UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

EDUARDO MENZEN

PROPOSTA DE UM MECANISMO DE RECUPERAÇÃO DE MENSAGENS ENVIADAS PARA UM CONTROLADOR DE FILAS

BENTO GONÇALVES

2024

EDUARDO MENZEN

PROPOSTA DE UM MECANISMO DE RECUPERAÇÃO DE MENSAGENS ENVIADAS PARA UM CONTROLADOR DE FILAS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de **Bacharel em Ciência da Computação** na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Daniel Notari

BENTO GONÇALVES

2024

EDUARDO MENZEN

PROPOSTA DE UM MECANISMO DE RECUPERAÇÃO DE MENSAGENS ENVIADAS PARA UM CONTROLADOR DE FILAS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de **Bacharel em Ciência da Computação** na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 01/07/2024

BANCA EXAMINADORA

____

Prof. Dr. Daniel Notari
Universidade de Caxias do Sul - UCS

____

Prof. Esp. Daniel Antônio Faccin
Universidade de Caxias do Sul - UCS

____

Prof. Me. Samuel Francisco Ferrigo
Universidade de Caxias do Sul - UCS

”Não há nada constante, exceto a mudanca.”

Heráclito

RESUMO

Este trabalho aborda a necessidade de desenvolver uma estrutura para recuperação de mensagens não inseridas em filas de processamento de mensagens para um sistema de Ponto de Venda (PDV) de uma rede de lojas de varejo. Um sistema PDV é responsável pela realização de transações de compra e venda, sendo crucial para a segurança e agilidade dos processos de grandes empresas de varejo. O problema identificado está relacionado aos procedimentos que dependem da emissão do documento fiscal, os quais, uma vez iniciados, não podem ser cancelados após a emissão do documento. O objetivo principal deste trabalho foi criar uma estrutura de recuperação de mensagens que não foram inseridas na fila de mensagens devido a falhas no sistema ou na infraestrutura. Os objetivos específicos compreenderam na concepção de uma estrutura de armazenamento em banco de dados local para as mensagens, a definição do procedimento de recuperação em caso de falhas, o desenvolvimento de um protótipo em linguagem de programação que incorpore essa estrutura e procedimento, e a validação da solução por meio de testes unitários. Uma das etapas do trabalho incluiu a implementação de um protótipo para testar a validade da solução proposta em um ambiente controlado. Foram desenvolvidas duas aplicações seguindo o padrão de arquitetura cliente-servidor: uma responsável pela criação, envio e reprocessamento de mensagens via requisições *Simple Object Access Protocol* (SOAP) e outra, consistindo em um *Web Service* SOAP que recebe as mensagens e as insere em filas de processamento. A aplicação cliente foi estruturada seguindo o padrão *Model-View-Controller* (MVC), separando claramente as responsabilidades e facilitando a manutenção do código. O servidor, contendo menos partes de código, não seguiu nenhum padrão específico e foi implementado com quatro classes e uma interface, cada uma com funcionalidades específicas. Para testar a eficácia da solução, foram realizados testes em dois cenários distintos: um sem sobrecarga e outro com sobrecarga de requisições no *Web Service*, utilizando o aplicativo *SoapUI* para simular a carga. Nos testes sem sobrecarga, todas as mensagens foram enviadas com sucesso. No cenário com sobrecarga, algumas mensagens falharam, mas o reprocessamento garantiu que todas as mensagens fossem eventualmente entregues, demonstrando a eficácia da solução proposta. O desenvolvimento do protótipo também passou por ajustes para melhorar a interface e a lógica de simulação de erros. Inicialmente, considerou-se o uso de diferentes sistemas operacionais e gerenciadores de banco de dados, mas posteriormente, optou-se por uma abordagem mais uniforme para simplificar o ambiente de teste. No final, a solução foi testada com sucesso, validando a proposta em um ambiente controlado e adverso.

Palavras-chave: Fila de Mensagens. Estrutura de Recuperação. Recuperação de Falhas. Protótipo. *Web Service*. Cliente-servidor.

LISTA DE FIGURAS

Figura 1 – Estrutura geral de um sistema cliente-servidor. 15
Figura 2 – Camadas da Invocação a Método Remoto (RMI). 17
Figura 3 – Interação de entidades *Web Services*. 18
Figura 4 – Entidades *Web Services* e suas interações. 19
Figura 5 – Uso de requisição HTTP *POST* na comunicação cliente-servidor do protocolo SOAP. 20
Figura 6 – Exemplo comparativo entre XML e JSON. 21
Figura 7 – *Status* das mensagens na estrutura de integração da plataforma *IBM Maximo Manage*. 23
Figura 8 – Fluxo geral do funcionamento da API do link de pagamento Cielo. 24
Figura 9 – Fluxo atual do processo de emissão de venda, sem tratamento para eventuais erros no envio de mensagem para fila. 27
Figura 10 – Trecho de código sem tratamento de erro adequado. 28
Figura 11 – Arquitetura atual de envio de mensagens para filas. 28
Figura 12 – Fluxo proposto do processo de emissão de venda, com tratamento para eventuais erros no envio de mensagem para fila. 29
Figura 13 – Proposta de solução com uma estrutura de armazenamento de mensagens enviadas para o controlador de filas. 29
Figura 14 – Nova arquitetura de envio de mensagens para filas. 30
Figura 15 – Casos de uso das novas funcionalidades do sistema PDV. 30
Figura 16 – Diagrama de sequência das novas funcionalidades do sistema PDV. 31
Figura 17 – Diagrama de classes. 32
Figura 18 – Modelo lógico da tabela para armazenamento das mensagens enviadas para o controlador de filas. 33
Figura 19 – Tela do protótipo simulador desenvolvido. 34
Figura 20 – Diagrama de sequência demonstrando o fluxo das mensagens na estrutura que foi criada para o protótipo simulador. 35
Figura 21 – Representação dos aplicativos utilizando a arquitetura cliente-servidor. . . . 36
Figura 22 – Organização estrutural do código do aplicativo servidor. 37
Figura 23 – Camada de controle. 38
Figura 24 – Camada de modelo composta pelas subcamadas BO e *DAO*. 39
Figura 25 – Camada de visão. 40
Figura 26 – Diagrama de classes do aplicativo cliente. 42
Figura 27 – Diagrama de classes do aplicativo servidor. 42
Figura 28 – Tabela de fila de mensagens do aplicativo servidor. 43
Figura 29 – Configuração do *SoapUI*, considerando uma estratégia de carga variável. . . 47

Figura 30 – Número de *threads* criadas na simulação pelo *SoapUI* ao longo do tempo. . 47
Figura 31 – Simulação de criação e envio de mensagens com sobrecarga no *Web Service*. 48
Figura 32 – Tela do aplicativo cliente após o reprocessamento das mensagens pendentes. 49

LISTA DE QUADROS

Quadro 1 – Número de *threads* ao longo do tempo, para vários valores de variância. . . 47

Quadro 2 – Resultados do aplicativo cliente nas simulações sem sobrecarga no *Web Service*. 48

Quadro 3 – Resultados do aplicativo cliente nas simulações em paralelo com o aplicativo de teste de carga. 49

Quadro 4 – Mensagens recebidas pelo aplicativo servidor durantes as simulações. . . . 49

LISTA DE ABREVIATURAS E SIGLAS

API Interface de Programação de Aplicação
AWT *Abstract Window Toolkit*
BO *Business Objects*
CRUD *Create, Read, Update, Delete*
DAO *Data Access Object*
DOM *Document Object Model*
ERP *Enterprise Resource Planning*
FIFO *First In, First Out*
GPL *General Public License*
HTTP *Hypertext Transfer Protocol*
HTML Linguagem de Marcação de HiperTexto
IBM *International Business Machines*
J2EE Java2 *Platform Enterprise Edition*
JDBC *Java Database Connectivity*
JMS Java *Messaging Service*
JSON *JavaScript Object Notation*
MVC *Model-View-Controller*
PCs Computadores Pessoais
PDV Ponto de Venda
POO Programação Orientada a Objetos
REST *Representational State Transfer*
RMI Invocação a Método Remoto
RPC Chamada de Procedimento Remoto
SOAP *Simple Object Access Protocol*
TCP/IP *Transmission Control Protocol/Internet Protocol*
URI *Uniform Resource Identifier*
UDDI *Universal Description, Discovery and Integration*
W3C *World Wide Web Consortium*
WSDL *Web Services Description Language*
XML *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO . . . 11
1.1 PROBLEMA DE PESQUISA . . . 12
1.2 OBJETIVOS . . . 12
1.2.1 Objetivos específicos . . . 12
1.3 ESTRUTURA DO TRABALHO . . . 13

