Um estudo sobre o middleware Apache ActiveMQ

1st Nicolly Ribeiro Luz
Faculdade de Computação (FACOM)
Universidade Federal de Uberlândia
Monte Carmelo, Brasil
nicolly.luz@ufu.br

***Abstract*—This paper explores the fundamental role of distributed systems in modern computing and highlights the critical importance of middleware in managing their inherent complexities. Distributed systems, defined as collections of computational programs utilizing resources across various distinct central points to achieve a shared objective, aim to eliminate bottlenecks and single points of failure. Middleware acts as an essential software layer, facilitating seamless communication and data exchange between diverse applications and components, thereby abstracting network complexities regardless of underlying hardware or software differences. This paper specifically focuses on Apache ActiveMQ, a prominent open-source message broker developed in Java. As a Message-Oriented Middleware (MoM), ActiveMQ is designed to enable robust and efficient message passing between applications in distributed environments. This article details ActiveMQ's architecture, functionalities, and its significant contribution to ensuring interoperability and resilience in distributed computing paradigms.**

***Index Terms*—Distributed Systems, Middleware, Message-Oriented Middleware, Apache ActiveMQ, Interoperability, Communication.**

I. Introdução

No cenário tecnológico em que vivemos atualmente, é inegável que a busca por sistemas mais robustos, verdadeiramente escaláveis e mais resilientes tem sido o grande fator motivacional para a migração e adoção em escala de sistemas distribuídos. Podemos pensar em Um sistema distribuído como uma coleção de programas de computador que utilizam recursos computacionais em vários pontos centrais de computação diferentes para atingir um objetivo comum e compartilhado [5]. Esses sistemas, que muitos também chamam de computação distribuída ou, em certas áreas, bancos de dados distribuídos, dependem crucialmente de uma comunicação e sincronização muito precisa entre seus diversos componentes na rede. A grande sacada por trás de toda essa arquitetura distribuída é justamente tirar os ”pesos mortos”, ou seja, aqueles gargalos que atrasam tudo e eliminar também os riscos de uma falha que derrube o sistema inteiro, o que, no fim das contas, nos dá uma performance e disponibilidade muito melhor. [5]

Agora, para que essa comunicação e coordenação funcionem de verdade em um ambiente tão plural, em que cada componente pode estar implementado de forma distinta dos demais, o middleware entra em cena com um papel absolutamente central. Pode-se descrever, genericamente, o middleware como uma ponte de software inteligente, em que ele se posiciona ali, entre as diferentes aplicações e componentes [2], agindo como um mediador essencial para executar sua missão, que é facilitar a troca de dados e a comunicação, fazendo com que as complexidades da rede (como diferenças de hardware, sistemas operacionais ou linguagens de programação) simplesmente desapareçam para o desenvolvedor. De forma a permitir que a gente se concentre nos aspectos que realmente importam, como a lógica de negócio do sistema. O middleware se transformou em uma ferramenta indispensável para gerenciar dados e permitir a comunicação eficiente entre diferentes componentes. É ele quem unifica tudo, integrando perfeitamente tecnologias, ferramentas e bancos de dados distintos em um sistema único e coeso.

E é justamente nesse universo do middleware que nosso artigo se baseia, o middleware que será abordado e discutido nesse trabalho é o Apache ActiveMQ. O Apache ActiveMQ é um popular corretor de mensagens de código aberto desenvolvido com base em Java. [1]. Ele funciona como um middleware orientado a mensagens (MoM), o que significa que foi projetado para enviar mensagens entre dois ou mais aplicativos, sua engenharia foi pensada para algo muito específico e vital: enviar e receber mensagens entre duas ou mais aplicações de forma robusta e eficiente. No decorrer dessa pesquisa, vamos desvendar a arquitetura por trás do Apache ActiveMQ, explorar suas funcionalidades chave e discutir como ele se tornou uma peça tão importante para garantir a interoperabilidade e a resiliência em ambientes de computação distribuída.

Fig. 1: Logo do ActiveMQ

II. Fundamentação

A. Entendendo o middleware

Uma vez que o assunto principal desse trabalho já foi introduzido, é preciso entender alguns conceitos essenciais para um melhor entndimento de como nosso middleware funciona.

Um middleware atua como uma camada intermediária entre componentes heterogêneos que foram desenvolvidos de forma independente uns dos outros para que eles interajam entre si a fim de alcançar um objetivo em commum. Conforme mostrado na figura a seguir, o middleware reside entre a camada de aplicação e a camada de plataforma que engloba o sistema operacional e os serviços de rede subjacentes [4].

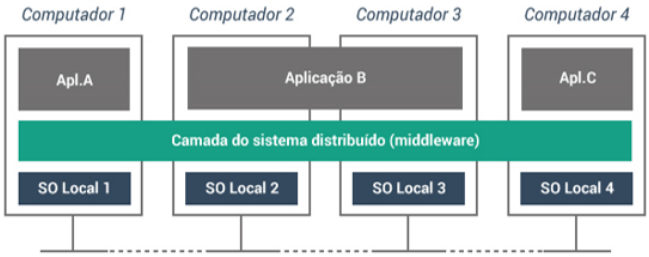

Fig. 2: Funcionamento de um Middleware

B. Entendendo o Middleware Orientado a mensagens

Como dito anteriormente, o middleware que será abordado aqui, Apache ActiveMQ, é um serviço orientado a mensagens (MoM). E por isso, vamos entender o que isso significa. Um middleware Orientado a mensagens permite que aplicativos distribuídos se comuniquem e troquem dados enviando e recebendo mensagens uns aos outros, de forma assíncrona, sem a necessidade de conhecimento mútuo ou sincronização direta. Os elementos básicos de um sistema MOM são clientes, mensagens e o provedor MOM (a implementação mais comum e robusta de um Provedor de Mensagens é o Message Broker, que será explicado mais adiante), que inclui uma API e ferramentas administrativas. Utilizando um sistema MOM, um cliente faz uma chamada de API para enviar uma mensagem a um destino gerenciado pelo provedor [4]. A chamada invoca os serviços do provedor para rotear e entregar a mensagem. Após o envio da mensagem, o cliente pode continuar a realizar outras tarefas, confiante de que o provedor a reterá até que um cliente receptor a recupere. É possível vizualizar essa ideia na figura abaixo:

