Repaso de computación en la Nube y servicios AWS

Temas: Servicio de integración de aplicaciones en AWS

Objetivos

La integración de aplicaciones es el proceso de lograr que los sistemas de software creados de forma independiente funcionen juntos sin intervención manual, permite a los desarrolladores crear aplicaciones que reutilizan los servicios y sistemas existentes. De esta forma, pueden hacer más con menos codificación y facilita la automatización, ya que las aplicaciones pueden comunicarse entre sí para manejar flujos de trabajo empresariales complejos.

Los objetivos de este repaso son profundizar estos temas en tres niveles: desde el punto de vista teórico, dsde el punto de vista páctico usando las herramientas de AWS y por codificación a través de conceptos rudimentarios de python.

Tareas

Prepara una presentación grupal para responder las siguientes preguntas y utiliza un ide de tu preferencia para las implementaciones solicitadas:

Preguntas

Amazon SNS (Simple Notification Service)

Descripción y casos de uso:

- **Pregunta**: Explica qué es Amazon SNS y describe al menos tres casos de uso en los que su implementación sería beneficiosa.
- **Respuesta Esperada**: Amazon SNS es un servicio de mensajería gestionado que coordina la entrega de mensajes a través de varios protocolos. Casos de uso incluyen notificaciones automáticas a usuarios finales, alertas en sistemas de monitoreo y coordinación de microservicios.

Endpoints y topics:

- **Pregunta**: Define qué es un endpoint y un topic en el contexto de Amazon SNS. ¿Cómo se relacionan estos conceptos?
- **Respuesta esperada**: Un endpoint es el destino donde se envían los mensajes (por ejemplo, una dirección de correo electrónico o un número de teléfono móvil). Un topic es un canal de

comunicación que agrupa varios endpoints. Los mensajes se publican en un topic y se distribuyen a todos los endpoints suscritos a ese topic.

Standard y FIFO Topics:

- **Pregunta**: Compara y contrasta los topics Standard y FIFO en Amazon SNS. ¿En qué escenarios se recomendaría utilizar cada uno?
- **Respuesta esperada**: Los topics Standard permiten una alta tasa de entrega de mensajes sin un orden garantizado. Los topics FIFO garantizan el orden de los mensajes y la entrega exacta una vez. Standard es útil para notificaciones generales, mientras que FIFO es adecuado para tareas que requieren orden estricto y procesamientos únicos.

Escenario de fanout:

- **Pregunta**: Describe un escenario de Fanout utilizando Amazon SNS. ¿Cómo se configura y qué beneficios proporciona?
- **Respuesta Esperada**: En un escenario de Fanout, un mensaje publicado en un topic SNS se distribuye a múltiples colas SQS, lambda functions u otros endpoints. Esto permite procesar el mismo mensaje en paralelo por diferentes sistemas, mejorando la eficiencia y separación de responsabilidades.

Precio de Amazon SNS:

- **Pregunta**: Explica cómo se estructura la tarifa de Amazon SNS. ¿Cuáles son los factores principales que afectan el costo?
- **Respuesta esperada**: Los costos de Amazon SNS se basan en el número de solicitudes de publicación de mensajes y el tipo de protocolo utilizado para la entrega (HTTP, SQS, Lambda, SMS, etc.). Los factores principales incluyen el número de mensajes enviados y el tipo de endpoints destinatarios.

Amazon SQS (Simple Queue Service) y Amazon MQ

Tipos de colas en Amazon SQS:

- **Pregunta**: Describe los diferentes tipos de colas en Amazon SQS y sus principales diferencias.
- **Respuesta esperada**: Amazon SQS ofrece colas Standard y FIFO. Las colas Standard permiten un alto rendimiento y no garantizan el orden de los mensajes. Las colas FIFO garantizan el orden y la entrega exacta una vez. FIFO es preferido para tareas que requieren procesamiento ordenado y único.

Precio y seguridad de Amazon SQS:

- **Pregunta**: ¿Cómo se calcula el precio de Amazon SQS y qué características de seguridad ofrece?

- **Respuesta esperada**: El precio de Amazon SQS se basa en la cantidad de solicitudes (envío y recepción de mensajes) y la cantidad de datos transferidos. Ofrece características de seguridad como cifrado en reposo y en tránsito, control de acceso mediante políticas IAM, y soporte para VPC endpoints.

