AWS Integration & Messaging

Section Introduction

When we start deploying multiple applications, they will inevitably need to communicate with one another.

There are two patterns of application communication:

1. **Synchronous communications**
   Application communicates directly with another application.

2. **Asynchronous / Event-based communications**
   Application sends a message to a queue, which is then consumed by another application.

Communication Examples

Synchronous:

- Buying Service → Shipping Service (direct call)

Asynchronous:

- Buying Service → Queue → Shipping Service

Section Introduction

Synchronous communication between applications can be problematic if there are sudden spikes of traffic.

For example:
What if you need to suddenly encode 1000 videos but usually it's just 10?

Solution: Decouple your applications

- **SQS**: Queue model
- **SNS**: Pub/Sub model
- **Kinesis**: Real-time streaming model

These services allow each part of the system to **scale independently** from the application.

Amazon SQS

What's a queue?

A queue is a temporary repository for messages sent by producers and read by consumers.

- **Producers** send messages to the **SQS Queue**
- **Consumers** poll messages from the **SQS Queue**

This decouples the components of a system, allowing them to scale independently.

Amazon SQS – Standard Queue

- Oldest offering (over 10 years old)
- Fully managed service, used to decouple applications

Attributes

- Unlimited throughput and unlimited number of messages in the queue
- Default message retention: 4 days (maximum: 14 days)
- Low latency: < 10 ms on publish and receive
- Message size limit: 256 KB per message

Delivery Characteristics

- **At-least-once delivery**: duplicate messages are possible
- **Best-effort ordering**: messages may arrive out of order

SQS – Producing Messages

- Messages are produced to SQS using the **SDK** via the `SendMessage` API.
- Once sent, the message is **persisted** in SQS until a **consumer deletes it**.
- **Message retention**: default 4 days, up to 14 days.

Example use case:

Send an order to be processed, with data such as:

- Order ID
- Customer ID
- Any other attributes

Notes:

- **SQS Standard** allows for **unlimited throughput**.
- Each message can be up to **256 KB** in size.

SQS – Consuming Messages

Consumers (such as EC2 instances, on-premise servers, or AWS Lambda) interact with SQS by:

1. **Polling SQS** for messages
    - Can receive up to **10 messages at a time**
2. **Processing the messages**
    - Example: insert the message into an **RDS database**
3. **Deleting the messages**
    - Use the `DeleteMessage` API to remove the message from the queue

SQS – Multiple EC2 Instances Consumers

Key Concepts

- Multiple consumers can **receive and process messages in parallel**
- **At-least-once delivery**: duplicates are possible
- **Best-effort ordering**: messages may be received out of order
- Consumers are responsible for **deleting messages** after processing
- Consumers can be **scaled horizontally** to increase processing throughput

SQS with Auto Scaling Group (ASG)

Architecture Overview

- **EC2 Instances** in an Auto Scaling Group (ASG) **poll for messages** from the SQS queue
- **CloudWatch Metric** `ApproximateNumberOfMessages` tracks the **queue length**
- A **CloudWatch Alarm** is triggered if the metric exceeds a defined threshold
- The **Auto Scaling Group scales out** to add more EC2 instances in response to the alarm

This setup ensures that the message processing capacity scales dynamically based on demand.

SQS to Decouple Between Application Tiers

Architecture Pattern

- A **front-end web application** sends requests using `SendMessage` to the **SQS Queue**
- A **back-end processing application** retrieves those messages using `ReceiveMessages`
- Both tiers can be placed in **Auto Scaling Groups** to handle varying loads

Benefits

- The SQS queue is **infinitely scalable**
- It allows for **decoupling** between front-end and back-end
- Each tier can **scale independently**, improving fault tolerance and scalability

Amazon SQS – Security

Encryption

- **In-flight encryption**: via HTTPS API
- **At-rest encryption**: using AWS KMS keys
- **Client-side encryption**: optional, performed by the client for custom encryption/decryption

Access Control

- **IAM policies**: used to manage access to the SQS API
- **SQS access policies** (similar to S3 bucket policies):
  - Enable **cross-account access** to SQS queues
  - Allow **other AWS services** (e.g., SNS, S3) to write to an SQS queue

SQS – Message Visibility Timeout

- After a message is **polled by a consumer**, it becomes **invisible** to other consumers.
- The **default visibility timeout** is **30 seconds**.
- This means the consumer has **30 seconds to process** the message.
- After the timeout expires, if the message has not been deleted, it becomes **visible again** in the queue and may be received by another consumer.

