Middlewares Orientados a Mensagens

Victor Emanuel Perticarrari Osório

[1]PPGCA — Programa de Pós-graduação em Computação Aplicada.
Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba — PR — Brasil

osorio@alunos.utfpr.edu.br

Abstract. *Message-oriented middlewares are a great solution for scaling real-time systems through asynchronous message processing. In this article, we analyse what is the challenge that these middlewares try to answer, when describing the limitations of client-server architecture it addresses. Those systems are then characterized by observing their main characteristics. Lastly, four common problems are described in systems that use this type of middleware.*

Resumo. *Middlewares orientados a mensagens são uma ótima solução para escalar sistemas em tempo real através do processamento assíncrono de mensagens. Nesse artigo analisa-se qual é o desafio que esses middlewares tentam responder, ao se descrever as limitações da arquitetura cliente-servidor que eles endereçam. Esses sistemas são caracterizados observando suas principais características. Finalmente, são apresentados quatro problemas comuns em sistemas que utilizam esse tipo de middleware.*

1. Introdução

A popularidade de *Middlewares* Orientados a Mensagens (MOM) tem crescido pela necessidade de integração entre diversos sistemas de forma assíncrona e confiável (Yongguo et al. 2019). Embora esse conceito exista desde o final da década de 1990, recentemente MOMs passaram a ter um papel central na arquitetura de sistemas complexos e distribuídos.

Além dos argumentos técnicos, uma possível explicação para essa popularização é o crescimento da base de usuários dos sistemas de internet. Nas últimas duas décadas, sistemas de internet se tornaram onipresente na vida moderna, fazendo com que aplicações que antes eram voltadas a uma pequena base de usuários tenham que lidar com uma enorme base e, em muitos casos, um crescimento massivo e inesperado de acessos simultâneos. Atualmente, aproximadamente 60% da população mundial é usuária da internet, significando um crescimento de 100% comparado com 2010, quando apenas 28,9% da população estava conectada (Ritchie et al. 2023). Serviços essenciais que antes eram exclusivamente físicos se tornam digitais, forçando as empresas a considerar na arquitetura dos sistemas requisitos de negócios como disponibilidade e responsividade (Treynor et al. 2017).

Novos requisitos e o aumento de demanda fazem surgir novas arquiteturas e plataformas para lidar com uma base de usuários crescente e o aumento da frequência das atualizações do software em produção. Essa mudança requer a reengenharia dos processos que

anteriormente eram síncronos, tornando-os assíncronos através do enfileiramento de mensagens para evitar gargalos de processamento (Fu et al. 2021). Nesse contexto, os MOMs surgem como uma evolução ao paradigma cliente-servidor, proporcionando maior disponibilidade, resiliência e uma redução do acoplamento entre sistemas (Goel et al. 2003).

O objetivo básico desse estudo é detalhar o estado da arte dos *Middlewares* Orientados a Mensagens. O documento está organizado em cinco seções. A Seção 2 detalha quais são as limitações da arquitetura Cliente-Servidor que motivaram a criação de um novo modelo de comunicação. A Seção 3 traz uma caracterização de um *Middleware* Orientado a Mensagens genérico. E a Seção 4 descreve os principais desafios na implementação de sistemas usando MOM e quais as possíveis soluções encontradas.

2. Limitações da Arquitetura Cliente-Servidor

Uma arquitetura cliente-servidor pode ser compreendida como um modelo computacional baseado em rede onde uma aplicação é dividida em dois processos distintos executados em computadores diferentes. O processo cliente é responsável por elaborar e enviar requisições ao processo servidor, aguardando respostas síncronas. O servidor, por sua vez, processa essas requisições e envia as respostas ao processo cliente (Hura 1995). O protocolo cliente-servidor mais comum para aplicações web é o *Hypertext Transfer Protocol* (*HTTP*) (Berners-Lee et al. 1996), utilizado tanto por navegadores quanto por aplicativos móveis. Ele é empregado em serviços *Representational State Transfer* (*REST*), *Remote Procedure Call* (*RPC*), *Simple Object Access Protocol* (*SOAP*) (Zur Muehlen et al. 2005) e GraphQL (Quiña-Mera et al. 2023). Embora seja amplamente utilizado, o HTTP não é o único protocolo cliente-servidor empregado em aplicações web. Em um cenário típico, uma aplicação se conecta a uma base de dados SQL ou NoSQL por meio de bibliotecas como a *Java Database Connection* (*JDBC*) (Lawrence et al. 2017).

Abaixo estão listadas as principais limitações dos protocolos cliente-servidor discutidos em sequência.

1. Disponibilidade e dependência entre serviços;
2. Acoplamento entre serviços;
3. Sincronicidade entre Requisição e Respostas;
4. Coordenação de interações de longa duração entre partes distribuídas.

2.1. Disponibilidade e dependência entre serviços

A primeira desvantagem de um protocolo cliente-servidor reside na incapacidade do cliente de executar suas tarefas caso o servidor ao qual deseja se conectar esteja indisponível ou não responda. Nenhum servidor poderá atingir uma disponibilidade absoluta de 100% devido à presença inevitável de problemas que podem levar a falhas. Portanto, é importante que cada serviço estabeleça metas de disponibilidade realistas, como, por exemplo, $99,99\%$. Para entender melhor o que gera uma indisponibilidade, deve-se olhar para as principais causas de interrupções no funcionamento do serviço: problemas com o próprio serviço ou problemas com dependências criticas do mesmo (Treynor et al. 2017).

Uma dependência é classificada como crítica quando seu mau funcionamento causa um mau funcionamento no serviço que depende dela. Existem várias estratégias para que uma dependência cliente-servidor não seja considerada crítica, tais

como retransmissões (Edstrom and Tilevich 2012), balanceamento de carga e *cache* (Lawrence et al. 2017), algumas dessas estratégias podem adicionar atrasos na comunicação.

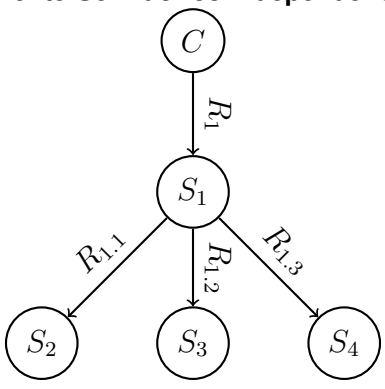
A dependência de um sistema é qualquer outro componente de software executado em um processo separado e que se comunica através da rede em um modelo cliente-servidor. Uma dependência pode ser considerada dependência de primeira ordem, quando o serviço depende diretamente dela, ou dependência de segunda ordem, quando é uma dependência direta de uma dependência de primeira ordem e assim por diante (Treynor et al. 2017).

