Servidor de notificaciones Push

Notificaciones para la web

Guillermo López Leal

Servidor de notificaciones Push: Notificaciones para la web

Guillermo López Leal

Agradecimientos

Quiero agradecer a las siguientes personas su ayuda con este proyecto y todo lo que le ha rodeado, ya que sin ellas, yo 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.
- 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

Lista de figuras

Lista de ejemplos

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 de dichas aplicaciones, es 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 los datos a mostrar a los usuarios dependiendo del momento del día u otras características. Aplicaciones que nos muestran los resultados de nuestro equipo favorito de fútbol, el tiempo que va a realizar en nuestra ciudad, o incluso mensajería instantánea son uno de los grandes usos de las tecnologías de internet para aplicaciones locales.

A lo largo de los años, las tecnologías sobre cómo descargar estos datos han ido variando, acomodándose a los momentos puntuales que han sucedido en internet, desde redes de módem lentas para lo 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 estos datos de internet 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 grandes paradigmas son los grandes usados a lo largo de la historia de internet y de las aplicaciones. Durante mucho tiempo sólo se usó las soluciones de polling y ahora cada vez más se usan las soluciones de push, sin embargo, su uso siempre ha sido dispar y, dependiendo de qué tipo de aplicaciones y cómo necesiten los datos, el uso de una o de otra es más importante.

Sin embargo, el primer método está desaconsejado por varios motivos que hacen que las redes se colapsen con su uso. El primero de ellos es porque usa un alto número de conexiones hacia el servidor de terceros, muchas de las cuales no tienen una utilidad real puesto que no siempre que se pregunta si tienen datos realmente se poseen.

Es por esto que los métodos de recogida de información basados en tecnologías Push son ampliamente usados para actualizar la información. Sin embargo, aunque estos métodos a primera vista puedan verse como más eficientes que los de polling, 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 móvil.

- 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

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.

Blah blah

rellenar

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,

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

y requieren de técnicas (como el NATting) para poder acomodar más equipos, con problemas que veremos más adelante.

- 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

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

final llevan a un uso de batería mayor debido a que se están recibiendo más datos, detalle que se explicará en la siguiente sección.

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.

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

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.

  Normally, this state is assigned by the network when the handset is transmitting a small amount of data. So it's common to use it when sending keep-alive packages.

  The inactivity time of this state is a little longer (30 seconds) and is konwn as T2 timer. When T2 timer is shotted, the handset will be moved to Cell_PCH or URA_PCH (depending on the type of network)

- Cell_PCH or URA_PCH (PCH: Paging Channel) (URA: UTRAN Registration Area)
  In this state the handset is not able to send any data except signalling information in order to be able to localize the handset inside the celullar network.

  In both states, the RRC connection is established and open, but it's rarely used.

  In this state, the handset informs the network every time the device change from one sector to another so the network is able to known exactly the BTS which is offering service to the device.

  The T3 timer defines the maximum time to be in a PCH state. This timer is longer than T1 and T2 and depends on each carrier. When it's fired the handset is moved to IDLE mode so if new data transmission is needed the handset will need near 2 seconds to restablish the channel and a lot of signalling messages.

- RRC_IDLE
  This is the most economical state since the handset radio is practically stopped.

