

Estudo de um middleware em Sistemas Distribuídos

Caso Apache ActiveMQ

Gabriel C. Fernandes¹

¹UNISINOS - Universidade do Vale do Rio dos Sinos
Caixa Postal 275 – 93.022-000 – São Leopoldo – RS – Brazil

`gabrielcastrofernandes@hotmail.com`

Abstract. *This is an article that describes the practical part involving a primary study on ActiveMQ middleware used in the Distributed Systems chair. The article seeks to show what were the steps taken in the process of creating applications (producers and consumers), services used and architecture implemented. And finally the result after the integration of all parties.*

Resumo. *Este é um artigo que descreve a parte prática envolvendo um estudo primário sobre o middleware ActiveMQ, usado na cadeira de Sistemas Distribuídos. O artigo busca mostrar quais foram os passos realizados no processo de criação das aplicações (produtores e consumidores), serviços utilizados e arquitetura implementada. E por fim o resultado após a integração de todas as partes.*

1. Introdução

Apache ActiveMQ é uma das soluções abertas que disponibilizam formas de aplicações se comunicarem em um formato assíncrono e de forma não estar acoplada a outras aplicações. ActiveMQ é um mediador de mensagens mais conhecido como message broker, em inglês. Usado para comunicação remota entre sistemas que usam a especificação do serviço de mensagem Java (Java Message Service). É escrito em Java, com API para muitas linguagens como C/C++, .NET, Perl, PHP, Python, Ruby e outras. Trabalhando diretamente entre a comunicação de sistemas para o fluxo de mensagens de forma simples e desatrelada. Também usa a JMS junto com o middleware orientado a mensagens (MOM – Message-oriented middleware) da Apache, sendo de alta disponibilidade, performance, escalabilidade e segurança para mensagens corporativas. O principal objetivo do ActiveMQ é disponibilizar uma base de padrões, aplicações que usem mensagens orientadas integradas com muitas linguagens e plataformas.[Snyder et al. 2011]

2. Conceitos técnicos

2.1. Linguagem escolhida

Para o caso de uso, foi utilizado o middleware juntamente com a API disponibilizada na linguagem .NET, conhecida como NMS (*.NET Message Service API*) que possibilita uma interface com o sistema de mensagens necessário. Além disso provém o uso de diversas características necessárias, como:

- Criação e disposição de conexões, sessões, produtores e consumidores
- Envio de mensagens para tópicos e filas

- Consumidor síncrono (bloqueio de recebimento, recebimento sem espera ou com tempo limite)
- Consumidor assíncrono (uso de `MessageListener` para ser despachado)
- Suporte para transação (enviando e reconhecendo várias mensagens em uma transação atômica)

2.2. Semântica

O ActiveMQ, assim como outros serviços de mensageria possibilitam o envio de mensagens em tópicos ou filas. Sendo o primeiro implementado com a semântica *Publish* e *Subscribe*, que visa o envio da mensagem através de uma aplicação para diversos “inscritos” ativos (0 para muitos), que receberam a mensagem. Diferente do segundo que possui uma semântica de *load balancer*. Uma mensagem é enviada diretamente para um consumidor quando ativo, caso contrário, a mensagem fica esperando o consumidor ser ativado. Também é possível enviar a mensagem para outros consumidores caso não seja reconhecida (*acknowledge*) antes de fechá-la.

2.3. Providers

Na API NMS, é disponibilizado um provedor (*provider*), cujo é um *Assembly* .NET que provem a implementação da API que possui conectividade com a JMS do middleware. Atualmente os provedores disponíveis para .NET são:

- AMQP
- ActiveMQ (OpenWire)
- STOMP
- MSMQ
- EMS
- WCF

E dentre eles, foi utilizado o OpenWire, por possuir um protocolo de conexão nativo, comunicação com ActiveMQ 5.x e disponibilizar todas as características necessárias para a implementação das aplicações. Também é um protocolo disponibilizado em diversas linguagens.

2.4. Serviços Amazon Web Services

Como parte do trabalho, e intuito de poder verificar o uso de aplicações distribuídas, foi utilizado os serviços AWS, possibilitando assim a distribuição da mensagem enviada para o *broker* e posteriormente ser consumida através de um consumidor em uma máquina criada em uma outra região do continente. Os serviços aqui aplicados foram:

- Amazon MQ – Serviço que possui já pronto o uso do ActiveMQ;
- Amazon EC2 – Criação de uma instância para executar a aplicação que irá consumir os dados enviados ao *broker*;
- Amazon SES (*Simple Email Service*) – Serviço usado para envio de e-mails;

3. Arquitetura criada

Como ideia e foco, foi pensado em distribuir duas aplicações de forma que fossem possíveis de se comunicar através de um endpoint e em locais diferentes. Sendo assim, foi utilizado primeiramente o Amazon MQ, serviço da AWS que disponibiliza a utilização do ActiveMQ através de máquinas distribuídas em diversas regiões do mundo e com configurações diferentes. Para o propósito, foi utilizada a instância mais simples e em uma região do Estados Unidos.

