Notificaciones Push

Para dispositivos móviles y la web

Guillermo López Leal

Notificaciones Push: Para dispositivos móviles y la web

Guillermo López Leal

Agradecimientos

Quiero agradecer a las siguientes personas su ayuda con este proyecto ya que sin ellas no podría haber hecho esto:

- Bernardo López y Carmen Leal por todo lo que han hecho siempre por mi. Gracias.
- Leticia Núñez por aguantarme. Gracias.
- Francisco Marzal por su inestimable ayuda durante la carrera y el proyecto.
- 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 del grupo de Open Web Device o de otros, que me ha enseñado un montón de cosas durante mi año allí. Gracias a todos.

Tabla de contenidos

1. Introducción 1
    Resumen 2
    Objetivos 2
    Descripción del servicio 2
2. Estado del arte 4
    Redes móviles: funcionamiento y problemas 4
        Redes LAN públicas o privadas 4
        Estados del dominio de circuitos 6
    Operadores 9
        WAP Push 9
    Internet 11
        GCM: Google Could Messaging 12
        APNS: Apple Push Notification Service 17
        Thialfi: el futuro de Google 17
3. Application Program Interface, API 18
    API between WebApp and the User Agent 19
    API between the User Agent and the Notification Server 21
    API between the Application Server and the Notification Server 25
    API entre WA y AS 27
    Tokens 27
        WAToken 27
        UAToken 28
        AppToken 28
    WakeUp 28
4. Seguridad 30
    Tokens de dispositivo 30
    Registro con clave pública-privada 32
    Notificaciones firmadas 32
    Verificación de notificaciones 34
    Ataque DDoS: flooding 34
    Comunicación cifrada vía SSL 34
5. Lecciones aprendidas 35
    Olvidar muchas cosas 35
    Investigar las alternativas. Siempre 35
    Hacer tests 35
    Jugar, mucho 35
    Tener claro qué se quiere y enfocarse en prioridades 35
    Conocer a gente que se haya pegado con las tecnologías 36
    Trabaja en abierto 36
    El hardware es muy importante 36

Lista de figuras

2.1. Esquema de estados radio 8
2.2. Funcionamiento WAP Push 10
2.3. Funcionamiento de GCM 15
4.1. Generación de token 31
4.2. Verificación de tokens 32
4.3. Registro de WA en el UA 32
4.4. Verificación de firma (en `src/common/crypto.js`) 33
4.5. Comprobación de firma (en `src/ns_as/ns_as_server.js`) 33

Lista de ejemplos

3.1. Multiple device messages 28
3.2. Message broadcast 28

Capítulo 1. Introducción

Una de las grandes ventajas que ha dado Internet al desarrollo de aplicaciones es, a parte de una distribución mucho más sencilla, la posibilidad de poder obtener datos de forma dinámica, la mayoría de veces pudiendo cambiar el comportamiento del propio programa gracias a los datos descargados desde la red o incluso actualizar la información a mostrar a los usuarios dependiendo del momento del día u otras características: nos muestran 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 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 momentos puntuales que han sucedido en Internet, desde redes de módem lentas que conocemos hoy en día 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, en muchas veces, incontrolada.

Así pues, la descarga de toda comunicación puede realizarse de diferentes maneras, ya sea de forma síncrona o asíncrona, pero se pueden agrupar en dos grandes grupos 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 a lo largo de la historia de Internet y de las aplicaciones. Durante mucho tiempo sólo se usó las soluciones de polling, debido a que no se había desarrollado una tecnología de push eficiente y que era más fácil 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 está desaconsejado por varios motivos que hacen que las redes se colapsen con su uso, o bien que sea muy ineficiente. El primero de ellos es porque usa un alto 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, ya que puede que realmente no haya datos, pero nosotros preguntamos si los hay, colapsando

[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 conjunción de estos dos problemas (realizar conexiones y que sean 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 tienen varias desventajas y son realmente válidos para conexiones estables y sin limitaciones, pero 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 sea implantado por todos los navegadores y que 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.

La principal idea de crear este servicio es la de usar de una forma mucho más eficiente los recursos de las redes móviles por lo que el uso de la batería será menor, a la vez que se reduciría el tráfico de señalización en la red de telefonía móvil, lo que permitiría a las empresas de telecomunicaciones el ahorrar recursos como ofrecer un mejor servicio a sus usuarios.