2 REFERENCIAL TEÓRICO . . . 14
2.1 TROCA DE MENSAGENS . . . 14
2.1.1 Cliente-Servidor . . . 14
2.1.2 RPC . . . 15
2.1.3 RMI . . . 16
2.1.4 Web Service . . . 17
2.1.5 Message Broker . . . 21
2.2 TRABALHOS RELACIONADOS . . . 22
2.2.1 IBM Maximo Manage . . . 22
2.2.2 API do Link de Pagamento Cielo . . . 23
2.3 CONSIDERAÇÕES FINAIS . . . 24

3 PROPOSTA DE SOLUÇÃO . . . 26
3.1 PROCESSO ATUAL . . . 26
3.2 MELHORIA NO PROCESSO ATUAL . . . 28

4 DESENVOLVIMENTO DA PROPOSTA DE SOLUÇÃO . . . 32
4.1 DESCRIÇÃO DO PROTÓTIPO SIMULADOR . . . 32
4.2 DESCRIÇÃO DA ARQUITETURA DO PROTÓTIPO . . . 36
4.2.1 Camada de Controle (Controller) . . . 38
4.2.2 Camada de Modelo (Model) . . . 38
4.2.3 Camada de Visão (View) . . . 39
4.3 AMBIENTE DE DESENVOLVIMENTO . . . 40
4.4 DIAGRAMA DE CLASSES . . . 41
4.5 MODELO DE DADOS . . . 41
4.6 CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO . . . 43

5 METODOLOGIA E RESULTADOS DOS TESTES . . . 46
5.1 AMBIENTE DE TESTE . . . 46

6 CONCLUSÃO . . . 50

REFERÊNCIAS **51**

1 INTRODUÇÃO

Grandes empresas de varejo, compostas por uma rede de lojas, necessitam que os processos de compra e venda de produtos e serviços sejam realizados por meio de um ou mais *softwares*, necessitando assim uma maior segurança e agilidade em seus processos. O *software* utilizado para a realização das vendas de produtos e serviços no caixa da loja é denominado Ponto de Venda (PDV). Muitas vezes, um sistema Ponto de Venda (PDV) realiza diversos procedimentos e a troca de mensagens[1] com outros sistemas, de forma síncrona ou assíncrona. Segundo Maziero (2020), a sincronização entre tarefas de um canal de comunicação pode ser síncrona, quando o envio e a recepção de mensagens bloqueiam as tarefas envolvidas até a conclusão da comunicação, ou pode ser assíncrona. Neste último caso, o envio e a recepção das mensagens não são bloqueantes, ou seja, não bloqueia a tarefa que esta sendo executada pelo sistema.

Segundo Tamae (2004), *Web Services* são serviços *web* que podem ser chamados através da internet para buscar dados ou interagir com outros serviços. Normalmente em um sistema PDV, essas mensagens são adicionadas em filas através da chamada de um *Web Service* ou em uma estrutura controladora de filas entre o produtor e o consumidor. Essas mensagens ficam em espera para serem processadas posteriormente pelo sistema consumidor da respectiva fila.

Possivelmente no momento em que ocorram as trocas de mensagens, o fluxo principal do programa não possa mais ser abortado para cancelar ou desfazer os procedimentos já realizados, normalmente devida ao fato de já existir um documento fiscal válido emitido, cujo cancelamento possa onerar ainda mais a situação de falha. Ou ainda, se deseja que o processamento de parte da informação não esteja conectado com o fluxo principal do programa, podendo o mesmo ser realizado de forma paralela ao fluxo principal ou posteriormente à finalização do mesmo. Estas peculiaridades na implementação da arquitetura de um sistema pode levar a escolha de uma arquitetura de comunicação assíncrona entre as partes constituintes do sistema. Segundo Fowler (2007) a comunicação assíncrona é crucial para construir sistemas escaláveis e resilientes. Essa abordagem permite que diferentes partes do sistema operem de forma independente, melhorando a capacidade de resposta e a eficiência, especialmente em situações de alta carga.

O fato de um processo ocorrer em partes distintas de um sistema, e não ser de forma atômica, instiga os arquitetos de um sistema a criarem estruturas e rotinas de controle, para evitar a perda de informação ou parte de um processo. Para Filho (2019) a qualidade de um processo não melhora simplesmente por estar de acordo com um padrão externo, mas sim à medida que o processo contribui para que um produto ou serviço seja entregue ao usuário ou

[1] Uma mensagem corresponde à transmissão de informação de um ponto a outro através de um circuito adequado (SAWAYA, 1999).

cliente com melhor qualidade, de forma mais rápida e com menor custo.

1.1 PROBLEMA DE PESQUISA

Um sistema PDV de uma loja de varejo, na qual será denominada neste trabalho de STORE, possui procedimentos que somente podem ser realizados após a emissão do documento fiscal, devido às dependências com o mesmo.

Estes procedimentos, realizados após a emissão do documento fiscal, estão normalmente relacionados à venda de serviços atrelados aos produtos da venda principal, que não compõem o documento fiscal, mas possuem dependência da existência do mesmo. Porém, após a emissão do documento fiscal, não é possível realizar o cancelamento do mesmo de forma automática, seja por motivos de limitação do próprio sistema ou ainda por motivo de regra de negócio. Assim sendo, após a emissão do documento fiscal, o processo de venda precisa ser finalizada de forma transparente para o usuário, como se todos os procedimentos posteriores à emissão do documento tivessem obtido sucesso em suas execuções.

Este tipo de situação acaba gerando perda de informações e não geração, das vendas dos serviços atrelados à venda principal,acarretando assim, trabalhos manuais para a geração das respectivas vendas para atender o cliente.

1.2 OBJETIVOS

O presente trabalho tem como objetivo principal desenvolver um protótipo de estrutura de recuperação de mensagens não inseridas numa fila de mensagens. Quando a inserção não ocorre em virtude da indisponibilidade do sistema de filas, seja esta de natureza sistêmica ou de infra-estrutura.

1.2.1 Objetivos específicos

Para que o objetivo do trabalho possa ser contemplado, determinaram-se os seguintes objetivos específicos:

- Descrever uma estrutura de armazenamento de mensagens enviadas para o sistema de filas;
- Descrever o processo de reenvio de mensagem para a fila de mensagens, no caso de falha de envio;
- Desenvolver, em linguagem de programação, um protótipo constituído pela estrutura de armazenamento, pelo processo de recuperação e um protótipo receptor de mensagens, para poder ser utilizado na validação da solução;

- Explicar a metodologia de testes de simulação e realizar testes unitários para a validação da solução.

1.3 ESTRUTURA DO TRABALHO

O presente trabalho possui a seguinte estrutura de organização: O primeiro capítulo se destina a dar uma ideia bem superficial da proposta do trabalho. No segundo capítulo, são abordados tópicos que servem como um referencial teórico para um melhor entendimento do problema abordado, e da solução proposta.

O terceiro capítulo se destina à descrever o problema abordado no presente trabalho, de uma forma mais aprofundada, assim como também a proposta de solução para o mesmo. Além disso, este capítulo descreve o protótipo de solução que será desenvolvimento para a validação da proposta de solução.

O quarto capítulo inicialmente descreve a arquitetura do protótipo em termos de um padrão de distribuição de tarefas e propósito, além da descrição detalhada da arquitetura em termos de *design de software* relacionado a organização e estrutura interna do protótipo desenvolvido e descrição do modelo de dados utilizado. Posteriormente descreve o ambiente utilizado para o desenvolvimento do protótipo da solução e por fim, o capítulo descreve o comportamento do protótipo simulador, os testes realizados e seus resultados.

O quinto e último capítulo deste trabalho se destina às conclusões e considerações finais.

