Ariel Segura

Arquitectura de software de referencia para objetos inteligentes en internet de las cosas

Trabajo Final Integrador presentado para la obtención del título de Licenciado en Sistemas
Departamento de Desarrollo Productivo y Tecnológico

Director del Trabajo Final Integrador
Diego Azcurra y Sebastian Martins

ARQUITECTURA DE SOFTWARE DE REFERENCIA PARA OBJETOS INTELIGENTES EN INTERNET DE LAS COSAS

Alumno

APU Ariel SEGURA

Directores

Mg. Diego AZCURRA y Lic. Sebastián MARTINS

TRABAJO FINAL PRESENTADO PARA OBTENER EL GRADO
DE
LICENCIADO EN SISTEMAS

DEPARTAMENTO
DE DESARROLLO PRODUCTIVO Y TECNOLOGICO
UNIVERSIDAD NACIONAL DE LANUS

DICIEMBRE, 2015

RESUMEN

La evolución de los sistemas embebidos, que tuvo lugar junto al bajo costo y ubicuidad de internet, ha tenido como resultado el paradigma Internet de las Cosas. El objetivo de este paradigma es convertir los objetos que nos rodean en objetos inteligentes de forma tal que, comunicándose a través de Internet, puedan percibir lo que sucede en su entorno y poder reaccionar frente a ello. Se han llevado a cabo numerosos trabajos en materia de estandarización para este paradigma como, por ejemplo, protocolos de comunicación, topologías de red y arquitecturas software de alto nivel que consideran, por sobre todas las cosas, cómo integrar todas las piezas de una solución de Internet de las Cosas en un solo sistema. Sin embargo, no se han identificado arquitecturas de software para la construcción del software de un objeto inteligente que describa los componentes básicos que se deben incluir. En este trabajo se propone una arquitectura de referencia para la construcción de objetos inteligentes en Internet de las Cosas cuyo objetivo es que los ingenieros de software tengan una estructura en la cual basarse y poder reutilizar.

ABSTRACT

The evolution of embedded systems that came along with the ubiquity of Internet -and its low cost- has given birth to the Internet of Things paradigm, which objective is to convert objects around us into smart objects that communicate by Internet and can perceive what is happening on its environment and react to it. Several researches have been carried out with regards to standards such as communication protocols, network topologies and high level software architectures that mainly consider the way components should be integrated in an Internet of Things application. Nevertheless, no software architecture that describes the basic components that a smart object should include has been found. In this grade thesis, a software architecture for smart objects is proposed in order to provide software engineers a base structure they can reuse.

DEDICATORIA

A mis viejos, Carlos y Romina, por estar y bancarme siempre.

A mis abuelos, "Quita", Olga, Luis y "Neneco", por tanto cariño y confianza.

A mi novia, Priscila, por la paciencia y el inquebrantable acompañamiento.

A mis amigos, los hermanos que la vida me dio.

AGRADECIMIENTOS

A la Universidad Nacional de Lanús por acogerme con generosidad de “alma mater” para que pudiera llevar a cabo mis estudios de Licenciatura en Sistemas.

Al profesor Mg. Diego Azcurra por confiar en mí desde un primer momento.

Al Dr. Ramón García-Martínez por el apoyo y la orientación que recibí durante toda mi carrera.

Al profesor Ing. Damián Santos por su gran aporte en mi desarrollo profesional.

Al profesor Lic. Sebastián Martins por sus comentarios y sugerencias sobre este trabajo.

A los profesores de la Licenciatura en Sistemas de la Universidad Nacional de Lanús por su enseñanza.

ÍNDICE

ÍNDICE DE FIGURAS

NOMENCLATURA

EEPROM	Electrically Erasable Programmable Read-Only Memory
HTTP	HyperText Transfer Protocol
I/O	Input/Output
IAAS	Infrastructure as a Service
IOT	Internet of Things (En español, Internet de las Cosas)
IP	Internet Protocol
iPAAS	Integration Platform as a Service
JSON	JavaScript Object Notation
LAN	Local Area Network
LCD	Liquid crystal display
M2M	Machine To Machine
MVC	Model-View-Controller
PAAS	Platform as a Service
RAML	Restful API Modelling Language
REST	Representational State Transfer
SAAS	Software as a Service
SD	Secure Digital
SOA	Software Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TFL	Trabajo Final de Licenciatura
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WAN	Wide Area Network
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language

1. INTRODUCCION

En este Capítulo se plantea el contexto del trabajo final de licenciatura (sección 1.1), se da una delimitación del problema (sección 1.2), se plantean los elementos de la solución propuesta (sección 1.3) y se resume la estructura del trabajo final de licenciatura (sección 1.4).

1.1. CONTEXTO DEL TRABAJO FINAL DE LICENCIATURA

Los avances tecnológicos de la última década nos han dejado en una posición en donde se puede percibir que el futuro de la tecnología va a tener lugar lejos de la computación tradicional, utilizada desde computadoras personales y smartphones o sistemas informáticos empresariales. La combinación de internet y las tecnologías emergentes, como la localización en tiempo real, sensores embebidos en objetos cotidianos y la comunicación a alta velocidad y bajo costo, han permitido la concepción de objetos inteligentes que tienen la capacidad de no sólo comprender lo que sucede a su alrededor sino también influir en el ambiente en el que se encuentra [Kortuem, G., et al, 2010].
Esta idea es la base del paradigma Internet de las Cosas donde muchos objetos (como sensores, actuadores, smartphones o electrodomésticos) en el ambiente pueden cooperar entre ellos para alcanzar objetivos en común [Atzori, L., et al, 2010]. La fortaleza principal de IOT radica en el impacto sobre los aspectos cotidianos de los potenciales usuarios [Atzori, L., et al, 2010]. De hecho este concepto de conectar las cosas que nos rodean a Internet está volviéndose cada vez más fuerte y se espera que 50 mil millones de dispositivos estén conectados a internet en 2020 [Evans, D., 2012].
Hay dos puntos de vista sobre aplicaciones IOT: desde el usuario y desde lo tecnológico.
Desde el punto de vista del usuario las aplicaciones que se esperan están sobre dos aspectos importantes: aspecto laboral y aspecto doméstico. En lo laboral se espera que las consecuencias de aplicar IOT se vean en campos como automatización e industrias manufactureras, áreas de logística y transporte de bienes [Atzori, L., et al, 2010]. En cuanto a lo doméstico, por otra parte, se espera que las aplicaciones tengan alto impacto en la calidad de vida de las personas por medio de la domótica, es decir, automatización en el hogar, asistentes virtuales o e-health. Por ejemplo, algunas posibles aplicaciones de IOT incluyen:

- Transporte y Logística: Se puede obtener información en tiempo real sobre el estado del tráfico, de la ruta y de la carga que un camión está transportando.
- Cuidado de la Salud: En los hospitales se puede mantener un inventario en tiempo real de los objetos sin necesidad de tener personal encargado de registrar los movimientos. Otra

aplicación interesante es la identificación de personas accidentadas y aviso en tiempo real a su familia.

- Ambientes inteligentes: La capacidad de transformar un hogar en un hogar inteligente gracias a la domótica es una aplicación que más desarrollo tuvo en los últimos años.

Los objetos inteligentes están construidos en base a sistemas embebidos que son computadoras pequeñas, de bajos recursos y autónomas que realizan el mismo trabajo infinitamente y están dotadas de un microcontrolador y dispositivos de entrada y salida. Estos objetos inteligentes siguen tres dimensiones [Kortuem, G., et al, 2010]:

- Awareness (O capacidad de percepción): Es la capacidad del objeto inteligente de censar, interpretar y reaccionar a eventos que ocurren en el mundo físico o digital.
- Representación: Se refiere a la capacidad de modelado que tiene un objeto inteligente, es decir, en términos de diseño del software, cómo se puede modelar un objeto inteligente.
- Interacción: Se refiere a la capacidad del objeto inteligente de comunicarse con el usuario en términos de entrada, salida, control y feedback (o devolución).

Respecto lo tecnológico, se han identificado dos aspectos claves que deben ser tenidos en cuenta a la hora de trabajar en IOT [Gubbi, J., et al, 2013]:

1. Un consenso común en la comunidad sobre usuarios, aplicaciones y conceptos. Una de las dificultades que atraviesa IOT en estos tiempos es que se la utiliza para muchos propósitos, normalmente con fines de marketing. Si bien muchas de estas soluciones incluyen de alguna manera comunicación entre objetos inteligentes, apenas muestran capacidad de interoperabilidad ya que por lo general son soluciones pensadas para cumplir requerimientos específicos [Internet of Things Architecture, 2013].

Arquitecturas de Software y protocolos de comunicación que procesen y transporten la información del contexto a donde sea necesario. Se han diseñado arquitecturas de software para aplicaciones IOT ([Atzori et al. 2010]; [IOT-A, 2013]; [Misra et al. 2015]) que serán explicadas en el capítulo 2.

1.2. DELIMITACIÓN DEL PROBLEMA

En el desarrollo de soluciones IOT se ha identificado como un gran inconveniente la tendencia a desarrollar soluciones para problemas particulares y acotados que tienen arquitecturas específicas y no reutilizables que carecen de escalabilidad ([Atzori et al. 2010]; [Carretero et al. 2013]; [IOT-A, 2013]; [Misra et al. 2015]). Si bien se han propuesto arquitecturas de software para Internet de las Cosas en varios trabajos ([Atzori et al. 2010]; [IOT-A, 2013]; [Misra et al. 2015]), no se han identificado arquitecturas de software prácticas específicas que definan los componentes y la

estructura que debe tener el software para la construcción de objetos inteligentes basados en sistemas embebidos. La ausencia de este tipo de arquitecturas puede traer como consecuencia que, quienes están dando sus primeros pasos en la construcción de esta clase de sistemas embebidos, no tengan una arquitectura como referencia en la cual basarse o poder reutilizar.

1.3. SOLUCIÓN PROPUESTA

La solución propuesta es una arquitectura de software para objetos inteligentes. Dicha solución presenta dos componentes: una arquitectura de software en capas que posee todos los componentes que un objeto inteligente necesita, presentándose a la misma en un vistazo general y luego en detalle, y un conjunto de vistas arquitectónicas que permiten tener diferentes perspectivas de la arquitectura propuesta y, por ende, del objeto inteligente, mostrando cómo se relacionan los componentes.

1.4. VISIÓN GENERAL DEL TFL

Este trabajo final de licenciatura está conformado por siete capítulos y un anexo: Capítulo de Introducción, Estado del Arte, Descripción del Problema, Solución, Prueba del Concepto, Conclusiones, Referencias y, por último, un Anexo.

En el Capítulo Introducción se plantea el contexto del trabajo final de licenciatura, se da una delimitación del problema, se plantean los elementos de la solución propuesta, se presentan las publicaciones del autor vinculadas a las investigaciones realizadas en el desarrollo del trabajo final de licenciatura y se resume la estructura del trabajo final de licenciatura.

En el Capítulo Estado del Arte se presentan distintas teorías y técnicas que son concurrentes con los objetivos de este TFL. Se presentan Conceptos de Arquitecturas de Software, Patrones Arquitectónicos de Software, Modelos de Representación para Arquitecturas de Software, Conceptos de Internet y de Objeto Inteligente, un marco contextual de Internet de las Cosas y un conjunto de Arquitecturas de Software que van a ser consideradas en el desarrollo de este trabajo.

En el Capítulo Descripción del Problema se identifica el problema de investigación de este trabajo final de licenciatura, se caracteriza el problema abierto y se concluye con un sumario de investigación.

En el Capítulo Solución se presenta: una Arquitectura de Software de Referencia para Objetos Inteligentes en Internet de las Cosas, del cual se abordan las cuestiones generales, propuesta de la arquitectura, relación entre componentes y un ejemplo que demuestra su aplicación. En primer lugar se presenta un vistazo general de la arquitectura propuesta cuyos componentes principales son la

capa de presentación, sistemas externos, servicios, lógica de negocio, acceso a datos, recurso virtual y recurso físico. Luego, se presenta un esquema detallado de la arquitectura que contiene los detalles de cada capa mostrando sus subcomponentes. Por último, se presentan tres vistas arquitectónicas que ayudan a comprender cómo se relacionan los componentes de la arquitectura propuesta.

En el Capítulo Prueba de Concepto, se presenta una prueba de concepto perteneciente a la arquitectura de software de referencia para objetos inteligentes en internet de las cosas. El caso de prueba de concepto es el caso del desarrollo de una Plataforma Multipropósito de Telemetría y Telecomando a través de Internet basada en Sistemas Embebidos, cuya solución está basada en la arquitectura propuesta en el presente trabajo.

En el Capítulo Conclusiones, se presentan las aportaciones de este trabajo final de licenciatura y se formulan las futuras líneas de investigación que se consideran de interés en base al problema abierto que se presenta en este trabajo.

En el Capítulo Referencias se listan todas las publicaciones consultadas para el desarrollo de esta investigación.

En el Capítulo Anexo se adjunta el resultado de la fase de codificación de la prueba del concepto en lenguaje C++.

2. ESTADO DE LA CUESTIÓN

En este capítulo se presenta el estado de la cuestión sobre distintas teorías, conceptos y técnicas que son concurrentes con los objetivos de este trabajo final de licenciatura. Se presentan Conceptos de Arquitecturas de Software (sección 2.1), Patrones Arquitectónicos de Software (Sección 2.1.1), Modelos de Representación para Arquitecturas de Software (sección 2.1.2), Conceptos de Internet (Sección 2.2) y de Objeto Inteligente (Sección 2.3), un marco contextual de Internet de las Cosas (Sección 2.4) y un conjunto de Arquitecturas de Software que van a ser consideradas en el desarrollo de este trabajo (Sección 2.4.1).

2.1. CONCEPTOS DE ARQUITECTURA DE SOFTWARE

La palabra arquitectura proviene del griego y es la conjunción de dos palabras: *arjé*, y *tektón*. Arjé significa al que manda, al principio mientras que tekton a construir o edificar, definiendo así al arquitecto como el encargado o el que define las bases y los principios de lo que se va a construir , y a la arquitectura como el conocimiento y la práctica que permite determinar las cuestiones básicas para construir un edificio [Morales, I. 2000].

La historia de la arquitectura tradicional data de miles de años atrás cuando se realizaron los primeros ladrillos secados al sol allá por el año 6000 A.C. Luego con el correr de los años aparecieron estilos arquitectónicos como los que llevan las obras construidas en la época de Alejandro Magno o los sistemas constructivos romanos, dirigidos a la eficacia: economía de tiempo, de recursos y de medios [Morales, I. 2000]. Entrada la edad media, aparece el estilo gótico con edificios de estructuras grandes (casi el doble de altura de los estilos anteriores). En la edad moderna se introdujo el concepto de ciencia, haciendo así que la experiencia abandone su papel en las innovaciones.

Si bien la arquitectura tradicional no es objeto de este trabajo, lo cierto es que existe un concepto similar al concepto en la ingeniería de software. De hecho, [Pressman, 2002] enuncia:

“Cuando hablamos de la arquitectura de un edificio, nos vienen a la cabeza diferentes atributos. A nivel más básico, pensamos en la forma global de la estructura física. Pero, en realidad, la arquitectura es mucho más. Es la forma en la que los diferentes componentes del edificio se integran para formar un todo unido. Es la forma en que el edificio encaja en su entorno y con los otros edificios de su vecindad. Es el grado en el que el edificio consigue su propósito fijado y satisface las necesidades de sus propietarios. […]. Son los pequeños detalles – el diseño de las

instalaciones eléctricas, del tipo de suelo, de la colocación de tapices y una lista casi interminable. Y, finalmente, es arte. ”

De la misma forma que en la arquitectura tradicional fallar al considerar escenarios claves, evitar decisiones o consecuencias a largo plazo pueden dejar una solución software con un riesgo muy alto de fracaso. Es por eso que se dice que la arquitectura de software es en realidad un conjunto de estructuras, propiedades y relaciones que conforman el “edificio” que dará soporte a la solución software que se va a desarrollar y su desarrollo es de vital importancia básicamente por tres razones claves [Pressman, 2002]:

- Facilita la comunicación entre las partes interesadas en el desarrollo del software
- Permite tomar decisiones tempranas y evitar un impacto mayor en una etapa posterior del desarrollo del proyecto
- Permite comprender fácilmente la estructura y el flujo de trabajo de sus componentes

Además, la arquitectura de software es considerada un puente entre la fase de diseño y la ingeniería de requerimientos [Sommerville, I., 2011] dado que tiene una relación directa entre decisiones de arquitectura y los requerimientos [Ferre, X. et al, 2003]. Sin embargo, [Sommerville, I., 2011] plantea que eventualmente pueden existir dos objetivos a la hora de pensar en arquitectura de software:

1. Como una forma de facilitar la discusión acerca del diseño del sistema: Esta visión es útil porque pueden efectuarse discusiones de alto nivel del sistema en etapas tempranas del ciclo de vida sin pensar en los detalles de implementación de las funcionalidades.
2. Como una forma de documentar una arquitectura que se haya diseñado: Esta visión está más orientada al mantenimiento y evolución del sistema dado que aquí se documentan los componentes, interfaces y conexiones, una vez desarrollado.

En cualquier caso, a lo largo de la historia de la ingeniería de software se ha identificado que, a pesar de que la mayoría de los sistemas son distintos entre sí, existen algunas similitudes entre las arquitecturas de los mismos. Estas similitudes suelen seguir lo que se conoce como patrón o estilo arquitectónico que captan la esencia de una arquitectura que se usó en diferentes sistemas de software [Sommerville, I., 2011].

En [Microsoft Patterns & Practices Team., 2009] se define a un patrón arquitectónico como un conjunto de principios que proveen un framework abstracto para una familia de sistemas que promueve la reutilización de componentes y diseños mediante soluciones a problemas recurrentes. Es decir, un patrón arquitectónico son un conjunto de decisiones y principios que encajan como solución en determinados tipos de problemas que suelen repetirse en varios sistemas. Uno de los beneficios que se destacan es que mediante estos patrones se logra un lenguaje común que todos los

arquitectos de software entienden. Por otra parte, y tal como se mencionó con anterioridad, proveen un medio de comunicación para discutir sin tener en cuenta detalles de implementación o de tecnologías a utilizar. Por ejemplo, al hablar de un patrón de diseño Cliente-Servidor, ya cualquier arquitecto que conozca el patrón entiende de lo que le están hablando y no necesita que le expliquen cómo funcionaría el software en tal arquitectura. Otra característica importante de la arquitectura de software respecto los patrones de diseño es que casi nunca se limitan a un solo estilo arquitectónico si no que suelen ser utilizados en conjunto para dar paso a la solución completa del sistema. Por ejemplo, puede tenerse una arquitectura orientada a servicios desarrollada mediante una arquitectura en capas y orientada a objetos. O como es el caso de aplicaciones web que combinan estilos arquitectónicos de N- Niveles para el despliegue de la aplicación, mientras que en la capa de presentación utilizan el patrón de diseño Modelo-Vista-Controlador (MVC).

Sommerville, I. [2011] también coincide en que los patrones son una forma de presentar, compartir y sobre todo reutilizar el conocimiento sobre los sistemas de software y además agregar que un patrón arquitectónico es una descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos de modo tal que se pueda demostrar que este estilo arquitectónico tuvo éxito en sistemas anteriores. Además, agrega que los patrones no son aptos para todos los problemas y es por eso que a la hora de definirlo, debe dejarse en claro qué problema resuelve, cuándo es conveniente utilizarlo y cuáles son sus fortalezas y debilidades.

En [Buschmann, F. et al, 1996] también se define a un patrón arquitectónico como principios y decisiones que resuelven un problema de diseño recurrente que aparece en situaciones de diseño específicas y que traen consigo pruebas de que ha tenido éxito anteriormente. Además, agregan que si bien presenta una estructura básica de la solución al problema, no contiene una solución detallada e implementada. Sólo contiene un esquema para una solución genérica a una familia de problemas que el ingeniero de software debe implementar en función de los requerimientos específicos que debe cumplir la arquitectura de software en cuestión.

A continuación se muestran algunos patrones arquitectónicos y de diseño comúnmente utilizados en el diseño arquitectónico de software (sección 2.1.1).

2.1.1. Patrones Arquitectónicos de Software

En esta sección se muestran los patrones y estilos arquitectónicos de software más importantes:

- Cliente/Servidor (Sección 2.1.1.1): Divide el sistema en dos aplicaciones donde el cliente realiza peticiones al servidor.

- Arquitectura basada en componentes (Sección 2.1.1.2): Descompone la aplicación en componentes funcionalmente reutilizables que exponen una interfaz de comunicación bien definida.
- Arquitectura en capas (Sección 2.1.1.3): Divide los objetivos de la aplicación en pilas de capas agrupadas.
- Orientada a objetos (Sección 2.1.1.4): Un paradigma basado en la división de responsabilidades para una aplicación o sistema en objetos reutilizables y "autónomos" que contienen los datos y el comportamiento relevante para ese objeto.
- Arquitectura Orientada a Servicios (SOA, del inglés Service-Oriented Architecture) (Sección 2.1.1.5): Se refiere a aplicaciones que exponen y consumen funcionalidades como un servicio usando contratos y mensajes.
- Modelo Vista Controlador (MVC) (Sección 2.1.1.6): Es un patrón para separar los roles de una aplicación en distintos componentes. El modelo representa los datos (Un dominio que incluye la lógica de negocio), la vista que representa la interfaz de usuario y el controlador que gestiona peticiones, manipula el modelo y ejecuta otras operaciones.
- Fachada de aplicación (Sección 2.1.1.7): Un componente que típicamente provee una interfaz simplificada a los componentes de negocio, comúnmente combinando múltiples operaciones de negocio en una sola operación que simplifica la lógica de negocio.

2.1.1.1 Cliente-Servidor

Uno de los patrones arquitectónicos más conocido es el modelo Cliente-Servidor (Figura 2.1). Las arquitecturas cliente-servidor son comúnmente utilizadas en sistemas distribuidos y se caracterizan por la separación de objetivos o funcionalidades en dos o más nodos en donde algunos nodos toman el rol de servidor y otros de cliente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores [Sommerville, I. 2011] a través de una red. Más genéricamente, este patrón describe la relación entre un cliente y uno o más servidores donde el cliente inicia una o más peticiones, espera por la respuesta y luego procesa la respuesta recibida. El servidor típicamente autoriza al usuario y lleva a cabo el procesamiento requerido para generar el resultado [Microsoft Patterns & Practices Team., 2009]. Claro está, que tanto cliente como servidor deben mantener el mismo protocolo de comunicación. Los principales componentes de este modelo entonces son [Sommerville, I. 2011]:

1. Un conjunto de servidores que ofrecen servicios a otros componentes. Como servidores web o de impresión.

2. Un conjunto de clientes que solicitan los servicios que ofrecen los servidores. Como navegadores web.
3. Una red que permite a los clientes acceder a dichos servicios. Como, por ejemplo, Internet.

Por lo general este patrón arquitectónico aplica cuando el software necesita servir a varios clientes, como aplicaciones web, procesos de negocio que necesitan ser utilizados a lo largo de una organización o se están desarrollando servicios para que sean consumidos por otras aplicaciones [Microsoft Patterns & Practices Team., 2009]. Otras aplicaciones para este patrón son: Centralización de almacenamiento de datos o cuando se necesitan soportar distintos tipos de dispositivos como dispositivos móviles y laptops.

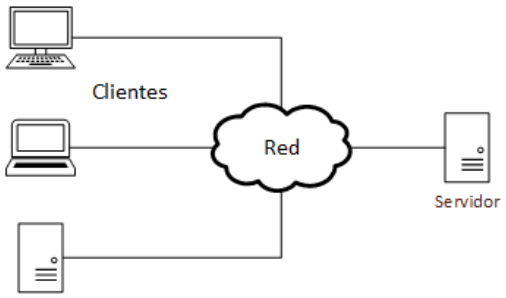

Figura 2.1. *Modelo Cliente-Servidor*

2.1.1.2 Arquitectura basada en componentes

El objetivo de este patrón es descomponer la aplicación en componentes funcionalmente reutilizables que exponen una interfaz de comunicación bien definida [Microsoft Patterns & Practices Team., 2009]. Los principios claves del uso de este estilo es el diseño de componentes que sean [Microsoft Patterns & Practices Team., 2009]:

- Reutilizables: Tal como se mencionó anteriormente estos componentes deben ser diseñados para ser reutilizados en diferentes escenarios y diferentes aplicaciones.
- Reemplazables: Los componentes pueden llegar a ser reemplazados por otros componentes similares. Es necesario que para esto, tengan interfaces similares.
- Agnósticos al contexto: Los componentes son diseñados para operar en diferentes ambientes y contextos. La información puntual y específica de cada escenario debería ser pasada al componente cuando sea oportuno, de ninguna manera debe estar incluida en él o debe ser accedida desde el componente.
- Extensibles: Un componente debe ser extensible para otorgar nuevas funcionalidades o comportamientos.

- Encapsulados: Los componentes exponen interfaces que permiten a quien lo invoquen usar su funcionalidad y así no revelan detalles internos de implementación o variables.
- Independientes: Los componentes son diseñados para tener un mínimo de dependencias con otros componentes. Por eso los componentes pueden ser desplegados en cualquier ambiente sin afectar otros componentes o sistemas.

Los casos típicos de componentes pueden ser controles de interfaz de usuario como botones o tablas, aunque también pueden ser componentes de software que estén ocultos al usuario y sirvan de apoyo a otros componentes dentro del mismo sistema.

Este patrón aplica cuando se tiene un conjunto de componentes o se pueden obtener a través de un tercero o bien cuando se piensa en la arquitectura del sistema como un conjunto de componentes listos para ser reemplazados y actualizados a nivel individual.

Por ejemplo en la figura 2.2 se puede observar la arquitectura de software para un sistema de compra en líneas. El mismo está compuesto por tres grandes componentes: Carrito de compras, procesador de pago y gestor de orden de compra.

En términos generales, el carrito de compras lleva un inventario de la compra que el usuario está realizando en el sitio en línea. Una vez que el usuario finaliza su pedido, le envía la información que necesita al procesador de pago para que efectúe el débito correspondiente en la cuenta del usuario y luego el gestor de orden de compra recibe los datos necesarios para disparar el flujo de logística correspondiente. Como se puede observar, los componentes son reutilizables y reemplazables, pues un procesador de pagos o carrito de compras se comportan de igual forma en la gran mayoría de los sistemas de compras en línea.

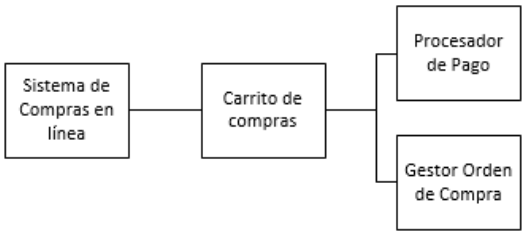

Figura 2.2. *Ejemplo de arquitectura basada en componentes*

2.1.1.3 Arquitectura en capas

El estilo arquitectónico basado en capas (Figura 2.3) se enfoca en agrupar componentes que tienen funcionalidad similar en distintas capas que son apiladas verticalmente una encima de la otra donde las funcionalidades de cada capa está relacionada a través del objetivo común de la capa [Microsoft Patterns & Practices Team., 2009]. Este estilo ayuda a estructurar aplicaciones que pueden ser

descompuestas en grupos de subtareas en las cuales cada grupo es un nivel de abstracción particular [Buschmann, F. et al, 1996].

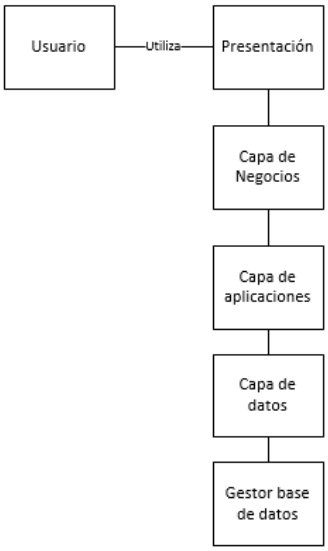

Figura 2.3. *Ejemplo de arquitectura en capas*

Es importante destacar que las capas pueden estar, o no, desplegadas en el mismo nodo. Es decir, ocasionalmente, por cuestiones de costos, rendimiento o mantenimiento, es conveniente tener cada capa desplegada en distintos servidores. Por ejemplo, en una típica aplicación web que tenga una arquitectura como la de la figura 2.3, la capa de presentación (que contiene la interfaz de usuario) debería estar desplegada en un nodo distinto al de la capa de datos por cuestiones de seguridad y rendimiento.

Los principios comunes para el uso de arquitectura basada en capas son [Microsoft Patterns & Practices Team., 2009]:

- Abstracción: Cada capa abstrae a sus capas subyacentes mediante una interfaz bien definida que permite la comunicación a lo largo del sistema. Esto permite que los detalles de implementación no sean conocidos entre capas.
- Encapsulación: No se deben asumir tipos de datos, ni métodos ni propiedades ya que las capas se abstraen de esta complejidad.
- Capas con funciones claramente definidas: La separación de funcionalidades debe ser clara. Las capas superiores le envían comandos a las capas inferiores y esperan una respuesta.
- Alta cohesión: Cada componente debe tener una responsabilidad bien definida de forma tal que una funcionalidad no deba ser resuelta en más de un componente o capa.
- Reutilizable: Las capas inferiores no deben depender de las capas superiores para poder ser reutilizables en otros escenarios.

- Bajo acoplamiento: La comunicación entre capas está basada en la abstracción y eventos para otorgar bajo acoplamiento entre capas.

Este patrón aplica cuando en un sistema existente hay capas que pueden ser reutilizables en otras aplicaciones, se tiene un conjunto de aplicaciones que exponen funcionalidades necesarias para el resto del negocio o bien la aplicación es muy compleja y necesita refactorizarse en capas para que se pueda dividir mejor el mantenimiento en pequeños equipos.

2.1.1.4 Arquitectura Orientada a Objetos

Un sistema orientado a objetos (Figura 2.4) se constituye con objetos que interactúan y mantienen su propio estado local y ofrecen operaciones sobre dicho estado [Sommerville, I., 2011]. El patrón arquitectónico orientado a objetos se basa en la división de responsabilidades para una aplicación en un conjunto reutilizables y autónomos de objetos que contienen la información y el comportamiento necesario para satisfacer las necesidades de negocio. Los objetos se comunican a través de interfaces llamando métodos y enviando y recibiendo mensajes.

Los principios claves de este patrón son [Microsoft Patterns & Practices Team., 2009]:

- Abstracción: Esto permite reducir una operación compleja en una generalización que retiene las características básicas de la operación. Ejemplos de abstracción se puede ver en objetos que exponen métodos “obtener” y “asignar” para escribir información en sus propiedades en lugar de exponerlas abiertamente para que cualquier componente las modifique.
- Composición: Los objetos pueden ser compuestos por otros objetos. Por ejemplo la Universidad en el diagrama de la figura 2.4 está compuesta por departamentos.
- Herencia: El paradigma orientado a objetos admite el concepto de herencia mediante el cual un objeto hereda de otro objeto funcionalidad y propiedades. Esto facilita el mantenimiento del software ya que la modificación en clases padres son inmediatamente heredadas en clases hijas. Por ejemplo, la clase Persona y Estudiante.
- Polimorfismo: El polimorfismo es la capacidad del paradigma orientado a objetos de sobreescribir el comporamiento de determinado objeto en base a las operaciones que debe soportar.
- Desacoplamiento: Los objetos pueden ser desacoplados de quienes lo consumen mediante la utilización de interfaces que abstraen la funcionalidad de la implementación.

El estilo arquitectónico orientado a objetos se suele utilizar cuando existe un modelo de datos que representan objetos del mundo real (como el ejemplo de la figura 2.4).

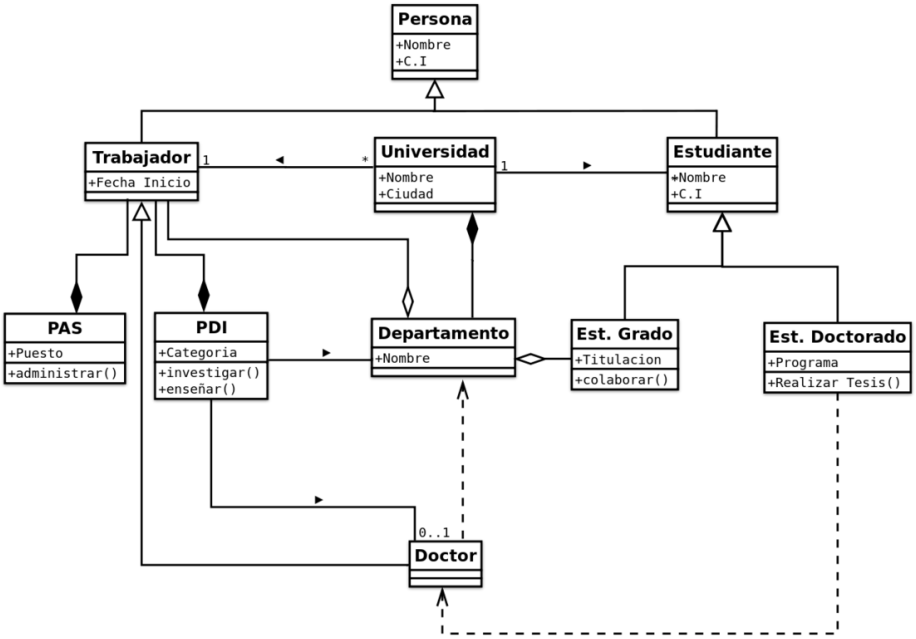

Figura 2.4. *Diagrama de clases para una Arquitectura Orientada a Objetos*
(De https://es.wikipedia.org/wiki/Diagrama_de_clases)

2.1.1.5 Arquitectura Orientada a Servicios

Las arquitecturas orientadas a servicios (SOA, del inglés Service Oriented Architecture) están basadas en las arquitecturas cliente-servidor y son una forma de desarrollar sistemas distribuidos en la que los componentes del sistema son servicios independientes y se ejecutan en computadoras distribuidas [Sommerville, I., 2011], aunque ocasionalmente pueden estar ejecutándose en el mismo servidor. Este tipo de arquitecturas permite ofrecer las funcionalidades del sistema como un conjunto de servicios pero también construir un sistema consumiendo funcionalidades a través de servicios ofrecidos por otros sistemas. Estos servicios están construidos generalmente en base a estándares y pueden ser invocados, publicados y descubiertos [Microsoft Patterns & Practices Team., 2009] (Figura 2.5).

Ocasionalmente suele confundirse entre SOA y Software como un servicio (SaaS, del inglés Software as a Service) y se utilizan los conceptos como sinónimo de forma errónea pues la diferencia entre SaaS y SOA es que el primero es un modelo de software de aplicación, mientras que el segundo es un modelo para la construcción de software y puede – o no – utilizar servicios SaaS [Merlino, H., 2014].

Las claves principales de SOA son las siguientes [Microsoft Patterns & Practices Team., 2009]:

- Los servicios son autónomos: Cada servicios es mantenido, desarrollado, desplegado y versionado de manera independiente.
- Los servicios son distribuibles: Pueden ser alojados en cualquier parte de una red, local o remotamente en tanto y en cuanto la red soporte los protocolos de comunicación requeridos.
- Los servicios están con bajo acoplamiento: Cada servicio es independiente de los demás y pueden ser reemplazados sin tener impacto en aplicaciones que lo consumen en tanto y en cuanto la interfaz con el servicio siga siendo compatible.
- Los servicios comparten esquemas y contratos, no clases: Cuando se comunican, los servicios exponen esquemas y contratos en lugar de clases internas.
- La compatibilidad está basada en una política: Esta política define características como transporte, protocolo y seguridad.

