Notificaciones Push

Para dispositivos móviles y la web

Guillermo López Leal

Notificaciones Push: Para dispositivos móviles y la web

Guillermo López Leal

Agradecimientos

- A Bernardo López y Carmen Leal, por todo lo que han hecho siempre por mi. Gracias.
- A Leticia Núñez, por aguantar mis peculiaridades.
- A Francisco Marzal, por su inestimable ayuda durante la carrera y el proyecto.
- A Fernando Rodríguez Sela, Fernando Jiménez Moreno, José Antonio Olivera Ortega, Ignacio Eliseo Barandalla Torregrosa y al resto de gente de Telefónica I+D, ya sea de la iniciativa Open Web Device u otras, por todo lo que me han durante el tiempo que pasé allí. Gracias a todos.
- A Manuel Touriño y Francisco Sánchez, por la revisión a fondo de este proyecto, por estar cuando se les necesita y por ser grandes compañeros de carrera.

Tabla de contenidos

Lista de figuras

Lista de ejemplos

Capítulo 1. Introducción

Una de las grandes ventajas que ha aportado Internet al desarrollo de aplicaciones es, aparte de una distribución mucho más sencilla, la posibilidad de poder obtener datos de forma dinámica. La mayoría de las veces se permite cambiar el comportamiento del propio programa gracias a los datos descargados desde la red o incluso actualizar la información que se va a mostrar a los usuarios dependiendo del momento del día u otras variables: los resultados de nuestro equipo favorito de fútbol, el tiempo que va a hacer en nuestra ciudad, o incluso mensajería instantánea, son algunos de los usos que se hacen de las tecnologías de Internet para aplicaciones locales.

A lo largo de los años, las tecnologías para descargar estos datos han ido variando, acomodándose a los cambios que han sucedido en Internet, desde las redes de módem lentas que conocemos actualmente, hasta redes de fibra óptica capaces de descargar un gran flujo de datos pasando por las olvidadas redes móviles, que son aquellas que más han sufrido esta descarga de datos de una manera, muchas veces, descontrolada.

Así pues, la descarga de toda comunicación puede realizarse de diferentes maneras, ya sea de forma síncrona o asíncrona, pudiendo agruparse en dos grandes categorías para conseguir esta información:

- **Poll.** Periódicamente se pide información al servidor de terceros.
- **Push.** El servidor manda información al cliente cuando hay información disponible para él.

Estos dos principales paradigmas son los más usados en la historia de Internet y de las aplicaciones. Durante mucho tiempo sólo se utilizaron las soluciones de polling, ya que no se había desarrollado una tecnología de push eficiente y eran más fáciles de implementar (preguntar cada cierto tiempo al servidor si había datos nuevos, manteniendo un pequeño intervalo en el cual cliente y servidor no estaban en sincronía. Sin embargo, ahora que la tecnología se ha ido desarrollando y se han creado nuevos paradigmas, el uso de notificaciones push está en auge, sobre todo gracias a los sistemas operativos móviles como Android o iOS), así como con nuevos estándares Web que permiten mantener sockets abiertos entre dos equipos diferentes para enviar datos, como WebSockets[1] y Server-sent Events[2].

Sin embargo, el primer método no es recomendable por varios motivos, ya que hace que las redes se colapsen con su uso o bien que éste sea muy ineficiente. El primero de ellos es porque usa un elevado número de conexiones hacia el servidor de terceros para preguntar si hay datos. En segundo lugar, porque esas conexiones no siempre son interesantes de realizar, puesto que puede que realmente no haya datos, pero al preguntar si los hay, se colapsa

[1] http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-17
[2] http://dev.w3.org/html5/eventsource/

tanto el servidor como la red con paquetes "inútiles". Y es la combinación de estos dos problemas (realizar conexiones para nada) lo que indica que debería haber otra tecnología de recogida de información más eficaz y amigable con las redes.

Es por esto que los métodos de recogida de información basados en tecnologías push son ampliamente usados para la descarga de datos de forma asíncrona, ya que sólo piden los datos cuando realmente son necesarios. Sin embargo, aunque estos métodos a primera vista puedan verse como más eficientes que los de polling, también cuentan con varias desventajas y pese a ser realmente válidos para conexiones estables y sin limitaciones, son muy poco efectivos y muy perjudiciales para las redes móviles, como las de telefonía.

Resumen

La creación de un sistema de notificaciones push, con un API sencilla y fácil de utilizar, transparente para los desarrolladores de aplicaciones, con el objetivo de que funcione en diferentes tipos de redes de datos, como celulares o ADSL, en el dominio de paquetes o de circuitos, y haga un uso mínimo de recursos y de batería, utilizando todas las peculiaridades de cada servicio para que sea la mejor solución en cada situación.

Objetivos

- Explicar por qué es un problema usar cualquiera de las dos soluciones planteadas al inicio del capítulo en dispositivos móviles y redes celulares, viendo cuál es el problema inicial, las singularidades de dichas redes móviles y cómo pueden mejorarse.

- Creación de una plataforma de mensajería push que sea amigable para los dispositivos móviles con las consideraciones expuestas en el punto anterior: que no use demasiada batería, que sea interesante para las operadoras móviles y que reduzca el consumo de ancho de banda y de señalización en la red celular.

- Implementación de este mecanismo de push, no sólo en su parte servidor si no en su parte cliente. Este desarrollo está englobado en la creación del sistema operativo para móviles Firefox OS.

- Estandarización al organismo W3C, para que lo implanten todos los navegadores y haya una mejor unión entre las redes de telefonía móvil y el mundo de Internet, para el intercambio de datos de forma eficiente.

Descripción del servicio

La plataforma del servidor de notificaciones se encargará de enviar mensajes push (pequeños mensajes, como conversaciones de chat, una estructura

JSON sobre la descripción de un partido de fútbol…) a terminales que están dentro de las redes móviles, exponiendo unas APIs claras y sencillas, tanto para el desarrollador de la aplicación web como para el creador de los sistemas operativos.

El principal objetivo de crear este servicio es usar de una forma mucho más eficiente los recursos de las redes móviles disminuyendo el uso de la batería, a la vez que se reduciría el tráfico de señalización en la red de telefonía móvil y se permitiría a las empresas de telecomunicaciones ahorrar medios y ofrecer un mejor servicio a sus usuarios.

Capítulo 2. Estado del arte

Este capítulo tiene como objetivo enumerar las diferentes tecnologías que se han usado a lo largo del tiempo para mandar mensajes push a los dispositivos móviles. Algunas de ellas han sido creadas por los operadores, por lo que tienen un carácter de buen uso de las redes móviles. Sin embargo, ninguna de ellas se ha impuesto sobre las creadas por las empresas de software o de Internet por varias razones que mostraremos más adelante.

Pero antes de comentar las diferentes propuestas que se han realizado en los últimos años, es necesario saber cómo funcionan las redes móviles a nivel radio y cómo los operadores tienen montada su infraestructura móvil.

Redes móviles: funcionamiento y problemas

Para crear un servicio que sea especialmente interesante para las redes móviles y que use pocos recursos, ancho de banda y en general, que cualquier operadora pueda pensar en él, hay que tener en cuenta los siguientes aspectos: el funcionamiento de las redes móviles que están desplegadas en el mundo, los detalles de las implementaciones, por qué se han hecho e incluso los acuerdos entre la capacidad de las líneas, los mensajes de señalización y otros muchos aspectos para que el servicio esté equilibrado entre los usuarios de las redes y los propietarios de estas.

Así pues, para conocer en profundidad el gasto de batería en los dispositivos móviles durante la conexión y por qué no se hace un buen uso de todos los recursos que aportan las redes móviles, es necesario empezar planteando un escenario satisfactorio que guste a todas las partes.

Redes LAN públicas o privadas

En primer lugar, hay dos tipos de redes principales para identificar los terminales dentro de las redes de telefonía, basadas en IP:

- **IPv4.** Son las direcciones IP más usadas en la actualidad y estandarizadas en el año 1981, en el RFC-791[1]. Se caracterizan por tener 2^{32} direcciones disponibles para asignar, equivalentes a 4.294.967.296 equipos. Sin embargo, el principal problema es que son demasiado pocas para todos los diferentes dispositivos que hay conectados en la actualidad a Internet y requieren de técnicas (como el NATting) para poder acomodar más equipos, con problemas que se analizan más adelante.

- **IPv6.** Es la evolución directa de IPv4, estandarizada como RFC-2460[2], pero sin compatibilidad hacia atrás. De esta forma, los equipos que funcionan sólo con IPv6 no son capaces de llegar a dispositivos equipados sólo

[1] http://tools.ietf.org/html/rfc791
[2] http://tools.ietf.org/html/rfc2460

con IPv4 y para que puedan hablar entre ellos, hay que realizar túneles o conversiones entre ambos tipos de direcciones. Esto cuenta con una serie de ventajas sobre IPv4, la principal es que el número de dispositivos con IP distinta y no repetida aumenta de forma dramática (hasta el punto de llegar a 340 sextillones de direcciones, o 2^{128}) y que, en un futuro, relegará a IPv4 a un segundo plano.

Además de los tipos de direcciones, ya sean IPv4 o IPv6, existe el problema de si dichas IPs están en redes públicas o privadas, las cuales limitan el acceso que tienen hacia otros dispositivos dentro o fuera de dichas redes. Esto puede ser beneficioso en algunos casos y realmente perjudicial en otros.

Así pues, existen estos dos tipos de redes, que se explican mediante:

- **LAN pública.** Los equipos conectados a este tipo de redes poseen una dirección IP (ya sea de versión 4 ó 6) que les permite ser visibles para el resto de dispositivos conectados a Internet y, en determinada medida, accesibles sin restricciones. Es decir, cualquiera puede saber si un usuario está en Internet y mantiene una conexión directa con los equipos a los que se quiere conectar (servidores web, de correo, de mensajería instantánea), y ellos pueden contactar con dicho usuario aunque no tengan conexión, puesto que es visible.

- **LAN privadas.** Nos encontramos dentro de un segmento de red que sólo es visible para otros equipos que lo compartan con nosotros. Esto hace que no tengamos una conexión directa con Internet y que haya que pasar por un dispositivo intermedio (normalmente un NAT) que permite salir a Internet no con nuestra dirección real, si no con la dirección que tenga dicho NAT y redirija nuestras peticiones hasta el exterior, a la vez que reconduce las respuestas desde el exterior hasta nuestro dispositivo.

Sin embargo, por diferentes motivos, ninguna de las soluciones es la panacea. En primer lugar, no contar con un control intermedio (normalmente un firewall) para protegerse de ataques, hace que tener una IP pública suponga una exposición de forma completa a cualquier dispositivo de Internet . Además, ser visible por todo el mundo, convierte en vulnerable cualquier petición que proceda del exterior, como escaneos de puertos, PINGs, o floods incontrolados. Es por esto, que la mayoría de operadores de telefonía móvil que disponen de servicios de datos, tratan de no dar IPs públicas a sus dispositivos para evitar estos problemas, que al final llevan a un mayor uso de batería debido ya que reciben más datos, detalle que se explicará en la siguiente sección.

Pero a su vez, tener un direccionamiento privado tampoco ayuda en demasía a las redes móviles. No ayuda puesto que, aunque en algunos casos sea útil como en el momento de recibir notificaciones push, el inconveniente suelen ser problemas de seguridad o gasto de batería. Tener un direccionamiento privado nos convierte en visibles sólo para nuestros vecinos en nuestra red privada y dado que la IP interna es diferente a la externa, cualquier intento

de conexión desde la red pública (Internet, por ejemplo) no llegaría a nuestra IP, ya que diferiría de la asignada para salir a Internet.

También se podría pensar que debido a que IPv4 tiene un rango de direcciones muy limitado, el uso de redes privadas con NATing está muy extendido, algo que es correcta. Pero también podría pensarse que usando IPv6 todos los problemas estarían resueltos y que al haber tantas posibilidades de direccionar dispositivos, el uso de mecanismos de push se irá perdiendo. Sin embargo, hay que tener en cuenta, como queda expresado anteriormente, que las operadoras suelen desplegar sistemas de firewall, por lo que aunque sus clientes dispongan de terminales con IPv6 pública, y por lo tanto, visibles desde Internet, se bloquearían algunas conexiones para así evitar la congestión en la red y un uso elevado de batería. De esta forma, habría que crear reglas específicas en los firewall para permitir conexiones entrantes hacia los dispositivos por parte de determinados lugares.

Redes fijas en el hogar

No se ha hablado de redes en el hogar puesto que la diferencia no es tan apreciable y permiten mantener equipos fuera de Internet, como televisiones o sistemas DLNA. Además, es transparente para los usuarios, ya que el peso lo lleva el router, que además de proporcionar acceso a Internet, actúa como NAT, facilitando sistemas 2 en 1.

Estados del dominio de circuitos

Como se ha comentado anteriormente, a continuación se procede a describir los estados radio más interesantes. En la especificación 3GPP TS 25.331[3] se pueden ver todos los estados de la capa RRC (Radio Resource Control):

Para simplificar, sólo están listados los estados de la red 3G, actualmente la más usada.

- **Cell_DCH (Canal dedicado).** Cuando el teléfono se encuentra en este estado radio es porque está transmitiendo una gran cantidad de datos y la red le ha puesto en un canal dedicado. El tiempo de inactividad es muy corto para este estado, conocido como el temporizador T1, y suele variar entre 5 y 20 segundos. Esto quiere decir que si se alcanza este temporizador, el teléfono cambiará su estado radio a Cell_FACH.

- **Cell_FACH.** En este estado, el teléfono está conectado a la red móvil y usa un canal compartido con otros terminales. Suele transmitir un pequeño número de paquetes, que normalmente son keep-alives. El temporizador de inactividad en este estado es un poco mayor (del orden de los 30 segundos) y es conocido como el temporizador T2. Cuando se dispara, el dispositivo se mueve por la radio al estado Cell_PCH o URA_PCH (dependiendo del tipo de red).

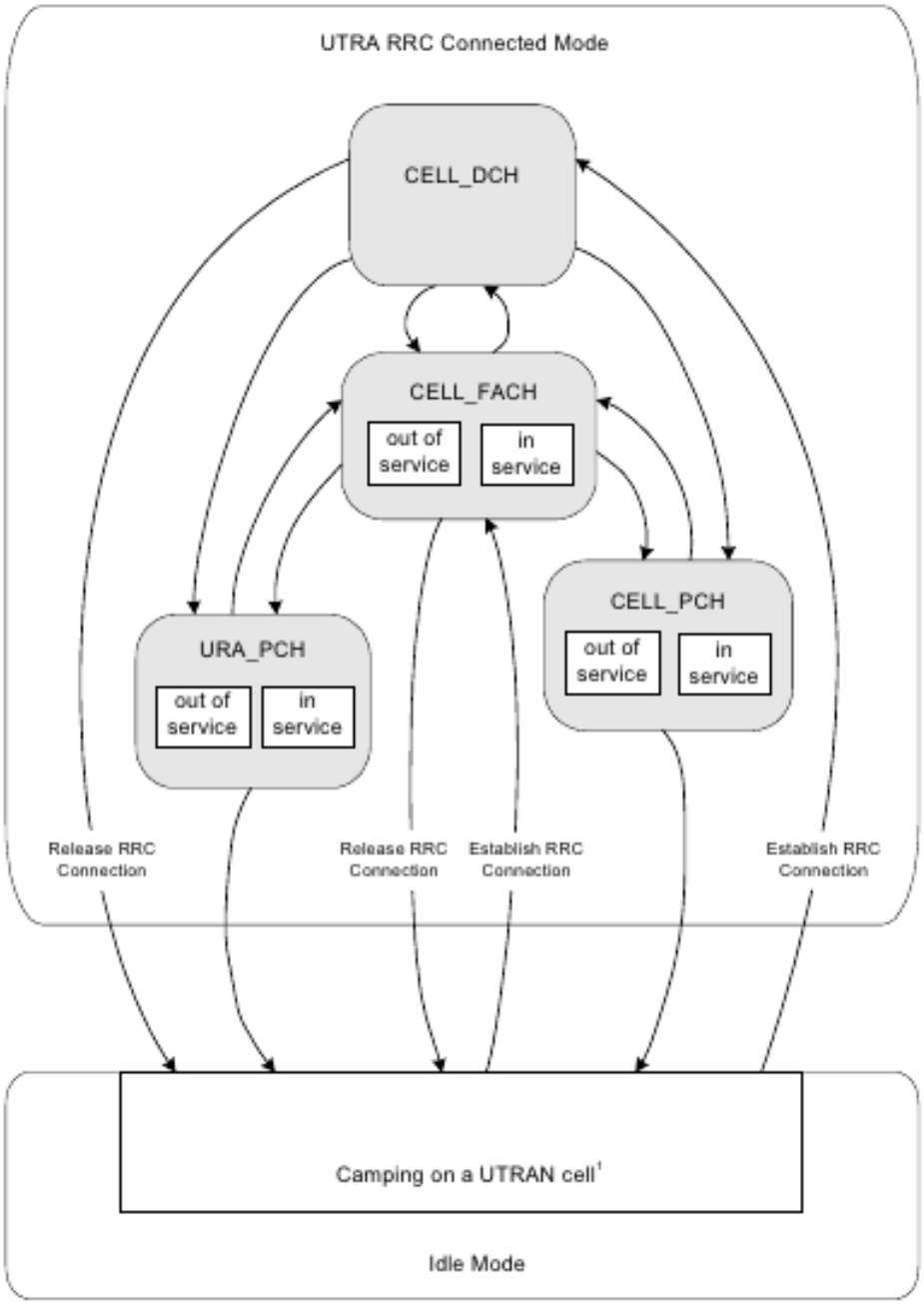
[3] http://www.3gpp.org/ftp/Specs/html-info/25331.htm

- **Cell_PCH o URA_PCH (PCH: Paging CHannel) (URA: UTRAN Registration Area).** En este estado, el teléfono no puede mandar ningún dato, excepto información de señalización para poder localizar el dispositivo dentro de la red móvil. En este caso, la conexión RRC está establecida y abierta, pero apenas es usada. En dicho estado, el dispositivo informa a la red cada cierto tiempo de que el dispositivo cambia de una celda (o sector) a otra, para que la red sea capaz de conocer exactamente en qué BTS (Base Transceiver Station, o nodo B) se encuentra ofreciendo servicio al dispositivo. El temporizador T3 define el tiempo máximo que un terminal puede estar en un estado PCH. Este temporizador es mayor que T1 y T2 y depende de cada operador de red. Cuando se dispara, el dispositivo se mueve al estado IDLE, por lo que cualquier dato que quiera transmitir el dispositivo le llevará unos dos segundos para restablecer el canal y una gran cantidad de mensajes de señalización.

- **RRC_IDLE.** Es el estado más económico en el que puede estar la radio, ya que prácticamente está dormida. También, se ha de señalar que el modem está escuchando los datos de célula, por lo que cada vez que detecta que el usuario cambia de un LAC (Localization Area Code, un grupo de múltiples BTS o NodosB) a otro, el dispositivo cambiará el estado PCH para informar a la red de su nueva ubicación. Así pues, cuando el dispositivo se encuentra en este estado, puede pasar a un estado más activo de red. Esta también sabe en qué LAC se encuentra moviéndose el dispositivo, por lo que si la red necesita informar al dispositivo de que está ocurriendo algo (una llamada, un SMS, alguien quiere mandarle datos), se manda un mensaje de broadcast, llamado Paging por toda la red LAC de BTS para localizar el dispositivo.