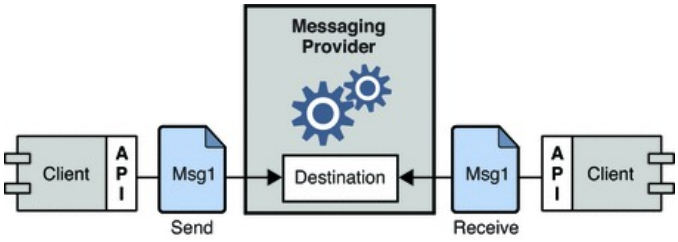

Fig. 3: Funcionamento de um Middleware Orientado a mensagens

Esses sistemas oferecem capacidade de armazenamento temporário para as mensagens, não exigindo que o emissor e o receptor estejam ativos durante a transmissão da mensagem. Diferentemente de sockets, suportam trocas de de mensagens que podem levar vários minutos em vez de alguns segundos ou milissegundos.

O MOM também oferece a abstração de filas de mensagens que pode ser acessada através da rede. É uma generalização do mecanismo de Mailbox presente em sistemas operacionais, que apresenta flexibilidade em relação a como programas podem depositar e retirar mensagens da fila. Dentro do contexto do MOM, a infraestrutura que provê e gerencia essas filas e mecanismos de armazenamento temporário é comumente chamada de Message Broker, o objetivo fundamental de um message broker é desacoplar (tornar independentes) os sistemas que precisam se comunicar. Em vez de uma aplicação enviar uma mensagem diretamente para outra, ela envia a mensagem para o message broker.

O broker, então, é um software, uma aplicação ou um servidor que atua como intermediário, responsável por garantir que essa mensagem seja entregue aos destinatários corretos . Ele atua como um ”posto de correio” centralizado, onde os produtores (aplicações que enviam mensagens) depositam suas mensagens e os consumidores (aplicações que recebem mensagens) as pegam quando necessário.

Além das estruturas de fila, produtores e consumidores, dentro do contexto dos Brokers também temos os tópicos [6]. As principais diferenças entre as filas e os tópicos são:

- **Filas:** As filas são espaços de armazenamento temporário para mensagens. Quando um produtor envia uma mensagem para uma fila, ela fica lá até que um consumidor a pegue. Cada mensagem em uma fila é geralmente consumida por apenas um consumidor.
- **Tópicos:** Já os tópicos, também são espaços de armazenamento, mas são usados para o modelo de comunicação publicar/assinar. Quando um produtor envia uma mensagem para um tópico, todos os consumidores que estão ”assinados” a esse tópico recebem uma cópia da mensagem. Isso é ideal para eventos ou notificações que precisam ser transmitidas para múltiplos sistemas.

O MoM se destaca por suas vantagens, principalmente pelos participantes não precisarem conhecer os endereços uns dos outros, basta se comunicarem com o Broker, além de que os participantes da comunicação não precisam se sincronizar para

trocar dados, bastam conectar-se para enviar ou receber mensagens, o que reduz o tempo ocioso durante a comunicação, e por fim, se um sistema falhar, o broker pode reter as mensagens até que o sistema volte a ficar online, evitando perda de dados e garantindo a continuidade das operações, se tornando uma peça fundamental na construção de sistemas distribuídos e na comunicação de seus componentes.

Ilustração de um Message Broker:

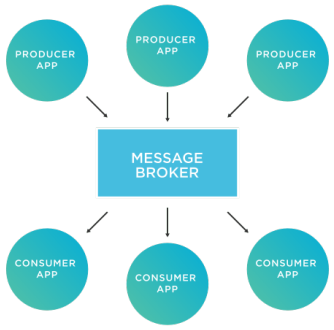

Fig. 4: Representação de um Message Broker

Apartir da explicação anterior, podemos dizer agora co clarez e entendimento que o Apache ActiveMQ é um message broker, e a seguir, vamos nos aprofundar especificamente em seu funcionamento e sua arquitetura.

III. Operação

Nesse tópico iremos abordar os recursos, a arquitetura, os serviços do Middleware Apache Active MQ, e como ele realmente funciona. Atualmente, o middleware Apache Active MQ possui duas versões, a versão *Classic*, que é a versão original do ActiveMQ, também conhecida como ActiveMQ 5 e a versão *Artemis* que é a versão mais recente e é considerada a próxima geração do ActiveMQ, com foco em melhor desempenho e escalabilidade em comparação com o *Classic*.

O ActiveMQ é um projeto de código aberto baseado em Java, desenvolvido pela Apache Software Foundation. É semelhante a outros sistemas de mensagens, como Apache Kafka, Amazon Easy Queue Service e RabbitMQ. O ActiveMQ utiliza a API Java Message Service (JMS), que descreve um padrão de software que pode ser usado para criar, enviar e receber mensagens. A versão Java Enterprise contém JMS, tornando-a disponível para uso por desenvolvedores Java na criação de aplicativos cliente que enviam, recebem e processam mensagens. Os clientes do ActiveMQ podem ser escritos em outras linguagens (como Node.js, Ruby e Python), mas o ActiveMQ é baseado em Java e, portanto, é mais adequado para aplicativos baseados em Java. [12].

