UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MIDDLEWARE ESCALÁVEL E RESILIENTE PARA COMUNICAÇÃO EM TEMPO REAL DE APLICAÇÕES WEB EM AMBIENTES DE COMPUTAÇÃO DISTRIBUÍDA

ARIEL ADONAI SOUZA

BLUMENAU
2022

ARIEL ADONAI SOUZA

MIDDLEWARE ESCALÁVEL E RESLIENTE PARA COMUNICAÇÃO EM TEMPO REAL DE APLICAÇÕES WEB EM AMBIENTES DE COMPUTAÇÃO DISTRIBUÍDA

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Francisco Adell Péricas, Ms. - Orientador

BLUMENAU
2022

MIDDLEWARE ESCALÁVEL E RESILIENTE PARA COMUNICAÇÃO EM TEMPO REAL DE APLICAÇÕES WEB MÓVEIS EM AMBIENTES DE COMPUTAÇÃO DISTRIBUÍDA

Por

ARIEL ADONAI SOUZA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof(a). Francisco Adell Péricas – Orientador(a), FURB
Membro:	Prof(a). Gilvan Justino – FURB
Membro:	Prof(a). Aurélio Faustino Hoppe – FURB

Blumenau, 07 de julho de 2022

Dedico este trabalho à minha família, amigos, professores e minha companheira, que me apoiaram em minha jornada até este momento.

AGRADECIMENTOS

Aos meus pais, Nelcy Renostro Souza e Mauri Souza, que sempre me apoiaram, incentivaram e possibilitaram eu chegar até aqui.

A minha companheira Isabella Clemente dos Santos por todo apoio, auxílio e compreensão durante a jornada do curso.

Ao meu orientador Francisco Adell Péricas pelo suporte e dedicação na orientação deste trabalho.

Eu não quero acreditar, eu quero saber.

Carl Sagan

RESUMO

Este trabalho apresenta a especificação e implementação de um *middleware* escalável e resiliente para comunicação entre aplicações web e aplicações em ambiente distribuído. O *middleware* faz uso do protocolo WebSocket para comunicação com as aplicações web e do protocolo AMQT para comunicação com os serviços. Além disso, a resiliência dele está relacionada à capacidade de entregar mensagens não entregues quando alguma aplicação estava desconectada. O *middleware* foi desenvolvido utilizando o *framework* Quarkus e Java 11, hospedado na AWS, onde alguns serviços dela foram utilizados para proporcionar um ambiente estável e com facilidade de escala. A partir dos resultados dos testes realizados foi possível observar que o *middleware* atendeu aos objetivos de escalabilidade, resiliência na entrega de mensagens e comunicação entre aplicações web e ambientes distribuídos. Ele se demonstrou uma solução viável para abstrair e simplificar a comunicação de aplicações web para dispositivos móveis e aplicações em ambientes distribuídos, retirando a necessidade de ambas as aplicações conversarem através de um mesmo protocolo e sem a necessidade de gerenciar a entrega das mensagens.

Palavras-chave: Middleware. Aplicações Web. AMQT. Sistemas Distribuídos.

ABSTRACT

This paper presents the specification and implementation of a scalable and resilient middleware for communication between web applications for mobile devices and applications in a distributed environment. The middleware makes use of the WebSocket protocol for communicating with web applications and the AMQT protocol for communicating with services. In addition, its resilience is related to the ability to deliver messages that were not delivered when some application was disconnected. The middleware was developed using the Quarkus framework and Java 11, hosted on AWS, where some services were used to provide a stable environment with ease of scale. From the results of the tests carried out, it was possible to observe that the middleware met the objectives of scalability, resilience in the delivery of messages and communication between web applications and distributed environments. It proved to be a viable solution to abstract and simplify the communication of web applications to mobile devices and applications in distributed environments, removing the need for both applications to talk through the same protocol and without the need to manage the delivery of messages.

Key-words: Middleware. Web Applications. AMQT. Distribuited Systems.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AMQT – Advanced Message Queuing Protocol

API – Application Programming Interface

AWS – Amazon Web Services

BFF – Backend for Frontend

CRUD – Create Read Update Delete

CSS – Cascading Style Sheets

DNS – Domain Name System

EC2 – Elastic Compute Cloud

ECR – Elastic Container Registry

HTML – Hiper Text Markup Language

HTTPS – Hiper Text Transfer Protocol Secure

HTTP – Hiper Text Transfer Protocol

JSON – JavaScript Object Notation

LAN – Local Area Networks

REST – REpresentational State Transfer

TLS – Transport Layer Security

URL – Uniform Resource Locator

UUID – Universally Unique IDentifier

SUMÁRIO

1 INTRODUÇÃO

O mundo está cada vez mais rodeado por aplicativos móveis. Estes aplicativos estão se tornando a principal ferramenta de comunicação das pessoas com a Internet, conforme citado por Oehlman e Blanc (2012). Diante desse cenário, é natural que surja uma demanda crescente de ferramentas e tecnologias que embarquem os seus softwares corporativos nos dispositivos móveis. Oehlman e Blanc (2012) também contam que em empresas com aplicações web, os desenvolvedores reconstroem grandes porções dessas aplicações web dentro dos aplicativos móveis para cada um dos diferentes dispositivos. “Para algumas companhias que constroem aplicativos móveis, esta é uma metodologia aceitável. É, contudo, uma das menos sustentáveis a longo prazo” (OEHLMAN; BLANC, 2012, p. 9). Os autores apontam para um futuro em que a demanda por desenvolvedores seria muito grande para conseguir manter todas essas aplicações por conta da grande variedade de dispositivos móveis disponíveis hoje no mercado.

Uma solução mais viável é desenvolver uma aplicação web para dispositivos móveis. Segundo Oehlman e Blanc (2012), aplicações web são uma forma de escrever aplicações que, quando feito da forma correta, permite adaptar as aplicações para dispositivos móveis sem a necessidade de reescrever muito código. “Um aplicativo web para dispositivos móveis é um aplicativo construído com as tecnologias web centradas no cliente em HTML, CSS e JavaScript, e é especificamente projetado para os dispositivos móveis” (OEHLMAN; BLANC, 2012, p. 9).

Segundo Tanenbaum e Steen (2008), em meados de 1980 ocorreram dois avanços tecnológicos que revolucionariam a computação até os dias atuais. O primeiro foi a evolução de microcontroladores de maior capacidade, que com o passar do tempo chegavam à capacidade de processamento de um *mainframe* por um preço muito menor. O segundo avanço foi o desenvolvimento das redes de computadores e o surgimento das redes locais, as Local Area Networks (LANs). As LANs permitiram que centenas de computadores próximos pudessem trocar informações na velocidade de alguns microssegundos. O resultado do surgimento destas duas tecnologias foi um modelo de computação de altíssima capacidade de processamento conectado em uma rede de alta velocidade, os sistemas distribuídos.

De acordo com Tanenbaum e Steen (2008, p. 1), um sistema distribuído é “um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente.” Eles ressaltam aspectos importantes: um sistema distribuído consiste em

computadores autônomos e usuários (programas, aplicações ou pessoas) que devem achar que tratam com um sistema único.

Segundo Tanenbaum e Steen (2008), em um ambiente de computação distribuída, por conveniência, é utilizado um modelo de comunicação assíncrona e existem diversos sistemas por trás de uma única aplicação. O RabbitMQ é uma aplicação que pode servir como *broker* para comunicação entre os serviços de maneira assíncrona, conforme descrito por Barthel (2009). O RabbitMQ utiliza o protocolo AMQT para comunicação com as aplicações, protocolo este que não é compatível com aplicações web. Diante desse cenário, é possível imaginar que seja complexo ou inviável uma aplicação web, que está fora deste ambiente, se comunicar com os diversos sistemas que possam existir.

Por mais que à primeira vista pareça complexo estabelecer a comunicação entre a aplicação web e os diversos sistemas em um ambiente distribuído, existe um tipo de software que soluciona esta situação: os *middlewares*. Um *middleware* funciona "[...] de forma essencial como uma camada oculta de tradução, o *middleware* permite a comunicação e o gerenciamento de dados para aplicativos distribuídos." (AZURE, 2021, p. 1). Com a utilização de um *middleware*, a aplicação web se comunica de forma transparente com os sistemas.

As plataformas em nuvem oferecem diversos serviços que podem ser configurados de forma simples e já iniciam com segurança, escala e alta disponibilidade. Essas plataformas são muito flexíveis e rápidas, permitindo que os sistemas escalem e as vezes de forma transparente. Podem ser grandes aliadas ao construir sistemas que precisem ser altamente disponíveis e com capacidade de escala.

Diante do cenário apresentado, levando em consideração o aumento do uso de dispositivos móveis, a facilidade e conveniência em implementar aplicações web para dispositivos móveis, este trabalho se propõe a implementar um *middleware* que possa atender a este contexto.

1.1 OBJETIVOS

O objetivo deste trabalho é apresentar um *middleware* que seja escalável e resiliente para comunicação cliente/servidor em tempo real entre aplicações web e sistemas em um ambiente de computação distribuída.

Os objetivos específicos são:

a) avaliar a escalabilidade do software usando AWS;

b) avaliar a resiliência na entrega de mensagens, ou seja, a mensagem poderá ser

enviada até mesmo se o destinatário estiver incomunicável no momento do envio (a entrega da mensagem deverá ocorrer quando o destinatário se conectar novamente);

c) atender a um contexto de comunicação entre um ambiente distribuído e aplicações web.

1.2 ESTRUTURA

A estrutura deste trabalho é apresentada em quatro capítulos. O primeiro apresenta a introdução e os objetivos gerais e específicos. O segundo capítulo apresenta a fundamentação teórica que embasará o trabalho. O terceiro capítulo apresenta o desenvolvimento do *middleware* e a sua especificação, onde são apresentados os casos de uso, fluxos de atividade, diagrama de arquitetura, trechos de código para complementar a explicação e a análise dos resultados. O quarto capítulo apresenta a conclusão do trabalho e as sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados os conceitos e fundamentos mais importantes para realização deste trabalho. A seção 2.1 trará informações sobre a história e evolução dos dispositivos móveis, o que são aplicações web para dispositivos móveis, o que é o padrão de arquitetura BFF, o que é o padrão REST e o que é o WebSocket. A seção 2.2 conta o início dos sistemas distribuídos, apresenta alguns conceitos que os definem e introduz a respeito da computação em nuvem. A seção 2.3 trará informações sobre o que é um *middleware* e traça um paralelo com o padrão de arquitetura BFF (Backend For Frontend).

2.1 APLICAÇÕES WEB PARA DISPOSITIVOS MÓVEIS

Antes de falar sobre o que é uma aplicação web para dispositivos móveis, é importante falar um pouco sobre os dispositivos móveis e como eles impactaram na internet e na vida das pessoas de maneira geral.

O mundo vive uma revolução *mobile* que começou na década de 2000. Duarte (2016) conta que os dispositivos móveis já existem desde a década de 1970, mas que a maior revolução aconteceu com a chegada da indústria dos apps somente em 2007. Foi a Apple quem deu o ponta pé inicial na indústria dos apps com a chegada do iPhone. O dispositivo trazia um conceito novo de experiência focada em *touchscreen* e a App store. A App store é uma loja de aplicativos que deu início a toda essa revolução. Através dela, qualquer desenvolvedor do mundo poderia escrever um app para o iPhone, publicá-lo e distribui-lo para o mundo todo. Até o momento da criação da App store, o desenvolvimento de aplicativos estava restrito a empresas credenciadas junto às fabricantes. Com a sua chegada, o desenvolvimento foi aberto a qualquer desenvolvedor.

Existem diversos tipos de dispositivos móveis, porém neste trabalho será tratado somente o *smartphone*. Um *smartphone* “é um telefone celular com um sistema operacional de verdade. Ou seja, o hardware é construído e depois é instalado um sistema operacional, que pode ser reinstalado outras vezes caso necessário, bem como aplicativos.” (DUARTE, 2016, p. 9).

Segundo a BBC News Brasil (2021), a chegada do 3G e a ampliação da infraestrutura global para o 3G foi essencial para que a revolução acontecesse. A geração anterior de conectividade móvel, o 2G, era muito lenta para navegar na internet, com velocidade máxima em cerca de 300 kbps, o que tornava inviável acessar sites através do 2G. O 3G oferecia uma capacidade muito maior de banda, cerca de 4 Mbps e em média era dez vezes mais rápida que a geração anterior. “Globalmente, o 3G permitiu a expansão dos telefones celulares

inteligentes, oferecendo uma experiência em movimento semelhante ao uso da internet por um computador conectado ao um cabo na parede. Sem a infraestrutura do 3G, os telefones celulares continuariam presos a mensagens de texto por SMS e conteúdo básico, e a revolução da mobilidade não teria sido possível.” (BBC NEWS BRASIL, 2021, p. 1).

A crescente adição de funcionalidades proveniente dos apps fez com que, gradativamente, muitos outros aparelhos e objetos fossem substituídos pelo *smartphone*. Câmeras, relógios de pulso, computadores de mesa e até consoles de jogos foram sendo substituídos à medida que os *smartphones* foram evoluindo, conforme documentado por BBC News Brasil (2021). O celular virou um objeto de uso diário e até mesmo essencial para ter uma vida social.

Segundo Isabela Mendes (2021), em 2021 96,1% dos usuários de internet no Brasil faziam uso da internet através de um *smartphone* e 71,6% faziam uso através de um computador. Fica evidente a importância de desenvolver aplicações web cada vez mais voltadas para o mundo móvel.

Oehlman e Blanc (2012) descrevem uma forma de criar aplicações web voltadas totalmente para dispositivos móveis. São aplicativos construídos com as tecnologias web centradas no cliente em HTML, CSS e JavaScript. Essa abordagem permite que os desenvolvedores possam reaproveitar trechos importantes das aplicações e não precisem criar aplicações totalmente diferentes para computadores e dispositivos móveis. Os autores utilizaram um conceito chamado de Bridging Framework, um *framework* que encapsula uma aplicação web dentro de um aplicativo para dispositivo móvel e fornece à aplicação web acesso a alguns recursos do telefone como câmera e acelerômetro. Desta forma, o usuário que utiliza o aplicativo tem a impressão de estar utilizando um aplicativo nativo, porém na realidade está utilizando uma aplicação web muito semelhante, se não igual, às aplicações web acessadas pelo navegador. Esses *frameworks* também possibilitam que o desenvolvedor possa compilar o aplicativo para diferentes plataformas (Android ou iOS) com poucas ou nenhuma alteração.

As aplicações web e aplicativos são geralmente o que é chamado de *frontend*. Grande parte das aplicações funcionam num modelo *frontend* e *backend*. O *frontend* é uma aplicação que não possui regras de negócio e não persiste os dados, ele serve somente uma interface visual de comunicação com a aplicação que mantem as regras de negócio, o *backend*. O *backend* pode ser uma ou mais aplicações que possuem as regras de negócio e de fato persistem os dados. Há diversas formas de integrar essas aplicações, dentre elas estão o padrão REST, o WebSocket e o padrão de arquitetura BFF.