El siguiente esquema representa los diferentes estados radio, ordenados por consumo energético en el dispositivo.

Figura 2.1. Esquema de estados radio



En definitiva, el estado más interesante para mantener un dispositivo móvil en la red celular es el modo Idle, puesto que no tiene una conexión de red abierta constantemente y el uso de recursos es de alrededor 100 veces menos entre este modo y el de máxima excitación (CELL_DCH), permitiendo que la batería dure más tiempo, haya menos señalización de red y se ocupen menos recursos, tanto en nuestro dispositivo como en la red.

Entonces, como se puede ver, se quiere mantener el dispositivo durante el máximo tiempo posible en este estado, por todas las razones expuestas anteriormente.

Operadores

Así pues, vistos los problemas que tiene la red móvil, cómo se comporta y cuál es el mejor estado de radio para mantener a un terminal para que el uso de batería y de recursos de red sea el menor posible, vamos a explicar las soluciones actuales de push, tanto las propuestas por las propias operadoras de telefonía, ya sea por parte del GSMA[4] como por OMA[5], dos de los organismos internacionales en la estandarización de protocolos para telefonía móvil.

WAP Push

Historicamente, los operadores móviles han ofrecido, y ofrecen, mecanismos reales para notificaciones push, conocidos como WAP Push. Este mecanismo es muy eficiente con la red, ya que incluso permite que no se tenga una conexión de datos abierta (con su contexto PDP), ya que no funciona bajo el dominio de paquetes, si no de circuitos. Su funcionamiento es muy similar a la recepción de un SMS o de una llamada de teléfono, en el cual el operador busca si el móvil está conectado, en qué área se encuentra disponible y posteriormente hace un broadcast a la célula para despertar al teléfono (mediante un mensaje de PAGING) y decir que tiene algo disponible para él.

Lo interesante de este sistema es que es algo integrado de forma perfecta en la red, ya que no son más que mensajes de SMS especialmente compuestos, por lo que usa los "restos" del espectro móvil para ser transmitidos y realmente no conllevan un gran coste para las operadoras.

Sin embargo, tiene un problema principal, y es que las operadoras quisieron cobrarlo, y aún hoy en día lo cobran. Para ellas apenas tiene un coste económico, sin embargo, tiene un gran ahorro de batería y de recursos de red si se hubiera ofrecido desde un primer momento de forma gratuíta y no hubieran obligado a crear soluciones propietarias que dañan las comunicaciones, como se hablará en la sección de Internet.

Así pues, su funcionamiento es muy sencillo. El servidor de terceros quiere enviarnos cualquier tipo de mensaje binario (aplicaciones, imágenes, vídeos, simple texto), por lo que en primer lugar se sube a un servidor que está en Internet, o bien en la red privada del operador, y después se manda un SMS con una estructura XML determinada que contiene una URL con la dirección que hay que descargar, que se compila y se envía como un SMS binario. Posteriormente, el mensaje es interpretado por el sistema operativo del móvil en cuestión y abre el navegador apuntando a la dirección en concreto que se había señado en el mensaje WAP Push, iniciando la descarga.

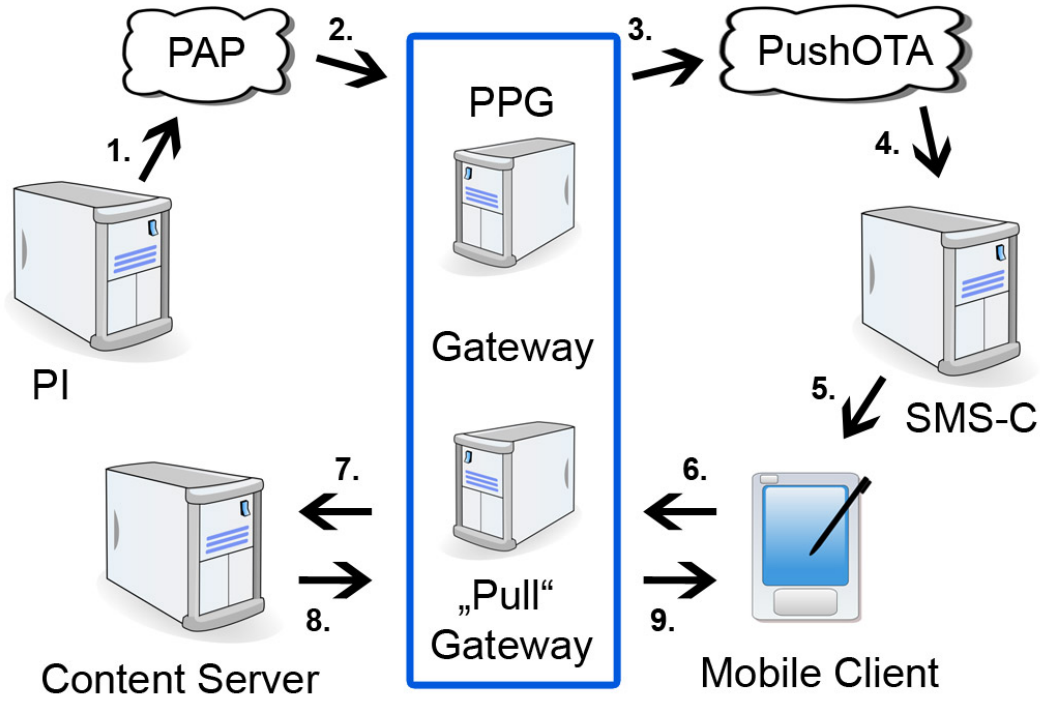
Como se puede observar, esto es un problema, puesto que está pensado para la descarga de datos desde Internet, y obliga a que se abra el navegador de forma predeterminada para la descarga del contenido.

[4] http://www.gsma.com/
[5] Open Mobile Alliance: http://openmobilealliance.org/

Sin embargo, tiene su parte positiva, y no es más que no utiliza un canal de datos dedicado en el dominio de paquetes, si no en el dominio de circuitos, y es una tecnología nativa en las redes móviles. Además, al ya haber despertado al teléfono, y por lo tanto a la radio, para descargar el SMS, el estado de consumo es alto, por lo que una segunda conexión a un servidor en Internet para descargar los datos reales no es tan perjudicial, sabiendo que sólo se conectará cuando haya datos que recoger, y no cada cierto tiempo.

Otro gran punto positivo es que es pueden coexistir tanto el WAP Push como otra plataforma de push, incluso aunque no funcione bajo la red de telefonía, como el Wi-Fi. Por ejemplo, podríamos recibir una notificación push por el método WAP Push y recoger los datos vía Wi-Fi si está disponible, por lo que normalmenta la descarga será más rápida, utilizará menos energía y bajará aún más la señalización necesaria para poder descargar, puesto que utiliza la red local y no la red proporcionada por el operador.

Figura 2.2. Funcionamiento WAP Push



Podemos explicar el funcionamiento de WAP push:

1. PI (Push Initiator), que es normalmente una aplicación que corre en un servidor web, se comunica usando el protocolo PAP (Push Access Protocol), que no es más que un XML usado para expresar las instrucciones de entrega del mensaje.

2. Este protocolo hace que el PI se entienda con el PPG (Push Proxy Gateway), que es el encargado de recibir el mensaje inicial. Este PPG es el principal actor para WAP Push, del que se han recogido varias de sus

ideas para el servidor de push que se desarrolla en este proyecto. Aparte de recibir el mensaje, es necesario que lo transforme o adapte para el receptor en concreto, guarde el mensaje por si el usuario no está conectado e incluso una conversión entre la dirección de recepción y el destinatario final.

3. Una vez procesado el mensaje, el PPG se comunica mediante PushOTA (Push On-The-Air) con el servidor central de SMS-C para pasar el mensaje (o mensajes) SMS que compondrán el contenido final. El protocolo PushOTA puede ser de varios tipos: una interfaz OTA-HTTP, una conexión OTA-WSP…

4. El mensaje es recibido por el SMS-C, que mirará a qué número va dirigido, cómo tiene que enrutarlo…

5. El mensaje se envía al dispositivo móvil del cliente usando la especificación normal de entrega de mensajes cortos SMS. Después, el sistema operativo del móvil interpreta el mensaje, viendo que es un mensaje WAP push y abre la página web que contiene el paquete binario que el proveedor quiere entregar. Esta apertura del navegador puede ser automática o bien aceptada por el usuario.

6. El navegador carga la dirección (que puede pasar por el PPG, pero no es obligatorio).

7. (el mismo paso que el anterior, en el caso de que exista proxy)

8. El servidor de terceros donde realmente está alojado el contenido devuelve la información requerida por la dirección que ha abierto el navegador del cliente. Este es el paso final, y el momento en que el usuario tiene los datos requeridos en un primer momento.

9. (pasando por el proxy intermedio si hiciera falta).

Así pues, podemos observar que el mecanismo de WAP push está diseñado para funcionar muy en sintonía con la propia red móvil. Esto hace que la transmisión sea muy sencilla y con un bajo coste para las operadoras, utilizando las redes de forma nativa para transmitir los datos necesarios para llegar al contenido final. Además, hace que los dispositivos puedan estar en modo de espera, con un poco uso de batería para ser despertados en el caso de que tengan notificaciones, de la misma manera que se hacen los SMS y las llamadas.

Internet

Una vez visto las diferentes plataformas push creadas por los organismos de los operadores de telefonía móvil, tenemos que movernos hacia tecnologías más modernas y que han sido creadas por empresas radicadas principalmente en Internet. Estas tecnologías son relativamente similares entre ellas, pero se difieren en las cuotas, autenticación… pero la idea sobre có-

mo mantener el canal de comunicación entre los dispositivos y el servidor de notificaciones es muy similar.

Así pues, vamos a ver ver cada una de estas tecnologías y a explicarlas brevemente.

GCM: Google Could Messaging

GCM o Google Cloud Messaging[6] es un sistema de notificaciones push creado por la empresa Google para su sistema operativo Android. Su finalidad es la misma que los demás sistemas: entregar notificaciones o mensajes que un servicio de terceros (o incluso la misma plataforma de Google) a los usuarios usando un dispositivo de una manera fácil, ordenada y controlada.

Lanzado en julio de 2012, es uno de los sistemas más grandes, puesto que está presente en todos los dispositivos Android que tengan una cuenta de Google asociada, los cuales son prácticamente la mayoría. El número de usuarios de este servicio no es especificado por Google, pero podría ser similar o inferior al número de dispositivos Android en el mercado, que supera los 600 millones de dispositivos[7].

> Google Cloud Messaging for Android (GCM) es un servicio que te permite enviar datos desde tu servidor a los usuarios con dispositivos Android. Esto puede ser un pequeño mensaje que dice a la aplicación que hay nuevos datos para ser descargados desde el servidor (por ejemplo, que un amigo ha subido un nuevo vídeo) o un mensaje que puede contener hasta 4KiB de información (por lo que las aplicaciones como las de mensajería instantánea pueden consumirlo directamente).
>
> El servicio GCM maneja todos los aspectos del encolamiento de los mensajes y entrega a la aplicación Android determinada que se puede ejecutar en el dispositivo.
>
> —GCM: Google Cloud Messaging for Android

Características. GCM tiene como características más interesantes:

- La aplicación a la que tiene que llegar la notificación no tiene por qué estar abierta. El servicio GCM se encarga de despertarla y que maneje la notificación.
- No hay ninguna interfaz para administrar la notificacición. GCM simplementa pasa datos en crudo desde que la notificación es recibida hasta que es entregada. La aplicación final es la que tiene la lógica necesaria para saber qué hacer con la notificación en todo momento. Por ejemplo, puede que la notificación sea "hay nuevos datos para descargar", por lo que no muestre ninguna interfaz, o bien un nuevo mensaje de un amigo, que sí es importante mostrar.

[6] http://developer.android.com/google/gcm/index.html
[7] XXXX FALTA REFERENCIA

- Requiere que la versión de Android instalada sea la 2.2 y además tiene que tener la aplicación de Google Play (el antiguo Market) configurada correctamente, esto es, con una cuenta de Google asociada. Pero no obliga a que la aplicación que reciba la notificación esté instalada por Google Play, si no que puede tener cualquier origen.

- Usa una conexión permanente con los servidores de Google para recibir las notificaciones y mensajes de control.

- Requiere que los servidores de terceros que envíen notificaciones estén registrados en la plataforma. Esto significa que Google puede revocar en cualquier momento el envío de notificaciones por parte de un desarrollador o empresa, simplemente no permitiendo el identificador único que se les proporciona en un primer momento.

- La identificación de la aplicación es de forma única. Esto significa que para cada aplicación que quiera recibir notificaciones push, en cada dispositivo, tiene un número único que la identifica de forma única dentro del servicio. Esta identificación única la da el servidor de GCM a la aplicación, que a su vez la tiene que enviar al servidor de terceros para que se envíe concretamente a esa instancia.

Así pues, el diagrama principal del envío y entrega de un mensaje de una forma muy abstracta es:

Nota

Suponemos que la aplicación instalada tiene un ID de registro que permite recibir notificaciones, además de que el servidor de terceros ha guardado ese ID y que el servidor tiene una clave de API que le permite identificarse como emisor de notificaciones.

1. El servidor de aplicación envía un mensaje a los servidores de GCM.

2. El servicio GCM encola el mensaje y lo guarda en el caso de que el dispositivo al que se tiene que entregar esté desconectado.

3. Cuando el dispositivo se encuentre online (puede que ya lo esté, o puede que no), el mensaje es mandado por un canal al dispositivo, que lo parsea el sistema operativo (Android, en este caso), y despierta a la aplicación destino en el caso que esté cerrada o la entrega directamente.

4. La aplicación procesa el mensaje y realiza cualquier tipo de evento relacionado con ella: sincronizar datos de fondo, mostrar una alerta…

GCM es un sistema que no es obligatorio para las aplicaciones Android publicadas en el Market de Google, sin embargo, su diseño hace que sea bastante eficiente para las redes móviles y haya un buen puñado de ventajas que hacen que los servicios de terceros tengan que tener menos lógica y preocuparse menos de la suerte que van a correr sus mensajes.

> **Nota**
> Cabe señalar, que antes de la introducción de GCM para los desarrolladores de Android (en julio de 2012), antes había un sistema muy similar para teléfonos Android, llamado Cloud to Device Messagging, o más conocido como C2DM[8]

Ventajas. GCM tiene una serie de ventajas sobre su predecesor CD2M (no explicado puesto que ha sido sustituido por esta versión) de las cuales habría que destacar:

- Tiene un mejor uso de batería, ya que puede encolar mensajes para ser entregados cuando el dispositivo está con una conexión activa (y por lo tanto tiene la radio móvil activada). Además, no tiene por qué notificar al dispositivo siempre que le llegue una nueva notificación, si no que puede esperar a que pase a estar activo (por ejemplo, que se encienda la pantalla) para mandar los datos.

- El uso de datos y la transferencia es más eficiente, por las explicaciones del punto anterior. Esto hace que se gaste menos batería y que el usuario sólo pueda recibir notificaciones cuando realmente son interesantes: cuando se usa el dispositivo, y mantener en cola las no prioritarias.

- El API es más sencilla, puesto que hay menos pasos para poder usar el servicio y además, el código del cliente en las aplicaciones es más claro y fácil de implementar, sobrescribiendo algunos métodos de las clases de las cuales se extiende.

- La migración del servicio es muy simple, y sólo hay que cambiar la URL de push para que apunte hacia otro servidor.

- Se eliminan las cuotas que había en el sistema anterior, permitiendo a los desarrolladores crecer de forma sencilla y sin nuevas peticiones. Además, se pueden consultar el estado de los mensajes en el perfil de Google.

- Permiten un mensaje (`payload`) de 4KiB, por lo que muchos de los datos que puedan ir por esta mensajería push no requerirán que las aplicaciones se conecten al servidor de terceros para recoger la información y realizar una segunda conexión.

- Se permiten hacer mensajes multicast, esto quiere decir que se puede hacer sólo un envío de mensaje push, pero indicando todos los receptores en el momento de enviarlo, se entregarán a todos. Incluso múltiples emisores pueden enviar mensajes a una misma aplicación (por ejemplo, una red social que habla sobre un jugador de fútbol y a la vez información sobre el equipo en el que juega).

- Hay un tiempo de vida máximo de 4 semanas para cada mensaje. Si un mensaje viene sin tiempo de vida, se presupone que el tiempo es máximo,

[8] https://developers.google.com/android/c2dm/

sin embargo, un mensaje muy interesante es del de un TTL (`time_to_live`) de 0 segundos, el cual se envía a los receptores sin guardarlo, por lo que si no está conectacto el receptor (o los receptores), se pierde.

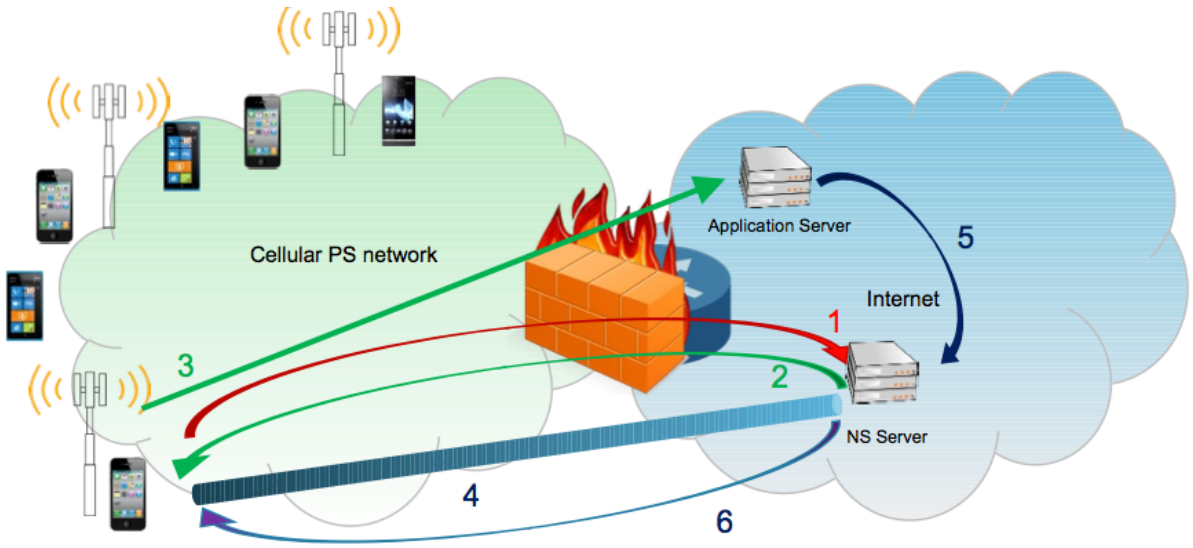
- Existe la posibilidad de sobrescribir mensajes antiguos (con la misma `collapse_key`), enviar mensajes sólo cuando el usuario esté activo (con `delay_while_idle`) e incluso comprobar si los mensajes van a ser entregados o no dependiendo de la respuesta que de el servicio.

A pesar de la enorme capacidad que tiene este sistema, con todas sus ventajas que se han mostrado anteriormente, sigue teniendo pequeños fallos que hacen que no sea ampliamente usado, por ejemplo:

1. **No es obligatorio.** Por lo que los diferentes desarrolladores no están obligados a usarlo en sus aplicaciones y pueden crear otros sistemas de push propietarios (como los que hacen la mayoría de aplicaciones de mensajería instantánea), lo que lleva a que la validez del sistema sea menor.