A. Como o ActiveMQ funciona

O ActiveMQ envia mensagens entre aplicativos clientes(produtores), que criam mensagens e as enviam para entrega, e consumidores, que recebem e processam mensagens quando necessário. O Message broker do ActiveMQ encaminha cada mensagem por um dos dois tipos de destinos:

Uma fila, onde um único cliente deve ser entregue (em um domínio de comunicação chamado ponto a ponto) ou para um tópico/assunto a ser enviado a outros clientes que assinam o tópico, em um modelo de mensagens chamado de publicar/assinar ou como é mais conhecido na língua inglesa *Publish/Subscribe* (pub/sub).

O ActiveMQ oferece a flexibilidade de enviar mensagens por filas e tópicos usando um único broker. No sistema de mensagens ponto a ponto, o broker atua como um balanceador de carga, roteando cada mensagem da fila para um dos consumidores disponíveis em um padrão round-robin. Ao usar o sistema de mensagens pub/sub, o broker entrega cada mensagem a todos os consumidores inscritos no tópico em questão [6]. Essa interação pode ser melhor compreendida com a figura[4]:

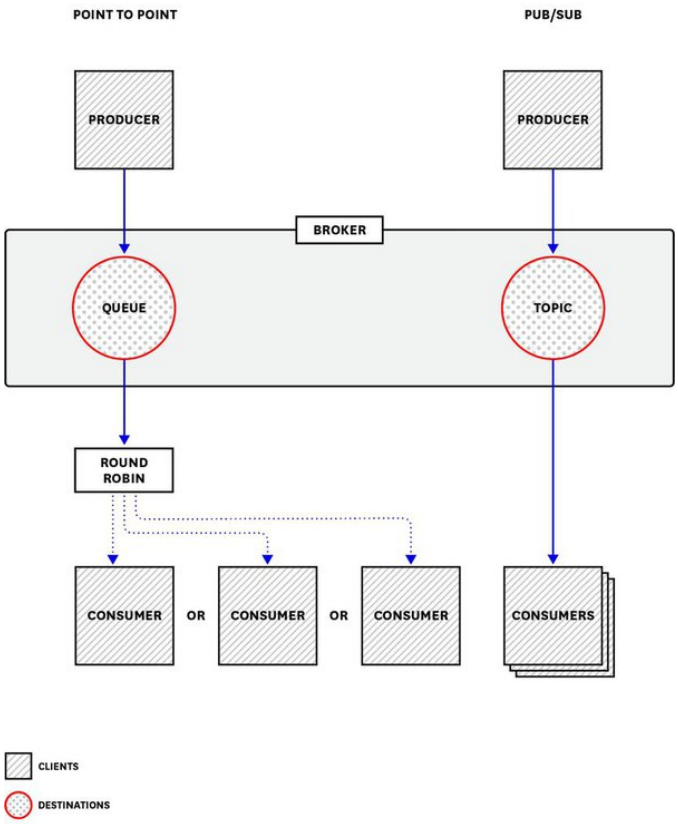

Fig. 5: Representação do funcionamento de filas e tópicos no Active MQ

B. Um passo a passo de como o ActiveMQ opera

1) Primeiro, uma aplicação (seja um produtor ou um consumidor) inicia uma conexão com o broker ActiveMQ usando algum dos protocolos de transporte que o middleware suporta (estes serão detalhados mais a frente).

2) Depois, o produtor cria uma mensagem e a envia para um destino específico no broker, que pode ser uma Fila ou um Tópico, mas esse destino vai estar especificado pelo produtor.
3) Quando o broker recebe a mensagem, ele a inspeciona e, se a persistência estiver configurada, ele a armazena-a no disco para garantir sua durabilidade, e em seguida, roteia a mensagem para o destino correto (fila ou tópico). Se for uma fila, ele enfileira a mensagem até que algum consumidor a solicite, e se for um tópico ele envia para todos os consumidores que são assinantes.
Obs: Se um assinante durável estiver offline, o broker persiste a mensagem até que ele volte.
4) Algum consumidor se conecta para solicitar as mensagens e após processa-la com sucesso, ele envia uma confirmação de volta ao broker para que a mensagem seja desenfileirada e não seja entregue a outro consumidor. Caso ele não envie a confirmação, o brooker pode reenviar a mensagem.

Com isso, podemos perceber que o funcionamento desse middleware é simples mas opera com muita robustez e eficiência.

C. API Java Message Service (JMS)

A API Java Message Service (JMS) é um padrão de mensagens que permite que componentes de aplicativos baseados na Java Platform Enterprise Edition (Java EE) criem, enviem, recebam e leiam mensagens. Ela permite comunicação distribuída, fracamente acoplada, confiável e assíncrona. E o Apache ActiveMQ é um dos provedores de mensageria mais populares que implementa a especificação JMS [12].

Cada mensagem enviada pelo ActiveMQ é baseada na especificação JMS e é composta de cabeçalhos, propriedades opcionais e um corpo. Os **cabeçalhos** de mensagens JMS contêm metadados sobre a mensagem, eles são definidos na especificação JMS e seus valores são definidos quando o produtor cria a mensagem ou quando o ActiveMQ a envia, também expressam as características da mensagem que determinam como o corretor e os clientes agem. **propriedades** funcionam de forma semelhante aos cabeçalhos e fornecem uma maneira de adicionar metadados opcionais a uma mensagem. Um **corpo** é a substância de uma publicação ActiveMQ. Texto ou dados binários podem ser o corpo de uma mensagem. O valor do cabeçalho JMSType da mensagem, que o produtor define especificamente quando a mensagem é gerada, especifica o que pode ser carregado no corpo da mensagem: um arquivo, um fluxo de bytes, um objeto Java, um fluxo primitivo Java, um conjunto de pares nome-valor ou uma string de texto. [6]

D. Arquitetura e recursos

Ao adentrar o todo o universo do Apache Active MQ, e com o auxílio de sua própria documentação, pudemos perceber que ele é formado por um conjunto de recursos e funcionalidades para suprir as demandas de promover comunicação entre sistemas e interoperabilidade. Dentre os recursos apresentados na documentação do ActiveMQ, adiante estão aqueles que julgamos mais relevantes no contexto de sistemas distribuídos.Todos os recursos citados a seguir foram retirados da documentação oficial do ActiveMQ. [7] A **conformidade com a JMS** é um bom ponto de partida para entendermos os recursos do ActiveMQ, pois ela oferece garantias importantes, como entrega de mensagens síncronas ou assíncronas, a depender do que se está esperando, entrega única e durabilidade de mensagens para assinantes.