A disponibilidade de um serviço pode ser calculada em função da frequência de falhas e do tempo médio até o reparo. A partir dessa observação e considerando $MTTF$ como o tempo médio que leva para uma falha acontecer, $MTTR$ o tempo médio que leva para o sistema voltar a funcionar após falha, a disponibilidade A de um serviço é definida pela Equação 1 (Treynor et al. 2017).

$$A = \frac{MTTF}{MTTF + MTTR} \tag{1}$$

Quando um serviço depende de outros serviços, a sua disponibilidade é limitada pela disponibilidade das suas dependências críticas (Treynor et al. 2017). Para exemplificar, considera-se um cliente C que acessa um servidor S_1 que, por sua vez, depende de outros três servidores S_2, S_3 e S_4 para atender uma requisição R_1. Ao responder à requisição R_1, o servidor S_1 fará as requisições $R_{1.1}$, $R_{1.2}$ e $R_{1.3}$ para os servidores S_2, S_3 e S_4, respectivamente, como representado na Figura 1. A disponibilidade de S_1 será limitada pelas disponibilidades de S_2, S_3 e S_4.

Figura 1. Arquitetura Cliente-Servidor com dependências de primeira ordem.



Fonte: Adaptado de Treynor et al. 2017.

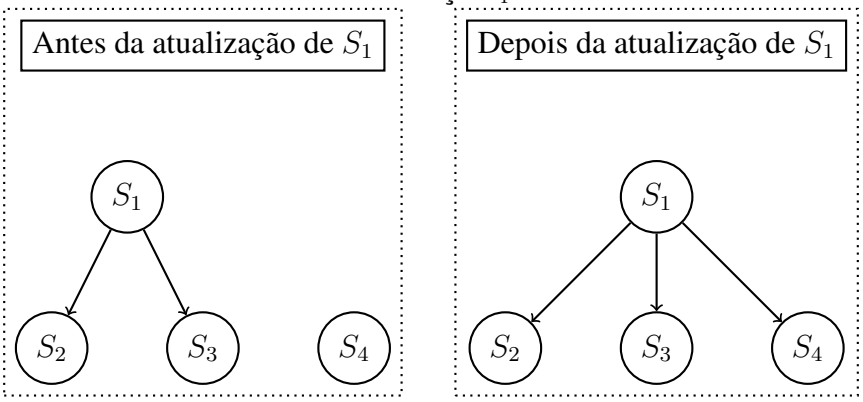
2.2. Acoplamento entre serviços

Um outra limitação da arquitetura cliente-servidor é o alto acoplamento entre os serviços. Um serviço só pode enviar requisições para outros serviços se conhecer a existência desse serviço, o que significa que deve conhecer o endereço do Protocolo de Internet

(*IP — Internet Protocol*), a porta para conexão e a Interface de Programação de Aplicação (*API — Application Programming Interface*) do serviço. A descoberta do IP e porta de conexão podem ser resolvidos usando estratégias de descobertas de serviços (Wang 2019; Richardson 2018), porém qualquer mudança de API deve acarretar uma atualização de todos os serviços que dependem dessa API.

Supondo que exista um serviço S_1 que já se comunica com S_2 e S_3, ao ser criado um novo serviço S_4, S_1, S_2 e S_3 só poderão se comunicar com S_4 se conhecerem a sua API. Se não houver nenhum mecanismo para a descoberta de S_4, ou algum mecanismo para o mesmo poder se registrar em S_1, será necessária uma atualização em S_1 para que ele saiba a existência de S_4, conforme pode ser visto na Figura 2.

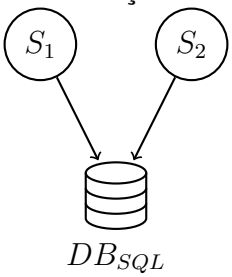
Figura 2. Adição de novas dependências. O serviço S_1 precisa ser atualizado para conhecer a existência do novo serviço S_4.



Fonte: Autoria própria

Esse acoplamento pode acontecer indiretamente quando há o compartilhamento de bases de dados relacionais, também conhecido como o padrão Base de Dados Compartilhada (*Shared database*) (Richardson 2018). Supondo que os serviços S_1 e S_2 utilizem a mesma base de dados DB_{SQL}, conforme ilustra a Figura 3, uma atualização em S_1 que demande atualização do esquema de uma tabela T_I, onde T_I também é usada por S_2, irá demandar atualização do serviço S_2.

Figura 3. Dependência indireta de serviços via base de dados compartilhada.



Fonte: Autoria própria

A necessidade de atualizações constantes pode ser uma das razões de indisponibilidade de serviços, pois se S_1 for atualizado juntamente com o esquema da tabela T_I em

DB_{SQL} sem a atualização de S_2, tanto S_1 quanto S_2 podem apresentar comportamentos indesejados ou mesmo indisponibilidade.

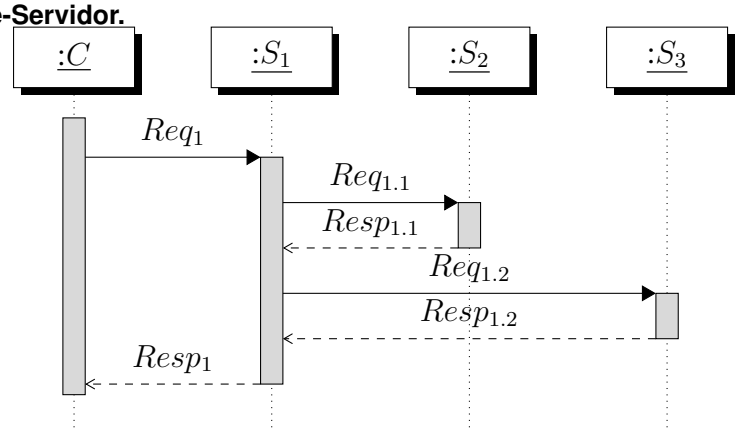
2.3. Sincronicidade entre Requisição e Respostas

Pela definição de uma arquitetura cliente-servidor, todo cliente deve elaborar uma mensagem que deve ser respondida pelo servidor de forma síncrona. A obrigação de responder sincronicamente não implica que todos os fluxos de um software cliente-servidor devam ser síncronos, apenas implica que toda requisição tenha uma resposta (Yildiz et al. 2008).

A limitação imposta pela sincronicidade entre requisição e resposta acontece porque processos complexos vão requerer várias chamadas de serviços, implicando em alto acoplamento entre serviços e alta latência pelo encadeamento das requisições.