Amazon MQ:

- **Pregunta**: Explica qué es Amazon MQ y sus ventajas frente a Amazon SQS.
- **Respuesta esperada**: Amazon MQ es un servicio de broker de mensajes gestionado compatible con estándares de la industria como AMQP, MQTT, OpenWire y STOMP. Proporciona capacidades avanzadas como persistencia de mensajes, transacciones y procesamiento de mensajes en secuencia, lo que lo hace ideal para migraciones de aplicaciones legacy.

AWS EventBridge y Amazon Step Functions

Diseño de Workflows Event-Driven con AWS EventBridge:

- **Pregunta**: Describe cómo diseñar una arquitectura event-driven utilizando AWS EventBridge. ¿Cuáles son las ventajas de usar EventBridge?
- **Respuesta esperada**: AWS EventBridge permite la creación de arquitecturas basadas en eventos mediante la entrega de eventos de diferentes fuentes a destinos como Lambda, SQS, y Step Functions. Las ventajas incluyen la integración simple, la gestión centralizada de eventos y el desencadenamiento de acciones en respuesta a eventos específicos.

Coordinación de servicios con AWS Step Functions:

- **Pregunta**: ¿Qué es AWS Step Functions y cómo facilita la coordinación de múltiples servicios AWS en flujos de trabajo sin servidor?
- **Respuesta esperada**: AWS Step Functions es un servicio de coordinación de flujos de trabajo que permite definir secuencias de tareas que interactúan con diversos servicios AWS. Facilita la construcción de aplicaciones distribuidas con flujos de trabajo visuales, seguimiento de estado y gestión de errores.

Tipos de Workflows y Amazon SWF:

- **Pregunta**: Compara AWS Step Functions con Amazon SWF. ¿En qué escenarios se prefiere uno sobre el otro?
- **Respuesta esperada**: AWS Step Functions proporciona una interfaz visual y facilita la creación de flujos de trabajo con múltiples servicios AWS. Amazon SWF, aunque más antiguo, ofrece control granular sobre los flujos de trabajo con soporte para procesos distribuidos complejos. Step Functions es ideal para nuevos proyectos y flujos de trabajo sin servidor, mientras que SWF es útil para migraciones de flujos de trabajo legacy.

Ejercicios prácticos

Simulación de Arquitectura de notificaciones:

- **Ejercicio**: Implementa una arquitectura en AWS donde un mensaje publicado en un topic SNS se distribuye a varias colas SQS y funciones Lambda. Configura y prueba la entrega y procesamiento de mensajes.

Implementación de un Workflow con Step Functions:

- **Ejercicio**: Diseña y despliega un flujo de trabajo con AWS Step Functions que coordine varios servicios, como Lambda, DynamoDB, y S3. Incluye pasos de éxito, fracaso y manejo de errores.

Monitoreo y optimización de mensajería:

- **Ejercicio**: Configura un sistema de monitoreo para las colas SQS y topics SNS utilizando CloudWatch. Ajusta las configuraciones para optimizar el rendimiento y minimizar los costos.

AWS Lab Learner

1. Amazon SNS (Simple Notification Service)

Ejercicio: Implementación de Notificaciones con Amazon SNS

Objetivo: Configurar un sistema de notificaciones utilizando Amazon SNS que envíe mensajes a múltiples endpoints (correo electrónico y SMS).
Pasos:

- Crea un topic SNS en la consola de AWS.
- Suscribe un endpoint de correo electrónico y un número de teléfono móvil al topic.
- Publica un mensaje en el topic y verificar la entrega en los endpoints suscritos.
- Configura una política de acceso que permita a un usuario específico publicar mensajes en el topic.

2. Amazon SQS (Simple Queue Service)

Ejercicio: Configuración de colas SQS y procesamiento de mensajes

Objetivo: Configurar colas SQS (Standard y FIFO) y crear un script en Python que envíe y reciba mensajes de estas colas.
Pasos:

- Crea una cola Standard y una cola FIFO en la consola de AWS.
- Escribe un script en Python que utilice el SDK de Boto3 para enviar mensajes a ambas colas.

- Escribe otro script en Python que reciba y procese mensajes de las colas.
- Implementa un mecanismo de manejo de errores y reintentos para el procesamiento de mensajes.

