UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: **rParking (Sistema de plazas de estacionamiento reservadas)**
Autores: Boccalari Ezequiel, Gonzalez Francisco
Directores: Lic. Diaz, Javier – Lic. Fava, Laura
Asesor profesional: Lic. Marra, Juan
Carrera: Licenciatura en Sistemas – Licenciatura en Informática

Resumen

La presente tesina consiste en el estudio de la problemática de las plazas de estacionamiento reservadas en un contexto de SmartCities (ciudades inteligentes), el análisis de una potencial solución abarcando las tecnologías correspondientes, tanto de software como de hardware, y el desarrollo y puesta en marcha de un sistema integral que provea la solución previamente diagramada.

Palabras Claves

Smart cities (ciudades inteligentes), IOT (Internet de las cosas), SmartParking (estacionamiento inteligente), javascript, node.js, angular.js, android, mongoDB, tiempo real, sensores de estacionamiento, RFID, plazas de estacionamiento reservadas

Conclusiones

Como conclusión, podemos decir que la solución propuesta es adecuada para resolver el tipo de problema planteado por esta tesina, ya que los resultados obtenidos se correlacionan con los resultados esperados y el desarrollo de la misma cumplió con los objetivos propuestos.

Trabajos Realizados

El desarrollo de un sistema informático que brinda una solución al problema de la gestión de las plazas de estacionamiento reservadas.

Trabajos Futuros

- *Integrar distintos sensores de ocupación y lectores RFID.*
- *Automatizar autenticación de plazas de estacionamiento.*
- *Implementación de aplicaciones mobile para otras plataformas.*

Fecha de la presentación: **Octubre, 2016**

Agradecimientos

A nuestros familiares por darnos todo el apoyo necesario desde el primer día de estudio en la facultad y ayudarnos a que sea posible formarnos como profesionales.

A nuestros amigos y compañeros de clase que nos acompañaron durante toda esta travesía.

A nuestros amigos y compañeros de trabajo con los cuales compartimos diariamente gran parte de nuestras vidas.

A los directores y al asesor profesional de esta tesina por el aporte que cada uno brindó para que sea posible llevarla a cabo.

A la Facultad de Informática de la UNLP y todos sus integrantes que nos acompañaron durante todo el ciclo académico.

ÍNDICE

CAPÍTULO 1 – Introducción

Con el propósito de brindar una solución a una problemática general nos embarcamos en un iniciativa de investigación, análisis y desarrollo tecnológico relacionado con ciudades inteligentes con el fin de proponer y crear un nuevo servicio que mejore algún aspecto de los habitantes de una ciudad.

En principio, nuestra intención fue abocarnos al concepto de estacionamiento inteligente ya que entendimos que el estacionamiento en los conglomerados urbanos de gran tamaño supone una complicación en la vida cotidiana de los ciudadanos. De esta manera pensamos en poder gestionar las plazas de estacionamiento de una ciudad con el fin de que las personas puedan tener acceso, mediante un sistema informático, al estado de las plazas y así optimizar el uso de las mismas. Una de las mejores maneras de obtener el estado de ocupación de una plaza de estacionamiento en tiempo real es utilizando sensores magnéticos de estacionamiento disponibles en el mercado, los cuales serían instalados en cada una de las plazas e integrados al sistema previamente mencionado.

Dado el gran costo de inversión que requiere llevar a cabo una solución de esa magnitud, por la cantidad de sensores y el costo de instalación de los mismos, decidimos reducir el alcance del proyecto limitándonos a gestionar plazas de estacionamiento reservadas ya que la cantidad de sensores a obtener e instalar es sustancialmente menor.

De todas maneras, el sistema se desarrollará lo suficientemente flexible, genérico y escalable con el fin de permitir ampliar el alcance a lo descripto anteriormente en caso de contar con la cantidad de sensores requeridos para una solución de mayor envergadura.

1.1 Motivación

Las ciudades suelen contar con diversos sectores que poseen plazas de estacionamiento reservadas exclusivamente para un grupo selecto/determinado de vehículos. Dichas plazas no pueden ser ocupadas por vehículos no autorizados, en cuyo caso, los infractores a dicha prohibición podrían ser sancionados. Además, en algunos casos, la disponibilidad de las plazas reservadas resulta crítica. Por ejemplo, una ambulancia trasladando un paciente con urgencia a un hospital debe poder estacionar en las zonas reservadas del hospital sin ningún tipo de inconvenientes. Por otro lado, una persona con discapacidad podría evitar esperas y búsquedas innecesarias, haciendo más fluido y simple el uso de dichas plazas. Actualmente, son pocas las ciudades que poseen algún tipo de sistema inteligente y/o automático que asista en el control de dichas plazas. Por lo general, estas están demarcadas de alguna manera (con señales o con pintura delimitando el espacio de la plaza en el asfalto) y la única manera de controlarlas es a través de un inspector que esté continuamente observando y garantizando que dichos sectores reservados no sean ocupados inadecuadamente. Esto incurre en un cierto margen de error humano, además de que en general no se tiene registro de la actividad y el uso de las plazas.

1.2 Objetivos y metodología a emplear

El objetivo principal de esta tesina es implementar un sistema de control sobre las plazas de estacionamiento reservadas en contextos en donde existan restricciones sobre los vehículos que pueden ocupar dichas plazas. A su vez, esto se enmarca en el concepto de Smart Cities (ciudades inteligentes) y aportaría a la ciudad que lo implemente la posibilidad de que los ciudadanos interesados puedan saber en cualquier momento la disponibilidad de las plazas reservadas y optimizar su uso.

Para llevar a cabo este proyecto, en primera instancia, se hará un estudio de las herramientas de software y hardware apropiadas para el desarrollo del mismo, teniendo en cuenta que como restricción necesitamos una solución en tiempo real. Como una segunda etapa, se definirá la arquitectura del sistema y se implementará los componentes que integran la misma utilizando las herramientas previamente estudiadas. En la última etapa, haremos una puesta en marcha del sistema, en un campo de prueba real con plazas de estacionamiento reservadas, para poder evaluar la respuesta de la solución desarrollada y compararla con los resultados esperados.

1.3 Estructura de la tesina

A continuación detallaremos los capítulos que componen esta tesina explicando el contenido y competencia de cada uno.

- *Capítulo 1 – Introducción*: Introducción general de la tesina, problemática a la que hace referencia y solución propuesta.
- *Capítulo 2 – Smart Cities y Smart Parking*: En este capítulo se da una perspectiva del concepto de Smart Parking a partir del enfoque en los conceptos de Smart Cities e IOT (Internet de las cosas).
- *Capítulo 3 – Estado del Arte de Smart Parking*: En esta sección desarrollamos el estado del arte con respecto a las plazas de estacionamiento reservadas.
- *Capítulo 4 – Análisis de Tecnologías*: Análisis de las diversas herramientas tecnológicas de software y hardware que a priori pensamos son las apropiadas para llevar a cabo la solución.
- *Capítulo 5 – Arquitectura Propuesta*: Detalle de la implementación del sistema llevado a cabo.
- *Capítulo 6 – Pruebas y Resultados*: En este capítulo damos un resumen

general de los resultados esperados, pruebas realizadas y resultados obtenidos.

- *Capítulo 7 – Conclusiones y Trabajos Futuros:* Por último brindamos las conclusiones generales tanto del proyecto como de la experiencia personal. A su vez, mencionamos los posibles trabajos futuros.

CAPÍTULO 2 – Smart Cities y Smart Parking

2.1 Smart Cities

No existe hoy en día una definición absoluta y estandarizada sobre el concepto de Smart Cities (ciudades inteligentes). En principio algunos de los puntos que son necesarios para considerar a un servicio provisto por una ciudad, como parte de un acercamiento a la idea de Smart Cities, tiene que ver con la recolección, análisis y puesta en disponibilidad de cualquier tipo de información que sea útil a todos los ciudadanos; y que el conocimiento y uso de dicha información, permita de alguna manera, mejorar la calidad de vida y el vínculo del ciudadano con los servicios que la ciudad provee. [1]

Los aspectos claves más importantes para el desarrollo de Smart Cities son:

- La existencia de una infraestructura digital moderna que permita el desarrollo de los servicios inteligentes (Internet, sensores, aplicaciones y redes sociales, aplicaciones móviles, etc.) combinada con una política de acceso abierto al consumo y la generación de la información pública en todo momento.
- Pensar y desarrollar los servicios centrados en el ciudadano y sus necesidad finales.
- Transparencia en la comunicación a los ciudadanos de los resultados y la performance alcanzada por los servicios inteligentes.

En un contexto tecnológico, el concepto de Smart City y el de Internet de las cosas (IoT) están muy relacionados ya que no consisten solamente en la conexión de cada vez mas personas, sino en el planteamiento de un mundo digital en el que todo podrá estar conectado entre sí.

Desde la perspectiva de los responsables municipales de los servicios

prestados en la ciudad, disponer de una ciudad inteligente va a ayudar a la gestión automática y eficiente de las infraestructuras urbanas, lo que aporta ventajas como reducciones de gastos, mejoras en si de los propios servicios prestados o incluso la posibilidad de crear nuevos servicios que respondan mejor a las necesidades específicas de cada ciudad, o de sus ciudadanos, e identificar posibles problemas futuros para reducir el impacto de los mismos.

Los ejes en los que suele incidir un proyecto de ciudad inteligente están relacionados con la movilidad urbana, eficiencia energética, gestión sustentable y sostenible de los recursos, gestión de las infraestructuras de la ciudad, gobierno abierto y participativo, seguridad pública, cultura y educación. Si bien las necesidades anteriores pueden estar agrupadas por áreas, se puede lograr un mayor potencial cuando se integran en una visión conjunta de otros servicios. Por ejemplo, una aplicación que ayude a gestionar el tráfico en tiempo real de una ciudad podría ayudar a identificar las zonas con mayor concentración de contaminación ambiental, información que a su vez puede utilizarse para analizar si las condiciones ambientales en determinadas zonas producen una mayor incidencia de enfermedades respiratorias.

En resumen, una ciudad inteligente provee a sus ciudadanos una serie de servicios basados en tecnologías modernas y digitales, que le permite a los mismos un uso más eficiente y una mayor interacción con los mismos, generando una retro alimentación al sistema global que permita mejorar su calidad a lo largo de su existencia. [2]

2.2 Movilidad Urbana

La movilidad en las ciudades se está convirtiendo en un problema en crecimiento. Es por este motivo que es una de las necesidades mas importantes a abordar bajo el concepto de ciudad inteligente. El concepto de movilidad hace referencia a la sostenibilidad, seguridad y eficiencia de las infraestructuras y sistemas de transporte de una ciudad, como así también a su accesibilidad local, nacional e internacional.

Uno de los mayores problemas en el ámbito de la movilidad urbana es la congestión del tráfico, el cual tiene un impacto negativo en la calidad de vida de los ciudadanos por varios motivos, entre los cuales se encuentra la pérdida de dinero por extensas búsquedas de estacionamiento y el consumo de combustible consecuente, un empeoramiento en la calidad del aire y contaminación ambiental debido al tiempo extra de circulación de los vehículos, pérdida de tiempo y disminución de productividad por la cantidad de horas desaprovechadas por los automovilistas.

Sin duda, la gestión de estacionamiento se trata de un servicio de gran utilidad. Las aplicaciones para este tipo de problema constan de sensores distribuidos por la ciudad que permiten identificar plazas de estacionamiento libres y consecuentemente mejorar la gestión de la ocupación de dichas plazas. De esta manera se minimiza el tiempo de circulación de los vehículos. [3]

2.3 Internet de las cosas (IoT)

El concepto de Internet de las cosas surge a partir de la idea de que cualquier “cosa” u objeto pueda ser capaz de comunicarse con otros objetos y personas a través de Internet, redes privadas u otros protocolos. Este concepto entró en foco de atención mundial en 2005 cuando la Unión Internacional de Telecomunicaciones (ITU por sus siglas en inglés) publicó el primer informe sobre el tema, donde sugería que Internet de las cosas, en un futuro, conectaría objetos repartidos por todo el mundo a través de la combinación de los avances tecnológicos en la identificación de objetos, redes de sensores, sistemas integrados y la nanotecnología. [4]

Según analistas, existen dos modos básicos de comunicación en Internet de las cosas, cosa-persona y cosa-cosa:

- Cosa – Persona: Las comunicaciones de este tipo abarcan una serie de tecnologías y aplicaciones en las cuales las personas interactúan con cosas y viceversa. También existen cosas que informan a las personas de

cambios en su estado, datos recolectados, etc.

- Cosa – Cosa: Abarca tecnologías y aplicaciones en donde los objetos interactúan sin que ningún humano haya iniciado la interacción ni sea receptor o intermediario. Los objetos pueden controlar otros objetos, tomar medidas correctivas y realizar notificaciones a las personas según sea necesario.

El desarrollo técnico de Internet de las cosas depende de una combinación en el avance de varios desarrollos:

1. En primer lugar, desarrollo de tecnologías con el fin de **identificar** objetos y dispositivos. La identificación tiene importancia ya que es necesaria para poder hacer referencia a objetos, personas y/o aplicaciones. Dentro del área de la identificación de objetos que no se encuentran conectados a Internet, la identificación por radiofrecuencia RFID es la tecnología más difundida y aceptada. RFID es un sistema de almacenamiento y recuperación de datos remoto capaz de ser leído y escrito sin contacto físico, vía ondas de radio usando antenas. Los componentes que integran este sistema son:
   - TAG RFID: compuesto por una antena, un transductor radio y un chip. El propósito de la antena es permitirle al chip transmitir la información de identificación obtenida del tag.
   - Lector RFID: capta la señal de un tag RFID, extrae la información y se la suministra al middleware.
   - Middleware RFID: proporciona procesamiento y almacenamiento de datos.
2. En segundo lugar se encuentra la **recopilación de datos**, es decir, desarrollar la capacidad de detectar cambios en el estado físico de las cosas mediante tecnologías como son los sensores. Los sensores recogen los datos de su entorno y los suministran para que puedan ser aplicados en la toma de decisiones.
3. En tercer lugar **dotar de inteligencia a las cosas**, es decir, que los objetos

cotidianos adquieran nuevas características inteligentes, ya sea procesamiento de datos y/o comunicación con otro objeto o persona. Un objeto inteligente podría reaccionar ante eventos o señales y tomar decisiones de manera autónoma o colaborativa con otros objetos o personas.

4. Por último, **lograr avances en la nanotecnología**. Esto significa lograr que cosas cada vez más pequeñas tengan la capacidad de comunicarse con otras cosas y conectarse a distintos tipos de redes. [5]

2.4 Smart Parking (Estacionamiento Inteligente)

En las ciudades con grandes cantidades de tránsito vehicular se observa que uno de los mayores problemas consiste en la búsqueda de estacionamiento. La misma incurre en grandes desperdicios de tiempo y combustible para los conductores, y esto no solo afecta a los que están intentando estacionar, sino a todos, ya que genera congestión de tráfico en las calles donde este fenómeno se produce con mayor frecuencia. [6]

Una posible solución radica, en principio, en conceptualizar al estacionamiento como un servicio provisto por la ciudad. Esto se puede enmarcar dentro del concepto de Smart Cities utilizando tecnología moderna para gestionar la disponibilidad de las plazas de estacionamiento y lograr un uso óptimo de las mismas. Esto es lo que se conoce como estacionamiento inteligente (Smart Parking).

Para que esta solución pueda considerarse realmente dentro del concepto de Smart Cities, la misma debe tener una manera de recolectar la información acerca del estado de disponibilidad de las plazas de estacionamiento y, además, proveer algún medio de acceso público a esta información para que los ciudadanos interesados puedan consultarla.

Existen diversas maneras de recopilar información acerca del estado de las plazas pero, posiblemente, la más adecuada sea detectar el uso de las mismas automáticamente. Esto se puede lograr mediante distintas tecnologías como por ejemplo, sensores de detección de estacionamiento, procesamiento de imágenes, parquímetros, etc.

Por otro lado, la publicación de esta información podría darse a través de aplicaciones web, aplicaciones para teléfonos inteligentes, pantallas digitales en puntos claves, y cualquier otro tipo de dispositivo electrónico o digital por el cual se pueda brindar información al ciudadano.

CAPÍTULO 3 – Estado del Arte de Smart Parking

3.1 Plazas de Estacionamiento Reservadas

Teniendo en cuenta el concepto de estacionamiento inteligente enmarcado en un contexto de Smart Cities, y también la existencia de lugares con plazas de estacionamiento reservadas para ciertos usuarios, es interesante proponer una solución para facilitar la gestión de dichas plazas mediante tecnologías de detección automática; detectando la ocupación de las mismas e identificando a los vehículos habilitados para estacionar. Además se podría optimizar el uso de estos lugares reservados, para que en horarios en donde no aplique la exclusividad del uso por parte de los usuarios habilitados, puedan ser utilizadas por cualquier conductor; y de esta manera aumentar la capacidad de estacionamientos disponibles en la ciudad. [7]

Por otro lado, la disponibilidad de la información respecto de la ocupación de las plazas reservadas permite que los usuarios que estén autorizados para estacionar puedan ver una mejora en la experiencia del uso, ya que pueden evitar búsquedas innecesarias y así ahorrar tiempo y combustible.

Hay que tener en cuenta también, que la implementación de un arquitectura de estas características puede resultar en una inversión muy costosa para cualquier ciudad. Por lo tanto, consideramos que resulta más viable comenzar con una primera aproximación, limitando el alcance de la solución a las plazas de estacionamiento reservadas y el uso autorizado de las mismas, ya que en general siempre será un subconjunto menor que el de plazas de estacionamiento disponibles en una ciudad.

3.2 Casos existentes

3.2.1 WORLDSENSING - Fastprk

En la ciudad de Moscú, Rusia, la empresa Worldsensing desplegó una solución de estacionamiento inteligente impulsada por el gobierno de la ciudad como respuesta al gran problema de congestión de tráfico que padecía. La solución, llamada “Fastprk” consiste en una red de sensores de ocupación de plazas de estacionamiento y de una plataforma de software asociada que fue integrada con los sistemas ya existentes de pago y cumplimiento de estacionamiento provistos por otras empresas.

El sistema también se adaptó a las plazas de estacionamiento para discapacitados, servicios de emergencia, ambulancias y policía, con la capacidad de verificar a los ocupantes de dichas plazas distinguiendo entre vehículos permitidos y no permitidos. También se instalaron pantallas digitales que muestran información en tiempo real acerca del estado de ocupación de las plazas de estacionamiento, las cuales le indican al conductor sobre las plazas que están libres en su proximidad. Esta información también se hizo accesible a través de un portal web y aplicaciones para teléfonos inteligentes.

El despliegue de la solución se llevó a cabo en varias etapas. La primera tuvo lugar en Noviembre de 2012 y fueron instalados cerca de 1000 sensores. Durante el 2013 y 2014 se fue incrementando la cantidad de sensores instalados alcanzando una cantidad de 2000 dispositivos en junio de 2013, 12000 dispositivos en enero de 2014 y finalmente 50000 a mediados del mismo año.

Como resultado de esta implementación, en la ciudad se logró reducir en gran medida la congestión de tráfico en zonas críticas y con ello los niveles de contaminación ambiental y estrés de los conductores. Por otro lado, se consiguió optimizar el pago y el control del estacionamiento ya que el sistema interrelaciona la información de las plazas activas con la información de los sistemas existentes

y permitió detectar las áreas con mayor porcentajes de ocupantes evitaban el pago de estacionamiento o incumplían la ley con estacionamientos no permitidos. [8]

3.2.2 SMARTPARKING - Detección inteligente de ocupación de plazas de estacionamiento reservadas para discapacitados

En la ciudad de Westminster, Inglaterra, se implementó un sistema de estacionamiento medido mediante el cual se detecta automáticamente la ocupación de plazas de estacionamiento delimitadas en la calle a través de sensores instalados en el asfalto. La ciudad cuenta con 3400 plazas equipadas con estos sensores, de las cuales 750 están reservadas para usuarios discapacitados y a su vez, 244 de estás últimas, son reservadas para personas específicas.

Con la motivación de evitar el uso indebido de las 244 plazas reservadas, la empresa Smart Parking, proveedora de dicha tecnología, sugirió una adaptación a estos sensores que permitiría determinar si una de estas plazas fue ocupada por un usuario habilitado, identificándolo mediante el uso de la tecnología RFID.

Actualmente el ayuntamiento de Westminster instaló, como parte de una prueba piloto, 64 sensores modificados (del total de las 244 plazas reservadas) y entregó los tags RFID a los usuarios correspondientes. Cuando una de estas es ocupada por un vehículo que no posee el tag de identificación apropiado, el sistema envía una señal de alerta al centro de operaciones correspondiente, la cual es atendida por un oficial a cargo que se encarga de dirigirse hasta el lugar para comunicarle al usuario inhabilitado que debe dejar de ocupar la plaza reservada.

Esta solución logró que los usuarios discapacitados tuvieran menos inconvenientes a la hora de estacionar en sus plazas de estacionamiento reservadas, ya que redujo la frecuencia con la que la encontraban ocupadas por

otros vehículos. [9]

3.3 Problemáticas Encontradas

Generalmente una gran parte del tráfico en las ciudades está generado por conductores intentando encontrar una plaza de estacionamiento desocupada. A priori se puede pensar que el tráfico es ocasionado por conductores dirigiéndose a otro sitio pero en realidad, los que buscan estacionamiento están mezclados con los anteriores y por lo tanto es difícil determinar cuantos vehículos transitan a baja velocidad con el fin de encontrar una plaza de estacionamiento libre.

Diversos estudios, en distintas ciudades del mundo, han demostrado que el tiempo en que estos vehículos permanecen buscando lugares libres para estacionar puede ser una de las fuentes importantes de la congestión del tránsito. Esto incurre en gastos innecesarios de tiempo y combustible, además de que se ha notado que en muchas ciudades, en donde hay calles que son propensas a generar embotellamientos, un gran porcentaje de vehículo representa a aquellos automovilistas que están buscando un estacionamiento libre. [10]

En la actualidad, en muchas ciudades se utilizan sistemas de estacionamiento inteligente diseñados para abordar el problema de la búsqueda de plazas de estacionamiento libre y así intentar aliviar la congestión generada por dichos vehículos. Una de las soluciones mas utilizadas consiste en desplegar una red de sensores de estacionamiento para obtener solamente información sobre la disponibilidad de las plazas y luego publicar está información a los ciudadanos interesados. En principio puede ser útil una solución de este estilo, pero por otro lado puede traer aparejado un efecto colateral con impacto negativo. Esto se da en el caso en que las plazas vacantes en una zona determinada sean limitadas y la cantidad de usuarios interesados en estacionar en dicha zona las supere en gran medida. Como consecuencia, dicha área puede verse más afectada debido al

arribo de los conductores interesados.