Supondo que exista um serviço S_1 que ao receber a requisição Req_1 deve iniciar dois processos nos serviços S_2 e S_3 através das requisições $Req_{1.1}$ e $Req_{1.2}$, respectivamente. Além das limitações já vistas nas Seções 2.1 e 2.2, haverá um impacto no tempo para geração da resposta $Resp_1$. Para que S_1 envie $Resp_1$ é preciso garantir que todos os serviços dependentes respondam corretamente, como pode ser visto na Figura 4.

Figura 4. Diagrama de sequência de uma comunicação síncrona usando arquitetura Cliente-Servidor.



Fonte: Autoria própria

2.4. Coordenação de interações de longa duração entre partes distribuídas

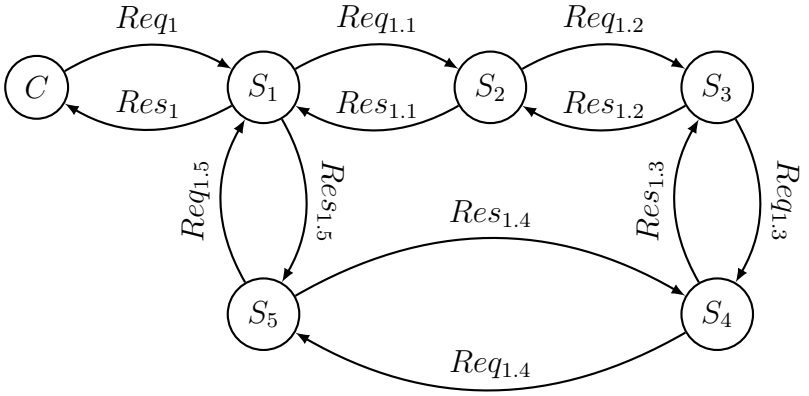
Outra limitação da arquitetura Cliente-Servidor é a ausência de um padrão para se construir soluções de gestão para longas interações (Zur Muehlen et al. 2005). Longas interações podem ser implementadas mediante duas estratégias: coreografia e orquestração (Terpák et al. 2016; Rudrabhatla 2018).

Quando existe a necessidade de se garantir a consistência de longas interações, esses processos são chamados de Transações de Longas Duração (*LLT — Long Lived Transactions*). LLT são processos não atômicos cuja execução pode durar um longo intervalo de tempo, sendo compostos por várias pequenas transações atômicas, garantindo, ao final, a consistência dos dados envolvidos (Garcia-Molina and Salem 1987). Saga é um padrão de implementação de LLT em que para cada transação atômica deve existir uma transação de compensação a ser executada caso a LLT seja abortada (Rudrabhatla 2018).

Processos com interações de longa duração são coreografados quando não há uma coordenação central, sendo a sequência de processos a serem executados definida pela troca de mensagens entre os serviços (Malyuga et al. 2020). Não há nenhum padrão para coordenação de coreografia, apesar de existirem padrões para definição de interface e API como a Linguagem de Descrição de Serviços Web (*WSDL — Web Services Description Language*) para SOAP e OpenAPI para REST. WSDL é a linguagem que possibilita expor um serviço SOAP para outros serviços poderem acessar suas funcionalidades através da geração de código para fácil integração (Christensen et al. 2001). Semelhantemente, o OpenAPI fornece as mesmas funcionalidades para serviços REST (OpenAPI 2021).

Na Figura 5 é mostrada como se dá a interação entre os serviços de um processo coreografado. Cada serviço recebe uma requisição e emite uma resposta para quem enviou a requisição e envia uma nova requisição para outro serviço. Quando S_1 recebe Req_1 de C, ele envia uma $Req_{1.1}$ para S_2 e uma Res_1 para C. A $Req_{1.1}$ gerará uma série de requisições até que a requisição $Req_{1.5}$ chegue a S_1 fechando o ciclo. Esse fluxo é característico de uma coreografia, não havendo uma coordenação central e os serviços executando processos em cadeia disparado por uma requisição. A principal vantagem dessa solução é que cada sistema deve conhecer apenas os sistemas que acessam, assim S_1 deve conhecer apenas S_2, S_2 deve conhecer apenas S_3 e assim por diante. No entanto, não é simples monitorar o processo, uma vez que ele é distribuído sem uma coordenação central.

Figura 5. Interações coreografadas de um processo de longa duração.

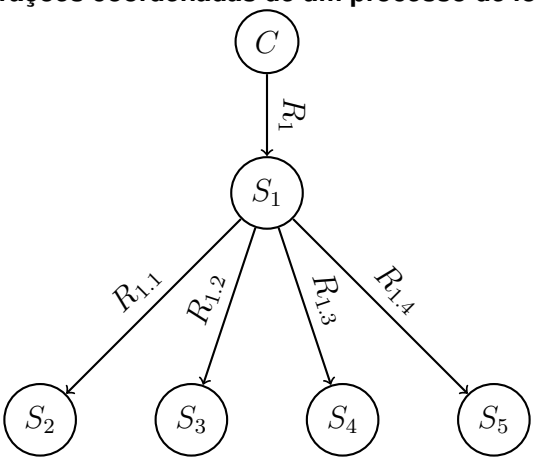


Fonte: Adaptado de Malyuga et al. 2020

Já processos orquestrados são coordenados por um serviço central que fará todas as chamadas aos demais serviços necessários para completar a execução, esperando as respostas e garantindo a confiabilidade e segurança no processo (Malyuga et al. 2020). Quanto mais complexa se torna a LLT, mais complexo se tornará o serviço coordenador, aumentando o tempo de resposta final e as dependências do mesmo (Yildiz et al. 2008). Também não existe um padrão de coordenação de orquestração.

A Figura 6 apresenta um sistema onde S_1 assume o papel de coordenar o processo de longa duração enviando requisições para S_2, S_3, S_4 e S_5. A principal desvantagem é que S_1 deverá conhecer todos os sistemas que deve acessar, mas em compensação a execução do processo poderá ser facilmente monitorada por S_1.

Figura 6. Interações coordenadas de um processo de longa duração.



Fonte: Adaptado de Malyuga et al. 2020

3. Middleware Orientado a Mensagens

Os *Middlewares* Orientados a Mensagens (MOM) surgem como uma solução para integração de sistemas heterogêneos sem a necessidade de acoplamento entre os mesmos. Inicialmente proposto como uma forma de conectar diversas empresas a fim de automatizar processos extremamente complexos, como negociações na bolsa de valores e leilões digitais (Banavar et al. 1999), as soluções existentes hoje buscam uma gestão escalável da informação em sistemas distribuídos por meio de eventos (Goel et al. 2003).