Além desse, podemos citar também que o ActiveMQ oferece uma ampla gama de opções de **conectividade**, incluindo suporte a protocolos como HTTP/S, multicast IP, SSL, STOMP, TCP, UDP, XMPP e outros (que serão abordados posteriormente), o que resulta em uma maior flexibilidade ao lidar com sistemas muito diversificados.

O ActiveMQ também oferece **Persistência e segurança**, pois ele oferece vários tipos de persistência onde a escolha fica a critério do desenvolvedor, e quanto a segurança, no ActiveMQ ela pode serTotalmente personalizado para o tipo de autenticação e autorização mais adequado às suas necessidades.

Ainda, o ActiveMQ fornece **APIs de cliente para diversas linguagens** além de Java, incluindo C/C++, .NET, Perl, PHP, Python, Ruby e outras. Isso abre portas para oportunidades em que o ActiveMQ pode ser utilizado fora do universo Java. Muitas outras linguagens também têm acesso a todos os recursos e benefícios oferecidos pelo ActiveMQ por meio dessas diversas APIs de cliente.

Outro recurso bastante interessante é a possibilidade de **Clusterização dos Brokers**, uma vez que Muitos Brokers ActiveMQ podem trabalhar juntos como uma rede federada de Brokers para fins de escalabilidade. Isso é conhecido como uma rede de corretores e pode suportar diversas topologias.

E por fim, citamos que o ActiveMQ possui uma **Administração drasticamente simplificada**, já que ele foi projetado pensando nos desenvolvedores. Dessa forma, não requer um administrador dedicado, pois oferece recursos de administração poderosos e fáceis de usar. Esta é apenas uma amostra dos recursos oferecidos pelo ActiveMQ, essa ferramenta oferece inúmeras outras funcionalidades para diferentes tipos de usabilidade.

E. Protocolos

O Active MQ [7] oferece suporte para uma variedade de clientes e protocolos entre linguagens de programação como Java, C, C++, C#, Ruby, Perl, Python, PHP, e os protocolos de comunicação:

- **OpenWire:** Este é, por assim dizer, o idioma nativo do ActiveMQ. Desenvolvido para oferecer alto desempenho, o OpenWire é a escolha ideal para clientes escritos em linguagens como Java, C, C++ e C#, garantindo uma comunicação otimizada e eficiente entre a aplicação e o broker.
- **STOMP (Simple Text Oriented Messaging Protocol):** Se a simplicidade e a portabilidade são a prioridade, o suporte ao STOMP se destaca. Sendo um protocolo leve e baseado em texto, ele facilita a comunicação com o

ActiveMQ a partir de uma vasta gama de linguagens, como C, Ruby, Perl, Python, PHP e até mesmo ActionScript/Flash.

- **AMQP (Advanced Message Queuing Protocol):** Para cenários que exigem um protocolo mais robusto e padronizado no nível corporativo, o ActiveMQ oferece suporte completo ao AMQP v1.0. Este protocolo é amplamente adotado e provê funcionalidades avançadas para garantir a entrega confiável e a interoperabilidade em ecossistemas de mensagens complexos.
- **MQTT (Message Queuing Telemetry Transport):** Com o avanço da Internet das Coisas (IoT), a necessidade de comunicação eficiente com dispositivos de recursos limitados tornou-se crucial. O suporte ao MQTT v3.1 posiciona o ActiveMQ como uma solução poderosa para este ambiente, permitindo que sensores, atuadores e outros dispositivos IoT estabeleçam conexões leves e eficientes para enviar e receber dados, mesmo em redes com largura de banda restrita.

Em resumo, a capacidade do ActiveMQ de ”falar” fluentemente tantos protocolos e interagir com clientes de diversas linguagens é o que o eleva a um componente de integração verdadeiramente versátil, tornando a adoção e a interoperabilidade um processo muito mais fluido.

F. Acoplamento flexível

O ActiveMQ oferece os benefícios do acoplamento flexível para a arquitetura de aplicações. O acoplamento flexível é comumente introduzido em uma arquitetura para mitigar o acoplamento flexível clássico de Chamadas de Procedimento Remoto (RPC). Esse design com acoplamento flexível é considerado assíncrono, onde as chamadas de qualquer uma das aplicações não têm relação uma com a outra; não há requisitos de interdependência ou temporização [7]. Da mesma forma, as aplicações consumidoras não se preocupam com a origem das mensagens ou como foram enviadas para o ActiveMQ. Esse é um benefício especialmente poderoso em ambientes heterogêneos, permitindo que os clientes sejam escritos usando diferentes linguagens e até mesmo protocolos de conexão diferentes. Da mesma forma, as aplicações consumidoras não se preocupam com a origem das mensagens ou como foram enviadas para o ActiveMQ. Esse é um benefício especialmente poderoso em ambientes heterogêneos, permitindo que os clientes sejam escritos usando diferentes linguagens e até mesmo protocolos de conexão diferentes.