2. **Mantiene una conexión abierta.** Como se ha comentado, uno de los grandes problemas, junto a la señalización, es la de mantener canales abiertos sin hacer nada en espera de que alguna vez atraviese un dato, en nuestro caso notificaciones.

Figura 2.3. Funcionamiento de GCM



Arquitectura. Así pues, podemos observar el flujo completo de trabajo de GCM:

1. En la primera utilización del servicio, el teléfono dotado con Android (la única plataforma que soporta GCM por el momento) tiene que disponer de una cuenta de correo válida en el sistema (de GMail o de Google Apps), a la cual se asociará un identificador único de dispositivo. Además, también se enviará una petición para recibir una cadena única que identificará a una aplicación en un dispositivo en concreto.

2. La aplicación que pidió la identificación, la recibe del servidor GCM.

3. La aplicación manda, por su propio canal de comunicación que haya decidido, dicho token único a un servidor de terceros, con algún tipo de identificación de usuario o de instancia que permitirá enviar contenido personalizado. Por ejemplo, si una persona registrada desea recibir notificaciones sólo sobre su equipo favorito, tenemos que enviar el dato que identifique a esa persona/usuario con el identificador enviado.

4. El dispositivo abre y mantiene una conexión TCP persistente con el servidor de GCM, mandando mensajes de keep-alives (que viene a ser un "sigo vivo") en intervalos de tiempo determinados. Estos intervalos son, por defecto, de 28 minutos, pero varía dependiendo del país y de la operadora, e incluso de forma adaptativa.

5. En cualquier momento, el servidor de terceros (llamado Application Server) manda un mensaje push al servidor GCM con los parámetros que se especifican en el servicio, incluyendo el identificador de dispositivo recibido en el paso 2 y enviado en el paso 3.

6. GCM envía, en la medida de sus posibilidades, el mensaje al dispositivo a través del canal TCP abierto, ya sea de forma inmediata, o bien con los parámetros indicados en el paso anterior (como `delay_while_idle` o `time_to_live`).

Problemas. Sin embargo, a pesar de todas las nuevas características y todo lo que ha trabajado Google en este sistema, hay varios problemas que pueden ser bastante graves y que harían que no confiáramos en este sistema

- Se necesita mantener siempre una conexión abierta, por lo que sigue ocupando espacio en la red móvil, y en concreto en el GGSN (GPRS Support Node), que es un hardware con un precio muy elevado que es donde se mantienen cada conexión abierta que tengamos, hasta un límite de 65535.

- Esa única conexión requiere de keep-alives para mantenerse activa, por lo que cada cierto tiempo estaremos cambiando de estado radio, con su correspondiente señalización, para indicar simplemente que estamos conectados.

- Es un punto único de fallo (SPOF: Single Point Of Failure) son los servidores de Google, por lo que si caen, la infraestructura cae, y no hay posibilidad de cambiar a otra, puesto que el protocolo es cerrado y propietario.

- Todos los mensajes van por los servidores de Google, por lo que puede ser un posible riesgo de seguridad, más aún viendo las diferentes leyes para regular Internet que han salido desde los EE.UU, que es donde Google está radicada.

- No hay garantía sobre el orden de entrega de los mensajes, por lo que algunos de estos enviados posteriormente podrían llegar antes que uno

enviado previamente. De esta forma, no se pueden crear dependencias entre mensajes.

APNS: Apple Push Notification Service

TODO XXX APNS o Apple Push Notification Service es otro de los sistemas creado por empresas creadoras de sistemas operativos para solventar el problema de notificaciones push, en este caso para dispositivos iOS, como el iPhone o el iPad.

Este sistema es muy similar en comportamiento a GCM pero con alguna variación, sobre todo en los tipos de mensajes que puede enviar los desarrolladores y cómo se tratan las notificaciones por el usuario.

Historia. Fue lanzado con la versión 3.0 de iOS, el 17 de junio de 2009, que correspondía . Ha sido mejorado con el paso del tiempo e introducido en los equipos de escritorio con la versión 10.7 de MacOS usando el centro de notificaciones, también presente en iOS 5.

Características.

Ventajas.

Problemas.

Thialfi: el futuro de Google

Thialfi es un nuevo protocolo diseñado y publicado por Google[9] que intenta arreglar algunos de los problemas que tienen los sistemas de notificaciones que pasan mensajes completos a través de la red, y el consiguiente problema de infraestructura y escalabilidad que podría resultar de tener millones de terminales conectados a la vez mandando muchos mensajes a través de la red y en espera de ser entregados.

Nota

Sin embargo, cabe notar que este sistema no está funcionando en los terminales móviles que llevan Android, si no que se usa el ya explicado GCM, y este es sólo con la intención de explorar nuevas posibilidades, y lo implementa Google Chrome en su versión de escritorio y móvil, así como la sincronización de contactos de manera experimental.

El principal cambio entre GCM y Thialfi es la vuelta de mensajes que no llevan contenido útil para la aplicación, si no que se basan en mandar números de versión sobre temas en concreto ("topics") a los que el usuario y las aplicaciones se suscriben para mantenerse en sincronía.

Características. Así pues, Thialfi tiene una serie de mejoras con respecto a los otros sistemas, como son:

[9] http://research.google.com/pubs/pub37474.html

- **Escalable.** Puede tener constancia de millones de clientes y de objetos, ya que el tamaño para ambos es pequeño, sin tener que guardar datos finales, si no simplemente números de versión.

- **Rápido.** Según las pruebas y el paper original, la entrega de los mensajes, en este caso mensajes con números de versión, se realizan en menos de 1 segundo desde que el servidor de terceros manda un mensaje hasta que el cliente lo recibe.

- **Seguro.** Incluso si un centro de datos se cae, el sistema es capaz de recuperarse, volviendo a registrar los diferentes dispositivos y enventualmente volviendo a un estado correcto.

- **Fácil de usar.** Las APIs son sencillas y ha sido implementadas en diferentes servicios usados por Google, como en la sincronización de datos de Chrome, contactos y en Google+.

Entonces, como se comenta en los puntos anteriores, todo son objetos que poseen identificadores (`ID`) únicos y números de versión que incrementan de forma monótona en cada actualización.

Ventajas. ...

- **Menor datos guardados.** Es una de las ventajas, pero a la vez es un gran inconveniente. Sólo hay que guardar los objetos que contienen un identificador y un número de versión, por lo que las bases de datos (BigTable, que usan en Google) necesitan menos espacio para poder funcionar, al no guardar mensajes (en GCM el `payload` máximo es de 4KiB). Además, sólo es necesario guardar el último mensaje para cada aplicación-objeto (básicamente es aumentar el número de versión), mientras que en un sistema como GCM hay que guardar múltiples mensajes.

- **No hay fugas de datos privados.** Al no tener mensaje las notificaciones, no hay riesgo de fuga de datos privados, puesto que lo único que se transmite son números de versión, que no dicen nada sobre el contenido.

- **Entrega siempre actualizada.** La entrega de los mensajes está asegurada, no porque siempre se tenga la última versión en un momento determinado, si no porque seguro que se va a conocer esa versión en un momento futuro. Esto quiere decir que lo único que reciben los clientes es un número de versión de un determinado objeto, que siempre es el máximo posible, por lo que si tenemos un cliente que se ha quedado desactualizado en, digamos, la versión `17`, y ha llegado la `18` y no ha podido ser entregada, cuando llegue la versión `19` el cliente sabrá que tiene que pedir tanto la `18` como la `19`, para estar en sincronía con el servidor, por lo que en cualquier momento lo único que almacena un servidor o un cliente es un determinado objecto, con un `ID` en una versión `V` determinada.

- **Buena recuperación ante errores.** Thialfi tiene una serie de mensajes de control que permiten recuperar el estado de los registros en caso de

fallo. Por ejemplo, si el servidor pierde todos los registros, se envía un mensaje a los clientes para que ejecuten `ReissueRegistrations()`, y se inicia un protocolo para devolver los registros.

- **Fácil implementación del cliente.** El número de líneas requeridas para la implementación en el cliente es muy bajo, debido a su uso basado en eventos y su fácil lógica, en la siguiente figura podemos ver algunos ejemplos:

  Figura 2.4. Líneas de código para implementar Thialfi

Application	Language	Network Channel	Client Lines of Code (Semi-colons)
Chrome Sync	C++	XMPP	535
Contacts	JavaScript	Hanging GET	40
Google+	JavaScript	Hanging GET	80
Android Application	Java	C2DM + Standard GET	300
Google BlackBerry	Java	RPC	340

  Como se puede observar, varía dependiendo del lenguaje de programación, desde las 40 u 80 en Google Contacts y Google+ hasta las 535 bajo un protocolo XMPP en Chrome Sync.

  Además, lo interesante que se ve en esta figura anterior es que Thialfi no obliga un único protocolo de aplicación, si no que puede funcionar sobre varios, como conexiones HTTP, XMPP o incluso RPC.

Problemas. Sin embargo, aunque tiene muchas ventajas, como las están enumeradas anteriomente, tiene problemas para ser un buen servidor para datos en la red móvil

- **Doble conexión para recuperar datos.** En un sistema en el que los servicios de terceros pueden enviar datos de payload, obliga a que el servidor de notificaciones sea más complejo, por el hecho de que tiene que guardar datos. Sin embargo, esto puede significar que los clientes no tengan que abrir una segunda conexión para recuperar los datos necesarios debido a que el propio mensaje de notificación ya lo tiene. Esto, a pesar de parecer un problema menor, es uno de los grandes problemas en las redes móviles, donde realizar una conexión es muy costoso (en términos de tiempo y de señalización) y la descarga de los datos lenta.

- **Mayor complejidad en el servidor.** Esto es debido a que requiere que se guarden los cambios o diferencias para cuando se conecten los dife-

rentes clientes, puedan recuperar los datos que pidan, como por ejemplo, los datos de la versión `17`. Sin embargo, puede haber casos en que los clientes no vayan a pedir la última versión, si no que tengan que pedir la versión `16` y `17`. Por lo que el servidor tiene que mantener esas diferencias de algún modo (en formato delta o con mensajes completos).

- **Mayor complejidad en las aplicaciones.** Esto es debido a que las aplicaciones no van a procesar el mensaje directamente, si no que tienen que discernir cuál es el número de versión, hacer una llamada al servidor de terceros y recibir los datos. Además, hay que tener en cuenta que pueden pedirse diferentes versiones para acabar en sincronía con el servidor, lo que aumenta la lógica para ver qué versión está dando el servidor y cómo se puede manejar con versiones más nuevas.

- **Servidores de aplicación más potentes.** El primer problema es la necesidad de que todos los clientes, una vez recibido un número de versión, tengan que obligatoriamente conectarse con el servidor de la aplicación para descargar los datos, lo que puede conllevar a muchísima capacidad de cálculo (si por ejempo los datos hay que calcularlos al vuelo y dependen de cada petición) y de ancho de banda, así como de medidas para evitar que muchos usuarios se conecten a la vez y puedan colapsar los sistemas.

Así pues, puede verse a Thialfi como un posible candidado a complementar, que no sustituir, a GCM, por los problemas mostrados más arriba. La necesidad de crear una segunda conexión, y los posibles problemas que puede acarrear a los servidores de aplicación el tener que trabajar con millones de peticiones, hace que no sea escalable para pequeñas empresas y malo para dispositivos móviles. Sin embargo, en la práctica se ha demostrado que es competente, pero con grandes granjas de servidores como los centros de datos de Google.

Capítulo 3. Application Program Interface, API

El servidor de notificaciones tiene diferentes API que expone hacia los elementos que hay en la arquitectura y que hacen que los mensajes se entreguen a los destinatarios correctos. En un primer momento, y para mantener la compatibilidad con el estándar [http://dvcs.w3.org/hg/push/raw-file/default/index.html] que propone el W3C se mantienen los métodos, pero sin embargo, se añaden algunos parámetros extras o se eliminan algunos métodos para añadir nuevas características, seguridad y evitar redundancia.

Pero para entender este capítulo, es necesario presentar a las diferentes instancias que forman parte del servidor de notificaciones:

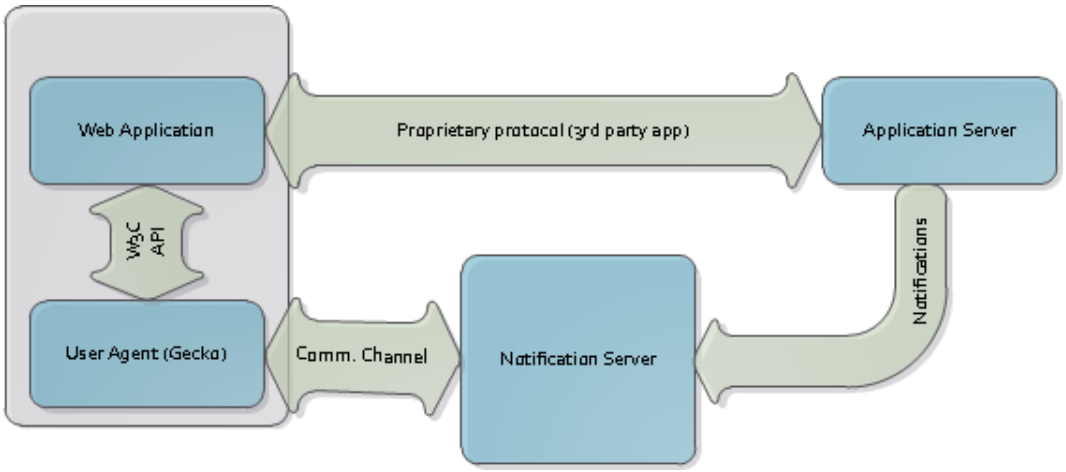
- **Aplicación web (WebApp: WA).** La aplicación del usuario que es ejecutada en el dispositivo.

- **Agente de usuario (User-Agent: UA).** Como el proyecto se inició bajo el paraguas de Firefox OS, cuyo motor de renderizado de páginas web, equivalente a aplicaciones web es conocido como el Agente de usuario, que en nuestro caso es Gecko, el motor de Mozilla.

- **Servidor de notificaciones (Notification Server: NS).** Es la infraestructura centralizada del servidor de notificaciones. Cualquiera puede desplegar una nueva instancia debido a que está liberado como código abierto [https://github.com/telefonicaid/notification_server]. Como el proyecto se inició bajo el paraguas de Firefox OS, cuyo motor de renderizado de páginas web, equivalente a aplicaciones web es conocido como el Agente de usuario, que en nuestro caso es Gecko, el motor de Mozilla.

- **Servidor de aplicaciones (Application Server: AS).** Es la parte servidora de la WA. Normalmente se podría definir como la presencia en Internet de la WA, que es la que recibe las URLs a las que tiene que hacer push y la que los realiza.

Figura 3.1. Instancias y comunicaciones



Así pues, se dividen la API en dos grandes grupos:

- **Externas.** Son aquellas que se están expuestas en los extremos del servidor, refiriéndose a la comunicación que tienen que realizar las aplicaciones (WA) con el agente de usuario (UA) y el servidor de terceros (AS) hacia el servidor de notificaciones NS). Son las usadas por desarrolladores externos para que su aplicación sepa recibir notificaciones y su servidor pueda enviarlas.

- **Internas.** Son las que los desarrolladores de aplicaciones no tienen que usar, ya que son transparentes para ellos y están creadas para que el protocolo funcione correctamente, llevando mensajes de señalización y de registro. Están involucrados principalmente el agente de usuario (UA) y el servidor de notificaciones central (NS).

Externas

Con estas APIs, los desarrolladores son capaces de hacer que los mensajes que quieran enviar desde un servidor externo lleguen correctamente hacia sus aplicaciones. La interfaz y el código es muy sencillo, para hacer que el desarrollo sea muy rápido.

> **Nota**
> Este API aún no está estandarizado por el W3C, por lo que lo aquí explicado es una proposición funcional.

Hay dos APIs principales externas expuestas para los desarrolladores de aplicaciones, que son:

Entre la WA y UA

Este API, llamada `navigator.mozPush`, es la principal vía de comunicación entre las aplicaciones y el agente de usuario, o sistema operativo.

Nota

Este API está prefijada en la primera versión funcional con `moz` puesto que se ha implementado por primera vez en un dispositivo por parte del navegador Mozilla Firefox y es el que tienen establecido para estas funcionalidades experimentales.

Está basada en el estándar propuesto[1] por el W3C, pero sin embargo tiene una serie de adiciones para añadir seguridad y eliminar la redundancia de alguno de sus métodos.

Con esta API, la aplicación es capaz de realizar dos funciones principales:

- Requerir permiso remoto para poder recibir una URI y comprobar si ya se tiene permiso, mediante la función `requestRemotePermission`.
- Revocar el recurso URI, para eliminar el registro de determinada aplicación, y no recibir más mensajes push, usando `revokeRemotePermission`

Así pues, la interfaz que define a `PushManager` (que es el contenedor que se expone a las aplicaciones), quedará de la forma siguiente:

Figura 3.2. Interfaz `PushManager`

```
interface PushManager {
  PushService requestRemotePermission (DOMString waToken,
                                       DOMString publicKey,
                                       optional DOMString algorithm);
  PushService checkRemotePermission ();
};
```

Método requestRemotePermission

Este método es el que tienen que llamar las aplicaciones para pedir permiso remoto para poder recibir las notificaciones. Así pues, es una petición que tiene dos resultados posibles, `onsucess` y `onerror`, dependiendo de si el resultado es satisfactorio o no. Como se he especificado más arriba, la signatura de la función es:

Figura 3.3. Función `requestRemotePermission`

```
requestRemotePermission (DOMString waToken,
                         DOMString publicKey,
                         optional DOMString algorithm);
};
```

Donde se ven los diferentes parámetros necesarios para poder operar correctamente, como son:

- **`waToken`.** Es el token único que tiene que generar la aplicación (ya sea en el servidor, o bien totalmente aleatorio) para identificarse de forma única

[1] http://dvcs.w3.org/hg/push/raw-file/default/index.html

junto a la clave pública en el servidor de notificaciones, que serán usados para crear una URL unívoca que servirá para mandar las notificaciones solamente a aquellas aplicaciones que hayan registrado ese par.

> **Nota**
> El desarrollador tiene la opción de hacer su `waToken` público y compartido, por lo que será usado para hacer broadcast de las notificaciones, o bien mantenerlo como un secreto para que sólo la instancia que contenga este token único sea la que reciba las notificaciones.

- **`publicKey`.** Es una llave pública que la aplicación tiene que tener guardada en el momento de su instalación (o bien transferida de forma segura por el servidor de terceros (`AS`: Application Server)) para poder ser usada en la comprobación de la firma de los mensajes por parte del componente `NS_AS` (Notification Server - Aplication Server) del servidor de notificaciones, para prevenir que en el sistema entren notificaciones y mensajes inválidos o maliciosos. Esta llave debe ser transmitida en Base64.

- **`algorithm`.** [Opcional]. Este parámetro indica el método de firmado usado por el parámetro `publicKey` para la comprobación de las notificaciones. El algoritmo predeterminado es el `RSA-SHA256`.

