AWS SQS and SNS

CS516 – Cloud Computing

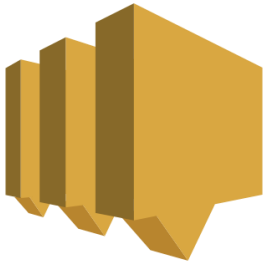
Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- SQS
  - Standard and FIFO queues
  - Group id and Deduplication id
  - Visibility timeout
- SNS
  - Message Attributes
  - Filtering
  - Dead letter queue
- Circuit breaker design pattern
- Microservices (SOA) architecture
- Event-driven architecture



SNS and SQS roles

1. For decoupling applications (microservices)

    1.1 It improves performance by making operations async.

    1.2 It improves fault-tolerance and reliability for temporary storing data.

2. For building event-driven applications. Event-driven architecture is trending.

Simple Queue Service - SQS

SQS offers a secure, durable, and available hosted queue that lets you **integrate** and **decouple** distributed software systems and components (microservices).

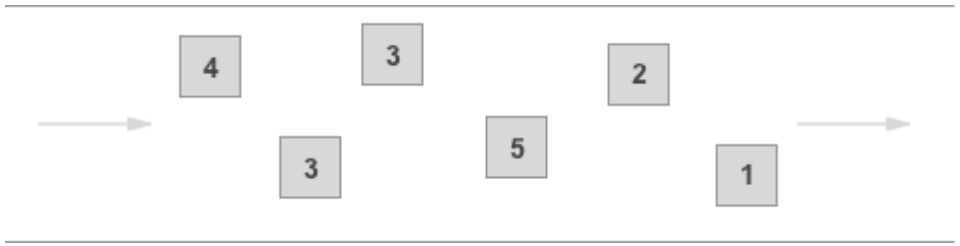
SQS and SNS provides nearly **unlimited scalability**. There are 2 types of queues, standard and FIFO.

Check out this video “AWS SQS Overview For Beginners” by the best channel on serverless technologies.

Standard queue

Use it when the throughput is important. Standard queues provide:

- **Unlimited throughput** - Standard queues support a nearly unlimited number of API calls per second. API calls include SendMessage, ReceiveMessage, DeleteMessage.
- **At-least-once delivery** - A message is delivered at least once, but occasionally more than one copy of a message is delivered.
- **Best-effort ordering** - Occasionally, messages are delivered in an order different from which they were sent.



FIFO queue

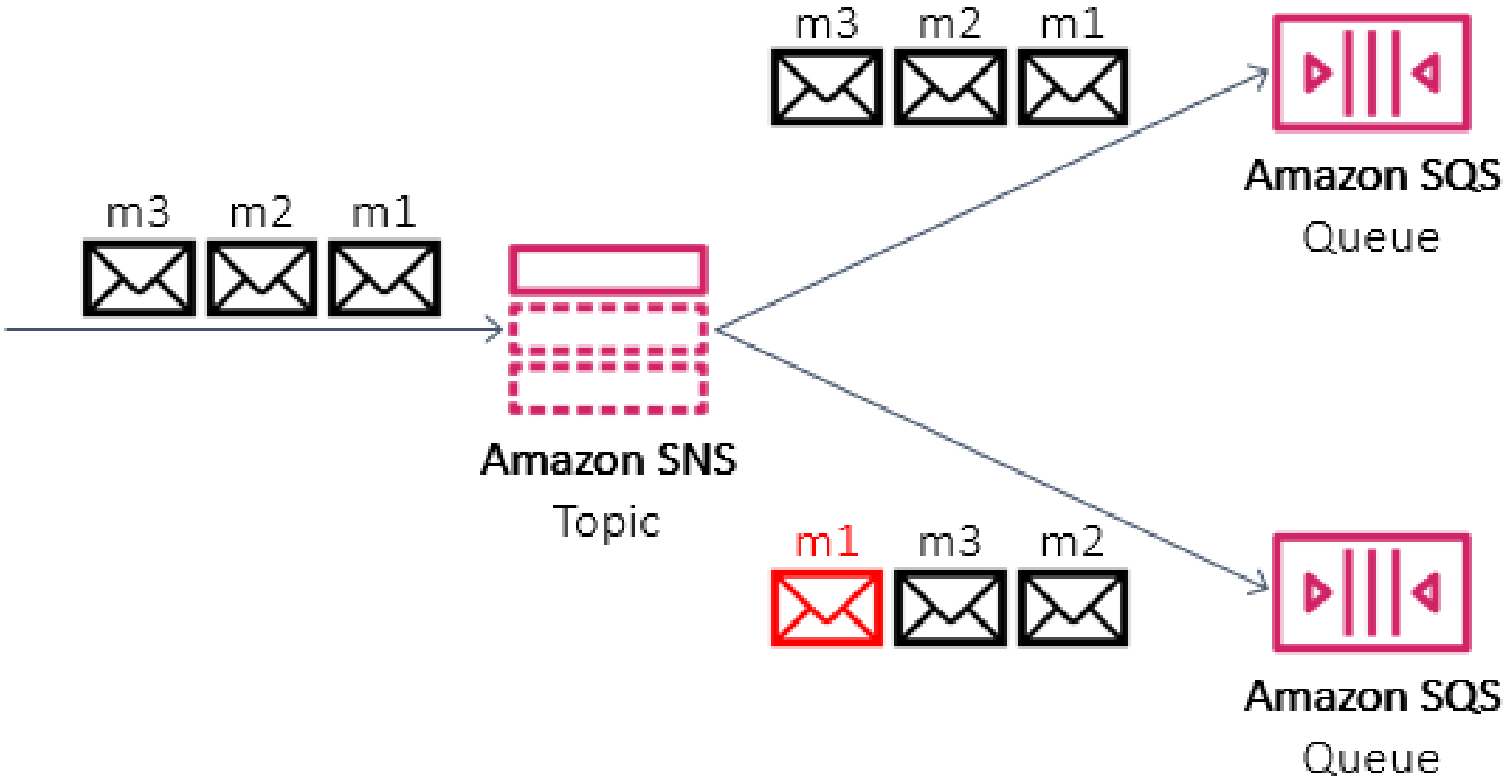
Use it when the order of events is important. FIFO queues guarantee consistency and strict message ordering.