2 REFERENCIAL TEÓRICO

Neste capítulo serão abordados aspectos fundamentais referente ao conceito de troca de mensagens, e das arquiteturas de comunicação e troca de mensagens cliente-servidor, *web service*, RPC e RMI, os quais proporcionaram uma base teórica para a compreensão do problema proposto e de sua solução. Além disso, este capítulo trará uma breve análise de trabalhos e soluções relacionadas ao tema do presente trabalho, como o aplicativo *IBM Maximo Manage* voltado para o uso de indústrias e a solução de pagamento *API do Link de Pagamento Cielo*, voltada para o varejo.

2.1 TROCA DE MENSAGENS

A troca de mensagens entre sistemas pode acontecer através da invocação de funções ou por meio de troca de dados estruturados ou não. Como exemplo de dados estruturados temos os documentos nos formatos *Extensible Markup Language* (XML) ou *JavaScript Object Notation* (JSON). Segundo Coulouris *et al.* (2013), as mensagens são enviadas entre os sistemas para coordenar suas atividades ou ainda para transferir informações entre processos.

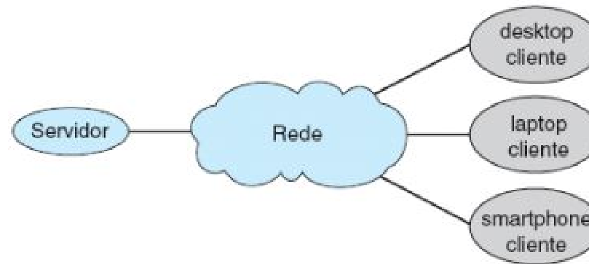
Programas executados em mais de uma máquina, as quais estão distribuídas e conectadas por meio de uma rede, precisam se comunicar mediante troca de mensagens, para sincronizar seus dados e suas execuções, realizando estas trocas de mensagens através da rede (MONTEIRO *et al.*, 2020).

Nas próximas seções deste trabalho, serão discutidas as arquiteturas de Cliente-Servidor, *Web Service*, RPC e RMI, que estão intimamente ligadas ao conceito de troca de mensagens entre sistemas ou processos.

2.1.1 Cliente-Servidor

Ao longo do tempo os Computadores Pessoais (PCs) aumentaram seu poder computacional, se tornando mais rápidos, além de mais baratos. Isto levou aos projetistas abandonarem a arquitetura de sistemas centralizados. Desta forma, terminais que antes eram conectados a sistemas centralizados, estão sendo substituídos por PCs e dispositivos móveis. O mesmo está acontecendo com as interfaces de usuários, que antes eram diretamente manipuladas por sistemas centralizados e agora estão cada vez mais sendo manipuladas quase sempre por uma interface web. Desta forma muitos sistemas se comportam como um sistema servidor para atender as solicitações realizadas por sistemas clientes. Um sistema desse tipo especializado e distribuído é denominado de sistema cliente-servidor (SILBERSCHATZ; GALVIN; GAGNE, 2015). A Figura 1 mostra a estrutura geral de um sistema cliente-servidor.

Figura 1 – Estrutura geral de um sistema cliente-servidor.



Fonte: (SILBERSCHATZ; GALVIN; GAGNE, 2015)

Segundo Silberschatz, Galvin e Gagne (2015) um sistema servidor consiste numa interface, a qual um sistema cliente pode realizar uma solicitação para executar uma ação. Por sua vez, o servidor executa a ação enviando os resultados ao sistema cliente. Para Elmasri R. A.; NAVATHE (2018) um servidor é um sistema composto de *hardware* e *software* que oferece serviços às máquinas clientes, entre eles acesso a arquivos, serviço de impressão ou ainda acesso a banco de dados. De acordo com Coulouris *et al.* (2013), o modelo de arquitetura cliente-servidor é considerado como o mais tradicional, além de ser o mais importante.

Este estilo de arquitetura possui ampla utilização em projetos físicos de sistemas distribuídos. A classificação dos processos entre clientes e servidores colabora para identificar as responsabilidades de cada sistema, assim como também identificar falhas de sistema de forma mais isolada. Um cliente ou um servidor podem ser componentes diferentes, como exemplo um *web service* ou um *cluster*[1] (MONTEIRO *et al.*, 2020).

2.1.2 RPC

A ideia geral da chamada de procedimento remota (Chamada de Procedimento Remoto (RPC)) é viabilizar que processos consigam chamar procedimentos remotos como se fossem locais, suportando diretamente o modelo cliente-servidor, de forma que os servidores ofereçam operações remotas por meio de interfaces de serviços chamadas pelos clientes diretamente como se estivessem fazendo uma chamada de procedimento local (MONTEIRO *et al.*, 2020).

A ideia central do RPC é fornecer uma abstração que permite que programas em um sistema distribuído chamem procedimentos em sistemas remotos como se fossem procedimentos locais. Isso simplifica o desenvolvimento de aplicações distribuídas, proporcionando uma interface familiar. A RPC lida com a complexidade da comunicação entre máquinas em uma rede distribuída, permitindo que um programa em uma máquina envie uma solicitação para a execução de um procedimento em outra máquina, ocultando a complexidade subjacente da comunicação de rede. Os desenvolvedores podem invocar procedimentos remotos sem se preocupar com os detalhes da implementação distribuída (BIRRELL; NELSON, 1984).

[1] Um cluster consiste num conjunto de máquinas conectadas por uma rede de alta velocidade (TANENBAUM; STEEN, 2007).

O paradigma RPC foi projetado com a intenção de abstrair os mecanismos das chamadas de procedimentos entre sistemas conectados através da rede. As mensagens no RPC são bem estruturadas, desta forma não são mais apenas pacotes de dados. Cada mensagem é endereçada a um *daemon* RPC que está escutando uma porta no endereço do sistema remoto e que contém um identificador especificando a função que deve ser executada e os parâmetros que devem ser passados para a mesma. Após a função ser executada conforme solicitado, e sua saída então é retornada para o solicitante através de uma mensagem separada (SILBERSCHATZ; GALVIN; GAGNE, 2015).

O conceito de RPC representa uma importante inovação intelectual na computação distribuída, sendo inicialmente atribuída a Birrell e Nelson no ano de 1984. O sistema de RPC subjacente faz a ocultação da codificação e decodificação de parâmetros e resultados, desta forma, os procedimentos de processos de computadores remotos podem ser chamados como se fossem procedimentos do mesmo espaço do endereçamento local (COULOURIS *et al.*, 2013).

2.1.3 RMI

A RMI possui conceitos similares à RPC, mas está relacionada ao mundo dos objetos distribuídos. Assim como na RPC, na RMI um objeto chamador pode invocar um método de outro objeto remoto, de forma que os detalhes subjacentes ficam ocultos para o usuário. Também suporta a programação com interfaces, sendo também construída sobre protocolos de requisição-resposta, podendo oferecer semânticas de chamadas. A RMI oferece um nível de transparência semelhante a RPC, ou seja, as chamadas locais e remotas utilizam a mesma sintaxe, mas no caso da RMI, as chamadas remotas normalmente expõem a natureza distribuída da chamada subjacente (COULOURIS *et al.*, 2013).

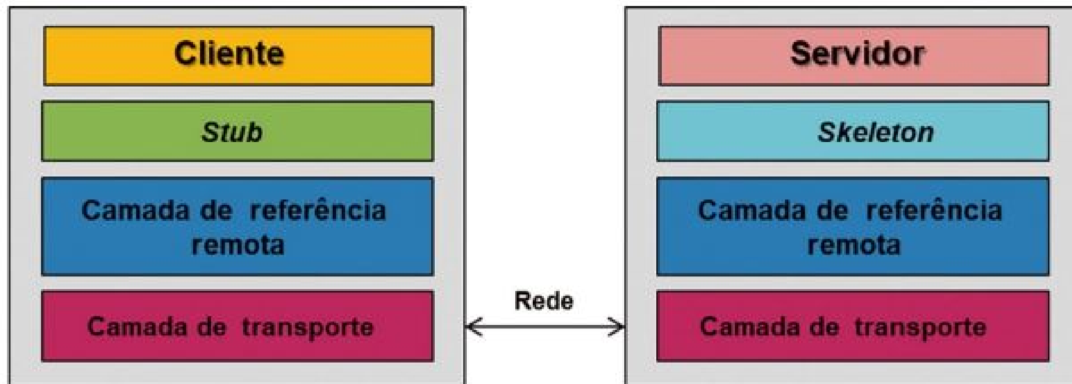
A ideia geral da RMI é semelhante à ideia da RPC, porém a RMI possui foco na invocação de objetos remotos, enquanto a RPC se concentra na invocação de procedimentos remotos. Seu objetivo é permitir que um objeto consiga invocar um método de outro objeto remoto. Pelo fato de oferecer suporte a objetos, a RMI oferece um maior potencial do que a RPC (MONTEIRO *et al.*, 2020).