Una solución propuesta a este problema consiste en permitir a los conductores, a partir de una notificación de una plaza libre, reservarla mediante algún sistema dedicado y de esta manera informar al resto de los conductores interesados para evitar que se trasladen al mismo punto y de esta manera reducir la cantidad de vehículos circulando en la misma zona. [11]

Teniendo en cuenta el costo de inversión que involucra una solución para ciudades con gran cantidad de plazas de estacionamiento -como La Plata- y considerando la reducida cantidad de sensores con que se cuenta/contamos, en esta tesina se propone una solución para gerenciar un conjunto reducido de plazas de estacionamiento pertenecientes al Rectorado de nuestra Universidad, ubicado en la calle 47 entre 6 y 7.

Las ciudades suelen contar con diversos sectores que poseen plazas de estacionamiento que están reservadas exclusivamente para un grupo selecto/determinado de vehículos. Dichas plazas no pueden ser ocupadas por vehículos no autorizados, en cuyo caso, los infractores a dicha prohibición podrían ser sancionados. Además, en algunos casos, la disponibilidad de las plazas reservadas resulta crítica. Por ejemplo, una ambulancia trasladando un paciente con urgencia a un hospital debe poder estacionar en las zonas reservadas del hospital sin ningún tipo de inconvenientes. Por otro lado, una persona con discapacidad podría evitar esperas y búsquedas innecesarias, haciendo más fluido y simple el uso de dichas plazas.

Llevar a cabo una solución para este enfoque/escenario implica un costo mucho menor de infraestructura debido a que se necesita una menor cantidad de dispositivos, cuya instalación, puesta a punto y mantenimiento resulta más viable operativa y económicamente. Por otro lado, este tipo de desarrollo podría servir como prototipo para ciudades interesadas en comenzar a integrar sistemas inteligentes.

CAPÍTULO 4 – Análisis de Tecnologías

4.1 Introducción

En este capítulo haremos el análisis pertinente de las tecnologías necesarias para comenzar a idear una solución a los problemas descriptos anteriormente. La misma debería permitir detectar de manera automática, eficiente y en tiempo real, la ocupación de una plaza reservada para poder actuar en consecuencia. Esto significa, que dicha información pueda ser accedida por quien corresponda para que pueda decidir si es una ocupación válida o no. A su vez, la solución debería contemplar automáticamente las ocupaciones válidas de las personas habilitadas para hacer uso de las plazas, de manera de evitar la autorización manual por parte los inspectores a cargo.

Por último, esta solución se debería modelar de manera que resulte flexible y adaptable a problemáticas similares que involucren la detección automática de ocupaciones de plazas de estacionamiento.

4.2 Sensores de ocupación

Redes de sensores inalámbricos (WSN- Wireless Sensor Network)

Una red de sensores inalámbricos (WSN por sus siglas en inglés) es una red que se auto-configura, formada por dispositivos (denominados nodos) que son desplegados en gran cantidad para percibir datos del mundo físico y que se comunican entre si por señales de radio.

Básicamente, cada nodo contiene una unidad de procesamiento con

capacidad de computación restringida, una memoria limitada, un dispositivo de comunicación de radio, una fuente de alimentación, y uno o más sensores conectados.

- Procesador: La tarea de esta unidad es la de procesar la información detectada localmente y la información registrada por otros dispositivos o sensores. Puede operar en diferentes modos: el modo “sleep” o modo dormido, se utiliza la mayor cantidad del tiempo para ahorrar energía; “idle” o inactivo, se usa cuando se espera recibir datos desde otro nodo, y “active” o activo cuando se envían o reciben datos con otros nodos sensores.
- Fuente de alimentación: Los nodos fueron pensados para desplegarse en ambientes variados, que pueden ser desfavorables en cuanto al acceso de energía eléctrica, por ende, deberían optimizar el uso de energía. También deberían proveer mecanismos que permitan al usuario final la opción de prolongar la vida útil de la red a expensas de disminuir el rendimiento.

  Las fuentes de energía mas comunes son baterías recargables, paneles solares y condensadores.
- Memoria: Se utiliza para almacenar programas y datos.
- Radio: Los nodos incluyen un radio inalámbrico de baja velocidad y poca cobertura. La velocidad mas típica es de 10-100 kbps y la cobertura es de 100 metros aproximadamente. La radiocomunicación es la actividad que generalmente consume mayor cantidad de energía, por lo que es importante incluir técnicas de ahorro de energía, como por ejemplo, que el nodo este en estado activo solamente en los momentos que requiera comunicarse.
- Sensores: Las redes de sensores pueden estar formadas por diferentes

tipos de sensores capaces de monitorizar una gran variedad de condiciones ambientales. Las características mas importantes de estos están basadas en (1) estado de desarrollo, describe la eficiencia de la ingeniería del sensor en relación al área donde se instalará, (2) capacidad de expansión, describe, en función a su tamaño y costo, la capacidad de expandirse y utilizarse en muchos sistemas distribuidos, (3) y el parámetro o tipo de medición que se obtiene con el sensor. Algunas aplicaciones precisan detectar varios parámetros al mismo tiempo, de manera que cada nodo puede tener varios sensores incluidos.

Las características técnicas mas importantes de una red de sensores son las siguientes:

1. Conexión inalámbrica: Los nodos de la red de sensores se comunican entre sí por radio para intercambiar y procesar los datos recolectados. La conectividad inalámbrica permite recuperar datos en tiempo real en puntos de difícil acceso, facilita la tarea de monitoreo en aquellos sitios donde los cables podrían dificultar la tarea, y reduce el costo de instalación.
2. Auto-organización: Los nodos se organizan entre si en una red ad-hoc, lo que significa que no necesitan una estructura de red preexistente. Cada nodo está programado para realizar un descubrimiento de su entorno, es decir, para reconocer cuales son sus nodos vecinos con los cuales puede interactuar a través de la radio.
3. Bajo consumo: Como las redes de sensores pueden instalarse en sitios remotos donde no se dispone de fuentes de alimentación, los nodos deben usar radios y procesadores de bajo consumo e implementar planes de eficiencia de uso de energía para poder funcionar por meses y/o años.

Existen varios roles en una red de sensores inalámbrica.

Los nodos pueden ser “nodos sensores”, equipados con diferentes tipos de sensores, se utilizan para monitorizar su entorno y transmitir las lecturas de los parámetros correspondientes a un “nodo recolector” o también denominado “estación base”.

Los nodos recolectores (sink) tienen la responsabilidad de recolectar las lecturas de los nodos sensores y reenviar esos datos a un servidor para realizar posteriores procesamientos y/o análisis de los mismos.

Por último, los “nodos actuadores” son dispositivos que se usan en el control del ambiente y responden ante las lecturas de los sensores u otras entradas. Por ejemplo, pueden ser los encargados de encender una luz cuando hay bajas condiciones de luminosidad, parámetro obtenido por algún nodo sensor. [12]

Estándar IEEE 802.15.4

Una red 802.15.4 es una parte de la familia de estándares de WPAN (Wireless Personal Area Network).

Las redes inalámbricas de área personal (WPANs por sus siglas en inglés) son usadas para transmitir información en distancias relativamente pequeñas. A diferencia de las redes inalámbricas de área local (WLANs), las conexiones realizadas con redes WPANs no requieren una infraestructura de red montada previamente. Esta característica permite implementar soluciones pequeñas, de bajo consumo y económicas para un amplio conjunto de dispositivos.

En 2003 el IEEE (Institute of Electrical and Electronics Engineers) publica el estándar 802.15.4 para redes personales inalámbricas de bajo consumo. En el

año 2006 se publica una actualización del estándar con algunas mejoras incluidas.

Este estándar define especificaciones para la capa física (PHY) y el control de acceso al medio (MAC) en redes inalámbricas con bajas tasas de transmisión de datos (LR-WPAN) donde se conectan dispositivos portables con autonomía limitada, por lo tanto requieren un uso de energía óptimo y eficiente.

Una red inalámbrica con baja transmisión de datos (LR-WPAN) es una red con bajo costo de comunicación que permite una conexión inalámbrica en aplicaciones con fuente de energía limitada y requisitos de rendimiento no muy estrictos. Los principales objetivos de una red LR-WPAN son facilidad de instalación, transferencia de datos fiable, bajo costo, y una duración razonable de batería manteniendo un protocolo simple y flexible.

En una red que implementa el estándar 802.15.4 pueden participar dos tipos de dispositivos: dispositivos con funciones completas (FFD por sus siglas en inglés) y dispositivos con funciones reducidas (RFD). Un dispositivo FFD tiene tres modos de funcionamiento, como un coordinador de una red de área personal (Coordinador PAN), solo como coordinador, o como dispositivo. Un dispositivo FFD puede comunicarse con dispositivos RFD o con otros dispositivos FFD, mientras que los dispositivos RFD solo pueden comunicarse con dispositivos FFD. Los dispositivos RFD están dirigidos a aplicaciones que son extremadamente simples que no tienen la necesidad de transmitir grandes cantidades de datos y quizás solo interactúen con un único dispositivo FFD a la vez. Por este motivo, los dispositivos RFD pueden ser implementados usado una mínima cantidad de recursos.

Una red WPAN debe incluir al menos un dispositivo FFD que opere como coordinador PAN. Dependiendo de los requerimientos de la aplicación, una red 802.15.4 LR-WPAN puede formarse a partir de dos topologías de red como muestra la Figura 1: una topología tipo estrella (star topology) o una topología

entre pares (peer-to-peer topology).

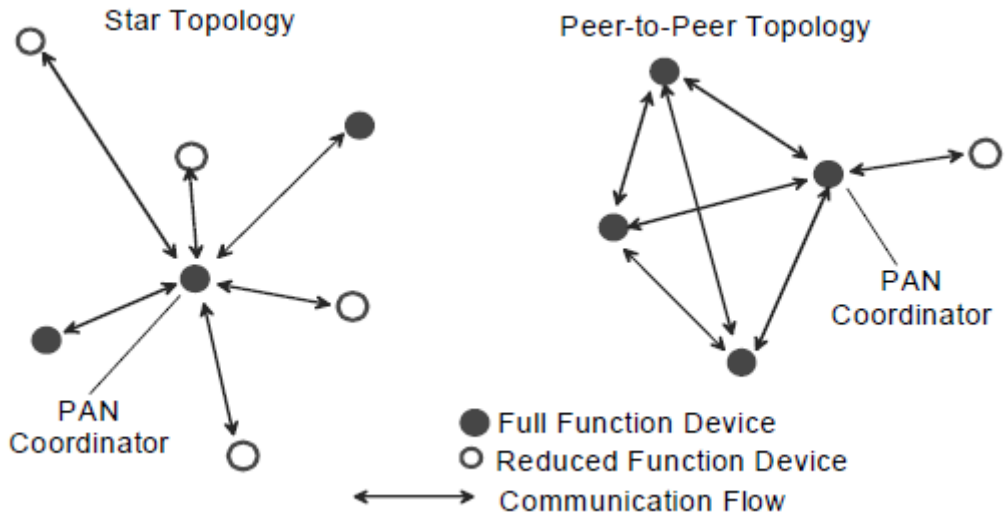

Figura 1. Ejemplos de topologías de una red LR-WPAN

En la topología tipo estrella, la comunicación es establecida entre dispositivos y un coordinador central (Coordinador PAN). Una topología de pares también tiene un coordinador PAN pero difiere de la topología tipo estrella en que cualquier dispositivo puede comunicarse con cualquier otro, en la medida de que estén en su radio de alcance. [13]

Arquitecturas para detección de ocupación de plazas de estacionamiento

Las arquitecturas que evaluamos para poder detectar la ocupación de plazas de manera automática fueron las siguientes:

- Arquitectura basada en el nivel de magnetismo: Esta arquitectura permite detectar la ocupación de una plaza de estacionamiento utilizando hardware dedicado que detecta las variaciones en el campo magnético de

la tierra que genera un vehículo.

- Arquitectura basada en reconocimiento óptico: Esta arquitectura permite detectar la ocupación de una o varias plazas de estacionamiento mediante el uso de cámaras de vídeo y un software específico capaz de reconocer la forma de un vehículo en las imágenes provistas por las cámaras y determinar una ocupación.

A continuación describimos en mayor detalle el funcionamiento de una arquitectura basada en el nivel de magnetismo. Para ello analizamos y evaluamos la solución brindada por la empresa Urbiotica, que provee diversos productos orientados a Ciudades Inteligentes.

En particular, la solución de Urbiotica para la detección de ocupación de plazas de estacionamiento consta de dos partes. Por un lado están los elementos físicos de la instalación (sensores y elementos de comunicación) y por otro la plataforma de software (U-Base). Las redes de sensores captan la información en tiempo real del estado de las plazas de estacionamiento y transmiten los datos por radio a los elementos de comunicación, quienes se encargan de reenviarlos a la plataforma de software U-Base utilizando la red de comunicaciones disponible (GPRS, Wi-Fi o Ethernet). La plataforma U-Base procesa los datos provenientes de la red de sensores y pone esa información a disposición de aplicaciones externas interesadas en consumirla. A continuación se detalla cada uno de los componentes.

U-SPOT (Elemento Sensor)

Es un sensor cuya funcionalidad es comunicar si una plaza de estacionamiento está ocupada o no. Para ello, envía información sobre los

cambios en el campo magnético terrestre debidos a la presencia de un vehículo. En la Figura 2 se puede observar la imagen de un U-SPOT.

Estos sensores pueden comunicarse con los elementos de comunicación U-FLAG en un radio de 50 metros utilizando el protocolo estándar de comunicación IEEE 802.15.4 a 2,4GHz. En cuanto a su autonomía, cuentan con una batería interna que les permite estar activos por hasta 10 años.

Por último, la confiabilidad de las mediciones de estos elementos es del 98% aproximadamente, es decir que la posibilidad de falsos positivos es baja.

Figura 2 – U-Spot

U-FLAG (Elemento de comunicación)

El U-FLAG, mostrado en la Figura 3, es un nodo enrutador que junta los datos enviados por los sensores de ocupación (U-SPOT) que hay a su alrededor y los redirige hacia el U-BOX mas cercano.

Estos componentes establecen una red Mesh, es decir, se interconectan unos con otros, para poder transmitir la información de ocupación obtenida hacia el U-BOX de destino.

Estos componentes, si no cuentan con alimentación eléctrica externa, tienen una autonomía de 4 días. Una alternativa puede ser utilizando otro componente de alimentación solar (U-SUN).

Figura 3 – U-Flag

U-BOX (Elemento de comunicación)

El U-BOX, Figura 4, es una pasarela de comunicaciones que recoge los datos de los sensores (U-SPOT) y enrutadores (U-FLAG). Estos datos son transmitidos hacia una aplicación en la nube (U-BASE) la cual es la encargada de procesar los datos y habilitar su disponibilidad para las aplicaciones clientes interesadas en consumir esa información.

El U-BOX actúa como máster en la red Mesh (todos los nodos interconectados) que forman los U-FLAG cercanos, y por el otro lado dispone de las interfaces necesarias para conectarse con la nube (U-BASE) a través de GPRS, Wi-Fi o Ethernet.

Este componente requiere alimentación eléctrica las 24 horas del día ya que su autonomía es solo de 4 horas. [14]

Figura 4 – U-Box

AMQP

Para integrar los sensores con el sistema y recibir en tiempo real las actualizaciones del estado de las plazas, Urbiotica provee distintos protocolos de comunicación, entre ellos el protocolo AMQP (**A**dvanced **M**essage **Q**ueuing **P**rotocol) que fue escogido por nosotros ya que funciona con una metodología de publish & subscribe a un canal de comunicación por el cual se recibe cada vez que el sensor registra un cambio de estado. También provee información acerca del estado del uso de la batería, temperatura y modo stand by (inactivo).

4.2 RFID

La tecnología RFID (del inglés **R**adio **F**requency **ID**entification) es una técnica que utiliza señales de radiofrecuencia para identificar cualquier tipo de objeto en un sistema. Básicamente, un sistema RFID consiste en pequeños receptores o tags adosados a objetos físicos que responden a señales inalámbricas de lectores RFID con información propia para ser utilizada según se necesite.

Hay muchos tipos de sistemas de RFID para distintos tipos de

aplicaciones y configuraciones, por ejemplo, distintas fuentes de alimentación, frecuencia en la que opera y diversas funcionalidades. Dependiendo de estas alternativas, el costo y dificultad de implementación de un sistema RFID puede variar sustancialmente. [15]

RFID presenta varias ventajas en comparación con otras tecnologías de identificación automática, como códigos de barra, tarjetas magnéticas, o sistemas biométricos. Estas ventajas son:

- Seguridad. Es un tipo de tarjeta que, por su diseño tecnológico, no puede duplicarse fácilmente. Cada una posee un código distinto y no permite que varios usuarios puedan tener una tarjeta duplicada. Es una diferencia fundamental en comparación con sistemas de banda magnética o código de barras donde la duplicación de tarjetas es bastante frecuente.
- No tiene necesidad de alineación. No necesita que la tarjeta sea pasada por una ranura en un determinado sentido, lo que le da una mayor practicidad y agilidad en su uso.
- Múltiples lecturas: Múltiples dispositivos pueden ser leídos simultáneamente, lo que permite ahorrar tiempo en comparación con otras tecnologías en las que es necesario alinear dispositivos u objetos para identificarlos uno por uno.
- Tarjetas sin desgaste. La tarjeta no tiene fricción alguna con el lector, por lo cual no se desgasta y su vida útil es prolongada.
- Etiquetas reutilizables. Algunos tipos de etiquetas RFID pueden ser leídas y escritas en múltiples ocasiones.

Aplicaciones de ejemplo

Los primeros ejemplos de aplicación comercial de tags RFID incluye el

tracking o seguimiento de vagones de tren, contenedores y automóviles.

Posteriormente comenzó a utilizarse para el cobro automático de peajes ubicándose tags RFID activos en los vehículos de los automovilistas interesados para que al cruzar una cabina de peaje se realice automáticamente el débito correspondiente.

A medida que los costos de manufactura de los tags fueron disminuyendo, comenzaron a emplearse en objetos de menor valor, expandiéndose a otras industrias, por ejemplo identificación de animales como mascotas, ganado o animales salvajes en peligro de extinción. Otro de los usos mas universales de los sistemas RFID consiste en tarjetas de proximidad para control de acceso de personas a lugares restringidos, o servicios públicos que utilizan tarjetas RFID para consultar datos de las personas.

Características de un sistema RFID

Existen tres componentes básicos en un sistema RFID como ilustra la Figura 5:

- El tag, etiqueta o transponder de RFID consiste en un circuito pequeño, integrado con una pequeña antena, capaz de trasmitir un número de serie único hacia un dispositivo de lectura, como respuesta a una petición. Dependiendo del tipo de tag, puede contener una batería o no.
- El lector, está compuesto por una antena, un módulo electrónico de radiofrecuencia y un módulo electrónico de control.
- Un controlador, middleware o host (comúnmente una PC) en el cual se ejecuta algún software de control conectado a una base de datos.

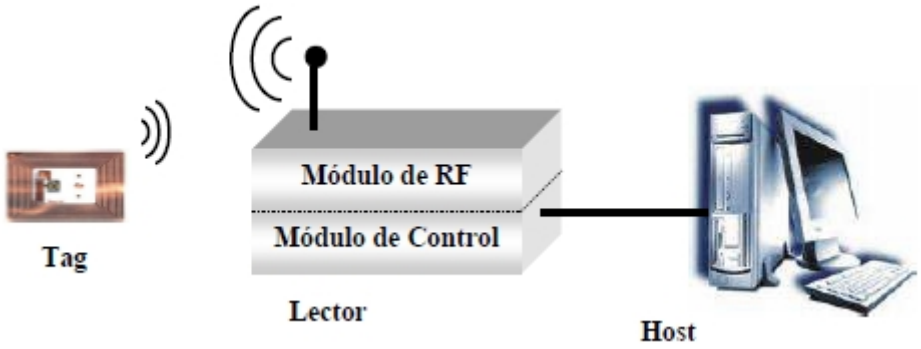

Figura 5 – Componentes de un sistema RFID

La tecnología RFID puede ser dividida principalmente en tres categorías:

- Sistemas pasivos, en los cuales los tags RFID no cuentan con una fuente de energía. Su antena recibe la señal de radiofrecuencia enviada por el lector y almacena esta energía en un capacitor, utilizándola para habilitar su circuito integrado y emitir una señal de respuesta. El rango de lectura puede ser muy limitado, con un máximo aproximado de 10 metros. Los tags en los sistemas de este tipo son económicos y de tamaño reducido.
- Sistemas activos, los cuales utilizan tags RFID con fuentes de energía integrada, como puede ser una batería por ejemplo. Este tipo de tags integra una electrónica mas sofisticada, lo que incrementa su capacidad de almacenamiento de datos y soportan un rango de lectura mayor que los sistemas pasivos, siendo de aproximadamente entre 20 y 100 metros de distancia. Los tags en los sistemas de este tipo son mas costosos y de mayor tamaño.
- Sistemas semi-activos: En este tipo de sistema el lector siempre es el encargado de iniciar la comunicación. Como los tags poseen una fuente de energía no necesitan la señal del lector para dotarse de energía (a diferencia de los tags pasivos), y esto le permite ser leídos a mayores

distancias y en un tiempo sustancialmente menor. Esto permite obtener lecturas de objetos en movimiento.

Tanto los tags activos, como los tags pasivos pueden adicionalmente ser clasificados de la siguiente manera:

- Solo lectura: En estos dispositivos los datos son programados en el tag durante su fabricación únicamente.
- Una escritura, muchas lecturas: Un tag de este tipo se puede reprogramar una única vez.
- Lectura y escritura: Estos tags pueden ser reprogramados múltiples veces. Este tipo de tags puede ser escrito por el lector, lo cual es una ventaja en comparación a las otras dos alternativas para aplicaciones que requieran de esta funcionalidad. [16]

4.3 NodeJS

Modelo orientado a eventos vs. Modelo multithreading

En la programación tradicional se ejecutan las operaciones de Entrada/Salida de la misma manera que se ejecutan las llamadas a funciones locales, es decir, el procesamiento no continua hasta que las operaciones o llamadas hayan finalizado. Este modelo de programación bloqueante proviene desde los tiempos de los sistemas “time-sharing” en los cuales cada proceso se correspondía con una tarea humana. En esos sistemas, los usuarios típicamente necesitaban terminar una tarea antes de decidir la siguiente tarea a realizar. Este modelo basado en un proceso por usuario no escalo muy bien ya que gestionar

varios procesos le genera una gran carga al sistema operativo, ya sea en memoria como así también por los costos producidos por los cambios de contexto de los mismos, y la performance de las tareas tendían a decaer luego de alcanzar una cantidad determinada.

