Darryn Campbell's Blog

Mobile computing and enterprise software development

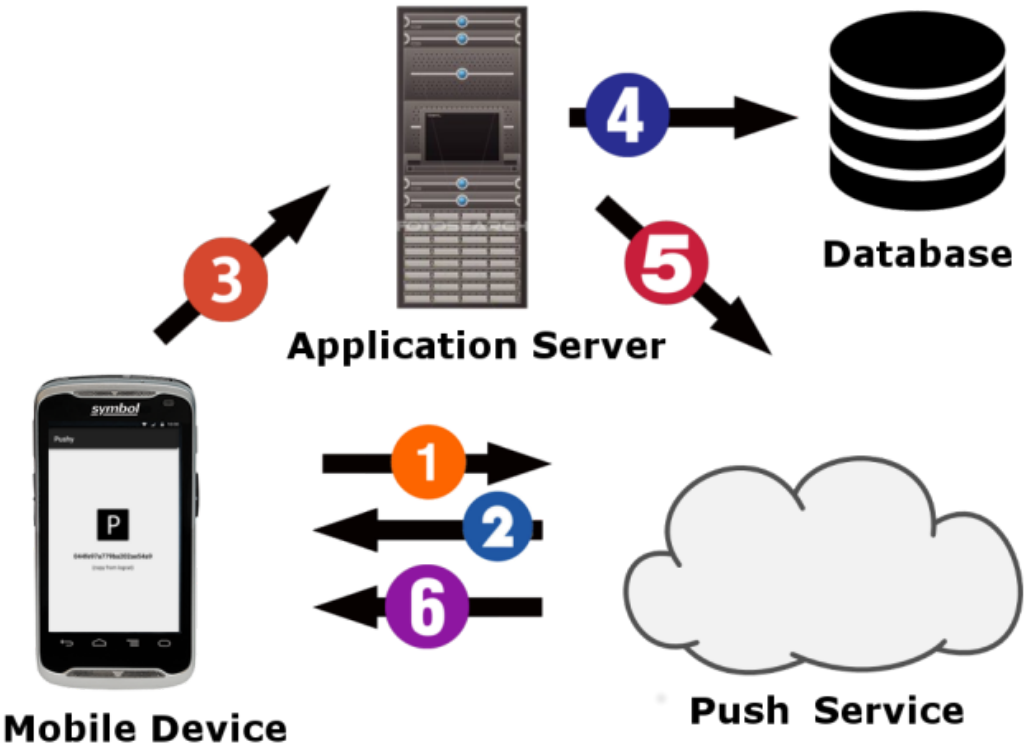
Push Messaging on AOSP Devices – Pushy.me

FEBRUARY 23, 2016JUNE 21, 2017 ~ DARRYN CAMPBELL

Please note: This blog has moved to my new site (http://www.darryncampbell.co.uk/?p=143), please update your bookmarks

Push Messaging

How push messaging works in general is common across all implementations, as shown in the following diagram:



1. The Android device requests a unique device ID from the push service

2. Once registered, the push service issues a unique ID to the device

3. After receiving the ID, the device will send this ID to the application server

4. The device ID should be stored for later use.

5. Whenever a push notification is required, the application server sends a message to the push service along with the device ID and application API key.

6. The push service delivers the message to the specified device.

The problem

Many of our Zebra mobile computers are shipped without Google Mobile Services (GMS) installed and these are referred to as "Android Open Source Platform" or AOSP devices. There are very good reasons to choose AOSP over GMS devices but customers doing so lose out on a wealth of Google provided services, most notably Maps, Location and Push messaging .

Push Messaging, the ability to send messages to mobile devices and have them instantly received, is almost exclusively implemented by Google Cloud Messaging (GCM) on Android. Unfortunately for customers with AOSP Android devices this means that most push implementations will **not** work for them, for example Pushlets (http://www.pushlets.com/), Azure push (https://azure.microsoft.com/en-

us/documentation/articles/mobile-services-javascript-backend-android-get-started-push/), Amazon notification system (https://aws.amazon.com/sns/), PhoneGap push (https://github.com/phonegap/phonegap-plugin-push) and the list goes on.