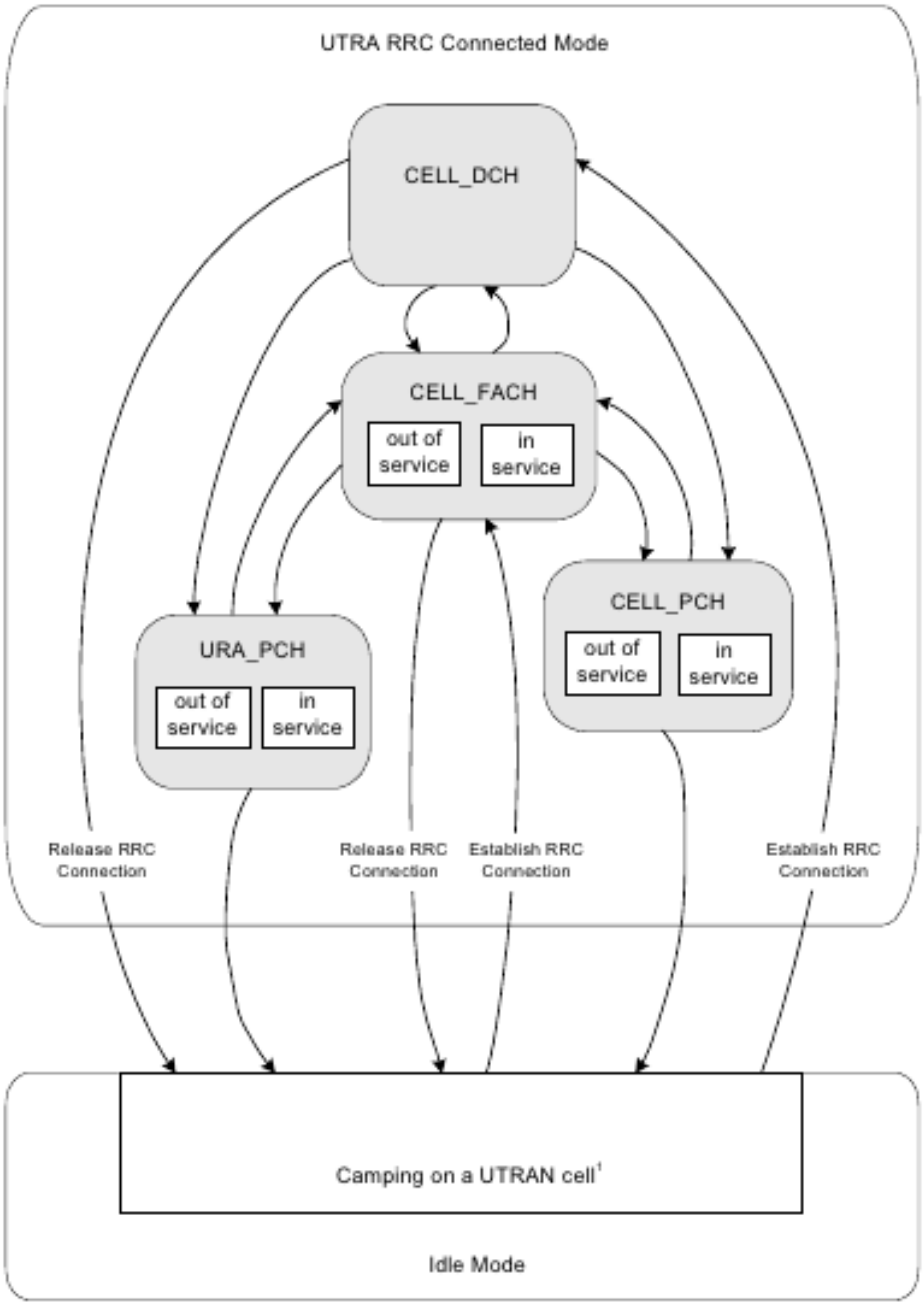
  In this state, the radio is only listening to radio messages quering the handset to "Wake Up" (paging messages).

  Also, the handset modem is listening the cell data so each time it detects that the user changed from one LAC (Localization Area Code - Group of multiple BTS) to another, the handset will change to the PCH state in order to inform the network.

  So when a handset is in this state, it can be Waked Up to a more active state and also the network knowns the LAC where the handset is moving,

so if the network needs to inform the handset it should send a broadcast paging message through all the LAC BTS in order to locate the handset.

The following scheme represent the different radio states ordered by power comsuption on the device:



Estado actual de servicios push

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

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

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.