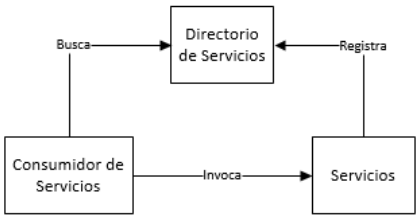

Figura 2.5. *Arquitectura Orientada a Servicios*

El estilo arquitectónico orientado a servicios se suele utilizar cuando en el sistema ya existen servicios que se pueden reutilizar o bien se planea consumir servicios de terceros. Otra variante es cuando se quiere construir un software que aglomera un conjunto de servicios en una interfaz de usuario o bien se está construyendo algo del tipo SaaS.

2.1.1.6 Modelo Vista Controlador

El patrón de diseño Modelo Vista Controlador (MVC, del inglés Model-View-Controller), a veces vinculado con el patrón arquitectónico de tres capas, separa presentación e interacción de los datos del sistema (Figura 2.6). Básicamente el software se estructura en tres componentes: El modelo maneja los datos del sistema y sus operaciones, la vista define y gestiona como se presentan esos datos al usuario y el controlador dirige la interacción del usuario con la interfaz de usuario mediante el manejo de botones, por ejemplo [Sommerville, I., 2011].

Buschmann, F. et al [1996] sugieren que la aplicación de este patrón resulta en la separación del software en tres áreas: entrada, procesamiento y salida. Y sostienen que el modelo es independiente de las representaciones que pueda tener la salida (en una interfaz de usuario, por ejemplo) como así

también de la entrada. La vista se encarga de mostrar la información al usuario al obtener los datos del modelo, incluso pueden haber múltiples vistas representando el mismo modelo. Los controladores (cada vista posee uno) reciben una entrada que típicamente son eventos de teclado o mouse y son traducidos en peticiones que se envían al modelo o la vista para que se pueda procesar la necesidad del usuario.

Este patrón se suele utilizar cuando existes múltiples formas de ver e interactuar con los datos, aunque también se utiliza cuando los requerimientos relativos a la interfaz de usuario aún son poco claros [Sommerville, I., 2011]. Esto es porque la separación del modelo de la vista y el controlador permite que existan múltiples vistas del mismo modelo.

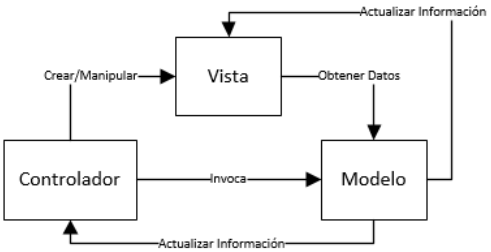

Figura 2.6. *Modelo Vista Controlador*

Para mantener refrescados los datos del modelo en todas las vistas, se implementa un patrón de diseño “Observer” u Observador mediante el cual la vista se suscribe al modelo y en cuanto se detecta alguna modificación, el modelo envía un mensaje a la vista para que refresque su contenido. De esta manera se pueden cambiar diversos módulos sin causar impactos grandes en el resto del sistema.

2.1.1.7 Fachada de Aplicación

El patrón Fachada de Aplicación o Application Facade es utilizado frecuentemente a la hora de diseñar el software pero más cercano a la etapa de implementación. La idea de este patrón es proporcionar una interfaz unificada para un conjunto de interfaces de un sistema, haciendo que los subsistemas sean más fáciles de utilizar. Por ejemplo, en la figura 2.7 se muestra como el patrón Fachada de Aplicación es aplicado en un sistema de tienda en línea para abstraer a los clientes el consumo de servicios ofrecidos por otros componentes como el gestor de stock, el carrito de compras o el gestor de pagos. El módulo de fachada en este caso es el componente central que se llama Tienda Online. Es importante observar que sin la existencia de este componente, es decir, sin

este patrón, los clientes deberían tener acceso directamente a los componentes de la derecha. Si bien esto puede efectuarse, tiene como desventaja que los clientes no quedan protegidos respecto cambios en los componentes del subsistema, no sólo de funcionalidad sino también de interfaz.

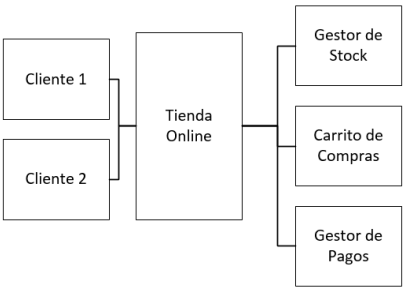

Figura 2.7. *Fachada de aplicación*

2.1.2. Modelos de Representación para Arquitecturas de Software

De la misma forma que en la arquitectura existen modelos de representación como el maquetado de un edificio o su representación en un plano XY para la documentación o comunicación de un diseño, en la ingeniería del software también se necesitan formas de modelar una arquitectura de software.

Una herramienta que se puede utilizar para el modelado de sistemas, y por ende arquitecturas de software, es el Lenguaje de Modelado Unificado [Fowler, M., 2004], un conjunto de diagramas que proveen a los arquitectos e ingenieros de software con herramientas para analizar, diseñar e implementar software, aunque también está contemplado su uso en otras áreas de negocio similares. UML es aceptado universalmente como el enfoque estándar al desarrollo de modelos de sistemas de software y está compuesto por 13 diferentes tipos de diagramas, aunque según estudios, la mayoría de los usuarios de UML sólo consideran 5 tipos de diagramas [Sommerville, I., 2011]:

- Diagramas de Actividad: que muestran las actividades incluidas en un proceso. En la figura 2.8 se muestra el diagrama de actividad para un sistema de tickets en el cual el pasajero solicita la reserva para que luego el distribuidor busque disponibilidad mediante el sistema de la aerolínea. Luego, el distribuidor ofrece vuelos disponibles en función de fechas y preferencias de precios. El pasajero selecciona el vuelo y mientras el distribuidor reserva un asiento, el pasajero paga. Por último, el ticket es emitido.
- Diagramas de caso de uso: que exponen las interacciones entre un sistema y su entorno. En la figura 2.9 se muestra el diagrama de actividad para un sistema de tickets en el cual el pasajero tiene la posibilidad de solicitar la reserva, seleccionar el vuelo y luego pagar por el

mismo. El distribuidor, por otra parte, tiene el objetivo de ofrecer vuelos al pasajero y luego expender el ticket. La aerolínea por último, reserva el vuelo para el pasajero.

- Diagramas de secuencias: que muestran las interacciones entre los actores y el sistema, y entre los componentes del sistema. En la figura 2.10 se muestra el diagrama de actividad para un sistema de tickets en el cual el pasajero solicita la reserva para que luego el distribuidor busque disponibilidad mediante el sistema de la aerolínea. La aerolínea itera su lista de vuelos y revisa su disponibilidad. Luego, el distribuidor filtra los vuelos enviados por la aerolínea en función de los precios deseados por el pasajero.
- Diagramas de clase: que revelan las clases de objeto en el sistema y las asociaciones entre estas clases. En la figura 2.11 se muestra el diagrama de actividad para un sistema de tickets en el cual el pasajero tiene la posibilidad de solicitar la reserva, seleccionar el vuelo y luego pagar por el mismo. El distribuidor, por otra parte, tiene el objetivo de ofrecer vuelos al pasajero y luego expender el ticket. La aerolínea por último, reserva el vuelo para el pasajero.
- Diagramas de estado: que explican cómo reacciona el sistema frente a eventos internos y externos. Los diagramas de estado no son estudiados en este trabajo.

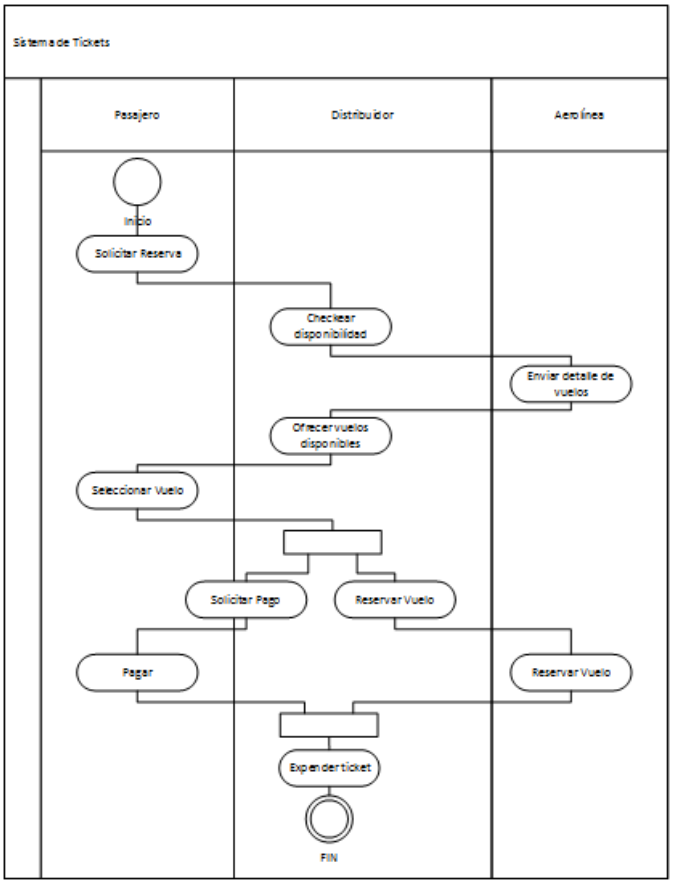

Figura 2.8. *Diagrama de actividad para un sistema de tickets.*

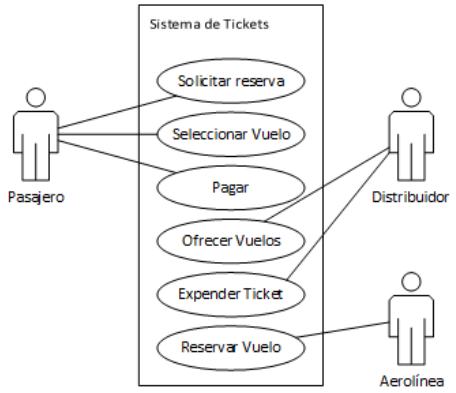

Figura 2.9. *Diagrama de casos de uso para un sistema de tickets.*

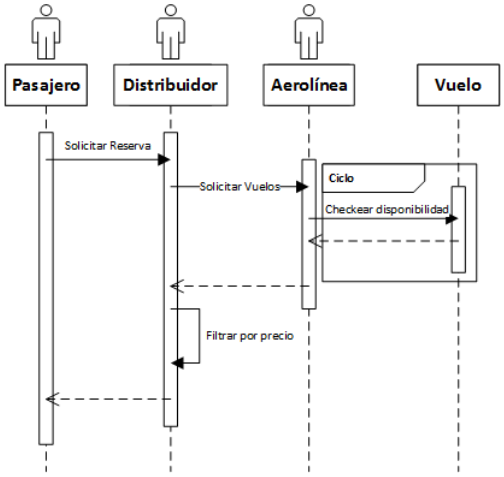

Figura 2.10. *Diagrama de secuencia para un sistema de tickets.*

Según la especificación de UML [Fowler, M., 2004] los diagramas se agrupan en tres categorías:

- Diagramas de Estructura: Que incluye diagrama de clases, de componentes y de despliegue. Un ejemplo de diagrama de despliegue puede ser el de la figura 12 en la cual el pasajero accede al sistema de tickets mediante una aplicación desplegada en algún nodo con dominio en el distribuidor. El distribuidor, por otra parte, accede a los datos de los vuelos mediante una aplicación desplegada en el dominio de la aerolínea. La aerolínea por último, posee dentro de su dominio una base de datos con la información necesaria para su negocio.
- Diagramas de Comportamiento: Que incluye diagramas de caso de uso, de actividad y de máquina de estado
- Diagramas de Interacción: Que incluye diagramas de secuencia, de comunicación, entre otros.

Además, estos diagramas pueden ser utilizados, según lo que quiera modelarse, con dos semánticas diferentes [Fowler, M., 2004]:

- Semántica estructural
- Semántica de comportamiento

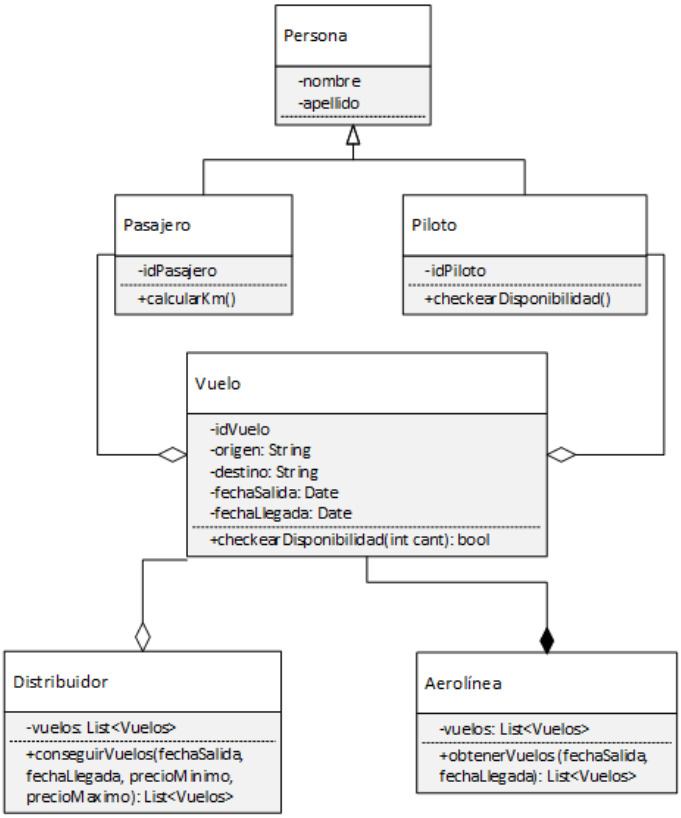

Figura 2.11. *Diagrama de clases de uso para un sistema de tickets.*

La semántica estructural, a veces conocida como semántica estática, hace referencia al significado conceptual del elemento en cuestión al definirlo a través de tipos de datos, relaciones y dependencias, y define la estructura estática de los objetos en un sistema. Los elementos en esta estructura representan conceptos de relevancia y pueden incluir implementaciones conceptuales del mundo real. Para esto, se suelen incluir clases, interfaces y componentes. Por otra parte, la semántica de comportamiento, a veces conocida como semántica dinámica, hace referencia a la comunicación que puede resultar entre los objetos estructurales a través del comportamiento. Es decir, muestran las relaciones en el comportamiento de las estructuras estáticas, demostrando cómo el objeto cambia a lo largo del tiempo. Estos tipos de semántica quedan ilustrados en la figura 2.13. Para más información se puede visitar [Fowler, M., 2004].

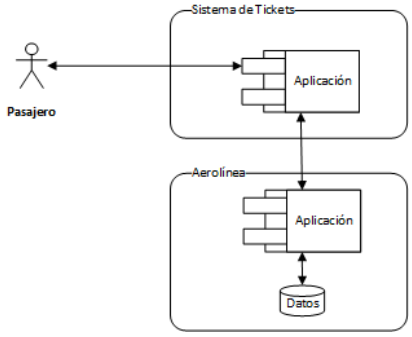

Figura 2.12. *Diagrama de despliegue.*

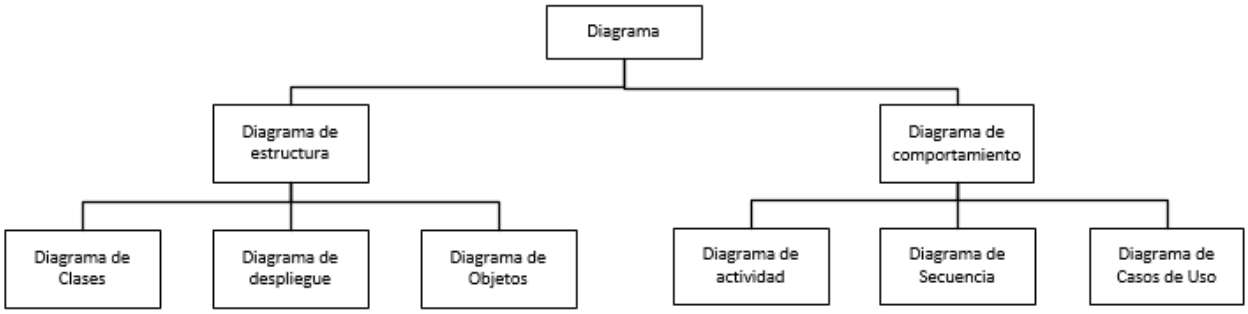

Figura 2.13. *Taxonomía de diagramas de estructura y comportamiento*

No obstante, comúnmente no alcanza un solo diagrama para representar la arquitectura entera. Para solucionar este problema, [Kruchten, P, 1995] propuso el modelo 4+1 que establece que para documentar una arquitectura hay que hacerlo desde cuatro diferentes puntos de vista y un quinto documento que integra las otras cuatro (Figura 2.14). Estas vistas son:

- Vista lógica, que ilustra la funcionalidad al usuario final
- Vista de procesamiento, que ilustra cuestiones de concurrencia y sincronización
- Vista física, que ilustra cuestiones de comunicación y hardware
- Vista de desarrollo, que ilustra detalles de la arquitectura para su codificación
- Vista de escenarios, que integra las otras cuatro vistas a través de casos de uso o escenarios

La vista lógica básicamente traduce requerimientos a diagramas de estructuras de datos cuya funcionalidad es un requerimiento. Indica las abstracciones clave en el sistema como objetos o clases [Sommerville, I., 2011], típicamente a través de diagramas de clases.

La vista de procesos contempla requerimientos no funcionales como rendimiento y disponibilidad, aunque también puede contener cuestiones de comunicación como redes LAN y WAN, entre otros. Indica la forma en que el sistema interactúa con los procesos que lo componen en tiempo real y es útil para discutir requerimientos no funcionales tal como se mencionó previamente [Sommerville, I., 2011].

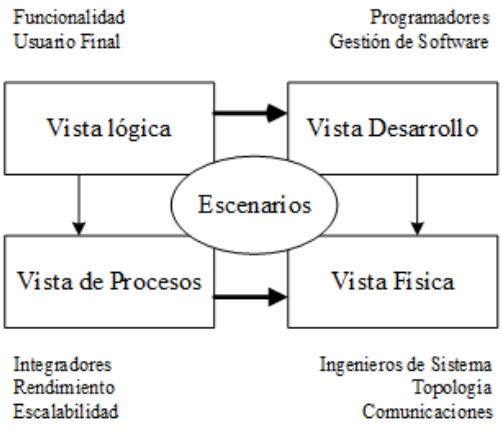

Figura 2.14. *Modelo 4+1*

La vista de desarrollo se refiere a la descomposición del sistema en subsistemas o módulos. Típicamente, se agrupan los módulos en paquetes y se hace referencia a ellos desde los subsistemas que los necesiten. Indica la descomposición del software en elementos que se implementen mediante un solo desarrollador o un equipo de desarrolladores, por lo que la utilidad de esta vista está dirigida hacia administradores y desarrolladores de software [Sommerville, I., 2011].

La física considera que haya diferentes configuraciones físicas en el sistema como topologías de red o distribución de los módulos del sistema. Esta vista expone el hardware del sistema y cómo los componentes de software se distribuyen a través de los procesadores en el sistema [Sommerville, I., 2011].

Los escenarios, por último, integran las cuatro vistas a través de diagramas de secuencia o de casos de uso. Los escenarios suelen mostrarse como una abstracción de los requerimientos más importantes y, a pesar de ser redundante con las otras cuatro vistas (y por eso "+1"), tiene dos principales propósitos [Kruchten, P, 1995]:

- Como un medio para el descubrimiento de elementos arquitectónicos durante el diseño arquitectónico
- Como un medio de validación e ilustración después de que el diseño arquitectónico ha finalizado.

2.2. INTERNET

Internet se ha convertido en un recurso casi esencial para la vida cotidiana del ser humano. No solo permite la comunicación a gran velocidad y bajo costo, también es utilizada a nivel gubernamental y organizacional en múltiples instituciones para intercambiar información en tiempo real. De hecho Internet surgió cuando algunas universidades lograron interconectar sus computadoras para compartir información. El desafío que tuvieron en ese entonces no era conectar las computadoras a una red si no conectar sus redes entre sí, definiendo así a internet como una red de redes.

A medida que más redes se pudieron incorporar a internet, más se facilitó el desarrollo de software distribuido, que es un conjunto de computadoras o aplicaciones independientes que aparece ante sus usuarios como un sistema consistente y único y por lo general tiene un modelo o paradigma único que se presenta a los usuarios. Con frecuencia, una capa de software que se ejecuta sobre el sistema operativo, denominada middleware, es la responsable de implementar este modelo [Tanenbaum, A. 2003]. El middleware es una capa de software que, como su nombre lo indica, es un software que se ubica entre dos capas o nodos y cumple una función particular. Por lo general este middleware no es

mostrado al usuario y cumple una función de integración entre dos aplicaciones o sistemas que necesitan convivir en un contexto heterogéneo como un sistema homogéneo.

Un ejemplo muy conocido de un sistema distribuido es World Wide Web (WWW), que no es ni más ni menos que un sistema que distribuye documentos de hipertexto. En sus comienzos, la WWW no era más que un conjunto de documentos estáticos pero con el crecimiento de internet, intranets, extranets y la misma WWW, el impacto que tuvo este sistema en los negocios, comercio, industria, educación, instituciones gubernamentales y en la vida cotidiana en general ha provocado que muchos de los sistemas distribuidos migren a ambientes web [Murugesan, S. et al, 2000], lo que a veces también se conoce como software como servicio (SaaS, del inglés Software as a Service) [Sommerville, I., 2011].

SaaS es un modelo de entrega de software en el que los datos y el software están alojados en servidores de Internet, comúnmente referenciados como la nube [Merlino, H., 2014]. Según [Laplante, 2008], SaaS es también a veces conocido como "software de suscripción" dado que separa conceptualmente la propiedad del software del usuario, es decir, el propietario del software es un proveedor que aloja el software y deja que el usuario lo consuma bajo demanda a través de alguna arquitectura cliente servidor a través de alguna red.

[Merlino, H., 2014] en su tesis doctoral comenta que la evolución del software hacia SaaS se dio gracias a la evolución de la informática y la masificación en el uso de Internet, tal como se mencionó anteriormente. Además, menciona que sobre los Centros de Datos (Data Center) se construido "un conjunto de capas tecnológicas las cuales interactúan entre sí" y que estas capas han dado lugar al SaaS. Estas capas son [Merlino, H., 2014]:

1. Infraestructura como un servicio (IaaS, del inglés Infraestructure as a Service)
2. Plataforma como un servicio (PaaS, del inglés Platform as a Service)
3. Software como un servicio (SaaS, del inglés Software as a Service)
4. Arquitectura orientada a servicios (SOA, del inglés Software Oriented Architecture)

El objetivo de IaaS es proveer al usuario de recursos computacionales (A veces físicos pero mayormente virtuales) bajo demanda. Estos recursos son ofrecidos mediante servicios y se dice que estos servicios son tangibles ya que representan equipos e infraestructura para el funcionamiento de un centro de datos [Merlino, H., 2014].

La capa PaaS ya no tiene como objetivo ofrecer recursos básicos sino herramientas de programación y/o plataformas de desarrollo, como Microsoft Azure o Force.com [Merlino, H., 2014]. En esta capa se encuentran sistemas operativos y servidores de bases de datos y se pierde control sobre los sistemas operativos, redes y demás. En este nivel aparece también lo que se conoce como iPaaS (Del inglés, Integration Platform as a Service o Plataforma de Integración como

servicio), cuyo objetivo es proveer al usuario una plataforma en la cual pueda construir software de integración para SaaS.

A diferencia de PaaS que está orientada al desarrollador, SaaS es un servicio que se ofrece al usuario final. SaaS es el eslabón de más alto nivel en el modelo de cloud computing y es un conjunto de servicios de aplicaciones que pueden ser accedidos a través de una red directamente a los equipos de los usuarios [Merlino, H., 2014].

En la figura 2.15 quedan ilustradas las diferencias entre los tres servicios ofrecidos en cloud computing y los recursos que ofrece el proveedor [Merlino, H., 2014]. Tal como se mencionó previamente, IaaS ofrece la infraestructura, es decir, los recursos computacionales desde el punto de vista de hardware y cuestiones de comunicación mientras que el usuario debe encargarse de configurar la plataforma, o sea sistemas operativos y aplicativos necesarios para que las aplicaciones se puedan ejecutar, y por último el software en sí como último eslabón. Por parte de PaaS, se le otorga al usuario además de la infraestructura una plataforma ya lista para poder desplegar aplicaciones en la misma y por último SaaS que ofrece todos los servicios, siendo el usuario final el objetivo de estos servicios que lo único que debe hacer es consumir el software que se le provee.

Figura 2.15. *Modelo de Cloud Computing [Merlino, H., 2014]*

Uno de los problemas que trajo esta migración hacia SaaS era que el acceso era exclusivamente a través de un navegador web y no era práctico el acceso directo a la información por parte de otros programas [Sommerville, I., 2011]. Para solucionar este problema se recurre a SOA que se basa en servicios para exponer las funcionalidades e información que rondan en un negocio particular. Si bien ni SaaS ni SOA requieren servicios web, son la mejor opción para implementarlos hasta el momento [Laplante, P., 2008]. El hecho de que sean la mejor opción es porque existen estándares y estilos que fueron ampliamente aceptados por la comunidad informática.

Un estilo arquitectónico para implementar los servicios web es REST (Representational State Transfer, es decir, transferencia de estado representacional) [Fielding. R, 2000] que trabaja con identificadores para sus recursos (URI o identificador universal de recurso) y se comunica mediante el protocolo HTTP. A través de los métodos que este protocolo expone, se entiende qué operación

realizar (Básicamente utiliza cuatro métodos, POST para crear, GET para leer, PUT para actualizar y DELETE para borrar). Cabe mencionar, que existe una notación comúnmente empleada para el intercambio de información entre agentes en los servicios web REST: JSON. JSON es un formato liviano para el intercambio de información y sus siglas significan JavaScript Object Notation, y está construido básicamente por dos estructuras [Introducing JSON, n.d.]:

- Una colección de pares nombre-valor
- Una lista ordenada de valores

Por ejemplo, un conjunto de empleados que tengan como atributos nombre y apellido, quedaría representado en un JSON de la siguiente manera:

```
{"empleados":[
   {"nombre":"Charly", "apellido":"García"},
   {" nombre ":"Ariel", " apellido ":"Segura"},
   {" nombre ":"Steve", " apellido ":"Jobs"}
]}
```

Como se puede observar, empleados es una lista (por la notación []) y cada empleado, es decir, cada ítem de la lista, es un objeto empleado cuyos atributos son “nombre” y “apellido”.

Si bien no existe un estándar que especifique cómo debe implementarse un servicio web REST, sí existe una especificación que define el diseño que un servicio web REST debe seguir: RAML [RAML 1.0]. RAML es un lenguaje comprensible por el humano que junta los principios de lo que se espera por parte de un servicio web REST. A continuación se muestra un ejemplo de cómo utilizar RAML para definir un servicio web que expone canciones, artistas y álbums [RAML, 2015]:

```
/canciones
  get
  post
  /{idCancion}
    get
    /file-content
      get
      post
/artistas
  get
  post
    /{idArtista}
      get
      /albums
        get
/albums
```

```
get
post
  /{idAlbum}
    get
    /canciones
      get
```

Otra alternativa para la construcción de servicios web radica en el uso de tecnologías XML. XML proviene del inglés y significa Lenguaje de Marcas Extensible (Extensible Markup Language) y es utilizado para almacenar datos de forma similar a JSON. El ejemplo de los empleados en XML queda de la siguiente manera:

```
<empleados>
   <empleado>
      <nombre>Charly</ nombre > <apellido>García</ apellido >
   </ empleado >
   < empleado >
      < nombre >Ariel</ nombre > < apellido >Segura</ apellido >
   </ empleado >
   < empleado >
      < nombre >Steve</ nombre > < apellido >Jobs</ apellido >
   </ empleado >
</empleados>
```

En [Sommerville, I., 2011] se describen dos estándares claves de SOA que son relevantes para este trabajo:

1. SOAP. Un estándar de intercambio de mensajes que soporta la comunicación entre servicios. Define el componente esencial y opcional de los mensajes transmitidos entre los servicios. La estructura de un mensaje SOAP es la siguiente [W3Schools]:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header>
...
</soap:Header>

<soap:Body>
...
```

```
  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

2. WSDL. El Lenguaje de Definición de Servicio Web (Web Service Definition Language) es un estándar para la definición de interfaz de servicio. Establece cómo deben definirse las operaciones de servicios y los enlaces de servicio. Un WSDL tiene una forma como la siguiente [W3Schools]:

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

2.3. ¿QUÉ ES UN OBJETO INTELIGENTE?

Lo primero que se piensa cuando se habla de una computadora es la típica laptop para consultar el correo electrónico, escribir una tesis o trabajar. Sin embargo, vivimos rodeados de computadoras. Las computadoras se encuentran presentes en gran variedad de elementos cotidianos como un horno microondas, un acondicionador de aire, un automóvil o un portón automático. A diferencia de las computadoras convencionales, interactúan directamente con el hardware y su software (llamado software embebido) debe reaccionar a eventos generados por el hardware y emitir a menudo señales de control en respuesta a tales eventos [Sommerville, I. 2011]. Estos sistemas embebidos son computadoras pequeñas, de bajos recursos y autónomas que realizan el mismo trabajo infinitamente y están dotadas de un microcontrolador y dispositivos de entrada y salida. Estos sistemas embebidos han evolucionado a lo largo de los años y han adquirido capacidades no sólo capacidades de cómputo, sino de conectividad, similares a las de computadoras hogareñas o de oficina. Muchos dispositivos hoy tienen la capacidad de conectarse a internet y consumir un servicio web como cualquier software que se ejecuta en una laptop.

En este contexto, surge el concepto de objeto inteligente que trata de un sistema embebido que puede entender y reaccionar a lo que está ocurriendo en su alrededor [Kortuem, G, et al., 2010] y que no solo se interesa por la interacción con el usuario sino también de la interacción con otros objetos inteligentes o incluso otro software, es decir, contempla la interacción con el mundo físico y virtual al mismo tiempo. Por ejemplo, dispositivos de medición de gases, relojes, semáforos, electrodomésticos de aire acondicionado, sistemas de control de riego, dispositivos de transporte público no tripulado, entre otros, pueden ser integrados con cualquier otro dispositivo o sistema externo a través de internet. Para esto, se necesita que los objetos inteligentes:

- Tengan capacidad de obtener información sobre su entorno a través de la medición y control de sensores y actuadores
- Tengan una configuración que se pueda adaptar a cada necesidad como pueden ser cuestiones de conectividad o seguridad
- Tener un módulo de gestión de seguridad y credenciales
- Permitir el control automático de tareas de forma tal que se programe el objeto inteligente para realizar tareas que hoy se realizan manualmente
- Puedan comunicarse con otros sistemas como servicios web o servicios en la nube

En [Guinard, D. 2011] se acude, sin embargo, al concepto de Smart gateways o puertas de enlace inteligentes, dado que no todos los sistemas embebidos tienen capacidad de cómputo suficiente para soportar protocolos como HTTP o TCP/IP. De esta manera, se puede otorgar capacidad de conectividad a aquellos dispositivos que por una limitación de recursos no puedan soportar los protocolos requeridos. De hecho [Kortuem, G, et al., 2010] sugiere que uno de los puntos claves es determinar dónde desplegar el software relacionado con un dispositivo, dando las siguientes opciones:

1. En el objeto inteligente: Cuando no hay gran demanda de cómputo como servicios web livianos que se puedan resolver en algunos bytes.
2. En gateways: Cuando los dispositivos no son lo suficientemente poderosos para ejecutar el software por cuenta propia.
3. En la nube: Esta solución mejora la disponibilidad de los servicios pero puede disminuir el rendimiento en términos de latencia y flujo de datos.

2.4. INTERNET DE LAS COSAS

Anteriormente se explicó brevemente la evolución de Internet a lo largo de la historia. Partiendo de un experimento entre universidades, pasando por el poder compartir información a distancia y

llegando a las aplicaciones web, SaaS y SOA, donde las personas no técnicas pueden compartir información o consumir diversos servicios como redes sociales. Es decir, primero se conectaron universidades, luego instituciones gubernamentales y empresas, luego personas… ¿y las cosas? ¿Y la heladera, el microondas, el acondicionador de aire o máquinas industriales? Y tantas cosas que nos rodean cotidianamente.

El término Internet de las Cosas (Internet of Things o IOT) se refiere a una evolución de lo que hoy se conoce como Internet para convertirla en una red interconectada de objetos que no solo recolectan información del ambiente e interactúa con el mundo físico, sino que usa estándares de Internet para proveer servicios de información [Gubbi, J et al, 2013]. Esto implica, además de tener determinada infraestructura, servicios y demás, la construcción de estos objetos inteligentes en sistemas embebidos de forma tal que tengan la conectividad necesaria para construir un ecosistema en donde todos los artefactos (Tanto hardware como software) estén interconectados [Holler, J., 2014]. De hecho, hablando de conectividad entre sistemas embebidos, vale la pena aclarar la diferencia entre dos conceptos que normalmente se usan como sinónimos: M2M (Machine To Machine) e IOT.

Según [Holler, J., 2014], M2M se refiere a soluciones que permiten la comunicación entre dispositivos del mismo tipo y aplicaciones específicas permitiendo a los usuarios finales capturar información sobre los eventos que suceden en el ambiente. Sin embargo, no es común que las soluciones M2M cubran escenarios en donde se necesite conexión de los dispositivos directamente a Internet. De hecho, el típico escenario en donde se aplica M2M es como el que se observa en la figura 2.16 en donde el dispositivo M2M se utiliza para monitorear o controlar un ambiente y se comunica con los servidores de aplicación mediante una red local (LAN) o amplia (WAN). Luego, el servicio funciona como una capa de abstracción para la aplicación que finalmente integra el dispositivo en el proceso de negocio.

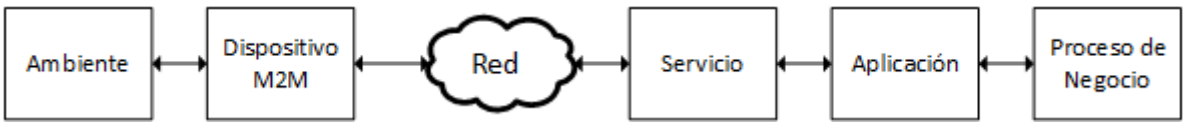

Figura 2.16. *Modelo de una solución M2M [Holler, J., 2014]*

IOT, sin embargo, además de contemplar los alcances de M2M, se refiere también a la conexión de estos dispositivos a Internet de manera que puedan conectarse con otros objetos, comunicarse e interactuar de la misma forma que las personas lo hacen hoy a través de la Web [Holler, J., 2014] y, por otra parte, dónde son aplicadas estas tecnologías. De hecho, la principal fortaleza del concepto

de IOT es el impacto que tendrá en los aspectos de la vida cotidiana en los potenciales usuarios [Atzori, L. et al, 2010]. Algunas aplicaciones de IOT son:

- Transporte y logística [Atzori, L. et al, 2010]: Vehículos, trenes y colectivos tienen cada vez más sensores, actuadores y capacidad de procesamiento. Esta información en complemento con la que pueda suministrar las rutas y los servicios de monitoreo de tránsito pueden generar información en tiempo real muy valiosa.
- Cuidado de la salud [Atzori, L. et al, 2010], [Misra, P. et al, 2015]: Las aplicaciones más comunes son el seguimiento de stock de objetos o personas en hospitales, identificación de personas (especialmente si han sufrido algún accidente) y la recolección automática de datos.
- Ambientes inteligentes [Atzori, L. et al, 2010], [Misra, P. et al, 2015]: La domótica y la automatización de fábricas permite un manejo más eficiente de recursos energéticos y económicos.