The customer's choice of push options for AOSP devices is very limited:

1. RhoConnect Push (http://docs.rhomobile.com/en/5.3.2/rhoconnect/push)
2. Pushy.me (https://pushy.me)
3. Building a custom solution from third party components, most commonly MQTT (https://developer.zebra.com/docs/DOC-2315) with some combination of RabbitMQ (https://www.rabbitmq.com/) or Mosquito (http://mosquitto.org/).

RhoConnect Push is only officially supported for applications built with the Rho framework so will not be an option for most deployments.

Building a custom solution whilst good in theory is a large barrier to adoption and prohibitive to many customers.

Enter Pushy.me, a third party push solution that runs well on both AOSP & GMS devices and provides a realistic alternative to GMS in the majority of deployments

What Pushy.me is

- A replacement for GCM downstream messaging (https://developers.google.com/cloud-messaging/downstream)
- A way to send a simple message from a server to a specific device
- A way to receive the push message on the device and act accordingly
- A hosted solution exposing a REST API
- A very similar user experience to GCM or APNS (Apple Push Notification Service)

What Pushy.me is not

- Free. There is a small monthly cost associated with each active device
- A replacement for GCM's upstream messaging or XMPP connection servers

Experimenting

If you just want to experiment with a working sample you can download the sample app from the Google Play store (https://play.google.com/store/apps/details?id=me.pushy.example) or build the app from the source in github (https://github.com/pushy-me/pushy-demo).

Once the application is running it will give you the device ID it received from the Pushy service (in this case 044fe97a779ba202ae54a9):

Send a test push message. As long as you have signed up for a Pushy.me account the easiest way to do this is via the Pushy.me website (https://pushy.me/docs/samples/android (https://pushy.me/docs/samples/android)), just enter the device ID and hit 'SEND TEST PUSH'. In a real deployment you will call a REST API from your application server similar to GCM downstream messaging.

Getting Started

First of all you need to create an account on the Pushy.me website (https://pushy.me/), you can sign up for a developer trial for free which allows you to test deployments for up to 100 devices. Once signed up you can create applications by going to the 'Account' tab and notice how you are given a unique API key for each application.

The Pushy website offers a great getting started guide (https://pushy.me/docs) and is organized into two workflows:

1. Customers starting out with a new application
2. Customers with an existing GCM implementation

Integrate the Pushy SDK

The Pushy SDK is available as a jar file and whilst it only exposes a small number of APIs to configure the service it nevertheless exposes enough to enable core push functionality. Copy the jar file to your Android project's libs/ folder and ensure it is built into the application

Full instructions can be found on the Pushy website (https://pushy.me/docs/sdk):

For Gradle, ensure the following is present in your build.gradle:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

Register Devices

The first client-side step in any push solution is for the device to register with the push server, in this case the Pushy.me server. This is handled by the Pushy API through a synchronous call:

```
// Acquire a unique registration ID for this device
String registrationId = Pushy.register(getApplicationContext());
```

Once you have a registration ID it needs to be sent to the application's backend server. Obviously since transferring the registration ID will involve a network call it is required to perform registration in a background thread. The combination of device ID and application API key are used when sending push messages to uniquely address a specific application on a specific device

Again, the online Pushy documentation gives an example of device registration (https://pushy.me/docs/registration)

Start listening and manifest changes

Having successfully registered the device, instruct the device to start listening to Push messages via the client API:

```
Pushy.listen(this);
```

It is recommended to have this call to listen() in the onCreate() method of your main launcher activity.

Pushy also requires additional manifest permissions, as follows:

As well as requiring changes to the manifest to register the receiver class, which will receive the push message.