- **High** (Not Unlimited) **Throughput** - If you use **batching**, FIFO queues support up to 3,000 messages per second (300 API calls, 10 messages per call).
- **Exactly-Once processing** - A message is delivered once and remains available until a consumer processes and deletes it. Duplicates aren't introduced into the queue.
- **First-In-First-Out delivery** - The order in which messages are sent and received is strictly preserved.

Message ordering and deduplication concepts also apply to SNS.

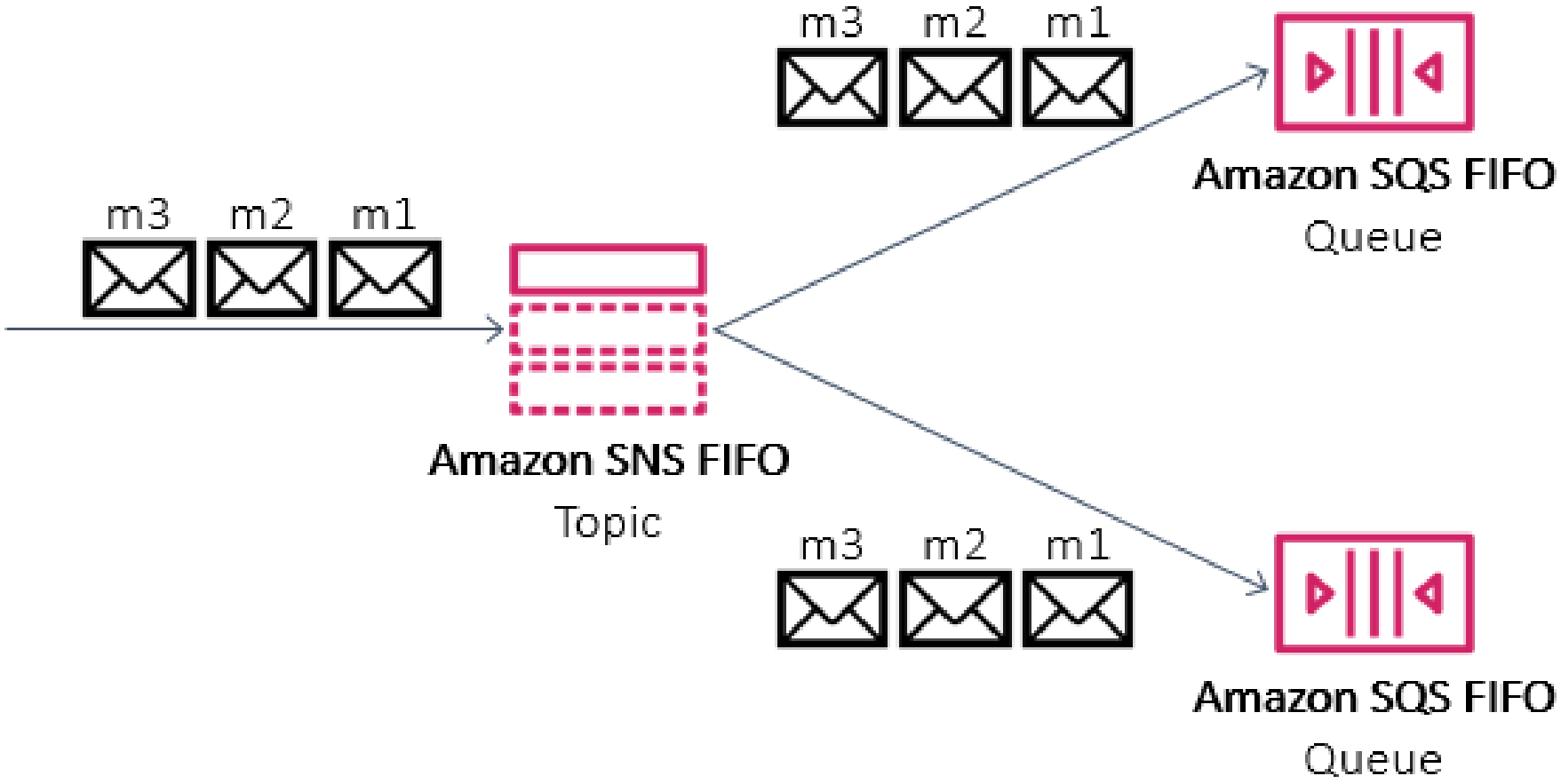
Best-effort message ordering

Standard SNS and SQS provide best-effort message ordering with rarely out of order delivery.



Strict message ordering

SNS FIFO and SQS FIFO provide strict message ordering.



Message deduplication ID

