UNIVERSIDADE FEDERAL DE ITAJUBÁ

CIÊNCIA DA COMPUTAÇÃO

TUTORIAL SOBRE AMAZON MQ

2020030477 - Flávio Augusto Aló Torres - flaviowet@gmail.com

2020007290 - Lucas Batista Pereira - lucas.bpereira1999@unifei.edu.br

2020002777 - Marcelo Robert Santos - marcelorobertsantos@unifei.edu.br

COM242 - SISTEMAS DISTRIBUÍDOS

Prof. Rafael Frinhani

INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO

UNIFEI - Itajubá

2023

UNIVERSIDADE FEDERAL DE ITAJUBÁ
INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
CIÊNCIA DA COMPUTAÇÃO
COM242 - SISTEMAS DISTRIBUÍDOS

Tutorial sobre Amazon MQ

1 Introdução

A troca de dados entre dispositivos é a base dos sistemas distribuídos, não só na troca de dados como também na troca de instruções e mensagens. Existem vários paradigmas de comunicação distribuída, como o ponta a ponta (P2P), o *publish / subscribe* (PS), a invocação remota de métodos (RMI), a chamada remota de procedimentos (RPC), o enfileiramento de mensagens (MQ), entre outros. Em qualquer uma dessas formas de comunicação é necessário robustez, velocidade e confiabilidade na transferência.

De forma mais específica, sistemas de enfileiramento de mensagens permitem a comunicação assíncrona entre diferentes componentes de um sistema distribuído. Essa comunicação é realizada por meio de uma fila de mensagens intermediária, que armazena as mensagens até que sejam entregues ao destinatário. Os sistemas envolvidos podem então se abstrair da linguagem da aplicação, necessitando apenas realizar a comunicação com um *broker* intermediário através de protocolos de mensagem suportados por ele. A comunicação é realizada ao publicar ou se inscrever em uma fila de mensagens, que as armazena até serem enviadas para todos os destinatários.

O presente trabalho busca ensinar sobre a ferramenta Amazon MQ, como ela funciona e como é possível instalar, utilizar e incorporar com outros serviços. Essa ferramenta, como o próprio nome diz, é baseada no enfileiramento de mensagens, e é integrada à plataforma *Amazon Web Services* (AWS), oferecendo grande solidez e integração com outros serviços.

1.1 Detalhes da Ferramenta

O Amazon MQ é um serviço de agente de mensagens gerenciado para Apache ActiveMQ e RabbitMQ que simplifica a configuração, a operação e o gerenciamento de agentes de mensagens na AWS. Com algumas etapas, o Amazon MQ pode provisionar seu agente de mensagens com suporte para atualizações de versões de software. Com o Amazon MQ é possível migrar aplicativos de agentes de mensagens existentes que dependem da compatibilidade com APIs como JMS ou protocolos como AMQP, AMQP, MQTT, OpenWire e STOMP. [Amazon MQ]

A princípio parece ser exatamente a mesma implementação do Apache ActiveMQ ou RabbitMQ, mas a vantagem está na conexão com a plataforma da AWS, onde é possível gerenciar credenciais, logs, métricas, regras de segurança e também incorporar com outros componentes da AWS, como a execução de funções com o AWS Lambda, armazenamento de dados com Simple Storage Service (S3) ou alguma opção de banco de dados também na plataforma, conexão com dispositivos de IoT com IoT Core, e vários outros serviços, podendo ser integrado com o AWS Elastic Compute Cloud (EC2), que é uma forma de computação em nuvem altamente customizável ou outras formas de computação em nuvem. [Fireship, Youtube]

Como observado, a plataforma da AWS implementa uma imensa quantidade de serviços, tanto proprietários quanto livres, dessa forma é possível que qualquer serviço procurado esteja presente na AWS, de banco de dados relacional à banco de dados em grafos, de containers Docker à computação quântica.

1.2 Realização da implementação

Sabendo então das possibilidades de integração e da vasta gama de serviços disponibilizados, a Amazon provê uma documentação simples para seus produtos, de forma que para serviços não tão requisitados existem poucos tutoriais de implementação específica.

