GCM Advanced Topics

This document covers advanced topics for GCM.

QUICKVIEW

- Learn more about GCM advanced features.

IN THIS DOCUMENT

Lifetime of a Message

When a 3rd-party server posts a message to GCM and receives a message ID back, it does not mean that the message was already delivered to the device. Rather, it means that it was accepted for delivery. What happens to the message after it is accepted depends on many factors.

In the best-case scenario, if the device is connected to GCM, the screen is on, and there are no throttling restrictions (see Throttling (#throttling)), the message will be delivered right away.

If the device is connected but idle, the message will still be delivered right away unless the `delay_while_idle` flag is set to true. Otherwise, it will be stored in the GCM servers until the device is awake. And that's where the `collapse_key` flag plays a role: if there is already a message with the same collapse key (and registration ID) stored and waiting for delivery, the old message will be discarded and the new message will take its place (that is, the old message will be collapsed by the new one). However, if the collapse key is not set, both the new and old messages are stored for future delivery.

> **Note:** There is a limit on how many messages can be stored without collapsing. That limit is currently 100. If the limit is reached, all stored messages are discarded. Then when the device is back online, it receives a special message indicating that the limit was reached. The application can then handle the situation properly, typically by requesting a full sync.

If the device is not connected to GCM, the message will be stored until a connection is established (again respecting the collapse key rules). When a connection is established, GCM will deliver all pending messages to the device, regardless of the `delay_while_idle` flag. If the device never gets connected again (for instance, if it was factory reset), the message will eventually time out and be discarded from GCM storage. The default timeout is 4 weeks, unless the `time_to_live` flag is set.

Finally, when GCM attempts to deliver a message to the device and the application was uninstalled, GCM will discard that message right away and invalidate the registration ID. Future attempts to send a message to that device will get a `NotRegistered` error. See How Unregistration Works (#unreg) for more information.

Although is not possible to track the status of each individual message, the Google APIs Console stats are broken down by messages sent to device, messages collapsed, and messages waiting for delivery.

Overview
Games
Location
Google+
Google Maps
Authorization
Google Play Services
Google Play In-app Billing
Google Cloud Messaging

Throttling

To prevent abuse (such as sending a flood of messages to a device) and to optimize for the overall network efficiency and battery life of devices, GCM implements throttling of messages using a token bucket scheme. Messages are throttled on a per application and per collapse key (#collapsible) basis (including non-collapsible messages). Each application collapse key is granted some initial tokens, and new tokens are granted periodically thereafter. Each token is valid for a single message sent to the device. If an application collapse key exhausts its supply of available tokens, new messages are buffered in a pending queue until new tokens become available at the time of the periodic grant. T hrottling in between periodic grant intervals may add to the latency of message delivery for an applicati llapse key that sends a large number of messages within a short period of time. Messages in the pendir eue of an application collapse key may be delivered before the time of the next periodic grant, if they are pi icked with messages belonging to a non-throttled category by GCM for netwc

Keeping t… …Sync

Whenever the a… ….android.c2dm.intent.REGISTRATION intent with a registratio… …future use, pass it to the 3rd-party server to complete the registration, ar… …completed the registration. If the server fails to complete the registration, it … …GCM.

There are also … …ecial care:

- Application up…
- Backup and res…

When an applic… …ate its existing registration ID, as it is not guaranteed to work with the new v… …e method called when the application is updated, the best way to achieve this validation is by storing the c… …application version when a registration ID is stored. Then when the application is started, compare the s… …ed value with the current application version. If they do not match, invalidate the stored data and start the registration process again.

Similarly, you should not save the registration ID when an application is backed up. This is because the registration ID could become invalid by the time the application is restored, which would put the application in an invalid state (that is, the application thinks it is registered, but the server and GCM do not store that registration ID anymore—thus the application will not get more messages).

Canonical IDs

On the server side, as long as the application is behaving well, everything should work normally. However, if a bug in the application triggers multiple registrations for the same device, it can be hard to reconcile state and you might end up with duplicate messages.