3. Amazon MQ

Ejercicio: Configuración y uso de Amazon MQ

Objetivo: Configurar un broker de mensajes con Amazon MQ y utilizarlo para enviar y recibir mensajes entre dos aplicaciones.
Pasos:

- Crea un broker de Amazon MQ en la consola de AWS.
- Configura dos aplicaciones (pueden ser scripts en Python) que se conecten al broker y se comuniquen a través de mensajes.
- Prueba la comunicación entre las aplicaciones y monitorear el tráfico de mensajes en la consola de Amazon MQ.
- Configura políticas de seguridad para restringir el acceso al broker.

4. AWS EventBridge

Ejercicio: Diseño de un flujo de trabajo basado en eventos

Objetivo: Utilizar AWS EventBridge para crear un flujo de trabajo que reaccione a eventos específicos y desencadene acciones en otros servicios de AWS.
Pasos:

- Crea un bus de eventos en AWS EventBridge.
- Configura una regla que escuche eventos específicos (por ejemplo, cambios en un bucket de S3).
- Configura acciones desencadenadas por la regla (por ejemplo, ejecutar una función Lambda).
- Prueba el flujo de trabajo subiendo un archivo a S3 y verificar que la función Lambda se ejecute correctamente.

5. AWS Step Functions

Ejercicio: Coordinación de Tareas con AWS Step Functions

Objetivo: Crea y ejecuta un flujo de trabajo utilizando AWS Step Functions que coordine varias tareas, incluyendo funciones Lambda y operaciones en DynamoDB.
Pasos:

- Crea un nuevo estado de máquina en AWS Step Functions.
- Define un flujo de trabajo que incluya al menos tres estados: una función Lambda que procese datos, una operación en DynamoDB, y un estado de espera.

- Implementa las funciones Lambda necesarias y crear la tabla DynamoDB.
- Ejecuta el flujo de trabajo y monitorear su progreso y resultados en la consola de AWS Step Functions.

6. Amazon Simple Workflow Service (SWF)

Ejercicio: Implementación de un Workflow con Amazon SWF

Objetivo: Configurar un flujo de trabajo simple utilizando Amazon SWF y demostrar cómo coordinar tareas distribuidas.
Pasos:

- Crea un dominio SWF en la consola de AWS.
- Define un flujo de trabajo que incluya al menos dos tareas: una tarea manual y una tarea automática.
- Escribe un script en Python que registre y ejecute las tareas del flujo de trabajo.
- Prueba el flujo de trabajo ejecutando el script y verificando la correcta finalización de las tareas.

Ejemplos de código

1. Simulación de Amazon SNS

Simulación de envío de notificaciones

```
class SNS:
    def __init__(self):
        self.topics = {}

    def create_topic(self, name):
        self.topics[name] = []
        return name

    def subscribe(self, topic, endpoint):
        if topic in self.topics:
            self.topics[topic].append(endpoint)
        else:
            raise Exception("Topic does not exist")

    def publish(self, topic, message):
        if topic in self.topics:
            for endpoint in self.topics[topic]:
```


```
            endpoint.notify(message)
        else:
            raise Exception("Topic does not exist")

class Endpoint:
    def __init__(self, name):
        self.name = name

    def notify(self, message):
        print(f"Notification to {self.name}: {message}")

# Ejemplo de uso
sns = SNS()
email = Endpoint("Email")
sms = Endpoint("SMS")

topic = sns.create_topic("MyTopic")
sns.subscribe(topic, email)
sns.subscribe(topic, sms)

sns.publish(topic, "Hello, world!")
```

2. Simulación de Amazon SQS

Simulación de colas

```
import queue

class SQS:
    def __init__(self):
        self.queues = {}

    def create_queue(self, name, fifo=False):
        self.queues[name] = queue.Queue()
        return name

    def send_message(self, queue_name, message):
        if queue_name in self.queues:
            self.queues[queue_name].put(message)
        else:
            raise Exception("Queue does not exist")

    def receive_message(self, queue_name):
```


```
        if queue_name in self.queues:
            return self.queues[queue_name].get()
        else:
            raise Exception("Queue does not exist")

# Ejemplo de uso
sqs = SQS()
queue_name = sqs.create_queue("MyQueue")

sqs.send_message(queue_name, "Hello, SQS!")
print(sqs.receive_message(queue_name))
```

