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
RCC State	**GMM State (2G/3G)**	**Description**

RCC State	GMM State (2G/3G)	Description
		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. 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,...
RCC State	**GMM State (2G/3G)**	**Description**

RCC State	GMM State (2G/3G)	Description
		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]

Also there is more information about Simple PUSH API here: [https://wiki.mozilla.org/WebAPI/SimplePush] [https://wiki.mozilla.org/WebAPI/SimplePush]

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

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

- register
- unregister
- registrations

navigator.push.register

This method allows the application to register a new channel.

```
navigator.push.register()
```

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.push.register();
req.onsuccess = function(e) {
  alert("Received URL: " + req.result.pushEndpoint);
};
req.onerror = function(e) {
  alert("Error registering app");
}
```

navigator.push.unregister

This method allows the application to unregister a previously registered channel.

```
navigator.push.unregister(endPointURL);
```

After register the application into the Notification Server, all received notification through the given URL will be delivered to the WA registered channel.

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 handler:

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

Inside the msg you'll receive the pushEndpoint URL so an app can register as many channels as it wants and with this attribute has a chance to differenciate one from another.

The complete example:

```
var emailEndpoint, imEndpoint;

// The user has logged in, now's a good time to register the channels
MyAppFramework.addEventListener('user-login', function() {
  setupAppRegistrations();
});

function setupAppRegistrations() {
 // Issue a register() call
 // to register to listen for a notification,
 // you simply call push.register
 // Here, we'll register a channel for "email" updates.
 // Channels can be for anything the app would like to get notifications for.
 var reqEmail = navigator.push.register();
 reqEmail.onsuccess = function(e) {
   emailEndpoint = e.target.result.pushEndpoint;
   storeOnAppServer("email", emailEndpoint); // This is the "Hand wavey" way that th
                                             // sends the endPoint back to the AppSe
 }

 // We'll also register a second channel for "im", because we're social and all abou
 var reqIm = navigator.push.register();
 reqIm.onsuccess = function(e) {
   imEndpoint = e.target.result.pushEndpoint;
   storeOnAppServer("im", imEndpoint);
 }
}

// Once we've registered, the AppServer can send version pings to the EndPoint.
// This will trigger a 'push' message to be sent to this handler.
navigator.mozSetMessageHandler('push', function handlePushMessage(message) {
  if (message.pushEndpoint == emailEndpoint)   // Yay! New Email! Steve and blue car
    getNewEmailMessagesFromAppServer(message.version);
  else if (message.pushEndpoint == imEndpoint) // Yay! An IM awaits. I wonder if it
    getNewChatMessagesFromAppServer();
});

// to unregister, you simply call..
AppFramework.addEventListener('user-logout', function() {
  navigator.push.unregister(emailEndpoint);
  navigator.push.unregister(imEndpoint);
});
```

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:

- HTTP API: Through the HTTP transport protocol the NS will deliver some information about server status.

- WebSocket API: This is the most important one since all the communications 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 API

This channel only offers one method to get server information.

about

This method responds an HTML page with general information about the running server like number of connections, number of process running...

status

This method is used to check if the server is available or not. Is designed to be used by load balancers when the server is under maintance.

The server will responde 200 (OK) if the server is enabled or 503 (Under Maintance).

To set the server into maintance mode (tell the load balancer that the server is not available) is needed to send a SIGUSR1 signal to the proccess:

```
kill -SIGUSR1 <pid>
```

To set the server into normal mode (tell the load balancer that the server is available) is needed to send a SIGUSR2 signal to the proccess:

```
kill -SIGUSR2 <pid>
```

WebSocket API

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

The websocket API supports multiple subprotocols identified each one with it's name:

- push-notification

Simple protocol defined by Mozilla and Telefonica and described here: [https://wiki.mozilla.org/WebAPI/SimplePush/Protocol] [https://wiki.mozilla.org/WebAPI/SimplePush/Protocol].

- push-notification-binary
  Binary version of the push-notification protocol

- push-extended-notification
  Telefonica extended solution which provides more functionalities

WebSocket: push-notification

Also know as "Simple push protocol" and defined by Mozilla and Telefonica.

This protocol is based in the Thialfi protocol [http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/pubs/archive/37474.pdf]. described by Google.

Also you can read more about this protocol in the Mozilla Wiki: [https://wiki.mozilla.org/WebAPI/SimplePush/Protocol] [https://wiki.mozilla.org/WebAPI/SimplePush/Protocol].

In order to use this subprotocol, the "push-notification" string shall be sent into the websocket handsake headers.

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:

hello

With this method the device is able to register itself.

The device is responsible to give the server a valid UAID. If the provided UAID is not valid or is "null", the server will respond with a valid one.

In next connections the UAID given by the server SHALL be used.

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

If it's not the first connection, the device can send a list of registered channels in order to syncronize client and server data. This mechanism allows a way to recover channels after a server crash.

