Messaging System

Overview

A message system enables **asynchronous** communication.

In modern cloud architecture, applications are **decoupled** into smaller, independent building blocks that are easier to develop, deploy and maintain. Message queues provide communication and coordination for these **distributed** applications.

Message queues can significantly **simplify coding** of decoupled applications, while improving **performance, reliability and scalability**.

A messaging system maintains a **queue** in its disks or memory, allowing it to store the messages **producers** add to the system while deleting messages that **consumers** have consumed/executed.

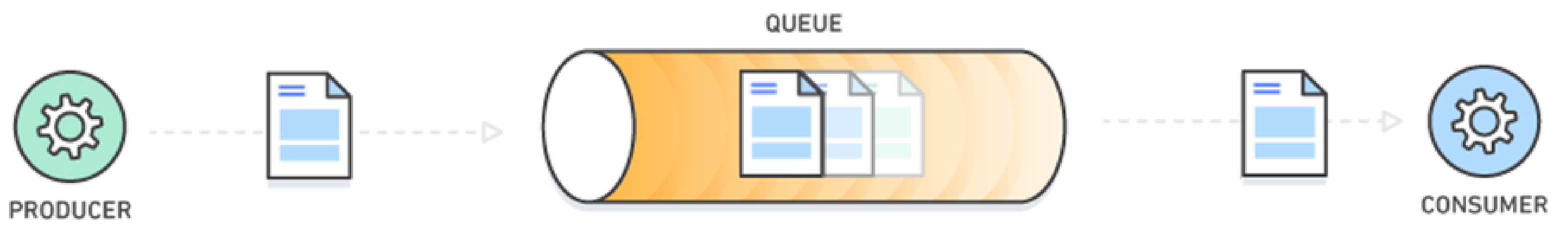
The **messages** are usually small, and can be things like **requests, replies, error messages,** or just **plain information**.

Messaging Patterns

Messaging systems may follow one of two **patterns**: message queuing(point-to-point) or a pub-sub pattern.

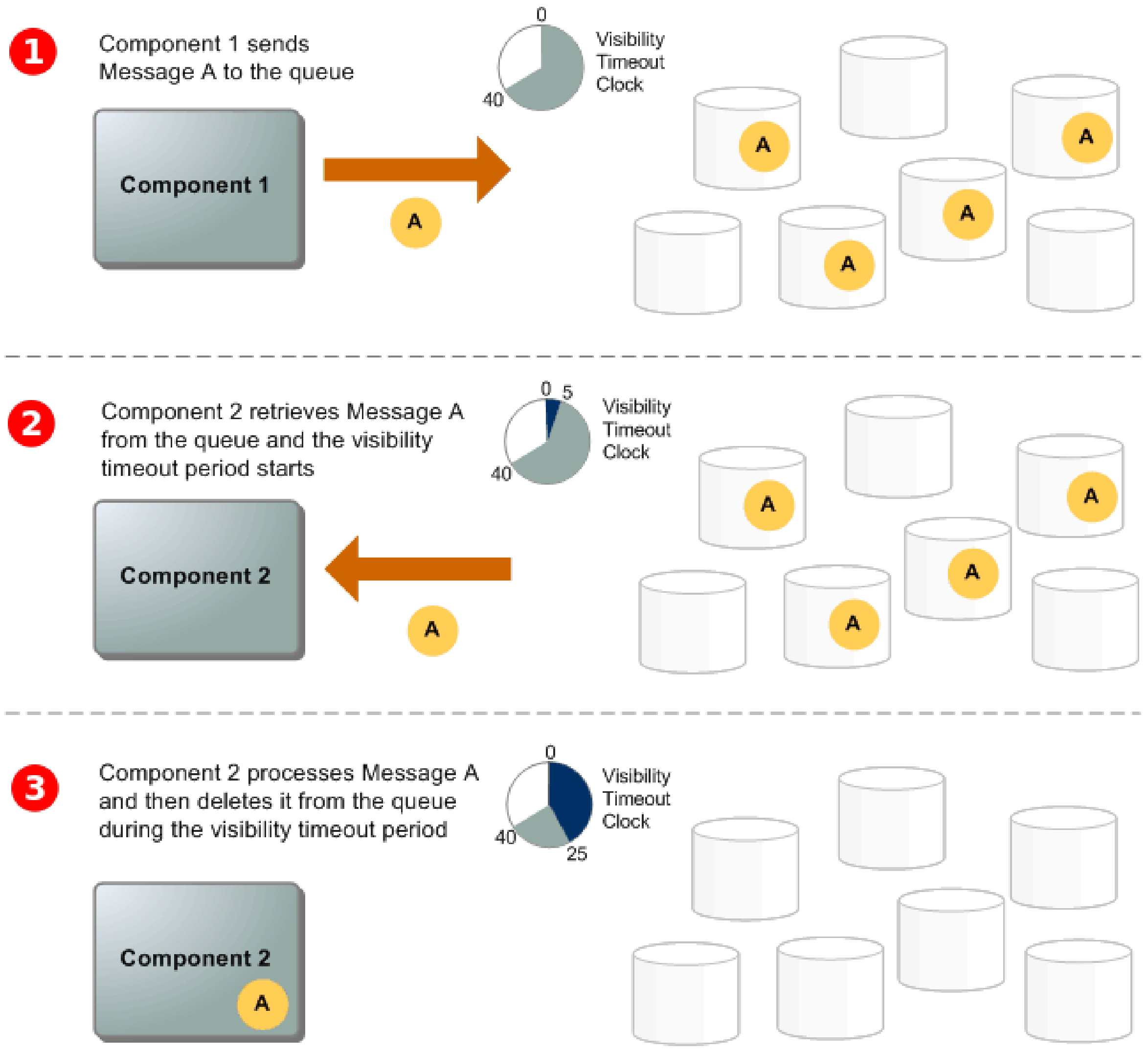
Message Queue (point-to-point)