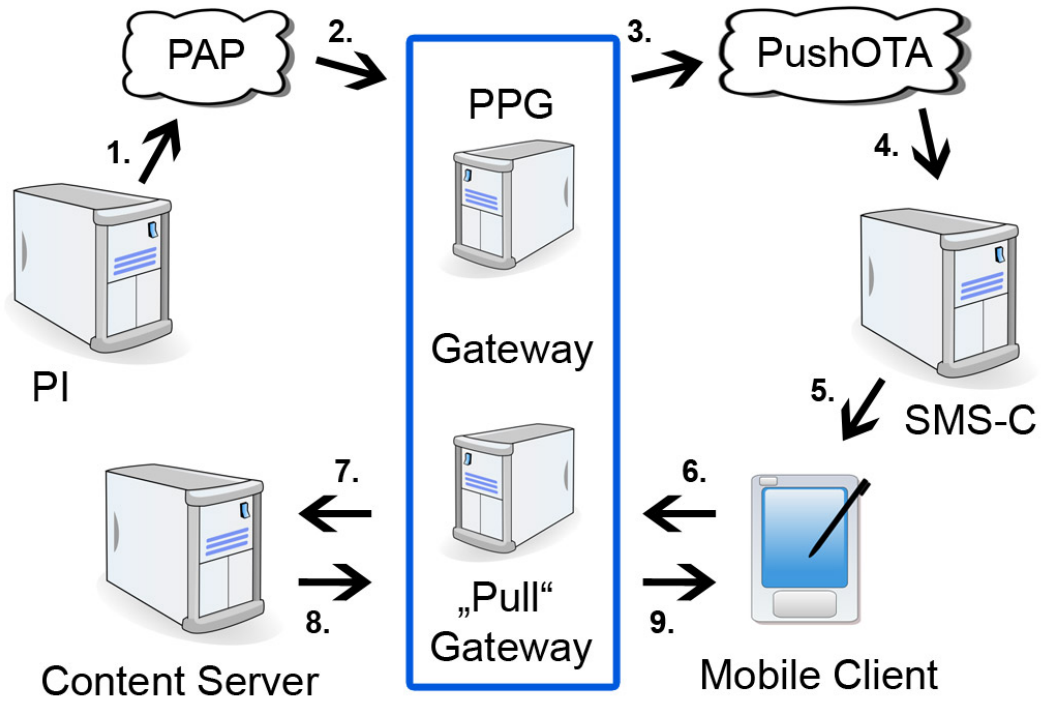
Así pues, su funcionamiento es muy sencillo. El servidor de terceros quiere enviarnos cualquier tipo de mensaje binario (aplicaciones, imágenes, vídeos, simple texto), por lo que en primer lugar se sube a un servidor que está en internet, o bien en la red privada del operador, y después se manda un SMS 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.

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 2.1. 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.

Capítulo 3. Notification server API

The Notification Server API is based on the W3C draft: [http://dvcs.w3.org/hg/push/raw-file/default/index.html] [http://dvcs.w3.org/hg/push/raw-file/default/index.html]

In order to understand this chapter, we'll present the different actors:

- WebApp (WA):
  The user's applications which is normally executed on the user device.

- User Agent (UA):
  Since this protocol born under the Firefox OS umbrella the "operating system" layer is known as the User Agent layer, in our case is the Gecko engine.

- Notification Server (NS):
  Centralized infrastructure of the notification server platform. This one can be freely deployed by anyone since it's open source: [https://github.com/telefonicaid/notification_server] [https://github.com/telefonicaid/notification_server]. The protocol also allows to use any server infrastructure the user wants

- Application server (AS):
  The WA server side. Normally the applications that runs on a mobile device use one or more Internet servers.

  Some of them will be deployed by the same developer as the client application.

  In our case, this server will be the one which send the notification to his clients/users.

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

The complete example:

```
var req = navigator.mozPush.requestURL(this.watoken, this.pb
req.onsuccess = function(e) {
  alert("Received URL: " + req.result.url);
  navigator.mozSetMessageHandler("push-notification", functi
    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 waiti
    },
    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 between the WA and the AS

This is a third party API which is independent of the PUSH protocol, so it's out of the scope of this document.

Anyway, through this API the publicURL received by the application should be send to his server.

Also this channel could be used to receive valid WATokens to be used during the WA registration.

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.