Timeline Example

1. `ReceiveMessage` request → message becomes invisible
2. During visibility timeout → message is not returned to other consumers
3. After timeout expires → message is returned again in a new `ReceiveMessage` call

SQS – Message Visibility Timeout (continued)

- If a message is **not processed within the visibility timeout**, it will be **delivered again**, potentially leading to **duplicate processing**.

- A consumer can call the `ChangeMessageVisibility` API to **extend the timeout** if more time is needed.
- **Trade-offs**:
  - If the timeout is **too high** (e.g., hours) and the consumer crashes, re-processing the message is **delayed**.
  - If the timeout is **too low** (e.g., seconds), there is a **higher risk of duplicate messages**.

Amazon SQS – Long Polling

- When a consumer requests messages from the queue, it can **wait for messages to arrive** if the queue is empty.
- This behavior is known as **Long Polling**.

Benefits of Long Polling

- **Reduces the number of API calls** to SQS
- **Increases efficiency** and **reduces latency** in your application

Configuration

- The wait time can be set between **1 second to 20 seconds** (20 seconds is recommended)
- Long Polling is **preferable to Short Polling**
- It can be enabled:
  - At the **queue level**
  - At the **API level** using the `WaitTimeSeconds` parameter

Amazon SQS – FIFO Queue

- **FIFO** stands for **First In, First Out**: messages are processed in the exact order they are sent.

Key Features

- **Ordering** is preserved using **Message Group ID** (required parameter)
- **Exactly-once send**: duplicates are removed using **Deduplication ID**
- **Messages are processed in order** by the consumer

Throughput Limits

- Up to **300 messages/second** without batching
- Up to **3000 messages/second** with batching

SQS with Auto Scaling Group (ASG)

Architecture Overview

- **EC2 Instances** in an Auto Scaling Group (ASG) **poll for messages** from the SQS queue
- **CloudWatch Metric** `ApproximateNumberOfMessages` tracks the **queue length**
- A **CloudWatch Alarm** is triggered if the metric exceeds a defined threshold
- The **Auto Scaling Group scales out** to add more EC2 instances in response to the alarm

This setup ensures that the message processing capacity scales dynamically based on demand.

Risk of Data Loss with High Load

When the **application load is too high**, some **transactions may be lost** if the system cannot keep up with the rate of incoming requests.

Architecture Overview

- The **application** handles incoming requests and attempts to **insert transactions** into a database
- The database could be:
  - **Amazon RDS**
  - **Amazon Aurora**
  - **Amazon DynamoDB**
- Even with **Auto Scaling**, without decoupling, spikes in traffic may cause transaction loss

SQS as a Buffer to Database Writes

Using **Amazon SQS** as a buffer helps **absorb spikes in traffic** and protects the database from being overwhelmed.

Architecture

- The **application** sends incoming requests to the **SQS Queue** using `SendMessage`
- The **queue** is **infinitely scalable**
- **Back-end consumers** (in an Auto Scaling Group) use `ReceiveMessages` to read from the queue
- Messages are then **inserted into the database**

Benefits

- **Smooths out spikes in load**
- **Decouples** application from database write throughput
- **Improves durability** and **scalability**

SQS to Decouple Between Application Tiers

Architecture Pattern

- A **front-end web application** sends requests using `SendMessage` to the **SQS Queue**
- A **back-end processing application** retrieves those messages using `ReceiveMessages`
- Both tiers can be placed in **Auto Scaling Groups** to handle varying loads

Benefits

- The SQS queue is **infinitely scalable**
- It allows for **decoupling** between front-end and back-end
- Each tier can **scale independently**, improving fault tolerance and scalability

Amazon SNS

Use Case

What if you want to **send one message to many receivers**?

