

Google Cloud Pub/Sub

APR 8, 2015

Google [released](#) the beta version of their [Google Cloud Pub/Sub API](#) in early March. For a beta version, it comes with pretty generous quota: 10,000 messages per second across up to 10,000 topics. I built a very simple demo app to and describe the implementation choices using [Integration Patterns](#).

Cloud Pub/Sub = Publish-Subscribe Channel and Competing Consumers

At the heart of the Cloud Pub/Sub is a classic [Publish-Subscribe Channel](#), which delivers a single message published to it to multiple subscribers. One advantage of this pattern is that adding subscribers is side-effect free, which is one reason a [Publish-Subscribe Channel](#) is sometimes considered more loosely coupled than a [Point-to-Point Channel](#), which delivers a message to exactly one subscriber. Adding consumers to a [Point-to-Point Channel](#) results in [Competing Consumers](#) and thus has a strong side effect.

Upon closer inspection, we realize that the Google Cloud Pub/Sub API actually implements both the [Publish-Subscribe Channel](#) and the [Competing Consumers](#) patterns. This is the reason Google can cite a variety of use cases in the [Overview Document](#):

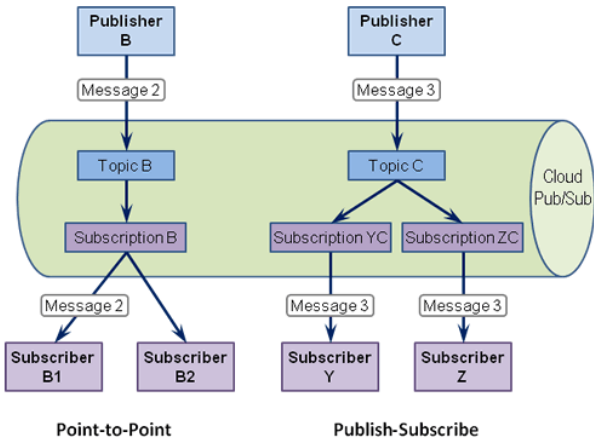
- Balancing workload
- Distributing event notifications
- Logging to multiple systems
- Data streaming
- ...

Topics and Subscriptions

Interestingly, the [Competing Consumers](#) aspect is not (yet) shown in the [Overview Diagram](#). The diagram makes clear, though, that the Pub/Sub implementation separates the concepts of *topic* and *subscription*:

- **Topics** represent the pub-sub aspect of the system: if multiple subscribers subscribe to the same topic, each of them will receive a copy of the message published to the topic.
- **Subscriptions** represent the competing consumers aspect of the system: if multiple subscribers pull messages from the same subscription, only one of them will receive the message.

I depicted this behavior in a modified version of Google's image:



In this example *Message 2* is only received by *Subscriber B1* as both *Subscriber B1* and *Subscriber B2* compete for the message from the same subscription. In contrast, both *Subscriber Y* and *Subscriber Z* each receive a copy of *Message 3* as they are subscribing to the same *Topic C*, but through separate subscriptions.

Separating topics and subscriptions is an elegant way to provide both [Publish-Subscribe Channel](#) and [Point-to-Point Channel](#) semantics through a single API. It also clarifies why Google separated the two concepts: each subscription represents a message queue that is fed by the pub-sub topics. If multiple subscriptions are attached to one topic, each of them receives a copy of each message.

REST API

Cloud Pub/Sub is accessed through a set of REST API's, which, once you are familiar with the concept, are fairly natural. A small, but important detail are the resource names for topics and subscriptions, which must follow the following format: `projects/project-identifier/collection/resource-name` where *collection* can be *topics* or *subscriptions*. Following this scheme, listing defined subscriptions can be as simple as making the following REST call, assuming you have authenticated via OAuth and your app is called `eaipubsub`:

GET `https://pubsub.googleapis.com/v1beta2/projects/eaipubsub/subscriptions`

The response is a simple JSON structure:

```
{
  "subscriptions": [
    {
      "name": "projects/eaipubsub/subscriptions/mysubscription",
```

ABOUT ME



Hi, I am [Gregor Hohpe](#), co-author of the book [Enterprise Integration Patterns](#). I like to work on and [write about](#) asynchronous messaging systems, service-oriented architectures, and all sorts of enterprise computing and architecture topics. I am also the Chief Architect at [Allianz SE](#), one of the largest insurance companies in the world.