Em qualquer sistema orientado a eventos, eventos são a unidade básica de dados, armazenados em espaços informacionais (Banavar et al. 1999) chamados de tópicos ou filas e podem ser entregues de dois modos: Ponto-a-Ponto ou Publicação/Subscrição (Yongguo et al. 2019; Fu et al. 2021).

As principais implementações de código aberto de MOMs são ActiveMQ, RabbitMQ, Mosquitto e Apache Kafka. A relação dos protocolos implementados por cada MOM é listada na tabela 1.

Tabela 1. Lista de protocolos implementados por cada MOM.

Apache Kafka	RabbitMQ	ActiveMQ	Mosquitto
Kafka	AMQP,	OpenWire, STOMP, MQTT, XMPP	MQTT, STOMP, REST, XMPP, AMQP

O Kafka, por sua complexidade, usa um protocolo próprio, mas as demais implementações usam os protocolos abertos: Protocolo avançado de enfileiramento de mensagens (*AMQP — Advanced Message Queuing Protocol*), Transporte de Telemetria com Filas de Mensagens (*MQTT — Message Queuing Telemetry Transport*), REST, Protocolo Extensível de Mensagens e Presença (*Extensible Messaging and Presence Protocol — XMPP*) e Protocolo de Mensagens Orientado a Texto Simples (*Simple Text*

Oriented Messaging Protocol — STOMP). Cada implementação possui sua especificidade e sua arquitetura interna que deve direcionar a escolha do protocolo em um determinado contexto. Por exemplo, o MQTT é um protocolo de comunicação leve para uso em dispositivos com capacidade de processamento limitado, sendo muito usado em dispositivos IoT (Yongguo et al. 2019). Já o Apache Kafka é uma plataforma Publicação/Subscrição baseada em *logs* de eventos distribuídos (Kleppmann 2021) capaz de processar uma grande vazão de mensagens em paralelo com alta confiabilidade (Wang et al. 2015; Aung et al. 2020).

Fu et al. 2021 e Yongguo et al. 2019 apresentam as seguintes características de um MOM: Elementos Arquiteturais, Mensagem, Gestão da Fila, Modo de Entrega de Mensagens, Modo de Consumo de Mensagens, Funcionalidades de *Cluster*, Garantias de Qualidade de Serviço e Métricas. Essas caraterísticas estão representadas na Figura 7 e descritas nas próximas subseções.

Figura 7. Mapa mental de um *Middleware* orientado a Mensagens.

Fonte: Autoria própria

3.1. Elementos Arquiteturais

Pode-se descrever a arquitetura de um sistema usando MOM associando suas estruturas como componentes, conectores e modelos de dados (Bass et al. 2013). Um componente é sempre um processo em execução, um conector é um protocolo pelo qual os componentes se comunicam e o modelo de dados são as entidades trocadas nesse sistema.

Em um sistema que utiliza MOM existem claramente três tipos de componentes que atuam independentemente:

- **Produtores** são todos os processos que produzem mensagens;
- **Consumidores** são todos os processos que consomem mensagens;
- ***Brokers*** são os processos responsáveis por direcionar as mensagens dos produtores para os consumidores que a desejam. Essas mensagens podem ser armazenadas, caso sejam persistentes.

Esses papéis não são excludentes, o que significa que um processo produtor pode ser um processo consumidor, e as funções de *broker* podem ser distribuídas pelos componentes em execução.

Os processos *brokers* podem possuir dois tipos de arquitetura: *Peer-to-Peer* e Primário/Réplica. Em uma arquitetura *Peer-to-Peer*, todos os *brokers* possuem responsabilidades compartilhadas e as mesmas funções, dividindo as responsabilidades dinamicamente entre os produtores e consumidores. Na arquitetura Primário/Réplica, um *broker* desempenha o papel principal de responder aos produtores e consumidores em suas requisições, enquanto as réplicas estarão disponíveis para assumir as responsabilidades do primário quando este estiver indisponível.

3.2. Mensagem

Para um MOM, uma mensagem é a unidade básica de comunicação entre as partes. Uma mensagem contêm dados de negócio e de controle, e possui em três partes (Yongguo et al. 2019; Wang et al. 2015):

- **Chave**: peça importante, mesmo que opcional, que irá definir o fluxo de consumo da mensagem através do particionamento e possibilitar agrupamento de mensagens em caso de processamento;
- **Cabeçalho**: informações de controle em formato chave-valor. As informações a serem enviadas como cabeçalho são o tipo da mensagem, informações de roteamento, informações de codificação ou códigos de rastreio;
- **Corpo**: informações de negócios da aplicação. A forma de codificação da mensagem deve ser definida pela aplicação.

Todos os *middlewares* possuem bibliotecas próprias em diversas linguagens e as mensagens podem ser enviadas por protocolos de comunicação exclusivos (Kafka) ou abertos, como no MQTT, AMQP, STOMP e XMPP. O formato de envio de mensagens é binário, mas alguns fornecem interfaces de configuração para que a mensagem seria codificada/decodificada pela própria biblioteca (Yongguo et al. 2019).

3.3. Gestão da Fila

Os MOMs podem garantir diversas maneiras de gerir como a mensagem deve ser armazenada, ou não. Por armazenamento deve-se compreender a durabilidade e a confiabilidade na qual a mensagem persistirá na fila/tópico (Yongguo et al. 2019; Fu et al. 2021).

Uma mensagem pode ser um dado efêmero como acessos a uma página na web ou dados persistentes como as informações de um usuário criado. Ambos os dados devem ser armazenados de forma diferente, logo o mecanismo de persistência deve definir se a mensagem é persistente ou transiente.

O MOM deve também garantir a integridade da mensagem durante o envio dela entre o produtor e o *broker* e entre o *broker* e o consumidor. Outra responsabilidade do MOM é garantir que a expansão da fila não impacte na integridade das mensagens pela impossibilidade de envios por falta de espaço. É necessário que o MOM apague mensagens antigas ou já consumidas para permitir a liberação espaço na fila ou o tópico.

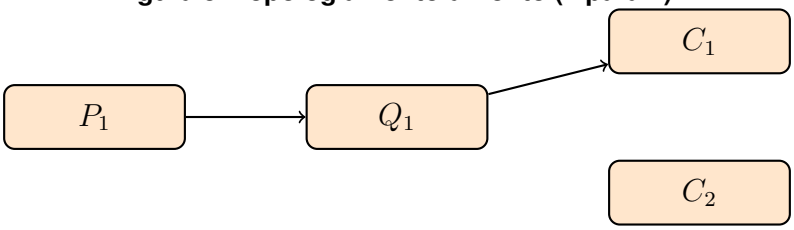
3.4. Modo de Entrega de Mensagens

Uma aplicação usando MOM pode funcionar mediante dois modos de entrega de mensagens (Yongguo et al. 2019; Fu et al. 2021; Hapner et al. 2003; Messaging 2022):

- **Ponto-a-Ponto** (*Point-to-Point — P2P*): em uma comunicação Ponto-a-Ponto, um produtor envia uma mensagem especificamente para um consumidor mediante uma fila. Uma mesma mensagem não será enviada para dois consumidores;
- **Publicação/Subscrição** (*Publish/Subscribe — Pub/Sub*): em uma comunicação Publicação/Subscrição, produtores publicam mensagens em tópicos e consumidores subscrevem a um ou mais tópicos. Um mesmo tópico pode ser consumido por vários consumidores ao mesmo tempo, e uma mesma mensagem pode ser enviada para vários consumidores.