De este modelo surge una alternativa, la programación Multithreading o multi-hilos. Un thread o hilo es, básicamente, un proceso liviano que comparte memoria con los demás threads dentro del mismo proceso. Mientras un thread está esperando a que finalice una operación de entrada/salida, otro thread puede asumir el control de la CPU (Unidad Central de Procesamiento) para un uso mas óptimo de la misma. Algunos sistemas permiten que los threads se ejecuten en paralelo en diferentes núcleos de CPU, lo cual requiere el uso de primitivas de sincronización para el acceso concurrente a datos en memoria compartida. [17]

Por ejemplo, aplicaciones como servidores Web llevan a cabo una cantidad significante de operaciones de entrada/salida, por lo tanto, el uso de múltiples threads permitiría a este tipo de aplicaciones hacer un mejor uso de los procesadores. En un sistema moderno de varios núcleos (multi-core) se pueden ejecutar threads simultáneamente en diferentes núcleos, es decir, con paralelismo real. En un sistema con un único núcleo, el procesador ejecuta un thread, cambia a otro thread y lo ejecuta, y así continuamente. Por ejemplo, el procesador cambia su contexto de ejecución para otro thread cuando el thread actual necesita escribir en un socket TCP. El cambio en el contexto de ejecución ocurre por qué completar la operación de escritura podría usar varios ciclos del procesador y, en lugar de desperdiciar ciclos de procesamiento esperando que la operación de escritura termine, el procesador inicia la operación de entrada/salida y cambia su contexto de ejecución para ejecutar otro thread, manteniéndose ocupado haciendo trabajo útil. Cuando la operación de

entrada/salida finaliza el thread esta nuevamente listo para ser ejecutado. [18]

La programación orientada a eventos, también llamada programación asíncrona, es un estilo de programación donde el flujo de ejecución es determinado por eventos. Los eventos son manejados por gestores de eventos (event handlers) o funciones callbacks. Una función callback es invocada cuando un evento significante ocurre, como por ejemplo, cuando se encuentra disponible el resultado de una consulta a la base de datos o cuando un usuario hace clic en un botón. [17]

Las operaciones de entrada/salida asíncronas son importantes para el modelo de programación orientada a eventos porque evitan que las aplicaciones se bloqueen mientras esperan que termine la ejecución de la operación. Por ejemplo, si una aplicación llena el buffer de un socket al estar escribiendo en el mismo y la operación de entrada/salida es bloqueante, el socket mantendría a la aplicación en espera hasta que haya lugar disponible para escritura en el buffer impidiéndole realizar un trabajo mas útil. En cambio, si la operación de entrada/salida es no bloqueante, el socket retornaría un mensaje indicando que por el momento no se puede seguir escribiendo en el buffer y notificaría posteriormente a quien corresponda cuando haya disponibilidad para escritura. Asumiendo que la aplicación se subscribe como interesada en este tipo de evento del socket, puede continuar realizando otras tareas sabiendo que recibirá la notificación de un evento cuando el buffer de escritura del socket tenga espacio disponible.

¿Que es NodeJS?

NodeJS es una plataforma construida sobre la máquina virtual de JavaScript V8, de código abierto, utilizado por Google Chrome. Fue implementada para crear aplicaciones de red de manera rápida y escalables.

Implementa un modelo de entrada/salida no bloqueante dirigido por eventos que lo hace ligero y eficiente, ideal para aplicaciones de tiempo real que manejan mucha intensidad de datos entre dispositivos distribuidos. [19]

NodeJS y V8 fueron desarrollados mayormente en C y C++ teniendo como objetivo lograr una alta performance y un bajo consumo de memoria.

A diferencia de otros entornos modernos, los procesos de NodeJS no se basan en multithreading (múltiples hilos) para dar soporte a la programación concurrente, sino que se basa en un modelo de eventos asíncronos de Entrada/Salida. Se puede pensar en un proceso servidor de NodeJS como un demonio single-threaded (único hilo) que tiene embebido el motor JavaScript V8 para soportar customización del mismo.

Para esta plataforma, el lenguaje de programación JavaScript es un complemento excelente ya que soporta eventos callbacks, es decir, permite crear funciones anónimas que se pueden registrar como gestoras o controladoras (handler) de eventos para ser ejecutadas cuando los mismos ocurran. [18]

NodeJS, al ser un entorno JavaScript en el lado del servidor, brinda algunos beneficios extras:

- Los desarrolladores pueden escribir aplicaciones web en un solo lenguaje de programación, lo que ayuda a reducir el cambio de contexto entre el desarrollo del cliente y el del servidor, y permite reutilizar código entre ambos.
- JSON (**J**ava**S**cript **O**bject **N**otation) es un formato de intercambio de datos actualmente muy utilizado y su sintaxis deriva de JavaScript.
- JavaScript es un lenguaje utilizado en varias bases de datos NoSQL (Por ejemplo, MongoDB), por lo tanto reduce la complejidad en la integración con las mismas.

- Utiliza la máquina virtual V8 que implementa el estándar ECMAScript, es decir, se pueden aprovechar las nuevas características del lenguaje JavaScript directamente en Node, sin necesidad de esperar a que los navegadores las implementen. [20]

Fundamentos de programación en NodeJS

El estilo de programación orientado a eventos de NodeJS es una de las características que lo define. Este modelo esta acompañado por un bucle de eventos (event loop). Un bucle de eventos es un estructura que principalmente lleva a cabo dos funciones en un bucle continuo, detección de eventos y propagación de eventos para su gestión. Durante su ejecución, el bucle de eventos debe detectar que eventos ocurren, y ante la ocurrencia de los mismos, debe determinar la función callback correspondiente e invocarla.

El bucle de eventos se puede definir como un thread ejecutándose dentro de un proceso, lo que significa que cuando un evento ocurre, el controlador (handler) del evento puede ejecutarse sin interrupciones. Es decir, dado un momento determinado a lo sumo solo habrá ejecutándose un controlador de eventos y cualquiera de estos en ejecución podrá completar su tarea sin ser interrumpido. Esto le permite a los programadores abstraerse de los requisitos necesarios para modelar la sincronización entre threads y no tener que preocuparse por el estado de la memoria compartida y la ejecución concurrente de los threads. [17]

NodeJS brinda la posibilidad de utilizar JavaScript del lado del servidor, de la misma manera que los navegadores brindan la posibilidad de utilizar JavaScript del lado del cliente. Por este motivo, es importante entender como funcionan los navegadores con el fin de entender como funciona NodeJS.

Ambos implementan un bucle de eventos y son no bloqueantes para realizar operaciones de entrada/salida.

Cuando es necesario llevar a cabo una operación de entrada/salida en el navegador, la misma se realiza fuera del bucle de eventos y cuando la misma finaliza se emite un evento que es manejado por la función callback correspondiente. En la Figura 6 se refleja lo mencionado anteriormente.

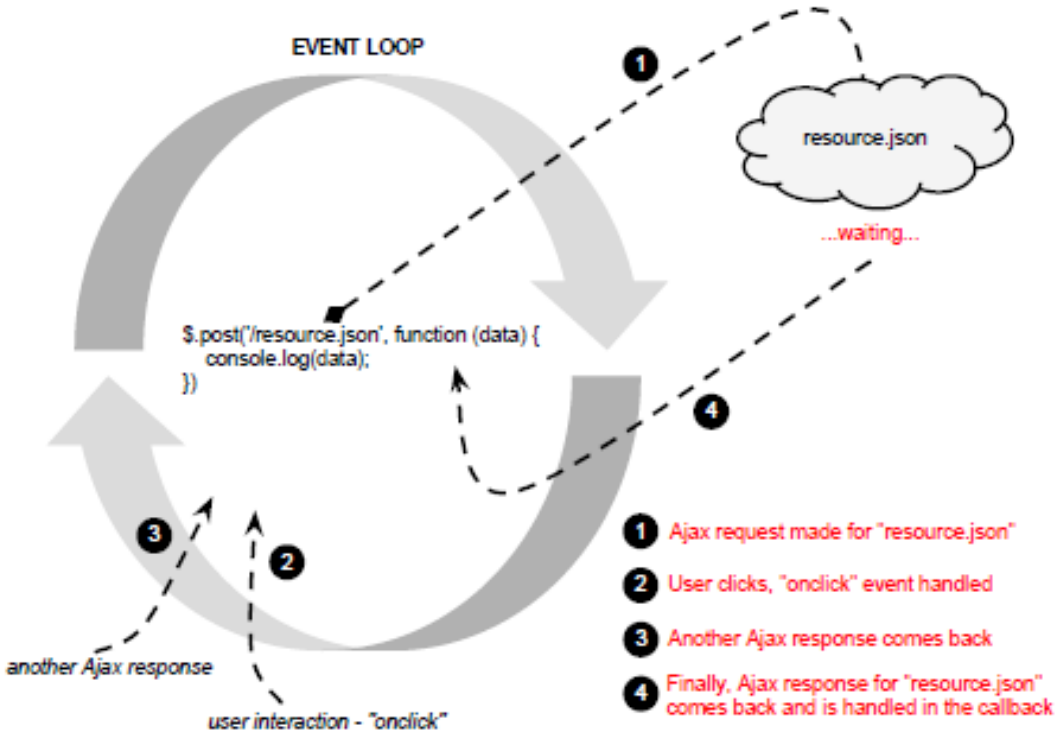

Figura 6: Operación de entrada/salida no bloqueante en un navegador web.

La operación de entrada/salida del ejemplo es llevada a cabo de manera asíncrona y no bloqueante respecto al principal hilo de ejecución. De esta manera, el bucle de eventos puede atender cualquier otra interacción o solicitud que provenga desde la página. Esto le permite al navegador poder responder a las necesidades del usuario y gestionar gran cantidad de interacciones del mismo con la página sin espera innecesaria.

En NodeJS, las operaciones de entrada/salida casi siempre son llevadas a cabo fuera del bucle principal de eventos (main event loop), como se muestra en la Figura 7, permitiendo al servidor recibir solicitudes y conexiones eficiente y constantemente.

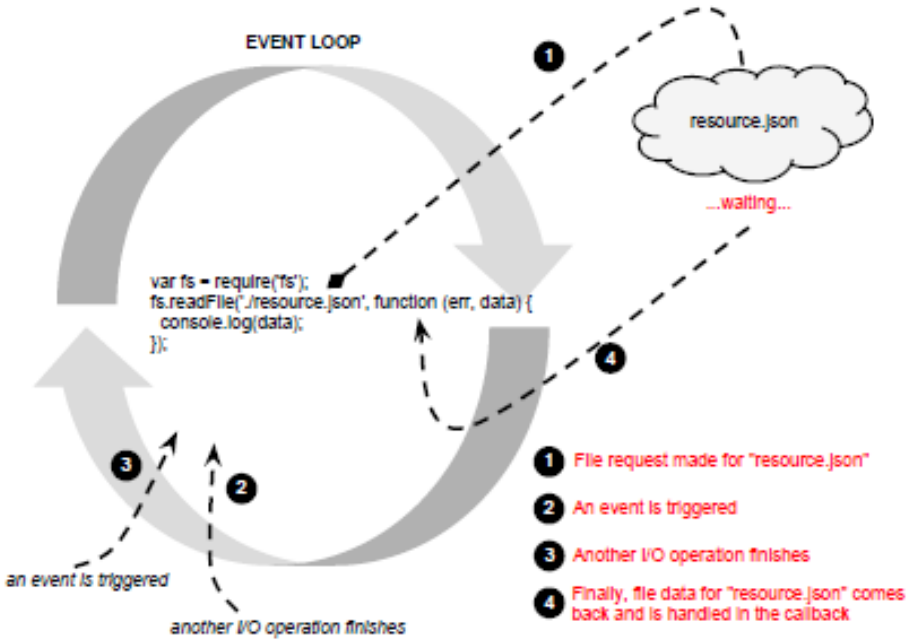

Figura 7: Operación de entrada/salida no bloqueante en Node.

En el ejemplo de la Figura 7, se procesa una solicitud de lectura del archivo “resource.json” (1) de manera asíncrona y se prosigue con la ejecución del script sin esperar que la operación de entrada/salida finalice. En segundo lugar se dispara un evento (2), luego finaliza una operación de entrada/salida diferente (3) y por último (4) se recuperan y se imprimen en consola los datos del archivo solicitados en el paso 1.

Al gestionar los eventos de manera asíncrona, rara vez permite que un proceso alcance un tiempo límite de entrada/salida ya que la latencia producida no sería lo suficientemente grande como para bloquear el servidor, como si

puede pasar en las operaciones de entrada/salida bloqueantes. Este modelo le permite a Node ser un servidor liviano y/o ligero para llevar a cabo las operaciones que, generalmente, un servidor que maneja comunicaciones sincrónicas las realiza de manera lenta.

Esta combinación de modelo de programación orientada a eventos junto a un modelo asíncrono de JavaScript del lado del servidor promueven la creación de aplicaciones en tiempo real con intensivo manejo de datos (DIRTy Application).

Las aplicaciones DIRTy (**D**ata-**I**ntensive **R**eal-**T**ime application) son el tipo de aplicaciones para las cuales fue diseñado NodeJS. Las aplicaciones de tiempo real son tendencia y están ganando territorio en la web, ya sea en aplicaciones web existentes que mejoran su comportamiento al incluir componentes de tiempo real, o con la aparición de nuevas aplicaciones con mayor capacidad de respuesta.

Este tipo de aplicaciones requieren una plataforma que pueda responder casi instantáneamente a un alto número de usuarios concurrentemente. Node está bien diseñado para estos casos, como así también, para aplicaciones con mucha demanda de operaciones de entrada/salida. [20]

Módulos

NodeJS implementa el estándar CommonJS para modularización. Cada módulo tiene su propio contexto, lo que significa que no afectan el contexto global de la aplicación ni interfieren con otros módulos.

En NodeJS, los módulos están referenciados por la ruta del archivo o por nombre. Aquellos que vienen integrados en NodeJS son cargados cuando el proceso Node arranca. Además de los del núcleo de NodeJS, existe una amplia

cantidad de módulos desarrollados por terceros (3rd-Party) que se pueden instalar usando el gestor de dependencias de NodeJS llamado NPM (Node Package Manager) o, por último, se pueden importar módulos desarrollados por uno mismo.

Cualquier módulo de cualquiera de los tipos mencionados anteriormente, expone una API pública que el programador puede utilizar, luego de que el módulo haya sido importado o requerido en el script correspondiente mediante la función require.

El sistema de módulos CommonJS es la única manera de compartir objetos o funciones entre archivos en Node. Para definir lo que se quiere exponer públicamente en cada módulo se utiliza el objeto export del módulo actual. Para hacer referencia al módulo en el que se está situado, existe la variable module, por lo tanto, con module.exports se exporta la funcionalidad deseada para los scripts que importen ese módulo.

Node trae incorporado varios módulos compilados dentro de su distribución. Estos son llamados módulos del núcleo (core modules), y su importación se hace por su nombre y no por la ruta del archivo, y son preferentemente cargados incluso si existen módulos de terceros (3rd-party modules) con el mismo nombre. Para importar módulos instalados mediante NPM se debe proveer de la ruta de archivo, ya sea absoluta o relativa al archivo actual. En el caso de que se indique el nombre del módulo solamente y el mismo no sea del núcleo, Node intentará buscarlo dentro de la carpeta node_modules en el directorio actual. Si falla la búsqueda del archivo, intentará localizar la carpeta node_modules del directorio padre y así continuamente hasta alcanzar el directorio raíz o encontrar el módulo que se está buscando.

La carpeta mencionada anteriormente node_modules es el directorio por defecto donde NPM descarga los módulos que van a ser usados en Node. [17]

Algunos de los principales módulos del núcleo de Node son los

siguientes:

- Buffer
  - Cuando se interactúa con el sistema de archivos o flujos de bytes en comunicaciones TCP, es necesario manipular datos binarios y Buffer es el objeto global que Node utiliza por defecto para el manejo de los mismos. Se corresponde con una asignación de memoria estática, fuera de la memoria heap (asignación dinámica) de la máquina virtual V8. El tamaño del Buffer se indica al momento de su creación y no puede modificarse posteriormente.
- Console
  - Exporta el objeto Console que permite enviar mensajes por la salida estándar del proceso STDOUT o STDERR en caso de transmitir errores.

    Los mensajes pueden ser de diferentes niveles: log, info, warning, debug, error.
- Events
  - Exporta el objeto EventEmitter. La mayoría de los módulos del núcleo de Node se desarrolla en torno a una arquitectura asíncrona dirigida por eventos en la cual ciertos grupos de objetos (llamados “emmiters”) periódicamente emiten eventos producen que objetos Function (“listeners”) sean ejecutadas.
  - Todos los objetos que emiten eventos son instancias de la clase EventEmitter. Estos objetos exponen una función #eventEmmiter.on() que permite vincular una o mas funciones al evento emitido por el objeto.
- File System
  - Este módulo provee funciones para manejar operaciones de

entrada/salida de archivos basándose en el estándar POSIX. Todas las funciones que provee se pueden ejecutar de manera asíncrona o de manera sincrónica.

- HTTP
  - Este módulo fue diseñado para soportar varias de las características del protocolo HTTP. Particularmente, permite hacer uso de mensajes HTTP (request y response) mediante un mecanismo (chunked transfer encoding) en el cual los mensajes se envía por partes (chunks).
- HTTPS
  - HTTPS es el protocolo HTTP sobre TLS/SSL. En Node esto se implementa en un módulo separado.
- NET
  - Esté módulo provee la funcionalidad para crear y gestionar servidores y clientes de manera asíncrona.
- UDP / DGRAM
  - Provee la implementación de sockets UDP.
- TIMER
  - Todas las funciones de este módulo son globales, es decir, no es necesario importarlas explícitamente mediante la función require(). Estas funciones permiten el manejo de contadores que realizarán una acción especificada culminado el tiempo que tienen programado.
- TLS/SSL
  - Este módulo provee funcionalidad para establecer comunicaciones mas seguras a través del protocolo TLS/SSL. [21]

4.4 Android

Las aplicaciones Android son escritas en Java. Las herramientas del SDK (Software Development Kit) de Android compilan el código, junto con otros recursos, y lo empaquetan en un archivo con extensión .apk. Un archivo APK (Android Package) contiene todo el contenido de una aplicación Android y es el archivo que utilizan los dispositivos Android para la instalación de la misma.

Una vez instalada en un dispositivo Android, cada aplicación se ejecuta en un entorno seguro debido a las siguientes características:

- El sistema operativo Android es un sistema Linux multiusuario en el que cada aplicación es un usuario diferente. Por defecto, el sistema operativo asigna a cada aplicación un único ID de usuario y define permisos para todos los archivos de la aplicación de manera tal que solo los pueda acceder la aplicación con el ID de usuario correspondiente.
- Cada proceso cuenta con su propia máquina virtual (VM por sus siglas en inglés) por lo que la ejecución del código de una aplicación es aislada de la ejecución de otras aplicaciones.
- Por defecto, cada aplicación se ejecuta en su propio proceso de Linux. Android inicia el proceso cuando cualquiera de los componentes de una aplicación necesita ser ejecutado. Por otro lado, puede decidir terminar con el proceso cuando ya no es necesario o cuando el sistema necesite recuperar memoria para otras aplicaciones y existen procesos con baja prioridad de ejecución.

Cada aplicación, por defecto, solo tiene acceso a los componentes que requiere para llevar a cabo su trabajo. Esto determina un entorno seguro en el cual cada aplicación no puede acceder a partes del sistema para el cual no tiene

los permisos necesarios. No obstante, existen métodos para que una aplicación pueda compartir datos con otras aplicaciones, o para que pueda acceder a los servicios que ofrece el sistema operativo.

Es posible que dos aplicaciones compartan datos si tienen el mismo ID de usuario de Linux, y en ese caso tendrían acceso a los archivos de cada una.

Para ahorrar recursos del sistema operativo, las aplicaciones con el mismo ID de usuario pueden ejecutarse en el mismo proceso Linux y compartir la máquina virtual.

Para que esto sea posible, las aplicaciones deben ser firmadas con el mismo certificado, caso contrario no podrán ejecutarse en el mismo proceso y por ende, no podrán compartir datos entre si.

Por otro lado, una aplicación puede solicitar permisos explícitamente para acceder a datos y/o recursos del dispositivo tales como contactos de usuario, mensajes SMS, cámara del dispositivo, Bluetooth, entre otros.

Principales componentes de una aplicación Android

Cada componente de una aplicación Android cumple un rol específico que ayuda a definir el comportamiento global de la misma. Algunos de ellos son puntos de entrada a la aplicación, ya sea para proveer acceso al sistema operativo o a otras aplicaciones. A continuación se detallan los cuatro tipos de componentes existentes en una aplicación Android.

Actividades

Una actividad (Activity) es el componente de la aplicación que provee una pantalla que implementa una interfaz de usuario para que los usuarios puedan

interactuar con la aplicación.

Una aplicación generalmente contiene múltiples actividades que están débilmente acopladas entre sí. Típicamente, una actividad en una aplicación es designada como actividad principal (main activity), la cual es presentada al usuario cuando se inicia la aplicación la primera vez. Cada actividad tiene la capacidad de iniciar otra actividad para permitir realizar diferentes acciones.

Cuando se inicia una nueva actividad la anterior se detiene pero el sistema operativo la preserva en una pila de actividades denominada “back stack”. La actividad mas recientemente iniciada se coloca en el tope de la pila y se muestra al usuario. Cuando el usuario deja de interactuar con la actividad actual y presiona el botón “atrás”, dicha actividad se quita del tope de la pila, el sistema operativo la destruye, y se resume la próxima actividad de la pila.

Una actividad provee varios métodos “callback” que le permiten ser notificada por el sistema operativo cuando se altera el ciclo de vida de la misma. Es decir, estos métodos se ejecutan cuando el sistema operativo crea, detiene, resume o destruye una actividad, y brindan la posibilidad de implementar un comportamiento específico que sea apropiado para el cambio de estado de la actividad.

Ciclo de vida de una Actividad

El ciclo de vida de una actividad está directamente afectado por su relación con otras actividades, sus tareas y la pila “back stack”. Una actividad puede tener estar en uno de los siguientes estados:

- Reanudada o en ejecución: la actividad está ejecutándose en primer plano en interacción con el usuario.
- Pausada: Otra actividad se encuentra ejecutándose en primer plano, pero

esta todavía es visible. Esto significa que otra actividad es visible por encima de la actividad pausada, ya sea al ser parcialmente transparente o por no cubrir la pantalla en su totalidad. La actividad pausada sigue estando en ejecución (mantiene su estado), pero puede ser destruida por el sistema operativo en situaciones donde la memoria disponible sea extremadamente baja.

- Detenida: Una actividad que se encuentra detenida está completamente ocultada por otra actividad, es decir, paso a ser ejecutada en segundo plano o en background. En este estado, la actividad tiene prioridad alta para ser destruida en caso de que el sistema operativo necesite reservar memoria en cualquier otro contexto de ejecución.

El sistema operativo puede quitar una actividad de la memoria ya sea pidiéndole que finalice, a través de la invocación de su método #finish(), o simplemente matando el proceso donde la misma se está ejecutando. Cuando posteriormente se inicia la actividad (luego de ser finalizada o destruida), la misma debe crearse nuevamente.