Ao configurar o serviço, um endpoint interno é criado, além dos endpoints para cada tipo de protocolo disponível (ex.: *OpenWire*). Esse caminho disponibilizado é usado mais adiante nas aplicações, de forma que seja possível o acesso delas para buscar ou escrever as mensagens.

Nessa etapa é importante salientar que a Amazon disponibiliza configurações padrões para acesso e segurança do *broker*, contudo, é possível permitir apenas portas e protocolos específicos, através da configuração dos Grupos de Segurança.

Uma vez configurada a parte do ActiveMQ e com o endpoint disponível, é possível criar a instância EC2 (*Elastic Compute Cloud*) e habilita-la para que a aplicação *Consumer* seja executada.

E por fim, o serviço de e-mail é disparado, enviando para os destinatários permitidos e selecionados para o recebimento.

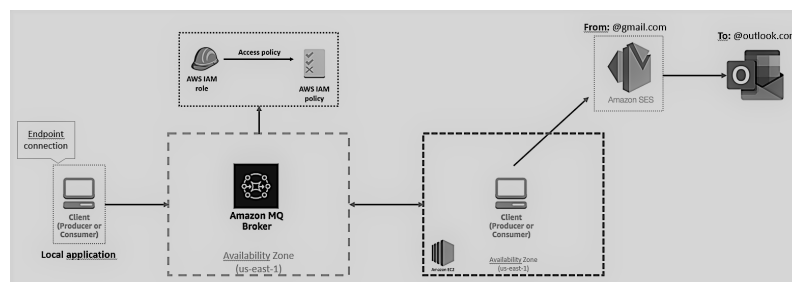


Figure 1. Arquitetura criada para o envio de mensagens de forma distribuída

3.1. Restrições de segurança e configurações

Como parte do sistema distribuído em instancias, em outra região do mundo, foi necessário criar políticas de acesso. Entre elas:

- Política de permissão para usuário que conecta os serviços Amazon MQ e Amazon SES.

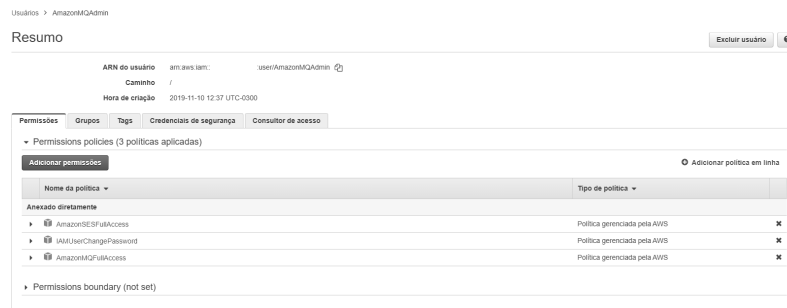


Figure 2. Políticas de permissão por usuário

- Regras de entrada (tipo de protocolo para tráfego, portas e IP's ou grupos CIDR)

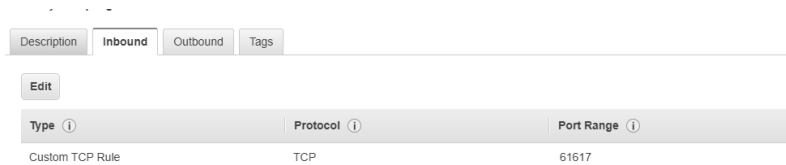


Figure 3. Regras de entrada para tráfego da rede



Figure 4. Regras de entrada para protocolo de RDP usado pela instância EC2

4. Implementação

Como descrito anteriormente, as aplicações realizadas, para comunicação entre as instâncias – do ActiveMQ e EC2 – são escritas em *.NET*.