GCM provides a facility called "canonical registration IDs" to easily recover from these situations. A canonical registration ID is defined to be the ID of the last registration requested by your application. This is the ID that the server should use when sending messages to the device.

If later on you try to send a message using a different registration ID, GCM will process the request as usual, but it will include the canonical registration ID in the `registration_id` field of the response. Make sure to replace the registration ID stored in your server with this canonical ID, as eventually the ID you're using will stop working.

Automatic Retry Using Exponential Back-Off

When the application receives a `com.google.android.c2dm.intent.REGISTRATION` intent with the `error` extra set as `SERVICE_NOT_AVAILABLE`, it should retry the failed operation (register or unregister).

In the simplest case, if your application just calls `register` and GCM is not a fundamental part of the application, the application could simply ignore the error and try to register again the next time it starts. Otherwise, it should retry the previous operation using exponential back-off. In exponential back-off, each time there is a failure, it should wait twice the previous amount of time before trying again. If the register (or unregister) operation was synchronous, it could be retried in a simple loop. However, since it is asynchronous, the best approach is to schedule a pending intent to retry the operation. The following steps describe how to implement this in the `MyIntentService` example used above:

1. Create a random token to verify the origin of the retry intent:

```
private static final String TOKEN =
        Long.toBinaryString(new Random().nextLong());
```

2. Change the `handleRegistration()` method so it creates the pending intent when appropriate:

```
...
if (error != null) {
 if ("SERVICE_NOT_AVAILABLE".equals(error)) {
```

```
    long backoffTimeMs = // get back-off time from shared preferences
    long nextAttempt = SystemClock.elapsedRealtime() + backoffTimeMs;
    Intent retryIntent = new Intent("com.example.gcm.intent.RETRY");
    retryIntent.putExtra("token", TOKEN);
    PendingIntent retryPendingIntent =
        PendingIntent.getBroadcast(context, 0, retryIntent, 0);
    AlarmManager am = (AlarmManager)
        context.getSystemService(Context.ALARM_SERVICE);
    am.set(AlarmManager.ELAPSED_REALTIME, nextAttempt, retryPendingIntent);
    backoffTimeMs *= 2; // Next retry should wait longer.
    // update back-off time on shared preferences
  } else {
    // Unrecoverable error, log it
    Log.i(TAG, "Received error: " + error);
}
...
```

The back-off time is stored in a shared preference. This ensures that it is persistent across multiple activity launches. The name of the intent does not matter, as long as the same intent is used in the following steps.

3. Change the `onHandleIntent()` method adding an `else if` case for the retry intent:

```
...
} else if (action.equals("com.example.gcm.intent.RETRY")) {
    String token = intent.getStringExtra("token");
    // make sure intent was generated by this class, not by a malicious app
    if (TOKEN.equals(token)) {
        String registrationId = // get from shared properties
        if (registrationId != null) {
        // last operation was attempt to unregister; send UNREGISTER intent agai
    } else {
        // last operation was attempt to register; send REGISTER intent again
    }
}
...
```

4. Create a new instance of `MyReceiver` in your activity:

```
private final MyBroadcastReceiver mRetryReceiver = new MyBroadcastReceiver();
```

5. In the activity's `onCreate()` method, register the new instance to receive the `com.example.gcm.intent.RETRY` intent:

```
...
IntentFilter filter = new IntentFilter("com.example.gcm.intent.RETRY");
filter.addCategory(getPackageName());
registerReceiver(mRetryReceiver, filter);
...
```

> **Note:** You must dynamically create a new instance of the broadcast receiver since the one defined by the manifest can only receive intents with the `com.google.android.c2dm.permission.SEND` permission. The permission `com.google.android.c2dm.permission.SEND` is a system permission and as such it cannot be granted to a regular application.

