Notification Server

Architecture

Fernando Rodriguez Sela
Guillermo Lopez Leal

Notification Server: Architecture

by Fernando Rodriguez Sela and Guillermo Lopez Leal

Table of Contents

List of Tables

List of Examples

Chapter 1. Introduction

Today mobile applications retrieve asynchronously information from multiple sites. Developers have two ways to retrieve this information:

- Polling: Periodically query the information to the server.
- Push: The server sends the information to the client when the required information is available.

The first method is strongly discouraged due to the large number of connections made to the server needlessly, because information is not available and you lose time and resources.

That is why the PUSH methods are widely used for information retrieval, anyway how PUSH platforms are currently working are misusing mobile radio resources and also consuming lot of battery.

This article aims to explain how to manage this kind of messaging, problems with existing solutions and finally how Telefónica Digital, within the framework of the development of Firefox OS operating system, a new solution designed friendlier to the network and low battery consumption on mobile terminals.

State of the art

Historically mobile operators offered (and offer) real mechanisms PUSH notifications, also known as WAP PUSH. WAP PUSH can "wake up" applications when any action is required of them by the server side (without interaction from the user). Sending WAP PUSH messages is done in the domain of circuits, the same used for voice and SMS, and that is why the user don't need to establish a data connection. These kind messages work properly out of the box.

WAP PUSH solutions works great when the user is registered in the mobile network, but if you are out of coverage or connected to a WiFi hotspot instead a celular network, you can not receive these messages.

Also, if we add that this messages implies an economic cost (basically it is a short message SMS) the effect is that major smartphone operating systems (Apple iOS and Google Android) have implemented a parallel solution that would work regardless of the mobile network to which the user belongs and it can run smoothly when they are using WiFi networks.

Current Internet solutions issues

Internet PUSH solutions are based on a public accesible server which handles all the notification delivery.

These solutions were designed without considering the mobile networks way of working and forces the handset to maintain an open socket with the server in order to avoid misnotifications.

This way of working increases the signalling and the handsets battery consume. For more information about this, please refer to the "Mobile network issues with current PUSH platforms" chapter

Service Description

The Notification Server platform is aimed to deliver push notifications (small messages like a real-time chat, a JSON data structure defining the goal of a soccer match) to web based terminals inside mobile networks.

The main objective of this service is to deliver these messages considering the way of working of the mobile radio so the battery consumption and traffic generated is reduced to the minimal. It is developed for working on stable Internet connections, like on Wi-Fi or Ethernet.

Adventages for developers

Since we want a service to be used, we think it to be very easy to use and to be great for developers.

Now, we point out some adventages with the use of this solution:

- Easy to use API: Based on web technologies.
- Reduce developer deployment consts
- More efficient use of the battery and network resources
- No registration process needed and no subscriptions
- Bigger payloads and more messages per application

Chapter 2. Mobile network issues with current PUSH platforms

This chapter explains why current solutions are bad for the mobile networks and how we designed this server to solve this issues.

In order to understand the complete problem, we need to introduce you on how the mobile nerworks work at radio level and also how the carriers have their network infrastructure. So, go ahead !

Mobile networks in a Private or Public LAN

Since on IPv4 the amount of free adresses is really low, celular networks were divided into the ones with real IPv4 adresses (normally for 3G modems) and private adressing model for handsets.

On the case of private networks, it's obvious that it's not possible to directly notify the handset when the server has a notification for it, so smartphone manufacturers decided to maintain opened channels with their servers so it's possible to notify handsets asynchronously.

On the other hand, if the handset has a public address, or is using IPv6, it's teorically possible to send the message directly making third party solutions unuseful, however in order to protect users, carriers can deploy firewalls to avoid direct access from Internet to the handset.

Mobile Network. Circuit domain states

In the 3GPP TS 25.331 specification, we can query all the circuit domain statues of the RRC Layer (Radio Resource Control).

In order to simplify, we only list the third generation (3G) states:

- Cell_DCH (Dedicated Channel)
  When the handset is in this state is because it has a dedicated channel on the mobile network.

  Normally the network sets a handsent into this state when it's transmitting a big amount of data.

  The inactivity time of this state is really short, known as T1 timer it should vary between 5 and 20 seconds. If T1 is fired, the handset will be changed to the Cell_FACH state.

- Cell_FACH
  In this state the handset is connected to the mobile network using a shared channel with other handsets.

  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.