A definição do modo de entrega depende de como a mensagem deve ser consumida. As Figuras 8, 9 e 10 exemplificam os modos de entrega exibindo o caminho que cada mensagem pode seguir do produtor ao consumidor. Na Figura 8, por exemplo, uma fila Q_1 conecta P_1 e C_1 enquanto C_2 fica em modo de espera para substituir C_1 em caso de falha. Caso P_1 envie uma mensagem para Q_1, essa só será entregue ao consumidor ativo, pois o modo Ponto-a-Ponto entrega uma mensagem apenas uma vez.

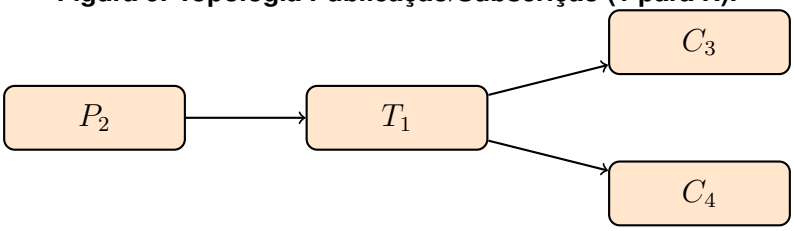
Figura 8. Topologia Ponto-a-Ponto (1 para 1).



Fonte: Adaptado de Yongguo et al. 2019

A Figura 9 representa um sistema, no modo Publicação/Subscrição, com um tópico T_1 conectando um produtor P_2 a dois consumidores C_3 e C_4. No modo Publicação/Subscrição, as mensagens são armazenadas em Tópicos, no caso T_1, e todas as mensagens do tópico devem ser entregues a todos os consumidores subscritos, C_3 e C_4. No caso abaixo, cada consumidor deve receber todas as mensagens armazenadas em T_1. Caso um novo consumidor subscreva-se em T_1, ele deve também receber todas as mensagens armazenadas em T_1.

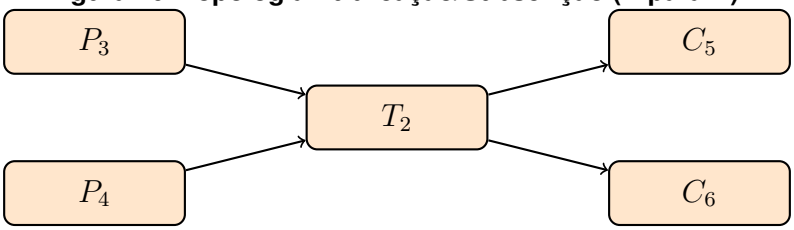
Figura 9. Topologia Publicação/Subscrição (1 para N).



Fonte: Adaptado de Yongguo et al. 2019

A Figura 10 tem um comportamento bem semelhante ao sistema da Figura 9, mas neste caso se tem mais de um produtor. Como P_3 e P_4 estão mandando mensagens para T_2, o tópico deve armazenar todas as mensagens recebidas e enviá-las para C_5 e C_6. Um tópico pode receber mensagens de vários produtores e deve encaminhar todas as mensagens recebidas para todos os consumidores subscritos a ele.

Figura 10. Topologia Publicação/Subscrição (N para N).



Fonte: Adaptado de Yongguo et al. 2019

3.5. Modo de Consumo de Mensagens

O mecanismo de envio de mensagens para os consumidores pode ser implementado de duas formas distintas (Fu et al. 2021): *Push* e *Pull*. O modo *Push* é quando o *broker* decide sobre o momento de enviar mensagens a um consumidor e a quantidade de mensagens para enviar. Já o modo *Pull* é quando cabe ao processo consumidor essa decisão.

Para sistemas implementados usando *Push* há um menor atraso na entrega das mensagens, mas o consumidor não conseguirá controlar o fluxo de consumo de mensagens, o que por resultar em uma alta carga de processamento e até possíveis falhas por falta de memória.

Em sistemas implementados usando *Pull*, o atraso pode ser maior, mas o consumidor poderá controlar quantas mensagens consegue consumir mantendo uma vazão constante, evitando pico de consumo de recursos como memória e processamento.

3.6. Funcionalidades de *Cluster*

Um *cluster* é uma coleção de processos em execução colaborando entre si para desempenhar uma tarefa. Para um MOM, atuar em *cluster* possibilita entregar alta disponibilidade que é um atributo de qualidade essencial (Yongguo et al. 2019).

Um MOM deve ser tolerante a falhas. Para prevenir que não exista um Ponto Único de Falha (*Single Point of Failure*) é preciso existir mais de um *brokers* em execução

atuando com redundância de forma que se um *broker* falhar, outro pode assumir sua função sem haver prejuízos à entrega de mensagens.

Outro requisito de qualidade importante para os MOMs é a possibilidade de balancear a carga de processamento de mensagens. O *broker* deve conseguir absorver picos de envio de mensagens e enviar aos consumidores um fluxo de mensagens suficientemente alto para não aumentar o atraso na entrega das mensagens, mas respeitando a capacidade de processamento do consumidor.