```
{
  messageType: "hello",
  uaid: "<a valid UAToken>",
  channelIDs: [<a list of channels to sync [OPTIONAL]>],
  wakeup_hostport: {
    ip: "<current device IP address>",
    port: "<TCP or UDP port in which the device is waiting for wake up notificati
  },
  mobilenetwork: {
    mcc: "<Mobile Country Code>",
    mnc: "<Mobile Network Code>"
  }
}
```

The wakeup_hostport 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 wakeup_hostport data so it's able to close the WebSocket channel to reduce signalling and battery consume.

The channelIDs array is sent by the client in order to synchronize server and client.

When the server receives a new hello message and the UAID provided by the client is a valid one (in other words, is the same returned to the client) the channelIDs list will be used to syncronize the server information with the client one.

For example, after a server crash, all client channels will be recovered with this simple method.

Another example, if the client uninstalled an app when the handset was offline, next time it connects will send tne channel list with one less, so the server will unregister this channel.

The server response can be one of these:

```
{
  messageType: "hello",
  uaid: "<a valid UAID>",
  status: 200
}
```

if it's connected through a permanent websocket, or:

```
{
  messageType: "hello",
  uaid: "<a valid UAID>",
  status: 201
}
```

if it's connected to a wakeup channel (UDP).

> **Note**
> This hello response differenciation is pending to change in order to use Websocket close status: on this Github Pull Request [https://github.com/telefonicaid/notification_server/issues/178].

```
{
  messageType: "hello",
  status: 4xx,
  reason: "<any reason>"
}
```

on any error case, like:

- 460: Error registering UAID

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

register

This method is used to register push channels. Each application can register as many channels as it wants. Each channel maintains an independent counter about the last version of the channel.

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 channel to receive Thialfi like notifications. A new endpoint URL will be delivered (through the WA-UA API).

No data is required at application level, only the UA client is responsible to generate a unique channelID for the handset. The channelID can be the same in different devices since the UAID will be used in the endpoint URL hash.

```
{
  messageType: "register",
  channelID: "<a new channelID>"
}
```

The server response can be:

```
{
  messageType: "register",
  status: 200,
  pushEndpoint: "<publicURL required to send notifications>",
  channelID: "<the channelID>"
}
```

```
{
  messageType: "register",
  status: 4xx,
  reason: "<any reason>"
}
```

on any error case, like:

- 457: Not valid channelID
- 408: Server is not ready yet

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

unregister

This method is used to unregister a push channel.

This shall be send to the notification server after a valid UA registration.

```
{
  messageType: "unregister",
  channelID: "<a new channelID>"
}
```

The server response can be:

```
{
  messageType: "register",
  channelID: "<a new channelID>"
  status: 202
}
```

```
{
  messageType: "register",
  status: 4xx,
  reason: "<any reason>"
}
```

on any error case, like:

- 408: Server is not ready yet

notification

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

All recieved notification(s) will have this structure:

```
{
  messageType: "notification",
  updates: [
    {
      channelID: "<channelID>",
      version: "<versionNumber>"
    },
    {
      channelID: "<channelID>",
      version: "<versionNumber>"
    },
    ...
  ]
}
```

On updates list, is returned all the list of pending notifications (last version of each channel)

desktop-notification

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

These notifications SHOULD be showed into the device notification area.

All recieved notification(s) will have this structure:

```
{
  messageType: "desktopNotification",
  updates: [
    {
      channelID: "<channelID>",
      _internal_id: "<id>",
      body: "<some text>"
    },
    {
      channelID: "<channelID>",
      _internal_id: "<id>",
      body: "<some text>"
    },
    ...
  ]
}
```

On updates list, is returned all the list of pending notifications (last version of each channel)

ack

For each received notification through notification, 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",
    updates: [
      {
        channelID: "<channelID>",
        version: "<versionNumber>"
      },
      {
        channelID: "<channelID>",
        version: "<versionNumber>"
      },
      ...
    ]
  }
```

Keep-alive algorithm

If it's needed a way to maintaint the socket open along time, a PING-PONG mechanism is also implemented.

The client sents an empty JSON object "{}" and the server will respond another empty object "{}" and/or a notification response (if pending notifications).

Wakeup websocket close

As explained before, when the client informs about the mobile network is in and the server has the required infrastructure in that mobile network, the websocket will be closed by the server after a predefined inactivity time (10 seconds).

When this timer fires, the websocket will be closed with the 4774 status code.

[WebSocket protocol] [http://tools.ietf.org/html/rfc6455#page-45].