Solution: Pub/Sub Model with SNS

- The **Buying Service** publishes a message to an **SNS Topic**
- The SNS Topic then **fans out** the message to multiple subscribers, such as:
  - **Email notification**
  - **Fraud Service**
  - **Shipping Service**
  - **SQS Queues**

Benefits

- Enables **Pub/Sub** architecture
- Decouples the **producer** from the **multiple consumers**
- **Direct integration** with various AWS services

Amazon SNS – Overview

- The **event producer** sends messages to **one SNS topic**
- You can have **as many event receivers** (subscriptions) as needed listening to that topic
- **All subscribers receive all messages**
  (note: **message filtering** is available as a new feature)

Limits

- Up to **12,500,000 subscriptions per topic**
- Up to **100,000 topics per account**

Supported Subscribers

- **SQS**
- **Lambda**
- **Kinesis Data Firehose**
- **HTTP(S) endpoints**
- **SMS & Mobile Notifications**
- **Emails**

SNS Integrates with Many AWS Services

Many AWS services can **directly publish notifications** to an SNS topic.

Examples of AWS Services Integrating with SNS

- **CloudWatch Alarms**
- **S3** (Object-created or deleted events)
- **Auto Scaling Groups** (Scaling notifications)
- **CloudFormation** (Stack state changes)
- **AWS Budgets**
- **AWS Lambda**
- **AWS DMS** (New replication events)
- **DynamoDB** (Streams or table events)
- **RDS Events**

SNS acts as a **central pub/sub hub** for AWS event-driven architecture.

Amazon SNS – How to Publish

Topic Publish (using the SDK)

1. **Create a topic**
2. **Create one or more subscriptions**
3. **Publish messages to the topic**

Direct Publish (for mobile apps SDK)

1. **Create a platform application**

2. **Create a platform endpoint**
3. **Publish messages to the platform endpoint**

Supported mobile push platforms include:

- **Google GCM**
- **Apple APNS**
- **Amazon ADM**

Amazon SNS – Security

Encryption

- **In-flight encryption** using the HTTPS API
- **At-rest encryption** using AWS KMS keys
- **Client-side encryption** is also supported for custom encryption/decryption

Access Control

- Use **IAM policies** to control access to the SNS API

SNS Access Policies

- Similar to **S3 bucket policies**
- Useful for:
  - **Cross-account access** to SNS topics
  - Allowing other AWS services (e.g., **S3**) to publish to an SNS topic

SNS + SQS: Fan Out

- **Push once** to an **SNS Topic**, and the message is **delivered to all subscribed SQS queues**
- This creates a **fully decoupled architecture** with **no data loss**
- **SQS** provides:
  - **Data persistence**
  - **Delayed processing**
  - **Retries**

Benefits

- You can **add more SQS subscribers** over time
- Ensure that **SQS access policies** allow SNS to publish to the queues
- Supports **Cross-Region Delivery**: messages can be sent to **SQS queues in other regions**

Application: S3 Events to Multiple Queues

- For a given combination of **event type** (e.g. `ObjectCreated`) and **prefix** (e.g. `images/`), you can configure **only one S3 Event rule**
- If you need to send the same S3 event to **multiple SQS queues**, use the **fan-out pattern** via **SNS**

Architecture

1. **Amazon S3** emits an event (e.g. object created)
2. The event is sent to an **SNS Topic**
3. The SNS Topic fans out the message to:
   - Multiple **SQS Queues**
   - Optionally, a **Lambda Function**

This setup enables **flexible and scalable multi-subscriber event handling**.

Application: SNS to Amazon S3 through Kinesis Data Firehose

Architecture Overview

- The **Buying Service** publishes messages to an **SNS Topic**
- The SNS Topic sends the messages to **Kinesis Data Firehose (KDF)**
- **KDF** delivers the data to **Amazon S3**

Notes

- **SNS can integrate with Kinesis**, enabling this kind of architecture
- You can use **any supported Kinesis Data Firehose destination**, such as:
  - Amazon S3
  - Amazon Redshift
  - Amazon OpenSearch
  - Custom HTTP endpoints

Amazon SNS – FIFO Topic

FIFO = First In First Out: guarantees message ordering in the topic.