2.1.1 REST

Saudate (2013) conta que REST é o que se pode chamar de estilo de desenvolvimento de *web services*. Foi originado na tese de doutorado de Roy Fielding, coautor do protocolo HTTP. O padrão REST é guiado pelas boas práticas de uso do HTTP como: uso adequado dos métodos; uso adequado de URL's; códigos de status padronizados; dentre outros. Por se tratar de um padrão e não de um protocolo, ele pode até ser utilizado em outros protocolos, porém o HTTP é o único que consegue ser totalmente compatível.

Conforme demonstrado por Saudate (2013), os serviços REST são divididos em recursos, que representam alguma entidade no sistema e possuem uma URL própria. Um exemplo seria o recurso `Cervejas` que pode ser representado com a URL `http://site.com/cervejas`, onde `http` é o protocolo, `site.com` é o nome de domínio do servidor e `/cervejas` é o recurso. Para os demais exemplos deste texto, serão omitidos o protocolo e o nome de domínio, ficando somente o recurso `/cervejas`.

O padrão REST define que a URL `/cervejas` deve realizar a interação com todas as cervejas do sistema. Para manipular um recurso específico, deve ser passada na URL um identificador. O identificador é fornecido logo após o recurso e separado por uma barra, por exemplo com o código de barras: `/cervejas/8712000031671`.

Os métodos HTTP são muito importantes dentro do padrão REST. O método GET é utilizado para obter dados, o método POST é utilizado para criação de novos recursos ou executar tarefas, o método PUT é utilizado para atualizar recursos e o método DELETE é utilizado para remover recursos.

As operações tradicionais de uma entidade são chamadas de CRUD (Create Read Update Delete). O padrão REST define os seguintes padrões:

a) método GET e URL de recurso sem identificador, obtém todos os recursos;
b) método GET e URL de recurso com identificador, obtém o recurso pelo identificador;
c) método POST e URL de recurso sem identificador, cria um recurso e o identificador é gerado pelo sistema;
d) método PUT e URL de recurso com identificador, atualiza o recurso pelo identificador;
e) método DELETE e URL de recurso com identificador, remove o recurso pelo identificador.

É muito comum existir cenários que fogem do escopo de uma CRUD. “Para obter resultados satisfatórios, deve-se sempre ter o pensamento voltado para a orientação a recursos. Tomemos como exemplo o caso do envio de e-mails: a URL deve ser modelada em formato de substantivo, ou seja, apenas /email. Para enviar o e-mail, o cliente pode usar o método POST - ou seja, como se estivesse ‘criando’ um e-mail” (SAUDATE, 2013, p. 31).

2.1.2 WebSocket

De acordo com Lombardi (2015), o WebSocket é um protocolo bidirecional, full-duplex (os dois dispositivos podem transmitir simultaneamente), que opera em cima do protocolo HTTP utilizando somente uma conexão. No protocolo HTTP, uma requisição parte do cliente abrindo uma conexão com o servidor, enviando os dados, recebendo a resposta do servidor e geralmente fechando a conexão. No protocolo WebSocket, a conexão inicia com uma requisição HTTP e fica aberta até que o cliente ou o servidor feche a conexão, sendo possível transmitir quaisquer dados do cliente para o servidor e vice-versa a qualquer momento.

Lombardi (2015) conta que existem alguns *hacks*, isto é, maneiras de trabalhar somente com HTTP de forma que pareça que existe uma conexão WebSocket na aplicação web. Uma delas é o chamado *long polling* que consiste no cliente constantemente realizar requisições ao servidor perguntando se existe algum dado novo. Utilizando o WebSocket a situação é diferente pois o cliente e o servidor possuem uma conexão bidirecional aberta o tempo todo. Isso permite o servidor enviar o dado sem a necessidade de o cliente ficar perguntando a todo momento se existe algum dado novo. Esses *hacks* até funcionam e o usuário pode até não perceber a diferença, porém do lado do servidor a história é diferente.

Tendo em vista os problemas que outras abordagens trazem, o WebSocket se torna uma solução mais viável para a utilização quando se tratar de aplicações web onde é necessário que o servidor se comunique com o cliente. Um cenário muito comum é em aplicações de chat, conforme apontado por Lombardi (2015).

2.1.3 Backend for Frontends

Vem crescendo muito o número de aplicações web para computadores após as diversas melhorias realizadas na infraestrutura global nas últimas décadas. Muitas dessas aplicações são interfaces de aplicações já existentes para computadores fora da web e que foram sendo migradas para o mundo web, conforme apontado por Newman (2015).

O mundo vive uma verdadeira revolução móvel nos últimos anos e essa revolução vem trazendo a necessidade dessas aplicações também estarem disponíveis no mundo móvel. Newman (2015) conta que isso traz um problema pois o mundo móvel é diferente do mundo web e possui necessidades diferentes. Seria necessário manter uma interface no *backend* para os dispositivos móveis e uma interface para as aplicações web.

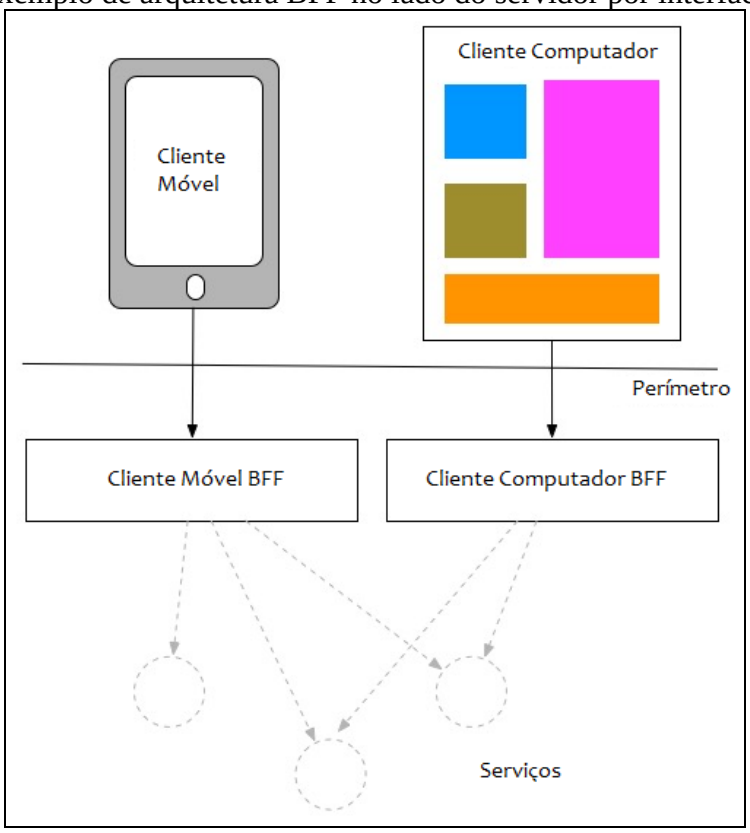
Newman (2015) aponta que as APIs tradicionais têm o propósito de servirem para uso geral. Essa estratégia funciona muito bem para aplicações web de computador e deixam flexível para que a aplicação que fará uso das APIs possa escolher como irá manipular os recursos. Quando se fala de dispositivos móveis, nem sempre essa estratégia será boa. Dispositivos móveis geralmente possuem menos área de tela, isso significa que podem exibir menos informações, devem reduzir a quantidade de conexões que precisam ser abertas para diminuir o consumo de bateria e consumir o mínimo de dados possível.

Tendo tudo isso em vista, é notável que dispositivos móveis possuem necessidades e propósitos diferentes das aplicações web para computadores, necessitando de APIs diferentes para atender as necessidades.

Para atender a situação de manter APIs tão diferentes para mesma aplicação, Newman (2015) propõe a utilização da arquitetura BFF (Backend for Frontends). A arquitetura BFF define que deva existir um *backend* para cada interface de usuário que necessite uma experiência diferente de utilização da API, como o caso dos dispositivos móveis e aplicações web para computadores.

Na Figura 1 é possível observar que existe um *backend* para cada interface de usuário, um para o cliente móvel e outra para o cliente computador. Os BFFs podem atuar como um *middleware*, ou seja, os clientes não se comunicam diretamente com os serviços, fazem isso sempre através do seu respectivo BFF.

Figura 1 – Exemplo de arquitetura BFF no lado do servidor por interface de usuário



Fonte: Newman (2015, p. 1).

Newman (2015) também conta que um BFF não precisa ser somente um tradutor de APIs. Um BFF pode abstrair diversas solicitações em uma, bem como executá-las em paralelo para otimizar o tempo da solicitação. Um exemplo que pode ser aplicado é um *e-commerce*. Pode-se supor que um usuário deseja consultar os itens da sua lista de desejo com a quantidade em estoque e preço do produto. Além disso, considerar que exista um serviço para obter a lista de desejo, um serviço para manipular o estoque e um serviço para controlar o preço, ou seja, será necessário obter dados de três fontes distintas. Um BFF pode realizar em paralelo essas operações de consulta em três serviços distintos enquanto o cliente precisou realizar somente uma solicitação ao BFF. Essa estratégia consome menos tempo e recurso do cliente.

2.2 SISTEMAS DISTRIBUÍDOS

Segundo Tanenbaum e Steen (2008), em meados de 1980 ocorreram dois avanços tecnológicos que revolucionariam a computação. O primeiro foi a evolução de microcontroladores de maior capacidade que foram evoluindo pouco a pouco. “De início, eram máquinas de 8 bits, mas logo se tornaram comuns CPUs de 16, 32 e 64 bits.” (TANENBAUM; STEEN, 2008, p. 1). Tanenbaum e Steen (2008) também relatam que com o

passar do tempo chegavam à capacidade de processamento de um *mainframe* por um preço muito menor. O segundo avanço foi o desenvolvimento das redes de computadores de alta velocidade e o surgimento das redes locais, as LANs, que permitiram que centenas de computadores próximos pudessem trocar informações na velocidade de alguns microssegundos. O surgimento dessas duas tecnologias possibilitou o desenvolvimento de um modelo de computação de altíssima capacidade de processamento conectado em uma rede de alta velocidade, os sistemas distribuídos.

A definição de um sistema distribuído é "um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente." (TANENBAUM; STEEN, 2008, p. 1). Existem alguns aspectos importantes dentro da definição: um sistema distribuído consiste em componentes autônomos, isto é, totalmente independentes e que não possuem dependência direta entre eles; os usuários (programas, aplicações ou pessoas) devem achar que tratam com um sistema único, ou seja, os componentes de um sistema distribuídos precisam colaborar entre si de alguma forma. Em princípio, nenhuma das premissas leva em consideração o tipo de computador ou a forma que irão se comunicar. Os sistemas podem ser executados em computadores muito potentes ou até mesmo em computadores muito fracos, bem como podem optar por diferentes formas de comunicação.

Tanenbaum e Steen (2008) descrevem que as diferenças na comunicação interna entre os diferentes sistemas e a organização interna dos sistemas deve ser ocultada para o usuário, indo de encontro à premissa de que os sistemas devem parecer um só para o usuário final. "Uma outra característica importante é que usuários e aplicações podem interagir com um sistema distribuído de maneira consistente e uniforme, independentemente de onde a interação ocorra." (TANENBAUM; STEEN, 2008, p. 2).

Diferente de sistemas centralizados (monolitos), os sistemas distribuídos em tese têm a capacidade de aumentar a sua escala, o que é uma consequência direta de utilizar componentes independentes. Outra consequência da utilização de componentes independentes é a possibilidade de algumas partes do sistema estarem avariadas e algumas partes funcionando, conforme explicado por Tanenbaum e Steen (2008).

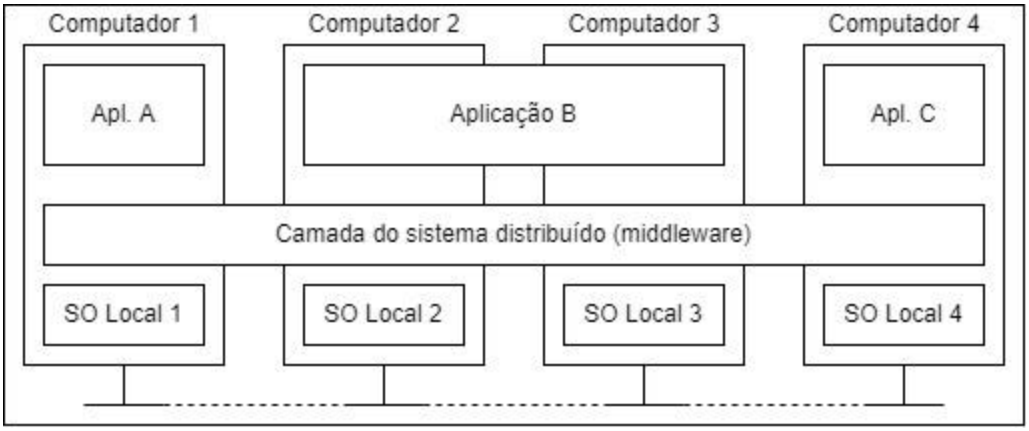
Segundo Tanenbaum e Steen (2008), em um ambiente de computação distribuída, por conveniência, é utilizado um modelo de comunicação assíncrona. Isto significa que um sistema A envia uma mensagem para um sistema B e não aguarda a resposta. Somente quando o sistema B puder processar a mensagem é que ele envia uma nova mensagem para o sistema A e o ciclo se repete: somente quando o sistema A puder processar é que a mensagem será processada. Este modelo é muito importante principalmente se tratando de escalabilidade em

ambientes geograficamente distantes ou de alta latência, onde a conexão pode falhar ou ser lenta, diferente de uma LAN. “[...] isso significa construir a aplicação requisitante de modo tal que ela use só comunicação assíncrona. Quando chega uma resposta, a aplicação é interrompida e um manipulador especial é chamado para concluir a requisição emitida anteriormente.” (TANENBAUM; STEEN, 2008, p. 7).

Tanenbaum e Steen (2008) ressaltam a importância da transparência para o usuário final ao comunicar com um sistema distribuído e descrevem uma forma muito utilizada de organizar o sistema de forma a torná-lo transparente. É comum que os sistemas sejam separados em camadas de tal forma que os sistemas fiquem numa camada mais alta e abaixo estejam a interface de comunicação e sistemas operacionais. Na camada de comunicação, um sistema distribuído pode ser chamado de *middleware*.

A Figura 2 “[...] mostra quatro computadores em rede e três aplicações, das quais a aplicação B é distribuída para os computadores 2 e 3. A mesma interface é oferecida a cada aplicação. O sistema distribuído proporciona os meios para que os componentes de uma única aplicação distribuída se comuniquem uns com os outros, mas também permite que diferentes aplicações se comuniquem. Ao mesmo tempo, ele oculta, do melhor e mais razoável modo possível, as diferenças em hardware e sistemas operacionais para cada aplicação.” (TANENBAUM; STEEN, 2008, p. 2).

Figura 2 – Exemplo de sistema distribuído organizado como middleware



Fonte: Tanenbaum e Steen (2008, p. 2).

2.2.1 Computação em nuvem

Segundo a Azure (2022), a computação em nuvem pode se resumir ao fornecimento de serviços distribuídos de computação de forma simples, flexível e sob demanda.