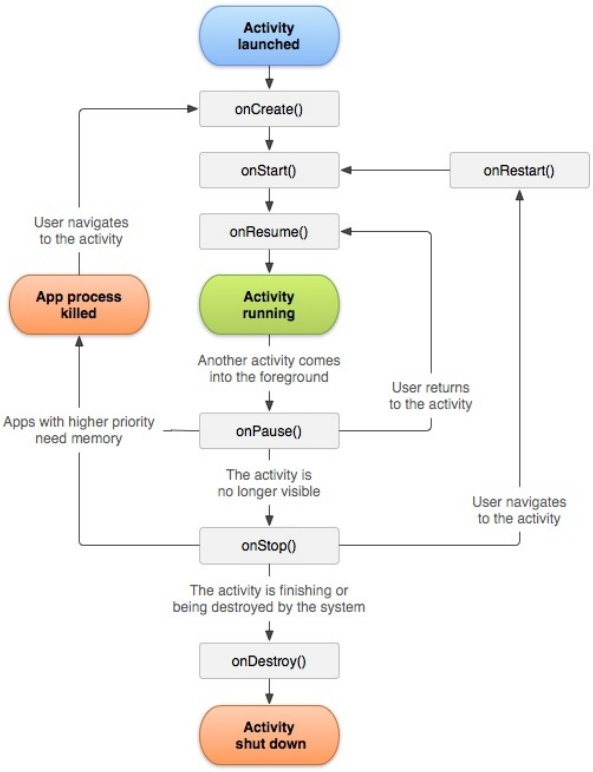

Figura 8 – Ciclo de vida de una Actividad

La Figura 8 define el ciclo de vida completo de una actividad. Con la implementación de los métodos callback (representados con los rectángulos) se puede controlar el comportamiento de la actividad en todos sus estados.

Servicios

Un servicio es un componente de una aplicación Android que permite llevar a cabo operaciones de ejecución prolongada en segundo plano (o background) y no provee interfaz de usuario. Un servicio puede iniciarse y comportarse de dos maneras diferentes:

- Unbounded Service: Un servicio es “iniciado” cuando un componente de

la aplicación inicia el servicio invocando el método #startService(). Una vez iniciado, el servicio puede ejecutarse en background indefinidamente, incluso si el componente que lo inició es destruido. Por lo general, un servicio iniciado de esta manera lleva a cabo una única operación en background y no retorna ningún resultado al componente que lo invocó.

- Bounded Service: Un servicio está ligado a un componente de la aplicación cuando se invoca mediante el método #bindService(). Un servicio iniciado de esta manera ofrece una interfaz de cliente-servidor para que los componentes de la aplicación puedan interactuar con él, enviándole solicitudes, obteniendo resultados, e incluso a través de comunicaciones entre procesos (IPC). Por lo general, este tipo de servicios se ejecuta mientras tenga algún componente de la aplicación vinculado, caso contrario el servicio es destruido.

Ciclo de vida de un servicio

El ciclo de vida de un servicio, representado en la Figura 9, es mas simple que el de una actividad. Sin embargo, se debe prestar mayor atención a la creación y destrucción de un servicio ya que el mismo puede estar ejecutándose en background sin que el usuario sea consciente de ello.

En función de la manera en que un servicio es iniciado, el ciclo de vida puede darse de dos maneras diferentes:

- Unbounded Service: El servicio es creado cuando un componente de la aplicación invoca el método #startService(). Una vez iniciado el servicio, el mismo se ejecuta indefinidamente y debe detenerse por su cuenta invocando el método #stopSelf(). Otro componente de la aplicación podría detener el servicio invocando el método #stopService(). En

cualquiera de las dos situaciones, cuando el servicio es detenido, el sistema operativo lo destruye.

- Bounded Service: El servicio es creado cuando un componente de la aplicación invoca el método #bindService(). El cliente luego se comunica con el servicio (servidor) a través de la interfaz IBinder, la cual describe un protocolo abstracto para interactuar con objetos remotos. El cliente puede cerrar la conexión y desvincularse del servicio invocando a método #unboundService(). En el momento en que el servicio no tenga vinculado ningún cliente, el sistema operativo destruye el servicio por lo que en este caso no es necesario que se detenga por su propia cuenta.

Puede darse el caso de que estas dos maneras de interactuar con un servicio no se den de manera aislada, es decir, un componente de la aplicación podría vincularse a un servicio que previamente había sido iniciado con el método #startService(). En este caso, los métodos #stopService() o #stopSelf() no detienen la ejecución del servicio hasta que todos los clientes se hayan desvinculado.

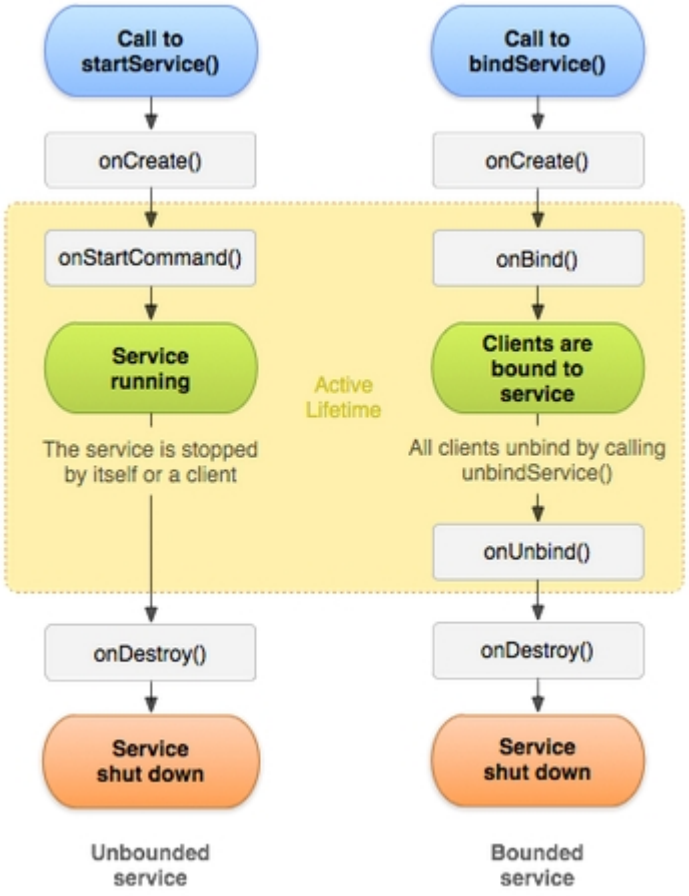

Figura 9– Ciclo de vida de los servicios. El diagrama de la izquierda muestra el ciclo de vida cuando se inicia el servicio con startService() y el diagrama de la derecha cuando se inicia con bindService()

Content Providers

Un componente de aplicación Content Provider es la interfaz estándar que permite compartir datos entre distintos procesos o aplicaciones. Gestiona el acceso a un conjunto de datos que pueden ser compartidos entre diferentes aplicaciones, es decir, una aplicación puede almacenar datos en cualquier lugar de almacenamiento (base de datos SQLite, file system, etc) y a través de un

content provider otra aplicación podría consultar dichos datos o incluso modificarlos.

Los Content Providers también son útiles para leer y/o escribir datos que son privados a la aplicación y no son compartidos, por ejemplo, se podría utilizar como interfaz a una base de datos SQLite.

Básicamente, un content provider brinda datos a una aplicación externa en forma de tablas, similares a las tablas de una base de datos relacional. Una fila representa una instancia de un tipo de datos que el provider gestiona, y cada columna en la fila representa un atributo de esa instancia.

Una aplicación puede acceder a los datos que son provistos por el content provider de otra aplicación a través de un objeto cliente denominado ContentResolver que implementa las funciones básicas CRUD (Create, Retrieve, Update, Delete) de un almacenamiento persistente. De esta manera, se produce una comunicación entre procesos (IPC) entre el objeto ContentResolver en el proceso de la aplicación cliente y el objeto ContentProvider en la aplicación que expone el componente Content Provider, es decir, la aplicación que permite el compartimiento de los datos.

La aplicación que comparte los datos a través de un content provider puede especificar permisos que son requeridos para aquellas aplicaciones que quieran acceder a los datos. Estos permisos aseguran que el usuario (aplicación cliente) conoce los datos de la aplicación que intentará consultar. Si la aplicación dueña de los datos no especifica ningún permiso no se le dará acceso a los datos a otras aplicaciones. Sin embargo, otros componentes dentro de la misma aplicación siempre tendrán permiso de lectura y/o escritura sobre los datos compartidos, independientemente de los permisos especificados.

Broadcast Receivers

Un broadcast receiver es un componente de aplicación que responde a mensajes de difusión de todo el sistema operativo, aunque también podrían ser emitidos por una aplicación.

Comúnmente un broadcast receiver cumple el rol de gateway con otros componentes, es decir, recibe un mensaje de difusión del sistema operativo o emitido por alguna aplicación y en función de eso inicia un servicio para que realice cierto trabajo basado en el evento ocurrido. Existen dos tipos de mensajes de difusión que pueden ser recibidos:

- Normales (Normal broadcasts): son completamente asíncronos. Todos los componentes Broadcast Receivers de este tipo de mensaje de difusión se ejecutan en un orden indefinido, y probablemente al mismo tiempo. Esto es mas eficiente, pero no se puede tener un control sobre el procesamiento de los mismos.
- Ordenados (Ordered broadcasts): son enviados a un Broadcast Receiver a la vez. Como cada uno se ejecuta por turnos en función de una prioridad asignada (dos broadcast receivers con la misma prioridad se ejecutan en un orden arbitrario), pueden propagar el resultado al siguiente broadcast receiver o incluso abortar la propagación del mensaje de difusión.

Los broadcast receivers se pueden registrar en una aplicación de dos maneras. Una es registrar dinámicamente una instancia de estos componentes utilizando el método #registerReceiver() y la otra es de manera estática a través de la declaración de un tag <receiver> en el archivo de configuración AndroidManifest.xml. Al ser registrados de cualquiera de estas dos formas, los

receivers son, por naturaleza, servicios entre aplicaciones. Es decir, otras aplicaciones externas puede usar estos componentes por lo que hay que tener en cuenta ciertas cuestiones de seguridad, como por ejemplo, se puede evitar que un broadcast receiver reciba mensajes desde otras aplicaciones externas declarándolo como no disponible mediante una propiedad de configuración o bien, al momento de enviar un mensaje de difusión o registrar un broadcast receiver, requerir cierto permiso para que reciban o envíen los mensajes solamente aquellos que estén habilitados para hacerlo.

Una solución para controlar el uso de los mensajes de difusión, en lugar de considerar los aspectos de seguridad mencionados anteriormente, es utilizar un servicio que provee la API de Android para registrar y enviar mensajes de difusión solamente dentro del proceso de ejecución de la aplicación. Este servicio se llama LocalBroadcastManager y brinda ciertas ventajas en comparación con enviar mensajes de difusión de manera global. Además de ser un mecanismo mas eficiente, mediante este servicio los datos que se envían como mensajes de difusión no saldrán de la aplicación, por lo que no hay que preocuparse por la exposición de datos privados a aplicaciones externas; tampoco es posible para otras aplicaciones enviar esos mensajes de difusión a nuestra aplicación, por lo cual no habría que preocuparse por cualquier agujero de seguridad que pueda ser aprovechado por aplicaciones externas.

Ciclo de vida de un Broadcast Receiver

Un Broadcast Receiver se ejecuta solamente el tiempo que dure la invocación a su método #onReceive(), donde se procesa el mensaje de difusión recibido. Una vez finalizada la ejecución de ese método, el sistema operativo considera que el componente no está mas activo y que puede ser finalizado.

Activación de los componentes

Las actividades, servicios y broadcast receivers son activados o iniciados por un mensaje asíncrono denominado intención (intent por sus siglas en inglés). Una intención es creada como una instancia de la clase Intent, la cual representa un mensaje para activar un componente específico o un tipo específico de componente, clasificándose como una intención explicita o implícita, respectivamente. En la figura 10 se muestra un ejemplo de la utilización de una intención implícita.

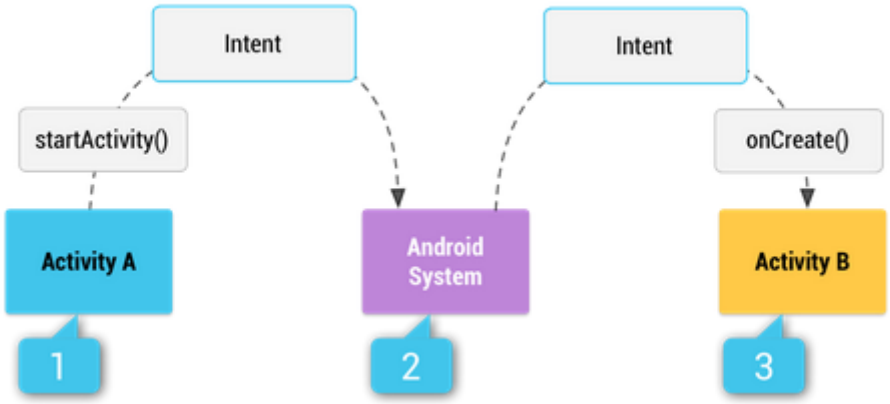

Figura 10. Intención implícita enviada desde una actividad hacia el sistema operativo para iniciar otra actividad.

Para las actividades y servicios, una intención define una acción que se necesita llevar a cabo. Por ejemplo, una intención puede transmitir una solicitud a una actividad para que muestre una imagen o una página web. En algunos casos, se puede iniciar una actividad para recibir un resultado específico y en ese caso, la actividad iniciada también retorna una intención con el resultado de la invocación (un ejemplo podría ser una intención creada para permitir que el

usuario seleccione un contacto personal y que sea retornado a nuestra aplicación, en ese caso la intención retornada incluiría una URI apuntando al contacto seleccionado por el usuario).

Para los componentes broadcast receivers la intención simplemente define el mensaje de difusión que se quiere enviar, por ejemplo, el sistema operativo envía un mensaje de difusión con una intención cuando el celular empieza a cargar su batería, la cual contiene un mensaje que describe la ocurrencia de ese evento.

El otro tipo de componente, content provider, no es activado a través de intenciones. En su lugar, son activados cuando se genera una solicitud proveniente de una instancia de ContentResolver, es decir, de un objeto cliente que inicia una comunicación para consumir los datos que expone el ContentProvider.

Para que cualquiera de estos componentes pueda ser iniciado, el sistema operativo debe saber de su existencia, y eso se logra declarando todos los componentes de la aplicación en el archivo principal de configuración denominado AndoridManifest.xml, el cual debe estar alojado en el directorio raíz de la aplicación. [22]

4.5 MongoDB

MongoDB es una base de datos NoSQL open-source orientada a documentos cuyos principales objetivos son proveer alta performance, alta disponibilidad y escalabilidad automática. [23]

Para poder justificar la decisión de utilizar MongoDB en vez de una RDBMS tradicional, debemos antes explicar las diferencias y motivaciones relacionadas con el uso de bases de datos NoSQL.

NoSQL

Se denomina base de datos NoSQL a toda BBDD que no almacena sus datos utilizando el modelo relacional tradicional de los RDBMS. Las BBDDs NoSQL pueden almacenar sus datos de variadas formas; en pares clave-valor, en forma de documentos, en base a columnas, etc.

Las motivaciones que llevaron al desarrollo y uso de BBDDs NoSQL tienen que ver con algunas falencias que poseen los RDBMS, en particular con respecto a los tiempos de respuesta y la escalabilidad.

Complejidad innecesaria

Los RDBMS proveen una gran variedad de herramientas para asegurar consistencia en los datos. Esto no es siempre deseable, e inclusive, muchas veces, las propiedades ACID de las bases de datos relacionales exceden las necesidades de ciertos tipos de aplicaciones.

Alto rendimiento

Algunas soluciones NoSQL proveen mayor rendimiento que las bases de datos tradicionales.

Escalabilidad horizontal

Las bases de datos NoSQL intentan resolver los siguientes problemas de los RDBMS:

- Escalamiento de los datos
- Performance de servidores únicos
- Esquemas de datos rígidos

En contraste con los RDBMS, las bases de datos NoSQL están diseñadas para escalar horizontalmente y no depender de hardware dedicado. Los nodos de un clúster NoSQL pueden agregarse y quitarse sin requerir el mismo esfuerzo operacional que requiere la misma operación en un RDBMS.

Evitar mapeo Objeto-Relacional

La mayoría de las DDBB NoSQL están diseñadas para almacenar estructuras de datos simples o más similares a las utilizadas en lenguajes OOP. Esto evita el costo del mapeo objeto-relacional que tradicionalmente se lleva a cabo en soluciones con lenguajes OOP que utilizan RDBMS para el almacenamiento de los datos. Esto es particularmente importante para aquellas aplicaciones que utilicen estructuras de datos simples y raramente puedan beneficiarse de las características de una base de datos relacional.

La caducidad de la metodología “One size fit’s it all”

“One size fit’s it all” es una frase en inglés que se refiere a la búsqueda de una única solución para todos los problemas. Esta es la manera con la que tradicionalmente se pensaban a las RDBMS, considerándolas prácticamente la única solución viable para el problema del almacenamiento de los datos de una aplicación.

Actualmente, este no es el caso, dado que hay un sinfín de aplicaciones para las cuales las RDBMS no son la solución apropiada, marcadas por estas dos principales tendencias:

- El crecimiento continuo de los volúmenes de datos (a guardar)
- El crecimiento continuo de la cantidad de datos a procesar en menor tiempo

Esto se da en numerosas aplicaciones en donde no hay una necesidad real de que se cumplan las metodologías ACID de las bases de datos tradicionales. Por ejemplo, una red social no requiere que en el mismo exacto momento, dos personas consultando el último posteo de una tercera, reciban el mismo dato exacto. Si bien no es el caso ideal, aceptar este tipo de inconsistencias temporales quita el gran peso que tienen los RDBMS a la hora de escalar y puede proveer una gran flexibilidad en el uso de los datos y la lógica de aplicación.

Obviamente, el movimiento NoSQL ha sido criticado desde varios sectores, particularmente indicando que no todos los RDBMS fallan a la hora del escalamiento o “sharding” o que quizás muchas de las aplicaciones a las cuales aplican las BBDD NoSQL, pueden ser atacadas con un RDBMS apoyado con un sistema de caching (como por ejemplo, memcached), y así obtener las bondades de los dos mundos. De todas maneras, distintos críticos del medio siguen defendiendo las bases de datos NoSQL, principalmente argumentando que:

- En muchos casos, los modelos de datos no encajan bien en el modelo de datos relacional y para ello se necesitan soluciones más flexibles
- Dada la simplicidad de la mayoría de los motores NoSQL, las operaciones de lectura y escritura (así como el escalamiento y el “sharding” de las

instancias) obtienen una perfomance realmente mayor al de los RDBMS para grandes volúmenes de datos. [24]

Con esta introducción general a las bases de datos NoSQL, podemos ahora, detallar un poco más los aspectos particulares de MongoDB.

Características principales de MongoDB

Como ya dijimos, MongoDB es una base de datos NoSQL orientada a documentos (se explicará más adelante). Las principales ventajas del uso de MongoDB son las siguientes:

- Alta Performance
  - La posibilidad de almacenar documentos embebidos reduce las operaciones de I/O en el sistema.
  - Soporta índices que permiten realizar consultas más rápidamente y se pueden utilizar objetos complejos como claves para tal caso.
- Alta Disponibilidad
  - Permite crear réplicas que aseguran:
    - Redundancia de datos.
    - Tolerancia automática ante fallas.
- Escalabilidad Automática
  - MongoDB provee escalabilidad horizontal de manera nativa:
    - Distribuyendo los datos automáticamente en un clúster (sharding).
    - Provee consistencia para aplicaciones de alto rendimiento y baja latencia.

- Documentos
  - Una tupla en MongoDB es un documento, que no es más que una estructura de datos de tipo clave-valor, similares a un objeto JSON. MongoDB permite que los valores de los claves en los documentos no sean solamente tipos simples, sino también compuestos, como arreglos y otros documentos (embebidos).

Documentos

El objeto básico en MongoDB es el documento. Este es un conjunto ordenado de claves y valores. La representación de dicho documento varía según el lenguaje de programación (puede ser un mapa, hash o diccionario). En JavaScript se los representa simplemente con un objeto.

Las claves de un documento son strings. Cualquier carácter UTF-8 puede ser usado en una clave, con sólo unas pocas excepciones:

- Una clave no puede contener el carácter \0 (Carácter nulo), el cual es utilizado para determinar el fin de una clave.
- Los caracteres . y $ tienen una semántica especial dentro del universo de MongoDB y su uso debería ser evitado. En general, los drivers de Mongo no permiten el uso de estos caracteres en lugares inapropiados.

Cabe destacar también que los documentos en Mongo son type-sensitive y case-sensitive (es decir, distingue entre mayúsculas y minúsculas y también entre distintos tipos nativos de datos).

Por ende, en Mongo los dos siguientes documentos son considerados distintos:

- {“foo”: 3}
- {“foo”: “3”}

Cómo lo son también estos:

- {“foo”: “bar”}
- {“Foo”: “bar”}

Otro punto importante a notar es que un documento no puede contener claves duplicadas.

Cómo último punto debemos mencionar el orden de los pares clave-valor. MongoDB distingue entre documentos según el orden, a pesar de que en muchos lenguajes de programación, dicho orden no es importante (o ni siquiera es mantenido).

Los drivers para MongoDB en dichos lenguajes generalmente implementan algún mecanismo para mantener el orden, en los casos en que sea necesario.

Colecciones

Una colección es simplemente un conjunto de documentos agrupados lógicamente. Si un documento es el análogo en MongoDB a lo que es una tupla en una BBDD relacional, una colección sería el análogo a una tabla.

En MongoDB las colecciones pueden ser heterogéneas, es decir, una colección puede tener elementos de distintos esquemas. Esto trae la pregunta: ¿Para qué entonces tener distintas colecciones? Hay varias respuestas para esto:

- Mantener distintos tipos de documentos en una misma colección complica muchísimo la tarea de los desarrolladores.

- Es mucho más rápido consultar un lista homogénea de documentos que una heterogénea ala cual a la vez tenemos que filtrar el tipo de documentos que estemos consultando.
- El tener documentos similares en una colección permite el aprovechamiento de la localidad de los datos.
- Tener colecciones homogéneas permite crear índices para mejorar la perfomance de las consultas.

De la misma manera que con las claves de un documento, las colecciones pueden tener cualquier secuencia de caracteres UTF-8 como nombres, con sólo las siguientes restricciones:

- El string vacío (“”) no es un nombre válido
- No pueden contener el carácter \0 (carácter nulo)
- No deben comenzar con el prefijo system. ya que dicho prefijo se utiliza para colecciones internas.
- No deberían contener el carácter “$” ya que se utiliza para algunas colecciones generadas automáticamente por algunos drivers de MongoDB

Bases de datos

Además de agrupar documentos en colecciones, MongoDB agrupa colecciones en bases de datos. Una instancia de MongoDB puede contener distintas bases de datos, cada una con permisos diferentes y distintos archivos en disco. La regla de facto general es usar una base de datos por aplicación.