Em relação às diferenças destas duas estratégias RPC e RMI, podemos destacar a questão de que no caso da RMI, pode-se utilizar todo o poder expressivo da programação orientada a objetos no desenvolvimento de programas distribuídos, incluindo o uso de objetos, classes e herança. Pode-se complementar a questão de que num sistema baseado em RMI, independentemente de objetos locais ou remotos, todos possuem referências exclusivas, as quais podem ser passadas como parâmetros, desta forma oferecendo uma semântica de passagem de parâmetros mais rica do que na RPC (COULOURIS *et al.*, 2013).

Quanto ao funcionamento da RMI, seu compilador denominado `rmic`, gera duas classes: *stub* e *skeleton*, conforme Figura 2. Desta forma, a classe implementa os métodos da inter-

face remota, sendo responsável por redirecionar a chamada do cliente ao objeto remoto (MONTEIRO *et al.*, 2020).

Figura 2 – Camadas da RMI.



Fonte: (MONTEIRO *et al.*, 2020)

A Figura 2 mostra que no lado do cliente, quando a classe *stub* é chamada devido à chamada de um método, o *stub* empacota os argumentos do método remoto e os envia para a classe *skeleton* do servidor. No servidor, o *skeleton* por sua vez recebe os dados e o método enviados pelo *stub* e os chama localmente. O resultado após a chamada do método, é enviada ao *stub* pelo *skeleton* (MONTEIRO *et al.*, 2020).

2.1.4 Web Service

Segundo Ferreira (2021), um *Web Service* é uma tecnologia que surgiu em meados da década de 1990, com o intuito de incorporar sistemas e realizar a comunicação entre eles através de tecnologias e protocolos.

Para Tamae (2004), *Web Services* são serviços web que podem ser buscados e chamado através da Internet. Após ser publicado num servidor web, um *Web Service* pode ser acessado ou chamado por outros programas ou até mesmo por outros *Web Services*, para obter dados, ou para interagir com outros serviços oferecidos por uma organização. Esses programas conseguem realizar desde uma simples troca de mensagem, como também transações mais complexas, como exemplo, o processo de compra de produtos.

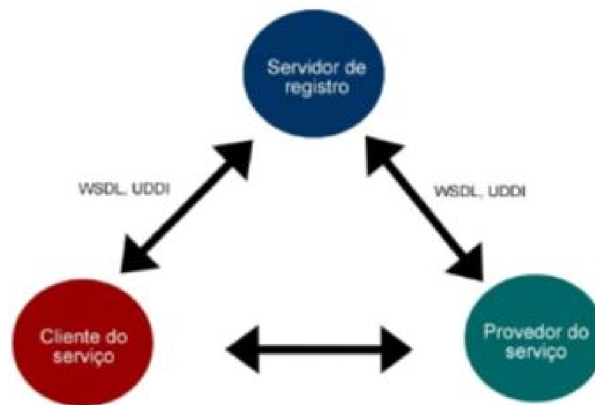
Um *Web Service* é uma tecnologia de comunicação que permite a interoperabilidade entre sistemas distribuídos na web. Ele facilita a troca de informações e a execução de operações entre diferentes aplicativos, independentemente da plataforma ou linguagem de programação utilizada. Em termos simples, um *Web Service* permite que aplicações se comuniquem e compartilhem dados pela internet, seguindo padrões e protocolos estabelecidos (ERL, 2009).

As tecnologias *Hypertext Transfer Protocol* (HTTP), XML, *Web Services Description Language* (WSDL) e *Universal Description, Discovery and Integration* (UDDI) são utilizadas para auxiliar nos processos de comunicação, segurança e administração de um *Web Service*. O

HTTP é um protocolo de comunicação responsável pelo transporte dos dados, enquanto o XML é uma linguagem de marcação, utilizada para facilitar o compartilhamento das informações. O WSDL é baseado em XML e descreve a formatação das mensagens e as interfaces de um *Web Service*. Um *Web Service* pode ser rapidamente encontrado de maneira fácil e dinâmica através do UDDI. Em termos de arquitetura, um *Web Service*, pode ser baseado numa arquitetura *Simple Object Access Protocol* (SOAP) ou *Representational State Transfer* (REST) (FERREIRA, 2021).

Conforme Ferreira (2021), as arquiteturas *Web Service* possuem três entidades interagindo entre si, para realizar publicação, busca e execução de operações, que são o provedor de serviço (*service provider*), o cliente de serviço (*service requestor*) e o servidor de registro (*service registry*). A figura Figura 3 mostra um exemplo deste tipo de arquitetura de *Web Service*, com as interações das entidades.

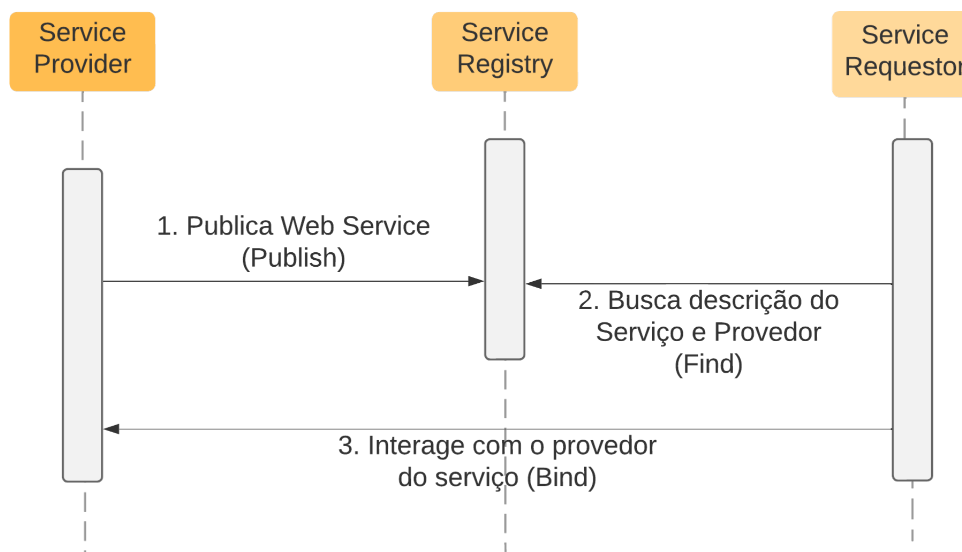
Figura 3 – Interação de entidades *Web Services*.



Fonte: (FERREIRA, 2021)

Pode-se dizer que o *service provider* publica ou hospeda o *Web Service* num servidor, o *service requestor* consomem o *Web Service*, ou seja, realiza uma interaçao com o *Web Service*. E o *service registry* armazena os aquivos com as descrições dos serviços publicados pelo service provider (FERREIRA, 2021). A Figura 4 mostra a sequencia de interações entre as entidades de um *Web Service*.

Figura 4 – Entidades *Web Services* e suas interações.



Fonte: O Autor (2023).

> Assim, de forma sintetizada, temos que o provedor do serviço publica o *Web Service* em um servidor, e os aplicativos clientes consomem o *Web Service* publicado. Para acessar o serviço, o aplicativo cliente usa a localização do *Web Service*, a qual é o seu URL, dessa maneira, o aplicativo cliente acessa o *Web Service* e envia parâmetros. Em troca, recebe as informações retornadas, geralmente, como uma estrutura em formato XML ou JSON (FERREIRA, 2021).