A pesar de estas aplicaciones Atzori y sus colegas mencionan algunos problemas como estandarización, direccionamiento y cuestiones de red, seguridad y privacidad que, si bien ya existen algunos aportes en la mayoría de las áreas, todavía quedan muchas problemáticas sin resolver. Sin embargo, a pesar de estas cuestiones, este concepto de conectar las cosas que nos rodean a Internet está volviéndose cada vez más fuerte y se espera que 50 mil millones de dispositivos estén conectados a internet en 2020 [Evans, D., 2012]. De hecho en la industria cada empresa está intentando imponer su modelo de IOT, como por ejemplo Cisco con “Internet of Everything” [Evans, D., 2012].

En el mercado hay ya varios dispositivos que apuntan al concepto de IOT que fueron lanzados posteriormente a la fecha de comienzo de este trabajo, algunos de los más relevantes son:

- Samsung ARTIK [ARTIK IoT]: Un dispositivo pensado para desarrolladores para que puedan construir su solución IOT partiendo desde un hardware base.
- Gateways Intel [Intel Gateways]: Un dispositivo pensado para actuar de Gateway entre varios dispositivos y algún servicio en la nube.
- Dispositivos SmartCitizen [Smart Citizen Dispositivos]: Dispositivos que se pueden conectar a un servicio en la nube, una aplicación móvil o de escritorio. Además tienen una API REST.

A continuación, se presentan las arquitecturas más relevantes sobre IOT que han sido consideradas en el desarrollo de este trabajo (sección 2.4.1)

2.4.1. Arquitecturas de Software en Internet de las Cosas

Tal como se mencionó con anterioridad, uno de los problemas que enfrenta IOT es la estandarización de protocolos de comunicación, políticas de seguridad y privacidad, y también de arquitecturas de software.

En [Atzori, L. et al, 2010] se propone una arquitectura orientada a servicios (Figura 2.17) para un middleware para IOT ya que según los autores es fundamental abstraer a los desarrolladores de IOT de cierta infraestructura de IOT. El hecho de haber elegido SOA, explican los autores, es porque la adopción de este tipo de arquitecturas permite descomponer sistemas complejos y monolíticos en aplicaciones consistentes de un ecosistema de componentes simples y bien definidos.

Esta arquitectura está compuesta básicamente por cinco capas:

- Aplicaciones
- Composición de Servicios
- Gestión de Servicios
- Abstracción de Objetos
- Objetos

Las aplicaciones (Applications) están en el extremo más alto exponiendo las funcionalidades del sistema a los usuarios finales. Esta capa no se considera en sí dentro del middleware pero consume las funcionalidades del mismo. A través del uso de servicios web estándares se espera que las aplicaciones se integren perfectamente a este software [Atzori, L. et al, 2010].

La capa de servicio es la típica capa de servicios que encapsula servicios de capas inferiores, esta capa no conoce la existencia de dispositivos. Es en realidad una capa que según los autores aparece en todas las arquitecturas SOA y provee funcionalidades para la composición de servicios ofrecidos por los objetos en servicios más complejos que brinden una funcionalidad más completa. En esta capa no existe incluso la noción de “dispositivos” y sólo son visibles los servicios que estos ofrecen.

La capa de gestión de servicio (Service Management) provee las funciones principales que se esperan de cada objeto en el contexto de IOT. Por ejemplo, descubrimiento del objeto, monitoreo de su estado y configuración de su servicio. También se incluye un repositorio de servicios para conocer cuáles son los servicios que están asociados a cada objeto en la red [Atzori, L. et al, 2010].

La abstracción de los objetos (Object abstraction) es simplemente una capa cuyo objetivo es lograr que los servicios sean agnósticos a cada tipo de dispositivo. El ecosistema IOT está repleto de objetos diferentes que tienen distintos objetivos y funcionalidades, y hasta incluso hablan distintos dialectos o protocolos. Es por eso que se necesita esta capa de abstracción. La capa transversal de confiabilidad, privacidad y gestión de la seguridad (Management of trust, privacy and security)

efectúa tareas de autenticación y autorización. [Atzori, L. et al, 2010] argumentan la existencia de esta capa debido a que la comunicación automática de objetos en el entorno personal representa un daño potencial ya que los dispositivos podrían llegar a estar compartiendo información confidencial sin la supervisión de un humano. Es por eso que el middleware necesita de esta funcionalidad, idealmente, afectando a la arquitectura entera.

Por último, los objetos son los dispositivos de los cuales se extrae información.

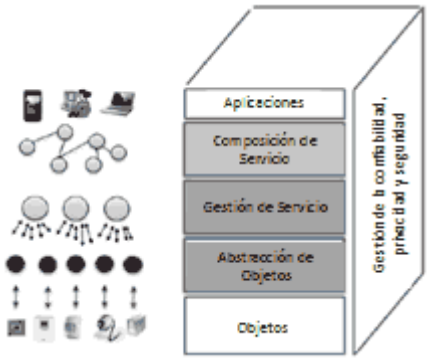

Figura 2.17. *Arquitectura SOA para Middleware en IOT [Atzori, L. et al, 2010]*

En [Misra, P. et al, 2015] se propone una arquitectura modular, escalable que soporte agregar o eliminar funcionalidades dependiendo de los requerimientos. Esta arquitectura de alto nivel (Figura 2.18) está compuesta por cuatro capas:

- Espacio físico y/o virtual
- Sensor como un servicio
- Gestión de datos
- Estadísticas/Análisis de datos

La capa más baja es la de espacio físico o virtual que es una colección de dispositivos compuestos por sensores o actuadores que en definitiva generan o consumen información del contexto en donde se encuentran. Es interesante el concepto que sugieren los autores de esta arquitectura para esta capa respecto los dispositivos ya que no sólo consideran los dispositivos físicos sino también virtuales, representando a aquellos a los que se accede mediante algún servicio. Respecto al tema aclaran que a pesar de que los dispositivos físicos contienen una interfaz para el mundo físico, los dispositivos virtuales son un tanto más diversos y consisten en censar entidades que pueden ser humanas como blogs, juegos de computadora en línea, calendarios electrónicos o redes sociales. La capa de sensor como un servicio representa una capa de abstracción para la entrada y salida de la capa de dispositivos. En esta capa también se exponen mecanismos para encender o apagar sensores o atributos específicos, cambiar su frecuencia de transmisión o los controles de calidad que el

dispositivo haga sobre sus mediciones. Los autores sugieren que la comunicación entre esta capa y las superiores sea a través de protocolos basados en REST.

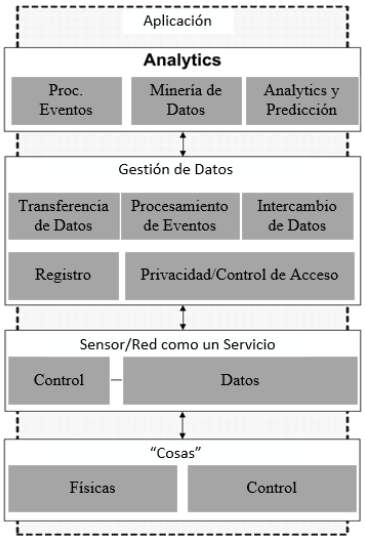

Figura 2.18. *Arquitectura de alto nivel para IOT [Misra, P. et al, 2015]*

La capa de gestión de datos tiene como objetivo generar cierta semántica a partir de los servicios que se ofrecen, es decir, los encapsula para ofrecer servicios más completos. Básicamente recolecta datos de los servicios que ofrecen la capa de sensor como un servicio y lo agrega a su registro (Registry) de forma tal que pueda ser periódicamente consultado. El acceso a los datos por parte del registro se realiza utilizando servicios web estándares.

Por último, la capa de estadísticas y análisis de datos se utiliza para obtener estadísticas o prever comportamientos o eventos en el entorno.

En [IOT-A, 2013] se propone un modelo arquitectónico de referencia que provee vistas y perspectivas de diferentes aspectos arquitectónicos, que son de interés de los participantes en un proyecto IOT, mediante las cuales se pueden obtener arquitecturas concretas para escenarios concretos.

En su publicación se menciona que la mayor contribución la da el Modelo de Referencia que provee conceptos y definiciones sobre los cuales se pueden construir arquitecturas IOT. El modelo de referencia a su vez consiste en tres submodelos:

- Submodelo de Dominio (Figura 2.19)
- Submodelo de Información (Figura 2.20)
- Submodelo de Comunicación (No estudiado en este trabajo)

El Submodelo de Dominio, que define los conceptos desde el punto de vista de la información en IOT. Este Submodelo de Dominio es la base del modelo de referencia y sirve como soporte a la arquitectura de referencia para que todas las arquitecturas concretas hagan referencia a los mismos

conceptos, introduciendo los principales conceptos como Dispositivos, Servicios IOT, Entidades Virtuales y la relación entre estos conceptos, como por ejemplo la relación “Los servicios exponen recursos”. Este submodelo fue construido de una forma abstracta de forma tal que no dependa de las tecnologías que se vayan a utilizar ni de los escenarios de uso que se vayan a resolver dado que nada garantiza que en unos años los dispositivos de los que hablamos ahora sigan existiendo o sigan siendo útiles. Sin embargo, al abstraerse, el concepto de dispositivo va a seguir teniendo el mismo significado.

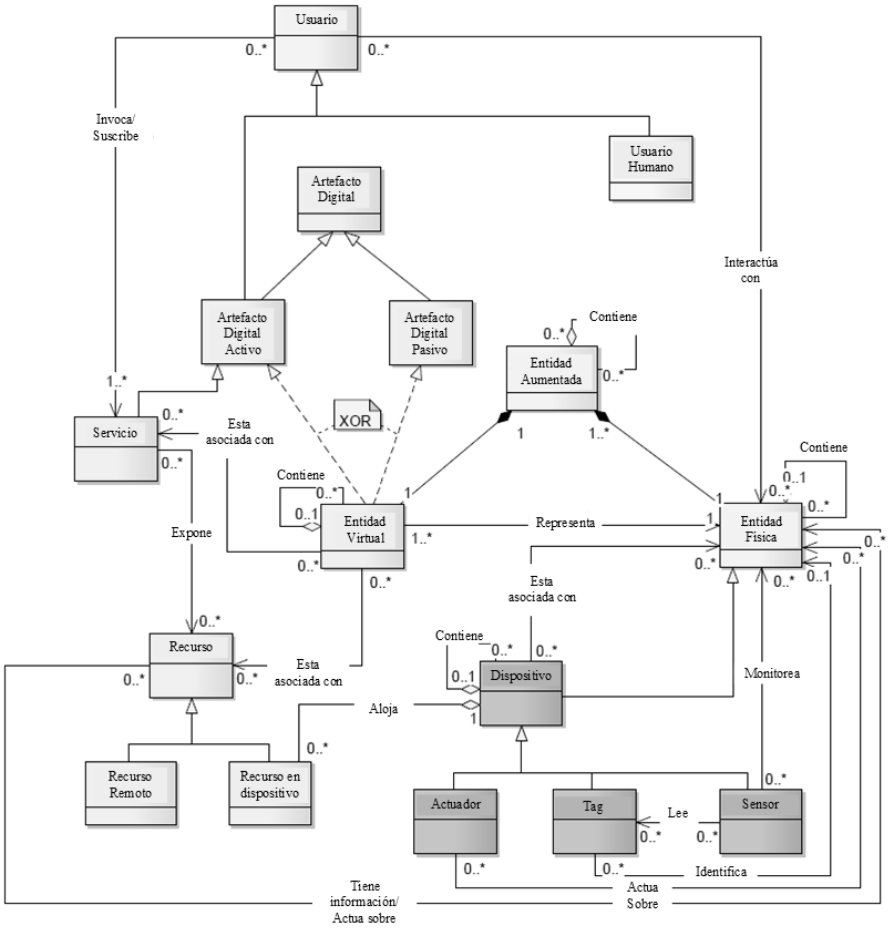

Figura 2.19. *Diagrama de clases para representar el Submodelo de Dominio para Internet de las Cosas [IOT-A, 2013]*

En el contexto de esta arquitectura se define como escenario clave aquel en el que el usuario interactúa con una entidad física en el mundo físico, definiendo así las dos claves para IOT, según estos autores. El usuario puede ser una persona pero también contemplan que sea un software, como por ejemplo un servicio o aplicación, que necesita interactuar con una entidad física. A diferencia del mundo físico en donde las interacciones normalmente suceden de forma directa, en IOT se contempla que estas interacciones sean remotas o mediadas por algún servicio que puede intercambiar información con la entidad física. Estas representaciones digitales de las entidades físicas reciben el nombre de entidades virtuales y hay varios ejemplos de las mismas como modelos

3D, avatars, objetos en el caso de la programación orientada a objetos y también cuentas en redes sociales pueden ser consideradas entidades virtuales.

En IOT las entidades virtuales tienen dos propiedades particulares:

- Son artefactos digitales y están asociadas a una sola entidad física, mientras que las entidades físicas pueden estar asociadas a más de una entidad virtual.
- Las entidades virtuales deberán ser representaciones sincronizadas de las propiedades de la entidad física de forma tal que si la entidad virtual sufre una modificación, tenga un impacto en la entidad física y viceversa.

Otro concepto importante dentro del Submodelo del Dominio es la entidad aumentada que es la encapsulación de una entidad física con una entidad virtual. Esto es lo que permite la comunicación entre una entidad física no sólo con una persona sino también con otro software, tal como se mencionó anteriormente. Esta entidad aumentada es lo que los autores de [IOT-A] consideran como "Cosa" dentro de internet de las cosas. El Submodelo de Información que en realidad surge a partir del Submodelo de Dominio y explica cómo se modela la información en IOT a través de una estructura abstracta que contiene relaciones y atributos, sin entrar en detalles de cómo se va a terminar representando esa información en los casos concretos. Por último, el Submodelo de Comunicación establece algunos lineamientos sobre cómo administrar diversos protocolos de comunicación.

Otro Modelo importante definido en [IOT-A, 2013] es el Modelo Funcional. Para explicar el concepto de Modelo Funcional primero hay que definir dos conceptos:

- Descomposición Funcional que hace referencia al proceso dividir componentes funcionales
- Componentes Funcionales que son los que arman la arquitectura de referencia

El objetivo principal de la descomposición funcional es por sobre todas las cosas utilizar la estrategia "divide y vencerás" para disminuir la complejidad de la solución en piezas más pequeñas y comprensibles. Uno de los resultados que produce la descomposición funcional es el modelo funcional que es una forma de representación abstracta de los grupos funcionales de componentes que integran la arquitectura de referencia.

En el modelo funcional se encuentran los siguientes grupos funcionales:

- Gestión de Procesamiento cubre los requerimientos de negocio respecto la posibilidad de construir servicios y aplicaciones sobre los servicios de IOT que se ofrecen
- Seguridad que cubre cuestiones de confiabilidad, seguridad y privacidad
- Gestión que cubre trasversalmente la interacción entre los demás grupos funcionales
- Organización de los servicios que actúa como un hub entre los demás grupos funcionales

- Entidad virtual y servicio IOT que incluyen funciones relacionadas con la obtención y el envío de datos desde y hacia los dispositivos, respectivamente. Ejemplos sobre este grupo funcional son: “Leer el valor del sensor 456” o “Encender el actuador 789”.

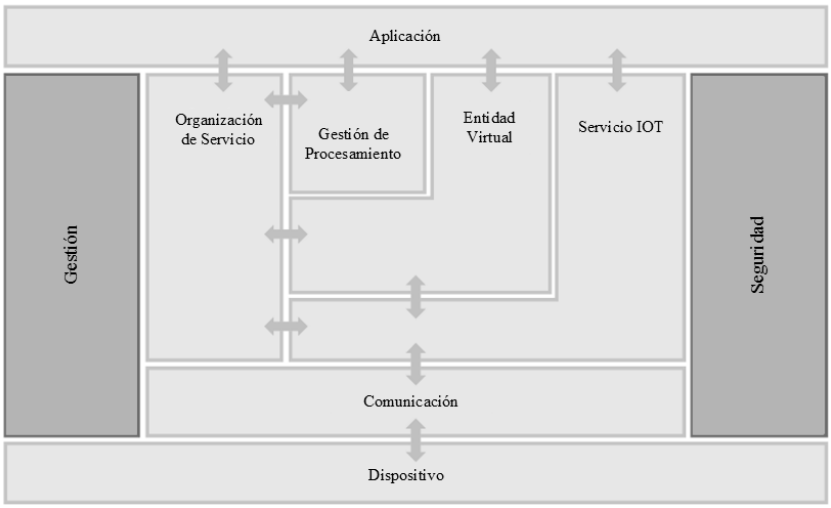

Figura 2.20. *Diagrama para representar el Modelo Funcional para Internet de las Cosas [IOT-A, 2013]*

En [Microsoft Patterns & Practices Team, 2009] se define un conjunto de componentes que están presentes en la mayoría de los artefactos software independientemente de los requerimientos que se tengan. Estos componentes se pueden ver en la figura 2.21 en donde se muestran de una forma abstracta y alto nivel cómo estos componentes se relacionan entre sí.

La capa de presentación contiene toda la funcionalidad relacionada con el usuario para administrar la relación sistema-usuario. Generalmente consiste en una serie de componentes que actúan de puente entre la capa de negocios y el usuario.

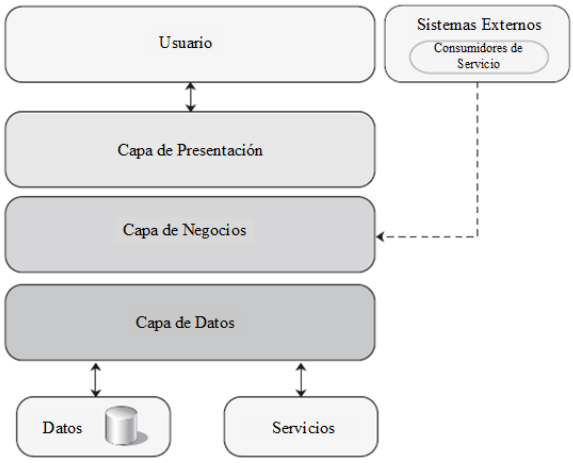

Figura 2.21. *Vista de alto nivel del modelo arquitectónico genérico presentado en [Microsoft Patterns & Practices Team, 2009]*

La capa de negocios implementa las funcionalidades centrales del sistema y las encapsula exponiendo sólo las funcionalidades necesarias para las capas superiores. Por lo general, tiene un conjunto de interfaces para que el resto de los componentes puedan consumir su información.

La capa de datos provee acceso a:

- Datos almacenados localmente
- Datos expuestos por sistemas externos dentro de la misma red

Además, al igual que la capa de negocios, provee interfaces para que el resto de los componentes puedan consumir su información.

En una vista más detallada de la arquitectura propuesta en [Microsoft Patterns & Practices Team, 2009] se presentan los subcomponentes que cada capa posee y se agregan la capa de servicios y las transversales (Figura 2.22).

En este escenario, los usuarios pueden acceder a la aplicación a través de la capa de aplicación que puede comunicarse directamente con la capa de negocios, o bien a través de la capa de servicios. La ventaja de este esquema es que mientras el usuario accede a través de la capa de presentación, el sistema puede exponer su información y funcionalidades a través de la capa de servicios para sistemas externos, soportando múltiples clientes y estimulando la reutilización de funcionalidades en varias aplicaciones.

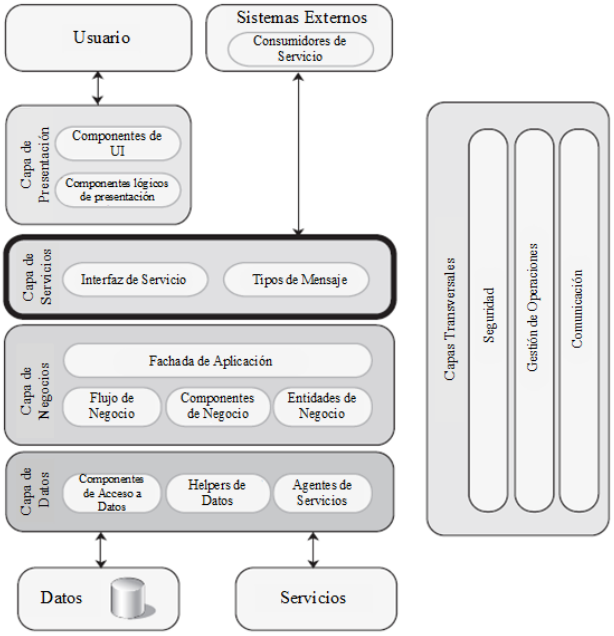

Figura 2.22. *Vista detallada del modelo arquitectónico genérico presentado en [Microsoft Patterns & Practices Team, 2009]*

Las capas transversales varían de un escenario a otro y deben ser identificadas para cada caso de uso. Este grupo de capas por lo general incluye funcionalidades como logging, caching,

validaciones, autenticación y manejo de errores. El hecho de identificar estas funcionalidades transversales es de extremada importancia dado que fomenta una mejor reusabilidad y mantenimiento, mientras que al mismo tiempo evita duplicar componentes software.

Las capas de esta vista detallada y sus subcomponentes son:

- Capa de Presentación:
    - Componentes de Interfaz de Usuario: Son los elementos visuales mediante los cuales se muestra o solicita información al usuario.
    - Componentes lógicos de presentación: Se refiere al componente software que define el comportamiento lógico de la aplicación de forma tal que sea independiente de la implementación específica de la interfaz de usuario.
- Capa de Negocios:
    - Application Facade o Fachada de aplicación: Es un componente opcional que provee una simplificación o abstracción a los componentes de la capa de negocios en sí. Esta capa suele combinar varias funcionalidades de las capas de negocios en una sola operación haciendo que sea más fácil utilizar esta capa. Además, reduce el grado de dependencia de las capas que consumen estas funcionalidades ya que no necesitan saber detalles de implementación.
    - Componentes lógicos de negocio: Estos componentes representan el modeo de datos que se encarga con la obtención, el procesamiento, la transformación y la gestión de los datos mediante la aplicación de reglas de negocio. Estos componentes lógicos de negocio pueden ser a su vez divididos en dos categorías: Componentes de flujo de negocio y Componentes de entidades de negocio. Los componentes de flujo de negocio son los que gestionan el flujo de datos en función de la funcionalidad a ofrecer. Es decir, muchas funcionalidades involucran la ejecución de varios pasos en un determinado orden. Esta capa se encarga de que los componentes de la capa de negocio trabajen en conjunto y de forma coordinada para alcanzar los objetivos de negocio. Los componentes de entidades de negocio, por otra parte, son los objetos o estructuras de datos que encapsulan la lógica y los datos necesarios para representar elementos del mundo real.
- Capa de Acceso a Datos:
    - Componentes de acceso a datos: Estos componentes abstraen la lógica requerida para acceder a los repositorios de datos, centralizando las funcionalidades troncales en un solo componente.

  - Agentes de Servicio: Cuando un componente de negocio necesita acceder a los datos cuyo proveedor es un servicio externo, puede resultar necesario implementar un componente que gestione la semántica de comunicación con ese servicio en particular.
- Capa de Servicios
  - Interfaces de servicio: Los servicios exponen una interfaz donde se envían todos los mensajes entrantes.
  - Tipos de Mensaje: Los servicios deben exponer no sólo las estructuras de datos y operaciones, sino también los tipos de datos y contratos de interfaz que definen la interfaz del servicio.

3. DESCRIPCIÓN DEL PROBLEMA

En este capítulo se identifica el problema de investigación de este trabajo final de licenciatura (sección 3.1), se caracteriza el problema abierto (sección 3.2) y se concluye con un sumario de investigación (sección 3.3).

3.1. IDENTIFICACIÓN DEL PROBLEMA DE INVESTIGACIÓN

En la sección anterior, han sido caracterizados los conceptos de IOT y sus distintas aplicaciones relativas a la sociedad como la salud, servicios de emergencia, transporte y domótica, así como también relativas a la industria y al mercado en general como servicios de retail, logística y ciudades inteligentes.

A los efectos de identificar el problema que se pretende resolver en el presente trabajo final de licenciatura, se debe empezar por identificar los inconvenientes que se presentan en la actualidad respecto las aplicaciones ya existentes y puestas en marcha. Actualmente, existe una tendencia de desarrollar soluciones para problemas particulares de internet de las cosas, con tipos específicos y acotados de dispositivos o una tecnología en particular [Atzori et al. 2010], lo cual termina en aplicaciones específicas con arquitecturas específicas con poco lugar para interactuar con otros sistemas [Internet of Things Architecture, 2013]. Lo cierto es que si bien se detecta esta problemática en varios trabajos ([Atzori et al. 2010]; [Carretero et al. 2013]; [IOT-A, 2013]; [Misra et al. 2015]), no se han identificado arquitecturas de software prácticas y concretas para la construcción de sistemas embebidos que puedan ser integrados al ecosistema de Internet de las Cosas. En [IOT-A, 2013], se propone un modelo arquitectónico de referencia, desde dispositivos físicos hasta el consumo y análisis de datos en las capas más superiores, mediante el cual a través de algunos requerimientos se puede obtener una arquitectura concreta. Sin embargo, si bien los autores identifican a los dispositivos como sistemas embebidos que interactúan entre el mundo digital y el físico, conectando entidades físicas del mundo real a internet, el modelo no contempla ni define qué arquitectura o qué componentes debe tener el software de un sistema embebido para que pueda formar parte de un ecosistema de IOT.

3.2. PROBLEMA ABIERTO

El problema abierto que se identifica en la presente sección, consiste en la necesidad de diseñar y construir una arquitectura de software que permita el desarrollo de objetos inteligentes basados en sistemas embebidos para su integración al ecosistema de IOT.
A pesar de la existencia de distintas arquitecturas de alto nivel como las presentadas en [Misra et al., 2015]; [Atzori el al., 2010]; [Gubbi, J., 2013] o [IOT-A, 2013], ninguna abarca específicamente la forma que debe tomar el software de un objeto inteligente para que se adapte al paradigma de internet de las cosas haciendo que quienes están dando sus primeros pasos en la construcción de objetos inteligentes tengan que definir su propia arquitectura para un problema en particular, sin tener un esquema de referencia en el cual basarse o poder reutilizar.

3.3. SUMARIO DE INVESTIGACIÓN

De lo expuesto precedentemente surgen las siguientes preguntas de investigación:

Pregunta 1: ¿Puede desarrollarse una arquitectura de software para sistemas embebidos que pueda utilizarse como referencia o punto de partida para el desarrollo de soluciones en internet de las cosas? En caso afirmativo: ¿Cuál es su estructura y qué elementos la componen?

Pregunta 2: ¿De existir tal arquitectura, es posible desarrollar vistas arquitectónicas que definan las relaciones entre sus componentes? De ser posible: ¿Cuáles?

Se proponen soluciones a los interrogantes planteados y su correspondiente validación en los próximos capítulos.

4. SOLUCION

En esta sección se presenta una Arquitectura de Software de Referencia para Objetos Inteligentes en Internet de las Cosas estructurada en cuatro partes: generalidades (sección 4.1), propuesta del modelo (sección 4.2), relación entre componentes (sección 4.3) y un ejemplo que muestra su aplicación (sección 4.4).

4.1. GENERALIDADES

En función del análisis realizado en el capítulo 3 correspondiente a la Descripción del Problema, se considera de interés citar nuevamente el problema abierto que se aborda en este trabajo final de licenciatura, recordando que el mismo se focaliza en el diseño de una arquitectura de referencia de software para objetos inteligentes en IOT. Tal como se mencionó en el capítulo 2 a un objeto inteligente se lo entiende como un sistema embebido que puede entender y reaccionar a lo que está ocurriendo en su alrededor [Kortuem, G, et al., 2010] y que no solo se interesa por la interacción con el usuario sino también de la interacción con otros objetos inteligentes o incluso otro software, es decir, contempla la interacción con el mundo físico y virtual al mismo tiempo. El concepto internet de las cosas implica la construcción de estos objetos inteligentes en sistemas embebidos de forma tal que tengan la conectividad necesaria para construir un ecosistema en donde todos los artefactos (Tanto hardware como software) estén interconectados [Holler, J., 2014].

La ausencia de una arquitectura con estos fines dificulta el desarrollo de soluciones haciendo que los existentes terminen siendo “a medida”, lo cual eleva costos y disminuye el grado de reutilización del software, dado que para cada solución se plantea una nueva arquitectura en vez de reutilizar componentes arquitectónicos de un problema similar anterior. Esto se debe a que el diseño arquitectónico de software es un proceso que debe tomar como entrada los requerimientos del sistema y tener como salida una arquitectura de software que los satisfaga. Por lo general, este proceso de diseño arquitectónico es iterativo, hasta que se asegura que su estructura es acorde a los requerimientos. Con la existencia de una arquitectura de referencia, se disminuirán considerablemente las iteraciones dado que solo se necesita adaptar una arquitectura ya existente a los requerimientos específicos del problema a resolver. Además las arquitecturas de referencia sirven como base para producir arquitecturas concretas que resuelven casos particulares [IOT-A, 2013]. Esta arquitectura actúa entonces como enlace entre la fase de ingeniería de requerimientos y diseño del software, describiendo la forma en que se organiza el sistema como un conjunto de componentes relacionados entre sí (Figura 4.1).

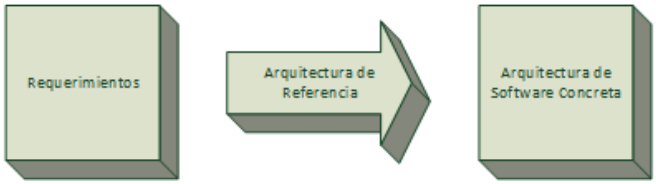

Figura 4.1. *Arquitectura de referencia en el diseño arquitectónico*

4.2. PROPUESTA DE ARQUITECTURA DE SOFTWARE DE REFERENCIA PARA OBJETOS INTELIGENTES EN INTERNET DE LAS COSAS

En esta sección se presenta un vistazo general de la arquitectura de software de referencia (Sección 4.2.1) y su modelo de referencia completo (sección 4.2.2). La misma se encuentra basada en el modelo propuesto por [Microsoft Patterns & Practices Team, 2009] explicado en el capítulo 2.

4.2.1. Vistazo General

Como se mencionó con anterioridad, la arquitectura propuesta en este trabajo está inspirada en el modelo que propuso [Microsoft Patterns & Practices Team, 2009] para sistemas distribuidos. En su presentación inicial se propuso un vistazo general para introducir los conceptos principales de la arquitectura y luego un diagrama de mayor nivel de detalle para explicar a bajo nivel cada componente. Para este trabajo se adoptó la misma estructura para presentar la arquitectura, siendo el vistazo general el objeto de este apartado (Figura 4.2). En las secciones 4.2.1.1 a 4.2.1.4 se explican las funcionalidades de la capa de presentación y sistemas externos, servicios, lógica del negocio, acceso a datos y recursos respectivamente.

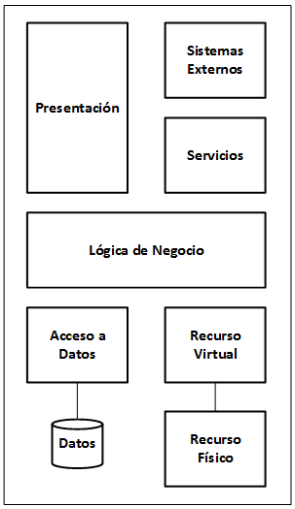

Figura 4.2. *Vistazo General de Arquitectura de Software de Referencia para Objetos Inteligentes en Internet de las Cosas*

4.2.1.1. Capas de Presentación y Sistemas Externos

El objetivo de la arquitectura es contemplar los requerimientos generales (Capítulo 2) que deben satisfacer los objetos inteligentes. En términos generales, los objetos inteligentes interactúan directamente con el usuario pero también lo hacen con otros objetos inteligentes, computadoras, dispositivos móviles, es decir, sistemas externos.

La interacción con el usuario y sistemas externos es claramente distinta, el usuario pretende entender lo que el objeto inteligente le comunica en un lenguaje claro, natural y comprensible, mientras que los sistemas externos necesitan otro tipo de lenguajes como protocolos, estándares y notaciones, como podrían ser el protocolo HTTP y la notación JSON. Por este motivo, las capas más altas de la arquitectura son los extremos de la comunicación con el usuario y sistemas externos.

La capa de presentación controla la entrada y salida del hardware con dos principales objetivos:

1) Abstraer al usuario de la complejidad en términos de electrónica, dado que se trata de un sistema embebido y
2) Abstraer a las capas inferiores de lo que el usuario ingresa o espera como respuesta en términos de formatos. Por ejemplo, la capa de presentación se puede encargar del control de un LCD inteligente, una pantalla táctil como así también botones y otros métodos de entrada.

La capa de sistemas externos, por otra parte, representa a las entidades que consumen información del objeto inteligente directamente a través de servicios, sin pasar por la capa de presentación. Es posible que el usuario esté del otro lado del sistema externo, monitorizando o controlando la información que llega, pero esto no es estrictamente necesario ya que el proceso de medición puede también ser automatizado a través de sistemas informáticos.

4.2.1.2. Capa de Servicios

La información que consumen el usuario o los sistemas externos es, a priori, distribuida por la capa de servicios. El objetivo de la capa de servicios es exponer la funcionalidad del sistema, implementada en las capas inferiores y para esto necesita comunicarse a través de protocolos, estándares y notaciones de forma tal que sea comprensible por parte de sistemas externos. Dependiendo los requerimientos puntuales del objeto inteligente, esta capa puede implementar protocolos de comunicación como HTTP para crear servicios web o bien protocolo Zigbee, ampliamente utilizado en el contexto de Internet de las Cosas. Otros protocolos puede ser Bluetooth aunque también se puede utilizar WiFi o comunicación vía señales infrarrojas.

4.2.1.3. Capa Lógica del Negocio

La capa que procesa y produce información para que pueda ser utilizada por el resto de las capas es la de lógica del negocio. Esta capa tiene las reglas necesarias para resolver el caso de uso que se plantea como requerimiento del objeto inteligente. Esto quiere decir que tiene que conocer cómo atender las peticiones que le envía la capa de servicios y como responder adecuadamente, como así también conocer el procedimiento que hay que seguir para obtener resultados correctos, cuándo hay entradas inválidas o cuándo son peticiones no autorizadas. La lógica del negocio es el núcleo de la aplicación software ejecutándose en el objeto inteligente. En esta capa se incluyen además los modelos de datos que ayudan a resolver el caso de uso como por ejemplo estructuras de datos o clases. Más adelante se observará una vista más detallada de esta capa para entender mejor su estructura y funcionamiento.

4.2.1.4. Capas de Recursos Virtuales, Recursos Físicos y Acceso a Datos

Los objetos inteligentes tienen la capacidad de consumir recursos. Por ejemplo, pueden tener integrados sensores de luz para regular el brillo de una pantalla, sensores de humedad para controlar un sistema de riego o también pueden disponer de conectores hacia lámparas o reflectores para controlar la iluminación en la calle.