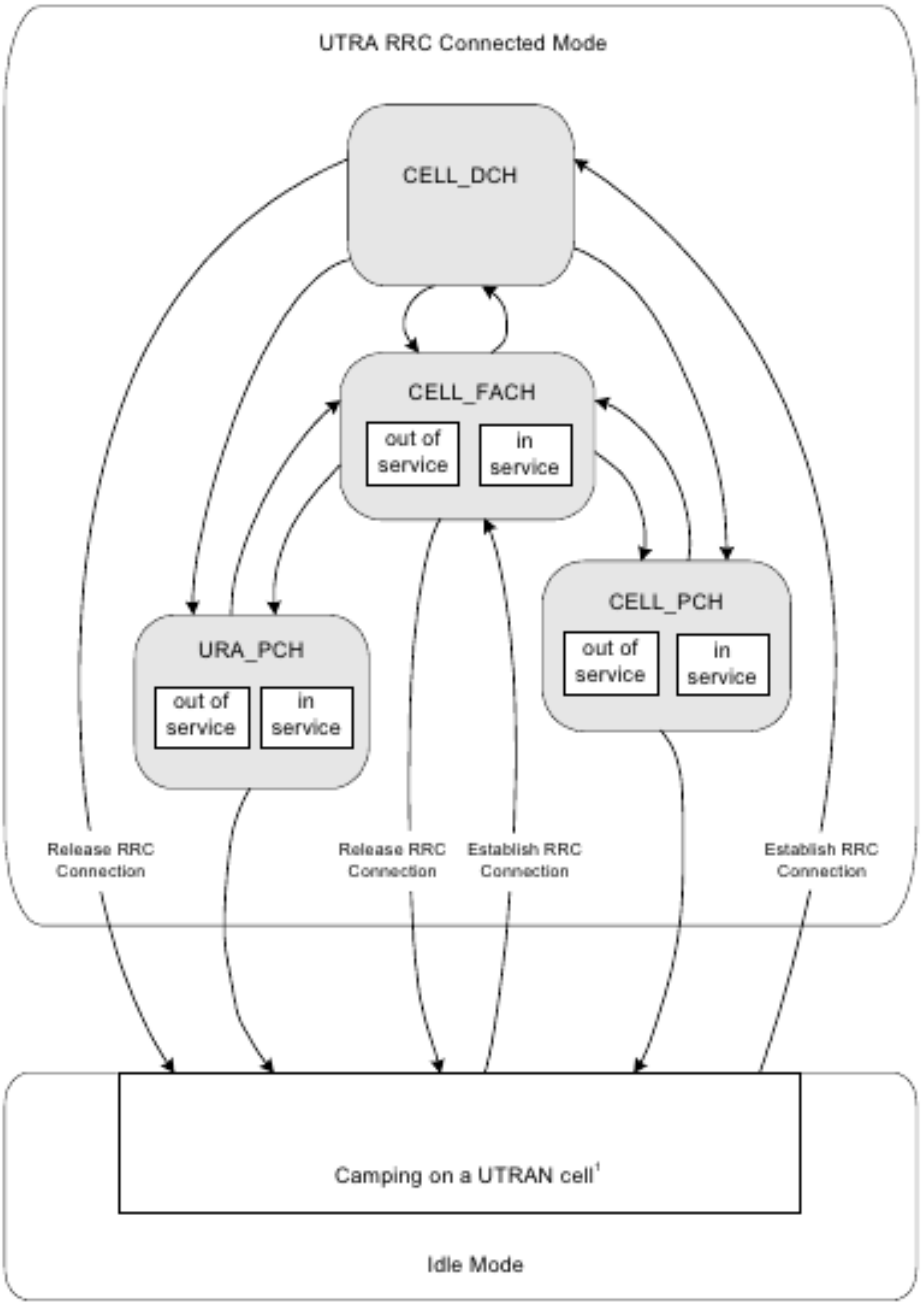
- RRC_IDLE
  This is the most economical state since the handset radio is practically stopped.

  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:



Mobile Network. Package domain states

In the 3GPP TS 23.060 specification, we can analyse all the package domain states of the GMM Layer (GPRS Mobility Management).

The package domain states are simpler than radio ones (only 3 states):

- READY (2G) / PMM_CONNECTED (3G)
  The handset has a PDP context established and is able to send and receive data.