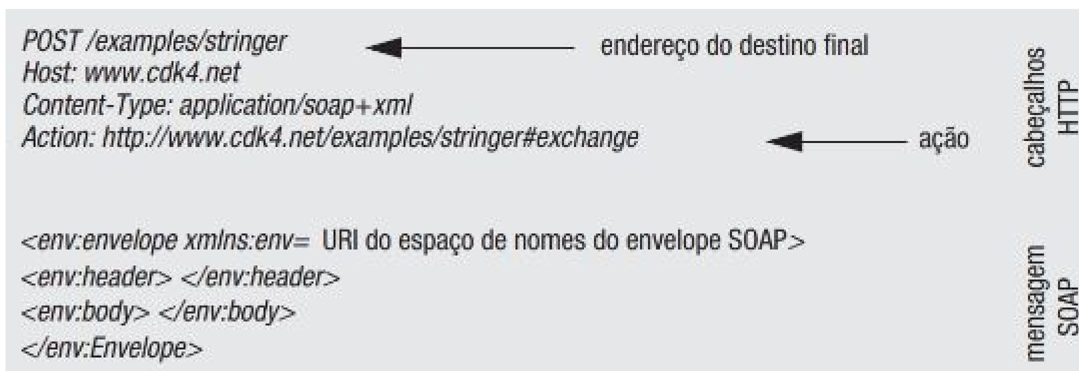
As duas abordagens mais comuns e amplamente empregadas para *Web Service* são a arquitetura e REST e o protocolo SOAP. Segundo (RODRIGUES *et al.*, 2020), REST é um padrão de arquitetura, que define como deve ser criado o serviço, enquanto RESTful corresponde ao serviço desenvolvido utilizando o estilo arquitetural REST.

O SOAP é um protocolo de arquitetura orientada a serviços, leve para trocar dados via XML, independentemente da linguagem ou plataforma de programação, seguindo o modelo cliente-servidor, chamando programas e aplicações remotamente via RPC (ROY; RAMANUJAN, 2001 apud FERREIRA, 2021).

SOAP é um protocolo baseado em XML para troca de mensagens utilizando, normalmente HTTP, como protocolo de transporte. As mensagens enviadas via SOAP possuem os elementos **envelope**, **header** (cabeçalho), **body** (corpo), **payload** (documento) e **fault** (falha). É comum utilizar no cabeçalho da mensagem, o SOAP *action*, o qual indica o endereço da mensagem (FERREIRA, 2021).

Para enviar uma mensagem SOAP, é exigido um protocolo de transporte, mas independe do protocolo de transporte utilizado. Os envelopes das mensagens não possuem nenhuma referência do endereço de destino, desta forma fica de responsabilidade do protocolo HTTP, ou qualquer outro protocolo usado no transporte das mensagens SOAP, especificar o endereço de destino (COULOURIS *et al.*, 2013).

Figura 5 – Uso de requisição HTTP *POST* na comunicação cliente-servidor do protocolo SOAP.



Fonte: (COULOURIS *et al.*, 2013)

A Figura 5 mostra de que forma o método HTTP *POST* é usado na transmissão de mensagem SOAP. Os cabeçalhos HTTP especificam o endereço *Uniform Resource Identifier* (URI) do destino e a ação que deverá ser executada e o corpo do protocolo HTTP transporta a mensagem SOAP (COULOURIS *et al.*, 2013).

Segundo Tihomirovs e Grabis (2016), a quantidade de dados enviados pelo SOAP pode provocar problemas de desempenho, devido à adição de cabeçalho e corpo complementares à mensagem. De acordo com Rodrigues *et al.* (2020) o REST é uma nova abordagem de implementação de *Web Services* baseada no protocolo HTTP. Desta forma, o SOAP comparado ao REST, pode ser considerado um protocolo ultrapassado.

O REST tende a ser "mais leve"em comparação com o SOAP, pois não adiciona camadas de dados para serem processadas na pilha de comunicação e transmissão. O mesmo foi projetado para dar suporte ao trabalho com arquivos de mídia, objetos de *hardware* específico. Qualquer *Web Service* definido com o princípio REST pode ser denominado RESTful, o qual usa verbos ou ações normais do protocolo HTTP, como GET, POST, PUT e DELETE, na execução de operações. Também permite a entrega de dados via texto puro, Linguagem de Marcação de HiperTexto (HTML), XML e JSON, sendo este último formato mais leve e legível para os seres humano (RODRIGUES *et al.*, 2020). A Figura 6 mostra um exemplo da diferença entre XML e JSON.

Os arquivos JSON cuja sigla significa *JavaScript Object Notation*, são formados por coleções de chave/valor e listas ordenadas de valores, enquanto que nos arquivos XML, o texto é delimitado por *tags*. Apesar do nome, o JSON é independentemente de linguagem, podendo ser utilizado em qualquer linguagem de programação e não somente com *JavaScript*, como o nome sugere. Além disso, é um formato de troca de dados leve e de fácil análise. (RODRIGUES *et al.*, 2020).

Figura 6 – Exemplo comparativo entre XML e JSON.

```xml
<?xml version="1.0"
      encoding="UTF-8"?>
<autenticacao>
  <usuario>meu_login</
usuario>
  <senha>minha_senha</
senha>
  <validacoes>
    <validacao>
      <chave>endereco</
chave>
      <valor>127.0.0.1</
valor>
    </validacao>
  </validacoes>
</autenticacao>
```

```json
{
   "usuario" :
"meu_login",
   "senha" :
"minha_senha",
   "validacoes" : {
      "validacao" : [
         {
            "chave"
: "endereco",
            "valor"
: "127.0.0.1"
         }
      ]
   }
}
```

Fonte: (RODRIGUES *et al.*, 2020)

2.1.5 Message Broker

Message broker é um componente importante em arquiteturas de sistemas distribuídos, facilitando a comunicação assíncrona entre diferentes partes de um sistema. Ele atua como um intermediário entre produtores e consumidores de mensagens, permitindo o envio, roteamento e processamento de mensagens em larga escala de forma eficiente, desta forma produtores e consumidores podem realizar uma comunicação de forma indireta, por intermétido do *message broker*.

De acordo com Tanenbaum e Steen (2007), sistemas distribuídos enfrentam desafios significativos de comunicação devido à heterogeneidade de hardware, protocolos de rede e requisitos de aplicativos. Nesse contexto, o *message broker* surge como uma solução para simplificar a integração e promover a interoperabilidade entre diferentes componentes de um sistema distribuído.

Stallings (2012) destaca a importância dos *message brokers* como parte integrante de sistemas distribuídos confiáveis e eficientes, fornecendo um canal de comunicação assíncrona e confiável. Os *message brokers* permitem que os sistemas distribuídos lidem com cargas de trabalho variáveis e distribuídas de maneira eficaz, garantindo a entrega confiável de mensagens mesmo em condições adversas.

2.2 TRABALHOS RELACIONADOS

Essa seção apresenta trabalhos cujas propostas de problemas e solução estão correlacionadas com a proposta do presente trabalho, aplicadas em campos distintos do conhecimento.

2.2.1 IBM Maximo Manage

O *IBM Maximo Manage* é um aplicativo integrante da *suite* de aplicativos *IBM Maximo Application Suite*[2] fornecida pela *International Business Machines* (IBM). O *IBM Maximo Manage* é uma plataforma gerenciadora de processos de negócios, fluxo de trabalho, ciclo de vida e de operações de ativos de uma empresa ou corporação, oferecendo uma visualização dos tipos de ativos e de suas condições, desta forma fornece recursos de planejamento, controle, auditoria e conformidade, ajudando a otimizar vários aspectos do negócio (IBM, 2021).

A plataforma *IBM Maximo Manage* é composta por uma estrutura de integração com suporte à arquitetura Java2 *Platform Enterprise Edition* (J2EE), com canais de publicação ou serviços corporativos que utilizam filas Java *Messaging Service* (JMS) para troca de dados com sistemas externos, desta forma, ela conta com um gerenciador de erro de fila iniciado quando uma condição de erro é identificada. A natureza do erro pode ser desde uma falha de comunicação, problemas de configurações ou ainda por validações de regras de negócio, ou lógica de processamento de entrada (IBM, 2021).

