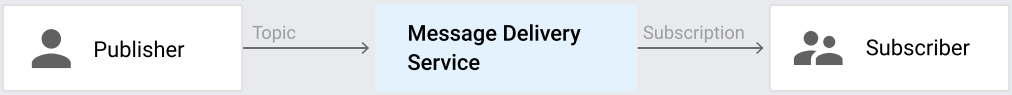
Pub/Sub provides a highly-available, scalable message delivery service. The tradeoff for having these properties is that the order in which messages are received by subscribers is not guaranteed. While the lack of ordering may sound burdensome, there are very few use cases that actually require strict ordering.

This document includes an overview of what message order really means and its tradeoffs, as well as a discussion of use cases and techniques for dealing with order-dependent messaging in your current workflow when moving to Pub/Sub.

On the surface, the idea of messages being in order is a simple one. The following image shows a bird's eye view of what it looks like for a message to flow through a message delivery service:

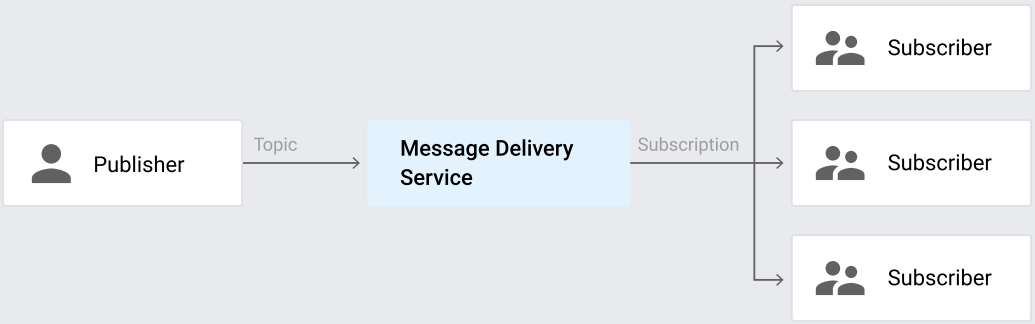


Your publisher sends a message on a topic into Pub/Sub. The message is then delivered to your subscriber via a subscription. In the presence of a single synchronous publisher, a single synchronous subscriber, and a single synchronous message delivery server—all running on a synchronous transport layer—the notion of order seems simple: messages are ordered by when the publisher successfully publishes them, by some measure of absolute time or sequence.

Even in this simple case, guaranteed ordering of messages would put severe constraints on throughput. The only way to truly guarantee order of messages would be for the message delivery service to deliver the messages one at a time to the subscriber, waiting to deliver the next message until the service knows the subscriber has received and processed the current message (generally via an acknowledgement sent from the subscriber to the service). Throughput of one message to the subscriber at a time is not scalable. The service could instead only guarantee that the first delivery of any message is in-order, allowing redelivery attempts to happen at any time. That would allow many messages to be sent to the subscriber

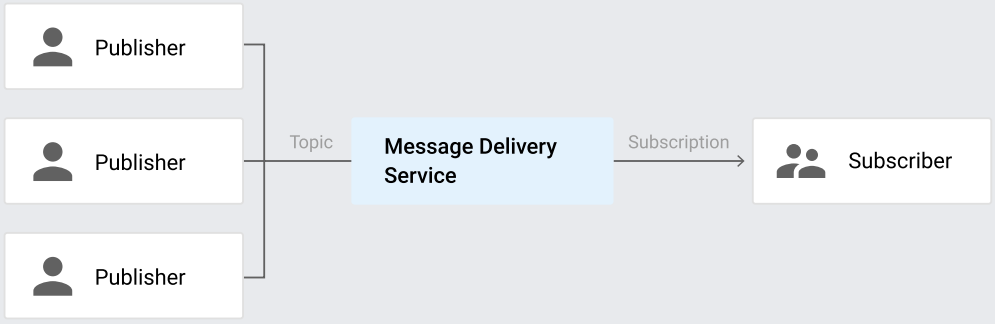
at once. However, even if ordering constraints are relaxed in this way, “order” makes less sense as you move away from the single publisher/message delivery service/subscriber case.