No caso do Amazon MQ, a implementação se dá através da comunicação com o console provido (do Active MQ ou Rabbit MQ), então cabe ao desenvolvedor utilizar dos *endpoints* da Amazon e configurar as regras de entrada para realizar a comunicação com o sistema de mensagens como se fosse um serviço independente da Amazon, e para tanto existem bibliotecas gerais que implementam as ações necessárias à comunicação.

É apresentado então no presente trabalho um tutorial para implementar o serviço do Amazon MQ, indicando as etapas de customização para uma implementação qualquer do usuário e em seguida são apresentadas implementações específicas para exemplo e modelo de uso.

2 Criando e configurando o Amazon MQ

Para seguir o tutorial, é recomendado iniciar no site da AWS, `https://aws.amazon.com/pt/`. Outros recursos podem ser encontrados no site `https://aws.amazon.com/pt/amazon-mq/resources/`, incluindo um guia de desenvolvedor, fóruns e tutoriais, e também nos vídeos "Working with Amazon MQ" e "Amazon MQ Demo - AWS", dos canais AWS User Group India e AWS.

Antes de iniciar o processo, deve-se ter uma conta válida na AWS. Uma questão a ser levada em consideração é que por mais que o serviço Amazon MQ tenha opções gratuitas, o usuário, ao criar a conta, deve cadastrar o seu cartão de crédito a fim de ter uma conta válida para utilizar os serviços da AWS.

Após realizar o login, o usuário será direcionado ao AWS Management Console, o qual oferecerá uma visão geral da plataforma e quais serviços existem nela. Para selecionar o Amazon MQ basta o usuário pesquisar por este serviço no campo

de busca localizado no canto superior esquerdo no painel central.

Já dentro do painel da Amazon MQ, haverá uma página dedicada à tecnologia e documentação da Amazon MQ. Na parte a direita da tela, há um botão chamado “Conceitos básicos” no quadro de “Criar Agentes”.

Figura 1: Criação de agente.

Após clicar em “Conceitos Básicos”, selecione o Apache ActiveMQ (que foi o software de mensagem de código aberto utilizado) e em seguida aperte “Próximo”.

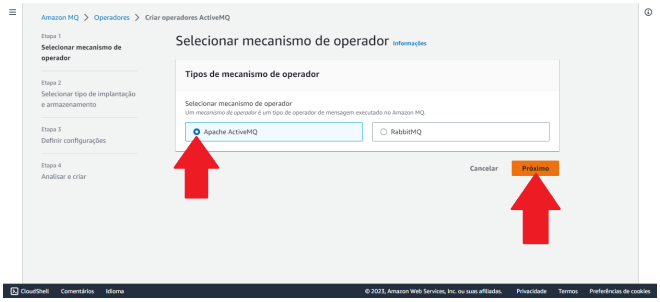

Figura 2: Criação de agente.

Em seguida, em “Modo de implantação”, selecione “Operador de instância única”. Essa configuração é o suficiente para fins didáticos. Feito isso, em “Tipo de armazenamento” selecione “Otimizado para durabilidade” (que é o padrão). Clique em “Próximo”.

Figura 3: Criação de agente.

Na próxima página, defina um nome (preferencialmente “MyBroker”) para o operador. Em “Tipo de instâncias de operador” selecione a instância “mq.t2.micro”. Mais abaixo na mesma página, coloque um nome de usuário (ex.: usuario_1) e uma senha (ex.: usuario_1_senha).

Após feitas essas configurações, clique em “Criar operador”. Essa ação leva em torno de 20 minutos para ocorrer. Depois disso seu agente já estará criado.

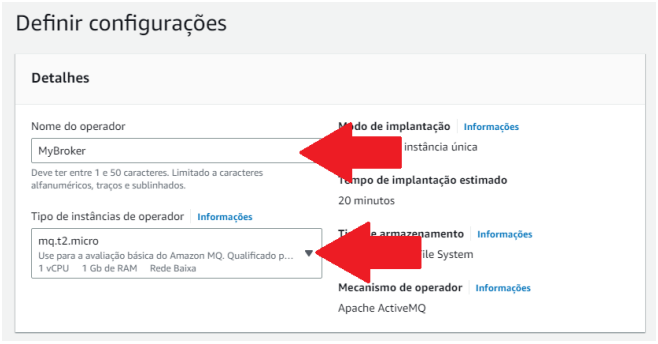

Figura 4: Criação de operador.