Además, los diferentes métodos serán llamados una vez se ejecute correctamente o con algún error la función, bien por un problema en la red, o bien por un problema en el servidor:

- **`onsuccess`.** Este método es invocado cuando la petición `requestRemotePermission` se ejecuta de forma satisfactoria. Lo que realmente hace es recoger un evento `success` que lanza el objeto `requestRemotePermission`. Así entonces, este evento tiene, como en la mayoría de los creados por objetos DOM en los navegadores, un atributo `detail` que contiene el mensaje satisfactorio:

  - **`event.detail.url`.** URL recibida desde el servidor de notificaciones y que será, probablemente, única para la instancia actual de la aplicación. Esta URL debe ser notificada por el medio más conveniente por la aplicación hacia su servidor de terceros, ya que será la usada para poder recibir notificaciones push.

- **`onerror`.** Método invocado cuando sucede algo no esperado en la respuesta del objeto creado por `requestRemotePermission`. Puede ser debido por múltiples factores, como fallo en la red (no disponible, o que se haya caído en medio de la tranmisión), fallo en el servidor de notificaciones (mensaje de error por no disponibilidad), o incluso por que el registro falle por tener un token incorrecto o duplicado.

  El evento que se captura también tiene el mensaje de error, que si bien no es informativo para las aplicaciones (debería ser completamente transparente), se muestra en:

- **`event.error`.** Mensaje de error. Bien generado por el propio sistema operativo o UA (User Agent) o bien el que es enviado por el servidor de notificaciones.

Ejemplo 3.1. Pedir permiso remoto y recibir URL

```
var watoken = 'mySecretWaToken';
var pbk = '…';

var push = navigator.mozPush;
var req = push.requestRemotePermission(watoken,
                                       encodeBase64(pbk));

req.onsuccess = function(event) {
  console.log('Yay! URL is: ' + event.detail.url);
};

req.onerror = function(event) {
  console.log('Oops!! Error is: ' + event.error);
};
```

Además, hay otro método, función o callback, llamado `onmessage`, que es ejecutado cada vez que la aplicación recibe un mensaje de notificación. Sólo es ejecutado cuando ha habido un `onsucess` previo, y no se ha producido ningún `onerror`.

- **`onmessage`.** Función que se llama una vez llega una notificación desde el sistema operativo. Hay que notar, que para que funcione esta opción, el objeto creado por `requestRemotePermission` tiene que seguir vivo y no haber sido borrado por el recolector de basura de JavaScript, por lo que se puede interpretar con que la aplicación está abierta, o bien se crea un objeto con una referencia fuerte, que significa que si la aplicación se cierra, el objeto sigue abierto y puede ser notificado.

  - **`event.detail.message`.** Mensaje completo. Es el `payload` que el servidor de las aplicaciones quería mandar desde un primer momento.

  - **`event.detail.url`.** [Opcional] URL a la que la notificación iba dirigida. Añadido por la posibilidad de que en un futuro las aplicaciones puedan pedir varias veces el permiso remoto, por lo cual recibirán diferentes URL a la que el servidor podría hacer push.

Nota

En la primera versión implementada en FirefoxOS, se utiliza el futuro estándar `setMessageHandler` del objeto `navigator` para poder ser despertado a pesar de que la aplicación esté cerrada.

```
navigator.mozSetMessageHandler("push-notification", function(msg) {
  alert("New Message with body: " + JSON.stringify(msg));
});
```

Método revokeRemotePermission

Este método realiza el paso contrario que el anterior, puesto que pide al servidor de notificaciones que se dejen de enviar notificaciones, esto es, que se quiere eliminar la URL o recurso dado en el paso anterior. Es un de-registro de URL. Puede servir en múltiples casos, como por ejemplo:

- Si la aplicación decide eliminar las notificaciones push por otro sistema y quiere dejar todo el sistema limpio, haciendo olvidar y eliminar todos los datos anteriores.
- El usuario decide editar las preferencias de la aplicación desde la configuración global del sistema y desactiva la opción de notificaciones push para determinadas aplicaciones. El sistema puede decidir no mandar las notificaciones a la aplicación o bien de-registrarla del servidor.
- El servidor de terceros ha sido comprometido y las aplicaciones tienen que re-registrarse para usar la nueva clave pública (PbK), por lo que en primer lugar tienen que de-registrarse para evitar que futuras notificaciones de orígenes malignos puedan volver a llegar a la aplicación.

Ejemplo 3.2. Pedir revocación de permiso (revocar URL)

```
var push = navigator.mozPush;
var req = push.revokeRemotePermission();

req.onsuccess = function(event) {
  console.log('Unregister successful');
};

req.onerror = function(event) {
  console.log('Oops!! Error is: ' + event.error);
};
```

Entre el WA y AS

La comunicación entre la aplicación web (WA) y el servidor de terceros (AS: Application Server) está fuera del alcance del servidor de notificaciones, ya que depende de cada aplicación el cómo se implementa.

Sin embargo, sí hay una serie de medidas que se pueden tomar para que esta transmisión de un dato importante sea seguro y no pueda ser interceptado por terceros que, aunque después sea complicado su uso porque depende de poder firmar con la clave privada las notificaciones, puede llevar a conocer datos privados de los clientes o instancias si se conoce el método de generación del `appToken` (que está contenido en la URL) y la clave pública.

- Que el envío se realice de forma segura, mediante HTTPS, para que sólo los extremos puedan conocer el contenido de la comunicación.
- Usar cifrado para evitar que un posible ataque de man-in-the-middle pueda recoger los datos y saber de qué se trata. Algo sencillo computacional-

mente ya que el cifrado se lleva a cabo en el cliente, que suelen tener menor potencia de cálculo que los servidores a los que van dirigidos.

- No pensar sólo en el envío HTTP, si no que otros sistemas podrían ser correctos, como XMPP, correo electrónico o incluso un mensaje SMS si fuera necesario.

- Por supuesto, asociar algún número que no permita saber a quién corresponde la URL enviada con qué usuario para un atacante externo, pero que el servicio pueda identificarlo de forma unívoca.

Entre el AS y NS

Este es el API que tienen que utilizar los servidores web para enviar las notificaciones de una forma estándar hacia el servidor de notificaciones para que éste las mande correctamente hacia los clientes.

El API se basa en protocolos estándar de la web y bien conocidos, como son los HTTP Post, mensajes JSON (JavaScript Object Notation) y las cabeceras de respuesta HTTP estándar, por lo que es muy sencilla la implementación y el protocolo, a parte de ser muy ligero y de haber cientos de programas software o librerías que implementan los métodos necesarios para poder enviar estas notificaciones.

Así pues, el servidor de terceros (AS: Application Server), necesita varios elementos antes de poder enviar su notificación, como son:

- Una clave privada, para poder firmas las notificaciones. Tiene que ser la correspondiente a la clave privada que tienen las aplicaciones guardada y con la que se han registrado.

- Una URL (o recurso unificado) a la que enviar las notificaciones. Habrá sido enviada por la aplicación en un paso anterior por algún método elegido por el desarrollador.

- Un mensaje que enviar (ya sea un número de versión o un mensaje con poco contenido o hasta 4KiB).

Entonces, una vez que el servidor de terceros tenga estos datos, es capaz de empezar a mandar notificaciones para que sean entregadas a las aplicaciones finales, siempre dependiendo de la URL y de la firma con la que se mandan dichas notificaciones.

> **Nota**
> Este API se corresponde a una simple interfaz REST que será mejorada en futuras versiones para incluir más características.

Así pues, se acepta una peticíon HTTP Post usada para mandar la notificación con el mensaje real dentro. El siguiente esquema debe ser enviado la `publicURL` que defina a la instancia de la aplicación para un usuario

determinado (explicado en los puntos anteriores), del estilo de: `https://push.telefonica.es/notify/SOME_RANDOM_TOKEN`

Figura 3.4. Notificación enviada

```
{
  messageType: "notification",
  id: "<ID used by the Application Server>",
  message: "<Message payload>",
  signature: "<Signed message>",
  ttl: "<time to live>",
  timestamp: "<Since EPOCH Time>",
  priority: "<prio>"
}
```

Así, como se puede ver la notificación tiene las siguientes características:

- Es una notificación con un estilo JSON, simple y sencillo, que es autocontenido y fácil de parsear y de crear.
- El tamaño de la notificación es mínimo, menor que si el sistema de intercambio de mensajes fuera un estricto XML.

Y define los siguientes atributos (obligatorios si no se indica lo contrario):

- **messageType.** Tiene que ser igual al texto `notification`. En un futuro podría haber más `messageType` para pedir información sobre la entrega de los mensajes o más funcionalidades.
- **id.** Identificador interno para el servidor de terceros, tiene que ser una cadena menor de 32 caracteres. Actualmente no tiene uso, pero en un futuro servirá para poder sobreescribir notificaciones (mandar otra notificación con el mismo `id` que el anterior y, si aún no se ha entregado, modificarla) y para poder preguntar cuál es el estado de la notificación: entregada, sin entregar, expirada…
- **message.** Es el texto que tiene que utilizar la aplicación y que el servidor de terceros envía. Tiene que ser un texto plano o bien estar en un formato JSON, pero convertido a cadena, por ejemplo, con la función `JSON.stringify` de JavaScript.
- **signature.** La firma con la clave privada de lo que se manda dentro de `message`. Se firma con `RSA-SHA256` por defecto, o con el algoritmo que se haya definido en el atributo `algorithm` pasado a la función `requestRemotePermission` cuando la aplicación ha pedido permiso a su sistema operativo.
- **ttl.** [Opcional] Tiempo de vida del mensaje. Está especificado en segundos, y puede ser desde 0, lo cual significaría que no se guarda en la base de datos, y si los destinatarios están desconectados, o no se puede entregar, la notificación se pierde; hasta 4 semanas (o 31 días, 2678400 segundos). El tiempo por defecto es de 4 semanas. Una vez superado ese tiempo, si aún sigue en el sistema (esto es, no se ha entregado), se elimina y no se entrega.

- **timestamp.** [Opcional] Tiempo desde el inicio UNIX, en segundos, para indicar cuándo salió la notificación desde el servidor de terceros. A esto se le suma el `ttl` para el tiempo máximo de vida en el sistema. Por defecto, se pone el momento en el que se recibe, si no está indicado.

- **priority.** [Opcional] Prioridad del mensaje. Varía entre `1` y `4`, donde 1 es más prioritario, y 4 es menos prioritario. Por defecto el valor que se indica es 4. Usado para mostrar al servidor si tiene que entregar la notificación en el momento o puede retrasarla hasta que haya más o el terminal esté despierto.

Pero además, se definen una serie de respuestas por parte del servidor dependiendo de si las notificaciones han sido aceptadas o no:

Mensajes sin códigos de error

Sólo hay un mensaje de OK, que indica que la notificación se ha procesado correctamente y va a poderse entregar a los dispositivos que están vinculados a la URL a la que se ha hecho HTTP Post.

```
Estado HTTP: 200
{
  status: "ACCEPTED"
}
```

Mensajes de error del servidor

Pero los mensajes más interesantes son los de error, que son todos aquellos que sean diferentes al de la cabecera HTTP con respuesta `200`. Dependiendo de la cabecera HTTP, el error varía, dando la razón en un formato escrito en el atributo `reason` de la respuesta, que es un JSON.

Nota

Estos mensajes de error están definidos en el fichero `/src/common/constants.js` a partir de la línea 28[2]

- **JSON inválido.** Se produce cuando la notificación no está correctamente formada, y no puede ser parseada como un JSON válido. Puede que sea debido a que no hay un atributo correctamente cerrado o contenga caracteres no parseables dentro del juego de caracteres de JavaScript (UTF-16).

```
Estado HTTP: 450
{
  status: "ERROR",
  reason: "JSON not valid error"
}
```

[2] https://github.com/telefonicaid/notification_server/blob/develop/src/common/constants.js#L28

- **URL incorrecta.** No se ha reconocido el formato de la URL y no se puede identificar a quién va dirigido.

```
Estado HTTP: 404
{
  status: "ERROR",
  reason: "Bad URL"
}
```

- **appToken inválido.** El appToken enviado no es válido o está vacío.

```
Estado HTTP: 451
{
  status: "ERROR",
  reason: "No valid appToken"
}
```

- **Se está usando un método HTTP no soportado.** No se está enviando una petición HTTP Post.

```
Estado HTTP: 405
{
  status: "ERROR",
  reason: "No valid HTTP method"
}
```

- **No es una notificación.** El atributo `messageType` no contiene la cadena `notification`, por lo que no es válido.

```
Estado HTTP: 452
{
  status: "ERROR",
  reason: "Not messageType=notification"
}
```

- **La notificación no está firmada.** La notificación no tiene una firma para poder verificar el origen.

```
Estado HTTP: 453
{
  status: "ERROR",
  reason: "Not signed"
}
```

- **Identificador no válido.** El identificador, que tiene que ser una cadena no nula, es inválido.

```
Estado HTTP: 454
{
  status: "ERROR",
  reason: "Bad id"
```

```
}
```

- **La notificación es demasiado grande.** Supera el límite del servidor de notificaciones, que está en 4KiB ó 4096 bytes.

```
Estado HTTP: 413
{
  status: "ERROR",
  reason: "Body too big"
}
```

- **Firma incorrecta.** La notificación está firmada, pero no es correctamente válida, esto es, la comprobación de la firma con la clave pública con la que la aplicación se registró ha devuelto que el mensaje ha sido modificado o es inválido.

```
Estado HTTP: 455
{
  status: "ERROR",
  reason: "Bad signature, dropping notification"
}
```

A partir de estos errores, el servidor de terceros es capaz de determinar que problema tiene el servidor de notificaciones con las notificaciones que está intentando enviar.

> **Nota**
> Los mensajes de error pueden cambiar de un servidor a otro, sobre todo si se añaden extensiones o más características, pero estos son los básicos que debería responder en caso de algún problema.

Así pues, las secciones anteriores son las APIs externas con las que tendrán que trabajar los desarrolladores de aplicaciones y gestores de servidores que intentan enviar notificaciones push.

Internas

Estas APIs y comunicaciones son las que se producen internamente, de forma transparente a los desarrolladores de las aplicaciones y del servidor que envía la notificación. Así pues, tiene que ser tratada por los desarrolladores que quieran implementar el protocolo en su plataforma, o sistema operativo.

Hay dos canales de comunicación diferentes, que son:

Entre el UA y el NS

Esta comunicación puede definirse como el estándar que tiene que seguir tanto el servidor de notificaciones como el sistema operativo o User Agent a la hora de implementar el protocolo para que puedan hablar entre ellos de forma correcta.

Está divida en dos partes principales, una interfaz HTTP, para recoger el token, y otra que usa el protocolo WebSockets, para el intercambio bidireccional de datos, así pues:

Interfaz HTTP

API HTTP: A través de este API, se puede descargar el token inicial y único que será usado para el registro posterior y demás operaciones.

> **Nota**
> Este API puede cambiar en un futuro y entre diferentes servidores de notificaciones, puesto que el token podría venir predeterminado en el mismo dispositivo o ser recogido mediante

Así pues, la petición actual es:

```
Host: https://push.telefonica.com/
GET /token
```

Y la respuesta sería:

```
Status: 200
Body: a4b26ecbd961c673f3526a8cc747758...
```

Donde todo el cuerpo de la respuesta (`res.body`) sería el token de dispositivo que es el que se tendría que usar para realizar los registros de dispositivo y de aplicaciones.

Interfaz WebSocket

API WebSockets: Este es el API bidireccional para el intercambio de datos, registros y notificaciones en algunos casos. Es el más complejo de todos ya que están involucrados muchos mensajes diferentes, pero todos son correctamente formados como JSON.

En primer lugar, es necesario abrir una conexión WebSockets con el servidor de notificaciones (en concreto con la instancia `NS_UA_WS`) para poder empezar a intercambiar datos.

Conexión WebSocket

Así pues, hay que utilizar el recurso necesario para iniciar la conexión, del estilo de `wss://localhost:8080`, usando el protocolo `push-notification`.

> **Nota**
> A partir de este momento, los siguientes métodos y comunicaciones se realizan por el canal WebSocket que se ha abierto en la sección anterior.

Registro de UA

Es el primer paso que hay que realizar para empezar a recibir notificaciones. Este paso significa el registro del dispositivo en el servidor de notificaciones, y la manera de recibirlas. Se realiza enviando un mensaje al servidor por la conexión WebSocket de la manera:

```
{
  messageType: "registerUA",
  data: {
    uatoken: "<a valid UAToken>",
    interface: {
      ip: "<current device IP address>",
      port: "<TCP or UDP port in which the device is waiting for wake up notificat
    },
    mobilenetwork: {
      mcc: "<Mobile Country Code>",
      mnc: "<Mobile Network Code>"
    }
  }
}
```

Donde los diferentes atributos tienen que ser:

- **messageType.** Tiene que ser siempre `registerUA`.
- **data.** Otros datos, compuestos por uatoken, interface y mobilenetwork.
  - **uatoken.** Es el token de dispositivo conseguido en las secciones anteriores.
  - **interface.** Contiene la IP y puerto en el que el dispositivo puede ser despertado. Por ahora, sólo se permite la escucha en puertos TCP o UDP.
    - **ip.** Dirección del dispositivo, ya sea con la IP privada o pública. En el caso de que sea pública, no hace falta que se rellene el dato de `mobilenetwork` y en el caso de privada, sí es necesario, puesto que se usará un proxy intermedio colocado dentro de la red móvil para poder encontrarle directamente.
    - **port.** Puerto de escucha, puede ser TCP o UDP. No necesitará saber qué le llega, si no simplemente que algo ha llegado ahí.
  - **mobilenetwork.** Indica los dos valores necesarios para identificar una red móvil correctamente, `mcc` y `mnc`. Las listas completas pueden consultarse en Internet[3]
    - **mcc.** Mobile Country Code.
    - **mnc.** Mobile Network Code.

[3] http://en.wikipedia.org/wiki/Mobile_country_code

Así pues, la respuesta del registro puede ser válida o no.

> **Nota**
> Hay que notar que las respuestas son asíncronas, y el envío de un mensaje de tipo `registerUA` no significa que su mensaje de respuesta sea el siguiente que llegue, por lo que hay que tratar cada mensaje por separado sin esperar una respuesta instantánea.

En el caso de una respuesta exitosa, se recibirá algo similar a:

```
{
  "messageType": "registerUA",
  "status": "REGISTERED",
  "pushMode": "WS",
  "WATokens": [],
  "messages": []
}
```

Donde se pueden encontrar varios atributos diferentes, los cuales son:

- **messageType.** Que tiene que ser igual a `registerUA`, ya que es la respuesta al tipo de mensaje enviado en un inicio.
- **status.** Si el valor es `REGISTERED`, es que el registro ha sido correcto.
- **pushMode.** Indica cuál es el modo en que las notificaciones van a llegar al dispositivo. Hay varios modos, los cuales pueden ser
  - **WS.** WebSocket. La conexión debe mantenerse abierta para poder recibir las notificaciones.
  - **UDP.** La conexión puede ser cerrada, ya que el dispositivo va a ser informado de que tiene nuevas notificaciones usando el canal UDP y el puerto abierto en el dispositivo. Últil en redes móviles.
  - **wapPush.** Utilizará WAP Push para informar al dispositivo de que hay nuevas notificaciones. No implementado en la primera versión

    > **Nota**
    > No están contempladas todas las posibilidades, ya que se puede añadir más en el futuro, dependiendo de los dispositivos que estén funcionando.