3.7. Garantias de Qualidade de Serviço

Sistemas baseados em mensagens necessitam assumir algumas garantias de qualidade para funcionar, as quais impactarão em como algumas funcionalidades são implementadas e quais as abordagens devem ser tomadas em casos de falhas (Fu et al. 2021). Como exemplos de garantias, pode-se citar (Fu et al. 2021):

- **Garantia de Entrega**: existem três tipos de garantias de entrega: "no máximo uma vez", "ao menos uma vez" e "exatamente uma vez". No modo "no máximo uma vez" (*at-most-once*), mensagens podem ser perdidas, pois elas não serão retransmitidas ao consumidor em caso de falha. Em "Ao menos uma vez" (*at-least-once*) mensagens podem ser recebidas mais de uma vez pelo consumidor, mas há a garantia de que todas as mensagens serão recebidas. E em "exatamente uma vez" (*exactly-once*) todas as mensagens são recebidas uma única vez pelo consumidor. Este último modo é mais custoso e não é garantido por todas as implementações de MOM;
- **Garantia de Ordenação**: existem três tipos de garantia de ordenação: sem-ordenação, ordenação-particionada e ordenação-global. O modo mais simples é o "sem-ordenação" (*no-ordering*), quando não há garantia de ordenação das mensagens recebidas. Quando a ordenação é garantida para um subconjunto de mensagens, uma partição, se tem o modo "ordenação-particionado" (*partition-ordering*). Já quando todas as mensagens recebidas estão ordenadas, se tem a "ordenação-global" (*global-ordering*), sendo o mais custoso dos modos por envolver a sincronização entre produtores e consumidores;
- **Confiabilidade**: falhas de hardware ou software podem acontecer em qualquer componente do sistema. Um MOM deve garantir que as mensagens sejam entregues mesmo havendo falhas;
- **Escalabilidade**: sistemas usando MOM podem escalar com a adição de novos recursos. Ao contrário dos sistemas cliente-servidor, há a possibilidade de se escalar por particionamento, criando novos processos em novas máquinas e redirecionando partições para esses novos processos;
- **Suporte a Transações**: falhas não podem gerar estados intermediários. Para isso, o suporte a transações deve garantir que operações sejam atômicas, o que significa que podem ser realizadas completamente ou descartadas.

3.8. Métricas

Sistemas usando MOM tem métricas de desempenho semelhantes aos sistemas cliente-servidor. Porém, como o processamento da mensagem é desacoplado da sua produção, o atraso do consumidor, o *lag*, é medido independentemente do atraso da mensagem. Assim, as métricas mais importantes para se verificar o desempenho são (Ehrenstein 2020):

- **Atraso**: é uma métrica relacionada à mensagem e definida como a diferença de tempo entre o momento em que a mensagem foi gerada no produtor e o momento que se finalizou o processamento da mesma no consumidor. O atraso pode ter algum impacto em certos processos, mas em outros processos não é importante, indicando apenas que existe uma economia de recursos de processamento para se evitar o uso exacerbado em momentos de pico (Fu et al. 2021);
- ***Lag***: é uma métrica do consumidor definida como a diferença entre o deslocamento da mensagem publicada mais recentemente e o deslocamento confirmado do consumidor. Embora seja comum ter variações naturais nas taxas de produção e consumo, a taxa de consumo não deve ser mais lenta do que a taxa de produção por um período prolongado (IBM 2023);
- **Vazão**: é uma métrica dos processos produtores e consumidores definida pelo número de mensagens processadas em um determinado intervalo de tempo. Valores médios e de pico podem ser monitorados para se verificar o desempenho do sistema.

O fluxo de mensagens em um sistema usando MOM pode ser descrito como um grafo em que cada nó é um tópico/fila e cada aresta uma função lambda (λ) que transformará a mensagem em uma operação com ou sem estado (Tsenos et al. 2021; Ehrenstein 2020). Neste grafo, as mensagens fluem entre vértices (nós), através de arestas. Se a vazão média das arestas produtoras for sempre superior que a vazão média das arestas consumidoras, o *lag* terá um crescimento constante, resultando em uma degradação do desempenho dos consumidores. Dessa forma, define-se como vazão sustentável a vazão máxima que um consumidor pode aceitar sem ocorrer aumento significante no *lag* (Ehrenstein 2020).

No Apache Kafka, a vazão máxima de um consumidor dependerá de vários fatores como o número de partições, número de consumidores atuando em conjunto, quantidade de réplicas configuradas no tópico, disponibilidade do *broker* e disponibilidade de recursos (processamento, acesso a disco, memória) no *broker*. Para conseguir a vazão máxima, é preciso ajustar todos esses fatores e as configurações do *broker* e dos clientes (Raptis and Passarella 2022).

4. Problemas comuns em Middlewares Orientados a Mensagens

Essa seção trata de alguns problemas que acontecem em sistemas baseados em eventos usando MOMs, descrevendo os mesmos e apresentando as soluções propostas. O principal objetivo de um MOM é entregar o maior número de mensagens com o menor atraso aos processos consumidores. Para que isso aconteça, todo o sistema deve operar harmonicamente com os recursos bem distribuídos para não existirem gargalos indesejados. O Apache Kafka foi escolhido para essa análise por ser um MOM que implementa balanceamento de carga para grupos de consumidores (Ezzeddine et al. 2022) e por apresentar uma maior vazão e um conjunto mais abrangente de funcionalidades (Yongguo et al. 2019; Fu et al. 2021).

Alguns problemas encontrados em sistemas que utilizam Apache Kafka são:

1. Desbalanceamento de Carga entre *brokers*;
2. Inanição do Consumidor;
3. Mensagens cronologicamente desordenadas;
4. Distribuição de carga para consumidores enviesada.

4.1. Desbalanceamento de Carga entre *brokers*

O desbalanceamento de carga entre *brokers* acontece quando um produtor produz mais mensagens para um subgrupo de partições alocadas em um subconjunto de *brokers*, fazendo com que exista uma diferença significativa de carga entre os *brokers* (Dedousis et al. 2018).

Para definir se há uma sobrecarga do *broker* é preciso primeiro que exista uma métrica que meça o nível de carga em cada *broker*, depois verificar se algum *broker* tem valores destoantes para essa métrica. O nível de carga de um broker é calculado a partir das vazões de entrada e saída do mesmo.

Para calcular a vazão de entrada $r_{b,p}$, considere um *broker* b pertencente ao conjunto de *brokers* disponíveis B, um produtor pr pertencente ao conjunto de produtores PR enviando mensagens a B, e uma partição p dentre o conjunto total de partições P_b no broker b. A vazão de entrada $r_{b,p}$ em um broker dada uma partição pode ser calculado como a soma de todas as vazões $r_{pr,p}$, conforme a Equação 2 (Dedousis et al. 2018).

$$r_{b,p} = \sum_{pr \in \{pr | pr \in PR, p \in P_{pr}\}} r_{pr,p}, \forall b \in B, \forall p \in P_b \tag{2}$$

Sendo assim, a vazão de entrada de um *broker* é dada por $R_b(P_b)$ que é a soma de todas as vazões $r_{b,p}$, conforme a Equação 3 (Dedousis et al. 2018).

$$R_b(P_b) = \sum_{p \in P_b} r_{b,p}, \forall b \in B \tag{3}$$

Já para a vazão de saída, $l_{c,p}$ é a vazão de uma partição p para um consumidor c pertencente ao conjunto de todos os consumidores C recebendo mensagens de B. Assim, a vazão de saída de um *cluster* $l_{b,p}$ é a soma das vazões de saída de cada partição p armazenada em b, conforme a Equação 4 (Dedousis et al. 2018).