Exemplo de comunicação baseada em chamadas de procedimento remoto:

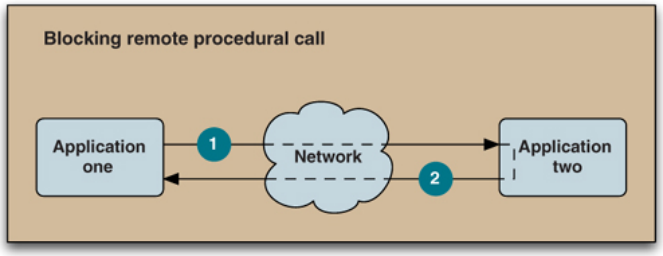

Fig. 6: Dois aplicativos fortemente acoplados usando chamadas de procedimento remoto para se comunicar

A aplicação 1 na figura acima é bloqueado até que a aplicação dois retorne o controle. Muitas arquiteturas de sistema usam RPC e são bem-sucedidas, mas há inúmeras desvantagens em um design tão fortemente acoplado: a mais notável é a maior quantidade de manutenção necessária, uma vez que mesmo pequenas mudanças repercutem em toda a arquitetura do sistema.

Exemplo de comunicação utilizando um middleware orientado a mensagens:

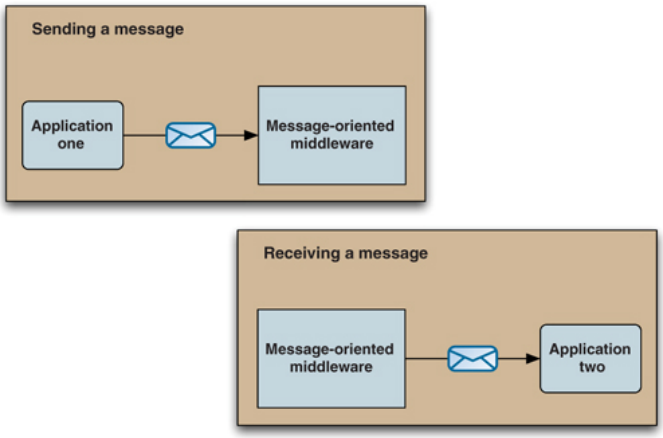

Fig. 7: Dois aplicativos fracamente acoplados usando mensagens JMS para se comunicar

A aplicação 1 na Figura acima envia uma mensagem ao MOM de forma unidirecional. Então, possivelmente algum tempo depois, a aplicação 2 recebe uma mensagem do MOM, de forma unidirecional. Nenhuma das aplicações tem conhecimento da existência da outra e não há tempo entre as duas aplicações. Esse estilo unidirecional de interação resulta em uma manutenção muito menor, pois as alterações em uma aplicação têm pouco ou nenhum efeito na outra. Por essas razões, aplicações fracamente acopladas oferecem grandes vantagens sobre arquiteturas fortemente acopladas ao considerar o design de aplicações distribuídas.

É aqui que o ActiveMQ entra em cena, oferecendo uma flexibilidade incrível na arquitetura de aplicações, permitindo que os conceitos relacionados ao acoplamento flexível se tornem realidade.

G. Memória e Armazenamento

Para armazenar mensagens aguardando envio aos usuários, o ActiveMQ utiliza memória. Cada mensagem ocupa parte da memória disponível até ser solicitada e enviada a um cliente, que então a processa e confirma o recebimento (a quantidade depende do tamanho da mensagem). O ActiveMQ libera a memória que estava sendo usada para aquela mensagem naquele momento. [1]

O ActiveMQ também grava mensagens no disco, em um armazenamento de mensagens (para onde as mensagens persistentes vão) ou em um armazenamento temporário (para onde as mensagens não persistentes vão quando o broker fica sem memória para armazená-las). [1]

Para a JVM na qual o ActiveMQ opera, o dispositivo host dedica parte de sua memória como memória heap. Por padrão, o Java é instruído pelo script de inicialização do ActiveMQ a construir um heap com tamanho máximo de 1 GB, mas essa quantidade pode ser alterada nos arquivos xml. [1]

IV. Testes

Nesse tópicos vamos demonstrar um guia passo a passo sobre como instalar e usar o Middleware. Também realizaremos testes para demonstrar como ele opera e como ajuda no desenvolvimento de um sistema distribuído.

Como já citado anteriormente, o Active MQ atualmente possui duas versões, a original, *Classic*, e a versão mais recente, a *Artemis*. Para nossos testes, como tenho intenção de apenas demonstrar como o middleware funciona na prática e levando em consideração que o *ActiveMQ Artemis* é uma reescrita com foco em performance bruta e flexibilidade de roteamento, decidi por testar a versão original que contém todos os aspectos que foram apresentados ao decorrer desse trabalho e é também uma abordagem mais simplificada para o entendimento.

A. *Guia de instalação do middleware*

Obs: As instruções a seguir foram executadas no sistema operacional Windows. [11]

Para que a instalação ocorra sem interconrrências, certifique-se de ter o Java Development Kit (JDK) ou Java Runtime Environment (JRE) instalado no seu computador.

Passo a passo para a instalação:

1) Faça o Download da última versão estável (latest stable release) do ActiveMQ de acordo com o seu sistema operacional, através do link: https://activemq.apache.org/components/classic/download/
2) Na sua pasta de Download, extraia o conteúdo do arquivo .zip que foi baixado de forma compactada.
3) Mova a pasta recém extraida para seu diretório *Program Files*.
4) Agora que o arquivo do ActiveMQ já está no diretório *Program Files*, percorra os seguintes arquivos: *apache-activemq-6.1.7/bin/win64/activemq*.
   Com isso, um terminal será aberto, copie o caminho para o localhost na porta 8161 em seu navegador (você está acessando a interface de administração web do Apache ActiveMQ).

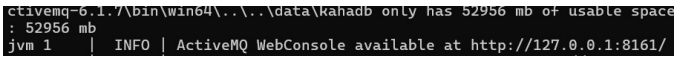

Fig. 8: Demonstração do caminho *localhost* no terminal

5) No navegador, um pop-up de login será aberto, digite *admin* para o campo de login e *admin* para o campo de senha, depois clique em Fazer login.