The message deduplication ID is the token used for deduplication of sent messages. If a message with a particular message deduplication ID is sent successfully, any messages sent with the same message deduplication ID are accepted successfully but aren't delivered during the 5-minute **deduplication interval**.

If content-based deduplication for the queue is enabled. The producer can omit the message deduplication ID.

Provide deduplication id if:

- Different contents but must be treated as duplicates.
- Identical content but different attributes must be treated as unique.

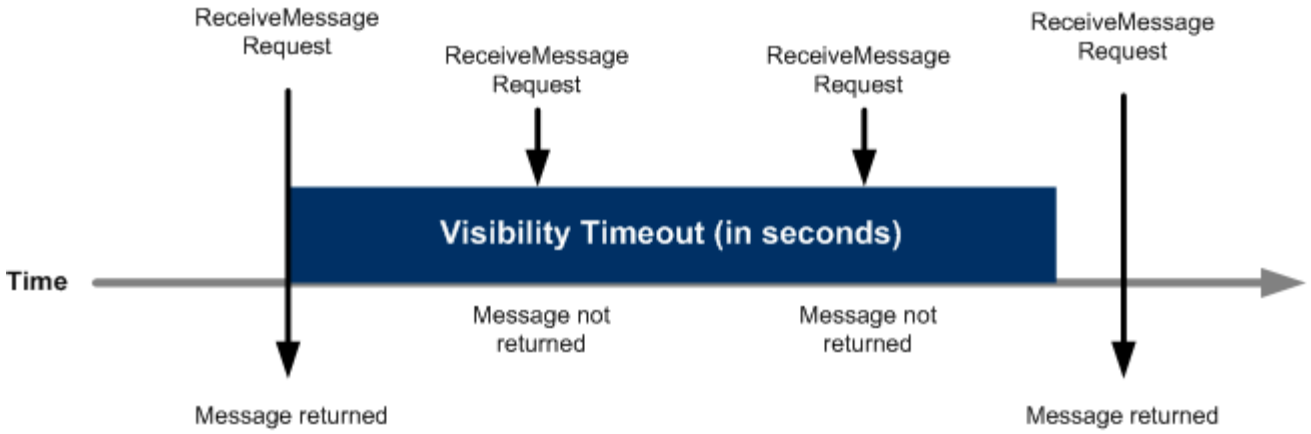
Message group ID

The message group ID is the tag that specifies that a message belongs to a specific message group. Messages that belong to the same message group are always processed one by one.