As plataformas de nuvem são flexíveis e rápidas, permitem que sejam contratados somente os recursos necessários por um determinado momento e liberam estes recursos quando eles não forem mais necessários. “[...] só paga pelos serviços individuais que precisar,

pelo tempo que os utilizar, sem a necessidade de contratos de longo prazo ou licenciamento complexo. [...] é semelhante à usada por serviços públicos, como água ou energia. Você paga apenas pelos serviços que utilizar e, quando parar de usá-los, não haverá custos adicionais nem taxas de cancelamento.” (AMAZON WEB SERVICES, 2022b, p. 1).

Azure (2022) descreve que as plataformas fornecessem serviços onde o valor pago é somente sobre do que foi utilizado, resultando em diminuição dos custos. Os recursos nas plataformas são flexíveis e possibilitam que os sistemas possam ser escalados de forma rápida, diferentemente de empresas que possuam sua própria estrutura de servidores. Tendo em vista uma empresa que possui a sua estrutura própria de servidores, mesmo que nem todos os recursos dos servidores estejam sendo consumidos, os servidores continuaram ligados e ociosos, gerando custos desnecessários à empresa. Outra situação é quando os sistemas precisam escalar além da capacidade dos servidores, onde é impossível aumentar a escala sem adicionar novos servidores, gerar ainda mais custos em momentos de ociosidade e lentidão na resposta da necessidade de escala.

A computação em nuvem elimina a necessidade de as empresas terem todos os custos de manter os servidores, rede, energia, configuração, além de especialistas para o gerenciamento da infraestrutura, conforme apontado pela Azure (2022). As plataformas em nuvem removem a necessidade de gerenciar toda uma pilha de hardware.

Além disso tudo, as plataformas conseguem resumir sistemas complexos em interfaces de simples configuração que já iniciam com segurança, escalabilidade e autogerenciamento. Um exemplo de serviço que a Amazon Web Service (2022a) fornece são os sistemas de Load Balancer, ou Balanceador de Carga. Este é um serviço responsável por servir como um ponto de entrada comum para outros sistemas e cuidam de distribuir as solicitações para as diversas instâncias daqueles sistemas. A Amazon Web Service (2022a) informa que pode fornecer esse serviço de balanceamento de carga com alta disponibilidade e com camadas de segurança como TLS ou HTTPS em poucos cliques, removendo toda a complexidade de gerenciamento desse sistema.

Atualmente há várias plataformas de computação em nuvem disponíveis no mercado, entre elas as mais conhecidas mundialmente são a Azure, a AWS e a Google Cloud.

2.3 MIDDLEWARE

O conceito de um *middleware* é vasto dentro do mundo da computação e possui diversas implementações e utilizações nos mais diversos componentes da computação. Existem *middlewares* de API, *middlewares* de comunicação com bancos de dados,

middlewares internos nas aplicações como estruturas que abstraem a complexidade de diversos componentes ou até mesmo uma integração com *hardware* em uma interface simples.

Um *middleware* funciona "[...] de forma essencial como uma camada oculta de tradução, [...] permite a comunicação e o gerenciamento de dados para aplicativos distribuídos." (AZURE, 2021, p. 1). Os *middlewares* também podem fornecer recursos e serviços comuns a aplicações, como autenticação, mensagerias, serviços de aplicações e APIs, conforme descrito por Red Hat (2018). Azure (2021) complementa que ele pode ser chamado "encanamento" pois ele conecta aplicações para que os dados sejam transportados através de um "cano", que seria o próprio *middleware*.

Red Hat (2018) conta que o desenvolvimento nativo em nuvem possui inúmeros benefícios, mas que também aumenta a complexidade devido à grande variedade de tecnologias e arquiteturas existentes nas nuvens. As empresas veem no *middleware* uma forma de reduzir a complexidade e aumentar a velocidade do desenvolvimento, uma vez que o *middleware* "[...] é capaz de oferecer suporte a ambientes de aplicação que funcionam de maneira estável e consistente em uma plataforma altamente distribuída." (RED HAT, 2018, p. 1). Isso está diretamente ligado aos diversos serviços que as plataformas em nuvem oferecem, serviços prontos que podem ser configurados rapidamente e que abstraem praticamente toda a complexidade do gerenciamento dos recursos desses serviços.

Segundo a Red Hat (2018), o *middleware* não é aplicado somente a um tipo de aplicação ou situação, ele engloba de tudo, desde aplicações web a ferramentas. Algumas das aplicações frequentes dos *middlewares* são:

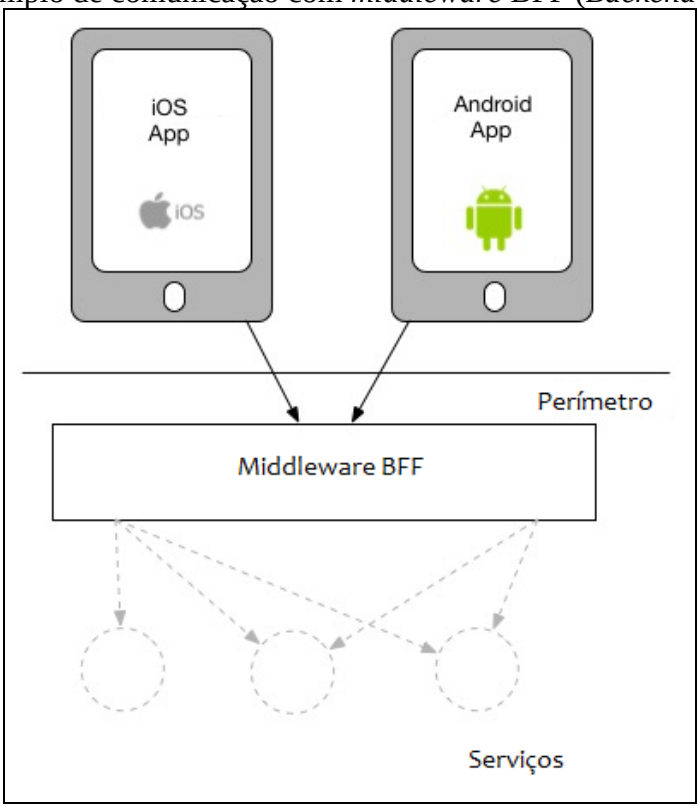
a) desenvolvimento de APIs, possibilitando a conexão de diversos produtos e serviços através de uma interface única;
b) transmissão de dados assíncrona, abstraindo do cliente a replicação do dado para diferentes zonas;
c) otimização de aplicações existentes, podendo transformar aplicações monolíticas em aplicações nativas em nuvem.

Os *middlewares* de API geralmente funcionam como uma entrada comum de comunicação, tratando e interpretando essa entrada para levar o dado para a aplicação correspondente e devolvendo o dado para o solicitante de acordo com o que a aplicação responder.

Na Figura 3, dois dispositivos móveis de sistemas operacionais diferentes estão se comunicando com diversos serviços através de um *middleware*, ou seja, para os dispositivos é

transparente a forma que os serviços se comunicam e como chegar neles. Toda a comunicação é realizada somente com o *middleware* e ele é responsável por repassá-la para os serviços, bem como coletar a resposta e entregar aos dispositivos. Por se tratar de sistemas distintos, é possível que o *middleware* precise tratar a comunicação de forma diferente para cada dispositivo.

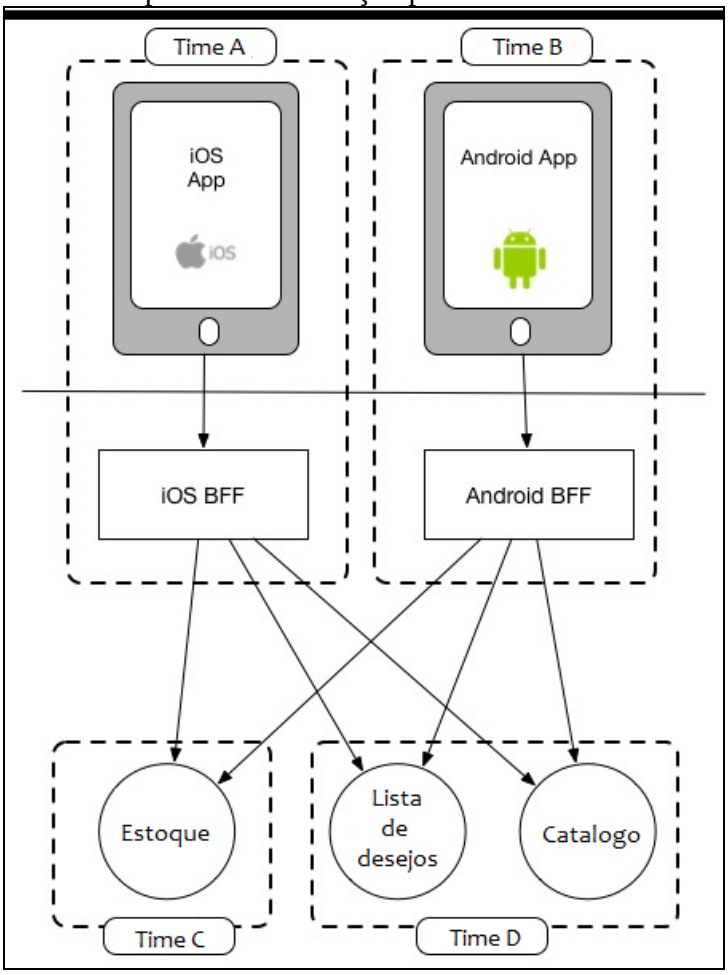
Figura 3 – Exemplo de comunicação com *middleware* BFF (*Backend For Frontend*)



Fonte: Newman (2015, p. 1).

Os *middlewares* podem realizar mais de uma ação para uma entrada, não necessariamente irão somente repassar o conteúdo para os serviços. Um exemplo pode ser encontrado nos BFFs, que pode executar diversas chamadas para as aplicações em paralelo, juntar tudo isso em uma única mensagem e devolver ao solicitante, conforme a Figura 4 demonstra.

Figura 4 – Exemplo de comunicação paralela com *middleware* BFF



Fonte: Newman (2015, p. 1).

Na Figura 4 encontram-se dois *middlewares* do tipo BFF, um para iOS e outro para Android. É possível notar que quando o Android ou o iOS solicitam para o seu *middleware* uma informação, três serviços são chamados ao mesmo tempo pelo *middleware*. A figura também contém a demarcação de times, indicando que são serviços que podem ser desenvolvidos por equipes separadas.

Os balanceadores de carga também podem ser chamados de *middleware* pois eles exercem esse papel de “encanamento”, recebendo as solicitações e distribuindo para várias instâncias de uma aplicação a fim de dividir a carga entre elas.

3 DESENVOLVIMENTO DO *MIDDLEWARE*

Neste capítulo serão abordadas as etapas de desenvolvimento do *middleware*. A seção 3.1 apresenta os requisitos funcionais e não funcionais do *middleware* desenvolvido. Na seção 3.2 são apresentados o diagrama de casos de uso, o diagrama entidade relacionamento, a arquitetura da aplicação e os diagramas de atividades, especificando e detalhando o funcionamento do sistema. A seção 3.3 detalha a implementação do *middleware*, destacando os principais pontos e traz exemplos para sua utilização. Na seção 3.4 é apresentada a análise dos resultados obtidos nesse trabalho.

3.1 REQUISITOS

Desenvolver aplicações web para dispositivos móveis é uma forma rápida e com a capacidade de reaproveitar trechos importantes das aplicações, tornando uma opção muito interessante a ser escolhida. Com a revolução móvel que está acontecendo, está havendo cada vez mais a utilização e demanda por aplicativos, exigindo cada vez mais servidores com maior capacidade de escala para comportar essa demanda crescente. A construção de aplicações distribuídas é uma grande aliada para comportar essa demanda cada vez mais crescente. Por estes motivos, este trabalho tem o objetivo de disponibilizar uma aplicação *middleware* escalável para comunicação de aplicações web e aplicações em um ambiente distribuído.

Os requisitos deste trabalho são:

a) o *middleware* deve permitir a troca de mensagens entre a aplicação web e um ambiente distribuídos (Requisito funcional – RF01);
b) o cliente deve poder se registrar no *middleware* (RF02);
c) o cliente deve receber as mensagens pendentes armazenadas no *middleware* logo após a conexão (RF03);
d) o serviço deve receber as mensagens pendentes armazenadas no *middleware* logo após a conexão (RF04);
e) as mensagens devem contar com um tempo de expiração enviado junto a mensagem (RF05);
f) o *middleware* deve armazenar as mensagens até serem enviadas ou expiradas (RF06);
g) o *middleware* deve contar com uma estrutura de resposta de mensagens (RF07);
h) o *middleware* deve esperar uma confirmação do recebimento da mensagem (RF08);

i) o *middleware* deve confirmar o recebimento da mensagem a quem enviou (RF09);
j) a entrega das mensagens precisa ser resiliente, ou seja, a mensagem deverá ser enviada até mesmo se o cliente estiver incomunicável no momento do envio (a entrega da mensagem ocorrerá quando o cliente se conectar novamente no *middleware*) (Requisito não funcional - RNF01);
k) precisa atender a um contexto de sistemas distribuídos (RNF02);
l) a comunicação entre cliente e *middleware* será através do protocolo WebSocket (RNF03);
m) a comunicação entre serviços e *middleware* será através de mensageria (RNF04);
n) o *middleware* fará uso do Redis para *cache* distribuído (RNF05);
o) o *middleware* fará uso do MongoDB como banco de dados (RNF06);
p) o *middleware* deve ser capaz de escalar horizontalmente sempre que a demanda chegar próxima da capacidade atual (RNF07);
q) o *middleware* deve reduzir a escala sempre que a demanda for muito menor que a capacidade atual (RNF08);
r) o *middleware* deve ser hospedado na AWS (RNF09).