Na estrutura de integração da plataforma *IBM Maximo Manage*, as mensagens são processadas em fila sequencial, uma por vez, seguindo a lógica *First In, First Out* (FIFO), ou seja, a primeira mensagem a entrar é a primeira a sair. Quando ocorre algum erro no processamento de alguma mensagem, o mecanismo de gerenciamento de erro é iniciado e a mensagem é sinalizada como tendo um erro. Dependendo da configuração do aplicativo gerenciador de erro de fila, a estrutura de integração pode realizar várias tentativas de processamento de uma mesma mensagem, antes de determinar que o erro da mesma requer intervenção. O sistema também pode enviar uma mensagem para várias contas de *e-mail*, informando que ocorreu um erro, e ainda, criar um registro contendo a mensagem com erro, podendo ser visualizado e tratado através do aplicativo *Reprocessamento de Mensagens*.

No aplicativo *Reprocessamento de Mensagens*, é possível alterar o *status* das mensagens que estão sinalizadas com erro, corrigir as mensagens modificando o seu conteúdo ou excluir as mesmas do banco de dados. A Figura 7 mostra os possíveis *status* das mensagens (IBM, 2021).

O *status* padrão para as mensagens sinalizadas com erro, é o *RETRY*. Até o problema da mensagem não ser corrigido, o sistema continua a reprocessá-la conforme a contagem configurada para novas tentativas. Quando o número de tentativas se esgotam, o sistema altera o *status* da mensagem para *HOLD* (IBM, 2021).

[2] <https://www.ibm.com/br-pt/products/maximo>

Figura 7 – *Status* das mensagens na estrutura de integração da plataforma *IBM Maximo Manage*.

Status	Descrição
RETRY	A mensagem está pronta para ser reprocessada pelo sistema.
HOLD	A mensagem não está pronta para ser reprocessada pelo sistema.

Fonte: (IBM, 2021)

2.2.2 API do Link de Pagamento Cielo

O Link de Pagamento da Cielo é uma funcionalidade agregada ao *site* da Cielo, a qual um lojista pode criar um *link* de pagamento através do *backoffice* do *site* da Cielo ou ainda integrar seu sistema de vendas com o *site* da Cielo, possibilitando assim também a criação de um *link* de pagamento via seu sistema de vendas, sem a necessidade de acessar o *site* da Cielo (CIELO, 2023).

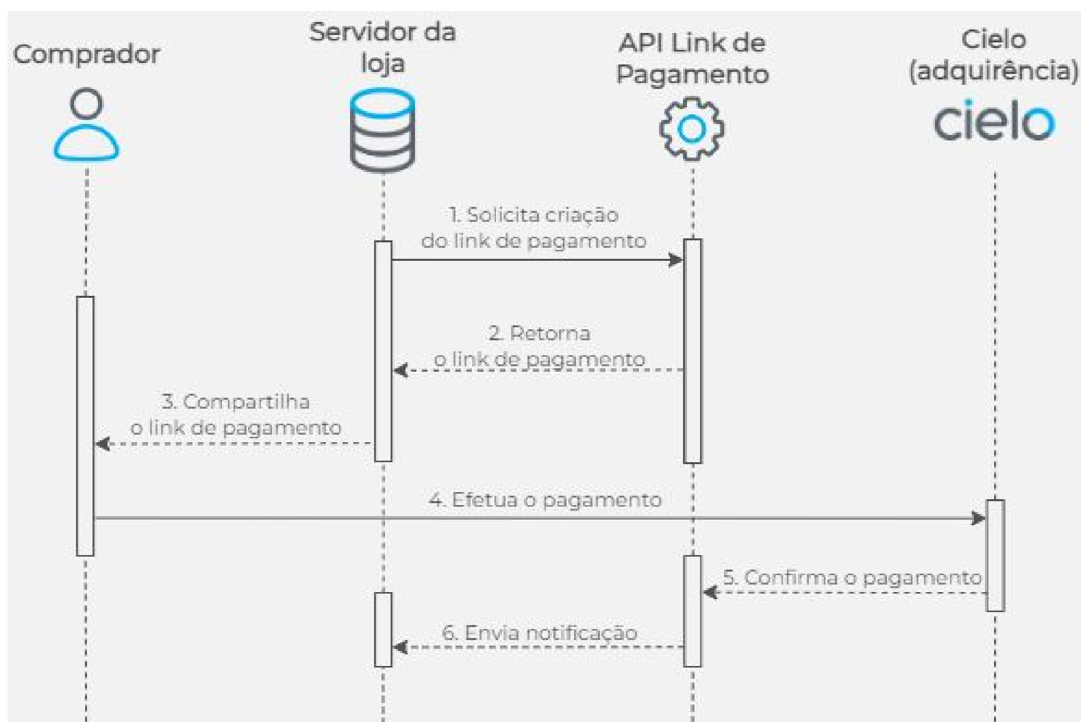
Uma vez criado, o link de pagamento de uma venda pode ser enviado ao cliente por qualquer canal de comunicação, entre eles, WhatsApp, Instagram, Facebook, e-mail e SMS. O cliente ao abrir o *link* de pagamento visualiza uma página personalizada com o logo do lojista e as opções de pagamento. Os meios de pagamento habilitados para esta funcionalidade são cartão de crédito, cartão de débito, QR Code Pay e PIX (CIELO, 2023).

A API do Link de Pagamento Cielo é uma Interface de Programação de Aplicação (API) REST que permite que os lojistas possam criar, editar e consultar *links* de pagamento através de seus próprios sistemas de venda e compartilhar o Link de Pagamento com os seus clientes. A Figura 8 mostra o fluxo geral do funcionamento da API do Link de Pagamento Cielo (CIELO, 2023).

O fluxo geral do processo de utilização e criação do Link de Pagamento da Cielo, conforme Figura 8 compreende os passos a seguir (CIELO, 2023):

1. O sistema do lojista envia uma solicitação de criação do link de pagamento para a API do Link de Pagamento;
2. A API Link de Pagamento retorna a URL do link de pagamento;
3. O sistema do lojista compartilha o link de pagamento com o cliente comprador;
4. O cliente acessa a URL do link e efetua o pagamento;
5. A Cielo, como adquirente, verifica as informações do pagamento autorizando ou não mesmo, retornando o resultado para a API do Link de Pagamento;
6. A API Link de Pagamento envia uma notificação de mudança de *status* ou finalização da transação para a URL de notificação configurada pelo lojista.

Figura 8 – Fluxo geral do funcionamento da API do link de pagamento Cielo.



Fonte: (CIELO, 2023)

No fluxo apresentado anteriormente, o Item 6 está relacionado a este trabalho, pelo fato de que na API do Link de Pagamento Cielo, o processo de envio de notificação é dotado de uma estrutura para tratamento de erro.

As notificações de conclusão de transação ou de alteração de *status* são enviadas para as URLs configuradas[3] no estabelecimento do lojista no site da Cielo. Desta forma, na conclusão de uma transação ou na alteração de seu *status*, a API do Link de Pagamento Cielo envia uma mensagem via POST no formato JSON para a URL cadastrada. Caso a URL cadastrada retornar algum erro ou não estiver disponível, a API irá realizar mais três tentativas com intervalo de uma hora entre cada POST. Também é possível acessar o pedido no site da Cielo, associado ao link de pagamento gerado e realizar o reenvio da notificação de forma manual.

2.3 CONSIDERAÇÕES FINAIS

Quando pensamos em sistemas corporativos, somos levados a imaginar vários *softwares* se comunicando. Isso não é um equívoco, pois cada vez mais temos a oferta de sistemas especialistas para cumprir funções específicas. Desta forma, os processos de uma empresa podem ser realizados com *softwares* diferentes, cada um com suas peculiaridades, mas integrando-se com os demais *softwares* da companhia.

[3] É possível configurar uma URL para notificação de conclusão de transação e outra para notificação de alteração de *status*, ou a mesma URL para ambas as situações.