All manifest changes (https://pushy.me/docs/manifest) as well as the API to start listening (https://pushy.me/docs/oncreate) are thoroughly documented on the Pushy.me website with full source code.

Receiving messages

The class which will receive the push message needs to extend an Android BroadcastReceiver (http://developer.android.com/reference/android/content/BroadcastReceiver.html) and is the class you registered in the previous section. This class' onReceive() method will be invoked regardless of whether your application is in the foreground or background:

```
package com.pushytest.testapp;
import android.app.Notification;
import android.app.NotificationManager;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class PushReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
       Toast.makeText(context, “Received Push: “ + intent.getStringExtra("message"
    }
}
```

This is a very simple receiver. A more featured example is given in the Pushy.me documentation for a broadcast receiver (https://pushy.me/docs/receiver)

Implementing / Updating the Backend

As previously mentioned Pushy acts as the push service to ensure messages sent to devices are received successfully and as far as network conditions allow, quickly.

It remains necessary to implement application logic on the server to:

- Have devices notify the application server of their device IDs and persist these IDs.
- Associate device IDs with users, or some other user friendly way of identifying devices
- Interface with the pushy service to send the required downstream messages.

Pushy has a number of sample code implementations that can be incorporated into existing application servers in PHP (https://pushy.me/docs/samples/php), Java (https://pushy.me/docs/samples/java) and .NET (https://pushy.me/docs/samples/dotnet). Since Pushy exposes a REST API the call itself is just an HTTP post, for testing purposes anything that will send an HTTP POST can be used and personally I like the Chrome Postman extension (https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop)

The raw HTTP Post is as follows:

```
POST /push?api_key=
HTTP/1.1
Host: https://pushy.me
Content-Type: application/json
Connection: close
Content-Length: 82

{"registration_ids":["a6345e3278adc55d3474f5"], "data":{"message":"Hello World!"}}
```

Integration with GCM

One of the main benefits of Pushy for many customers will be the ability to quickly update an existing GCM solution to leverage Pushy as the transport medium. Pushy documentation (https://pushy.me/docs/registration) stresses this ease of transition as each stage of implementation in the guide is accompanied by a corresponding 'GCM Migration' section.

Register Devices

Find

```
String registrationId = GoogleCloudMessaging.getInstance(context).register(SENDER_
```

Replace with

```
String registrationId = Pushy.register(getApplicationContext());
```

Start listening and manifest changes

Find

```
super.onCreate(savedInstanceState);
```

Append

```
Pushy.listen(this);
```

Receiving messages

Update the existing GCM broadcast receiver as follows:

- Avoid checking the intent action for com.google.android.c2dm.intent.RECEIVE
- Link the receiver class to the AndroidManifest.xml declaration

Sending a message from the application server

Find

```
https://android.googleapis.com/gcm/send
```

Replace with

```
https://pushy.me/push?api_key=
```

Staging your device

In terms of delivering a Pushy enabled application to your device there are no special steps; the Pushy API is built into your Android APK so can be deployed by any MDM solution, manually loaded, delivered via the Play Store or through AppGallery.

On the client Pushy leverages MQTT to listen for incoming push connections, it is therefore necessary to ensure port 1883 is opened and the application is able to communicate with the backend Pushy service (https://pushy.me (https://pushy.me)). Since communication is over HTTPS port 443 needs to also be open.

On Premises deployments

Whilst available as a hosted solution, Pushy does support on-premises deployments. Customers interested in on-premises deployments are encouraged to contact Pushy support (https://pushy.me/support) directly.

Other Reading

Whilst the primary reason for most Zebra customers to choose a non-GCM push solution will be the need to accommodate AOSP devices, further justification can be found in the following blog (http://eladnava.com/google-cloud-messaging-extremely-unreliable/). Disclaimer: The linked blog is written by a Pushy developer.

POSTED IN PUSH, ZEBRA_TECHNOLOGIES
AOSP PUSH PUSHY.ME