3. Simulación de Amazon MQ

Simulación de broker de mensajee

```
import threading

class MessageBroker:

    def __init__(self):

        self.topics = {}


    def create_topic(self, name):

        self.topics[name] = []

        return name


    def subscribe(self, topic, endpoint):

        if topic in self.topics:

            self.topics[topic].append(endpoint)

        else:

            raise Exception("Topic does not exist")
```


```
    def publish(self, topic, message):
        if topic in self.topics:
            for endpoint in self.topics[topic]:
                endpoint.notify(message)
        else:
            raise Exception("Topic does not exist")


class Endpoint:
    def __init__(self, name):
        self.name = name

    def notify(self, message):
        print(f"Message to {self.name}: {message}")

# Ejemplo de uso
broker = MessageBroker()
service1 = Endpoint("Service1")
service2 = Endpoint("Service2")

topic = broker.create_topic("MyTopic")
broker.subscribe(topic, service1)
broker.subscribe(topic, service2)

broker.publish(topic, "New message for services")
```

4. Simulación de AWS Step Functions

Simulación de flujos de trabajo

```
class StateMachine:
    def __init__(self, states):
        self.states = states
        self.current_state = list(states.keys())[0]

    def execute(self, data):
        while self.current_state:
            state = self.states[self.current_state]
            print(f"Executing {self.current_state}")
            self.current_state, data = state(data)

def state1(data):
    data["value"] += 1
    return "State2", data

def state2(data):
    data["value"] *= 2
    return None, data

states = {
    "State1": state1,
    "State2": state2,
}
```

```
sm = StateMachine(states)

data = {"value": 1}

sm.execute(data)

print("Final data:", data)
```

5. Simulación de Amazon SWF

Simulación de servicio de flujo de trabajo simple

```
class SWF:
    def __init__(self):
        self.workflows = {}

    def register_workflow(self, name, workflow):
        self.workflows[name] = workflow

    def start_workflow(self, name, data):
        if name in self.workflows:
            self.workflows[name](data)
        else:
            raise Exception("Workflow does not exist")

def workflow(data):
    print("Starting workflow")
    data["step1"] = "completed"
    data["step2"] = "completed"
```


```
    print("Workflow data:", data)


# Ejemplo de uso
swf = SWF()
swf.register_workflow("MyWorkflow", workflow)
data = {"initial": "data"}
swf.start_workflow("MyWorkflow", data)
```

6. Simulación de AWS EventBridge

Simulación de eventos

```
class EventBridge:
    def __init__(self):
        self.rules = []

    def create_rule(self, event_pattern, targets):
        self.rules.append({"pattern": event_pattern, "targets": targets})

    def put_event(self, event):
        for rule in self.rules:
            if all(item in event.items() for item in rule["pattern"].items()):
                for target in rule["targets"]:
                    target.notify(event)

class Target:
    def __init__(self, name):
```

```
        self.name = name


    def notify(self, event):
        print(f"Target {self.name} received event: {event}")


# Ejemplo de uso
event_bridge = EventBridge()
lambda_target = Target("LambdaFunction")


event_bridge.create_rule({"source": "my.app"}, [lambda_target])
event_bridge.put_event({"source": "my.app", "detail": "example event"})
```

Código

Ejercicio 1: Sistema completo de notificaciones con Amazon SNS

Descripción: Implementar un sistema de notificaciones que soporte múltiples tipos de endpoints (email, SMS, webhooks) y permita la gestión de suscripciones y publicaciones de mensajes.

Requisitos:

- Crea una clase SNSEndpoint para representar diferentes tipos de endpoints.
- Implementa métodos para agregar y eliminar endpoints de un topic.
- Implementa métodos para publicar mensajes a los endpoints suscritos.
- Simula el envío de notificaciones a diferentes tipos de endpoints.

Código inicial:

```
class SNS:
    def __init__(self):
        self.topics = {}
    def create_topic(self, name):
        self.topics[name] = []
```


```
        return name

    def subscribe(self, topic, endpoint):
        if topic in self.topics:
            self.topics[topic].append(endpoint)
        else:
            raise Exception("Topic does not exist")

    def unsubscribe(self, topic, endpoint):
        if topic in self.topics:
            self.topics[topic].remove(endpoint)
        else:
            raise Exception("Topic does not exist")
    def publish(self, topic, message):
        if topic in self.topics:
            for endpoint in self.topics[topic]:
                endpoint.notify(message)
        else:
            raise Exception("Topic does not exist")
class SNSEndpoint:
    def __init__(self, name, endpoint_type):
        self.name = name
        self.endpoint_type = endpoint_type

    def notify(self, message):
```

```
        print(f"Notification to {self.endpoint_type} {self.name}: {message}")


# Implementación adicional requerida...
```

Ejercicio 2: Simulación avanzada de Amazon SQS con prioridades y retries

Descripción: Crear una simulación avanzada de Amazon SQS que soporte colas con prioridad y manejo de reintentos en caso de fallos.

