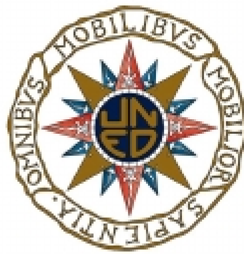
Universidad Nacional de Educación a Distancia

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

E.T.S. de Ingeniería Informática

Código (31105151) Trabajo de Fin de Máster



REDES M2M DESCENTRALIZADAS Y DISTRIBUIDAS DE IOT DE SENSORES Y ACTUADORES INTELIGENTES EN COMPUTACIÓN UBICUA

Autor: Óscar Moya Martínez
Director: Ismael Abad Cardiel

Curso 2019 – 2020, Convocatoria de Junio

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Código (31105151) Trabajo de Fin de Máster

REDES M2M DESCENTRALIZADAS Y DISTRIBUIDAS DE IOT DE SENSORES Y ACTUADORES INTELIGENTES EN COMPUTACIÓN UBICUA

Autor: Óscar Moya Martínez
Director: Ismael Abad Cardiel

Trabajo de Fin de Máster Tipo B

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 22/06/2020

Quién suscribe:

Autor(a): Óscar Moya Martínez
D.N.I/N.I.E/Pasaporte.: 44864851Q

Hace constar que es la autor(a) del trabajo:

Título completo del trabajo.

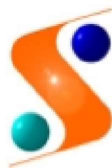
Redes M2M descentralizadas y distribuidas de IoT de sensores y actuadores inteligentes en computación ubicua

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.



Impreso TFdM05_AutorPbl. Autorización de publicación y difusión del TFM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09

www.issi.uned.es

RESUMEN

La interconexión de dispositivos en computación ubicua supone todo un reto arquitectónico donde entran en juego diversas disciplinas de ingeniería informática. Algunas soluciones actuales están basadas en tener un nodo central o plataforma que actúa de mediador entre las máquinas. Estas soluciones de nodos centrales obligan a tener una máquina de tipo computadora destinada a este fin y puede ser apropiada para redes de dispositivos grandes, como Internet o intranets grandes a nivel de una ciudad. Sin embargo, con el IoT una máquina inteligente debe tener entidad propia y no depender de un nodo central para funcionar. Se trata por tanto de proponer una verdadera red de máquinas descentralizadas y distribuidas donde la comunicación es entre las máquinas directamente y donde la inteligencia reside en los sensores y actuadores.

LISTA DE PALABRAS CLAVE

M2M, Plataforma IoT, Medios de Transmisión, Computación Ubicua, API REST, Wi-Fi, Arquitecturas de Sistemas Software, Patrones de Diseño Software, Smartphone, Android, Servicios, Jobs, Worker

Índice de contenido

1. Introducción....22
 1.1. Un Poco de Historia....22
 1.2. Objetivos y Alcance del Trabajo....23
 1.3. Estructura del Trabajo....24
  1.3.1. Primera Parte (Capítulos 2, 3 y 4)....24
  1.3.2. Segunda Parte (Capítulos 5, 6, 7 y 8)....25
  1.3.3. Tercera Parte (Capítulo 9)....25
2. Dispositivos Restringidos y Microordenadores para M2M, IoT y la Computación Ubicua....26
 2.1. NXP....27
 2.2. Arduino....28
 2.3. Raspberry PI....29
 2.4. Smart TV Box....30
3. Redes M2M....31
 3.1. Medios de Transmisión....31
  3.1.1. Bluetooth....32
  3.1.2. Red de Telefonía Móvil....34
  3.1.3. Wi-Fi....36
  3.1.4. RFID y NFC....38
  3.1.5. ZigBee....41
  3.1.6. Z-Wave....43
  3.1.7. LoRaWAN....44
  3.1.8. SigFox....44
  3.1.9. Neul....45
  3.1.10. Thread....45
 3.2. Tecnologías Software para la Interoperabilidad entre Procesos en Redes IP M2M....45
  3.2.1. Servicio Web SOAP....45
  3.2.2. Servicio Web REST....47
  3.2.3. Websockets....50
  3.2.4. MQTT....51
  3.2.5. DDS....54
  3.2.6. AMQP....54
   3.2.6.1 AMQP 0.91....55
   3.2.6.2 AMQP 1.0....56
  3.2.7. ZeroMQ....56
  3.2.8. CoAP....59
  3.2.9. NNG....61
  3.2.10. 6LoWPAN....62
  3.2.11. OMA Lightweight M2M....63
4. Las Plataformas IoT....66
 4.1. Modelos de Comunicación en Plataformas IoT....67
  4.1.1. Modelo 1 de Comunicación en Plataformas IoT....68
  4.1.2. Modelo 2 de Comunicación en Plataformas IoT....68
  4.1.3. Modelo 3 de Comunicación en Plataformas IoT....69
  4.1.4. Modelo 4 de Comunicación en Plataformas IoT....69

4.1.5. Modelo 5 de Comunicación en Plataformas IoT..............................70
4.2. Reflexiones sobre las Plataformas IoT...71
5. Arquitecturas de Sistemas Software para la Interoperabilidad de Sensores y Actuadores...72
5.1. Topologías de Red Descentralizadas para M2M de Sensores y Actuadores..72
5.1.1. Topología de Red de Malla Completa..72
5.1.2. Topología de Red de Malla Parcial..73
5.2. Arquitectura Cliente-Servidor de Sensores y Actuadores......................74
5.2.1. Dispositivo Servidor..75
5.2.2. Dispositivo Cliente..75
5.2.3. Dispositivo Cliente y Servidor...75
5.3. Modelos de Comunicación de Sensores y Actuadores...........................76
5.3.1. Modelo 1 de Comunicación de Sensores y Actuadores...................76
5.3.2. Modelo 2 de Comunicación de Sensores y Actuadores...................77
5.3.3. Modelo 3 de Comunicación de Sensores y Actuadores...................78
6. Protocolos y Patrones de Diseño Software para la Interoperabilidad entre Sensores y Actuadores...80
6.1. Patrón de Diseño Software REST...80
6.2. Patrón de Diseño Software Front Controller..84
6.2.1. Estructura Fija de Carga Útil..85
6.2.2. Estructura Variable de Carga Útil...86
6.3. Patrón de Diseño Software Proxy...88
6.4. Patrón de Diseño Software de Varios Procesos.....................................92
6.5. Patrones de Diseño Software de Mensajería...93
6.6. Otros Patrones de Diseño Software Utilizados en los Prototipos...........93
7. Descubrimiento de Máquinas en una red M2M Descentralizada..................95
7.1. Polling...95
7.2. Push..99
7.3. Difusión...99
7.4. Multienvío...101
7.5. Protocolos de Mensajería de Tópicos...104
7.6. Resumen sobre Este Capítulo..105
8. Los Prototipos en Android...107
8.1. Plataformas Android Para el Desarrollo de los Prototipos....................107
8.2. Arquitectura de los Prototipos...108
8.3. Servicios e Hilos de Ejecución en Android en la Plataforma de Smartphones/Tablets...111
8.3.1. Los Servicios y los Problemas en Android en la Plataforma de Desarrollo para Smartphones/Tablets..112
8.4. Servicios e Hilos de Ejecución en los Prototipos.................................113
8.4.1. Broadcast Receiver...115

8.4.1.1 Broadcast Receiver Estático..115
8.4.1.2 Broadcast Receiver Dinámico...117
8.4.2. JobService..119
8.4.3. Services..122
8.4.4. HandlerThread..128
8.4.5. SensorEventListener..130
8.4.6. Workers...132
8.4.7. Optimizador de Batería y la Lista Blanca de Android....................136
8.5. API de Persistencia Room en los Prototipos..138
8.6. Configurar la Ubicación en los Prototipos...141
9. Pruebas Empíricas y Comparativa entre los Protocolos de Comunicaciones en los Prototipos...143
9.1. Comparativa de Tiempos...145
9.2. Comparativa de Tamaño de Mensajes con TrafficStats de Android.....148
9.3. Comparativa de Tiempos y Tamaño de los Mensajes con Wireshark...152
9.4. Conclusiones de la Comparativa...153
10. Conclusiones y Trabajo Futuro..157
Referencias...160
Listado de siglas, abreviaturas y acrónimos..164
Anexo A. Documentación Adicional de los Prototipos....................................166
Recursos de Programación Utilizados...166
Dispositivos Físicos Android Utilizados...166
Manual de Operación (Instalación/Ejecución)..166
Manual de Usuario...167
Documentación del Código Fuente...167
Anexo B. Exportaciones de Wireshark de los Protocolos de Comunicaciones 168
Traza Completa http...168
Traza Enviado http...168
Traza Recibido http..168
Traza Completa Sockets TCP..169
Traza Enviado Sockets TCP..169
Traza Recibido Sockets TCP...169
Traza Completa Sockets UDP...170
Traza Enviado Sockets UDP...170
Traza Recibido Sockets UDP..170
Traza Completa CoAP...171
Traza Enviado CoAP...172
Traza Recibido CoAP..172
Traza Completa ZeroMQ TCP...173
Traza Enviado ZeroMQ TCP...173

Traza Recibido ZeroMQ TCP..173

Índice de figuras

Figura 2.1: Dispositivo Restringido para IoT. NXP i.MX RT1020 Crossover MCU con Arm Cortex-M7 core....27
Figura 2.2: Dispositivo Restringido para IoT. Arduino MKR1000....28
Figura 2.3: Microordenador Raspberry PI 4....29
Figura 2.4: Microordenador Smart TV Box. Bqeel....30
Figura 3.1: Redes Bluetooth omnidireccionales....32
Figura 3.2: Redes de telefonía móvil....34
Figura 3.3: Tarjetas SIM M2M de Movistar....36
Figura 3.4: Redes con punto de acceso central para conectar dispositivos alámbricos e inalámbricos....37
Figura 3.5: Red Wi-Fi con repetidores de señal....38
Figura 3.6: Etiqueta RFID....39
Figura 3.7: Circuito Integrado RFID....39
Figura 3.8: Tarjeta de Transporte RFID....40
Figura 3.9: ZigBee. Topología en árbol....42
Figura 3.10: ZigBee. Topología en estrella....42
Figura 3.11: ZigBee. Topología en malla....42
Figura 3.12: LoRaWan. Topología en Estrella....44
Figura 3.13: Petición raw http por POST a un servidor SOAP....46
Figura 3.14: Respuesta raw http de un servidor SOAP....46
Figura 3.15: Petición SOAP del body http....46
Figura 3.16: Respuesta SOAP del body http....46
Figura 3.17: Petición raw POST a un servidor REST....48
Figura 3.18: Petición raw GET a un servidor REST....48
Figura 3.19: Petición raw DELETE a un servidor REST....48
Figura 3.20: Petición raw PUT a un servidor REST....48
Figura 3.21: Respuesta raw de servidor REST....49
Figura 3.22: Documento json de la petición POST....49
Figura 3.23: Petición SOAP con autenticación básica OASIS....49
Figura 3.24: Petición REST http con el parámetro api-key....50
Figura 3.25: Comunicación con un broker para el paso de mensajes con el protocolo MQTT....52
Figura 3.26: Protocolo del tipo productor/consumidor con colas para el paso de mensajes....54
Figura 3.27: Protocolo AMPQ 0.91....56
Figura 3.28: Protocolo ZeroMQ. Modelos de Mensajería....58
Figura 3.29: Cliente CoAP....60
Figura 3.30: Captura Petición (rojo) – Respuesta (azul) CoAP con Wireshark....60
Figura 3.31: Captura Petición (rojo) – Respuesta (azul) http REST con Wireshark....60
Figura 3.32: Protocolo NNG. Patrón de mensajería survey (encuesta)....62
Figura 3.33: Protocolo NNG. Patrón de mensajería bus (Routing)....62
Figura 3.34: Redes 6LoWPAN....63
Figura 3.35: Servidor Leshan. Dispositivos conectados (Mi ordenador)....64
Figura 3.36: Servidor Leshan. Detalle del dispositivo msi....64
Figura 3.37: Captura de Wireshark cuando se enlaza el cliente con el servidor en Leshan....64
Figura 3.38: Captura de Wireshark cuando se hace clic en el botón read de Battery con un navegador web....64

Figura 3.39: Captura del debug del navegador web Chrome (petición) al servidor Leshan....65
Figura 3.40: Captura del debug del navegador web Chrome (respuesta) del servidor Leshan....65
Figura 4.1: Modelo 1 de Comunicación en Plataformas IoT....68
Figura 4.2: Modelo 2 de Comunicación en Plataformas IoT....68
Figura 4.3: Modelo 3 de Comunicación en Plataformas IoT....69
Figura 4.4: Modelo 4 de Comunicación en Plataformas IoT....69
Figura 4.5: Modelo 5 de Comunicación en Plataformas IoT....70
Figura 5.1: Formula para calcular el número de enlaces según el número de nodos....72
Figura 5.2: Topología de red de malla completa para 6 nodos....73
Figura 5.3: Topología de red de malla parcial para 6 nodos....74
Figura 5.4: Modelo 1 de Comunicación de Sensores y Actuadores....76
Figura 5.5: Modelo 2 de Comunicación de Sensores y Actuadores....77
Figura 5.6: Modelo 3 de Comunicación de Sensores y Actuadores....78
Figura 6.1: Petición GET http que devuelve información de un dispositivo....80
Figura 6.2: Información de un dispositivo....81
Figura 6.3: Petición GET http que devuelve la última lectura de la batería....81
Figura 6.4: Lectura del sensor de batería de un dispositivo....81
Figura 6.5: Código Android/Java que devuelve el documento json con información del dispositivo con AsynHttpServer en el prototipo TFM Server....82
Figura 6.6: Petición GET CoAP que devuelve la última lectura de la batería....83
Figura 6.7: Código Android/Java que devuelve el documento json con información del dispositivo con CoAP en el prototipo TFM Server....83
Figura 6.8: Bloque de Código Android/Java para añadir recursos en CoAP usando la reflexión en Java en el prototipo TFM Server....83
Figura 6.9: Bloque de Código Android/Java para añadir recursos en CoAP sin la reflexión en Java en el prototipo TFM Server....84
Figura 6.10: Estructura fija de carga útil que se envía desde el prototipo TFM Client al prototipo TFM Server mediante sockets TCP....85
Figura 6.11: Front Controller de los sockets TCP en el prototipo TFM Server....85
Figura 6.12: Command de los sockets TCP para obtener las lecturas del sensor de batería en el prototipo TFM Server....86
Figura 6.13: Estructura fija de carga útil que se envía desde el prototipo TFM Client al prototipo TFM Server mediante sockets UDP....87
Figura 6.14: Clase cliente que usa el patrón Proxy en el prototipo TFM Client.89
Figura 6.15: Superclase del patrón Proxy en el prototipo TFM Client....90
Figura 6.16: Clase Proxy del patrón Proxy en el prototipo TFM Client....90
Figura 6.17: Clase Action concreto del protocolo CoAP del patrón Proxy en el prototipo TFM Client....91
Figura 7.1: Clase controladora UnicastSubscribeController que busca dispositivos por la técnica de polling en el prototipo TFM Client....97
Figura 7.2: Command que envía una petición broadcast para anunciar su presencia en el prototipo TFM Server....100
Figura 7.3: Controlador que recibe una petición broadcast en el prototipo TFM Client....101

Figura 7.4: Command que envía una petición multicast en el prototipo TFM Server....102
Figura 7.5: Controlador que recibe una petición multicast en el prototipo TFM Client....103
Figura 8.1: Arquitectura de los prototipos....109
Figura 8.2: Diagrama de Procesos BPMN en el prototipo TFM Server....113
Figura 8.3: Diagrama de Procesos BPMN en el prototipo TFM Client....114
Figura 8.4: Broadcast Receiver de Tipo ACTION_BOOT_COMPLETE en el prototipo TFM Server....116
Figura 8.5: Broadcast Receiver de tipo estático ACTION_BOOT_COMPLETE en el manifest.xml del prototipo TFM Server....116
Figura 8.6: Broadcast Receiver de tipo WIFI_STATE_CHANGED_ACTION en el prototipo TFM Server....118
Figura 8.7: Invocación desde MainActivity para registrar y quitar el registro del receiver en los prototipos....118
Figura 8.8: Código para registrar y quitar el registro del receiver en los prototipos....119
Figura 8.9: Clase auxiliar JobServicesUtil de los prototipos para construir y lanzar un JobService....120
Figura 8.10: ServicesFacade de tipo JobService utilizado en ambos prototipos....121
Figura 8.11: Un JobService de Android en el manifest.xml de la app....122
Figura 8.12: Como se lanza un Service de Android en el prototipo TFM Server....124
Figura 8.13: Service de Android....126
Figura 8.14: Service de Android de los sockets TCP en el prototipo TFM Server....126
Figura 8.15: Declaración de Service de Android en el manifest.xml de los sockets TCP en el prototipo TFM Server....128
Figura 8.16: HandlerThread de Android de los sockets TCP en el prototipo TFM Server....129
Figura 8.17: SensorEventListener para leer el sensor de proximidad en el prototipo TFM Server....131
Figura 8.18: Como se lanza un Worker periódico de Android en ambos prototipos....134
Figura 8.19: ManagerWorker (Worker periódico) de Android en ambos prototipos....135
Figura 8.20: Ventana de mantenimiento para que las apps puedan hacer uso de la CPU y de la red....136
Figura 8.21: Intent implícito para abrir el optimizador de batería de las apps de Android....137
Figura 8.22: Captura de pantalla del emulador (API LEVEL 27), cuando se añade el prototipo TFM Server a la lista blanca....137
Figura 8.23: Acceso a la web generada por debug-db....139
Figura 8.24: Forward de puertos al emulador de Android Studio....139
Figura 8.25: Web generada por debug-db del prototipo TFM Server....140
Figura 8.26: Modelos Entidad-Relación de los prototipos....140

Figura 9.1: Captura de la configuración de los Test Servidores en el prototipo TFM Client....144
Figura 9.2: Captura del resultado del test de batería ALL desde el punto de vista TFM Client (Tiempos)....146
Figura 9.3: Captura del resultado del test de batería ALL desde el punto de vista TFM Server (Tiempos)....147
Figura 9.4: Captura del resultado del test de batería ALL desde el punto de vista TFM Client (Tamaño Mensajes)....148
Figura 9.5: Captura del resultado del test de batería ALL desde el punto de vista TFM Server (Tamaño Mensajes Recibidos)....150
Figura 9.6: Captura del resultado del test de batería ALL desde el punto de vista TFM Server (Tamaño Mensajes Respondidos)....150
Figura 9.7: Captura del resultado del test de batería LAST desde el punto de vista TFM Client (Tiempos)....154

Índice de tablas

Tabla 3.1: Clasificación de dispositivos Bluetooth....................................33
Tabla 3.2: Versiones de Bluetooth...33
Tabla 3.3: Generaciones de telefonía móvil..35
Tabla 3.4: Versiones Wi-Fi más conocidas...37
Tabla 7.1: Cuadro resumen de mecanismos para el descubrimiento de máquinas en redes M2M...106
Tabla 9.1: Cuadro resumen del resultado del test de batería ALL desde el punto de vista TFM Client (Tiempos)...146
Tabla 9.2: Cuadro resumen del resultado del test de batería ALL desde el punto de vista TFM Client (Tamaño mensajes real con TrafficStats).........................149
Tabla 9.3: Cuadro resumen del resultado del test de batería ALL desde el punto de vista TFM Client (Tamaño mensajes sólo carga protocolo con TrafficStats) 149
Tabla 9.4: Cuadro resumen del resultado del test de batería ALL con Wireshark (Tamaño Mensajes)...152
Tabla 9.5: Cuadro resumen del resultado del test de batería ALL con Wireshark (Tiempos)...152
Tabla 9.6: Cuadro resumen del resultado desde todos los puntos de vista del test de batería ALL...153
Tabla 9.7: Cuadro resumen del resultado del test de batería LAST desde el punto de vista TFM Client (Tiempos)..154
Tabla 9.8: Cuadro resumen de cuando utilizar los protocolos probados.........156

CAPÍTULO 1

1. INTRODUCCIÓN

Uno de los objetivos de este trabajo de investigación es proponer una verdadera arquitectura máquina a máquina (M2M) entre máquinas de distinta naturaleza, como pueden ser la conexión entre distintos sistemas operativos o entre diferentes máquinas que utilicen distintos lenguajes de programación. O bien, máquinas que, funcionalmente, hacen cosas diferentes pero que juntas completan de forma más eficiente su funcionalidad, colaborando las máquinas entre sí. El concepto M2M es relativamente nuevo pero la esencia es la misma a P2P (Peer to Peer) que se basa en el intercambio de información entre personas, en general distantes entre sí. El término M2M es similar a P2P, pero el intercambio de información se produce entre máquinas remotas.

La comunicación de las máquinas en las redes M2M tiene un propósito bien definido y es conectar las máquinas para que colaboren entre sí conformando una red de máquinas de Internet de las Cosas (IoT) con sensores que miden el espacio físico donde están las personas y actuadores que influyen en este espacio físico.

La computación ubicua se basa en el concepto de que la informática y los ordenadores se encuentren en todas partes y en cualquier momento, integrando los ordenadores en objetos cotidianos y otros objetos que pueden estar ocultos a la vista de las personas. Por tanto, si entramos a una sala, habitación o un espacio donde existe un sistema ubicuo no vemos el típico ordenador de escritorio con su pantalla ni un armario o cabina de servidores, si no que las computadoras están repartidas por la sala en forma de objetos corrientes y una red de sensores y actuadores que pueden estar ocultos a la vista de las personas.

1.1. UN POCO DE HISTORIA

Hace ya bastante tiempo, en los tiempos de los mainframes o servidores que ocupaban hasta una habitación, los servidores eran los más potentes del mercado y los clientes llamados terminales, eran llamados terminales tontos, debido a que eran ordenadores muy simples. Los terminales tontos tenían una pantalla, teclado y puede que un ratón y abrían un conexión telnet o similar con el servidor para trabajar. El terminal tonto no podía hacer prácticamente nada sin el servidor central, porque todo el procesamiento y la lógica de la aplicación residía en el mainframe.

Después empezaron a surgir ordenadores de tipo teclado. Un ordenador se embebía en un teclado y se conectaba a una pantalla de televisión o monitor de tubo de rayos catódicos. Algunos de estos ordenadores son los Spectrum, los Commodore 64 y los Amstrad. En esta época todos los bytes del programa contaban porque la memoria era escasa y muy cara, estos ordenadores solían contar con 64 KB de memoria RAM y los mejores con 128 KB.

Más adelante, empezaron a surgir los ordenadores personales o los PC (Personal Computer) con muy pocos recursos y bastantes lentos, pero en pocos años empezaron a ganar potencia y recursos y empezaron a realizar tareas bastantes complejas que sólo estaban destinadas a los servidores de tipo mainframes.

Cuando los ordenadores personales eran aún lentos, la red de comunicaciones como Internet era lenta también, con los modems de 56 Kb y con las tecnologías de ADSL y cable que no llegaban a un 1 Mb. En esta época, todo iba lento y había que armarse de paciencia para realizar cualquier tarea.

Con el tiempo, la velocidad de la red de comunicaciones y la potencia de los ordenadores personales mejoraron de forma espectacular. Casi al mismo tiempo, han ido surgiendo otros dispositivos como los smartphones, tablets y smart TV, que si bien al principio no tenían mucha capacidad, la han ido ganando en poco tiempo, al igual que los ordenadores personales. Este paradigma es el que ha estado predominando durante bastantes años atrás, donde todo (ordenadores, portátiles, smartphones, tablets, las redes de comunicaciones y los servidores) es muy rápido.

En esta época en la que todo trabaja muy rápido, ha surgido también el concepto de la computación en nube, que se basa en que todos los programas y servicios estén centralizados en plataformas en Internet, que sería un paradigma similar a los mainframes y los terminales del primer párrafo. De manera similar y más recientemente, está cogiendo fuerza el concepto de CoT (Cloud of Things) como plataformas IoT en la nube, para gestionar dispositivos del IoT en la nube.

Actualmente, con el paradigma de las redes M2M, la computación ubicua y el IoT, volvemos a tener que lidiar y trabajar de nuevo con dispositivos con muy pocos recursos y por si fuera poco, algunos de estos dispositivos pueden funcionar con una batería y hay que maximizar la vida de la batería.

Desarrollar software para las redes M2M y el IoT, no resulta ser una tarea sencilla, debido a que los programadores se han acostumbrado a desarrollar software para ordenadores personales y dispositivos como smartphones de varios GB de RAM y procesadores con varios núcleos, servidores de aplicaciones potentes y redes de comunicaciones que alcanzan velocidades del orden de Mb o Gb por segundo.

1.2. OBJETIVOS Y ALCANCE DEL TRABAJO

En este trabajo se propone una arquitectura software donde no es necesario disponer de una plataforma IoT para que las máquinas puedan trabajar juntas y puedan comunicarse entre sí. La plataforma IoT puede existir, pero es un nodo más de la red M2M. La comunicación entre los sensores y actuadores no pasa por una plataforma IoT, si no que son los sensores y los actuadores los que se comunican directamente entre sí. De esta forma, conseguimos una red M2M distribuida y descentralizada de sensores y actuadores.

Este trabajo, bajo el paraguas de la computación ubicua, se centra más en la parte de más alto nivel de la interoperabilidad de las máquinas, como los protocolos software, el formato de intercambio de información y en la ingeniería de software requerida de sensores y actuadores. Dejando más de lado la ingeniería y tecnología de las comunicaciones que conforman la capa física y de enlace de datos de las comunicaciones y la capa de hardware de los sensores y actuadores.

También se incluyen dos prototipos que cubren los aspectos de la arquitectura M2M descentralizada propuesta y que están basados en la plataforma de desarrollo Android y su SDK en Java. Android, al estar basado en Linux, puede instalarse en casi cualquier dispositivo y puede ser una buena plataforma de inicio para desarrollar los prototipos, especialmente porque todo el software relacionado es freeware y el hardware puede ser ejecutado con emuladores y los smartphones o tablets de Android están más al alcance de la mano que otros dispositivos del mercado.

En los prototipos se han utilizado varios protocolos software y se ha realizado una comparativa de tiempos y de tamaño de los mensajes, con el objetivo de investigar si realmente los protocolos IoT probados son más rentables que otros protocolos más tradicionales.

1.3. ESTRUCTURA DEL TRABAJO

Este trabajo de investigación se centra en tres partes diferenciadas que se resumen a continuación:

1.3.1. Primera Parte (Capítulos 2, 3 y 4)

En esta parte se investiga el panorama actual de la computación ubicua, el M2M y el IoT. Consta de los siguientes capítulos:

- **Capítulo 2.** Se trata de una pequeña introducción a los considerados dispositivos restringidos y los microordenadores, como nuevo soporte hardware para el desarrollo de aplicaciones informáticas y las tecnologías de la información y comunicación.
- **Capítulo 3.** Esta parte es la investigación del estado del arte actual de las tecnologías de la información y las comunicaciones. Se investigan si algunas tecnologías tradicionales destinadas a las redes de computadores (servicios web) y reutilizadas algunas en la computación móvil (smartphones) pueden reutilizarse para M2M, la computación ubicua y el IoT. También se investigan nuevas tecnologías, algunas de ellas desarrolladas explícitamente para M2M, la computación ubicua y el IoT. Dependiendo de su interés para M2M, el IoT y de la información disponible, algunas se analizan en más profundidad que otras.
- **Capítulo 4.** Por su importancia, también se analizan las plataformas IoT en la actualidad y su relación con la computación ubicua de los sensores y los actuadores.

1.3.2. Segunda Parte (Capítulos 5, 6, 7 y 8)

Se hace un análisis y se llega hasta el nivel de implementación para conseguir una arquitectura descentralizada de máquinas inteligentes de sensores y actuadores que se encuentran automáticamente basándose en la investigación de la primera parte.

- **Capítulo 5.** En este capítulo se hace hincapié en el objetivo fundamental de este trabajo que es conseguir una arquitectura descentralizada de sensores y actuadores.
- **Capítulo 6.** Se baja de nivel de abstracción y se proponen tecnologías, protocolos y patrones de diseño software para conseguir el objetivo de la arquitectura descentralizada.
- **Capítulo 7.** Se analizan los problemas y se proponen algunas soluciones para que las máquinas tengan la capacidad de autoconfigurarse por si mismas.
- **Capítulo 8.** Se comentan las cuestiones y los problemas más importantes en el desarrollo de los prototipos, junto a las soluciones adoptadas.

1.3.3. Tercera Parte (Capítulo 9)

Esta parte sólo tiene un capítulo y su objetivo principal es la de probar todo el trabajo realizado y realizar una comparativa entre los distintos protocolos, tomando como indicadores el tiempo en procesar las solicitudes y el tamaño de los mensajes enviados y recibidos. No se tratan de pruebas funcionales de los prototipos, que también se han realizado, si no se trata de probar que protocolo ofrece mejores prestaciones. Se comparan 3 protocolos tradicionales (http, sockets TCP y sockets UDP) y 2 protocolos IoT (CoAP y ZeroMQ TCP)

CAPÍTULO 2

2. DISPOSITIVOS RESTRINGIDOS Y MICROORDENADORES PARA M2M, IOT Y LA COMPUTACIÓN UBICUA

Este capítulo es una pequeña extensión de la introducción del capítulo 1.1 que habla sobre la historia y evolución de los sistemas informáticos, con el objetivo de tener una pequeña panorámica actual del software que se desarrolla para un hardware restringido. En un contexto de M2M, IoT y la computación ubicua, es muy importante entender desde el principio, que el hardware puede tener ciertas limitaciones, de forma que nuestro software puede funcionar en un ordenador personal y en un smartphone de la actualidad pero no lo hará (o lo hará con muchas dificultades) en un hardware restringido.

De forma general, la palabra restringido, indica algo que está limitado y reducido. En dispositivos y hardware restringido, indica que el hardware está limitado en potencia y es reducido (puede ser pequeño en tamaño), en comparación con los ordenadores personales de la actualidad. Esto es parcialmente verdadero, debido a que existe hardware muy restringido de forma que recuerdan a los primeros ordenadores, pero existe otro hardware, no tan restringido, de hasta pequeños microordenadores, que tienen la capacidad de un smartphone actual de gama media-alta en la actualidad o a un ordenador personal actual de gama baja.

Por tanto, los dispositivos restringidos son utilizados en dispositivos finales como los sensores y los actuadores y que, dependiendo del dispositivo, pueden tener interfaces para conectarse a otros dispositivos como a un microordenador. Estas conexiones se pueden hacer con cables físicos directos o bien pueden ser dispositivos autónomos y que pueden tener capacidades de red para poder conectarse a otros dispositivos. Los microordenadores, sin embargo, se pueden utilizar para el IoT pero también para otros propósitos.

Hoy en día, existen fabricantes de este hardware considerado restringido y de microordenadores, algunos de ellos se resumen en las siguientes subsecciones.

2.1. NXP

NXP es un compañía a nivel internacional que tiene su sede en los países bajos, aunque tiene repartidas oficinas por todo el mundo. Por la web consultada en [1], tienen una amplia gama de productos, desde sensores, fuentes de alimentación, procesadores, microcontroladores y productos destinados a nivel industrial, de la automoción, la domótica y las ciudades inteligentes. Respecto a los microcontroladores, tiene un amplia gama de productos muy restringidos y otros no tan restringidos.

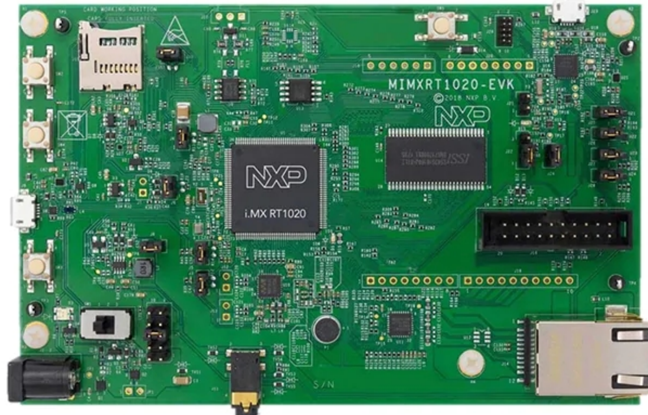


Figura 2.1: Dispositivo Restringido para IoT. NXP i.MX RT1020 Crossover MCU con Arm Cortex-M7 core

El dispositivo anterior, según NXP, está destinado para el IoT y está equipado con:

- **Procesador**: Cortex-M7 500 MHz
- **Memoria RAM**: 256 KB.
- **Memoria Flash** (para el sistema operativo): 64 MB.
- **Conectividad**: Wi-Fi, Bluetooth, Bluetooth Low Energy, ZigBee y Thread.

Otros dispositivos NXP tienen tan sólo 100 MHz y 64 KB de memoria RAM, mientras que otros dispositivos llegan al 1,2 GHz de frecuencia de procesador, dependiendo de la aplicación y el uso que se vaya a dar al dispositivo.

2.2. ARDUINO

Arduino es una compañía que ha ganado mucha popularidad por su lema de desarrollo software y hardware libres. Mientras que NXP del apartado anterior parece más destinado a profesionales y al sector de la industria, Arduino está más accesible y destinado a un consumidor medio, entendiendo como consumidor medio, a estudiantes, usuarios avanzados o profesionales de la informática que deseen aprender Arduino y realizar proyectos a nivel personal o incluso aplicarlos a nivel profesional o industrial. Como curiosidad, algunos dispositivos NXP cuentan con interfaces de conexión con Arduino.

Arduino cuenta con un IDE de desarrollo propio y con una serie de placas que poseen diferentes funcionalidades. La más importante es el microcontrolador principal llamado Arduino Uno, aunque existen otras variantes. A esta placa principal se le pueden conectar sensores y periféricos, como por ejemplo periféricos que ofrecen conectividad Wi-Fi.

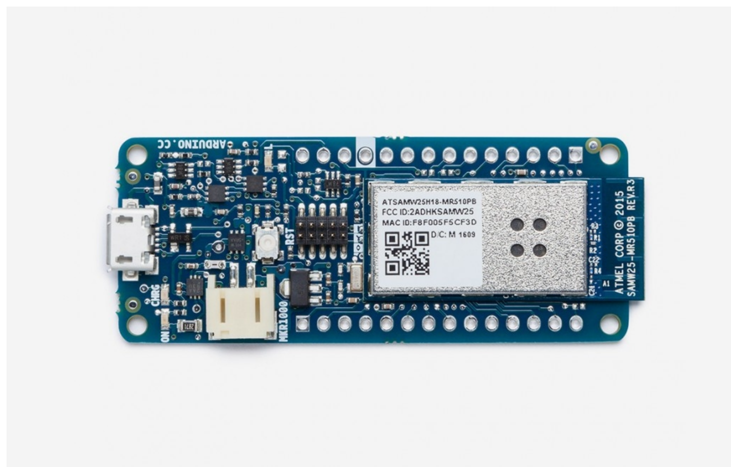


Figura 2.2: Dispositivo Restringido para IoT. Arduino MKR1000

El dispositivo Arduino de la figura anterior está destinado para el IoT y está equipado con:

- **Procesador**: 48 MHz.
- **Memoria RAM**: 32 KB.
- **Memoria Flash** (para el sistema operativo): 256 KB.
- **Conectividad**: Wi-Fi.

En [2] se puede descargar su IDE de desarrollo y comprobar la gama de placas electrónicas de que dispone.

2.3. RASPBERRY PI

Raspberry PI fabrica placas electrónicas de un único modelo, a diferencia de Arduino que fabrica diferentes placas electrónicas para diferentes propósitos. La Raspberry PI ha ido evolucionando con el tiempo y ganando potencia. El último modelo Raspberry PI 4 podría decirse que es un microordenador.

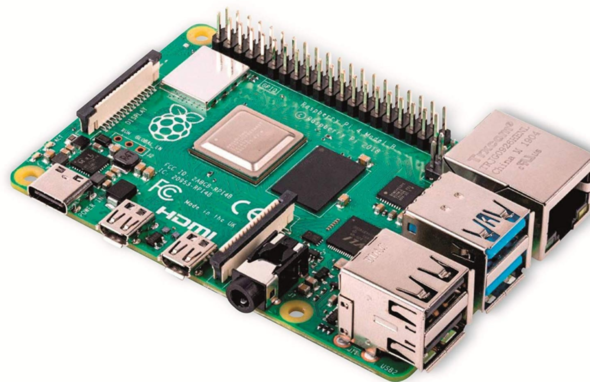


Figura 2.3: Microordenador Raspberry PI 4

Dada su potencia, la Raspberry PI se puede utilizar para más cosas o proyectos que el IoT y se utiliza como videoconsola retro, centro multimedia, smart TV o servidor de ficheros. La Raspberry PI 4 está equipada con [3]:

- **Procesador**: 1.5GHz 64-bit quad-core Cortex-A72.
- **Memoria RAM**: 2, 4 o 8 GB.
- **Memoria Flash** (para el sistema operativo): MicroSD de varios GB (256 GB).
- **Conectividad**: Wi-Fi, Bluetooth y Ethernet de cable.

La Raspberry PI tiene como sistema operativo principal un Rasbian, basado en una Debian Linux, pero se puede instalar hasta un Windows 10 o un Android.

Volviendo un poco al IoT, Rasberry PI cuenta con una interfaz GPIO que son los conectores de 40 pins (similares a los discos duros IDE PATA antiguos) y que pueden extender su funcionalidad con otros periféricos, que ya no son fabricados por Rasberry PI. Incluso venden adaptadores para conectar Rasberry PI a placas de Arduino de sensores para aprovechar estas placas en una Rasberry PI.

Si bien es cierto que Rasberry PI no está orientado sólo al IoT, es posible que con el tiempo fabrique dispositivos Rasberry PI destinados para el IoT y que dispongan de sensores incorporados en la propia placa.

Dada la popularidad ganada por las Rasberry PI y su número de ventas, otros fabricantes han construido clones de estas placas electrónicas, algunas incluso más potentes.

2.4. SMART TV BOX

Estos dispositivos se utilizan como centro multimedia, conectándolos a una televisión, convertimos una televisión convencional en una smart TV. En general, suelen llevar el sistema operativo Android. Para el IoT no tienen mucho interés, debido a que no suelen llevar casi nada de sensores incorporados, seguramente para abaratar costes, pero existe una fuerte competición en el mercado dada la amplia gama de productos de este tipo que se está vendiendo. Lo más interesante es su bajo coste y la potencia que consiguen.

Figura 2.4: Microordenador Smart TV Box. Bqeel

El Smart TV Box está equipado con [4]:

- **Procesador**: RK3318 Quad-Core 64bit Cortex-A53
- **Memoria RAM**: 4 GB.
- **Memoria Flash** (para el sistema operativo): 64 GB.
- **Conectividad**: Wi-Fi, Bluetooth 4 y Ethernet de cable.

El dispositivo anterior incorpora Android y para IoT se podrían construir dispositivos de este tipo, pero con menos potencia, para abaratar costes e incorporar los sensores en el propio dispositivo, tal y como llevan incorporados los sensores en un smartphone con Android.

CAPÍTULO 3

3. REDES M2M

El término M2M se refiere a Machine To Machine y consiste en el intercambio de información de las comunicaciones entre máquinas remotas entre sí. La comunicación M2M es una nueva tecnología de comunicaciones donde un gran número de dispositivos inteligentes pueden comunicarse de forma autónoma con otros dispositivos y tomar decisiones de forma colaborativa sin intervención humana [5].

No todas las máquinas pueden formar parte de una red de máquinas M2M. Por lo tanto, una máquina M2M debe tener las siguientes capacidades:

- Enviar información a otros dispositivos.
- Recibir información de otros dispositivos.
- Procesar la información que le llega de otras máquinas. El procesado implica la serialización de datos cuando se envía por la red de comunicaciones y el proceso inverso de deserializar los datos cuando se reciben.
- Debe tener capacidades alámbricas o inalámbricas para conectarse a una red. Por alámbricas se refiere a que tenga un adaptador de red y que el cable conectado a la tarjeta de red se conecte al cableado de la infraestructura de red. Por inalámbrica, puede ser por Wi-Fi o bien con tecnología inalámbrica de datos como 3G, 4G o 5G, utilizada en los smartphones.

Algunos ejemplos de máquinas que pueden formar parte de una red M2M, pueden ser un ordenador con tarjeta de red alámbrica, un portátil con Wi-Fi, un smartphone, una tablet, pero también dispositivos que tengan un sensor y que publiquen sus lecturas en la red para que otros dispositivos las lean.

Otros ejemplos de máquinas que no pueden formar parte de una red M2M son una nevera, una lavadora, un despertador o un microondas, siempre y cuando estas máquinas no tengan capacidades de conexión a red. Sin embargo, cada vez es más frecuente que máquinas corrientes, como las neveras, las cuales nunca han tenido capacidades de conexión a una red, las tengan actualmente.

3.1. MEDIOS DE TRANSMISIÓN

Los medios de comunicación o transmisión se refiere a la capa física o al medio físico por el que viaja la información, hay dos tipos [6]:

- **Alámbrica:** el medio físico es un cable. Hay diversos tipos de cables para transmitir la información, a su vez pueden dividirse en cables eléctricos (utilizan la electricidad para transmitir la información) y los cables ópticos (utilizan ondas de luz para transmitir la información). Este tipo de transmisión se conoce como medios guiados.

- **Inalámbrica:** el medio físico no utiliza un cable, si no que utiliza el aire en forma de ondas electromagnéticas para la transmisión de datos. Dentro del espectro electromagnético tenemos varias tecnologías actuales que funcionan en diferentes bandas de frecuencia cuya medida es el hercio. Este tipo de transmisión se conoce como medios no guiados.

En las redes M2M de sensores y actuadores de IoT nos interesa especialmente la tecnología inalámbrica, dejando un poco de lado a la tecnología alámbrica, sin perjuicio de que una máquina con tecnología alámbrica, como un ordenador personal o un microordenador pueda formar parte de una red M2M.

Dentro de la tecnología inalámbrica para las redes M2M tenemos varias posibilidades, algunas son más conocidas y usadas, mientras que otras son relativamente nuevas, están en proceso de construcción o todavía no han llegado al mercado, en cualquier caso, están presentadas en las siguientes subsecciones.

3.1.1. Bluetooth

La tecnología Bluetooth es omnidireccional y permite la transmisión de datos a diferentes dispositivos a través de objetos y por tanto los dispositivos pueden comunicarse entre sí, sin tener que estar alineados, tal y como muestra la siguiente figura.

Figura 3.1: Redes Bluetooth omnidireccionales

El Bluetooth no consume mucha energía si el alcance es bastante limitado, como 10 o 15 metros. Si queremos más distancia el consumo de energía es mayor. El Bluetooth se utiliza hoy en día con bastante frecuencia en smartphones, tablets, equipos de música, mandos a distancia de algunos smart TV y controladores de juegos (gamepads) de ordenadores y videoconsolas. Debido a su alcance limitado con un consumo de energía bajo y a la baja velocidad de transferencia de datos, tampoco son un buen candidato para las redes de dispositivos móviles M2M donde haya cierta cantidad de dispositivos o donde haya que transmitir una cantidad considerable de datos. Además, el emparejado o enlace entre los dispositivos suele ser bastante complejo entre dispositivos, obligando a intercambiar unas claves para

establecer la conexión, este proceso de emparejado no es automático y lo tiene que hacer una persona.

Sin embargo, las nuevas versiones de Bluetooth están mejorando mucho, bajando la potencia de consumo y el alcance de la transmisión y el Bluetooth se está especializando en el Internet de las Cosas [7], convirtiéndose en un competidor de las redes Wi-Fi, especialmente porque a partir de la versión 4.1, un dispositivo Bluetooth soporta el protocolo IPv6 y, en teoría, puede estar conectado directamente a Internet. Los dispositivos Bluetooth se clasifican en 4 clases según su potencia y alcance tal y como muestra la siguiente tabla:

Clase	Potencia máxima permitida (mW: miliwatios)	Alcance aproximado
Clase 1	100 mW	100 metros
Clase 2	2.5 mW	5-10 metros
Clase 3	1 mW	1 metro
Clase 4	0.5 mW	0.5 metro

Tabla 3.1: Clasificación de dispositivos Bluetooth

Los dispositivos de la clase 2, son los más utilizados en los dispositivos móviles y son los más interesantes para las redes M2M, debido al compromiso entre el consumo de energía y el alcance que tienen. Un dispositivo de una clase es compatible con otro dispositivo de cualquier otra clase.

Actualmente, existen varias versiones de Bluetooth que se resumen en la siguiente tabla en base a su ancho de banda y las características más importantes:

Versión	Ancho de Banda	Características más importantes para M2M
1.2	1 Mbit/s	
2.0	3 Mbit/s	
3.0	24 Mbit/s	Mejoras sustanciales en el ancho de banda
4.0 (Bluetooth Low Energy)	32 Mbit/s	Se mejora el consumo de energía, Diseñado para funcionar con IoT (puede funcionar con el protocolo IP)
5.0	50 Mbit/s	Tasa de transferencia elevada, La versión 5.1 permite localizar físicamente a otros dispositivos

Tabla 3.2: Versiones de Bluetooth

Debido a su alcance limitado, el Bluetooth también se conoce para formar redes PAN (Personal Area Network) como una red de dispositivos cercanos a la persona. Para las redes M2M tiene especial interés a partir de su versión 4.0 que da soporte al protocolo IP y su bajo consumo de energía, para formar redes M2M pequeñas o personales. Para redes más grandes, otro dispositivo con Bluetooth puede compartir su conexión a Internet con Wi-Fi o con tecnología alámbrica, de manera que los dispositivos de la red PAN tengan acceso a Internet, sin embargo, puede ser complejo o no ser posible que un

dispositivo fuera de la red PAN tenga acceso a los dispositivos de la red PAN. Además, otro inconveniente es que de momento no existen máquinas con el rol de punto de acceso, tal y como existen para las redes Wi-Fi, si no que es otro dispositivo como un smartphone o un ordenador personal los que comparten su conexión a Internet.

3.1.2. Red de Telefonía Móvil

Las comunicaciones de la telefonía móvil han ido y continúan evolucionando de forma constante en el tiempo. Todas se caracterizan porque usan una tarjeta SIM con un número de teléfono que se introduce en un teléfono móvil y que nos da acceso a la red del operador de telefonía. La telefonía móvil funciona con estaciones base que dan cobertura a los nodos de la red (como los teléfonos móviles) y cada estación base puede soportar un número elevado de nodos, tal y como se muestra en la siguiente figura.

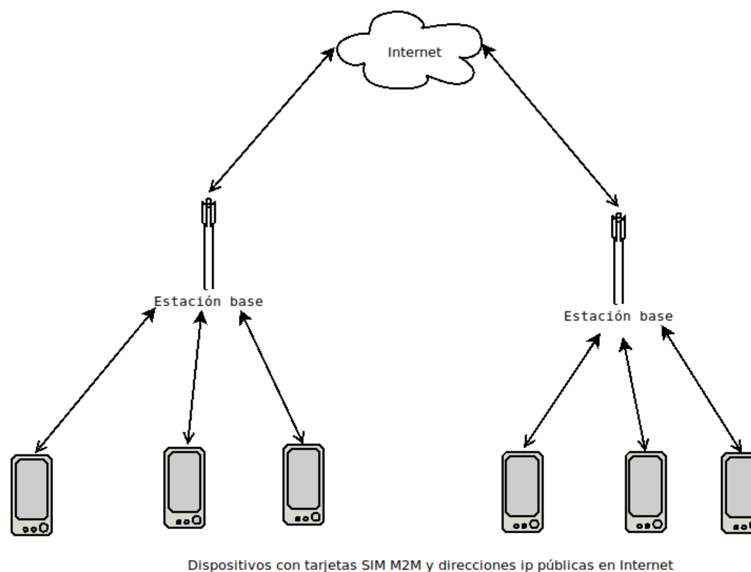


Figura 3.2: Redes de telefonía móvil

Hoy en día, es común catalogar las versiones de telefonía móvil por la generación, actualmente se utilizan la 3G y la 4G. Una generación puede entenderse como una versión, y en cada generación se definen las especificaciones.

Las generaciones más antiguas, anteriores a la 3G, sólo permitían transmitir y recibir voz, mensajes de texto plano, mensajes multimedia y acceso a la web con una velocidad de transferencia muy baja (entre 56 Kbit/s y 114 Kbit/s), lo que no les hacían muy adecuados para una red M2M.

A partir de la generación 3G, la velocidad mejoró de forma considerable y un dispositivo con 3G y cobertura 3G permite navegar por Internet de forma fluida con una velocidad de unos 2 Mbit/s.

La generación 4G permite velocidades muy altas, entre 100 Mbit/s y 1 Gbit/s, lo que las hace equiparables a las redes LAN cableadas Ethernet con

cables UTP de categoría 5 y 6. Como pasó con la red 3G, los operadores de telefonía móvil tienen que actualizar las estaciones base y por tener un móvil 4G no garantiza que la velocidad vaya a ser 4G.

La última generación es la 5G que está desarrollada pero aún no se ha llegado a implantar de forma generalizada, pero está previsto que se implantará en el año 2020. Es muy rápida y permitirá velocidades de transferencia de desde 1 Gbit/s hasta los 20 Gbit/s, según algunas fuentes de Internet (consultar tabla 3.3). La arquitectura y capacidades de la próxima generación 5G de teléfonos y redes inalámbricas son conducidas por la demanda y los requerimientos del tráfico entre personas y del tráfico M2M [8].

Todas las generaciones modernas tienen tasas de transferencia muy altas y funcionan con el protocolo IP, lo que las hace muy adecuadas para el Internet de las Cosas y las redes M2M, especialmente la esperada 5G con la tasa de transferencia más alta y su baja latencia permitirá comunicar dispositivos en tiempo real. Además, el 5G consume menos energía y permite un número mayor de dispositivos conectados por estación base. En la siguiente tabla se resumen las tasas de transferencia:

Versión	Ancho de Banda
Anterior a 3G	Entre 56 Kbit/s y 114 Kbit/s
3G	2 Mbit/s
4G	Entre 100 Mbit/s y 1 Gbit/s
5G	Entre 1 Gbit/s y 20 Gbit/s *

Tabla 3.3: Generaciones de telefonía móvil

** Como aún no se ha implantado el 5G la velocidad puede variar según el artículo de Internet. Lo más oficial es que la velocidad será a partir de 1 Gbit/s*

Sobre el alcance de una estación base de telefonía móvil, el operador de telefonía divide el territorio en celdas, y en cada celda hay una estación base. El alcance de una celda suele rondar entre los 400 metros y los 20 kms, dependiendo de la densidad de población y la geografía, es posible que algunas celdas se solapen, pudiendo estar en un punto geográfico con cobertura por varias estaciones base.