- **WATokens.** Array que indica todas las aplicaciones que están registradas en el dispositivo. Se mandan las URLs completas, por lo que se puede comprobar si falta alguna. En el caso de que el dispositivo tenga más de las recibidas, puede re-registrar la aplicación determinada o bien eliminarla, según el caso. Si no ha recibido una aplicación que debería estar registrada, simplemente deberá registrarla.
- **messages.** Array con las diferentes notificaciones que están pendientes de recibir por el dispositivo. Por cada una de ellas, y si el procesado es

correcto, se debe enviar un mensaje de confirmación, o ACK para indicar que se ha recibido correctamente y no enviar duplicados en un futuro.

En el caso de una respuesta incorrecta, la respuesta será[4]

```
{
  "messageType": "registerUA",
  "status": "ERROR",
  "reason": "<razón>"
}
```

Donde la razón `<razón>` puede ser:

- **Data received is not a valid JSON package.** El mensaje enviado no es válido.
- **UAtoken not valid for this server. Get a new one.** El token no es válido para el servidor actual, lo que quiere decir que se rechazará hasta que no se consiga uno nuevo.
- **Failed registering UAtoken.** Hubo un fallo interno en el servidor al registrar el dispositivo.

> **Nota**
> Esta función de registro tiene que realizarse cada vez que cambie la conexión móvil o WiFi del dispositivo, para actualizar la información en el servidor.

> **Nota**
> A partir de este momento, los siguientes métodos y comunicaciones necesitan tener una conexión autorizada, esto es, que haya registrado anteriormente un dispositivo mediante el método `registerUA`

Registro de WA

Esta función es la usada para el registro de aplicaciones a partir de una clave pública y un token de aplicación. Es la capa inferior de la función `requestRemotePermission` del objeto `navigator.push` expuesto para los desarrolladores de aplicaciones.

El formato es como sigue:

```
{
  "data":{
    "watoken": "hola",
    "pbkbase64": "hola"
  },
  "messageType":"registerWA"
}
```

[4] https://github.com/telefonicaid/notification_server/blob/develop/src/common/constants.js#L39

Donde los diferentes atributos se definen como:

- **data.** Contiene los datos necesarios para realizar el registro, los cuales son:
  - **watoken.** Es el token de aplicación que hay que registrar.
  - **pbkbase64.** Es la clave pública, codificada en formato Base64 para evitar problemas con los saltos de línea u otro tipo de caracteres o símbolos extraños, que corresponde a la aplicación que ha solicitado el registro.
- **messageType.** Tiene que ser igual a la cadena `registerWA`.

Con la posible respuesta, por el mismo canal y de forma asíncrona que puede ser:

```
{
  "watoken": "<watoken>",
  "messageType": "registerWA",
  "status": "REGISTERED",
  "url": "https://push.telefonica.com/notify/5819b005..."
}
```

Que se corresponden con los siguientes atributos:

- **watoken.** Es el token que ha generado la respuesta actual. De nuevo, comentar que las respuestas son asíncronas, por lo que tiene que ir reflejado a qué petición inicial iba dirigida.
- **messageType.** Tiene que ser igual al mensaje original, por lo tanto `registerWA`. Permite tratar al mensaje correctamente en el código.
- **status.** Tiene que ser igual a la cadena `REGISTERED`. En caso contrario, o que venga el texto `ERROR`, el registro no ha sido correcto.
- **url.** URL o recurso con la dirección única de notificación. Tiene que enviarse esta URL al objeto que ha originado la petición desde la aplicación.

También hay unos códigos de error similares a los del método `registerUA` respondiendo con un mensaje del estilo:

```
{
  "messageType": "registerUA",
  "status": "ERROR",
  "reason": "<razón>"
}
```

Con el atributo `<razón>` dando una explicación del error en forma humana, con un claro mensaje descriptivo.

Recepción de notificación

Confirmación de recepción, o ACK

De-registro de WA

De-registro de UA

Ping-Pong (o keepalives)

Internas

Estas APIs y comunicaciones son las que se producen internamente, de forma transparente a los desarrolladores de las aplicaciones y del servidor que envía la notificación. Así pues, tiene que ser tratada por los desarrolladores que quieran implementar el protocolo en su plataforma, o sistema operativo.

The UA-NS API is divided in two transport protocols:

- POST API: Through the HTTP POST transport protocol the NS will deliver valid UATokens to the device.
- WebSocket API: This is the most important one since all the communications (except to recover tokens) with the NS SHALL be driven through this API.
  On future releases will be supported another channels as Long-Polling solutions in order to cover devices which don't support Web Sockets.

HTTP POST API

This channel only offers one method to get a valid UAToken.

GET UA TOKEN

This method SHOULD be protected to avoid DoS attacks getting millions of valid tokens, in any case, this is out of the scope of this protocol.

The TOKEN method is called with a simple URL: `https://<notification_server_base_URL>/token`

The server will respond with an AES encrypted valid token. This token SHALL be used to identify the device in future connections.

WebSocket API

Through this channel the device will register itself, his applications, and also will be used to deliver PUSH notifications

All methods sent through this channel will have the same JSON structure:

```
{
  messageType: "<type of message>",
  ... other data ...
}
```

In which messageType defines one of these commands:

registerUA

With this method the device is able to register itself.

When a device is registering to a notification server, it SHALL send his own valid UAToken and also the device can send additional information that can be used to optimize the way the messages will be delivered to this device.

```
{
  messageType: "registerUA",
  data: {
    uatoken: "<a valid UAToken>",
    interface: {
      ip: "<current device IP address>",
      port: "<TCP or UDP port in which the device is waiting for wake up notific
    },
    mobilenetwork: {
      mcc: "<Mobile Country Code>",
      mnc: "<Mobile Network Code>"
    }
  }
}
```

The interface and mobilenetwork optional data will be used by the server to identify if it has the required infrastructure into the user's mobile network in order to send wakeup messages to the IP and port indicated in the interface data so it's able to close the WebSocket channel to reduce signalling and battery consume.

The server response can be:

```
{
  status: "REGISTERED",
  statusCode: 200,
  messageType: "registerUA"
}
```

```
{
```

```
  status: "ERROR",
  statusCode: 40x,
  reason: "Data received is not a valid JSON package",
  messageType: "registerUA"
}
```

```
{
  status: "ERROR",
  statusCode: 40x,
  reason: "Token is not valid for this server",
  messageType: "registerUA"
}
```

```
{
  status: "ERROR",
  statusCode: 40x,
  reason: "...",
  messageType: "registerUA"
}
```

registerWA

This method is used to register installed applications on the device. This shall be send to the notification server after a valid UA registration.

Normally, this method will be used each time an application requires a new push notification URL (through the WA-UA API) or also each time the device is powered on and is re-registering previously registered applications.

The required data for application registration is the WAToken and the public key.

```
{
  messageType: "registerWA",
  data: {
    uatoken: "<a valid UAToken>",
    watoken: "<the WAToken>",
    pbkbase64: "<BASE64 coded public key>"
  }
}
```

The server response can be:

```
{
  status: "REGISTERED",
  statusCode: 200,
```

```
  url: "<publicURL required to send notifications>",
  messageType: "registerUA"
}
```

```
{
  status: "ERROR",
  statusCode: 40x,
  reason: "...",
  messageType: "registerWA"
}
```

The device service should redirect the received URL to the correct application.

getAllMessages

This method is used to retrieve all pending messages for one device.

This will be used each time the device is Waked Up, so it's polling pending messages.

```
{
  messageType: "getAllMessages",
  data: {
    uatoken: "<a valid UAToken>"
  }
}
```

The server response can be:

```
{
  messageType: "getAllMessages",
  [
    <Array of notifications with the same format
     as defined in the notification method>
  ]
}
```

notification

This message will be used by the server to inform about new notification to the device.

All recieved notification will have this structure:

```
{
  messageType: "notification",
```

```
  id: "<ID used by the Application Server>",
  message: "<Message payload>",
  timestamp: "<Since EPOCH Time>",
  priority: "<prio>",
  messageId: "<ID of the message>",
  url: "<publicURL which identifies the final application>"
}
```

ack

For each received notification through notification or getAllMessages, the server SHOULD be notified in order to free resources related to this notifications.

This message is used to acknoledge the message reception.

```
{
  messageType: "ack",
  messageId: "<ID of the received message>"
}
```

Capítulo 4. Notification Server Architecture

This chapter explains how is the server designed to be able to process millions of messages per second.

Technologies used

The server infrastructure had been build using high performance languages and also high performance database and message queuing systems.

MongoDB

RabbitMQ

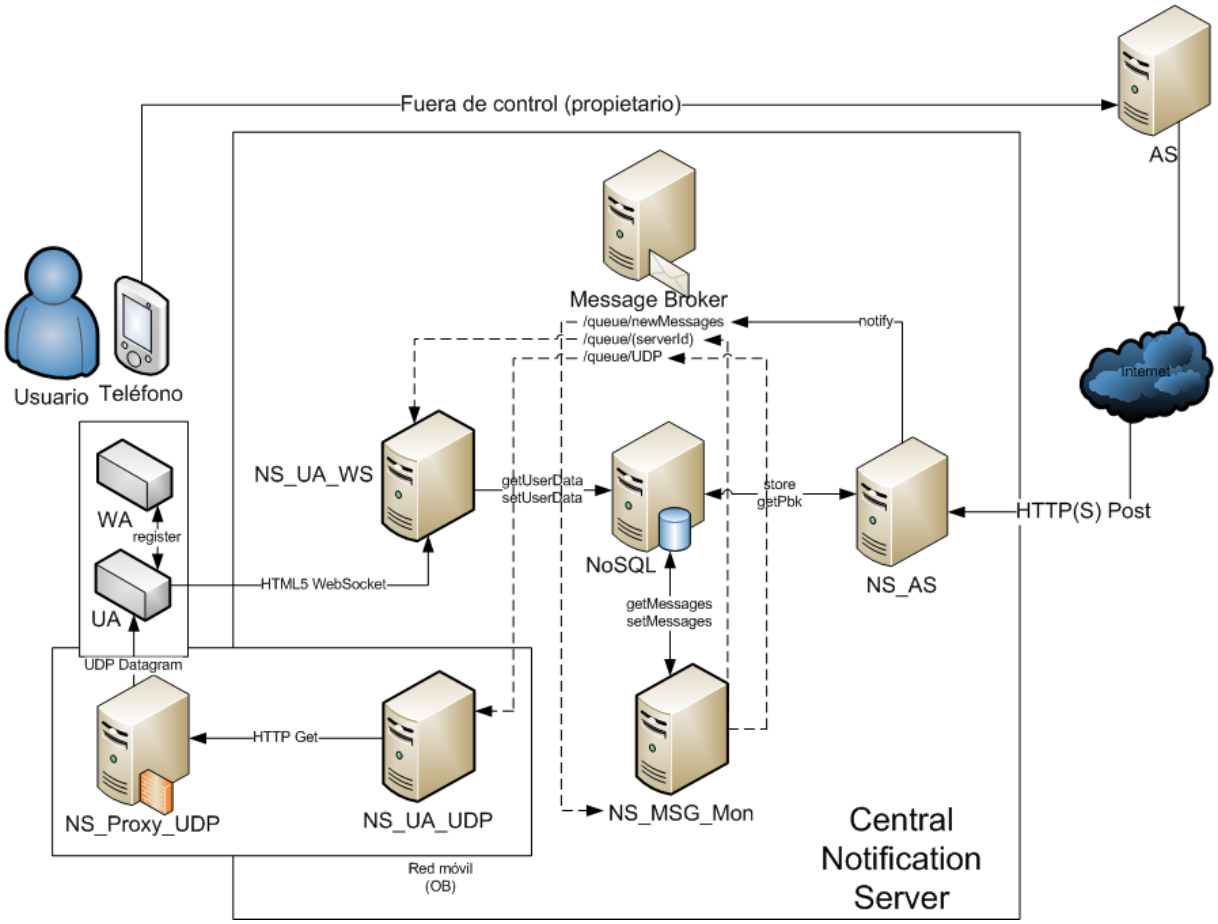
Node.js

Types of servers

In order to be able to scale horizontally and vertically with no limits all the server platform infrastructure had been splitted in several boxes one of them dedicated to a particular task and also independent of the rest so it can be scalled independently of the rest ones.

The names of each box follows this scheme: NS-<type_of_client>

Figura 4.1. Esquema general de la arquitectura



NS-UA-WS

The NS_UA_WS server is the frontend for mobile devices. This server will attend the clients using HTTP protocols (HTTP GET and WebSockets)

This server offers to channels:

- Retrieve a valid User Agent token used to identify each handset. This token will be delivered via HTTP GET method to the /token URL.
- Maintain a Websocket connection with the clients. This WebSocket will be maintained open in order to deliver push messages through it.

  This server will store on the MongoDB all registered nodes and applications. Also will receive from the Message Queue all the messages to be delivered to the connected handsets.

NS-UA-UDP

The NS_UA_UDP server is the responsible to intermediate between the central Notification Server infrastructure and each NS_WakeUp deployed in each OB.

As told before, this server will be connected to the message queue and for each received wake-up petition this server will retrieve from the MongoDB the NS_WakeUp server address and send a HTTP message to it querying to wake-up a handset.

NS-WakeUp

The NS_WakeUp server is a proxy between the central NS servers and the client equipment (device). This service will receive petitions through a standard HTTP port and will send UDP datagrams or TCP packets (for pinging purposes) inside the OB private network to the private IP of the client equipment. This server must be placed inside the OB private network or in a zone that must see that private IPs.

The Wake-Up Proxy server is responsible to ping to the correct client inside each OB (using UDP datagrams). It must be placed inside the OB or in a zone that can see the devices inside that private network.

This server will receive the ping orders through a standard HTTP port which will be connected to InterNodo network to receive the data from the VDC inside Telefónica network.

NS-AS

The NS_AS server is the frontend for application servers. This server will attend the third party servers through HTTPS POST petitions.

This server will expose a HTTPS POST in /notify method in order to receive messages from the third party application servers. The received messages will be stored on the MongoDB and will notify other servers through the Message Queue.

NS-Monitor

The NS_Monitor is the responsible to deliver messages to the correct recipient. So this server will be monitoring all inbound messages, deliver them and verify if re-deliver is needed.

The monitor reads the /newMessage queue (which frontends from the NS_AS puts all the received messages), and finds in the database which nodes need to be notified, sending the message to the correct queue the node (user agent) is subscribed to.

Message Queue (RabbitMQ)

A Message Queue cluster is used to act as a message dispatcher between all the other servers. RabbitMQ or ActiveMQ will be used.

This is a standard Message Queue which supports STOMP or AMQP protocol.

Because huge load, this server will be deployed in cluster mode.

NO-SQL Database (MongoDB)

A MongoDB cluster is needed to use as persistent storage system. It is used to save the registered devices, registered apps and received messages.

This is a non relational database.

Because huge load, this server will be deployed in cluster mode.

Capítulo 5. Tecnologías

Ahora, una vez hablado de cómo es la arquitectura y la primera implementación, hay que comentar sobre las tecnologías usadas y por qué se han escogido para la parse del servidor, comparando con otras soluciones similares.

Así pues, una de las grandes decisiones que hay que tomar al iniciar un proyecto de software son las tecnologías y lenguajes de programación que se van a utilizar, incluyendo a las herramientas, editores, gestores de bugs y de incidencias, método de comentarios en el código y un sin fin de características que harán derivar el proyecto hacia el éxito o el fracaso.

Node.js

Node.js[1] es un nuevo paradigma de programación para aplicaciones web en la parte del servidor, basado en JavaScript, muy rápido y con una serie de características muy importantes, como se verá más adelante.

> Node.js es una plataforma creada sobre el intérprete de JavaScript de Chrome para crear fácilmente aplicaciones de red rápidas y escalables. Node.js usa un sistema de eventos, un modelo de entrada-salida no bloqueante que hace que sea ligero y eficiente, perfecto para aplicaciones en tiempo real que requieran un uso intensivo de datos y que funcione a través de dispositivos distribuidos.
>
> —Node.js

Esta nueva plataforma surgió en el año 2009 por Ryan Dahl, y es patrocinada por la empresa Joyent, y su creación se debió a la necesidad de crear un sistema que pudiera trabajar correctamente con sitios web que aceptaran peticiones push para el intercambio de datos, por lo que la idea de esta tecnología era especialmente interesante para el proyecto del servidor de notificaciones.

Está compuesta de diferentes paquetes independientes, que son:

- **Motor JavaScript V8 de Google Chrome.** Un intérprete creado ad-hoc para Google Chrome, que revolucionó a la competencia por ser el más rápido y usar unas ideas muy novedosas e inteligentes. Genera código máquina para varias plataformas, como 32-bits, 64-bits, ARM o MIPS antes de ejecutar el código, en vez de directamente interpretarlo como hacían otros motores JavaScript. Además, utiliza una serie de técnicas antes de ejecutar que permiten optimizar (en varios pasos) y cachear resultados de partes del código que se usan mucho, como bucles o algunas funciones.

- **La capa de abstracción libUV.** Node.js se basó en la librería `libev` de abstracción de plataforma que estaba solamente disponible para Linux.

[1] http://nodejs.org/

Una vez que el desarrollo continuó y fue necesario utilizar otras plataformas, como Windows, se decidió crear `libUV`, que no es más que la evolución de `libev` para soportar otras plataformas y dar un API estándar para los desarrolladores de forma transparente al sistema que lo está ejecutando.

libUV contiene la mayoría de las características que luego definen a Node.js. Así pues, se implementan sockets TCP no bloqueantes, tuberías, el protocolo UDP, temporizadores, creación de procesos hijos e hilos, paso de mensajes vía IPC y sockets, varias funciones asíncronas como DNS o acceso a ficheros del disco…

Así pues, livuv obliga a un estilo de programación asíncrono y basado en eventos, y su trabajo principal es el de proveer un bucle de eventos y notificaciones basadas en callbacks para operaciones de entrada y salida y otras actividades.

- **Núcleo.** Escrito principalmente en JavaScript haciendo uso de las API e interfaces que provee V8 y otros módulos para un trabajo comunitario, es el código principal que después, mediante otras APIs se expone al desarrollador que va a utilizar Node.js.

  Así pues, podemos encontrarnos módulos como el de criptografía[2] (`crypto`) escritos enteramente en C, haciendo uso de las extensiones de V8, mientras que por ejemplo el que provee la interfaz para el manejo de sistemas de ficheros[3] (`fs`) está enteramente escrito en JavaScript, y su sintaxis es más fácil de entender y leer.

Para su diseño se tomó ideas prestadas de otros sistemas similares, como es Twisted para Python, libevent para C o EventMachine par Ruby. La mayoría de los programas en JavaScript se ejecutan en el contexto del navegador, puesto que siempre han sido los mayores exponenentes y consumidores de este lenguaje, sin embargo, Node.js es ejecutado en la parte del servidor, e incluso puede ser usado como lenguaje de scripting (sustituyendo incluso a Bash o Python) y posee un REPL, una consola interactiva.

Usuarios. Actualmente Node.js, a pesar de ser una tecnología muy reciente y joven, está siendo usada por muchas grandes empresas de internet para escalar sus sistemas. El hecho de que Yahoo! haya empezado a migrar algunos de sus sistemas desde PHP hacia Node.js, implementando diferenes frameworks y módulos mientras hacían la transición, hace pensar que el recorrido de esta plataforma es muy amplio y con muchas empresas grandes detrás interesadas en que siga siendo funcional, agregue más módulos y características y que sea líder en el mercado de los desarrollos de servicios de backend. Además, Microsoft, LinkedIn (para su aplicación móvil) o incluso Walmart también han decidido cambiar algunos de sus sistemas para hacer uso de Node.js.