Figura 5: Criação de usuário.

Para poder ter acesso ao ActiveMQ Web Console ou mesmo realizar as conexões, será necessário configurar as regras de entrada. Para acessar essas configurações, selecione seu broker.

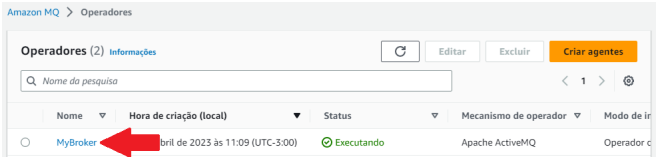

Figura 6: Selecionando o broker.

Após selecionado o broker, vá na seção “Segurança e rede” e clique em seu “Grupo de segurança”, conforme a figura abaixo. Uma nova guia se abrirá.

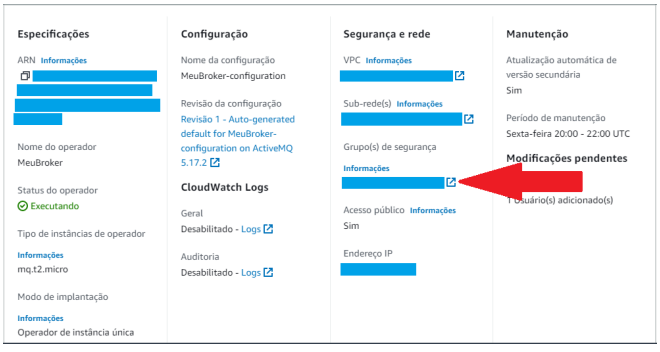

Figura 7: Selecionando o grupo de segurança.

Na nova guia, selecione seu grupo de segurança conforme a figura 8.

Mais abaixo na página do seu grupo de segurança, clique

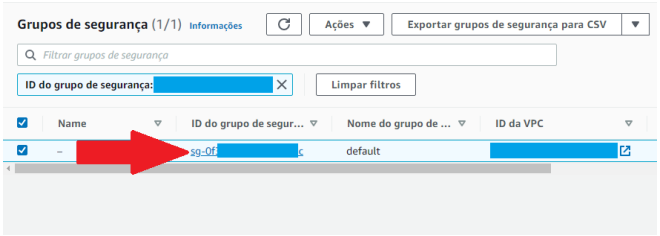

Figura 8: Selecionando o grupo de segurança.

em editar regras de entrada conforme a figura 9. Adicione 3 regras, inserindo os dados especificados na figura 10:

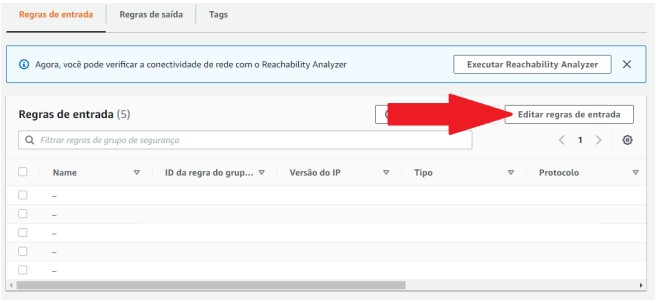

Figura 9: Selecionando o grupo de segurança.

Figura 10: Criando as regras de entrada.

O ActiveMQ deverá estar funcionando normalmente quando os passos anteriores forem concluídos. Com isso, o acesso ao console do ActiveMQ pode ser feito, através da endpoint. Para descobrir como conectar dispositivos ao broker, siga os passos seguintes.

3 Desenvolvimento

3.1 Problemática proposta

A aplicação didática desenvolvida no projeto foi a utilização do Amazon MQ para realizar a comunicação entre dois dispositivos: um simulando o cliente e o outro o servidor. O *backend* realiza três tarefas básicas:

1. Resposta a uma mensagem de texto
2. Alterações em um arquivo de texto
3. Cálculo de uma função

A máquina cliente solicita ao servidor que execute uma dada tarefa, que a executa retornando uma mensagem de resposta. No caso exemplo adotado, foram utilizados como clientes um ESP8266 e um aplicativo Flutter. Para servidor (*backend*), foi utilizado um código em Java em um notebook.