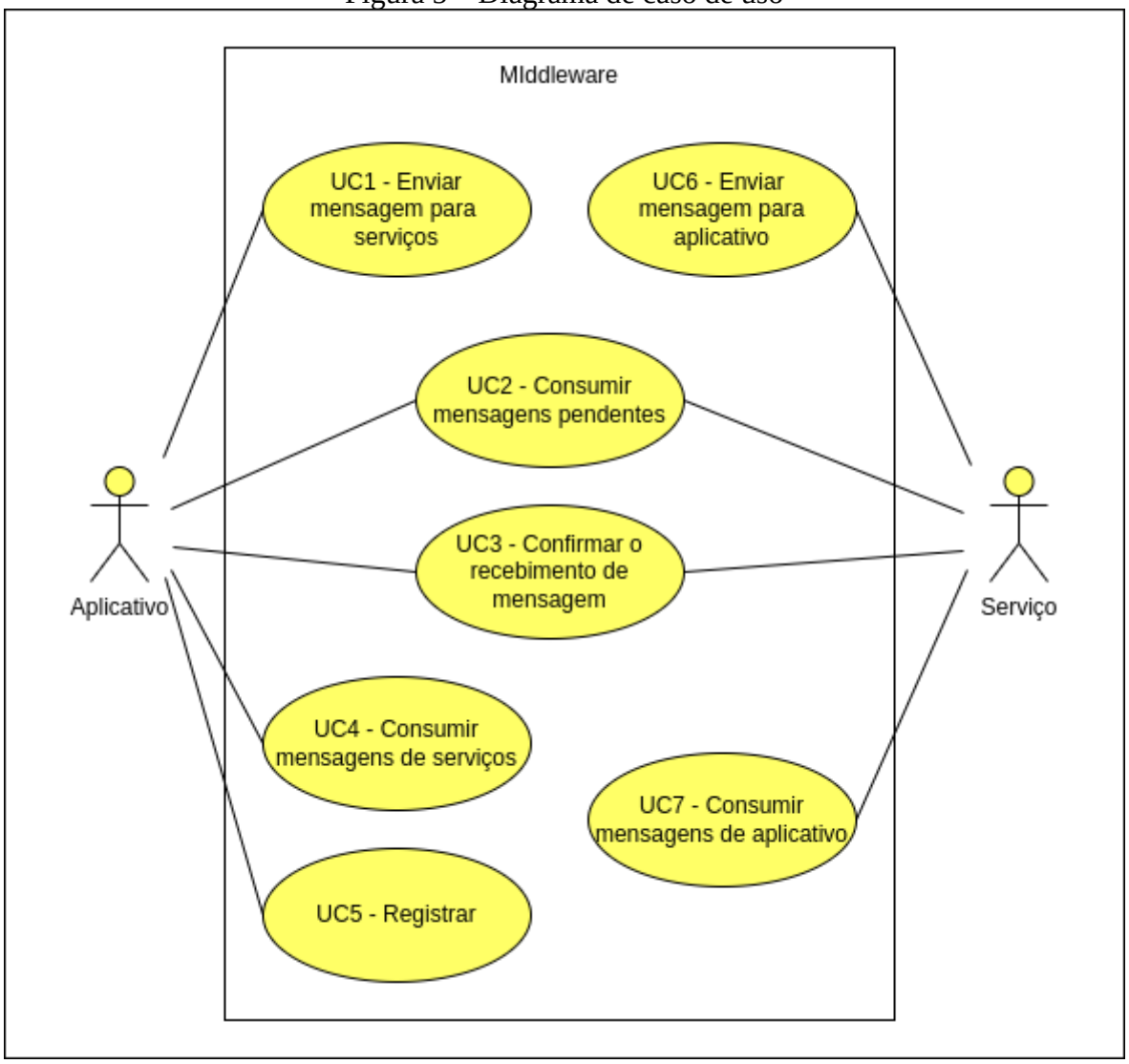
3.2 ESPECIFICAÇÃO

Nesta seção é apresentada a especificação do *middleware* desenvolvido. É apresentado o diagrama de casos de uso, o diagrama entidade relacionamento, a arquitetura da aplicação e os diagramas de atividades. Todos os diagramas foram feitos através da ferramenta Draw.io.

3.2.1 Diagrama de caso de uso

O diagrama de caso de uso da Figura 5 foi elaborado com base nos requisitos elencados na seção anterior.

Figura 5 – Diagrama de caso de uso



Fonte: elaborado pelo autor.

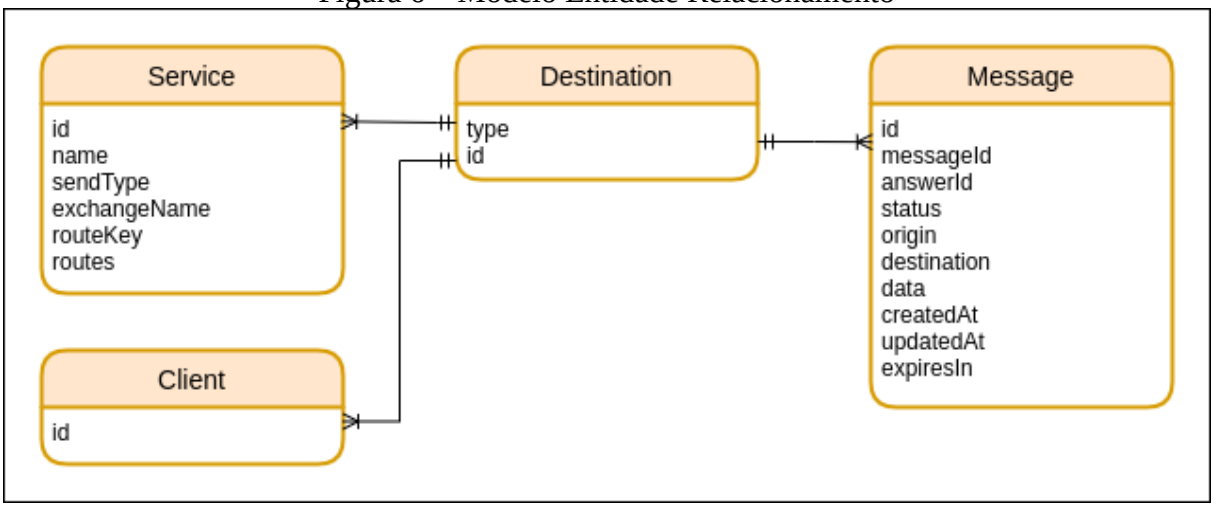
Os casos de uso são apresentados na Figura 5, no caso de uso `UC1 - Enviar mensagem para serviços`, o aplicativo envia uma mensagem para o *middleware* com destino em um serviço. No caso de uso `UC2 - Consumir mensagens pendentes`, tanto o aplicativo quanto o serviço recebem as mensagens que estão pendentes para eles, isto é, recebem mensagens que foram enviadas enquanto estavam desconectados. No caso de uso `UC3 - Confirmar o recebimento de mensagens`, tanto o aplicativo quanto o serviço confirmam o recebimento de uma mensagem; somente após a confirmação o *middleware* remove a pendência da mensagem. Este recurso permite que o recebedor da mensagem controle quando confirmar a mensagem, possibilitando que rotinas disparadas pelo recebimento de mensagens possam ser interrompidas abruptamente e serem reiniciadas quando as mensagens forem recebidas novamente, uma vez que o *middleware* só remove a pendência da mensagem quando o recebedor confirma o recebimento, e a mensagem pendente

sempre é reenviada através do caso de uso `UC2`. No caso de uso `UC4 – Consumir mensagens de serviços`, o aplicativo consome do *middleware* mensagens que foram enviadas por serviços para o aplicativo. No caso de uso `UC5 – Registrar`, o aplicativo se registra em uma instância no *middleware*, indicando para o *middleware* onde o aplicativo está e possibilitando a comunicação. No caso de uso `UC-6 – Enviar mensagem para aplicativo`, o serviço envia uma mensagem para o *middleware* com destino a um aplicativo. No caso de uso `UC-7 Consumir mensagens de aplicativo`, o serviço consome do *middleware* mensagens que foram enviadas por aplicativos para o serviço.

3.2.2 Modelo Entidade Relacionamento

O *middleware* faz uso do MongoDB como banco de dados e por se tratar de um banco de dados de documentos, não existe relacionamento entre as entidades, ou seja, não existe chave estrangeira. O Modelo Entidade Relacionamento (MER) na Figura 6 representa as entidades do *middleware*.

Figura 6 – Modelo Entidade Relacionamento



Fonte: elaborado pelo autor.

A entidade `Service` representa um serviço. Esta entidade guarda quais são as rotas desse serviço e as informações necessárias para o envio das mensagens ao serviço que são a forma de envio (*Queue* ou *Exchange*), o nome da *Exchange* e o nome da fila. A rota é um campo obrigatório a ser informado pelo aplicativo ao enviar uma mensagem para o *middleware*, pois é através dela que o *middleware* irá identificar o destino.

A entidade `Client` representa um cliente que sempre será uma instalação de aplicativo se comunicando com o *middleware*.

A entidade `Message` representa uma mensagem que o *middleware* recebeu, tanto de um serviço quanto de um aplicativo. Através do campo `origin` é possível identificar se é

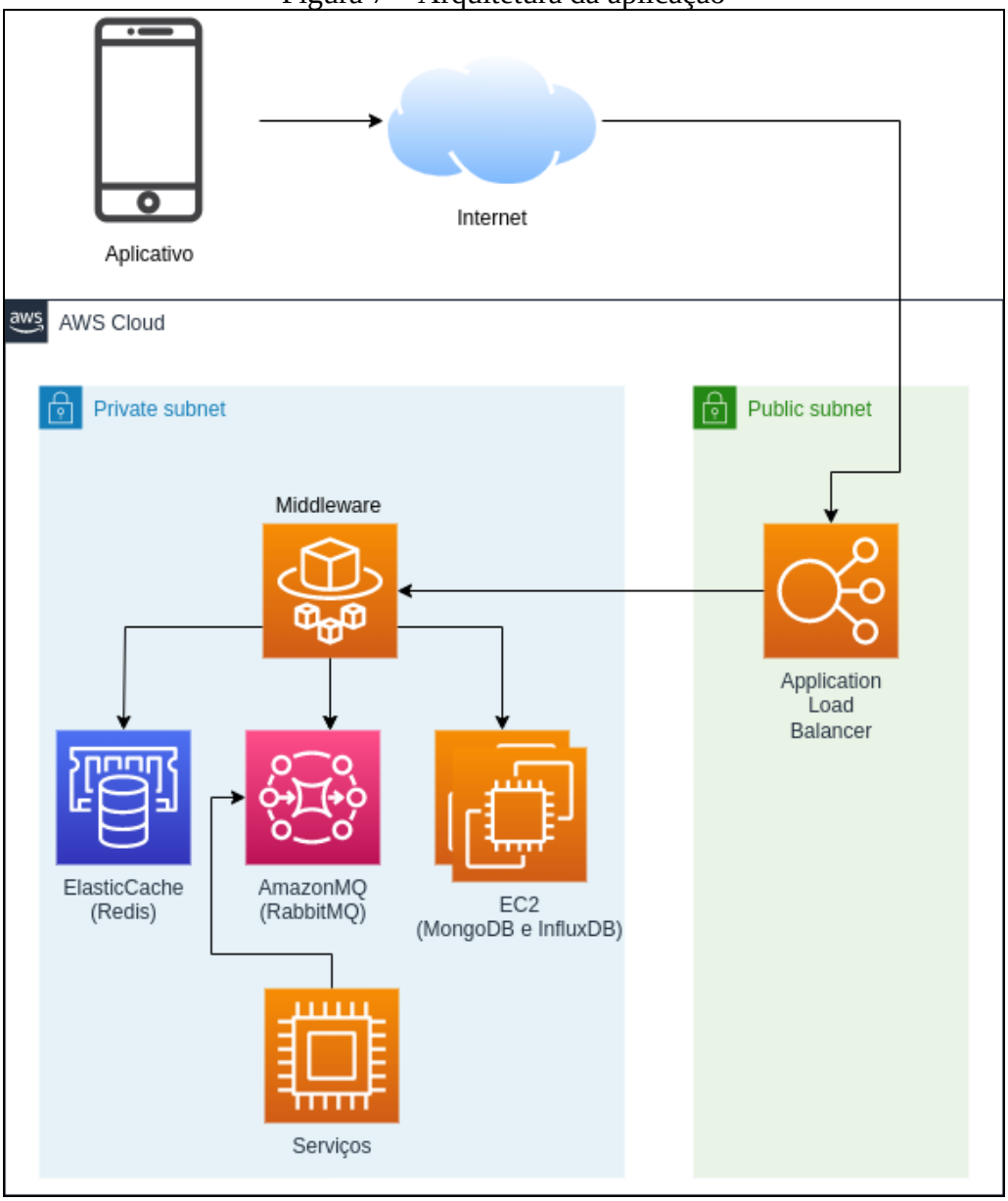
mensagem de um serviço ou de aplicativo. Esta entidade guarda o ID da mensagem gerado por quem enviou a mensagem, um ID de resposta de mensagem, a origem (serviço ou aplicativo), o destino (serviço ou aplicativo), os dados da mensagem e o tempo de expiração da mensagem. O campo `data` é um objeto qualquer que guarda os dados da mensagem, dados estes que podem ser qualquer objeto JSON e se trata da mensagem que está sendo trafegada entre o aplicativo e o serviço. É como se a entidade `Message` fosse um encapsulamento carregando a mensagem real que está dentro do campo `data`.

A entidade `Destination` representa o destino de uma mensagem. Ela representa os atributos `origin` e `destination` da entidade `Message`. O campo `type` identifica se é um serviço ou um aplicativo e o campo `id` guarda o identificador do `Service` ou do `Client`.

3.2.3 Arquitetura da aplicação

O *middleware* foi hospedado na AWS e foram utilizados diversos recursos da própria AWS para sua implementação. A Figura 7 apresenta uma visão geral de como o *middleware* foi organizado e quais serviços da AWS foram utilizados.

Figura 7 – Arquitetura da aplicação



Fonte: elaborado pelo autor.

Na Figura 7 é possível ver que diversos serviços da AWS são utilizados. As conexões dos aplicativos chegam por um Application Load Balancer que tem o papel de distribuir as conexões entre as instâncias do *middleware*. O *middleware* está sendo executado dentro do Fargate, o que já deixa evidente que o *middleware* roda em um *container* Docker e que existirão diversas instâncias em execução simultaneamente. O *middleware* faz uso dos serviços ElasticCache (Redis), AmazonMQ (RabbitMQ), EC2 com o MongoDB e EC2 com o InfluxDB instalado. Além dos serviços, a rede é dividida em duas Subnets: uma pública e uma privada. A Subnet pública é a que possui saída para internet e os recursos nela possuem um IP na internet. Já a Subnet privada possui saída para internet através de um NAT e os recursos nela não possuem um IP na internet. Essa divisão é importante para melhorar a organização

dos recursos e dificultar qualquer acesso indevido aos recursos internos, sendo possível acessá-los somente pelo balanceador de carga.

O *middleware* faz uso do ElasticCache para registro dos clientes, ou seja, guardar a informação de quais clientes estão conectados em cada instância do *middleware*. Isto é necessário pois caso seja necessário enviar uma mensagem para um cliente, a mensagem é enviada na realidade para a instância do *middleware* onde o cliente está conectado.

O AmazonMQ é necessário para a comunicação do *middleware* com os serviços, bem como para comunicação entre as instâncias do *middleware*. Cada instância do *middleware* cria uma fila exclusiva de entrada no AmazonMQ para que ela possa receber mensagens. Os serviços também fazem uso do AmazonMQ pois é através dele que a comunicação entre eles acontece.

O MongoDB é utilizado para armazenar o cadastro de serviços e as mensagens que estão sendo trafegadas pelo *middleware*.