[2] https://github.com/joyent/node/blob/master/src/node_crypto.cc
[3] https://github.com/joyent/node/blob/master/lib/fs.js

Para qué se usa

Node.js se utiliza en la parte del backend como la programación lógica del servidor de notificaciones. Esto quiere decir que es el que controla todo lo que está ocurriendo en el servicio, expone las APIs de terceros, mantiene las conexiones abiertas, hace intercambio de datos, guarda en la base de datos, monitoriza los mensajes por si hay que borrarles y un sin fín más de funcionalidades.

Así pues, es el componente principal del servicio, el que está en continuo contacto con el mensaje enviado, desde la entrada, incluyendo la verificación, hasta la entrega a los diferentes terminales a los que iba dirigido. Es el maestro de ceremonias que hace que todo funcione correctamente.

Por qué se ha elegido

Hay multitud de tecnologías para programar servicios web, pasando por cientos de diferentes lenguajes de programación que proveen miles de frameworks diferentes entre los que elegir. Sin embargo, una tecnología robusta, sencilla, con mucha comunidad detrás para buscar cualquier duda y tener un montón de módulos ha hecho que el proyecto se decantara por Node.js por las siguientes razones:

- Es asíncrono y basado en eventos, lo que permite mucha más granularidad y no bloquear la ejecución. Además, no se tienen que lanzar hilos por cada petición (o varias), si no que está todo en un bucle de eventos al que se van añadiendo acciones.

- Es JavaScript, y está alineado con la tecnología del proyecto que lo engloba, que es FirefoxOS.

- Es muy rápido de programar y se pueden hacer pruebas de concepto de forma rápida, además, que sea JavaScript ayuda a detectar de forma más fácil los errores, y que el tiempo desde hacer un cambio hasta verlo reflejado sea mínimo, sin tener que esperar a grandes tiempos de compilación.

- Despliegue muy sencillo y con multitud de opciones, ya que permite elegir si se quiere correr detrás de un balanceador de carga (como HAProxy), directamente conectado a la red y que sea Node.js el que responda…

Así pues, aunque hay otras elecciones posibles, como podría ser PHP, o incluso Python con pyev[4], el que se creyó que tendría menos uso de CPU y memoria y tendría más posibilidades de ser "hackeado" por la comunidad, fue Node.js.

[4] https://code.google.com/p/pyev/

RabbitMQ

RabbitMQ es una plataforma de mensajería para aplicaciones, fácil de usar, que funciona en la mayoría de los sistemas operativos, de código abierto y con multitud de wrappers para diferentes lenguajes de programación, como Node.JS, C++, Java, Python…

Una plataforma de mensajería permite a las aplicaciones conectarse y escalar de forma sencilla y rápida. Las aplicaciones pueden conectarse con otras aplicaciones, o bien diferentes componentes pueden ser independientes y constituir entre todos una gran aplicación o incluso poder enviar o recibir datos. La mensajería es asíncrona, por lo que se desacoplan las aplicaciones y se basa en eventos (de recepción, de envío).

Algunas de las grandes características de RabbitMQ son:

- **Confiabilidad.** RabbitMQ tiene una serie de funcionalidades que hace que la entrega de los mensajes pueda ser confiable, mediante la persistencia de mensajes (el mensaje no se guarda sólo en RAM, si no también en disco, recuperable en caso de que haya un error en el sistema), aceptaciones de entrega (puede ser automático o manual, por ejemplo, se quiere decir que un mensaje está recibido cuando se ha procesado correctamente), confirmación de publicación (si ha habido fallos al enviar mensajes a las colas) y alta escalabilidad.

- **Enrutado flexible.** Los mensajes son enrutados a través de intercambios antes de llegar a las colas, y RabbitMQ provee varias técnicas de enrutado, como hacer round-robin, el típico publicador-suscriptor o incluso se pueden mezclar técnicas simples con complejas para poder realizar cualquier enrutado imaginable.

- **Clustering.** Se pueden conectar varias instancias de RabbitMQ en una red local, haciendo un único y simple servidor.

- **Federación.** Es similar al modo cluster pero con mayor toleracia a fallos, usando redes que son más propensas a fallos, como en una red WAN, por ejemplo, teniendo las instancias en diferentes centros de datos distribuidos por el mundo.

- **Colas con alta disponibilidad.** Las colas pueden crearse en modo espejo en varias máquinas en un mismo cluster, por lo que si una máquina falla, la cola estará replicada en otra, y el fallo será transparente para los programas que usen este sistema de mensajería. Además, automáticamente se comprobará y se intentará unir de nuevo los nodos caídos.

- **Clientes múltiples.** Como se ha comentado más arriba, RabbitMQ ofrece muchos clientes para lenguajes de programación populares, ya sea para desarrollos de backend, como C++, Node.js, Python, Ruby o incluso Erlang, como para clientes frontend, como JavaScript en el navegador.

Para qué se usa

El uso principal en el servidor de notificaciones es la de pasar mensajes entre los diferentes actores que están en ejecución, de forma desacoplada y funcionando de forma independiente, esperando a eventos que desencadenen una reacción dependiendo de qué contengan y qué haya que realizar según el mensaje.

El uso más simple que se da es el de algunas intancias creando contenido, que son los que publican y envían los mensajes a las colas, y otros que son los consumidores, los que reciben los mensajes y realizan con ellos diferentes tareas. Así pues, los suscriptores están apuntados a determinadas colas o tópicos, esperando a que lleguen mensajes que alguien les mete por el otro extremo. Tenemos el caso de la cola `newMessages`, la cual es escrita por el servidor que provee la interfaz a los servidores de terceros, llamado Applicacion Server (`AS`), metiendo, bien los mensajes completos (cuando su tiempo de vida sea 0), o bien el identificador que él ha creado y ha guardado en la base de datos, para que el consumidor lo lea, y entienda que tiene que entregar los mensajes dependiendo de algunos parámetros que se pasan en dicho mensaje en la cola.

Por qué se ha elegido

En un primer momento el software de mensajería utilizado era ActiveMQ[5] ya que en el plan tecnológico se prefería el uso de esta tecnología, sin especificar las razones.

Sin embargo, dado el uso que se estaba dando en el servidor de notificaciones, en el cual no requeríamos que hubiera una persistencia de datos, que era uno de los puntos flojos de ActiveMQ, se empezó a usar el protocolo STOMP[6], que es el predeterminado de ActiveMQ.

El problema surgió cuando hubo que probar el despliegue, añadiendo pruebas de alta disponibilidad, tolerancia a fallos y durabilidad en el tiempo, momento en el que se descartó ActiveMQ debido a que era muy débil en cuando alguno de sus nodos fallaran, así que como el cluster no era como tal, si no que era solamente maestro-esclavo, lo que hacía que la escalabilidad fuera muy mala.

A partir de ese momento se buscaron otras soluciones de mensajería, viendo Apollo[7], que es una nueva implementación de ActiveMQ basada en eventos y mucho más eficaz que su hermano mayor, ZeroMQ[8], descartada por su gran curva de aprendizaje y su API a muy bajo nivel, y al final RabbitMQ, con el cual se hicieron más pruebas de carga, se decidió que sería la fundación final para los problemas de mensajería que necesitábamos, obligándonos a

[5] http://activemq.apache.org/
[6] http://stomp.github.com/
[7] http://activemq.apache.org/apollo/
[8] http://www.zeromq.org/

cambiar el protocolo de intercambio de datos, de STOMP a AMQP[9], nativo de RabbitMQ y estandarizado e implementado por la mayoría de los sistemas de colas (por lo que en un futuro se podría cambiar el software de mensajería sin que diera problemas el servidor de notificaciones).

Nota
Hay que destacar que debido a los problemas que se encontraron en ActiveMQ el plan tecnológico está siendo revisado y es muy probable que en un futuro cercano se prefiera la opción de RabbitMQ en vez de ActiveMQ.

MongoDB

Es una base de datos NoSQL escalable, orientada al almacén de documentos, con alto rendimiento, de código libre y escrita en C++. Es rápida y provee una flexibilidad que no dan las bases de datos relacionales, sobre todo en la manera como se guardan los datos, ya que no están ancladas a esquemas fijos que no pueden cambiarse.

Creado por la empresa 10gen[10] en octubre de 2007, utiliza JavaScript para las consultas (el API es accesible y se escribe como JavaScript), usando el motor de renderizado SpiderMonkey de Mozilla.

La forma de guardado de los datos es en formato BSON, o Binary JSON, por lo que es totalmente compatible con JavaScript y no hay que hacer un post-procesado de la información para utilizarla de forma agradable en el backend, ya que es un tipo de dato nativo más para JavaScript, que es el lenguaje de programación de la parte de backend, con Node.js.

Así pues, MongoDB es una base de datos muy rápida, muy flexible y con cientos de parámetros de configuración que hace que funcione desde una sola instancia hasta cientos de máquinas trabajando en paralelo en modo cluster, haciendo réplicas o muchas más maneras diferentes.

Algunas de sus características son:

- **Esquemas libres.** Que se guardan en un formato JSON, por lo que son fácilmente parseables por la mayoría de los lenguajes de programación. Además, permite realizar cambios al vuelo a los documentos, agregando o eliminando contenido sin tener problemas con si el esquema se repite o no, porque no hay tal.

- **Soporte completo de índices.** Aunque se guarden los documentos sin esquema, se pueden crear índices en aquellos campos que vayan a estar siempre. Por defecto, el índice genérico se realiza sobre el campo `_id` que es como la clave primaria para MongoDB. Sin embargo, se puede

[9] http://www.amqp.org/
[10] http://www.10gen.com/

indexar por cualquier campo, incluso haciéndoles combinados para buscar de forma más rápida.

- **Replicación y alta disponibilidad.** Una de las razones por las que MongoDB es muy usado es porque tiene multitud de opciones para replicar los datos entre diferentes servidores, ya sea en redes locales LAN o redes más grandes, WAN.

- **Sharding.** Más allá de poder replicar la base de datos entre diferentes instancias, se puede crear la opción de dividir la base de datos dependiendo de los datos de entrada, y dirigir las escrituras hacia un lado u otro. Esto es interesante por ejemplo si se necesita cumplir alguna ley que obligue que en un servicio mundial, los datos de los usuarios de algún determinado país esté en un centro de datos en concreto.

- **Modificaciones atómicas.** Algunas bases de datos NoSQL rompen las convenciones de las bases de datos SQL usadas desde hace tiempo, como la atomicidad (operaciones que o bien se ejecutan en su totalidad, o no se ejecutan), que podría llevar a pérdida de información e incongruencias. Así pues, MongoDB posee un motor que le permite realizar este tipo de opciones sin coste alguno.

- **Map-reduce.** El famoso algoritmo map-reduce, cada vez más utilizado para realizar operaciones sobre grandes conjuntos de datos, es un ciudadano de primer nivel en MongoDB, lo que permite realizar operaciones de agregación y procesado de datos de forma sencilla.

- **GridFS.** Permite guardar ficheros de cualquier tamaño en la base de datos, y recogerlos de forma sencilla, incluyendo replicación, sharding y todas las características que tiene MongoDB descritas más arriba.

Para qué se usa

Al ser una base de datos, se utiliza para guardar la información que envían los diferentes actores o instancias que están en ejecución en el servidor de notificaciones. Así por ejemplo, se guardan las notificaciones enviadas y correctas, los registros de los diferentes dispositivos con sus aplicaciones enlazadas y los posibles mensajes…

Por qué se ha elegido

Por proveer una flexibilidad mucho mayor que las bases de datos SQL tradicionales, y porque el formato de los mensajes, ya sean de los recibidos y los que se intercambian en las colas y de los datos entre las diferentes instancias tienen una característica en común: son JSON. Así pues, cualquier dato del sistema puede guardarse sin problemas en la base de datos, haciendo incluso de respaldo de los datos en caso de fallo del sistema.

Capítulo 6. Seguridad

La seguridad es uno de los puntos más importantes en cualquier sistema informático, y por supuesto no ha podido obviarse en este proyecto. Tener un sistema seguro es una tarea complicada, que requiere de mucho esfuerzo y dedicación, y sobre todo, de investigar cuáles pueden ser los fallos del sistema que pueden ser abusados. Aún así, cualquier sistema, por muy seguro que sea, siempre puede ser mejorado, y debe serlo, puesto que en el servidor de notificaciones estamos moviendo mensajes, que pueden ir en claro, e identificando quién los manda y hacia qué terminales, por lo que puede ser un grave problema si tenemos algún fallo de seguridad.

Así pues, ha habido diferentes elementos del sistema que ha habido que poner más énfasis en la seguridad, bien porque eran susceptibles de mostrar información personal, bien porque se creía que podrían recibir ataques por fuerza bruta o de denegación de servicio o incluso porque son aquellos que están expuestos tanto a los terminales como a Internet, y siempre son un dulce muy apetitoso para personas con no buenas intenciones.

Tokens de dispositivo

El primer punto de entrada al sistema de notificaciones push es tener un token de dispositivo válido que permitirá que el UA (User Agent, explicado en el capítulo "API") se registre de forma correcta en el sistema.

Entonces, había que garantizar de algún modo que este token fuera tanto único como válido en nuestro sistema para poder controlar el abuso, y si los registros de dispositivos venían de donde nosotros queríamos, esto es: de dispositivos con Firefox OS en un primer lugar.

A lo largo del tiempo, se han desarrollado muchos mecanismos de intercambio de información para realizar conexiones de usuario de forma controlada, o identificación de aplicaciones sin tener que dar los nombres de usuario y contraseñas a servicios de terceros, como OAuth[1]. Sin embargo, en caso de la primera versión del servidor de notificaiones, se quería algo más sencillo, lo que se tradujo en los siguientes preceptos:

- **Verificación sencilla.** No debería requerir el uso de mucho cómputo para comprobar si el token de usuario es válido o no.

- **Rápido de generar.** Generación al vuelo, por lo que tendría que hacerse de forma rápida, ya que el cálculo era la respuesta a una petición HTTP y no se podía dejar dicha conexión HTTP abierta, por el uso de recursos en el sistema.

[1] http://oauth.net/2/

Posibilidad de cambiarse en un futuro. Si se decide a usar un nuevo sistema en el futuro, como pudiera ser OAuth, Mozilla Persona[2] (con lo que encajaría más aún en el sistema Firefox OS) o incluso que el token viniera preconfigurado desde fábrica no debería suponer un cambio drástico en la infraestructura.

Así pues, la decisión en la implementación de generar un token[3], se quedó con el siguiente código:

Figura 6.1. Generación de token

```
function token() {}

token.prototype = {
  serialNumber: 1,

  // The TOKEN shall be unique
  get: function() {
    // SerialNumber + TimeStamp + NotificationServer_Id + CRC
    var rawToken = this.serialNumber++ + "#" + Date.now() +
                   "#" + process.serverId + "_" + uuid.v1();

    // CRC
    rawToken += "@" + crypto.hashMD5(rawToken);

    // Encrypt token with AES
    return crypto.encryptAES(rawToken, cryptokey);
  },
};
```

Así, en la figura anterior, podemos ver cómo está generado el token de dispositivo:

1. Usamos un número de serie que aumenta en una unidad cada vez que hay una nueva petición.
2. Añadimos el momento de generación actual, en milisegundos (según el estándar de JavaScript).
3. Añadimos el PID del proceso que lo ha generado.
4. Y un UUID (versión 1) para añadir aleatoriedad
5. Calculamos el hash MD5 del `rawToken` inicial, que se lo añadimos al token inicial con un `@` entre medias. Esto lo usamos como CRC.
6. Devolvemos el token generado cifrado con AES y la clave `cryptoKey` que está configurada en el fichero `config.js` que deben obligatoriamente compartir todos las instancias del servidor de notificaciones que generen to-

[2]https://support.mozilla.org/es/kb/que-es-y-como-funciona-browserid . Mozilla Persona es un sistema de identificación similar a OpenID, con la diferencia de que Persona usa direcciones de correo en vez de URLs, lo que resulta más natural para la identificación

[3]Se puede ver el módulo que realiza esta generación y verificación en: https://github.com/telefonicaid/notification_server/blob/develop/src/common/token.js

kens, pues es el principal medio para descifrar el token para realizar las operaciones y se usa para comprobar si es válido en el sistema.

Así pues, se puede observar que el token puede ser comprobado mediante la verificación con la función `verify(token)`, esto es, descifrando el token y comprobando su CRC. En primer lugar, si el descifrado no es correcto, el token recibido no es válido, por lo que podemos rechazarlo. Pero además, y en segundo lugar, podemos comprobar si el CRC (el hash MD5 calculado en la figura anterior) es válido, separando por el caracter `@` que habrá después de descifrar y comprobando si la segunda parte concuerda con la primera parte, habiéndole aplicado la misma función de hash.

Figura 6.2. Verificación de tokens

```
// Verify the given TOKEN
verify: function(token) {
  if(!token)
    return false;

  // Decrypt token
  var rawToken = crypto.decryptAES(token, cryptokey).split('@');

  // CRC Verification
  return (rawToken[1] == crypto.hashMD5(rawToken[0]));
}
```

Así pues, un posible ataque que podría intentarse es la suplantación de identidad, ya que sabiendo el token de dispositivo o `UAToken` es posible el registro de terceras partes como otro dispositivo. Sin embargo, esto no se considera un problema en el servidor de notificaciones y más en la parte cliente, aunque en todo caso, este token único tiene que estar gestionado por la seguridad que de el sistema operativo, o User Agent (`UA`) que implementa este protocolo.

Registro con clave pública-privada

Todas las aplicaciones que quieran recibir notificaciones tienen que registrarse con dos parámetros: el primero es un identificador único y el segundo una clave pública codificada en base64[4], que es la encargada de la verificación de las firmas de cada notificación. Esto se realiza desde la propia aplicación web, con el API que hay entre la WA y el UA, mediante la función:

Figura 6.3. Registro de WA en el UA

```
DOMRequest navigator.mozPush.requestURL(DOMString watoken, DOMString pbk)
```

Así pues, es obligatorio que haya un par de claves públicas-privadas (soportando sólo `RSA-SHA256` en la primera versión) para poder registrarse y posteriormente enviar y recibir correctamente notificaciones.

[4]RFC4648, página 5, punto 4, "Base 64 Encoding": http://tools.ietf.org/html/rfc4648#page-5

Notificaciones firmadas

Como se ha comentado en la sección anterior, en el registro es obligatorio contar con una clave pública-privada basada en RSA-SHA256. Esto no se hace simplemente por añadir un atributo más al registro, si no por proteger a los dispositivos que están dentro de la red móvil. Así, esta idea se ha implementado para permitir notificaciones sólo desde lugares a los que se haya aceptado mandar notificaciones, esto es, aquellos registrados con una clave pública que puede verificar la firma de la notificación enviada con la clave privada que sólo tendrán los servidores de la aplicación (servidor AS).