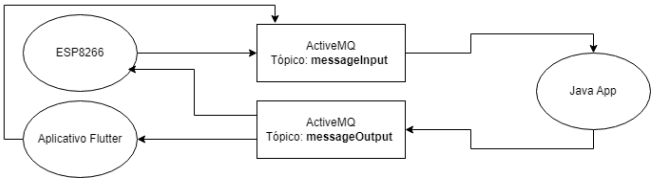

Figura 11: Diagrama de resposta a uma mensagem de texto.

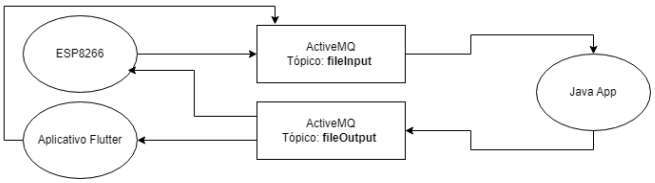

Figura 12: Diagrama de alteração em um arquivo de texto.

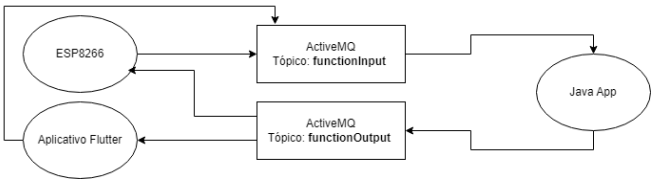

Figura 13: Diagrama do cálculo de uma função.

Os repositórios das implementações estão disponíveis no GitHub através do seguinte link: https://github.com/orgs/Amazon-MQ-Seminario-UNIFEI/repositories.

3.2 Desenvolvimento em Flutter

Flutter é um *framework* para desenvolvimento multiplataforma que utiliza a linguagem Dart. Ele é uma boa opção para desenvolvimento de aplicativos e para a realização de interface de usuário para programas *desktop* e *web*, portanto é interessante realizar a integração do Amazon MQ como uma forma de visualizar os dados e integrar aplicações em cenários reais.

Para o ambiente de testes, foi utilizada uma máquina com sistema operacional Windows 10, utilizando Flutter 3.7.5 na linguagem Dart versão 2.19.2 e o programa Visual Studio Code.

Dado que o Active MQ suporta vários protocolos de mensagens, é possível escolher qual protocolo se encaixa melhor na implementação utilizando as bibliotecas disponibilizadas para Flutter. O protocolo com maior implementação é claramente Http, mas para fins didáticos foi escolhido utilizar outro protocolo, a princípio foi escolhido o protocolo AMQP (*Advanced Message Queuing Protocol*), porém a biblioteca implementa o protocolo AMQP na versão 0.9.2 e o AWS apenas suporta AMQP no mínimo na versão 1.0.0, então foi utilizada a biblioteca *stomp_dart_client* que implementa o protocolo STOMP através de Web Sockets (Pub.dev).

Para utilizar tal protocolo no Amazon MQ, é necessário configurar a entrada de acesso de um protocolo TCP personalizado - web socket - na porta 61619, e o código em Flutter

utilizará a mesma porta e o protocolo web socket seguro (wss) na URL.

Na implementação é necessário então importar os arquivos da biblioteca:

```
import 'package:stomp_dart_client/stomp.dart';
import 'package:stomp_dart_client/stomp_config.dart';
import 'package:stomp_dart_client/stomp_frame.dart';
```

Em seguida, é definido o cliente para a comunicação. Nesse passo, é necessário utilizar as credenciais e o *endpoint* de acesso, que foram omitidos para segurança, mas basta criar um arquivo "credenciais.dart" e inserir as variáveis ali.