Como las colecciones, las bases de datos se identifican por nombre. Este

tiene más restricciones que las colecciones, puesto que puede ser cualquier secuencia de caracteres UTF-8 que cumplan con lo siguiente:

- Que no sea un string vacío (“”)
- No contener los siguientes caracteres: /, \, ., ", *, <, >, :, |, ?, $, (espacio simple) ni el \0 (carácter nulo). Básicamente, solo ASCII alfanumérico.
- Los nombres son case-sensitive, incluso aunque el sistema de archivos del SO no lo sea.
- Los nombres están limitados a 64 bytes.

Estas restricciones tienen que ver con que cada base de datos se guarda en archivos. Esto también explica algunas de las otras restricciones que se detallaron anteriormente.

También existen ciertos nombres de bases de datos reservadas que se pueden acceder pero tienen cierta semántica especial:

- admin
  - Es la base de datos “root” de la instancia. Si un usuario es agregado a la base de datos admin, ese usuario automáticamente obtiene permisos para el resto de las bases. También hay ciertos comandos que solo pueden realizarse desde esta base, como listar todas las bases existentes o dar de baja el servicio.
- local
  - Esta base nunca se replica (en contextos “sharding”) y se puede utilizar para guardar colecciones que deberían ser locales al servidor.
- config
  - Cuando se usa MongoDB configurado para “sharding”, esta base de datos se utiliza internamente para mantener datos acerca de los nodos. [25]

4.6 AngularJS

AngularJS es un framework estructural escrito en JavaScript para crear aplicaciones web dinámicas. Permite utilizar los mismos documentos HTML como templates para las vistas de la aplicación y además, el framework maneja el data binding del modelo de datos con la vista, y también provee inyección de dependencias entre componentes. [26]

De alguna manera AngularJS puede pensarse como la versión del modelo estructural MVC para la web. Las vistas son páginas HTML, con la sintaxis extendida que provee AngularJS para hacer el data binding. El modelo son los datos de la aplicación y los controladores son objetos JavaScript que pueden ser asociados a las vistas y son los que responden a las acciones del usuario y manipulan los datos.

Lo interesante de utilizar un framework como AngularJS en vez de simple JavaScript es que este último no provee, de forma nativa, ninguna manera de organizar la aplicación (más allá de separar distintos scripts en distintos archivos ".js"). Si quisiéramos mantener algún orden necesitaríamos hacerlo nosotros mismos y de alguna manera, los distintos frameworks como AngularJS, Ember, Backbone, etc, proveen justamente eso.

AngularJS en particular, es un framework estable que se especializa en el modelo MVC y nos permite organizarnos adecuadamente, separando las distintas capas en objetos con distinta semántica (controladores, modelo, vistas), además de que permite definir otro tipo de artefactos útiles como servicios (usualmente usados para obtener datos de manera remota), directivas (permiten extender la sintaxis de HTML para agregar nuevos componentes lógicos), etc.

AngularJS permite separar distintas partes de nuestra aplicación en módulos, lo que permitiría construir aplicaciones basadas en N módulos que pueden agregarse según la necesidad propia de cada sistema.

Por último, AngularJS internamente gestiona los eventos mediante un servicio $digest que, básicamente, procesa los cambios en la vista y/o modelo para llevar a cabo eficientemente el data-binding en dos direcciones, lo cual es compatible con la manera en la que el navegador maneja su propio ciclo de eventos, como se ilustra en la Figura 11.

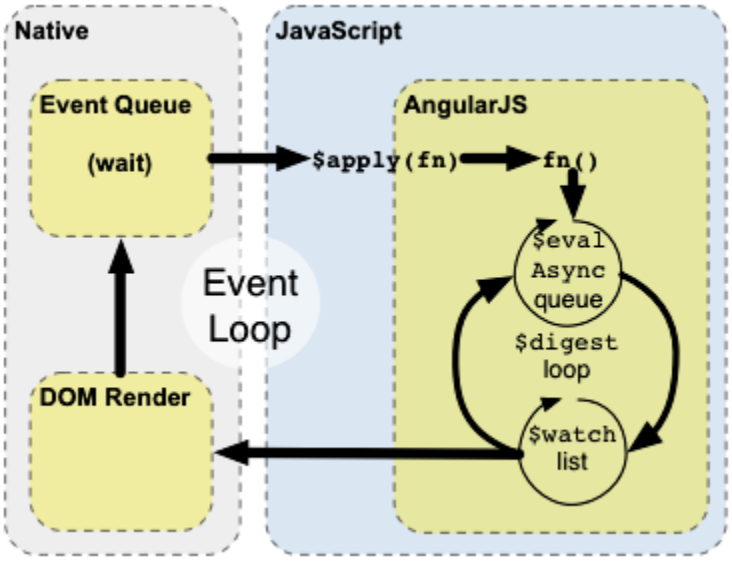

Figura 11 – Integración con el ciclo de eventos del navegador

Data Binding

En los frameworks web tradicionales, los controladores combinan los datos del modelo con los templates definidos, para crear una vista que luego se le retornará al usuario. De esta manera, la vista refleja solamente los datos que el modelo expone al momento renderizar la vista.

AngularJS hace esto de una manera diferente. En lugar de fusionar el modelo y el template de la vista para luego reemplazar un elemento de la página

HTML (elemento DOM), AngularJS crea templates en tiempo de ejecución para la vista. Es decir, las vistas son templates que son interpolados dinámicamente en vivo. Esta característica es una de las más importantes en el framework, ya que evita al desarrollador el trabajo de resolver como se van a mostrar los datos en la vista.

Este aspecto es el que permite la real separación del modelo MVC, en donde el controlador solo se encarga de manipular los datos del modelo, y el binding automático entre el modelo y la vista se hace cargo de la presentación de los datos. En la Figura 12 se muestra una representación del binding bidireccional de AngularJS.

Sin adentrarnos demasiado en los detalles del framework, lo que hace AngularJS es recordar los valores que tiene el modelo en todo momento. Cuándo AngularJS cree que los valores del modelo puedan haber cambiado, ejecuta un proceso que intenta encontrar los modelos “dirty” (en inglés, sucios) para detectar potenciales cambios en el modelo.

Esta modalidad es llamada dirty checking (chequeo de “suciedad”). Este proceso es relativamente eficiente. Cada vez que pueda haber un potencial cambio en el modelo, AngularJS ejecutará el dirty checking en su ciclo de eventos para mantener consistencia entre el modelo y la vista.

En AngularJS el data binding se define en la vista (en los templates HTML), simplemente utilizando la notación {{ name }} en donde name es el nombre de una de las variables del modelo. El modelo es simplemente un objeto mantenido por angular, llamado $scope, cuya explicación se dará más adelante.

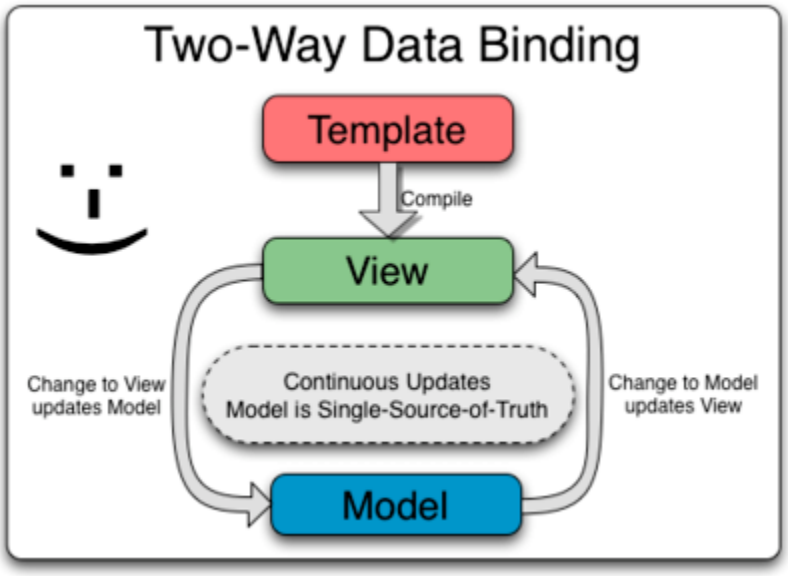

Figura 12 – Data-Binding de AngularJS en dos direcciones.

Módulos

Cuándo programamos en JavaScript, uno puede alojar toda la funcionalidad del código en el namespace global (o sea, en el scope del documento). En muchos casos, esta metodología no es conveniente, ya que puede causar colisiones de nombres que son difíciles de detectar y por ende provocar pérdida de tiempo en el desarrollo.

AngularJS permite encapsular nuestra lógica en componentes testeables y reusables utilizando módulos.

Un módulo es una unidad funcional de código que puede reutilizarse en distintas aplicaciones.

Además, la aplicación main de AngularJS, es a su vez, un módulo, el cual podemos hacer que dependa de otros módulos que encapsulen el código de

nuestra aplicación.

El uso de módulos nos provee varias ventajas:

- Mantener limpio el namespace global
- Facilitar la escritura de tests ya que podemos testear módulos individuales con funcionalidades aisladas.
- Facilitar la reutilización de código en distintas aplicaciones.
- Permitir que nuestra aplicación cargue los distintos módulos que la contienen en cualquier orden a medida que los necesitamos.

La API de Angular permite crear módulos simplemente con un método al cual debemos indicarle el nombre que tendrá el módulo, y los otros módulos de los cuales depende.

Scopes

El scope es un elemento fundamental de cualquier aplicación Angular. Se usan prácticamente en todo el framework y por lo tanto es importante comprender correctamente su uso y funcionamiento.

Los scopes en angular se refieren al modelo de la aplicación (la M de MVC). O sea, los scopes funcionan como el contexto de ejecución de las expresiones que escribimos en las vistas. También, el objeto $scope es donde definimos la lógica de nuestra aplicación (los métodos que manipulan los datos).

Sencillamente, funcionan como el pegamento entre el modelo, los controladores y las vistas. Esto significa que si la vista cambia un elemento del scope, este se verá actualizado en el modelo, cuando lo consultemos en un controlador. Y si un controlador modifica el estado de alguna variable del scope, la vista también se actualizará automáticamente.

Root Scope

En AngularJS los scopes se orden de manera jerárquica, similar a como lo hacen los contextos de ejecución en JavaScript. Cuándo el framework comienza a correr y crea el binding para el objeto ng-app (objeto principal de la aplicación Angular), este crea también el primer scope, llamado $rootScope. Este es el padre eventual de todos los demás scopes que luego se creen. Este es lo más cercano a tener un contexto global en Angular, y así como no se aconseja utilizar el namespace global de JavaScript para alojar nuestra lógica; tampoco es conveniente utilizar excesivamente el $rootScope para nuestra funcionalidad.

En AngularJS, cada controlador tendrá su propio scope que será hijo del $rootScope.

El objeto $scope es simplemente un objeto JavaScript, por lo tanto podemos manipularlo con total libertad. Todas las propiedades del objeto $scope estarán accesibles en los templates de las vistas.

Funciones básicas de los scopes:

- Proveer observers para los cambios del modelo
- Proveer la posibilidad de propagar cambios de modelo en la aplicación
- Pueden ser anidados para aislar funcionalidad y propiedades de modelo
- Proveer el contexto de ejecución en donde se evalúan las expresiones de la vista

En general, un scope se crea cuando se inicia una aplicación Angular, cuando se crea un nuevo controlador o cuando se instancia una directiva (otro

componente importante de Angular que se detallará más adelante).

Controllers

Los controllers (controladores) en AngularJS nos permiten “expandir” la funcionalidad básica de un $scope en una vista. En esencia no es más que una función JavaScript que recibe el $scope de la vista, y nos permite manipularlo para darle al modelo su estado inicial, así como setear funciones adicionales al scope, que pueden ser invocadas desde la vista asociada.

Lo único que debe hacer el desarrollador, es crear la función que oficia de constructor para el controlador. Esta función es la que recibe el $scope como parámetro y en la ejecución de la misma es en donde establecemos toda la lógica para inicializar y manipular los datos.

También, las funciones declaradas en el $scope son accesibles de la misma manera que sus datos, así que fácilmente podemos hacer un binding de los eventos de clic en botones u otros elementos de la vista (usando la directiva ng-click) a funciones dentro del $scope.

Hay varias maneras de declarar un controlador, aunque la que se considera más apropiada es la utilizar el método controller que proveen los módulos. De esa manera, podemos crear un controlador dentro de un módulo sin “ensuciar” el contexto global de JavaScript.

Para crearlo, al método controller se le pasa por parámetro el nombre del controlador y luego la función de construcción del mismo.

También hay más de una manera de asociar una vista con un controlador, pero de nuevo, la preferida es utilizando la funcionalidad de routing que viene con angular, la cual nos permite, para un cierto path, renderizar un template HTML dentro de un div, y asociar a esta vista con un cierto controlador.

De esta manera, nos quedan separadas las responsabilidades con respecto a la representación de los datos y la manipulación de los mismos, permitiendo intercambiar vistas y controladores cambiando únicamente los parámetros de routing. La Figura 13 ejemplifica lo anteriormente detallado.

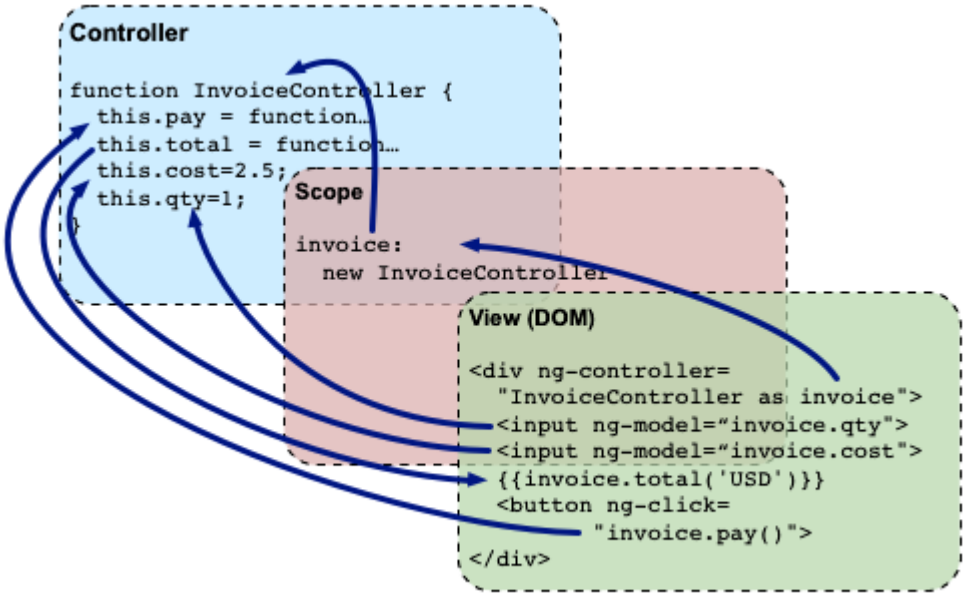

Figura 13 – Relación View-Scope-Controller

Expresiones

Las expresiones son la manera que tenemos en AngularJS de expresar el binding de datos en la vista. Usualmente se utiliza la notación {{ }}, en donde todo lo que pongamos entre las llaves dobles, será evaluado por angular dentro del correspondiente $scope.

También podemos utilizar expresiones dentro de un controlador, para, por ejemplo, establecer un listener que se evaluará cada vez que angular ejecute el

ciclo de eventos para controlar si hubo cambios en la vista o modelo, y en caso de que la expresión utilizada en el listener haya cambiado con respecto al último valor evaluado, la función de callback asociada a dicha expresión será ejecutada.

Las expresiones cumplen con las siguientes características:

- Todas las expresiones son ejecutadas dentro del contexto de un $scope y tienen acceso a todas su variables y funciones
- Las expresiones no arrojan errores de tipos o referencia
- No permiten sentencias de control (if, for, etc)
- Aceptan filtros o cadenas de filtros (los filtros es explicarán a continuación)

Filtros

Un filtro provee un medio para formatear los datos que le mostramos al usuario en la vista. En angular hay varios filtros ya predefinidos, pero también podemos crear los nuestros fácilmente. Un filtro es utilizado mediante el símbolo |, dentro de las dobles llaves en una expresión dentro de un template. Por ejemplo, la expresión {{ name | uppercase }}, obtendrá primero la propiedad name del $scope asociado, y luego ese valor será pasado por el filtro uppercase, un filtro predefinido en angular que cambia todos los caracteres de la entrada a mayúsculas.

Los filtros también pueden recibir parámetros adicionales (además de la entrada) que permiten personalizar el comportamiento del mismo. Por ejemplo, la expresión {{ 123.456789 | number:2 }} devuelve el valor 123.46, ya que el filtro number es un filtro que formatea la entrada a valor número y además puede recibir un parámetro con la precisión de los dígitos decimales que queremos

obtener.

Para crear un filtro nosotros mismos, sólo hace falta invocar al método filter de un módulo AngularJS, al cual le pasamos como parámetro el nombre del filtro y una función de construcción. Esta última, al ser invocada debe devolver la función filtro en sí, que necesita recibir al menos 1 parámetro (la entrada del filtro) y luego tantos parámetros como argumentos adicionales queramos para el mismo.

Directivas

Las directivas son otros de los elementos más importantes dentro del framework.

Una directiva permite “extender” el lenguaje HTML nativo con nuevos tags con funcionalidad provista por angular o por nosotros mismos. Es la manera que tiene angular de crear nuevos elementos HTML. Por ejemplo, la directiva ng-app, es la que permite a angular iniciar todos los elementos necesarios del framework. Dentro de ella es que angular buscar expresiones para reemplazar, otras directivas para procesar y vistas para renderizar.

El ciclo de vida de las directivas es suficientemente complejo y escapa al alcance de este trabajo, por lo tanto vamos a resumir un poco la explicación del mismo y cómo podemos hacer para crear nuestras propias directivas.

La manera más directa de entender las directivas es que son simplemente funciones que corren en el contexto de un elemento HTML. Esta función es la que dota al elemento de nueva funcionalidad, esto incluyendo también la posibilidad de transformar y manipular los elementos DOM del tag. Angular expone por defecto varias directivas que permiten hacer infinidad de cosas, como por ejemplo la directiva ng-click, que hace un binding del evento

mouseUp del elemento a una función dentro del $scope de la vista, o la directiva ng-class que permite modificar las clases CSS asociadas a un elemento según cambia el modelo en el contexto del $scope. Como es usual, para crear nuestras propias directivas, basta con llamar a la función “directive” de un módulo, pasándole como parámetro el nombre de la directiva y el constructor.

Cuándo angular está parseando el documento, busca llamadas a directivas en diversos sitios:

- Tags
- Atributos
- Comentarios
- Clases

La función que crea la directiva es ejecutada sólo la primera vez que es encontrada la llamada para cada directiva, y para cada vez que la directiva es invocada, angular ejecuta las funciones de binding definidas en el constructor de la misma, y le asocia a la directiva su propio $scope. De ahí en adelante, el ciclo de vida de la directiva toma el control.

Vistas y routing

Angular provee un módulo llamado angular-route, muy útil para manejar la navegación de nuestra aplicación. Con él, podemos definir el layout de nuestra vista (de toda la aplicación), y luego crear distintas vistas más pequeñas que se pueden cargar y descargar a medida que navegamos en la aplicación.

Para crear un template layout, lo único que debemos hacer, es, dentro de nuestro HTML que funciona de vista para la aplicación, declarar un elemento con la directiva ng-view. Esta directiva es usada por el modulo de routing y es en

donde renderizará las vistas más pequeñas que definan el template para cada ruta.

Luego, simplemente tenemos que llamar al método config del módulo que define nuestra aplicación y pasarle el objeto $routeProvider que es el que nos permite configurar las rutas.

Por último, para cada ruta que queramos definir, podemos llamar al método when del $routeProvider que, entre varias otras cosas, nos permite definir para una cierta ruta (por ejemplo, /, /home, /login, etc.) la vista asociada a dicha ruta y el controlador que manejará el $scope de la misma.

De este modo, cada vez que una ruta en angular cambia, el framework hace lo siguiente:

- Crea un nuevo scope
- Elimina la última vista, lo cual limpia el último scope
- Enlaza el nuevo scope con el nuevo template
- Enlaza el controlador asociado (o uno nuevo, implícito) al nuevo scope
- Emite un evento que indica que la vista fue cargada y enlazada
- Ejecuta la función del atributo onLoad de la vista, si es que existe

Este módulo también nos provee la posibilidad de manipular la navegación, utilizando el servicio $location, el cual tiene métodos que permiten saber el path actual en el cual se encuentra la aplicación y también cambiar de path, disparando los eventos asociados al cambio que ya hemos detallado.

Inyección de dependencias

En general hay tres maneras de que un objeto obtenga punteros a los

otros objetos de los cuales depende:

- Crearlos internamente en el objeto dependiente
- Referenciarlos asumiendo que están en el contexto global (o sea, que sea una variable global)
- Pasarle los objetos al dependiente en el momento en que los necesitaba

Con el patrón de inyección de dependencias, estamos usando la última de estas tres opciones. Las otras dos presentan varias dificultades, como lo que ya discutimos con respecto a ensuciar el scope global y hacer casi imposible el desacoplamiento de los componentes. Este patrón, elimina la mala práctica de tener dependencias hard-codeadas, posibilitando inclusive la resolución de dependencias en tiempo de ejecución.

La idea de la inyección de dependencias es encontrar los objetos de los que otro objeto depende de manera automática, e inyectarlos en el momento en el que son necesarios.

En AngularJS el objeto $injector se encarga de esto. Éste es el responsable de la creación e instanciación de todos los objetos de la aplicación (módulos, controladores, servicios, etc).

Cuándo un módulo se inicializa, el $injector es el responsable de crearlo y de pasarle todos los objetos de los que depende (previa instanciación de cada uno de ellos).

El objeto $injector tiene varias maneras de resolver las dependencias que el objeto a instanciar necesita. Estas son:

- Por inferencia:
    - En este modo, AngularJS asume que los nombres de los parámetros de la función constructora del objeto, son los nombres de las dependencias que necesita. Por ejemplo, si la función que

crea un controlador es: `function ($scope, $rootScope, MiServicio) {}`…, el $injector intentará suplir las dependencias con los objetos llamados “$scope”, “$rootScope” y “MiServicio” respectivamente.

- Explícita:
  - Angular provee una manera de indicar explícitamente cuáles son las dependencias a inyectar. Para ello, solo tenemos que definir la propiedad $inject de la función creadora del objeto. Esta propiedad debe ser un arreglo de strings, con los nombres de los objetos a inyectar. En este caso, el orden es importante, ya que angular inyectará el objeto cuyo nombre es el primer objeto del arreglo, en el primer parámetro de la función, y así sucesivamente.
  - Iniline
    - La ultima manera es virtualmente igual a la anterior, con la única diferencia que no seteamos una variable en la función, sino que, cuando estamos creando un objeto dentro de un módulo, en vez de pasarle la función constructora, podemos pasar un arreglo en donde los primeros n-1 elementos, son las dependencias del objeto, y el n-ésimo elemento es la función constructora, que tiene n-1 parámetros.