TOPICS

[ALL RAMBLINGS](#)

[Architecture](#) (12) [Cloud](#) (3)
[Conversations](#) (8) [Design](#) (26)
[Events](#) (27) [Gregor](#) (4)
[Integration](#) (13)
[Messaging](#) (11) [Modeling](#) (5)
[Patterns](#) (7) [Visualization](#) (3)
[WebServices](#) (5) [Writing](#) (11)

POPULAR RAMBLINGS

[Starbucks does not use 2-phase commit](#)
[Is this architecture?](#)
[RESTful Conversations](#)
[Hub and Spoke](#)
[Correlation and Conversations](#)
[Diagram-driven Design](#)

RECENT

[25 Years of OOP](#)
[How to Scale an Organization? The same way you scale a system!](#)
[The Architect's Penthouse Bookshelf](#)
[If software eats the world, you better use version control!](#)
[If you never kill anything, you will live among zombies. And they will eat your brain.](#)
[Movie Star Architects](#)
[Virtualization Matryoshka](#)
[Is This Architecture? Look for Decisions!](#)
[Same Old Architecture - Best of Ramblings](#)

```

        "topic": "projects/eaipubsub/topics/sometopic",
        "pushConfig": { },
        "ackDeadlineSeconds": 60
    }
}
}

```

Java Client API

As with most Cloud API's, Cloud Pub/Sub comes with a variety of client libraries. I tend to have mixed feelings with generated client libraries for REST API's as I fear they negate some of the dynamic aspects and simplicity of the REST style. For building little sample applications, though, they do provide a nice starting point. I set out to build simple console applications that setup topics and subscriptions and then publish and consume messages. Wrapping all this into a little helper class resulted in a little less than 100 lines of code, which is not too bad. However, since Google recently open sourced Stubby aka [gRPC](#), which also includes client stub generators, one wonders whether these could also be used for the Cloud API's.

If you setup your project from scratch, you'll need to import about 10 libraries as described in the `readme.html` file in the download.

Authenticating

Working with a stand-alone application requires you to authenticate using private keys:

```

private void createClient(String private_key_file, String email) throws IOException, GeneralSecurityException {
    HttpTransport transport = GoogleNetHttpTransport.newTrustedTransport();
    GoogleCredential credential = new GoogleCredential.Builder()
        .setTransport(transport)
        .setJsonFactory(JSON_FACTORY)
        .setServiceAccountScopes(PubsubScopes.all())
        .setServiceAccountId(email)
        .setServiceAccountPrivateKeyFromP12File(new File(private_key_file))
        .build();
    pubsub = new Pubsub.Builder(transport, JSON_FACTORY, credential)
        .setApplicationName("eaipubsub")
        .build();
}

```

You would want to use a custom `HttpRequestInitializer` to enable automatic retry upon failures for anything but trivial demos.

Creating Topics

Most of the methods throw exceptions based on error HTTP responses, such as creating objects that already exist. You can swallow the exceptions or check for existence first like I did for topic creation:

```

Topic createTopic(String topicName) throws IOException {
    String topic = getTopic(topicName);
    Pubsub.Projects.Topics topics = pubsub.projects().topics();
    ListTopicsResponse list = topics.list(project).execute();
    if (list.getTopics() == null || !list.getTopics().contains(new Topic().setName(topic))) {
        Topic newTopic = topics.create(topic, new Topic()).execute();
        return newTopic;
    } else {
        return new Topic().setName(topic);
    }
}

```

As a topic only contains the name, I simply instantiate a new one if the chosen topic already exists. `getTopic` does the magic translation from `sometopic` to `/projects/eaipubsub/topics/sometopic`. Many methods are available only through the `projects()` collection, which is likely the result of the translation from the REST URI's.

Publishing Messages

Publishing a message is similarly straightforward:

```

List<String> publishMessage(String topicName, String data) throws IOException {
    List<PubsubMessage> messages = Lists.newArrayList();
    messages.add(new PubsubMessage().encodeData(data.getBytes("UTF-8")));
    PublishRequest publishRequest = new PublishRequest().setMessages(messages);
    PublishResponse publishResponse = pubsub.projects().topics()
        .publish(getTopic(topicName), publishRequest)
        .execute();
    return publishResponse.getMessageIds();
}

```

You can publish more than one message at a time and you will receive a list of published message ID's in return. Messages carry base64 encoded data, which makes publishing them from the [API Explorer](#) a little less convenient. You can also add name-value attributes to the message, e.g. to carry metadata.