Código 1: Configuração do cliente para acesso em Flutter

```
StompClient stompClient = StompClient(
config: StompConfig(
  url: 'wss://$endpointAppCredential.amazonaws.com:61619',
  onConnect: onConnectCallback,
  onDisconnect: onDisonnectCallback,
  onDebugMessage: onDebugCallback,
  onWebSocketError: onErrorCallback,
  onStompError: onErrorCallback,
  onUnhandledFrame: onErrorCallback,
  onUnhandledMessage: onErrorCallback,
  onUnhandledReceipt: onErrorCallback,
  stompConnectHeaders: {
    'login': userAppCredential,
    'passcode': passcodeAppCredential
  },
),
);
```

Após a declaração do cliente, é necessário criar uma conexão com o *broker* para o envio e recebimento de mensagens através do método StompClient().activate(), que pode ser executado na inicialização da aplicação. O encerramento da aplicação irá cortar a conexão, mas é possível se desconectar a qualquer momento de forma segura utilizando o método StompClient().deactivate(). Observe que aqui o *StompClient()* deve ser o objeto declarado anteriormente, com as configurações necessárias.

Flutter é um *framework* nativamente assíncrono, dessa forma as funções de *callback* são feitas como uma *stream* de dados, rodando paralelamente à aplicação principal. Esses dados podem então ser retornados para a função principal reativamente.

Uma forma de visualizar a *stream* de dados é justamente ao se inscrever em um tópico ou fila, que é feita através do método StompClient().subscribe, e pode ser incorporado à uma função maior para o envio da mensagem, como no exemplo a seguir (as impressões dentro da condição *kDebugMode* são para realização de debug durante o desenvolvimento da aplicação):

Código 2: Inscrição em um tópico em Flutter

```
void subscribeToTopic(String topic, StreamController theStream) {
  if (kDebugMode) {
    print("Subscribing to $topic");
  }
  stompClient.subscribe(
    destination: '/topic/$topic',
    callback: (message) {
      String decodedMessage = utf8.decoder.convert(message.binaryBody!);
      if (kDebugMode) {
        print("Received Message in topic $topic:\n$decodedMessage");
      }
      theStream.add(decodedMessage);
    },
  );
}
```

A mensagem retornada é um objeto StompFrame contendo *headers* sobre a mensagem, o comando recebido e os dados em bytes que são então convertidos para String dentro do *callback* da função de inscrição.

Dentro dessa própria função de *callback* ou ao ouvir os dados com a *stream* pode-se então realizar o tratamento dos dados e execução de funções, podendo ser tratados a partir de sua origem, do comando recebido ou por convenções de comandos dentro da própria mensagem.

Já para enviar uma mensagem, basta estar conectado e selecionar o destino e mensagem a serem enviados, como no exemplo a seguir:

Código 3: Envio de uma mensagem em Flutter

```
void sendMessage(String message, String topic) {
  if (kDebugMode) {
    print("Sending to $topic");
  }
  stompClient.send(
    destination: '/topic/$topic',
    body: message,
  );
}
```

Foram utilizados tópicos nesse exemplo, porém é possível se inscrever e enviar mensagens para filas também, basta selecionar o destino como '/queue/$nome$' onde $nome é o nome da fila de destino.

Para a exibição dos dados, basta utilizar os dados da *stream* que foram passados durante a inscrição no tópico ou fila. A inscrição pode ser feita em um botão lançando por exemplo a função socket.subscribeToTopic("messageOutput", _mensagemStream);, onde *socket* é um objeto que contém o StompClient e as funções implementadas. Os dados podem ser exibidos da seguinte maneira:

Código 4: Exibição dos dados da stream de mensagens em Flutter

```
Row(
children: [
  Text(
    "Mensagem: ",
    style: h3style,
  ),
  StreamBuilder(
    stream: _mensagemStream.stream,
    builder: (context, snapshot) {
      return Text(snapshot.data != null
```

```
11              ? snapshot.data.toString()
12              : "", style: resultStyle);
13        },
14      ),
15    ],
16    ),
```

Já o envio das mensagens é trivial, basta configurar o texto que será enviado da forma que couber à aplicação e passá-lo para a função de envio com o tópico selecionado também da forma que mais couber à aplicação. No exemplo desenvolvido, o tópico é selecionado com um *DropdownButton*, sendo cada item *DropdownMenuItem* um valor para selecionar o tópico e o texto é escrito em um *TextField* onde o controlador desse campo contém a mensagem a ser enviada. O envio em si é feito através de um botão que reúne os valores e realiza a comunicação, como mostrado a seguir:

Código 5: Botão para envio da mensagem ao tópico em Flutter

```
1  ElevatedButton(
2  onPressed: () {
3    try {
4      socket.sendMessage(
           _textEditingController.text, topic)
           ;
5      ScaffoldMessenger.of(context).
           showSnackBar(SnackBar(
6          content: Text("Mensagem enviada a
               $topic")));
7    } catch (e) {
8      setState(() {
9        errResult = e.toString();
10     });
11   }
12 },
13 child: const Text("Enviar dado"),
14 ),
```

Os possíveis erros podem ser ser tratado nessa função e a confirmação de envio pode ser realizada conforme o exemplo, enviando uma mensagem para a tela do usuário.

A interface final (mostrada na Figura 14) não foca em um ótimo design e sim em seu funcionamento para as funções requisitadas, sendo composta pelos títulos de ajuda, os campos para seleção de tópico e envio de mensagem, o botão para realizar o envio, campos para ouvir as mensagens e um botão flutuante que realiza a inscrição em todos os tópicos previamente definidos.

3.3 Desenvolvimento do backend em Java

A utilização de um backend em Java para realizar a comunicação entre cliente-servidor utilizando o ActiveMQ é uma abordagem popular para aplicações que exigem uma alta escalabilidade e confiabilidade na troca de mensagens. O ActiveMQ é um sistema mensageiro que oferece uma variedade de recursos, incluindo troca de mensagens assíncrona, filas de mensagens, tópicos e muito mais. Além disso, o ActiveMQ pode ser facilmente integrado a outras tecnologias, como o Java, por meio do uso de APIs.

Ao desenvolver um backend em Java para utilizar o ActiveMQ, é possível criar um sistema altamente escalável, que permite que vários clientes se comuniquem de maneira assíncrona com o servidor. Isso é especialmente útil em aplicações

Figura 14: Interface final da aplicação em Flutter.

que exigem uma troca de informações em tempo real, como em sistemas de chat, sistemas de monitoramento, e-commerce, entre outros. O uso de filas e tópicos do ActiveMQ também permite que as mensagens sejam armazenadas e entregues de forma confiável, garantindo que as mensagens não sejam perdidas mesmo em caso de falhas de rede ou do servidor.

Para essa aplicação, foi utilizada a versão 11.0.18 do Java Development Kit (JDK) rodando em uma maquina com sistema operacional GNU-linux utilizando a distro Ubuntu versão 20.04.6 LTS. O plataforma de desenvolvimento escolhida foi a Eclipse, pois ela já acompanha ferramentas que serão necessárias para realizar a conexão do aplicativo Java com o broker do ActiveMQ, como o Mavem.

Após criar um novo projeto Java no eclipse, a primeira etapa é adicionar os pacotes necessários no Java class path. Para isso basta converter o Java project em Mavem project. O exemplo a seguir mostra quais dependências devem ser adicionadas no arquivo pom.xml do projeto Mavem.

Código 6: Adicionar dependencias Java

```
1  <dependencies>
2      <dependency>
3          <groupId>org.apache.activemq</
               groupId>
4          <artifactId>activemq-client</
               artifactId>
5          <version>5.15.8</version>
6      </dependency>
7      <dependency>
8          <groupId>org.apache.activemq</
               groupId>
9          <artifactId>activemq-pool</
               artifactId>
10         <version>5.15.8</version>
11     </dependency>
12 </dependencies>
```

3.3.1 Visão geral do sistema

A aplicação backend foi desenvolvida utilizando uma classe para cada tópico de dados de entrada, por isso, foram criadas três classes responsáveis por lidar com cada tópico. A classe “Consumer_ACK” é responsável por retornar um sinal enviado do cliente para o próprio cliente, seu funcionamento lembra a função ”ping” nos sistemas GNU-Linux. A classe ”Consumer_LOG” é responsável por armazenar os sinais enviados por um cliente em um arquivo de LOG’s no mesmo dispositivo onde a aplicação java está sendo executada. A classe ”Consumer_SOMA” é responsável por retornar a soma de todos os números contidos na mensagem enviada pelo cliente ao próprio cliente. A classe principal ”Server” é responsável por executar as três classes citadas acima simultaneamente.