- STANDBY (2G) / PMM_IDLE (3G)

The handset isn't transmitting anything but the PDP context is not closed, so it maintains a valid IP address.

In this state the handset don't consume any resource but the network is maintaining his IP address as a valid one, so it's very important to try to maintain the handset in this state in order to be able to Wake Up it and change to a PMM_CONNECTED state in order to transmit/receive information.

- IDLE (2G) / PMM_DETACHED (3G)
  In this state, the handset hasn't a PDP context established so it hasn't a valid IP address.

Mobile Network. States relation

In this section we show the relation between RRC and GMM states.

In order to simplify this table, we only consider the handset is only using data channels, so no voice nor SMS (circuit domain) is being used.

Table 2.1. RCC - GMM relation

RCC State	GMM State (2G/3G)	Description
Cell_DCH	READY/PMM_CONNECTED	The handset is transmiting or receiving data information using a dedicated channel or a HSPA shared channel.
Cell_FACH	READY/PMM_CONNECTED	The handset had been transmiting or receiving data some seconds ago and due to inactivity had been moved to the Cell_FACH RCC state. Also it's possible that the handset is transmiting or receiving small amount of data like pings, keep-alives, cell updates,...
Cell_PCH/URA_PCH	READY/PMM_CONNECTED	The handset had been in Cell_FACH some seconds ago and due to inactivity had been moved to this less resource consume state. However, the signalling channel is available and is able to change to a data transmission state like FACH or DCH with a little amount of signalling.
Cell_PCH/URA_PCH	STANDBY/PMM_IDLE	The handset is not transmiting nor receiving any amount of data and also the signalling connection is closed.
RCC State	**GMM State (2G/3G)**	**Description**

RCC State	GMM State (2G/3G)	Description
		However the IP address is maintained by the network and associated to this handset. This is one of the most interesting states since the PDP context is not closed, the IP address is still valid and the handset is not consuming baterry, network traffic,... As soon as the handset needs to restablish the data channel the radio state will be changed to FACH or DCH.
RRC_IDLE	STANDBY/PMM_IDLE	This state is the same as the previous one since the radio state is IDLE.
RRC_IDLE	IDLE/PMM_DETACHED	The handset is not transmitting nor receiving anything and also it hasn't any PDP context established, so no IP address is available for this handset. Normally this state is after 24h of innactivity in the package domain.
RCC State	**GMM State (2G/3G)**	**Description**

Mobile Network. Signalling storms

This is a carrier well-know effect after the big adoption of smartphones around the word.

As we explained in previous sections, each time the network decides to move a handset from one state to another is needed to restablish channels and starts a negotiation between the network and the handset with the signalling protocol.

Since nowadays handsets are sending keep-alives to maintain their connections opened, the efect is that the handsets is continously changing from one state to another producing a lot of signalling in the network and also consumes a lot of battery resources.

Mobile Network. Battery comsuption

The battery comsuption depends on the Radio state. The following list represent the amount of battery needed on each state represented in relative units:

- RRC IDLE: 1 relative unit
- Cell_PCH: < 2 relative unit
- URA_PCH: < or equal than Cell_PCH

- Cell_FACH: 40 relative units
- Cell_DCH: 100 relative units

Chapter 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.

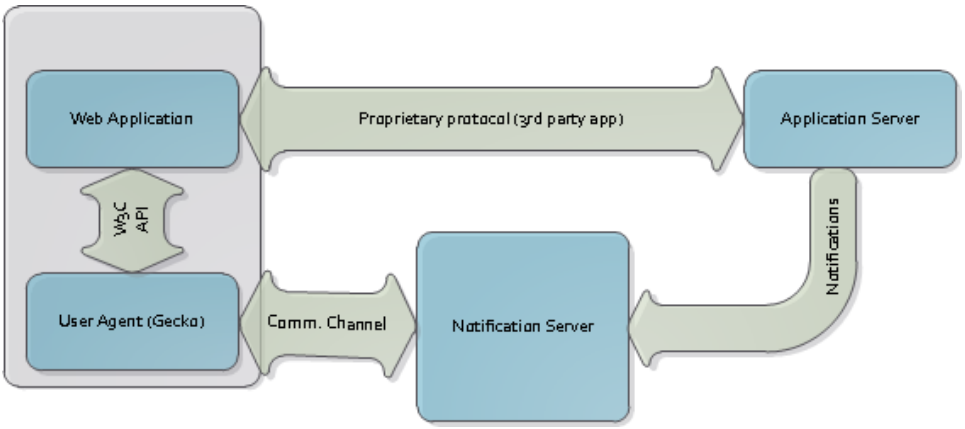
- User Agent (UA):
  Since this protocol born under the Firefox OS umbrella the "operating system" layer is known as the User Agent layer, in our case is the Gecko engine.