Creating Subscriptions

Creating subscriptions is also easy if you keep in mind the naming convention and the object hierarchy:

```

Subscription subscribeTopic(String subscriptionName, String topicName) throws IOException {
    String sub = getSubscription(subscriptionName);
    Subscription subscription = new Subscription()
        .setName(sub)
        .setAckDeadlineSeconds(15)
        .setTopic(getTopic(topicName));
    try {
        return pubsub.projects().subscriptions().create(sub, subscription).execute();
    } catch (GoogleJsonResponseException e) {
        if (e.getStatusCode() == HttpURLConnection.HTTP_CONFLICT) {
            return subscription;
        } else {
            throw e;
        }
    }
}

```

```
}

```

Here I opted to swallow the exception associated with the 409 "Conflict" return code if the subscription already exists. As explained above, it's important to distinguish the naming (and thus the identity) of topics and subscriptions to make sure you achieve the desired channel semantics -- my first attempt did not.

Consuming Messages

Last, but not least, consuming, or "pulling", messages mostly involves navigating collections and decoding data while avoid NPE's. Messages have to be explicitly acknowledged -- more on that below. I am passing a flag to my method to auto-acknowledge messages right after they were received. If you desire (local) transactional semantics, you postpone acknowledgment until after successfully processing the message.

```
List<String> pullMessage(Subscription subscription, int maxMessages, boolean doAck) throws IOException {
    PullRequest pullRequest = new PullRequest()
        .setReturnImmediately(true)
        .setMaxMessages(maxMessages);
    PullResponse response = pubsub.projects().subscriptions().pull(subscription.getName(), pullRequest).execute();
    List<ReceivedMessage> messages = response.getReceivedMessages();
    List<String> ackIds = Lists.newArrayList();
    List<String> data = Lists.newArrayList();
    if (messages != null) {
        for (ReceivedMessage receivedMessage : messages) {
            PubsubMessage message = receivedMessage.getMessage();
            if (message != null) {
                byte[] bytes = message.decodeData();
                if (bytes != null) {
                    data.add(new String(bytes, "UTF-8"));
                }
            }
            ackIds.add(receivedMessage.getAckId());
        }
    }
    if (doAck) {
        AcknowledgeRequest ackRequest = new AcknowledgeRequest().setAckIds(ackIds);
        pubsub.projects().subscriptions().acknowledge(subscription.getName(), ackRequest).execute();
    }
    return data;
}
```

You can pull one or multiple messages per pull request and also chose whether to block until a message arrives or whether to return immediately by means of the `setReturnImmediately` property.

A Tiny Demo App

Packaging all of this along with some fields and a constructor into a wrapper class allows you to send messages like this:

```
static final String PROJECT = "eaipubsub";
static final String TOPIC = "sometopic";


public static void main(String[] args) throws IOException, GeneralSecurityException {
    PubSubWrapper pubsub = PubSubWrapper.getInstance(
        System.getProperty("PRIVATE_KEY_FILE_PATH"),
        System.getProperty("SERVICE_ACCOUNT_EMAIL"),
        PROJECT);

    String data = (args.length > 0) ? args[0] : "hello pub/sub api";
    pubsub.createTopic(TOPIC);
    List<String> messageIds = pubsub.publishMessage(TOPIC, data);
    if (messageIds != null) {
        for (String messageId : messageIds) {
            System.out.println("Published with a message id: " + messageId);
        }
    }
}
```

Integration Patterns

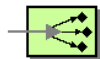
Let's look at how different [Integration Patterns](#) find themselves in the Cloud Pub/Sub implementation. This will also be a nice test of time for the patterns to see whether the vocabulary established through the patterns is still relevant with cloud API's, something we did not even dream off when we wrote the book in 2003!

Publish-Subscribe Channel



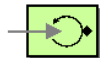
Cloud Pub/Sub has classic [Publish-Subscribe Channel](#) semantics. Not part of the pattern, but of the related discussion is the notion of a *topic hierarchy*, as for example described in the (somewhat dated) [OASIS WS-Topics](#) specification. A topic hierarchy organizes the topic as a hierarchy or tree and is often coupled with wildcard subscriptions that allow subscribing to "higher-up" nodes, which equates to subscribing to all topics underneath that node. Cloud Pub/Sub does not implement a topic hierarchy -- topics must be simple names. Topic names can contain a period to create topics such as `mytopic.subtopic`, albeit without any related semantics.

Competing Consumers



As illustrated above, Cloud Pub/Sub supports [Competing Consumers](#) by having multiple consumers pull messages from the same subscription. This feature allows for example for the distribution of workload across multiple consumers while guaranteeing that each message is consumer by only one consumer.