Servicios

En AngularJS, un servicio es un objeto singleton que es creado solo una vez por aplicación y cuando es necesario. Los servicios nos permiten agrupar funcionalidad relacionada, y además, por ser singletons, compartir información entre distintos controladores manteniendo el desacoplamiento de los mismos.

Angular ya provee varios servicios que son de gran utilidad como $http para manejar pedidos AJAX entre otros, que nos facilitan también la creación de nuevos servicios.

El framework nos hace muy fácil la tarea de crear un nuevo servicio: simplemente basta con registrarlo con un nombre y una función de construcción. Una vez registrado, el compilador de angular puede referenciarlo por nombre y cargarlo como dependencia en los objetos que lo requieran.

Para registrar un servicio hay varias maneras, pero la más común y flexible es usando el método “factory” de un módulo angular. Este método recibe un string como nombre del servicio y una función (o un arreglo cuyo último elemento es una función) y debe devolver el objeto singleton que implementa el servicio.

Para utilizar un servicio, sólo hace falta registrarlo como dependencia de otro objeto angular (un controlador, una directiva, un filtro, otro servicio, etc.) por nombre. El framework se hará cargo de instanciarlo una vez y pasar dicha instancia como dependencia. [27]

4.7 Reflexivo del análisis de tecnologías

Luego de haber analizado en detalle las distintas tecnologías descriptas, optamos por el uso de una arquitectura full-stack JavaScript para el desarrollo de distintos tipos de aplicaciones. El uso en conjunto de NodeJS, AngularJS y MongoDB provee varias ventajas a la hora de desarrollar una aplicación. En principio, se optimiza la tarea de implementación ya que se utiliza un único lenguaje (JavaScript) para desarrollar tanto el frontend como el backend, y también para gestionar la base de datos. Además, esta característica permite que la integración de estos componentes sea mas eficiente ya que se acoplan

de manera natural por tener todas un factor común que es el lenguaje de programación JavaScript.

Puntualmente NodeJS nos permitió diseñar una arquitectura que se adecue a las necesidades que requiere nuestra propuesta de solución, cumpliendo con el objetivo de que la respuesta del sistema sea en tiempo real.

CAPÍTULO 5 – Arquitectura Propuesta

5.1 Propuesta de desarrollo: rParking

Se propone como solución desarrollar un sistema integral que permita a los actores interesados tener información en tiempo real sobre el estado de las plazas de estacionamiento reservadas y proveer un mecanismo para dar aviso a quién corresponda en caso de detectar una ocupación no autorizada, para que se actúe en consecuencia.

Para implementar esta solución se deben instalar sensores de ocupación en las plazas reservadas para detectar el ingreso de un vehículo y lectores RFID para su identificación y autenticación. Cuando un vehículo estaciona en una de las plazas se activaría el sensor de estacionamiento, indicando que la plaza fue ocupada. Posteriormente, el sistema determinará si el vehículo estacionado es el autorizado, mediante una señal recibida por un lector RFID. Si esta autenticación falla, el sistema registraría un caso de estacionamiento no permitido en dicha plaza, e informaría a un supervisor para que tome la medida necesaria. Tanto el supervisor como los conductores autorizados podrían consultar la información del estado de las plazas.

Los conductores autorizados deberían poseer un tag RFID con el cual podrán ser autenticados en el sistema como conductores acreditados para estacionar en las plazas reservadas.

A su vez, algunas de las plazas en el sistema podrían configurarse de manera tal que no requieran autenticación mediante tags RFID, permitiendo que cualquier conductor pueda ocuparlas sin necesidad de estar registrado.

También se podrían configurar ciertas plazas para que pueda ser utilizadas por ciertos conductores en particular, con el objeto de reservar dichas plazas sólo

a aquellos que tienen permitido utilizarlas.

Esta reserva podría cancelarse en caso de que el usuario de la plaza notifique que la misma no será ocupada, para lograr un uso óptimo del espacio.

Con el objetivo de probar los desarrollos propuestos, se hará una prueba piloto de las aplicaciones, en el estacionamiento perteneciente a Presidencia de la Universidad de La Plata, ubicado en la calle 47 e/ 6 y 7.

5.2 Arquitectura definida

La arquitectura que proponemos está compuesta por diversos componentes de hardware y software:

- Sensores de ocupación que serán instalados en las plazas reservadas.
- Lectores y tags RFID para la detección y autenticación de usuarios registrados.
- Un módulo de software que constituye el back-end de la aplicación.
- Aplicaciones clientes (web y mobile) que se comunican con el back-end mediante una API de servicios.

Se usarán tecnologías que aseguren interoperabilidad y disponibilidad a largo plazo:

- Se separará el módulo de comunicación con los sensores de estacionamiento, del core (núcleo) del sistema principal, con objeto de desacoplar ambas partes y permitir una rápida adaptación a distintos tipos de sensores y/o estándares.
- De la misma manera se trabajará con los lectores y tags RFID.
- Por otro lado, los componentes de la aplicación periféricos al núcleo se

desarrollarán como componentes intercambiables e interoperables entre sí, con baja cohesión y utilizando tecnologías y herramientas de software libre.

Con respecto a las herramientas de software, uno de los puntos más novedosos es la utilización de Node.js, una tecnología para el back-end pensada para aplicaciones en tiempo real; y MongoDB, una tecnología de base de datos NoSQL que permite acceso de tiempo constante a los datos independientemente del volumen de los mismos.

A su vez, para el front-end de la aplicación, se desarrollarán dos clientes distintos que accedan a la misma API (una API de servicios REST):

- Aplicación Web: Será desarrollada con algún framework JavaScript que permita escribir aplicaciones web ricas.
- Aplicación mobile: Se construirá una aplicación mobile desarrollada con el SDK de alguna de las plataformas para teléfonos inteligentes disponibles.

Por último, se intentará que el hardware utilizado en la ciudad y los servicios de las aplicaciones inteligentes implementados evolucionen lo más independientemente posible, para poder lograr un fácil crecimiento tanto en la infraestructura de la ciudad, como en los servicios prestados.

Con la integración de estos componentes se lograría una arquitectura óptima, representada en la Figura 14, que involucra todas las partes necesarias (actores, objetos, acciones, eventos) para brindar una solución al problema de estacionamiento en plazas reservadas. A su vez, la arquitectura propuesta es lo suficientemente flexible como para poder adaptarse a problemáticas similares, tales como brindar información de plazas de estacionamiento libre. Además, la solución se adaptaría al cambio y/o agregado de sensores de distintos fabricantes ya que solo bastaría con resolver la comunicación entre los dispositivos y nuestra implementación del módulo de sensores.

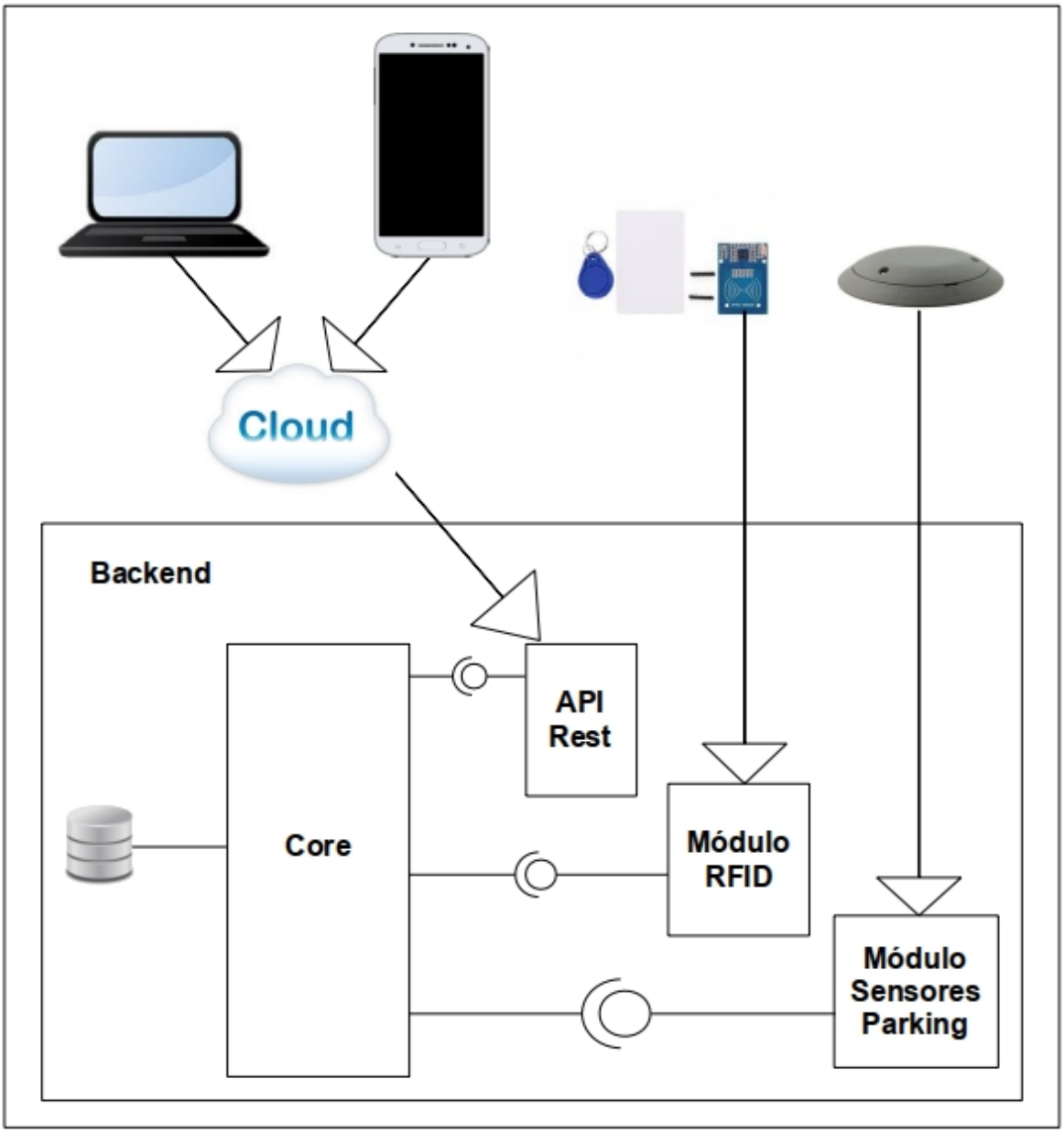

Figura 14 . Diagrama de la arquitectura de la solución

5.3 Funcionalidades provistas

A partir del análisis funcional se hizo un relevamiento de las funcionalidades necesarias con las cuales debería contar el sistema “rParking” para llegar a la solución deseada.

Las funcionalidades que se implementaron en el front-end web de la aplicación fueron las siguientes:

- Módulos de alta, baja y modificación de:
  - Usuarios
  - Plazas de estacionamientos
  - Sensores de ocupación
  - Lectores RFID
  - Tags RFID
  - Vehículos
- Módulo de configuraciones generales del sistemas
- Módulo que permite inicializar el modelo de sensores de ocupación y plazas de estacionamiento en función de los datos provistos por Urbiotica.
- Módulo tipo dashboard que permite visualizar en tiempo real el estado de ocupación de las plazas de estacionamiento y a su vez permite a los usuarios autorizados desestimar alertas de ocupación inválidas en caso de que sea necesario.
- Módulo que permite visualizar las alertas de ocupación detallando fecha, hora y código de plaza de estacionamiento.

Las funcionalidades que se implementaron en el front-end mobile de la

aplicación fueron las siguientes:

- Pantalla para visualizar el estado de ocupación de las plazas de estacionamiento en tiempo real.
- Pantalla para visualizar las alertas de ocupación, detallando fecha, hora y código de la plaza de estacionamiento correspondiente.

Las distintas configuraciones disponibles en el sistema permiten adecuarse a diferentes escenarios que pueden darse en función de distintas variantes que se presentan en los casos reales; por ejemplo, disponibilidad de hardware (con cuántos lectores RFID se cuenta, exclusividad de ciertas plazas, liberación de ciertas plazas para un período arbitrario de tiempo, etc.).

En principio, existe la posibilidad de configurar el sistema según se tenga un lector RFID por plaza, o solamente uno para todas las plazas del lugar. Para este último caso, es lógicamente imposible determinar exactamente que plaza se está autenticando cuando se consume un evento de lectura de tag de RFID. Para mitigar este problema, se utiliza algún criterio para poder decidir que plaza es la que se está autenticando. Otra manera de reducir el impacto de este inconveniente es controlando la cantidad de autos estacionando al mismo tiempo.

En el caso de tener un lector por plaza, el evento de lectura de un tag es automáticamente asociado al lector que lo procesa y este último está vinculado a una plaza en particular, por lo cual, podemos determinar con exactitud que se está intentando autenticar en una plaza específica.

A su vez, cada plaza podrá ser configurada con un conjunto arbitrario de tags habilitados, pudiendo configurar cero (la plaza puede ser ocupada por cualquier usuario que disponga de un tag), uno o más (sólo podrán ocupar esa plaza correctamente aquellas personas que dispongan los tags correspondientes).

También, para sea más eficiente el uso de las plazas en aquellos casos en donde personas con exclusividad no hagan uso de las plazas que tienen asociadas (por ejemplo, licencias por enfermedad, vacaciones, etc.), la restricción de uso exclusivo a un grupo de tags, puede ser invalidada para un período de tiempo determinado, y como consecuencia esa plaza puede ser ocupada por cualquier conductor que posea un tag habilitado.

Por último, existe la posibilidad de configurar una plaza para que la misma no requiera de autenticación por RFID. En es caso, todas las ocupaciones de dicha plaza serán reconocidas como correctas.

5.4 Pantallas del sistema

A continuación se muestran las diferentes pantallas de los front-end web y mobile de la aplicación, y se detalla la utilidad de cada una.

Front-end Web

Login: Permite ingresar nombre de usuario y contraseña para iniciar sesión en la aplicación.

Figura 15 - Login

Dashboard: Permite visualizar el estado de las diferentes plazas de estacionamiento en tiempo real. También brinda la opción de autenticar un estacionamiento de forma manual, desestimando el alerta correspondiente.

A su vez, permite simular la lectura de un tag RFID para autenticar la ocupación de una plaza.

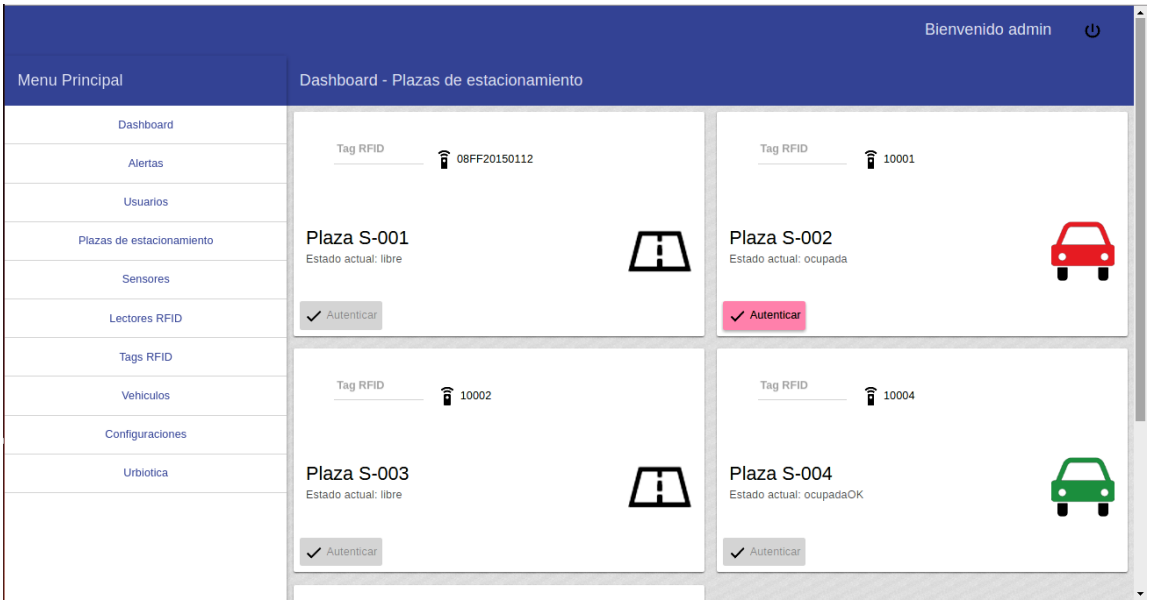

Figura 16 - Dashboard estado plazas de estacionamiento

Alertas: Muestra un listado de las alertas ordenadas cronológicamente, desde las reciente hasta la mas antigua, indicando la fecha, hora y plaza de estacionamiento correspondiente.

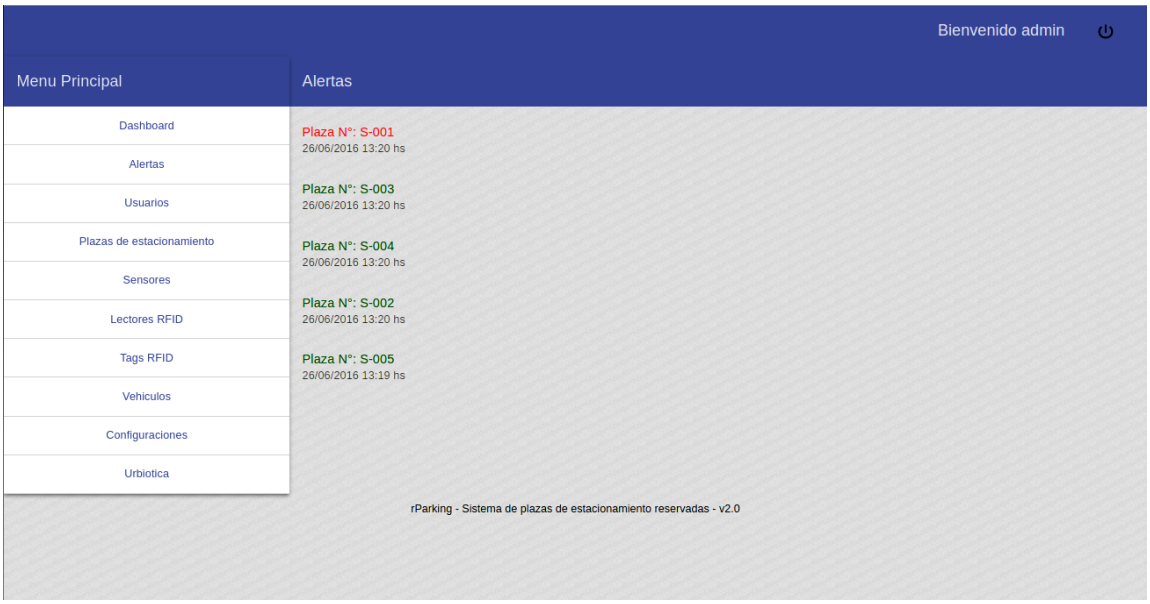

Figura 17 - Listado de Alertas

Configuraciones globales: Permite configurar los aspectos generales de

la aplicación.

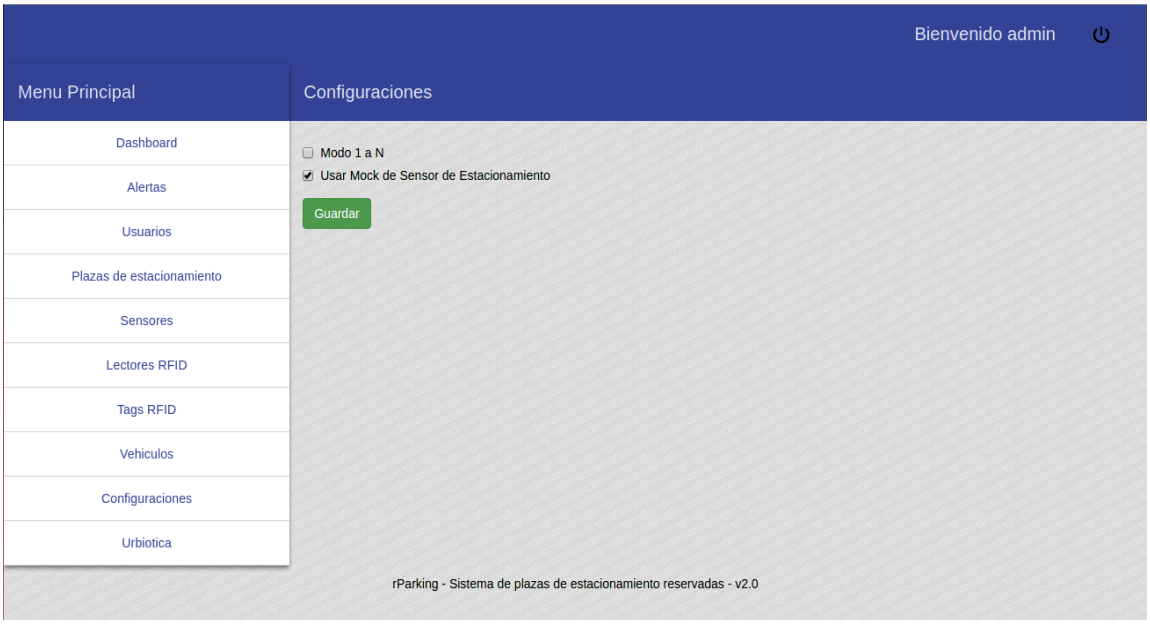

Figura 18 - Configuraciones globales del sistema

Setup Urbiotica: Esta funcionalidad esta pensada para usar como startup

de la aplicación ya que se conecta con los servicios de Urbiotica y obtiene los datos necesarios para inicializar el modelo de la aplicación.