Capítulo 2. Estado del arte

Este capítulo tiene como objetivo listar 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 cuáles son las diferentes propuestas que se han realizado en los últimos años, necesitamos 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 la redes móviles y que use pocos recursos, ancho de banda y en general, cualquier operadora pueda pensar en él, hay que plantear en primer lugar cómo es el funcionamiento de las redes móviles que están despliegadas a lo largo del mundo, conocer los detalles de las implementaciones y el por qué se han hecho e incluso los acuerdos que se han tenido que llegar entre la capacidad de las líneas, los mensajes de señalización y diferentes aspectos para que el servicio esté balanceado entre los usuarios de las redes y los propietarios de estas.

Es por todo esto que la necesidad de conocer en profundidad en qué gastan los dispositivos móviles su batería a lo largo del tiempo de conexión y por qué no se hace un buen uso de todos los recursos que nos da las redes móviles, es necesario para empezar a plantear un escenario satisfactorio que guste a todas las partes.

Redes LAN públicas o privadas

En primer lugar, tenemos 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, el cual equivalen a 4.294.967.296 equipos. Sin embargo, el principal problema es que son demasiado pocas para todos los diferentes dispositivos que hay hoy en día conectados a Internet, y requieren de técnicas (como el NATting) para poder acomodar más equipos, con problemas que veremos más adelante.

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

- **IPv6.** Es la evolución directa de IPv4, estandarizada como RFC-2460[2], pero sin compatibilidad hacia atrás, por lo que equipos funcionando sólo con IPv6 no son capaces de llegar a dispositivos equipados sólo con IPv4 y para que puedan hablar entre ellos hay que realizar túneles o conversiones entre ambos tipos de direcciones.. Tiene 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, tenemos el problema de si dichas IPs están en redes públicas o privadas, los cuales limitan el acceso que tienen hacia otros dispositivos dentro o fuera de dichas redes, que puede ser beneficioso en algunos casos y realmente perjudicial en otros.

Así pues, tenemos 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 que sean visibles por el resto de dispositivos conectados a Internet y, en determinada medida, accesible sin restricciones. Esto quiere decir que cualquiera puede saber que estamos en Internet y tenemos una conexión directa con los equipos a los que queremos conectarnos (servidores web, de correo, de mensajería instantánea) de una manera directa, y ellos pueden contactar con nosotros aunque no tengamos conexión, puesto que somos visibles.

- **LAN privadas.** Estamos dentro de un segmento de red que sólo es visible para otros equipos que compartan dicho segmento con nosotros. Esto hace que no tengamos una conexión directa con Internet, si no que tengamos 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 reencamine nuestras peticiones hasta el exterior, a la vez que reencaminando las respuestas desde el exterior hasta nuestro dispositivo.

Sin embargo, ninguna de las soluciones es la panacea por diferentes cuestiones. En primer lugar, tener una IP pública hace que estemos expuestos de forma completa a cualquier dispositivo de Internet que pueda vernos si no tenemos un control intermedio (normalmente un firewall) para protegernos de ataques. Además, al ser visible por todo el mundo, somos vulnerables a cualquier petición que nos venga del exterior, como escaneos de puertos, PINGs, o floods incontrolados. Es por esto, que la mayoría de operadores de telefonía móvil que tienen servicios de datos, intentan evitar el dar IPs públicas a sus dispositivos para evitar estos problemas, que al final llevan a un uso de batería mayor debido a que se están recibiendo más datos, detalle que se explicará en la siguiente sección.

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

Pero a su vez, el tener un direccionamiento privado tampoco ayuda en demasía a las redes móviles. No ayuda ya que, aunque se haya comentado que el dispositivo sea accesible desde Internet suele ser un inconveniente (problemas de seguridad, gasto de batería), en algunos casos nos es muy útil, como en el momento de recibir notificaciones push. El tener un direccionamiento privado nos hace que sólo seamos visibles para nuestros vecinos en nuestra red privada, y que la IP interna sea diferente a la externa, por lo que cualquier intento de conexión desde la red pública (Internet, por ejemplo) no pueda llegar a nuestra IP, ya que difiere la que nosotros tenemos asignada con la que salimos a Internet.