Fig. 9: Pop-up do login no navegador

6) Ao fazer login, você terá acesso ao console do ActiveMQ, onde poderá ver filas, tópicos, gerenciar mensagens, e monitorar o status do broker.

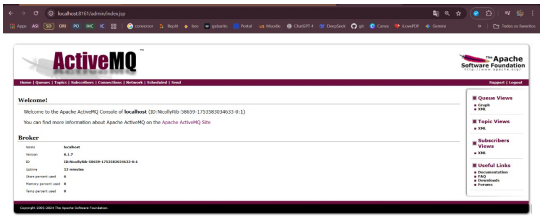

Fig. 10: Console do ActiveMQ

7) Pronto, agora você já pode usufruir das funcionalidades do Apache Active MQ.

Fig. 11: Funcionalidades do Apache ActiveMQ

Para desligar o broker, volte ao mesmo terminal onde você o iniciou (ou abra um novo, navegue até a pasta *bin*) e use o comando: *activemq stop*.

B. Testes realizados no middleware

Agora que a instalação já foi concluída, vamos iniciar a etapa de testes do middleware para ver como seu funcionamento ocorre na prática.

Para o teste, optei por implementar uma fila (Queue), por ela ser mais simples de entender e portanto, mais didática.

1) O primeiro passo para a implementação é garantir que o ActiveMQ está funcionando, ou seja, que você conseguiu acessar o console de administração.
2) Feito isso, é necessário criar um novo projeto Java na sua IDE de preferência, eu optei por utilizar o IntellijIDEA.
3) Ao criar um novo projeto java, defina O *Build System* como *Maven*
4) Preencha os detalhes do projeto:
   - **GroupId:** org.example
   - **ArtifactId:** activemq-queue-test (ou testeActiveMQ)
5) Clique em *Create* e aguarde o IntelliJ criar o projeto e indexar os arquivos.

Com o projeto criado:

6) Configure o *pom.xml*, (Que são são as dependências essenciais) No painel "Project" do IntelliJ (geralmente à esquerda), expanda seu projeto e localize o *pom.xml*, adicione as seguintes dependências:

```
<dependencies>
      <dependency>
         <groupId>org.apache.activemq</groupId>
         <artifactId>activemq-client</artifactId>
         <version>6.1.7</version>
      </dependency>

      <dependency>
         <groupId>jakarta.platform</groupId>
         <artifactId>jakarta.jakartaee-api</artifactId>
         <version>10.0.0</version>
         <scope>provided</scope>
      </dependency>

      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.13.2</version>
         <scope>test</scope>
      </dependency>
  </dependencies>
```

   - **`activemq-client:6.1.7`:**
     Esta é a dependência central para interagir com o broker Apache ActiveMQ Classic. Ela fornece as classes cliente necessárias para estabelecer conexão, enviar e receber mensagens de filas e tópicos. Sem ela, o código Java não conseguiria se comunicar com o ActiveMQ.

   - **`jakarta.jakartaee-api:10.0.0`:**
     Esta dependência traz as APIs da plataforma Jakarta EE, incluindo a API Jakarta Messaging (que é a evolução da Java Message Service - JMS). Ela é essencial porque as classes e interfaces usadas para a mensageria no seu código Java (como `ConnectionFactory`, `Session`, `Message`) são definidas por esta API.

   - **`junit:4.13.2`:**
     Esta dependência é do popular framework de testes unitários JUnit. Embora não seja diretamente necessária para a funcionalidade principal de mensageria, ela é fundamental para escrever e executar testes automatizados que garantam o correto funcionamento das suas classes de Produtor e Consumidor.

7) Agora que as dependências já foram adicionadas, crie os arquivos dos produtores e consumidores, optei por chamá-los de ***QueueSend e QueueReceive*** respectivamente.
8) Para facilitar a visualização e a facilidade de acompanhamento, optei por publicar todo o código-fonte, incluindo os exemplos detalhados e comentados passo a passo, na plataforma GitHub.

   Você pode acessá-lo diretamente pelo link:github.com/nini1504/SistemasDistribuidos /ArtigoActiveMQ/TestesRealizados

9) Se preferir, baixe todo o projeto e apenas o abra em sua IDE de preferência, mas cuidado, caso faça isso, verifique se as dependências e propriedades estão em conformidade com a sua máquina.
10) Com as Classes devidamente criadas e o Apache ActiveMQ em funcionamento, execute primeiro a Classe *QueueReceive*, pois é uma boa prática iniciar o consumidor primeiro para que ele esteja "escutando" quando as mensagens chegarem.
11) Agora, execute a Classe *QueueSend* e digite uma mensagem no console do *QueueSend*, depois aperte Enter.
12) Feito isso, vá até o console do ActiveMQ em seu navegador e verifique se a fila foi criada e também se os dados estão de acordo com o que foi executado.