Una peculiaridad del ancho de banda es que varían si el dispositivo está en movimiento o bien está en reposo. Por tanto, si vamos circulando con un coche el ancho de banda será menor, mientras que si estamos parados el ancho de banda es mayor. Cuando conectamos nuestro dispositivo móvil a la tarifa de datos del operador de telefonía nos da una dirección IP pública en Internet. A pesar de que la dirección IP es pública y debería ser accesible desde cualquier parte del mundo, por ejemplo desde el ordenador de una casa, el operador de telefonía corta el acceso al dispositivo móvil por razones de seguridad, es decir el tráfico entrante al dispositivo está cortado y está permitido el tráfico saliente del dispositivo. Sin embargo, la empresa de telecomunicaciones Movistar ofrece tarjetas SIM M2M, que aparentemente son tarjetas SIM corrientes, tal y como muestra la siguiente figura:

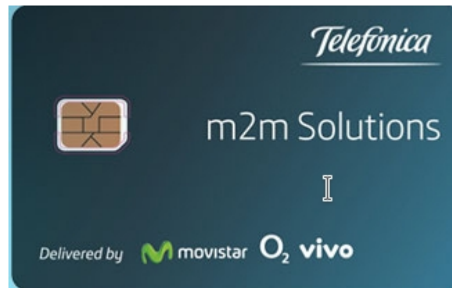


Figura 3.3: Tarjetas SIM M2M de Movistar

En el enlace de [9] puede consultarse alguna información sobre este tipo de tarjetas SIM. Además, Movistar ofrece aplicaciones para la gestión de las tarjetas y los dispositivos que las incorporan. Aunque en el enlace no lo especifican, es de suponer que este tipo de tarjetas SIM permitirán el tráfico entrante a los dispositivos, aunque es posible que pueda estar restringido y no esté abierto a todo Internet. Tampoco especifican el alcance de la gestión de los dispositivos, pero no irá más allá de la gestión de las comunicaciones, la conectividad y el estado de las SIM M2M de nuestra red. Este tipo de tarjetas SIM M2M, junto a la tecnología 4G y 5G son muy adecuadas para formar una red M2M de dispositivos a nivel global. Esta solución tiene el inconveniente de la cobertura del operador, si tenemos máquinas en zonas remotas sin cobertura o con mala cobertura o edificios que por sus características arquitectónicas, como muros muy gruesos o algunas zonas de los edificios como los sótanos donde hay poca cobertura o ninguna, la red de tarjetas SIM M2M no nos servirá y tendremos que buscar otras soluciones. Este tipo de tarjetas SIM M2M las tienen que estar usando las empresas eléctricas en sus modernos contadores de luz, los cuales envían la lectura del contador de luz a la central de la empresa eléctrica y de esta forma no tiene que trasladarse físicamente una persona de la empresa eléctrica al domicilio de los consumidores para tomar la lectura. Otro ejemplo de uso que hacen en [9] es para las máquinas de vending. Este tipo de máquinas venden bebidas y comida a los usuarios. La máquina de vending puede avisar a la central de que se están agotando las existencias, pueden informar de averías en la máquina e incluso se podría configurar de forma remota los precios de los productos que se venden. Todo esto ahorra costes de desplazamientos de los trabajadores a las máquinas de vending.

3.1.3. Wi-Fi

Las redes Wi-Fi están muy evolucionadas hoy en día, son muy rápidas y seguras usando un buen algoritmo de cifrado. Las redes Wi-Fi están gobernadas por la Wi-Fi Alliance [10]. En cada red Wi-Fi, hay un punto de acceso Wi-Fi que conectará a todos los dispositivos en el mismo direccionamiento IP. Este tipo de redes tienen el aspecto de la siguiente figura:

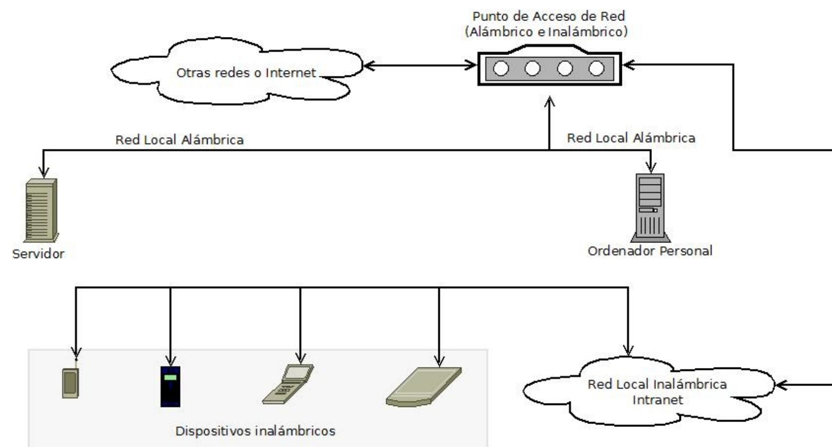


Figura 3.4: Redes con punto de acceso central para conectar dispositivos alámbricos e inalámbricos

Al igual que las tecnologías inalámbricas anteriores, las redes Wi-Fi han ido evolucionando con el tiempo y el IEEE ha estado publicando nuevos estándares y los fabricantes han ido implantándolos. Los más usados y conocidos se resumen en la siguiente tabla:

Versión	Ancho de Banda
IEEE 802.11b	11 Mbit/s
IEEE 802.11g	54 Mbit/s
IEEE 802.11n (Wi-Fi 4)	600 Mbit/s
IEEE 802.11ac (Wi-Fi 5)	1 Gbit/s
IEEE 802.11ax (Wi-Fi 6)	Entre 10 Gbit/s y 40 Gbit/s *

Tabla 3.4: Versiones Wi-Fi más conocidas

** Al ser una tecnología nueva, puede haber discrepancias de la velocidad máxima del Wi-Fi 6 publicada en Internet. Algunas fuentes sugieren que será a partir de los 11 Gbit/s*

Las versiones n y ac, son las versiones más utilizadas hoy en día en el entorno doméstico y profesional. Debido a la dificultad de recordar la versión del IEEE 802.11x de Wi-Fi para un usuario o consumidor medio, la Wi-Fi Alliance, una organización que se encarga de promover y certificar la tecnología Wi-Fi, ha puesto un nombre más fácil de recordar, tal y como muestra en la tabla 3.4 con los nombres de Wi-Fi x entre paréntesis.

Los dispositivos Wi-Fi son compatibles entre sí, independientemente de la versión que utilicen. Sin embargo, hay una excepción, las redes Wi-Fi operan en la banda de frecuencia de los 2,4 GHz y para conseguir más velocidad de transferencia utilizan la banda de los 5 GHz. Por tanto, un dispositivo que sólo puede operar en la banda de los 2,4 GHz no podrá conectarse a un punto de acceso que sólo ofrece la banda de los 5 GHz.

Respecto a la distancia o alcance de las redes Wi-Fi, las redes Wi-Fi pueden abarcar una área pequeña, como una casa, pero también áreas más grandes, como un edificio, una estación de tren, un aeropuerto, un barrio de una ciudad o un pueblo. En general, un punto de acceso Wi-Fi corriente no suele pasar de los 50 metros de distancia de alcance, distancia que se puede

ver acortada, debido a que las redes Wi-Fi son susceptibles a perder potencia al atravesar obstáculos, como paredes y muebles. Para conseguir más distancia, se pueden utilizar repetidores de señal que amplifican la señal desde cientos de metros hasta algunas decenas de kilómetros de distancia, tal y como muestra la siguiente figura:

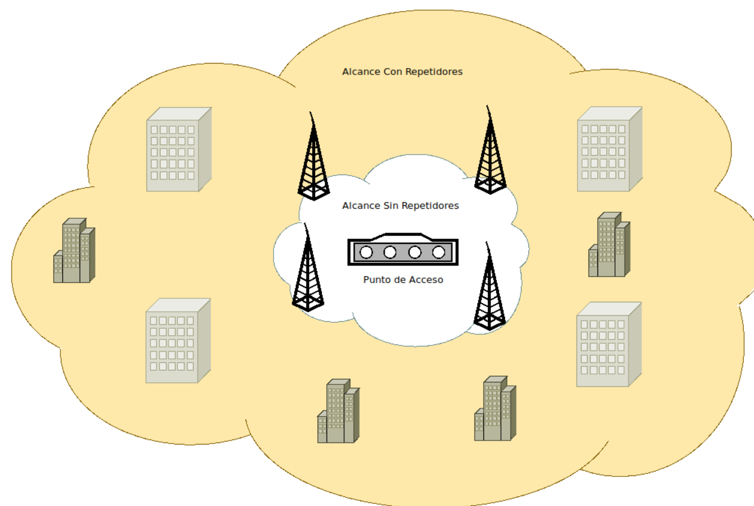


Figura 3.5: Red Wi-Fi con repetidores de señal

Para las redes M2M descentralizadas y distribuidas de pequeño y mediano tamaño las redes Wi-Fi son muy interesantes ya que facilita la gestión de forma aislada de todos los dispositivos de una red privada Wi-Fi, tiene tasas de transferencia muy altas, funciona con el protocolo IP y permite conectar dispositivos a corta (pocos metros) y media distancia (algunas decenas de kilómetros con repetidores). Uno de los inconvenientes que puede tener Wi-Fi es su alto consumo energético para dispositivos muy restringidos con poca batería, Wi-Fi no sería la solución más rentable.

Dado el inconveniente sobre el consumo energético y para ampliar al radio de alcance de un punto de acceso Wi-Fi convencional, la Wi-Fi Alliance para dar soporte a los sensores y actuadores de bajo consumo que funcionan a batería y al Internet de las Cosas, está desarrollando o ha desarrollado pero aún no ha llegado a implantarse, el estándar IEEE 802.11ah (Wi-Fi Halow).

Wi-Fi Halow funciona en frecuencias bajas del orden de los 900 MHz, frente a los 2,4 y 5 GHz actuales, de esta forma la onda de radiofrecuencia es más amplia (se consigue más alcance y atraviesa mejor los obstáculos) y hace falta menos potencia para transmitir en bandas más bajas. En contra, la velocidad de transferencia de Wi-Fi Halow será mucho menor que la Wi-Fi convencional.

3.1.4. RFID y NFC

El término RFID se refiere a Radio Frequency Identifier. En la tecnología RFID, un dispositivo RFID transmite información por ondas de radio a otro dispositivo. Cada dispositivo RFID almacena su identificador único y puede que

otra información de interés. La forma física de estos dispositivos RFID es en forma de etiquetas RFID como la siguiente figura:

Figura 3.6: Etiqueta RFID

Esta forma de las etiquetas, que pueden ser como pegatinas adhesivas de la figura anterior, no es única y puede variar, ya que se pueden encontrar etiquetas RFID en otros colores y otras formas, e incluso en forma de llavero para colgar junto a las llaves. La cuestión es que este envoltorio protege el circuito integrado RFID como la siguiente figura:

Figura 3.7: Circuito Integrado RFID

RFID puede funcionar en diferentes bandas de frecuencia y tienen 3 modos de funcionamiento:

- **Modo pasivo**: la etiqueta RFID no tiene fuente de alimentación propia (batería), la señal que emite el lector es suficiente para alimentar el circuito integrado de la etiqueta RFID y que esta realice una respuesta al lector. La distancia de operación es muy pequeña, de unos pocos centímetros y son mucho más baratas que las etiquetas activas.
- **Modo activo:** la etiqueta RFID tiene fuente de alimentación propia y la utiliza para enviar la señal al lector. El rango de operación es mucho más elevado, de decenas de metros, y este tipo de etiquetas son mucho más caras que las pasivas. Las baterías utilizadas son de larga duración, entre 1 año y 10 años.
- **Modo semipasivas:** este tipo de etiquetas tienen una fuente de alimentación externa pero se comunican con el lector como una etiqueta pasiva (con la energía que proviene del lector). Este tipo de etiquetas permiten construir circuitos integrados más complejos y permiten almacenar más información que las pasivas.

A nivel de consumo actual medio, las etiquetas RFID más utilizadas son las pasivas (debido a su bajo coste) y se utilizan hoy en día para las tarjetas de crédito, llamadas tarjetas contacless, tarjetas de identificación para pasar a salas con acceso restringido y en los bonos de transporte público [11], donde se puede ver el circuito integrado RFID, como en la esquina inferior izquierda de la siguiente figura:

Figura 3.8: Tarjeta de Transporte RFID

En la industria y a nivel profesional, las etiquetas RFID se están utilizando para el inventariado de equipos y piezas y la trazabilidad de mercancías [12], substituyendo a los tradicionales códigos de barras. Aunque las etiquetas RFID son más caras que los tradicionales códigos de barras, la respuesta de una etiqueta RFID suele ser más rápida y sobre todo es más cómoda de leer, debido a que la tecnología RFID es omnidireccional y el lector y la etiqueta RFID no tienen que estar alineados para comunicarse, a diferencia de los códigos de barras y QR.

Para las redes M2M una máquina puede leer la información de la conexión Wi-Fi de red desde una etiqueta RFID para autoconfigurarse y conectarse de forma automática a la red Wi-Fi.

También se puede usar para hacer un seguimiento de máquinas o personas. Por ejemplo, en un museo se puede hacer un seguimiento de las personas, dándoles una tarjeta RFID para entrar al museo por un torno y esta tarjeta se utilizará para salir por otro torno, devolviendo la tarjeta RFID a un empleado del museo. De esta forma se puede saber en qué salas se ha estado más tiempo, cuanto tiempo se ha estado en total en el museo o cuantas personas hay en una sala en un momento determinado, conformando una red M2M de etiquetas RFID. Para ello, se puede instalar un lector RFID en cada puerta de acceso de cada sala para realizar el seguimiento.

La tecnología NFC (Near Field Communication) es tecnología inalámbrica RFID de corto alcance de unos pocos centímetros que funciona en la banda base de los 13,56 MHz. NFC se ha popularizado debido a su integración con los teléfonos móviles. Hoy en día se utiliza el NFC para realizar compras con el móvil en comercios (en vez de usar la tarjeta de crédito) y como bono de transporte, para utilizar el móvil como si fuera una tarjeta de transporte de la figura 3.8.

Para una red M2M las aplicaciones de NFC pueden ser más o menos las mismas que con las tecnologías RFID y los códigos de barras, pero con un radio de alcance muy limitado usando un teléfono móvil. Por ejemplo, siguiendo con el ejemplo del museo, cada obra de arte podría tener un etiqueta NFC de manera que nos pasase una URL que abriríamos en el navegador web de nuestro teléfono con información ampliada de la obra de arte, de esta forma cada obra de arte se puede entender como un nodo NFC de la red, esto sería similar al uso de los códigos QR, pero es mucho más cómodo y rápido al no tener que escanear el código con la cámara del móvil.

3.1.5. ZigBee

ZigBee es un competidor directo con Bluetooth en cuanto a que ZigBee, al igual que el Bluetooth, se utilizan para redes de área personal (PAN). ZigBee es una tecnología inalámbrica omnidireccional de bajo consumo energético, baja tasa de transferencia de datos (250 Kb/s) y baja latencia de red, lo que los hace ideales para el uso en la domótica donde hay sensores y actuadores que necesitan consumir poca energía, no necesitan transmitir grandes volúmenes de datos y se necesita una respuesta lo más rápida posible [13].

ZigBee está compuesto de varios tipos de dispositivos a nivel lógico que conforman una red ZigBee [14]:

- **Coordinador**: sólo hay un coordinador por red. Controla la red y mantiene los caminos o saltos para llegar a los dispositivos finales. Según el IEEE 802.15.4, el coordinador, se conoce en ZigBee como un dispositivo de funcionalidad completa que no usa batería, si no que va conectado directamente a la red eléctrica.
- **Router**: pueden haber varios routers en una misma red. Se encargan de redirigir el tráfico a los dispositivos finales desde el coordinador, un router puede gestionar las rutas de varios dispositivos finales o de más routers. También ejecutan código de usuario y ofrece el nivel de aplicación. Según el IEEE 802.15.4, este dispositivo se conoce también como un dispositivo de funcionalidad completa.
- **Dispositivo Final**: dispositivos de funcionalidad reducida que funcionan a batería. Son los sensores y actuadores de la red.

Existen 3 tipos de topologías de red en ZigBee:

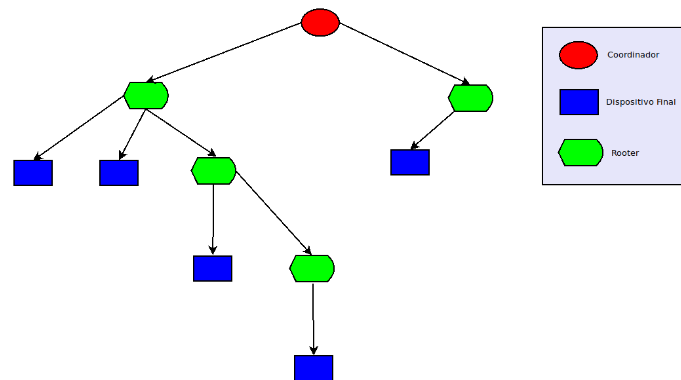


Figura 3.9: ZigBee. Topología en árbol

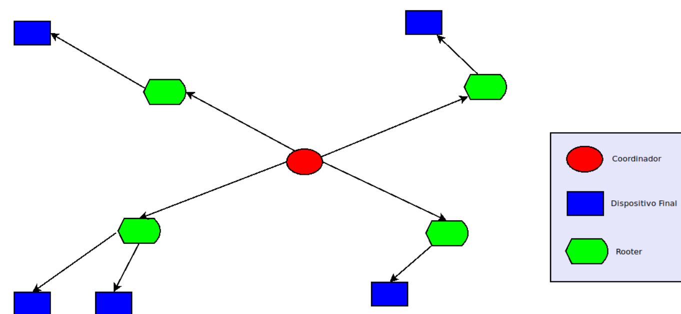


Figura 3.10: ZigBee. Topología en estrella

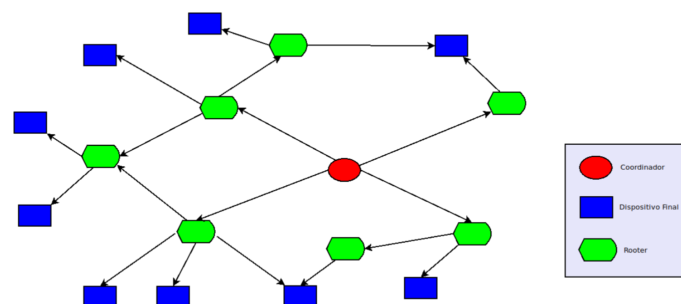


Figura 3.11: ZigBee. Topología en malla

La topología más interesante es la topología en malla, en el que la red puede crecer con más libertad y en la que pueden existir distintos caminos para llegar a un dispositivo final, de forma que si cae un nodo intermedio, se puede llegar por otro camino.

La topología en estrella es un tipo especial de topología en árbol, en estrella es como un árbol en el que la profundidad del árbol llega hasta el nivel 2. Los routers de la figura 3.10, se pueden eliminar y dejar sólo los dispositivos finales y el coordinador.

La topología en árbol tiene la ventaja de que se puede extender con más niveles, este tipo de topología en la jerga de redes de comunicaciones se conoce también como topología en estrella extendida. El mayor inconveniente de ZigBee es que si cae el coordinador nos quedamos sin red, al contrario de Bluetooth, en el que todos los dispositivos son autónomos. Pero esto también ocurre en una red Wi-Fi, si cae el punto de acceso Wi-Fi, la red cae por completo.

ZigBee está cogiendo popularidad debido a que varios fabricantes están usando esta tecnología en sus dispositivos, como los Amazon Echo y los Philips Hue, se puede buscar en Amazon la palabra ZigBee y comprobar los resultados. Para los fabricantes, es una tecnología interesante, debido a que ZigBee requiere de menos electrónica (son más baratos) que Bluetooth en los dispositivos finales y puede albergar a 65535 dispositivos distribuidos en subredes de 255 dispositivos cada una, muchos más dispositivos que Bluetooth.

En la práctica, el coordinador asumirá el rol de router, y cualquier dispositivo final puede ser también router para conectar dispositivos y formar redes más grandes, independientemente de que pueda haber alguna máquina que asuma sólo el rol de router. Si buscamos las bombillas de Philips Hue en Internet, sólo existe una máquina, llamada puente, que se entiende como pasarela (es el coordinador y el router) para llegar a los sensores y actuadores de Philips y que ejecuta el código de usuario.

3.1.6. Z-Wave

Mientras que ZigBee es un protocolo de conexión de un estándar abierto por la IEEE 802.15.4, Z-Wave se trata de un protocolo de conexión cerrado gobernado por la Alliance Z-Wave, que garantiza la interoperabilidad de los dispositivos que utilizan esta tecnología. Z-Wave se usa con el mismo fin que ZigBee, para la domótica. Z-Wave tiene presencia también en el mercado actual, se puede buscar en Internet Z-Wave para comprobar todos los gadgets que usan esta tecnología.

Al ser Z-Wave un protocolo cerrado no hay mucha información en Internet de como funciona, pero básicamente es muy similar a ZigBee. Z-Wave funciona con una topología en malla y los tipos de dispositivos son los mismos (coordinador, router y dispositivo final) que ZigBee, aunque en otros sitios de Internet, distinguen sólo dos dispositivos: Controlador y Esclavo. Por otra parte, Z-Wave funciona a una banda de frecuencia distinta a ZigBee.

Al parecer Z-Wave, al igual que ZigBee no utiliza, las ventajas de utilizar el protocolo IP y todo el nivel de aplicaciones ya existente para la capa IP, si no que utiliza una capa de aplicaciones propia, al igual que ZigBee.

Si usamos un protocolo de estas características para una red de máquinas M2M siempre es mejor usar un protocolo abierto como ZigBee, en vez de usar uno cerrado como Z-Wave. Los protocolos abiertos son, en general, más accesibles, más baratos, más fáciles de implantar, más documentados y más fáciles de utilizar que los cerrados.

3.1.7. LoRaWAN

LoRaWAN es un estándar de red de comunicaciones inalámbrica de tipo LPWAN (low-power WAN). Se trata de una tecnología de bajo consumo energético, una baja tasa de transferencia(de 0,3 Kb/s a 50 Kb/s) y de medio alcance para redes WAN (Wide Area Network), de 5 kms con obstáculos y 15 kms sin obstáculos [15].

La topología de red que utiliza LoRaWAN es una red de redes en estrella, tal y como muestra la siguiente figura:

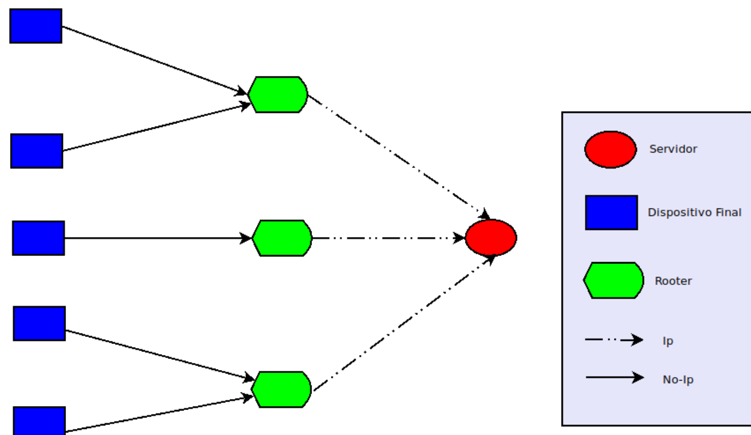


Figura 3.12: LoRaWan. Topología en Estrella

LoRaWAN no utiliza el protocolo IP para conectar los dispositivos finales con los routers y utiliza una pila de protocolos de aplicaciones IoT propia, al igual que lo hace ZigBee y Z-Wave. La comunicación desde los router al servidor o servidores, si que utiliza el protocolo IP.

Los routers son específicos para redes LoRaWAN y en la parte servidora, se trata de un software que se instala en un ordenador personal o un ordenador de tipo servidor. LoRaWAN es un estándar abierto y existen varios servidores opensource.

Los servidores LoRaWAN son plataformas IoT para recoger los datos de los sensores y enviar órdenes a los actuadores. De esta forma se puede desplegar una red de máquinas M2M en un área extensa muy eficiente, sin embargo, no se trata de una red de máquinas descentralizada. Las plataformas IoT se tratan en el capítulo 4.

3.1.8. SigFox

SigFox es una compañía francesa proveedora de redes de tipo LPWAN. Funciona de manera similar a LoRaWAN, donde hay un backend SigFox a modo de servidor/plataforma IoT. El alcance es de unos 30 a 50 kms sin obstáculos y de 3 a 10 kms con obstáculos. La tasa de transferencia es muy baja de 10 a 1000 bps. Al parecer, SigFox tampoco usa el protocolo IP para los dispositivos finales.

Una diferencia muy notable frente LoRaWAN, es que con LoRaWAN podemos desplegar una red LoRaWAN privada (comprando las tecnologías de microchips), mientras que con SigFox, debemos tener cobertura del operador SigFox [16].

Para las redes M2M, se pueden conseguir desplegar muchas máquinas de forma distribuida, pero no se trata de una red descentralizada, debido a la existencia del backend SigFox a modo de plataforma IoT.

3.1.9. Neul

Neul es otra tecnología inalámbrica de tipo LPWAN. La diferencia más notable entre Neul, SigFox y LoRaWAN, son los chips inalámbricos que están instalados en las máquinas finales, cada tecnología de comunicación usa su propio chip. Al igual que sus otros competidores, se trata de una tecnología de bajo consumo energético, baja tasa de transferencia y de medio alcance [17].

3.1.10. Thread

Es un protocolo basado en el IEEE 802.15.4 y en el 6loWPAN centrado en la domótica. Se trata de un protocolo basado en el protocolo IP para IoT que funciona en redes inalámbricas de baja frecuencia para los sensores y actuadores de bajo consumo [18]. Se trata pues de un claro competidor de ZigBee, Z-Wave, Wi-Fi Hallow y Bluetooth low energy por el dominio de los protocolos IoT y las redes M2M.

Esta tecnología es relativamente nueva pero hay un proyecto en marcha de Connected Home over IP que está siendo impulsado por Google, Amazon y Apple [19]. También está siendo impulsado por ZigBee, el cual tiene que estar interesado en que se sigan vendiendo sus chips inalámbricos.

3.2. TECNOLOGÍAS SOFTWARE PARA LA INTEROPERABILIDAD ENTRE PROCESOS EN REDES IP M2M

La interoperabilidad se define como la capacidad de que dos o más máquinas (procesos) puedan comunicarse entre sí. Para comunicar máquinas, dispositivos u ordenadores directamente sin nodos centrales, tenemos varias posibilidades. Se analizan y se investigan desde los tradicionales servicios web que han sido usados en redes de computadores y actualmente en smartphones, hasta algunas tecnologías y protocolos IoT basados en el protocolo IP para dispositivos restringidos.

3.2.1. Servicio Web SOAP

SOAP (Simple Object Access Protocol) es un sistema de mensajería sin estado de sesión que usa el protocolo http para el transporte de los datos y en cuyo body http se envía un documento xml con los metadatos SOAP y la carga útil del mensaje. Un servicio web es una tecnología que proporciona servicios de red transportados por http como forma de construir aplicaciones de red de componentes distribuidos (servicios) independientes del lenguaje y de la

plataforma [20]. Con ejemplos se entiende y se explica mejor (lo que está en negrita es lo más relevante de las figuras 3.13 y 3.14):

```
POST http://localhost:8080/AOSServer/SOAPWebServiceGD HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
Content-Length: 615
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Figura 3.13: Petición raw http por POST a un servidor SOAP

```
HTTP/1.1 200 OK
Server: GlassFish Server Open Source Edition  5.0
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition  5.0  Java/Oracle Corporation/1.8)
Content-Type: text/xml; charset=utf-8
Transfer-Encoding: chunked
```

Figura 3.14: Respuesta raw http de un servidor SOAP

Las figuras anteriores es una petición con su respuesta mediante el protocolo http, lo más importante es el Content-Type que es text/xml, es decir que en el body de la petición http viaja un documento xml y en la respuesta viaja también de vuelta otro documento xml para el cliente, estos documentos xml son como las siguientes figuras:

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:soap="http://soap.ws.aos.uned.es/">
    <soapenv:Header/>
    <soapenv:Body>
        <soap:createDoc>
            <createDocRequestBean>
                <extensionDocumento>pdf</extensionDocumento>
                <fichero>RXN0byBlcyB1biBmaWNoZXJvIGRlIHRleHRvIHBhcmEgaGFjZXIgcHJ1ZWJhcw==</fichero>
                <idAplicacion>id app</idAplicacion>
                <nombreDocumento>test name</nombreDocumento>
            </createDocRequestBean>
        </soap:createDoc>
    </soapenv:Body>
</soapenv:Envelope>
```

Figura 3.15: Petición SOAP del body http

```xml
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:createDocResponse xmlns:ns2="http://soap.ws.aos.uned.es/">
            <createDocResponseBean>
                <idDocumento>id doc</idDocumento>
                <respuestaServicio>OK</respuestaServicio>
            </createDocResponseBean>
        </ns2:createDocResponse>
    </S:Body>
</S:Envelope>
```

Figura 3.16: Respuesta SOAP del body http

En la figura 3.15 se puede observar el documento xml de la petición SOAP. Este documento xml tiene varios elementos:

- **Envelope**: es el elemento raíz del documento xml de la petición SOAP y tal y como su nombre sugiere es una envoltura (elemento raíz) de todo el documento xml de la petición SOAP.
- **Header:** en este nivel se establecen los metadatos para enviar al servidor. En la figura 3.15, está vacío, no hay headers. En el header podemos introducir elementos propios o bien algunos ya establecidos

por OASIS, un organismo que publica los estándares SOAP. Los headers se usan mucho para securizar los servicios web y la trazabilidad de las peticiones.

- **Body**: en el body viaja la información útil de la petición que tiene varias partes:
  - **createDoc**: es el nombre de la invocación remota.
  - **createDocRequest**: es un objeto que contiene los siguiente atributos, que son los que tienen la información útil de la petición:
    - **extensionDocumento**.
    - **fichero**: cadena de texto codificada en base 64.
    - **idAplicacion**.
    - **NombreDocumento**.

Lo más relevante es que el objeto createDocRequest ha sido serializado para ser enviado en el body de una petición http con formato xml.

La respuesta de la figura 3.16 es similar, tiene el envelope sin header y tiene el objeto createDocResponse con dos atributos. El objeto del payload también ha sido serializado en la respuesta.

SOAP se diseñó con estas premisas para que distintos lenguajes de programación y distintos sistemas operativos se comunicasen entre sí, consiguiendo una interoperabilidad muy alta. Sin embargo, cuando buscamos servidores SOAP para Android no encontramos ninguno, así pues, si queremos usar SOAP en nuestras redes M2M descentralizadas y distribuidas tendríamos que construir uno para Android o para la plataforma sobre la que vayamos a trabajar.

Quizás no haya servidores SOAP para Android porque, hoy en día, SOAP se está enfocando más a entornos empresariales y corporativos, donde la comunicación se produce entre ordenadores personales y distintos servidores de tipo computadora para transferir datos, y se puede implementar una seguridad muy fuerte usando certificados de aplicación, para firmar con la clave privada del certificado la cabecera SOAP y cifrar el body de SOAP, independientemente de si usamos certificados a nivel de servidor como SSL o TLS.

Para finalizar, comentar que si bien es cierto que no hay servidores SOAP para Android, si que existen clientes SOAP para consumir servicios web SOAP desde Android.

3.2.2. Servicio Web REST

REST es una alternativa a SOAP para la interoperabilidad de máquinas heterogéneas con distintas arquitecturas, distintos sistemas operativos y distintos lenguajes de programación utilizando tecnología de servicios web.

REST utiliza también llamadas http sin estado de sesión. En SOAP, en general todas las invocaciones remotas se hacen por el método POST del protocolo http. En REST, se utilizan otros métodos, los básicos serían los siguientes, ya que hay otros menos comunes y parece que hay cierto interés en que cada vez haya nuevos métodos http:

- **POST**: crea un nuevo recurso.
- **GET**: consigue un recurso.
- **DELETE**: borra un recurso.
- **PUT**: modifica un recurso.

Un recurso es un dato, una fila de una base de datos o un fichero de texto o de cualquier tipo. Algunos ejemplos de peticiones http con REST pueden ser las siguientes:

```
POST http://localhost:8080/AOSServer/RestWebServiceGD/createDoc HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: application/json
Content-Length: 182
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Figura 3.17: Petición raw POST a un servidor REST

```
GET http://localhost:8080/AOSServer/RestWebServiceGD/readDoc?idAplicacion=id%20app&idDocumento=id%20doc HTTP/1.1
Accept-Encoding: gzip,deflate
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Figura 3.18: Petición raw GET a un servidor REST

```
DELETE http://localhost:8080/AOSServer/RestWebServiceGD/deleteDoc/id%20App/id%20Doc HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: application/json
Content-Length: 0
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Figura 3.19: Petición raw DELETE a un servidor REST

```
PUT http://localhost:8080/AOSServer/RestWebServiceGD/updateDoc HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: application/json
Content-Length: 143
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Figura 3.20: Petición raw PUT a un servidor REST

En las peticiones POST y PUT la carga útil viaja en el body con un Content-Type application/json, es decir, la carga útil http es un documento json. Mientras que en el método GET, la carga útil viaja en la propia URL de la petición. Una respuesta de alguno de las invocaciones anteriores puede ser la siguiente:

```
HTTP/1.1 200 OK
Server: GlassFish Server Open Source Edition  5.0
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition  5.0  Java/Oracle Corporation/1.8)
Content-Type: application/json
Content-Length: 93
```

Figura 3.21: Respuesta raw de servidor REST

La figura anterior el servidor ha devuelto 200 (OK) y como en las peticiones, en la respuesta viaja un documento json (Content-Type). Un ejemplo de documento json puede ser el siguiente, que corresponde con la petición POST:


```
{
  "extensionDocumento": "pdf",
  "fichero": "RXN0byBlcyB1biBmaWNoZXJvIGRlIHRleHRvIHBhcmEgaGFjZXIgcHJ1ZWJhcw==",
  "idAplicacion": "id app",
  "nombreDocumento": "test name"
}
```


Figura 3.22: Documento json de la petición POST

Vistas las figuras anteriores, REST es, en general, más sencillo que SOAP, debido principalmente a que no necesitamos un analizador para examinar la carga SOAP (envelope). SOAP se empieza a complicar cuando necesitamos analizar el header. En los ejemplos anteriores de SOAP, el header estaba vacío. Un ejemplo de petición SOAP con header es la autenticación básica de OASIS, como la siguiente figura:


```
<soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
                   xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
                   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
        <wsu:Timestamp wsu:Id="TS-6E6AF83BAB514E1647156113815061517">
            <wsu:Created>2019-06-21T17:29:10Z</wsu:Created>
            <wsu:Expires>2019-06-21T18:02:30Z</wsu:Expires>
        </wsu:Timestamp>
        <wsse:UsernameToken wsu:Id="UsernameToken-6E6AF83BAB514E1647156113815061416">
            <wsse:Username>user</wsse:Username>
            <wsse:Password
                    Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">
                pass
            </wsse:Password>
            <wsse:Nonce
                    EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary">
                Q2Smy94tLjwkl75ubS9HMA==
            </wsse:Nonce>
            <wsu:Created>2019-06-21T17:29:10.609Z</wsu:Created>
        </wsse:UsernameToken>
    </wsse:Security>
</soapenv:Header>
```


Figura 3.23: Petición SOAP con autenticación básica OASIS

OASIS es un consorcio internacional sin fines de lucro que orienta el desarrollo, la convergencia y la adopción de los estándares de e-business [21].

En la práctica, se utilizan librerías de terceros para analizar este tipo de cabeceras SOAP. Si el lenguaje de programación que estamos usando, no tiene soporte para analizar las cabeceras SOAP, tendremos que construir un analizador. En java, existe la librería WSS4J para este propósito, se pueden consultar más información en [22].