También podría pensarse que debido a que IPv4 tiene un rango de direcciones muy limitado, el uso de redes privadas con NATing está muy extendido, frase que es correcta. Pero también podría pensarse que usando IPv6 todos los problemas están resueltos, y que al haber tantas posibilidades de direccionar dispositivos el uso de mecanismos de push se irá perdiendo, pero hay que tener en cuenta, como se ha dicho 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, algunas conexiones serán bloqueadas, para evitar congestión en la red y un alto uso de batería, por lo que 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 nos permite 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 darnos acceso a Internet, hace de NAT, por lo que solemos tener sistemas 2 en 1.

Estados del dominio de circuitos

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

Para simplificar, sólo están listados los estados de la red 3G, que ahora mismo es la más usada.

- **Cell_DCH (Canal dedicado).** Cuando el teléfono está 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.

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

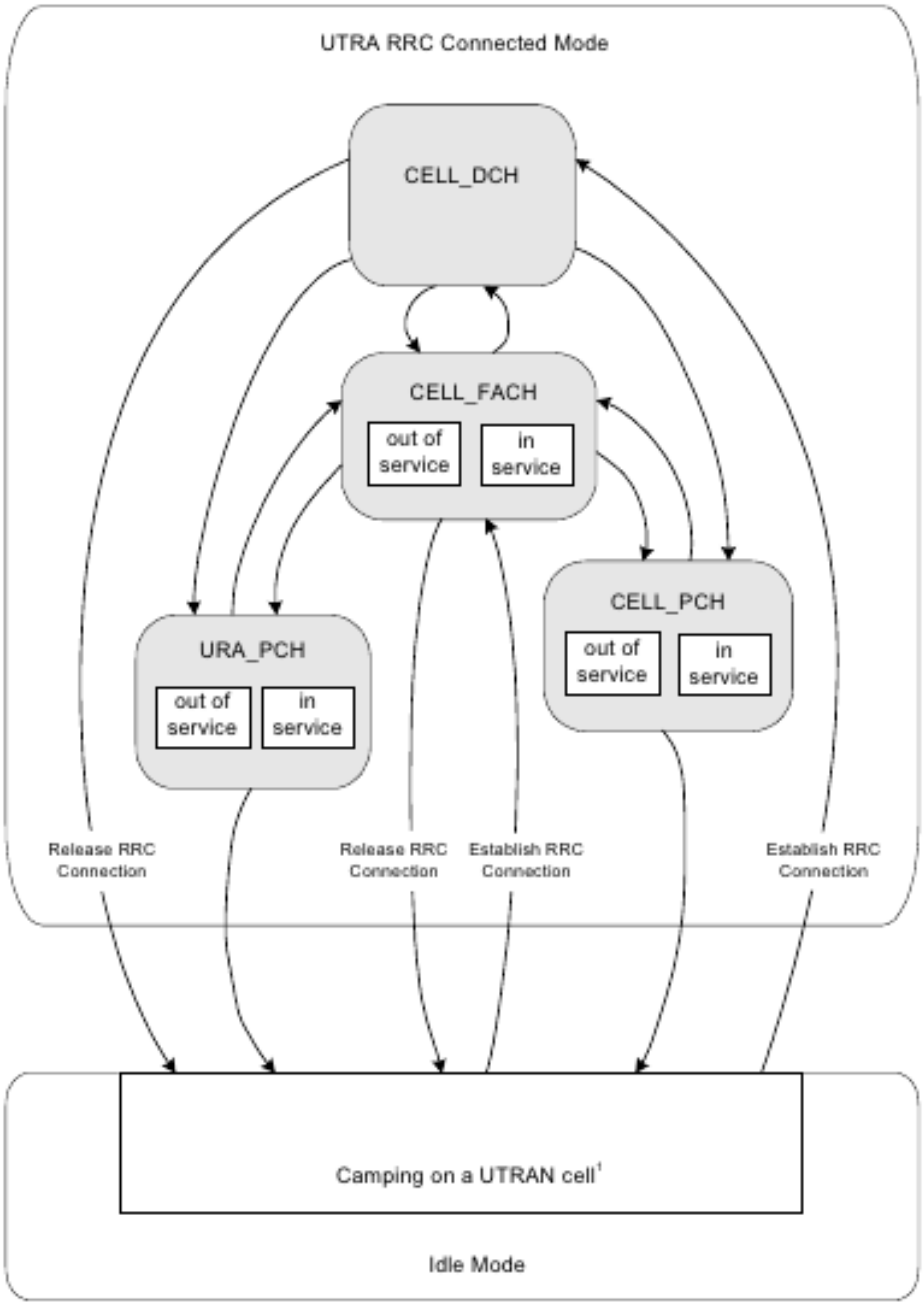
- **Cell_FACH.** En este estado, el teléfono está conectado a la red móvil y usa un canal compartido con otros terminales. Suele estar transmitiendo un pequeño número de paquetes, que normalmente son keep-alives. El temporizador de inactividad en este estado es un poco mayor (del orden de los 30 segundos)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