Código 7: Servidor

```
1 public class Server
2 {
3     public static void main(String[] args)
          throws Exception {
4         Consumer_LOG.main(null);
5         Consumer_ACK.main(null);
6         Consumer_SOMA.main(null);
7     }
8 }
```

3.3.2 Configuração da conexão ao broker

Todas as três classes necessitam primeiramente configurar a conexão ao broker, configurar o callback das mensagens e se inscrever nos tópicos correspondentes das classes. Além disso, as configurações de cada classe necessita de parâmetros pré-definidos para realizar as conexões, um exemplo das configurações podem ser visualizadas a seguir.

Código 8: Exemplo de parâmetros necessários

```
1 String broker = "ssl://xxx.xxx.com:8883";
2 String clientId = "JavaConsumer_ACK";
3 String topic_in = "messageInput";
4 String topic_out = "messageOutput";
5 String username = "usuario";
6 String password = "Usuario12345@";
```

Tendo esses parâmetros definidos, será possível configurar o servidor MQTT, realizar a conexão ao broker e se inscrever no tópico correspondente da classe:

Código 9: Configuração do servidor MQTT

```
1 MqttClient mqttClient = new MqttClient(
      broker, clientId, new MemoryPersistence
      ());
2 MqttConnectOptions connOpts = new
      MqttConnectOptions();
3 connOpts.setUserName(username);
4 connOpts.setPassword(password.toCharArray()
      );
5 connOpts.setCleanSession(true);
6
7 mqttClient.subscribe(topic_in);
```

Um detalhe é que até aqui todas as configurações mostradas são válidas paras as três classes, com a única diferença sendo os nomes dos tópicos que são diferentes para cada classe. Porém a configuração do callback varia, pois ela irá depender do que o desenvolvedor necessita que seja feito com os dados recebidos do ActiveMQ. A seguinte implementação do callback corresponde a classe ”Consumer_ACK”, que serve de base para as callback’s das outras classes:

Código 10: Exemplo de callback

```
1  mqttClient.setCallback(new MqttCallback() {
2      @Override
3      public void connectionLost(Throwable
           throwable) {
4          System.out.println("Consumer_ACK:
               Conexao perdida com o broker: "
                + throwable.getMessage());
5      }
6      @Override
7      public void messageArrived(String s,
           MqttMessage mqttMessage) throws
           Exception {
8          System.out.println("Consumer_ACK:
               Mensagem recebida: " + new
               String(mqttMessage.getPayload()
               ));
9          mqttClient.publish(topic_out,
               message);
10     }
11     @Override
12     public void deliveryComplete(
           IMqttDeliveryToken
           iMqttDeliveryToken) {
13     }
14 }
```

3.3.3 Processamento e retorno de dados

A classe ”Consumer_LOG” é um exemplo de como armazenar uma mensagem de um cliente em um arquivo de texto. Para isso é necessário converter a mensagem MQTT em uma string e logo depois registrar essa mesma string em um arquivo de texto. A classe RegistrarMensagem será responsável por realizar o armazenamento dos logs.

Código 11: Conversão e inicio do armazanamento da mensagem

```
1 String arquivo_input = new String(
      mqttMessage.getPayload());
2 RegistrarMensagem.main(arquivo_input);
```

Na classe RegistrarMensagem ocorre a abertura de um arquivo de texto com o nome ”RegistroLogs” caso ele não exista.

Código 12: Criação do arquivo

```
1 String fileName = "local/do/RegistoLogs.txt
      ";
2 File file = new File(fileName);
3 try {
4     boolean criado = file.createNewFile();
5     if (criado)
6     {
7     //System.out.println("Debug: Arquivo
          criado com sucesso!");
8     }else {
9     //System.out.println("Debug: Arquivo ja
          existe!");
```

```
10        }
11    }catch(IOException e){
12        e.printStackTrace();
13    }
```

No arquivo é escrito a ordem numérica da mensagem, a mensagem em si e a data do dia em que a mensagem foi registrada.