$$l_{b,p} = \sum_{c \in \{c | c \in C, p \in P_c\}} l_{c,p}, \forall b \in B, \ \forall p \in P_b \tag{4}$$

Assim, a vazão de saída do *broker* é representada por $L_b(P_b)$ e definida como a soma de todas as vazões de saída das partições atribuídas a b, conforme a Equação 5 (Dedousis et al. 2018).

$$L_b(P_b) = \sum_{p \in P_b} l_{b,p}, \forall b \in B \tag{5}$$

Dessa forma, pode-se calcular a intensidade de tráfego τ_b no *broker* b conforme a Equação 6 (Dedousis et al. 2018).

$$\tau_b(P_b) = R_b(P_b)/L_b(P_b), \forall b \in B \tag{6}$$

Com base nos valores de τ_b para $\forall b \in B$, pode-se definir se um *broker* b está com sobrecarga se o valor de τ_b se desvia da média significativamente conforme a Equação 7 (Dedousis et al. 2018), onde $\mu(B)$ é a média da intensidade de tráfego de todos os *brokers* B, $\sigma(B)$ é o desvio padrão da intensidade de tráfego e ϵ é um parâmetro a ser definido empiricamente.

$$ol_b(B, P_b) = \begin{cases} 1, & \tau_b(P_b) > \mu(B) + \epsilon \cdot \sigma(B) \\ 0, & else \end{cases} \tag{7}$$

4.2. Inanição do Consumidor

A inanição do consumidor acontece quando a vazão média dos produtores de um tópico é maior que a vazão média de um grupo de consumidores subscrito nesse tópico, fazendo com que o *lag* desse grupo de consumidores cresça continuamente. Um grupo de consumidores são vários processos que consomem mensagens coordenadamente distribuindo a carga entre si. Uma mesma mensagem não é enviada a dois processos do mesmo grupo de consumidores (Bang et al. 2018; Ehrenstein 2020).

A vazão do grupo de produtores pode ser momentaneamente maior que a vazão dos consumidores, mas para que o *lag* tenha um crescimento constante é preciso que a soma das vazões média de todos os produtores seja maior que a soma das vazões médias de todos os consumidores dos tópicos envolvidos.

A solução proposta para esse problema é baseada no descarte de mensagens, ela, porém, só pode ser aplicado em sistemas onde uma mensagem descartada não resulta na perda de informações (Bang et al. 2018). Quando mensagens não podem ser descartada, é necessário aumentar a vazão de saída através do aumento da capacidade de processamento ou da adição de novas partições e processos consumidores (Ezzeddine et al. 2022).

4.3. Mensagens cronologicamente desordenadas

No Apache Kafka, mensagens são geradas pelos processos produtores e, no momento da geração, cada mensagem é associada ao seu horário de criação (*timestamp*) e a uma chave. As mensagens são enviadas posteriormente ao *broker* de forma assíncrona e, em caso de indisponibilidade do *broker*, esse envio será retardado. Como o *timestamp* de uma mensagem é gerado pelo produtor e um mesmo consumidor lê mensagens de diversos produtores, há a possibilidade de existirem mensagens cronologicamente desordenadas para uma mesma chave (Wang et al. 2021).

A existência de mensagens cronologicamente desordenadas é um problema particularmente relevante para consumidores com estado interno que se baseiam no *timestamp* das mensagens, como a agregação de valores em um intervalo de tempo, ou o processamento de séries temporais. Esse problema é comum em aplicações usando a biblioteca Kafka Streams, as quais são um tipo específico de processo que atuam como consumidor e produtor ao mesmo tempo, por possibilitar agrupar mensagens em sequências temporais usando o *timestamp* (Wang et al. 2021).

4.4. Distribuição de carga para consumidores enviesada

O Apache Kafka distribui a carga entre os consumidores através da atribuição de partições a cada consumidor pertencente a um grupo. Assim, a distribuição de carga entre os consumidores é enviesada quando a soma das vazões de um consumidor é muito superior à média da soma das vazões dos outros consumidores pertencentes ao mesmo grupo (Ezzeddine et al. 2022).

A distribuição de carga enviesada acontece porque os algoritmos de atribuições de partições não considera a vazão e o atraso das partições, podendo atribuir a um mesmo processo as partições com maior vazão. O principal indício da distribuição de carga enviesada é o crescimento do atraso em um subconjunto específico de partições.

5. Conclusão

Como visto nesse estudo, *Middlewares* Orientados a Mensagens são uma ferramenta para a construção de sistemas distribuídos. MOMs possibilitam criar sistemas com menor acoplamento entre serviços ao se adotar o modo Publicação/Subscrição, permitindo a criação de fluxos de dados complexos. Eles foram desenvolvidos como uma alternativa às limitações da arquitetura cliente-servidor, buscando aumentar a disponibilidade pela redução de dependências diretas e ao processamento assíncrono de mensagens.

As diversas implementações de MOM podem ser usadas em uma grande variedade de sistemas, desde sistemas embarcados via protocolos como o MQTT, ou sistemas com altíssima vazão usando o Apache Kafka. Sistemas baseados em mensagens podem usar MOMs como barramento de trocas de mensagens, permitindo que novos serviços sejam adicionados sem a necessidade de se alterar os serviços já implantados através do modo Publicação/Subscrição, ou conectar dois serviços diretamente através do modo Ponto-a-Ponto. Esses sistemas podem coexistir com APIs REST, ou outros protocolos cliente-servidor, em uma arquitetura mista, explorando as potencialidades de ambos os estilos arquiteturais.

Essa pesquisa reuniu esforços na caracterização dos MOMs para uma melhor compreensão de como sistemas usando MOM podem ser descritos. Durante a pesquisa, percebeu-se que há alguns problemas comuns nas implementações de sistemas baseados em mensagens. A solução desses problemas considera que as mensagens são passíveis de descarte, uma premissa que só pode ser aplicada a alguns tipos de mensagens, como dados de sensores e séries temporais.

Referências

[Aung et al. 2020] Aung, T., Min, H. Y., and Maw, A. H. (2020). Enhancement of fault tolerance in kafka pipeline architecture. In *Proceedings of the 11th International Conference on Advances in Information Technology*, IAIT2020, New York, NY, USA. Association for Computing Machinery.

[Banavar et al. 1999] Banavar, G., Chandra, T. D., Strom, R. E., and Sturman, D. C. (1999). A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, page 1–18, Berlin, Heidelberg. Springer-Verlag.