C. Resultados observados

1) ***1° teste: Um produtor e 1 consumidor****:* Para esse teste, executei primeiro o consumidor (que ficou esperando até que alguma mensagem fosse enviada), e posteriormente executei o produtor e enviei a mensagem *Hello World*, o consumidor recebeu como o esperado e dessa forma, enviei mais duas mensagens.

```
--- Produtor Pronto para Enviar Mensagens ---
Enviando para a fila: 'minha.primeira.fila' em tcp://localhost:61616
Digite sua mensagem e pressione ENTER. Digite 'sair' para encerrar.
Sua mensagem (ou 'sair'): hello world
  [ENVIADO] Mensagem #1: 'hello world'
Sua mensagem (ou 'sair'): msg 2
  [ENVIADO] Mensagem #2: 'msg 2'
Sua mensagem (ou 'sair'): o serviço ja está pronto!
  [ENVIADO] Mensagem #3: 'o serviço ja está pronto!'
Sua mensagem (ou 'sair'):
```

Fig. 12: Mensagens enviadas pelo Produtor (QueueSend)

```
--- Consumidor Pronto para Receber Mensagens ---
Esperando mensagens na fila: 'minha.primeira.fila' em tcp://localhost:61616
Pressione Ctrl+C para encerrar o consumidor a qualquer momento.
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [RECEBIDO] Mensagem: 'hello world'
```

Fig. 13: Primeira mensagem recebida pelo consummidor (QueueReceive)

```
[INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
[RECEBIDO] Mensagem: 'msg 2'
[INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
```

Fig. 14: Segunda mensagem recebida pelo consummidor (QueueReceive)

```
[INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
[RECEBIDO] Mensagem: 'o serviço ja está pronto!'
[INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
```

Fig. 15: Terceira mensagem recebida pelo consummidor (QueueReceive)

Por fim, encerramos o produtor e pudemos observar no console do ActiveMQ as informações de tudo que foi feito.

```
Sua mensagem (ou 'sair'): sair
Produtor finalizou o envio. Total de mensagens enviadas: 3
Produtor encerrado e recursos liberados.

Process finished with exit code 0
```

Fig. 16: Encerramento do produtor

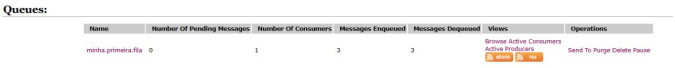

Fig. 17: Console do ActiveMQ após o encerramento do serviço

Nessa última figura podemos observar as seguintes informações:

- Havia apenas um Consumidor conectado
- 3 mensagens foram enfileiradas e desenfileiradas pois o consumidor recebeu todas.
- Nenhuma mensagem ficou pendente, pois todas foram recebidas pelo consumidor.

*2) **2°teste: Um produtor e dois consumidores**:* Como explicado anteriormente, em uma fila, cada mensagem é consumia por apenas um consumidor, e com esse teste pude perceber que as mensagens enviadas pelo produtor são recebidas de forma intercalada pelos consumidores, o consumidor 1 recebe a mensagem 1, o consumidor 2 recebe a mensagem 2, e assim por diante... Nas capturas de tela abaixo esse funcionamento é mostrado de forma bem clara.

```
--- Produtor Pronto para Enviar Mensagens ---
Enviando para a fila: 'minha.primeira.fila' em tcp://localhost:61616
Digite sua mensagem e pressione ENTER. Digite 'sair' para encerrar.
Sua mensagem (ou 'sair'): ola
  [ENVIADO] Mensagem #1: 'ola'
Sua mensagem (ou 'sair'): oii
  [ENVIADO] Mensagem #2: 'oii'
Sua mensagem (ou 'sair'): msg 123
  [ENVIADO] Mensagem #3: 'msg 123'
Sua mensagem (ou 'sair'): msg 322
  [ENVIADO] Mensagem #4: 'msg 322'
Sua mensagem (ou 'sair'): msg 555
  [ENVIADO] Mensagem #5: 'msg 555'
Sua mensagem (ou 'sair'): msg 221
  [ENVIADO] Mensagem #6: 'msg 221'
Sua mensagem (ou 'sair'):
```

Fig. 18: Mensagens enviadas pelo produtor

```
--- Consumidor Pronto para Receber Mensagens ---
Esperando mensagens na fila: 'minha.primeira.fila' em tcp://localhost:61616
Pressione Ctrl+C para encerrar o consumidor a qualquer momento.
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [RECEBIDO] Mensagem: 'oii'
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [RECEBIDO] Mensagem: 'msg 123'
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [RECEBIDO] Mensagem: 'msg 555'
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
```

Fig. 19: Mensagens recebidas pelo consumidor 1

```
--- Consumidor Pronto para Receber Mensagens ---
Esperando mensagens na fila: 'minha.primeira.fila' em tcp://localhost:61616
Pressione Ctrl+C para encerrar o consumidor a qualquer momento.
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [RECEBIDO] Mensagem: 'msg 322'
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
  [RECEBIDO] Mensagem: 'msg 221'
  [INFO] Nenhuma mensagem recebida em 10 segundos. Continuo esperando...
```

Fig. 20: Mensagens recebidas pelo consumidor 2

Obs: A primeira mensagem ”ola”, foi enviada para outro consumidor que foi fechado assim que recebeu a mensagem, por isso ela não aparece como recebida pelos demais.

Podemos observar no console do ActiveMQ que as informações são acumulativas, por isso na figura abaixo é possível vizualizar que houveram 2 consumidores conectados e um total de 9 mensagens enviadas entre todos os consumidores que já estiveram vinculados com a fila.

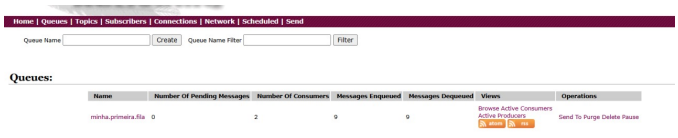

Fig. 21: Console do ActiveMQ após o encerramento do serviço

3) ***3°teste: Um produtor e um consumidor, em que o produtor é executado primeiro****:* Para essa simulação, eu quis vizualizar o que aconteceria caso o produtor enviasse a mensagem para a fila mas não houvesse ninguém para recebe-la. E o que aconteceu foi, ela ficou armazenada na fila (com status de pendente) até que um consumidor se conectasse para ”pegá-la”.

Fig. 22: Mensagem enviada pelo produtor

Veja que, no console do ActiveMQ, é possível ver que não há nenhum consumidor conectado e há uma mensagem pendente.