- 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

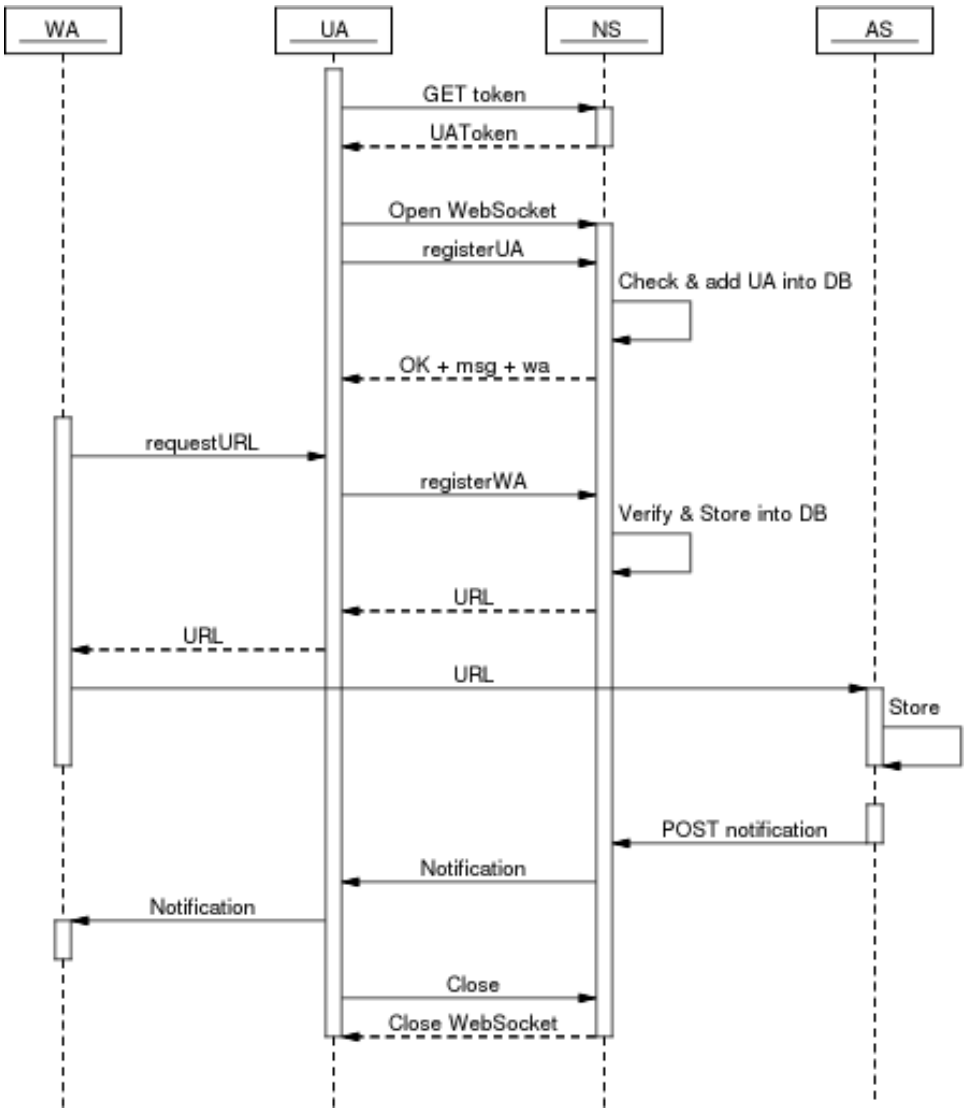
- Application server (AS):
  The WA server side. Normally the applications that runs on a mobile device use one or more Internet servers.

  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.