API between the Application Server and the Notification Server

This chapter explains the different APIs available to third party servers

Generic API

about

This method responds an HTML page with general information about the running server like number of connections, number of process running...

status

This method is used to check if the server is available or not. Is designed to be used by load balancers when the server is under maintance.

The server will responde 200 (OK) if the server is enabled or 503 (Under Maintance).

To set the server into maintance mode (tell the load balancer that the server is not available) is needed to send a SIGUSR1 signal to the proccess:

```
kill -SIGUSR1 <pid>
```

To set the server into normal mode (tell the load balancer that the server is available) is needed to send a SIGUSR2 signal to the proccess:

```
kill -SIGUSR2 <pid>
```

Simple PUSH API

With this API the Application server is able to update version number to specified channel.

This is a simple HTTP API (PUT method).

This version accepts only one HTTP PUT method used to update version number of a channel. The following payload SHALL be POSTED to the endpointURL: `https://server:port/v1/notify/SOME_RANDOM_TOKEN`

```
version=<version_number>
```

and for desktop notifications:

```
body=<any text>[&ttl=<ttl>]
```

The server response can be one of the following:

- STATUS: 200
- STATUS: 404 = Channel not found
- STATUS: 404 = Bad body received
- STATUS: 404 = Bad version received

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.

channelID

This token identifies the user or group of users and on extended API SHALL be a secret but in simple push API (thialfi like) it's not needed to be a secret.

If this token is UNIQUE (and secret, of course) will identify a unique instance of the application related (normally) to one user. In this case the returned URL will be unique for this channelID. On simple push, each device with same channelID will receive a unique endpointURL.

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, ...

UAID

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.

endpointURL

Automatic generated token by the notification server which identifies the application + user/device 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 web-socket interfaces in order to pull all pending messages.

The WakeUp server offers a simple HTTP API:

status

This method is used to check if the server is available or not. Is designed to be used by load balancers when the server is under maintance.

The server will responde 200 (OK) if the server is enabled or 503 (Under Maintance).

To set the server into maintance mode (tell the load balancer that the server is not available) is needed to send a SIGUSR1 signal to the proccess:

```
kill -SIGUSR1 <pid>
```

To set the server into normal mode (tell the load balancer that the server is available) is needed to send a SIGUSR2 signal to the proccess:

```
kill -SIGUSR2 <pid>
```

Wake up method

To wakeup a device, sent a HTTP GET method to the WakeUp server with the ip and port parameters. The protocol is optional:

http(s)://server:port/?
ip=1.2.3.4&port=5678&protocol=[PROTOCOL_TCPv4 | PROTOCOL_UDPv4]

ip and port are mandatory and refers to the device ip and port where the Wakeup agent is listening.

protocol is optional (by default UDPv4 is used) the numeric values:

PROTOCOL_UDPv4 = 1

PROTOCOL_TCPv4 = 2

Chapter 4. Log traces

This chapter describes each log message, his ID and how to interpret it.

It covers NOTIFY, ERROR and CRITICAL traces. DEBUG and INFO traces are not documented.

Next sections describes each log level:
<xi:include></xi:include>
<xi:include></xi:include>
<xi:include></xi:include>

Command reference

Table of Contents

Name

load_mcc_mnc_onmongo — loads a mobile operator list into the central MongoDB

Synopsis

`awk -f scripts/load_mcc_mnc_onmongo.awk` [mcc_mnc_list.txt]

Description

Reads the mcc_mnc_list.txt file and loads all the mobile operators list into the central MongoDB database.

This command should only be used for the first system provision.

Name

add_wakeupserver_ip.sh — links a wakeup server IP and Port to a MCC-MNC pair

Synopsis

`scripts/add_wakeupserver_ip.sh` [mcc] [mnc] [wakeup server URL]

Description

Updates the operators collection linking a WakeUp server to the mcc-mnc pair.

This command should only be used each time the WakeUp server address is changed.

It's important to note that the MNC SHALL be 2 digits and MCC 3 digits. Fill with 0s if necessary

With an empty URL into the third parameter, the WakeUp server will be disabled into next runnings

Example

Enabling:

scripts/add_wakeupserver_ip.sh 214 07 http://1.2.3.4:4567

Disabling:

scripts/add_wakeupserver_ip.sh 214 07

Name

empty_mongo.sh — simple script to clean all MongoDB collections

Synopsis

```
scripts/empty_mongo.sh
```

Description

Drops all notification server collections from the MongoDB.

Use it if you really know what're you doing.

Example

scripts/empty_mongo.sh

Name

getloginfo — shows the detailed log trace description

Synopsis

`scripts/getloginfo` [log ID (in hexadecimal 0xABCD)]

Description

When the server emits a NOTIFICATION, ERROR or CRITICAL trace, you can see a unique ID in it.

This command allows you to recover more information about the error without need to refer the main document.

Example

scripts/getloginfo 0x1234