GCM Cloud Connection Server

> **Note:** To try out this feature, sign up using this form (https://services.google.com/fb/forms/gcm/).

The GCM Cloud Connection Server (CCS) allows third party servers to communicate with Android devices by establishing a persistent TCP connection with Google servers using the XMPP protocol. This communication is asynchronous and bidirectional.

You can continue to use the HTTP request mechanism to send messages to GCM servers, side-by-side with CCS which uses XMPP. Some of the benefits of CCS include:

- The asynchronous nature of XMPP allows you to send more messages with fewer resources.
- Communication is bidirectional—not only can the server send messages to the device, but the device can send messages back to the server.
- You can send messages back using the same connection used for receiving, thereby improving battery life.

The upstream messaging (device-to-cloud) feature of CCS is part of the Google Play services platform. Upstream messaging is available through the `GoogleCloudMessaging` (/reference/com/google/android/gms/gcm/GoogleCloudMessaging.html) APIs. To use upstream messaging and the new streamlined registration process, you must set up (/google/play-services/setup.html) the Google Play services SDK.

QUICKVIEW

- Get an introduction to key CCS terms and concepts.
- Learn how to send and receive both upstream and downstream messages in CCS.

IN THIS DOCUMENT

SEE ALSO

Getting Started
CCS and User Notifications
Signup Form

> **Note:** For an example of an XMPP server, see GCM Server (server.html#xmpp).

CCS vs. GCM HTTP

CCS messaging differs from GCM HTTP messaging in the following ways:

- Upstream/Downstream messages
  - GCM HTTP: Downstream only: cloud-to-device.
  - CCS: Upstream and downstream (device-to-cloud, cloud-to-device).
- Asynchronous messaging
  - GCM HTTP: 3rd-party servers send messages as HTTP POST requests and wait for a response. This mechanism is synchronous and causes the sender to block before sending another message.
  - CCS: 3rd-party servers connect to Google infrastructure using a persistent XMPP connection and send/receive messages to/from all their devices at full line speed. CCS sends acknowledgements or failure notifications (in the form of special ACK and NACK JSON-encoded XMPP messages) asynchronously.
- JSON
  - GCM HTTP: JSON messages sent as HTTP POST.
  - CCS: JSON messages encapsulated in XMPP messages.

This document describes how to use CCS. For general concepts and information on how to use GCM HTTP, see the GCM Architectural Overview (gcm.html).

How to Use CCS

GCM Cloud Connection Server (CCS) is an XMPP endpoint, running on `http://gcm.googleapis.com` port 5235.

CCS requires a Transport Layer Security (TLS) connection. That means the XMPP client must initiate a TLS connection. For example in smack, you would call `setSocketFactory(SSLSocketFactory)`, similar to “old

style SSL" XMPP connections and https.

CCS requires a SASL PLAIN authentication mechanism using `<your_GCM_Sender_Id>@gcm.googleapis.com` (GCM sender ID) and the API key as the password, where the sender ID and API key are the same as described in Getting Started (gs.html).

You can use most XMPP libraries to interact with CCS.

Sending messages

The following snippets illustrate how to perform authentication in CCS.

Client

```
<stream:stream to="gcm.googleapis.com"
        version="1.0" xmlns="jabber:client"
        xmlns:stream="http://etherx.jabber.org/streams"/>
```

Server

```
<str:features xmlns:str="http://etherx.jabber.org/streams">
 <mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
   <mechanism>X-OAUTH2</mechanism>
   <mechanism>X-GOOGLE-TOKEN</mechanism>
   <mechanism>PLAIN</mechanism>
 </mechanisms>
</str:features>
```

Client

```
<auth mechanism="PLAIN"
xmlns="urn:ietf:params:xml:ns:xmpp-sasl">MTI2MjAwMzQ3OTMzQHByb2plY3RzLmdjbS5hb
mRyb2lkLmNvbQAxMjYyMDAzNDc5FzNAcHJvamVjdHMtZ2EtLmFuZHJvaWQuY29tAEFJe
mFTeUIzcmNaTmtmbnFLZEZiOW1oekNCaVlwT1JEQTJKV1d0dw==</auth>
```

Server

```
<success xmlns="urn:ietf:params:xml:ns:xmpp-sasl"/>
```

Message Format

CCS uses normal XMPP `<message>` stanzas. The body of the message must be:

```
<gcm xmlns:google:mobile:data>
    JSON payload
</gcm>
```