Defining message order can be complicated, depending on your publishers and subscribers. First of all, it is possible you have multiple subscribers processing messages on a single subscription:



In this case, even if messages come through the subscription in order, there are no guarantees of the order in which the messages will be processed by your subscribers. If order of processing is important, then subscribers would need to coordinate through some ACID storage system such as Cloud Firestore (/firestore/) or Cloud SQL (/sql/).

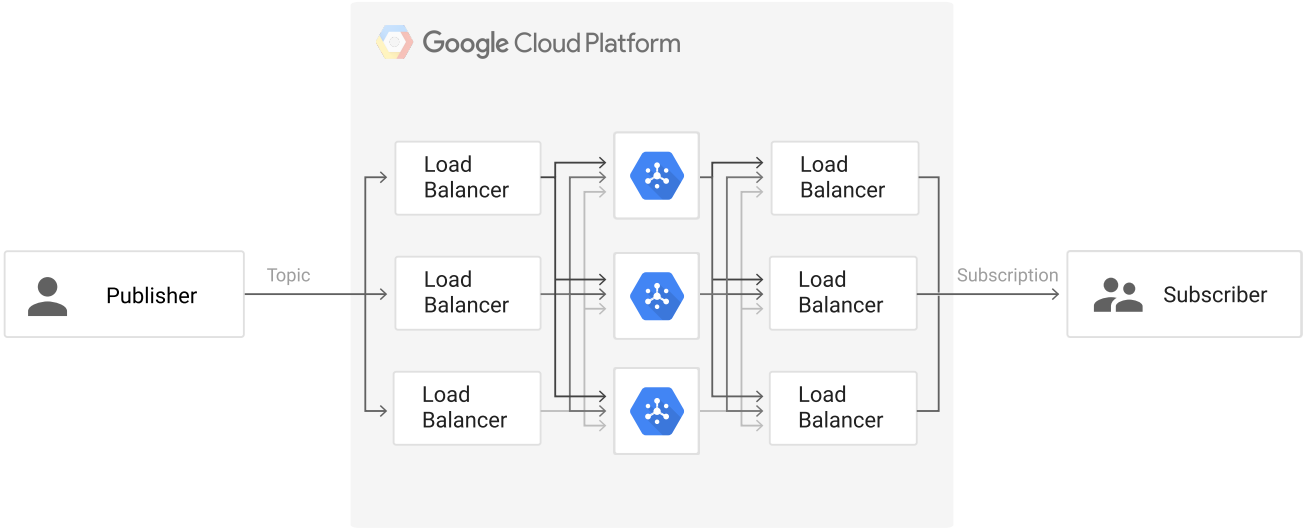
Similarly, multiple publishers on the same topic can make order difficult:



How do you assign an order to messages published from different publishers? Either the publishers themselves have to coordinate, or the message delivery service itself has to attach a

notion of order to every incoming message. Each message would need to include the ordering information. The order information could be a timestamp (though it has to be a timestamp that all servers get from the same source in order to avoid issues of clock drift), or a sequence number (acquired from a single source with ACID guarantees). Other messaging systems that guarantee ordering of messages require settings that effectively limit the system to multiple publishers sending messages through a single server to a single subscriber.

If the abstract message delivery service used in the examples above were a single, synchronous server, then the service itself could guarantee order. However, a message delivery service such as Pub/Sub is not a single server, neither in terms of server roles nor in terms of number of servers. In fact, there are layers between your publishers and subscribers and the Pub/Sub system itself. Here is a more detailed diagram of what is going on when Pub/Sub is the message delivery system:



As you can see, there are many paths a single message can take to get from the publisher to the subscriber. The benefit of such an architecture is that it is highly available (no single server outage results in systemwide delays) and scalable (messages can be distributed across many servers to maximize throughput). The benefits of distributed systems like this have been instrumental to Google products such as Search, Ads, and Gmail that build on top of the same systems that run Pub/Sub.