6. In the activity's `onDestroy()` method, unregister the broadcast receiver:

```
unregisterReceiver(mRetryReceiver);
```

How Unregistration Works

There are two ways to unregister a device from GCM: manually and automatically.

An Android application can manually unregister itself by issuing a `com.google.android.c2dm.intent.UNREGISTER` intent, which is useful when the application offers a logoff feature (so it can unregister on logoff and register again on logon). See the Architectural Overview (gcm.html#unregistering) for more discussion of this topic. This is the sequence of events when an application unregisters itself:

1. The application issues a `com.google.android.c2dm.intent.UNREGISTER` intent, passing the package name as an extra.
2. When the GCM server is done with the unregistration, it sends a `com.google.android.c2dm.intent.REGISTRATION` intent with the `unregistered` extra set.
3. The application then must contact the 3rd-party server so it can remove the registration ID.
4. The application should also clear its registration ID.

An application can be automatically unregistered after it is uninstalled from the device. However, this process does not happens right away, as Android does not provide an uninstall callback. What happens in this scenario is as follows:

1. The end user uninstalls the application.
2. The 3rd-party server sends a message to GCM server.
3. The GCM server sends the message to the device.
4. The GCM client receives the message and queries Package Manager about whether there are broadcast receivers configured to receive it, which returns `false`.
5. The GCM client informs the GCM server that the application was uninstalled.
6. The GCM server marks the registration ID for deletion.
7. The 3rd-party server sends a message to GCM.
8. The GCM returns a `NotRegistered` error message to the 3rd-party server.
9. The 3rd-party deletes the registration ID.

> **Note:** The GCM client is the Google Cloud Messaging framework present on the device.

Note that it might take a while for the registration ID be completely removed from GCM. Thus it is possible that messages sent during step 7 above gets a valid message ID as response, even though the message will not be delivered to the device. Eventually, the registration ID will be removed and the server will get a `NotRegistered` error, without any further action being required from the 3rd-party server (this scenario happens frequently while an application is being developed and tested).

Send-to-Sync vs. Messages with Payload

Every message sent in GCM has the following characteristics:

- It has a payload limit of 4096 bytes.
- By default, it is stored by GCM for 4 weeks.

But despite these similarities, messages can behave very differently depending on their particular settings. One major distinction between messages is whether they are collapsed (where each new message replaces the preceding message) or not collapsed (where each individual message is delivered). Every message sent in GCM is either a "send-to-sync" (collapsible) message or a "message with payload" (non-collapsible message). These concepts are described in more detail in the following sections.

Send-to-sync messages

A send-to-sync (collapsible) message is often a "tickle" that tells a mobile application to sync data from the server. For example, suppose you have an email application. When a user receives new email on the server, the server pings the mobile application with a "New mail" message. This tells the application to sync to the server to pick up the new email. The server might send this message multiple times as new mail continues to accumulate, before the application has had a chance to sync. But if the user has received 25 new emails, there's no need to preserve every "New mail" message. One is sufficient. Another example would be a sports application that

updates users with the latest score. Only the most recent message is relevant, so it makes sense to have each new message replace the preceding message.

The email and sports applications are cases where you would probably use the GCM `collapse_key` parameter. A *collapse key* is an arbitrary string that is used to collapse a group of like messages when the device is offline, so that only the most recent message gets sent to the client. For example, "New mail," "Updates available," and so on

GCM allows a maximum of 4 different collapse keys to be used by the GCM server at any given time. In other words, the GCM server can simultaneously store 4 different send-to-sync messages, each with a different collapse key. If you exceed this number GCM will only keep 4 collapse keys, with no guarantees about which ones they will be.

Messages with payload

Unlike a send-to-sync message, every "message with payload" (non-collapsible message) is delivered. The payload the message contains can be up to 4kb. For example, here is a JSON-formatted message in an IM application in which spectators are discussing a sporting event:

```
{
  "registration_id" : "APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx...",
  "data" : {
    "Nick" : "Mario",
    "Text" : "great match!",
    "Room" : "PortugalVSDenmark",
  },
}
```

A "message with payload" is not simply a "ping" to the mobile application to contact the server to fetch data. In the aforementioned IM application, for example, you would want to deliver every message, because every message has different content. To specify a non-collapsible message, you simply omit the `collapse_key` parameter. Thus GCM will send each message individually. Note that the order of delivery is not guaranteed.