- **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 este estado, el dispositivo informa a la red cada cierto tiempo 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 un montón de mensajes de señalización.

- **RRC_IDLE.** Es el estado más económico en el que puede estar la radio, ya que prácticamente está dormida. También, hay que decir que el modem está escuchando a 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 está en este estado, puede ser despierto a un estado más activo de red y también la red sabe en qué LAC se encuentra moviéndose el dispositivo, por lo que si la red necesita informar al dispositivo de que está ocurriendo algo en la red (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, ordenador por consumo energético en el dispositivo.

Figura 2.1. Esquema de estados radio



Así pues, como se puede ver, 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), lo que hará que la batería dure más tiempo, tengamos menos señalización de red, y estemos ocupando menos recursos, tanto en nuestro dispositivo como en la red.

Entonces, como podemos ver, queremos mantener al dispositivo 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.

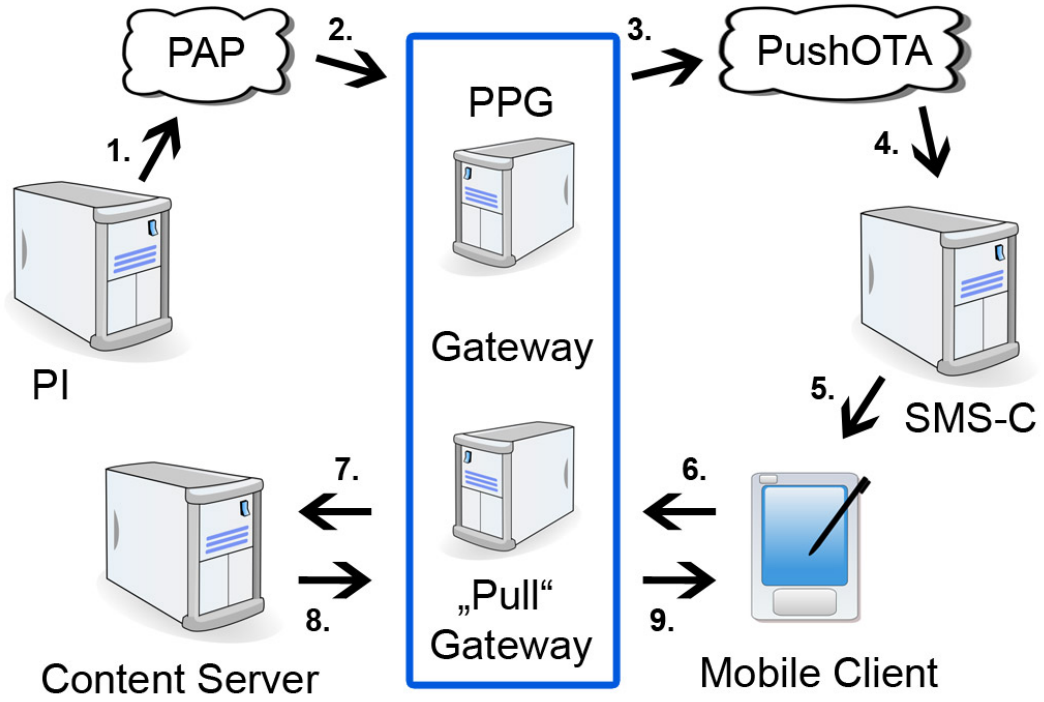
Como se puede observar, esto es un problema, puesto que está pensado para la descarga de datos desde Internet, y obliga a que se abra el navegador de forma predeterminada para la descarga del contenido.

[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. A parte 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 cómo 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.