4.1. Produtor

A aplicação Produtor foi desenvolvida de forma simples, com objetivo de comunicar-se com o serviço Amazon MQ, utilizando o *endpoint* disponibilizado na criação do *broker*. Após conectado, é selecionada a fila para o envio da mensagem e realizado o envio dela.

```

class MessageProduction
{
    public MessageProduction()
    {
        this.connectionFactory = new ConnectionFactory(Globals.WIRE_LEVEL_ENDPOINT);
    }
}

namespace Constants
{
    public static class Globals
    {
        public static readonly String WIRE_LEVEL_ENDPOINT = "ssl://b-11536-1.mq.us-east-1.amazonaws.com:61617";
        public static readonly String ACTIVE_MQ_USERNAME = "admin";
        public static readonly String ACTIVE_MQ_PASSWORD = "adm!00**&h";

        public static readonly String QUEUE_NAME = "EMAIL.ALERT";
    }
}

```

Figure 5. Construtor e constantes do projeto

```

public void SendMessageQueue()
{
    using (IConnection connection = this.connectionFactory.CreateConnection(
        Globals.ACTIVE_MQ_USERNAME, Globals.ACTIVE_MQ_PASSWORD))
    {
        connection.Start();

        using (ISession session = connection.CreateSession(AcknowledgementMode.AutoAcknowledge))
        using (IDestination producerDestination = session.GetQueue(Globals.QUEUE_NAME))
        using (IMessageProducer producer = session.CreateProducer(producerDestination))
        {
            producer.DeliveryMode = MsgDeliveryMode.NonPersistent;

            String msg = string.Format("Information sended to server MQ of Amazon!!!");

            ITextMessage message = session.CreateTextMessage(msg);

            producer.Send(message);

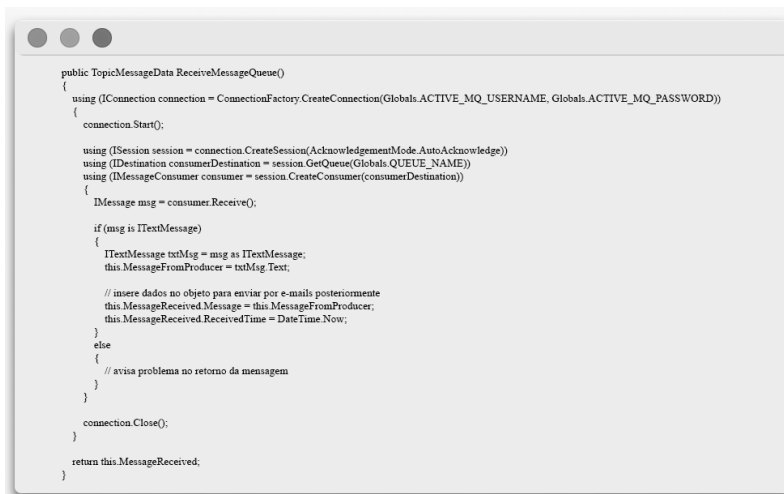
            Console.WriteLine("INFO: Successful in sending message to server");
        }
    }
}

```

Figure 6. Método para envio da mensagem para a fila

4.2. Consumidor

O consumidor por sua vez, também realiza o acesso via *endpoint* para visualizar e buscar as mensagens necessárias no *broker*.



```

public TopicMessageData ReceiveMessageQueue()
{
    using (IConnection connection = ConnectionFactory.CreateConnection(Globals.ACTIVE_MQ_USERNAME, Globals.ACTIVE_MQ_PASSWORD))
    {
        connection.Start();

        using (ISession session = connection.CreateSession(AcknowledgementMode.AutoAcknowledge))
        using (IDestination consumerDestination = session.GetQueue(Globals.QUEUE_NAME))
        using (IMessageConsumer consumer = session.CreateConsumer(consumerDestination))
        {
            Message msg = consumer.Receive();

            if (msg is IMessage)
            {
                IMessage txtMsg = msg as IMessage;
                this.MessageFromProducer = txtMsg.Text;

                // insere dados no objeto para enviar por e-mails posteriormente
                this.MessageReceived.Message = this.MessageFromProducer;
                this.MessageReceived.ReceivedTime = DateTime.Now;
            }
            else
            {
                // avisa problema no retorno da mensagem
            }
        }

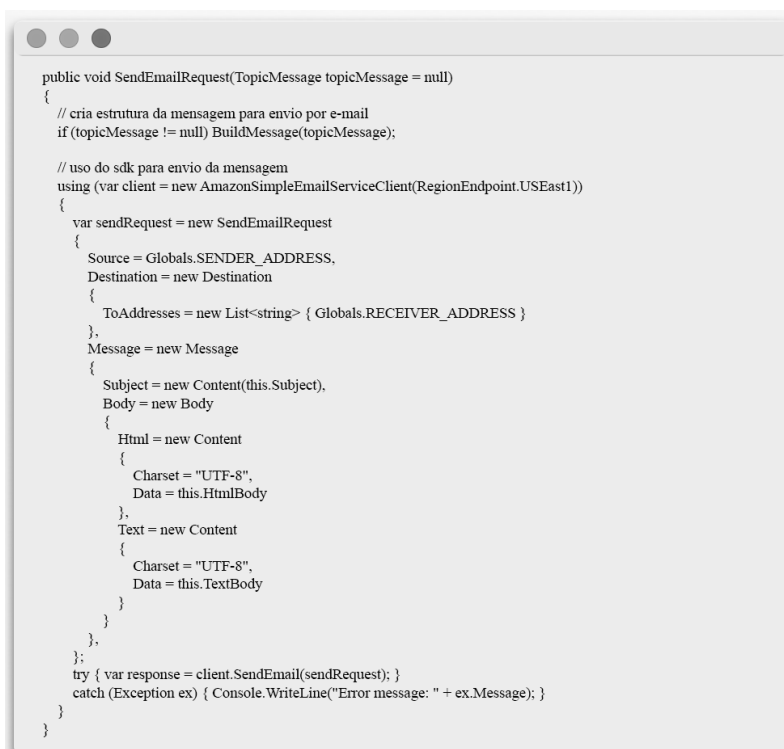
        connection.Close();
    }

    return this.MessageReceived;
}