Polling Consumer, Event-driven Consumer



My simple code example implements a rather old-school [Polling Consumer](#) and therefore does not include a `pushConfig` in the subscription. If you are building cloud applications, you are much more likely to implement an [Event-Driven consumer](#), which can act as a WebHook receiver. Cloud Pub/Sub is happy to [support both](#).

Durable Subscriber



Cloud Pub/Sub subscribers are [Durable Subscribers](#): messages published to a topic with an active subscription are received even while the application is not running and are stored until the application pulls the message. This also leads to the "Poison Message" syndrome (indeed a missing pattern in our [book](#)): if a particular message causes the consuming application to crash, that same message will be pulled again on application restart, causing the application to crash again. This happened to me while building the simple Cloud Pub/Sub demo when I sent a non-base64-encoded message from the API Explorer that caused `decodeData` to return `null`. A [Channel Purger](#) may be helpful in such situations

Message Expiration



Cloud Pub/Sub does implement a general [Message Expiration](#) of 7 days as explained in the [Quota Policy](#). However, this property is not setttable on a per-message basis, so the messaging system cannot automatically discard messages whose value decreases rapidly with increasing age. There also is no [Dead Letter Channel](#). On the upside 7 days at 10,000 messages a second adds up to a nice chunk of storage.

Transactional Client



Cloud Pub/Sub supports [Transactional Clients](#) via an explicit message acknowledgment: messages consumers have to acknowledge message consumption within a configurable time interval of 15 -600 seconds. Messages that were not acknowledged are returned to the subscription to be pulled again.

I tested this behavior with a client that pulls, but never acknowledges messages. As expected, a message published to the associated topic is repeatedly redelivered after the specified acknowledgment deadline. After starting another subscriber that does acknowledge messages on the same subscription (thus operating in the [Competing Consumers](#) model), the "bad" subscriber kept on receiving the message for some time until eventually the "good" receiver got it and acknowledged it. Fairness across multiple competing subscribers to a single subscription is not explicitly defined in the spec, but it seems that in case of a wrecked consumer other consumers will eventually have a chance to pull the messages that were not acknowledged by the "bad" consumer. It's probably best to assume the distribution of messages across multiple consumers on a single subscription is random.

Idempotent Receiver



Cloud Pub/Sub usually delivers messages exactly once and in order. However, it does [not guarantee](#) this behavior. Therefore, the application layer has to ensure subscribers are [Idempotent Receivers](#). This is not an omission, but surely results in a significant simplification of the implementation. Applications that require idempotency at the application level can use additional mechanisms, such as a list of received message ID's, to implement this behavior.

Sadly we did not design an icon for [Idempotent Receiver](#) when we wrote the book. I just made this one up. Maybe it'll make it into the 2nd edition...

Google Cloud Pub/Sub and Integration Patterns

It's satisfying to see that we can discuss much of what Google Cloud Pub/Sub does using the pattern language described in [Enterprise Integration Patterns](#) 12 years ago. It's also nice to see that messaging semantics blend well with REST API's and cloud services, something that seemed very far away in 2003 -- even Web Services were described as an [Emerging Standard](#) back then!

In case of the Google Cloud Pub/Sub system, I am particularly happy to see it become available as I was lucky enough to observe some of the humble beginnings in my days at Google CorpEng

Join my Team!

Are you a top-notch cloud infrastructure or application architect that wants to help establish a private cloud infrastructure for one of the largest insurance companies in the world? And re-architect critical core applications to efficiently run on it? Then consider [joining my team](#) at Allianz in Munich! I currently have two listings on LinkedIn: [Cloud Infrastructure Architect](#) and [Principal Software Architect](#), but am also looking for other highly qualified individuals in the areas of IoT, Data Architecture, High Performance Computing, etc. Apply directly on [Allianz Careers](#)!



Share: Tweet [Share](#)

+16 Recommend this on Google

Follow: Follow @ghohpe { 1,682 followers

[SUBSCRIBE TO](#)

[FEED](#)

More On: [MESSAGING](#) [INTEGRATION](#) [CLOUD](#) [ALL RAMBLINGS](#)



Gregor is the Chief IT Architect of Allianz SE. He is a frequent speaker on asynchronous messaging and service-oriented architectures and co-authored *Enterprise Integration Patterns* (Addison-Wesley). His mission is to make integration and distributed system development easier by harvesting common patterns and best practices from many different technologies.
www.eaipatterns.com