El funcionamiento es el mismo que el sistema PGP con el correo firmado, ya que del contenido (atributo `message` de la notificación JSON enviada por POST) se crea una firma con la llave privada (que sólo la tendrá el servidor que envía la notificación, o AS), que se manda al servidor de notificaciones (a una URL determinada, que es capaz de saber la clave pública que verifica dicha firma) y comprueba si es correcta o no. Si lo es, deja pasar la notificación al sistema y, eventualmente, se entregará. Si no es válida, por la razón que sea, la notificación es rechazada y ni siquiera entra al sistema.

Figura 6.4. Verificación de firma (en `src/common/crypto.js`)

```
verifySignature: function(data, signature, publicKey) {
  var algorithm = 'RSA-SHA256';
  var verifier = crypto.createVerify(algorithm);
  verifier.update(data);
  return verifier.verify(publicKey, signature, 'hex');
},
```

Figura 6.5. Comprobación de firma (en `src/ns_as/ns_as_server.js`)

```
//Get the PbK for the apptoken in the database
dataStore.getPbkApplication(apptoken, function(error, pbkbase64) {
  if (error) {
    return callback(errorcodesAS.BAD_MESSAGE_BAD_SIGNATURE);
  }
  var pbk = new Buffer(pbkbase64 || '', 'base64').toString('ascii');
  if (!crypto.verifySignature(normalizedNotification.message,
                              json.signature, pbk)) {
    log.debug('NS_AS::onNewPushMessage --> Rejected. Bad signature');
    return callback(errorcodesAS.BAD_MESSAGE_BAD_SIGNATURE);
  }
  ...
  // La firma es válida, continuar el flujo normal
  ...
```

Pero, ¿qué se consigue con esto? En primer lugar mantener acotados los servicios que pueden mandar notificaciones puesto que un desarrollador tiene control sobre la llave pública, puesto que es la que concuerda con su llave privada. Si pierde la llave privada, no podrá hacer nada, por lo que no podrá volver a mandar notificaciones. Pero además, si la recoge una persona

malintencionada y decide empezar a mandar notificaciones causando una riada de mensajes, puede rápidamente actualizar su aplicación, poniendo su nueva clave pública en sus ficheros de configuración y la privada en sus servidores, y todos los dispositivos con esta nueva instalación se registrarán de nuevo (los dispositivos comprueban entre actualizaciones de la aplicación si la llave pública de la aplicación ha cambiado, y si es así, se registran de nuevo) por lo que no recibirán mensajes desde la llave robada, si no desde la nueva, ya que el sistema ni las dejará pasar.

Y en segundo lugar, se puede hacer un control de abuso llegado el caso. Por ejemplo, si un servicio no hace más que enviar mensajes de broadcast hacia el interior, se puede revocar todas los registros de aplicaciones que tienen dicha clave y se protegerá de forma fácil al usuario.

Verificación de notificaciones

En cuanto llega una notificación, o dicho de otra manera, cualquier mensaje mediante una petición `POST` a la interfaz que expone el servidor de notificaciones hacia el exterior (en este caso se habla del `API` hacia el Application Server o `AS`), hay que comprobar una serie de criterios para verificar que la notificación es correcta.

En primer lugar, se comprueba que el mensaje enviado es un mensaje `JSON` bien formado, con la función `JSON.parse(notification)` nativa de JavaScript. Si es así, se pasa a comprobar el resto de campos, de la forma:

- Que el atributo `messageType` de la notificación es igual a la cadena `notification`.

- Que esté correctamente firmado, como se explica en la sección anterior. Esto significa que tenga el atributo `signature` y además sea válido.

- Que tenga un identificador externo, esto es, que el atributo `id` no puede estar sin definir (`undefined` al parsear el `JSON`) o ser nulo, puesto que en un futuro se podría agregar la opción de sobreescribir notificaciones y poder preguntar por el estado de la entrega de una notificación a través de este identificador único para el servicio de terceros o `AS`.

- Que el atributo `message` e `id` no superen los 4KiB. Esto es debido a que la longitud máxima que se permite en el servidor es de 4KiB. Probablemente en el futuro se cambie la restricción de tamaño para la parte de `id` y se obligue a un tamaño más pequeño, ya que 4096 bytes de identificación suficiente para cualquier aplicación.

A partir de este momento, la notificación, si es válida, pasa al sistema que lo entregará a los dispositivos a los que iba destinada, eliminando algunos campos, como `signature` (para evitar ataques replay), y lo dejará de la siguiente manera:

Figura 6.6. Notificación normalizada

```
{
  "messageType": "notification",
  "id": 1234,
  "message": "Hola",
  "ttl": 2592000,
  "timestamp": "1356102444115",
  "priority": 1
}
```

Ataque DDoS: flooding y replay

Una manera muy fácil de colapsar sistemas informáticos expuestos a Internet es mediante ataques que inunden las comunicaciones y acaben con los recursos de forma rápida, mediante miles de ordenadores haciendo peticiones de forma continuada y de una manera distribuida, siendo imposible casi discernir cuáles bloquear porque la mayoría son peticiones válidas. Es por esto que hay que tener en cuenta muchas variables en el diseño de sistemas, a la vez que hay que minimizar el riesgo de lo que se deja abierto a terceras partes, y siempre monitorizar todo lo que está pasando para intentar identificar comportamientos extraños que pudieran entrañar un problema al sistema.

Una de las medidas tomadas es la normalización de las notificaciones (ver figura precedente), como se comenta en la sección anterior, eliminando la firma, ya que sería extremadamente fácil hacer ataques de repetición (replay) hacia el sistema, inundando todas las instancias con mensajes iguales pero totalmente válidos, al estar firmados. Así pues, aunque la notificación pueda ser enviada de nuevo, al no tener las aplicaciones o un servicio de terceros la firma del mensaje (que está compuesta gracias a la clave privada que sólo lo tiene el primer AS), el mensaje no se valida y es rechazado.

Comunicación cifrada vía SSL

Una de las maneras más extendidas por Internet para intercambiar información de forma segura es el uso de certificados reconocidos por alguna de las CA (Certificate Authority) de confianza. A pesar de que este sistema es bastante frágil[5] y ha tenido numerosas[6] debilidades[7], sigue siendo el más usado y el que soportan la mayoría de los navegadores.

Es por eso, que todas las interfaces que expone el servidor de notificaciones hacia terceras partes, ya sean a los dispositivos (UA) o a los servidores de aplicación (AS) usan SSL (Secure Sockets Layer) para evitar que una posible persona malintencionada (normalmente conocido como Man-In-The-Middle

[5] BlackHat USA 2011: SSL And The Future Of Authenticity: http://www.youtube.com/watch?v=Z7Wl2FW2TcA
[6] Comodo creó certificados válidos para Google, Yahoo o Skype dirigidos a terceras partes: http://www.schneier.com/blog/archives/2011/03/comodo_group_is.html
[7] Detectado un certificado para dominios de Google emitido por una autoridad no competente: http://arstechnica.com/security/2013/01/turkish-government-agency-spoofed-google-certificate-accidentally/

o MITM) pueda interceptar la información y la descifre, con la obvia pérdida de privacidad.

Así pues, en la parte de los dispositivos se emplea dos métodos diferentes:

- **HTTPS.** La descarga del token inicial, en la primera versión del servidor, se realiza mediante un método `GET` a una dirección HTTP, por ejemplo `https://push.telefonica.com/token`. Esto permite que sólo los extremos de la conexión, en este caso el UA y el NS conozcan el contenido.

- **WSS (WebSockets Secure).** WebSockets también tiene una extensión segura para el intercambio de datos. Necesita un certificado SSL de la misma manera que una conexión HTTPS normal. De hecho, WebSockets se inicia mediante una petición HTTP, y después de hace un `Upgrade` para poder pasar el protocolo a WebSockets. Una vez negociada la conexión y actualizada a WebSockets, ya es posible transmitir información cifrada entre las dos partes. Por este canal se descargan las notificaciones, las llamadas de registro y confirmación de mensajes…

UDP para notificaciones

Uno de los hechos diferenciadores del servidor de notificaciones, como se ha ido comentando durante todo el desarrollo de este proyecto, es que no es necesario mantener un socket o una conexión abierta entre el teléfono (UA) y el propio servidor para ser notificado de que hay información pendiente.

Así pues, se ha decidido que el paquete que se envía desde el servidor de notificaciones (NS) hacia el dispositivo (UA) sea del tipo UDP, por las siguientes cuestiones:

1. El paquete UDP, no llevará ningún tipo de contenido. Y si lo lleva, será descartado. Esto es debido a que no es necesario que haya corrección de errores, ni aceptación (también llamados ACK), no importa el orden (de hecho, no hay orden implícito), y tampoco es necesario un control de congestión.

2. Utilizar TCP hubiera hecho que la señalización en la red sea mucho mayor, ya que para hacer un simple "ping" para que el dispositivo (UA) se entere de que tiene una nueva notificación, sería necesario tener control de aceptación, retransmisión, temporizadores, orden, control de congestión… y en general demasiado cargado para algo que no requiere ningún `payload` para ser leído.

Nota

Probablemente en el futuro se añada algún mecanismo de información de cambio de puerto, por ejemplo: como es fácil mandar un paquete UDP a los diferenes dispositivos para que se despierten a un puerto por defecto (como es el caso de la primera versión, que está

fijado en el `5000`), estos automáticamente se conectan al servidor de notificaciones para descargar los mensajes, pero que, al ser un ataque, no habría. La idea es que si esto sucede, el dispositivo se reregistre de nuevo en el servicio indicando otro puerto, para que haya menos posibilidades de inundar el servidor.

Entrega de notificaciones a las aplicaciones

Un problema de seguridad que podría ocurrir es que haya dos aplicaciones instaladas en el dispositivo del usuario, y que ambas tengan permiso para recibir notificaciones push, pero una de ellas sea una malintencionada que se quiera hacer pasar por la original.

Así pues, el dilema viene cuando una aplicación es la verdadera y la otra es una que la intente suplantar. Esto quiere decir que la aplicación web verdadera genera un token de aplicación (o `WAToken`) de una manera poco aletoria y que es fácilmente adivinable. En este caso, si alguien sabe cuál es el algoritmo de generación de dicho token, podría replicarlo y registrarse de nuevo en el sistema como la aplicación "original". La duda en este caso surge de saber cuál es la aplicación original, cuál es la copia e incluso cuál se ha registrado en primer lugar. Así pues, la decisión es la de mandar la notificación sólo a la primera aplicación que se haya registrado con un token determinado, y rechazar el registro de sucesivas aplicaciones con igual identificador.

Nota

Sin embargo, esto podría cambiar en un futuro, pero como se comenta, hay demasiadas variables para poder elegir la aplicación real, puesto que no hay manera de discernir ahora mismo cuál es, o si una aplicación que intente suplantar a otra se haya registrado antes que la real.

Capítulo 7. Lecciones aprendidas

Durante el concepto inicial, desarrollo e implementación del servidor de push, se encontraron muchas variables que han hecho cambiar ligeramente el rumbo tomado por el proyecto. Sin embargo, la idea general seguía presente y los cambios que se fueron realizando eran debidos a cosas que se creía que eran más interesantes de tener, como mensajes de deregistro, posibilidad de encolar mensajes y mandar sólo un mensaje final, añadir ACKs específicos de aplicación y no usar los propios de TCP…

SQL contra NoSQL

Una de las primeras decisiones tomadas en el proyecto fue la de usar una base de datos no relacional, también llamadas No-SQL. En este caso se eligió MongoDB debido a que era la seleccionada por el plan tecnológico de la empresa. La decisión entre elegir una base de datos No-SQL frente a una SQL convencional (como puede ser MySQL u Oracle) fue simplemente la de no tener un esquema estricto para cada uno de los datos que se iban a introducir, ya que se creía que el esquema cambiaría en el futuro según se fuera iterando (y así sucedió de hecho), y permitiría un prototipado de la plataforma mucho más rápido.

Sin embargo, el cambiar radicamente la mentalidad entre una solución SQL a algo No-SQL, muy relajado y sin un esquema fijo para todos los "elementos" de una misma "tabla" podría suponer un peligro, puesto que las nociones de SQL y muchas de las decisiones para estos sistemas se toman a partir de dichos conocimientos.

Así pues una de las primeras cosas fue "dejar de pensar de forma relacional", con todo lo que ello implica, como la pérdida de alguna de las propiedades ACID[1]. E incluso la noción de tabla y de esquema.

Mientras que en las bases de datos SQL hay un montón de restricciones, los esquemas son muy fijos y hay que respetar todas las normas ACID, además de intentar normalizar la base de datos para poder cumplir las cuatro normas básicas, en NoSQL depende mucho de cuál sea la base de datos seleccionada el qué hacer, básicamente porque cada una tiene sus características y rompen las propiedades ACID de diferentes maneras.

Así pues, algunas de las características de la base de datos elegida, MongoDB, hacen que las consultas sean mucho más rápidas cuando esté todo en el mismo sitio, lo que quiere decir que es mejor duplicar datos a hacer varias consultas a la vez. De hecho, si hay que hacer varias consultas, es mejor que no sean complejas, y que se encargan los diferentes backend de unir los datos, ya que así liberamos de tareas pesadas a los mismos gestores. Por ejemplo, hacer un `JOIN` en MongoDB es muy pesado, pero sin embargo

[1] ACID: Atomicity, Consistency, Isolation y Durability

es muy rápido si hacemos dos consultas por separado y las unimos en el código del programa.

Investigar nuevas tecnologías

Uno de los mayores retos del proyecto ha sido el uso de una tecnología nueva pero muy emergente y con una comunidad enorme detrás como es Node.js. Este lenguaje de programación permite hacer iteraciones muy rápidas, con muchos módulos para añadir características de una manera muy sencilla. Sin embargo, es JavaScript, lo que hace pensar a muchas personas que no es válido para desarrollo de backend, como sí lo podría ser PHP, Ruby o Python, expresamente diseñados para eso, ya que existe el concepto de que JavaScript sólo se ejecuta en el navegador de los usuarios, y su único cometido es el de manipular el DOM[2] de la página y poder hacer interacciones y cambios en tiempo real, incluso pudiendo hacer peticiones AJAX para agregar contenido de forma dinámica.

Sin embargo JavaScript es uno de los lenguajes más denostados y minusvalorados, ya que posee un montón de características que lo hacen especialmente rápido y, sobre todo, divertido de programar, amén de una velocidad de ejecución cada vez más rápida gracias a la competencia que se ha ido desarollando los últimos años entre los principales desarrolladores de navegadores de Internet, que al final son los principales valedores de crear intérpretes de JavaScript más complejos, rápidos y eficientes.

La idea detrás de Node.js es que sea asíncrono y esté basado en un bucle de eventos, por lo que toda la programación hay que dirigirla hacia allí. Además, hay que tener en cuenta que JavaScript no tiene hilos, pero tampoco es necesario por la implementación del lenguaje (usando callbacks y eventos). Así pues, el cambio de mentalidad también es significativo, ya que es poco común en la programación estructurada tipo C o C++ el uso de eventos (aunque obviamente hay librerías para poder usarlos), mientras que en JavaScript, y en concreto en Node.js, lo son todo.

Así pues, muchas veces el código queda poco ordenado, debido a que la programación no es lineal, si no que puede ocurrir que una entrada empiece por la primera función, pero que tenga que llamar a la tercera que a su vez llama a la continuación de la primera para devolver la ejecución a la inicial. Sin embargo, pasada la diversión final con callbacks y eventos, Node.js ha destacado como un lenguaje muy potente, versátil y sobre todo, rápido y eficiente.

Tener tests unitarios y funcionales

Crear test y pruebas es algo que debería estar obligado en todos los proyectos, pero que, por unas cuestiones u otras, no lo suele estar, y se suele llevar a un plano secundario, muchas veces con la documentación. Sin embargo,

[2] DOM: Document Object Model

es una de las piezas fundamentales en un sistema que esté en desarrollo. Dada la naturaleza de la informática, quizás cambios que se piensen que no afectan a otras partes, sí lo podrían hacer, y ahí es donde entran los tests.

Así pues, es fácil pensar en tener tests para prácticamente la totalidad del código y de las funciones que se van usando durante el programa. Simplemente es elegir la tecnología más fácil y sencilla para que dichos tests se ejecuten y que crearlos no sea algo que lleve más tiempo que el propio código que se quiere probar.

Así pues, en el caso del servidor de notificaciones, se ha empleado Node.js como lenguaje para crear los tests, tanto unitarios como funcionales, apoyándose en algunos casos de módulos específicamente diseñados para ello, como Vows, que permite ejecutar tests en paralelo, esperar a que unos terminen antes de que empiecen otros, y que se haga de forma muy sencilla y muy visual. Todo ello totalmente integrado en el fichero `Makefile` para ser ejecutado como `make tests` hace que la ejecución y comprobación de funcionalidades y de regresiones sea rápida y sencilla.

Priorización y enfoque en tareas

Uno de los principales enemigos de sacar adelante un proyecto es la falta de priorización de las tareas. Esto quiere decir que muchas veces lo más importante que hay que hacer para que un proyecto salga adelante no es lo mismo que lo que desarrolladores piensan. Quizas añadir una funcionalidad que va a usar el 5% de la gente y require el 20% del tiempo debido a su complejidad, podría priorizarse después de una que vaya a usar el 80% de la gente y que requiera ese 20% de tiempo.

Así pues, alguna de las funcionalidades que el servidor de notificaciones se han dejado para una futura versión 2 han sido fruto de estos acuerdos: cosas que son muy interesantes de tener, y que ayudarán de una gran manera a los desarrolladores que lo usen, pero que requieren demasiado esfuerzo para programarse que podría concentrarse en otras partes más fundamentales y que, en general, afectarán a más gente y harán incluso que esas nuevas features interesantes sean más fáciles y rápidas de implementar en un futuro.

Trabajar en abierto

Como se ha comentado anteriormente, la naturaleza de este proyecto era que fuera abierto y que pudiera ser usado por todo el mundo, a la vez que esperar nuevos colaboradores o personas que encontraran problemas y ayudaran en su arreglo hasta otras que les gustaría añadir nuevas funcionalidades y se encarguen de dichos nuevos escenarios.

Entonces, es sencillo entender que toda la tecnología, lenguajes de programación, compiladores, sistemas, programas sean abiertos. Pero es más, los sistemas donde se guarda el código fuente, los repositorios, también están

abiertos, en la plataforma GitHub, lo que da mucha más visibilidad y facilidad a terceras personas de acercarse y ayudar.

Como se ha comentado anteriormente, los tests es una parte fundamental del desarrollo de una aplicación o servicio, así pues, aprovechando la excelente integración entre GitHub (el servicio donde se sube el código fuente) y el sistema de integración continua TravisCI, se decidió usarlos en cojunción para que cada vez que alguien suba un cambio en el código, automáticamente se genere una compilación y se lancen los tests, enviando correos y avisos a las personas que están configuradas en caso de que algo haya ido mal o se encuentren regresiones.

La importancia del hardware

Siempre se piensa que cuanto más capacidad y más rápido mejor, pero en el caso de sistemas reales también hay que tener en cuenta la ubicación de los servidores, la conectividad, la latencia y el presupuesto.

Lo más lógico es ver las diferentes necesidades de cada parte del sistema, y hacer pruebas de carga en diferentes escenarios: red congestionada, red disponible al 100%, red con fallos intermitentes, discos duros con poca velocidad, con mucha velocidad, poca RAM, mucha RAM…

Además, también es necesario probar de forma incesante todas las instancias. En el caso del servidor de notificaciones, se detectó que Node.js sólo permite que se ocupe 1.4GiB de memoria por cada proceso, por lo que sería necesario lanzar varios procesos a la vez del mismo servidor o usar otros sistemas para poder balancear la carga, como usando el módulo `cluster`[3] disponible de forma nativa.