Esta integração entre os sistemas pode ser feita de forma eficiente, mediante trocas de mensagens entre eles, utilizando diferentes tipos de arquiteturas, como Cliente-Servidor, *Web Service*, RPC e RMI. Os dados podem ser estruturados nos formatos XML ou JSON. A utilização de dados de forma estruturada tem a vantagem de torná-los mais claros para o ser humano, facilitando assim a manutenção destes *softwares*.

Em relação às arquiteturas, a arquitetura Cliente-Servidor além de ser a mais importante, é a mais tradicional e consiste numa interface em que um sistema cliente pode realizar uma solicitação a um sistema servidor, para que o mesmo execute uma ação. Já um *Web Service* consiste em serviços web utilizando o modelo Cliente-Servidor sendo o SOAP e o REST as arquiteturas comumente utilizadas. A primeira troca dados via XML através de uma chamada RPC e a segunda, considerada mais avançada, é baseada no protocolo HTTP entregando dados via HTML, XML e JSON. Os paradigmas RPC e RMI também são baseados no modelo Cliente-Servidor e foram projetados com o intuito de viabilizar que processos consigam invocar procedimentos ou objetos remotos como se fossem locais. Os conceitos da RMI são similares à RPC, porém seu foco está relacionado ao mundo dos objetos, permitindo assim que um objeto possa invocar um método de outro objeto remoto.

Pode-se destacar alguns trabalhos relacionados com a proposta de solução do problema apresentado neste trabalho, entre elas as propostas comercias como o *IBM Maximo Manage* e a *API do Link de Pagamento Cielo*. A proposta da empresa IBM consiste numa estrutura de integração e de troca de dados através de filas de processamento gerenciadas pela própria plataforma, sendo possível definir parâmetros para o reprocessamento de mensagens que obtiveram erro em seu processamento. Já a proposta da *CIELO* é uma funcionalidade agregada ao próprio *backoffice* de seu *site*, disponível para os lojistas clientes da *CIELO* à qual é possível criar um *link* de pagamento pelo próprio *backoffice* ou ainda o lojista pode integrar esta funcionalidade com seu próprio sistema de vendas. Esta solução, quando integrada com o sistema de vendas do lojista, realiza interações com o mesmo através de envio de notificações, as quais são reprocessadas no máximo 3 vezes num intervalo de uma hora, em caso de erro de envio da notificação.

O próximo capítulo deste trabalho é composto por uma seção que apresenta o processo atual da rede de loja STORE e um breve histórico na evolução do processo de venda da rede de lojas. Nesta mesma seção é reiterado o problema apresentado no primeiro capítulo do trabalho. Nas seções seguintes, será apresentada a proposta de solução para o problema atual, a estrutura de armazenamento proposta e a prototipação da solução, para a validação da mesma.

3 PROPOSTA DE SOLUÇÃO

Neste capítulo será abordada a proposta de solução para o problema considerado no presente trabalho, além de explorarmos o desenvolvimento de um protótipo para realizar a validação da solução do problema proposta neste trabalho.

3.1 PROCESSO ATUAL

A rede de lojas STORE é composta por um sistema PDV presente em cada loja que se comunica com o sistema *Enterprise Resource Planning* (ERP) centralizado, ou seja, o sistema PDV das lojas está em constante comunicação com o ERP localizado no centro administrativo da rede de lojas STORE. Além do sistema ERP centralizado na matriz da rede de lojas STORE, o sistema PDV das lojas se comunica de forma síncrona com outros sistemas satélites que fazem parte do grupo de empresas o qual a rede de lojas STORE faz parte. Estas comunicações normalmente são para o recebimento de parcelas de contratos destas outras empresas do grupo, ou ainda para a realização de venda de produtos específicos.

A rede de lojas STORE realiza a venda de produtos mercantis como, por exemplo, produtos elétricos, eletrônicos e móveis. Além disso, também oferta para seus clientes a venda de serviços como garantia estendida e título de capitalização, os quais ao critério do cliente podem ser recebidos de forma unificada na mesma compra, junto aos produtos mercantis.

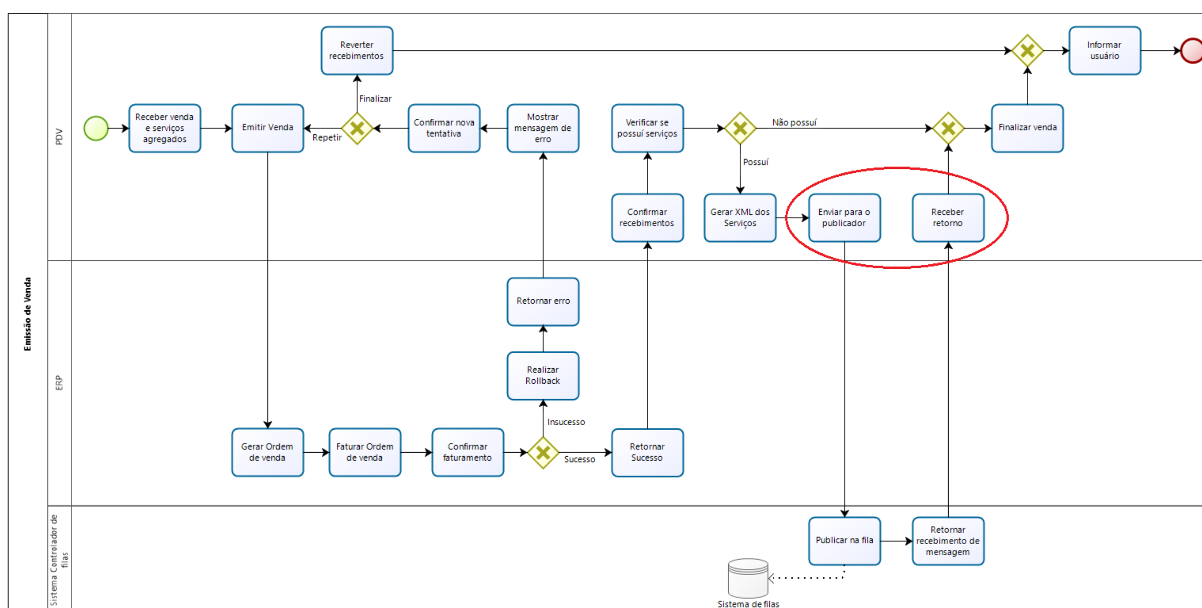
Nos primórdios do processo, era ofertado inicialmente apenas o serviço de garantia estendida na modalidade de pós-venda para os clientes. Porém, a empresa verificou a necessidade de se ofertar também de maneira unificada, visando um atendimento do cliente de uma forma mais eficiente. O processo de unificação de recebimento foi feito primeiramente com o serviço de garantia estendida. O título de capitalização foi uma oportunidade de negócio que surgiu posteriormente ao serviço de garantia estendida. O processo de geração do bilhete do título de capitalização é semelhante à geração do bilhete de garantia, desta forma utiliza a mesma estrutura lógica de comunicação entre os sistemas envolvidos, semelhante à venda da garantia estendida.

No desenvolvimento da unificação do recebimento de garantia estendida, optou-se por utilizar um sistema de filas para realizar o faturamento da venda de serviço, uma vez que esta deve estar atrelada ao documento de venda do produto, e o seu faturamento ocorre em um sistema satélite fora da estrutura do PDV e do ERP. Esta escolha foi feita com o intuito de evitar possíveis erros de comunicação com os sistemas que realizam o faturamento das vendas de serviços, erros que podem ocorrer devido a sobrecarga ou queda de comunicação. Além disso, também considerou-se que o faturamento da venda de garantia só pode ser realizado após a emissão da venda dos produtos, já que está atrelado ao documento de venda do produto que

está sendo segurado.

O fluxo do processo atual de venda pode ser verificado através da Figura 9, o qual mostra o envio de mensagem para um sistema controlador de filas de mensagem, via uma chamada *Web Service*. Mas que não existe nenhum tratamento específico no caso de se ter algum problema no enviou de mensagens ou no retorno do envio das mensagens para o sistema controlador de filas. A escolha desta arquitetura, sem um tratamento de erro no envio das mensagens para o controlador de filas, se deve pelo fato de que, neste ponto do processo, não se pode mais cancelar o documento da venda dos produtos mercantis.

Figura 9 – Fluxo atual do processo de emissão de venda, sem tratamento para eventuais erros no envio de mensagem para fila.