En el contexto de este trabajo se diferencia un recurso virtual de un recurso físico siendo el recurso virtual la representación abstracta del recurso físico en el software, mientras que el recurso físico implica un objeto tangible, es decir, si se considera que el objeto inteligente tiene un sensor de humedad, pues el sensor de humedad va a ser el recurso físico que va a ser accedido desde un recurso virtual – componente software – que efectúa mediciones sobre el mismo. Vale aclarar, sin embargo, que es posible que el recurso virtual este asociado con otro recurso virtual de otro objeto inteligente o sistema externo, como por ejemplo, un objeto inteligente radicado en Buenos Aires que quiere medir la humedad relativa en un campo de Santiago del Estero a través de otro objeto inteligente.

Otra funcionalidad generalmente deseable en los objetos inteligentes es la capacidad de almacenar datos temporal o permanentemente. Entre estos datos se pueden encontrar configuraciones o datos de aplicación necesarios para la ejecución del software. Lo cierto es que existen muchos mecanismos de almacenamiento, como por ejemplo EEPROM y FLASH, aunque también existen objetos inteligentes que almacenan su información en memorias SD o incluso tienen una base de datos interna embebida. En este contexto, la capa de acceso a datos tiene como objetivo abstraer al

resto de los componentes de la complejidad que requiere acceder al repositorio de datos para recuperar y guardar información.

4.2.2. Modelo Completo

En esta sección se presenta de forma detallada la arquitectura de software para objetos inteligentes en internet de las cosas. Esta versión está basada en la de la figura 4.2 y se muestra en la figura 4.3. En las secciones 4.2.2.1 a 4.2.2.5 se explican las funcionalidades de la capa presentación, servicios, lógica de negocio, capas transversales, recursos y acceso a datos, respectivamente.

4.2.2.1. Capa de Presentación

Tal como se mencionó en la sección 4.2.1.1, la capa de presentación es la capa encargada de interactuar directamente con el usuario, abstrayéndolo de los tipos de datos que se estén utilizando o cuestiones electrónicas. Esta capa tiene dos subcomponentes:

- Hardware I/O, que tiene como objetivo controlar la entrada y salida por hardware
- Lógica de Presentación, que tiene como objetivo ordenarle al componente Hardware I/O la manera en que tiene que comportarse

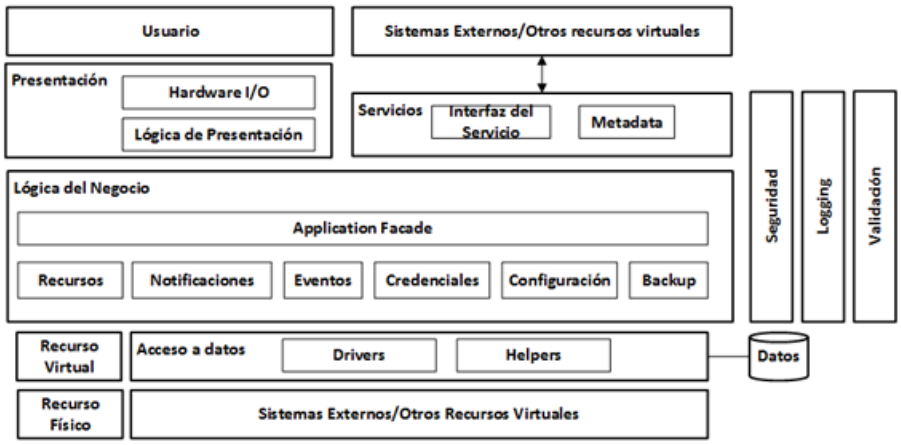

Figura 4.3. *Detalle de Arquitectura de Software de Referencia para Objetos Inteligentes en Internet de las Cosas. La misma se encuentra basada en la propuesta por [Microsoft, 2009] y es una vista detallada del vistazo general.*

La razón por la cual existen estos dos componentes es que resulta conveniente separar los datos de su presentación, consiguiendo así un desacoplamiento entre modelo de datos y vista. Por otra parte, permite centralizar todo lo relativo a hardware de presentación (Por ejemplo, LCDs inteligentes, LEDs y teclados) en un solo componente, lo cual abarata costos de mantenimiento. Este esquema de

capa de presentación es similar al propuesto por el patrón Model-View-Controller (MVC). En donde la vista sería Hardware I/O y controlador Lógica de Presentación.

4.2.2.2. Capa de Servicios

La capa de servicios, encargada de exponer las funcionalidades que ofrece y la información que produce la capa lógica del negocio, está compuesta por dos subcomponentes:

- Interfaz del Servicio, que ofrece los endpoints necesarios para que los servicios puedan ser consumidos
- Metadata, que expone información acerca del servicio para conocer de antemano cómo se debe consumir el servicio

El componente Metadata sería análogamente similar al concepto de WebService Definition File (WSDL) de SOAP o RAML, para que cuando el usuario quiera consumirlo, primero entienda cómo funciona. En este caso, no es necesario tener un archivo de definición pero sí es importante que la interfaz quede claramente definida en un solo componente. Este concepto a veces es conocido también como contrato, dado que establece un acuerdo, entre servidor y cliente, en cómo intercambiar información. Otro aspecto importante de este contrato es la interfaz del servicio. En términos de servicios web, por ejemplo, la interfaz de servicio define los endpoints a los cuales el usuario puede hacer llamados para consumir el servicio. Es importante que los endpoints no varíen sin que el usuario sea notificado (Esto implicaría que sus aplicaciones dejen de funcionar).

4.2.2.3. Capa Lógica de Negocio

Recordando lo mencionado en la sección 4.2.1.3, la capa lógica del negocio implementa las funcionalidades necesarias para satisfacer los requerimientos del objeto inteligente. Si bien es cierto que los requerimientos varían de un escenario a otro, la realidad es que en términos generales hay requerimientos que se mantienen o repiten. A continuación se enuncian los componentes que satisfacen estos requerimientos generales:

- Recursos: el objetivo de un objeto inteligente es proveer información sobre su contexto tanto al usuario como al entorno de internet de las cosas. Para esto, es necesario que tenga un módulo de recursos que le permita efectuar mediciones y controlar los recursos que tiene a su alcance.
- Notificaciones: volviendo a lo mencionado en el punto anterior, los objetos inteligentes tienen la capacidad de informar a su entorno lo que sucede en su contexto. Un módulo de

notificaciones ayuda a mantener actualizados a aquellos sistemas o usuarios que necesiten tener la información en tiempo real.

- Eventos: en el entorno del objeto inteligente ocurren eventos permanentemente. Estos eventos pueden ser cambios de temperatura, detección de movimientos, aumento de los niveles de humedad o decremento de la presión sanguínea. Es importante que el objeto inteligente pueda procesar adecuadamente estos eventos de forma tal que se puedan tomar decisiones con información precisa obtenida en tiempo y forma.
- Credenciales: la mayoría de los sistemas tienen diferentes niveles de seguridad. Típicamente, se define un usuario administrador y luego distintos perfiles de acceso de forma tal que cada perfil tiene acceso a diferentes funcionalidades. Para autenticar usuarios se pueden usar contraseñas, lectores de huellas digitales, llaves o tarjetas inteligentes.
- Configuración: los objetos inteligentes deben ser parametrizables, es decir, deben permitir al usuario final cambiar la configuración para que funcione en el contexto deseado. Esta configuración, dependiendo del objeto inteligente, podría incluir dirección IP, dirección MAC, un nombre, entre otros.
- Backup: el manejo y recuperación de errores es un aspecto clave de los sistemas embebidos. Un adecuado componente de backup permite un rápido recupero ante algún desastre que haya dejado al objeto inoperante.

El hecho de que estas capas existan en esta arquitectura de referencia no implica que deban existir en la implementación de todos los objetos inteligentes. Es posible que no sea deseable, por ejemplo, un componente de credenciales y que sí sea necesario agregar otro componente, esto varía según los requerimientos. Un componente importante que aún no ha sido mencionado es Application Facade. Este componente responde a un patrón de diseño mediante el cual se encapsulan las funcionalidades que ofrecen sus subcomponentes. En el caso de lógica del negocio, encapsula las funcionalidades ofrecidas por los componentes enunciados con anterioridad.

4.2.2.4. Capas Transversales

En la mayoría de los sistemas existen funcionalidades que son de interés en más de un componente (en este caso, lógica del negocio y servicios). Este es el caso de las funcionalidades que ofrecen las capas transversales. En esta arquitectura, estas capas están compuestas por los siguientes componentes:

- Seguridad: este componente está encargado de la verificación de certificados, de cifrado y descifrado de información y de la implementación de políticas de acceso.

- Logging: sobre todo cuando en la presencia de fallas, es de interés conocer qué fue lo que sucedió. El componente Logging otorga la funcionalidad de registrar lo que los componentes han realizado para poder detectar la causa del fallo.
- Validación: como en todo sistema, los datos pasan por un proceso de validación. En este componente se implementan las reglas necesarias para determinar si un dato es válido o no.

4.2.2.5. Capa Recursos Físicos, Recursos Virtuales y Acceso a Datos

Las capas recurso virtual y acceso a datos son las de más bajo nivel dentro de un objeto inteligente. La capa de recurso virtual es quizás la más importante dentro de la arquitectura dado que de ella depende la información extraída del contexto. Para lograr este objetivo, el recurso virtual debe incluir las librerías o componentes software necesario para interactuar con el recurso físico asociado. Esto quiere decir que si, por ejemplo, el recurso físico es un componente ZigBee, pues el recurso virtual deberá poder dialogar ZigBee con el recurso físico. Cabe aclarar que estos recursos virtuales obtienen información de entrada para el objeto inteligente, mientras que los de las capas superiores son para consumir la información de salida del objeto inteligente.
La capa de acceso a datos, por otra parte, está compuesta por dos componentes:

- Drivers, que ayudan al componente a tener acceso al repositorio de datos
- Helpers, que dan soporte para la extracción y transformación de datos desde el repositorio

4.3. RELACIÓN ENTRE COMPONENTES

En esta sección se describe cómo se relacionan los distintos componentes de la arquitectura de la sección 4.2. Para esto, se hace referencia al modelo 4+1 de [Kruchten, P., 1995]. Para este trabajo se optó por omitir la vista de desarrollo y de escenarios, dado que ambos diagramas dependen exclusivamente de los requerimientos específicos para el objeto inteligente, cosa que la presentada en este trabajo carece, por ser una arquitectura de referencia. En las secciones 4.3.1 a 4.3.3 se documentan entonces la vista lógica, vista de procesamiento y vista física, respectivamente.

4.3.1. Vista Lógica

Tal como se mencionó en capítulos anteriores la vista lógica de una arquitectura soporta los requerimientos funcionales (lo que el sistema debe proveer a los usuarios en términos de servicios) [Kruchten, P., 1995]. Si bien se pueden usar cualquier tipo de diagramas para documentar esta vista,

los más comunes son Diagramas de Clase UML y Diagramas Entidad-Relación. Para este trabajo, se optó por diagramas de clase UML. En la figura 4.4 se puede observar la vista lógica de la arquitectura de referencia. En las secciones 4.3.1.1 a 4.3.1.4 se explican por separado las relaciones entre los componentes más importantes.

4.3.1.1 Capas Superiores

En esta sección se explica la relación entre las capas de presentación, servicio, Application Facade y capas transversales (Figura 4.5).

La clase Application Facade utiliza la capa de presentación para enviarle la información que debe mostrar y también de qué modo y dónde mostrarlo. Para implementar esta funcionalidad deberá proveer alguna interfaz del tipo “Mostrar (Mensaje)”.

Por ejemplo, si la capa de presentación controla una pantalla LCD, debe conocer no sólo la información a mostrar sino también si mostrarlo en uno o varios renglones (en caso en que sea texto) y en qué sección de la pantalla ubicarlo. Esta clase también deberá aceptar entradas por parte del usuario como entradas por pantalla táctil, botones o teclados, por lo tanto, la interfaz que provea deberá contemplar un método como “Leer (): Mensaje”.

La clase servicios le provee a la clase Application Facade la interfaz necesaria para comunicarse con sistemas externos a través de un protocolo específico. La interacción entre Servicios y Application Facade se va a dar entre métodos como “Leer (): Mensaje” o “Enviar (Mensaje)”, donde mensaje es información codificada y escrita en alguna notación en particular. Por ejemplo, si una aplicación Android se quiere comunicar con un objeto inteligente, pues puede hacerlo perfectamente a través de servicios web (asumiendo que la clase servicios expone servicios web) y notación JSON. En este escenario, mensaje va a ser un mensaje objeto JSON.

4.3.1.2 Capas de Recursos

En esta sección se explica la relación entre las capas relativas a los recursos virtuales y Application Facade (Figura 4.6).

Los recursos son quizás los componentes más importantes un objeto inteligente, dado que a través de ellos se puede procesar y publicar información. Los recursos virtuales deben proveer una interfaz que permita efectuar lecturas y escrituras sobre el recurso físico que tiene asociado a través de su dirección.

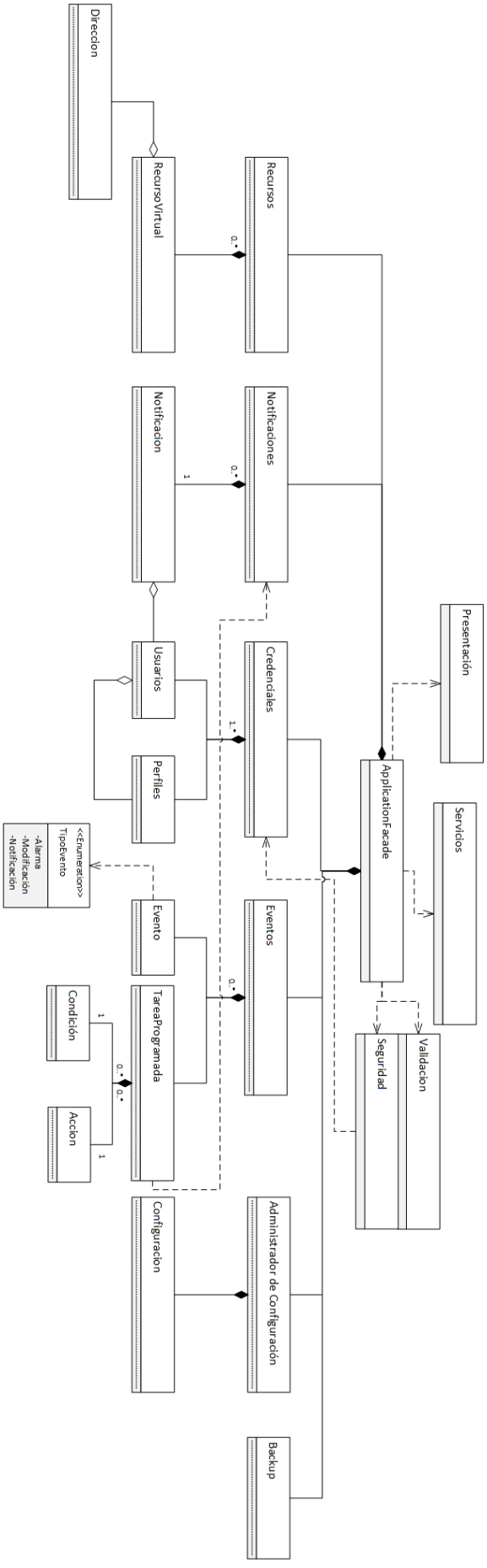

Figura 4.4. *Vista Lógica de la Arquitectura de Referencia*

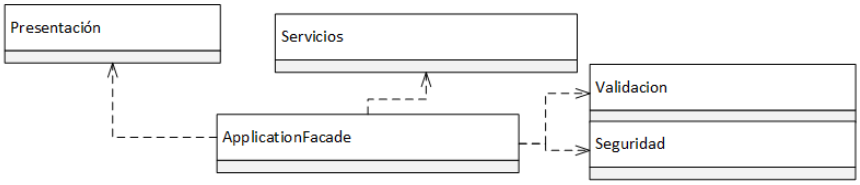

Figura 4.5. *Relación entre capas superiores*

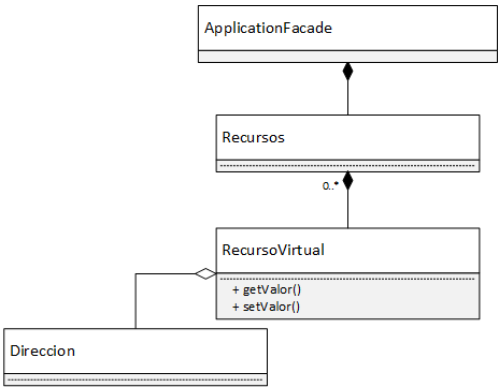

Figura 4.6. *Relación entre capas de recursos*

En la práctica, los recursos virtuales implementan una forma de lectura en particular, es decir, la lectura de un sensor de humedad puede diferir en la de un sensor de temperatura, por lo tanto es posible que existan diferentes implementaciones para los métodos getValor y setValor.

La dirección debe proveer información suficiente para que se pueda encontrar el dispositivo. Por ejemplo, en un microcontrolador la dirección podría proveer los pines en los cuales los recursos físicos son conectados.

Esta información es finalmente gestionada por la clase Recursos, que mantiene una colección de recursos virtuales y le brinda una interfaz al Application Facade para agregar o quitar elementos de esa colección. A su vez, esta clase se encarga del aprovisionamiento de nuevos recursos virtuales.

4.3.1.3 Capas de Credenciales

En esta sección se explica la relación entre las capas relativas a las credenciales, Application Facade y Notificaciones (Figura 4.7).

La clase de credenciales está compuesta por un conjunto de usuarios y perfiles. A través de estos datos, deberá proveer una interfaz para validar las credenciales con las cuales el usuario se está autenticando ante la aplicación. Por ejemplo, si están utilizando un soporte de tarjeta magnética para la autenticación, pues las credenciales deberán ser válidas para el acceso y autorización de la operación que el usuario desee realizar.

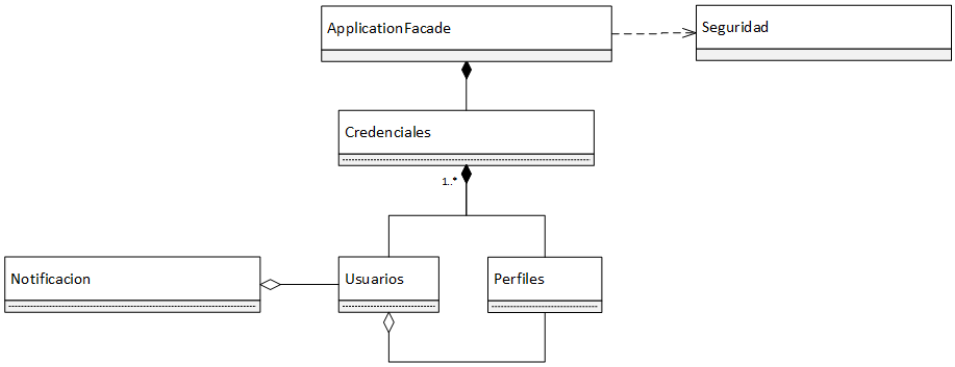

Figura 4.7. *Relación entre capas relativas a las credenciales*

Por otra parte, el módulo de credenciales también se comunica con el módulo de notificaciones dado que las notificaciones tienen como objetivo grupos de usuarios. Es decir, no todos los usuarios deben recibir las mismas notificaciones.
Por último, la clase de seguridad es utilizada en este módulo para desencriptar o encriptar el conjunto de datos entrante o saliente, respectivamente.

4.3.1.4 Capas de Eventos

En esta sección se explica la relación entre las capas relativas a los eventos, Application Facade y Notificaciones (Figura 4.8). La clase Eventos es la que gestiona este módulo y está compuesta por una colección del tipo Evento y otra del tipo Tarea Programada. La clase Evento contiene información acerca del tipo de evento y algún mensaje. A modo de ejemplo en la figura se agregan eventos del tipo Alarma, Modificación y Notificación. Por otra parte, la clase Tarea Programada está compuesta por dos clases: Condición y Acción. La clase condición debe contener al menos dos operandos y un operador y debe proveer alguna interfaz para que se pueda evaluar su estado (Por verdadero o falso). La clase Acción simplemente debe proveer una interfaz para que la misma sea ejecutada. De esta manera se pueden ejecutar las acciones de una tarea programada, cuando las condiciones son verdaderas. Por último, la relación entre una tarea programada y notificaciones está dada por la posibilidad de que una acción tenga como objetivo notificar a los usuarios de algún evento en particular.

4.3.2. Vista de Procesamiento

Al contrario de la vista lógica, la vista de procesamiento considera los requerimientos no funcionales, como rendimiento y disponibilidad. Esta vista estudia el procesamiento de datos desde el punto de vista de los sistemas distribuidos, abarcando cuestiones de concurrencia e integridad,

como así también cuestiones de comunicación. Sin embargo, para esta arquitectura se adopta a la vista de procesamiento como un medio para especificar los algoritmos que se utilizan para ofrecer la funcionalidad requerida. Para documentar esta vista se optó por el uso de diagramas de actividades UML.

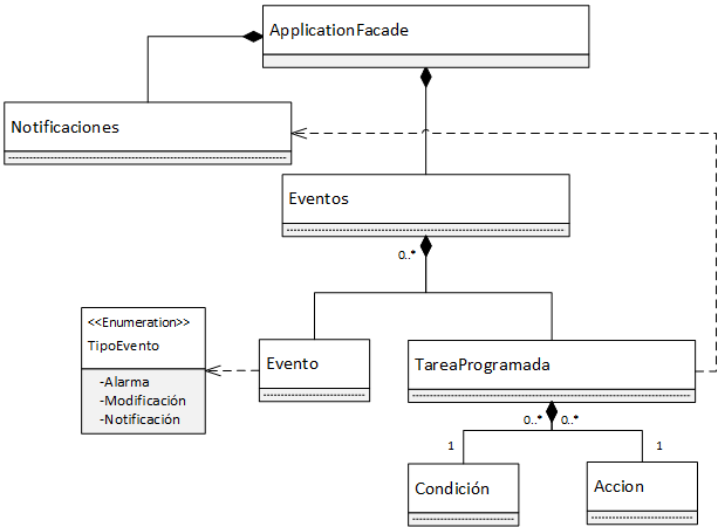

Figura 4.8. *Relación entre capas relativas a las credenciales*

El diagrama de la figura 4.9 muestra el flujo de datos para verificar que un usuario tiene permisos para ejecutar la petición que realizó.

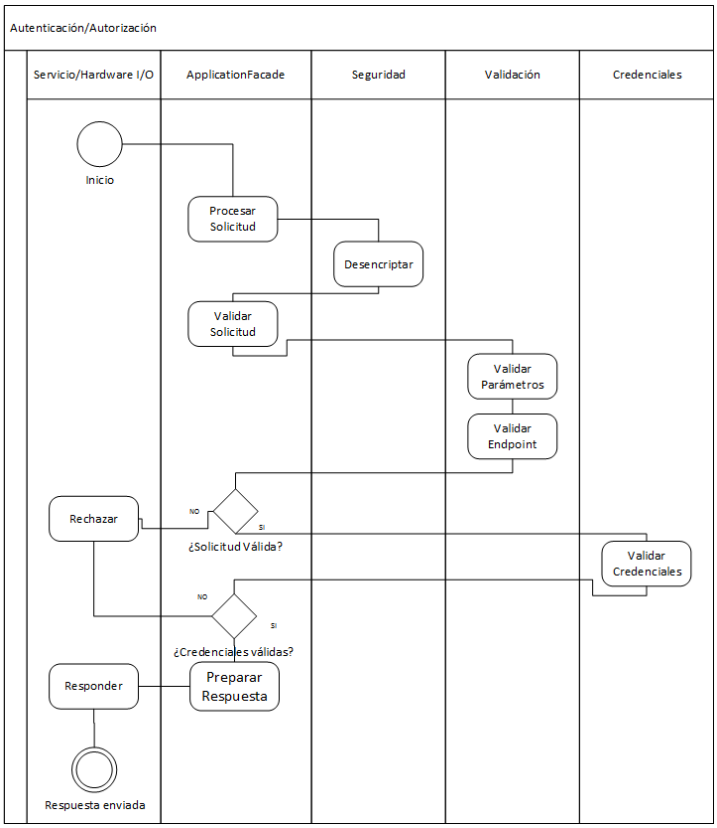

Figura 4.9. *Diagrama de Actividad UML para la autenticación y autorización*

El comienzo del flujo lo da el componente de Servicio o Hardware I/O (de la capa de presentación). El primero puede validar contra un usuario y contraseña o bien un token que la aplicación haya otorgado, mientras que el segundo puede utilizar una tarjeta magnética, lector de huellas digitales o reconocimiento facial. En cualquiera de los dos casos, la solicitud es procesada por la capa de seguridad desencriptando el mensaje y luego se envía a la capa de validación, que se encarga de validar que los parámetros y el recurso al que se quiere acceder tienen un formato adecuado y aceptable por el objeto inteligente. En el caso en que la validación tenga resultado negativo, se rechaza la petición y si es positivo, se validan las credenciales en la capa de credenciales. En esta capa se determina si los datos enviados en el mensaje son auténticos y además si el usuario tiene permisos para acceder al recurso. Por último, si las credenciales son inválidas, se rechaza la respuesta y si son válidas, se prepara la respuesta y se la envía de vuelta. Este flujo de datos se da siempre que entre una petición en el objeto inteligente (En el resto de las figuras se omite para no repetirlo en todos los diagramas).

El diagrama de la figura 4.10 muestra el flujo de datos para asignar un valor a un recurso virtual. El comienzo del flujo lo da el componente de Servicio o Hardware I/O e instantáneamente le transmite el mensaje a ApplicationFacade para que delegue la petición en el subcomponente adecuado. En este caso, se trata del Administrador de Recursos que itera sobre su colección de recursos y le envía el valor que se envió para que se le asigne. El recurso virtual cuando lo recibe escribe el valor en el puerto de salida que tiene asociado. Eventualmente, aunque el gráfico no lo contemple, se puede dar la situación en que sea necesario obtener una retroalimentación del valor asignado – sobre todo en aplicaciones críticas como reactores nucleares o sistemas de frenado automático – para validar que efectivamente el nuevo valor se encuentra en el puerto de salida.

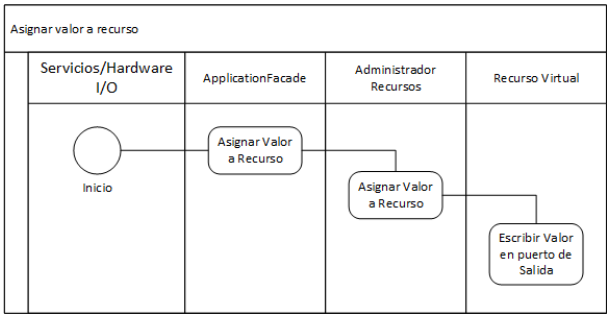

Figura 4.10. *Diagrama de Actividad UML para asignar valor a un recurso virtual*

El diagrama de la figura 4.11 muestra el flujo de datos para leer un valor de un recurso virtual. El comienzo del flujo lo da el componente de Servicio o Hardware I/O e instantáneamente le transmite el mensaje a ApplicationFacade para que delegue la petición en el subcomponente adecuado. En este caso se trata del Administrador de Recursos que itera sobre su colección de recursos y le envía

un mensaje para que retorne su valor. El recurso virtual lee el valor en su puerto de entrada y se lo devuelve al administrador de recursos. Este se lo devuelve a ApplicationFacade para que prepare la respuesta y finalmente sea enviada a través de la capa de Servicios o Hardware I/O.

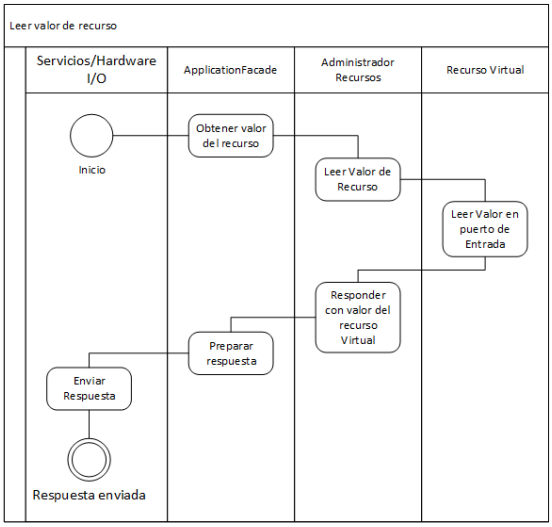

Figura 4.11. *Diagrama de Actividad UML para leer un valor de un recurso virtual*

El diagrama de la figura 4.12 muestra el flujo de datos para leer un valor de un recurso virtual. El comienzo del flujo lo da el componente de Servicio o Hardware I/O e instantáneamente le transmite el mensaje a ApplicationFacade para que delegue la petición en el subcomponente adecuado. En este caso se trata del Administrador de Recursos que asigna un espacio en memoria para el nuevo recurso virtual. Le comunica al recurso cuales son los parámetros de inicialización y luego de que el mismo los procesa, el administrador de recursos le envía el nuevo recurso virtual al administrador de datos para que lo persista. Por último, ApplicationFacade recibe el nuevo recurso para que sea enviado en una respuesta a través de la capa de Servicio o Hardware I/O.

El diagrama de la figura 4.13 muestra el flujo de datos para ejecutar una tarea programada. El comienzo del flujo lo da el componente ApplicationFacade para que se delegue la petición en el subcomponente adecuado. En este caso se trata del Administrador de Eventos que itera por cada una de las tareas programadas verificando si las condiciones que tiene son verdaderas o falsas. Si las condiciones de una tarea programada son verdaderas, itera por cada una de sus acciones y las ejecuta. Si las condiciones no se cumplen, la tarea se omite.

Por último, el diagrama de la figura 4.14 muestra el flujo de datos para restaurar el objeto inteligente a su configuración por defecto. El comienzo del flujo lo da el componente de Servicio o Hardware I/O e instantáneamente le transmite el mensaje a ApplicationFacade para que delegue la petición en el subcomponente adecuado. En este caso se trata del Administrador de Configuración que asigna los valores de fábrica a la configuración actual y se la envía al administrador de datos para que guarde los cambios. Luego, la nueva configuración es enviada a ApplicationFacade para

que prepare la respuesta, de forma tal que la capa de Servicios o Hardware I/O pueda enviarla. Finalmente, el objeto inteligente se reinicia para que su nueva configuración tenga efecto.

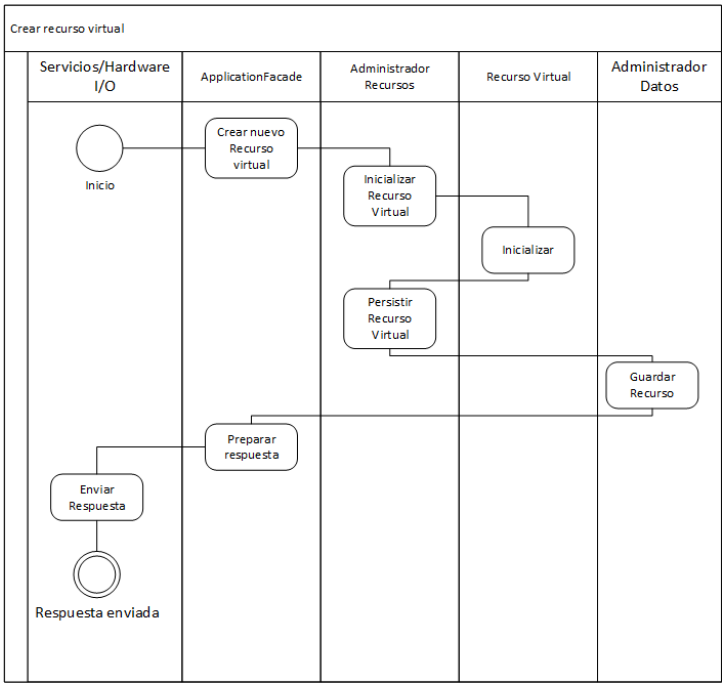

Figura 4.12. *Diagrama de Actividad UML para crear un recurso virtual*

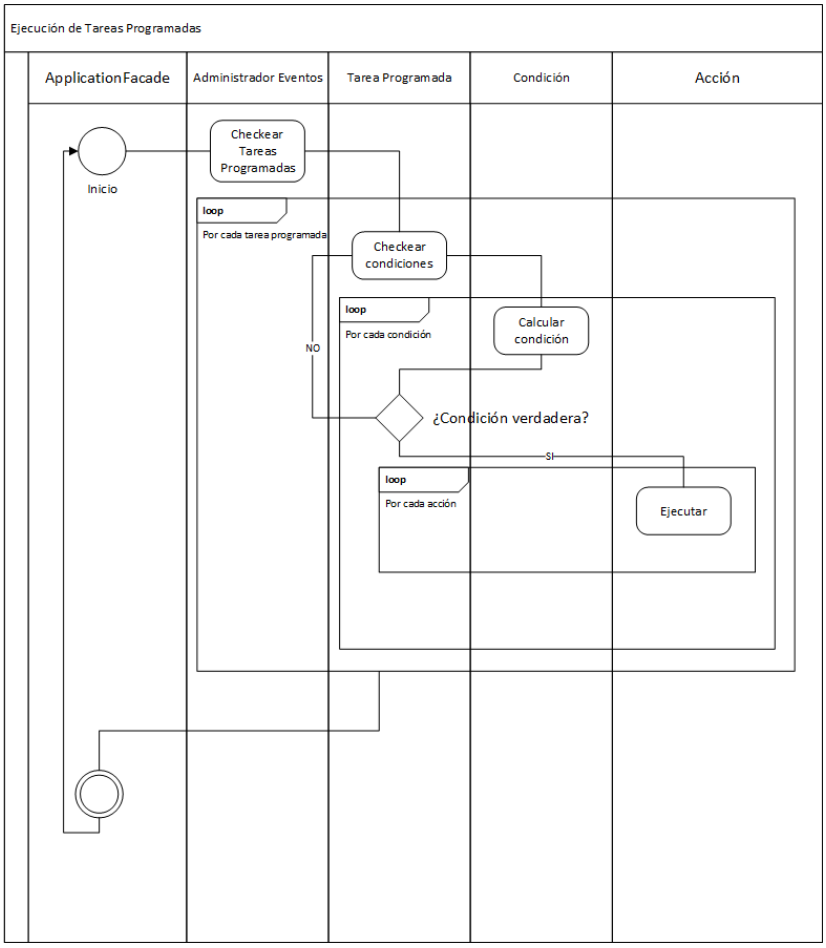

Figura 4.13. *Diagrama de Actividad UML para ejecutar una tarea programada*

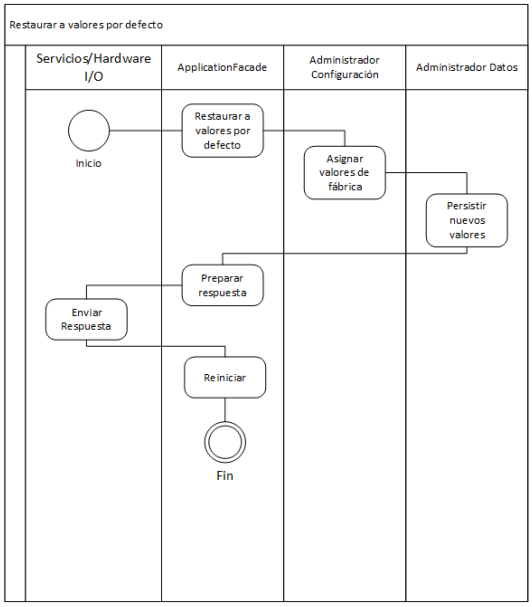

Figura 4.14. *Diagrama de Actividad UML para restaurar a valores por defecto*

4.3.3. Vista Física

Esta vista representa cómo se distribuyen los componentes entre los distintos nodos del sistema. En otras palabras, se ubica cada parte del software en un nodo, de forma tal que se mapeen software y hardware.