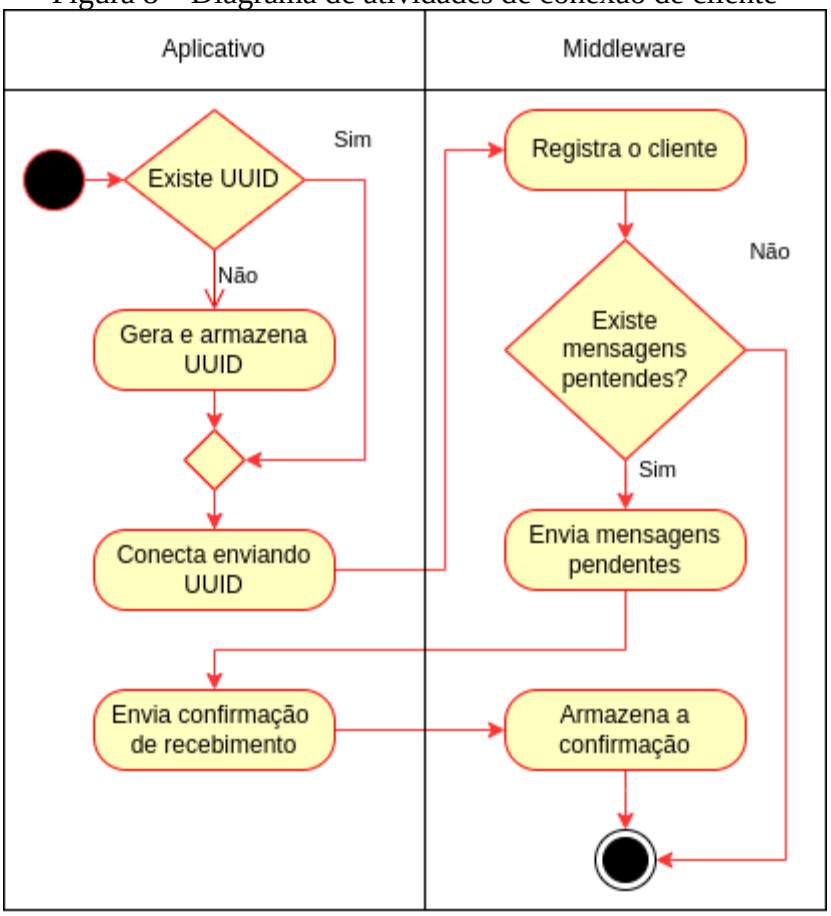
O InfluxDB é utilizado para armazenar as métricas que o *middleware* gera durante a sua execução.

3.2.4 Diagramas de atividades

Esta seção contém diversos diagramas de atividades para especificar o fluxo de atividades dentro do *middleware*.

A Figura 8 apresenta o diagrama de atividades da conexão de um cliente. Antes da conexão, o cliente verifica se já gerou um UUID e o gera caso ainda não exista. O UUID é enviado junto da requisição de conexão. O *middleware* recebe o UUID e registra o cliente naquela instância do *middleware* de forma que qualquer instância do *middleware* possa saber que o cliente está registrado e é possível comunicar. O *middleware* verifica no banco de dados se existem mensagens pendentes para enviar ao cliente e realiza o envio caso existam. O cliente recebe as mensagens pendentes, processa e confirma o recebimento de cada uma delas. O *middleware* armazena a confirmação, tirando a pendência da mensagem.

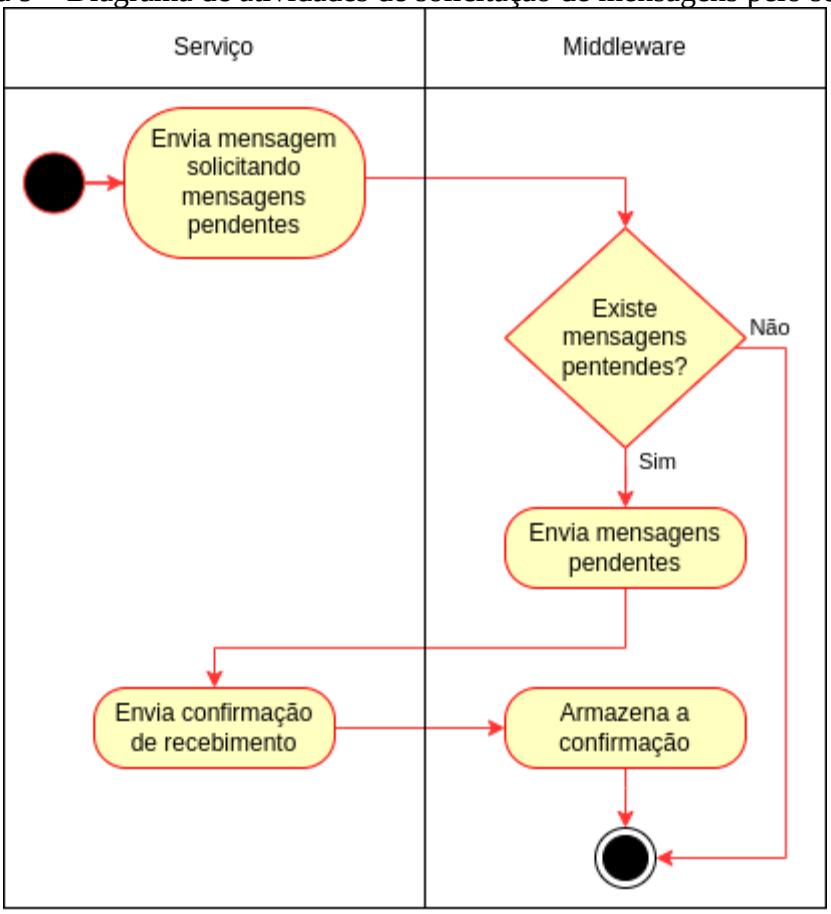
Figura 8 – Diagrama de atividades de conexão de cliente



Fonte: elaborado pelo autor.

A Figura 9 apresenta o diagrama de atividades da solicitação de mensagens pelo serviço. Quando uma instância nova de um serviço iniciar, o serviço deve solicitar ao *middleware* as mensagens que estão pendentes para o serviço. O serviço realiza isso enviando uma mensagem ao *middleware* solicitando as mensagens pendentes. O *middleware* verifica se existem mensagens pendentes para enviar ao serviço e realiza o envio caso existam. O serviço recebe as mensagens pendentes, processa e confirma o recebimento de cada uma delas. O *middleware* armazena a confirmação, tirando a pendência da mensagem.

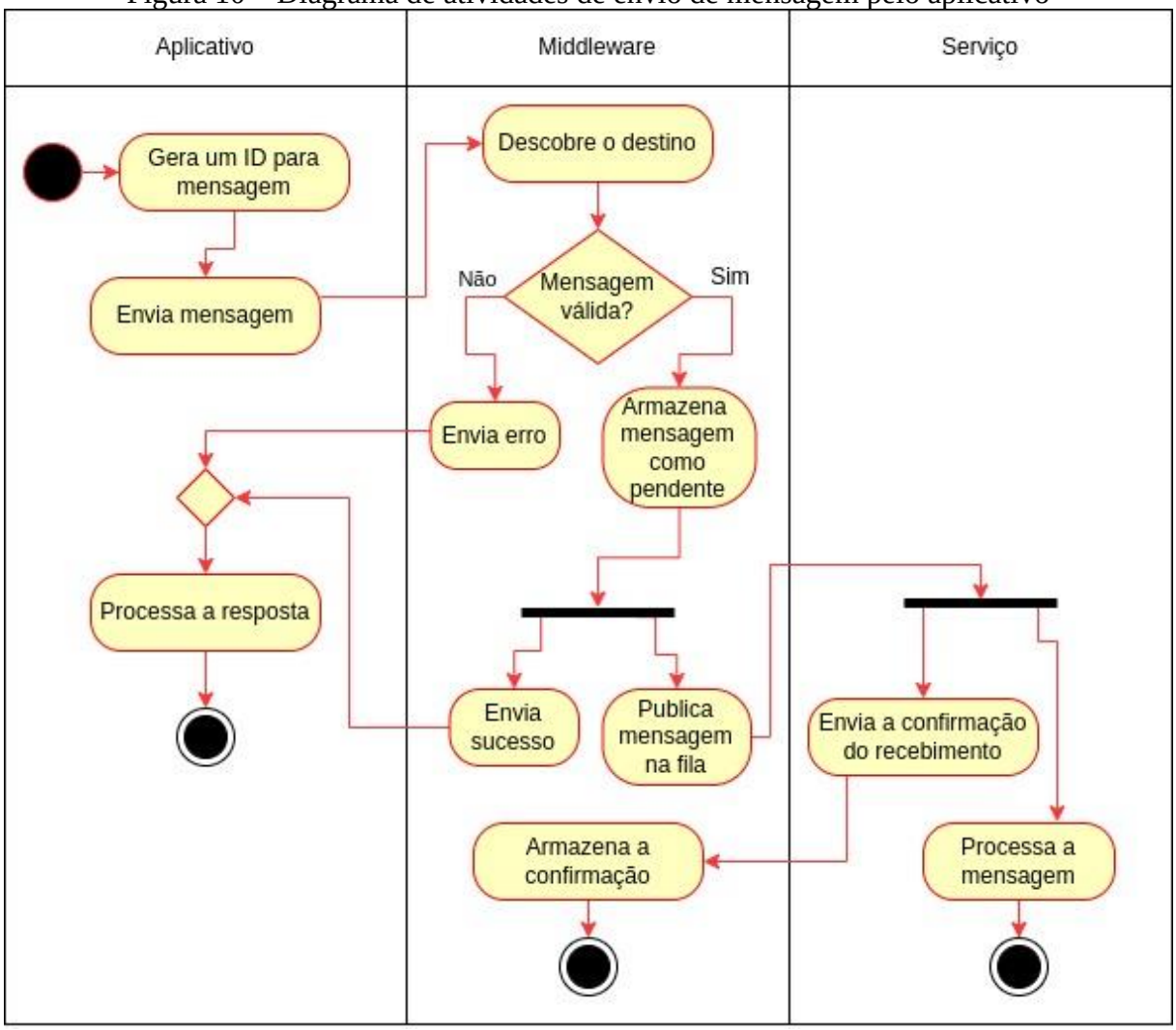
Figura 9 – Diagrama de atividades de solicitação de mensagens pelo serviço



Fonte: elaborado pelo autor.

A Figura 10 apresenta o diagrama de atividades de envio de mensagens pelo aplicativo. Primeiro o aplicativo gera um UUID para a mensagem e realiza o envio para o *middleware*. O *middleware* processa a mensagem e descobre o destino dela. Caso a mensagem não seja válida, o *middleware* envia uma resposta de erro para o aplicativo. Caso a mensagem seja válida, a mensagem é armazenada como pendente, e em paralelo é respondido para o cliente que a mensagem foi enviada com sucesso e a mensagem é repassada para o serviço. O serviço confirma o recebimento ao *middleware* e processa a mensagem recebida.

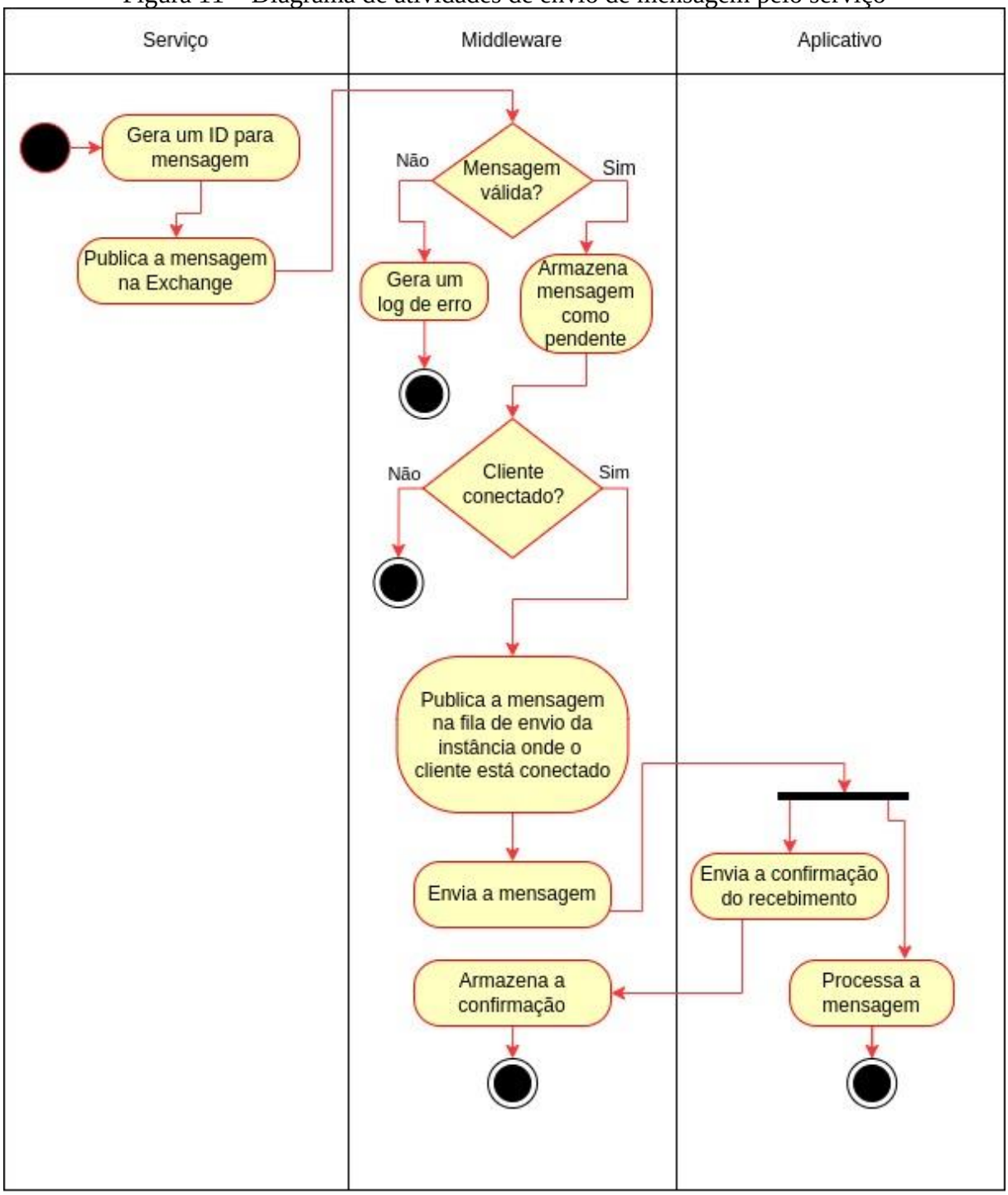
Figura 10 – Diagrama de atividades de envio de mensagem pelo aplicativo



Fonte: elaborado pelo autor.

A Figura 11 apresenta o diagrama de atividades de envio de mensagem pelo serviço. Primeiro o serviço gera um UUID de mensagem e publica na *Exchange* de notificações do *middleware*. O *middleware* verifica se a mensagem é válida. Caso a mensagem não seja válida, é gerado um log de erro e o processo é encerrado. Caso a mensagem seja válida, a mensagem é armazenada como pendente e é verificado se o cliente está conectado. Caso o cliente não esteja conectado, o processo é encerrado. Caso o cliente esteja conectado, a mensagem é publicada na fila de envio da instância do *middleware* onde o cliente está conectado. A instância recebe a mensagem e repassa para o cliente. O aplicativo em paralelo responde o *middleware* com uma confirmação de recebimento e processa a mensagem recebida.