Features (similar to SQS FIFO)

- **Ordering** is preserved using **Message Group ID**
  - All messages in the same group are delivered in order
- **Deduplication**:
  - Using a **Deduplication ID**
  - Or **Content-Based Deduplication**

Subscribers

- Can include both:
  - **SQS Standard queues**
  - **SQS FIFO queues**

Throughput

- **Limited throughput**, same as **SQS FIFO**:
  - 300 messages/second without batching
  - 3000 messages/second with batching

SNS FIFO + SQS FIFO: Fan Out

Use this setup when you need:

- **Fan-out** (send to multiple subscribers)
- **Ordering** (First In First Out delivery)
- **Deduplication** (prevent duplicate messages)

Architecture

- The **Buying Service** publishes to an **SNS FIFO Topic**
- The SNS FIFO Topic fans out the message to multiple **SQS FIFO Queues**, such as:

    - **Fraud Service**
    - **Shipping Service**

This configuration ensures **reliable delivery**, **exact ordering**, and **duplicate protection** across multiple subscribers.

SNS – Message Filtering

Overview

- SNS uses **JSON filter policies** to control which messages are delivered to specific **subscriptions**
- If a subscription has **no filter policy**, it receives **all messages**

Example Use Case

A **Buying Service** publishes a message about a new transaction:

```
{
  "Order": 1036,
  "Product": "Pencil",
  "Qty": 4,
  "State": "Placed"
}
```

Different subscribers can apply **filter policies** based on the `State`:

- **SQS Queue (Placed orders)**: receives messages with `State: Placed`
- **Email Subscription (Cancelled orders)**: receives messages with `State: Cancelled`
- **SQS Queue (Declined orders)**: receives messages with `State: Declined`
- **SQS Queue (All)**: receives all messages (no filter policy)

Benefit

Allows **targeted delivery** of messages to the appropriate consumers.

Amazon Kinesis Data Streams

Used to **collect and store streaming data in real-time** from a variety of producers.

Typical Data Sources

- **Click Streams**
- **IoT Devices**
- **Metrics & Logs**

Producers

- **Kinesis Agent**
- **Custom Applications**

Consumers

- **AWS Lambda**
- **Amazon Kinesis Data Firehose**
- **Managed Service for Apache Flink**
- Other real-time processing applications

Key Benefit

Provides a scalable and durable platform for **real-time data ingestion and processing**.

Kinesis Data Streams – Key Features

- **Retention**: configurable up to **365 days**
- **Replay support**: consumers can **reprocess old data**
- **Immutability**: data **cannot be deleted** manually; it expires after the retention period
- **Data size**: up to **1 MB** per record
  (designed for **many small** real-time records)
- **Ordering guarantee** for records with the same **Partition Key**
- **Encryption**:
  - **In-flight** via HTTPS
  - **At-rest** via AWS KMS

Developer Tools

- **Kinesis Producer Library (KPL)**: helps build high-performance producers
- **Kinesis Client Library (KCL)**: helps build high-performance consumers

Kinesis Data Streams – Capacity Modes

Provisioned Mode

- You **manually configure** the number of **shards**
- Each shard provides:
  - **1 MB/s** or **1000 records/s ingestion**
  - **2 MB/s egress**
- Manual **scaling** by adding/removing shards
- **Billing**: charged **per shard per hour**

On-Demand Mode

- **No provisioning or manual management** required
- Default capacity: **4 MB/s in** or **4000 records/s**
- **Auto-scales** based on observed peak usage in the **last 30 days**
- **Billing**: charged **per stream per hour** and **per GB** of data in/out

Amazon Data Firehose

Producers

- **Applications**
- **Clients** via **SDK**
- **Kinesis Agent**
- **Kinesis Data Streams**
- **Amazon CloudWatch** (Logs & Events)
- **AWS IoT**

Processing

- Firehose can perform **data transformation** using an optional **AWS Lambda function**
- Accepts records **up to 1 MB**
- Performs **batch writes** to destinations

- Supports **S3 backup bucket** for:
  - **All data**
  - Or only **failed data**

Destinations

- **Amazon S3**
- **Amazon Redshift**
- **Amazon OpenSearch**
- **HTTP Endpoint**
- **3rd-party Partners** (e.g., Datadog)
- **Custom destinations**

Firehose is fully managed and scales automatically to handle incoming data.