The following sequence diagram shows a tipical message flow between actors:



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

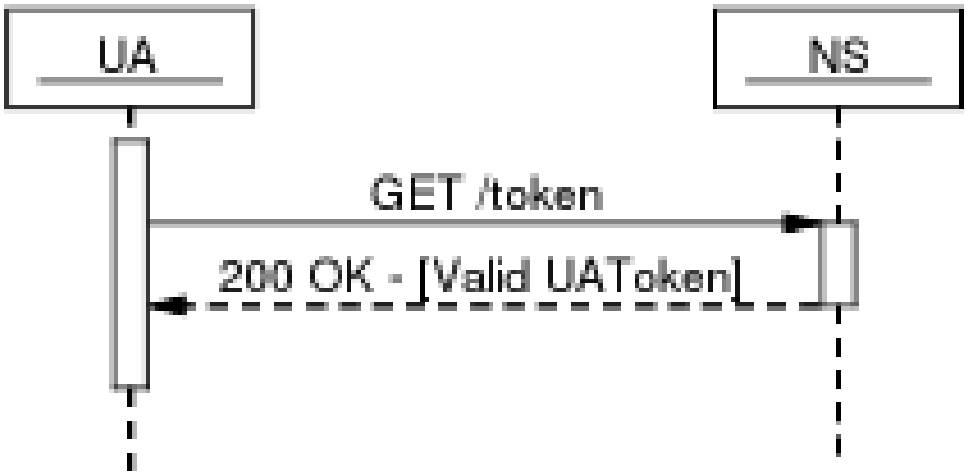
This channel only offers one method to get a valid UAToken.

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 n
    },
    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:

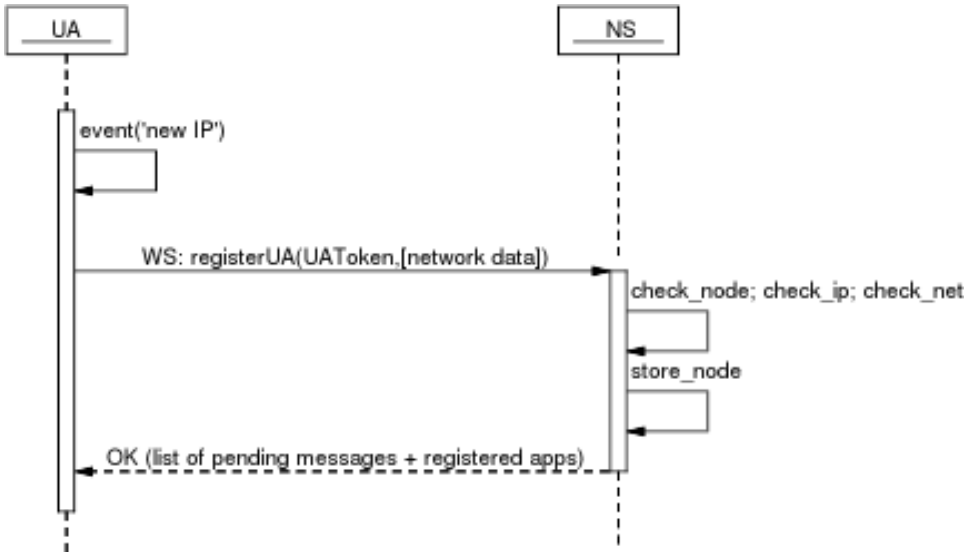
```
{
  status: "REGISTERED",
  statusCode: 200,
  messageType: "registerUA"
}
```

```
{
  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"
}
```

This method is also used to anounce a new IP address or a network change.



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.

Example 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.

Example 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.