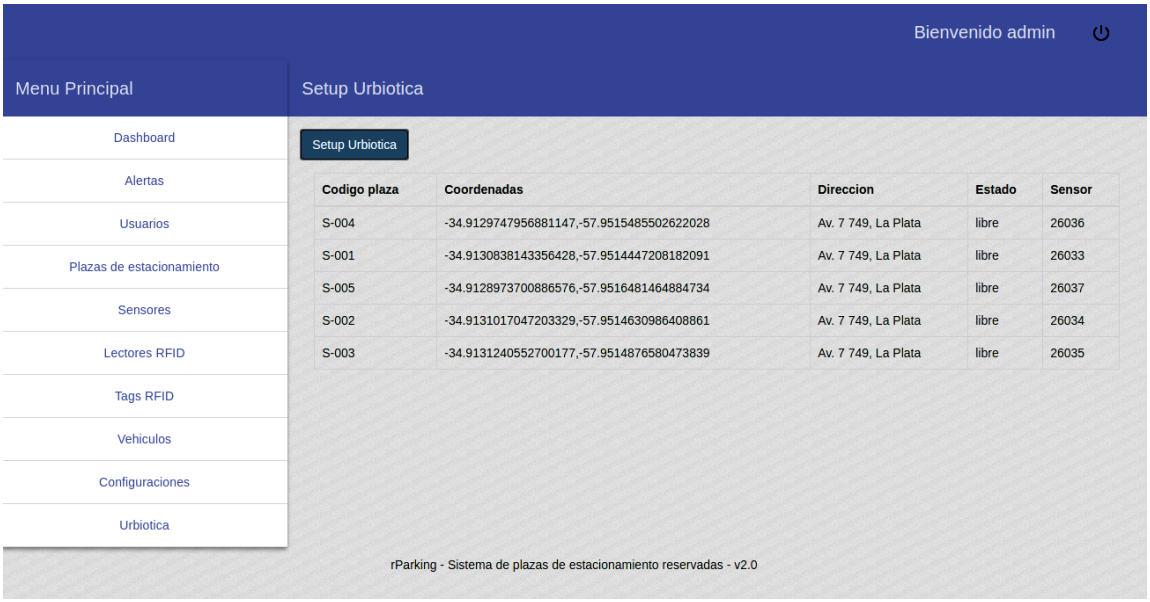

Figura 19 - Configuración inicial Urbiotica

ABM: Estas pantallas permiten dar de alta, modificar y eliminar las entidades correspondientes. También muestran un listado de las mismas.

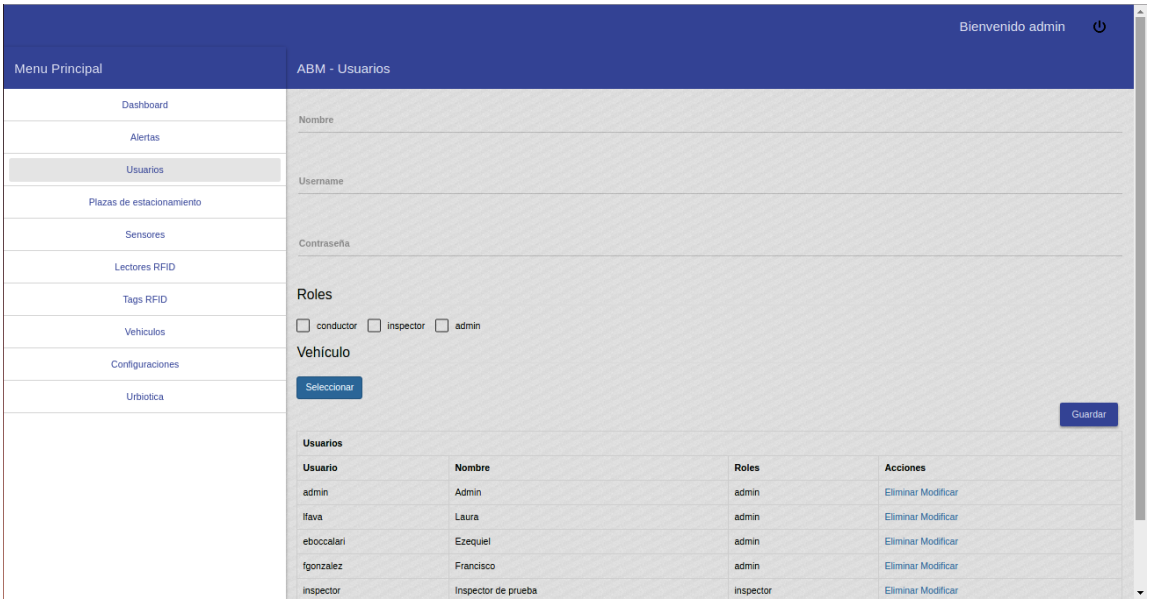

Figura 20 - ABM Usuarios

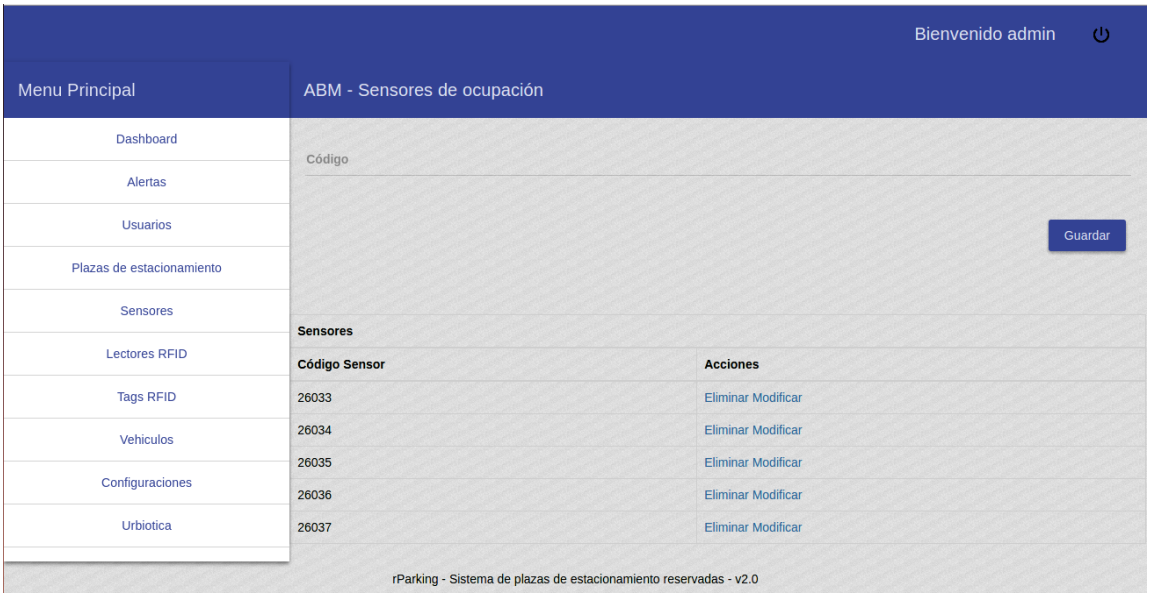

Figura 21 - ABM Sensores de ocupación

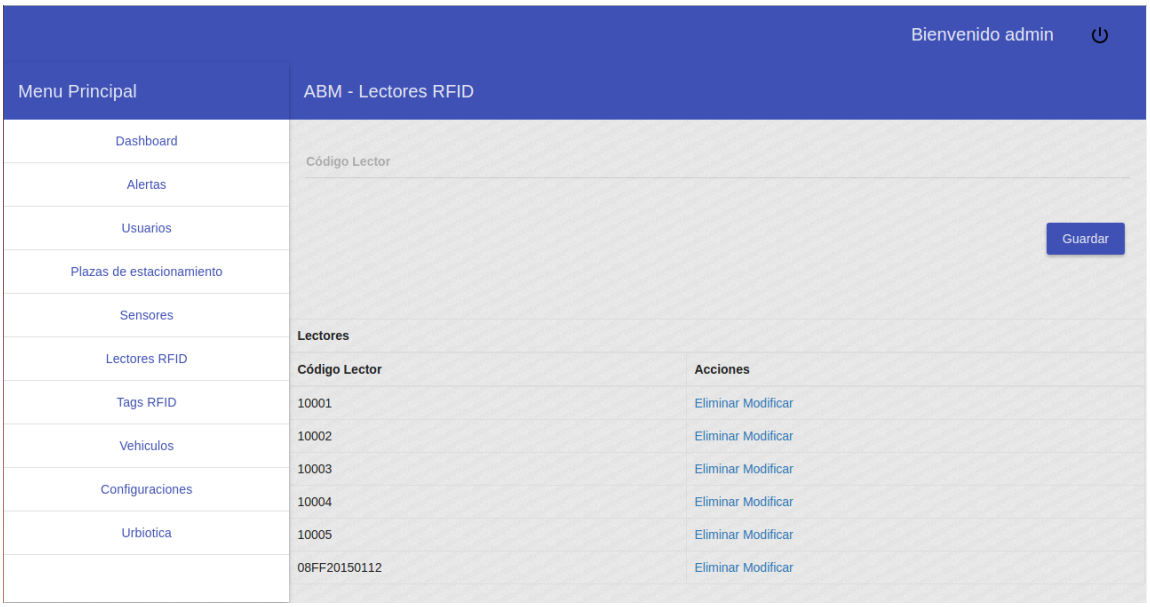

Figura 22 - ABM Lectores RFID

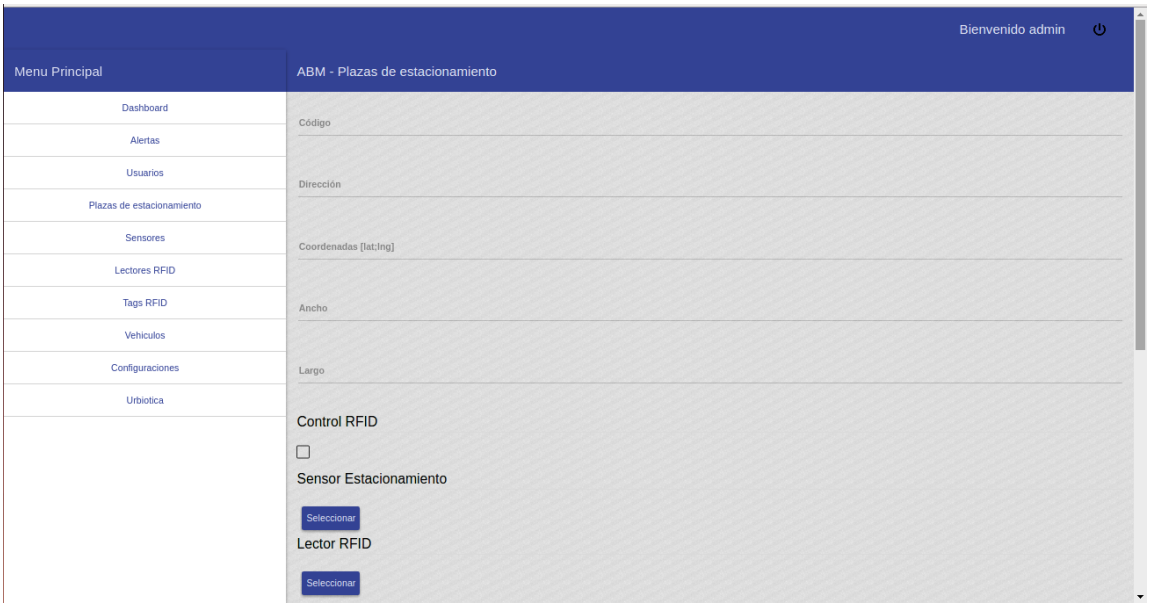

Figura 23 (a) - ABM Plazas de estacionamiento

Figura 23 (b) - ABM Plazas de estacionamiento (cont.)

Figura 24 - ABM Tags RFID

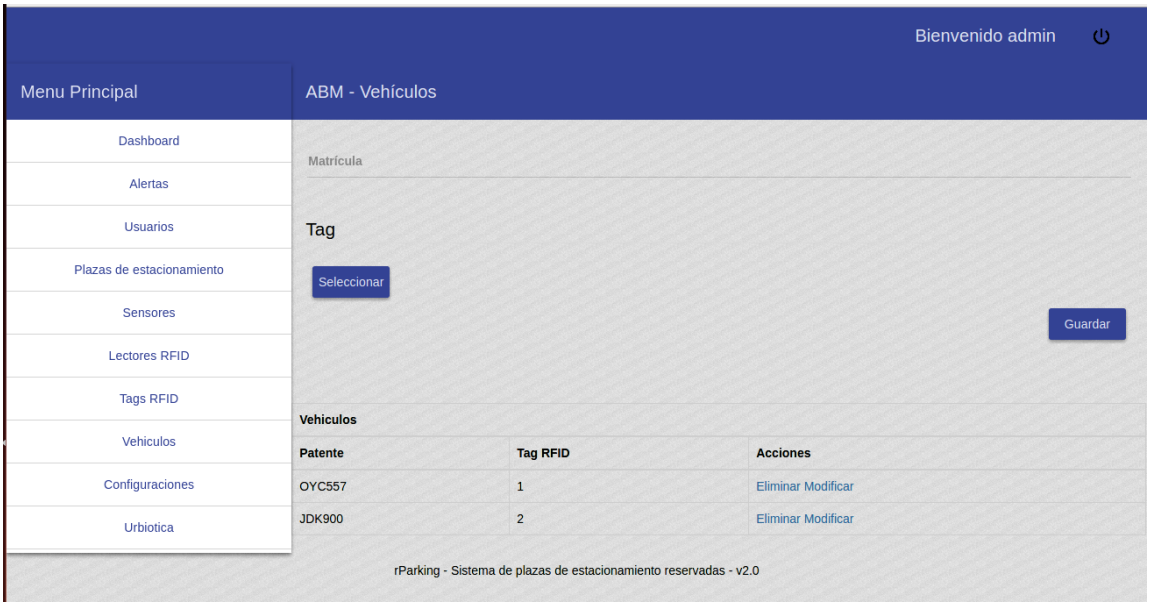

Figura 25 - ABM Vehículos

Front-end Mobile

Login: Permite ingresar nombre de usuario y contraseña para iniciar sesión en la aplicación.

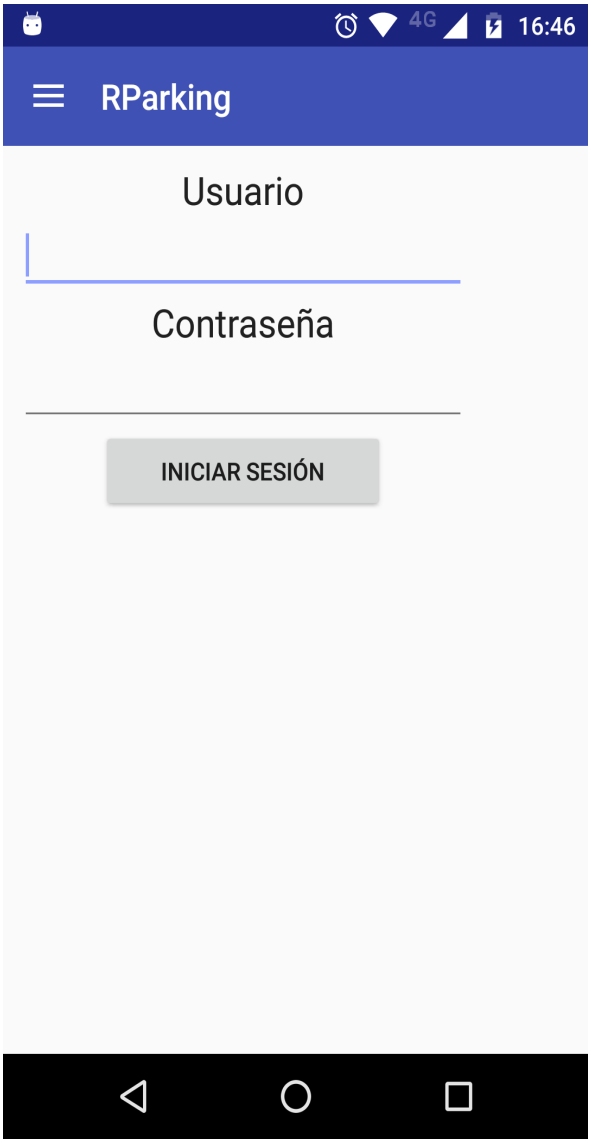

Figura 26 - Login - Aplicación Android.

Dashboard: Permite visualizar el estado de las diferentes plazas de estacionamiento en tiempo real.

Figura 27 - Dashboard plazas de estacionamiento - Aplicación Android.

Alertas: Muestra un listado de las alertas ordenadas cronológicamente, desde las reciente hasta la mas antigua, indicando la fecha, hora y plaza de estacionamiento correspondiente. También permite desestimar una alerta de estacionamiento incorrecto y de esa manera, autenticarlo como correcto.

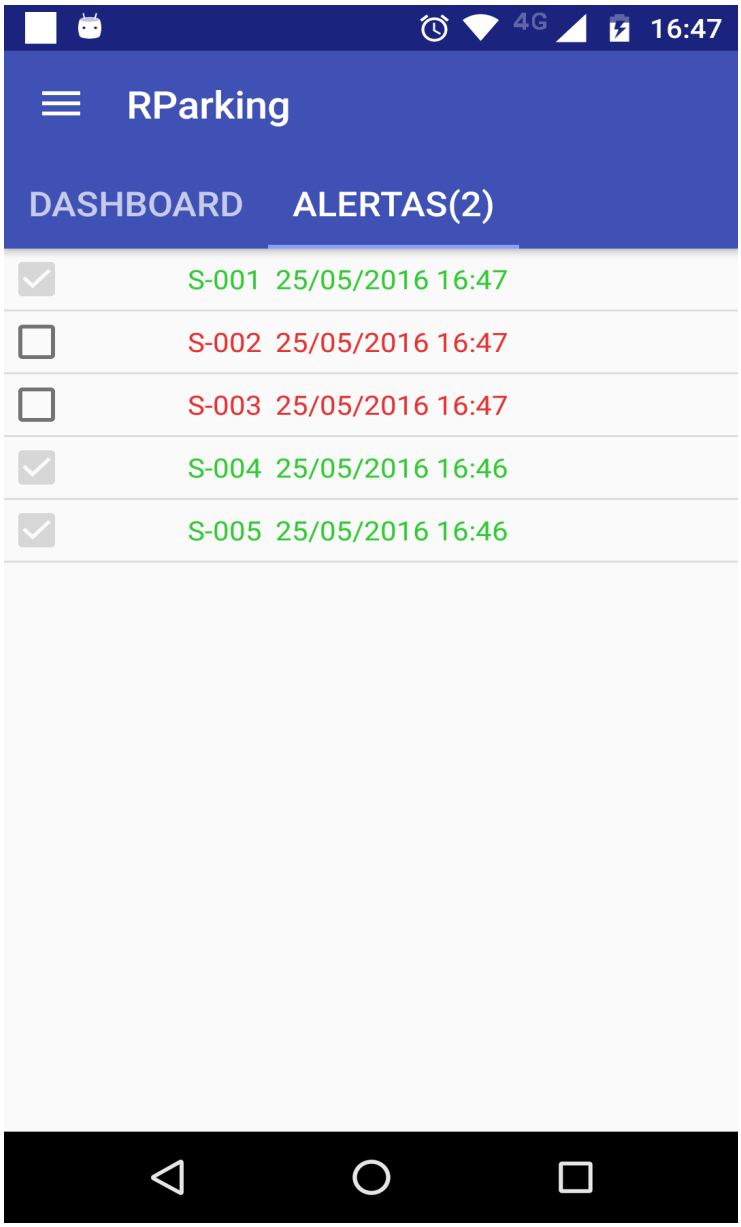

Figura 28 - Listado de Alertas en la aplicación Android

Menú lateral: Este menú permite acceder a las funcionalidades generales de la aplicación, así como también finalizar la sesión iniciada.

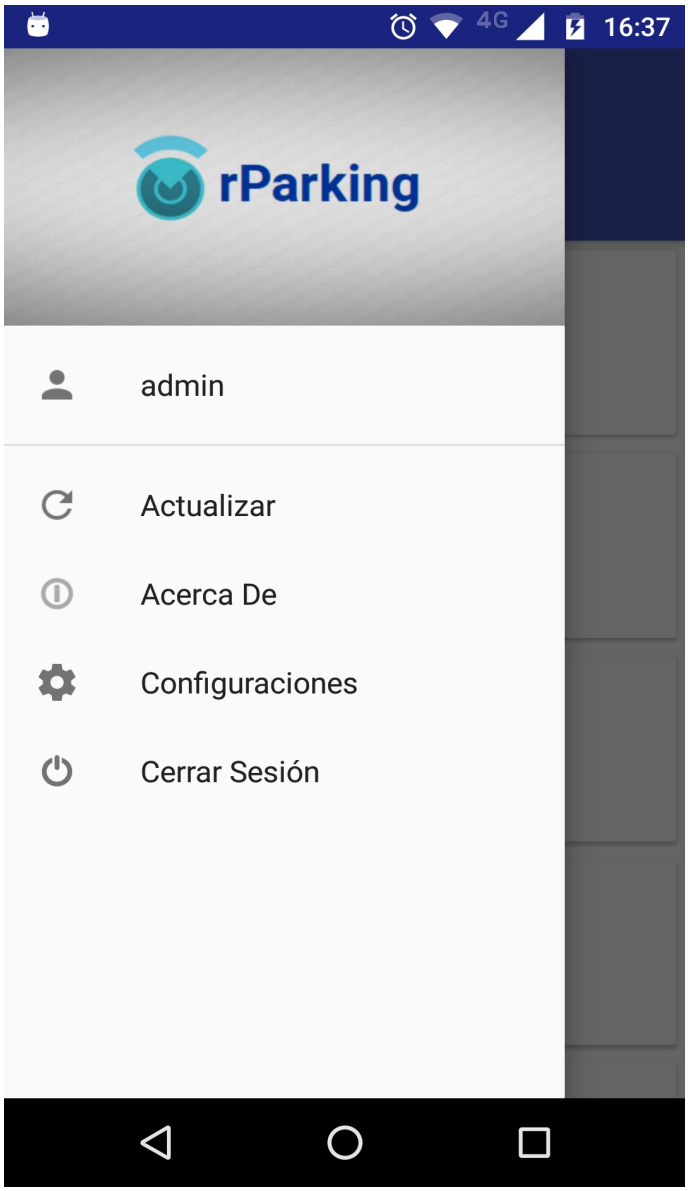

Figura 29 - Menú Lateral – Aplicación Android

Configuraciones aplicación: Muestra la/s posibles configuraciones que permite la aplicación.

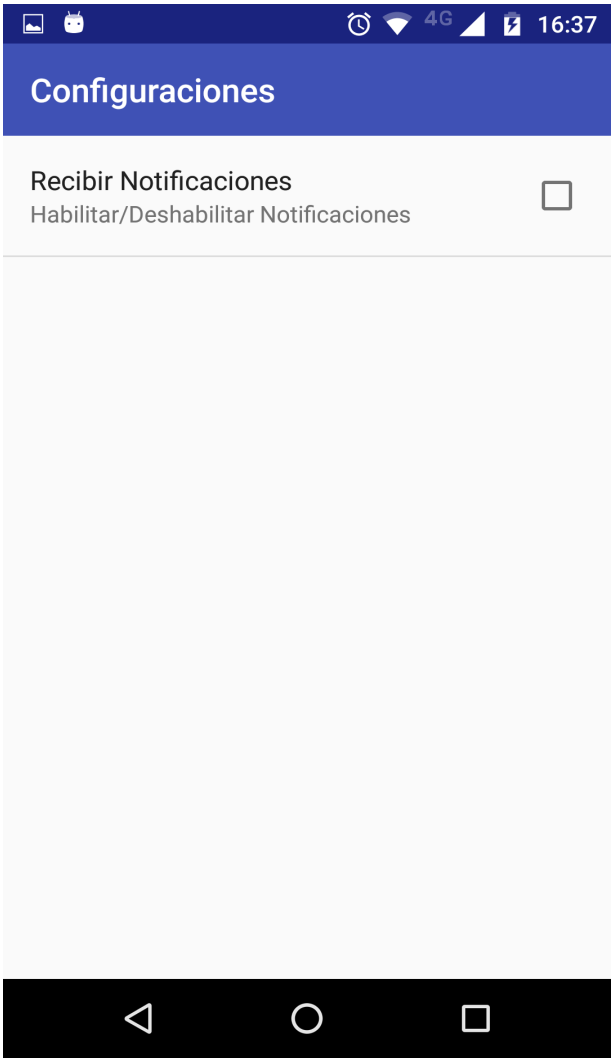

Figura 30 - Configuraciones - Aplicación Android

Acerca De: Muestra información específica de la aplicación.