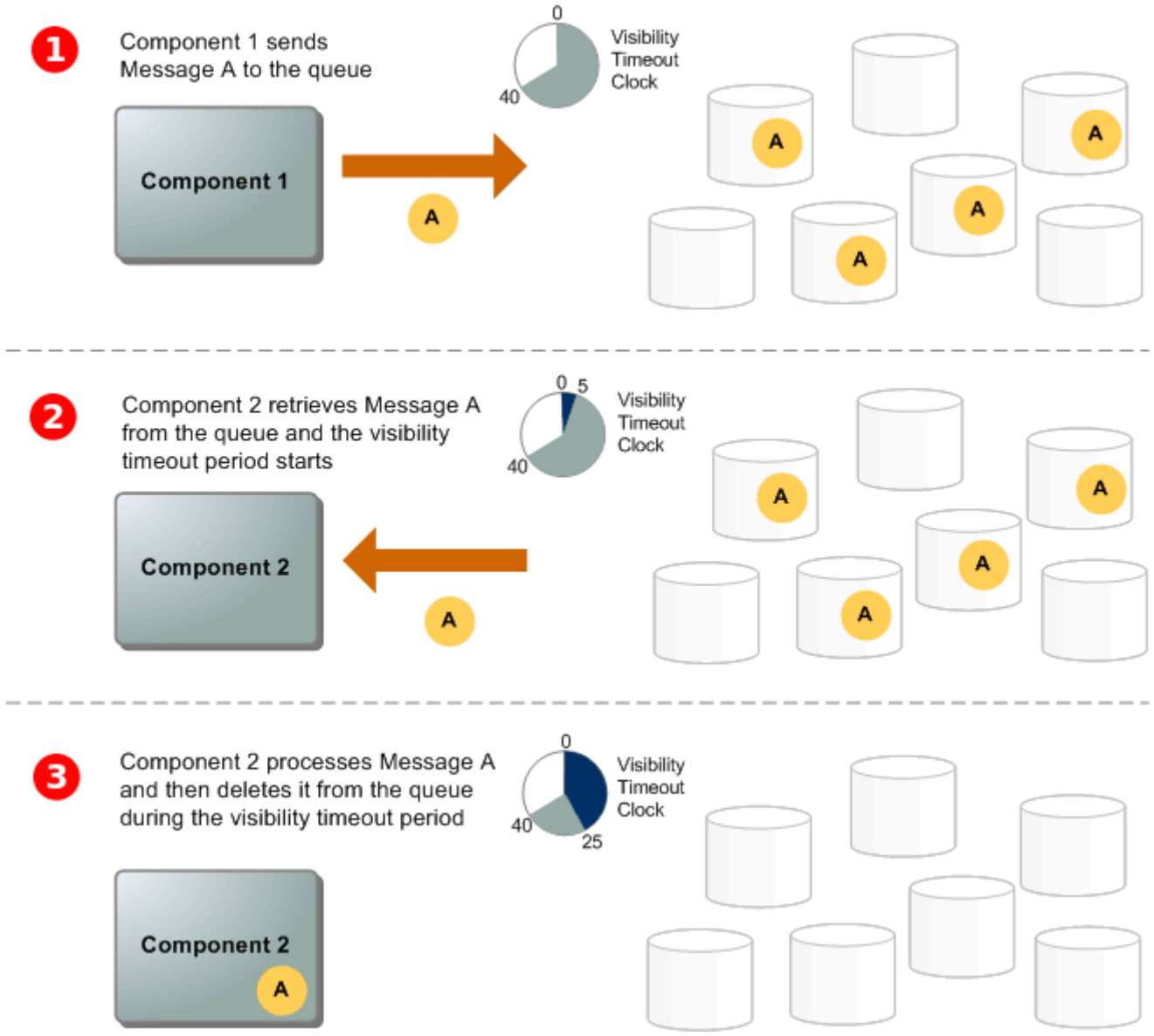
- Max number of messages in the queue is 20,000. Use group ID to avoid it.
- There could be bottleneck on the group of messages if the current message is not being processed properly by a consumer. Use group ID to prevent it.

SQS visibility timeout

When a consumer receives and processes a message from a queue, the message **remains** in the queue. SQS **doesn't automatically delete** the message. Because SQS is a **distributed system**, there's no guarantee that the consumer actually receives the message due to a connectivity issue, or an issue in the consumer application. Thus, the consumer **must delete** the message from the queue after receiving and processing it.



Message lifecycle



SQS short and long polling

Amazon SQS provides short polling and long polling to receive messages from a queue. By default, queues use short polling.