Figura 11 – Diagrama de atividades de envio de mensagem pelo serviço



Fonte: elaborado pelo autor.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

O *middleware* foi desenvolvido em Java 11 utilizando o *framework* Quarkus na versão 2.8.0 e pronto para ser executado em *container* Docker. O desenvolvimento ocorreu utilizando o padrão de arquitetura MVC (Model-View-Controller), mais o Repository e utilizando algumas bibliotecas como Lombok, para agilizar alguns itens do desenvolvimento, também foi utilizado Quartz para gerenciamento de tarefas agendadas (Jobs) e Hibernate com Panache para persistência.

Para persistência das mensagens e serviços foi utilizado o banco de dados de documentos MongoDB. Para comunicação entre os serviços e o próprio *middleware* foi utilizado o RabbitMQ através do protocolo AMQT. Para comunicação com os clientes foi utilizado o protocolo WebSocket. Para as métricas da aplicação, foi implementada uma rotina que publica as métricas que a própria aplicação gera no InfluxDB. Ele foi hospedado na AWS e foram utilizados os seguintes serviços da AWS para o funcionamento dele: AmazonMQ como *broker* de mensagens, ElasticCache como *cache* distribuído, EC2 para hospedar uma instalação do MongoDB e uma instalação do InfluxDB, e Fargate para executar os *containers* do *middleware*.

Por se tratar de um *middleware* de comunicação com o objetivo de escalabilidade, foram desenvolvidas aplicações sem interface gráfica para simular um teste de carga e nenhuma aplicação interativa foi desenvolvida. Todas essas aplicações de teste de carga foram desenvolvidas em Java 11.

3.3.1.1 Comunicação *middleware-middleware*

Quando uma instância de *middleware* necessita enviar uma mensagem para um cliente, ela precisa fazer isso enviando uma mensagem para a instância do *middleware* onde o cliente está conectado. Para que isso seja possível, quando uma instância do *middleware* inicia, é gerado um id único e criam-se duas filas exclusivas no RabbitMQ: uma fila se registra numa *Exchange* de notificação utilizada para recebimento de mensagens dos serviços e a outra fila se registra numa *Exchange* que recebe as mensagens para enviar aos clientes.

Sempre que um serviço qualquer envia uma mensagem para um aplicativo através do *middleware*, a mensagem é persistida no banco de dados como pendente, é verificado no Redis qual instância do *middleware* o cliente está conectado e publica-se a mensagem na fila exclusiva daquela instância. A instância que recebe a mensagem checa se o cliente ainda está conectado e realiza o envio através do WebSocket. Caso o cliente não esteja mais conectado

no momento do envio, a mensagem permanece como pendente no banco de dados até que o cliente se conecte novamente.

3.3.1.2 Cadastro de serviços e rotas

O cadastro de serviços é realizado através de uma API HTTP. Para o *middleware* é importante saber o nome do serviço, qual o nome da *Exchange*, qual o nome da fila e quais são as rotas para um determinado serviço. Este cadastro é necessário para que o *middleware* saiba para onde enviar uma mensagem enviada pelo aplicativo. O destino da mensagem é encontrado através da rota que é enviada na mensagem que precisa coincidir com a rota de algum serviço. Uma rota deve ser única por serviço, pois é com base nela que o *middleware* sabe para onde enviar a mensagem.

Existe uma rota fixa que não possui cadastro que é a rota do próprio *middleware*. Caso o aplicativo deseje enviar uma mensagem para o *middleware*, ele deve utilizar `middleware` como rota.

3.3.1.3 Registro de cliente

Sempre que um cliente se conecta numa instância do *middleware*, ele é obrigado a enviar o seu identificador no ato da conexão através do *query param* `id`. Isto permite que a instância grave uma chave no Redis indicando que o cliente em questão está conectado em tal instância, possibilitando que qualquer outra instância possa descobrir para onde enviar uma mensagem para o cliente. O registro do cliente dura somente 30s. Este tempo é o suficiente pois este tempo é atualizado a cada iteração do cliente com o *middleware*.

Para que a conexão não seja encerrada a cada 30s de inatividade, o *middleware* implementou o recurso de ping-pong do protocolo WebSocket. A cada 20s o *middleware* dispara um *ping* para o cliente. Caso o cliente não responda em até 5s, o *middleware* interpreta isto como um *timeout* e encerra a conexão. Caso o cliente responda dentro dos 5s, o registro do cliente é atualizado e ele ganha mais 30s de conexão. O Quadro 1 apresenta o trecho de código que realiza o *ping*.

Quadro 1 – *Job* interno do *middleware* que realiza o *ping*

```
01 @Scheduled(
02     every = Config.WEBSOCKET_CHECK_PING_JOB_INTERVAL,
03     identity = "WebsocketPing"
04 )
05 void ping() {
06     Set<WebsocketSession> sessions = this.service.getSesions();
07     sessions.parallelStream().forEach(websocketSession -> {
08         Session session = websocketSession.getSession();
09         if (session.isOpen()) {
10             try {
11                 if (isPongLate(websocketSession)) {
12                     this.logger.info("Clossing session " +
    session.getId() + " because client does not respond");
13                     session.close(
14                         new CloseReason(
15                             CloseCodes.NORMAL_CLOSURE,
16                             "Client does not respond"
17                         )
18                     );
19 
20                 } else if (isNeedSendPing(websocketSession)) {
21                     session.getBasicRemote().sendPing(
22                         ByteBuffer.allocate(0)
23                     );
24                     websocketSession.setLastPing(Instant.now());
25                 }
26             } catch (IOException e) {
27                 this.logger.error(e.getMessage(), e);
28             }
29         }
30     });
31 }
```

Fonte: elaborado pelo autor.

O quadro 1 mostra o *job* de que realiza o *ping*. Este job é executado a cada 10s e todas as conexões são verificadas naquele momento. Com essa abordagem, nem todas as conexões estão prontas para o *ping* ser enviado. Na linha 20 é verificado se é necessário enviar um *ping* ao cliente (se já atingiu 20s de inatividade) e o *ping* é enviado caso necessário. Na linha 11 é verificado se o *pong* está atrasado (se já atingiu 5s de inatividade após o envio do *ping*) e a conexão é encerrada caso isso tenha acontecido. O Quadro 2 mostra o trecho de código onde o *middleware* recebe o *pong* enviado pelo cliente e atualiza o registro.

Quadro 2 – Recebimento do *pong* e atualização do registro

```
01 public Uni<Void> onPongMessage(WebsocketPongEvent event) {
02     WebsocketSession session = getWebsocketSession(event.getId());
03     session.setLastPong(Instant.now());
04 
05     this.clientService.updateRegistration(session);
06 
07     return Uni.createFrom().voidItem();
08 }
```

Fonte: elaborado pelo autor.

Sempre que uma conexão WebSocket é encerrada, o *middleware* remove a chave do cliente Redis, ou seja, o cliente perde o registro.

3.3.1.4 Envio de mensagem pelo aplicativo

A conexão do aplicativo com o *middleware* é estabelecida através do protocolo WebSocket e todos os dados trafegados são objetos JSON. O aplicativo pode enviar qualquer mensagem que ele desejar para um serviço, desde que a mensagem seja encapsulada na estrutura de mensagem do *middleware*. Essa estrutura de mensagem é composta por um objeto JSON com os campos: `id`, `answerId`, `route`, `data` e `ttl`. O campo `id` deve ser preenchido com um UUID gerado aleatoriamente pelo aplicativo. O campo `answerId` serve para responder uma mensagem recebida e ele pode ser nulo. O campo `route` deve ser preenchido com a rota do serviço de destino. O campo `data` deve ser preenchido com qualquer objeto JSON e é o campo que guarda a mensagem que será entregue para o serviço. O campo `ttl` deve ser preenchido com a data de expiração da mensagem.

O Quadro 3 mostra uma mensagem enviada para o serviço de rota `chat`. Nela é possível observar que o campo `data` possui diversas propriedades que são relevantes para um serviço de chat e são os dados que realmente importam para o serviço.

Quadro 3 – Exemplo de mensagem enviada pelo aplicativo

```
01 {
02     "id": "818e875f-2cdb-401d-ac81-6860ad588bb5",
03     "route": "chat",
04     "ttl": 1652494335000,
05     "data": {
06         "to": "fulano20",
07         "message": "Olá. Tudo bem com vc?",
08         "date": 1652494335000
09     }
10 }
```

Fonte: elaborado pelo autor.

O Quadro 4 mostra a mensagem recebida pelo serviço que foi enviada no quadro 3. É possível notar que a mensagem é muito semelhante, pois a única diferença é a inclusão do campo `from` que guarda o identificador do cliente que enviou a mensagem.

Quadro 4 – Exemplo de mensagem recebida pelo serviço

```
{
    "id": "818e875f-2cdb-401d-ac81-6860ad588bb5",
    "ttl": 1652494335000,
    "route": "chat",
    "from": "25281752-b4f5-43d5-9d1e-44d3f9a963f4",
    "data": {
        "to": "fulano20",
        "message": "Olá. Tudo bem com vc?",
        "date": 1652494335000
    }
}
```

Fonte: elaborado pelo autor.

Quando o aplicativo deseja confirmar o recebimento de uma mensagem, ele deve enviar uma mensagem para o *middleware* informando no campo `answerId` o id da mensagem que será confirmada e no campo `data` um campo `type` com valor `CONFIRM_MESSAGE`, conforme demonstrado no Quadro 5.

Quadro 5 – Exemplo de confirmação de mensagem pelo aplicativo

```
{
    "id": "6fbd855f-a171-4d77-b012-65b98d3cc2af",
    "answerId": "1797339b-700c-4e7f-ab7e-8af9e624b866",
    "route": "middleware",
    "ttl": 1652494335000,
    "data": {
        "type": "CONFIRM_MESSAGE"
    }
}
```

Fonte: elaborado pelo autor.

3.3.1.5 Envio de mensagem pelo serviço

O envio de mensagens pelo serviço é muito semelhante ao do aplicativo. As diferenças principais são: é enviado tudo para uma *Exchange* do *middleware* (não por WebSocket) e as mensagens têm como destino um `clientId` e não uma rota. Da mesma forma que o aplicativo, o serviço pode enviar qualquer mensagem que quiser, desde que esteja encapsulada na estrutura de mensagem do *middleware*, conforme demonstrado no Quadro 6.

Quadro 6 – Exemplo de confirmação de mensagem pelo aplicativo

```
{
    "id": "1797339b-700c-4e7f-ab7e-8af9e624b866",
    "type": "SEND_MESSAGE",
    "clientId": "25281752-b4f5-43d5-9d1e-44d3f9a963f4",
    "ttl": 1652236374000,
    "data": {
        "action": "ADD_NEW_MESSAGE",
        "from": "fulano10",
        "message": "Olá. Tudo bem com vc?",
        "date": 1652494335000
    }
}
```

Fonte: elaborado pelo autor.

O Quadro 6 traz um exemplo de mensagem que o serviço de chat envia para um cliente (aplicativo). As mensagens de serviço não possuem rotas, portanto, a distinção do que fazer com cada mensagem é feita pelo campo `type` que pode variar entre `SEND_MESSAGE`, `CONFIRM_MESSAGE` ou `GET_PENDING_MESSAGES`. O tipo `SEND_MESSAGE`, como exemplificado no quadro 6, realiza o envio de uma mensagem para um cliente. O tipo `CONFIRM_MESSAGE` tem o objetivo de confirmar o recebimento de uma determinada mensagem, igual ao aplicativo. Por fim, o tipo `GET_PENDING_MESSAGES` tem o objetivo de solicitar ao *middleware* as mensagens pendentes para o serviço.

3.3.1.6 Geração e publicação de métricas

O *middleware* gera métricas de diversas operações que são realizadas nele e realiza a publicação de forma assíncrona no InfluxDB. A publicação é feita em lote e de forma assíncrona para não afetar o desempenho das rotinas que geram métricas com a publicação delas. As métricas geradas são: quantidade e tamanho de mensagens enviadas e recebidas por WebSocket, quantidade e tamanho de mensagens enviadas e recebidas de serviços, quantidade de clientes conectados e quantidade de novas conexões. O Quadro 7 mostra um trecho de código de publicação de métrica.

Quadro 7 – Publicação de métrica de nova conexão

```
01 @ConsumeEvent(value = "websocket-onopen", blocking = true)
02 public Uni<Void> onOpen(String sessionId) throws IOException {
03     this.logger.info("New Session " + sessionId);
04     this.metricsService.publish(new NewConnectionMetric());
05     ...
06 }
```

Fonte: elaborado pelo autor.

No Quadro 7, na linha 4 é realizada a publicação da métrica de uma nova conexão. A publicação é sempre realizada através do serviço (camada interna da aplicação) de métricas. A classe `NewConnectionMetric` herda da classe `Metric` que é a classe base de métricas e possui uma estrutura base para publicar um ponto no InfluxDB.

3.3.2 Operacionalidade da implementação

Por se tratar de uma aplicação que funciona no *backend* e de forma transparente, ela não possui telas ou uma interatividade com o usuário final. Esta seção se destina a descrever como é possível construir uma infraestrutura para o *middleware* e como configurá-lo corretamente.

O *middleware* foi hospedado na AWS, foi criado um projeto Terraform para construir a infraestrutura deste trabalho e disponibilizado num repositório Git [1]. O *middleware* foi desenvolvido para funcionar em *containers* Docker e a construção da aplicação deve ser feita à parte do Terraform. O projeto Terraform irá exigir que um domínio seja configurado no projeto para que o Route53 possa criar os registros de DNS.

3.3.2.1 Construção da infraestrutura

Dentro do repositório Git é possível encontrar a pasta `terraform`, que contêm o projeto Terraform que é utilizado para a construção da infraestrutura. O Terraform é uma aplicação que automatiza a construção de recursos nas nuvens. É necessário instalá-lo antes de seguir com os demais passos. Algumas configurações devem ser realizadas no projeto Terraform para que seja possível executá-lo para construir a infraestrutura. Devem ser populados os dados da conta da AWS no arquivo `providers.tf` (Apêndice A) e devem ser alteradas as variáveis no arquivo `variables.tf` (Apêndice B). Existem alguns comentários nos arquivos que descrevem o que precisa de ajuste e todos os parâmetros possuem uma descrição para ajudar na identificação. Após as configurações, basta solicitar ao Terraform que construa a infraestrutura de acordo com a documentação oficial.

Com a infraestrutura construída, é necessário instalar e configurar o MongoDB, e o InfluxDB. Basta executar o `user_data` que está na instância EC2 dos respectivos recursos.

3.3.2.2 Construção do *middleware*

Dentro do repositório Git é possível encontrar a pasta `middleware`, que contém os códigos fonte do *middleware*. Nesta pasta existe o arquivo `Dockerfile` que é responsável por construir a imagem Docker do *middleware*. Basta construir uma imagem Docker chamada `middleware:latest` utilizando o `Dockerfile` que a aplicação será compilada e disponibilizada na imagem Docker. O projeto Terraform irá gerar um repositório ECR para publicação da imagem do *middleware*. Este repositório é chamado de `middleware` e devem ser seguidos os passos que a própria AWS disponibiliza para subir a imagem gerada para o repositório. Após construir o projeto Terraform e subir a imagem do *middleware* para o repositório, em alguns instantes o *middleware* estará pronto para comunicação.