En la figura 4.15, se muestra un diagrama de despliegue de alto nivel, que describe cómo se ubica al objeto inteligente en un modelo orientado a servicios. En el diagrama se puede observar cómo el objeto inteligente provee servicios a los consumidores y, para generar una respuesta a esa petición, consume los valores que lee de los recursos, los procesa y luego emite una respuesta. La idea de pensar a los objetos inteligentes como un Gateway, es decir, un intermediario, entre el consumidor y el recurso permite contemplar que hay recursos como sensores de temperatura, de luz o de humedad que no tienen la capacidad de integrarse a un ambiente de internet de las cosas.

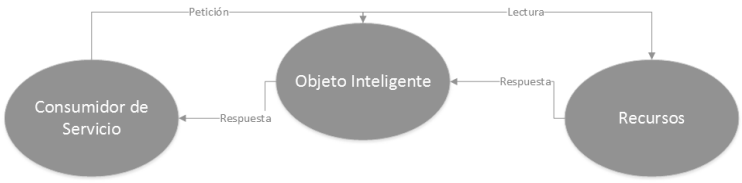

Figura 4.15. *Modelo Orientado a Servicios de un Objeto Inteligente en Internet de las Cosas.*

La figura 4.16 muestra la vista física de la arquitectura de referencia y es una versión detallada del modelo orientado a servicios propuesto en la figura 4.15. La vista comienza con el usuario

realizando peticiones al objeto inteligente a través de actuadores o software externo como pueden ser sistemas empresariales, aplicaciones web, o bien otros objetos inteligentes. Por ejemplo, en el caso de la biblioteca, la comunicación podría darse entre el sistema de reservas y el objeto inteligente, aunque si un usuario administrador quisiera acceder directamente al objeto inteligente, también podría hacerlo. Luego, dentro del objeto inteligente se encuentran aquellos componentes que resuelven las peticiones generadas por el entorno. Básicamente, a través de ApplicationFacade se delega el trabajo en los componentes de negocio y las componentes transversales (Se las reemplazo por un solo componente para ahorrar espacio en el diagrama), y los recursos virtuales ofrecen una interfaz hacia los recursos físicos del objeto inteligente (Como pueden ser sensores de presencia o actuadores para encender una lámpara) como así también una interfaz hacia otros recursos virtuales, que pueden estar alojados en otro objeto inteligente. Notar que los recursos físicos del objeto inteligente impactan directamente sobre los objetos del mundo físico real, es decir, objetos no abstractos que el usuario puede ver y tocar. Por último, el componente de acceso a datos junto a su repositorio también es incluido dentro del objeto inteligente.

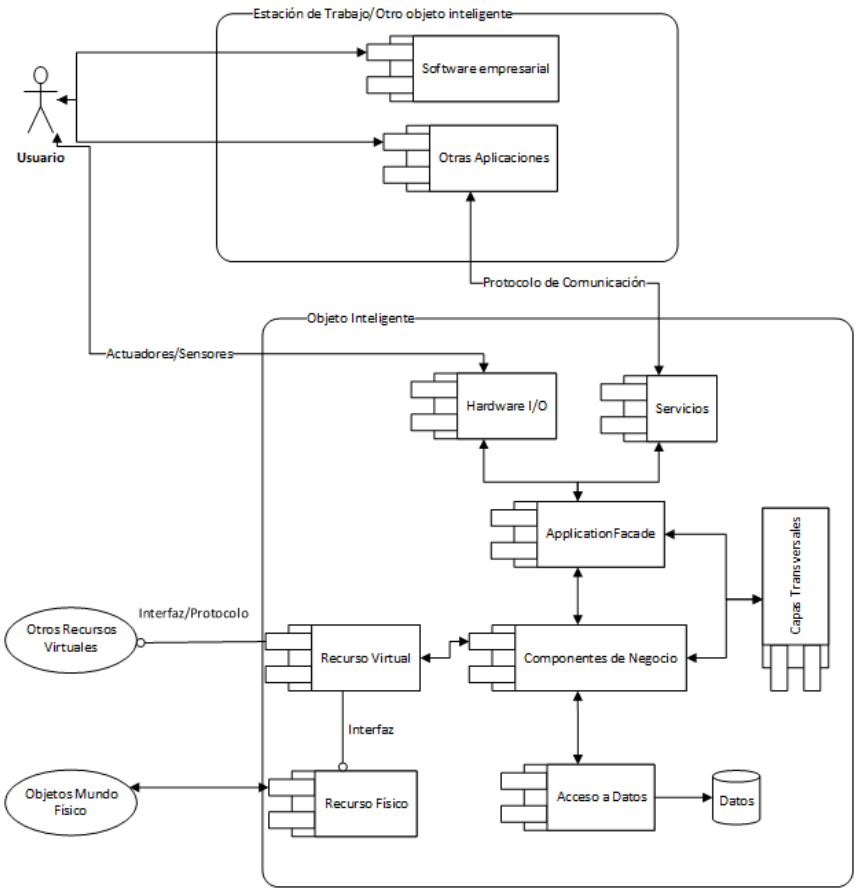

Figura 4.16. *Vista Física de la Arquitectura de Referencia*

4.4. UN EJEMPLO PRÁCTICO

En esta sección se propone un ejemplo práctico para mostrar cómo se puede utilizar esta arquitectura para resolver una problemática actual. En la sección 4.4.1 se enuncia el problema y en la sección 4.4.2 una solución en base a la arquitectura de referencia propuesta.

4.4.1. Enunciado del Ejemplo

La Universidad Nacional de Lanús tiene como misión automatizar el sistema de riego de un campo en Lobos, Provincia de Buenos Aires. Para esto, se van a fabricar dispositivos que cuentan con los sensores y actuadores necesarios para monitorizar y regular la humedad relativa, temperatura y luminosidad del terreno y las plantas. De esta manera, se prevé un menor consumo de energía eléctrica y de agua. El centro de monitoreo va a estar ubicado en el campus de Remedios de Escalada de la Universidad y van a estar trabajando, en distintas tareas, veinte personas monitoreando el campo a través de un sitio web.

Dado que el centro de monitoreo no trabaja 24x7, se necesita que los dispositivos notifiquen a la guardia de turno en caso que haya algún desperfecto en el sector asociado a cada dispositivo, es decir, si por ejemplo se detecta un incremento en los niveles de humedad, la persona que se encuentre de guardia deberá recibir una notificación en la aplicación móvil que tiene instalada en su celular. Además, el dispositivo, al estar conectado al sistema de riego, tiene la capacidad de abrir o cerrar el paso de agua, de forma tal que puede regular automáticamente la humedad, al igual que la temperatura y luminosidad con los ventiladores y lámparas, respectivamente. En caso en que la persona de guardia, al monitorizar los datos, crea que esta por producirse algún problema, puede remota y manualmente disparar las acciones necesarias para prevenirlo, como abrir un grifo o encender un ventilador. Si el guarda necesita ir personalmente a solucionar el inconveniente, dado que en el campo no hay señal inalámbrica de internet, es necesario visualizar los valores que se están midiendo en un LCD que el dispositivo tendrá.

En cuanto a los usuarios, se necesita que exista un usuario administrador capaz de reiniciar el dispositivo a valores por defecto, en caso de algún desperfecto pero el resto de los usuarios no pueden acceder a esta funcionalidad. La autenticación de los usuarios se dará vía usuario y contraseña.

Por último, respecto la configuración del dispositivo, se necesita poder configurarlo para que pertenezca a distintas subredes (Dependiendo donde sea instalado), por lo tanto las propiedades de red deben poder ser modificadas. La configuración debe ser almacenada en memoria EEPROM.

4.4.2. Solución

Para solucionar el problema enunciado en la sección 4.4.1 se procedió a adaptar la arquitectura de referencia a la problemática propuesta. Esto es, un vistazo general de la arquitectura concreta (Sección 4.4.2.1), detalle de la arquitectura concreta (Sección 4.4.2.2), una vista lógica (Sección 4.4.2.3), una vista de procesamiento (Sección 4.4.2.4) y una vista física (Sección 4.4.2.5).

4.4.2.1 Vistazo General

El vistazo general de la arquitectura concreta se puede observar en la figura 4.17. La misma comienza con una capa de sistemas externos que, para este escenario en particular, está compuesta por dos aplicaciones: Una para smartphones y otra web. Esta capa interactúa directamente con la capa de servicios, que es implementada a través de servicios web. En cuanto a la capa de presentación, del enunciado se extrae que la misma va a estar encargada de mostrar mensajes sobre un display LCD. Luego, la capa lógica de negocios se transforma en este caso en lógica de sistema automático de riego, implementando las funcionalidades requeridas. La capa acceso a datos se encarga de la lectura y escritura sobre memoria EEPROM y por último, los recursos virtuales son cuatro:

1. Medidor de Iluminación: Que efectúa mediciones sobre el sensor de iluminación
2. Medidor de Humedad: Que efectúa mediciones sobre el sensor de humedad
3. Medidor de Temperatura: Que efectúa mediciones sobre el sensor de temperatura
4. Controlador de Lámpara: Que acciona valores sobre la lámpara

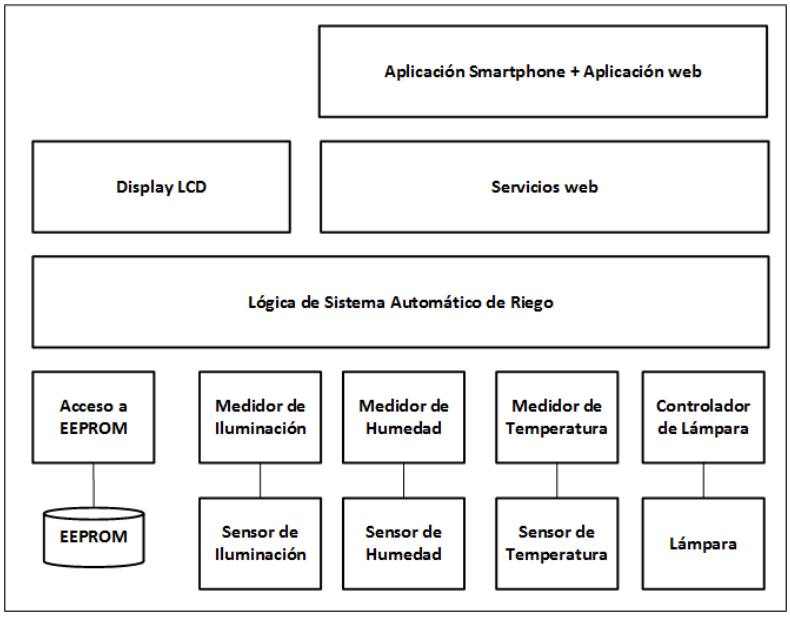

Figura 4.17. *Vistazo de la Arquitectura Concreta*

4.4.2.2 Arquitectura Detallada

La vista a continuación de la arquitectura parte de la planteada en la sección 4.4.2.1, aumentando el nivel de detalle de los componentes (Figura 4.18).

La capa de presentación muestra sus dos subcomponentes: Display LCD y Lógica de Presentación. El primero contiene los drivers necesarios para manejar el hardware del Display LCD, es decir, le envía la información que tiene que mostrar. La lógica de presentación, de forma complementaria y anterior, pre-procesa la información que se debe mostrar de forma tal que quede prolija y presentable en el display LCD, es decir, se agregan saltos de línea y detalles de formato.

La capa de Servicios Web, por definición, expone la funcionalidad del sistema hacia la aplicación web y smartphone. Internamente, está compuesta por dos subcapas: La primera, interfaz del servicio, que efectivamente envía y recibe datos hacia y desde el exterior y la segunda, metadata, que expone información necesaria para que los sistemas externos sepan cómo está estructurada la capa de servicios y así poder consumirla. En este caso, por ser servicios web, la interfaz del servicio está basada en protocolo HTTP y la metadata está descripta en un archivo XML.

La capa de lógica de negocio nuevamente se divide en varios componentes que dan soporte a las funcionalidades del sistema. Para este enunciado en particular, no hay variaciones en la cantidad de capas y todas cumplen las mismas funcionalidades que tienen por definición, al igual que las capas transversales.

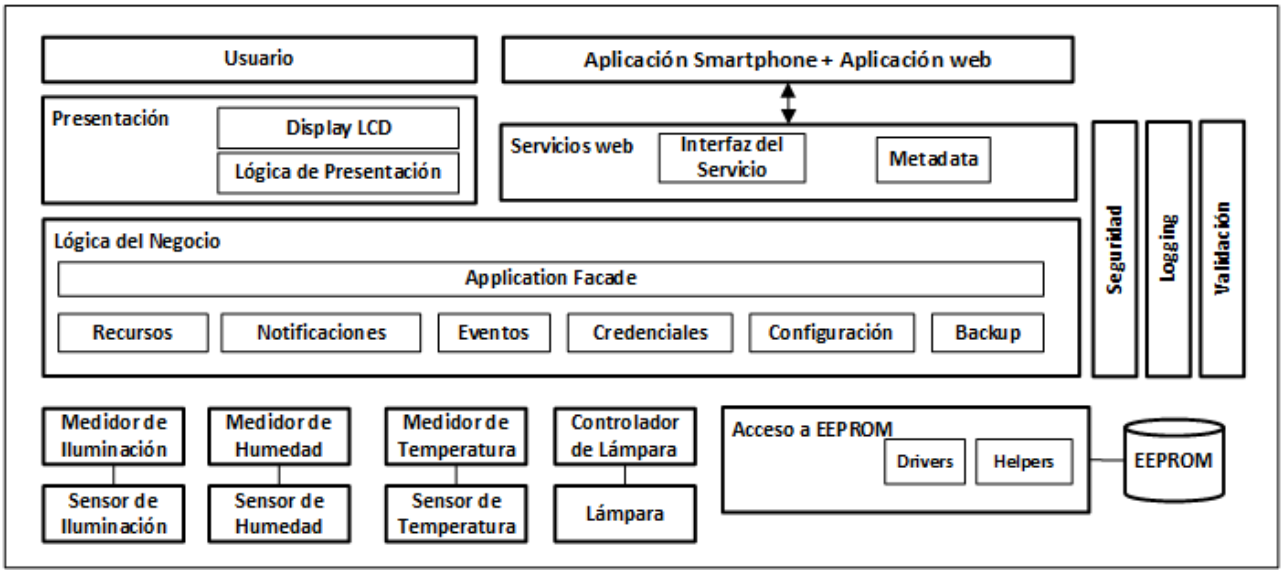

Figura 4.18. *Vista detallada de la Arquitectura Concreta*

La capa de recurso virtual en este escenario se transformó en cuatro capas: Medidor de Iluminación, de Humedad, de Temperatura y Controlador de Lámpara, que son los recursos identificados en el

enunciado. Cada recurso debe ofrecer una interfaz de leer y escribir valores y, a su vez, estos están asociados a los recursos físicos sensor de iluminación, de humedad, de temperatura y una lámpara, respectivamente.
El acceso a datos, por último, se transformó en un acceso a memoria EEPROM que es donde, por enunciado, se almacenarán los datos. Para esto, el acceso a EEPROM necesita del apoyo de drivers que conozcan cómo escribir y leer desde ese tipo de memoria.

4.4.2.3 Vista lógica

En esta sección se muestra la vista lógica (Figura 4.19) de la arquitectura. En las secciones 4.4.2.3.1 a 4.4.2.3.4 se explican por separado las relaciones entre los componentes más importantes.

4.4.2.3.1 Capas Superiores

En esta sección se explica la relación entre las capas de presentación, servicio, Application Facade y capas transversales (Figura 4.20).
En este escenario, la capa de presentación debe implementar la funcionalidad de refrescar el display LCD con el mensaje que se le envíe. Para esto, la clase Presentación implementa el método refrescarDisplay(String mensaje) el cual es accedido desde la clase Control del Riego.
La capa de servicios provee como interfaz un método que es resolverPetición(). En este caso, es resolver el llamado mediante servicios web por lo tanto la resolución consistirá en el análisis de parámetros que hayan sido enviados y los recursos que hayan sido solicitados. Luego, el control del riego decide cuáles son los procedimientos a seguir para esa petición en particular y le responde a la capa de servicios con el mensaje que tiene que devolver.
En cuanto las capas transversales, no hay ningún requerimiento en particular por lo que quedan reservadas para la validación de algunos parámetros o cifrado/descifrado de información que eventualmente se necesiten.

4.4.2.3.2 Capas de Recursos

En esta sección se explica la relación entre las capas relativas a los recursos virtuales y Application Facade (Figura 4.21). Para el ejemplo propuesto, se identificaron cuatro tipos de recursos virtuales: Sensor de Humedad, de Temperatura, de Iluminación y un controlador de lámpara.

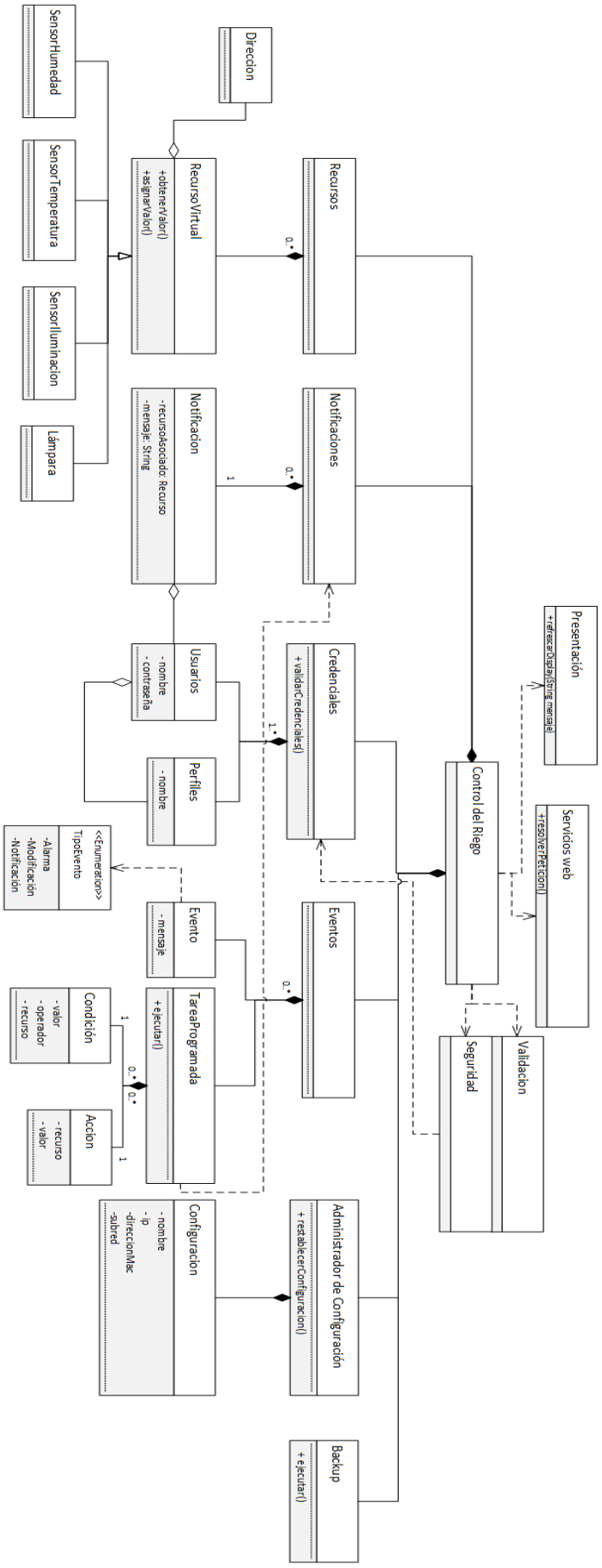

Figura 4.19. *Vista Lógica de la Arquitectura de Concreta*

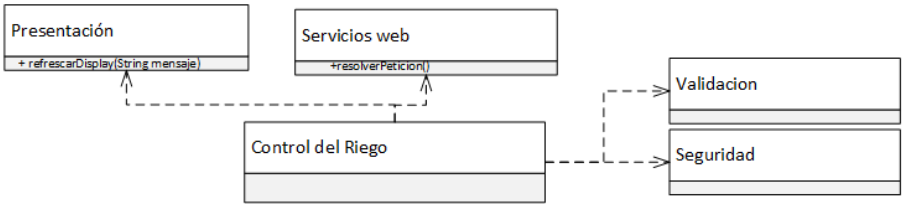

Figura 4.20. *Relación entre capas superiores*

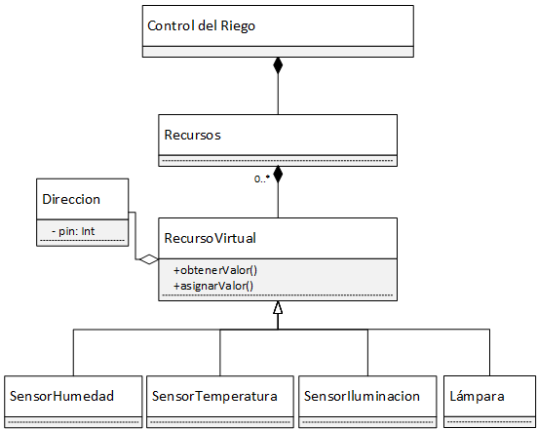

Figura 4.21. *Relación entre capas de recursos*

Cada uno de estos recursos deberá implementar apropiadamente los métodos obtenerValor() y asignarValor() definidos en la clase Recurso Virtual, dado que, por ejemplo, la forma de medir la humedad, asumiendo que es un sensor diferente y los métodos de medición varían, es distinta a la forma de medir la iluminación.

La forma en la que los recursos virtuales efectivamente miden o asignan valores sobre los dispositivos físicos está dada a través de la dirección que se les provee. En este caso, la dirección está compuesta por un pin que representa el puerto en el cual el recurso está físicamente conectado.

4.4.2.3.3 Capas de Credenciales

En esta sección se explica la relación entre las capas relativas a las credenciales, Application Facade y Notificaciones (Figura 4.22).

La clase de credenciales está compuesta por un conjunto de usuarios y perfiles. Los usuarios tienen como atributos un usuario y contraseña. A través de estos datos, deberá proveer una interfaz para validar las credenciales con las cuales el usuario se está autenticando ante la aplicación. Los perfiles por otra parte, solo incorporan como atributo un nombre que los identifica. Por otra parte, el módulo de credenciales también se comunica con el módulo de notificaciones, que a su vez contienen como atributos un recurso virtual (dado que los mensajes que se envían están asociados a un recurso en particular).

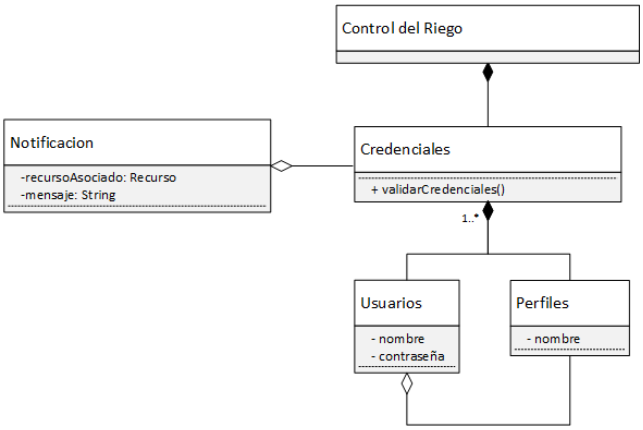

Figura 4.22. *Relación entre capas relativas a las credenciales*

4.4.2.3.4 Capas de Eventos

En esta sección se explica la relación entre las capas relativas a los eventos, Application Facade y Notificaciones (Figura 4.23).

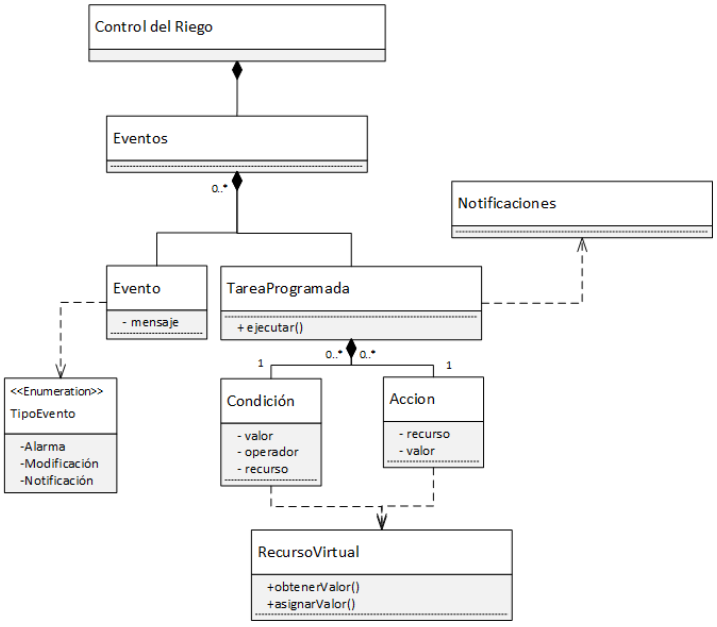

Figura 4.23. *Relación entre capas relativas a los eventos*

En el tramo de tareas programadas, una condición tiene como atributos un valor, un operador y un recurso, de forma tal que se puedan resolver expresiones del tipo “Si el valor de la temperatura es mayor a 21, entonces…”. El operador es entonces del tipo Mayor, Menor, Igual o Distinto y el recurso es en realidad una referencia a un recurso virtual.

La acción, por otra parte, consiste en asignar un valor a un recurso virtual como por ejemplo “Asignar valor encendido a la lámpara”. Por eso, sus atributos son un recurso virtual y un valor literal.

En lo que respecta a los eventos, estos solo tienen un atributo mensaje que contiene información sobre el evento en sí.

4.4.2.4 Vista Procesamiento

En esta sección se muestran las diferencias principales entre la vista de procesamiento de la arquitectura de referencia y la concreta.

El diagrama de la figura 4.24 muestra el flujo de datos para verificar que un usuario tiene permisos para ejecutar la petición que realizó. El comienzo del flujo lo da únicamente el componente de Servicio, siendo esta la primera diferencia con la arquitectura de referencia en la cual la capa de Presentación podía también iniciar este flujo. La solicitud es procesada por la capa de seguridad que desencripta el mensaje y luego se envía a la capa de validación, que se encarga de validar que los parámetros y el recurso al que se quiere acceder tienen un formato adecuado y aceptable por el objeto inteligente. En el caso en que la validación tenga resultado negativo, se rechaza la petición y si es positivo, se validan usuario y contraseña en la capa de credenciales. El resto del algoritmo, sigue tal cual al mencionado en la arquitectura de referencia.

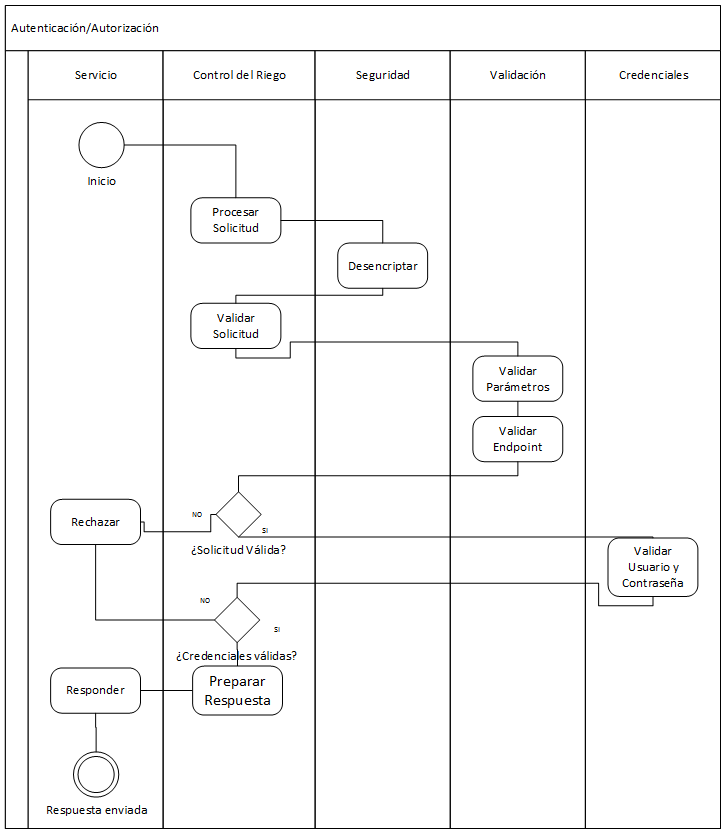

Figura 4.24. *Diagrama de Actividad UML para la autenticación y autorización*

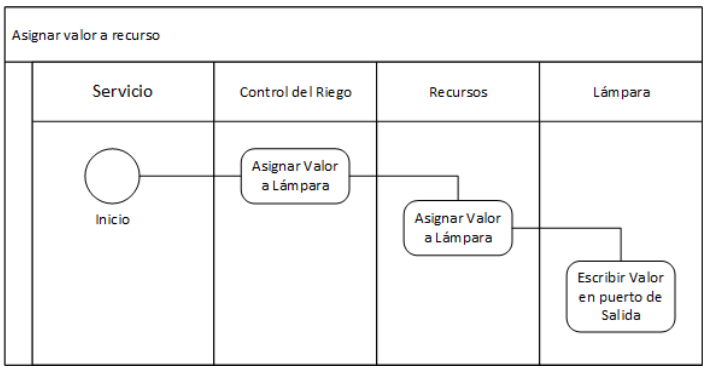

Figura 4.25. *Diagrama de Actividad UML para asignar valor a un recurso virtual*

El diagrama de la figura 4.25 muestra el flujo de datos para asignar un valor a un recurso virtual. En este ejemplo, el recurso virtual es la lámpara. El comienzo del flujo lo da el componente de Servicio que es el que procesa la petición de los sistemas externos, es decir, la aplicación web o smartphone. El Administrador de Recursos itera sobre su colección de recursos hasta que lo encuentra y le envía el valor que se envió desde la capa de servicios para que se le asigne. La lámpara cuando lo recibe escribe el valor en el puerto de salida que tiene asociado.

El diagrama de la figura 4.26 muestra el flujo de datos para leer un valor de un recurso virtual. En este ejemplo, los recursos virtuales que aceptan lecturas son sensores: de temperatura, humedad e iluminación. El comienzo del flujo lo da únicamente el componente de Servicio y el Administrador de Recursos itera sobre su colección de recursos y le envía un mensaje al sensor en cuestión para que retorne su valor. El mismo lee el valor en su puerto de entrada y se lo devuelve al administrador de recursos. Este se lo devuelve a Control del Riego para que prepare la respuesta y finalmente sea enviada a través de la capa de Servicios.

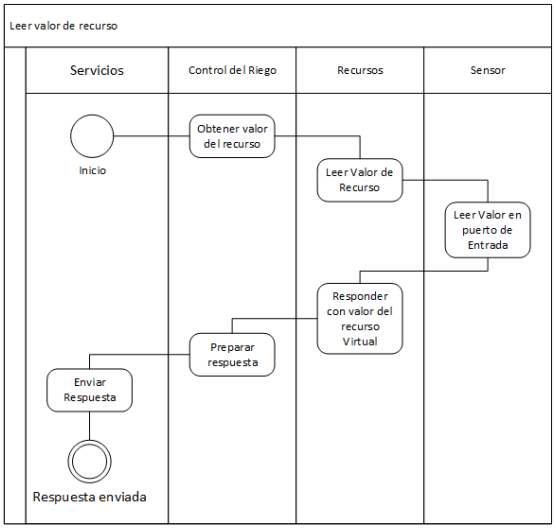

Figura 4.26. *Diagrama de Actividad UML para leer un valor de un recurso virtual*

El diagrama de la figura 4.27 muestra el flujo de datos para ejecutar una tarea programada. El comienzo del flujo lo da el componente Control del Riego para que se delegue la petición en el Administrador de Eventos que itera por cada una de las tareas programadas verificando si las condiciones que tiene son verdaderas o falsas, esto es, solicitar el valor del recurso asociado y compararlo con el valor esperado. Si las condiciones de una tarea programada son verdaderas, itera por cada una de sus acciones y las ejecuta. Si las condiciones no se cumplen, la tarea se omite.

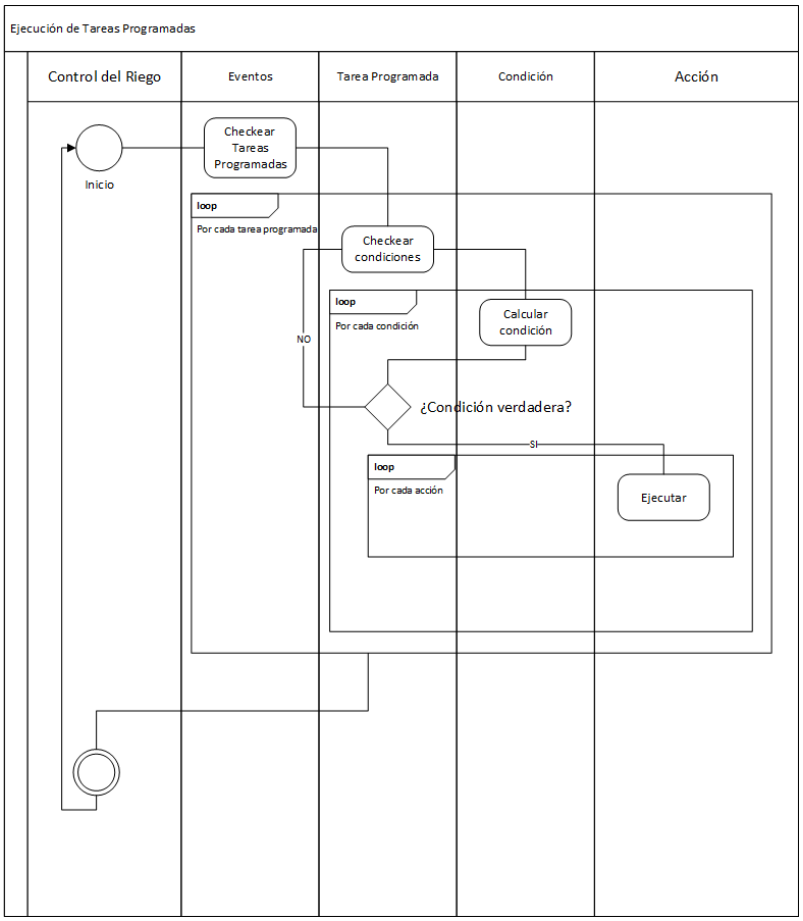

Figura 4.27. *Diagrama de Actividad UML para ejecutar una tarea programada*

4.4.2.5 Vista Física

La figura 4.28 muestra la vista física de la arquitectura concreta. La vista comienza con el usuario realizando peticiones al objeto inteligente a través de la aplicación web o smartphone. Embebidos en el objeto inteligente se encuentran el display LCD y la capa de servicios web, que resuelven las peticiones generadas por el entorno. Básicamente, a través de Sistema Controlador del Riego se delega el trabajo en los componentes de negocio y las componentes transversales, y los recursos virtuales ofrecen una interfaz hacia los recursos físicos del objeto inteligente, en este caso se trata

de un medidor de iluminación que lee valores sobre el sensor de iluminación, un medidor de humedad que lee valores sobre el sensor de humedad, un medidor de temperatura que lee valores sobre el sensor de temperatura y un controlador de lámpara que escribe valores sobre la lámpara.