En REST, es más sencillo si se utilizan cabeceras, estas viajan como un parámetro más de la petición http, como la siguiente figura que se envía el parámetro api-key:

```
DELETE http://localhost:8080/AOSServerSecure/RestWebServiceGD/deleteDoc/id%20App/id%20Doc/ HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml
api-key: soy un api-key de prueba
Content-Length: 0
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Figura 3.24: Petición REST http con el parámetro api-key

Para hacer funcionar REST en dispositivos y más concretamente en Android, necesitamos un servidor http, pero no nos sirve un servidor http que se instala directamente con un apk en Android.

Necesitamos un servidor http de bajo nivel, existen estos servidores http en Android, en forma de dependencias Java, cuyas respuestas se pueden manipular a nivel de código fuente y con los que podremos proporcionar una respuesta dinámica a los clientes, y dada una petición, poder ejecutar código en el propio dispositivo para la lectura de los sensores y proporcionar órdenes a los actuadores. Algunos de estos servidores http de bajo nivel para Android son:

- **OKhttp**
- **android-async-http**
- **Google Volley**

De las 3 anteriores se ha probado android-async-http, en otro trabajo del máster en la asignatura de computación ubicua y el resultado ha sido bueno en pruebas de implementación de un prototipo.

Para finalizar este apartado, la tendencia es que los API REST se están enfocando más a las apps móviles y la comunicación con los servidores y para la transferencia de datos entre un navegador web y un servidor. La razón es que REST es más ligero (una petición REST lleva menos carga de metadatos que SOAP, como todo el envelope de SOAP), es más fácil de utilizar en los dispositivos móviles y REST se suele utilizar junto a documentos json que son más fáciles de analizar con JavaScript que los documentos xml de SOAP.

3.2.3. Websockets

Se trata de un tipo especial de socket TCP persistente que funciona en el puerto 80, aunque nada impide usarlo en otro puerto. Por persistente, se entiende que un cliente inicia la conexión con un servidor, pero la conexión TCP se mantiene entre ambos, cliente y servidor. De esta forma el cliente puede enviar información al servidor y al mismo tiempo, el servidor puede enviar información al cliente, consiguiendo una comunicación bidireccional full duplex. Los websockets están diseñados para ser utilizados en un servidor de aplicaciones web y los clientes pueden ser navegadores web, pero también pueden ser utilizados entre dos procesos cualquiera que no tengan que ver con la web.

En el protocolo http normal, las peticiones de los clientes no son persistentes y cuando el servidor ofrece una respuesta al cliente, ambos cierran la conexión en curso. Si es necesario se abre otra conexión entre ambos para la transmisión de datos. Este escenario de no persistencia es correcto, porque un servidor puede recibir miles de peticiones de muchos clientes y mantener las conexiones consumen recursos de la máquina servidora, y es mejor liberar las conexiones para ahorrar recursos del servidor. También se hace así por razones de seguridad, realizar un ataque de denegación de servicio con conexiones persistentes sería mucho más sencillo que con sockets no persistentes.

Para las redes M2M y el IoT puede ser muy útil utilizar estos sockets persistentes, en OMA Lightweight M2M, se ha comprobado que una vez enlazado el cliente con el servidor, el servidor puede enviarle peticiones a los dispositivos clientes, para obtener información de los sensores o para enviarle órdenes a los actuadores. La comunicación se simplifica mucho, cada dispositivo sólo tiene que tener un cliente y enlazarse con la plataforma IoT, es decir, no es necesario que un sensor implemente una parte servidora para recibir peticiones y otra parte cliente para comunicarse con el servidor.

Para un escenario descentralizado, donde no exista una plataforma IoT, los sensores pueden tomar el rol de servidores y los actuadores de clientes, una vez enlazados, ambos pueden comunicarse. Como se ha dicho, las conexiones persistentes consumen espacio en la memoria RAM del dispositivo y habría que experimentar si este modelo, finalmente, resulta más rentable en dispositivos restringidos que disponen de muy poca memoria RAM.

Para la plataforma Android existen los servidores http, de los que se habló en los servicios web REST, en forma de dependencias Java y que soportan los websockets, como android-async-http y Okhttp.

3.2.4. MQTT

MQTT (Message Queue Telemetry Transport) es un protocolo ligero basado en el paso de mensajes entre diversos actores/dispositivos por el método del publicador/subscriptor, también conocido como productor/consumidor [23]. En esencia, no es un protocolo nuevo, debido a que ya existen protocolos similares de este tipo como JMS (Java Messaging Service) que usan diversas implementaciones como, entre otras, ActiveMQ y HornerQ. Sin embargo, MQTT está optimizado para consumir muy poco ancho de banda y para dispositivos restringidos con ciertas limitaciones en la capacidad procesamiento y batería, es decir, está especialmente diseñado para ser utilizado en dispositivos IoT y redes M2M.

Por tanto, MQTT es un protocolo muy interesante para dispositivos en redes M2M grandes y que también puede utilizarse para redes pequeñas. MQTT necesita de un nodo central, llamado broker, que recibe los mensajes y los reenvía a los receptores/consumidores. Funciona con tópicos o temas. El funcionamiento en este tipo de protocolos es el siguiente:

- Un tópico o tema es una cola de mensajes que tiene el broker (nodo central). En un tópico pueden haber 0 o más productores y 0 o más consumidores.
- Se pueden tener tantos tópicos como se desee en el broker.
- Un productor se registra como tal en un tópico, con el fin de enviar mensajes a los consumidores.
- Un consumidor se registra como tal en un tópico, con el fin de recibir mensajes de los productores.
- Los productores y los consumidores no se conocen entre sí, es el broker (nodo central), el que relaciona los mensajes entre los dispositivos.
- Se trata por tanto de un protocolo de muchos a muchos, donde un productor o varios productores envían un mensaje al tópico que es recibido por todos los consumidores.

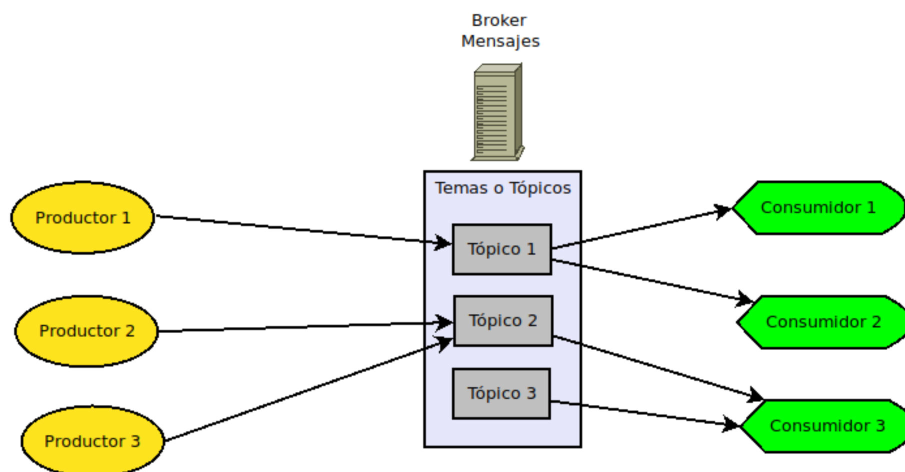


Figura 3.25: Comunicación con un broker para el paso de mensajes con el protocolo MQTT

En la figura anterior, se puede observar como:

- El **Tópico 1** tiene un productor y dos consumidores
- El **Tópico 2** tiene dos productores y dos consumidores
- El **Tópico 3** tiene cero productores y un consumidor.

Por tanto, la comunicación de los dispositivos se hace a través del Broker. Cuando un dispositivo produce un mensaje para un tópico del broker, el broker reenvía el mensaje a todos los consumidores que se han subscrito al tópico. La comunicación es asíncrona, el productor envía el mensaje pero no sabe si los consumidores lo han recibido o que consumidores lo reciben, esto puede ser un inconveniente en algunos escenarios cuando el productor necesita una respuesta del consumidor, este modelo de comunicación no lo

proporciona y hay que implementar de forma separada los callbacks de la respuesta.

En general, estos protocolos de productor/consumidor se disparan cuando se produce algún evento con cierta importancia. En nuestra arquitectura, no tendría porque ser así. Por ejemplo, podríamos crear un tópico de temperatura en el broker, entonces un sensor podría enviar cada cierto tiempo y de forma regular la temperatura que detecta al tópico de la temperatura. En el otro lado estarían los consumidores de la temperatura que se ha subscrito al tópico de temperatura para recibir el mensaje de la lectura de la temperatura y realizar alguna acción con la temperatura, en caso de ser necesario.

MQTT es un protocolo muy interesante para redes M2M, aunque la red M2M resultante es distribuida pero no es descentralizada debido a que el broker es un componente software que tenemos que tener instalado en algún dispositivo que hace de middleware de todos los mensajes entre los productores y los consumidores. Sin embargo, están fabricando routers con el protocolo MQTT incorporado a modo de broker de los mensajes, de manera similar a que hace tiempo que los routers incorporan un servidor DHCP (protocolo para la gestión y asignación dinámica de direcciones IP de los clientes). Esto enmascara o disimula la centralización de los mensajes del broker en un dispositivo de tipo router. Router que tenemos que tener obligatoriamente para las comunicaciones, si este router incorpora MQTT no tenemos que tener una máquina o dispositivo en exclusiva a modo de broker. De momento, hay routers de este tipo a nivel industrial y de momento no parece que hayan routers a nivel de consumo medio, por ejemplo puntos de acceso Wi-Fi con el software de broker MQTT instalado. Se puede consultar en Internet router MQTT para ver los modelos que existen en el mercado.

Otro punto a considerar para usar MQTT es si existe una librería que ayude en la implementación del lenguaje de programación que vamos a utilizar para nuestra red de dispositivos. Esta librería tendría que estar instalada en los productores y en los consumidores, es por tanto una librería de tipo cliente para producir y consumir los mensajes del broker. Existe una librería desarrollada por Eclipse llamada Paho para algunos lenguajes de programación y plataformas como Java, Android y C, entre otros lenguajes. Se puede consultar la lista completa en [24] de los lenguajes soportados, aunque hay bastantes, faltan algunos como PHP e IOS, sin embargo, para PHP tenemos la librería phpMQTT y hay otra dependencia para IOS llamada CocoaMQTT.

Respecto al software que se instala en una máquina de tipo computadora para tener un broker, existe Mosquito, también desarrollado por Eclipse que está disponible para todas las plataformas como Windows, Linux, Mac e incluso para la versión Debian de una Rasberry PI. Con lo cual, la interoperabilidad de MQTT entre los distintos lenguajes de programación y plataformas software es muy alta.

3.2.5. DDS

DDS (Data Delivery Service) es otro protocolo basado en suscripción y publicación, similar a MQTT, que necesita también de un middleware para el paso de mensajes y que ha sido diseñado para IoT. Por los resultados obtenidos durante la investigación, se está usando mucho más MQTT para la interoperabilidad de dispositivos, debido principalmente a que DDS no es gratuito.

DDS ha sido diseñado por la OMG al igual que CORBA [25]. DDS parece que tiene muchos componentes y plugins y parece un poco complejo de utilizar, al menos en una primera impresión.

Dado que DDS no es gratuito, y como consecuencia, la experimentación a nivel de implementación de los prototipos con este protocolo va a ser muy limitada, no se continúa investigando este protocolo.

3.2.6. AMQP

AMQP son las siglas de Advanced Message Queuing Protocol y se trata de otro protocolo basado en el envío de mensajes de tipo productor/consumidor mediante colas. De forma general, los protocolos de tipo productor/consumidor de tipo cola funcionan como la siguiente figura:

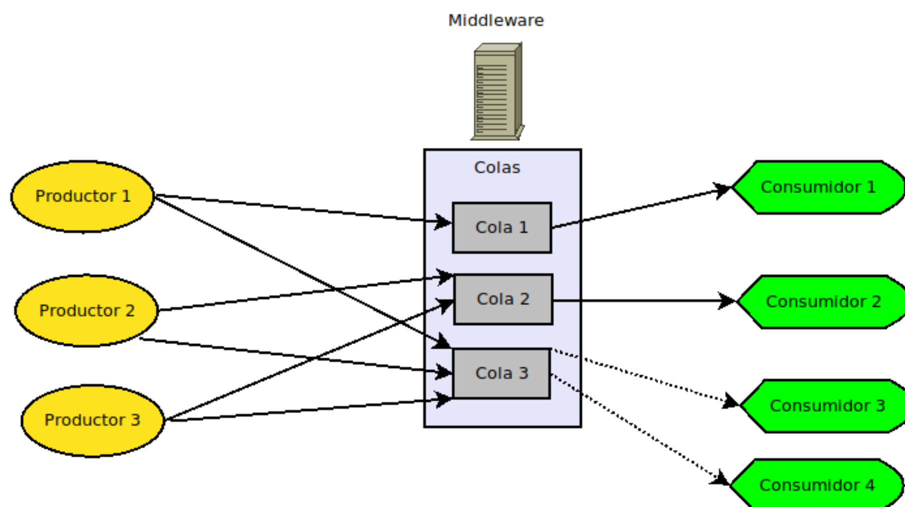


Figura 3.26: Protocolo del tipo productor/consumidor con colas para el paso de mensajes

En la figura anterior, se observa como:

- La cola 1 tiene un productor y un consumidor.
- La cola 2 tiene dos productores y un consumidor.
- La cola 3 tiene tres productores y 2 consumidores. Las líneas discontinuas indican que sólo uno de ellos procesará cada mensaje (ver explicación en el párrafo siguiente).

En este modelo de colas, pueden haber varios productores registrados a una cola, y varios consumidores registrados a una cola pero la diferencia fundamental respecto a los tópicos, como en MQTT, es que sólo un consumidor recibe y procesa el mensaje. Es decir, que si hay varios consumidores registrados a una cola, sólo uno de ellos recibe y procesa el mensaje y los otros consumidores no lo reciben ni lo procesan. La rotación entre unos consumidores y otros de una misma cola la hace el middleware que gestiona las colas, puede ser una estrategia por Round Robin, o bien otras estrategias más sofisticadas, como enviar al consumidor que esté menos ocioso o ocupado.

Este tipo de arquitecturas de mensajes se utilizan, especialmente, en entornos de alto rendimiento y mucha carga de trabajo donde hay muchos productores y pocos consumidores. Los consumidores se ven desbordados por las peticiones concurrentes de los productores y mediante las colas se pone orden a la ejecución de las peticiones con la cola FIFO, de esta forma los consumidores ejecutan petición a petición por orden de llegada y no tienen que procesar todas las peticiones al mismo tiempo. Si el consumidor está caído o fuera de línea, el middleware se guarda las peticiones a la espera de que el consumidor pueda atender las peticiones. La arquitectura es muy fácilmente escalable, porque si el consumidor de una cola está desbordado, podemos añadir más consumidores para balancear la carga de trabajo de la cola entre más consumidores. Al ser las colas (y los tópicos) un sistema asíncrono, donde no es del tipo petición-respuesta síncrona, si se necesita una respuesta por parte de los consumidores, hay que utilizar callbacks para ofrecer una respuesta a los productores.

Para las redes M2M, tiene más sentido en un entorno de sistemas centralizados con plataformas IoT, donde los dispositivos son los productores y las plataformas IoT son los consumidores.

Si quisiéramos crear un entorno de comunicación descentralizado, cada dispositivo debería tener su propia cola para consumir los mensajes, es decir, si tenemos 20 dispositivos, deberíamos de tener 20 colas, que serían gestionadas en el middleware.

Existen dos versiones de este protocolo con bastantes diferencias que se comentan en las siguientes subsecciones.

3.2.6.1 AMQP 0.91

De forma resumida, este protocolo funciona de la siguiente manera:

- Un productor envía un mensaje al intercambiador.
- El broker en base a los metadatos que contiene el mensaje y algunas estrategias de enrutamiento, envía el mensaje a una o a varias colas en base a las vinculaciones del intercambiador con las colas.
- La cola, es una estructura de datos de tipo FIFO y cuando le llega el turno al mensaje envía el mensaje a un sólo consumidor

En AMQP, se añade una capa más, que es la del intercambiador que enruta el mensaje a la cola o a las colas correspondientes mediante los vínculos del intercambiador con las colas. Es decir, el productor no envía el mensaje a la cola del middleware directamente, si no que el productor envía el mensaje al intercambiador, y este ya lo distribuye a la cola o las colas correspondientes. La principal ventaja de este protocolo es que con una petición de un productor podemos enviar a varias colas y por tanto el mensaje llegará a varios consumidores. En la siguiente figura se puede observar esta arquitectura:

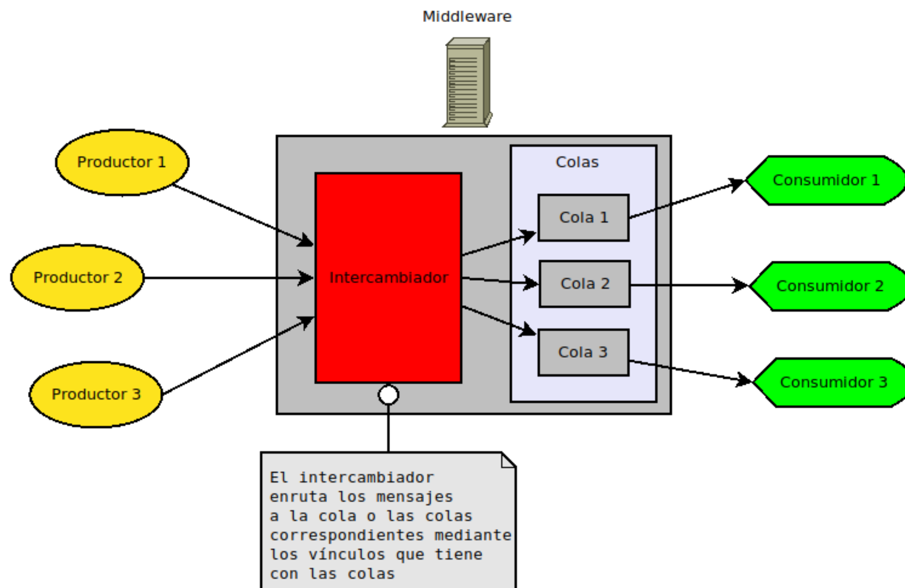


Figura 3.27: Protocolo AMPQ 0.91

En otras arquitecturas pueden existir varios intercambiadores, consiguiendo una arquitectura muy escalable.

3.2.6.2 AMQP 1.0

La versión 1.0 ya ha sido aprobada como protocolo internacional y estándar de OASIS. La versión 1.0 es incompatible y bastante diferente con la versión de la serie 0.x [26], donde al parecer van a eliminar los brokers, los vínculos con las colas y puede funcionar con tópicos y de punto a punto.

Las implementaciones, las librerías y los broker de AMQP en su versión 1.0 escasean y por tanto, no tiene sentido seguir investigando sobre este protocolo en su versión 1.0.

3.2.7. ZeroMQ

ZeroMQ es una librería orientada a mensajes mediante patrones de mensajería, como el productor/consumidor. La diferencia principal con respecto a otras soluciones, como MQTT, es que ZeroMQ puede funcionar sin un broker o un middleware para el paso de mensajes [27] (brokerless y de ahí

viene el Zero), consiguiendo una arquitectura de clientes simples con la suficiente inteligencia para gestionar y enrutar los mensajes.

En el protocolo ZeroMQ se utiliza la noción de socket, pero en ZeroMQ han añadido una capa de abstracción por encima. En términos generales, los sockets tradicionales utilizan datagramas no orientados a conexión síncronos y flujo de bytes (streams) orientados a conexión síncronos. La comunicación de los sockets tradicionales es: de (1) un cliente y un servidor , (2) muchos clientes y un servidor y (3) un cliente y muchos servidores. En ZeroMQ la comunicación puede ser asíncrona y permite una relación de muchos clientes y muchos servidores. El protocolo ZeroMQ soporta los siguientes patrones de mensajería:

- **Request-Replay**: la comunicación es muchos clientes y muchos servidores. Se utiliza para implementar distribución de tareas y llamadas a procedimientos remotos (RPC). Tiene dos sockets síncronos REQ y REP y dos sockets asíncronos DEALER y ROOTER. Estos dos últimos se pueden utilizar con colas.
- **Exclusive Pair**: es como Request-Replay, pero sólo y exclusivamente puede haber un cliente y un servidor. Usa un tipo de socket llamado PAIR.
- **Pipeline**: conecta un cliente con varios nodos de tipo pipes, para que todos juntos y de forma paralela procesen una petición para el servidor. Usa dos sockets llamados PUSH y PULL.
- **Publish-Subscribe**: es el clásico productor/consumidor con tópicos del que ya se ha comentado en otras subsecciones anteriores, pero sin middleware, aunque puede funcionar también con uno. Conecta un productor con varios consumidores. Usa 4 tipos de sockets PUB, XPUB, XSUB y SUB. Los sockets XPUB y XSUB se utilizan para trabajar con brokers, mientras que los sockets PUB y SUB trabajan sin un broker.
- **Client-Server**: se encuentra en estado de borrador. Varios clientes inician la comunicación con un servidor. El servidor puede comunicarse con todos los clientes.
- **Radio-dish**: también se encuentra en estado de borrador. Es como el patrón del productor/consumidor pero en vez de utilizar tópicos se utiliza la noción de grupo. Según parece la diferencia fundamental con el patrón de productor/consumidor es que un consumidor puede entrar en un modo de silencio y descartar los mensajes que recibe al haber superado su límite máximo de recepción de mensajes.

Las siguientes figuras ilustran los distintos patrones de mensajería de ZeroMQ y los sockets ZeroMQ utilizados en los patrones, las ilustraciones han sido extraídas directamente de [28].

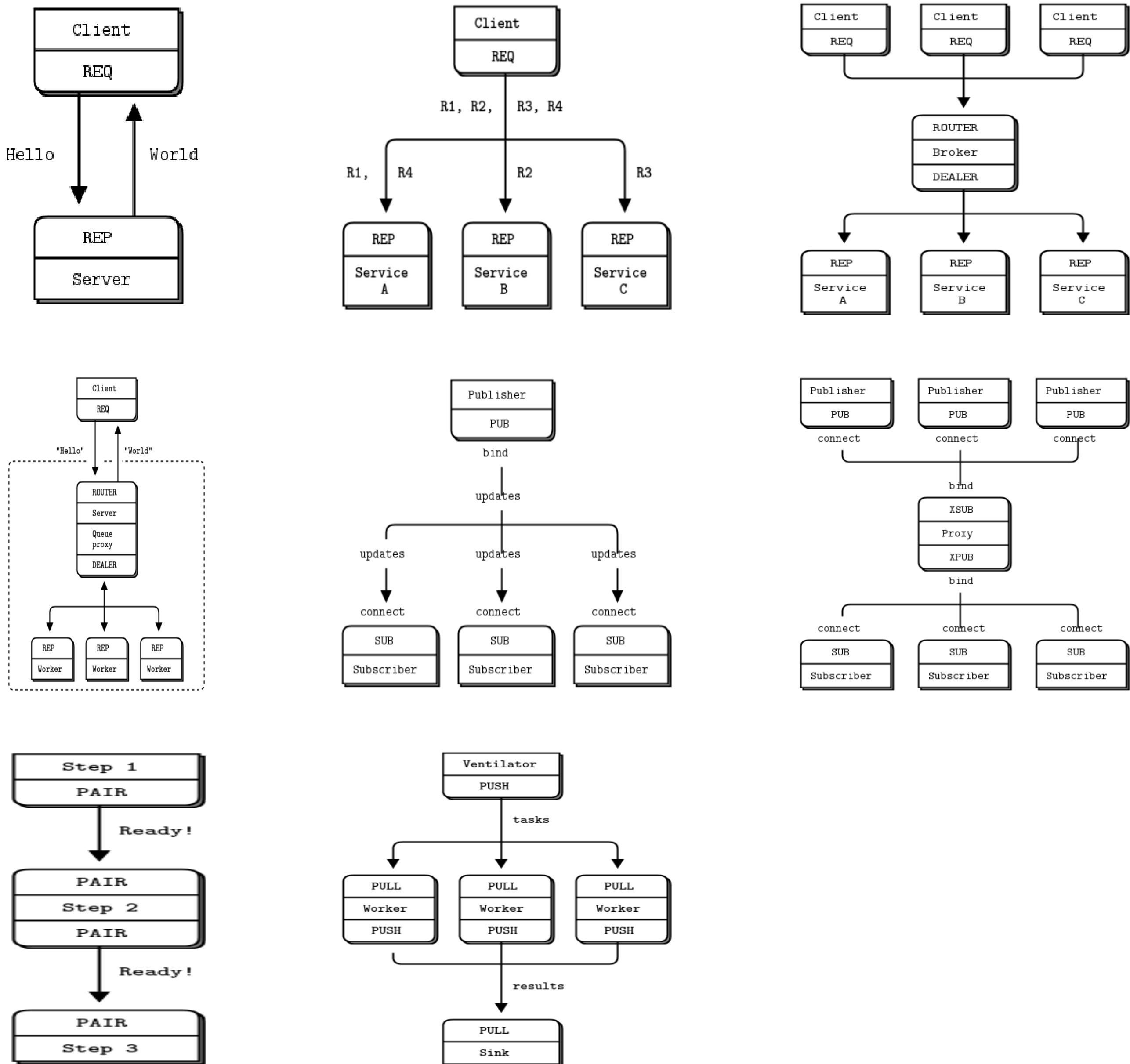


Figura 3.28: Protocolo ZeroMQ. Modelos de Mensajería

Vistas las figuras anteriores de esta sección, se puede observar como el protocolo ZeroMQ es muy completo y puede trabajar de distintos modos. ZeroMQ también puede funcionar con brokers, tal y como sugieren algunas de las figuras anteriores. También puede funcionar con colas y tópicos.

Por otra parte, la librería ZeroMQ está disponible para los lenguajes de programación más populares, entre ellos, Java, C, C++, C# y PHP, y también

está para la plataforma Android, por tanto la interoperabilidad de este protocolo es muy alta.

Para las redes M2M y el IoT, el protocolo ZeroMQ es muy completo y cubre toda la casuística de la comunicación entre los distintos actores. Por ser tan completo y con una naturaleza que puede ser algo distinta de las comunicaciones con sockets tradicionales, el protocolo ZeroMQ puede llevar un tiempo entenderlo y dominarlo en su totalidad.

3.2.8. CoAP

CoAP (Constrained Application Protocol) es otro protocolo nuevo a nivel de aplicación para redes IP diseñado para M2M y el IoT. Como su nombre de Constrained indica (restringido en inglés), es para redes y dispositivos con pocos recursos [29].

Este protocolo funciona de manera muy parecida al patrón de diseño de un API REST utilizado en el protocolo http con sus métodos más corrientes como GET, POST, PUT y DELETE, donde el servidor o un dispositivo, como un sensor, declara sus recursos en forma de URLs, pero en vez de usar el protocolo http se utiliza el protocolo CoAP, es decir, en las URLs, en vez de usar el formato de URL:

- **http://host/recurso**
- Se utiliza el formato de URL: **coap://host/recurso**

Otras diferencias entre http y CoAP es que los mensajes CoAP son, por diseño, más pequeños con longitudes de cabecera (headers) más pequeñas que en http. Los mensajes CoAP viajan en un formato binario que ocupa menos espacio que el simple texto plano sin comprimir utilizado en el protocolo http (ver test más adelante). Otra diferencia es que el protocolo http funciona con TCP, mientras que el protocolo CoAP funciona con UDP.

CoAP puede ver reducida su interoperabilidad, debido a que no podemos realizar una petición CoAP directamente con un navegador web (aunque hay plugins para alguna versión antigua de Firefox) o bien con un proceso estándar que realice peticiones http o un comando del sistema operativo como curl, mientras que con el protocolo http si que podemos hacerlo. Por este motivo, los desarrolladores CoAP han implementado un servidor CoAP a modo de proxy llamado crosscoap que traduce desde el protocolo http a CoAP y a la inversa, desde el protocolo CoAP a http para ganar interoperabilidad entre los procesos de las máquinas que no puedan o tengan la capacidad de usar CoAP.

CoAP tiene alternativas de transporte usando SMS en una red de comunicaciones móviles, mediante una URL con el siguiente formato:

- **coap+sms://numero_telefono/recurso**

CoAP está disponible para los lenguajes de programación más populares, entre ellos, Java, JavaScript, Python y C#. En su página también comentan que la versión de Java llamada Californium funciona en Android.

Como parece un protocolo tan interesante para redes M2M y el IoT se ha analizado una petición http REST y otra CoAP con el fin de comparar ambos protocolos. En [30] tienen una serie de utilidades de Californium que podemos descargar con git y luego compilar con maven. Una vez construidos los artefactos y obtenidos los ejecutables, hay que levantar el servidor CoAP (cf-server-2.1.0-SNAPSHOT.jar) y después lanzar el cliente (cf-browser-2.1.0-SNAPSHOT.jar). El cliente tiene una interfaz gráfica con JavaFX y se ha realizado una petición GET al recurso hello, tal y como muestra la siguiente figura:

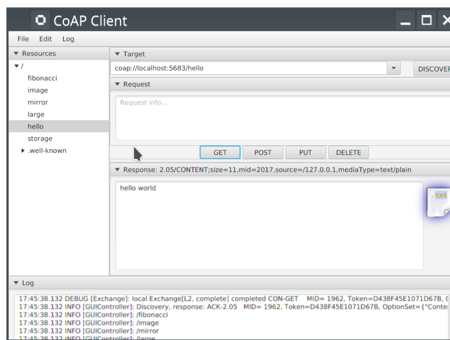
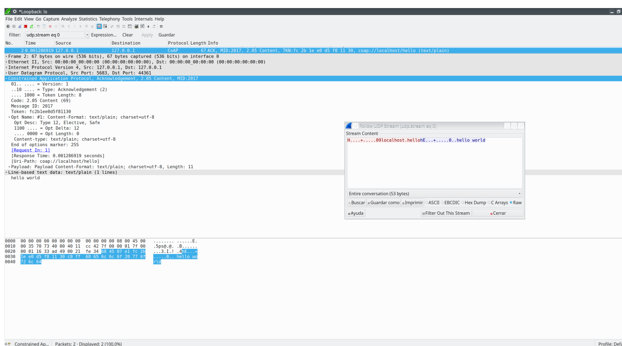


Figura 3.29: Cliente CoAP



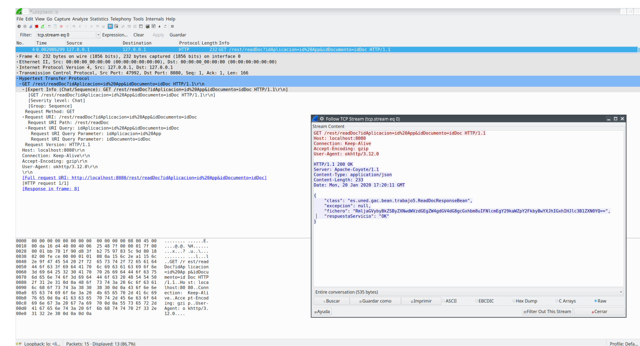


Figura 3.30: Captura Petición (rojo) – Respuesta (azul) CoAP con Wireshark	*Figura 3.31: Captura Petición (rojo) – Respuesta (azul) http REST con Wireshark*

Con Wireshark, un software destinado para analizar paquetes de red, se puede capturar la petición y la respuesta CoAP, ver las figuras anteriores. Las peticiones y respuestas de CoAP y REST no son equivalentes, portan distinta carga en la petición y distinta carga en la respuesta, pero para el experimento son válidas igualmente, ya que tal y como muestran las anteriores figuras, el protocolo CoAP ocupa menos espacio que http REST. El protocolo http tiene más carga de cabeceras en la petición y en la respuesta, mientras que CoAP casi no tiene casi cabeceras en la petición y en la respuesta. También es más

corta la primera parte de la petición CoAP, desde la capa de transporte hasta (UDP) hasta el nivel físico (frame).

La carga útil de CoAP en la petición y en la respuesta viajan en texto plano (text/plain) y no viajan en binario (al menos en la prueba realizada), tal y como sugieren en las especificaciones del protocolo, si no que deben hacer binaria la cabecera, de hecho, sin investigar más en el protocolo CoAP, la cabecera CoAP no se entiende, mientras que en http las cabeceras se pueden leer directamente y son entendibles a simple vista. Otra cosa muy interesante de CoAP es que incorpora un servicio de descubrimiento DISCOVER para descubir de forma automatizada los recursos del servidor. REST no lo tiene incorporado de serie en las propias especificaciones del protocolo, aunque se podría implementar a mano para REST.

CoAP es un protocolo muy interesante para redes M2M con baja latencia de red donde podemos tener una red de sensores y actuadores distribuidos y descentralizados. Un inconveniente de CoAP es una pérdida de interoperabilidad en comparación con el protocolo http que está mucho más extendido.

3.2.9. NNG

NNG es otra librería ligera que implementa varios patrones de mensajería sin brokers, de forma similar a ZeroMQ. NNG es una librería que substituye a nanomsg, con mejoras respecto a su antecesora. Por este motivo, NNG significa nanomsg-next-gen.

No hay mucha información sobre NNG ni sobre nanomsg. Sobre NNG podemos encontrar algo en [31] y [32] donde explican con cierto detalle el protocolo. NNG está especialmente implementado para el lenguaje C, aunque se puede encontrar algo para otros lenguajes de programación. No hay mucho y la interoperabilidad de este protocolo es mucho más reducida que en otros protocolos de este tipo.

Respecto a los patrones de mensajería soportados por NNG y sin entrar en muchos detalles son los siguientes:

- **Pipeline (A One-Way Pipe) *.**
- **Request/Reply (I ask, you answer) *.**
- **Pair (Two Way Radio) *.**
- **Pub/Sub (Topics & Broadcast) *.**
- **Survey (Everybody Votes):** se trata de un patrón de tipo encuesta, donde todos los clientes votan hasta que expira la encuesta y donde los clientes reciben una respuesta individualizada. Se utiliza para servicios de descubrimiento y algoritmos de votación. La siguiente figura ilustra la de este patrón de comunicación (la figura se ha extraído directamente de [31]).

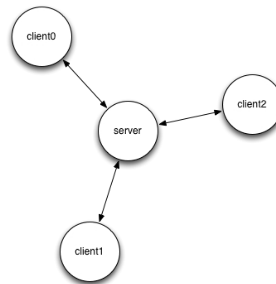


Figura 3.32: Protocolo NNG. Patrón de mensajería survey (encuesta)

- **Bus (Routing):** este patrón se utiliza para conectar todos los nodos de una red como una malla, es decir cada nodo está conectado a todos los demás nodos de la red. Un mensaje enviado por un nodo se envía a todos los demás nodos de la red de malla. La siguiente figura muestra este patrón de mensajería (la figura se ha extraído directamente de [31]).

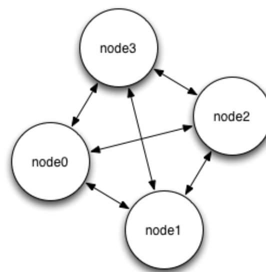


Figura 3.33: Protocolo NNG. Patrón de mensajería bus (Routing)

** Los 4 primeros protocolos son similares a los ya comentados en ZeroMQ. En la guía de ZeroMQ están más ampliamente explicados que en la documentación de NNG, por lo que se remite al apartado de ZeroMQ de este trabajo.*

Comentar que los ejemplos encontrados son escasos y son para el protocolo IPC (Inter-Process Communication). Es decir, para comunicar procesos dentro de la misma máquina, aunque se supone que también funciona sobre TCP.

3.2.10. 6LoWPAN

Esta tecnología es muy interesante porque es independiente del medio de transmisión físico y puede funcionar con el estándar IEEE 802.15.4 [33] de tecnología inalámbrica de bajo consumo como ZigBee, pero también con tecnología Wi-Fi. Más que una tecnología inalámbrica, es la capa de red que faltaba para conseguir la independencia entre el medio de transmisión y la capa de aplicaciones, la independencia la consigue utilizando el protocolo IPv6, de esta forma todos los dispositivos finales como sensores y actuadores pueden tener una dirección IP y se pueden usar todos los protocolos de aplicaciones ya inventados de las capas superiores como http y otros.

Recordemos que ZigBee, Z-Wave, LoRaWAN, SigFox y Neul utilizan protocolos propios de aplicaciones IoT, cada uno el suyo.

6LowPAN surge por la necesidad de que todos los dispositivos, incluyendo los de bajo consumo y bajo poder computacional, como pueden ser los sensores y actuadores, puedan utilizar el protocolo IP en el IoT.

Como su nombre indica, 6LowPAN está pensado para redes PAN, de tipo pequeñas o personales, al igual que ZigBee, pero está red PAN puede formar parte de una red IP más grande como puede ser una red local IP, una intranet IP o Internet, tal y como muestra la siguiente figura:

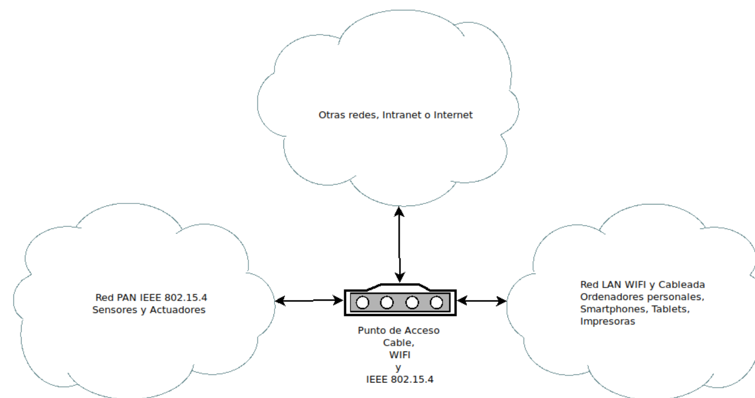


Figura 3.34: Redes 6LoWPAN

Así pues, mientras que el protocolo IP convencional funciona muy bien en dispositivos potentes y conectados a la red eléctrica, 6loWPAN es un complemento perfecto para dispositivos restringidos de bajo consumo que funcionan a batería y de esta forma, construir redes de máquinas M2M descentralizadas y distribuidas de sensores y actuadores. Aunque el IEEE 802.15.4 tiene poco alcance, se pueden desplegar varios puntos de acceso de este tipo, y cada sensor y actuador puede tener una dirección IP

3.2.11. OMA Lightweight M2M

Desarrollado por la Open Mobile Alliance [34], se trata de un protocolo ligero, diseñado para redes M2M y la administración de dispositivos de IoT. OMA Lightweight M2M funciona sobre CoAP y define los recursos de los dispositivos mediante el patrón de diseño de un API REST. Actualmente, existen pocas implementaciones sobre este protocolo, pero hay una desarrollada por Eclipse en Java, llamada Eclipse Leshan y como no hay mucha documentación, de como funciona Leshan ni OMA Lightweight M2M, la investigación se va a orientar a analizar esta implementación de Leshan para comprobar como funciona. En [35] definen el protocolo con las siguientes características:

- Usa por debajo a CoAP
- Soporte de capa de transporte UDP y SMS (al igual que CoAP)

- Modelo de recurso simple basado en objetos (Patrón de diseño API REST)
- Funcionalidades básicas de M2M: servidor LWM2M, control de acceso, dispositivo, conectividad, actualización de firmware, ubicación, estadísticas de conectividad
- Seguridad basada en DTLS (Datagram TLS: proporciona privacidad en la transferencia de datagramas UDP).

En [35] tienen un servidor y un cliente listo para usarse, para ello hay que seguir las instrucciones que hay en [36] y ejecutar el servidor y después el cliente para registrar el cliente en el servidor. Se adjuntan algunas capturas de pantalla del servidor que proporcionan.

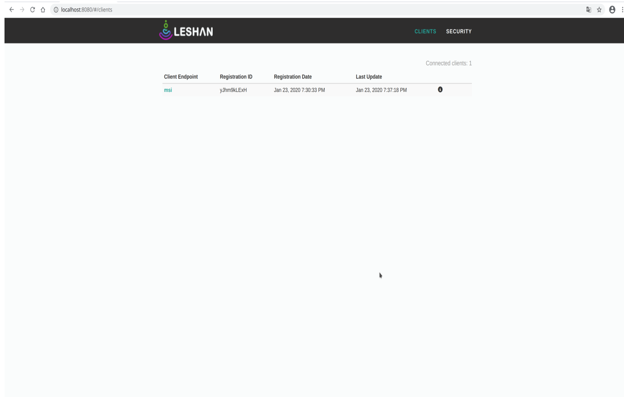


Figura 3.35: Servidor Leshan. Dispositivos conectados (Mi ordenador)

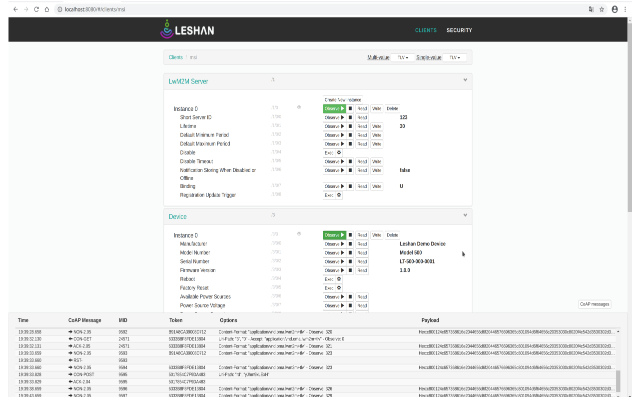


Figura 3.36: Servidor Leshan. Detalle del dispositivo msi

En la figura 3.36 (abajo) se pueden ver las peticiones CoAP con un formato binario, en la columna payload, cosa que no se consiguió ver cuando se analizó el protocolo CoAP en su correspondiente sección. Se han capturado algunos paquetes con Wireshark y el resultado se muestra en las siguientes figuras:

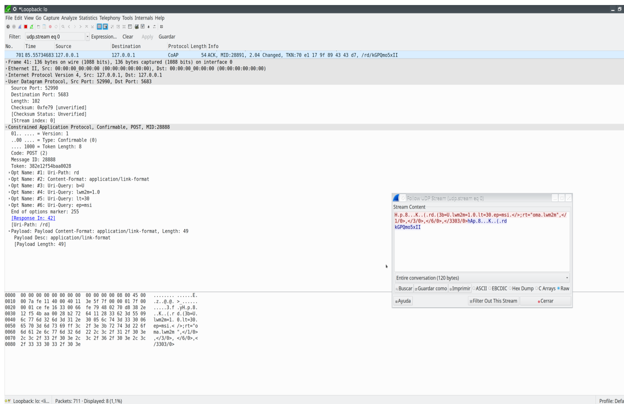


Figura 3.37: Captura de Wireshark cuando se enlaza el cliente con el servidor en Leshan

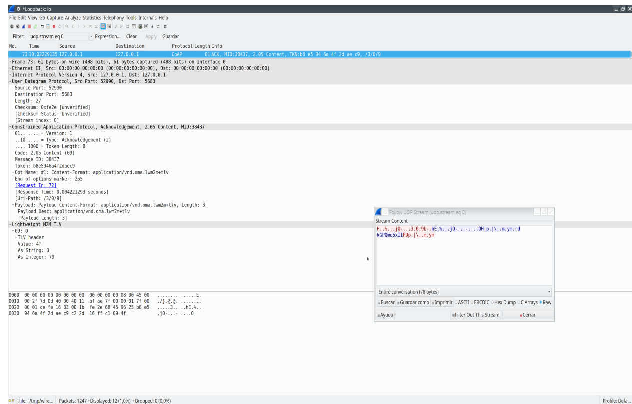


Figura 3.38: Captura de Wireshark cuando se hace clic en el botón read de Battery con un navegador web

En las figuras anteriores de Wireshark se puede ver como viaja en formato binario el payload de la petición y la respuesta con Content-Format distintos a text/plain. En la figura 3.45 se puede observar como se ha devuelto el entero 79 pero en el stream content del socket UDP (la ventana más pequeña a la derecha de la captura) no se puede observar ningún 79, es decir viaja en formato binario. Siguiendo con la figura 3.45, se puede observar como el protocolo Lightweight M2M está encima del protocolo CoAP. También se ha captura la petición y la respuesta con el navegador web Chrome al hacer clic en la pantalla y el resultado se ilustra en las siguientes figuras:

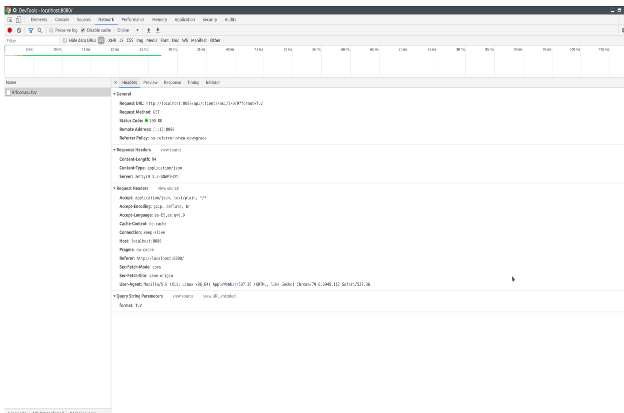


Figura 3.39: Captura del debug del navegador web Chrome (petición) al servidor Leshan

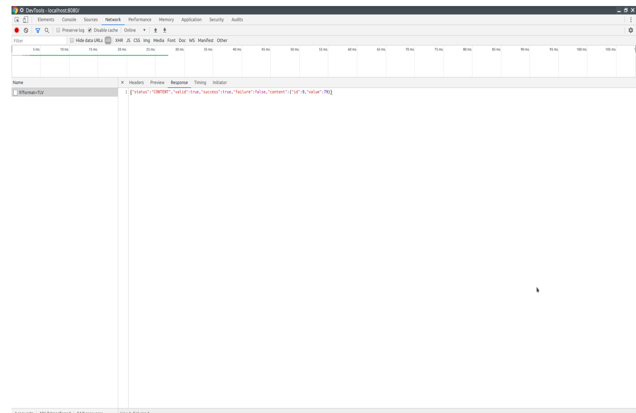


Figura 3.40: Captura del debug del navegador web Chrome (respuesta) del servidor Leshan

En la figura 3.40 se puede observar como se devuelve el 79 al navegador web para que este sea pintado en la pantalla por el propio navegador web. En la figura 3.39 se puede observar el servidor, un Jetty 9.1.z. Desafortunadamente, durante la investigación, se estuvo analizando los servlets pero se descubrió que los contenedores Java J2EE no pueden funcionar en Android porque no dan soporte a JMX, así pues, casi con total seguridad es que no podremos arrancar un servidor Leshan en un Android. Por otra parte, Leshan y por consiguiente OMA LwM2M está pensado para la gestión centralizada de dispositivos en un servidor Leshan y que los sensores y actuadores se gestionen a través del servidor. Sin entrar en más detalles y formulando una hipótesis, se puede decir que deben usar algún tipo de socket persistente, similar a los websockets que mantienen estable la conexión para conseguir una comunicación bidireccional full duplex entre el servidor y los clientes, sin renegociar de nuevo cada petición. Esto permite que el servidor pueda iniciar la comunicación con los dispositivos clientes y es mucho más rentable que estar haciendo constantes peticiones y respuestas con el protocolo http tradicional renegociando cada socket comunicación que se abre y se cierra. En general, el enfoque utilizado en OMA LwM2M es similar a las plataformas IoT, pero en Leshan lo han hecho a muy bajo nivel ofreciendo librerías tanto para clientes y servidores y optimizando la comunicación al máximo con sockets persistentes. Gracias a estas librerías podemos construir nuestra propia plataforma IoT y empotrar la librería de los clientes en los sensores y actuadores. Las plataformas IoT se tratan en el siguiente capítulo.

CAPÍTULO 4

4. LAS PLATAFORMAS IOT

El IoT (Internet of Things) o el Internet de las Cosas se basa en el concepto de que todos los dispositivos estén conectados a la red, como puede ser Internet. Un dispositivo puede ser cualquier cosa, como una nevera, un sensor de temperatura, una taza de café o un smartphone. De manera más formal, IoT es una infraestructura de red global dinámica con capacidad de autoconfiguración basada en protocolos de comunicación estándar e interoperables donde objetos físicos y virtuales poseen identidades, atributos físicos y personalidades virtuales, utilizan interfaces inteligentes y están perfectamente integrados en la red de información [37].

Desde el punto de vista funcional, el IoT busca ser útil y hacer la vida más fácil a las personas y a las empresas. Son las personas y empresas las que compran y comprarán estas cosas inteligentes y útiles con conexión a Internet.

Para llevar a cabo la tarea de hacer realidad el IoT, algunos fabricantes de software han desarrollado software conocido como plataformas IoT. Por definición una plataforma es un marco de desarrollo centralizado donde se ofrecen aplicaciones y servicios. Existen varias plataformas en el mercado con el concepto de computación en la nube, las cuales se anuncian recientemente también como plataformas IoT, como por ejemplo Microsoft Azure y Google Cloud que han ofrecido y ofrecen soluciones basadas en centralizar todas las aplicaciones y servicios de los usuarios en su plataforma, y actualmente ofrecen soluciones para gestionar dispositivos para el IoT. Este tipo de plataformas IoT en la nube tienen su nombre propio y se las llama CoT (Cloud of Things).

Aunque las plataformas y por consiguiente las plataformas IoT son arquitecturas centralizadas, dada su importancia y su interés con el IoT, la computación ubicua y las redes M2M, las plataformas IoT se van a analizar en este capítulo.

En la práctica, las plataformas IoT se basan en instalar un software en un servidor de tipo computadora, este software puede estar en la nube o bien, dependiendo de la plataforma IoT, se puede instalar en un servidor de nuestra propiedad. Algunas de estas plataformas pueden ser sistemas operativos completos como TyniOS. Este software o sistema operativo actúa de nodo central para todos los dispositivos. Por tanto, las plataformas IoT actúan de backend y tienen las siguientes funcionalidades:

- Pueden tener una base de datos para almacenar los datos, acciones y peticiones que provienen de los dispositivos.
- Procesar la información de los sensores para tomar decisiones y actuar sobre el medio físico, en base a la lectura de los sensores y los actuadores que estén registrados en la plataforma.
- Gestionar dispositivos y tienen la responsabilidad de la conectividad entre los dispositivos.

- Comunicarse con otras plataformas.
- Deberían tener una interfaz de usuario amigable que las hagan fáciles de utilizar y configurar. También deben de presentar los datos del medio físico recogido de los sensores de forma amigable, presentando los datos en forma de tablas y gráficas.

La comunicación de los dispositivos con la plataforma y a la inversa, de los dispositivos con la plataforma, se puede hacer con protocolos estándares de la pila de protocolos TCP/IP (API REST, CoAP, MQTT, entre otros), o bien con protocolos propietarios de la plataforma, cuya implementación puede que no esté disponible para todos los dispositivos. Por defecto, si la plataforma usa protocolos estándares y abiertos siempre será más fácil de utilizar.

La idea principal de las plataformas IoT es integrar los sensores y actuadores en la plataforma. Para este objetivo se pueden elegir varios enfoques que vendrán determinados por la propia plataforma y también por la capacidad de comunicación de los sensores y actuadores. De forma genérica, puede haber, al menos 5 enfoques o modelos, tal y como muestran las siguientes subsecciones de este capítulo.

4.1. MODELOS DE COMUNICACIÓN EN PLATAFORMAS IOT

En las siguientes subsecciones se ilustra el modelo de operación normal de todos los actores implicados en la comunicación, es decir, las lecturas de los sensores y las acciones de los actuadores y no se muestra la posible configuración remota que pudiera tener un sensor que puede ser configurado de forma remota desde una plataforma IoT o bien la posible comunicación que inicia un actor ante un evento importante detectado por un sensor o un actuador.

Por tanto, los modelos presentados a continuación son muy simples y muy rígidos, se ha hecho así de forma intencionada para mostrar un posible funcionamiento normal de operación y simplificado entre los dispositivos y la plataforma IoT. Es decir, estos modelos no cubren toda la casuística de operación que pueden tener todos los actores de los modelos. Por ejemplo en el modelo 1, no todos los sensores tienen que enviar sus lecturas a la plataforma IoT, si no que otras veces y dependiendo de la naturaleza del sensor y para que se utilice, es posible que sea la plataforma IoT la que solicite la lectura al sensor. En la sección 5.3 sobre los modelos descentralizados sin plataformas IoT se detalla algo más en que momentos puede ser más aconsejable usar un modelo u otro.

4.1.1. Modelo 1 de Comunicación en Plataformas IoT

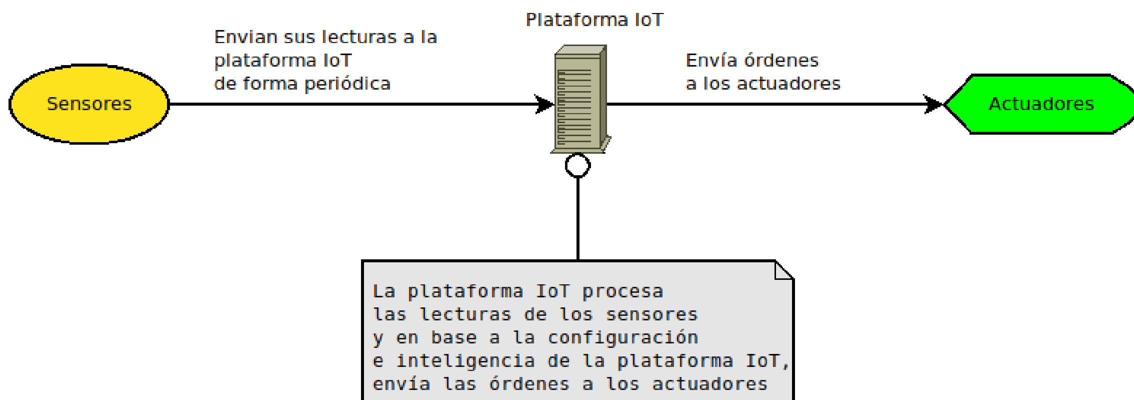


Figura 4.1: Modelo 1 de Comunicación en Plataformas IoT

Los dispositivos envían la información de sus sensores a la plataforma que registra los valores. La plataforma IoT procesa los mensajes y en base a unas reglas definidas envía las órdenes a los actuadores, es decir toda la inteligencia del modelo reside en la plataforma.

En este escenario, los actuadores estarán durmiendo la mayor parte del tiempo, mientras que los sensores tendrán que despertar de forma periódica o regular para leer su sensor y enviarlo a la plataforma IoT.

4.1.2. Modelo 2 de Comunicación en Plataformas IoT

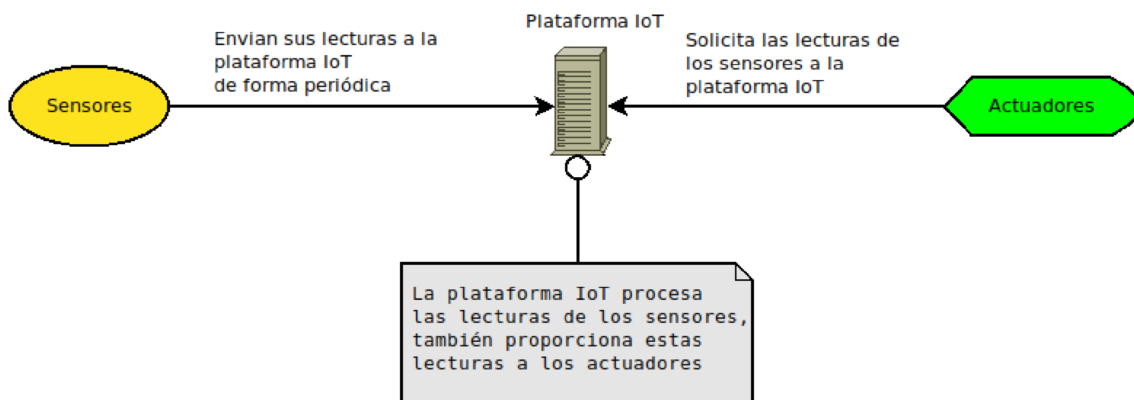


Figura 4.2: Modelo 2 de Comunicación en Plataformas IoT

Todos los dispositivos, sensores y actuadores, realizan peticiones de forma regular a la plataforma IoT. Los sensores con el fin de que queden registradas sus lecturas en la plataforma IoT. Los actuadores en base a las lecturas registradas por los sensores en la plataforma IoT, podrán ser capaces de realizar sus acciones.

En este caso, los actuadores tienen bastante inteligencia, ya que pueden ser autónomos de la inteligencia que pudiera tener la plataforma IoT, sin embargo, la lectura puede ser analizada previamente por la plataforma IoT y si

detecta algo extraño en la lectura de un sensor exponer un valor como erróneo o inválido que puede ser procesado por el actuador.

En este escenario, tanto los sensores como los actuadores tendrán que despertar de forma regular en una operación normal, siendo este modelo el que más gasto energético tiene en los dispositivos.

4.1.3. Modelo 3 de Comunicación en Plataformas IoT

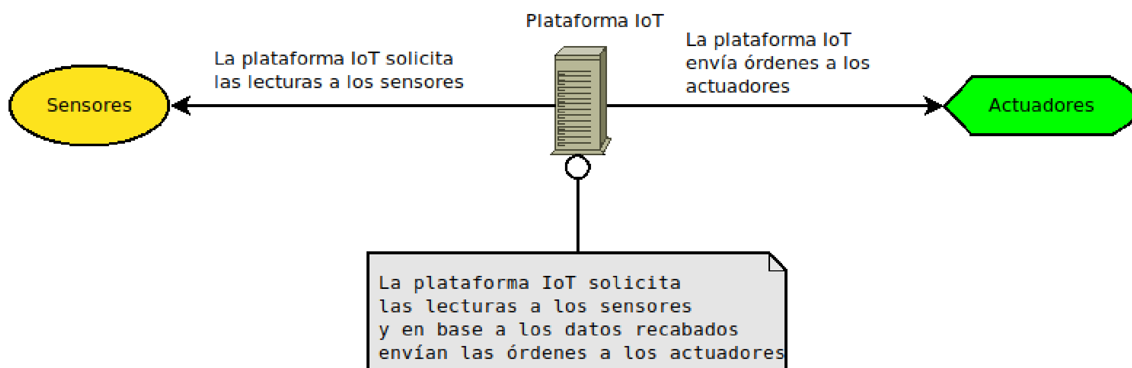


Figura 4.3: Modelo 3 de Comunicación en Plataformas IoT

En este caso la plataforma IoT es el controlador de la red y el que solicita las lecturas a los sensores y en base a las lecturas, envía las órdenes a los actuadores. En este modelo toda la inteligencia, o la mayor parte, reside en la plataforma IoT.

Este modelo proporciona el menor gasto energético en los dispositivos, debido a que tanto los sensores como los actuadores sólo tienen que despertar cuando se lo pida la plataforma IoT.

Este modelo es muy similar al protocolo SNMP, donde el administrador SNMP es el controlador y gestor de la red de los agentes. En el protocolo SNMP los dispositivos o nodos de la red, también inician la comunicación con el administrador SNMP cuando el agente detecta un evento importante en el dispositivo que monitoriza.

4.1.4. Modelo 4 de Comunicación en Plataformas IoT

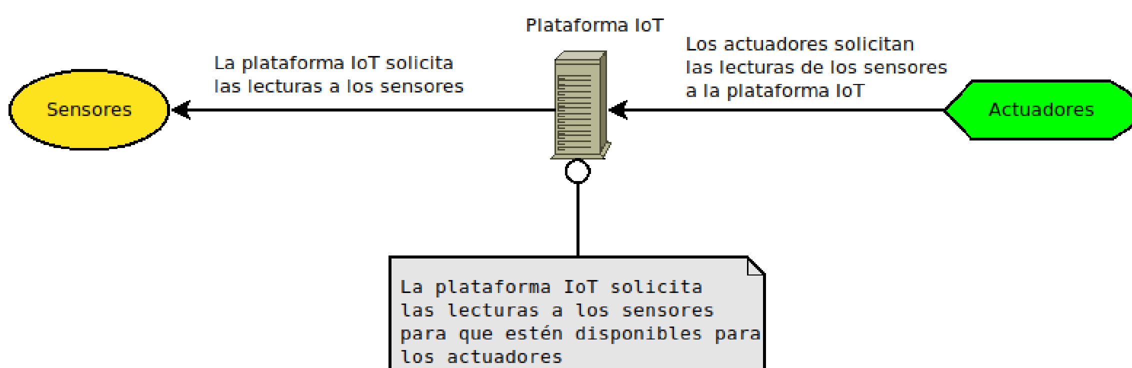


Figura 4.4: Modelo 4 de Comunicación en Plataformas IoT

Los sensores pueden estar durmiendo la mayor parte del tiempo y despertar cuando se lo pida la plataforma IoT. Por su parte los actuadores realizan peticiones regulares a la plataforma IoT para obtener la lectura de los sensores. Este modelo casa bien con un modelo de sensores con batería y pocos recursos, mientras que los actuadores son más potentes y tienen cierta inteligencia para tomar las decisiones de accionar sus interruptores para modificar el medio físico, sin contar que la plataforma IoT puede analizar previamente las lecturas de los sensores y decidir que datos exponer a los actuadores. Un ejemplo de inteligencia compartida puede ser un actuador sobre un calentador de agua eléctrico y un sensor de temperatura. El actuador realiza peticiones regulares a la plataforma IoT para obtener la lectura del sensor, pero esta lectura es filtrada previamente por la plataforma IoT, de manera que existe un horario programado en la plataforma IoT para impedir que el actuador del calentador caliente el agua por la noche. La plataforma IoT puede devolver una palabra clave, como NO, o cualquier otra y el actuador ya sabe que no tiene que calentar el agua. Sin embargo, si devuelve un valor numérico, por ejemplo 35, que sería grados centígrados, el actuador activaría el calentador eléctrico para calentar el agua.

4.1.5. Modelo 5 de Comunicación en Plataformas IoT

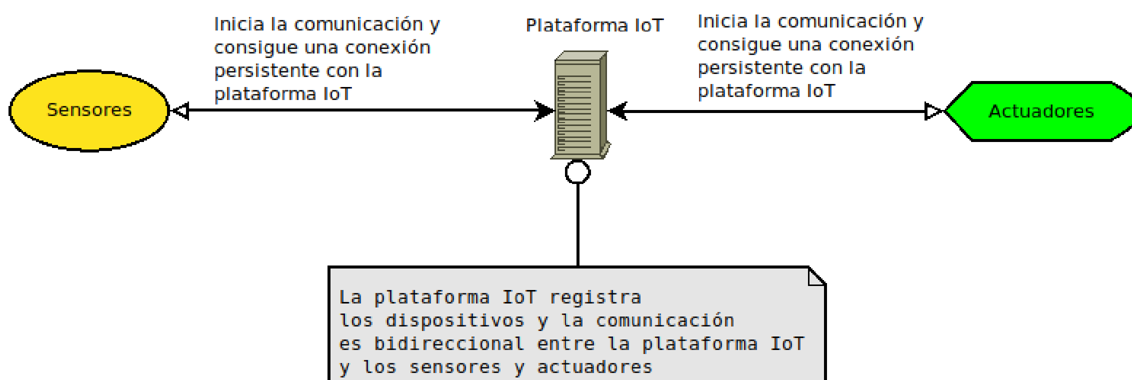


Figura 4.5: Modelo 5 de Comunicación en Plataformas IoT

Este modelo está inspirado en los websockets y el protocolo OMA Lightweight M2M. Los clientes, los sensores y actuadores inician la comunicación con la plataforma IoT (el servidor) y esta registra los dispositivos, proporcionando un socket persistente que permanece estable y que permite una comunicación bidireccional.

Este modelo cubre todas las necesidades de comunicación, desde la plataforma IoT se pueden configurar los dispositivos de forma remota y los sensores y actuadores pueden enviar peticiones a la plataforma IoT de manera regular, si es que así se ha configurado en la plataforma. Por su parte la plataforma puede pedir información a los sensores y enviar órdenes a los actuadores.

Esta forma de trabajar simplifica mucho la lógica en los dispositivos clientes, los cuales, sólo tienen que conectarse a la plataforma y no tienen

parte servidora para recibir peticiones. Por contra, los dispositivos que funcionan a batería deben estar parcialmente despiertos para mantener la persistencia del socket.

Respecto a la plataforma IoT, al estar instalada en una computadora con suficientes recursos y conectada a la red eléctrica, no hay tanto problema del consumo de memoria RAM y consumo energético para mantener las conexiones persistentes con los dispositivos.

4.2. REFLEXIONES SOBRE LAS PLATAFORMAS IOT

Las plataformas IoT son y serán fundamentales en redes grandes de sensores y actuadores, porque la arquitectura centralizada de redes de cientos o miles de dispositivos en la red, en la práctica, resulta más sencilla de administrar, tal y como ocurre con las redes de ordenadores personales tradicionales en entornos corporativos.

Hoy en día, existen bastantes plataformas IoT genéricas o más específicas, como pueden ser plataformas de domótica centradas en el hogar como OpenHAB. Analizar y experimentar con algunas de estas plataformas en profundidad daría para escribir un trabajo completo de investigación. Con las plataformas IoT podemos conseguir una red de M2M distribuida de sensores y actuadores donde:

- La mayor parte o toda la inteligencia reside en la plataforma IoT y no reside en los sensores y actuadores.
- La red M2M resultante de sensores y actuadores es centralizada en la plataforma IoT, en vez de una red M2M de sensores y actuadores descentralizada.
- Una caída de la plataforma IoT inutilizaría la red de sensores y actuadores. Se puede mitigar utilizando plataformas IoT de respaldo, si cae la plataforma IoT principal otra tomaría su lugar.

Por los motivos de los puntos anteriores se deja de lado la línea de investigación de las plataformas IoT.

CAPÍTULO 5

5. ARQUITECTURAS DE SISTEMAS SOFTWARE PARA LA INTEROPERABILIDAD DE SENSORES Y ACTUADORES

Después de la investigación de los capítulos anteriores y del estado del arte actual, en este capítulo se van a proponer algunas arquitecturas software de tipo distribuidas y descentralizadas teniendo en cuenta las tecnologías de del capítulo 3, que no dependan de nodos centrales o plataformas IoT del capítulo 4 para trabajar y donde la inteligencia reside en los sensores y actuadores.

Este capítulo trata el nivel lógico de más alto nivel de la arquitectura software entre los sensores y actuadores y todavía no se proponen protocolos ni patrones de diseño software para la interconexión de los sensores y actuadores, aunque en el modelo 3 de este capítulo si que se nombran los sockets persistentes. Estas cuestiones se tratan con más profundidad en el capítulo 6.

Una plataforma IoT, un ordenador personal, una impresora, una nevera, una televisión o un smartphone pueden formar parte de la red de máquinas, pero lo más importante es que la red M2M de sensores y actuadores continúe trabajando a pesar de que se caiga la plataforma IoT. Es decir, la plataforma IoT puede existir en la red M2M y porque, en la práctica, es mucho más cómodo hacerlo así para configurar de forma centralizada los dispositivos y para tener de forma centralizada los datos y presentarlos al usuario en una aplicación web. Pero, si apagamos la plataforma IoT, la red de máquinas de sensores y actuadores continúa trabajando y podremos configurar los dispositivos, ya no de forma centralizada, pero si podremos hacerlo, conectándonos a cada dispositivo, desde un ordenador personal, un smartphone o desde el propio dispositivo, en el caso de que tuviera una pantalla o alguna interfaz de usuario para poder hacer la configuración de forma local al dispositivo.

5.1. TOPOLOGÍAS DE RED DESCENTRALIZADAS PARA M2M DE SENSORES Y ACTUADORES

5.1.1. Topología de Red de Malla Completa

La red de malla completa es un tipo de topología de red, en la cual, cada nodo está conectado con todos los demás nodos de la red. En las redes cableadas y de forma estricta, esto significa que cada nodo debe tener un cable con cada uno de los demás nodos de la red, aplicando la siguiente fórmula:

$$N = \frac{n(n-1)}{2}$$

Figura 5.1: Formula para calcular el número de enlaces según el número de nodos

Si tenemos 6 nodos, resulta que n = 6, entonces N = 15 enlaces. Tal y como muestra la siguiente figura:

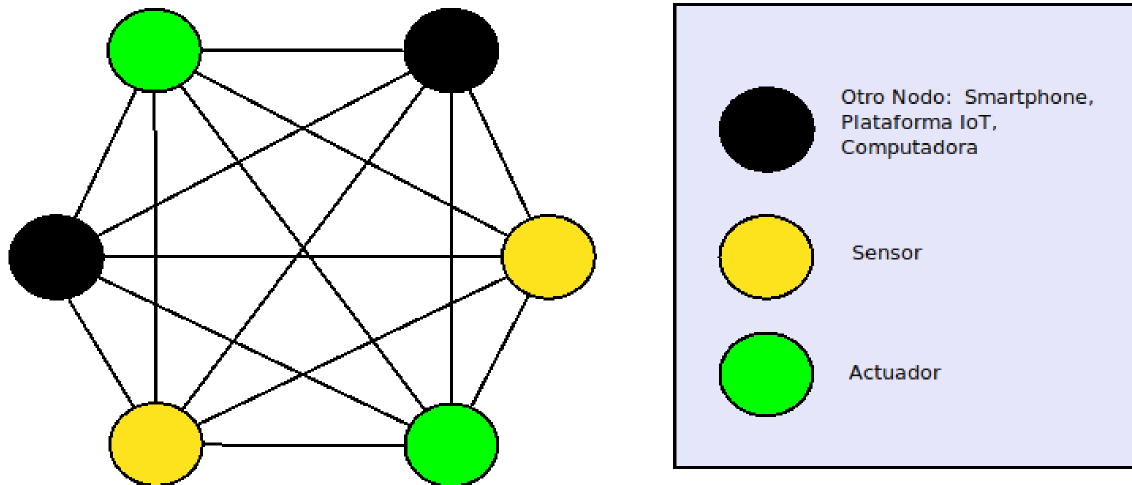


Figura 5.2: Topología de red de malla completa para 6 nodos

A medida que añadimos más nodos a la red el número de cables crece de manera inviable y en la práctica se consiguen simular topologías de malla de manera lógica con una topología de red física en estrella, sin todos los cables conectados para cada nodo mediante switches de red o enrutadores.

En sistemas inalámbricos, no tenemos el problema de los cables, pero cada nodo no está conectado directamente con todos los demás directamente, por ejemplo con tecnología Wi-Fi Direct, si no que se puede conseguir una topología de red en malla lógica con una topología de red física en estrella, al igual que en sistemas cableados, con puntos de acceso inalámbricos o enrutadores.

En la figura 5.2 se puede observar como existen sensores y actuadores, pero también pueden existir en la red otros nodos como las plataformas IoT.

Este tipo de topologías de red en malla, aunque sea a nivel lógico, resultan muy interesantes en entornos descentralizados para el IoT y las redes de máquinas M2M. Sin embargo, puede que cada nodo no necesite conectarse con todos los demás nodos. Por ejemplo, puede que un sensor no necesite conexión con otro sensor, lo que nos lleva al siguiente apartado.

5.1.2. Topología de Red de Malla Parcial

Dado que existe la posibilidad de que cada nodo no necesite tener enlace con todos los demás nodos de la red, existen también las topologías de red de malla parcial. Esta topología es similar a la completa, pero podemos tener menos enlaces entre los nodos, para una red de 6 nodos podemos tener 13 enlaces, en vez de 15 enlaces, como en la siguiente figura:

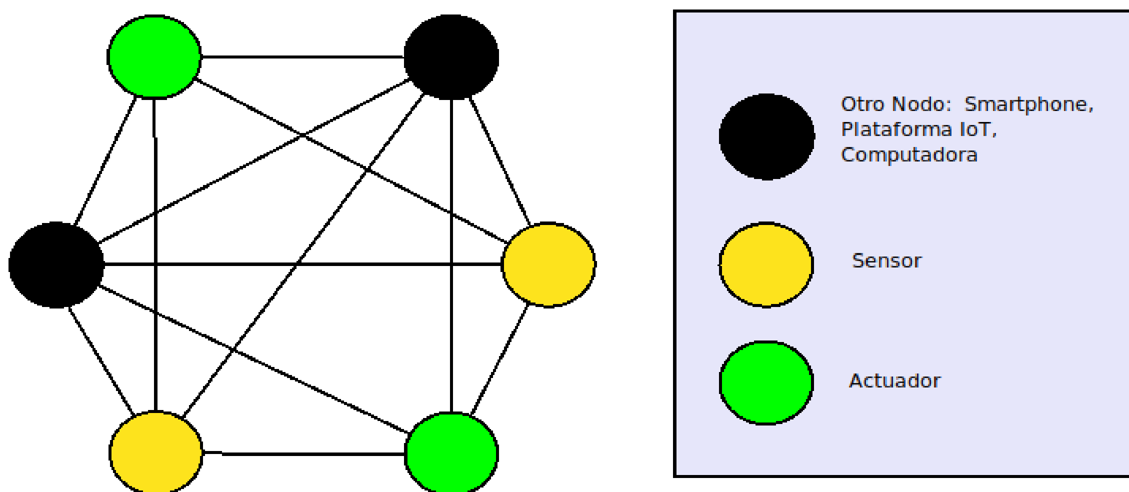


Figura 5.3: Topología de red de malla parcial para 6 nodos

En la figura anterior, se han eliminado los enlaces entre los sensores y los enlaces entre los actuadores, ha sido a modo de ejemplo, pero pueden existir otras configuraciones de enlaces. Al igual que en la topología de malla completa, los enlaces entre los nodos se harán a nivel lógico a través de un punto de acceso o enrutador, es decir, no hará falta que para cada nodo de la red tenga un enlace directo, mediante un cable o con Wi-Fi Direct con todos los demás nodos.

Este tipo de red en malla parcial es muy interesante para conseguir la red descentralizada de máquinas, donde no todas las máquinas tengan enlace con todas las demás máquinas. Por ejemplo, un actuador que controla el mecanismo de una persiana, no necesita un sensor de temperatura y este enlace no se hará entre este tipo de sensor y este tipo de actuador.

La malla parcial se construye de forma dinámica por las máquinas que se conectan a la red. Para conseguirlo, una máquina vendrá con una configuración de fábrica de las máquinas que necesita. Un ejemplo puede ser el sistema de calefacción de una vivienda que es controlado por un actuador. El actuador buscará sensores de temperatura en la vivienda pero, en base a su configuración, ignorará el resto de sensores.

5.2. ARQUITECTURA CLIENTE-SERVIDOR DE SENSORES Y ACTUADORES

Las topologías presentadas en la sección 5.1 son del tipo cliente-servidor, donde cada nodo puede ser cliente (inicia la comunicación) y al mismo tiempo servidor (recibe la comunicación). Dado que no tenemos máquinas con la caché de servidor, es decir no es necesaria una plataforma IoT, la computación en la nube o middlewares de mensajes (o al menos no tenemos una máquina de escritorio exclusiva a modo de middleware de paso de mensajes), llegados a este punto conviene aclarar que es un cliente y que es un servidor en un entorno de máquinas descentralizadas.

5.2.1. Dispositivo Servidor

En general, un servidor es un software que está corriendo en una máquina y que espera peticiones, peticiones que le llegan de los clientes. Así pues, un servidor no suele hacer nada, a no ser que tenga tareas programadas para realizar ciertas acciones o comandos. Por tanto, un servidor procesa básicamente las peticiones de los clientes y les proporciona una respuesta.

La palabra servidor siempre denota tener algo de más importancia o de caché que los clientes, sin embargo, dos servidores pueden comunicarse entre sí. Para nuestro caso, hay que entender que el que inicia la comunicación es el cliente y el que recibe la petición es el servidor, independientemente de que ambos puedan tener la categoría de servidor.

En una arquitectura descentralizada y distribuida, un servidor puede ser un sensor, por ejemplo un sensor lumínico que lee de forma periódica la luz ambiente de su sensor interno del dispositivo y que expone, con el patrón de diseño API REST, sus lecturas para que otros dispositivos clientes puedan hacer uso de esta lectura.

5.2.2. Dispositivo Cliente

Un cliente es otro software que corre, generalmente, en otra máquina y que hace peticiones a los servidores. Los clientes realizan peticiones a los servidores por muchas razones, por ejemplo, para obtener datos de una base de datos remota o para que realicen cálculos.

En nuestro escenario, un cliente puede ser un dispositivo que dispone de un actuador para encender una bombilla y que hace peticiones a un servidor/sensor lumínico para decidir si enciende la bombilla en función de las lecturas que recibe del servidor/sensor.

5.2.3. Dispositivo Cliente y Servidor

Es habitual que un servidor sea al mismo tiempo cliente, debido a que procesa las peticiones de sus clientes y al mismo tiempo, este servidor (que es también cliente) hace peticiones a otro servidor.

En un entorno corporativo de computadoras de tipo servidores y clientes la distinción entre ambos está clara. Los clientes son las estaciones de trabajo donde trabajan las personas y los servidores son las máquinas remotas que tienen más recursos y que son utilizadas por todos los clientes. Un servidor puede ser un servidor web, un servidor de usuarios, un servidor de datos o un servidor de impresión.

En una red de máquinas de sensores y actuadores, un sensor puede ser servidor y al mismo tiempo cliente. De tipo servidor porque expone o sirve sus lecturas para ser leídas por otros dispositivos y al mismo tiempo puede ser cliente porque envía las lecturas del día anterior a una plataforma IoT o el sensor puede ser cliente también porque detecta algo muy importante que requiere ser notificado.

5.3. MODELOS DE COMUNICACIÓN DE SENSORES Y ACTUADORES

Estos modelos son similares a los modelos presentados en las plataformas IoT pero en este caso, la plataforma IoT no existe en el modelo. Estos modelos son muy simples para ver como podrían trabajar juntos los sensores y actuadores en una arquitectura descentralizada.

Como ya se dijo en las plataformas IoT, un modelo cubre la casuística de un caso concreto y particular y no hay que elegir, para toda la red de máquinas, un modelo concreto y aplicarlo de forma sistemática en todos los sensores y actuadores.

En cada modelo se explica un ejemplo para ver un caso de uso que puede ser más adecuado que otro a un caso particular.

Por simplicidad, en los modelos se han omitido otros actores distintos a los sensores y actuadores, como pueden ser ordenadores personales, plataformas IoT o smartphones, que también pueden formar parte de la red de máquinas, aunque pueden nombrarse en el ejemplo del modelo.

Las líneas discontinuas de las respuestas del modelo 1 y del modelo 2 están por el hecho de que la respuesta puede ser síncrona, pero también ser asíncrona o bien puede no existir la respuesta, como ocurre con los tópicos o las colas. El protocolo ZeroMQ, a falta de investigar este protocolo en profundidad, también permite hacer peticiones multipunto de un emisor a varios receptores.

5.3.1. Modelo 1 de Comunicación de Sensores y Actuadores

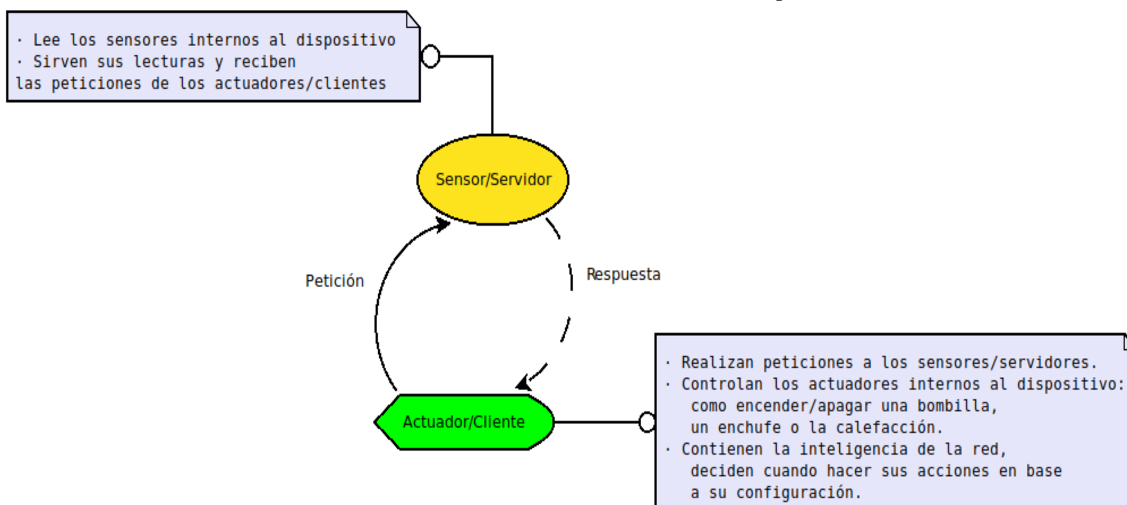


Figura 5.4: Modelo 1 de Comunicación de Sensores y Actuadores

Los actuadores son los controladores de la red y realizan de forma periódica y regular peticiones a los sensores para obtener la lectura de los sensores.

Un ejemplo de este tipo puede ser un dispositivo empotrado en un sistema de aire acondicionado de un museo. Este dispositivo tiene un actuador que regula la potencia del sistema de aire acondicionado. Para realizar su labor, necesita de las lecturas de los sensores que estarán diseminados por todo el museo. Estos sensores pueden ir conectados a la red eléctrica en dispositivos realmente muy restringidos y más pequeños que el actuador y no tenemos el problema de que se les pueda terminar la batería.

El actuador realiza cada 5 minutos, peticiones a todos los sensores del museo para obtener sus lecturas y el actuador regula la potencia del aire acondicionado, en base a su configuración y a la inteligencia que pueda tener y el registro del pasado de las personas que suelen venir ese día de la semana al museo.

A su vez y de forma opcional, el actuador puede enviar la información que ha conseguido a un servidor de backend o una plataforma IoT para registrar los valores que ha obtenido de los sensores cada cierto tiempo, por ejemplo cada hora. Lo más importante de entender del ejemplo es que si la plataforma IoT o servidor de backend se apaga, el sistema de aire acondicionado continúa funcionado con normalidad.

5.3.2. Modelo 2 de Comunicación de Sensores y Actuadores

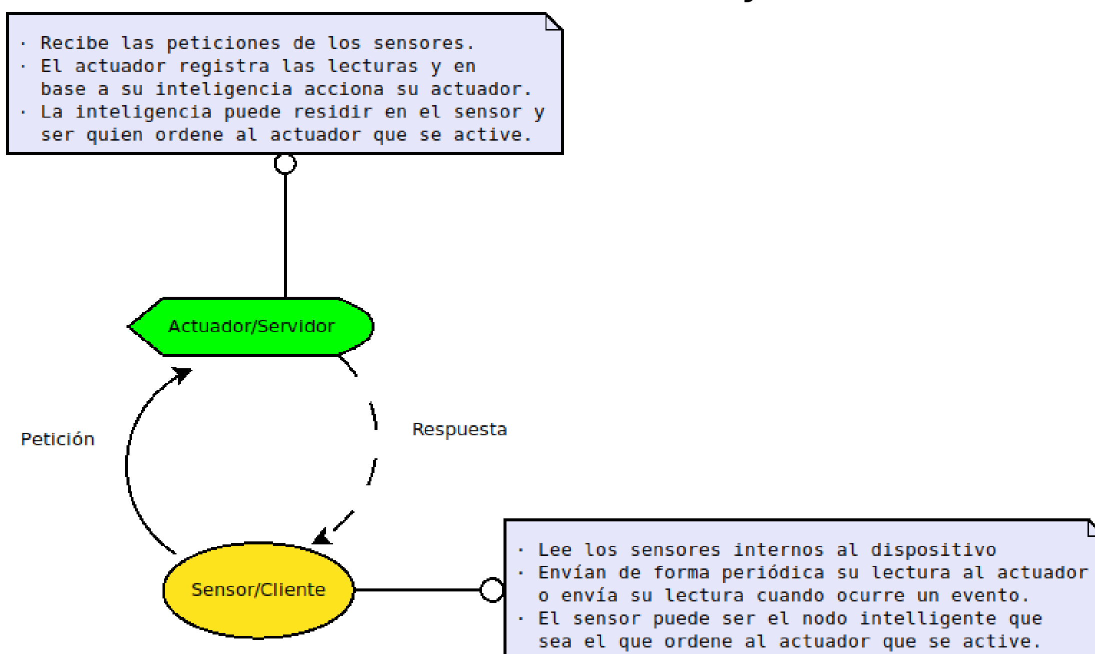


Figura 5.5: Modelo 2 de Comunicación de Sensores y Actuadores

En este modelo, los sensores son los que controlan la red. Los sensores pueden enviar de forma periódica o regular peticiones a los actuadores, o bien los sensores pueden enviar las peticiones sólo cuando ocurra un evento importante. Los sensores pueden ser inteligentes y directamente ordenar a un actuador que se active o que se desactive.

Siguiendo con el ejemplo del museo del apartado anterior, por la noche el museo cierra y se conecta un sistema de alarma. Este sistema de alarma cuenta con sensores de movimiento, de presión y de presencia de calor e infrarrojos para detectar posibles intrusos. Los sensores están conectados a la red eléctrica y están leyendo su sensor de forma regular, de forma que cuando detectan un intruso envían una petición al actuador.

Para filtrar posibles falsos positivos de un sensor, el actuador necesita que varios sensores, al menos que dos sensores, hayan detectado al intruso para hacer sonar la alarma. En cualquier caso, sea un sensor o varios sensores, el actuador tiene una tarjeta SIM M2M y envía un SMS a la empresa de seguridad para que se registre el evento. El operador de seguridad humano examina y evalúa de forma remota al actuador y comprueba si se trata de un falso positivo o no.

En el caso de que sólo un sensor detecte al intruso y el SMS falle por cualquier causa, el actuador intenta enviar la alarma por Internet con una petición http al sistema de seguridad. Si todo falla, el actuador intenta filtrar la lectura del sensor y si detecta valores muy extraños, por ejemplo el sensor de presión detecta 10 toneladas de presión, no hace sonar la alarma y ordena al sensor que se apague en la respuesta que le proporciona. Pero si ha detectado valores dentro de un umbral posible, el actuador por precaución hace sonar la alarma y continuará intentando avisar por cualquier medio a la empresa de seguridad.

5.3.3. Modelo 3 de Comunicación de Sensores y Actuadores

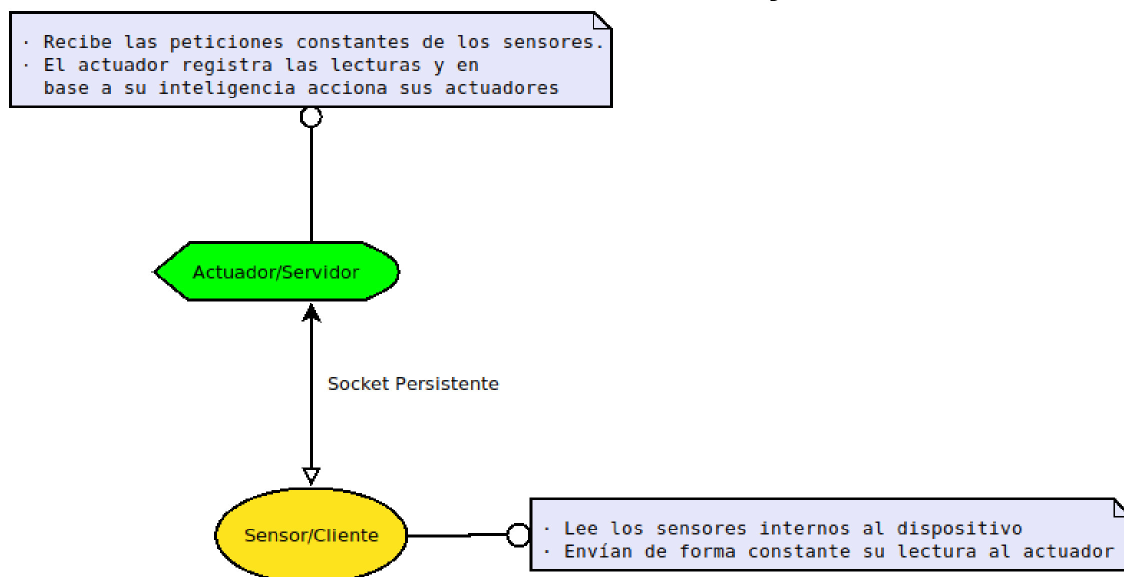


Figura 5.6: Modelo 3 de Comunicación de Sensores y Actuadores

Los sensores mantienen un socket persistente con el actuador para enviar de la manera más rápida posible sus lecturas al actuador. Este escenario se puede dar en sistemas críticos, donde es necesario obtener el

valor de los sensores en tiempo real. Con los sockets persistentes se mantiene una comunicación más rápida, debido a que no hay que renegociar toda la conexión por cada envío de datos.

Un ejemplo de este tipo puede darse en un sistema de estabilidad de un avión de pasajeros. Los sensores del avión son de tipo acelerómetro y están midiendo y enviando de forma constante las lecturas al actuador por su socket persistente. Si varios de estos sensores detectan una inclinación peligrosa del avión, el actuador marcará una alarma visual y sonora en el panel de control para que el piloto corrija la inclinación. En caso ya de alcanzar una inclinación fatal y sin que el piloto haya corregido el rumbo, el actuador corregirá automáticamente la inclinación.

Estos aviones deben tener instalado un servidor central a modo de plataforma IoT. Este servidor de backend está monitorizando de forma constante que todo funciona correctamente. Si la plataforma IoT observa que el actuador está funcionando incorrectamente podría invalidar la orden de corregir la inclinación. También podría ocurrir que el servidor central tenga una avería y no funcione correctamente, en este caso los sensores de acelerómetro y el actuador del sistema de estabilidad podrían seguir funcionando con normalidad, porque la comunicación no pasa por el servidor de backend, si no que son los propios dispositivos los que se comunican directamente.

CAPÍTULO 6

6. PROTOCOLOS Y PATRONES DE DISEÑO SOFTWARE PARA LA INTEROPERABILIDAD ENTRE SENSORES Y ACTUADORES

En esta sección se va a hablar de como se puede conseguir conectar, lo más directamente posible, los sensores y actuadores para obtener una red distribuida y descentralizada de máquinas. Para obtener los resultados deseados, necesitamos que los dispositivos, sean sensores o actuadores, puedan levantar procesos o servicios para escuchar las peticiones que les puedan llegar de otros dispositivos clientes, sean estos sensores, actuadores o cualquier tipo de nodo de la red, como un ordenador o un smartphone. Recordemos que para conseguir el objetivo de red descentralizada, tenemos que implementar una topología de red en malla parcial, como la red de la figura 5.3. Mezclar los protocolos de comunicaciones con los patrones de diseño software es necesario debido a que, según el protocolo que estemos usando podremos aplicar un patrón de diseño software u otro, o simplemente, el protocolo de comunicaciones ya ofrece un patrón de diseño software y no sea necesario aplicar ninguno. Un patrón de diseño software son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto [38].

En las siguientes subsecciones todo el código y las estructuras de datos presentadas pertenecen a los prototipos implementados en el trabajo y se presentan las partes más importantes de los patrones software utilizados en los prototipos. Todo el código fuente se encuentra en:

- Proyecto TFM Server: paquete es.uned.tfm.server.servers.controller
- Proyecto TFM Client: paquete es.uned.tfm.client.clients.controller

6.1. PATRÓN DE DISEÑO SOFTWARE REST

REST ya se ha comentado en la sección 3.2.2 como un API de servicios web para la interoperabilidad de procesos, donde se realizan llamadas http por los métodos POST, GET, PUT y DELETE. El protocolo http es un protocolo de tipo solicitud-respuesta, los clientes (normalmente un navegador) envían una petición al servidor Web, que es un software que contiene información basada en Web y que la proporciona como consecuencia de las peticiones de sus clientes [39].

Por otra parte, en la sección 3.2.2 no se ha comentado nada sobre el patrón de diseño que utiliza para la definición de recursos. En el patrón de diseño software REST se definen los recursos mediante URLs, como el siguiente ejemplo:

```
GET http://host:10101/rest/device
```

Figura 6.1: Petición GET http que devuelve información de un dispositivo

La ejecución de la llamada de la figura anterior por GET al dispositivo (host: es la dirección IP o el hostname de la máquina), devuelve un documento json como el siguiente:


```
[
  {
    "hostname": "host",
    "id": 1,
    "identifier": "00:08:22:28:09:5F",
    "label": "*",
    "netAdapter": "wlan0",
    "sensors": [
      {
        "id": 1,
        "sensorActivated": true,
        "sensors": "Battery"
      },
      {
        "id": 2,
        "sensorActivated": true,
        "sensors": "Proximity"
      }
    ]
  }
]
```


Figura 6.2: Información de un dispositivo.

De forma similar se podría utilizar otra invocación como la siguiente para obtener la última lectura del sensor de batería:

```
GET http://host:10101/rest/device/sensors/battery/last
```

Figura 6.3: Petición GET http que devuelve la última lectura de la batería

Y nos devuelve un documento json como el siguiente:


```
{
  "correct": 1,
  "id": 14818,
  "isCharging": false,
  "prLevelRemaining": 52,
  "sensorReadDate": "2020-05-11 08:13:10,162",
  "sensorsEntity": {
    "devicesEntity": {
      "id": 1,
      "identifier": "00:08:22:28:09:5F",
      "label": "*",
      "netAdapter": "wlan0"
    },
    "id": 1,
    "sensorActivated": true,
    "sensors": "Battery"
  }
}
```


Figura 6.4: Lectura del sensor de batería de un dispositivo

Los identificadores se pueden resolver de la misma forma que se hace en la tecnología RFID, cada dispositivo tendría un identificador único. De

hecho, cada dispositivo podría tener un chip RFID que identificaría de forma única un dispositivo y un lector para leer el chip RFID y exponerlo en el servicio REST. Otra forma, sería guardar en un circuito impreso el identificador del dispositivo y leerlo desde ahí. Pero la forma más sencilla es almacenar la dirección física del adaptador de red Wi-Fi o dirección MAC, que se supone que es única en el mundo. Los fabricantes de adaptadores de red graban en la tarjeta de red de forma permanente este tipo de identificadores, de tal forma que cada adaptador tiene un identificador único, aunque sean de distinto fabricante. En los prototipos, este identificador es el atributo identifier que es la dirección física del adaptador de red wlan0 que corresponde con el adaptador de red Wi-Fi utilizado en un smartphone de Android.

En el prototipo de TFM Server, que lee sus sensores internos y los sirve por red para otros dispositivos, tiene levantado un servidor web con la librería AsyncHttpServer y el código fuente tiene el siguiente aspecto:

```
1 this.server.get("/rest/device", (request, response) -> {
2    try {
3        String json = DevicesEntity.toJson(this.devicesDao.getAllEntitiesWithSensors());
4        response.setContentType("application/json");
5        response.code(ConstantsUtil.RESPONSE_OK);
6        response.send(json);
7    } catch (Exception e) {
8        response.code(ConstantsUtil.RESPONSE_ERROR);
9        response.send(e.toString());
10       LogsUtil.log("AsyncHttpServerConfig.init() --> /rest/device ERROR: ",
11               LogsUtil.Levels.ERROR, e);
12    } finally {
13        response.end();
14    }
15 });
```

Figura 6.5: Código Android/Java que devuelve el documento json con información del dispositivo con AsynHttpServer en el prototipo TFM Server

La figura anterior muestra el código para conseguir el documento json de la figura 6.2 que contiene la información de un dispositivo y devolverlo al cliente. El objeto server es un objeto del tipo AsyncHttpServer de Android que nos permite escuchar las peticiones http. En este objeto se define uno de los recursos de los que dispone el servidor que es /rest/device y que será invocado por el método GET. Para cada recurso del servidor, se pueden definir recursos con bloques de código fuente de este tipo y de esta forma aplicar de forma sistemática el patrón de diseño API REST. Con REST no es necesario aplicar ningún patrón de diseño porque implícitamente estamos usando uno, la definición de recursos en forma de URL. En los ejemplos anteriores se puede observar como el protocolo utilizado en las invocaciones son llamadas del protocolo http, sin embargo, en la sección 3.2.8, el protocolo CoAP usa el mismo patrón de diseño software REST para definir los recursos. Sin embargo, en las pruebas con el prototipo y el protocolo CoAP, CoAP no admite definir los recursos utilizando el símbolo barra inclinada “/” para separar los recursos. Así pues, utilizando un workaround, se ha utilizado el símbolo de guión bajo “_”

para separar los recursos y una petición para solicitar información del dispositivo tiene el siguiente aspecto:

```
GET coap://host:10201/device_sensors_battery_last
```

Figura 6.6: Petición GET CoAP que devuelve la última lectura de la batería

El documento json devuelto por CoAP es el mismo que el mostrado en la figura 6.4. A nivel de código fuente, CoAP nos obliga a declarar los recursos de otra forma que en AsynHttpServer, extendiendo de la clase CoapResource, como muestra la siguiente figura:

```
1 public static class CoapDevicesResource extends CoapResource {
2
3    private static final String resource = "devices";
4
5    public CoapDevicesResource() {
6        super(resource);
7    }
8
9
10  public CoapDevicesResource(String name) {
11        super(name);
12    }
13
14    @Override
15    public void handleGET(CoapExchange exchange) {
16        try {
17           exchange.respond(CoAP.ResponseCode.VALID,
             DevicesEntity.toJson(devicesDao.getAllEntitiesWithSensors()),
             MediaTypeRegistry.APPLICATION_JSON);
18        } catch (Exception e) {
19           LogsUtil.log("CoapDevicesResource.handleGET() --> ERROR: ", LogsUtil.Levels.ERROR, e);
20           exchange.respond(CoAP.ResponseCode.INTERNAL_SERVER_ERROR, e.toString());
21        }
22
23    }
24 }
```

Figura 6.7: Código Android/Java que devuelve el documento json con información del dispositivo con CoAP en el prototipo TFM Server

La figura anterior define el recurso device. Se trata de una clase estática que es interna a la clase que añade los recursos CoAP. El bloque de código de esta clase externa que añade los recursos se presenta en la siguiente figura:

```
1 try {
2    Class<CoapResource>[] resourcesClasses = (Class<CoapResource>[]) this.getClass().getClasses();
3    for (Class<CoapResource> resourceClass : resourcesClasses) {
4        CoapResource coapResource = resourceClass.getDeclaredConstructor().newInstance();
5        this.server.add(coapResource);
6    }
7 } catch (Exception e) {
8    LogsUtil.log("CoapServerController.init() --> ERROR: ", LogsUtil.Levels.ERROR, e);
9 }
```

Figura 6.8: Bloque de Código Android/Java para añadir recursos en CoAP usando la reflexión en Java en el prototipo TFM Server

El bloque de código anterior utiliza la reflexión en Java para conseguir todas las clases internas y mediante un bucle, en this.server.add se van añadiendo todos los recursos. Esta forma de trabajar evita que tengamos que repetir un this.server.add por cada recurso que añadamos, si tuviéramos 100 recursos, tendríamos 100 instrucciones de tipo this.server.add o bien añadir los 100 recursos dentro de una instrucción this.server.add, lo que, en cualquier caso, empieza a ser inmantenible y propenso a errores, por el olvido de añadir algún recurso. Es decir, evita tener que escribir un bloque de código como el siguiente con 3 recursos:

```
1 CoapCommand command1 = new CoapDevicesCommand("devices", this.service.getApplicationContext());
2 CoapCommand command2 = new CoapSensorsBatteryALLCommand("devices_sensors_battery_all",
                          this.service.getApplicationContext());
3 CoapCommand command3 = new CoapSensorsBatteryLASTCommand("devices_sensors_battery_last",
                          this.service.getApplicationContext());
4 this.server.add(
        command1,
        command2,
        command3);
```

Figura 6.9: Bloque de Código Android/Java para añadir recursos en CoAP sin la reflexión en Java en el prototipo TFM Server

Con la reflexión, la mejora es evidente, nunca tendremos que modificar el bloque de código de añadir recursos CoAP y lo más importante, es que nos aseguramos que se añadirán todos los recursos definidos al servidor CoAP.

6.2. PATRÓN DE DISEÑO SOFTWARE FRONT CONTROLLER

Front Controller es un patrón de diseño muy versátil que puede ser aplicado a varios protocolos de comunicaciones y diferentes situaciones. En el patrón de diseño Front Controller todas las peticiones pasan por el mismo proceso de comunicaciones que está esperando las peticiones de los clientes en un puerto específico. Entendiendo un proceso de comunicaciones como una clase, un bloque de código fuente, que recibe la petición y toda la carga útil que lleva la petición, la procesa y envía una respuesta al cliente. La carga útil del mensaje puede ser, básicamente, de dos formas:

- Una estructura fija, como un documento json. Este documento json contendrá metadatos para que el Front Controller sepa como tratar la petición. La parte que no son de metadatos puede ser variable y obviarse en la petición al serializar los objetos.
- Una estructura variable, en forma de clave valor, similar a los parámetros de las invocaciones http. Algunos de estos parámetros ayudarán al Front Controller a procesar la petición.

El patrón de diseño Front Controller se apoya en la carga dinámica de clases y otro patrón de diseño software llamado Command.

Para los ejemplos de las siguientes subsecciones se va a utilizar los sockets, debido a que, como no hay protocolo de aplicaciones, podemos implementar ambos tipos de estructura para la carga útil de los mensajes.

6.2.1. Estructura Fija de Carga Útil

Cuando la estructura es fija conocemos de antemano que datos como máximo podemos enviar y recibir. Así pues, tanto cliente como servidor conocen de antemano que clases de modelo de datos y por tanto que estructuras de datos (los documentos json) van a viajar por la red de comunicaciones. Se dice que datos como máximo vamos a enviar porque si uno de los parámetros es nulo puede omitirse en la serialización de los datos y que no viaje a nulo por la red de comunicaciones. En los prototipos, ningún dato es nulo y no se da este último caso. El prototipo TFM Client envía un documento json con sockets TCP como el siguiente:

```
{
  "command": "SocketsTCPSensorsBatteryCommand",
  "tcpSocketsRequestParamsBean": {
    "operation": "LAST"
  }
}
```

Figura 6.10: Estructura fija de carga útil que se envía desde el prototipo TFM Client al prototipo TFM Server mediante sockets TCP

En el prototipo TFM Server se procesa la petición como la siguiente figura:

```
1 tcpSocketsRequestBean = TCPSocketsRequestBean.fromJson(jsonRequest);
2 String command = tcpSocketsRequestBean.getCommand();
3 SocketsTCPCommand socketTCPCommand = (SocketsTCPCommand)
            Class.forName("es.uned.tfm.server.servers.controller.commands.socketstcp." +
            command).newInstance();

4 String jsonResponse = socketTCPCommand.execute(this.service.getApplicationContext(),
          tcpSocketsRequestBean.getTcpSocketsRequestParamsBean(),
          ServerLogInvocationsEntity);
```

Figura 6.11: Front Controller de los sockets TCP en el prototipo TFM Server

La figura anterior es la clase SocketsTCPServerController, como hay mucho código relacionado con el manejo de los sockets TCP en Android/Java, sólo se muestra la parte del código fuente que es relevante para el patrón de diseño Front Controller. Este controlador hace:

- En la línea 1 de la figura anterior deserializamos el documento json al objeto Java TcpSocketsRequestBean, este objeto se instancia en líneas anteriores al código.
- En la línea 2, obtenemos una cadena de texto que representa una clase Java, es decir la cadena SocketsTCPSensorsBatteryCommand.
- En la línea 3 utilizamos carga dinámica de clases en Java para instanciar la clase cuyo nombre es SocketsTCPSensorsBatteryCommand.

- Por último, en la línea 4, ejecutamos el método execute de la clase SocketsTCPSensorsBatteryCommand. El parámetro interesante para esta sección es el segundo que contiene el tipo de operación, en nuestro caso es LAST, como en la figura 6.10. Los commands se pueden encontrar en el prototipo TFM Server en el paquete que indica la línea 3 (que está verde dentro del Class.forName). Un command realiza una acción en concreto, en este caso devolver la última lectura del sensor de batería registrada tal y como muestra el siguiente código:

```
1 public class SocketsTCPSensorsBatteryCommand implements SocketsTCPCommand {
2    @Override
3    public String execute(Context applicationContext,
                          TCPSocketsRequestParamsBean request,
                          ServerLogInvocationsEntity serverLogInvocationsEntity) {
4        String jsonResponse;
5        String operation = request.getOperation();
6        switch (operation) {
7            case "ALL":
8                jsonResponse =
                    BatterySensorEntity.toJson(batterySensorDao.getAllEntitiesOrderByIdDesc());
9                break;
10            case "LAST":
11                jsonResponse = BatterySensorEntity.toJson(batterySensorDao.getLast());
12                break;
13            default:
14                jsonResponse = SensorsEntity.toJson(sensorsDao.getEntityById(1L));
15                break;
16        }
17        return jsonResponse;
18    }
19 }
```

Figura 6.12: Command de los sockets TCP para obtener las lecturas del sensor de batería en el prototipo TFM Server

En la figura anterior se ha omitido algunas partes del código no relevantes para que el ejemplo sea más fácil de entender. Básicamente este command:

- En la línea 5 obtenemos el tipo de operación que viene del documento json de la request, en nuestro caso es LAST.
- Este operation es tratado en el switch de la línea 6 y en función de esto se ejecuta el bloque del case LAST que obtiene un documento json con última lectura del sensor de batería que ha registrado el dispositivo.
- Este documento json se devuelve al controlador de la figura 6.11 y este se encarga de devolver el documento json al cliente remoto. Como no es relevante para el ejemplo, en la figura 6.11 no se muestra como un socket TCP devuelve el documento json.

6.2.2. Estructura Variable de Carga Útil

Con una estructura variable de carga útil los parámetros no están fijados en las clases del modelo de datos y se utiliza estructuras dinámicas de datos. Podemos usar una cadena de texto utilizando algún separador como el

símbolo &, de manera similar al protocolo http como separador de parámetros en la URL. Pero es mejor usar otra técnica, como en los prototipos, que se utiliza un HashMap. Un HashMap es una estructura variable de tipo clave-valor como muestra la siguiente figura:

```
{
  "command": "SocketsUDPSensorsBatteryCommand",
  "params": {
    "operation": "LAST"
  }
}
```

Figura 6.13: Estructura fija de carga útil que se envía desde el prototipo TFM Client al prototipo TFM Server mediante sockets UDP

En la figura anterior tenemos una parte fija que es el atributo command, que indica el nombre de la clase que se instanciará en el servidor. Por otra, tenemos el atributo params que es el HashMap que contiene los parámetros. En la figura anterior, sólo tiene un parámetro, que es el atributo operation con el valor LAST para conseguir la última lectura del sensor, pero podría tener más parámetros.

Respecto al código fuente que trata la petición en el servidor con el patrón Front Controller es muy similar, sólo cambia la forma de obtener el atributo operation.

Sobre cuando usar una estructura fija o variable, en los prototipos no es muy relevante porque las peticiones enviadas son muy sencillas, apenas tienen carga útil. Sin embargo se puede complicar bastante si enviamos parámetros de consulta para consultar por fecha, por identificadores de lectura o por cualquier otro parámetro o si enviamos órdenes al servidor para realizar acciones como reconfigurar o apagar el sensor. De forma general:

- Las estructuras fijas son más fáciles de utilizar y se cometen menos errores al utilizarlas. Las estructuras fijas son mejores si las funcionalidades cambian poco o nada con el tiempo.
- Las estructuras variables son más versátiles y más fáciles de mantener, pero son más difíciles de utilizar. Las estructuras variables son la mejor opción si las funcionalidades cambian de forma frecuente y queremos tener retrocompatibilidad entre clientes antiguos y servidores más actualizados.

Para nuestro caso, una estructura variable es la mejor opción, si un servidor se actualiza con nuevos parámetros y nuevas funcionalidades y un cliente aún no ha podido actualizarse, en teoría, el cliente podrá seguir utilizando el servidor como hasta ahora. Sin embargo, con una estructura fija podemos tener más problemas, debido a que la estructura que envía el cliente puede que no concuerde con la estructura que espera el servidor y se pueden producir excepciones al deserializar los documentos json en el servidor.

6.3. PATRÓN DE DISEÑO SOFTWARE PROXY

El patrón de diseño software Proxy es similar al patrón Front Controller, en cuanto a que tenemos un proceso escuchando en un puerto, sin embargo, con el patrón Proxy, toda la complejidad de las comunicaciones y del formato de intercambio de datos se abstrae en forma de funciones o métodos y objetos a nivel de código fuente. De esta forma, cuando queremos obtener la lectura de un sensor, invocamos un método local como clientes y obtenemos la lectura de un sensor remoto, sin preocuparnos de las comunicaciones, es decir, de los sockets ni de los puertos ni de si se envía un documento json o un documento xml.

En el patrón Proxy y en general en cualquier diseño software con protocolos de comunicaciones, podemos tener peticiones con carga fija o carga variable, tal y como se ha explicado en el apartado 6.2.1 y 6.2.2 respectivamente.

Este tipo de patrones se utilizan en los servicios web como SOAP, los cuales, se obtiene un objeto Proxy y luego se realiza una invocación local, a nivel de código fuente, cuando en realidad se está haciendo un invocación remota que se transmite por una red de comunicaciones.

En la parte del servidor, tenemos un dispatcher transparente, que intercepta las peticiones y enruta las peticiones y la carga útil a un método concreto del servidor de forma que el desarrollador de la app tiene unos métodos de entrada para tratar las peticiones. La implementación de un dispatcher transparente se puede conseguir usando la reflexión en Java, el nombre del método puede venir como un parámetro de la petición y el dispatcher, usando la reflexión, invocar a un método concreto de forma local en el servidor.

Para esta sección se va a comentar el prototipo TFM Client donde se hace un uso más intensivo del patrón Proxy. En el prototipo TFM Server, sólo se ha utilizado en el protocolo ZeroMQ, debido a que ya se ha usado el patrón Front Controller y el patrón REST en otros protocolos.

En el prototipo TFM Client tenemos 5 clientes de protocolos y la app sólo utiliza uno al mismo tiempo para realizar peticiones regulares a los servidores. Por ejemplo, si usamos el cliente CoAP, la app utiliza el protocolo CoAP para realizar las invocaciones remotas. Además, el cambio entre un cliente y otro es en caliente, así pues, si estamos usando CoAP y habilitamos ZeroMQ, el cliente realiza, una vez terminada la última operación de CoAP, las siguientes invocaciones remotas utilizando el protocolo ZeroMQ.

En el patrón Proxy hay al menos 3 clases implicadas y una clase cliente que utiliza el patrón Proxy. La clase cliente es un controlador que realiza de forma recurrente las invocaciones, pero no sabe que protocolo de comunicaciones va a utilizar. Todo el código se encuentra en el prototipo TFM Client en el paquete es.uned.tfm.client.clients.controllers. Esta clase cliente/controladora tiene el siguiente aspecto:

```
1  public class RemoteBatterySensorController extends HandlerThread {
2    private boolean isControllerRun = true;
3    private Actions actions;
4    // Constructor
5    public RemoteBatterySensorController(
                   RemoteBatterySensorService service, Map<String, Object> params) {
6        super("RemoteBatterySensorController", HandlerThread.NORM_PRIORITY);
7        this.service = service;
8        this.params = params;
9        this.actions = new Proxy();
10   }

11   @Override
12   public void run() {
14     while(this.isControllerRun){
15       int refresh = //[..] Parametro de la app, tiempo que duerme el controlador, ej: 1000 ms
16       try{
17           String bssid = //[..] String con la dirección física del punto de acceso
18           String label = //[..] Etiqueta de localización
19           List<RemoteSensorsEntity> listSensorsEntity = //[..] Sensores Remotos de la BD local

20          for(RemoteSensorsEntity remoteSensor : listSensorsEntity){
21             String jsonResponse =
                              this.actions.doAction(this.service, null, remoteSensor, this.params);
22          }
23       } catch (Exception e) {
24          //[..] Tratar la excepcion
25       }
26       try {
27         //[..] Dormir el controlador
28         Thread.sleep(refresh);
29       } catch (InterruptedException e) {
30          //[..] Tratar la excepcion
31     }
32   }
33 }
```

Figura 6.14: Clase cliente que usa el patrón Proxy en el prototipo TFM Client

Como anteriores veces, se han omitido partes del código para que el ejemplo sea más fácil de entender y se han comentado algunas partes. Es una clase que extiende de HandlerThread. Se trata de un manejador de hilos de Android, similar a los Thread clásicos de Java pero con algunos extras para el manejo de hilos en Android.

Este controlador se ejecuta de manera indefinida en el bucle while del método run, hasta que se interrumpe el hilo de ejecución y se pone a false el booleano isControllerRun. Las líneas más importantes para el patrón Proxy son:

- En la línea 3 se declara Actions que es la superclase del patrón Proxy, se comenta después.
- En la línea 9 se instancia Actions con el objeto proxy del patrón Proxy.
- En la línea 19 se obtienen todos los sensores remotos que hay registrados en la base de datos local del prototipo TFM Client en base al bssid (punto de acceso) y al label (etiqueta de ubicación).

- En las líneas 20 y 21, para cada uno de los sensores remotos, realizamos la invocación remota a través del patrón Proxy. Como puede observarse, no sabemos que protocolo de comunicaciones vamos a utilizar.

Continuamos con las clase Actions, esta clase es la superclase del patrón Proxy, puede ser una interfaz o bien una clase abstracta. En el prototipo TFM Client es una clase abstracta que tiene el método doAction como la siguiente figura:

```
1 public abstract class Actions {


2 public abstract String doAction(
            Context context,
            PrefFavoriteNetworkClientEntity networkClient,
            RemoteSensorsEntity remoteSensor,
            Map<String, Object> params);
3 }
```

Figura 6.15: Superclase del patrón Proxy en el prototipo TFM Client

La siguiente figura muestra la clase Proxy del patrón de diseño software Proxy, esta clase es la que decide que protocolo de comunicaciones se va a utilizar:

```
1  public class Proxy extends Actions {
2
3    @Override
4    public String doAction(Context context,
                          PrefFavoriteNetworkClientEntity networkClient,
                          RemoteSensorsEntity remoteSensor, Map<String,
                          Object> params) {
6        String jsonResponse;
7        Actions action;
8        PrefFavoriteNetworkClientDao daoFavoriteNetworkClient =
                             PrefFavoriteNetworkClientDaoImpl.getInstance(context);
9        try {
10           networkClient = daoFavoriteNetworkClient.getFavoriteEntity();
11           String controllerNameClass = networkClient.getControllerClassName();
12           action =
             (Actions) Class.forName("es.uned.tfm.client.clients.controller.proxies." +
                                    controllerNameClass).newInstance();
13           jsonResponse =
                     action.doAction(context, networkClient, remoteSensor, params);
14       }catch(Exception e){
15            jsonResponse = e.toString();
16            LogsUtil.log("Proxy.doAction().Exception. ERROR: ",
17                          LogsUtil.Levels.ERROR, e);
18       }
19       return jsonResponse;
20    }
21 }
```

Figura 6.16: Clase Proxy del patrón Proxy en el prototipo TFM Client

Por líneas quedaría:

- En la línea 1 se extiende de la superclase Actions del patrón Proxy.
- En la línea 10 y 11, obtenemos el nombre de la clase que se instanciará. Esta cadena de texto está guardada en una tabla de la base de datos local del dispositivo. En la aplicación real, se comprueba si existe algún

cliente favorito, si no existe, se corta el flujo de ejecución con una excepción.

- En la línea 12, se instancia la clase de un Action concreto que representa un protocolo de comunicaciones con la carga dinámica de clases.
- En la línea 13 se ejecuta el Action concreto del patrón Proxy.

Como tenemos 5 protocolos de comunicaciones en el prototipo TFM Client, tenemos 5 Actions concretos, cada protocolo tiene sus peculiaridades, se va a comentar el protocolo CoAP cuyo código tiene el siguiente aspecto:

```
1  public class CoapClientActions extends Actions {
2
3    private Context context;
4
5    @Override
6    public String doAction(Context context,
                           PrefFavoriteNetworkClientEntity networkClient,
                           RemoteSensorsEntity remoteSensor, Map<String,
                           Object> params) {
7      this.context = context;
8      String jsonResponse;
9      try {
10          Method method =
                 this.getClass().getDeclaredMethod(String.valueOf(params.get("action")),
                 PrefFavoriteNetworkClientEntity.class,
                 RemoteSensorsEntity.class,
                 Map.class);
11          jsonResponse = (String) method.invoke(this, networkClient, remoteSensor, params);
12     } catch (Exception e) {
13          LogsUtil.log("CoapClientActions.doAction().Exception. ERROR: ",
                        LogsUtil.Levels.ERROR, e);
14          jsonResponse = e.toString();
15        }
16      return jsonResponse;
17    }

18   public String deviceSensorsBatteryLASTAction(
                  PrefFavoriteNetworkClientEntity networkClient,
                  RemoteSensorsEntity remoteSensor,
                  Map<String, Object> params){

19      String hostname = remoteSensor.getRemoteDevicesEntity().getHostname();
20      int port = networkClient.getPort();
21      String context = "/devices_sensors_battery_last";
22      String url = "coap://" + hostname + ":" + port + context;
23      params.put("url", url);
24      return this.sendMessageGET(networkClient, remoteSensor, params, null);
25
26   }
27 }
```

Figura 6.17: Clase Action concreto del protocolo CoAP del patrón Proxy en el prototipo TFM Client

La clase de la figura anterior muestra parcialmente el código, la clase real tiene más métodos de invocación. Se observa que tiene, en la línea 6, el método doAction de la superclase Actions y además tiene el método de deviceSensorsBatteryLASTAction en la línea 18. Por líneas:

- Dentro del método doAction, en la línea 10, usando la reflexión en Java se consigue el método que se invocará dentro de la clase, en nuestro caso es deviceSensorsBatteryLASTAction. Este parámetro no viene de la clase cliente del controlador ni del patrón Proxy, si no que viene del servicio de Android RemoteBatterySensorService que a su vez lanza el hilo de ejecución del controlador (el método run), de esta forma desacoplamos al controlador del método de la clase Java de ejecutar una acción en concreto, como en este caso, obtener la última lectura de la batería.
- En la línea 11, se ejecuta, con la reflexión de Java, el método deviceSensorsBatteryLASTAction.
- En el método deviceSensorsBatteryLASTAction, líneas 18 y siguientes, prepara la invocación para el método sendMessageGET que contiene el código preciso para el protocolo CoAP.

Todas las clases relacionadas con los clientes de red en el prototipo TFM Client son buenas candidatas para ser empaquetadas en una librería externa como un componente. Un componente es una unidad independiente desplegable sin estado observable que puede ser usado por terceras partes [40]. Para conseguirlo, debemos de definir y documentar una interfaz clara para hacer un buen uso de la librería y empaquetar en un JAR todas las clases relacionadas y subir este JAR a un repositorio de componentes como Nexus, instalado en nuestra red local. Una vez que está subido al Nexus de nuestra red local, los proyectos que vayan a utilizar la librería deberán configurar gradle, añadiendo el repositorio de Nexus y añadiendo la librería al fichero gradle como cualquier otra. De esta forma, si varios proyectos utilizan este componente de clientes de red, las modificaciones y ampliaciones son muy sencillas, sólo tenemos que modificar la librería, compilarla y volver a subir otra versión de la librería al Nexus. Los proyectos clientes deben de compilar su proyecto con la nueva versión de la librería y ya tienen las modificaciones o ampliaciones listas para usarse. Si no se hiciera de esta forma y tuviéramos 20 proyectos con clientes de red, habría que copiar y pegar las modificaciones de los clientes de red a mano en los 20 proyectos.

6.4. PATRÓN DE DISEÑO SOFTWARE DE VARIOS PROCESOS

Otra aproximación es usar un proceso por puerto de conexión de forma que cada puerto de conexión realice una tarea en concreto, como por ejemplo, un proceso puede devolver la temperatura de un sensor y otro puerto la presión de otro sensor. Esto implica que debemos de tener varios procesos en funcionamiento en el mismo dispositivo, si tenemos 10 sensores en el dispositivo, implica tener 10 procesos en el dispositivo, de forma que cada proceso consume memoria del dispositivo, en contrapartida de los hilos de ejecución que podemos diseñar con el patrón Front Controller, Proxy y un API REST, mediante un proceso, se pueden recibir diferentes tipos de peticiones.

En este caso, cada proceso escuchando en cada puerto, espera una estructura de datos de carga fija.

En los prototipos este patrón se utiliza en el prototipo TFM Server debido a que tenemos 5 servidores en 5 puertos distintos para recibir peticiones, aunque se podrían haber reutilizado alguno, debido a que 3 utilizan el protocolo TCP (http, sockets TCP y ZeroMQ TCP) y 2 el protocolo UDP (sockets UDP y CoAP).

En el prototipo TFM Client se levantan dos sockets UDP, uno para recibir peticiones broadcast con el protocolo IPv4 y otro socket UDP para recibir peticiones por multicast.

6.5. PATRONES DE DISEÑO SOFTWARE DE MENSAJERÍA

Por completitud se resume este apartado. Los patrones de mensajería ya han sido comentados en secciones anteriores, algunos son más tradicionales como los tópicos y las colas, mientras que otros son más recientes e innovadores. En todo caso, se enumeran a continuación:

- **Colas**: comentado en la sección 3.2.6 en el protocolo AMPQ
- **Tópicos**: comentado en la sección 3.2.4 en el protocolo MQTT.
- **Pipeline**: comentado en la sección 3.2.7 en el protocolo ZeroMQ.
- **Request-Replay**: comentado en la sección 3.2.7 en el protocolo ZeroMQ.
- **Exclusive-Pair**: comentado en la sección 3.2.7 en el protocolo ZeroMQ.
- **Client-Server**: comentado en la sección 3.2.7 en el protocolo ZeroMQ.
- **Radio-dish**: comentado en la sección 3.2.7 en el protocolo ZeroMQ.
- **Survey**: comentado en la sección 3.2.9 en el protocolo NNG.
- **Bus**: comentado en la sección 3.2.9 en el protocolo NNG.

En los prototipos no se han utilizado ninguno de estos patrones de mensajería, debido a que no ha sido esencialmente necesario y su uso hubiera sido un poco forzado.

6.6. OTROS PATRONES DE DISEÑO SOFTWARE UTILIZADOS EN LOS PROTOTIPOS

En los prototipos se han utilizado algunos patrones de diseño software más que se resumen a continuación:

- **DAO (Data Access Object):** este patrón se utiliza para el acceso a la base de datos. En cada objeto DAO se definen que operaciones se pueden realizar con una entidad/tabla de la base de datos. DAO es

requerido por Room. Room es una librería de persistencia de base de datos SQLite recomendada por Google.

- **Singleton:** se utiliza en las clases DAO para conseguir el objeto que da acceso a las bases de datos.
- **Facade:** se utiliza en la clase de los servicios, oculta la complejidad para tratar con los servicios de la app.
- **MVVM (Modelo-Vista-VistaModelo):** se trata de un patrón recomendado por Google para el desarrollo de apps. Básicamente combina el patrón MVC (Modelo Vista Controlador) y el patrón Observer. Suele requerir al menos dos clases y un documento xml:
    - El documento xml que representa el diseño estático de la vista.
    - Una clase activity o fragment que maneja la vista. Esta clase es la controladora de la vista, maneja como se renderiza la vista y maneja los eventos de usuario.
    - La clase View Model que contiene el código para el manejo del modelo de datos.

  Desde la clase activity o fragment observamos atributos de la clase de modelo, cuando el modelo cambia, la vista cambia automáticamente. De esta forma, se separa el código que maneja la vista, del código que maneja el modelo de datos de la app.

CAPÍTULO 7

7. DESCUBRIMIENTO DE MÁQUINAS EN UNA RED M2M DESCENTRALIZADA

Los sensores y actuadores de una red M2M deben ser capaces de encontrar a otros dispositivos que necesiten dentro de la red de máquinas a la que estén conectadas. Esta problemática de máquinas inteligentes que se encuentran sin intervención de un operador humano, se trata en las siguientes subsecciones de este capítulo. Básicamente, tenemos dos escenarios:

- Un usuario humano se conecta a un punto de acceso con un ordenador o con un smartphone. El dispositivo que maneja el usuario encuentra de forma automática que sensores y actuadores existen en la red del punto de acceso para interactuar con ellos.
- Una máquina se conecta al punto de acceso y esta máquina busca a otros dispositivos que necesite, por ejemplo un actuador necesita de varios sensores.

Para buscar máquinas en una red M2M podemos utilizar varias técnicas, en los prototipos se han utilizado algunas de las siguientes subsecciones.

7.1. POLLING

De forma general, esta técnica consiste en interrogar de forma regular o periódica un recurso o varios recursos para obtener sus estados o bien para actualizarlos. En nuestro contexto de descubrimiento de máquinas, cada máquina conectada a la red buscará y sondeará de forma periódica a otras máquinas, a pesar de que es posible que no sea necesario buscar de nuevo a otras máquinas porque no hay cambios en la red de máquinas.

Esta técnica no resulta ser muy eficiente debido a que genera mucho tráfico de red, la comunicación es uno a uno múltiple y obliga a las máquinas a despertarse de forma regular para buscar a otras máquinas. Otro inconveniente es que puede haber un retardo de tiempo, en el cual una máquina está conectada a la red, pero aún no ha sido descubierta por otras máquinas.

La comunicación uno a uno múltiple utilizando comunicación unidestino no es eficiente y puede crear grandes retardos de tiempo, especialmente cuando la red es grande. Además, la mayor parte del trabajo lo realiza la máquina que está buscando a otras máquinas.

En los prototipos se ha utilizado esta técnica para descubrir máquinas. Aunque el concepto es sencillo, a nivel de implementación no resulta ser tan sencillo porque tenemos que programar todo el proceso. El prototipo TFM Client es la máquina que busca a otras máquinas en la red por la técnica de polling. El código de todo el proceso se muestra en la siguiente figura:

```
1 public class UnicastSubscribeController extends HandlerThread {
2
3    private UnicastSubscribeService service;
4    private Map<String, Object> params;
5    private boolean isControllerRun = true;
6    private Actions actions;
7
8    public UnicastSubscribeController(String name) {
9        super(name);
10   }
11
12    public UnicastSubscribeController(UnicastSubscribeService service, Map<String, Object> params) {
13        super("UnicastSubscribeController", HandlerThread.NORM_PRIORITY);
14        this.service = service;
15        this.params = params;
16        this.actions = new Proxy();
17    }
18
19
20 @Override
21 public void run() {
22
23    PreferencesUtil preferencesUtil = new PreferencesUtil(this.service);
24    ServicesFacade servicesFacade = ServicesFacade.getInstance();
25    PrefFavoriteNetworkClientDao daoFavoriteNetworkClient = PrefFavoriteNetworkClientDaoImpl.getInstance(this.service);
26    SubscribeRemoteDeviceBll subscribeRemoteDeviceBll = new SubscribeRemoteDeviceBll(this.service);
27    while (this.isControllerRun) {
28        LogsUtil.log("UnicastSubscribeController.run() Controller is running", LogsUtil.Levels.DEBUG);
29        int refresh = //[..] Obtener el tiempo de refesco para sondear dispositivos, Ej: 60000 ms
30        int timeoutICMP = //[..] Obtiene el timeout del protocolo ICMP. Ej: 1000 ms
31        int timeoutPORT = //[..] Obtiene el timeout del protocolo TCP/UDP. Ej: 1000 ms
32        PrefFavoriteNetworkClientEntity prefFavoriteNetworkClientEntity = daoFavoriteNetworkClient.getFavoriteEntity();
33        try {
34          String myIpDevice = NetworkUtil.getHostname(this.service);
35          String subnetMask = NetworkUtil.getSubnetMask(NetworkUtil.WLAN0);
36          SubnetUtils subnetUtils = new SubnetUtils(myIpDevice, subnetMask);
37          String[] listHostsCandidates = subnetUtils.getInfo().getAllAddresses();
38          LogsUtil.log("UnicastSubscribeController.run() --> Scanning Unicast " + listHostsCandidates.length + " hosts", LogsUtil.Levels.DEBUG);
```

```
39      try {
40          List<String> listHostsReachables = NetworkUtil.getListHostReachable(Arrays.asList(listHostsCandidates), timeoutICMP, myIpDevice);
41          List<String> listHostsFinally =    NetworkUtil.getListHostReachablesByProtocolAndPort(listHostsReachables,
                                                                  prefFavoriteNetworkClientEntity.getPort(), timeoutPORT);
42          for (String hostnameFinally : listHostsFinally) {
43              RemoteDevicesEntity remoteDevices = new RemoteDevicesEntity();
44              remoteDevices.setHostname(hostnameFinally);
45              RemoteSensorsEntity remoteSensor = new RemoteSensorsEntity();
46              remoteSensor.setRemoteDevicesEntity(remoteDevices);
47              String jsonResponse = this.actions.doAction(this.service, null, remoteSensor, this.params);
48              subscribeRemoteDeviceBll.subscribe(jsonResponse, ConstantsUtil.UNICAST_POOLING);
49          }
50       } catch (Exception e) {
51          LogsUtil.log("UnicastSubscribeController.run().Exception. ", LogsUtil.Levels.INFO, e);
52      }
53
54      } catch (Exception e) {
55    LogsUtil.log("RemoteBatterySensorController.run().Exception. FATAL ERROR: ", LogsUtil.Levels.ERROR, e);
56  }
57 }
```

Figura 7.1: Clase controladora UnicastSubscribeController que busca dispositivos por la técnica de polling en el prototipo TFM Client

La clase anterior tiene mucho código y se ha omitido alguna parte no relevante pero que dificulta la comprensión y la explicación del proceso. Se trata de otra clase controladora que es ejecutada por otro servicio de Android de forma similar a la clase controladora que invoca de forma remota a un servidor para obtener las lecturas del sensor de batería. A nivel funcional, el proceso realiza el sondeo/escáner en 5 pasos:

1. Líneas 34 a 37. Para realizar un escáner de la red, necesitamos saber todas las direcciones IP de la red en base a nuestra propia dirección IP y la máscara de red. En este paso, el proceso se apoya en una clase de la propia app llamada NetworkUtil y de una librería de Apache para realizar subnetting llamada SubnetUtils.
2. Línea 40. El segundo paso es realizar un escáner rápido a todas las máquinas obtenidas en el paso anterior mediante el protocolo ICMP y guardar que máquinas de la red están vivas. Es decir, con ICMP hacemos un ping rápido para saber que máquinas están vivas para el siguiente paso. De nuevo nos apoyamos en la clase de la app NetworkUtil que realiza esta labor. Para realizar un ping con Android se utiliza la clase InetAddress y su método isReachable(timeout). Aunque algunas JVM de algunos sistemas operativos no permiten realizar esta operación, a no ser que se tenga permiso de root (super usuario), según el JavaDoc de Android, si que se permite realizar esta operación de ping y se han hecho pruebas y funciona. Contra más pequeño es el timeout más rápido termina el escáner, pero es posible que un dispositivo receptor no responda a tiempo y no se descubra algún dispositivo. El ping puede estar cortado, por razones de seguridad por los firewalls, aunque en general, en las redes locales de puntos de acceso Wi-Fi no hay ningún problema.
3. Línea 41. Con la lista de máquinas vivas según el protocolo ICMP del paso anterior, se hace un test al puerto del protocolo TCP y del protocolo UDP. El puerto está guardado en las preferencias de la propia app como el cliente favorito y es configurable por el usuario. Esta línea se apoya también en la clase NetworkUtil de la propia app.
4. Líneas 43 a 47. Para cada host dentro del bucle for, si el host destino pasa la prueba anterior, se hace finalmente una petición para obtener el documento json de la figura 6.2 con la información relativa al dispositivo para ser guardada en el paso posterior. Comentar que para esta petición, se reutiliza el patrón Proxy del capítulo 6.3 con el cliente favorito configurado en el prototipo TFM Client.
5. El último paso es la línea 48, donde se guarda en la base de datos SQLite local del prototipo TFM Client toda la información relativa al documento json. Esta línea hace uso de la clase de negocio SubscribeRemoteDeviceBll y su método subscribe. En este método, se comprueba que el SSID del punto de acceso no sea desconocido al que está conectado el prototipo TFM Client, según Android con el texto <unknown ssid>. Este nombre de SSID puede darse por motivos de

permisos en Android. Después se deserializa el documento json. Si todo es correcto y no hay excepciones en el proceso de deserialización, se guarda la información del documento json a un modelo relacional de tablas de base de datos dentro de la caché de la app del propio dispositivo. La información se guarda aquí para que otros servicios de la propia app utilicen esta información para saber a que dispositivos tienen que hacer las peticiones, por ejemplo, para obtener la última lectura del sensor de un dispositivo que está conectado a un punto de acceso en concreto.

7.2. PUSH

La técnica de push consiste en notificar cuando cambia el estado de un recurso o para anunciar que un recurso es nuevo. Para una red de máquinas, si una máquina es nueva en la red, se trata de un nuevo recurso. La técnica de push se puede implementar utilizando la técnica de polling del apartado anterior para descubrir a otras máquinas. La diferencia fundamental es que la máquina nueva notifica de su presencia al resto de máquinas, enviando un documento json como el de la figura 6.2 en el paso 4 de polling y aprovechando la conexión, la máquina nueva recibe una respuesta del tipo de máquina remota que es, recibiendo otro documento json de la figura 6.2, teniendo ambas máquinas consciencia de la existencia de las dos. Dicho con otras palabras, mientras la máquina nueva descubre a otras máquinas, las máquinas ya desplegadas en la red saben de la existencia de la nueva máquina.

Esta estrategia es también uno a uno múltiple pero es una técnica más rápida para construir de forma dinámica la red de malla parcial de máquinas. En los prototipos no se han implementado está técnica de push uno a uno múltiple, si no que se han utilizado las técnicas de difusión del apartado 7.3 y de multidifusión del apartado 7.4 que son más eficientes.

7.3. DIFUSIÓN

La difusión es soportada por el protocolo UDP y consisten en enviar un mensaje a la dirección de difusión (broadcast) de la red con el fin de que todos los nodos de la red reciban la petición. Esta dirección es la última dirección IP de la red de la máscara de red y en el protocolo IPv4 se utiliza para este propósito.

Una máquina nueva entra a la red y envía un petición broadcast al punto de acceso para anunciar su presencia en la red. El punto de acceso reenvía a cada máquina de la red la petición con la información de la máquina nueva. Cada máquina ya desplegada en la red debe tener un socket UDP levantado para recibir las peticiones de la máquina nueva en un puerto específico y obtener la información, si la máquina no tiene este socket levantado en el puerto especificado, el dispositivo receptor ignora la petición.

Un problema es que el punto de acceso debe soportar y permitir la difusión de mensajes y en general no está soportada ni permitida por los

enrutadores en Internet, con el fin de no generar un mensaje que se propague por todos los nodos de Internet del mundo.

En el protocolo IPv6 no existe este concepto de broadcast y se utiliza la multienvío del apartado 7.4.

En los prototipos se ha utilizado esta técnica de broadcast. El prototipo TFM Server es la app que anuncia su presencia con esta técnica de broadcast y cuyo código es el siguiente:

```
1 public class BroadcastCommand implements DevicePublishCommand {
2
3     @Override
4     public String execute(Context context, Map<String, Object> params) {
5         String response = "";
6         try (DatagramSocket socketClient = new DatagramSocket()) {
7             DevicesDaoImpl deviceDao = DevicesDaoImpl.getInstance(context);
8             String jsonRequest = DevicesEntity.toJson(deviceDao.getAllEntitiesWithSensors());
9             byte[] bufferRequest = new byte[jsonRequest.length()];
10            String hostname = NetworkUtil.getHostname(context);
11            String broadcast = NetworkUtil.getBroadcast(hostname);
12            InetAddress inetAddresBroadcast = InetAddress.getByName(broadcast);
13            socketClient.setSoTimeout(ConstantsUtil.DEVICE_PUBLISH_TIMEOUT);
14            socketClient.setBroadcast(true);
15            DatagramPacket request =
16                  new DatagramPacket(bufferRequest, bufferRequest.length, inetAddresBroadcast,
17                                     ConstantsUtil.DEVICE_PUBLISH_BROADCAST_PORT);
18            request.setData(jsonRequest.getBytes("UTF-8"));
19            socketClient.send(request);
20        } catch (Exception e) {
21            LogsUtil.log("BroadcastCommand.execute().Exception. ERROR: ",
22                           LogsUtil.Levels.ERROR, e);
23            response = e.toString();
24        }
25        return response;
26    }
27 }
```

Figura 7.2: Command que envía una petición broadcast para anunciar su presencia en el prototipo TFM Server

El command de la figura anterior es ejecutado por la clase controladora DevicePublishController en el prototipo TFM Server, y este controlador es ejecutado desde un servicio de la app. Este controlador es muy sencillo y la estructura principal es muy similar al controlador de la figura 7.1, extiende de HandlerTread y se ejecuta su método run que invoca a este command de broadcast o el command del apartado 7.4 de los multienvíos.

Observando el código de la figura 7.2, se trata de una petición corriente por sockets UDP, pero hay algunos matices:

- En la línea 11 la dirección de envío es la de broadcast. La clase NetworkUtil de la app calcula esta dirección en base a la dirección IP obtenida en la línea anterior. Es muy importante pasar a este método una dirección IPv4, en IPv6 no existe esta dirección de broadcast. También se ha probado la dirección 255.255.255.255 y funciona, pero está en desuso y para evitar problemas no se ha utilizado.
- En la línea 14 se establece que el envío va a ser de tipo broadcast.

- En la línea 19 se realiza el envío pero no se espera ninguna respuesta por parte de ningún host.

En el prototipo TFM Client se recibe la petición de broadcast con el siguiente código:

```
1 @Override
2  public void run() {
3     ServicesFacade servicesFacade = ServicesFacade.getInstance();
4     SubscribeRemoteDeviceBll subscribeRemoteDeviceBll =
                                  new SubscribeRemoteDeviceBll(this.service);
5     try {
6         this.server = new DatagramSocket(ConstantsUtil.DEVICE_SUBSCRIBE_BROADCAST_PORT);
7         while (this.isControllerRun) {
8             try {
9                 byte[] bufferRequest = new byte[BUFFER];
10                final DatagramPacket request =
                                new DatagramPacket(bufferRequest, bufferRequest.length);
11                this.server.receive(request);
12                try {
13                    String ssid = NetworkUtil.getSSIDFromCurrentWifi(this.service);
14                    String bssid = NetworkUtil.getBSSIDFromCurrentWifi(this.service);
15                    if(ConstantsUtil.UNKNOWN_SID.equals(ssid)){
16                      //Imprime un log
17                    }else {
17                        String jsonRequest = new String(request.getData(),
19                                     0, request.getLength(), Charset.forName("UTF-8"));
21                        subscribeRemoteDeviceBll.subscribe(jsonRequest, ConstantsUtil.BROADCAST);
22                    }
23                } catch (Exception e) {
24                    //Imprime un log
25                }
26            } catch (Exception e) {
27                //Imprime un log
28            }
29        }
30    } catch (SocketException e) {
31      //Imprime un log y trata la excepcion fatal del socket
32    }
33 }
```

Figura 7.3: Controlador que recibe una petición broadcast en el prototipo TFM Client

El controlador de la figura anterior funciona de manera similar a los ya comentados en secciones anteriores. Se trata de un hilo de ejecución que es lanzado desde un servicio de la app. Al igual que en polling, este controlador guarda la petición, el documento json de la figura 6.2, en la base de datos del prototipo TFM Client en la línea 21. Lo más reseñable es el puerto de escucha, que está contenida en la constante DEVICE_SUBSCRIBE_BROADCAST_PORT, cuyo valor es 10001, para recibir peticiones por broadcast.

7.4. MULTIENVÍO

El multienvío o multicast consiste en enviar una petición a un grupo de nodos y que sólo este grupo de nodos reciba la petición, pudiendo tener varios grupos en la red. El multienvío se implementa en los encaminadores a nivel del protocolo IP con las clases de red tipo D en el protocolo IPv4. Estas direcciones no representan nodos o máquinas reales en la red, si no como se ha comentado, representan grupos de nodos. Esto obliga a que los nodos

receptores deben registrarse previamente en un grupo para recibir las peticiones. Existen protocolos y varias estrategias de encaminamiento para este propósito que varían del protocolo IPv4 e IPv6. En la práctica, en IPv4 se envía un paquete UDP por la red a una dirección de multienvío. En el prototipo TFM Server envía el multienvío con el siguiente código:

```
1 public class MulticastCommand implements DevicePublishCommand {
2
3    @Override
4    public String execute(Context context, Map<String, Object> params) {
5        String response = "";
6        try (DatagramSocket socketClient = new DatagramSocket()) {
7            DevicesDaoImpl deviceDao = DevicesDaoImpl.getInstance(context);
8            String jsonRequest = DevicesEntity.toJson(deviceDao.getAllEntitiesWithSensors());
9            byte[] bufferRequest = new byte[jsonRequest.length()];
10            InetAddress inetAddresBroadcast =
                                  InetAddress.getByName(ConstantsUtil.MULTICAST_IP_SENSORS);
11            socketClient.setSoTimeout(ConstantsUtil.DEVICE_PUBLISH_TIMEOUT);
12            DatagramPacket request =
13               new DatagramPacket(bufferRequest, bufferRequest.length,
                 inetAddresBroadcast, ConstantsUtil.DEVICE_PUBLISH_MULTICAST_PORT);
                 request.setData(jsonRequest.getBytes("UTF-8"));
14            socketClient.send(request);
15        } catch (Exception e) {
16            LogsUtil.log("MulticastCommand.execute().Exception. ERROR: ",
                          LogsUtil.Levels.ERROR, e);
17            response = e.toString();
18        }
19        return response;
20    }
21 }
```

Figura 7.4: Command que envía una petición multicast en el prototipo TFM Server

La figura anterior es una petición corriente UDP por multicast, pero como en broadcast, existen una serie de matices:

- En la línea 10 la dirección de envío está guardada en una constante de la app, en MULTICAST_IP_SENSORS de la clase ConstantsUtil, cuyo valor es 230.0.0.1, reservada para que los sensores publiquen su presencia.
- En la línea 14 se envía la petición, pero no se espera ninguna respuesta, al igual que en broadcast.

En el prototipo TFM Client se recibe la petición multicast de la siguiente manera:

```
1  private MulticastSocket server;
2  private InetAddress group;

3  @Override
4  public void run() {
5     ServicesFacade servicesFacade = ServicesFacade.getInstance();
6     SubscribeRemoteDeviceBll subscribeRemoteDeviceBll =
                   new SubscribeRemoteDeviceBll(this.service);
7
8     try {
9         this.server = new MulticastSocket(ConstantsUtil.DEVICE_MULTICAST_BROADCAST_PORT);
10        this.group = InetAddress.getByName(ConstantsUtil.MULTICAST_IP_SENSORS);
11        this.server.joinGroup(this.group);
12
13        while (this.isControllerRun) {
14            try {
15                byte[] bufferRequest = new byte[BUFFER];
16                final DatagramPacket request =
                              new DatagramPacket(bufferRequest, bufferRequest.length);
17                this.server.receive(request);
18                try {
19                    String jsonRequest =
                               new String(request.getData(), 0,
                                           request.getLength(), Charset.forName("UTF-8"));
20                    subscribeRemoteDeviceBll.subscribe(jsonRequest, ConstantsUtil.MULTICAST);
21                } catch (Exception e) {
22                  //Imprime un log
23                }
24            } catch (Exception e) {
25              //Imprime un log
26            }
27        }
28    } catch (Exception e) {
29    //Imprime un log y trata la excepcion fatal del socket
30    }
31 }
32
33  @Override
34   public boolean quit() {
35     this.isControllerRun = false;
36     try {
37         this.server.leaveGroup(this.group);
38         this.server.close();
39     } catch (Exception e) {
40         LogsUtil.log("MulticastSubscribeController.quit() ERROR: ", LogsUtil.Levels.ERROR, e);
41     }
42     return true;
43  }
```

Figura 7.5: Controlador que recibe una petición multicast en el prototipo TFM Client

Observamos que el bloque de código de la figura anterior se encuentra un método run, por lo que es un hilo de ejecución de tipo HandlerThread como en controladores anteriores, a su vez, este controlador es lanzado por un servicio de la app. La recepción de una petición multicast UDP es poco un diferente a una recepción unicast UDP en los siguientes puntos:

- En la línea 9, donde es un objeto del tipo MulticastSocket, en vez de un DatagramSocket.

- Otra diferencia fundamental es que nos registramos en un grupo multicast en la línea 10 y 11. La constante MULTICAST_IP_SENSORS tiene el valor 230.0.0.1.
- En multicast debemos dejar el grupo cuando el servicio se para. Esto se hace en el método quit, en la línea 37.

El resto de código es muy similar a un socket UDP unicast, este controlador también registra el sensor remoto en la base de datos local de la app, tal y como se hace por el método broadcast y unicast múltiple.

7.5. PROTOCOLOS DE MENSAJERÍA DE TÓPICOS

En los prototipos no se ha implementado el descubrimiento de máquinas con protocolos de mensajería a nivel de aplicación con tópicos, debido a que necesitamos un broker de mensajes y va en contra de uno de los objetivos fundamentales de este trabajo, que es la descentralización de máquinas, pero otra forma de conseguir, a nivel teórico, el multienvío y publicar la presencia de dispositivos, es subir al nivel de aplicación y utilizar un protocolo de mensajería de tópicos, como puede ser MQTT y que pueda funcionar con cualquier versión del protocolo IP. Esto implica que debemos de tener un broker de mensajes que puede estar instalado en el punto de acceso y que sea lo más transparente a los dispositivos, al igual que es transparente un servidor DHCP instalado en un punto de acceso.

Dejando fuera los protocolos de enrutamiento del nivel IP, los broker de mensajería se pueden utilizar para el descubrimiento de máquinas. Esta estrategia sólo tiene sentido y resultaría ser eficiente si:

- Como se ha comentado, que el broker esté instalado en el propio punto de acceso y cuya dirección IP es bien conocida por los nodos de la red porque la dirección IP del punto de acceso forma parte de la configuración DHCP que reciben los clientes.
- En una máquina cuya dirección IP es conocida de antemano por los clientes, por ejemplo, la penúltima dirección IP de la máscara de red (la anterior dirección IP a la dirección de broadcast) y que el servidor DHCP nunca asignará a otro nodo de la red.
- Que el broker tenga un socket UDP para recibir peticiones broadcast o multicast para ser encontrado con facilidad. En este caso, un cliente envía una petición broadcast o multicast a un puerto específico a la red en busca de un broker. El broker responde la petición broadcast o multicast con un documento json con toda la información necesaria para utilizar el broker, como su dirección IP, versión, tipo de broker y los tópicos que maneja.

En cualquiera de los casos, la nueva máquina envía el mensaje de su anunciamiento como nueva máquina a un tópico predefinido para este fin del broker y a continuación la máquina nueva puede registrarse en este tópico

para recibir las futuras peticiones de las siguientes máquinas nuevas, en caso de ser necesario.

El broker puede utilizar la técnica de comunicación uno a uno múltiple para transmitir el mensaje de anunciamiento del nuevo dispositivo a todos los demás dispositivos, pero en este caso el mensaje es certero y cada máquina nueva no genera un tráfico denso de red, debido a que el broker tiene un mapa completo de las máquinas que están registradas en el tópico.

7.6. RESUMEN SOBRE ESTE CAPÍTULO

Las técnicas comentadas sobre el descubrimiento de máquinas no son excluyentes y se pueden utilizar todas de forma conjunta o sólo 1, según las características de la red a la que estemos conectados. En la siguiente tabla se muestra un cuadro resumen, por orden de más eficiente (arriba) a menos eficiente (abajo) con las características de cada uno de los mecanismos.

Mecanismo	Características
Multicast	· No genera tráfico denso de red. Las peticiones multicast van dirigidas sólo a los miembros del grupo que se han registrado previamente. · Compatible con IPv4 y IPv6. · Sólo funciona con el protocolo UDP. · Es sencilla la implementación en las máquinas (sensores/actuadores) emisoras y receptoras del multicast. · Hay que implementar mecanismos para enlazarse con un grupo y dejar el grupo. · Se puede utilizar en puntos de acceso de redes de área local privadas. En este escenario de red privada, es posible que requiera configuración adicional si tenemos varias subredes repartidas en varios enrutadores o puntos de acceso. · Para máquinas conectadas directamente Internet, tendremos que contratar con la operadora de Internet el multicast y puede que no sea posible hacerlo. · Todo el trabajo de los multienvíos lo hace la electrónica de red y libera a los dispositivos (sensores/actuadores) de realizar un envío uno a uno múltiple.
Brokers MQTT	· Requiere de un software que de momento no está extendido en los puntos de acceso y tendremos que dedicar una máquina de tipo computadora para el multienvío de los mensajes. · Las máquinas (sensores/actuadores) necesitan una librería para entenderse con el broker. Si cambiamos de broker tendremos que reimplementar la mensajería en las máquinas. · No genera tráfico denso de red. El broker sólo envía los mensajes a los consumidores que se han registrado al tópico. · Compatible con IPv4 y IPv6. · Hay que implementar mecanismos para enlazarse a los tópicos y dejarlos.

	· Se puede utilizar en Internet, el broker puede estar conectado a Internet y no requiere nada especial de enrutamiento de electrónica de red, debido a que el broker funciona en la capa de aplicación. · Todo el trabajo de los multienvíos lo hace el broker y libera a los dispositivos (sensores/actuadores) de realizar un envío uno a uno múltiple para buscar a otros dispositivos. · Dependiendo del broker, puede funcionar con TCP y/o UDP.
Broadcast	· Sólo funciona en IPv4 y en redes de área local privadas. · Muy simple su uso, no requiere mecanismos para enlazarse ni para desenlazarse a un grupo de multienvío. · Es sencilla la implementación en las máquinas (sensores/actuadores) emisoras y receptoras del broadcast. · Genera tráfico denso en la red, el broadcast se envía a todas las máquinas conectadas, sean estas o no interesadas en la comunicación. · Sólo funciona con el protocolo UDP.
Push y Polling Unicast Múltiple	· Los escáneres de las máquinas (sensores/actuadores) generan mucho tráfico de red. · Todo el trabajo pesado de búsqueda lo realizan las máquinas (sensores/actuadores), puede provocar un gasto energético importante en dispositivos restringidos. · Puede funcionar con TCP y UDP. · La implementación de los escáneres en las máquinas (sensores/actuadores) es mucho más compleja que la implementación con multicast y con broadcast. · Se tarda cierto tiempo en escanear la red, si la red es grande pueden haber retardos de tiempo importantes, debido a que la comunicación es uno a uno múltiple. · Puede funcionar en redes de área local privadas y puede funcionar en Internet con ciertas restricciones. Si estamos en una red acotada SIM M2M es posible su uso. Si estamos en una red de Internet con miles de dispositivos es inviable realizar los escáneres. · Con push unicast se consigue formar la red de malla parcial en menos tiempo, que con sólo la técnica de polling.

Tabla 7.1: Cuadro resumen de mecanismos para el descubrimiento de máquinas en redes M2M

CAPÍTULO 8

8. LOS PROTOTIPOS EN ANDROID

Después de los capítulos anteriores se puede abordar las cuestiones más importantes del desarrollo en Android de los prototipos. En el capítulo 5 se ha presentado una arquitectura de malla parcial con varios modelos de comunicación descentralizados, todos ellos del tipo cliente-servidor. En el capítulo 6 se ha presentado como manejar las comunicaciones entre los distintos actores (sensores/actuadores) de la comunicación mediante protocolos y patrones de diseño software. Y en el capítulo 7 se han presentado mecanismos para que las máquinas pueden encontrarse entre sí para formar la red de malla parcial descentralizada de forma dinámica.

8.1. PLATAFORMAS ANDROID PARA EL DESARROLLO DE LOS PROTOTIPOS

En Android tenemos varias plataformas para desarrollar apps. Según el dispositivo para el que estemos desarrollando, tenemos las siguientes posibilidades:

- **Smartphones/Tablets:** es posiblemente la plataforma más conocida y extendida de todas. Los dispositivos son pantallas inteligentes táctiles como pueden ser teléfonos inteligentes y tablets. La última versión estable es la versión 10 [41], pero ya están publicando la 11.
- **Wear OS:** se trata del sistema operativo para relojes inteligentes [42].
- **Android TV:** plataforma de desarrollo de apps para televisores inteligentes [43].
- **Android para coches:** se trata de la plataforma de Android Auto, es un asistente para coches integrado en el sistema de infoentretenimiento del vehículo [44].
- **Android Things:** Es una plataforma Android para IoT cuyo hardware base es una Raspberry PI 3B y un NXP iMX7D [45]. Aunque el autor de este trabajo no dispone del hardware necesario, podría haber comprado un kit de desarrollo. Hubiera sido la elección de la plataforma más natural para este trabajo pero el proyecto Android Things está de momento descontinuado, a favor de los asistentes como los altavoces inteligentes que están más de moda en la actualidad.
- **Chrome OS:** es un sistema operativo basado en Google y utiliza el navegador web Chrome como base para la interfaz de usuario [46]. Chrome OS se puede encontrar en dispositivos como Smart TV Box y portátiles.
- **Google Assistant:** son los asistentes de Google en forma de altavoces inteligentes [47], están muy de moda en la actualidad y parece que los fabricantes están siguiendo esta línea de trabajo por el número de ventas, por este motivo la línea de trabajo de Android Things está ahora

mismo descontinuada. Google Assistant también está disponible para otras plataformas, como smartphones, tablets y smart TV. Como los smartphones y tablets tienen un micrófono se podría haber implementado una app con un asistente de voz que escuchara comandos de voz y que ejecute acciones. Google tiene un servicio de voz online que convierte la voz en texto que puede ser tratado por la app. No se ha implementado en los prototipos, pero seguramente habría problemas para escuchar la voz y utilizar el servicio de Google, cuando la app deja de estar en primer plano y cuando el dispositivo entra en modo de suspensión en la plataforma de smartphones y tablets. Resumiendo, este servicio funciona con conectividad a Internet, la app envía la voz capturada del micrófono, Google la procesa y la app recibe la cadena de texto procesada por el servicio de Google.

De las anteriores plataformas, se ha escogido la primera de los smartphones y tablets, debido principalmente a que tengo hardware para realizar las necesarias pruebas con dispositivos físicos. Android Things está descontinuada y no dispongo de hardware, y el resto no tengo el hardware para poder realizar las pruebas. Si que tengo una Android TV, pero carece de sensores y los actuadores que tiene están limitados a una televisión. En Android TV, se podría implementar una app similar a TFM Client, adaptada a la interfaz de usuario de Android TV pero con el fin de mostrar información de los sensores remotos de la red con gauges y gráficas.

8.2. ARQUITECTURA DE LOS PROTOTIPOS

En total se han implementado dos prototipos, cada uno, tiene distintas funcionalidades:

- **TFM Server (sensor):**
  - Lee los sensores internos al dispositivo (batería y proximidad) donde está instalado y guarda las lecturas en una base de datos SQLite local al dispositivo.
  - Realiza invocaciones periódicas por broadcast o multicast para publicar su presencia al resto de dispositivos de la red.
  - Tiene en total 5 servidores levantados para recibir invocaciones remotas del prototipo TFM Client (actuador) o de cualquier cliente y servir las lecturas de sus sensores y para servir la configuración de este dispositivo.
- **TFM Client (actuador):**
  - Maneja sus actuadores internos, el altavoz y la cámara flash del dispositivo donde está instalada.
  - Recibe publicaciones por broadcast y multicast de otros dispositivos.
  - Busca a otros dispositivos por unicast (multienvío uno a uno múltiple).

- ○ Realiza invocaciones remotas al prototipo TFM Server y en función de las lecturas remotas maneja sus actuadores.

A nivel funcional, la idea fundamental es que si tapamos el sensor de proximidad del prototipo TFM Server (cámara delantera), se encienda el flash de la cámara del prototipo TFM Client. Otro aspecto funcional es si el prototipo TFM Server tiene menos del 20% de batería, el prototipo TFM Client emite una alarma sonora mediante el altavoz interno. En siguiente figura se puede observar de forma gráfica la arquitectura:

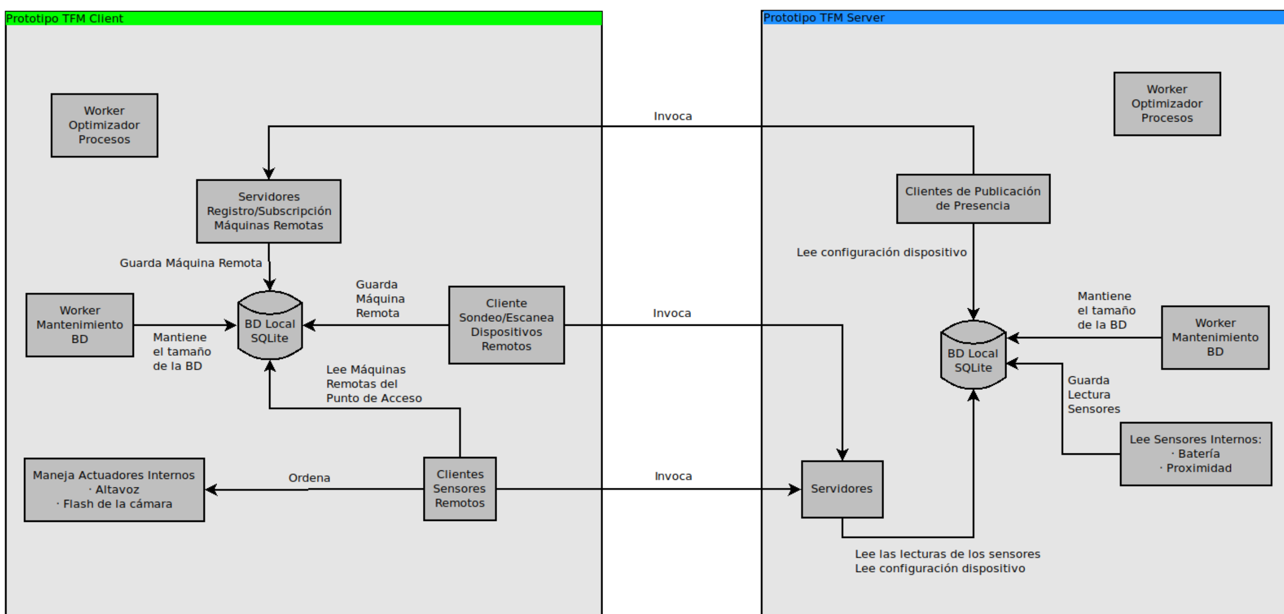


Figura 8.1: Arquitectura de los prototipos

En la figura anterior se observa como existe una base de datos central en ambos prototipos, donde todos los procesos leen o almacenan información. Por tanto, la base de datos local actúa de buffer para todos los procesos. En la figura anterior también se observa como unos procesos son productores de información y otros procesos consumen la información de la base de datos local. Por prototipos:

- **TFM Server.**
  - ○ <u>Productores:</u>
    - ▪ Procesos que leen los sensores internos para almacenar estas lecturas en la base de datos local.
  - ○ <u>Consumidores:</u>
    - ▪ Los servidores consumen la información de los sensores generada y la configuración del dispositivo de la base de datos local y la sirven a los clientes remotos.

        - El proceso de publicación consume la configuración del dispositivo de la base de datos y la envía por broadcast o multicast.

- **TFM Client.**
    - Productores:
        - Los servidores de broadcast y multicast almacenan la información que reciben de los clientes remotos (TFM Server) en la base de datos local.
        - El cliente que escanea la red por unicast también almacena la información de los dispositivos que encuentra en la base de datos local.
    - Consumidores:
        - Los procesos que realizan invocaciones a los sensores remotos consumen la información de los dispositivos remotos que se han encontrado los productores.

En la figura anterior también hay dos procesos de tipo worker en ambos prototipos. Un worker es un servicio en segundo plano que realiza tareas como actualizar datos de un servidor remoto o, como en este caso, mantener estable la base de datos manteniendo un número de filas en algunas tablas de la base de datos de la app. Dada la importancia de los procesos, los servicios y los workers en background en Android se tratan en la sección siguiente.

Es importante diseñar una arquitectura escalable, tal y como dicen en [48] el éxito es fácil, el éxito del mantenimiento es difícil. La arquitectura propuesta de los prototipos es escalable y permite añadir nuevos servicios, nuevos workers, nuevos métodos de búsqueda de dispositivos y nuevos sensores.

Para finalizar este apartado, es importante comentar que esta arquitectura corresponde con el modelo 1 de comunicaciones de la sección 5.3.1, donde los actuadores (TFM Client) realizan peticiones regulares y periódicas a los sensores remotos (TFM Server) para obtener la información del valor de sus lecturas. También es importante recalcar que ambos prototipos son clientes y al mismo tiempo son servidores, tal y como se explicó en el apartado 5.2.3:

- TFM Client es cliente porque realiza invocaciones remotas para obtener lecturas de sensores remotos y al mismo tiempo es servidor porque tiene dos servidores UDP para recibir publicaciones de presencia de dispositivos remotos por broadcast y multicast. También es cliente porque realiza invocaciones por unicast a los servidores para descubrir máquinas remotas.
- TFM Server es servidor porque recibe invocaciones remotas para servir la lectura de sus sensores y para servir su configuración. También es

cliente porque realiza peticiones broadcast o multicast para anunciar su presencia.

8.3. SERVICIOS E HILOS DE EJECUCIÓN EN ANDROID EN LA PLATAFORMA DE SMARTPHONES/TABLETS

El API de Android provee de una serie de herramientas para lanzar servicios. De forma general, la definición clásica de un servicio o daemon (según el sistema operativo) es un proceso que se ejecuta automáticamente en background (segundo plano), es decir, no hay ningún usuario que toque un botón para lanzar el servicio, si no que el servicio se ejecuta por si solo ante ciertos eventos del sistema, por ejemplo, cuando arranca el sistema operativo. El usuario no suele percibir que se ha lanzado el servicio porque el proceso no tiene interfaz gráfica que informe al usuario del trabajo que está realizando el proceso. No obstante y en general en cualquier sistema operativo con entorno gráfico, es buena práctica informar al usuario que existe un proceso en ejecución o que está realizando algún trabajo finito, se puede informar al usuario en la bandeja del sistema del sistema operativo o en un entorno con Android, en la barra de notificaciones.

En este contexto, cuando hablamos de procesos, simplemente se debe entender como un bloque de código fuente que la CPU procesa para su ejecución. Es decir, no se está utilizando el término proceso única y exclusivamente como una unidad de ejecución independiente y se puede observar en la tabla de procesos y el administrador de procesos del sistema operativo.

Los prototipos levantan varios de estos procesos autónomos, como los mostrados en la figura 8.1, como el proceso que lee de forma periódica el sensor de batería, como el proceso que levanta un servidor http o como el proceso que realiza peticiones por broadcast a la red para publicar su presencia. De forma resumida, estos procesos se deben comportar de la siguiente manera y deben tener las siguientes características:

- Algunos procesos realizan un pequeño trabajo y se duermen un cierto tiempo para luego despertar y continuar con el trabajo repetitivo, como el proceso de publicación de broadcast. Estos procesos deben despertarse de forma periódica y en momentos, más o menos, exactos y periódicos, por ejemplo cada minuto.
- Otros procesos deben de estar en ejecución de forma indefinida, como los servidores de los protocolos de comunicaciones que procesan las peticiones, como un servidor de sockets TCP.
- Los procesos deben seguir funcionando cuando se sale de la app de manera normal, se dice normal porque un cierre forzado cerrará todos los procesos relacionados con la app.
- Los procesos deben rearrancarse cuando el sistema operativo se reinicie y vuelva a estar operativo sin que el usuario tenga que abrir de nuevo la app.

- Los procesos deben funcionar con normalidad cuando el dispositivo entra en modo suspensión.
- Siguiendo con el punto anterior, algunos de estos procesos deben de tener conectividad de red cuando el dispositivo entra en modo de suspensión, un ejemplo son los actuadores que realizan invocaciones remotas a los sensores.
- Los procesos deben ser configurables para que el usuario pueda, entre otras cosas, parar o arrancar los procesos o configurar alguna parte importante del proceso como la frecuencia de tiempo en la que el proceso se despierta para realizar su trabajo.

8.3.1. Los Servicios y los Problemas en Android en la Plataforma de Desarrollo para Smartphones/Tablets

En la guía de Android Developer [49], hay abundante información sobre como lanzar procesos, servicios, workers y jobs (trabajos) pero no existe un API concreto que se comporte y que tenga por si sola todas las características comentadas de los puntos anteriores del apartado 8.3. Por tanto, se tienen que combinar varias de estas API para conseguir el resultado deseado y configurar las apps con ciertos permisos para que puedan funcionar como es debido en una plataforma para smartphones y tablets.

Es problemático cuando salimos de la app, Android cierra los procesos de la app por si mismo en base a sus estrategias de optimización de procesos sin previo aviso. Y es especialmente problemático cuando el dispositivo entra en modo de suspensión, los procesos dejan de funcionar o funcionan a intervalos de tiempo con ventanas de tiempo que establece Android, si además estos procesos necesitan conectividad de red este problema se agrava aún más.

Otro agravante son las restricciones de procesos en segundo plano que se imponen a partir de Android 8 (API LEVEL 26) que se pueden encontrar en la guía de Android [50]. Por si fuera poco, algunos fabricantes de smartphones tienen sus propios optimizadores de batería y que restringen la ejecución de procesos en segundo plano, especialmente si estos precisan de conectividad de red.

Todos estos problemas vienen determinados por el ahorro de batería y al mal uso que hacen algunas apps, que sin motivo justificado, drenan la batería de los smartphones y tablets con sus procesos en segundo plano. Por consiguiente, Android ha puesto cartas en el asunto y ha impuesto restricciones para los procesos en segundo plano, especialmente a partir de la versión 8 de Android, lo que ha dificultado enormemente el desarrollo de los prototipos para la plataforma de smartphones y tablets.

En la siguiente sección se va a explicar con detalle como se ha conseguido para que las apps funcionen a pesar de todos los problemas comentados y encontrados en esta sección.

8.4. SERVICIOS E HILOS DE EJECUCIÓN EN LOS PROTOTIPOS

Los prototipos levantan una serie de procesos que deben estar de forma permanente en ejecución y otros pueden dormir un tiempo y después levantarse para realizar su trabajo y luego dormirse de nuevo para levantarse después y continuar trabajando. Algunos de estos procesos pueden encolarse y ejecutarse con una cierta frecuencia pero no son críticos para el funcionamiento de los prototipos, estos últimos son los workers de mantenimiento. En las siguientes figuras, usando la notación BPMN que es más rica para definir y diseñar procesos, se puede observar las arquitecturas de servicios y procesos que se ha seguido en los dos prototipos, usando las API de Android:

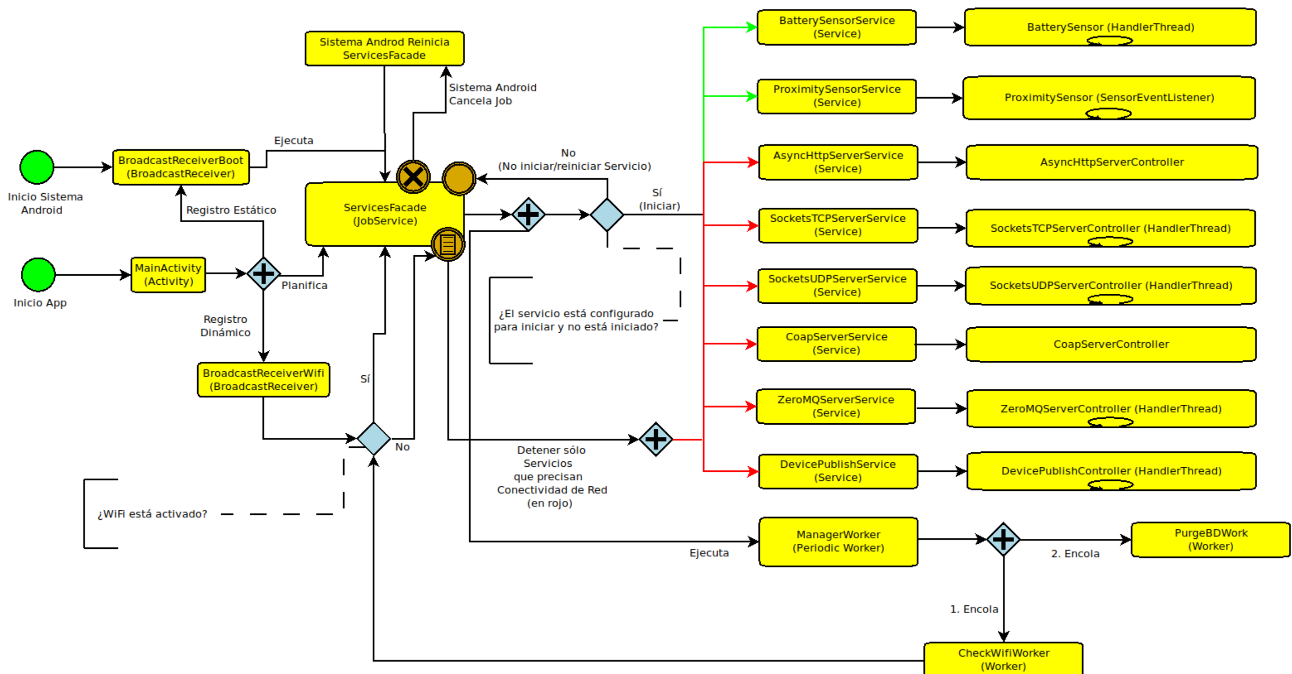


Figura 8.2: Diagrama de Procesos BPMN en el prototipo TFM Server

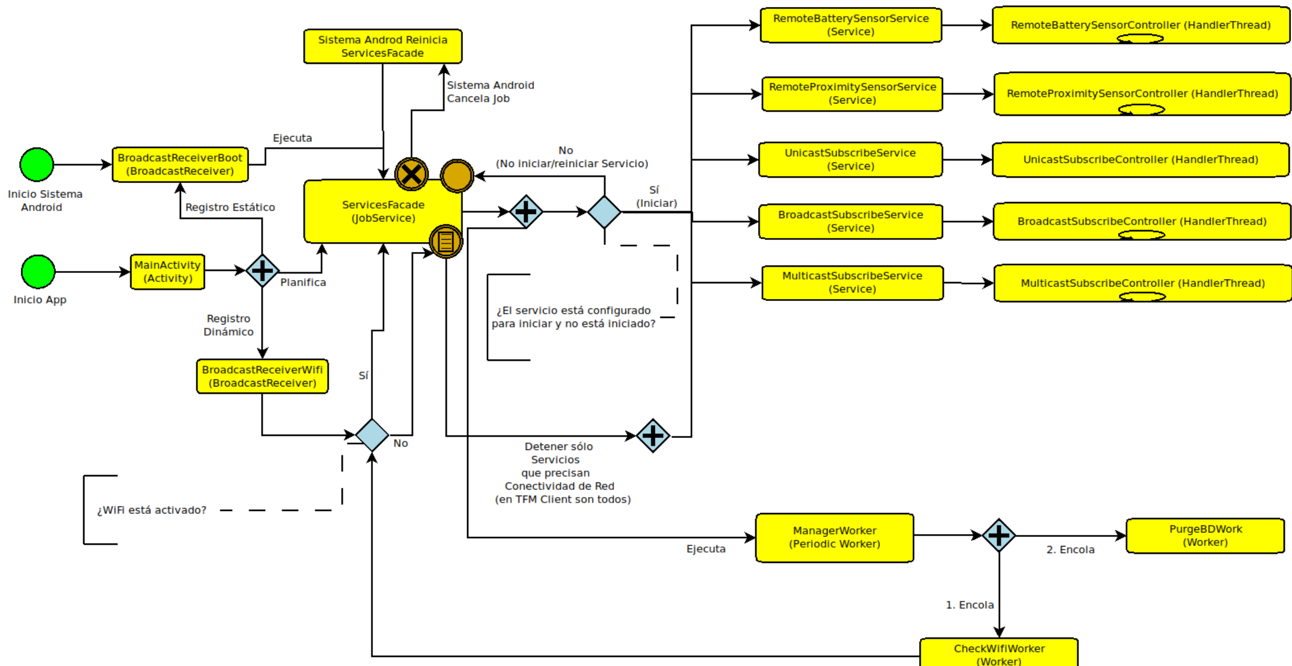


Figura 8.3: Diagrama de Procesos BPMN en el prototipo TFM Client

Ambas arquitecturas de procesos son similares pero realizan funcionalidades distintas, se van a comentar de forma genérica las dos. En las cajas amarillas, si existe una palabra entre paréntesis, indica que tipo de API de procesos de Android se está utilizando. El proceso BPMN comienza en un círculo verde, se observa que hay dos en cada prototipo, puede ser porque el usuario abre la app, o bien porque el sistema Android se inicia. El proceso completo de ambos prototipos no tiene representación de fin de proceso BPMN con un círculo rojo porque el funcionamiento normal del proceso completo BPMN, en ambos prototipos, es que se ejecuten indefinidamente. Puede haber un fin de proceso completo BPMN si el dispositivo se apaga o bien se realiza un cierre forzado de la app, en cuyo en este último caso toda la app se cierra y todos sus procesos e hilos de ejecución relacionados terminan. También un proceso puede terminar porque el usuario lo desactiva desde la pantalla de configuración de la app.

Es importante comentar que, la primera vez que se instala la app, al menos se debe ejecutar una vez la app para que Android registre el Broadcast Receiver estático de tipo boot.

Una vez que, se abre la app o se inicia el sistema Android, se lanza el job ServicesFacade que es de tipo JobService. Esta pieza software es el núcleo de los procesos en ambos proyectos y es la responsable de mantener, lanzar y, en su caso, parar el resto de procesos de las apps.

ServicesFacade lee la configuración de la app (por simplicidad, en el diagrama no está representado, pero si está indicado en la bifurcación del condicional, en el rombo azul vacío que está más arriba) y en función de esto,

inicia los procesos de la app, siempre que el proceso en particular no esté ya iniciado.

ServicesFacade lanza dos tipos de procesos, por un lado, lanza procesos de tipo Service y por otro de tipo Worker.

Los procesos de tipo Service lanzan a su vez hilos de ejecución de tipo HandlerThread, SensorEventListener y otras veces no se lanza nada particular del API de Android. No se lanza nada de Android si el API que estamos usando no requiere que se lance ningún hilo o proceso en particular. Por ejemplo, en CoapServerController, el API del core de CoAP ya tiene sus mecanismos internos para gestionar sus hilos cuando se arranca el servidor CoAP.

En el prototipo TFM Server, se observa que como hay flechas verdes, que indican que el servicio no requiere conectividad de red y otras flechas están en rojo, que indica que el servicio precisa conectividad de red.

Por otra parte, están los workers, que realizan tareas de mantenimiento en ambos prototipos y que no requieren una ejecución inmediata y que se puede posponer su ejecución para un momento posterior.

Todas estas API de Android se comentan en las subsecciones siguientes de este capítulo.

8.4.1. Broadcast Receiver

Los Broadcast Receiver son utilizados en las apps para recibir eventos del sistema. Android lanza un evento, como por ejemplo, el inicio del sistema, y las apps que están escuchando este evento lo reciben y ejecutan código de la propia app. A medida que Android ha ido subiendo de versión, se han ido prohibiendo los Broadcast Receiver estáticos [51], que son declarados en el manifest.xml de la app y que pueden ser recibidos y procesados por las apps, cuando estas se encuentran en segundo plano. En los dos prototipos se utilizan dos, uno de cada tipo, de tipo estático y de tipo dinámico.

8.4.1.1 Broadcast Receiver Estático

Las apps utilizan este Broadcast Receiver de tipo estático, concretamente se trata del Broadcast Receiver que escucha el evento de cuando el sistema Android inicia. Esto posibilita que los procesos de las apps arranquen cuando el dispositivo se reinicie. Para declarar un Broadcast Receiver de este tipo, tenemos que hacer tres cosas. La primera de todas, es que el Broadcast Receiver no esté prohibido en la guía de Android, especialmente si en versiones recientes de Android se encuentra prohibido, no recibiremos el Broadcast Receiver del sistema y la app no funcionará como se espera. La segunda es escribir el código fuente como la siguiente figura:

```
1  public class BroadcastReceiverBoot extends BroadcastReceiver {
2
3      @Override
4      public void onReceive(Context context, Intent intent) {
5          LogsUtil.initLogsDao(context);
6
7          final String action = intent.getAction();
8          LogsUtil.log("BroadcastReceiverBoot.onReceive().
                   Context: " + context.toString()
                   + " Action: " + action, LogsUtil.Levels.INFO)
9          if (Intent.ACTION_BOOT_COMPLETED.equals(action)) {
10             JobInfo.Builder builder =
                       JobsServicesUtil.buildJob(context,
                                                  JobsServicesUtil.JOB_ID_SERVICES_FACADE,
                                                  ServicesFacade.class);
11             JobsServicesUtil.launchJob(context, builder.build());
12         }
13     }
14 }
```

Figura 8.4: Broadcast Receiver de Tipo ACTION_BOOT_COMPLETE en el prototipo TFM Server

Por líneas, se comentan las partes más importantes de la figura anterior del Broadcast Receiver:

- En la línea 1, la clase extiende de BroadcastReceiver y esto nos obliga a extender del método onReceive.
- En la línea 9, se filtra en el condicional (bloque if) que tipo de evento se ha recibido, si es del tipo ACTION_BOOT_COMPLETE, se ejecuta su bloque interno que básicamente construye y lanza el JobService de ServicesFacade a través de una clase auxiliar JobsServicesUtil que forma parte de la app, esta clase auxiliar se comenta en la subsección 8.4.2 de los JobService.

En tercer y último lugar, se debe declarar en el manifest.xml el Broadcast Receiver de la siguiente manera:

```
<manifest>
    <application>
        <receiver
            android:name=".broadcast.receivers.BroadcastReceiverBoot"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Figura 8.5: Broadcast Receiver de tipo estático ACTION_BOOT_COMPLETE en el manifest.xml del prototipo TFM Server

La declaración del receiver se encuentra dentro del elemento application. El receiver tiene 4 atributos:

- **name:** es la ruta de la paquetería y el nombre de la clase de la figura 8.4.

- **enabled:** si es true, es que está habilitado para su ejecución y false es que no está habilitado para su ejecución.
- **exported:** si está a true, indica que el receiver puede ser invocado por procesos externos a la app, como será invocado desde fuera de la app, debe estar a true.
- Dentro del elemento intent-filter se encuentra el subelemento action que indica el name del tipo de receiver que es, como se ha comentado de tipo BOOT_COMPLETED.

Algunas versiones recientes de Android y en especial las capas que instalan los fabricantes encima de Android, permiten desactivar a voluntad del usuario que apps tienen permiso para iniciarse de forma automática, por lo que es posible que los servicios de los prototipos no se inicien a no ser que se conceda el permiso de inicio automático. Normalmente este ajuste, suele estar en los ajustes de Android de la propia app o bien suele estar por los ajustes de la batería de Android.

8.4.1.2 Broadcast Receiver Dinámico

Este tipo de receiver se registra en el propio código de la app, por ejemplo en una activity de la app y una vez que se cierra la activity se debe quitar el registro del receiver. En los prototipos se utiliza uno de estos tipos de receiver en el MainActivity, actividad principal y de inicio de la app. Concretamente, la funcionalidad de este broadcast es detectar el cambio de estado de la conexión Wi-Fi, si esta se encuentra activa o inactiva. Si el Wi-Fi está activo, se levantan todos los servicios de red de la app, siempre que estos estén habilitados para ello en la configuración de la app y no se encuentren ya iniciados. Si se desactiva el Wi-Fi, todos los servicios de red de la app se desactivan.

Por tanto, el objetivo de este receiver es la de optimización de procesos de los prototipos, si no está activo el Wi-Fi, no tiene sentido que los servicios de red estén levantados y se ahorra batería, CPU y memoria en el dispositivo.

El receiver concreto es android.net.wifi.WIFI_STATE_CHANGED y está dentro de la lista de receivers no permitidos que se pueden declarar en el manifest.xml, a partir de la versión 7.0 de Android. La razón fundamental de no permitir receivers de este tipo, es porque, si tenemos 20 apps con este receiver y se activa el Wi-Fi en el dispositivo, todos los receivers de las apps se ponen en marcha casi al mismo tiempo y se produce una sobrecarga muy importante en el dispositivo y Android ha prohibido este y otros receivers similares.

Así pues, este receiver sólo funcionará cuando la app se encuentre en la MainActivity de los prototipos. Para declarar un receiver de este tipo, se debe de realizar en dos pasos, el primero es declarar el Broadcast Receiver como tal, como la siguiente figura:

```
1 public class BroadcastReceiverWifi extends BroadcastReceiver {
2
3    @Override
4    public void onReceive(Context context, Intent intent) {
5        final String action = intent.getAction();
6
7        LogsUtil.log("BroadcastReceiverWifi.onReceiver(). Action = " + action,
                     LogsUtil.Levels.INFO);

8        if(action.equals(WifiManager.WIFI_STATE_CHANGED_ACTION)){
9            ServicesFacade servicesFacade = ServicesFacade.getInstance();
10            if(servicesFacade != null){
11                int isWifiEnabled = intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE,0);
12                switch(isWifiEnabled){
13                    case WifiManager.WIFI_STATE_ENABLED:
14                        servicesFacade.init();
15                        break;
16                    case WifiManager.WIFI_STATE_DISABLED:
17                        servicesFacade.stopNetworkServices();
18                        break;
19                }
20            }
21        }
22    }
23 }
```

Figura 8.6: Broadcast Receiver de tipo WIFI_STATE_CHANGED_ACTION en el prototipo TFM Server

La declaración no cambia si se trata de un receiver estático o dinámico como en este caso, por líneas:

- En la línea 8 se filtra para comprobar que se trata del evento WIFI_STATE_CHANGED_ACTION.
- Dentro del bloque del condicional, básicamente se comprueba en el switch, si el Wi-Fi está activo, ServicesFacade inicia los servicios, y si el Wi-Fi está inactivo, ServicesFacade para los servicios de red.

La declaración de forma programática de registrar y quitar el registro se presenta en las siguiente figuras:

```
1 public class MainActivity extends AppCompatActivity {
2    private BroadcastReceiversFacade broadcastReceiversFacade;
3
4    @Override
5    protected void onCreate(Bundle savedInstanceState) {
6            //[..] Codigo de la Activity
7             this.broadcastReceiversFacade = new BroadcastReceiversFacade();
8             this.broadcastReceiversFacade.registerWifi(this);
9             //[..] Codigo de la Activity
10        }
11     }
12     @Override
13     protected void onStop() {
14         super.onStop();
15         if (this.broadcastReceiversFacade != null) {
16             this.broadcastReceiversFacade.unregisterWifi(this);
17         }
18     }
19 }
```

Figura 8.7: Invocación desde MainActivity para registrar y quitar el registro del receiver en los prototipos

```
1 public class BroadcastReceiversFacade {
2
3     private BroadcastReceiverWifi broadcastReceiverWifi;
4
5     public void registerWifi(Activity activity){
6         this.broadcastReceiverWifi = new BroadcastReceiverWifi();
7         IntentFilter filter = new IntentFilter();
8         filter.addAction(WifiManager.WIFI_STATE_CHANGED_ACTION);
9         try {
10            activity.registerReceiver(this.broadcastReceiverWifi, filter);
11        } catch (IllegalStateException e) {
12            //[..] Imprime Log
13        } catch (Exception e) {
14            //[..] Imprime Log
15        }
16    }
17
18    public void unregisterWifi(Activity activity){
19        try {
20            activity.unregisterReceiver(this.broadcastReceiverWifi);
21        } catch (IllegalStateException e) {
22            //[..] Imprime Log
23        } catch (Exception e) {
24            //[..] Imprime Log
25        }
26    }
27
28 }
```

Figura 8.8: Código para registrar y quitar el registro del receiver en los prototipos

En la figura 8.7 se debe registrar el receiver en el onCreate de la activity y cuando se para la activity se debe de quitar el registro.

En la figura 8.8, en el método registerWifi, se registra la activity para que escuche el tipo de receiver en el IntentFilter WIFI_STATE_CHANGED_ACTION. En el método unregisterWifi se quita el registro del receiver de la actividad y la app ya no escuchará más eventos de broadcast.

8.4.2. JobService

Un JobService es proceso que una vez construido y lanzado por la app, es ejecutado por el planificador de trabajos de Android [52]. El planificador inicia el trabajo en el hilo principal de la app, por lo que hay que implementar mecanismos para no bloquear la app, como los Thread clásicos de Java o los HandlerThread de Android. En este tipo de trabajos se pueden definir parámetros muy interesantes, para que los procesos de los prototipos continúen trabajando, a pesar de que la app pasa a segundo plano y a pesar de que el dispositivo entra a modo dormido con la pantalla apagada.

La verdad es que esta API está un poco escondida en la guía de Android Developer. En ambos prototipos, los JobService se utilizan de la misma forma, en la siguiente figura se muestra la clase auxiliar que construye y lanza un JobService:

```
1  public class JobsServicesUtil {
2
3     public static final int JOB_ID_SERVICES_FACADE = 1;
4
5     public static JobInfo.Builder buildJob(Context context, int jobId, Class<?> nameJobService){
6         ComponentName serviceComponent = new ComponentName(context, nameJobService);
7         JobInfo.Builder builder = new JobInfo.Builder(jobId, serviceComponent);
8         builder.setMinimumLatency(1000);
9         builder.setOverrideDeadline(5000);
10        builder.setPersisted(true);
11        builder.setRequiresCharging(false);
12        return builder;
13    }
14
15    public static void launchJob(Context context, JobInfo build){
16        JobScheduler jobScheduler = context.getSystemService(JobScheduler.class);
17        jobScheduler.schedule(build);
18    }
19 }
```

Figura 8.9: Clase auxiliar JobServicesUtil de los prototipos para construir y lanzar un JobService

Esta clase auxiliar es utilizada por el Broadcast Receiver de tipo boot de la figura 8.4, en la líneas 10 y 11 y también es utilizada por la clase MainActivity de la misma forma en el método onCreate de la activity. Se comentan las cuestiones más importantes de la figura anterior:

- En la línea 3 hay una constante que identifica de manera única el JobService, como sólo hay un JobService en cada prototipo, sólo hay un identificador. Esta constante se utiliza en el método buildJob, en el parámetro jobId de tipo int.
- En la línea 5, en la signatura del método buildJob, se puede ver también el parámetro genérico nameJobService que en nuestro caso será ServicesFacade que extiende de JobService.
- Dentro del método buildJob se define el comportamiento que tendrá el JobService, una vez ejecutado por el planificador de trabajos de Android. En la línea 8 y 9, se define que el trabajo se retrasará 1000 ms en ejecutarse y como máximo tardará en ejecutarse 5000 ms.
- En la línea 10 se indica que el trabajo se volverá a planificar cuando el sistema Android se reinicie, en realidad se han hechos pruebas y es indiferente si este parámetro está a true o false, el trabajo siempre se replanifica.
- La línea 11 es muy importante e indica que el JobService no requiere que el cargador de batería esté conectado al dispositivo para funcionar.
- Dentro del método launchJob se lanza o mejor dicho se planifica el trabajo con el API de Android con el objeto construido en el método anterior buildJob.

Finalmente, en la siguiente figura se muestra el JobService que utilizan ambos prototipos, como es habitual utilizando el API de Android, esta clase

extiende de JobService y se presenta parcialmente el código en la siguiente figura (en los dos prototipos se implementa de la misma forma):

```
1 public class ServicesFacade extends JobService implements ServicesFacadeInterface {
2
3    private static ServicesFacade servicesFacade;
4    private PreferencesUtil preferencesUtil;
5    public ServicesFacade() {
6        LogsUtil.log("*** ServicesFacade.Constructor().
             No usar este constructor de forma manual en el codigo ***", LogsUtil.Levels.INFO);
7    }
8
9    @Override
10    public boolean onStartJob(JobParameters params) {
11        LogsUtil.initLogsDao(this);
12        LogsUtil.log("*** ServicesFacade.onStartJob() *** ", LogsUtil.Levels.INFO);
13        this.preferencesUtil = new PreferencesUtil(this);
14        servicesFacade = this;
15        this.init();
16        return true;
17    }
18
19    @Override
20    public boolean onStopJob(JobParameters params) {
21        LogsUtil.log("*** ServicesFacade.onStopJob() *** ", LogsUtil.Levels.INFO);
22        return true;
23    }
24 }
```

Figura 8.10: ServicesFacade de tipo JobService utilizado en ambos prototipos

En la línea 1 de la figura anterior, la clase ServicesFacade también implementa una interfaz donde se definen los métodos necesarios de los Service y que se implementan en esta clase. Dichos métodos se comentan en la sección 8.4.3. El resto de líneas se comentan a continuación:

- En la línea 3 hay un atributo privado estático del mismo tipo que la clase que se asigna en la línea 14 o más bien se asigna la referencia del objeto ServicesFacade a este atributo privado. De esta forma se puede conseguir el único objeto del JobService de las apps y utilizar los métodos para iniciar y parar los servicios.
- En la línea 5 hay un constructor por defecto, que es requerido por los JobService, no tiene nada en su interior, sólo se imprime un log.
- En la línea 10, comienza el método onStartJob de los JobService, dentro de este método tenemos un método privado init en la línea 15 que contiene toda la lógica de inicio de los procesos de las apps.
- La línea 16 es muy importante, donde se devuelve true y de esta forma se indica al planificador de trabajos de Android que el JobService continúa ejecutándose según los parámetros que hemos definido en el buildJob de JobServicesUtil. Es decir, que de esta forma mantenemos el wakelock cuando este método termina su ejecución. Un wakelock es un bloqueo que mantiene nuestra app despierta, o al menos mantenemos está parte de los procesos de la app despierta.
- En la línea 20 se encuentra el método onStopJob que pertenece también al JobService. En este método retornamos también true, es muy

importante este retorno, con true le indicamos al planificador que nuestro JobService no ha terminado y que debe de volver a replanificar el JobService de ServicesFacade. Este método no es invocado en ningún caso por los propios prototipos porque lo deseable es que siempre esté en ejecución, pero puede ser invocado por el propio planificador de trabajos de Android cuando lo estime oportuno en base a sus estrategias de planificación, por excepciones o en base a los parámetros de construcción que se han definido en JobServicesUtil. En JobServicesUtil no se ha definido ningún parámetro que indique al planificador de trabajos de Android que deba detener el JobService.

De forma resumida, el quid de la cuestión es devolver true en onStartJob para mantener el wakelock y devolver true en onStopJob, para replanificar el JobService, en caso de parada. Es posible que el JobService se replanifique y que puedan existir varias instancias de JobService corriendo, pero esto no ocurre, debido a que el JobService tiene un identificador único y sólo mantiene un JobService en memoria. Pero lo que si que ocurre es que al pararse el JobService se cree otra instancia, por este motivo todos los objetos definidos como atributos del JobService deben declararse con static para que se mantengan las referencias de los atributos entre los distintos objetos de JobService. Estos atributos estáticos son los Intent que representan los Service del API de Android y que se presentan en la siguiente subsección.

Por último, hay que declarar el JobService en el manifest.xml de la app de la siguiente manera:

```
<manifest>
    <application>
      <service
        android:name=".services.ServicesFacade"
        android:enabled="true"
        android:exported="true"
        android:permission="android.permission.BIND_JOB_SERVICE"/>
    </application>
</manifest>
```

Figura 8.11: Un JobService de Android en el manifest.xml de la app

En la figura anterior, se observa como los atributos son similares a los Broadcast Receiver y de hecho tienen el mismo significado, las diferencias son:

- El elemento es service, en vez de receiver.
- Tiene un permiso en el atributo permission que indica que se trata de un JobService.

8.4.3. Services

Un Service de Android [53] es un proceso que se ejecuta con el fin de realizar tareas de larga duración en segundo plano (background) y se caracterizan porque no tienen interfaz de usuario, de esta forma si la app se

cierra de manera normal, el Service de Android continúa trabajando en segundo plano. Como ya se ha comentado, los procesos de los prototipos son tareas que deben ejecutarse de manera indefinida.

Los Services de Android se ejecutan por defecto en el hilo principal de la app, por lo que hay que implementar mecanismos para no bloquear el hilo principal, lanzando otros hilos como Threads, HandleTreads o AsyncTask. Hay varios tipos de Services en Android:

- **Foreground (primer plano):** son servicios que muestran una notificación en la barra de notificaciones de Android, por lo que son visibles por el usuario.
- **Background (segundo plano):** son servicios que no muestran nada en la interfaz de Android, por tanto, el usuario no percibe que se está ejecutando el servicio.
- **Bind (enlace):** son servicios destinados a la comunicación entre servicios de tipo cliente-servidor. Un servicio de este tipo puede ser declarado dentro de la app (servidor) y otros servicios pueden comunicarse-enlazarse (cliente) con este servicio de tipo servidor. Los servicios clientes pueden estar dentro de la misma app o bien pueden ser servicios clientes de otras apps. Cuando el servicio servidor no tiene enlaces con servicios clientes, el servicio servidor se destruye.

Lo más natural, para los prototipos es utilizar los servicios en background, pero Android impone restricciones muy severas para este tipo de servicios [54] a partir de Android 8.0 (API LEVEL 26) que, de forma resumida, sólo permiten su ejecución cuando la app se encuentra en primer plano de alguna manera. Los servicios de tipo enlace no ha sido particularmente necesarios en los prototipos, por lo que en los prototipos se utilizan los servicios en foreground. A continuación se muestra en el prototipo TFM Server en su clase ServicesFacade como levantar un servicio en foreground (página siguiente):

```
1  public class ServicesFacade extends JobService implements ServicesFacadeInterface {
2
3    private static Intent socketsTCPServerService;
4    public static final int NOTIFICATION_SOCKETS_TCP = 2;
5
6    public void startSocketsTCPServerService() {
7      if(!this.isServiceRunning(SocketsTCPServerService.class)){
8         try {
9             socketsTCPServerService = new Intent(this, SocketsTCPServerService.class);
10            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
11                this.startForegroundService(socketsTCPServerService);
12            }else{
13                this.startService(socketsTCPServerService);
14            }
15        } catch (Exception e) {
16            LogsUtil.log("ServicesFacade.startSocketsTCPServerService() --> ERROR. Motivo: ",
                           LogsUtil.Levels.ERROR, e);
17        }
18     }else{
19        LogsUtil.log("ServicesFacade.startSocketsTCPServerService() -->
                       The service is already running ", LogsUtil.Levels.INFO);
21     }
22   }


23   @Override
24   public void stopSocketsTCPServerService() {
25       if (socketsTCPServerService != null) {
26           try {
27               this.stopService(socketsTCPServerService);
28           } catch (Exception e) {
29               LogsUtil.log("ServicesFacade.stopSocketsTCPServerService() --> ERROR. Motivo: ",
                               LogsUtil.Levels.ERROR, e);
30           }
31       }
32   }

33   private boolean isServiceRunning(Class<?> serviceClass) {
34      boolean isServiceRunning = false;
35      ActivityManager manager = (ActivityManager)
                                       this.getSystemService(Context.ACTIVITY_SERVICE);
36      for (ActivityManager.RunningServiceInfo
                    runningService : manager.getRunningServices(Integer.MAX_VALUE)) {
38         if (serviceClass.getName().equals(runningService.service.getClassName())) {
39              isServiceRunning = true;
40         }
41      }
42      return isServiceRunning;
43   }
44 }
```

Figura 8.12: Como se lanza un Service de Android en el prototipo TFM Server

El código de la figura anterior pertenece al prototipo de TFM Server, concretamente a la forma de levantar y parar el servicio que levanta el servidor de los sockets TCP. Se observan dos atributos, el tipo Intent, es el propio Service de Android que debe ser declarado de tipo Intent, ya se ha comentado que es de tipo static para que las referencias de memoria sobrevivan, en caso de que se reinicie el JobService (ServicesFacade). El otro atributo es la constante que se utiliza para lanzar la notificación en foreground, cuyo valor es 2. También hay 3 métodos que son lanzados por el propio JobService cuando arranca, en el método init del propio JobService (en la figura no se muestra dicho método), los 3 métodos de la figura se comentan a continuación:

- El primero es startSocketsTCPServerService. En la línea 7, se comprueba si el servicio ya se encuentra en ejecución a través del método privado isServiceRunning. Si no está en ejecución, se crea el servicio en la línea 9, como un Intent y después en la línea 10, si la versión de Android es igual o superior a Android 8.0 (la O es de Oreo), se ejecuta la instrucción 11 y si no la instrucción 13. Esto es así, porque los servicios en foreground en Android 8.0 y superiores obliga a hacerlo así. Si el servicio ya se encuentra en ejecución, entonces imprimimos un log en la línea 16. Hay que comprobar si el servicio ya se encuentra en ejecución, porque puede ocurrir, que el servicio ya esté en ejecución y que el usuario inicie la app, en cuyo caso, se levantaría de nuevo el servicio de los sockets TCP, provocando una excepción y sobrescribiendo la dirección de memoria del atributo socketsTCPServerService.
- El segundo método es stopSocketsTCPServerService que para el servicio si la dirección de memoria de socketsTCPServerService no es nula. También existe un stopForegroundService, pero con stopService funciona igualmente con los Android probados.
- Por último tenemos el método privado y auxiliar isServiceRunning que utilizan el resto de servicios de las apps. Básicamente, hace una llamada al sistema para comprobar si el servicio del prototipo se encuentra en ejecución, si lo está devuelve true y si no está en ejecución devuelve false.

Sólo falta la implementación del propio Service de Android, siguiendo con la explicación del ejemplo se detalla el servicio de los sockets TCP en la siguiente figura (página siguiente):

```
1 public class SocketsTCPServerService extends Service {
2
3     private SocketsTCPServerController controller;
4
5     @Override
6     public int onStartCommand(Intent intent, int flags, int startId) {
7         PreferencesUtil preferencesUtil = new PreferencesUtil(this);
8         if (this.controller == null) {
9             try {
10                int port = Integer.valueOf(preferencesUtil.getStringPreference(
                        PreferencesUtil.KEY_PREF_SERVER_SOCKETS_TCP_PORT,
                        ConstantsUtil.SERVER_SOCKETS_TCP_PORT));

11                int timeout = Integer.valueOf(preferencesUtil.getStringPreference(
                        PreferencesUtil.KEY_PREF_SERVER_SOCKETS_TCP_TIMEOUT,
                        ConstantsUtil.SERVER_SOCKETS_TCP_TIMEOUT));

12                this.controller = new SocketsTCPServerController(this, port, timeout);
13                this.controller.start();

14                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
15                    this.startForeground(ServicesFacade.NOTIFICATION_SOCKETS_TCP,
                              NotificationsUtil.showNotificacionWithoutProgressIndeterminate
                              (ServicesFacade.NOTIFICATION_SOCKETS_TCP, ConstantsUtil.APP_NAME,
                              "Service: Sockets TCP Started", this, true,
                               R.drawable.ic_glyph_solid_cloud_upload_alt,
                               NotificationCompat.PRIORITY_LOW, true, false));
16                }else{
17                    NotificationsUtil.showNotificacionWithoutProgressIndeterminate
                              (ServicesFacade.NOTIFICATION_SOCKETS_TCP, ConstantsUtil.APP_NAME,
                              "Service: Sockets TCP Started", this, true,
                               R.drawable.ic_glyph_solid_cloud_upload_alt,
                               NotificationCompat.PRIORITY_LOW, true, true);
18                }
19                LogsUtil.log("SocketsTCPServerService.onStartCommand() --> OK",
                              LogsUtil.Levels.INFO);
20
21            } catch (Exception e) {
22                LogsUtil.log("SocketsTCPServerService.onStartCommand() --> ERROR",
                              LogsUtil.Levels.ERROR, e);
23                preferencesUtil.setBooleanPreference
                              (PreferencesUtil.KEY_PREF_SERVER_SOCKETS_TCP_ACTIVATED, false);
24                SimpleToast.error(this.getApplicationContext(),
                                  this.getApplicationContext().
                                  getString(R.string.err_sockets_tcp_server_onstartcommnad));
25            }
26        }
27        return super.onStartCommand(intent, flags, startId);
28    }
29
30    @Override
31    public void onDestroy() {
32       NotificationsUtil.remoteNotification(this, this, ServicesFacade.NOTIFICATION_SOCKETS_TCP);
33       if (this.controller != null) {
34          boolean isStop = this.controller.quit();
35          LogsUtil.log("SocketsTCPServerService.onDestroy() --> OK isStop: " + isStop,
                          LogsUtil.Levels.INFO);
36       }
37       super.onDestroy();
38    }
39
40    @Override
41    public IBinder onBind(Intent intent) {
42        return null;
43    }
44 }
```

Figura 8.14: Service de Android de los sockets TCP en el prototipo TFM Server

A continuación se comentan las líneas más importantes de la figura anterior del Service:

- En la línea 1, vemos como la clase extiende de Service, como es habitual utilizando las API de Android.
- Los services tienen tres métodos, el primero comienza en la línea 6 onStartCommand, que como su nombre sugiere es el bloque de inicialización del Service:
    - La línea 8, se hace una comprobación adicional, de si el controlador es nulo, no es especialmente necesario, pero así nos aseguramos de no levantar dos veces el mismo servidor o hilo de ejecución.
    - En las líneas 10 y 11 se obtienen el port y el timeout de las propias preferencias de la app para levantar el servidor de sockets TCP.
    - En la línea 12, creamos una instancia del controlador de los sockets TCP.
    - En la línea 13, lanzamos el hilo de ejecución de los sockets TCP.
    - Las siguientes líneas del método onStartCommand, lanzan un notificación en función de la versión de Android. Si la versión de Android es igual o superior a Android Oreo, en la línea 15, se lanza el servicio como un startForegroundService y hay que pasarle la notificación que se mostrará en la barra de notificaciones de Android. Para que el comportamiento sea el mismo en versiones inferiores a Android Oreo, se muestra una notificación permanente en la línea 17.
    - En la línea 27 se invoca al método super de la superclase Service. Este retorno se utiliza para las políticas de volver a recrear el Service, en caso de que se produzca algún evento o parada inesperada. Con super delegamos en el sistema Android que decida que hacer con el Service y no forzamos ninguna política de reinicio.
- El siguiente método es onStopCommand, donde se define el bloque de código de parada del Service:
    - Se elimina la notificación permanente en la línea 32.
    - En la línea 33 se protege el bloque de código de parada con el condicional si el controlador no es nulo.
    - En las siguientes líneas dentro del bloque condicional, se invoca al método quit del controlador para detener el hilo de ejecución del controlador. También existe un quitSafety, pero este provoca un bloqueo y el servicio no termina nunca, porque espera a que finalice el socket TCP, cosa que no ocurre porque está de forma permanente en ejecución. Así pues, con quit forzamos el cierre del socket TCP.
    - En la línea 37, invocamos a super.onDestroy para notificar al sistema de que el servicio ha sido parado o destruido a todos los efectos.

- Por último tenemos el método onBind, para enlazar servicios, en los prototipos no se utiliza y simplemente hay que devolver null en la línea 42.

Para finalizar, tenemos que declarar el Service en el manifest de la app de la siguiente manera:

```
<manifest>
    <application>
      <service
        android:name=".services.servers.SocketsTCPServerService"
        android:enabled="true"
        android:exported="true" />
    </application>
</manifest>
```

Figura 8.15: Declaración de Service de Android en el manifest.xml de los sockets TCP en el prototipo TFM Server

Los atributos son idénticos a los ya explicados en los Broadcast Receiver y los JobService de Android.

8.4.4. HandlerThread

Pasamos a los hilos de ejecución de Android. Los HandlerThread [55] extienden de la clase Thread y son similares a estos, pero en los HandlerThread han puesto una capa por encima para manejar varias instancias de objetos Runnables (hilos de ejecución) que son procesados por un Looper [56]. Un Looper, se ejecuta de manera indefinida hasta que alguien lo detiene y el Thread termina su ejecución en su método run principal. Los HandlerThread son similares a los Timer clásicos de Java. En los prototipos, se utilizan los HandlerThread pero de forma similar a los clásicos Threads de Java, de la siguiente forma (página siguiente):

```
1  public class SocketsTCPServerController extends HandlerThread {
2
3      private SocketsTCPServerService service;
4      private int port;
5      private int timeout;
6      private boolean isServerRunning;
7      private ServerSocket server;
8
9      public SocketsTCPServerController(SocketsTCPServerService service, int port, int timeout) {
10         super("SocketsTCPServerController", HandlerThread.NORM_PRIORITY);
11         this.service = service;
12         this.port = port;
13         this.timeout = timeout;
14         this.isServerRunning = true;
15     }
16
17     @Override
18     public void run() {
19        try {
20          this.server = new ServerSocket(this.port);
21          while (this.isServerRunning) {
22             try {
23             //[..] Codigo principal del hilo
24             } catch (Exception e) {
25             //[..] Tratar la excepcion, el hilo continua ejecutandose
26             }
27      } catch (Exception e) {
28          //[..] Tratar la excepcion, parar la ejecucion del hilo
29     }
30
31     @Override
32     public boolean quit() {
33         this.isServerRunning = false;
34         try {
35             this.server.close();
36         } catch (IOException e) {
37             LogsUtil.log("SocketsTCPServerConfig.quit() --> ERROR: ", LogsUtil.Levels.ERROR, e);
38         }
39         return true;
40     }
41 }
```

Figura 8.16: HandlerThread de Android de los sockets TCP en el prototipo TFM Server

La clase de la figura anterior es similar a extender de Thread, tiene atributos, un constructor para inicializar los atributos que más tarde se usarán cuando se ejecute el hilo en el clásico método run de los Thread y el método quit que sí que es específico de los HandlerThread. Se comentan las líneas más importantes relacionadas con los HandlerThread:

- En la línea 10, en el constructor, debemos invocar a super con el nombre único de este manejador de hilos (el nombre de la clase) y la prioridad del proceso. Esta prioridad del proceso debe ser un valor extraído desde android.os.Processandroid.os.Process y no de java.lang.Thread.
- En la línea 18, comienza el código run, que es cuando el hilo se encuentra en ejecución. En el esqueleto del código run, un bucle while se ejecutará de manera indefinida en base al booleano isServerRunning. Si ocurre un error dentro del bucle while, el try captura la excepción y el hilo no se para y continúa su ejecución, por ejemplo, aquí se pueden producir errores al tratar una petición, como una excepción al

deserializar un documento json a un objeto. Por el contrario, si se produce una excepción en el try principal (línea 19), el hilo detendrá su ejecución porque se produce fuera del bucle. Un clásico error puede ser porque el puerto del socket se encuentre abierto por otro proceso y no tiene sentido mantener el hilo de ejecución en funcionamiento.

- En la línea 32, es el bloque de código quit, específico para detener el hilo de ejecución y que no se encuentra en los Thread clásicos. Básicamente, de forma preventiva se pone a false la condición de continuar ejecutando el bucle while de la línea 21, y a continuación se cierra el socket de conexión TCP. El método quit devuelve true para indicar al Handler que se detenga.

En los prototipos se han utilizado los HandlerThread sin esta capa por encima para el manejo de hilos. Esto ha sido debido a la desconfianza a las colas, los Handler y los Looper de Android y a los optimizadores de batería y se abordó en primer lugar como un Thread clásico. Esto fue inspirado también, porque en primer lugar se utilizó un AsyncTask [57] que es otra forma de lanzar tareas como hilos de ejecución. Los AsyncTask de Android tienen una cola, que al tener más de 5 hilos en ejecución, ya no admite ninguno más en ejecución (recordemos que los hilos de los prototipos se ejecutan indefinidamente), y es una sorpresa que al implementar el sexto servicio, este no se ejecuta hasta que no se para otro. Los AsynTask están pensados para realizar tareas asíncronas que pueden posponerse en el tiempo, si por algún motivo la app lanza muchos hilos de ejecución, mediante la cola de AsyncTask se limitan los hilos de ejecución que están en marcha al mismo tiempo y pueden ser muy útiles para no sobrecargar la CPU y la batería de los dispositivos, pero en el contexto de los prototipos no es tipo de comportamiento deseado, porque podemos tener más de 5 hilos de ejecución al mismo tiempo.

La forma de lanzar un HandlerThread se puede encontrar en la figura 8.13, en las líneas 12 y 13, instanciando el objeto y luego invocando al método start.

8.4.5. SensorEventListener

Esta clase de Android es la responsable de recibir eventos cuando un sensor obtiene un valor nuevo. Se entiende como valor nuevo, a un valor que es distinto de un valor de la lectura anterior del sensor. En el prototipo se leen dos tipos de sensores, el de batería, en el que se lee de una forma sin utilizar esta clase SensorEventListener y el sensor de proximidad que sí que hay que leerlo con SensorEventListener. De alguna forma es similar a un Broadcast Receiver, y cuando la app se registra para leer el sensor de proximidad, se queda en ejecución de forma indefinida esperando eventos de los sensores, hasta que quitamos el registro de continuar escuchando eventos de los sensores, momento en el que se detiene su ejecución. La forma en la que se

ha implementado en el prototipo TFM Server para el sensor de proximidad se muestra en la siguiente figura:

```
1  public class ProximitySensor implements SensorEventListener {
2
3      private final Context context;
4      private SensorManager sensorManager;
5      private Sensor sensorProximity;
6
7      public ProximitySensor(Context context) {
8          this.context = context;
9          this.sensorManager = (SensorManager)
                                 this.context.getSystemService(Context.SENSOR_SERVICE);
10         this.sensorProximity = this.sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
11     }
12
13     public void execute() {
14         this.sensorManager.registerListener(this, this.sensorProximity,
                                               SensorManager.SENSOR_DELAY_NORMAL);
15     }
16
17     public void stop() {
18         this.sensorManager.unregisterListener(this, this.sensorProximity);
19     }
20
21     @Override
22     public void onSensorChanged(SensorEvent event) {
23        if (event.sensor.getType() == Sensor.TYPE_PROXIMITY) {
24          float[] valuesSensor = event.values;
25          float value = valuesSensor[0];
26          //Guarda la lectura en la base de datos del dispositivo
27        }
28
29     }
30
31     @Override
32     public void onAccuracyChanged(Sensor sensor, int accuracy) {
33
34     }
35
36 }
```

Figura 8.17: SensorEventListener para leer el sensor de proximidad en el prototipo TFM Server

La figura anterior es un resumen de la clase original ProximitySensor y se muestra el código relevante relacionado con un SensorEventListener. Por líneas:

- En la línea 1 la clase ProximitySensor implementa a SensorEventListener.
- En la línea 9 se obtiene el gestor de sensores de Android del sistema.
- En la línea 10 se obtiene el tipo de sensor de proximidad.
- El método execute de la línea 13 se utiliza desde la clase Service (SensorProximityService) de la app para registrar esta clase como Listener de sensores con una velocidad de lectura SENSOR_DELAY_NORMAL.
- El método stop de la línea 17, es el contrario, cuando se para el Service, se ejecuta este método para no seguir escuchando el evento de leer el sensor de proximidad.

- Al implementar la clase con la interfaz SensorEventListener, nos obliga a sobrescribir dos métodos:
  - El primero, onSensorChanged, se ejecuta cuando cambia el valor del sensor. Protegemos el método con un condicional en la línea 23 para asegurarnos que el evento es realmente el evento del sensor de proximidad. Dentro del condicional, obtenemos el valor del sensor de proximidad en la línea 25 como un float. La unidad se mide en cms (centímetros)
  - El segundo onAccuracyChanged no se utiliza, pero la interfaz nos obliga a sobrescribirlo, en este caso se puede dejar vacío.

En la clase original de ProximitySensor hay más código, está relacionado con el guardado de la lectura del sensor en una tabla de la base de datos local de la app del prototipo y que no es importante para el ejemplo y los SensorEventListener.

Y por otro lado, también está el código para gestionar dos comportamientos del sensor de proximidad:

- En un emulador de Android, el método onSensorChanged se ejecuta de manera constante, seguramente y por los valores obtenidos del sensor, tiene bastante decimales y un cambio infinitesimal provoca que se ejecute onSensorChanged de manera constante, es posible que se deba a un bug del emulador.
- En cambio, en los dispositivos físicos probados, el comportamiento es que sólo se ejecuta el método onSensorChanged en las siguientes circunstancias:
  - Si tapamos la cámara delantera, se obtiene la lectura 0,0.
  - Si dejamos de tapar la cámara delantera, se obtiene un número mayor que 0,0 que puede ser 1,0 ó 5,0, según el dispositivo.

Este comportamiento es configurable en el prototipo TFM Server, marcando un check de si la app está instalada en un dispositivo emulado (lee de forma constante el sensor de proximidad en el dispositivo) o bien la app está instalada en un dispositivo físico, y sólo devuelve los valores cerca, cuyo valor es 0,0 y lejos, cuyo valor es mayor que 0,0.

8.4.6. Workers

Los Workers [58] es un API reciente de Android para lanzar procesos en segundo plano. Siempre es mejor usar un API reciente porque funciona en versiones recientes de Android y ofrecen, en teoría, retrocompatibilidad con versiones más antiguas de Android. Los Workers, tienen características interesantes, como las siguientes:

- Pueden funcionar cuando la app no está en primer plano.

- Pueden funcionar cuando el dispositivo se encuentra dormido con la pantalla apagada.
- Se pueden programar tareas que se ejecuten de forma periódica.
- Se pueden ejecutar completamente en background, es decir, sin tener que mostrar una notificación en la barra de notificaciones de Android.

Viendo las características anteriores, uno puede pensar que son perfectos para los procesos de los prototipos, pero como en anteriores ocasiones, esta API está pensada para optimizar la batería y el consumo de CPU del dispositivo y no sirve para tareas que se deban ejecutar de forma inmediata, con una frecuencia de tiempo establecida y para procesos infinitos. Por tanto, otras características que no son tan interesantes para los procesos de los prototipos son:

- El tiempo mínimo de una tarea periódica es de 15 minutos, por lo que no sirve para invocar remotamente a un servidor cada segundo para leer un sensor remoto.
- Los Workers son encolados y pueden ser suspendidos por Android. Para los procesos de tipo servidor, como levantar un socket TCP, por si mismo, no tiene mucho sentido usar el término de Worker, que denota un trabajo finito, cuando el trabajo de un servidor es infinito, que es procesar peticiones de los clientes.