A message queue is a form of asynchronous **point-to-point** communication. Many producers and consumers can use the queue, but each message is **processed only once**, by a **single consumer**.



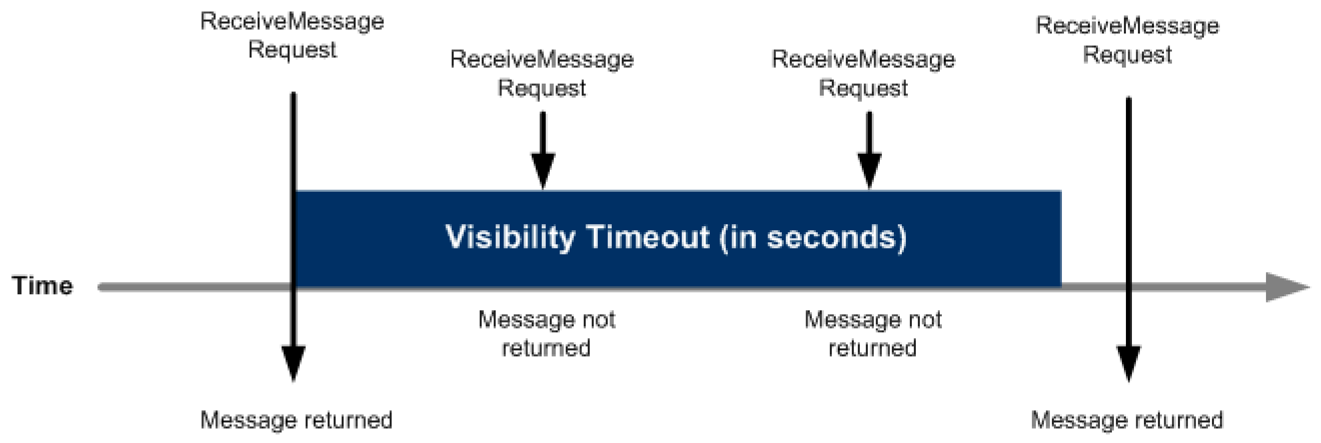
Case Study: Amazon SQS

Achieving reliability through **data redundancy** and **visibility timeout**.


1
Component 1 sends Message A to the queue
Component 1
A
0
40
Visibility Timeout Clock
2
Component 2 retrieves Message A from the queue and the visibility timeout period starts
Component 2
0
5
40
3
Component 2 processes Message A and then deletes it from the queue during the visibility timeout period
25

1 A producer (component 1) **sends** message A to a queue, and the message is distributed across the Amazon SQS servers **redundantly**.