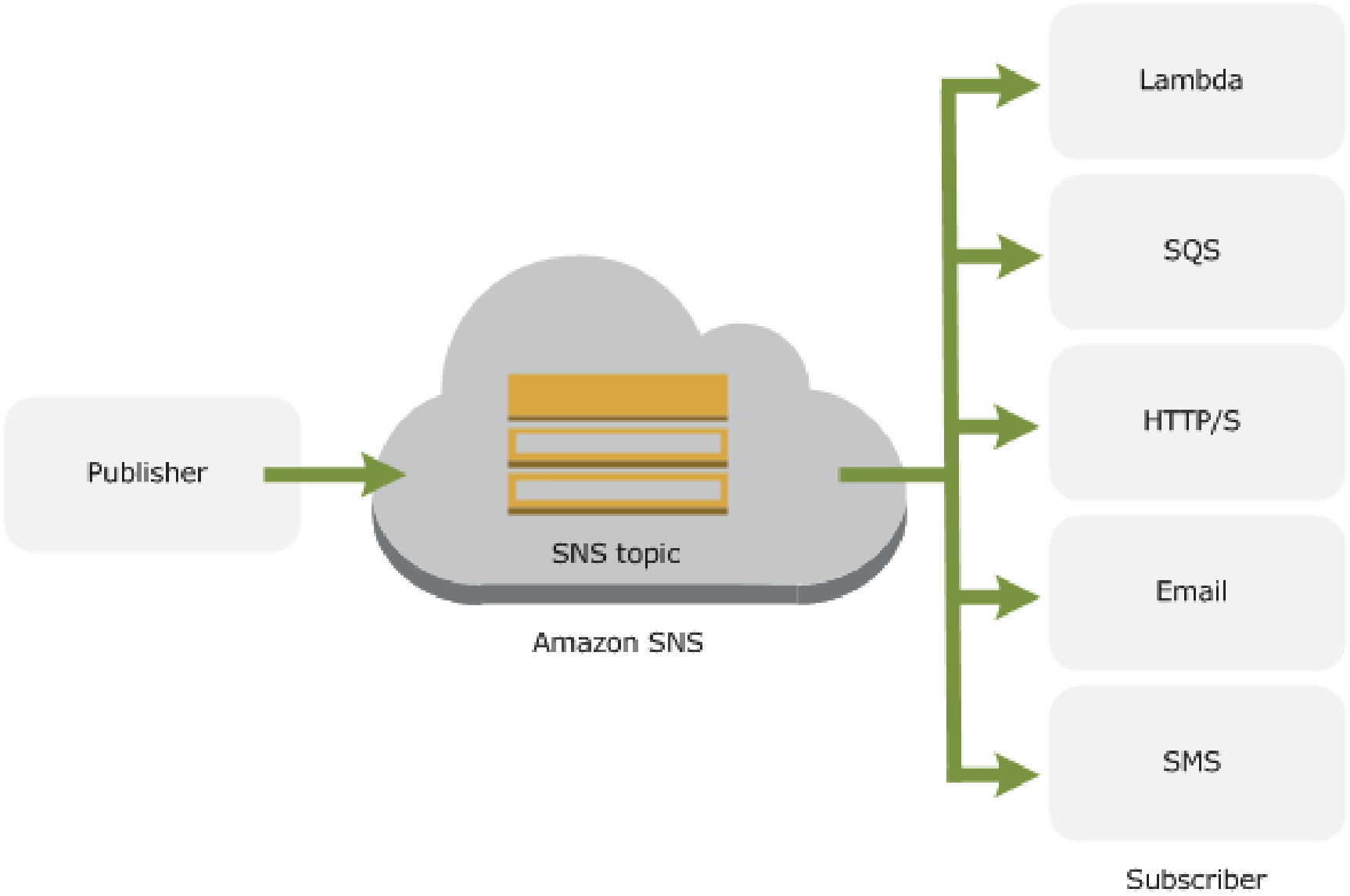
- With **short polling**, the ReceiveMessage request queries only a **subset of the servers** to find messages that are available to include in the response. Amazon SQS sends the response right away, even if the query found **no messages**.
- With **long polling**, the server keeps the client connection open and the ReceiveMessage request queries **all of the servers** for messages. Amazon SQS sends a response after it collects **at least one available message**, up to the maximum number of messages specified in the request. Amazon SQS sends an empty response only if the polling wait time expires.

Simple Notification Service - SNS

Simple Notification Service (SNS) is a fast, flexible, fully-managed **push** notification service that sends messages to subscribers from the message publisher.

You can use SNS to send emails from your application to the users and send emails to yourself when events occur in your AWS account in conjunction with CloudWatch and so on.

Read more about Amazon SNS


Lambda
SQS
HTTP/S
Email
SMS
Subscriber
Publisher
SNS topic
Amazon SNS

Application-to-application (A2A)

Amazon SNS lets you decouple publishers from subscribers. This is useful for application-to-application messaging for microservices, distributed systems, and serverless applications.