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[8]

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

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

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.

- 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 bas-

tante 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.

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[9]

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).

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

- 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, 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.

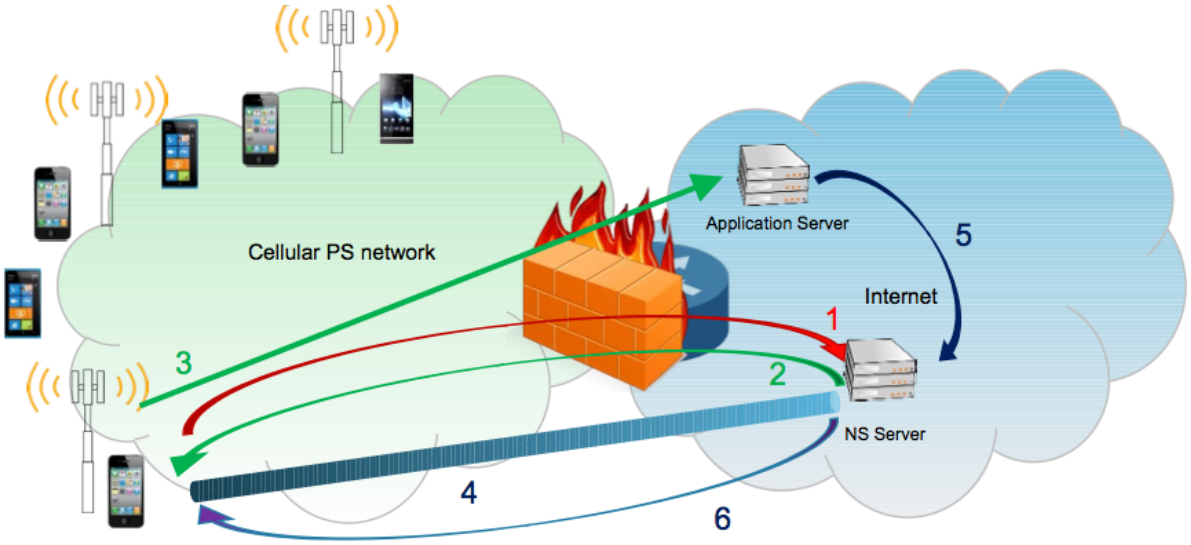
- Existe la posibilidad de sobrescribir mensajes antiguos (con la misma `collapse_key`), enviar mensajes sólo cuando el usuario esté activo (con `delay_while_idle`) e incluso comprobar si los mensajes van a ser entregados o no dependiendo de la respuesta que de el servicio.

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

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.

Thialfi: el futuro de Google

Thialfi es un nuevo protocolo diseñado y publicado por Google[10] 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 las posibilidades, y lo implementa Google Chrome en su versión de escritorio y móvil.

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 "topics" a los que el usuario y las aplicaciones se suscriben.

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

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 estaba proponiendo el W3C, se mantiene por compatibilidad hacia atrás, pero sin embargo, se sobreescriben algunos métodos para introducir mejoras de seguridad.

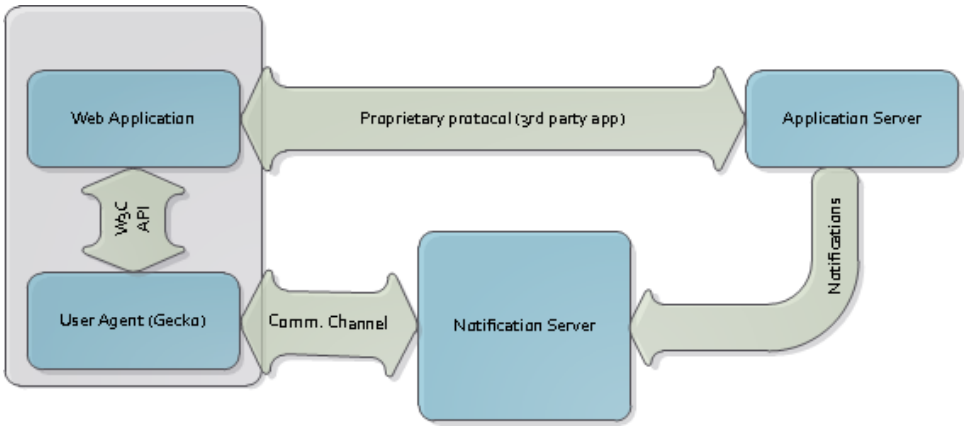
Para entender este capítulo, es necesario presentar a los diferentes actores que hacen aparición:

- **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.