Amazon Data Firehose

> *Formerly known as **Kinesis Data Firehose***

Key Characteristics

- **Fully managed**, **serverless**, and **auto-scaling**
- **Pay-as-you-go** pricing model
- Works in **near real-time** with **buffering** based on **size or time**

Supported Destinations

- **AWS services**:
  - Amazon Redshift
  - Amazon S3
  - Amazon OpenSearch Service
- **Third-party integrations**:
  - Splunk
  - MongoDB
  - Datadog
  - NewRelic
- **Custom HTTP Endpoints**

Supported Data Formats

- **Input formats**: CSV, JSON, Parquet, Avro, Raw Text, Binary
- **Output transformations**:
  - Convert to **Parquet** or **ORC**
  - Compress using **gzip** or **snappy**

Transformations

- Use **AWS Lambda** for **custom data transformations**
  (e.g., convert CSV to JSON)

Kinesis Data Streams vs Amazon Data Firehose

Feature	Kinesis Data Streams	Amazon Data Firehose

Purpose	Streaming data collection	Load streaming data into destinations
Control	Requires producer & consumer code	Fully managed
Latency	Real-time	Near real-time
Scaling	Provisioned or On-Demand	Automatic scaling
Data Storage	Up to 365 days	No data storage
Replay Capability	Supported	Not supported
Destinations	Custom consumers	S3, Redshift, OpenSearch, 3rd party, HTTP

SQS vs SNS vs Kinesis

SQS (Simple Queue Service)

- **Consumer pulls** data
- Data is **deleted after being consumed**
- Supports **many workers** (consumers)
- **No throughput provisioning** required
- **FIFO ordering** only with FIFO queues
- Supports **individual message delay**

SNS (Simple Notification Service)

- **Pushes data** to many subscribers
- Supports up to **12,500,000 subscribers**
- Data is **not persisted** (lost if not delivered)
- **Pub/Sub** model
- Up to **100,000 topics**
- **No throughput provisioning** required
- Integrates with **SQS for fan-out**
- **FIFO capability** when publishing to SQS FIFO queues

Kinesis Data Streams

- **Standard mode**: consumer **pulls** data (2 MB/s per shard)
- **Enhanced fan-out**: SNS-style **push** (2 MB/s **per shard per consumer**)
- Supports **data replay**
- Designed for **real-time big data**, analytics, and ETL
- Guarantees **ordering at the shard level**
- Data **expires after configurable retention period**
- Can be run in **provisioned** or **on-demand** mode

Amazon MQ

When to Use Amazon MQ

- **SQS** and **SNS** are **cloud-native** and use **AWS proprietary protocols**
- Legacy or on-premises applications often use **open protocols**, such as:
  - **MQTT**

    - **AMQP**
    - **STOMP**
    - **Openwire**
    - **WSS**

Migration Strategy

- Instead of **re-engineering** applications to use SQS/SNS, use **Amazon MQ** as a **drop-in replacement**

Features

- **Managed message broker service**
- **Supports queues** (similar to SQS)
- **Supports topics** (similar to SNS)
- **Runs on servers**, can be deployed in **Multi-AZ** with **failover**
- **Less scalable** than SQS/SNS, but ideal for compatibility with existing systems

Amazon MQ – High Availability

Multi-AZ Architecture

Amazon MQ supports **high availability** by running brokers in **multiple Availability Zones** within a region.

Example: Region `us-east-1`

- **AZ 1 (`us-east-1a`)**:
    - Amazon MQ Broker (ACTIVE)
- **AZ 2 (`us-east-1b`)**:
    - Amazon MQ Broker (STANDBY)
- **Shared storage**: uses **Amazon EFS**

Behavior

- **Automatic failover**: if the active broker fails, the **standby broker takes over**
- Ensures **durability** and **resilience** for message processing

Client-Side

- Clients must support **failover connection logic**