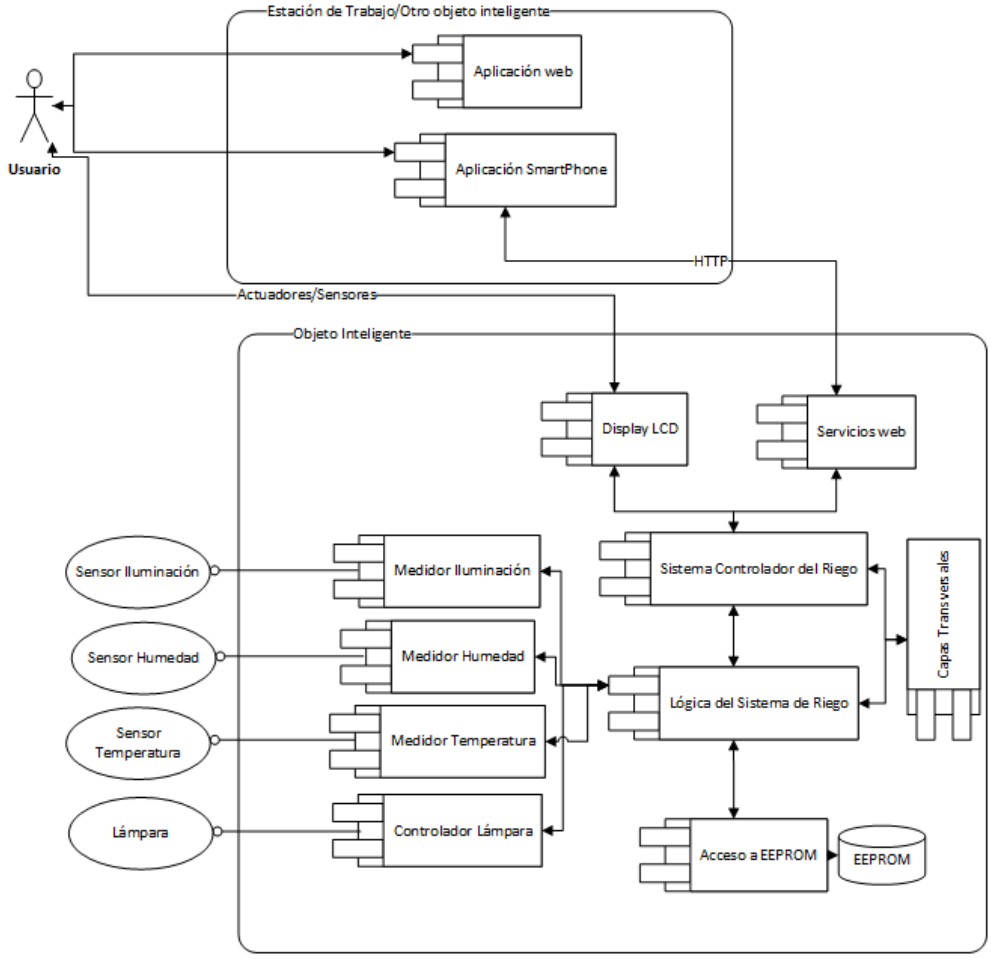

Figura 4.28. *Vista Física de la Arquitectura Concreta*

5. PRUEBA DE CONCEPTO

En este capítulo se presenta una prueba de concepto perteneciente a la arquitectura de software de referencia para objetos inteligentes en internet de las cosas. En la sección 5.1 se analiza el caso del desarrollo de una Plataforma Multipropósito de Telemetría y Telecomando a través de Internet basada en Sistemas Embebidos y en la sección 5.2 se muestra su solución basada en la arquitectura propuesta en el capítulo 4.

5.1. DESCRIPCIÓN DE PRUEBA DE CONCEPTO: Plataforma Multipropósito de Telemetría y Telecomando a través de Internet basada en Sistemas Embebidos

En esta sección se analiza el caso de validación correspondiente a la construcción de una Plataforma Multipropósito de Telemetría y Telecomando a través de Internet basada en Sistemas Embebidos. En la sección 5.1.1 se enuncia la descripción del negocio del caso de validación, en la sección 5.1.2 los requerimientos generales de la plataforma y en la sección 5.1.3 se muestran los requerimientos técnicos del sistema.

5.1.1. Descripción del Negocio

Esta prueba de concepto se desarrolló en el marco de un Proyecto de Investigación en el Laboratorio de Sistemas Industriales perteneciente al Grupo de Investigación en Sistemas de Información en la Universidad Nacional de Lanús. En su presentación [Azcurra, D. 2013] menciona: “En múltiples procesos de la actividad humana, ya sea en el ámbito industrial, el productivo, el comercial, el gubernamental, el agrícola, el de salud o el doméstico puede resultar necesario o conveniente disponer de mecanismos de medición o monitoreo remoto de determinados sensores y/o de control a distancia de ciertos dispositivos. A modo de ejemplo se puede citar la administración centralizada de varias máquinas, la supervisión remota de una planta de producción o, en el ámbito doméstico, la implementación de sistemas de alarmas, de control de iluminación, de adecuación térmica o de control de riego. Actualmente el mercado satisface esta demanda de manera parcial, ya que muchas veces se trata de soluciones muy costosas, orientadas a mercados específicos o implican quedar “atados” con un proveedor. El objetivo general de este proyecto es el desarrollo de una plataforma multipropósito de telemetría y telecomando a través Internet basada en la tecnología de sistemas embebidos, caracterizada principalmente por:

- Ser aplicable para múltiples mercados.

- Ser versátil y flexible.
- Ser de fácil instalación y operación.
- Ser configurable por el usuario sin que este requiera de una formación especializada.
- Ser segura y robusta.
- Ser de bajo costo.
- Funcionar independiente de proveedores de servicios.
- Funcionar a través de Internet.
- Permitir el acceso desde cualquier navegador web estándar o teléfono celular."

Además, menciona que la plataforma está prevista para ser incorporada en "diversos sectores de la sociedad, como ser el industrial -particularmente en el sector de pequeñas y medianas empresas-, el comercial y el doméstico." y que aquellos que se dediquen al desarrollo de soluciones innovadoras, pueden encontrar en la misma un "vehículo para nuevas aplicaciones".

5.1.2. Requerimientos Generales de la Plataforma

Los requerimientos generales definen mediante un lenguaje natural, junto con diagramas, lo que el sistema debe hacer desde el punto de vista de un usuario [Sommerville, 2011]. Partiendo de esta definición, el objetivo del proyecto fue desarrollar una plataforma basada en sistemas embebidos mediante la cual el usuario pueda automatizar y controlar las diversas actividades que realiza, de forma remota. Debe actuar como controlador ante dispositivos de entrada y salida tales como sensores y teclados, actuadores y visualizadores, respectivamente. Se espera que, en distintos sectores del mercado, su uso tenga ventajas como [Azcurra, D. 2013]:

- Aumento de productividad
- Mejora en el uso de recursos energéticos
- Mayor cuidado de los recursos naturales
- Reducción de costos
- Prevención de accidentes
- Aumento de la competitividad a través de la actualización tecnológica
- Rápido acceso a la información

Las tres principales características de la plataforma son ser multipropósito, de bajo costo y de fácil comunicación. Además se espera que la plataforma sea escalable vertical y horizontalmente, disminuyendo el impacto de nuevos requerimientos funcionales y la posibilidad de interactuar con nuevos componentes sin tener que realizar un proceso de reingeniería.

Para una mejor comprensión de los objetivos del proyecto, se realizó una arquitectura general de la plataforma (Figura 5.1). Esto es, sin mayor grado de detalle, un panorama de alto nivel de la arquitectura del sistema, que muestra la distribución de funciones a través de los módulos del mismo.

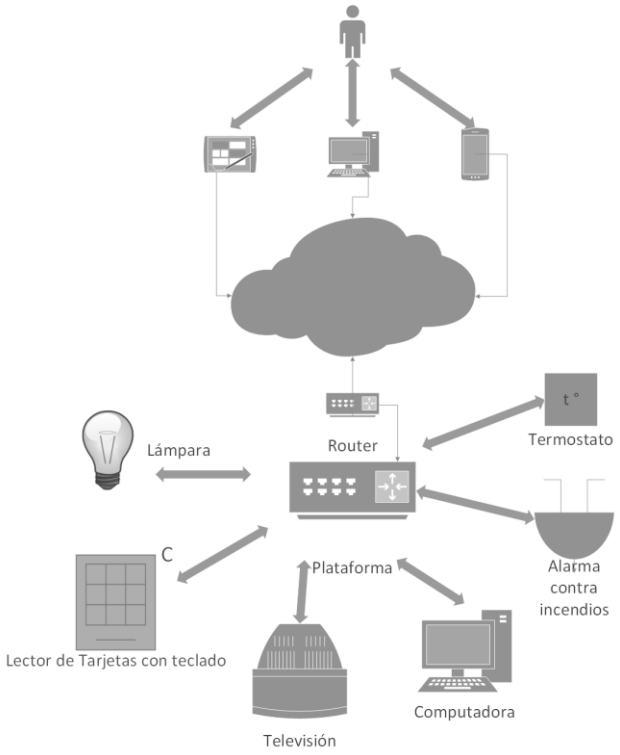

Figura 5.1. *Arquitectura del Sistema de la prueba de concepto*

5.1.3. Requerimientos del Sistema

Los requerimientos del sistema definen, de forma detallada y en un nivel técnico, lo que el sistema debe hacer para satisfacer los requerimientos descriptos en la sección 5.1.2. Los requerimientos son:

- El sistema permitirá que los usuarios puedan administrar los dispositivos que deseen controlar y/o medir mediante la plataforma, realizando altas, bajas y modificaciones de los mismos.
- El sistema tendrá un usuario administrador.
- Cada grupo de usuarios tendrá asociado un perfil de acceso.
- El sistema permitirá la creación de nuevos usuarios.
- El sistema permitirá que los usuarios obtengan información sobre los dispositivos conectados a la plataforma (como por ejemplo la visualización de datos).
- El sistema almacenará la información mencionada en el ítem anterior de los distintos dispositivos conectados así como las acciones que realicen los usuarios. Estas acciones

incluyen: ABM dispositivos, ABM usuarios, inicio de sesión, cierre de sesión, reinicio de la plataforma, actualización de configuración y log de eventos.

- La plataforma tendrá un módulo de log in.
- La plataforma permitirá el control de los dispositivos conectados. Este control incluye asignarle un valor al dispositivo.
- La plataforma permitirá la programación de tareas y eventos.
- La plataforma permitirá realizar un backup de la configuración.
- La plataforma permitirá el acceso a usuarios desde aparatos móviles como celulares y tablets.
- La plataforma tendrá un esquema transaccional. No queda en estados intermedios. Es decir, se perderán aquellas tareas que no se hayan completado antes de que la plataforma se reinicie, se apague o que el usuario haya perdido la conexión o cerrado sesión.
- La plataforma permitirá realizar un backup de la configuración necesaria para la migración hacia otra plataforma.

5.2. SOLUCIÓN DEL CASO DE PRUEBA DE CONCEPTO

En esta sección se propone una solución al caso de validación correspondiente a la construcción de una Plataforma de Telemetría y Telecomando a través de Internet basada en Sistemas Embebidos, basada en la arquitectura de referencia propuesta en el capítulo 4. Para resolver los requerimientos del caso enunciado en la sección 5.1, se optó por desarrollar una plataforma programada como servidor web que trabaje con el protocolo HTTP interactuando con los distintos clientes (como navegadores web, aplicaciones móviles y sistemas empresariales) que consumen su información, a través de servicios web. En la sección 5.2.1 los resultados de la fase de diseño, incluida la arquitectura concreta para el presente caso de validación y en la sección 5.2.2 los resultados de la codificación del software.

5.2.1. Diseño del Software

Se entiende al diseño arquitectónico como un proceso mediante el cual se entiende cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema [Sommerville, 2011]. Este proceso tiene como resultado un diseño de datos, uno arquitectónico, uno de interfaz y uno de componentes [Pressman, 2002]. En esta sección solo se muestran los resultados del proceso de diseño arquitectónico (Sección 5.2.1.1).

5.2.1.1. Arquitectura del Software

En esta sección se muestra la arquitectura concreta para la prueba del concepto propuesta en la sección 5.1. En la sección 5.2.1.1.1 se presenta un vistazo general de la arquitectura concreta, en la sección 5.2.1.1.2 la arquitectura concreta completa y en la sección 5.2.1.1.3 la relación entre los componentes.

5.2.1.1.1. Vistazo General

El vistazo general de la arquitectura se puede observar en la figura 5.12. La misma comienza con una capa de aplicación web que interactúa directamente con la capa de servicios, que expone la funcionalidad del negocio a través de servicios web. La capa de presentación se ocupa de atender los eventos ocurridos en el botón de reset y también controla el encendido y apagado de LEDs que permiten visualizar el estado de la plataforma. La capa Plataforma de Telemetría y Telecomando es la capa de negocios que encapsula las funcionalidades que satisfacen los requerimientos definidos en la sección 5.1.3. Luego, la capa acceso a datos se encarga de la lectura y escritura sobre memoria EEPROM y microSD. Por último, los recursos virtuales son cinco:

5. Pin: Que puede trabajar como actuador o sensor dado que se conecta a puertos de entrada y salida que deben conectarse mediante cables. Esto permite alcanzar el concepto multipropósito de la plataforma dado que no se puede conectar cualquier tipo de dispositivo, actuador o sensor.
6. Medidor de Humedad: Que efectúa mediciones sobre el sensor de humedad. Es una implementación particular para un sensor de humedad ampliamente utilizado en la electrónica: DHT11.
7. Medidor de Temperatura: Que efectúa mediciones sobre el sensor de temperatura. Es una implementación particular para un sensor de temperatura ampliamente utilizado en la electrónica: DHT11.
8. ZigBee: Que otorga conectividad con dispositivos ZigBee, estándar de comunicación inalámbrica para transferencia de datos de bajo consumo. No fue implementado para esta prueba de concepto.
9. X10: Que otorga conectividad con dispositivos X10, protocolo de comunicación que utiliza la red eléctrica para la transferencia de datos. No fue implementado para esta prueba de concepto.

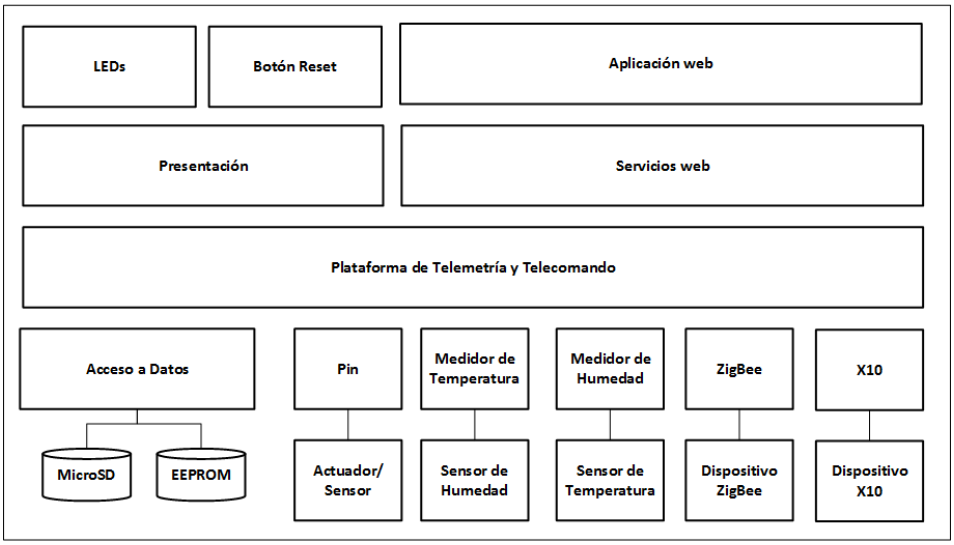

Figura 5.2. *Vistazo general de la arquitectura de la prueba del concepto*

5.2.1.1.2. Arquitectura Completa

La vista de la arquitectura mostrada a continuación (Figura 5.3) parte de la arquitectura planteada en la sección 5.2.1.1.1, aumentando el nivel de detalle de los componentes. En las secciones 5.2.1.1.2.1 a 5.2.1.1.2.5 se explican las funcionalidades de la capa presentación, servicios, lógica de negocio, capas transversales, dispositivos y acceso a datos, respectivamente.

5.2.1.1.2.1. Capa de Presentación

La capa de presentación tiene una sola subcomponente y es la librería de Arduino que permite acceder al hardware de entrada y salida por los puertos digitales y analógicos del microcontrolador (Para esta prueba de concepto se utilizó un Arduino Mega). Esta capa permite conocer el estado del botón de reset y controlar el estado de los LEDs de la plataforma.

5.2.1.1.2.2. Capa de Servicios Web

La capa de Servicios Web que expone la funcionalidad del sistema hacia la aplicación web, está compuesta por dos subcapas: La primera, una API (Del inglés, Application Programming Interface) RESTful, que efectivamente envía y recibe datos hacia y desde el exterior a través de servicios web REST a través del protocolo de comunicación HTTP, y la segunda, RAML, que expone información necesaria para que los sistemas externos sepan cómo está estructurada la capa de servicios y así poder consumirla.

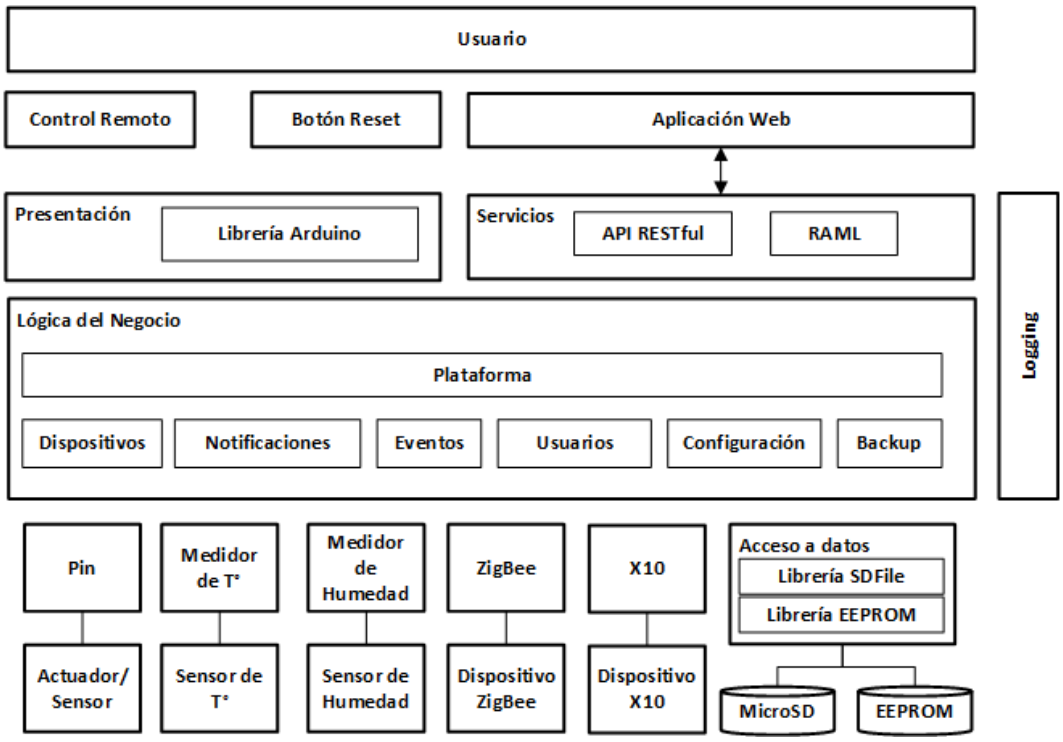

Figura 5.3. *Vista detallada de la arquitectura de la prueba del concepto*

5.2.1.1.2.3. Capa de Negocio

La capa de Plataforma de Telemetría y Telecomando se divide en varios componentes que dan soporte a las funcionalidades del sistema. Estos son:

- Dispositivos: Es el módulo de recursos que permite la medición y el control a distancia de los distintos tipos de dispositivos. Permite el alta, baja y modificación de los mismos.
- Notificaciones: Permite el intercambio de información del contexto de la plataforma y de lo que está sucediendo en tiempo real.
- Eventos: Este componente es el que permite la programación de tareas y gestión de eventos.
- Usuarios: Es el módulo que gestiona el acceso a la plataforma. Además de contar con un usuario administrador, permite la creación y modificación de nuevos y otros usuarios.
- Configuración: Este módulo permite personalizar la configuración de la plataforma. Incluye datos de red como dirección IP y MAC.
- Backup: Componente que efectúa copias de resguardo de la configuración de la plataforma.

Por último, la capa Plataforma encapsula todas las funcionalidades de las capas mencionadas anteriormente ofreciendo una única interfaz a las capas de presentación y servicios.

5.2.1.1.2.4. Capas Transversales

Si bien por definición se incluyeron tres capas transversales (Seguridad, Validación y Logging), para la prueba del concepto solo fue necesario incluir una capa de Logging dado que los requerimientos no exigían grandes políticas de seguridad o validación. Dicho esto, la capa transversal a toda la lógica de negocio es la capa de Logging, que ofrece la funcionalidad de registrar lo que sucede en el entorno de la plataforma. Esto puede ser actividad del usuario o eventos que suceden a partir de tareas programadas.

5.2.1.1.2.5. Capas de Dispositivos y Acceso a Datos

Para esta prueba de concepto se diseñaron cinco capas de dispositivos: Pin, Medidor de Temperatura, Medido de Humedad, ZigBee y X10. Cada una de estas, incluye librerías y software específico para que se efectúe la comunicación entre el recurso virtual y el físico. Por ejemplo, el Medidor de temperatura en este caso contiene una librería desarrollada por la comunidad de Arduino que trabaja con el sensor de Temperatura DHT11. El dispositivo del tipo Pin contiene la librería Arduino que le permite comunicarse con los pines del microcontrolador. Lo mismo ocurre con ZigBee y X10 que, aunque no fueron implementados en el desarrollo del prototipo, por diseño deben incluir librerías que permitan la comunicación mediante estos protocolos.
La capa de Acceso a Datos, por otra parte, debe persistir y leer datos de las memorias microSD y EEPROM. Para esto, incluye dos librerías: Librería SDFile y librería EEPROM, ambas desarrollada por la comunidad de Arduino.

5.2.1.1.3. Relación entre Componentes

En esta sección se describe cómo se relacionan los distintos componentes de la arquitectura de la sección 5.2.1.1. En las secciones 5.2.1.1.3.1 a 5.2.1.1.3.3 se documentan entonces la vista lógica, vista de procesamiento y vista física, respectivamente.

5.2.1.1.3.1. Vista Lógica

En la figura 5.4 se puede observar la vista lógica de la arquitectura de referencia. En las secciones 5.2.1.1.3.1.1 a 5.2.1.1.3.1.4 se explican por separado las relaciones entre los componentes más importantes.

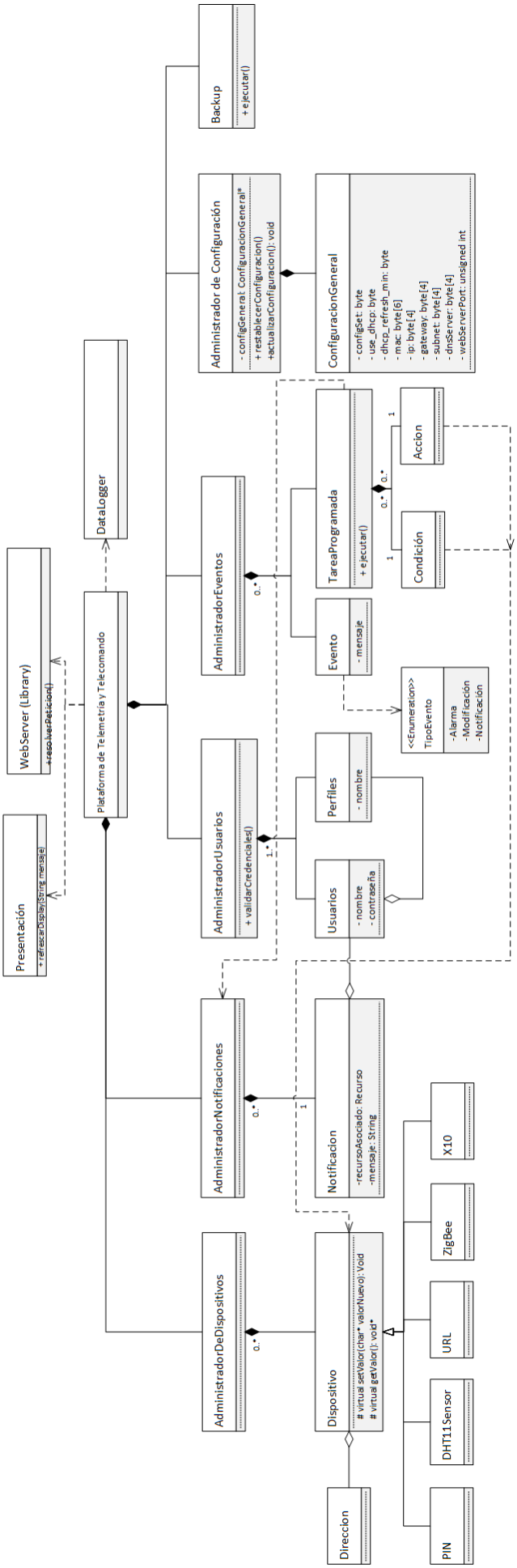

Figura 5.4. *Vista Lógica de la Arquitectura de la prueba del concepto*

5.2.1.1.3.1.1 Capas Superiores

En esta sección se explica la relación entre las capas de servicio, Application Facade y capas transversales (Figura 5.5).

La clase “Plataforma de Telemetría y Telecomando” (De ahora en adelante “Plataforma”) es la que en la definición de la arquitectura se llamó Application Facade y es la que encapsula las funcionalidades del resto de las capas de negocio. Para lograr esto, se agregan como atributos los distintos componentes de negocio: Administrador de eventos, de dispositivos y de usuarios, entre otros.

La clase “Webserver” es la que en la definición de la arquitectura se llamó Servicios y ofrece la interfaz necesaria para comunicarse con sistemas externos que, en este escenario, es una aplicación web y la interfaz es el protocolo HTTP. Esta clase está basada en la librería para Arduino llamada “Webduino” y puede encontrarse en [Webduino, 2012]. Webserver ofrece métodos como “enviar”, “imprimir” y “leer” para comunicarse con el cliente vía HTTP. Por otra parte, la metadata se expone mediante un archivo RAML [RAML, 2015] y los mensajes se envían en formato JSON.

Respecto las capas transversales, sólo una se encuentra desacoplada de la clase Plataforma y es la capa de Logging, que en este escenario se llama DataLogger. Las capas de Validación y Seguridad se encuentran embebidas en la clase Plataforma dado que su pequeño tamaño no justificó el armado de una nueva clase. La clase DataLogger puede recibir distintos argumentos como pueden ser un string, un usuario o un dispositivo. Estos datos son guardados en un archivo de log para dejar registrados distintos eventos.

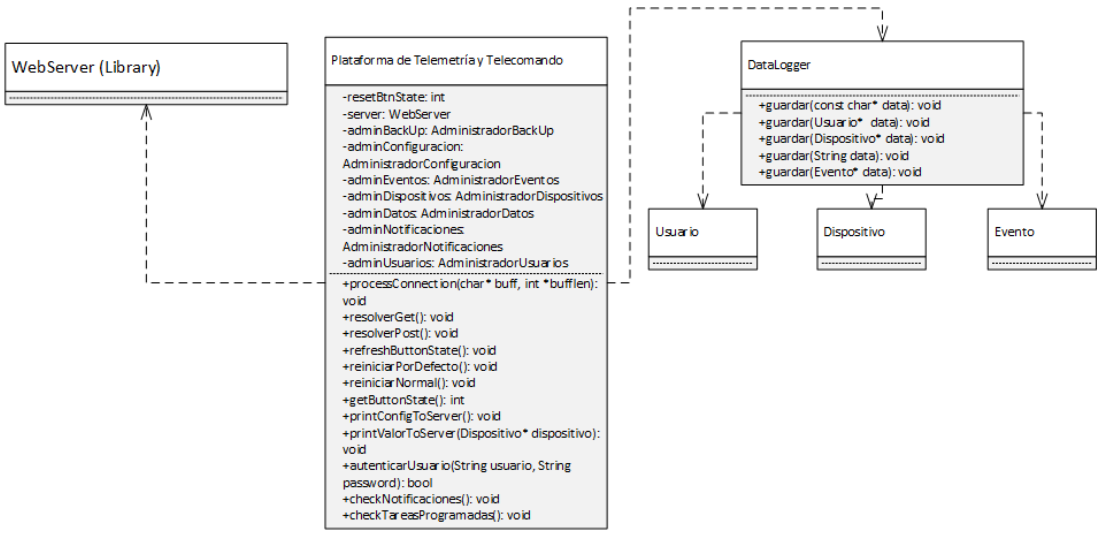

Figura 5.5. *Capas superiores de la vista lógica de la prueba del concepto*

5.2.1.1.3.1.2 Capas de Recursos

En esta sección se explica la relación entre dispositivos y plataforma. En la figura 5.6 se puede observar el diagrama de clases utilizado para demostrar esta relación. En la misma se identifican por lo menos cinco tipos de dispositivos:

- Pin, que representa aquellos dispositivos que se conectan a la plataforma mediante conectores simples (positivo y negativo)
- DHT11 Sensor, que representa un sensor particular que mide la temperatura y la humedad relativa. Se conecta también por pin pero la forma de medición varía para este sensor en particular y por eso se implementa en una clase separada. Este tipo de dispositivo es luego separado en DHT11Temp (Para medir temperatura) y DHT11Hum (Para medir humedad), dado que el procedimiento para leer la temperatura y la humedad varían (El sensor es el mismo).
- URL, que representa aquellos recursos o dispositivos que no se encuentran físicamente conectados a la plataforma pero se puede acceder a sus valores mediante la URL que proporcionan. Son los que en la definición de la arquitectura se llamaron recursos virtuales.
- ZigBee, que representa dispositivos ZigBee. También son recursos virtuales dado que no son conectados directamente a la plataforma.
- X10, que representa dispositivos X10. Si bien estos dispositivos son conectados físicamente a la plataforma, los dispositivos X10 también son considerados recursos virtuales.

Cada uno de estos recursos deberá implementar apropiadamente los métodos getValor() y setValor() definidos en la clase Dispositivos, como así también los métodos setEsDeEntrada() y getEsDeEntrada().

Por otra parte, tal como se mencionó en el capítulo de solución, la forma en la que los recursos virtuales efectivamente miden o asignan valores sobre los dispositivos físicos está dada a través de la dirección que se les provee. En este caso, la dirección está compuesta por un tipo de dirección que representa finalmente al tipo de dispositivo y un valor en formato de cadena de caracteres. Luego, cada dispositivo deberá conocer cómo resolver la dirección para leer o escribir valores sobre el recurso físico. Por ejemplo, en el caso del dispositivo de tipo Pin, el procedimiento para resolver la dirección es:

1. Leer el valor de la dirección. Este representa un puerto de entrada y salida de la plataforma.
2. Encontrar el puerto de entrada y salida a través del Administrador de Dispositivos.
3. Encontrar el puerto de entrada y salida del microcontrolador a través del puerto de entrada y salida de la plataforma.
4. Efectuar escritura o lectura sobre el puerto de entrada y salida del microcontrolador.

Este procedimiento dista completamente del que poseen los dispositivos URL, que una vez que encuentran el valor de la dirección, efectúan una petición a dicha URL (Puede ser una petición HTTP, por ejemplo) con los parámetros que dicho recurso virtual requiera.

Respecto al administrador de dispositivos, el mismo ofrece una interfaz para la instanciación de los mismos a través de la clase DispositivoFactory. La ventaja de utilizar un componente dedicado a esta tarea es que toda la lógica necesaria para la creación de dispositivos se centraliza en un solo punto. Por otra parte, el administrador de dispositivos cuenta con una lista de puertos de entrada y salida para la construcción de dispositivos del tipo Pin y posee una colección con los dispositivos creados y almacenados en la plataforma.

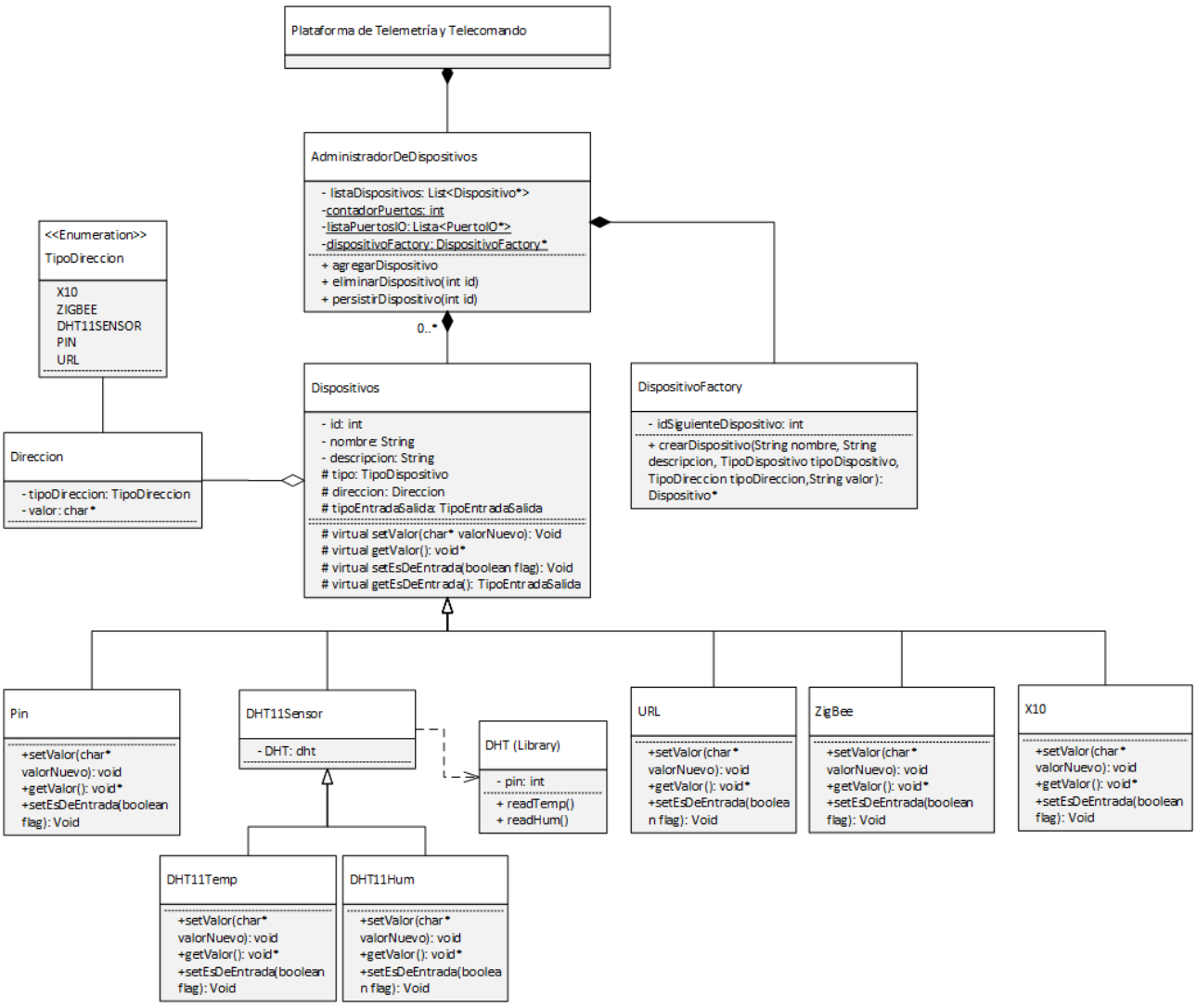

Figura 5.6. *Capas de dispositivos de la vista lógica de la prueba del concepto*

5.2.1.1.3.1.3 Capas de Credenciales

El esquema de credenciales está compuesto por una clase de Credenciales, Notificación, Usuarios y Perfiles (Figura 5.7). La clase de credenciales está compuesta por un conjunto de usuarios y perfiles

y provee una interfaz para validar las credenciales y autenticar a los usuarios mediante usuarios y contraseñas. Por otra parte, se comunica con el módulo de notificaciones dado que cada notificación tiene asociada un grupo de usuarios afectados.