---

[1] O projeto Terraform, bem como todos os códigos fonte deste trabalho, podem ser encontrados no GitHub do autor (https://github.com/arielrenostro/tcc2).

3.3.2.3 Configuração do *middleware*

O projeto Terraform cria um registro de DNS que aponta direto para o Load Balancer do *middleware*. Este registro é formatado com `middleware` seguido do domínio. Exemplo: se o domínio se chama `furb.br`, o registro será criado como `middleware.furb.br`. Este é o registro de DNS que deve ser utilizado para o aplicativo/cliente acessar o *middleware*.

A única configuração que precisa ser realizada no *middleware* é o cadastro dos serviços e suas rotas. O cadastro, edição, exclusão e consulta são realizados por uma API REST. A seguir serão apresentados exemplos dessas requisições, onde será omitido o *host* da URL.

O *endpoint* de **cadastro de serviço** serve para realizar o registro de um novo serviço. Para realizar a operação deve ser executada uma requisição com método POST, URL `/services` e o corpo descrito no Quadro 8.

Quadro 8 – Corpo da requisição de cadastro de serviço

```
01 {
02     "name": "ChatServer",
03     "sendType": "EXCHANGE",
04     "exchangeName": "chat-server",
05     "routeKey": "input",
06     "routes": [
07         "chat"
08     ]
09 }
```

Fonte: elaborado pelo autor.

O Quadro 8 demonstra um JSON com o conteúdo necessário para o cadastro de um serviço. O campo `name` indica o nome do serviço, o campo `sendType` indica o tipo de envio e pode variar entre `EXCHANGE` e `QUEUE`, o campo `exchangeName` indica o nome da *Exchange* (caso exista), o campo `routeKey` indica o *RouteKey* ou o nome da fila e o campo `routes` indica quais são as rotas do serviço.

O *endpoint* de **edição de serviço** serve para realizar uma alteração no registro de um serviço. Para realizar a operação deve ser executada uma requisição com método PUT, URL `/services/{id}`, onde o `{id}` na URL indica o id do serviço e o corpo descrito no Quadro 9.

Quadro 9 – Corpo da requisição de alteração de serviço

```
{
    "id": "507f1f77bcf86cd799439011",
    "name": "ChatServer",
    "sendType": "EXCHANGE",
    "exchangeName": "chat-server",
    "routeKey": "input",
    "routes": [
        "chat"
    ]
}
```

Fonte: elaborado pelo autor.

O Quadro 9 demonstra um JSON com o conteúdo necessário para a alteração do cadastro de um serviço. O campo `id` deve ser repassado sem alterações, o campo `name` indica o nome do serviço, o campo `sendType` indica o tipo de envio e pode variar entre `EXCHANGE` e `QUEUE`, o campo `exchangeName` indica o nome da *Exchange* (caso exista), o campo `routeKey` indica o *RouteKey* ou o nome da fila e o campo `routes` indica quais são as rotas do serviço.

O *endpoint* de **exclusão de serviço** serve para remover o registro de um serviço. Para realizar a operação deve ser executada uma requisição com método DELETE, URL `/services/{id}`, onde o `{id}` na URL indica o id do serviço e sem corpo. O retorno será o serviço removido.

O *endpoint* de **consulta de serviço** serve para obter os registros dos serviços. Para realizar a operação deve ser executada uma requisição com método GET, URL `/services` e sem corpo. Será retornada uma lista com todos os serviços cadastrados.

3.4 ANÁLISE DOS RESULTADOS

Nesta seção são apresentados os resultados obtidos através dos experimentos realizados com a aplicação desenvolvida. A seção 3.4.1 apresenta o ambiente e aplicações utilizados para a realização dos testes. Na seção 3.4.2 é feito um teste de estresse controlado na aplicação, sendo o foco observar quando era necessário incrementar uma instância ao *middleware*. Na seção 3.4.3 é feita a análise da capacidade de resiliência.

3.4.1 Ambiente de testes

Os testes foram divididos em dois principais grupos: provar a resiliência e provar a escalabilidade. Para os testes de escalabilidade, foi desenvolvida uma aplicação de serviço de chat e uma aplicação cliente de chat. Para os testes de resiliência foi utilizado o serviço de chat e executados testes manuais com ferramentas para avaliar a comunicação WebSocket para simular os clientes.

A aplicação de serviço cria e consome uma fila no RabbitMQ, aguarda o recebimento de mensagens que deverão ser encaminhadas para outro cliente (contido na mensagem), processa a mensagem e publica na fila de notificações do *middleware* para que ele encaminhe a mensagem ao cliente.

A aplicação de cliente consome uma *Exchange* de *broadcast* para receber as instruções de quantas conexões/clientes e quantas mensagens por minuto serão disparadas ao *middleware* por cada conexão/cliente. Os testes sempre são iniciados pelo cliente enviando mensagens ao *middleware*. A aplicação cliente também gera métricas, como tempo de resposta do *middleware* para responder o `ok` da mensagem enviada, taxa de envio por minuto, taxa de recebimentos por minuto, novas conexões, desconexões, mensagens enviadas com sucesso, mensagens enviadas com falha e *timeouts* no envio de mensagens ao *middleware*.

O *middleware* foi disponibilizado em instâncias Fargate com 0,5 de CPU e 1GB de RAM. As aplicações desenvolvidas para o teste (cliente e serviço) foram disponibilizadas em um *cluster* ECS (Elastic Container Service) com 3 instâncias EC2 c6a.xlarge (4 vCPU e 8GB de RAM). Foram escaladas 4 instâncias de cliente limitadas em 1024 unidades de CPU e 2GB de RAM cada, e 4 instâncias de serviço limitadas em 1024 unidades de CPU e 2GB de RAM cada. As quantidades de instâncias do *middleware* foram definidas à medida que os testes foram sendo executados.

Todas as aplicações periféricas (AmazonMQ, ElasticCache, MongoDB e InfluxDB) e de testes (cliente e serviço) foram superdimensionadas no trabalho para que não fossem impeditivos para o *middleware* escalar e foram monitoradas durante o trabalho afim de garantir que estivessem sempre operando em níveis aceitáveis de consumo de recurso.

3.4.2 Testes de estresse

Os testes de estresse são uma forma de avaliar a escalabilidade da aplicação conforme a carga for aumentando. Os testes no *middleware* foram realizados durante 10 minutos cada, com a quantidade de clientes/conexões fixa, onde cada cliente realizava o envio de uma mensagem por segundo. Foram coletadas as métricas de quantidade de instâncias, consumo de CPU, consumo de memória e tempo de resposta para responder os clientes. O consumo de CPU representa diretamente o estresse que a instância está sofrendo para atender a todas as mensagens que estão sendo trafegadas através dele. O tempo de resposta pode trazer um sinal mais claro de que a instância está sobrecarregada, pois à medida que a carga aumenta, o tempo de resposta tende a aumentar também. É importante monitorar as aplicações periféricas, como o banco de dados, pois o aumento no tempo de resposta pode estar

relacionado a alguns desses periféricos e não necessariamente a instância do *middleware*. Este não foi o caso do *middleware*, pois todas as aplicações periféricas foram superdimensionadas e o aumento do tempo de resposta foi ocasionado pela sobrecarga na própria instância do *middleware*. A Tabela 1 traz os resultados dos testes executados.

Tabela 1 – Resultados dos testes de estresse

Clientes	Taxa de envio	Instâncias	CPU	RAM	Tempo resposta médio	Tempo resposta p(95)
60	3.600/min	1	26%	22%	6ms	10ms
120	7.200/min	1	45%	23%	7ms	10ms
180	10.800/min	1	66%	23%	8ms	9ms
240	14.440/min	1	84%	23%	9ms	19ms
300	18.000/min	1	98%	26%	47ms	159ms
300	18.000/min	2	69%	28%	29ms	54ms
300	18.000/min	3	47%	30%	16ms	25ms
400	24.000/min	3	59%	30%	15ms	21ms
500	30.000/min	3	79%	30%	31ms	93ms
600	36.000/min	3	95%	30%	81,6ms	543ms
600	36.000/min	4	72%	27%	20ms	27ms

Fonte: elaborado pelo autor.

Na Tabela 1 cada linha representa um teste. A coluna `Clientes` representa a quantidade de clientes conectados simultaneamente, a coluna `Taxa de envio` representa a quantidade de mensagens enviadas por minuto por todos os clientes, a coluna `Instâncias` representa a quantidade de instâncias de *middleware*, a coluna `CPU` representa o percentual máximo de CPU atingido pelo *middleware*, a coluna `RAM` representa o percentual máximo de memória RAM atingida pelo *middleware*, a coluna `Tempo de resposta médio` representa a média dos tempos máximos de resposta do *middleware* e a coluna `Tempo de resposta p(95)` representa o percentil de 95 dos tempos de resposta do *middleware*, ou seja, quase os tempos de resposta máximos do *middleware*.

A Tabela 1 demonstra, através dos testes realizados no *middleware*, que ele conseguiu performar normalmente com uma única instância até 240 clientes e 14.440 mensagens por minuto. A partir dos 300 clientes e 18.000 mensagens por minuto, o *middleware* começou a consumir praticamente toda a CPU e o tempo de resposta subiu consideravelmente. Repetindo o mesmo teste com duas instâncias o resultado foi melhor, consumindo cerca de 69% de CPU e com os tempos de resposta mais baixos. Repetindo o mesmo teste com três instâncias o resultado foi ainda melhor, consumindo cerca de 47% de CPU e com tempos de resposta ainda mais baixos. O cenário se tornou estável e muito semelhante até a marca de 500 clientes e 30.000 mensagens por minuto, onde foi necessário acrescentar uma nova instância para que os tempos de resposta voltassem ao mesmo patamar de antes.

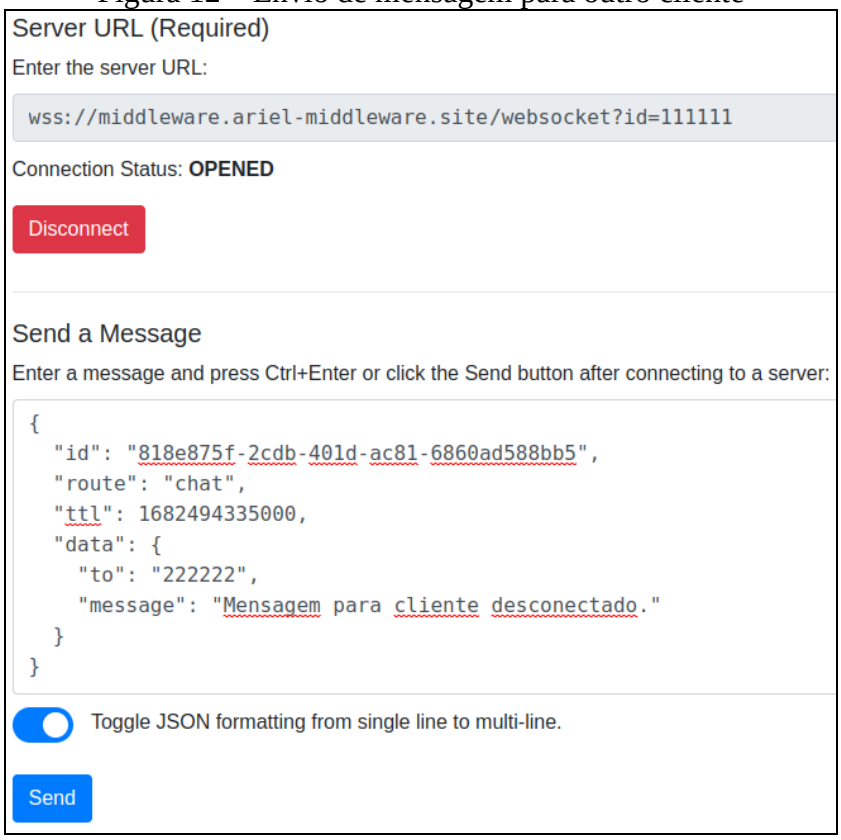
3.4.3 Análise de resiliência

A análise da resiliência foi realizada através de um teste onde é enviada uma mensagem para um cliente desconectado. Se o cliente receber a mensagem quando conectar, o *middleware* atende a este requisito.

O teste foi realizado simulando o cliente de forma manual com uma ferramenta de conexão WebSocket e utilizando o serviço de chat desenvolvido para os testes de estresse. O teste foi realizado conectando um cliente no *middleware*, efetuando o envio de uma mensagem para um cliente não conectado, analisando o comportamento do *middleware*, conectando o cliente, confirmando o recebimento da mensagem e analisando novamente o comportamento do *middleware*.

A Figura 12 dá início ao teste e mostra o envio da mensagem para outro cliente através do *middleware*.

Figura 12 – Envio de mensagem para outro cliente



Fonte: elaborado pelo autor.

Na Figura 12 foi conectado o cliente `111111` e realizado o envio de uma mensagem para o serviço de chat com destino no cliente `222222`, conforme apresentado no quadro `Send a Message`. A Figura 13 mostra as mensagens no banco de dados do *middleware* após o envio da mensagem.