[Bang et al. 2018] Bang, J., Son, S., Kim, H., Moon, Y.-S., and Choi, M.-J. (2018). Design and implementation of a load shedding engine for solving starvation problems in

apache kafka. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–4.

[Bass et al. 2013] Bass, L., Clements, P., and Kazman, R. (2013). *Software architecture in practice*. SEI series in software engineering. Addison-Wesley, Upper Saddle River, NJ, 3rd ed edition.

[Berners-Lee et al. 1996] Berners-Lee, T., Fielding, R. T., and Frystyk, H. (1996). Hypertext Transfer Protocol – HTTP/1.0. *Request for Comments*, 1945:1–60.

[Christensen et al. 2001] (2001). Web Services Description Language (WSDL) 1.1. (Acessado em 12 de agosto de 2023).

[Dedousis et al. 2018] Dedousis, D., Zacheilas, N., and Kalogeraki, V. (2018). On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 76–86, Vienna. IEEE.

[Edstrom and Tilevich 2012] Edstrom, J. and Tilevich, E. (2012). Reusable and extensible fault tolerance for restful applications. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 737–744.

[Ehrenstein 2020] (2020). Scalability Benchmarking of Kafka Streams Applications. (Acessado em 19 de agosto de 2023).

[Ezzeddine et al. 2022] Ezzeddine, M., Migliorini, G., Baude, F., and Huet, F. (2022). Cost-Efficient and Latency-Aware Event Consuming in Workload-Skewed Distributed Event Queues. In *Proceedings of the 2022 6th International Conference on Cloud and Big Data Computing*, ICCBDC '22, pages 62–70, New York, NY, USA. Association for Computing Machinery.

[Fu et al. 2021] Fu, G., Zhang, Y., and Yu, G. (2021). A fair comparison of message queuing systems. *IEEE Access*, 9:421–432.

[Garcia-Molina and Salem 1987] Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data - SIGMOD '87*, pages 249–259, San Francisco, California, United States. ACM Press.

[Goel et al. 2003] Goel, S., Sharda, H., and Taniar, D. (2003). Message-Oriented-Middleware in a Distributed Environment. In Böhme, T., Heyer, G., and Unger, H., editors, *Innovative Internet Community Systems*, pages 93–103, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hapner et al. 2003] (2003). JSR 914: Java Message Service (JMS) API. `https://jcp.org/en/jsr/detail?id=914`, (Acessado em 27 de setembro de 2023).

[Hura 1995] Hura, G. (1995). Client-server computing architecture: an efficient paradigm for project management. In *Proceedings for Operating Research and the Management Sciences*, pages 146–152.

[IBM 2023] (2023). IBM Cloud Docs / Event Streams / Consuming messages. `https://cloud.ibm.com/docs/EventStreams?topic=EventStreams-consuming_messages`, (Acessado em 22 de setembro de 2023).

[Kleppmann 2021] Kleppmann, M. (2021). Thinking in events: From databases to distributed collaboration software. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, DEBS '21, page 15–24, New York, NY, USA. Association for Computing Machinery.

[Lawrence et al. 2017] Lawrence, R., Brandsberg, E., and Lee, R. (2017). Next generation jdbc database drivers for performance, transparent caching, load balancing, and scale-out. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 915–918, New York, NY, USA. Association for Computing Machinery.

[Malyuga et al. 2020] Malyuga, K., Perl, O., Slapoguzov, A., and Perl, I. (2020). Fault Tolerant Central Saga Orchestrator in RESTful Architecture. In *2020 26th Conference of Open Innovations Association (FRUCT)*, pages 278–283, Yaroslavl, Russia. IEEE.

[Messaging 2022] (2022). Jakarta Messaging. `https://jakarta.ee/specifications/messaging/3.1/jakarta-messaging-spec-3.1.html`, (Acessado em 04 de julho de 2023).

[OpenAPI 2021] (2021). OpenAPI Specification v3.1.0. `https://spec.openapis.org/oas/v3.1.0`, (Acessado em 15 de julho de 2023).

[Quiña-Mera et al. 2023] Quiña-Mera, A., Fernandez, P., García, J. M., and Ruiz-Cortés, A. (2023). GraphQL: A Systematic Mapping Study. *ACM Computing Surveys*, 55(10):1–35.

[Raptis and Passarella 2022] Raptis, T. P. and Passarella, A. (2022). On efficiently partitioning a topic in apache kafka. In *2022 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 1–8.

[Richardson 2018] Richardson, C. (2018). *Microservices patterns: with examples in Java*. Manning Publications.

[Ritchie et al. 2023] Ritchie, H., Mathieu, E., Roser, M., and Ortiz-Ospina, E. (2023). Internet. *Our World in Data*. https://ourworldindata.org/internet.

[Rudrabhatla 2018] Rudrabhatla, C. K. (2018). Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, 9(8).

[Terpák et al. 2016] Terpák, J., Horovčák, P., and Lukáč, M. (2016). Mathematical models creation using orchestration and choreography of web services. In *2016 17th International Carpathian Control Conference (ICCC)*, pages 739–742.

[Treynor et al. 2017] Treynor, B., Dahlin, M., Rau, V., and Beyer, B. (2017). The Calculus of Service Availability. *ACM Queue*.

[Tsenos et al. 2021] Tsenos, M., Zacheilas, N., and Kalogeraki, V. (2021). Dynamic Rate Control in the Kafka System. In *Proceedings of the 24th Pan-Hellenic Conference on Informatics*, PCI '20, pages 96–98, New York, NY, USA. Association for Computing Machinery.

[Wang et al. 2021] Wang, G., Chen, L., Dikshit, A., Gustafson, J., Chen, B., Sax, M. J., Roesler, J., Blee-Goldman, S., Cadonna, B., Mehta, A., Madan, V., and Rao, J. (2021). Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2602–2613, New York, NY, USA. Association for Computing Machinery.

[Wang et al. 2015] Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., and Stein, J. (2015). Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655.

[Wang 2019] Wang, Y. (2019). Towards service discovery and autonomic version management in self-healing microservices architecture. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, page 63–66, New York, NY, USA. Association for Computing Machinery.

[Yildiz et al. 2008] Yildiz, B., Fox, G., and Pallickara, S. (2008). An orchestration for distributed web service handlers. In *2008 Third International Conference on Internet and Web Applications and Services*, pages 638–643.

[Yongguo et al. 2019] Yongguo, J., Qiang, L., Changshuai, Q., Jian, S., and Qianqian, L. (2019). Message-oriented middleware: A review. In *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, pages 88–97.

[Zur Muehlen et al. 2005] Zur Muehlen, M., Nickerson, J. V., and Swenson, K. D. (2005). Developing web services choreography standards—the case of REST vs. SOAP. *Decision Support Systems*, 40(1):9–29.