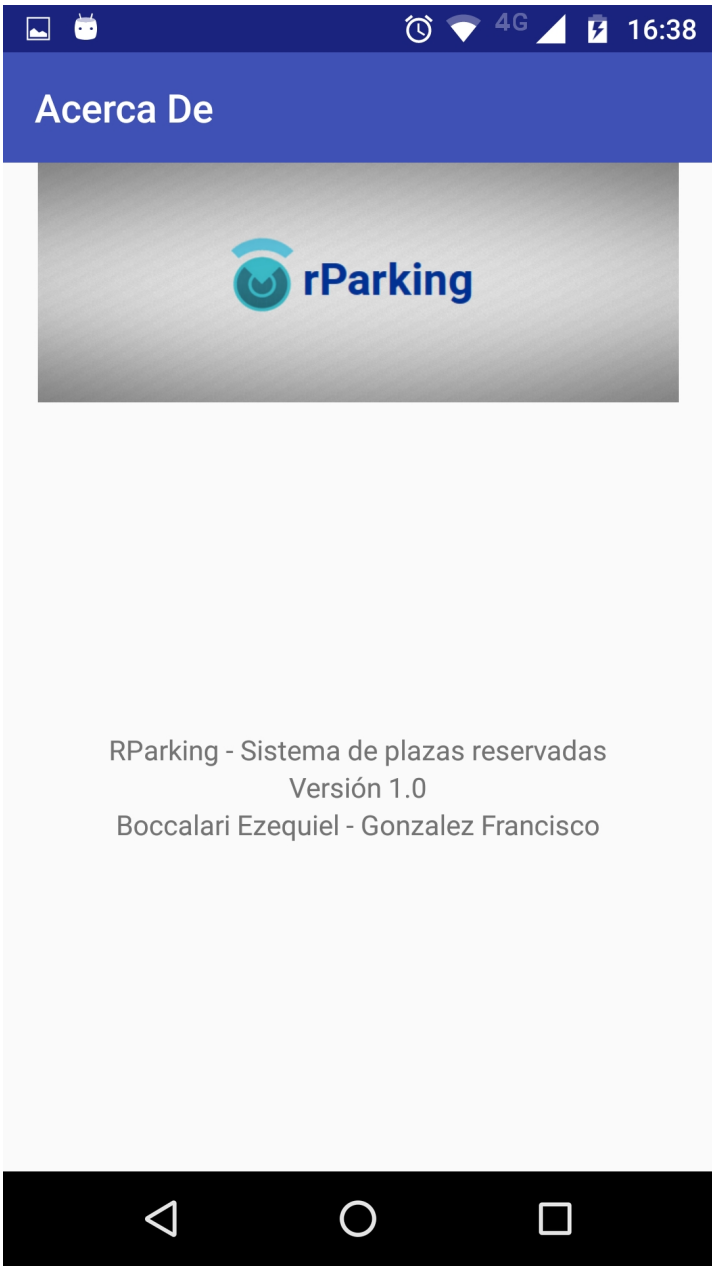

Figura 31 - Acerca de – Aplicación Android.

CAPÍTULO 6 – Pruebas y Resultados

6.1 Resultados Esperados

Se espera con este desarrollo poder optimizar y asegurar el uso adecuado de las plazas de estacionamiento reservadas. En consecuencia, deberá lograrse un uso más eficiente y óptimo de las plazas.

Por otro lado se pretende que este sistema otorgue un valor agregado a la mecánica ya existente con respecto a las plazas de estacionamiento reservadas integrando todas las partes asociadas. Esta integración permitiría la rápida y automática comunicación de la información entre las partes y les brindará a los conductores interesados una visualización, en tiempo real, acerca del estado de la disponibilidad de las plazas.

6.2 Prueba de campo del sistema rParking

Para poder llevar a cabo las pruebas de campo fue necesario diferenciarlas en dos categorías. Por un lado comprobaciones técnicas, para evaluar la performance y conectividad, y por otro lado comprobaciones funcionales para evaluar los distintos casos de uso y/o funcionalidades que provee el sistema.

Las pruebas se realizaron en dos etapas, evaluándose las mismas características en ambas pero con diferente estado de los sensores. De manera mas especifica, uno de los sensores en la primer iteración de pruebas no funcionaba de manera correcta, por tal motivo fue necesario llevar a cabo una segunda iteración.

Las siguientes son imágenes de referencia de la instalación física en el estacionamiento del rectorado de la UNLP de los sensores de ocupación (Figura 32) y el componente U-Flag (Figura 33) de Urbiotica.

Figura 32 – De izquierda a derecha y de arriba hacia abajo, imágenes de los sensores de ocupación S-001, S-002, S-003, S-004 y S-005

Figura 33 – Componente U-FLAG

6.3 Resultados obtenidos

Los resultados obtenidos en ambas iteraciones de pruebas son los siguientes:

Pruebas - iteración número 1

Pruebas técnicas			
Nombre	**Descripción**	**Resultado**	**Observaciones**
Tiempo de respuesta de ocupación de una plaza.	Evaluación de los tiempos de respuestas en la ocupación de una plaza de estacionamiento.	Muy bueno	*Promedio de 1 segundo. Esta medida fue tomada observando el tiempo que transcurría desde el momento en que el auto se posaba arriba del sensor y el sistema reflejaba ese cambio.
Tiempo de respuesta de autenticación de una plaza.	Evaluación de los tiempos de respuesta en la autenticación de una plaza de estacionamiento usando un lector RFID para la lectura de un tag RFID.	Excelente	
Funcionamiento de los sensores de ocupación.	Evaluación de los sensores de ocupación (U-SPOT) instalados.	Muy buena.	Excepto sensor “S-002” el cual tuvo un compartimiento errático en algunas ocasiones.
Conectividad	Evaluar respuesta del sistema utilizando 3G, 4G y Wi-Fi.	3G:Buena 4G:Muy Buena Wi-Fi:Excelente	

Tabla 1 – Evaluaciones técnicas en la primer iteración de pruebas.

A continuación presentamos algunos resultados estadísticos con respecto a la latencia de procesamiento de los eventos en el sistema durante la primera iteración de pruebas.

Latencia de consumo de mensaje de la cola AMQP (en segundos)	
Mínima	0,00191
Máxima	0,02287
Media	0,00300
Varianza	0,00001
Latencia de lecturas RFID (en segundos)	
Mínima	0,00600
Máxima	0,01400
Media	0,00850
Varianza	0,00001
Latencia de ocupación de plazas de estacionamiento (en segundos)	
Mínima	0,01700
Máxima	0,01500
Media	0,00410
Varianza	0,00001
Latencia de liberación de plazas de estacionamiento (en segundos)	
Mínima	0,00200
Máxima	0,09400
Media	0,00300
Varianza	0,00038

Tabla 2 – Latencia de procesamiento de eventos en la primer iteración de pruebas.

Pruebas funcionales		
Nombre	**Descripción**	**Resultado**
Plazas con tag exclusivos	Configurar una plaza para que solo puedan autenticarse correctamente determinados tags.	Funciona correctamente.
Plazas sin tag exclusivos	Configurar una plaza para que se pueda autenticar cualquier tag RFID dado de alta en el sistema.	Funciona correctamente.
Plazas invalidadas por fecha	Configurar una plaza para que la autenticación de la misma no quede restringida a los tags exclusivos en un período determinado. Es decir, que cualquier tag dado de alta en el sistema podría autenticar una ocupación aunque la plaza tenga configurado tags RFID exclusivos.	Funciona correctamente.
Lector RFID exclusivo	Se configura una plaza para que solamente pueda ser autenticada su ocupación con un lector RFID en particular.	Funciona correctamente.
Plazas sin control RFID	Configurar una plaza para que no controle autenticación RFID, es decir, ante una eventual ocupación la misma se marca como ocupada correctamente.	Funciona correctamente.
Plaza, lector y tag exclusivo.	Se configura una plaza para que solamente uno o varios tags exclusivos puedan autenticar la ocupación a través de un único lector RFID.	Funciona correctamente.

Tabla 3 – Evaluaciones funcionales en la primer iteración de pruebas.

Pruebas - iteración número 2

Los resultados de los testeos de la primer iteración fueron muy alentadores, con excepción del sensor S-002, donde en algunas ocasiones se observaron comportamientos erráticos. Después de analizada la instalación y efectuadas algunas calibraciones se procedió a realizar un segundo testeo. Para esto fue necesario también reemplazar el componente U-FLAG que no estaba

funcionando adecuadamente. Los resultados de esta segunda prueba se encuentran sintetizados en la Tabla 2 que figura a continuación.

Pruebas técnicas			
Nombre	**Descripción**	**Resultado**	**Observaciones**
Tiempo de respuesta de ocupación de una plaza.	Evaluación de los tiempos de respuestas en la ocupación de una plaza de estacionamiento.	Muy bueno	*Promedio de 1 segundo. Esta medida fue tomada de forma arbitraria, observando el tiempo que transcurría desde el momento en que el auto se posaba arriba del sensor y el sistema reflejaba ese cambio.
Tiempo de respuesta de autenticación de una plaza.	Evaluación de los tiempos de respuesta en la autenticación de una plaza de estacionamiento usando un lector RFID para la lectura de un tag RFID.	Excelente	
Funcionamiento de sensores de ocupación.	Evaluación de los cinco sensores de ocupación (U-SPOT) instalados.	Muy buena.	Se realizaron las mismas pruebas pero esta vez haciendo mas exhaustivas las mismas sobre el sensor S-002. El mismo funciono correctamente en todos los casos, presentando una latencia mayor en comparación al resto de los sensores pero aceptable de todas formas.
Conectividad	Evaluar respuesta del sistema utilizando 3G, 4G y Wi-Fi.	3G:Buena 4G:Muy Buena Wi-Fi:Excelente	

Tabla 4 – Evaluaciones técnicas en la segunda iteración de pruebas.

A continuación presentamos los mismos resultados estadísticos con respecto a la latencia de procesamiento de los eventos en el sistema para la segunda prueba.

Latencia de consumo de mensaje de la cola AMQP (en segundos)	
Mínima	0,001967
Máxima	0,009717
Media	0,002524
Varianza	0,000001
Latencia de lecturas RFID (en segundos)	
Mínima	0,00750
Máxima	0,01210
Media	0,00980
Varianza	0,00001
Latencia de ocupación de plazas de estacionamiento (en segundos)	
Mínima	0,00200
Máxima	0,01700
Media	0,00400
Varianza	0,00001
Latencia de liberación de plazas de estacionamiento (en segundos)	
Mínima	0,00200
Máxima	0,02500
Media	0,00300
Varianza	0,00003

Tabla 5 – Latencia de procesamiento de eventos en la segunda iteración de pruebas.

Pruebas funcionales		
Nombre	**Descripción**	**Resultado**
Plazas con tag exclusivos	Configurar una plaza para que solo puedan autenticarse correctamente determinados tags.	Funciona correctamente.
Plazas sin tag exclusivos	Configurar una plaza para que se pueda autenticar cualquier tag RFID dado de alta en el sistema.	Funciona correctamente.
Plazas invalidadas por fecha	Configurar una plaza para que la autenticación de la misma no quede restringida a los tags exclusivos en un período determinado. Es decir, que cualquier tag dado de alta en el sistema podría autenticar una ocupación aunque la plaza tenga configurado tags RFID exclusivos.	Funciona correctamente.
Lector RFID exclusivo	Se configura una plaza para que solamente pueda ser autenticada su ocupación con un lector RFID en particular.	Funciona correctamente.
Plazas sin control RFID	Configurar una plaza para que no controle autenticación RFID, es decir, ante una eventual ocupación la misma se marca como ocupada correctamente.	Funciona correctamente.
Plaza, lector y tag exclusivo.	Se configura una plaza para que solamente uno o varios tags exclusivos puedan autenticar la ocupación a través de un único lector RFID.	Funciona correctamente.

Tabla 6 – Evaluaciones funcionales en la segunda iteración de pruebas.

Finalmente, en las instancias de pruebas se mostró la aplicación rParking en sus dos modalidades, móvil y web, a los encargados de la playa de estacionamiento quienes se mostraron muy interesados en la misma y la consideraron simple de usar y beneficiosa para su trabajo. Por otro lado valoraron la velocidad del tiempo de respuesta del sistema desde el momento en el que el vehículo hace una ocupación de la plaza de estacionamiento hasta que se ve reflejado en las aplicaciones.

CAPITULO 7 – Conclusiones y Trabajos Futuros

7.1 Conclusiones

A lo largo de este informe de tesina se han analizado los problemas relacionados a la disponibilidad de plazas de estacionamiento en general, acotándonos finalmente a la gestión de plazas reservadas. Debido al tipo de problema al que nos enfrentamos, fue necesario realizar una investigación en profundidad de varias tecnologías, tanto de software como de hardware, para resolverlo de una manera eficiente y eficaz.

La investigación implicó que nos interiorizáramos, por un lado, en la tecnología de sensores de ocupación, y por otro, en tecnologías modernas de desarrollo de software como son NodeJS, MongoDB y AngularJS. A su vez, nos encontramos con el reto de integrar hardware dedicado (sensores de ocupación) y software en un mismo sistema.

Para cada una de estas etapas, se ha descripto oportunamente un marco teórico, y se han expuesto los desafíos encontrados y su forma de resolución.

Con respecto a la solución propuesta, lo que podemos concluir, teniendo en cuenta los resultados esperados y obtenidos, es que la misma es adecuada para resolver el tipo de problema planteado inicialmente.

Por un lado, nos enfocamos en diseñar una arquitectura que permita procesar los eventos en tiempo real y, en base a las mediciones realizadas, llegamos a la conclusión de que las herramientas elegidas son apropiadas para este tipo de sistema. Al mismo tiempo, comprendemos que hay un punto en donde se observa la mayor parte de la latencia y es el tipo de conexión entre el backend y el frontend, ya sea 3G, 4G o WI-FI (que fueron las redes probadas). De todas maneras, dicha latencia es baja y resulta poco perceptible para la

usabilidad del sistema. Por lo tanto, podemos determinar que los resultados obtenidos se correlacionan con los resultados esperados, y que el desarrollo de esta tesina cumplió con los objetivos que nos propusimos inicialmente.

Para finalizar, en lo personal, este trabajo nos ha permitido experimentar en profundidad con tecnologías no estudiadas en la carrera. Mas específicamente, el hecho de lograr una interacción entre componentes hardware y software con los cuales nunca habíamos trabajado antes. Por otro lado logramos adquirir conocimiento en tecnologías de desarrollo de software que son tendencia actualmente; como NodeJS, que nos brindó una alternativa para el desarrollo del backend utilizando el mismo lenguaje de programación que se utilizó para el desarrollo del frontend y la gestión de la base de datos. En cuanto al frontend, AngularJS nos pareció un excelente framework para organizar y estructurar el desarrollo del cliente web. Por último, utilizar MongoDB nos permitió conocer un nuevo paradigma de base de datos, orientado a proveer una mejor performance en aplicaciones de tiempo real, a diferencia de lo que tradicionalmente estábamos acostumbrados a utilizar en la carrera y en nuestras experiencias laborales; además de que, al estar desarrollado en JavaScript, se adecua naturalmente al full-stack JS en el cual está montado el sistema.

A modo de apreciación personal, el desarrollo de este trabajo fue una experiencia muy grata y enriquecedora; por todo lo mencionado anteriormente y por los desafíos y problemas que nos encontramos y logramos solucionar.

7.2 Líneas de trabajo futuras

Dado que el sistema se pensó de forma tal que se pueda extender fácilmente y que soporte la integración de distintos componentes, existen varias mejoras y puntos de extensión que se detallan a continuación.

- Agregar sensores de ocupación y lectores RFID de diferentes fabricantes.
- Combinar hardware actual con otros componentes (de ocupación y autenticación) que incrementen la confiabilidad del sistema.
- Automatizar autenticación de una plaza de estacionamiento para reemplazar lecturas RFID manuales.
- Implementar aplicaciones mobile para otras plataformas.

REFERENCIAS

[1] UK Government - Department for Business, Innovation and Skills, "SMART CITIES: Background paper", 2013

[2] SAMUEL IDOWU, NADEEM BARI, “A development framework for Smart City Services”, Master's Thesis, Lulea University Of Technology, Lulea, Sweden, 2012

[3] Fundación Telefónica, "Smart Cities: un primer paso hacia la internet de las cosas", 2011

[4] International Telecommunication Union, "ITU Internet Reports: The Internet of Things", Geneva, 2005

[5] Jordán Pascual Espada, Tesis Doctoral, “Diseño de objetos virtuales colaborativos orientados a servicios en el marco de Internet de las cosas”, Universidad de Oviedo, 2012

[6] YANFENG GENG, CHRISTOS G. CASSANDRAS, “A new “Smart Parking” System Infrastructure and Implementation”, Division of Systems Engineering, and Center for Information and Systems Engineerin, Boston Univeristy, Brookline, MA, 02446, USA, 2012

[7] DONALD SHOUP, "Ending the Abuse of Disabled Parking Placards", ACCESS Almanac, Number 39, 2011

[8] WorldSensing, "Fastprk, Moscow, THE LARGEST SMART PARKING PROJECT IN THE WORLD",

[9] ITS International, "Smart detection of disabled bay abuse", 2016

[10] DONALD SHOUP, "Cruising for Parking", ACCESS Almanac, Number 30, 2007

[11] Hongwei Wang, Wenbo He, "A Reservation-based Smart Parking System", Department of Computer Science & Engineering, Department of Electrical Engineering, University of Nebraska-Lincoln, NE, USA, 2011

[12] Antonio Liñán Colina, Alvaro Vives, Antoine Bagula, Marco Zennaro, Ermanno Pietrosemoli, "INTERNET DE LAS COSAS", 2015

[13] IEEE Computer Society, Low-Rate Wireless Personal Area Networks (LR-WPANs), 3 Park Avenue, New York, NY 10016-5997, USA, ISBN:978-0-7381-6684-1, 2006

[14] Urbiotica, "Products", http://www.urbiotica.com/en/products/, Barcelona, España

[15] Stephen A. Weis, "RFID (Radio Frequency Identification): Principles and Applications", MIT CSAIL, 2007

[16] Jorge Alberto Alvarado Sánchez, Tesis, "Sistema de Control de Acceso con RFID", Centro de Investigación y de Estudios Avanzados del Instituto Politíecnico Nacional, Departamento de Ingeniería Eléctrica, Sección de Computación, México, D.F., 2008

[17] Pedro Texeira, "Professional Node.js", John Wiley & Sons, Inc., Indianapolis, USA, ISBN: 978-1118185469, 2013

[18] Stefan Tilkov, Steve Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs", IEEE Computer Society, www.computer.org/internet/, 2010

[19] NODE.JS, “About”, http://nodejs.org/about/, 2014

[20] Mike Cantelon, Marc Harter, T.J. Holowaychuk, Nathan Rajlich, "Node.js in Action", Manning Publications https://www.manning.com/books/node-js-in-action, ISBN: 9781617290572, 2013

[21] NODE.JS, "Node.js Documentation", https://nodejs.org/api/,

[22] Android, "Introduction to Android", https://developer.android.com/guide/index.html,

[23] MongoDB, "The MongoDB 3.2 Manual", https://docs.mongodb.org/manual/, 2016

[24] Christof Strauch, "NoSQL Databases", University Hochschule der Medien (Stuttgart Media University), Stuttgart, http://www.christof-strauch.de/nosqldbs.pdf

[25] Kristina Chodorow, "MongoDB: The Definitive Guide", Second Edition, O’Reilly Media, Inc., United States of America., ISBN: 978-1-449-34468-9, 2013

[26] AngularJS, "What Is Angular?", https://docs.angularjs.org/guide/introduction, 2016

[27] Brad Green, Shyam Seshadr, "AngularJS", First Edition, O’Reilly Media, Inc., United States of America., ISBN: 978-1-449-34485-6, 2013

GLOSARIO

ABM: Módulo de Alta, Baja y Modificación.

ACID: Atomicity, Consistency, Isolation, Durability (Atomicidad, Consistencia, Aislamiento, Durabilidad) se refiere a las propiedades de las transacciones en una base de datos.

AJAX: Asynchronous JavaScript and XML (JavaScript asincrónico y XML). Tecnología utilizada en navegadores para hacer llamados asincrónicos a servidores web.

API: Application Programming Interface(Interfaz de Programación de Aplicaciones) Es el conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

CommonJS: Proyecto con el objetivo de especificar un ecosistema JavaScript por fuera de los navegadores.

DDBB: Bases de datos.

GPRS: General Packet Radio Service (servicio general de paquetes vía radio). Servicio de transmisión de datos mediante conmutación de paquetes.

IEEE: Institute of Electrical and Electronics Engineers (Instituto de Ingeniería Eléctrica y Electrónica).

IoT: Internet of Things (Internet de las cosas).

JSON: JavaScript Object Notation (Notación de Objetos JavaScript). Estándar de formato para intercambio de datos en forma de objetos JavaScript.

MVC: Model–view–controller (Modelo–vista–controlador). Patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario.

NoSQL: (a veces llamado "no sólo SQL") es una amplia clase de sistemas de gestión de bases de datos que difieren del modelo clásico del sistema de gestión de bases de datos relacionales.

NPM: Node Package Manager. Manejador de paquetes por defecto para Node.js

OOP: Object-Oriented Programming (Programación orientada a objetos). Paradigma de programación.

RDBMS: Relational DataBase Management System (Sistema de gestión de bases de datos relacionales).

REST: Representational State Transfer (Transferencia de Estado Representacional). Estilo de arquitectura software para sistemas hipermedia distribuidos

RFID: Radio-Frequency Identification (Identificación por radiofrecuencia). Sistema de almacenamiento y recuperación de datos remoto.

SDK: Software Development Kit (Kit de desarrollo de software). Conjunto de herramientas de desarrollo de software que le permite al programador o desarrollador de software crear aplicaciones para un sistema concreto.

Smart Cities: Ciudades Inteligentes.

Smart Parking: Estacionamiento inteligente.

UTF-8: 8-bit Unicode Transformation Format (Formato de transformación Unicode de 8 bits). Formato de codificación de caracteres.