En el caso de MongoDB, las pruebas indicaron que el rendimiento de la base de datos mejoraba de una forma drástica si las instancias tenían poca memoria RAM pero un disco duro SSD, cuyo rendimiento es unas 10 veces superior a los discos duros magnéticos tradicionales.

[3] http://nodejs.org/api/cluster.html

Capítulo 8. Futuro: v2 y siguientes

Este capítulo está dedicado a todas aquellas ideas que se han quedado por el camino o que no son tan importantes para implementar en la primera versión del servidor de notificaciones, pero que podrían ser interesantes para futuras versiones o que se han pedido por otros proyectos (como pueden ser de mensajería instantánea).

Así pues, algunos de ellos tienen más prioridad que otros, algunos pueden tener más capacidad real de ser usados que otros, pero es posible que muchos de ellos lleguen a implementarse finalmente, y formen parte de la especificación estandarizada por el W3C.

Sobreescribir notificaciones

Una de las características más interesantes que permiten otros servidores de notificaciones es la posibilidad de sobreescribir notificaciones enviadas previamente en el tiempo pero que además no hayan sido entregadas, por lo que podría añadir más información, o bien cancelarse.

Ejemplo 8.1. Notificación sobre el desarrollo de un partido de fútbol

En el caso de una aplicación de resultados de fútbol, podría ser que cada notificación tuviera un estándar definido para indicar los goles, cambios de jugadores, tarjetas y demás incidencias. Podría ser que en un momento determinado, uno de los equipos marque un gol, y se requiera avisar a todos los usuarios para que la información se actualice. Sin embargo, se podría pensar en enviar un mensaje push con sólo la información determinada, pero, ¿y si alguien no ha recibido los anteriores? Requiere mucha más lógica por parte de la aplicación el recibir todos los mensajes, ordenarlos y pintarlos de una forma cronológica, cuando el orden de entrega no está garantizado.

Así entonces, la idea podría ser la de que el mensaje tuviera un identificador fijo para el partido, por ejemplo `valencia-madrid-liga-20130120` y que cada vez que hubiera un cambio que haya que enviar a los usuarios se añadiera más contenido, sobreescribiendo a los dispositivos que aún no hayan recibido la notificación con esta nueva (en vez de tener `N` mensajes, habría `1` mensaje con `N` eventos), y siempre tendrían lo último, independientemente del número de eventos que haya habido antes. En el caso de dispositivos que estén recibiendo notificaciones que ahora se sobreescriben, simplemente sería implementar la lógica de descartar la información anterior y pintar todo de nuevo, o bien hacer caso omiso de lo antiguo y utilizar sólo los nuevos eventos.

Recepción segura de token de usuario

Actualmente la recepción del token de dispositivo/usuario que se utiliza como primer paso para registrar a alguien en el servidor de notificaciones no es

demasiado segura, ya que a todo el mundo que pide un token, se le asigna uno válido. Esto debería cambiar en un futuro e integrarse con otros sistemas de autenticación de usuarios, para dar tokens sólo a los usuarios autorizados.

Así pues, podría ser que en un futuro este token pudiera venir de forma preinstalada en el propio dispositivo, que tenga que utilizarse una tecnología como OAuth2 para recoger el token o que incluso haya que identificarse como comprador del terminal y asignarlo a la cuenta de usuario para poder recibir notificaciones.

Sin embargo, este token depende mucho del entorno en el que se esté ejecutando el servidor de notificaciones, ya que este token no sale nunca del dispositivo, y podría asignarse de la mejor manera que el proveedor del servicio disponga, ya que la parte de cómo recibir el token no está estandarizada, ya que se supone que ha habido un intercambio seguro de información para recibirlo y el token que tiene el usuario/dispositivo es válido para todo el sistema.

Cambio en el tamaño máximo

En un primer momento, y como se comenta más atrás en el proyecto, el tamaño máximo de mensaje está restringido a simplemente 4KiB de información. Esta elección está hecha porque es la estándar de Google Cloud Messaging (GCM), y supone un tamaño aceptable para enviar datos a cualquier aplicación.

Sin embargo, esto podría cambiar en un futuro dependiendo del uso que se esté dando a la plataforma: si los desarrolladores prefieren mandar mensajes pequeños o grandes, si mandan el contenido cifrado, si existen otras características que aún no se están dando en las aplicaciones o en Internet…

Poder enviar números de versión

Este es una de las características fundacionales de Thialfi, el sistema definido por Google y que se usa en Google Chrome para la sincronización y los contactos. Sin embargo, el sistema actual del servidor de notificaciones tiene la opción de mandar números de versión, no de forma implícita, si no poniéndolo de forma explícita en el payload del mensaje enviado.

Así entonces en un futuro podría haber estas notificaciones:

```
{
  messageType: "notification",
  id: "<ID interno para el AS>",
  version: "<Número de versión>",
  signature: "<Firma>",
  ttl: "<Tiempo de vida>",
  timestamp: "<Desde EPOCH>",
  priority: "<Prioridad 1-4>"
}
```

Nota
Como recordatorio, el formato actual es:

```
{
  messageType: "notification",
  id: "<ID interno para el AS>",
  payload: "<Número de versión>",
  signature: "<Firma>",
  ttl: "<Tiempo de vida>",
  timestamp: "<Desde EPOCH>",
  priority: "<Prioridad 1-4>"
}
```

Como se puede comprobar viendo ambos mensajes, el primero es un subconjunto del segundo, ya que con el método actual, el servidor es capaz de enviar números de versión a las aplicaciones, construidas de tal manera que el mensaje que reciben no contiene un `payload` real, si no que es una versión que tiene que estar sincronizada con el servidor de la aplicación, y que se encargará de ir a realizar una segunda conexión para traer los datos reales, ya manejables por la aplicación.

Ejemplo 8.2. Uso de números de versión en aplicación del banco

Un uso real de números de versión podría ser la aplicación del banco, que quiere enviar que hay nuevos movimientos en la cuenta corriente, pero no desea enviar ningún dato personal a través de la red no gestionada por ellos, por lo que podría mandar un número de versión para que la propia aplicación se conecte con los servidores del banco, sin dejar ningún resto de información personal en el servidor de notificaciones, y descargue la información requerida para estar actualizado.

Implementar soporte para IPv6

El soporte de IPv6 debería ser una de las prioridades de la segunda versión del servidor, básicamente porque su uso es cada vez más extendido (más que por ser algo voluntario, por ser algo obligatorio), y puede tener características muy gratificantes para el servidor y de la manera que se envían las notificaciones.

La idea sobre IPv6 es la de aumentar el número de direcciones posibles para que cualquier dispositivo tenga una IP diferente a cualquier otro y por lo tanto sea único en la red, haciendo que pueda ser directamente accesible, sin tener que pasar por proxies intermedios que redirigan la petición hacia otras direcciones.

Esto es un gran avance que podría acabar directamente con la funcionalidad del servidor de notificaciones, ya que, al estar el dispositivo directamente en Internet, con una IP pública, y siendo accesible desde cualquier lado, no hay necesidad de un proxy intermedio (como es el servidor de notificaciones) o

de mantener conexiones abiertas (como se hace en los otros sistemas) ya que los servicios que quieran enviar algo al teléfono lo podrían hacer directamente.

Sin embargo, hay un problema, y es que los proveedores de telefonía móvil pueden decidir varias cosas para que sus dispositivos no sean directamente accesibles por Internet:

- **Cortafuegos.** Para evitar que los dispositivos que están realmente conectados en la red se protejan ante posibles ataques, podrían implementarse sistemas de cortafuegos o firewall que permitirían cortar algunos tipos de conexiones que puedan ser perjudiciales para los dispositivos, como peticiones de PING, posibles ataques DDoS, que haría que la radio móvil esté siempre en el estado de máxima excitación y la batería de los terminales se gaste de forma rápida.

- **IP Privadas.** Aunque IPv6 provea miles de millones de diferentes direcciones y se pueda identificar a cada dispositivo por una IP diferente, las operadoras podrían decidir seguir creando redes privadas como ocurre ahora mismo en las redes IPv4, para proteger de la misma manera el acceso desde internet hacia los teléfonos de forma incontrolada. Sin embargo, cumpliría lo mismo que haría el cortafuegos propuesto anteriormente, pero además, al estar en un segmento de red que no se puede llegar por ser privado, tendría el problema de que estos dispositivos no podrían ser contactados directamente, por lo que necesitarían un proxy intermedio para poder hacer las peticiones.

Así pues, aunque IPv6 podría pensarse que podría matar al servidor de notificaciones, como se ha comentado anteriormente, es muy probable que no lo haga, ya que en ambos casos necesitaríamos reglas especiales en los firewall para permitir que sólo el rango de direcciones de funcionamiento del servidor de notificaciones pueda acceder a los dispositivos o bien tener una de las instancias dentro de la red móvil, como ocurre con el despliegue en IPv4, donde es necesario que haya algo dentro de la red móvil para poder hacer `PING` directamente a los terminales sin tener una conexión abierta y que sean capaces de responder a notificaciones.

Implementar prioridades

Esto podría ser una de las grandes nuevas funcionalidades en la segunda versión del servidor de notificaciones, ya que permite a los desarrolladores no molestar a los usuarios constantemente y esperar a que estén mirando el dispositivo para poder mostrar las notificaciones. GCM tiene un sistema similar, con la propiedad `delay_while_idle` en el JSON enviado, que permite que no se entregue hasta que el dispositivo esté de nuevo conectado a la red y el usuario realizando algún tipo de transferencia de datos.

Sin embargo, tener sólo dos estados quizás es demasiado poco para los desarrolladores, porque se podrían necesitar otros valores para diferentes

prioridades, encolarlas y poderlas enviar de forma instantánea o bien esperar a que haya varias, no teniendo que ser todas del mismo origen, para enviar a la vez.

Ejemplo 8.3. Información sobre lugares cercanos y ofertas

Un uso interesante podría ser la posibilidad de tener una aplicación que informe a un servidor de cuál es la ubicación del dispositivo y quiera enviar ofertas de lugares cercanos a donde se encuentre, como bares, restaurantes o conciertos de música. Sin embargo, el recibir anuncios constantemente podría ser muy molesto para el usuario (incluso podría llegar a desinstalar la aplicación), por lo que se podría indicar al servidor cuál es la prioridad del mensaje: esto es, si se obliga a que se mande la notificación sea cual sea el estado del dispositivo, o que se pueda encolar y esperar hasta que el usuario esté de nuevo activo (por ejemplo, con la pantalla encendida o navegando por Internet) o se reciba una notificación de una prioridad superior, momento en que se enviarían todas.

PINGs de backup

Aunque la idea de este servidor de notificaciones es que no haga falta tener un canal abierto, constante, vacío, a la espera de hacer `PING` de vez en cuando para comprobar si hay datos, sí podría ser interesante en determinadas conexiones, como la de WebSockets.

La idea es tener este mecanismo de `PING` que se realice cada cierto tiempo (por definir, puede ser 10 minutos o media hora) y que aprovechando el `PONG` del servidor, pueda enviar la lista de registros de aplicaciones y los posibles mensajes, en el caso de que haya habido un fallo y no se haya podido entregar.

Sin embargo, esta es una de las tareas menos prioritarias porque se estaría volviendo a los sistemas anteriores que requerirían mantener un canal abierto para preguntar de vez en cuando, el famoso polling, o bien abrir un canal para exclusivamente decir que se está conectado.

Soporte WAP push

El soporte UDP que se da en el servidor es nativo de la red: se usa toda la infraestructura montada para despertar al teléfono, que reciba el paquete, vea que es un `PING` de notificación, se conecte al servidor y descargue los datos.

Pero puede mejorarse con la introducción de WAP push, aunque sólo sería en casos muy excepcionales, porque ya se ha visto cómo funciona esta tecnología, abriendo el navegador del usuario para descargar los datos, algo que haría que se sintiera extraño, a menos que se pudiera hacer de fondo y se pudiera pasar a la aplicación en concreto.

La prioridad es baja por los problemas que tiene este sistema como se comenta en el párrafo anterior y porque es un servicio de pago para los desarrolladores o servicios que quieran mandar notificaciones push, además de que los usuarios tengan que dar su número de teléfono para poder hacerlo funcionar.

Control de abuso

Actualmente no hay ningún control de abuso para la primera versión del servidor. Se controlan los orígenes frente a una base de datos que se puede ir rellenando, pero no hay nada automatizado que lidie con todo el proceso.

Lo que se quiere para la segunda versión es la posibilidad de que esto sea automático, para que cuando el servidor detecte un alto número de conexiones, normalmente erróneas, sea suficientemente inteligente de ir añadiendo las IPs de origen a una lista negra y no permita que vuelva a conectarse en un futuro.

Sin embargo, no sólo sería para el control de quién se está conectando, si no incluso para aplicaciones que inundan el sistema con mensajes no válidos o que nunca serán entregados, o incluso que se envíen una y otra vez sin control.

Soporte a diferentes servidores de notificaciones

Uno de los grandes pilares de Internet ha sido siempre la descentralización de los servicios, pero la interoperabilidad entre ellos. Así por ejemplo ocurre con el correo electrónico, ya que las personas son capaces de escribirse entre ellas aunque no sean del mismo proveedor (por ejemplo, esto no ocurre en los sistemas de mensajería instantánea de hoy día), ya que los protocolos están muy bien definidos y es capaz de enrutarse mensajes entre diferentes servicios, con la capacidad de que lleguen de unos a otros sin problemas.

Con esta misma idea ha nacido el servidor de notificaciones, queriendo que sea federado y que cualquier persona pueda instalarlo, incluso creando su propio servidor desde cero, mientras se respeten los estándares.

Así pues, la idea de que haya una red de servidores de notificaciones federado es algo lógico. Podría ser interesante que un servidor de Telefónica reciba alguna notificación que vaya a un servidor de Vodafone y sea suficientemente inteligente el enrutado para poder reencaminar la notificación sin molestar a la red interna de cada servidor.

Es por esto que quizás en versiones futuras se cambie la forma en que se da la URL de push (a la que el servidor de terceros tiene que enviar su notificación en formato JSON con un HTTP POST), de una URL normal, del estilo `http://push.telefonica.es/notify/abcdefghijklmnopqrs-`

`tuvwxyz` a algo similar a una dirección de correo, y sea el propio sistema federado DNS el que sepa cómo reencaminar el paquete, del estilo de `abcdefghijklmnopqrstuvwxyz@push.telefonica.com` y que mediante DNS pueda resolver alguno de las entradas `SRV`[1] en el DNS donde se colocaría alguna información para enrutar el mensaje de forma correcta.

Control de presencia

Una de los principales argumentos de los desarrolladores de aplicaciones de mensajería instantánea es que requieren tener un control sobre si el usuario está conectado, desconectado o en estado ausente. Eso es sencillo para ellos si crean un propio socket contra sus servidores y mantienen la conexión y el estado ellos, sabiendo si se cae la conexión, usando APIs del sistema saber si la pantalla está encendida y avisar del estado…

Sin embargo, el principal problema es que los usuarios no suelen tener solo una aplicación de este tipo, si no que suelen tener varias, muchas veces dependiendo del círculo de amigos, o de donde tengan más contactos. Así pues, podría ocurrir que ese socket se multiplicara por dos, tres, o incluso 4, según el número de aplicaciones que lo requieran.

Es por esto que una de las ideas es la de proveer un API a los servicios para comprobar si alguien está conectado/desconectado o en estado ausente. Esto se podría hacer de una forma muy sencilla tal y como está montado el servidor de aplicaciones, ya que sería hacer una petición a la URL que queremos saber si se puede entregar, se miraría el estado del nodo o de los nodos que corresponden a dicha URL y se devolvería cuál es el estado actual: conectado, desconectado, ausente o indefinido (cuando estemos bajo UDP y no tengamos una conexión abierta)

Esto trasladaría la carga de la red desde los terminales y la red de telefonía al servidoe de notificaciones y a Internet, porque no tienen que estar respondiendo los terminales continuamente, ni manteniendo canales abiertos, si no que es el propio servidor el que sabe si el dispositivo puede ser alcanzable o no.

Control de entrega de notificaciones

Muchas de las aplicaciones de mensajería instantánea requieren saber si el mensaje enviado ha sido entregado o no. Podría ser para mostrar un pequeño icono al usuario para que sepa que el otro lado ha recibido el mensaje, o bien porque se quiere llevar el control de si se entrega el mensaje correctamente o si el dispositivo está conectado.

Así pues, este sistema proveería un API para poder preguntar sobre el estado de entrega de un mensaje, no sólamente si se ha entregado o no, si no

[1] http://www.ietf.org/rfc/rfc2782.txt

quizás en qué momento, y si no se ha entregado, por qué razón: 2aplicación desintalada, el dispositivo está desconectado…

Ejemplo 8.4. Enviar mensajes sólo si los anteriores han llegado

Un escenario típico para algunos servicios es la de que si no se han entregado los mensajes anteriores, no tiene sentido enviar mensajes posteriores. Esto podría ser válido en aplicaciones que manden un solo `PING` al usuario. Probablemente no sea interesante mandar varias notificaciones, si no saber si la anterior se ha entregado, y si es así, mandar un nuevo mensaje. En el caso de que no se haya entregado, en algún momento se entregará y el mensaje siguien sería el mismo, por lo que no se envía, no ocupa espacio en el servidor de notificaciones, y reduce la carga en la red móvil, al sólo tener que enviar un único mensaje.

Glosario de términos

M

MongoDB
: Base de datos no relacional que guarda sus datos en un formato BSON, similar a JSON, y cuyo lenguaje de consulta es JavaScript, teniendo funciones de reducción, mapeo…

N

Notification Server
: Servidor de notificaciones. Es el conjunto general en la parte centralizada, está compuesto de diferentes partes.

Node.js
: Plataforma de desarrollo en la parte servidora escrita principalmente en JavaScript. Permite una programación rápida, basada en eventos y asíncrona.

R

RabbitMQ
: Sistema de mensajería escrito en Erlang, muy rápido, con muchas configuraciones para las colas y preferencias para alta disponibilidad y alta carga.

Notification Server
: Servidor de notificaciones. Es el conjunto general en la parte centralizada, está compuesto de diferentes partes.

U

User Agent
: Agente de usuario. Es el programa que ejecuta una instancia web. En este caso, podría ser el propio sistema operativo Firefox OS o el navegador Firefox.

W

WebApp
: Aplicación web. Programa instalado en el dispositivo, basado en tecnologías web.

Bibliografía

PDFs

[NikMar-libuv] Nikhil Marathe. Copyright © 2012 Nikhil Marathe. *An Introduction to libuv*. Libro online [http://nikhilm.github.com/uvbook/An%20Introduction%20to%20libuv.pdf]

Libros

[NikMar-libuv] Nikhil Marathe. Copyright © 2012 Nikhil Marathe. *An Introduction to libuv*. Libro online [http://nikhilm.github.com/uvbook/An%20Introduction%20to%20libuv.pdf]

Artículos

[NikMar-libuv] Nikhil Marathe. Copyright © 2012 Nikhil Marathe. *An Introduction to libuv*. Libro online [http://nikhilm.github.com/uvbook/An%20Introduction%20to%20libuv.pdf]