En cada prototipo, se utilizan dos Workers de Android con la finalidad de mantenimiento de la app:

- **CheckWifiWorker:** dado que el Broadcast Receiver para comprobar si la conexión Wi-Fi se encuentra activa o inactiva se encuentra prohibida cuando la app está en segundo plano a partir de la versión 7 de Android, se ha implementado esta funcionalidad con este Worker. Recordemos que esta funcionalidad para los servicios de red cuando el Wi-Fi está inactivo y se levantan todos los servicios de red que ya estuvieran levantados.
- **PurgeBDWorker:** los procesos de los prototipos guardan información en algunas tablas de datos de manera periódica para su funcionamiento y con el tiempo estas tablas crecerán de forma inmanejable para un dispositivo, tanto en espacio y tanto en rendimiento, las tablas con muchas filas son siempre más costosas de consultar. La finalidad de este Worker es mantener las tablas de datos dentro de un rango de filas para no sobrecargar la memoria secundaria del dispositivo y para que las consultas sean más ágiles. Para ello, el Worker cuando se despierta, realiza un count de las tablas de datos y si estan sobrepasan un umbral, se eliminan las primeras x filas, siendo x un número entero positivo.

Como tenemos dos Workers, lo más natural es tener dos Worker de tipo periódico. Lo cierto es que después de la implementación y las pruebas, en general, sólo uno de los Workers se suele ejecutar, por tanto la ejecución es muy errática e imprecisa. La solución pasa por implementar un Worker de tipo periódico y que este Worker periódico, lance los dos Workers de tipo de una sola ejecución. La forma de lanzar un Worker periódico se muestra en la siguiente figura que pertenece al JobService de ServicesFacade:

```
1 @Override
2 public void startManagerWorker() {
3    Constraints constraints = new Constraints.Builder()
             .setRequiresCharging(false)
             .build();

4     PeriodicWorkRequest worker =
            new PeriodicWorkRequest.Builder
                             (ManagerWorker.class, 15, TimeUnit.MINUTES, 10, TimeUnit.MINUTES)
                             .setConstraints(constraints)
                             .build();
5    WorkManager.getInstance()
           .enqueueUniquePeriodicWork("ManagerWorker", ExistingPeriodicWorkPolicy.KEEP , worker);
6 }
```

Figura 8.18: Como se lanza un Worker periódico de Android en ambos prototipos

Se comentan por líneas la figura anterior:

- En la línea 3 se definen la restricción que el dispositivo no requiere estar cargando para que el Worker se ejecute.
- En la línea 4 se construye el objeto Worker de tipo periódico. Se asignan las restricciones y el tiempo establecido de ejecución es de 15 minutos y después se le indica al planificador de Workers que durante 10 minutos puede lanzar el Worker.
- En la línea 5 se encola el Worker, con el nombre único ManagerWorker y con la política KEEP de no cancelar el trabajo y relanzarlo si este ya se encuentra en ejecución.

El objeto worker es de tipo PeriodicWorkRequest y el código se presenta en la siguiente figura:

```
1  public class ManagerWorker extends Worker {
2
3
4
5      public ManagerWorker(@NonNull Context context, @NonNull WorkerParameters workerParams) {
6          super(context, workerParams);
7      }
8
9      @NonNull
10     @Override
11     public Result doWork() {
12         LogsUtil.log("ManagerWorker.doWork(). Starting ID: " + this.getId(),
                        LogsUtil.Levels.INFO);
13         Result result = Result.failure();
14         try{
15             Constraints constraints = new Constraints.Builder()
                     .setRequiresCharging(false)
                     .build();

16             OneTimeWorkRequest worker1 = new OneTimeWorkRequest.Builder
                         (CheckWifiWorker.class).
                          setConstraints(constraints)
                         .setInitialDelay(1, TimeUnit.MINUTES).build();

17             OneTimeWorkRequest worker2 = new OneTimeWorkRequest.Builder
                         (PurgeBDWorker.class)
                         .setConstraints(constraints)
                         .setInitialDelay(5, TimeUnit.MINUTES).build();

18             WorkManager.getInstance().beginWith(worker1).then(worker2).enqueue();
19             result = Result.success();
20         }catch(Exception e){
21             LogsUtil.log("ManagerWorker.doWork(). ERROR: ", LogsUtil.Levels.ERROR, e);
22         }
23
25         LogsUtil.log("ManagerWorker.doWork(). END RESULT " +
                       result.toString() + " ID " + this.getId(), LogsUtil.Levels.INFO);
26         return result;
27     }
28 }
```

Figura 8.19: ManagerWorker (Worker periódico) de Android en ambos prototipos

A continuación se comenta por líneas la figura anterior:

- En la línea 1 se observa que la clase ManagerWorker extiende de Worker, como es habitual utilizando la API de Android.
- En la línea 5 está el constructor requerido por los Workers.
- En la línea 11 comienza el trabajo del Worker en el método doWork.
- En la línea 15 se define la restricción de no requerir que el dispositivo esté cargando para los Workers de tipo de una sola ejecución.
- En la línea 16 se define el Worker de CheckWifiWorker con las restricciones y se establece un retraso en su ejecución de 1 minuto.
- En la línea 17 se define el otro Worker de PurgeBDWorker con las restricciones y que se retrase su ejecución 5 minutos.

- La línea 18 es muy interesante, se indica al planificador de Workers que encole los trabajos, primero que empiece con el worker1 (CheckWifiWorker) y cuando este termine que continúe con el worker2 (PurgeDBWorker). De esta forma con los retrasos en la ejecución y el orden se gana cierto control en la ejecución de los trabajos.
- En la línea 26 se devuelve el resultado de la ejecución del Worker periódico en función de como ha ido la ejecución.

Es importante resaltar que los Workers que se lanzan desde este Worker periódico son del tipo OneTimeWorkRequest, es decir de un solo uso y no son periódicos. Pero como son replanificados por un PeriodicWorkRequest se consigue que se ejecuten de manera indefinida.

Respecto a la implementación de estos OneTimeWorkRequest, son idénticos a un Worker de tipo periódico, lo único que cambia naturalmente es que el código específico que se realiza en el método doWork.

8.4.7. Optimizador de Batería y la Lista Blanca de Android

Android a partir de la versión 6.0 introduce una serie de mejoras que ayudan a alargar la duración de la batería de los dispositivos. La forma de conseguir alargar la duración de la batería es imponer restricciones a como las apps pueden hacer uso de la red y de la CPU cuando el dispositivo entra a modo descanso [59]. En la siguiente figura, extraída de [59] se puede ver el modelo que han seguido:

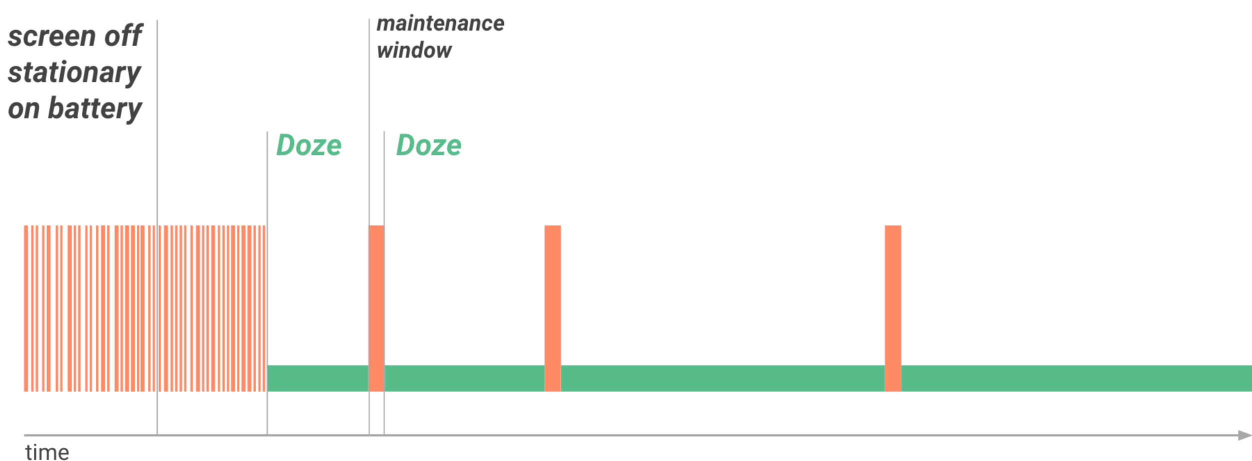


Figura 8.20: Ventana de mantenimiento para que las apps puedan hacer uso de la CPU y de la red

Cuando el dispositivo entra a modo descanso, durante el periodo doze (dormitar), ninguna app puede hacer uso de la CPU, pero especialmente no puede hacer uso de la red. Después se abre una ventana de tiempo corta para que las apps realicen sus actividades. A medida que va pasando el tiempo,

mientras el dispositivo continúa en modo descanso, los periodos de doze se van alargando con el tiempo y por tanto las ventanas en el que las apps pueden continuar con sus actividades se dilata con el tiempo.

Este comportamiento no es el deseable en los prototipos, en los que se espera que los procesos estén funcionando cuando el dispositivo entra a modo dormido y especialmente porque casi todos los procesos son de red en los prototipos, a excepción de los Workers y los procesos que leen los sensores internos al dispositivo.

Por fortuna, existe una lista blanca de Android para que las apps puedan funcionar cuando el dispositivo está en modo descanso sin restricciones. Se puede solicitar este permiso para incluir una app en esta lista blanca pero provoca un baneo en la tienda de la PlayStore y aconsejan abrir el Intent que lleva a la configuración, a modo de acceso directo. En los prototipos se ha implementado con esta premisa de acceso directo con el siguiente código y que no requiere de ningún permiso adicional:

```
1  public void openSettingsBatteryOptimizations() {
2    Intent intent = new Intent();
3    intent.setAction(Settings.ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS);
4    this.activity.startActivity(intent);
5 }
```

Figura 8.21: Intent implícito para abrir el optimizador de batería de las apps de Android

Este Intent se ejecuta desde los dos prototipos desde Menu/Servicios y haciendo tic en el botón Configurar, a continuación se muestra una captura de pantalla del emulador, una vez añadida la app a la lista blanca de Android:

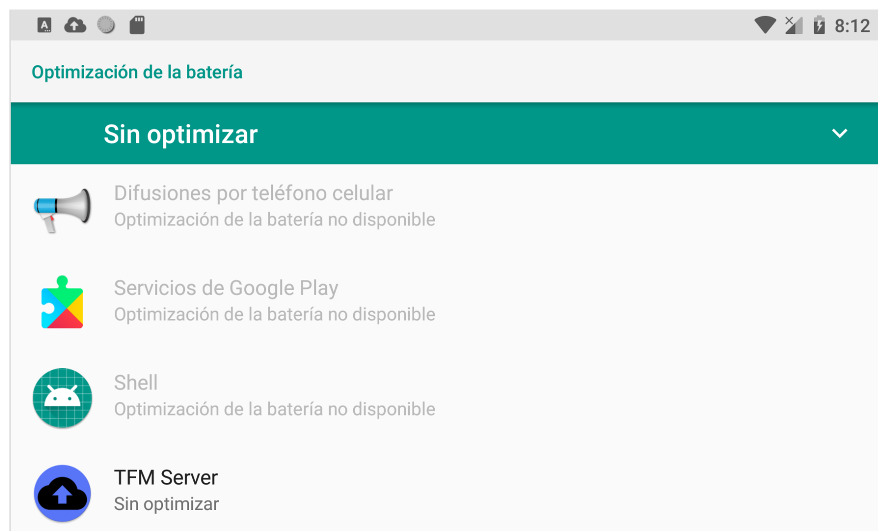


Figura 8.22: Captura de pantalla del emulador (API LEVEL 27), cuando se añade el prototipo TFM Server a la lista blanca

Algunos fabricantes de smartphones y tablets instalan una capa por encima de Android y tienen sus propios optimizadores de batería, por lo que es posible que se deban de realizar otras acciones adicionales en esa capa de

abstracción para desactivar el optimizador de batería en los prototipos. En modelos con la capa de MIUI de Xiaomi, este ajuste se encuentra en ajustes de las apps, hay que buscar el prototipo en el listado de las apps instaladas y a continuación ir al apartado de Ahorro de batería del prototipo y seleccionar sin restricciones.

8.5. API DE PERSISTENCIA ROOM EN LOS PROTOTIPOS

El último componente importante que queda por comentar es la persistencia de los prototipos y que es utilizada de forma intensa por todos los procesos de las apps. En los prototipos se utilizan dos formas de almacenar datos, por un lado están las preferencias de la app que se almacenan en la caché de la app en formato de documento xml y se utiliza el API de Preferences de Android para el acceso y la modificación de este documento xml. Y por otro lado, está el almacenaje de datos en una base de datos local a los prototipos. En este capítulo se va hablar sobre este último método de persistencia en Android mediante bases de datos.

En Android se utiliza SQLite como base de datos para almacenar datos. Por defecto, el API de SQLite es de muy bajo nivel, hay que escribir bastante código fuente y es bastante complejo manejarse con SQLite. Por este motivo, han ido surgiendo diversas API de persistencia que facilitan y simplifican el manejo de bases de datos SQLite desde Android. Un API de este estilo y que está soportada oficialmente por Google es Room [60]. Para usar Room se tiene que añadir la dependencia al fichero de gradle de la app. En los prototipos todo el código relacionado con la base de datos se encuentra en el paquete database, de forma resumida se comenta el contenido de este paquete:

- **AppDatabase:** esta clase es muy importante, contiene métodos para obtener los objetos dao y es donde se añaden que clases de entidad (tablas) va a tener la base de datos.
- **ConvertersDatabase:** esta clase tiene un conversor de Time a Date y de Date a Time. En SQLite, las fechas y las horas se guardan en Time que es tipo Long. Este conversor facilita el uso de las fechas y las horas desde el código fuente en Java.
- **Entities:** este paquete contiene las entidades. Una entidad es una clase que representa una tabla en la base de datos SQLite.
- **Dao:** este paquete contiene dos tipos de clases por cada clase de entidad. En Room se utiliza una interfaz que contiene las operaciones CRUD de una entidad y después se utiliza una clase que implementa a esta interfaz. El nombre de las clases que implementan las interfaces terminan con el sufijo Impl.

En el paquete dao, existe la clase AppDatabaseVersion que contiene en su constructor el código de inicio de algunas entidades (tablas) para precargar algunos datos que luego son utilizados por los procesos de los prototipos.

Room genera el código de bajo nivel de SQLite, para consultar este código generado por Room se puede ir a la carpeta generated del proyecto y consultar alguna clase dao del paquete database.

Ni Room ni Google facilitan ninguna herramienta para consultar de forma sencilla la base de datos generada por Room ni los datos que contiene. En SQLite se almacenan las tablas en un fichero que está dentro de la caché de la app y tampoco resulta muy accesible, concretamente se encuentra en esta ruta: /data/user/0/es.uned.tfm.server/databases/appDatabase en el prototipo TFM Server, en el prototipo TFM Client, hay que cambiar la ruta del paquete de del texto server por el texto client. Sin embargo, existe la librería debug-db que, de forma automática, levanta un servidor web y que genera una web con la información de todas las bases de datos de la app y todas las tablas de cada base de datos. Con debug-db también se pueden consultar las preferencias guardadas con el API de Preferences de Android. Este tipo de generadores como debug-db resultan muy útiles porque facilitan el desarrollo de las apps y también pueden resultar útiles más adelante para depurar las apps cuando el dispositivo físico no se encuentra conectado al ordenador, debido a que podemos consultar la web generada por debug-db para investigar que está fallando. Para acceder a la web generada por debug-db hay que escribir en un navegador web:

```
http://host:8080
```

Figura 8.23: Acceso a la web generada por debug-db

Donde host es la dirección IP del prototipo. Si se utiliza el emulador proporcionado por Android Studio, se puede visualizar desde un navegador web del ordenador cambiando host por la dirección de loopback (localhost) y haciendo forward para redireccionar el puerto escribiendo el siguiente comando:

```
$INSTALACION_SDK$/sdk/platform-tools/adb forward tcp:8080 tcp:8080
```

Figura 8.24: Forward de puertos al emulador de Android Studio

La siguiente captura de pantalla muestra la web generada por debug-db en un dispositivo físico del prototipo TFM Server:

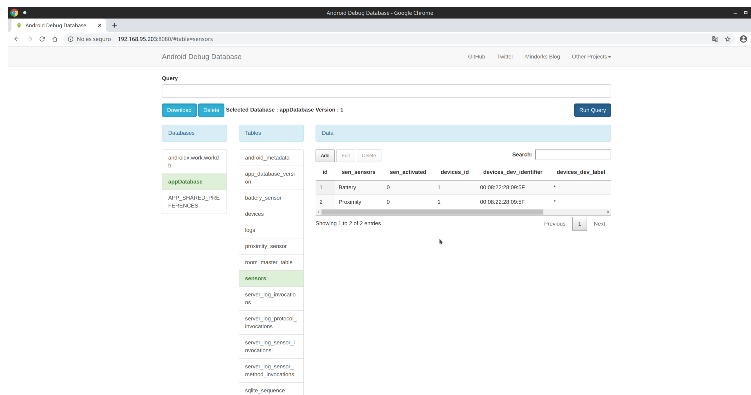


Figura 8.25: Web generada por debug-db del prototipo TFM Server

Consultando la web se descubre que el API de los Workers de Android tienen su propia base de datos SQLite para planificar trabajos. Para terminar esta sección se presenta el modelo entidad-relación de ambos prototipos:

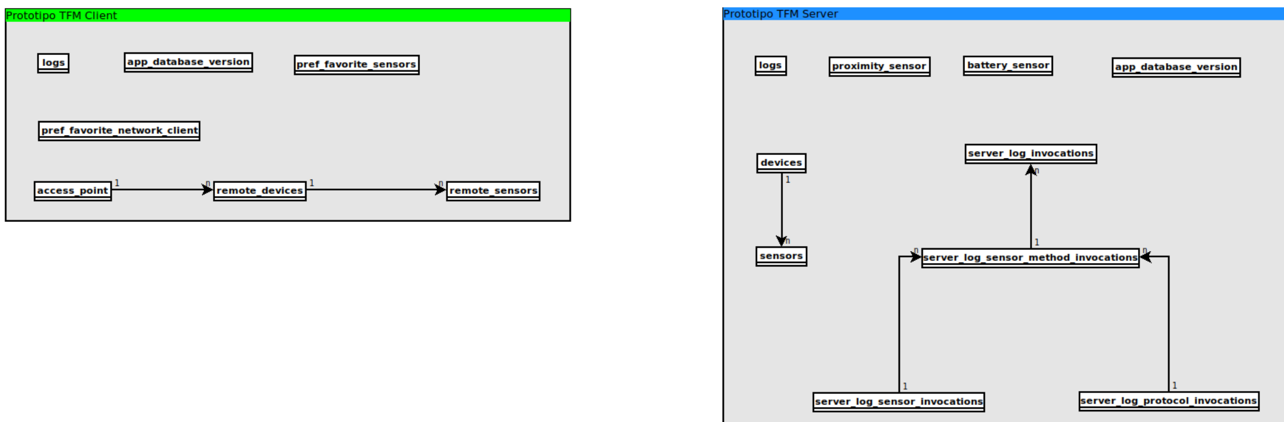


Figura 8.26: Modelos Entidad-Relación de los prototipos

Tablas de TFM Client:

- **logs:** esta tabla contiene los logs que se generan desde el código fuente con la clase LogsUtil. Es muy útil para depurar de forma remota un dispositivo.
- **app_database_version:** contiene una fila con la versión de la base de datos.
- **pref_favorite_sensors:** contiene la configuración de los sensores remotos.
- **pref_favorite_network_client:** contiene todos los clientes, sólo uno de ellos puede ser favorito. Se entiende favorito como el cliente de red predeterminado para realizar las invocaciones remotas.

- **access_point:** contiene todos los puntos de acceso que ha registrado el dispositivo.
- **remote_devices:** son todos los dispositivos encontrados en un punto de acceso.
- **remote_sensors:** son todos los sensores registrados de un dispositivo remoto.

Tablas TFM Server:

- **logs:** esta contiene los logs que se generan desde el código fuente con la clase LogsUtil. Es muy útil para depurar de forma remota un dispositivo.
- **app_database_version:** contiene una fila con la versión de la base de datos.
- **proximity_sensor:** se almacenan las lecturas del sensor de proximidad.
- **battery_sensor:** se almacenan las lecturas del sensor de batería.
- **devices:** dato precargado, sólo hay una fila con la información de red de la interfaz wlan0 y la etiqueta de ubicación.
- **sensors:** un dispositivo tiene sensores. Contiene datos precargados con información de los sensores.
- El resto de las tablas que empiezan por server_log, se utilizan para guardar información estadística de las peticiones de red que le llegan al prototipo. Esta información es útil para llevar un control del estado de un sensor remoto, cuantas peticiones recibe, cuantos datos recibe y si el sensor está respondiendo en un tiempo razonable a las peticiones.

Hay más tablas en ambos prototipos pero son de uso por el propio motor de SQLite para realizar sus procesos internos.

8.6. CONFIGURAR LA UBICACIÓN EN LOS PROTOTIPOS

Otro punto importante a comentar en los prototipos es que en un punto de acceso se pueden tener varios sensores de proximidad y se pueden tener varios actuadores que consuman estos sensores. Por ejemplo, en el contexto de los prototipos, existe un sensor de proximidad y varios actuadores que consumen de forma remota la lectura del sensor de proximidad. Si desplegamos otro sensor de proximidad, los actuadores encontrarán a este sensor de proximidad remoto y harán las peticiones a los dos sensores remotos de proximidad, y puede que no sea el escenario deseable.

En estos casos tanto el prototipo TFM Server y TFM Client, tienen una etiqueta de ubicación. Por defecto está establecido con el carácter comodín * que indica que los dispositivos clientes (actuadores) harán las peticiones, sólo aquellos dispositivos servidores (sensores) que tengan la etiqueta de ubicación

del carácter *. Un sensor de proximidad se puede configurar con la etiqueta comedor y otro sensor de proximidad se puede configurar con la etiqueta cocina. De esta forma, los actuadores que despleguemos en la cocina habrá que configurarlos con la etiqueta cocina y con la palabra comedor para los actuadores desplegados en el comedor.

La ubicación se cambia con el API de Preferences de Android. Se puede cambiar en ambos prototipos en: Menú/Configuración/Ubicación Dispositivo.

CAPÍTULO 9

9. PRUEBAS EMPÍRICAS Y COMPARATIVA ENTRE LOS PROTOCOLOS DE COMUNICACIONES EN LOS PROTOTIPOS

Los prototipos están preparados para realizar unas pruebas empíricas con el fin de poder comparar los protocolos de comunicaciones utilizados. Por tanto, no se tratan de pruebas funcionales para probar la funcionalidad de los prototipos. En los prototipos hay en total 5 protocolos de comunicaciones, los 3 primeros son clásicos y los dos últimos son protocolos IoT:

- **Protocolo http.** Son llamadas http corrientes.
- **Sockets TCP.** Son los clásicos sockets TCP de Java.
- **Sockets UDP.** Son los clásicos sockets UDP de Java.
- **CoAP.** Es un protocolo IoT que utiliza UDP.
- **ZeroMQ TCP.** Otro protocolo IoT. La versión para Java sólo está disponible para el protocolo TCP y sólo se ha podido probar TCP.

Para realizar las pruebas, lo más sencillo es utilizar dos dispositivos físicos, un dispositivo físico con el prototipo TFM Server instalado y el otro dispositivo físico con el prototipo TFM Client instalado y conectarlos al mismo punto de acceso Wi-Fi, esperar a que se encuentren con algún mecanismo de descubrimiento y lanzar las pruebas. Con emuladores es bastante más complejo y no ha sido posible hacerlo porque la redirección del protocolo UDP no está soportada con los emuladores y dos de los protocolos utilizan UDP como protocolo de transporte.

Otra variante de prueba es utilizar un emulador con el prototipo TFM Client instalado y trucar las tablas access_point, remote_devices y remote_sensors de la base de datos de la app con la librería debug-db para que apunte a la dirección IP de un dispositivo físico con el prototipo TFM Server instalado.

En las pruebas empíricas hay dos indicadores que se toman como medición:

- En el prototipo TFM Server se hace a nivel de petición/respuesta:
  - El tiempo en milisegundos en procesar cada petición y proporcionar una respuesta.
  - El tamaño real de un mensaje entrante y el tamaño real del mensaje saliente. La unidad de medida utilizada es el byte.
- En el prototipo TFM Client, se mide el proceso a nivel global, es decir de todas las peticiones/respuestas por protocolo:
  - Se mide el tiempo en milisegundos del procesamiento de todas las peticiones por protocolo.

- Se mide el tamaño real de todos los mensajes, entrantes y salientes.

Para los tiempos, una típica forma de medir es con marcas de tiempo, se toma una marca de tiempo al inicio y cuando se termina se toma otra marca de tiempo y luego se resta el tiempo de fin menos el tiempo de inicio. En algoritmos que sólo consumen CPU resulta ser muy fiable, sin embargo, cuando estamos usando alguna API, la siguiente línea no tiene porque ejecutarse de forma inmediata, por ejemplo, ZeroMQ, encola la respuesta y todavía no ha enviado el mensaje.

Cuando se habla del tamaño real es la carga útil del mensaje, el documento json que se recibe o se devuelve, más la carga que porta el protocolo de comunicaciones, como pueden ser cabeceras y metadatos que utiliza el protocolo. Para el tamaño se ha utilizado la clase Android TrafficStats que permite extraer estadísticas de red por app. Existe otra clase llamada NetworkStatsManager pero es más compleja de utilizar y el resultado ha sido similar después de las pruebas.

Existe en el prototipo TFM Server una clase auxiliar DataTrafficStatsUtil en el paquete util que se encarga de medir y guardar en la tabla server_log_invocations las estadísticas del tiempo y los tamaños de los mensajes que más adelante se puede visualizar mediante unos gráficas de líneas en el prototipo TFM Server.

Una vez que está todo listo, esto es, tenemos los dos prototipos en línea y el prototipo TFM Server tiene todos los servidores levantados, hay que ir a la pantalla del prototipo TFM Client de las pruebas ubicada en el Menú/Test Servidores, tal y como se muestra en la siguiente figura:

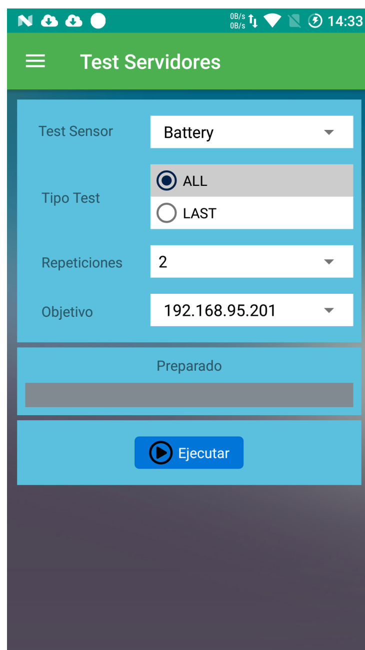


Figura 9.1: Captura de la configuración de los Test Servidores en el prototipo TFM Client

En esta pantalla se puede configurar hasta cierto punto los test, de arriba a abajo:

- **Test Sensor:** se puede elegir entre el test del sensor de Battery o de Proximity.
- **Tipo Test:** si es ALL, se devuelve un documento json con todas las filas de la tabla battery_sensor o proximity_sensor. Si es LAST, se devuelve la última fila guardada de battery_sensor o proximity_sensor.
- **Repeticiones:** indica el número de peticiones por protocolo que se hará en las pruebas, el máximo son 100. Como son 5 protocolos, el máximo son 500 peticiones.
- **Objetivo:** este campo indica la dirección IP del dispositivo remoto donde está instalado el prototipo TFM Server. Este campo es dinámico y se extrae de los dispositivos que se han descubierto.

Una vez que se configura la pantalla el dispositivo se puede hacer tic en el botón de Ejecutar y la app comienza el test. Cuando termina, los datos se pueden consultar en el prototipo TFM Server, navegando desde el Menú/Servidores y escogiendo el sensor y el tipo test, si ha sido ALL (Todas las Lecturas) o LAST (Última Lectura).

Las pruebas se han realizado todas con dos dispositivos físicos, con una tablet en el dispositivo TFM Server para visualizar mejor las gráficas y de esta forma realizar capturas más grandes y se ha utilizado un smartphone en el prototipo TFM Client.

Al tratarse de pruebas empíricas otras ejecuciones pueden proporcionar resultados diferentes, especialmente para la comparativa de los tiempos de ejecución.

9.1. COMPARATIVA DE TIEMPOS

Las pruebas para los tiempos se han configurado en la pantalla de los test de TFM Client de la siguiente manera (todos los resultados de tiempos están en milisegundos):

- **Test Sensor:** Battery
- **Tipo Test:** ALL. Son 30 filas de la tabla battery_sensor, con un tamaño del mensaje de payload en cada una de las respuestas de 7983 bytes. Como son 50 repeticiones por protocolo, en total se retransmiten 399150 bytes por protocolo.
- **Repeticiones:** 50, con más repeticiones la gráfica en la tablet empieza a visualizarse con demasiadas líneas y es más complejo entender el gráfico. Como son 50 y 5 protolocos, en total son 250 peticiones/respuestas.

- **Objetivo:** 192.168.95.202. La dirección IP donde se encuentra instalado el prototipo TFM Server.

Para los tiempos del proceso global tomado desde TFM Client, hay que tener en cuenta, para cada uno de los protocolos, después de una petición/respuesta simple, hay un tiempo de espera de 500 milisegundos con el fin de que respiren las máquinas y darle tiempo a TrafficStats para que registre el tamaño de las peticiones y el tamaño de las respuestas.

El resultado global desde el punto de vista de TFM Client se muestra en la siguiente captura de pantalla:

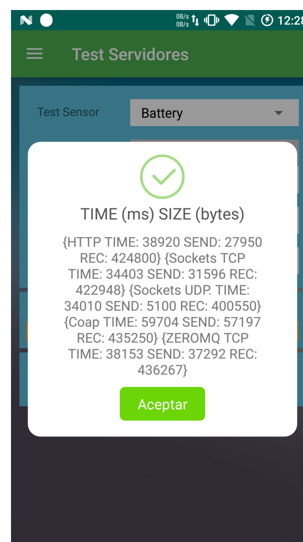


Figura 9.2: Captura del resultado del test de batería ALL desde el punto de vista TFM Client (Tiempos)

En la siguiente tabla se muestra el resumen con algunos cálculos adicionales (todos los tiempos están en milisegundos):

Protocolo	Tiempo Con Retardo	Tiempo Sin Retardo *	Petición/Respuesta Media **
HTTP	38920	13920	278,4
Sockets TCP	34403	9403	188,06
Sockets UDP	34010	9010	180,2
CoAP	59704	34704	688,14
ZeroMQ TCP	38153	13153	263,06

Tabla 9.1: Cuadro resumen del resultado del test de batería ALL desde el punto de vista TFM Client (Tiempos)

** Como son 50 peticiones/respuestas y son 500 ms de retardo en cada petición/respuesta, resulta que son 25000 ms, por lo que hay que restar el tiempo 25000 al tiempo con retardo*

*** Se divide el tiempo sin retardo por las 50 peticiones/respuestas*

Viendo la tabla anterior, se observa como el protocolo de sockets UDP tiene menor tiempo de respuesta media, mientras que en CoAP el tiempo de respuesta es bastante elevado.

La siguiente captura muestra la gráfica de los tiempos, desde el punto de TFM Server:

Figura 9.3: Captura del resultado del test de batería ALL desde el punto de vista TFM Server (Tiempos)

Hay que destacar que este tiempo de la figura anterior es el que se tarda el servidor en procesar la petición y proporcionar una respuesta. Según el protocolo, que estemos usando, estas marcas de tiempo de inicio y fin, se podrán poner antes o después. Si estamos usando un API de bajo nivel como los sockets TCP y UDP, tenemos más libertad para colocar las marcas de inicio y fin, mientras que si usamos una librería externa, estamos más limitados para poner estas marcas de tiempo y por tanto las marcas de tiempo son ligeramente mejores usando API de terceros.

Un claro beneficiario de esta ventaja es el protocolo ZeroMQ porque usa cola para el retorno que lanza un hilo de ejecución para la respuesta, es decir, se toma la marca de fin de tiempo, pero en realidad la respuesta aún no se ha enviado al cliente.

AsyncHttpServer también se beneficia de esta ventaja, aunque tiene un callback para la respuesta, le falta un interceptor para capturar el comienzo verdadero de la petición, cosa que hará que suban un poco los ms.

Con los sockets UDP se consiguen buenos tiempos y están por debajo de los sockets TCP, lo que resulta normal, por el tiempo de negociación en el protocolo TCP.

CoAP, en el lado del servidor, procesa la petición entre los sockets UDP y los sockets TCP de media, pero a veces consigue tiempos muy bajos, menores que los sockets UDP.

9.2. COMPARATIVA DE TAMAÑO DE MENSAJES CON TRAFFICSTATS DE ANDROID

Las pruebas para del tamaño de los mensajes se han configurado en la pantalla de los test de TFM Client de la siguiente manera:

- **Test Sensor:** Battery
- **Tipo Test:** ALL. Son 30 filas de la tabla battery_sensor, con un tamaño del mensaje de payload en cada una de las respuestas de 7983 bytes. Como son 10 repeticiones por protocolo, en total se retransmiten 79830 bytes por protocolo.
- **Repeticiones:** 10, con más repeticiones la gráfica en la tablet empieza a visualizarse con demasiadas líneas y es más complejo entender el gráfico. Como son 10 y 5 protolocos, en total son 50 peticiones/respuestas.
- **Objetivo:** 192.168.95.202. La dirección IP donde se encuentra instalado el prototipo TFM Server.

A continuación se muestra la captura de la pantalla desde el punto de vista de TFM Client:

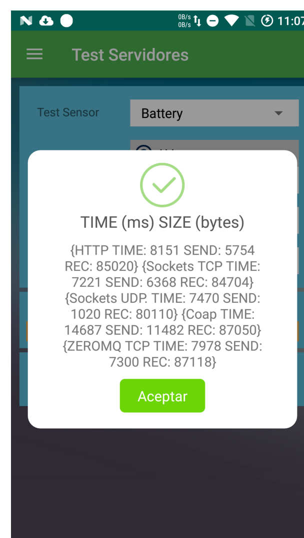


Figura 9.4: Captura del resultado del test de batería ALL desde el punto de vista TFM Client (Tamaño Mensajes)

La unidad de medida de todas las tablas es el byte. En la siguiente tabla se calcula el promedio de los bytes enviados y recibidos en total, la carga útil y la carga del protocolo:

Protocolo	Enviado Total	Recibido Total	Enviado Medio *	Recibido Medio **
HTTP	5754	85020	575,4	8502
Sockets TCP	6368	84704	636,8	8470,4
Sockets UDP	1020	80110	102	8011
CoAP	11482	87050	1148,2	8705
ZeroMQ TCP	7300	87118	730	8711,8

Tabla 9.2: Cuadro resumen del resultado del test de batería ALL desde el punto de vista TFM Client (Tamaño mensajes real con TrafficStats)

** Como son 10 peticiones se divide entre 10 para calcular el promedio*

*** Como son 10 peticiones se divide entre 10 para calcular el promedio*

Con los cálculos de la tabla anterior se calcula la carga que porta el protocolo, es decir sin la carga útil:

Protocolo	Enviado Medio Sólo Carga Protocolo *	Recibido Medio Sólo Carga Protocolo **
HTTP	575,4	519
Sockets TCP	541,8	487,4
Sockets UDP	28	28
CoAP	1148,2	722
ZeroMQ TCP	688	728,8

Tabla 9.3: Cuadro resumen del resultado del test de batería ALL desde el punto de vista TFM Client (Tamaño mensajes sólo carga protocolo con TrafficStats)

** http y CoAP no tienen carga útil en el envío. Los otros protocolos: (1) sockets TCP la carga útil son 95 bytes, (2), sockets UDP la carga útil son 74 bytes y en (3) ZeroMQ la carga útil son 42 bytes, luego, en estos protocolos hay que restar estos bytes para obtener la carga del protocolo*

*** La carga útil de respuesta son 7983 bytes, luego se resta 7983 al recibido medio de todos los protocolos*

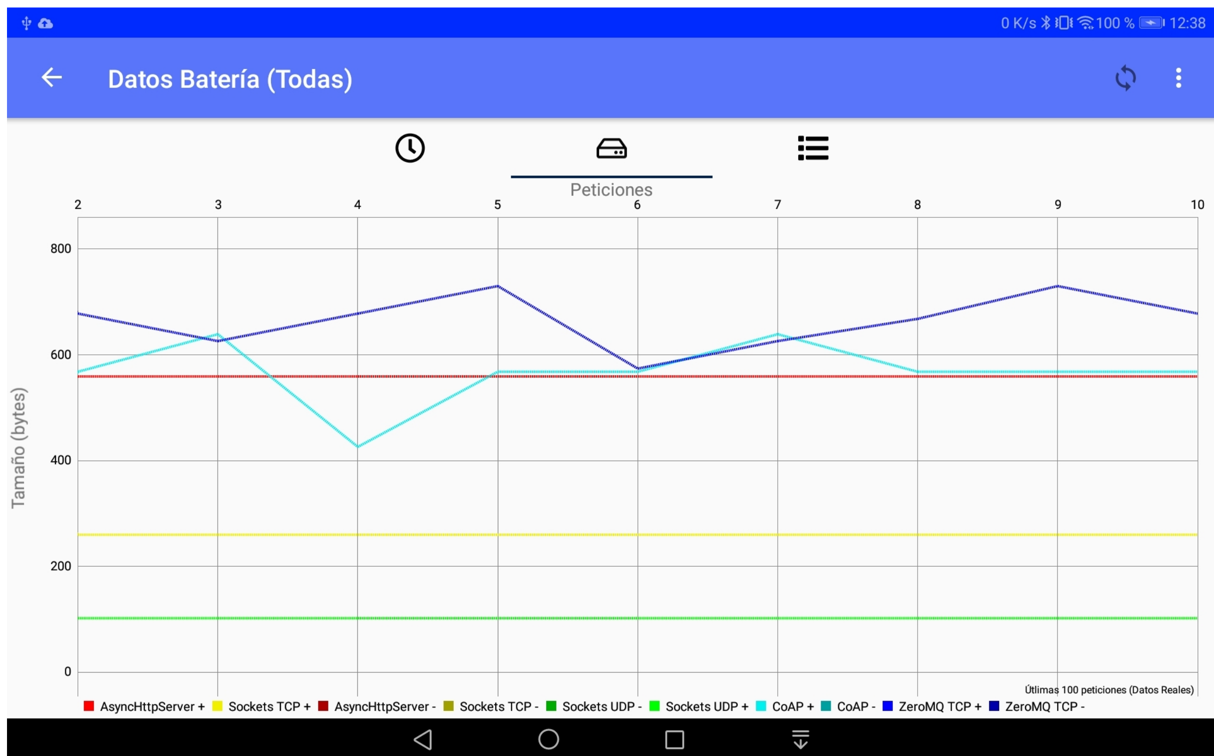


Figura 9.5: Captura del resultado del test de batería ALL desde el punto de vista TFM Server (Tamaño Mensajes Recibidos)

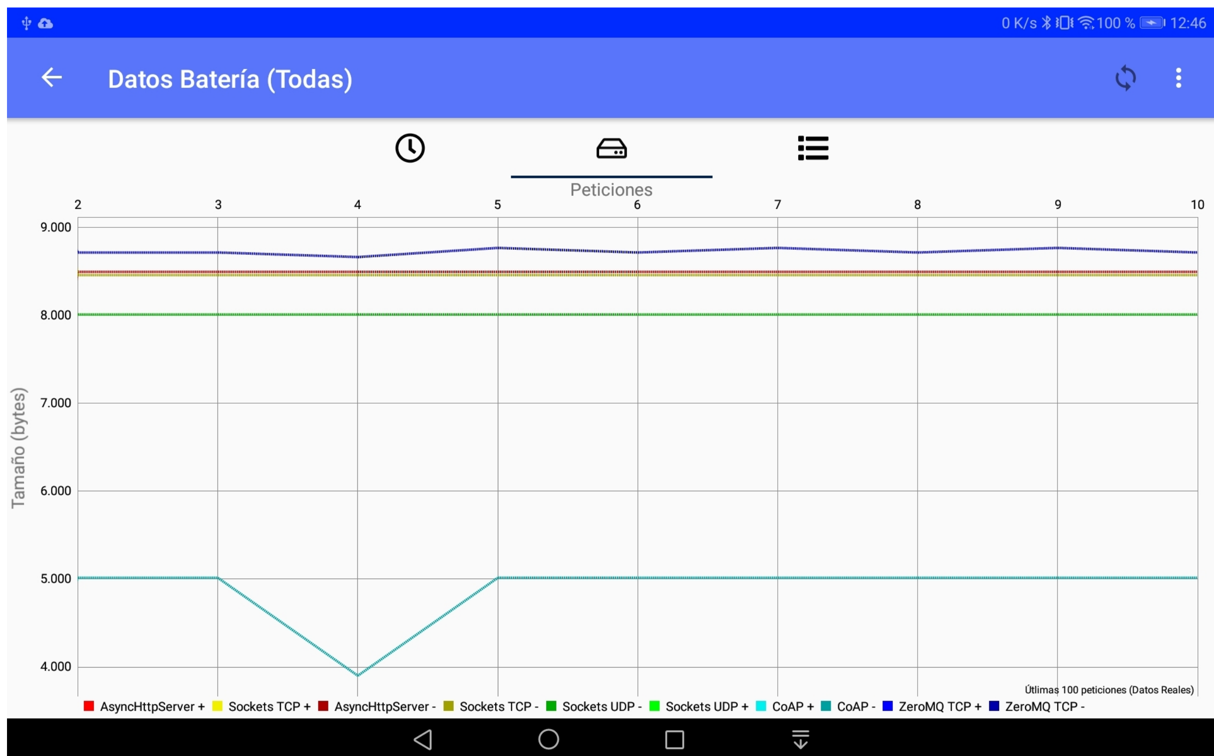


Figura 9.6: Captura del resultado del test de batería ALL desde el punto de vista TFM Server (Tamaño Mensajes Respondidos)

La clase de Android TrafficStats toma los tamaños de los mensajes de recepción y respuesta, sin embargo se han observado algunas incoherencias entre el tamaño informado por TrafficStats entre ambos prototipos y un tamaño aceptable y creíble de las peticiones.

Las incoherencias observadas desde el punto de vista de TFM Server son:

- En ZeroMQ TCP, dado que tiene la cola de retorno para devolver el mensaje, cuando se toma la marca de tiempo, aún no se ha enviado el mensaje de retorno y se ha puesto un tiempo de espera de 10 milisegundos para dar tiempo a ZeroMQ a enviar el mensaje de retorno y después tomar el tamaño de los mensajes con TrafficStats.
- En CoAP, el mensaje de respuesta no se encola y se responde de forma síncrona al cliente, sin embargo, el tamaño de la respuesta es muy bajo, de unos pocos cientos de bytes. La solución es similar a ZeroMQ TCP, poniendo un tiempo de espera, pero esta vez de 100 ms, esto perjudica el tiempo de forma notable la respuesta de CoAP pero como estamos midiendo el tamaño, no tiene relevancia. Aún así después de la modificación, se ven fluctuaciones en la gráfica.
- En la petición número 1, en el tamaño de los bytes recibidos, todos los protocolos reciben, a excepción del protocolo UDP, valores incongruentes, pero en las siguientes peticiones/respuestas se estabiliza y da valores dentro de un margen congruente, las capturas de pantalla se toman sin tener en cuenta la primera petición.

Desde el punto de vista TFM Client, se han observado las siguientes incoherencias:

- En el protocolo http, no se envía ninguna carga útil debido a que se hace una petición GET a una url, sin embargo, el protocolo http tiene bastante carga de cabeceras, como la figura 3.39 y el tamaño del envío (SEND) debería ser mayor que otros protocolos.
- En CoAP, el tamaño de envío es demasiado grande, y más cuando no se envía ninguna carga de trabajo útil, porque se hace de forma similar al protocolo http, con una petición GET a una url, así que o bien CoAP no funciona tan bien o bien la medición de TrafficStats no es muy buena.

En la figura 9.5 se muestra la gráfica de los mensajes recibidos por parte de TFM Server, el protocolo de los sockets TCP muestra un tamaño de recepción un poco bajo, respecto al informado por TFM Client. El resto de protocolos tiene un tamaño coherente entre los dos prototipos.

En la figura 9.6 son los mensajes que se devuelven al cliente desde TFM Server, se observa como CoAP tiene un tamaño de mensaje bastante menor del esperado y un pico de 1000 bytes por debajo en la cuarta petición. Los

otros protocolos muestran un tamaño de mensaje de respuesta coherente con el prototipo TFM Client.

9.3. COMPARATIVA DE TIEMPOS Y TAMAÑO DE LOS MENSAJES CON WIRESHARK

Dado que en la sección 9.2 se han encontrado algunas inconsistencias entre ambos prototipos utilizando la clase TrafficStats de Android, se va a comprobar que los datos obtenidos directamente con los prototipos son correctos con la herramienta Wireshark. Esta herramienta ya ha sido utilizada en la sección 3.2.8 cuando se analizó el protocolo CoAP. Los resultados de Wireshark han sido exportados a varias hojas de cálculo que se detallan en el anexo B de esta memoria de trabajo.

La prueba consiste en realizar sólo una petición/respuesta y sumar todo el tráfico saliente (enviado) y todo el tráfico entrante (recibido) informado por Wireshark. Como la herramienta Wireshark debe estar instalada en un ordenador, se han desarrollado 3 clientes para PC de escritorio, uno para los sockets TCP, otro para los sockets UDP y otro para ZeroMQ. Para http se puede usar un navegador web y para CoAP se puede utilizar el proporcionado por CoAP.

Protocolo	Enviado Total	Recibido Total	Enviado Sólo Carga Protocolo *	Recibido Sólo Carga Protocolo **
HTTP	1275	8814	1275	831
Sockets TCP	764	8726	669	743
Sockets UDP	190	8608	116	625
CoAP	1358	8929	1358	946
ZeroMQ TCP	1214	8951	1172	968

Tabla 9.4: Cuadro resumen del resultado del test de batería ALL con Wireshark (Tamaño Mensajes)

** http y CoAP no tienen carga útil en el envío. Los otros protocolos: (1) sockets TCP la carga útil son 95 bytes, (2) sockets UDP la carga útil son 74 bytes y en (3) ZeroMQ la carga útil son 42 bytes, luego, en estos protocolos hay que restar estos bytes para obtener la carga del protocolo*

*** La carga útil de respuesta son 7983 bytes, luego se resta 7983 al recibido medio de todos los protocolos*

Cuando se analiza el tráfico con Wireshark también se puede obtener el tiempo que ha tardado una petición/respuesta completa restando el tiempo de la última fila de la traza completa al tiempo de la primera fila, se detallan los tiempos en la siguiente tabla:

Protocolo	Tiempo
HTTP	218
Sockets TCP	211
Sockets UDP	103
CoAP	441
ZeroMQ TCP	119

Tabla 9.5: Cuadro resumen del resultado del test de batería ALL con Wireshark (Tiempos)

9.4. CONCLUSIONES DE LA COMPARATIVA

En la siguiente tabla se resume todos los datos obtenidos de las pruebas empíricas, los tiempos están en milisegundos y el tamaño de los mensajes que porta el protocolo (sin carga útil) enviados y recibidos están en bytes:

Protocolo	Enviado Wireshark	Enviado TFM Client	Enviado TFM Server
HTTP	1275	575,4	558
Sockets TCP	669	541,8	165
Sockets UDP	116	28	28
CoAP	1358	1148,2	496
ZeroMQ TCP	1172	688	626
Protocolo	**Recibido Wireshark**	**Recibido TFM Client**	**Recibido TFM Server**
HTTP	831	519	617
Sockets TCP	743	487,4	517
Sockets UDP	625	28	28
CoAP	946	722	-3106
ZeroMQ TCP	968	728,8	767
Protocolo	**Tiempo Wireshark**	**Tiempo TFM Client**	**Tiempo TFM Server**
HTTP	218	278,4	40
Sockets TCP	211	188,06	38
Sockets UDP	103	180,2	35
CoAP	441	688,14	30
ZeroMQ TCP	119	263,06	25

Tabla 9.6: Cuadro resumen del resultado desde todos los puntos de vista del test de batería ALL

Las conclusiones que podemos extraer es que la clase TrafficStats de Android es un poco imprecisa al medir el tamaño de los mensajes, especialmente, en el prototipo TFM Server, en el prototipo TFM Client, los datos se asemejan, en general, bastante más a los datos obtenidos por Wireshark. Además, la clase TrafficStats no parece que tenga en cuenta algunas cabeceras http y no mide demasiado bien el protocolo UDP (sockets UDP y CoAP).

Respecto a los tiempos, el cálculo desde TFM Server, sólo tiene en cuenta el procesamiento del servidor y las lecturas desde Wireshark y desde TFM Client son algo más realistas de cuanto tiempo tarda en procesarse una solicitud al completo (cliente-servidor). Además desde Wireshark se ha utilizado un ordenador personal con red cableada y es normal que los tiempos sean algo más bajos que desde TFM Client.