Fig. 23: Console do ActiveMQ com mensagem pendente

Quando a classe do consumidor é executada, ela automaticamente recebe a mensagem que estava na fila, e mensagem passa de pendente para desenfileirada, além de que o número de consumidores passa a ser 1.

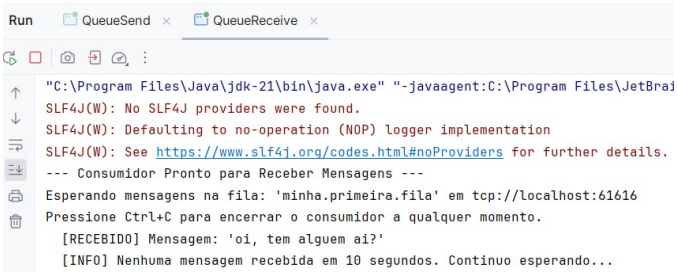

Fig. 24: Mensagem recebida pelo consumidor

Fig. 25: Console do ActiveMQ após o consumidor receber a mensagem

Com esses testes pudemos observar o funcionamento do middleware com filas na prática e em diversos cenários probabilísticos, podendo assim clarear todas as informações que foram apresentadas ao longo desse trabalho e ter um melhor entendimento de todos os conceitos descritos anteriormente.

V. Considerações finais

Ao longo deste trabalho, exploramos o complexo domínio dos sistemas distribuídos, reconhecendo algumas de suas limitações e a necessidade de alternativas eficientes para garantir a interação e uma coordenação precisa entre diferentes componentes de um sistema distribuído. Por isso, o middleware ganha força como uma parte importante da arquitetura, funcionando como uma ponte inteligente que simplifica as limitações da rede e as características específicas de cada tecnologia, permitindo que as aplicações possam focar em suas lógicas de negócios fundamentais.

Este estudo se aprofundou no middleware Apache ActiveMQ, um middleware orientado a mensagens (MoM) e baseado na linguagem Java que foi projetado para facilitar a troca robusta e eficiente de mensagens entre diferentes componentes no contexto de sistemas distribuídos. Ao longo desse artigo, explicamos como essa ferramenta funciona, seus elementos básicos, o sistema de mensageria e o mais importante, o papel crucial do Message Broker.

A estrutura do ActiveMQ se mostra bastante promissora e eficaz, apoiada por uma ampla gama de funcionalidades que o destacam como uma boa opção dentre os outros middlwares. Além disso, sua conectividade multiprotocolo garante a interoperabilidade com uma impressionante gama de linguagens e sistemas, e não se limitando apenas ao escossitema do Java. Sua capacidade de persistência e segurança plugáveis, juntamente com a possibilidade de clusterização de brokers, configuram o ActiveMQ como uma escolha inteligente para arquiteturas que exigem muita resiliência e flexibilidade.

As experiências práticas realizadas com o ActiveMQ Classic 6.1.7, descritas em nosso guia passo a passo, evidenciaram claramente o funcionamento das filas em um message broker. Pudemos observar de perto como o broker organiza a entrega de mensagens em um modelo MoM, e também vários casos de uso para o middleware, explorando diversos cenários que podem occorer na prática.

Em resumo, o Apache ActiveMQ representa a essência do acoplamento flexível em sistemas distribuidos, permitindo que as aplicações funcionem sem necessidade de conhecimento mútuo e sincronização direta, além de aumentar a resiliência a falhas e mitigar as limitações das chamas de procedimentos remoto. Ele não apenas facilita a forma de comunicação em rede, como também oferece uma base sólida para sistemas que requerem alta disponibilidade e escalabilidade.

É importante ressaltar que as referências que não foram citadas ao longo do trabalho, mas estão descritas na última seção serviram de base para o estudo e para a formulação da implementação do middleware como também para a execução dos testes.

REFERENCES

[1] Apache ActiveMQ. *Apache ActiveMQ Home Page*. Disponível em: https://activemq.apache.org/. Jul. 2025.

[2] AWS. "O que é Middleware?" Disponível em: https://aws.amazon.com/pt/what-is/middleware/. Jul. 2025.

[3] OpenLogic. "What is Apache ActiveMQ?" Disponível em: https://www.openlogic.com/blog/what-apache-activemq. Jul. 2025.

[4] Oracle. "Message-Oriented Middleware (MOM)." Disponível em: https://docs.oracle.com/cd/E19340-01/820-6424/aeraq/index.html. Jul. 2025.

[5] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2th ed., Pearson Education, 2007.

[6] Quora. "What is ActiveMQ and its use and function?" Disponível em: https://www.quora.com/What-is-ActiveMQ-and-its-use-and-function. Jul. 2025.

[7] B. Snyder, D. S. Posnett, and J. C. D. Santos, *ActiveMQ in Action*. Manning Publications, 2011. Disponível em: https://www.manning.com/books/activemq-in-action. Jul. 2025.

[8] DZone. "Getting Started with JMS: ActiveMQ Explained in Simple Way." Disponível em: https://dzone.com/articles/getting-started-with-jms-activemq-explained-in-simple-way. Jul. 2025.

[9] AWS. "Amazon MQ: Working with a Java Example." Disponível em: https://docs.aws.amazon.com/pt_br/amazon-mq/latest/developer-guide/amazon-mq-working-java-example.html. Jul. 2025.

[10] Mware. *Message Brokers*. Disponível em: https://www.vmware.com/topics/message-brokers. Jul. 2025.

[11] M. Ambrosio. "Mensageria Responsiva com JMS e ActiveMQ." *Medium*, [Data de publicação, ex: 15 de maio de 2017]. Disponível em: https://mmarcosab.medium.com/mensageria-responsa-com-jms-e-active-mq-f1cb6fd6dc35. Jul. 2025.

[12] Oracle. *Java Message Service (JMS)*. Disponível em: https://www.oracle.com/java/technologies/java-message-service.html. Jul. 2025.