2 When a consumer (component 2) is ready to process messages, it **consumes** messages from the queue, and message A is returned. While message A is being processed, it **remains** in the queue and **isn't returned to subsequent receive requests** for the duration of the visibility timeout.



Amazon SQS doesn't automatically delete the message. Because Amazon SQS is a distributed system, there's no guarantee that the consumer actually receives the message (for example, due to a connectivity issue, or due to an issue in the consumer application).

3 The consumer (component 2) **deletes** message A from the queue to prevent the message from being received and processed again when the visibility timeout expires.

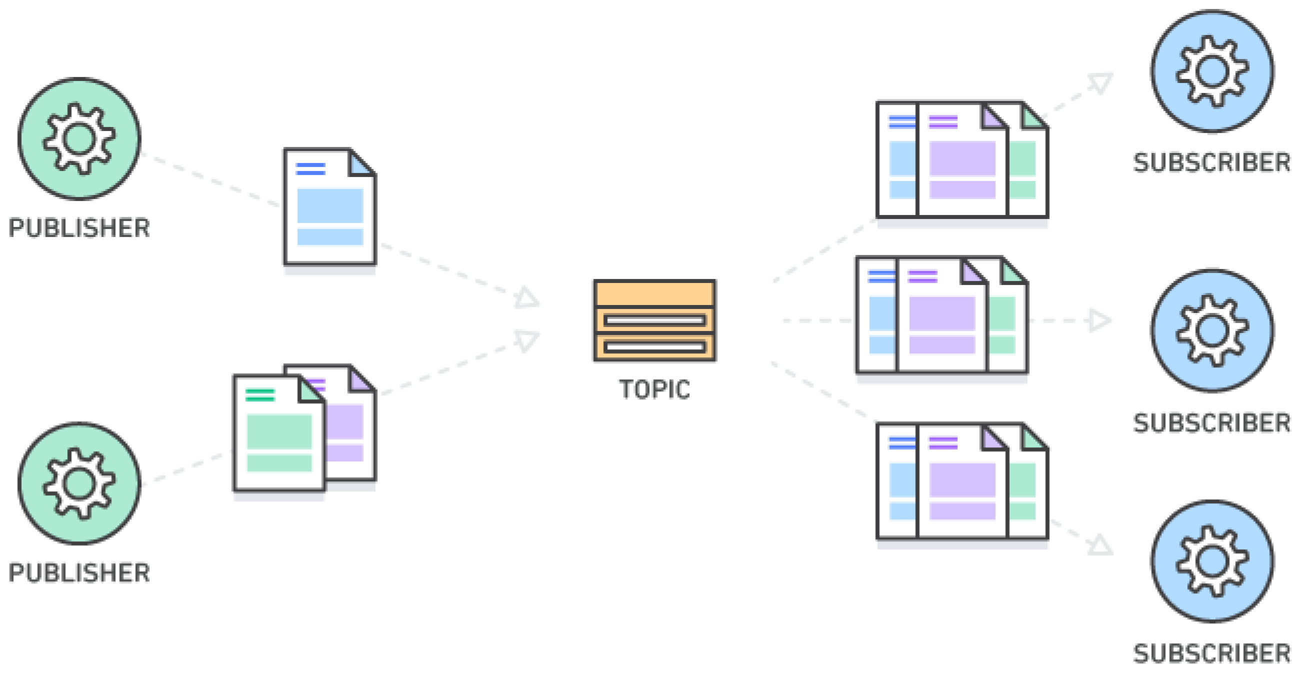
Pub/Sub Messaging

A pub/sub model allows messages to be broadcasted asynchronously across multiple sections of the applications. The core component that facilitates this functionality is something called a **Topic**. The **publisher** will **push** messages to a Topic, and the Topic will instantly **push** the message to all the **subscribers**.

Pub/sub messaging can be used to enable **event-driven** architectures.

An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with **microservices**. An **event** is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website.

Event-driven architectures have three key components: event **producers**, event **routers**, and event **consumers**. A producer publishes an event to the router, which filters and pushes the events to consumers. Producer services and consumer services are **decoupled**.



The **subscribers** to the message topic often perform **different functions**, and can each do something different with the message in parallel.