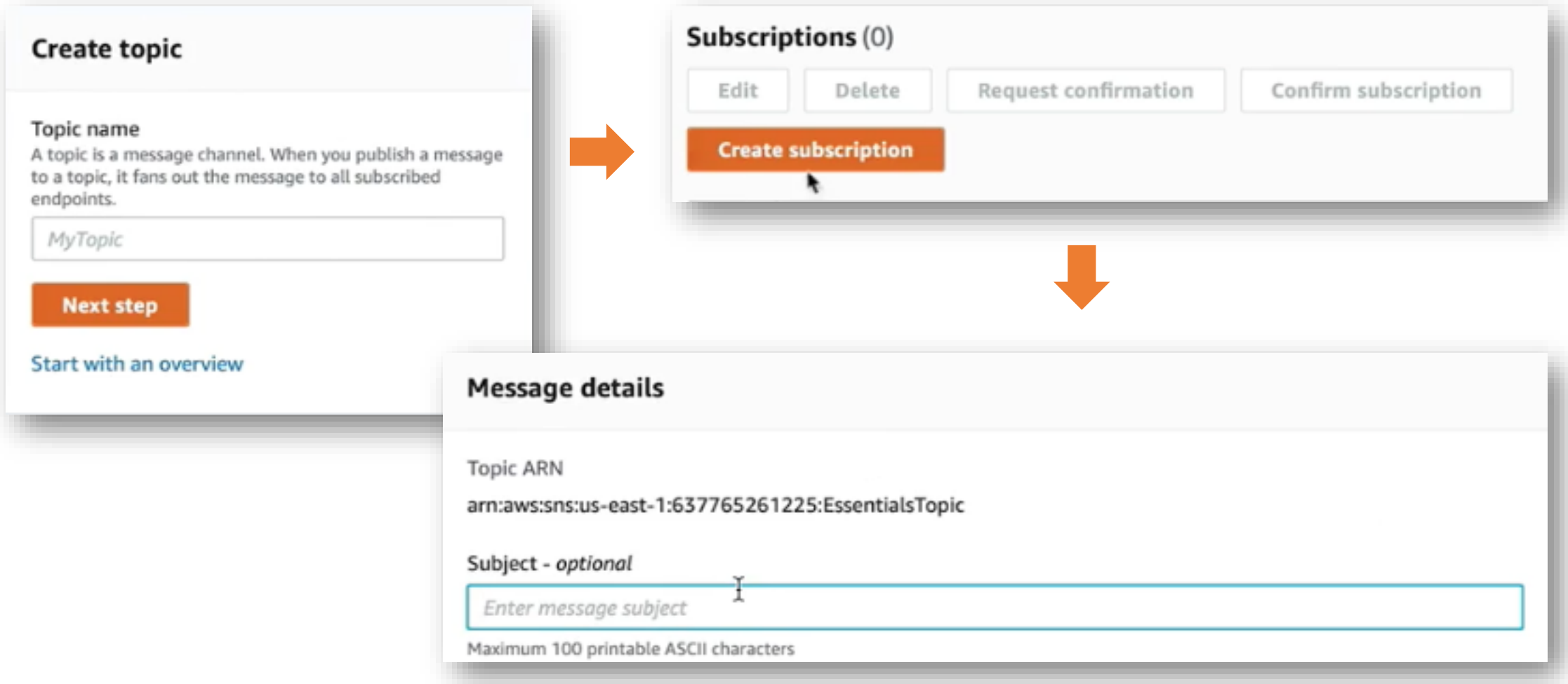
Application-to-person (A2P)

Amazon SNS lets you send push notifications to mobile apps, text messages to mobile phone numbers, and plain-text emails to email addresses. You can fan out messages with a topic, or publish to mobile endpoints directly.

SNS Basic Components

1. **Publishers** communicate **asynchronously** with subscribers by producing and sending a message to a topic.
2. **Topic** is a logical access point and communication channel.
3. **Subscribers** consume or receive the message or notification over one of the supported protocols when they are subscribed to the topic.

Create a Topic, Subscription, Publish



Message attributes

Message attributes let you provide any **arbitrary metadata** (such as timestamps, geospatial data, signatures, and identifiers) about the message.

You can use message attributes to make your messages filterable using subscription filter policies. You can apply filter policies to topic subscriptions. When a filter policy is applied, a subscription receives only those messages that have attributes that the policy accepts.

Sample SNS payload


```
{
   "Type": "Notification",
   "MessageId": "a1b2c34d-567e-8f90-g1h2-i345j67klmn8",
   "TopicArn": "arn:aws:sns:us-east-2:123456789012:MyTopic",
   "Message": "message-body-with-transaction-details",
   "Timestamp": "2019-11-03T23:28:01.631Z",
   "SignatureVersion": "4",
   "Signature": "signature",
   "UnsubscribeURL": "unsubscribe-url",
   "MessageAttributes": {
      "customer_interests": {
         "Type": "String.Array",
         "Value": "[\"soccer\", \"rugby\", \"hockey\"]"
      },
      "store": {
         "Type": "String",
         "Value":"example_corp"
      },
      "event": {
         "Type": "String",
         "Value": "order_placed"
      },
      "price_usd": {
         "Type": "Number",
         "Value":210.75
      }
   }
}
```

Raw message delivery

Message attributes are NOT sent if you send the message to HTTPS endpoint, Kinesis, and SQS. Because these 3 services expect a raw message or the message as is.