API between WebApp and the User Agent

This API is mainly based on the W3C draft as specified in [http://dvcs.w3.org/hg/push/raw-file/default/index.html] [http://dvcs.w3.org/hg/push/raw-file/default/index.html]

With this API the application is able to register itself into the Notification Server and recover the public URL to be used as notification URL by his Application Server (AS).

This API (under the navigator.mozPush object) defines these methods:

- requestURL
- getCurrentURL

navigator.mozPush.requestURL

This method allows the application to register it self into the notification server.

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

This method should receive this two parameters:

- watoken: The WA Token used to identify the user of the application.
  The application developer can decide to use the same WAToken for all his users or a group of them so the notification will act as a broadcast message

  It's very important to note that this token (mainly if used to identify one particular user) SHALL be a secret. It's recommended that this token will be generated by the server using a SHA hash based on the login details (as an identification cookie).

  If this parameter is not provided, a randomized one will be generated by the UA engine.

- pbk: This parameter will contain a RSA Public key coded in BASE64.
  This public key will be used by the notification server to validate the received messages signature, so the private key will be used by the AS to sign the messages.

It's under definition to send two parameters or only one which will be a JSON object:

```
navigator.mozPush.requestURL({
  watoken: <watoken>,
  pbk: <Base64 codified public key>
```

```
})
```

Finally this method will response asynchronously with the URL to be sent to the AS in order to be able to send notifications.

```
var req = navigator.mozPush.requestURL(this.watoken, this.pbk);
req.onsuccess = function(e) {
  alert("Received URL: " + req.result.url);
};
req.onerror = function(e) {
  alert("Error registering app");
}
```

navigator.mozPush.getCurrentURL

This method allows the application to recover a previously requested URL to the UA API, so it's not needed to ask for it to the notification server.

```
navigator.mozPush.getCurrentURL()
```

This methods will response asynchronously with the URL to be sent to the AS in order to be able to send notifications.

```
var req = navigator.mozPush.getCurrentURL();
req.onsuccess = function(e) {
  alert("URL = " + req.result.url);
};
req.onerror = function(e) {
  alert("Error getting URL");
}
```

After register the application into the Notification Server, all received notification through the given URL will be delivered to all user agents which registered the pair (WAToken + PBK).

Since the notifications will be received by the UA it's needed a way to notify each application. The current specification is using the new System Messages infrastructure defined in FirefoxOS.

In this case, the application shall register to the "push-notification" event handlser:

```
navigator.mozSetMessageHandler("push-notification", function(msg) {
```

```
  alert("New Message with body: " + JSON.stringify(msg));
});
```

The complete example:

```
var req = navigator.mozPush.requestURL(this.watoken, this.pbk);
req.onsuccess = function(e) {
  alert("Received URL: " + req.result.url);
  navigator.mozSetMessageHandler("push-notification", function(msg) {
    alert("New Message with body: " + JSON.stringify(msg));
  });
};
req.onerror = function(e) {
  alert("Error registering app");
}
```

API between the User Agent and the Notification Server

With this API the client device is able to register his applications and itself into the selected notification server.

This API isn't yet standarised, anyway the one explained here is an on working proposal.

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>"
}
```

API between the Application Server and the Notification Server

With this API the Application server is able to send asynchronous notifications to his user's without heavy infrastructure requirements or complex technical skills.

This is a simple REST API which will be improved in future releases.

This version accepts only one HTTP POST method used to send the notification payload. The following payload SHALL be POSTED to the publicURL which defines the application and user, like: `https://push.telefonica.es/notify/SOME_RANDOM_TOKEN`

```
{
  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>",
}
```

The server response can be one of the following:

```
STATUS: 200

{
  status: "ACCEPTED"
}
```

```
STATUS: 40x

{
  status: "ERROR",
  reason: "JSON not valid"
}
```

```
STATUS: 40x

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

```
STATUS: 40x

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

```
STATUS: 40x

{
  status: "ERROR",
  reason: "You must sign your message with your Private Key"
}
```