Dado que CoAP ha salido muy mal parado en las pruebas realizadas, se ha pensado que CoAP no está pensado para un volumen de datos tan grande, aunque la verdad es que 7983 bytes no son muchos datos y se ha repetido la misma prueba pero de tipo LAST del sensor de batería con 50 repeticiones y el resultado ha sido el siguiente:

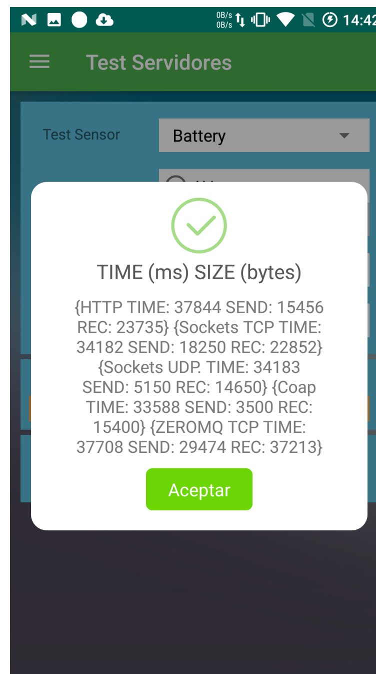


Figura 9.7: Captura del resultado del test de batería LAST desde el punto de vista TFM Client (Tiempos)

Protocolo	Tiempo Con Retardo	Tiempo Sin Retardo *	Petición/Respuesta Media **
HTTP	37844	12844	256,88
Sockets TCP	34182	9182	183,64
Sockets UDP	34183	9183	183,66
CoAP	33588	8588	171,76
ZeroMQ TCP	37708	12708	254,16

Tabla 9.7: Cuadro resumen del resultado del test de batería LAST desde el punto de vista TFM Client (Tiempos)

** Como son 50 peticiones/respuestas y son 500 ms de retardo en cada petición/respuesta, resulta que son 25000 ms, por lo que hay que restar el tiempo 25000 al tiempo con retardo*

*** Se divide el tiempo sin retardo por las 50 peticiones/respuestas*

En la tabla 9.7 se puede observar como el mejor tiempo es el de CoAP y curiosamente los sockets TCP y UDP han tardado prácticamente lo mismo. Respecto a los tamaños y fiándonos de las lecturas de TrafficStats en el prototipo TFM Client, CoAP también sale mejor parado con tamaño de los mensajes mucho más pequeños, cuando la carga útil del mensaje es menor.

Por protocolos se pueden extraer las siguientes conclusiones:

- El protocolo http tiene la ventaja de estar muy extendido y es muy interoperable y dado los datos obtenidos, sólo tendría sentido si vamos a

desarrollar una web para acceso remoto a los dispositivos y no para la comunicación directa entre dispositivos.

- Con los sockets TCP se han conseguido buenos tamaños de mensajes y buenos tiempos. Los sockets TCP están un poco penalizados ya que portan 95 bytes de carga en los mensajes enviados. Así como están desarrollados se podrían utilizar para la red de máquinas descentralizadas, se han hecho pruebas de estrés y funcionan bien, respondiendo a todas las peticiones, sin embargo, si el tamaño de los mensajes no es problema porque la red de comunicaciones es de alta velocidad, un buen candidato para TCP es utilizar ZeroMQ TCP.
- Los sockets UDP se han comportado muy bien y han sido los claros vencedores de las pruebas en todos los aspectos, excepto en la prueba que ha ganado CoAP por tiempos en el tipo LAST del test de batería. Además, al someterlos a una prueba de estrés desde un ordenador, el cliente ha llegado a perder algún paquete provocando un time out. Se podrían mejorar usando hilos de ejecución y de esta forma sincronizar mejor una petición con su respuesta. Si vamos a enviar y recibir pocos bytes y si los dispositivos no van a estar sometidos a una excesiva carga de trabajo, los sockets UDP tal y como están desarrollados se podrían utilizar para la red de máquinas.
- CoAP ha destacado en los tiempos, cuando no hay muchos datos que procesar y su simplicidad en la implementación, con la misma librería de core tenemos el cliente CoAP y el servidor CoAP. Cuando hay bastantes datos y viendo la traza de Wireshark del anexo B, se repiten mucho las cabeceras binarias y no parece que resulte ser muy eficiente repetirlas, al menos, en la implementación de Californium probada. CoAP está pensado para transmitir pocos bytes, cuando el datagrama tiene más de 8192 bytes, la petición o la respuesta falla. Es por este motivo que el máximo de tamaño de la respuesta se ha fijado en 7983 bytes. Su API tiene mecanismos para elevar este máximo pero no ha sido posible hacerlo.
- Se nota que el protocolo ZeroMQ está muy bien desarrollado, consiguiendo tiempos muy buenos en el lado del servidor y tiempos bastante buenos en el lado del cliente con sus colas y sus hilos de ejecución, además nos ofrece otros modelos de mensajería que pueden llegar a ser muy útiles. La implementación de ZeroMQ también es sencilla, como la de CoAP. ZeroMQ tiene cierta sobrecarga de datos en la carga que porta en el protocolo, pero para las redes de alta velocidad como el Wi-Fi, ZeroMQ funcionará muy bien. En una red de baja velocidad de 250 Kb/s pueden suponer retardos apreciables.

Protocolo	Config. Remota	Red de Baja Velocidad	Red de Alta Velocidad	Tamaño Mensajes Pequeño (< 1K)	Tamaño Mensajes Medio (1K < 65K)	Tamaño Mensajes Grande (> 65K)
HTTP	X					
Sockets TCP		X	X	X	X	X
Sockets UDP		X	X	X	X	No funciona
CoAP		X		X	No funciona a partir de 8192 bytes	No Funciona
ZeroMQ TCP			X	X	X	X

Tabla 9.8: Cuadro resumen de cuando utilizar los protocolos probados

** Una X indica si el protocolo es apto para el indicador analizado*

Resumiendo y explicando la tabla anterior:

- Utilizar http si vamos a utilizar una web para configuración remota o para mostrar información del dispositivo remoto vía navegador web.
- Los sockets TCP de Java funcionan muy bien en todos los escenarios. Las pruebas de estrés han sido correctas y los sockets TCP no han perdido paquetes.
- Los sockets UDP son muy simples y permiten una comunicación rápida y de pequeño tamaño. Cuando el tamaño de los mensajes es mayor que 65K ya no es posible su uso por las limitaciones del protocolo TCP/IP. Durante las pruebas desde el ordenador de escritorio se ha hecho una prueba de estrés y el cliente pierde algún paquete, cuando se envían varios decenas o cientos de peticiones en el mismo segundo. Se podría intentar mejorar la implementación actual de los sockets UDP, especialmente en el lado cliente que es donde se pierden los paquetes.
- CoAP tiene sentido usarlo cuando la red es de baja velocidad y/o cuando el tamaño de los mensajes es pequeño, menor que 1K. No funciona cuando el tamaño de los mensajes es mayor que 8192 bytes y cercano a ese limite el rendimiento es peor que otras alternativas analizadas.
- ZeroMQ funciona muy bien, es bastante rápido, sobre todo en la prueba que se hizo desde Wireshark pero tiene cierta sobrecarga de tamaño de los mensajes en el protocolo y no lo hace apto para redes de baja velocidad.

CAPÍTULO 10

10. CONCLUSIONES Y TRABAJO FUTURO

Uno de los objetivos fundamentales de la propuesta de este trabajo es la construcción dinámica de una red M2M descentralizada de malla parcial donde los sensores y actuadores pueden funcionar de forma autónoma sin necesidad de una plataforma IoT. La plataforma IoT puede formar parte de la red M2M pero es una máquina más de la red M2M, si la plataforma IoT se apaga, los sensores y actuadores de la red continúan trabajando con normalidad.

Se ha conseguido materializar la propuesta con la implementación de dos prototipos para la plataforma Android de smartphones y tablets. Por un lado, el prototipo TFM Server lee sus sensores internos y por otro lado el prototipo TFM Client realiza peticiones por red para obtener las lecturas de los sensores remotos descubiertos para accionar sus actuadores internos. En la implementación de los prototipos, los sensores anuncian su presencia por broadcast o multicast y los actuadores están escuchando la red para registrar los sensores remotos. Los actuadores tienen también un mecanismo de búsqueda por unicast múltiple para escanear la red y buscar sensores remotos.

En los prototipos se han utilizado varios patrones de diseño software, especialmente en el código fuente relacionado con la red M2M, lo que facilita el mantenimiento y la reutilización de código. Aunque existe un cierto trabajo inicial para implementar los patrones de diseño software, a largo plazo, las modificaciones y las ampliaciones de los prototipos resultarán ser mucho más sencillas que si no se usan patrones de diseño software.

También se ha diseñado una forma bastante simple pero efectiva para que varios sensores del mismo tipo puedan coexistir y no provoquen colisiones en el mismo punto de acceso con varios actuadores que necesiten el mismo tipo de sensor. Se ha conseguido con el etiquetado de la ubicación de forma que podemos tener varios sensores del mismo tipo en un punto de acceso pero sólo algunos actuadores harán uso de un sensor remoto, es decir, los actuadores harán peticiones sólo a aquellos sensores que compartan la misma etiqueta de ubicación.

Durante el desarrollo de los prototipos han surgido bastantes dificultades para mantener los procesos vivos y en funcionamiento cuando el dispositivo físico (smartphone o tablet) entra en modo suspensión o cuando se sale de la app del prototipo. Las dificultades han sido motivadas porque la plataforma de smartphones y las tablets de Android no está diseñada para funcionar como dispositivo IoT, con procesos en segundo plano que están de forma permanente y constante en ejecución, si no que está diseñada para satisfacer la experiencia de usuario maximizando la duración de la batería del dispositivo, especialmente en versiones recientes de Android. Por tanto, y a pesar de que se ha conseguido que los procesos de los prototipos continúen vivos y en funcionamiento, la plataforma de Android para smartphones y tablets, tal y como está diseñada, no es una buena candidata como plataforma para los dispositivos IoT y habría que eliminar las restricciones de los procesos en

segundo plano de la plataforma Android de smartphones y tablets. En el resto de aspectos, la plataforma Android, en los dispositivos físicos probados (smartphones y tablets), se ha comportado de forma correcta y ha respondido muy bien a las pruebas realizadas, tanto en aspectos de la red de comunicaciones como la escritura y lectura masiva de datos mediante SQLite con el API Room.

Respecto a las tecnologías inalámbricas, no hay duda que en la actualidad existen muchos medios de transmisión inalámbricos, algunos son muy conocidos, como los infrarrojos, el Bluetooth, el Wi-Fi, la comunicación por satélite por microondas y la comunicación terrestre de telefonía móvil. Otros medios de transmisión están cogiendo popularidad, como ZigBee y Z-Wave que están presentes en los gadgets actuales y están al alcance de los consumidores. Por otra parte, algunas tecnologías que han tenido presencia en el pasado y continúan teniendo presencia en el mercado actual, como Bluetooth o Wi-Fi se están adaptando o se han adaptado para dar soporte a M2M y al Internet de las Cosas.

Respecto a los protocolos de comunicaciones, están surgiendo nuevos protocolos IoT, como CoAP y ZeroMQ, por lo que existe en el mercado actual, cierto interés en estas tecnologías para dispositivos restringidos y microordenadores. En los prototipos se han utilizado 5 protocolos (http, sockets TCP, sockets UDP, CoAP y ZeroMQ TCP) y en una aplicación real para producción no sería necesario utilizar todos los protocolos al mismo tiempo. Se puede reservar http para configuración remota e información del dispositivo vía navegador web. CoAP se puede utilizar cuando el volumen de datos a transmitir es pequeño y ZeroMQ se puede utilizar cuando la cantidad de datos a transmitir supere las limitaciones de tamaño de los mensajes de CoAP. Una razón de usar CoAP y ZeroMQ puede estar motivada principalmente a que es posible que, con el tiempo, estos protocolos se transformen en un estándar de facto y la comunidad de arquitectos y desarrolladores software utilicen estos protocolos de manera masiva en sus proyectos IoT, aunque también puede ocurrir justo lo contrario, que los protocolos se abandonen y tengamos librerías de protocolos en nuestros proyectos IoT que ya no se van a actualizar nunca más. Esto nos lleva a un trabajo futuro y sería crear nuestro propio protocolo IoT con las características deseables para un protocolo IoT:

- Baja latencia de red.
- Bajo consumo de ancho de banda con tamaño de los mensajes lo más reducidos posible.
- Ser ligero, necesita poca memoria RAM y necesita poco espacio en disco para funcionar.
- Que esté disponible para una amplia gama de lenguajes de programación para facilitar la interoperabilidad entre distintas máquinas.

Como trabajo futuro de los prototipos, estos básicamente se pueden ampliar con más funcionalidades como:

- Habría que ampliar los prototipos con nuevos sensores y que estos puedan ser consumidos por los actuadores.
- Implantar y experimentar con más protocolos de comunicaciones, entre ellos los websockets y más protocolos IoT y sólo quedarse con los que ofrezcan mejores tiempos de respuesta y mejores tamaños de mensajes.
- Usar algún protocolo de mensajería como MQTT con fines muy específicos, como el descubrimiento de las máquinas. MQTT puede ser muy útil para una red de sensores y actuadores desplegada en Internet y para redes privadas conformadas por varios puntos de acceso y varias subredes privadas.
- Implantar en los prototipos, mecanismos de comunicación con plataformas IoT. Se podría utilizar alguna plataforma IoT existente, construir una plataforma IoT desde cero o bien se podría construir una plataforma IoT con las librerías que proporciona OMA Lightweight M2M.
- Desarrollar los prototipos para que puedan ser operados sin pantalla. Recordemos que los prototipos se han desarrollado para la plataforma de smartphones y tablets que sí tienen pantalla. Una forma de conseguirlo es reutilizando el servidor web para las llamadas API REST. Este servidor web es capaz de devolver páginas html y por tanto pueden ser accesibles por un navegador web desde otro dispositivo, como un ordenador personal o bien se puede crear una app que encuentre los dispositivos de la red y que después abra un navegador web para navegar por las páginas html del dispositivo remoto, accediendo a su configuración y a la información que proporciona de los sensores y actuadores que tiene instalado. Se pueden crear webs muy ricas en estos dispositivos con gráficas y listados no demasiado extensos porque todo el procesamiento de la vista la realiza el cliente en el navegador web. Este servidor ya se encuentra desplegado en el prototipo TFM Server, pero puede ser desplegado también en TFM Client de la misma forma.

Referencias

[**1**]: NXP, `https://www.nxp.com/`, 2020

[**2**]: Arduino, `https://store.arduino.cc/`, 2020

[**3**]: Raspberry, `https://www.raspberrypi.org/products/raspberry-pi-4-model-b/`, 2020

[**4**]: Bqeel, `https://www.bqeel.net/product/b07rn85dn4.html`, 2020

[**5**]: P. Kumar, R. Verma, A. Prakash, A. Agrawal, K. Naik, R. Tripathi, M. Alsabaan, T. Khalifa, T. Abdelkader, A. Abogharaf, *Machine-to-Machine (M2M) Communications: A Survey*. Elsevier, 2016

[**6**]: B. Forouzan, *Transmisión de datos y redes de comunicaciones (Cuarta Edición)*. Mc Graw Hill, 2006

[**7**]: M. Bhargava, *IoT Projects with Bluetooth Low Energy*. Packt, 2017

[**8**]: Y. Mehmood, N. Haider, M. Imran, A. Timm-Giel y M. Guizani, *M2M Communications in 5G: State-of-the-Art Architecture, Recent Advances, and Research Challenges*. IEEE Communications Magazine, 2017

[**9**]: Movistar SIMS M2M, `http://www.movistar.es/grandes-empresas/soluciones/fichas/machine-to-machine-m2m/`, 2020

[**10**]: J. T. J. Penttinen, *The Telecommunications Handbook: Engineering Guidelines for Fixed, Mobile and Satellite Systems*. Wiley, 2015

[**11**]: D. Bikash, *Radio Frequency Identification Technology: AnOverview of its Components, Principles and Applications*. International Journal of Science, Engineering and Technology Research (IJSETR), 2016

[**12**]: Dibyendu Bikash Datta, *Radio Frequency Identification Technology: AnOverview of its Components, Principles and Applications*. International Journal of Science, Engineering and Technology Research (IJSETR), 2016

[**13**]: D. Gislason, *Zigbee Wireless Networking*. Newnes, 2008

[**14**]: J. Vasseur y A. Dunkels, *Interconnecting Smart Objects with IP*. Elsevier, 2010

[**15**]: J. Haxhibeqiri, E. De Poorter, I. Moerman y J. Hoebeke, *A Survey of LoRaWAN for IoT: From Technologyto Application*. MDPI, 2018

[**16**]: A. Kurniawan, *Sigfox Development with SiPy, LoPy4, FiPy*. Lulu, 2020

[**17**]: C. González, V. García, B. Pelayo, J. Cueva, *Protocols and Applications for the Industrial Internet of Things*. IGI Global, 2018

[**18**]: C. Perry, *Mastering IOT: Build modern IoT solutions that secure and monitor your IoT infraestructure*. Packt, 2019

[**19**]: Martin De Saulles, *The Business of Data: Commercial Opportunities and Social Challenges in a World Fuelled by Data*. Innovation and Technology Horizons, 2020

[**20**]: M. L. Liu, *Computación Distribuida*. Pearson Addison Wesley, 2004

[**21**]: J. Garbajosa, F. Soriano, J. Moreno, *Tecnologías Software Orientadas a Servicios*. Fundación madri+d para el Conocimiento Velázquez, 2007

[**22**]: Apache WSS4J, `https://ws.apache.org/wss4j/`, 2020

[**23**]: Gastón C. Hillar, *MQTT Essentials - A Lightweight IoT Protocol*. Packt, 2017

[**24**]: Eclipse Paho, `https://www.eclipse.org/paho/downloads.php`, 2020

[**25**]: Z. Mahmood, *Connected Environments for the Internet of Things: Challenges and Solutions*. Springer, 2017

[**26**]: B. Adryan, D. Obermaier, P. Fremantle, *The Technical Foundations of IoT*. Artech House, 2017

[**27**]: F. Akgul, *ZeroMQ*. Packt, 2013

[**28**]: ZeroMQ - The Guide, *http://zguide.zeromq.org/page:all*. , 2020

[**29**]: R. Zurawski, *Industrial Communication Technology Handbook*. CRC Press, 2015

[**30**]: Eclipse Californium Tools, `https://github.com/eclipse/californium.tools`, 2020

[**31**]: Nanomsg Trade Mark, `https://nanomsg.org/gettingstarted/nng/index.html`, 2020

[**32**]: NNG Contributors, `https://github.com/nanomsg/nng`, 2020

[**33**]: I. Ishaq, D. Carels, G. K. Teklemariam, J. Hoebeke, F. Van den Abeele, E. De Poorter, I. Moerman y P. Demeester, *IETF Standardization in the Field of the Internet of Things (IoT): A Survey*. MDPI, 2013

[**34**]: Open Mobile Alliance, `https://www.omaspecworks.org/`, 2020

[**35**]: Eclipse Leshan, `https://www.eclipse.org/leshan/`, 2020

[**36**]: Eclipse Leshan Git, `https://github.com/eclipse/leshan#test-server-sandbox`, 2020

[**37**]: CLÚSTER ICT-AUDIOVISUAL DE MADRID, *Internet de las Cosas: Objetos Interconectados y dispositivos inteligentes*. Madrid Network, 2013

[**38**]: E. Gamma, *Patrones de Diseño*. Pearson Addison Wesley, 2009

[**39**]: D. Parsons, *Desarrollo de Aplicaciones Web Dinámicas con XML y Java*. Anaya, 2008

[**40**]: C. Szyperski, D. Gruntz y S. Murer, *Component Software: Beyond Object-Oriented Programming*. (Segunda Edición) Addison Wesley, 2002

[**41**]: Google Official Site for Android App Developers, `Android 10` `https://developer.android.com/about/versions/10`, 2020

[**42**]: Google Official Site for Android App Developers, `Wear OS` `https://developer.android.com/wear`, 2020

[**43**]: Google Official Site for Android App Developers, `Android TV` `https://developer.android.com/tv`, 2020

[**44**]: Google Official Site for Android App Developers, `Android Auto` `https://developer.android.com/cars`, 2020

[**45**]: Google Official Site for Android App Developers, `Android Things` `https://developer.android.com/things`, 2020

[**46**]: Google Official Site for Android App Developers, `Chrome OS` `https://developer.android.com/chrome-os/intro`, 2020

[**47**]: Google Official Site for Android App Developers, `Google Assistant` `https://developers.google.com/assistant`, 2020

[**48**]: R. Malveau y T. Mowbray, *Software Architect Bootcamp*. Prentice Hall, 2000

[**49**]: Google Official Site for Android App Developers, `Guía` `https://developer.android.com/guide`,

[**50**]: Google Official Site for Android App Developers, `Background Execution Limits` `https://developer.android.com/about/versions/oreo/background`, 2020

[**51**]: Google Official Site for Android App Developers, `Broadcasts overview` `https://developer.android.com/guide/components/broadcasts`, 2020

[**52**]: Google Official Site for Android App Developers, `JobService` `https://developer.android.com/reference/android/app/job/JobService`, 2020

[**53**]: Google Official Site for Android App Developers, `Descripción general de los servicios` `https://developer.android.com/guide/components/services?hl=es-419`, 2020

[**54**]: Google Official Site for Android App Developers, `Límites de ejecución en segundo plano` `https://developer.android.com/about/versions/oreo/background?hl=es-419`, 2020

[**55**]: Google Official Site for Android App Developers, HandlerThread
https://developer.android.com/reference/android/os/HandlerThread, 2020

[**56**]: Google Official Site for Android App Developers, Looper
https://developer.android.com/reference/android/os/Looper, 2020

[**57**]: Google Official Site for Android App Developers, AsyncTask
https://developer.android.com/reference/android/os/AsyncTask, 2020

[**58**]: Google Official Site for Android App Developers, Cómo programar tareas con WorkManager
https://developer.android.com/topic/libraries/architecture/workmanager?hl=es-419, 2020

[**59**]: Google Official Site for Android App Developers, Cómo optimizar tu app para Descanso y App Standby
https://developer.android.com/training/monitoring-device-state/doze-standby?hl=es-419, 2020

[**60**]: Google Official Site for Android App Developers, Room Persistence Library
https://developer.android.com/topic/libraries/architecture/room, 2020

Listado de siglas, abreviaturas y acrónimos

6LoWPAN	IPv6 over Low - Power Wireless Personal Area Networks
AMPQ	Advanced Message Queuing Protocol
API	Application Programming Interface
BLE	Bluetooth Low Energy
BPMN	Business Process Model and Notation
CoAP	Constrained Application Protocol
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CRUD	Create Read Update Delete
DAO	Data Access Object
DDS	Data Delivery Service
DHCP	Dynamic Host Configuration Protocol
DTLS	Datagram Transport Layer Security
FIFO	First In, First Out
GB	Gigabyte
Gb	Gigabit
Hz	Hercio
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
JAR	Java ARchive
JMX	Java Management Extensions
JVM	Java Virtual Machine
LAN	Local Area Network
LPWAN	Low Power Wide Area Network
M2M	Machine to Machine
MAC	Media Access Control Address
MAC	Media Access Control
MB	Megabyte
Mb	Megabit
MQTT	Message Queue Telemetry Transport

MVC	Modelo Vista Controlador
MVVM	Modelo-Vista–VistaModelo
NFC	Near field communication
NNG	Nanomsg-next-gen
OASIS	Organization for the Advancement of Structured Information Standards
OMA	Open Mobile Alliance
OMG	Object Management Group
P2P	Person to Person
PAN	Personal Area Network
PC	Personal Computer
QR	Quick Response code
RAM	Random Access Memory
REST	REpresentational State Transfer architectural style
RFID	Radio Frequency Identification
RPC	Remote Procedure Call
SDK	Software Development Kit
SIM	subscriber identity module
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
SSID	Service Set Identifier
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
UTP	Unshielded Twisted Pair
WAN	Wide Area Network
WSS4J	Web Service Security for Java
ZeroMQ	Zero Message Queuing

Anexo A. Documentación Adicional de los Prototipos

Recursos de Programación Utilizados

Recurso	*Versión*	*Descripción*
Kubuntu	18.04	Sistema operativo base para el desarrollo
Java	Open JDK 1.8.0_252	Lenguaje de programación utilizado
Android Studio	3.6.1	IDE de desarrollo Android para implementar apps nativas en Java
Gradle	5.6.4	Herramienta para la gestión y construcción de proyectos Java de Android (incluido en Android Studio)
Android Emulator	Imagen Nexus 5X API 25	Emulador Android para smartphones
Android Emulator	Imagen Nexus 5X API 27	Emulador Android para tablets
Android Emulator	Imagen Nexus 10X API 25	Emulador Android para smartphones

Dentro de cada prototipo se pueden consultar todas las dependencias o librerías externas que se han utilizado en cada uno de ellos. El fichero build.gradle de las apps es donde se indican las dependencias. Este fichero se encuentra en:

- $PATH_PROYECTO$/app/build.gradle

Cada dependencia utilizada está comentada para saber su uso en el o los prototipos, dentro del elemento dependencies del fichero build.gradle.

Dispositivos Físicos Android Utilizados

Dispositivo	*Versión Android*	*Tipo*
Yiayu S3	7.1 (API LEVEL 25)	Smartphone
Redmi Note 8 Pro	9.0 (API LEVEL 28)	Smartphone
HUAWEI MediaPad T5	8.0 (API LEVEL 26)	Tablet

Manual de Operación (Instalación/Ejecución)

El código fuente de ambos prototipos se suministra comprimido en formato zip. Descomprimir los archivos TFM Server.zip y TFM Client.zip. Después abrir el proyecto/carpeta descomprimida con Android Studio que debe ser instalado previamente. En este trabajo no se ha proveído de los binarios para la ejecución directa (sin IDE de desarrollo), ya que es más fácil hacerlo desde el propio Android Studio.

Una vez abierto el proyecto con Android Studio, este lo construye y lo deja listo para su ejecución. Es posible que se tenga que descargar algún SDK

o nos solicite la actualización de alguna dependencia. Si todo sale bien, para la ejecución tenemos dos posibilidades:

- Ejecutar la app en un emulador. El emulador ya está instalado en Android Studio y necesitamos una imagen del sistema operativo Android. Para descargar una imagen se puede acudir a Tools/AVD Manager y desde ahí descargar una imagen. Los prototipos se han probado en los emuladores de la tabla de los recursos de programación utilizados de este anexo.
- Ejecutar la app en un dispositivo físico. Tenemos que habilitar las herramientas de desarrollador en el dispositivo Android y desde aquí habilitar la depuración USB. Para instalar la app en el dispositivo tendremos que habilitar la opción de orígenes desconocidos (permitir la instalación de aplicaciones de origen desconocido). Por último, necesitaremos un cable USB (el del cargador) y conectar el smartphone o tablet al PC.

Para ambos casos, tenemos que hacer clic en el botón de Run App o Debug situado en la barra de herramientas de Android Studio y seleccionar la imagen para el emulador o el dispositivo físico para la ejecución de la app.

Manual de Usuario

Los prototipos disponen de un manual de usuario con los aspectos más importantes y relevantes en el manejo de los prototipos. Para acceder a un manual de un prototipo hay que navegar, desde la propia app, a Menú/Ayuda.

Documentación del Código Fuente

Dentro de cada proyecto, se encuentra la carpeta javadoc, que tiene la documentación del código fuente. Para visualizarla, se puede abrir el fichero index.html. En la documentación se puede ver para que sirve cada paquete y todas las clases y métodos utilizados en los proyectos.

Anexo B. Exportaciones de Wireshark de los Protocolos de Comunicaciones

Traza Completa http

No.	Time	Source	Destination	Protocol	Length	Info
3	2.422935853	192.168.95.2	192.168.95.202	TCP	66	43068 > 10101 [FIN, ACK] Seq=1 Ack=1 Win=502 Len=0 TSval=446817191 TSecr=1152871
4	2.423045833	192.168.95.2	192.168.95.202	TCP	74	43070 > 10101 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=446817191 TSecr=0 WS=128
5	2.423149861	192.168.95.2	192.168.95.202	HTTP	541	GET /rest/device/sensors/battery/all HTTP/1.1
6	2.512719162	192.168.95.202	192.168.95.2	TCP	66	10101 > 43060 [ACK] Seq=1 Ack=476 Win=365 Len=0 TSval=1154929 TSecr=446817192
7	2.512833470	192.168.95.202	192.168.95.2	TCP	74	10101 > 43070 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=1154929 TSecr=446817191 WS=256
8	2.512884291	192.168.95.2	192.168.95.202	TCP	66	43070 > 10101 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=446817281 TSecr=1154929
9	2.515015935	192.168.95.202	192.168.95.2	TCP	66	10101 > 43068 [ACK] Seq=1 Ack=2 Win=340 Len=0 TSval=1154930 TSecr=446817191
10	2.626820492	192.168.95.202	192.168.95.2	TCP	66	10101 > 43068 [FIN, ACK] Seq=1 Ack=2 Win=340 Len=0 TSval=1154941 TSecr=446817191
11	2.626866442	192.168.95.2	192.168.95.202	TCP	66	43068 > 10101 [ACK] Seq=2 Ack=2 Win=502 Len=0 TSval=446817395 TSecr=1154941
12	2.637028623	192.168.95.202	192.168.95.2	TCP	163	10101 > 43060 [PSH, ACK] Seq=1 Ack=476 Win=365 Len=97 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
13	2.637061283	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=98 Win=501 Len=0 TSval=446817405 TSecr=1154942
14	2.638441820	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=98 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
15	2.638472461	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=1546 Win=501 Len=0 TSval=446817407 TSecr=1154942
16	2.638790326	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=1546 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU
17	2.638826998	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=2994 Win=501 Len=0 TSval=446817407 TSecr=1154942
18	2.638975349	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=2994 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU
19	2.639012562	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=4442 Win=501 Len=0 TSval=446817407 TSecr=1154942
20	2.639106522	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=4442 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU
21	2.639131119	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=5890 Win=501 Len=0 TSval=446817408 TSecr=1154942
22	2.639249896	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=5890 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU
23	2.639279837	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=7338 Win=496 Len=0 TSval=446817408 TSecr=1154942
24	2.641740339	192.168.95.202	192.168.95.2	HTTP	809	HTTP/1.1 200 OK (application/json)
25	2.641763174	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=8081 Win=501 Len=0 TSval=446817410 TSecr=1154942
39	28.500084592	192.168.95.2	192.168.95.202	TCP	66	43070 > 10101 [FIN, ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=446843269 TSecr=1154929
40	28.500097209	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [FIN, ACK] Seq=476 Ack=8081 Win=501 Len=0 TSval=446843269 TSecr=1154942
41	28.524409149	192.168.95.202	192.168.95.2	TCP	66	10101 > 43070 [FIN, ACK] Seq=1 Ack=2 Win=87040 Len=0 TSval=1157530 TSecr=446843269
42	28.524427938	192.168.95.2	192.168.95.202	TCP	66	43070 > 10101 [ACK] Seq=2 Ack=2 Win=64256 Len=0 TSval=446843293 TSecr=1157530
43	28.525337080	192.168.95.202	192.168.95.2	TCP	66	10101 > 43060 [FIN, ACK] Seq=8081 Ack=477 Win=365 Len=0 TSval=1157531 TSecr=446843269
44	28.525347028	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=477 Ack=8082 Win=501 Len=0 TSval=446843294 TSecr=1157531

Traza Enviado http

No.	Time	Source	Destination	Protocol	Length	Info
3	2.422935853	192.168.95.2	192.168.95.202	TCP	66	43068 > 10101 [FIN, ACK] Seq=1 Ack=1 Win=502 Len=0 TSval=446817191 TSecr=1152871
4	2.423045833	192.168.95.2	192.168.95.202	TCP	74	43070 > 10101 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=446817191 TSecr=0 WS=12
5	2.423149861	192.168.95.2	192.168.95.202	HTTP	541	GET /rest/device/sensors/battery/all HTTP/1.1
8	2.512884291	192.168.95.2	192.168.95.202	TCP	66	43070 > 10101 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=446817281 TSecr=1154929
11	2.626866442	192.168.95.2	192.168.95.202	TCP	66	43068 > 10101 [ACK] Seq=2 Ack=2 Win=502 Len=0 TSval=446817395 TSecr=1154941
13	2.637061283	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=98 Win=501 Len=0 TSval=446817405 TSecr=1154942
15	2.638472461	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=1546 Win=501 Len=0 TSval=446817407 TSecr=1154942
17	2.638826998	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=2994 Win=501 Len=0 TSval=446817407 TSecr=1154942
19	2.639012562	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=4442 Win=501 Len=0 TSval=446817407 TSecr=1154942
21	2.639131119	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=5890 Win=501 Len=0 TSval=446817408 TSecr=1154942
23	2.639279837	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=7338 Win=496 Len=0 TSval=446817408 TSecr=1154942
25	2.641763174	192.168.95.2	192.168.95.202	TCP	66	43060 > 10101 [ACK] Seq=476 Ack=8081 Win=501 Len=0 TSval=446817410 TSecr=1154942

Traza Recibido http

No.	Time	Source	Destination	Protocol	Length	Info
6	2.512719162	192.168.95.202	192.168.95.2	TCP	66	10101 > 43060 [ACK] Seq=1 Ack=476 Win=365 Len=0 TSval=1154929 TSecr=446817192
7	2.512833470	192.168.95.202	192.168.95.2	TCP	74	10101 > 43070 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=1154929 TSecr=446817191 WS=256
9	2.515015935	192.168.95.202	192.168.95.2	TCP	66	10101 > 43068 [ACK] Seq=1 Ack=2 Win=340 Len=0 TSval=1154930 TSecr=446817191
10	2.626820492	192.168.95.202	192.168.95.2	TCP	66	10101 > 43068 [FIN, ACK] Seq=1 Ack=2 Win=340 Len=0 TSval=1154941 TSecr=446817191
12	2.637028623	192.168.95.202	192.168.95.2	TCP	163	10101 > 43060 [PSH, ACK] Seq=1 Ack=476 Win=365 Len=97 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
14	2.638441820	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=98 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
16	2.638790326	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=1546 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
18	2.638975349	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=2994 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
20	2.639106522	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=4442 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
22	2.639249896	192.168.95.202	192.168.95.2	TCP	1514	10101 > 43060 [ACK] Seq=5890 Ack=476 Win=365 Len=1448 TSval=1154942 TSecr=446817192 [TCP segment of a reassembled PDU]
24	2.641740339	192.168.95.202	192.168.95.2	HTTP	809	HTTP/1.1 200 OK (application/json)

Traza Completa Sockets TCP

No.	Time	Source	Destination	Protocol	Length	Info
2	1.582466075	192.168.95.2	192.168.95.202	TCP	74	57994 > 10102 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=447294357 TSecr=0 WS=128
3	1.699007771	192.168.95.202	192.168.95.2	TCP	74	10102 > 57994 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=1202648 TSecr=447294357 WS=256
4	1.699094137	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=447294474 TSecr=1202648
5	1.701024335	192.168.95.2	192.168.95.202	TCP	162	57994 > 10102 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=96 TSval=447294476 TSecr=1202648
6	1.704061799	192.168.95.202	192.168.95.2	TCP	66	10102 > 57994 [ACK] Seq=1 Ack=97 Win=87040 Len=0 TSval=1202648 TSecr=447294476
7	1.786063763	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=1 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
8	1.786114711	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=1449 Win=64128 Len=0 TSval=447294561 TSecr=1202656
9	1.786234152	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=1449 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
10	1.786265332	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=2897 Win=64128 Len=0 TSval=447294561 TSecr=1202656
11	1.787344866	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=2897 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
12	1.787374735	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=4345 Win=64128 Len=0 TSval=447294562 TSecr=1202656
13	1.787505661	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=4345 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
14	1.787535742	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=5793 Win=64128 Len=0 TSval=447294562 TSecr=1202656
15	1.787628798	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=5793 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
16	1.787656780	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=7241 Win=63488 Len=0 TSval=447294563 TSecr=1202656
17	1.787692733	192.168.95.202	192.168.95.2	TCP	810	10102 > 57994 [PSH, ACK] Seq=7241 Ack=97 Win=87040 Len=744 TSval=1202656 TSecr=447294476
18	1.787707334	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=7985 Win=64128 Len=0 TSval=447294563 TSecr=1202656
19	1.788016076	192.168.95.202	192.168.95.2	TCP	66	10102 > 57994 [FIN, ACK] Seq=7985 Ack=97 Win=87040 Len=0 TSval=1202656 TSecr=447294476
20	1.788354358	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [FIN, ACK] Seq=97 Ack=7986 Win=64128 Len=0 TSval=447294563 TSecr=1202656
21	1.794027385	192.168.95.202	192.168.95.2	TCP	66	10102 > 57994 [ACK] Seq=7986 Ack=98 Win=87040 Len=0 TSval=1202657 TSecr=447294563

Traza Enviado Sockets TCP

No.	Time	Source	Destination	Protocol	Length	Info
2	1.582466075	192.168.95.2	192.168.95.202	TCP	74	57994 > 10102 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=447294357 TSecr=0 WS=128
4	1.699094137	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=447294474 TSecr=1202648
5	1.701024335	192.168.95.2	192.168.95.202	TCP	162	57994 > 10102 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=96 TSval=447294476 TSecr=1202648
8	1.786114711	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=1449 Win=64128 Len=0 TSval=447294561 TSecr=1202656
10	1.786265332	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=2897 Win=64128 Len=0 TSval=447294561 TSecr=1202656
12	1.787374735	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=4345 Win=64128 Len=0 TSval=447294562 TSecr=1202656
14	1.787535742	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=5793 Win=64128 Len=0 TSval=447294562 TSecr=1202656
16	1.787656780	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=7241 Win=63488 Len=0 TSval=447294563 TSecr=1202656
18	1.787707334	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [ACK] Seq=97 Ack=7985 Win=64128 Len=0 TSval=447294563 TSecr=1202656
20	1.788354358	192.168.95.2	192.168.95.202	TCP	66	57994 > 10102 [FIN, ACK] Seq=97 Ack=7986 Win=64128 Len=0 TSval=447294563 TSecr=1202656

Traza Recibido Sockets TCP

No.	Time	Source	Destination	Protocol	Length	Info
3	1.699007771	192.168.95.202	192.168.95.2	TCP	74	10102 > 57994 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=1202648 TSecr=447294357 WS=256
6	1.704061799	192.168.95.202	192.168.95.2	TCP	66	10102 > 57994 [ACK] Seq=1 Ack=97 Win=87040 Len=0 TSval=1202648 TSecr=447294476
7	1.786063763	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=1 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
9	1.786234152	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=1449 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
11	1.787344866	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=2897 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
13	1.787505661	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=4345 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
15	1.787628798	192.168.95.202	192.168.95.2	TCP	1514	10102 > 57994 [ACK] Seq=5793 Ack=97 Win=87040 Len=1448 TSval=1202656 TSecr=447294476
17	1.787692733	192.168.95.202	192.168.95.2	TCP	810	10102 > 57994 [PSH, ACK] Seq=7241 Ack=97 Win=87040 Len=744 TSval=1202656 TSecr=447294476
19	1.788016076	192.168.95.202	192.168.95.2	TCP	66	10102 > 57994 [FIN, ACK] Seq=7985 Ack=97 Win=87040 Len=0 TSval=1202656 TSecr=447294476
21	1.794027385	192.168.95.202	192.168.95.2	TCP	66	10102 > 57994 [ACK] Seq=7986 Ack=98 Win=87040 Len=0 TSval=1202657 TSecr=447294563

Traza Completa Sockets UDP

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.95.2	192.168.95.202	UDP	116	46511 > 10103 Len=74
7	0.103720457	192.168.95.202	192.168.95.2	UDP	625	10103 > 46511 Len=7983

Traza Enviado Sockets UDP

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.95.2	192.168.95.202	UDP	116	46511 > 10103 Len=74

Traza Recibido Sockets UDP

No.	Time	Source	Destination	Protocol	Length	Info
7	0.103720457	192.168.95.202	192.168.95.2	UDP	625	10103 > 46511 Len=7983

Traza Completa CoAP

No.	Time	Source	Destination	Protocol	Length	Info
3	3.926012540	192.168.95.2	192.168.95.202	UDP	83	59840 > 10201 Len=41
5	4.106065421	192.168.95.202	192.168.95.2	UDP	573	10201 > 59840 Len=531
6	4.110002134	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
7	4.124880754	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
8	4.128046078	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
9	4.140792047	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
10	4.142889653	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
11	4.151266763	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
12	4.152528455	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
13	4.161154361	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
14	4.162230284	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
15	4.171448825	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
16	4.172181285	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
17	4.181307226	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
18	4.182532636	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
19	4.195014331	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
20	4.196115969	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
21	4.206037213	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
22	4.206957375	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
23	4.216727351	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
24	4.217558895	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
25	4.227737155	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
26	4.228509793	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
27	4.244501071	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
28	4.246046938	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
29	4.260849949	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
30	4.261871030	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
31	4.333428070	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
32	4.334589228	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
33	4.350632736	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
34	4.351835722	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
35	4.367308156	192.168.95.202	192.168.95.2	UDP	362	10201 > 59840 Len=320

Traza Enviado CoAP

No.	Time	Source	Destination	Protocol	Length	Info
3	3.926012540	192.168.95.2	192.168.95.202	UDP	83	59840 > 10201 Len=41
6	4.110002134	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
8	4.128046078	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
10	4.142889653	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
12	4.152528455	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
14	4.162230284	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
16	4.172181285	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
18	4.182532636	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
20	4.196115969	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
22	4.206957375	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
24	4.217558895	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
26	4.228509793	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
28	4.246046938	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
30	4.261871030	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
32	4.334589228	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43
34	4.351835722	192.168.95.2	192.168.95.202	UDP	85	59840 > 10201 Len=43

Traza Recibido CoAP

No.	Time	Source	Destination	Protocol	Length	Info
5	4.106065421	192.168.95.202	192.168.95.2	UDP	573	10201 > 59840 Len=531
7	4.124880754	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
9	4.140792047	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
11	4.151266763	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
13	4.161154361	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
15	4.171448825	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
17	4.181307226	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
19	4.195014331	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
21	4.206037213	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
23	4.216727351	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
25	4.227737155	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
27	4.244501071	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
29	4.260849949	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
31	4.333428070	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
33	4.350632736	192.168.95.202	192.168.95.2	UDP	571	10201 > 59840 Len=529
35	4.367308156	192.168.95.202	192.168.95.2	UDP	362	10201 > 59840 Len=320

Traza Completa ZeroMQ TCP

No.	Time	Source	Destination	Protocol	Length	Info
3	2.901906574	192.168.95.2	192.168.95.202	TCP	74	54124 > 10202 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=450257707 TSecr=0 WS=128
4	2.944988829	192.168.95.202	192.168.95.2	TCP	74	10202 > 54124 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=1498970 TSecr=450257707 WS=256
5	2.945017148	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=450257750 TSecr=1498970
6	2.954502449	192.168.95.2	192.168.95.202	TCP	76	54124 > 10202 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=10 TSval=450257759 TSecr=1498970
7	2.957944286	192.168.95.202	192.168.95.2	TCP	66	10202 > 54124 [ACK] Seq=1 Ack=11 Win=87040 Len=0 TSval=1498971 TSecr=450257759
8	2.964971850	192.168.95.202	192.168.95.2	TCP	77	10202 > 54124 [PSH, ACK] Seq=1 Ack=11 Win=87040 Len=11 TSval=1498972 TSecr=450257759
9	2.964990835	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=11 Ack=12 Win=64256 Len=0 TSval=450257770 TSecr=1498972
10	2.965790886	192.168.95.2	192.168.95.202	TCP	120	54124 > 10202 [PSH, ACK] Seq=11 Ack=12 Win=64256 Len=54 TSval=450257770 TSecr=1498972
11	2.973965598	192.168.95.202	192.168.95.2	TCP	119	10202 > 54124 [PSH, ACK] Seq=12 Ack=65 Win=87040 Len=53 TSval=1498973 TSecr=450257770
12	2.973983858	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=65 Ack=65 Win=64256 Len=0 TSval=450257779 TSecr=1498973
13	2.975774327	192.168.95.202	192.168.95.2	TCP	93	10202 > 54124 [PSH, ACK] Seq=65 Ack=65 Win=87040 Len=27 TSval=1498973 TSecr=450257770
14	2.975783525	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=65 Ack=92 Win=64256 Len=0 TSval=450257780 TSecr=1498973
15	2.979394256	192.168.95.2	192.168.95.202	TCP	152	54124 > 10202 [PSH, ACK] Seq=65 Ack=92 Win=64256 Len=86 TSval=450257784 TSecr=1498973
16	3.012302161	192.168.95.202	192.168.95.2	TCP	66	10202 > 54124 [ACK] Seq=92 Ack=151 Win=87040 Len=0 TSval=1498977 TSecr=450257784
17	3.014367964	192.168.95.202	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=92 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
18	3.014377203	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=1540 Win=64128 Len=0 TSval=450257819 TSecr=1498977
19	3.014720656	192.168.95.202	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=1540 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
20	3.014726692	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=2988 Win=64128 Len=0 TSval=450257819 TSecr=1498977
21	3.015758305	192.168.95.202	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=2988 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
22	3.015762957	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=4436 Win=64128 Len=0 TSval=450257820 TSecr=1498977
23	3.016013381	192.168.95.202	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=4436 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
24	3.016018993	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=5884 Win=64128 Len=0 TSval=450257821 TSecr=1498977
25	3.016205912	192.168.95.202	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=5884 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
26	3.016211338	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=7332 Win=64128 Len=0 TSval=450257821 TSecr=1498977
27	3.016288150	192.168.95.202	192.168.95.2	TCP	820	10202 > 54124 [PSH, ACK] Seq=7332 Ack=151 Win=87040 Len=754 TSval=1498977 TSecr=450257784
28	3.016293545	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=8086 Win=64128 Len=0 TSval=450257821 TSecr=1498977
29	3.017314720	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [FIN, ACK] Seq=151 Ack=8086 Win=64128 Len=0 TSval=450257822 TSecr=1498977
30	3.020943229	192.168.95.202	192.168.95.2	TCP	66	10202 > 54124 [FIN, ACK] Seq=8086 Ack=152 Win=87040 Len=0 TSval=1498977 TSecr=450257822
31	3.020979012	192.168.95.2	192.168.95.202	TCP	66	54124 > 10202 [ACK] Seq=152 Ack=8087 Win=64128 Len=0 TSval=450257826 TSecr=1498977

Traza Enviado ZeroMQ TCP

No.	Time	Source	Destination	Protocol	Length	Info
3	2.901906574	192.168.95.2	192.168.95.2	TCP	74	54124 > 10202 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=450257707 TSecr=0 WS=128
5	2.945017148	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=450257750 TSecr=1498970
6	2.954502449	192.168.95.2	192.168.95.2	TCP	76	54124 > 10202 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=10 TSval=450257759 TSecr=1498970
9	2.964990835	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=11 Ack=12 Win=64256 Len=0 TSval=450257770 TSecr=1498972
10	2.965790886	192.168.95.2	192.168.95.2	TCP	120	54124 > 10202 [PSH, ACK] Seq=11 Ack=12 Win=64256 Len=54 TSval=450257770 TSecr=1498972
12	2.973983858	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=65 Ack=65 Win=64256 Len=0 TSval=450257779 TSecr=1498973
14	2.975783525	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=65 Ack=92 Win=64256 Len=0 TSval=450257780 TSecr=1498973
15	2.979394256	192.168.95.2	192.168.95.2	TCP	152	54124 > 10202 [PSH, ACK] Seq=65 Ack=92 Win=64256 Len=86 TSval=450257784 TSecr=1498973
18	3.014377203	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=1540 Win=64128 Len=0 TSval=450257819 TSecr=1498977
20	3.014726692	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=2988 Win=64128 Len=0 TSval=450257819 TSecr=1498977
22	3.015762957	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=4436 Win=64128 Len=0 TSval=450257820 TSecr=1498977
24	3.016018993	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=5884 Win=64128 Len=0 TSval=450257821 TSecr=1498977
26	3.016211338	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=7332 Win=64128 Len=0 TSval=450257821 TSecr=1498977
28	3.016293545	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=151 Ack=8086 Win=64128 Len=0 TSval=450257821 TSecr=1498977
29	3.017314720	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [FIN, ACK] Seq=151 Ack=8086 Win=64128 Len=0 TSval=450257822 TSecr=1498977
31	3.020979012	192.168.95.2	192.168.95.2	TCP	66	54124 > 10202 [ACK] Seq=152 Ack=8087 Win=64128 Len=0 TSval=450257826 TSecr=1498977

Traza Recibido ZeroMQ TCP

No.	Time	Source	Destination	Protocol	Length	Info
4	2.944988829	192.168.95.2	192.168.95.2	TCP	74	10202 > 54124 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=1498970 TSecr=450257707 WS=256
7	2.957944286	192.168.95.2	192.168.95.2	TCP	66	10202 > 54124 [ACK] Seq=1 Ack=11 Win=87040 Len=0 TSval=1498971 TSecr=450257759
8	2.964971850	192.168.95.2	192.168.95.2	TCP	77	10202 > 54124 [PSH, ACK] Seq=1 Ack=11 Win=87040 Len=11 TSval=1498972 TSecr=450257759
11	2.973965598	192.168.95.2	192.168.95.2	TCP	119	10202 > 54124 [PSH, ACK] Seq=12 Ack=65 Win=87040 Len=53 TSval=1498973 TSecr=450257770
13	2.975774327	192.168.95.2	192.168.95.2	TCP	93	10202 > 54124 [PSH, ACK] Seq=65 Ack=65 Win=87040 Len=27 TSval=1498973 TSecr=450257770
16	3.012302161	192.168.95.2	192.168.95.2	TCP	66	10202 > 54124 [ACK] Seq=92 Ack=151 Win=87040 Len=0 TSval=1498977 TSecr=450257784
17	3.014367964	192.168.95.2	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=92 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
19	3.014720656	192.168.95.2	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=1540 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
21	3.015758305	192.168.95.2	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=2988 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
23	3.016013381	192.168.95.2	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=4436 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
25	3.016205912	192.168.95.2	192.168.95.2	TCP	1514	10202 > 54124 [ACK] Seq=5884 Ack=151 Win=87040 Len=1448 TSval=1498977 TSecr=450257784
27	3.016288150	192.168.95.2	192.168.95.2	TCP	820	10202 > 54124 [PSH, ACK] Seq=7332 Ack=151 Win=87040 Len=754 TSval=1498977 TSecr=450257784
30	3.020943229	192.168.95.2	192.168.95.2	TCP	66	10202 > 54124 [FIN, ACK] Seq=8086 Ack=152 Win=87040 Len=0 TSval=1498977 TSecr=450257822