AWS SNS, Kubernetes, and Event Handling

Question 1:

What's the difference between SNS and SQS?

Response:

SNS (Simple Notification Service) and SQS (Simple Queue Service) are two AWS messaging services with distinct purposes.

1. Amazon SNS (Simple Notification Service)

- Purpose: Publish/subscribe messaging service for broadcasting messages to multiple subscribers.

- Delivery Model: Push-based delivery.

- Features: Supports multiple subscriber types (HTTP/S, Lambda, Email, SQS, SMS).

- Message Retention: No retention; messages are pushed immediately.

- Use Case: Broadcasting notifications, event-driven architectures.

2. Amazon SQS (Simple Queue Service)

- Purpose: Point-to-point messaging for decoupling distributed systems and enabling asynchronous processing.

- Delivery Model: Pull-based delivery.

- Features: Retains messages for up to 14 days, FIFO support, dead-letter queues.

- Use Case: Task queues, decoupling microservices.

Key Differences:

- SNS is for pub/sub, broadcasting messages to multiple subscribers.

- SQS is for point-to-point, reliable queuing for asynchronous processing.

Use Cases:

- Use SNS for broadcasting notifications.

- Use SQS for task processing or decoupling.

- Combine SNS and SQS for fan-out messaging patterns.

Question 2:

How can I register my server with SNS and how would events arrive?

Response:

1. Expose an HTTP endpoint in your server to receive SNS messages. For example, using Akka HTTP:

Code:

```scala

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.server.Directives._

val route =
  path("sns") {
    post {
      entity(as[String]) { body =>
```

println(s"Received SNS message: $body")
complete(StatusCodes.OK)
}
}
}

Http().newServerAt("0.0.0.0", 8080).bind(route)
```

2. Register the endpoint with SNS using the AWS SDK.

Code:
```scala
import software.amazon.awssdk.services.sns.SnsClient
import software.amazon.awssdk.services.sns.model.SubscribeRequest

val snsClient = SnsClient.builder().build()
val subscribeRequest = SubscribeRequest.builder()
.topicArn("arn:aws:sns:region:account-id:topic-name")
.protocol("http")
.endpoint("http://<server-ip>:8080/sns")
.build()

snsClient.subscribe(subscribeRequest)
```

3. Handle SNS SubscriptionConfirmation by making an HTTP GET request to
```

the SubscribeURL.

Code:

```scala
path("sns") {
  post {
    entity(as[String]) { body =>
      val snsMessage = parseJson(body)
      snsMessage.get("Type") match {
        case "SubscriptionConfirmation" =>
          val confirmUrl = snsMessage.get("SubscribeURL").asString()
          Http().singleRequest(HttpRequest(uri = confirmUrl))
        case "Notification" =>
          println(s"Received SNS Notification: ${snsMessage.get("Message")}")
      }
      complete(StatusCodes.OK)
    }
  }
}
```

4. Use Kubernetes Service Discovery or a single subscriber to relay messages to all pods.

Question 3:

Do I need a load balancer if I'm using Kubernetes for my normal REST APIs? If so, when SNS fires an event, I still want every instance in the cluster to respond.

Response:

You might need a load balancer depending on your use case. A load balancer typically routes external traffic to the cluster, distributing it across pods. However, if every instance in the cluster must respond to SNS events, you'll need a fan-out mechanism.

Options:

1. **Direct Subscription:**

- Expose each pod's endpoint (e.g., NodePort or ClusterIP).

- Each pod subscribes directly to SNS and processes events individually.

2. **Single Entry Point with Internal Broadcasting:**

- Use a single public endpoint (e.g., LoadBalancer or Ingress) to subscribe to SNS.

- The receiving pod broadcasts messages to all pods in the cluster.

3. **SNS to Message Broker:**

- Use a message broker like RabbitMQ or NATS as an intermediary.

- All pods subscribe to the broker and receive events.

Example for internal broadcasting in Scala:

```scala

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.actor.ActorSystem
import scala.concurrent.Future

object Broadcast {
  implicit val system: ActorSystem = ActorSystem("sns-broadcast")
  import system.dispatcher

  def broadcastToAllPods(event: String): Future[Unit] = {
                        val podIps = List("pod1.default.svc.cluster.local",
"pod2.default.svc.cluster.local")
    val broadcastFutures = podIps.map { podIp =>
      val url = s"http://$podIp:8080/sns/internal"
      Http().singleRequest(HttpRequest(
        method = HttpMethods.POST,
        uri = url,
        entity = HttpEntity(ContentTypes.`application/json`, event)
      ))
    }
    Future.sequence(broadcastFutures).map(_ => ())
  }
}
```

Let me know if further examples or clarifications are needed!