Overview

Games

Location

Google+

Google Maps

The JSON payload for server-to-device is similar to what the GCM http endpoint uses, with these exceptions:

- There is no support for multiple recipients.
- `to` is used instead of `registration_ids`.
- CCS adds the field `message_id`, which is required. This ID uniquely identifies the message in an XMPP connection. The ACK or NACK from CCS uses the `message_id` to identify a message sent from 3rd-party servers to CCS. Therefore, it's important that this `message_id` not only be unique, but always present.
- For ACK/NACK messages that are special control messages, you also need to include a `message_type` field in

the JSON message. For example:

```
message_type = ('ack' OR 'nack');
```

Authorization
Google Play Services
Google Play In-app Billing
Google Cloud Messaging
Getting Started
Architectural Overview
Cloud Connection Server
User Notifications
GCM Client
GCM Server
Advanced Topics
Migration
Reference
Google Play Distribution

For each message a device sends to the serv u need to send an ACK message. You never need to send a NACK messag ssage, CCS will just resend it.

CCS also send o-device message. If you do not receive either, it means that the TCP connectio peration and your server needs to resend the messages.

Message Exa

Here is an XMF ssage from a 3rd-party server to CCS:

```
<message i
  <gcm xml
  {
      "to"                                  )" replaces "registration_ids"
      "mes                                   // new required field
      "dat
      {
          "hello":"world",
      }
      "time_to_live":"600",
      "delay_while_idle": true/false
  }
  </gcm>
</message>
```

Here is an XMPP stanza containing the ACK/NACK message from CCS to 3rd-party server:

```
<message id="">
  <gcm xmlns="google:mobile:data">
  {
      "from":"REGID",
      "message_id":"m-1366082849205"
      "message_type":"ack"
  }
  </gcm>
</message>

<message id="">
  <gcm xmlns="google:mobile:data">
  {
      "from":"REGID",
      "message_id":"m-1366082849205"
      "error": ERROR_CODE,
      "message_type":"nack"
  }
  </gcm>
</message>
```

Upstream Messages

Using CCS and the GoogleCloudMessaging (/reference/com/google/android/gms/gcm/GoogleCloudMessaging.html) API, you can send messages from a user's device to the cloud.

Here is how you send an upstream message using the GoogleCloudMessaging (/reference/com/google/android/gms/gcm/GoogleCloudMessaging.html) API. For a complete example, see Getting Started

(gs.html#gs_example):

```
GoogleCloudMessaging gcm = GoogleCloudMessaging.get(context);
String GCM_SENDER_ID = "Your-Sender-ID";
AtomicInteger msgId = new AtomicInteger();
String id = Integer.toString(msgId.incrementAndGet());
Bundle data = new Bundle();
// Bundle data consists of a key-value pair
data.putString("hello", "world");
// "time to live" parameter
int ttl = [0 seconds, 4 weeks]

gcm.send(GCM_SENDER_ID + "@gcm.googleapis.com", id, ttl, data);
```

This call generates the necessary XMPP stanza for sending the upstream message. The message goes from the app on the device to CCS to the 3rd-party server. The stanza has the following format:

```
<message id="">
  <gcm xmlns="google:mobile:data">
  {
      "category":"com.example.yourapp", // to know which app sent it
      "data":
      {
          "hello":"world",
      },
      "message_id":"m-123",
      "from":"REGID"
  }
  </gcm>
</message>
```

Here is the format of the ACK expected by CCS from 3rd-party servers in response to the above message:

```
<message id="">
  <gcm xmlns="google:mobile:data">
  {
      "to":"REGID",
      "message_id":"m-123"
      "message_type":"ack"
  }
  </gcm>
</message>
```

Flow Control

Every message sent to CCS receives either an ACK or a NACK response. Messages that haven't received one of these responses are considered pending. If the pending message count reaches 1000, the 3rd-party server should stop sending new messages and wait for CCS to acknowledge some of the existing pending messages.

Conversely, to avoid overloading the 3rd-party server, CCS will stop sending if there are too many unacknowledged messages. Therefore, the 3rd-party server should "ACK" received messages as soon as possible to maintain a constant flow of incoming messages. The aforementioned pending message limit doesn't apply to these ACKs. Even if the pending message count reaches 1000, the 3rd-party server should continue sending ACKs to avoid blocking delivery of new messages.

ACKs are only valid within the context of one connection. If the connection is closed before a message can be ACKed, the 3rd-party server should wait for CCS to resend the message before ACKing it again.