```

Figure 7. Método para buscar a mensagem da fila

E por fim a mensagem é tratada de forma a ser enviada através do serviço da Amazon.



```

public void SendEmailRequest(TopicMessage topicMessage = null)
{
    // cria estrutura da mensagem para envio por e-mail
    if (topicMessage != null) BuildMessage(topicMessage);

    // uso do sdk para envio da mensagem
    using (var client = new AmazonSimpleEmailServiceClient(RegionEndpoint.USEast1))
    {
        var sendRequest = new SendEmailRequest
        {
            Source = Globals.SENDER_ADDRESS,
            Destination = new Destination
            {
                ToAddresses = new List<string> { Globals.RECEIVER_ADDRESS }
            },
            Message = new Message
            {
                Subject = new Content(this.Subject),
                Body = new Body
                {
                    Html = new Content
                    {
                        Charset = "UTF-8",
                        Data = this.HtmlBody
                    },
                    Text = new Content
                    {
                        Charset = "UTF-8",
                        Data = this.TextBody
                    }
                }
            }
        };

        try { var response = client.SendEmail(sendRequest); }
        catch (Exception ex) { Console.WriteLine("Error message: " + ex.Message); }
    }
}

```

Figure 8. Método para envio da mensagem usando o serviço de envio de e-mails

5. Resultado

Como resultado, ao executar a aplicação que envia as mensagens, o console mostra a quantidade de mensagens ainda enfileiradas (Figura 9), esperando serem consumidas por algum consumidor.

Nota-se que na figura 6, o modo de envio é “Não persistente” ou `textitNonPersistent`, o que significa que a mensagem é consumida pelos consumidores ativos no momento e após a finalização, a mensagem é destruída, não sendo armazenada para futuros consumidores. Para isso seria necessário mudar para o modo “Persistente” que armazena a mensagem em disco. Por fim, a mensagem é enviada através do serviço e-mails para o e-mail configurado (Figura 10).



Figure 9. Painel de mensagens ActiveMQ da AWS

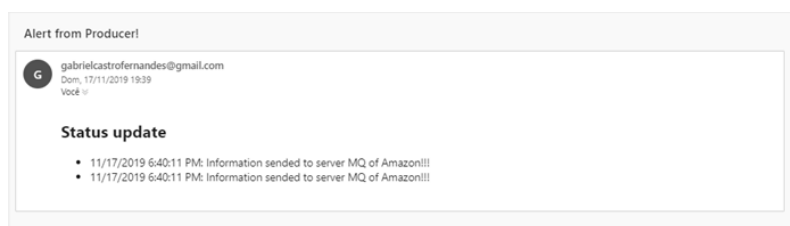


Figure 10. Envio da mensagem para um e-mail configurado

6. Conclusão

O middleware ActiveMQ mostra-se um grande concorrente frente a outros serviços da mesma categoria, como o RabbitMQ ou Kafka, se diferenciando em questões como ser Open Source que possibilita a melhoria contínua da ferramenta, e disponibilidade de uso comercial sem custo em alguns casos.

Quanto ao seu desempenho, é satisfatório para pequenas implementações, principalmente quando ligadas aos serviços em nuvem, como o da AWS, utilizado no decorrer do projeto. Isso visa e contribui muito para que o sistema possua uma latência menor na entrega da mensagem à fila e posteriormente consumida por alguma aplicação.

A utilização em uma arquitetura *Serverless* – que aqui não foi abordado – é de grande valia pois pode identificar momentos de maior entrada de mensagens em brokers, assim aumentando as propriedades de uma instancia, fazendo com que o sistema se comporte de forma estável, independente do volume de mensagens.

References

Snyder, B., Bosanac, D., and Davies, R. (2011). *ActiveMQ In Action*. Manning.