GCM will store up to 100 non-collapsible messages. After that, all messages are discarded from GCM, and a new message is created that tells the client how far behind it is. The message is delivered through a regular `com.google.android.c2dm.intent.RECEIVE` intent, with the following extras:

- `message_type`—The value is always the string "deleted_messages".
- `total_deleted`—The value is a string with the number of deleted messages.

The application should respond by syncing with the server to recover the discarded messages.

Which should I use?

If your application does not need to use non-collapsible messages, collapsible messages are a better choice from a performance standpoint, because they put less of a burden on the device battery.

Setting an Expiration Date for a Message

The Time to Live (TTL) feature lets the sender specify the maximum lifespan of a message using the `time_to_live` parameter in the send request. The value of this parameter must be a duration from 0 to 2,419,200 seconds, and it corresponds to the maximum period of time for which GCM will store and try to deliver the message. Requests that don't contain this field default to the maximum period of 4 weeks.

Here are some possible uses for this feature:

- Video chat incoming calls
- Expiring invitation events
- Calendar events

Background

GCM will usually deliver messages immediately after they are sent. However, this might not always be possible. For example, the device could be turned off, offline, or otherwise unavailable. In other cases, the sender itself might request that messages not be delivered until the device becomes active by using the `delay_while_idle` flag. Finally, GCM might intentionally delay messages to prevent an application from consuming excessive resources and negatively impacting battery life.

When this happens, GCM will store the message and deliver it as soon as it's feasible. While this is fine in most cases, there are some applications for which a late message might as well never be delivered. For example, if the message is an incoming call or video chat notification, it will only be meaningful for a small period of time before the call is terminated. Or if the message is an invitation to an event, it will be useless if received after the event has ended.

Another advantage of specifying the expiration date for a message is that GCM will never throttle messages with a `time_to_live` value of 0 seconds. In other words, GCM will guarantee best effort for messages that must be delivered "now or never." Keep in mind that a `time_to_live` value of 0 means messages that can't be delivered immediately will be discarded. However, because such messages are never stored, this provides the best latency for sending notifications.

Here is an example of a JSON-formatted request that includes TTL:

```
{
  "collapse_key" : "demo",
  "delay_while_idle" : true,
  "registration_ids" : ["xyz"],
  "data" : {
    "key1" : "value1",
    "key2" : "value2",
  },
  "time_to_live" : 3
},
```

Receiving Messages from Multiple Senders

GCM allows multiple parties to send messages to the same application. For example, suppose your application is an articles aggregator with multiple contributors, and you want each of them to be able to send a message when they publish a new article. This message might contain a URL so that the application can download the article. Instead of having to centralize all sending activity in one location, GCM gives you the ability to let each of these contributors send its own messages.

To make this possible, all you need to do is have each sender generate its own project number. Then include those IDs in the sender field, separated by commas, when requesting a registration. Finally, share the registration ID with your partners, and they'll be able to send messages to your application using their own authentication keys.

This code snippet illustrates this feature. Senders are passed as an intent extra in a comma-separated list:

```
Intent intent = new Intent(GCMConstants.INTENT_TO_GCM_REGISTRATION);
intent.setPackage(GSF_PACKAGE);
intent.putExtra(GCMConstants.EXTRA_APPLICATION_PENDING_INTENT,
        PendingIntent.getBroadcast(context, 0, new Intent(), 0));
String senderIds = "968350041068,652183961211";
intent.putExtra(GCMConstants.EXTRA_SENDER, senderIds);
ontext.startService(intent);
```

Note that there is limit of 100 multiple senders.