- Raw message delivery is **disabled**


```
{
  "Type": "Notification",
  "MessageId": "dc1e94d9-56c5-5e96-808d-cc7f68faa162",
  "TopicArn": "arn:aws:sns:us-east-2:111122223333:ExampleTopic1",
  "Subject": "TestSubject",
  "Message": "This is a test message.",
  "Timestamp": "2021-02-16T21:41:19.978Z",
  "SignatureVersion": "1",
  "Signature": "FMG5tlZhJNHLHUXvZgtZzlk24FzVa7oX0T4P03neeXw8ZEXZx6z35j2FOTuNYShn2h0bKNC/zLTnMy
  "SigningCertURL": "https://sns.us-east-2.amazonaws.com/SimpleNotificationService-010a507c183
  "UnsubscribeURL": "https://sns.us-east-2.amazonaws.com/?Action=Unsubscribe&SubscriptionArn=a
}
```


- Raw message delivery is **enabled**

```
This is a test message.
```

Amazon SNS message filtering

By default, an Amazon SNS topic subscriber receives every message published to the topic. To receive a subset of the messages, a subscriber must assign a filter policy to the topic subscription.

You need to pass the **message attributes** (MessageAttributes) parameter which is a map consists of a key and value.

A policy that accepts messages

If any single attribute in this policy doesn't match an attribute assigned to the message, the policy rejects the message.

```
"MessageAttributes": {
   "customer_interests": {
      "Type": "String.Array",
      "Value": "[\"soccer\", \"rugby\", \"hockey\"]"
   },
   "store": {
      "Type": "String",
      "Value":"example_corp"
   },
   "event": {
      "Type": "String",
      "Value": "order_placed"
   },
   "price_usd": {
      "Type": "Number",
      "Value":210.75
   }
}
```

Payload

```
{
   "store": ["example_corp"],
   "event": [{"anything-but": "order_cancelled"}],
   "customer_interests": [
      "rugby",
      "football",
      "baseball"
   ],
   "price_usd": [{"numeric": [">=", 100]}]
}
```

Policy

A policy that rejects messages

If any mismatches occur, the policy rejects the message. The encrypted attribute isn't present in the message, this causes the message to be rejected regardless of the value assigned to it.

```
"MessageAttributes": {
   "customer_interests": {
      "Type": "String.Array",
      "Value": "[\"soccer\", \"rugby\", \"hockey\"]"
   },
   "store": {
      "Type": "String",
      "Value":"example_corp"
   },
   "event": {
      "Type": "String",
      "Value": "order_placed"
   },
   "price_usd": {
      "Type": "Number",
      "Value":210.75
   }
}
```

Payload

```
{
   "store": ["example_corp"],
   "event": ["order_cancelled"],
   "encrypted": [false],
   "customer_interests": [
      "basketball",
      "baseball"
   ]
}
```

Policy

Message delivery status logging

Amazon SNS supports logging the delivery status of notification messages using CloudWatch. Amazon SNS sends the following metrics to CloudWatch:

- NumberOfMessagesPublished
- NumberOfNotificationsDelivered
- NumberOfNotificationsFailed
- NumberOfNotificationsFilteredOut
- PublishSize – Message size etc.

Based on these metrics, you can set a policy for ASG.

Example log for successful SMS delivery


```
{
    "notification": {
        "messageId": "34d9b400-c6dd-5444-820d-fbeb0f1f54cf",
        "timestamp": "2016-06-28 00:40:34.558"
    },
    "delivery": {
        "phoneCarrier": "My Phone Carrier",
        "mnc": 270,
        "destination": "+1XXX5550100",
        "priceInUSD": 0.00645,
        "smsType": "Transactional",
        "mcc": 310,
        "providerResponse": "Message has been accepted by phone carrier",
        "dwellTimeMs": 599,
        "dwellTimeMsUntilDeviceAck": 1344
    },
    "status": "SUCCESS"
}
```

Example log for failed SMS delivery


```
{
    "notification": {
        "messageId": "1077257a-92f3-5ca3-bc97-6a915b310625",
        "timestamp": "2016-06-28 00:40:34.559"
    },
    "delivery": {
        "mnc": 0,
        "destination": "+1XXX5550100",
        "priceInUSD": 0.00645,
        "smsType": "Transactional",
        "mcc": 0,
        "providerResponse": "Unknown error attempting to reach phone",
        "dwellTimeMs": 1420,
        "dwellTimeMsUntilDeviceAck": 1692
    },
    "status": "FAILURE"
}
```

Unsuccessful message delivery

SNS retries if message is not delivered. Policies can be:

- 3 times immediately in the no-delay phase
- 2 times (1 second apart) in the pre-backoff phase
- 10 times (with exponential backoff from 1 second to 60 seconds)
- 35 times (60 seconds apart) in the post-backoff phase

If retry is not successful, the message is sent to the **dead-letter queue** further analysis or manual reprocessing.