Código 13: Inscrição de dados no arquivo

```
1  FileWriter fw = new FileWriter("local/do/
       RegistoLogs.txt", true);
2  BufferedWriter bw = new BufferedWriter(fw);
3
4  bw.write("LOG: " + ++retorno + " | Mensagem
       recebida: " + palavra + " | Data de
       hoje: " + dataFormatada + "\n");
5
6  bw.close();
7  fw.close();
```

A classe ”Consumer_SOMA” é responsável por retornar a soma do todos os números em uma mensagem. É importante ressaltar o detalhe de que todos as palavras/símbolos não numéricas serão ignoradas, por exemplo: Nas mensagens ”10 + 10”, ”10 lucas 10” e ”10 - 10”, o retorno será sempre ”20”, as palavras/símbolos ”+”, ”lucas” e ”-” serão ignorados, porque só interessa a aplicação os números, pois com esses números será realizado um somatório o qual será retornado ao cliente.

O trecho de código abaixo demonstra como a lógica foi implementada em Java.

Código 14: Apuração da string e a soma numérica

```
1  String message = new String(mqttMessage.
       getPayload());
2  String[] parts = message.split(" ");
3  int k = parts.length;
4  int soma = 0;
5  for (int i = 0; i < k; i++)
6  {
7      try {
8          Integer.parseInt(parts[i]);
9          soma += Integer.parseInt(parts[i]);
10     }catch (NumberFormatException e) {
11         try {
12         Double.parseDouble(parts[i]);
13         soma += Double.parseDouble(parts[i
               ]);
14     }catch (NumberFormatException e2)
15     {
16         //System.out.println("Debug: " +
               parts[i] + "nao eh um numero");
17     }
18   }
19 }
20 String soma_return = Integer.toString(soma)
       ;
```

Alguns pontos chave dessa operação:

1. A função split() é utilizada para subdividir uma string em uma array de strings, ”parts” herda em cada posição de seu vetor uma palavra de ”message”.

2. ”k” armazena a quantidade de tokens separados por um espaço, por exemplo: A mensagem ”10 + 10” tem três tokens.

3. Caso ocorra algum Exception durante o ”parse” das palavras, significa que a palavra pode ser número Double ou uma palavra/simbolo não numérico.

4. Se a palavra for considerada um número, esse número será somado a variável ”soma”.

3.4 Desenvolvimento no ESP8266

Para configurar o ESP8266, é necessário possuir um. Com ele em mãos, conecte-o ao computador. Com a Arduino IDE (ou qualquer outra IDE compatível), instale a biblioteca PubSubClient (MQTT) (disponível no gerenciador de bibliotecas do Arduino IDE). Além disso, efetue todas as configurações necessárias para poder enviar códigos para seu dispositivo (podem haver diferenças nas instalações devido aos fabricantes). Feitas essas coisas, é só alterar o código colocando as informações relacionadas ao Amazon MQ e ao Wi-Fi disponível. Envie o código para o microcontrolador. A cada 60 segundos uma mensagem de solicitação é enviada a partir do ESP8266 para os brokers, onde são encaminhadas para o servidor Java. A resposta do servidor Java pode ser vista no monitor serial da Arduino IDE.

4 Resultados e Conclusões

O trabalho apresentado teve como objetivo entender o funcionamento do Amazon MQ, oferecer um tutorial de configuração e exemplificar implementações de cliente e servidor através de aplicações para ESP, Flutter e em Java.

Espera-se que as informações presentes neste trabalho sejam úteis para a implementação do sistema de enfileiramento de mensagens Amazon MQ em diferentes sistemas, lembrando que a Amazon possui documentação sobre qualquer serviço disponibilizado, para futura integração deste com outros serviços.

É concluído então que a proposta apresentada foi alcançada, a comunicação distribuída implementada contém dois clientes e um servidor, realizando funções de forma remota através do sistema do enfileiramento de mensagens do Amazon MQ utilizando dois protocolos de mensagem e três linguagens de programação diferentes.

5 Materiais adicionais

Reforça-se o uso dos sites já mencionados: o site inicial do Amazon MQ, https://aws.amazon.com/pt/amazon-mq/, recursos que podem ser encontrados no site https://aws.amazon.com/pt/amazon-mq/resources/, incluindo um guia de desenvolvedor, fóruns e tutoriais, e também os vídeos “Working with Amazon MQ” e “Amazon MQ Demo - AWS”, dos canais AWS User Group India e AWS.