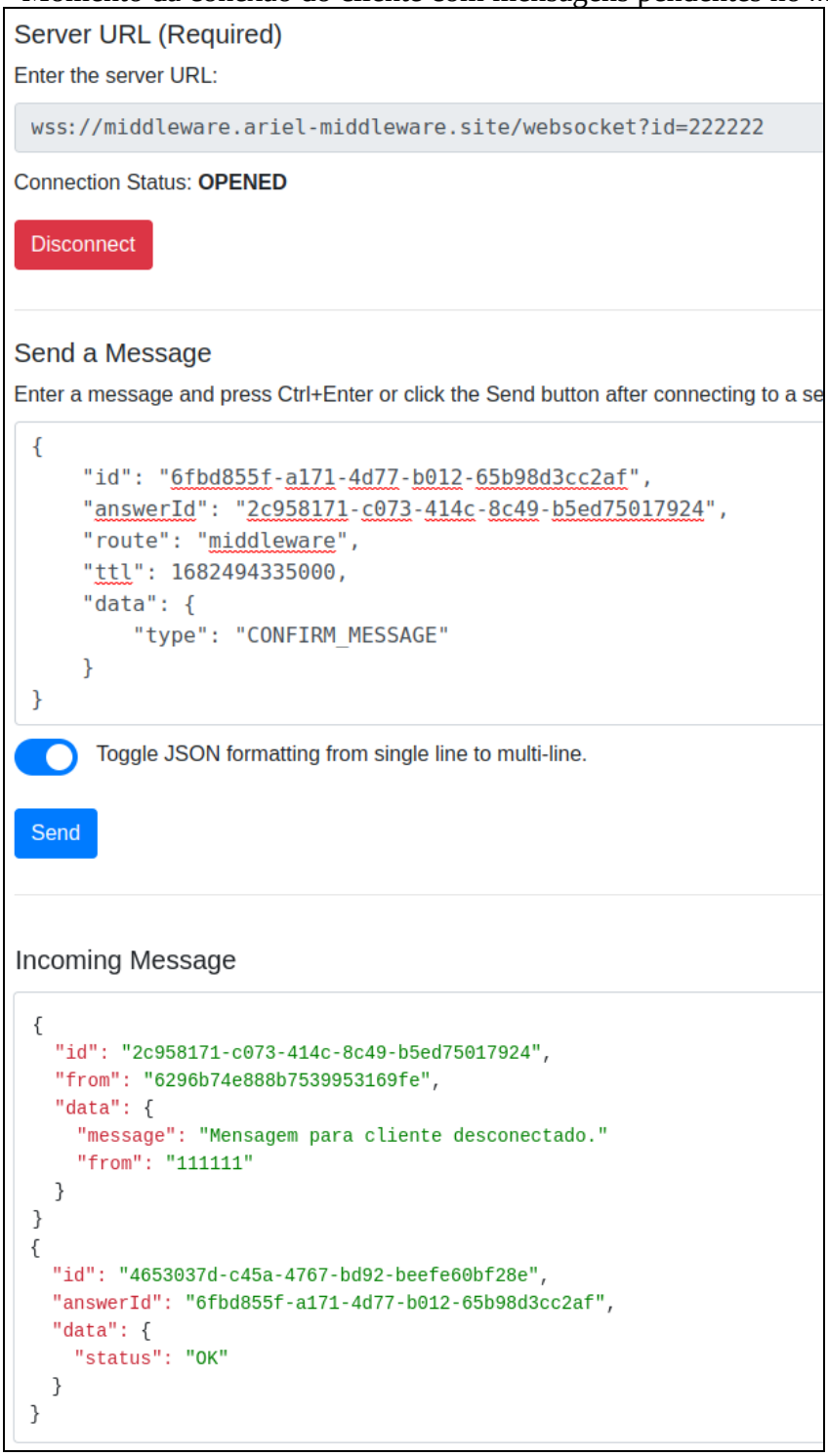
Figura 13 – Mensagens no banco de dados para cliente desconectado

	{} data	{} origin	{} destination	{} status ▲ 1
1	{"to": "222222", "message	{"_id": "111111",	{"_id": "6296b74e88	DELIVERED
2	{"message": "Mensagem par	{"_id": "6296b74e8	{"_id": "222222", "	PENDING

Fonte: elaborado pelo autor.

Na Figura 13 a linha 1 contém a mensagem enviada pelo cliente `111111` com o seu status indicando que a mensagem foi entregue ao serviço. A linha 2 contém a mensagem enviada pelo serviço com destino no cliente `222222` que ainda está pendente, indicando que a mensagem ainda não foi entregue ao cliente. A Figura 14 mostra o momento da conexão do cliente `222222` ao *middleware*.

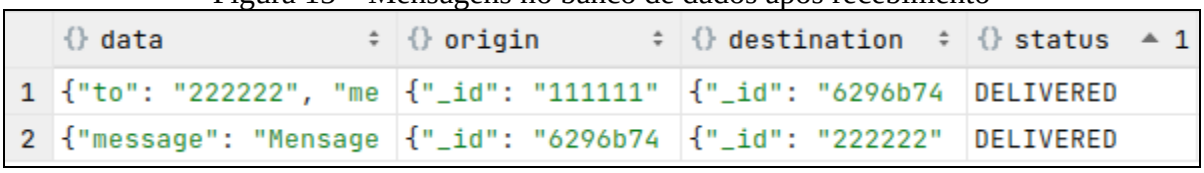
Figura 14 – Momento da conexão do cliente com mensagens pendentes no *middleware*



Fonte: elaborado pelo autor.

A Figura 14 mostra a conexão do cliente `222222` e a confirmação do recebimento da mensagem. No quadro `Incoming Message` mostra a mensagem pendente recebida pelo cliente e logo depois um `ok` do *middleware*. A mensagem de `ok` é referente a mensagem enviada pelo cliente no quadro `Send a Message`. A Figura 15 mostra as mensagens no banco de dados do *middleware* após o envio das mensagens.

Figura 15 – Mensagens no banco de dados após recebimento



	{} data	{} origin	{} destination	{} status ▲ 1
1	{"to": "222222", "me	{"_id": "111111"	{"_id": "6296b74	DELIVERED
2	{"message": "Mensage	{"_id": "6296b74	{"_id": "222222"	DELIVERED

Fonte: elaborado pelo autor.

Na Figura 15, a linha 1 contém a mensagem enviada pelo cliente `111111` sem nenhuma alteração. A linha 2 contém a mensagem enviada pelo serviço com destino no cliente `222222` que agora está com a situação indicando que foi entregue ao cliente.

4 CONCLUSÕES

Este trabalho apresentou a implementação de um *middleware* para comunicação de aplicações web e serviços em um ambiente distribuído. Os objetivos específicos foram atingidos, sendo eles a avaliação da escalabilidade da aplicação usando a AWS, a avaliação da resiliência na entrega de mensagens e atender ao contexto de aplicações distribuídas e aplicações web. A comunicação com o cliente foi desenvolvida utilizando o WebSocket, protocolo que está implementado nos navegadores o que possibilitando a implementação de aplicações web se comunicando com o *middleware*. Além disso, tira a necessidade de o cliente conhecer toda a complexidade do ambiente distribuído, bastando somente informar a rota. A comunicação com o serviço foi desenvolvida utilizando o protocolo AMQT, ou seja, por mensageria e possibilita que aplicações de serviço possam enviar mensagens para clientes sem precisar saber onde e se o cliente está conectado, bastando que o serviço envie uma mensagem ao *middleware* que ele irá realizar o trabalho de encontrar o cliente e enviar a mensagem.

Para o desenvolvimento do *middleware*, optou-se em utilizar o *framework* Quarkus pela sua integração com diversas bibliotecas Java, pelo prometido aumento na velocidade do desenvolvimento e pela compilação nativa do projeto que visa aumentar o desempenho e diminuir o consumo de recursos da instância. De maneira geral, o *framework* se demonstrou muito estável, foi simples iniciar um novo projeto, houve um ganho real na velocidade do desenvolvimento e contou com diversas parametrizações que ajudaram a refinar alguns parâmetros durante os testes do trabalho realizado, como por exemplo a quantidade de conexões abertas com o banco de dados. Porém, algumas bibliotecas como o Lombok ainda não estavam integradas com o Quarkus e a implementação do RabbitMQ para o Quarkus não atendia ao contexto deste trabalho. A limitação do Lombok impossibilitou que este trabalho fosse compilado de forma nativa e foi necessário compilar o projeto para Java 11 e rodar através de uma JVM. A limitação com relação ao RabbitMQ implicou na necessidade de implementar a sua utilização manualmente e não através das estruturas do Quarkus.

A utilização da AWS para hospedar o *middleware* ajudou muito no desenvolvimento do trabalho pois tirou a complexidade e a necessidade de gerenciar os *clusters* do Redis e RabbitMQ, possibilitou escalar as instâncias do *middleware* através do Fargate sem dificuldades e proporcionou um ambiente estável para realização dos testes.

Durante o desenvolvimento e testes do *middleware*, algumas rotinas passaram por melhorias e otimizações em função da análise do desempenho a fim de diminuir o estresse do

próprio *middleware*, bem como de alguns componentes periféricos a ele, principalmente o banco de dados. Alguns ajustes na quantidade de conexões com banco de dados, Redis e consumidores do RabbitMQ foram essenciais para a performance da aplicação.

Por fim, é possível concluir que este trabalho tem o potencial de abstrair, simplificar e em casos em que o cliente é impossibilitado de implementar o protocolo AMQT, possibilitar a comunicação entre aplicações web para dispositivos móveis e aplicações em ambiente distribuído. O *middleware* implementado permite que as aplicações conversem cada uma com seu protocolo, tirando a necessidade das aplicações de implementarem um mesmo protocolo para comunicação. Ele traz a utilização do protocolo WebSocket para comunicação com o cliente que pode ser utilizado por qualquer aplicação web (nativa em navegadores) e a utilização do protocolo AMQT para comunicação com os serviços, protocolo este que é muito difundido, possui bibliotecas para sua utilização em diversas linguagens de programação, tornando a utilização simples. E em um ambiente distribuído, por convenção, comunica-se de forma assíncrona, aumentando a chance de a aplicação em questão já fazer uso deste protocolo. Além disso, tanto o cliente quanto o serviço podem confiar que a mensagem será entregue ao destino quando o destinatário se conectar novamente.

4.1 EXTENSÕES

Com a conclusão do desenvolvimento deste *middleware*, surgiram sugestões para possíveis melhorias e trabalhos futuros.

a) a geração dos identificadores de cliente e mensagem são realizados pelo próprio cliente. Isto é ruim pois existe uma pequena chance de colisão dos identificadores. A sugestão é a geração dos identificadores ser por conta do *middleware* ou implementar uma forma de garantir que mesmo que o cliente gere os seus identificadores, eles nunca irão colidir;
b) para agilizar o desenvolvimento do trabalho, o banco de dados não foi distribuído. A sugestão é distribuir o banco de dados e balancear a carga nos bancos de dados, possibilitando que o *middleware* se torne ainda mais escalável. Uma abordagem bastante utilizada é a utilização de um nó de leitura e outro nó de escrita, isto não é suportado hoje pelo *middleware*;
c) este trabalho se limitou a tratar tudo dentro de uma única região da AWS. A estrutura da aplicação foi montada de forma que ela possa ser distribuída para diversas regiões da AWS e do mundo, fazendo com que a resposta da aplicação fosse mais rápida através de uma boa distribuição de DNS, fazendo com que os

clientes e serviços sempre interajam com o *middleware* que está mais próximo dele geograficamente;

d) possibilitar a comunicação com mais de um RabbitMQ, permitindo que o *middleware* seja inserido em mais de um grupo de serviços e possibilitando até mesmo uma comunicação Serviço – Serviço através do dele;
e) não possuir distinção de WebSocket para cliente e RabbitMQ para serviço, permitindo que um serviço também se comunique por WebSocket com o *middleware*.

REFERÊNCIAS

AMAZON WEB SERVICES. **Elastic Load Balancing**. 2022. Disponível em: https://aws.amazon.com/pt/elasticloadbalancing. Acesso em: 22 mar. 2022a.

AMAZON WEB SERVICES. **Preço AWS**. 2022. Disponível em: https://aws.amazon.com/pt/pricing. Acesso em: 22 mar. 2022b.

BARTHEL, Joern. **Getting started with AMQP and RabbitMQ**. 2009. Disponível em: https://www.infoq.com/articles/AMQP-RabbitMQ. Acesso em: 12/07/2022.

BBC NEWS BRASIL. **Internet móvel**: a revolução tecnológica do smartphone. a revolução tecnológica do smartphone. 2021. Disponível em: https://www.bbc.com/portuguese/internacional-55973855. Acesso em: 17 mar. 2022.

DUARTE, Luiz. **Criando apps para empresas com Android**. Gravataí: Luiztools, 2016. 242 p.

ISABELA MENDES (Brasil). **Entenda a importância de ter um site responsivo**. 2021. Disponível em: https://www.surfedigital.io/blog/entenda-a-importancia-de-ter-um-site-responsivo. Acesso em: 18 mar. 2022.

LOMBARDI, Andrew. **WebSocket**: lightweight client server communications. 1. ed. Sebastopol, CA: O'Reilly Media, Inc., 2015.

MICROSOFT AZURE. **O que é Computação em Nuvem?** 2022. Disponível em: https://azure.microsoft.com/pt-br/overview/what-is-cloud-computing. Acesso em: 22 mar. 2022.

MICROSOFT AZURE. **O que é middleware?** 2022. Disponível em: https://azure.microsoft.com/pt-br/overview/what-is-middleware. Acesso em: 20 nov. 2021.

NEWMAN, Sam. **Pattern**: backends for frontends. Backends For Frontends. 2015. Disponível em: https://samnewman.io/patterns/architectural/bff. Acesso em: 20 mar. 2022.

OEHLMAN, Damon; BLANC, Sébastien. **Aplicativos Web Pro Android**: desenvolvimento pro Android usando HTML5, CSS3 & JavaScript. 1. ed. Rio de Janeiro: Ciência Moderna, 2012.

RED HAT. **Middleware**. 2018. Disponível em: https://www.redhat.com/pt-br/topics/middleware/what-is-middleware. Acesso em: 20 mar. 2022.

SAUDATE, Alexandre. **REST**: construa api's inteligentes de maneira simples. São Paulo: Casa do Código, 2013. 314 p.

TANENBAUM, Andrew S.; STEEN, Maarten Van. **Sistemas Distribuídos**: princípios e paradigmas. 2. ed. São Saulo: Pearson, 2008.

APÊNDICE A – Arquivo providers.tf

Nesta seção, o Quadro 10 apresenta o arquivo providers.tf do projeto do Terraform.

Quadro 10 – Arquivo providers.tf

```
01 provider "aws" {
02   # se existe perfil aws, pode ser utilizado aqui:
03   # profile             = "ariel"
04
05   # ID da conta AWS
06   allowed_account_ids = [123456789123]
07
08   # região da AWS
09   region              = "us-east-2"
10
11   default_tags {
12     tags = {
13       CreatedBy   = "Terraform"
14       Application = "Middleware"
15     }
16   }
17 }
```

Fonte: elaborado pelo autor.

APÊNDICE B – Arquivo variables.tf

Nesta seção, o Quadro 11 apresenta o arquivo variables.tf do projeto do Terraform.

Quadro 11 – Arquivo variables.tf

```
01 variable "env" {
02   description = "Ambiente da aplicação"
03   type        = string
04   default     = "stg"
05 }
06 # São duas formas de gerenciar o DNS (Route53): pelo Terraform,
07 criado manualmente;
08 # Este método é gerenciado pelo Terraform.
09 # Quando utilizado este método, deve ser criado o módulo "dns"
10 #variable "domain" {
11 #  description = "Nome do domínio que será criado no Route53"
12 #  type        = string
13 #  default     = "ariel-middleware.site"
14 #}
15 # Este método é o manual.
16 # Quando utilizado este método, deve ser removido o módulo "dns".
17 variable "dns" {
18   default = {
19     route53 = {
20       zone_id = "XXXXXXXXXXX"
21       domain  = "ariel-middleware.site"
22     }
23   }
24 }
25
26 variable "mq_username" {
27   description = "Usuário do RabbitMQ"
28   type        = string
29   default     = "admin"
30 }
31
32 variable "mq_password" {
33   description = "Senha do RabbitMQ"
34   type        = string
35   default     = "@Batata-1234"
36 }
37
38 variable "bastion_public_key" {
39   description = "Bastion SSH Public Key"
40   type        = string
41   default     = "ssh-rsa xxxx"
42 }
43
44 variable "mongodb_public_key" {
45   description = "MongoDB SSH Public Key"
46   type        = string
47   default     = "ssh-rsa xxxx"
48 }
49
50 variable "influxdb_public_key" {
51   description = "InfluxDB SSH Public Key"
52   type        = string
53   default     = "ssh-rsa xxxx"
54 }
```

Fonte: elaborado pelo autor.