Hopefully, you now understand why order of messages is fairly complex and why Pub/Sub de-emphasizes the need for order. To attain availability and scalability, it is important that you minimize your reliance on order. The reliance on order can take several forms, each described below with some typical use cases and solutions.

Typical Use Cases: Queue of independent tasks, collection of statistics on events

Use cases where order does not matter at all are perfect for Pub/Sub. For example, if you have independent tasks that need to be performed by your subscribers, each task is a message where the subscriber that receives the message performs the action. As another example, if you want to collect statistics on all actions taken by clients on your server, you can publish a message for each event and then have your subscribers collate messages and update results in persistent storage.

Typical Use Cases: Logs, state updates

In use cases in this category, the order in which messages are processed does not matter; all that matters is that the end result is ordered properly. For example, consider a collated log that is processed and stored to disk. The log events come from multiple publishers. In this case, the actual order in which log events are processed does not matter; all that matters is that the end result can be accessed in a time-sorted manner. Therefore, you could attach a timestamp to every event in the publisher and make the subscriber store the messages in some underlying data store (such as Cloud Firestore (/firestore/)) that allows storage or retrieval by the sorted timestamp.

The same option works for state updates that require access to only the most recent state. For example, consider keeping track of current prices for different stocks where one does not care about history, only the most recent value. You could attach a timestamp to each stock tick and only store ones that are more recent than the currently-stored value.

Typical Use Cases: Transactional data where thresholds must be enforced

Complete dependence on the order in which messages are processed is the most complicated case. Any solution that enforces strict ordering of messages is going to come at the expense of performance and throughput. You should only depend on order when it is absolutely necessary, and when you are sure that you won't need to scale to a large number of messages per second. In order to process messages in order, a subscriber must either:

- Know the entire list of outstanding messages and the order in which they must be processed, or
- Have a way to determine from all messages it has currently received whether or not there are messages it has not yet received that it needs to process first.

You could implement the first option by assigning each message a unique identifier and storing in some persistent place (such as Cloud Firestore (/firestore/)) the order in which messages should be processed. A subscriber would check the persistent storage to know the next message it must process and ensure that it only processes that message next, waiting to process other messages it has received when they come up in the complete ordering. At that point, it is worth considering using the persistent storage itself as the message queue and not relying on message delivering through Pub/Sub.

The latter is possible by using Cloud Monitoring (/monitoring/) to keep track of the **`pubsub.googleapis.com/subscription/oldest_unacked_message_age`** metric (see Supported Metrics (/monitoring/api/metrics) for a description). A subscriber would temporarily put all messages in some persistent storage and ack the messages. It would periodically check the oldest unacked message age and check against the publish timestamps of the messages in storage. All messages published before the oldest unacked message are guaranteed to have been received, so those messages can be removed from persistent storage and processed in order.

Alternatively, if you have a single, synchronous publisher and a single subscriber, you could use a sequence number to ensure ordering. This approach requires the use of a persistent counter. For each message, the publisher performs the following:

The subscriber performs the following:

Either of these solutions introduces latency in publishing and processing messages; there needs to be a synchronous step in the publisher to create the order and a delay on out-of-order messages in the subscriber to enforce order.

When first using a message delivery service, having messages delivered in order seems like a desirable property. It simplifies the code necessary to process messages where order matters. However, providing messages in order comes at great cost to availability and scalability, no matter what message delivery system you use. For Google products built on the same infrastructure in Pub/Sub, availability and scalability are vitally important features, which is why the service does not offer in-order message delivery. Whenever possible, design your applications to avoid a dependency on message order. You will then be able to scale easily, with Pub/Sub scaling to deliver all of your messages quickly and reliably.

If you decide to implement some form of message ordering with Pub/Sub, see Cloud Firestore (/firestore/) and Cloud SQL (/sql/) to learn more about how to implement the strategies described in this document.