```
STATUS: 40x

{
  status: "ERROR",
  reason: "Bad signature, dropping notification"
```

```
}
```

```
STATUS: 40x

{
  status: "ERROR",
  reason: "Try again later"
}
```

```
STATUS: 40x

{
  status: "ERROR",
  reason: "No valid AppToken"
}
```

API entre WA y AS

Este es el API que es independiente del protocolo de push propuesto, por lo que está fuera del alcance de este proyecto.

Sin embargo, a través de este API la publicURL recibida por la aplicación a través del API entre la WA y el UA, debería ser mandada al servidor de terceros.

Recomendaciones. Sin embargo, hay una serie de pautas que deberían seguir los desarrolladores para pasar la información de una forma segura a través de la red.

- Usar un canal seguro de envío de datos, como puede ser una conexión SSL con su servidor.
- Cifrar el contenido de la URL con algún algoritmo que permita descifrarlo en la parte servidora.
- 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.

Tokens

The tokens are an important part of this API since it identifies each (user) actor (device and applications) in a unique or shared way.

WAToken

This token identifies the user or group of users and SHALL be a secret.

If this token is UNIQUE (and secret, of coures) will identify a unique instance of the application related (normally) to one user. In this case the returned URL will be unique for this WAToken.

If this token is shared by different devices of the SAME user (and secret), will identify a unique user with multiple devices. In this case, the returned URL will be unique per user but each URL will identify multiple devices the user is using.

Ejemplo 3.1. Multiple device messages

This can be used by applications in which the user require the same information across his devices, like the mobile and the desktop app. Can be used, for example, by e-mail clients.

Finally, if a developer decides to deliver the same WAToken to all his users (in this cases is obviously not a secret one), then the returned URL will identify all instances of the same application. In this case each notification received in the publicURL will be delivered to ALL the devices which have the application installed (and registered). This will be a BROADCAST message.

Ejemplo 3.2. Message broadcast

This can be used by applications in which all users require exactly the same information at the same time, like weather applications, latest news, ...

UAToken

This token identifies each customer device in a unique way.

This token is also used as an identification key since this isn't a random one. This token is an AES encrypted string which will be checked for validaty each time it's used.

This token should be delivered after identify the user in a valid way, anyway this identification procedure is out of the scope of this specification.

AppToken

Automatic generated token by the notification server which identifies the application + user as in a unique fashion.

This token is included in the publicURL which identifies the application, and normally is a SHA256 hashed string with the WATokent + the Public Key.

WakeUp

When the handset is inside a mobile operator network, we can close the websocket to reduce battery comsuption and also network resources.

So, when the NS has messages to the WA installed on a concrete UA it will send a UDP Datagram to the handset.

When the mobile receives this datagram, it SHALL connect to the websocket interfaces in order to pull all pending messages.

Capítulo 4. 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 4.1. Generación de token

```
function token() {}

token.prototype = {
  serialNumber: 1,

  // The TOKEN shall be unique
  get: function() {
    // SerialNumber + TimeStamp + NotificationServer_Id + CRC -> RAWToken
    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 com-

[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

partir todos las instancias del servidor de notificaciones que generen tokens, 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 4.2. 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]));
}
```

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 4.3. 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.

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 ha-

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

ce 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 4.4. 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 4.5. 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.
- 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 peryo smite 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.

holas

Ataque DDoS: flooding

…

Comunicación cifrada vía SSL

…

Capítulo 5. 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…

Olvidar muchas cosas

Una de las primeras decisiones que se tomó en el proyecto fue usar una base de datos no relacional, las llamadas No-SQL. En el caso del proyecto, 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 íbamos a introducir, ya que se creía que el esquema cambiaría (así sucedió de hecho), y permitiría un prototipado de la plataforma mucho más rápido.

Sin embargo, el cambiar radicamente el chip 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 se siguen teniendo nociones de SQL y muchas de las decisiones para estos sistemas se toman a partir de dichos conocimientos.

…

Investigar las alternativas. Siempre

XXX: node.js y algoritmos

Hacer tests

…

Jugar, mucho

…

Tener claro qué se quiere y enfocarse en prioridades

…

Conocer a gente que se haya pegado con las tecnologías

…

Trabaja en abierto

…

El hardware es muy importante

… y no siempre lo caro es lo mejor, o lo rápido…

…