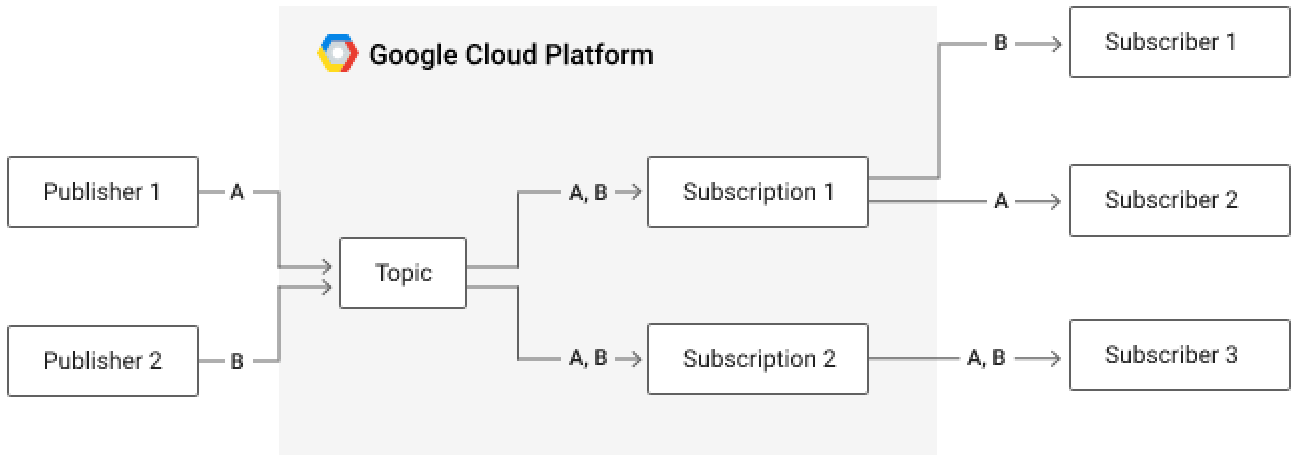
In the pubsub model, the messaging system doesn't need to know about any of the subscribers. It doesn't track which messages have been received and it doesn't manage the load on the consuming process. Instead, the **subscribers** track which messages have been received and are responsible for **self-managing** load levels and scaling.

Case Study: Google Pub/Sub

There are several key concepts in a Pub/Sub service:

- **Message**: the **data** that moves through the service.
- **Topic**: a named entity that represents a **feed of messages**.
- **Subscription**: a named entity that represents an **interest** in receiving messages on a particular topic.
- **Publisher** (also called a producer): creates messages and sends (publishes) them to the messaging service on a specified topic.
- **Subscriber** (also called a consumer): receives messages on a specified subscription.

The following diagram shows the basic flow of messages through Pub/Sub:



In this scenario, there are two **publishers** publishing messages on a single **topic**. There are two **subscriptions** to the topic.

The first subscription has two subscribers, meaning messages will be **load-balanced** across them, with each subscriber receiving a **subset** of the messages. The second subscription has one subscriber that will receive **all** of the messages.

The bold letters represent messages. Message A comes from Publisher 1 and is sent to Subscriber 2 via Subscription 1, and to Subscriber 3 via Subscription 2. Message B comes from Publisher 2 and is sent to Subscriber 1 via Subscription 1 and to Subscriber 3 via Subscription 2.

More differences between point-to-point and pub/sub systems : https://cloud.google.com/solutions/event-driven-architecture-pubsub

Judging Performance

A messaging service can be judged on its performance in three aspects: **scalability, availability, and latency**. These three factors are often at odds with each other, requiring **compromises** on one to improve the other two.

A **scalable** service should be able to handle increases in load.

A system's **availability** is measured on how well it deals with different types of issues, gracefully failing over in a way that is unnoticeable to end users.

Latency is a time-based measure of the **performance** of a system. A service generally wants to minimize latency wherever possible.

References*

https://aws.amazon.com/message-queue/
https://aws.amazon.com/pub-sub-messaging/
https://en.wikipedia.org/wiki/Message_queue
https://blog.iron.io/message-queue-vs-publish-subscribe/
https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-basic-architecture.html

https://www.bmc.com/blogs/pub-sub-publish-subscribe/
https://aws.amazon.com/event-driven-architecture/