Requisitos:

- Crea una clase SQSQueue que soporte colas con diferentes prioridades.
- Implementa un mecanismo para reintentar mensajes fallidos.
- Simula el procesamiento de mensajes con un sistema de prioridad y reintentos.

Código inicial:

```
import heapq
import time
import random

class SQSQueue:
    def __init__(self):
        self.queue = []

    def send_message(self, message, priority=0):
        heapq.heappush(self.queue, (priority, message))

    def receive_message(self):
        if self.queue:
            return heapq.heappop(self.queue)[1]
        else:
            return None

    def process_message(self, message):
        try:
            # Simular procesamiento de mensajes
            if random.choice([True, False]):
                raise Exception("Error processing message")
            print(f"Processed message: {message}")
        except Exception as e:
            print(f"Failed to process message: {message}. Retrying...")
            self.send_message(message, priority=1)
```

Implementación adicional requerida...

Ejercicio 3: Sistema de mensajería completo con Amazon MQ

Descripción: Implementar un sistema de mensajería completo que soporte la publicación y suscripción de mensajes entre múltiples servicios.

Requisitos:

- Crea una clase MessageBroker para manejar la publicación y suscripción de mensajes.
- Implementa la capacidad de agregar y eliminar suscripciones.
- Simula la comunicación entre múltiples servicios utilizando threads para representar servicios concurrentes.

Código inicial:

```
import threading

class MessageBroker:
    def __init__(self):
        self.topics = {}

    def create_topic(self, name):
        self.topics[name] = []
        return name

    def subscribe(self, topic, endpoint):
        if topic in self.topics:
            self.topics[topic].append(endpoint)
        else:
            raise Exception("Topic does not exist")

    def unsubscribe(self, topic, endpoint):
        if topic in self.topics:
            self.topics[topic].remove(endpoint)
        else:
            raise Exception("Topic does not exist")

    def publish(self, topic, message):
        if topic in self.topics:
            for endpoint in self.topics[topic]:
                threading.Thread(target=endpoint.notify, args=(message,)).start()
```


```
        else:
            raise Exception("Topic does not exist")

class Endpoint:
    def __init__(self, name):
        self.name = name

    def notify(self, message):
        print(f"Message to {self.name}: {message}")

# Implementación adicional requerida...
```

Ejercicio 4: Orquestación de tareas con AWS Step Functions

Descripción: Implementar una simulación de AWS Step Functions que coordine varias tareas representadas por funciones en Python.

Requisitos:

- Crea una clase StateMachine que gestione el flujo de trabajo.
- Define múltiples estados y transiciones entre ellos.
- Implementa el manejo de errores y reintentos en cada estado.
- Prueba la ejecución de un flujo de trabajo completo.

```
class StateMachine:
    def __init__(self, states):
        self.states = states
        self.current_state = list(states.keys())[0]

    def execute(self, data):
        while self.current_state:
            state = self.states[self.current_state]
            print(f"Executing {self.current_state}")
            try:
                self.current_state, data = state(data)
            except Exception as e:
                print(f"Error in {self.current_state}: {e}. Retrying...")
                # Implementar lógica de reintento...

def state1(data):
    data["value"] += 1
    return "State2", data
```


```
def state2(data):
    if data["value"] % 2 != 0:
        raise Exception("Value is not even")
    data["value"] *= 2
    return None, data

states = {
    "State1": state1,
    "State2": state2,
}

sm = StateMachine(states)
data = {"value": 1}
sm.execute(data)
print("Final data:", data)
```

Ejercicio 5: Diseñar Workflows con AWS EventBridge

Descripción: Crear una simulación de AWS EventBridge que permita el diseño de flujos de trabajo basados en eventos y la interacción con diferentes servicios.

Requisitos:

1. Crear una clase EventBridge que gestione la creación de reglas y la publicación de eventos.
2. Implementar la capacidad de agregar múltiples objetivos para cada regla.
3. Simular la generación y manejo de eventos en un flujo de trabajo.