Chapter 4. Notification Server Architecture

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

Technologies used

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

MongoDB

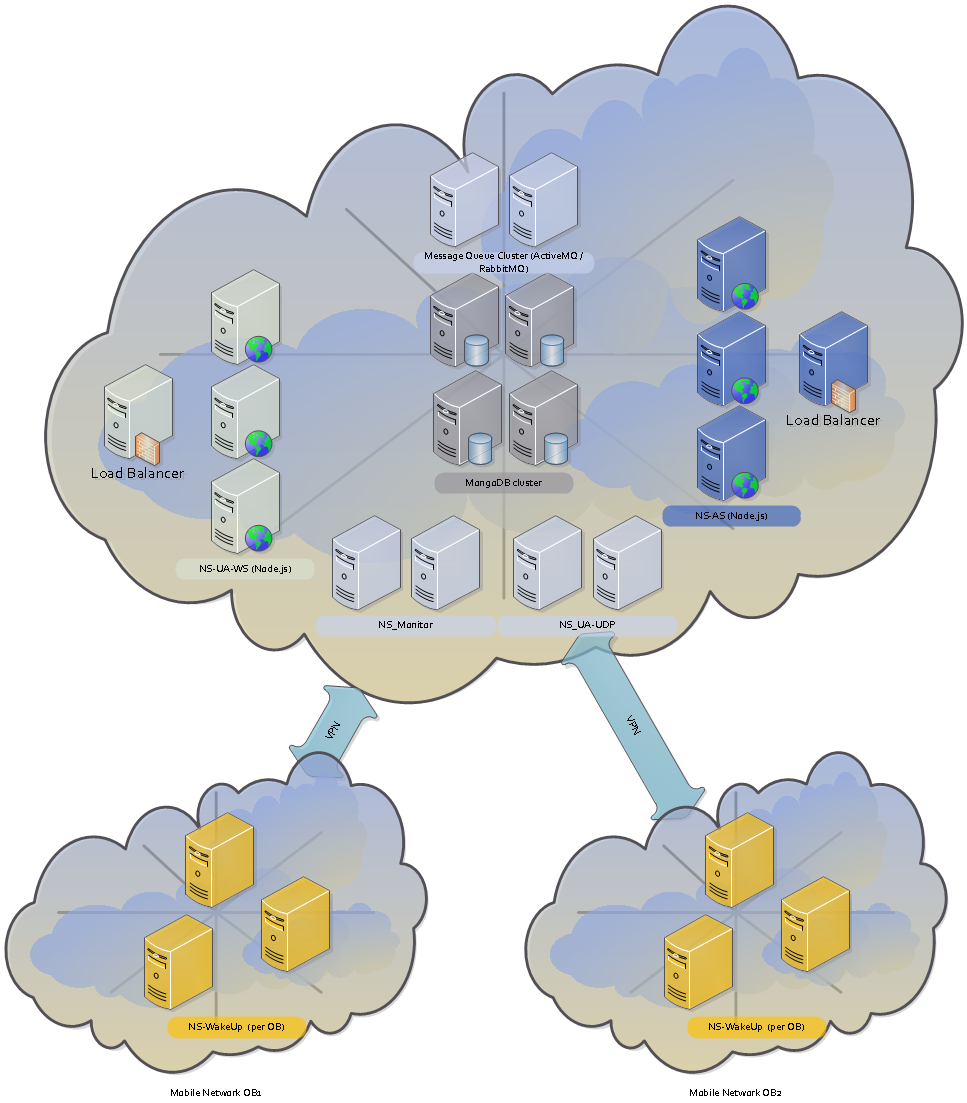
RabbitMQ

Node.JS

Types of servers

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

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



NS-UA-WS

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

This server offers to channels:

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

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

NS-UA-UDP

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

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

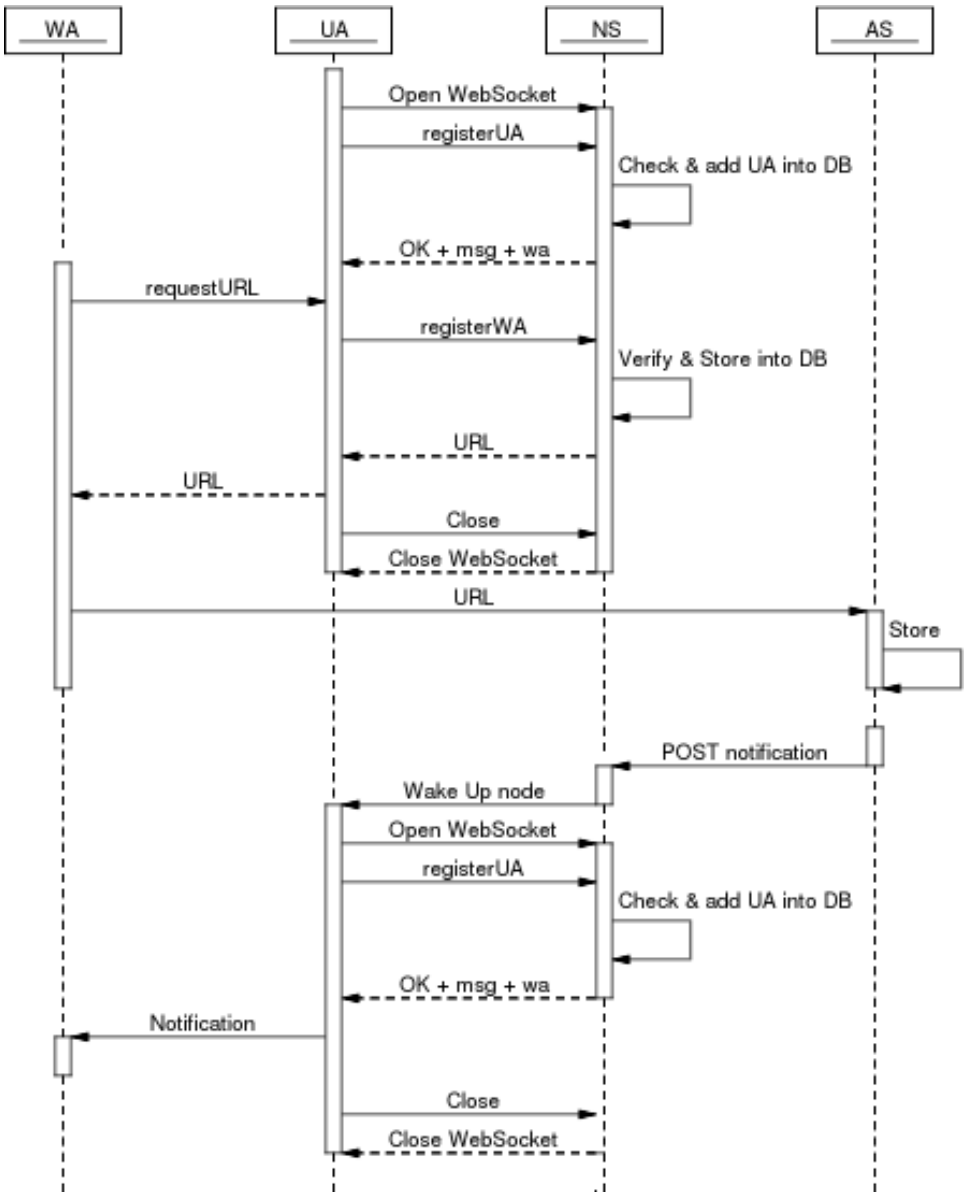
NS-WakeUp

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

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

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

The following scheme shows who the notification is sent with a wakeup server:



NS-AS

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

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

NS-Monitor

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

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

Message Queue (RabbitMQ)

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

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

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

NO-SQL Database (MongoDB)

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

This is a non relational database.

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

Chapter 5. Notification Server Performance

This chapter shows some performance test done to the notification server

Chapter 6. Security

This chapter explains all security topics related with the notification server

The security can be splitted into different areas:

Identify the nodes

[Talk about AES encrypted UAToken]

Identify the applications

[Talk about WAToken (must be a secret!)]

Verify the origin

[Talk about notification signature validation]

Possible attacks and how to mitigate it

- An evil AS wants to send notifications: Needs to known the private key
- An evil WA wants to receive notifications from another WA: Need to know the WAToken which SHALL be a secret.
- An evil device wants to register as another one: Needs to know the UAToken (managed by the OS)
- DDoS attacks: Abuse controls (see future work)

Chapter 7. Notification Server Deployment

This chapter explains how is the notification server will be deployed

Chapter 8. Lessons learned

In this chapter we will explain the lessons we learned during the daily work in a high scalable and high performance server infrastructure

Chapter 9. Future ideas for next releases

The following sections will point out future ideas to be developed in next server releases.

IPv6 support

Enque low priority messages

Enque low priority notifications which will be sent in a common connection, this will consider the TTL of the message.

Backup pings

If the client is not notified after some time, it will connect to the server and check if some message is pending to be delivered.

If the server had messages for the client, an alarm should be fired, since some issue happened on the wakeup platform.

In order to reduce network signalling, this PING should be only be shotted when a network connection had been done by any other application on the handset.

WAP Push

Use WAP Push to wake up handsets. The MSISDN will be retrieved by the central server after an integration with the carrier.

Because privacy reasons, the MSISND won't be shared with 3rd. party systems.

Abuse control

Add abuse control on the external APIs

Support multiple subnetworks

Many carriers have more than one private network since the have more smartphones than the class A segment.

Also, M2M solutions normally have their own private subnetwork.

The Notification Server shall be well integrated on multiple subnetworks carriers.

Increase delivery controls

Add support on AS API to know the delivery status of the messages.

Presence

Offer a presence API to AS.

Support delegated modes and HUB systems

Some carriers consider that multiple notification platforms should be interconnected between them.