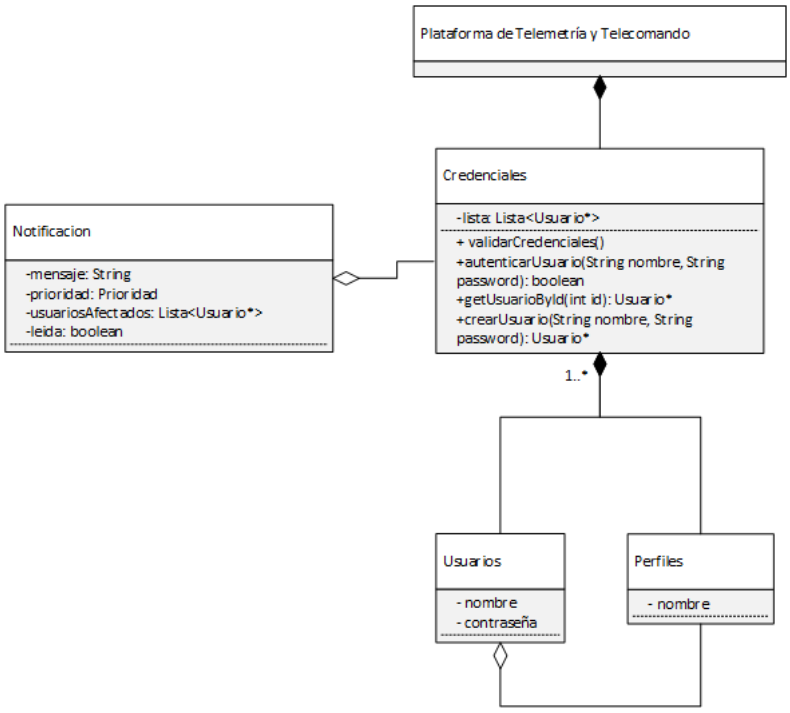

Figura 5.7. *Capas de credenciales de la vista lógica de la prueba del concepto*

5.2.1.1.3.1.4 Capas de Eventos

La clase Eventos es la que gestiona este módulo y está compuesta por una colección del tipo Evento y otra del tipo Tarea Programada. La clase Evento contiene información acerca del tipo de evento (Alarma, Modificación de alguna información del sistema o del valor de un dispositivo o notificación de alguna tarea programada), un mensaje, una descripción y una prioridad, que puede ser baja, media, alta o urgente. Las tareas programadas están compuestas por dos clases: Condición y Acción, que son clases padres y abstractas de las condiciones y acciones concretas, respectivamente. Por definición, la clase condición debe contener al menos dos operandos y un operador y debe proveer alguna interfaz para que se pueda evaluar su estado (Por verdadero o falso). La clase Acción simplemente debe proveer una interfaz para que la misma sea ejecutada. Dicho esto, las clases concretas de condiciones son dos:

- Condición Dispositivo-Dispositivo (CondiciónDD), que tienen como primer y segundo operando a un dispositivo. Esto es útil cuando se quieren comparar valores entre dispositivos, como la medición de temperatura en dos puntos en un terreno.

- Condición Dispositivo-Literal, que tienen como primer operando un dispositivo y segundo operando a un valor literal. Esto es útil cuando se quieren comparar los valores que mide el dispositivo contra un valor fijo.

Las clases de acciones también se dividen en dos:

- Acción por notificación, que permite enviar un mensaje a una lista de usuarios con determinada prioridad. El valor a enviar en el mensaje está definido en la clase abstracta Acción.
- Acción a dispositivo, que permite asignar un valor a un dispositivo. El valor que se le asigna es el definido en la clase abstracta Acción.

De esta manera se pueden ejecutar las acciones de una tarea programada, cuando las condiciones son verdaderas. Combinando todos los casos se pueden establecer tareas programadas de la siguiente forma:

> "Si la temperatura del frente es mayor que la temperatura del costado y la humedad es mayor al 80%, entonces encender la ventilación y notificar a los usuarios del primer piso de manera urgente que hay problemas de refrigeración en la sala."

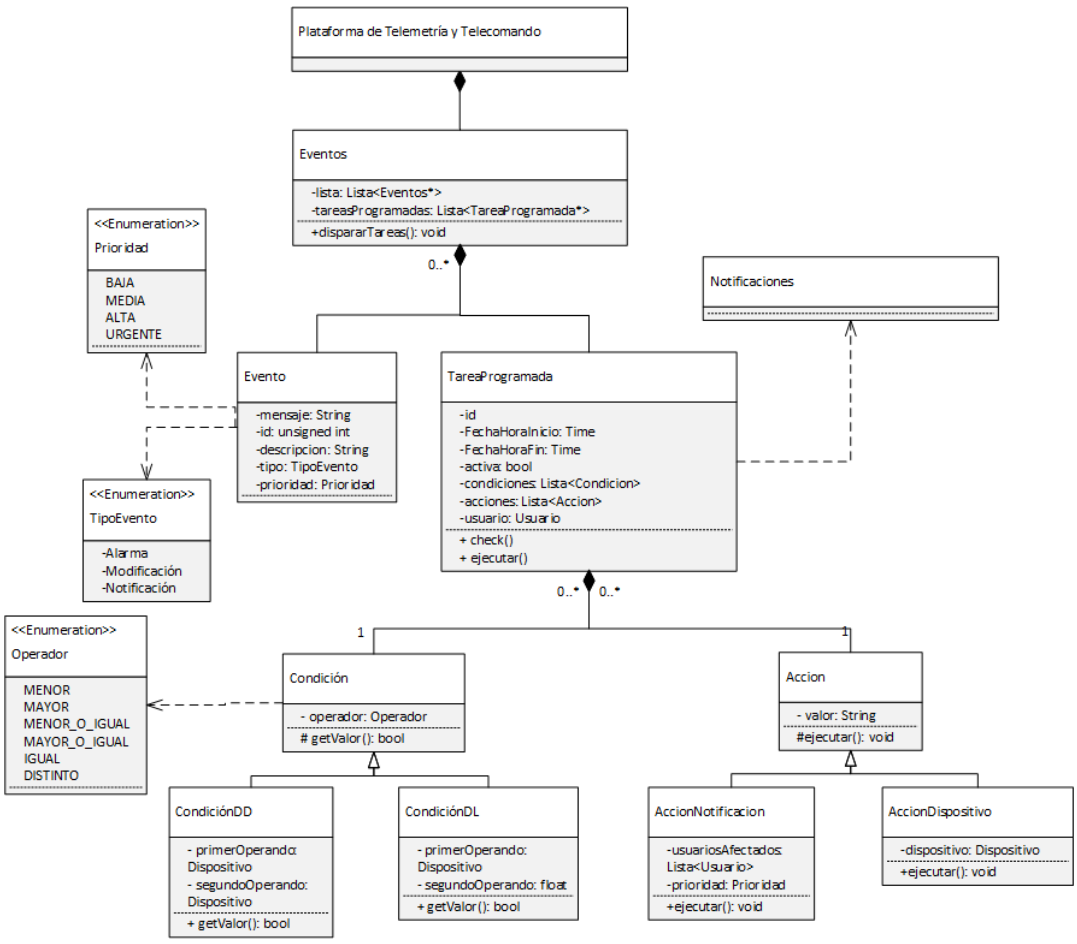

Figura 5.8. *Capas de eventos de la vista lógica de la prueba del concepto*

5.2.1.1.3.2. Vista de Procesamiento

En esta sección se muestran los diagramas que componen la vista de procesamiento para la presente prueba de concepto. Cabe aclarar que fueron omitidos aquellos diagramas que no varían respecto a su definición en el capítulo de solución.

El diagrama de la figura 5.9 muestra el flujo de datos para verificar que un usuario tiene permisos para ejecutar la petición que realizó. El comienzo del flujo lo da únicamente el componente de Servicio, es decir, a través de peticiones de sistemas externos mediante servicios web. Dado que en los requerimientos no se hizo hincapié en la seguridad de la plataforma, su nivel de seguridad es relativamente bajo y es por este motivo que los componentes de validación y seguridad quedaron embebidos en la capa de la plataforma que procesa la solicitud.

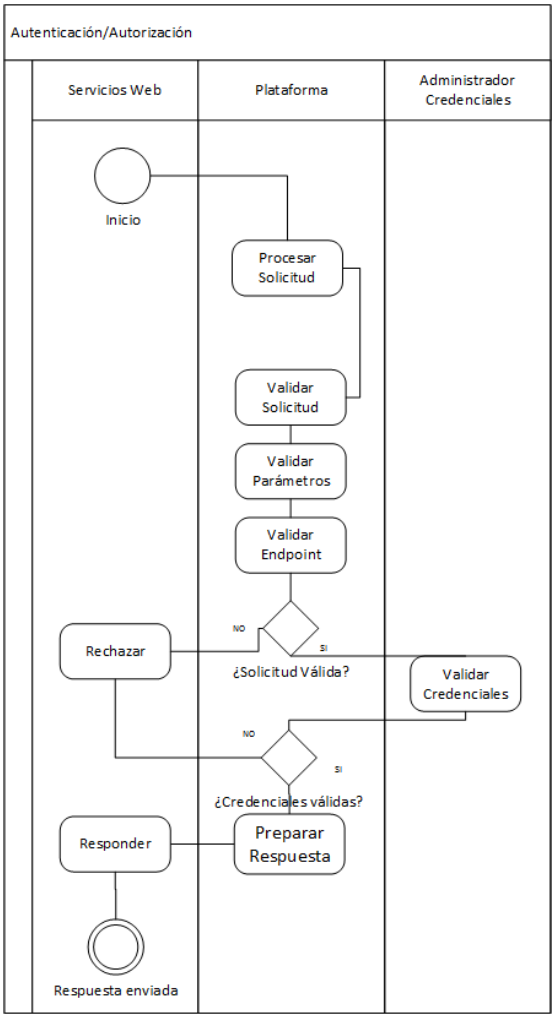

Figura 5.9. *Diagrama de Actividad UML para la autenticación y autorización (Prueba de concepto)*

El diagrama de la figura 5.10 muestra el flujo de datos para asignar un valor a un dispositivo. El comienzo del flujo lo da el componente de servicios que es el que procesa la petición de los sistemas externos. El Administrador de dispositivos itera sobre su colección de dispositivos hasta

que lo encuentra y le envía el valor que se envió desde la capa de servicios para que se le asigne. El dispositivo luego escribe el valor en el puerto de salida que tiene asociado.

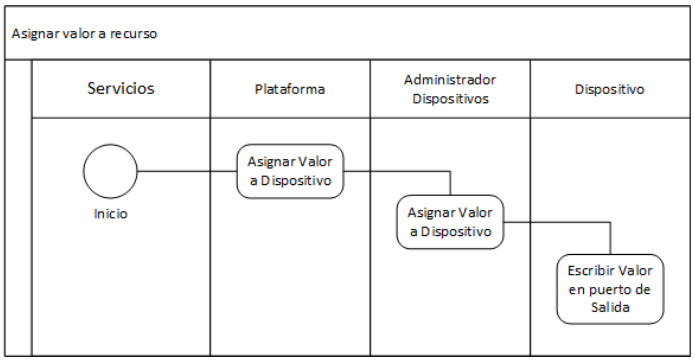

Figura 5.10. *Diagrama de Actividad UML para asignar valor a un dispositivo (Prueba de concepto)*

El diagrama de la figura 5.11 muestra el flujo de datos para leer un valor de un dispositivo. El comienzo del flujo lo da únicamente el componente de servicios y el Administrador de Dispositivos itera sobre su colección de dispositivos y le envía un mensaje al dispositivo en cuestión para que retorne su valor. El mismo lee el valor en su puerto de entrada y se lo devuelve al administrador de dispositivos. Este se lo devuelve a Plataforma para que prepare la respuesta y finalmente sea enviada a través de la capa de Servicios.

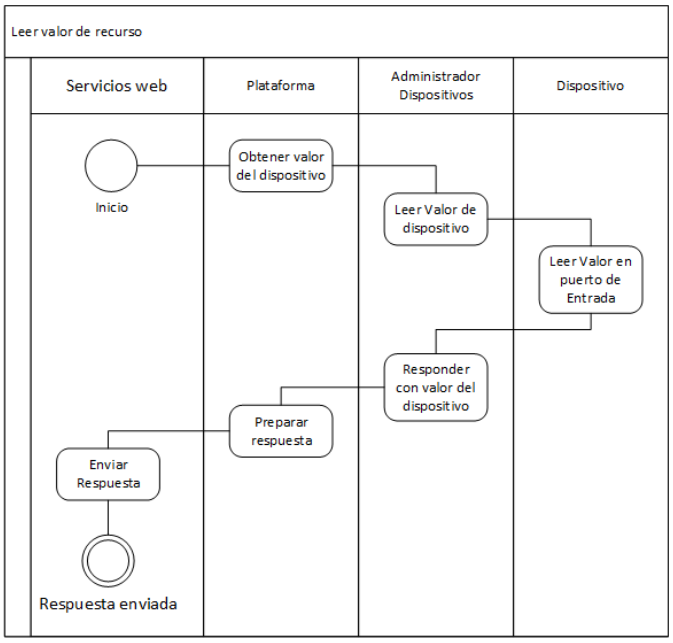

Figura 5.11. *Diagrama de Actividad UML para leer un valor de un dispositivo (Prueba de Concepto)*

El diagrama de la figura 5.12 muestra el flujo de datos para ejecutar una tarea programada. El comienzo del flujo lo da el componente Plataforma para que se delegue la petición en el Administrador de Eventos que itera por cada una de las tareas programadas verificando si las

condiciones que tiene son verdaderas o falsas. Para determinar el estado de cada condición, las mismas pueden seguir dos flujos dependiendo el tipo de condición:

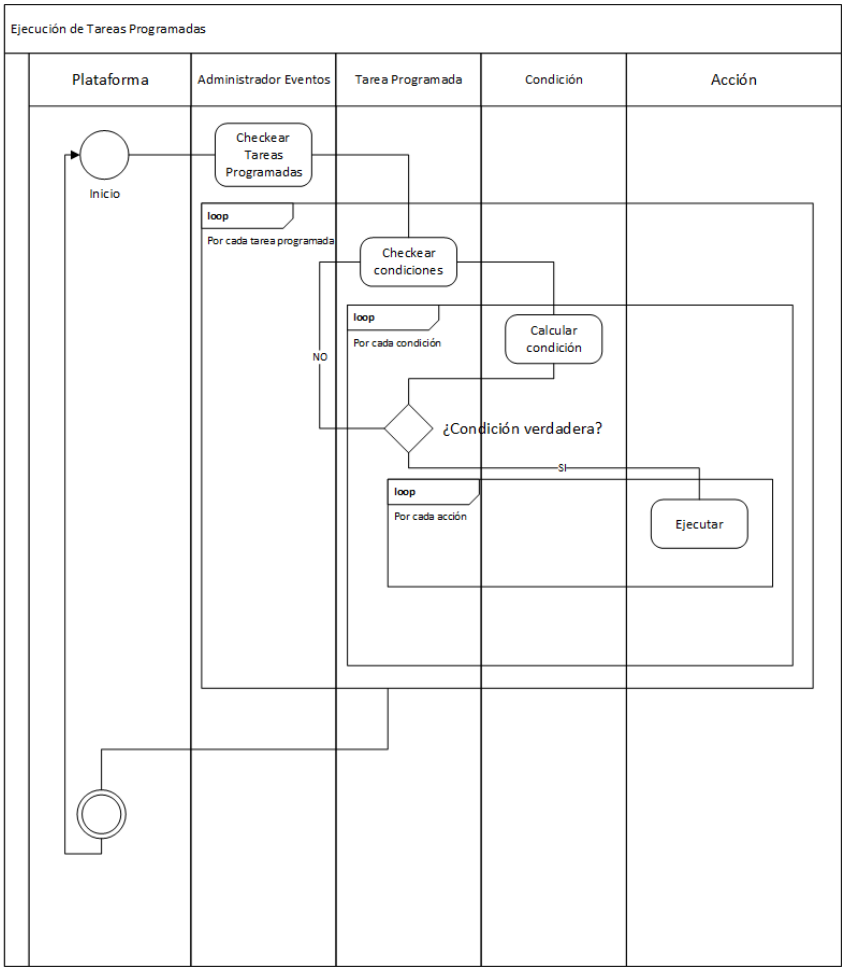

Figura 5.12. *Diagrama de Actividad UML para ejecutar una tarea programada (Prueba de concepto)*

- Condición Dispositivo – Dispositivo: Para resolver este tipo de condición se obtiene el valor de ambos dispositivos y luego se los compara mediante el operador de la condición (Figura 5.13).
- Condición Dispositivo – Literal: Para resolver este tipo de condición se obtiene el valor del dispositivo y luego se lo compara con el valor literal mediante el operador de la condición (Figura 5.14).

Si las condiciones de una tarea programada son verdaderas, se itera por cada una de las acciones de la tarea programada y se las ejecuta. Si las condiciones no se cumplen, la tarea se omite. Para ejecutar las acciones se pueden seguir dos procedimientos según el tipo de acción:

- Acción dispositivo: Para ejecutar este tipo de acción se obtiene el valor a asignar y se lo asigna al dispositivo (Figura 5.15).

- Acción notificación: Para ejecutar este tipo de acción se obtiene el mensaje a enviar y se lo envía al administrador de notificaciones junto al conjunto de usuarios asignados (Figura 5.16).

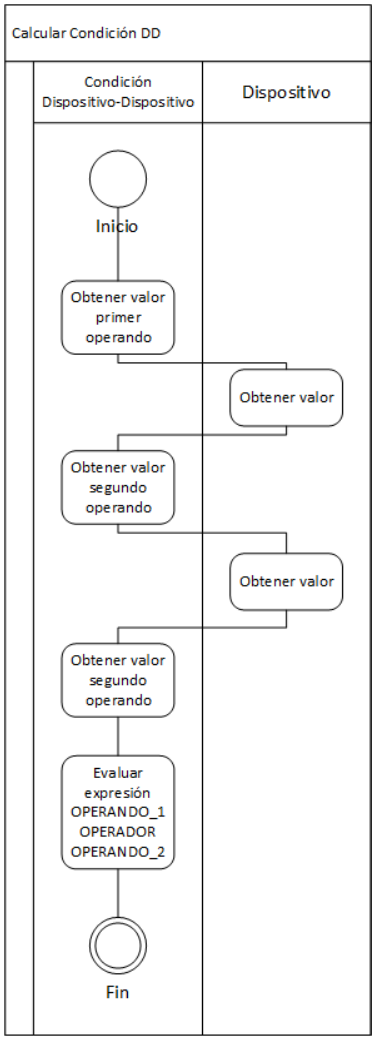

Figura 5.13. *Diagrama de Actividad UML para calcular el estado de una condición dispositivo - dispositivo*

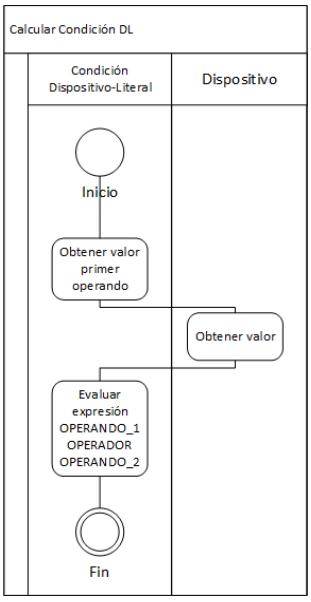

Figura 5.14. *Diagrama de Actividad UML para calcular el estado de una condición dispositivo – valor literal*

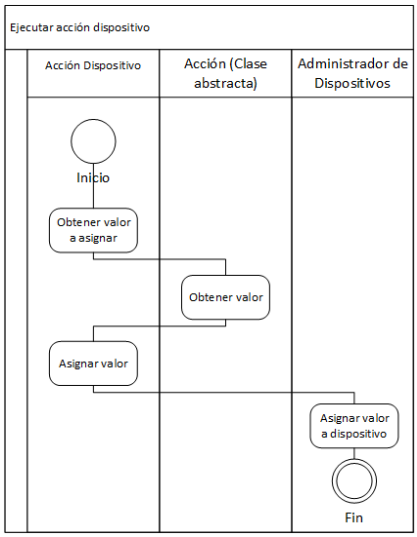

Figura 5.15. *Diagrama de Actividad UML para ejecutar una acción en un dispositivo*

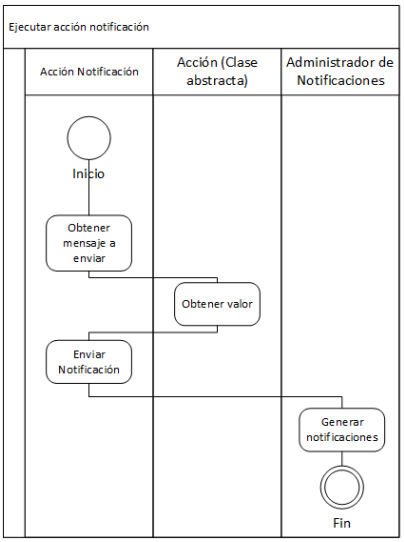

Figura 5.16. *Diagrama de Actividad UML para ejecutar una acción mediante una notificación*

5.2.1.1.3.3. Vista Física

La vista (Figura 5.17) comienza con el usuario realizando peticiones al objeto inteligente a través de actuadores, o sistemas externos como la aplicación web embebida en la plataforma u otras aplicaciones que consuman su información a través de servicios web. Básicamente, a través del componente Plataforma se delega el trabajo en los componentes lógicos de la plataforma y la capa de logging. Los dispositivos identificados son seis y cada uno tiene asociado un dispositivo. Cabe destacar que la relación entre los dispositivos virtuales y los dispositivos o recursos físicos no se limita a ser de uno a uno, sino que, como en el caso de los dispositivos DHT11, ambos usan el mismo sensor DHT11 para realizar mediciones, sólo cambia el mecanismo medición.

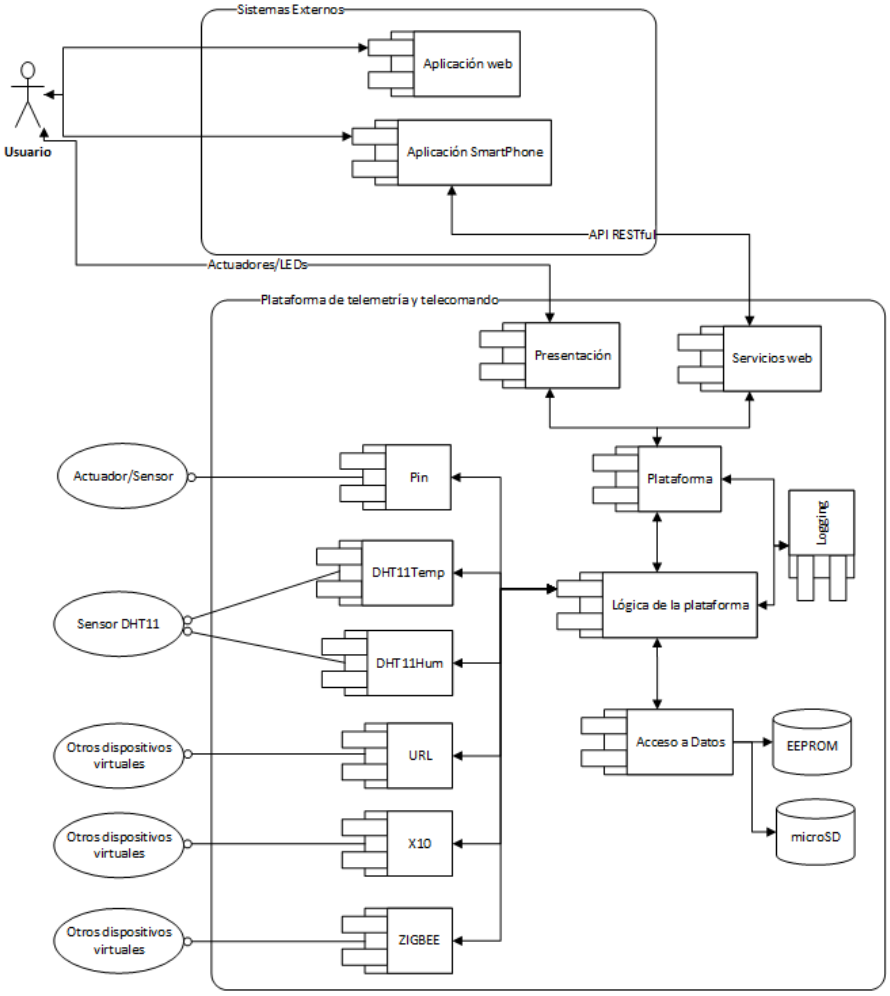

Figura 5.17. *Vista Física de la Arquitectura de la prueba de concepto*

5.2.2. Codificación

La codificación del firmware correspondiente a la arquitectura diseñada se llevó a cabo bajo el paradigma orientado a objetos, el cual modela el software en términos similares a los que utilizan las personas para describir objetos del mundo real como un vehículo, un animal o una persona [Deitel et al, 2008]. Si bien en el escenario en el que se desenvuelve este trabajo no es del todo tangible, es decir, no representan objetos que se puedan visualizar en la vida cotidiana dado que no se puede tocar un administrador de eventos, este enfoque permite que se desarrolle código simple de mantener y comprender. Por otra parte, la codificación se llevó a cabo utilizando el lenguaje de programación C/C++, visto que el hardware que se utilizó para la plataforma es compatible con la tecnología Arduino, programable en C/C++. El fabricante del hardware utilizado provee una implementación de lenguaje C que nos abstrae de la complejidad relacionada con el manejo de hardware a bajo nivel. Los programas en Arduino tiene una estructura fija: un setup y una función

loop que se ejecuta infinitamente. Un típico programa de Arduino para prender y apagar un led tiene la siguiente forma:

```
int led = 13;

// La función setup que se invoca luego de reiniciar el arduino
void setup() {
pinMode(led, OUTPUT);
}
// la función loop que se invoca una y otra vez hasta que el arduino se apague
void loop() {
digitalWrite(led, HIGH);
delay(1000);
digitalWrite(led, LOW);
delay(1000);
}
```

En el caso de la plataforma el código tiene la siguiente forma:

```
Plataforma* plataforma;
void setup()
{
   plataforma = new Plataforma();
   // otro código de inicialización
}
void loop()
{
   char buff[64];
   int len = 64;
   plataforma->processConnection(buff, &len);
   plataforma->checkNotificaciones();
}
```

El código fuente desarrollado en C/C++ puede separarse en dos archivos: Archivo encabezado (con extensión .h) y Archivo fuente (con extensión .cpp). La ventaja de esto es principalmente la separación entre la interfaz y la implementación, es decir, se separa lo que hace de cómo lo hace. En el capítulo 8, se muestran como quedan los componentes principales de la arquitectura codificados en lenguaje C++ bajo el paradigma orientado a objetos. Por una cuestión de espacio, sólo se muestran los archivos de encabezado.

6. CONCLUSIONES

En este Capítulo se presentan las aportaciones de este trabajo final de licenciatura (sección 6.1) y se formulan las futuras líneas de investigación que se consideran de interés en base al problema abierto que se presenta en este trabajo (sección 6.2).

6.1. APORTACIONES DEL TFL

En esta sección se presentan los aportes derivados de las preguntas de investigación del TFL (Sección 6.1.1) y las conclusiones generales (Sección 6.1.2).

6.1.1. Aportes derivados de las preguntas de investigación del TFL

A continuación se procede a responder a las preguntas que se formularon en la sección 3.3 correspondiente al Sumario de Investigación.

Pregunta 1: ¿Puede desarrollarse una arquitectura de software para sistemas embebidos que pueda utilizarse como referencia o punto de partida para el desarrollo de soluciones en internet de las cosas? En caso afirmativo: ¿Cuál es su estructura y qué elementos la componen?

En este trabajo se ha demostrado por construcción que fue posible desarrollar una arquitectura de software para sistemas embebidos tal como se muestra en el capítulo 4; en el cual se ha presentado una arquitectura en capas basada en el modelo propuesto en [Microsoft Patterns & Practices Team, 2009].

En líneas generales (Sección 4.2.1), la arquitectura propuesta consta de cuatro niveles:

- Dos capas superiores: Presentación y Sistemas Externos
- Capa de servicios
- Capa de lógica de negocio
- Capa de recursos virtuales, recursos físicos y acceso a datos

Las dos capas superiores que son las de presentación y sistemas externos (Sección 4.2.1.1) abordan la interacción del objeto inteligente con entidades externas como personas u otro software; la primera capa interactúa directamente con el usuario final a través de sensores y actuadores mientras que la segunda capa representa otro software que puede estar desplegado en un servidor, en la nube u otro objeto inteligente.

Luego, la capa de servicios (Sección 4.2.1.2) tiene como objetivo exponer la funcionalidad del objeto inteligente, implementada en las capas inferiores, a través de uno o más protocolos de comunicación (Como HTTP) o servicios (Como servicios web). Es importante que en esta capa se haga uso de estándares, notaciones y buenas prácticas de ingeniería de software de forma tal que la integración con sistemas externos sea más sencilla.

La capa de negocio (Sección 4.2.1.3) contiene el modelo de datos necesario para satisfacer las necesidades del negocio, es decir, los requerimientos. También contiene cierta lógica y reglas que son las que finalmente producen la información necesaria para que pueda ser distribuida por la capa de servicios o informada a través de la capa de presentación.

Las capas de recursos virtuales, recursos físicos y acceso a datos (Sección 4.2.1.4) son principalmente capas de bajo nivel de donde se obtienen los recursos necesarios para que el objeto inteligente produzca información. En estas capas se encuentran sensores y actuadores, como así también archivos o bases de datos y el acceso a otros objetos inteligentes.

Los recursos físicos son entidades del mundo físico tangible que por lo general permiten la obtención de información del contexto en donde se encuentran desplegados. Un ejemplo de recurso físico es un sensor de temperatura.

Los recursos virtuales son representaciones abstractas generalmente de recursos físicos de los cuales se quiere obtener información. Estas representaciones están en realidad desplegadas en el software por medio de objetos o estructuras de datos que a través de métodos o funciones puedan efectuar un intercambio de datos con el recurso físico. Por ejemplo, el recurso virtual que represente a un sensor de temperatura deberá tener la capacidad de efectuar mediciones sobre el mismo.

La capa de acceso a datos provee funcionalidades para almacenar datos temporal o permanentemente en el objeto inteligente como configuraciones o datos necesarios para que el software pueda ejecutarse. El objetivo de esta capa es abstraer al resto de los componentes de los detalles de implementación del repositorio de datos, ya que los datos pueden ser almacenados en distintos tipos de memoria como EEPROM o FLASH.

Luego, para cada una de las capas previamente enunciadas se describieron los subcomponentes que poseen (Sección 4.2.2):

- Capa de Presentación (Sección 4.2.2.1) que posee dos subcomponentes:
    - Hardware I/O que tiene como objetivo controlar la entrada y salida por hardware (Como LCDs inteligentes y teclados) y
    - Lógica de Presentación cuyo objetivo es ordenar al componente Hardware I/O la manera en que tiene que comportarse.

  La existencia de estos dos componentes permite tener en el sistema embebido un patrón de diseño similar a MVC.

- Capa de Servicios (Sección 4.2.2.2) que posee dos subcomponentes:
    - Interfaz del Servicio que ofrece los endpoints necesarios para que los servicios puedan ser consumidos
    - Metadata que expone información acerca del servicio para conocer de antemano cómo se debe consumir el servicio.
- Capa de Lógica de Negocio (Sección 4.2.2.3) que, si bien es cierto que los requerimientos varían de un escenario a otro, la realidad es que por lo menos se pueden identificaron los siguientes subcomponentes:
    - Recursos que proveen información sobre su contexto a través de la medición y control de los recursos que el objeto inteligente tiene a su alcance. Esta información es enviada a través de la capa de servicios o presentación.
    - Notificaciones que proveen información sobre lo que sucede en el entorno en tiempo y forma de forma tal que se puedan tomar decisiones con información precisa.
    - Credenciales que se encargan de mantener políticas de seguridad y así evitar el acceso no autorizado a determinados recursos del objeto inteligente.
    - Configuración que permite la parametrización del objeto inteligente de forma tal que se pueda adecuar al contexto en el que se va a utilizar. Esta configuración, dependiendo del objeto inteligente, podría incluir dirección IP, dirección MAC, un nombre, entre otros.
    - Backup que se encarga de cuestiones de manejo y recuperación de errores al establecer copias de resguardo de la información del objeto inteligente.
- Capas Transversales (Sección 4.2.2.4) que ofrecen funcionalidades de interés a varias capas como lógica de negocio y servicio. La cantidad de capas transversales varían según el escenario pero en términos generales se identificaron los siguientes subcomponentes:
    - Seguridad que se encarga de la veritifación de certificados y cifrado de información.
    - Logging que se encarga de mantener un historial de los eventos ocurridos durante el procesamiento de datos.
    - Validación que se encarga de ofrecer servicios de validación para varios escenarios como por ejemplo validación de fechas.
- Capa de Recursos Físicos, Recursos Virtuales y Acceso a Datos (Sección 4.2.2.5) en donde se identifican los siguientes subcomponentes:
    - Drivers o Librerías del recurso virtual que permiten obtener el software necesario para interactuar con el recurso físico asociado.
    - Drivers que permiten que el componente de acceso a datos efectivamente acceda al repositorio de datos.

- Helpers que dan soporte para la extracción y transformación de datos desde el repositorio.

Pregunta 2: ¿De existir tal arquitectura, es posible desarrollar vistas arquitectónicas que definan las relaciones entre sus componentes? De ser posible: ¿Cuáles?

Sí, en la sección 4.3 se ha demostrado por construcción que fue posible desarrollar vistas arquitectónicas para definir las relaciones entre los componentes de la arquitectura planteada en el capítulo 4; haciendo referencia al modelo 4+1 de [Kruchten, P., 1995].

Las vistas arquitectónicas construidas para esta arquitectura son tres:

- Vista Lógica (Sección 4.3.1) que representa un soporte a los requerimientos funcionales, es decir, lo que el objeto inteligente debe proveer a los usuarios en términos de servicios. Para documentar esta vista se utilizaron diagramas de clases UML.
- Vista de Procesamiento (Sección 4.3.2) que representa el flujo de datos en el procesamiento de datos del objeto inteligente especificando los algoritmos que se utilizan para ofrecer las funcionalidades requeridas. Para documentar esta vista se utilizaron diagramas de actividades UML.
- Vista Física (Sección 4.3.3) que representa cómo se distribuyen los componentes entre los distintos nodos del sistema, es decir, muestra cómo se ubica cada parte del software en un nodo de forma tal que se mapeen software y hardware. Para esta vista se utilizaron diagramas de despliegue UML.