Código Inicial:

```
class EventBridge:
    def __init__(self):
        self.rules = []

    def create_rule(self, event_pattern, targets):
        self.rules.append({"pattern": event_pattern, "targets": targets})

    def put_event(self, event):
        for rule in self.rules:
            if all(item in event.items() for item in rule["pattern"].items()):
                for target in rule["targets"]:
                    target.notify(event)

class Target:
    def __init__(self, name):
```


```
        self.name = name

    def notify(self, event):
        print(f"Target {self.name} received event: {event}")

# Implementación adicional requerida...
```

Ejercicio 6: Sistema de Notificaciones con Prioridades y Reintentos

Descripción:
Implementar un sistema de notificaciones que soporte la asignación de prioridades a los mensajes y un mecanismo de reintentos en caso de fallo al enviar la notificación.

Requisitos:

- Crear una clase NotificationSystem que gestione topics y suscripciones.
- Permitir la asignación de prioridades a los mensajes publicados.
- Implementar un mecanismo de reintento con un límite máximo de intentos para los mensajes fallidos.
- Simular el envío de notificaciones a endpoints con diferentes tipos (email, SMS, webhook) y simular posibles fallos de entrega.

Objetivos:

- Implementar métodos para crear topics y suscripciones.
- Crear un mecanismo de encolado de mensajes con prioridades.
- Diseñar un sistema de reintentos con un contador de intentos y un límite máximo.
- Implementar clases de endpoints que representen diferentes tipos de notificaciones y simulen fallos aleatorios.

Ejercicio 7: Sistema de colas con mensajes prioritarios y expiración

Descripción:

- Desarrollar un sistema de colas que soporte mensajes prioritarios y la expiración de mensajes no procesados dentro de un tiempo determinado.

Requisitos:

- Implementar una clase PriorityQueue que soporte múltiples niveles de prioridad.
- Añadir un mecanismo para establecer tiempos de expiración en los mensajes.
- Implementar un sistema que elimine los mensajes expirados de la cola.
- Simular el envío, recepción y procesamiento de mensajes, considerando tanto las prioridades como las expiraciones.

Objetivos:

- Crear métodos para enviar y recibir mensajes con prioridad y tiempo de expiración.
- Implementar una función que verifique y elimine mensajes expirados de la cola periódicamente.
- Simular diferentes escenarios de procesamiento, incluyendo casos donde algunos mensajes expiran antes de ser procesados.

Ejercicio 8: Broker de mensajes con gestión de sesiones

Descripción:
Desarrollar un broker de mensajes que soporte la gestión de sesiones de clientes, permitiendo la suscripción y publicación de mensajes de forma persistente.

Requisitos:

- Crear una clase MessageBroker que gestione la creación y eliminación de topics.
- Implementar una clase Session para representar sesiones de clientes con métodos de conexión y desconexión.
- Permitir que los clientes se suscriban a topics y reciban mensajes incluso después de desconectarse y reconectarse.
- Simular la publicación de mensajes y la entrega a clientes conectados y desconectados.

Objetivos:

- Implementar métodos para la gestión de sesiones de clientes.
- Diseñar un sistema de entrega de mensajes que almacene mensajes no entregados hasta que el cliente se reconecte.
- Simular la creación de topics, suscripciones y la publicación y recepción de mensajes con sesiones persistentes.

Ejercicio 9: Sistema de Notificaciones Distribuidas con MQ

Descripción:
Desarrollar un sistema de notificaciones distribuidas utilizando una simulación de Amazon MQ, permitiendo la publicación y suscripción de mensajes entre múltiples servicios distribuidos.

Requisitos:

- Crear una clase DistributedMessageBroker que gestione la publicación y suscripción de mensajes entre múltiples servicios.
- Implementar un mecanismo para la gestión de conexiones y sesiones de clientes.

- Permitir la definición de diferentes tipos de mensajes y su procesamiento por diferentes servicios.
- Simular la publicación y recepción de mensajes en un entorno distribuido, gestionando conexiones y sesiones de clientes.

Objetivos:

- Implementar métodos para la gestión de conexiones y sesiones de clientes.
- Diseñar un sistema de publicación y suscripción de mensajes entre múltiples servicios distribuidos.
- Simular la creación de conexiones, publicación y recepción de mensajes en un entorno distribuido.