You can have a dead-letter queue for SQS as well.

Application Decoupling

Many applications start to grow in complexity as they mature, making it harder for developers to maintain code or add new features. Decoupling is achieved by having services like SNS and SQS in between 2 applications or microservices.

Decoupling provides:

- Easier to maintain code and change implementations
- Cross-platform, different languages and technologies (**microservices**)
- Independent releases (microservices)
- Better application performance by doing the action asynchronously.
- Fault tolerant – You can keep pushing messages back to the queue if the listener service is not working. Also know as the **circuit breaker design** pattern.

Circuit breaker design pattern

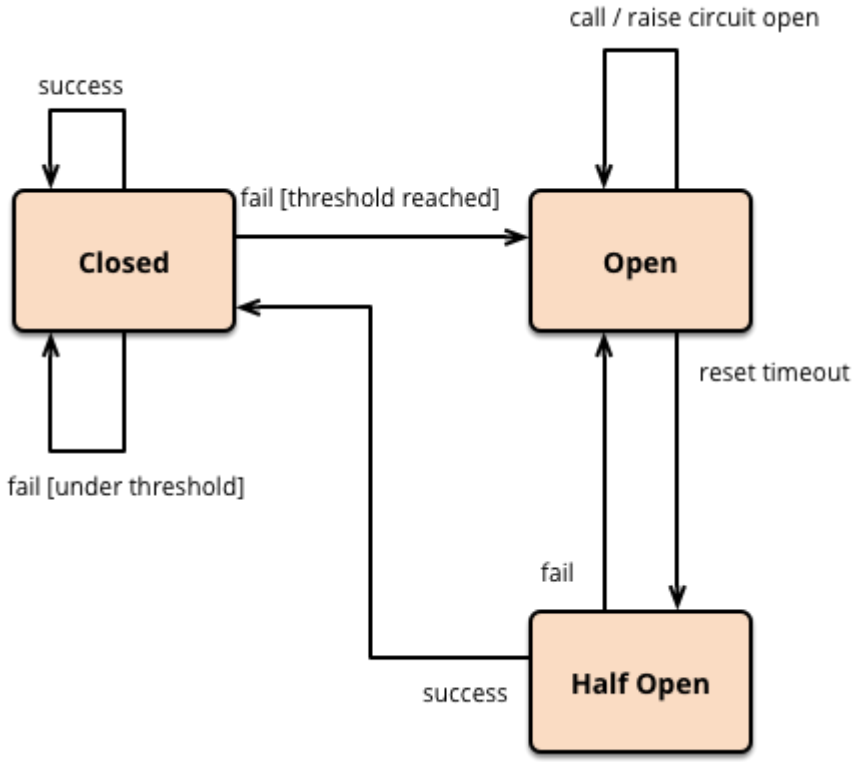
It is used to detect failures and encapsulates the logic of preventing a failure from constantly recurring, during maintenance, temporary external system failure or unexpected system difficulties.

For example, if the database is not available, you can’t keep inserting records into it. Instead, keep it in the queue and send data when the database is back.

Circuit breaker design pattern

Circuit states:

- **Closed** – Underlying system is working. All transactions are success.
- **Open** – The underlying system is **not** working. All transactions failed.
- **Half open** – Sending some requests to see whether the underlying system is available or not.



Popular circuit breaking library: [resilience4j]

Microservices architecture

Service-Oriented Architecture (SOA) - an architectural style that supports service orientation. Application components provide services to the other components through **network**. A **service** is a **discrete unit** of functionality that can be accessed remotely and acted upon and updated **independently**. For example, authorization, account, inventory, shipping. You can use different technologies each service.

Microservice architecture is a variant of the service-oriented architecture (SOA) structural style that arranges an application as a collection of **loosely coupled (decoupled) services**. Microservices are easier to maintain, scale out, **automate the deployment** (CI/CD) and add features.

Event-driven architecture

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** like an item being placed in a shopping cart on an e-commerce website. Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a **notification** that an order was shipped).

Event-driven architectures have three key components:

1. event producers (like SNS publishers)
2. event routers (like SNS topics)
3. event consumers (like SNS subscribers)

Event-driven service in AWS: Amazon EventBridge