6.1.2. Conclusiones Generales

La arquitectura presentada en este trabajo es una aportación a las arquitecturas existentes en la comunidad de internet de las cosas dado que al momento del desarrollo de este trabajo final de licenciatura no se han encontrado trabajos que abarquen la estructura que debe tener un software embebido para la construcción de objetos inteligentes que pertenezcan a este ecosistema.

En la sección 4.4 se ha aplicado la arquitectura propuesta con un ejemplo práctico y luego en el capítulo 5 con un caso de prueba de concepto, en donde se aborda la construcción de un software embebido para una plataforma multipropósito de telemetría y telecomando a través de internet.

Se estima que el diseño modular, basado en capas y componentes de la arquitectura propuesta que adaptar e implementar una solución con la misma sea rápido y fácil de probar haciéndola ideal no sólo para pruebas de conceptos en fases de prototipado sino también para implementaciones finales.

6.2. FUTURAS LÍNEAS DE INVESTIGACIÓN

Durante el proceso de investigación llevado a cabo en este trabajo final de licenciatura han surgido las siguientes futuras líneas de investigación:

I. Internet de las cosas es un ambiente heterogéneo en el cual no sólo conviven miles de objetos inteligentes sino también diversos sistemas de información los cuales sirven no sólo para consumir información sino también para proveerla. La integración de estos sistemas de información por lo general se da a través de servicios. En este contexto, se considera necesario incorporar un patrón de publish/suscribe (Publica/Suscribe) en la arquitectura de forma tal que permita la suscripción de servicios con el objeto inteligente.

II. Se considera de interés experimentar la flexibilidad de la arquitectura propuesta en términos de requerimientos o necesidades de negocio, ya que su estructura permite agregar, quitar o reemplazar componentes según sea necesario.

III. En varios trabajos de la comunidad de internet de las cosas disponibles en la literatura (A modo de ejemplo: [IOT-A, 2013]; [Misra, P., et al, 2015]; [Kortuem, G., et al, 2010]; [Holler, J., 2014]; [Guinard, D., 2011]; [Gubbi, J., et al, 2013]; [Chui, M., et al, 2010]; [Atzori, L., 2010], entre otros) se ha mencionado como eje importante el manejo de la seguridad en los objetos inteligentes en internet de las cosas. Esta clase de sistemas tiene una particularidad y es que, a diferencia de otros sistemas de información, altos niveles de seguridad pueden resultar perjudiciales para la ejecución del software embebido. Por ejemplo, en un sistema bancario tradicional se suelen asignar no sólo usuarios y contraseñas, sino también dispositivos electrónicos u otras formas de autenticación que elevan el nivel de seguridad de la cuenta bancaria. Sin embargo, en sistemas embebidos tener niveles tan altos pueden no ser deseables en el caso en que se necesite interrumpir inmediatamente la ejecución del software ya que lograr el acceso inmediato al objeto inteligente quizás demande más de la cuenta. Es por eso que, al ser un punto tan complejo, se decidió dejarlo de lado para el desarrollo de esta arquitectura.

IV. La arquitectura presentada ha sido pensada y diseñada para su implementación en sistemas embebidos que operen directamente sobre su hardware (es decir, sin un sistema operativo). La evolución de la tecnología ha permitido la construcción de sistemas embebidos con sistema operativo de base, de forma tal que el software para el objeto inteligente se ubique sobre una capa de abstracción. De aquí, se desprende que es necesario continuar con la experimentación de la arquitectura propuesta en sistemas operativos embebidos.

V. Como desprendimiento del punto anterior, surge la necesidad de diseñar y experimentar la adaptación del software de objetos inteligentes ya existentes que deseen migrar o adaptar sus componentes a la arquitectura presentada.

7. BIBLIOGRAFÍA

ARTIK IoT, (2015). "*ARTIK IoT"*. https://www.artik.io/. (Link existente al 24/05/2015)

Atzori, L., Iera, A., Morabito, G., (2010). "*The Internet of Things: A survey*."

Azcurra, D. (2012). "*Formulario de presentación del proyecto: Plataforma Multipropósito de Telemetría y Telecomando a través de Internet Basada en Sistemas Embebidos*", convocatoria I+D+i Amílcar Herrera 2012 de la Secretaria de Ciencia y Técnica de la Universidad Nacional de Lanús (Argentina)

Buschmann, F; Meunier, R; Rohnert, H; Sommerlad, P; Stal, M. 1996. "*Pattern-Oriented Software Architecture: A System of Patterns"*. John Wiley & Sons.

Carretero, J., Daniel García, J. (2013). "*The Internet of Things: Connecting the world."*

Chui, M., Loffler, M., Roberts, R. (2010). "*The Internet of Things*." http://www.mckinsey.com/insights/high_tech_telecoms_internet/the_internet_of_things (Link existente al 24/05/2015)

Evans, D. (2012). "*The Internet of Everything How More Relevant and Valuable Connections Will Change the World"*.

Ferre, X., Juristo, N., Moreno, A., Sanchez, I. (2003). "*A Software Architectural View of Usability Patterns. 2nd Workshop on Software and Usability Cross-Pollination"*.

Fielding. R., (2000) "*Architectural styles and the design of network-based software architectures."* PhD Thesis

Fowler, M. (2004). "*UML distilled: A brief guide to the standard object modeling language (3rd Ed.)"*. Boston: Addison-Wesley.

Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). "*Internet of Things (IoT): A vision, architectural elements, and future directions*."

Guinard, D. (2011). "*A Web of Things Application Architecture – Integrating the Real-World into the web.*" PhD Thesis, ETH Zurich, 2011. webofthings.org/dom/thesis.pdf (Link existente al 15/11/2015)

Groba, Ch., Clarke, S. (2010). "*Web Services on embedded systems – A Performance Study*" webofthings.org/wot/2010/pdfs/152.pdf (Link existente al 15/11/2015)

Holler, J. (2014). "*From machine-to-machine to the internet of things introduction to a new age of intelligence*."

Intel Gateways. (2015). Intel Gateways https://software.intel.com/en-us/iot/hardware/gateways. (Link existente al 24/11/2015)

Internet of Things Architecture. (2013). http://www.iot-a.eu/public/public-documents/copy_of_d1.2/at_download/file (Link existente al 15/10/2015)

IOT-A. (2013). http://www.iot-a.eu/public/public-documents/d1.5/at_download/file (Link existente al 15/10/2015)

Introducing JSON. http://www.json.org/. (Link existente al 16/11/2015)

Kendall, K., Kendall, J. (2005). "*Análisis y Diseño de sistemas, Sexta edición*". ISBN 9780136089162. Ed Prentice Hall

Kortuem, G., Kawsar, F., Fitton, D., & Sundramoorthy, V. (2010). "*Smart objects as building blocks for the Internet of things*." IEEE Computer Society.

Laplante, P., Zhang, J., & Voas, J. (2008). "*What's in a Name? Distinguishing between SaaS and SOA"*.

Merlino, H. (2014). *"Inclusión de Servicios en Aplicaciones Basados en Patrones de Usabilidad. Caso UNDO/REDO."* Tesis Doctoral en Ciencias informáticas. Facultad de Informática. Universidad Nacional de La Plata. *http://sedici.unlp.edu.ar/handle/10915/44290* (Link existente al 28/11/2015)

Microsoft Patterns & Practices Team. (2009). ".*NET application architecture guide (2nd ed.)"*. ISBN 978-0735627109. Microsoft.

Misra, P., Simmhan, Y., Warrior, J. (2015). "*Towards a Practical Architecture for the Next Generation Internet of Things."*

Morales, I. (2000). "*Introducción a la arquitectura: Conceptos fundamentales"* (1.st ed.). Barcelona: UPC.

Murugesan, S., Deshpande, Y., Hansen, S., & Ginige, A. (2000). "*Web Engineering: A New Discipline for Development of Web-Based Systems."* Lecture Notes in Computer Science Web Engineering, 3-13.

Pressman, R. (2002) "*Ingeniería del Software, un enfoque práctico*" ISBN 9786071503145. Ed. McGraw-Hill.

RAML 1.0., (2015). "*RAML"* http://docs.raml.org/specs/1.0/ (Link existente al 24/11/2015)

RAML (2015). "*RAML"* http://raml.org/ (Link existente al 24/11/2015)

Smart Citizen Dispositivos. https://smartcitizen.me/pages/store (Link existente al 24/11/2015)

Sommerville, I. (2011). "*Ingeniería de software* (*9a. ed*.)". ISBN 9786073206037. Ed. Pearson.

Tanenbaum, Andrew S. (2003) "*Redes de computadoras, Cuarta Edición*". ISBN 9789702601623. Ed. Prentice Hall.

W3C Data. http://www.w3.org/2013/data/ (Link existente al 15/11/2015)

W3Schools. http://www.w3schools.com/xml/xml_services.asp (Link existente al 15/11/2015)

Webduino. (2012) https://code.google.com/p/webduino/ (Link existente al 15/11/2015)

8. ANEXO

En esta sección se adjunta la documentación anexa a los capítulos de este T.F.L.

8.1 Resultados de la codificación

8.1.1 Clase Plataforma

```
#ifndef PLATAFORMA_H
#define PLATAFORMA_H
#include <Arduino.h>
#include <Ethernet.h>
#include <SPI.h>
#include "AdministradorDispositivos.h"
#include "AdministradorBackUp.h"
#include "AdministradorConfiguracion.h"
#include "AdministradorDatos.h"
#include "AdministradorEventos.h"
#include "AdministradorNotificaciones.h"
#include "AdministradorUsuarios.h"
#include "WebServer.h"
#include "lib/DataLogger.h"
#include "Util.h"
#define RESET_PIN A0
#include "Definicion.h"
#include "Time.h"

class Plataforma
{
public:
    static char* rootDirectory;
    Plataforma();
    virtual ~Plataforma();
    void processConnection(char *buff, int *bufflen);
    void resolverPost();
    void resolverGet(char* url_tail);
    void refreshButtonState();

    int getButtonState();
    // set command that's run when you access the root of the server
    void setDefaultCommand(WebServer::Command *cmd);

    // set command run for undefined pages
    void setFailureCommand(WebServer::Command *cmd);

    // add a new command to be run at the URL specified by verb
    void addCommand(const char *verb, WebServer::Command *cmd);

    void printFileToServer(char* filename);
    bool autenticarUsuario(char*  nombre, char*  password);

    void reiniciarPorDefecto();
    void reiniciarNormal();

    void procesarREST(WebServer::ConnectionType type, char *url_tail);

    void checkNotificaciones(Usuario* usuario);
    void checkTareasProgramadas();
    void printSerialTime();
    void configurarTime(int hr,int minute,int sec,int day, int month, int yr);
protected:
private:
```

```
    char* getSubPath(char* url);
    char* getResource(char* url);
    void procesarPeticionDispositivos(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionReiniciar(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionConfiguracion(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionUsuarios(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionPerfiles(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionNotifications(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionEvents(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionTasks(WebServer::ConnectionType type, char *url_tail);
    void procesarPeticionPorts(WebServer::ConnectionType type, char *url_tail);
    void procesarPerfilUsuario(int idUsuario, WebServer::ConnectionType type, char
*url_tail);


    void procesarValorDispositivo(int idDispositivo, char *url_tail,
WebServer::ConnectionType type);
    void procesarDireccionDispositivo(int idDispositivo, char *url_tail,
WebServer::ConnectionType type);
    void procesarActualizarDispositivo(int idDispositivo);
    void procesarEliminarDispositivo(int idDispositivo);
    void enviarTodosLosDispositivos();
    void procesarCrearDispositivos();
    void procesarEliminarTodosLosDispositivos();
    void procesarAsignarValorADispositivo(Dispositivo* dispositivo);
    void enviarDispositivo(int idDispositivo);
    void enviarDispositivo(Dispositivo* dispositivo);

    void printDeviceToServer(Dispositivo* dispositivo);
    void printDeviceToServerJSON(Dispositivo* dispositivo);
    void printAddressToServer(Direccion* direccion);
    void printUsuarioToServer(Usuario* usuario);
    void printPerfilToServer(PerfilDeUsuario* perfil);
    void enviarPuerto(PuertoIO* puerto);

    void printConfigToServer();
    void procesarActualizarConfiguracion();
    void printValorToServer( Dispositivo* dispositivo);
    void printNotificacionToServer(Notificacion* notificacion);

    void enviarTodosLosUsuarios();
    void procesarCrearUsuario();
    void procesarEliminarTodosLosUsuarios();
    void enviarUsuario(int idUsuario);
    void enviarUsuario(Usuario* usuario);
    void procesarActualizarUsuario(int idUsuario);
    void procesarEliminarUsuario(int idUsuario);

    void enviarTodosLosPerfiles();
    void procesarCrearPerfil();
    void procesarEliminarTodosLosPerfiles();
    void enviarPerfil(int idPerfil);
    void enviarPerfil(PerfilDeUsuario* perfil);
    void procesarActualizarPerfil(int idPerfil);
    void procesarEliminarPerfil(int idPerfil);

    void enviarTodasLasNotificaciones();
    void procesarEliminarTodasLasNotificaciones();
    void enviarNotificacion(int idNotificacion);
    void enviarNotificacion(Notificacion* notificacion);
    void procesarActualizarNotificacion(int idNotificacion);
    void procesarEliminarNotificacion(int idNotificacion);

    void rechazarPeticion();
    int buttonState;         // variable for reading the pushbutton status
    WebServer* server;
    AdministradorBackUp* adminBackUp;
    AdministradorConfiguracion* adminConfiguracion;
    AdministradorDispositivos* adminDispositivos;
    AdministradorDatos* adminDatos;
    AdministradorEventos* adminEventos;
```

```
    AdministradorNotificaciones* adminNotificaciones;
    AdministradorUsuarios* adminUsuarios;
    Usuario* usuarioActual;


#endif // PLATAFORMA_H
```

8.1.2 Dispositivo

```
/*
 * File:   Dispositivo.h
 * Author: Ariel
 *
 * Created on 3 de noviembre de 2013, 12:07
 */

#ifndef DISPOSITIVO_H
#define  DISPOSITIVO_H

#ifndef DIRECCION_H
#include "Direccion.h"
#endif

enum TipoDispositivo { ANALOGICO, DIGITAL };
enum TipoEntradaSalida { ENTRADA, SALIDA , NOAPLICA};
class Dispositivo {
public:

    Dispositivo(int id,char* nombre, char* descripcion, TipoDispositivo tipoDispositivo,
                Direccion* direccion);
    Dispositivo();
    Dispositivo(const Dispositivo& orig);
    virtual ~Dispositivo();

    // getters
    int getId();
    Direccion* getDireccion();
    char* getDescripcion();
    TipoDispositivo getTipoDispositivo();
    char* getNombre();

    // setters
    void setDireccion(Direccion* direccion);
    void setDescripcion(char* descripcion);
    void setTipoDispositivo(TipoDispositivo tipoDispositivo);
    void setNombre(char* nombre);
    void setId(int id);

    String toString();

    // Funciones Virtuales
    virtual float getValor()=0;
    virtual void setValor(float valor)=0;
    virtual void setValor(char* valor)=0;
    virtual void setEsDeEntrada(bool valor)=0;
    virtual TipoEntradaSalida getEsDeEntrada()=0;

    static char* tipoDispositivoToString(TipoDispositivo tipo);
    static char* tipoEntradaSalidaToString(TipoEntradaSalida tipo);
protected:
    Direccion* direccion;
    TipoDispositivo tipoDispositivo;
    TipoEntradaSalida esDeEntrada;
private:
    int id;
    char nombre[NOMBRE_SIZE];
    char descripcion[DESCRIPCION_SIZE];

};
```

```
#endif   /* DISPOSITIVO_H */
```

8.1.3 Pin

```
#ifndef PIN_H
#define PIN_H

#include "Dispositivo.h"
#include <Arduino.h>

class Pin : public Dispositivo
{
    public:
        /** Default constructor */
        Pin(int id,char* nombre, char* descripcion, TipoDispositivo tipoDispositivo,
Direccion* direccion, TipoEntradaSalida tipoEntradaSalida);
        /** Default destructor */
        virtual ~Pin();
        float getValor();
        void setValor(float valor);
        void setEsDeEntrada(bool valor);
        TipoEntradaSalida getEsDeEntrada();
        void setValor(char* valor);
    protected:

    private:

};

#endif // PIN_H
```

8.1.4 DHT11

```
#ifndef DHT11_H
#define DHT11_H

#include "Dispositivo.h"

#ifndef DIRECCION_H
#include "Direccion.h"
#endif
#define DHTTYPE DHT11   // DHT 11
#include "DHT.h"
#include <stdlib.h>
class DHT11Class : public Dispositivo
{
    public:
        /** Default constructor */
        DHT11Class(int id,char* nombre, char* descripcion, TipoDispositivo
tipoDispositivo, Direccion* direccion);
        /** Default destructor */
        virtual ~DHT11Class();
        virtual float getValor()=0;
        void setValor(float valor);
        void setEsDeEntrada(bool valor);
        TipoEntradaSalida getEsDeEntrada();
    protected:
        dht DHT;
    private:

};

#endif // DHT11_H
```

8.1.5 DHT11 - Temperatura

```
#ifndef DTH11TEMP_H
#define DTH11TEMP_H

#include "DHT11Class.h"


class DHT11Temp : public DHT11Class
{
    public:
        DHT11Temp(int id,char* nombre, char* descripcion, TipoDispositivo
tipoDispositivo, Direccion* direccion);
        virtual ~DHT11Temp();
        float getValor();
        void setValor(char* valor);
    protected:
    private:
};

#endif // DTH11TEMP_H
```

8.1.6 DHT11 – Humedad

```
#ifndef DHT11HUM_H
#define DHT11HUM_H

#include "DHT11Class.h"


class DHT11Hum : public DHT11Class
{
    public:
        DHT11Hum(int id,char* nombre, char* descripcion, TipoDispositivo tipoDispositivo,
Direccion* direccion);
        virtual ~DHT11Hum();
        float getValor();
        void setValor(char* valor);
    protected:
    private:
};

#endif // DHT11HUM_H
```

8.1.7 Administrador de Configuración

```
#ifndef CONFIGURACIONGENERAL_H
#define CONFIGURACIONGENERAL_H
#include <Arduino.h>  // for type definitions
#include "Time.h"

class ConfiguracionGeneral
{
    public:
        ConfiguracionGeneral();
        virtual ~ConfiguracionGeneral();
        void guardarCambios();
        byte GetconfigSet() { return configSet; }
        void SetconfigSet(byte val) { configSet = val; }
        byte Getuse_dhcp() { return use_dhcp; }
        void Setuse_dhcp(byte val) { use_dhcp = val; }
        byte Getdhcp_refresh_min() { return dhcp_refresh_min; }
        void Setdhcp_refresh_min(byte val) { dhcp_refresh_min = val; }
        void setupNetwork();
        void print_EEPROM_Settings();
        byte* Getmac() { return mac; }
```

```
        void Setmac(byte val[6]) {
            mac[0] = val[0];
            mac[1] = val[1];
            mac[2] = val[2];
            mac[3] = val[3];
            mac[4] = val[4];
            mac[5] = val[5];
        }
        byte* Getip() { return ip; }
        void Setip(byte val[4]) {
             ip[0] = val[0];
             ip[1] = val[1];
             ip[2] = val[2];
             ip[3] = val[3];
        }
        byte* Getgateway() { return gateway; }
        void Setgateway(byte val[4]) {
             gateway[0] = val[0];
             gateway[1] = val[1];
             gateway[2] = val[2];
             gateway[3] = val[3];
        }
        byte* Getsubnet() { return subnet; }
        void Setsubnet(byte val[4]) {
             subnet[0] = val[0];
             subnet[1] = val[1];
             subnet[2] = val[2];
             subnet[3] = val[3];
        }
        byte* GetdnsServer() { return dnsServer; }
        void SetdnsServer(byte val[4]) {
             dnsServer[0] = val[0];
             dnsServer[1] = val[1];
             dnsServer[2] = val[2];
             dnsServer[3] = val[3];
        }
        unsigned int GetwebServerPort() { return webServerPort; }
        void SetwebServerPort(unsigned int val) { webServerPort = val; }
        void set_EEPROM_Default();
        void read_EEPROM_Settings();
        void setTime(int hr,int min,int sec,int day, int month, int yr);

    protected:
    private:
        byte configSet;
        byte use_dhcp;
        byte dhcp_refresh_min;
        byte mac[6];
        byte ip[4];
        byte gateway[4];
        byte subnet[4];
        byte dnsServer[4];
        unsigned int webServerPort;
        TimeElements* dateTime;
};

#endif // CONFIGURACIONGENERAL_H
```

8.1.8 Condición Dispositivo - Literal

```
#ifndef CONDICIONDL_H
#define CONDICIONDL_H

#include "Condicion.h"
#include "Dispositivo.h"

class CondicionDL : public Condicion
{
```

```
    public:
        CondicionDL(Dispositivo* operando1, Operador operador, float operando2);
        void setOperando1(Dispositivo* operando1);
        void setOperando2(float operando2);
        Dispositivo* getOperando1();
        float getOperando2();
        virtual ~CondicionDL();
        bool getValor();

    protected:
    private:
        bool evaluarExpresion();
        Dispositivo* operando1;
        float operando2;
};

#endif // CONDICIONDL_H
```

8.1.9 Condición Dispositivo – Dispositivo

```
#ifndef CONDICIONDD_H
#define CONDICIONDD_H

#include "Condicion.h"
#include "Dispositivo.h"

class CondicionDD : public Condicion
{
    public:
        CondicionDD(Dispositivo* operando1, Operador operador, Dispositivo* operando2);
        void setOperando1(Dispositivo* operando1);
        void setOperando2(Dispositivo* operando2);
        Dispositivo* getOperando1();
        Dispositivo* getOperando2();
        virtual ~CondicionDD();
        bool getValor();

    protected:
    private:
        bool evaluarExpresion();
        Dispositivo* operando1;
        Dispositivo* operando2;
};

#endif // CONDICIONDD_H
```

8.1.10 Administrador de Dispositivos

```
#ifndef ADMINISTRADORDISPOSITIVOS_H
#define ADMINISTRADORDISPOSITIVOS_H

#include "Lista.h"
#include "Util.h"
#include "PuertoIO.h"
#include "DispositivoFactory.h"

class AdministradorDispositivos
{
public:
    AdministradorDispositivos();
    Lista<Dispositivo*>* getLista();
    static Lista<PuertoIO*>* getPuertosIO();
    static PuertoIO* getPuertoIO(int id);
    Dispositivo* getDispositivoById(int id);
    static DispositivoFactory* getDispositivoFactory();
    void agregarDispositivo(Dispositivo* dispositivo);
    void eliminarDispositivo(Dispositivo* dispositivo);
```

```
    void eliminarDispositivo(int id);
    void actualizarDispositivo(int id, char* nombre, char* descripcion, char* valor,
TipoDispositivo tipoDispositivo, TipoEntradaSalida tipoEntradaSalida);
    Dispositivo* crearDispositivo(char* nombre, char* descripcion, TipoDispositivo
tipoDispositivo, TipoDireccion tipoDireccion,char* valor, TipoEntradaSalida
tipoEntradaSalida);
    virtual ~AdministradorDispositivos();
protected:
private:
    static int contadorPuertos;
    Lista<Dispositivo*> *lista;
    static Lista<PuertoIO*> *listaPuertosIO;
    static DispositivoFactory* dispositivoFactory;

};

#endif // ADMINISTRADORDISPOSITIVOS_H
```

8.1.11 Capa de Servicios (librería webduino)

```
/* -*- Mode: C++; tab-width: 2; indent-tabs-mode: nil;  c-file-style: "k&r"; c-basic-
offset: 2; -*-

   Webduino, a simple Arduino web server
   Copyright 2009-2012 Ben Combee, Ran Talbott, Christopher Lee, Martin Lormes

   Permission is hereby granted, free of charge, to any person obtaining a copy
   of this software and associated documentation files (the "Software"), to deal
   in the Software without restriction, including without limitation the rights
   to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
   copies of the Software, and to permit persons to whom the Software is
   furnished to do so, subject to the following conditions:

   The above copyright notice and this permission notice shall be included in
   all copies or substantial portions of the Software.

   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
   IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
   FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
   AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
   LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
   OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
   THE SOFTWARE.
*/

#ifndef WEBDUINO_H_
#define WEBDUINO_H_

#include <string.h>
#include <stdlib.h>

#include <EthernetClient.h>
#include <EthernetServer.h>

/************************************************************************
 * CONFIGURATION
 ************************************************************************/

#define WEBDUINO_VERSION 1007
#define WEBDUINO_VERSION_STRING "1.7"

#if WEBDUINO_SUPRESS_SERVER_HEADER
#define WEBDUINO_SERVER_HEADER ""
#else
#define WEBDUINO_SERVER_HEADER "Server: Webduino/" WEBDUINO_VERSION_STRING CRLF
#endif

// standard END-OF-LINE marker in HTTP
```

```
#define CRLF "\r\n"

// If processConnection is called without a buffer, it allocates one
// of 32 bytes
#define WEBDUINO_DEFAULT_REQUEST_LENGTH 32

// How long to wait before considering a connection as dead when
// reading the HTTP request.  Used to avoid DOS attacks.
#ifndef WEBDUINO_READ_TIMEOUT_IN_MS
#define WEBDUINO_READ_TIMEOUT_IN_MS 1000
#endif

#ifndef WEBDUINO_FAIL_MESSAGE
#define WEBDUINO_FAIL_MESSAGE "<h1>EPIC FAIL</h1>"
#endif

#ifndef WEBDUINO_AUTH_REALM
#define WEBDUINO_AUTH_REALM "Webduino"
#endif // #ifndef WEBDUINO_AUTH_REALM

#ifndef WEBDUINO_AUTH_MESSAGE
#define WEBDUINO_AUTH_MESSAGE "<h1>401 Unauthorized</h1>"
#endif // #ifndef WEBDUINO_AUTH_MESSAGE

#ifndef WEBDUINO_SERVER_ERROR_MESSAGE
#define WEBDUINO_SERVER_ERROR_MESSAGE "<h1>500 Internal Server Error</h1>"
#endif // WEBDUINO_SERVER_ERROR_MESSAGE

// add '#define WEBDUINO_FAVICON_DATA ""' to your application
// before including WebServer.h to send a null file as the favicon.ico file
// otherwise this defaults to a 16x16 px black diode on blue ground
// (or include your own icon if you like)
#ifndef WEBDUINO_FAVICON_DATA
#define WEBDUINO_FAVICON_DATA { 0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x10, \
                                0x10, 0x02, 0x00, 0x01, 0x00, 0x01, 0x00, \
                                0xb0, 0x00, 0x00, 0x00, 0x16, 0x00, 0x00, \
                                0x00, 0x28, 0x00, 0x00, 0x00, 0x10, 0x00, \
                                0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x01, \
                                0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0xff, 0x00, 0x00, 0x00, \
                                0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x00, \
                                0x00, 0xff, 0xff, 0x00, 0x00, 0xcf, 0xbf, \
                                0x00, 0x00, 0xc7, 0xbf, 0x00, 0x00, 0xc3, \
                                0xbf, 0x00, 0x00, 0xc1, 0xbf, 0x00, 0x00, \
                                0xc0, 0xbf, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0xc0, 0xbf, 0x00, 0x00, 0xc1, 0xbf, \
                                0x00, 0x00, 0xc3, 0xbf, 0x00, 0x00, 0xc7, \
                                0xbf, 0x00, 0x00, 0xcf, 0xbf, 0x00, 0x00, \
                                0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                0x00, 0x00 }
#endif // #ifndef WEBDUINO_FAVICON_DATA

// add "#define WEBDUINO_SERIAL_DEBUGGING 1" to your application
// before including WebServer.h to have incoming requests logged to
// the serial port.
#ifndef WEBDUINO_SERIAL_DEBUGGING
#define WEBDUINO_SERIAL_DEBUGGING 0
#endif
#if WEBDUINO_SERIAL_DEBUGGING
```

```
#include <HardwareSerial.h>
#endif

// declared in wiring.h
extern "C" unsigned long millis(void);

// declare a static string
#define P(name)   static const prog_uchar name[] PROGMEM

// returns the number of elements in the array
#define SIZE(array) (sizeof(array) / sizeof(*array))

/************************************************************************
 * DECLARATIONS
 ************************************************************************/

/* Return codes from nextURLparam.  NOTE: URLPARAM_EOS is returned
 * when you call nextURLparam AFTER the last parameter is read.  The
 * last actual parameter gets an "OK" return code. */

typedef enum URLPARAM_RESULT { URLPARAM_OK,
                               URLPARAM_NAME_OFLO,
                               URLPARAM_VALUE_OFLO,
                               URLPARAM_BOTH_OFLO,
                               URLPARAM_EOS         // No params left
                             };

class WebServer: public Print
{
public:
    // passed to a command to indicate what kind of request was received
    enum ConnectionType { INVALID, GET, HEAD, POST, PUT, DELETE, PATCH };

    // any commands registered with the web server have to follow
    // this prototype.
    // url_tail contains the part of the URL that wasn't matched against
    //          the registered command table.
    // tail_complete is true if the complete URL fit in url_tail,  false if
    //          part of it was lost because the buffer was too small.
    typedef void Command(WebServer &server, ConnectionType type,
                         char *url_tail, bool tail_complete);

    // constructor for webserver object
    WebServer(const char *urlPrefix = "", int port = 80);

    // start listening for connections
    void begin();

    // check for an incoming connection, and if it exists, process it
    // by reading its request and calling the appropriate command
    // handler.  This version is for compatibility with apps written for
    // version 1.1,  and allocates the URL "tail" buffer internally.
    void processConnection();

    // check for an incoming connection, and if it exists, process it
    // by reading its request and calling the appropriate command
    // handler.  This version saves the "tail" of the URL in buff.
    void processConnection(char *buff, int *bufflen);

    // set command that's run when you access the root of the server
    void setDefaultCommand(Command *cmd);

    // set command run for undefined pages
    void setFailureCommand(Command *cmd);

    // add a new command to be run at the URL specified by verb
    void addCommand(const char *verb, Command *cmd);

    // utility function to output CRLF pair
    void printCRLF();
```

```
    // output a string stored in program memory, usually one defined
    // with the P macro
    void printP(const prog_uchar *str);

    // inline overload for printP to handle signed char strings
    void printP(const prog_char *str)
    {
        printP((prog_uchar*)str);
    }

    // output raw data stored in program memory
    void writeP(const prog_uchar *data, size_t length);

    // output HTML for a radio button
    void radioButton(const char *name, const char *val,
                     const char *label, bool selected);

    // output HTML for a checkbox
    void checkBox(const char *name, const char *val,
                  const char *label, bool selected);

    // returns next character or -1 if we're at end-of-stream
    int read();

    // put a character that's been read back into the input pool
    void push(int ch);

    // returns true if the string is next in the stream.  Doesn't
    // consume any character if false, so can be used to try out
    // different expected values.
    bool expect(const char *expectedStr);

    // returns true if a number, with possible whitespace in front, was
    // read from the server stream.  number will be set with the new
    // value or 0 if nothing was read.
    bool readInt(int &number);

    // reads a header value, stripped of possible whitespace in front,
    // from the server stream
    void readHeader(char *value, int valueLen);

    // Read the next keyword parameter from the socket.  Assumes that other
    // code has already skipped over the headers,  and the next thing to
    // be read will be the start of a keyword.
    //
    // returns true if we're not at end-of-stream
    bool readPOSTparam(char *name, int nameLen, char *value, int valueLen);

    // Read the next keyword parameter from the buffer filled by getRequest.
    //
    // returns 0 if everything weent okay,  non-zero if not
    // (see the typedef for codes)
    URLPARAM_RESULT nextURLparam(char **tail, char *name, int nameLen,
                                 char *value, int valueLen);

    // compare string against credentials in current request
    //
    // authCredentials must be Base64 encoded outside of Webduino
    // (I wanted to be easy on the resources)
    //
    // returns true if strings match, false otherwise
    bool checkCredentials(const char authCredentials[45]);

//  201 Created
//  The request has been fulfilled and resulted in a new resource being created.
    void httpCreated();

//  202 Accepted
//The request has been accepted for processing, but the processing has not been
completed.
```

```
//The request might or might not eventually be acted upon, as it might be disallowed when
processing actually takes place.
    void httpAcepted();

//204 No Content
//The server successfully processed the request, but is not returning any content.
//Usually used as a response to a successful delete request.
    void httpNoContent();

    // output headers and a message indicating a server error
    void httpFail();

//  404 Not Found
//The requested resource could not be found but may be available again in the future.
// Subsequent requests by the client are permissible.
    void httpNotFound();

    // output headers and a message indicating "401 Unauthorized"
    void httpUnauthorized();

    // output headers and a message indicating "500 Internal Server Error"
    void httpServerError();

//  501 Not Implemented
//The server either does not recognize the request method,
//or it lacks the ability to fulfil the request. Usually this implies future availability
    void httpNotImplemented();

    void httpMethodNotAllowed();

//504 Gateway Timeout
//The server was acting as a gateway or proxy and did not receive a timely response from
the upstream server.
    void httpGatewayTimeout();

//507 Insufficient Storage (WebDAV; RFC 4918)
//The server is unable to store the representation needed to complete the request
    void httpInsufficientStorage();

    // output standard headers indicating "200 Success".  You can change the
    // type of the data you're outputting or also add extra headers like
    // "Refresh: 1".  Extra headers should each be terminated with CRLF.
    void httpSuccess(const char *contentType = "text/html; charset=utf-8",
                     const char *extraHeaders = NULL);

    // used with POST to output a redirect to another URL.  This is
    // preferable to outputting HTML from a post because you can then
    // refresh the page without getting a "resubmit form" dialog.
    void httpSeeOther(const char *otherURL);

//  304 Not Modified
//Indicates that the resource has not been modified since the version specified by the
request
//headers If-Modified-Since or If-None-Match.
//This means that there is no need to retransmit the resource, since the client still has
a previously-downloaded copy.
    void httpNotModified();

    // implementation of write used to implement Print interface
    virtual size_t write(uint8_t);
    virtual size_t write(const char *str);
    virtual size_t write(const uint8_t *buffer, size_t size);
    size_t write(const char *data, size_t length);

private:
    EthernetServer m_server;
    EthernetClient m_client;
    const char *m_urlPrefix;

    unsigned char m_pushback[32];
    char m_pushbackDepth;
```

```
    int m_contentLength;
    char m_authCredentials[51];
    bool m_readingContent;

    Command *m_failureCmd;
    Command *m_defaultCmd;
    struct CommandMap
    {
        const char *verb;
        Command *cmd;
    } m_commands[8];
    char m_cmdCount;

    void reset();
    void getRequest(WebServer::ConnectionType &type, char *request, int *length);
    bool dispatchCommand(ConnectionType requestType, char *verb,
                         bool tail_complete);
    void processHeaders();
    void outputCheckboxOrRadio(const char *element, const char *name,
                               const char *val, const char *label,
                               bool selected);

    static void defaultFailCmd(WebServer &server, ConnectionType type,
                               char *url_tail, bool tail_complete);
    void noRobots(ConnectionType type);
    void favicon(ConnectionType type);
};


#endif // WEBDUINO_H_
```