Fonte: O Autor (2023).

Na arquitetura mostrada na Figura 9, o sistema PDV se comunica com o ERP, para gerar e faturar a ordem de venda, num processo atômico, ou seja, se em alguma parte do processo der algum erro, este pode ser desfeito totalmente. Posteriormente, o PDV publica as mensagens no *message broker* por meio de um *Web Service*. O *message broker* é responsável pelo armazenamento das mensagens na respectiva fila e seu processamento, que ocorre através da comunicação com outro sistema para a geração da venda de serviços.

A Figura 10 mostra um trecho do algoritmo em *C#* que o sistema PDV utiliza para efetivar uma venda de serviço de garantia, através do envio de mensagem para o *message broker*. Verifica-se que na linha 346 o código aguarda um retorno do método chamado, sem realizar o bloqueio do cliente PDV, assim como também é possível observar que se tiver alguma exceção na execução do método chamado, o tratamento de erro na linha 350 do código apenas irá gravar no visualizador de eventos do *Windows* do computador que está executando o sistema PDV, não travando os processos seguintes.

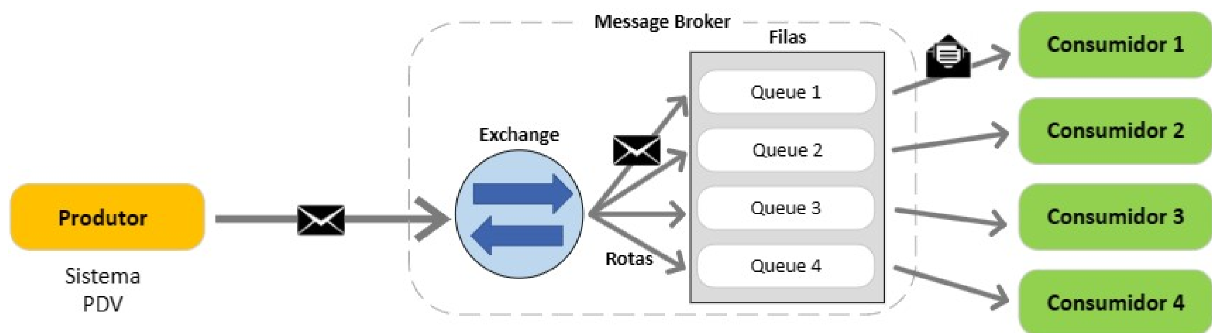
Figura 10 – Trecho de código sem tratamento de erro adequado.

```
346                     await CallQueueWarrantyEffective(warrantyEffectiveModel);
347                 }
348                 catch (Exception ex)
349                 {
350                     ApplicationLog.Log(typeof(FiscalDocumentModel).Name, ex.Message, LSRetailPosis.LogTraceLevel.Error);
351                 }
```

Fonte: Adaptado pelo Autor (2023).

A Figura 11 mostra a arquitetura atual de envio de mensagens para as filas de processamento. O PDV faz o papel de produtor enviando mensagens para o *message broker*, indicando a rota que a mesma pertence. Cada rota corresponde à uma fila, desta forma são necessárias inúmeras filas pois cada tipo de venda corresponde à uma fila especifica. Quando a mensagem chega na estrutura de fila, o *Message Broker* classifica e armazena a mensagem em sua fila de destino, até que seja processada (consumida) por outro sistema consumidor da respectiva fila.

Figura 11 – Arquitetura atual de envio de mensagens para filas.



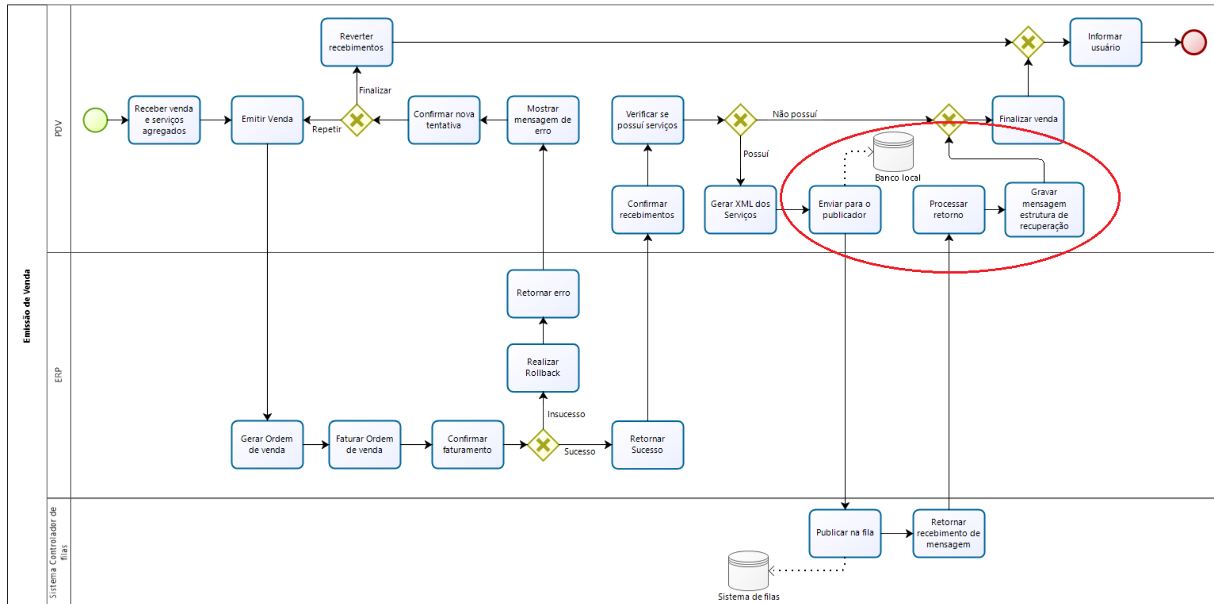
Fonte: O Autor (2023).

Este tipo de arquitetura possibilita a troca de mensagem entre vários sistemas diferentes, além de oferecer uma maior disponibilidade de operação, em relação ao caso em que a comunicação entre os sistemas fosse realizada de forma direta. É possível verificar na Figura 11 que o sistema produtor consegue enviar mensagens para várias filas e sistemas diferentes, sem se preocupar com a complexidade de comunicação entre o sistema produtor e o consumidor, pois toda esta complexidade fica a cargo do *message broker*.

3.2 MELHORIA NO PROCESSO ATUAL

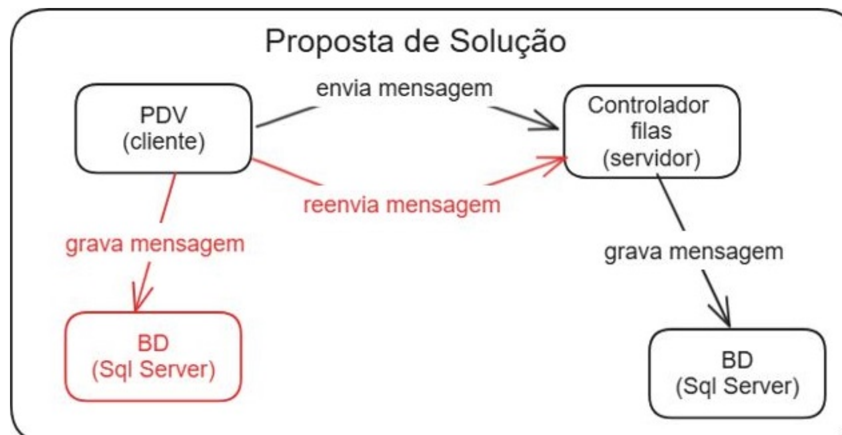
A proposta de solução contempla uma rotina que irá gravar todas as mensagens enviadas para o *message broker* numa estrutura de armazenamento. A Figura 12 mostra em que ponto do fluxo do processo esta nova rotina deve ser inserida para tratar os eventuais erros que não recebem o devido tratamento, no processo atual do PDV. Neste novo fluxo conforme mostra a Figura 13, independentemente de sucesso ou não, a mensagem enviada para o sistema controlador de filas será gravada na estrutura (tabela) de recuperação de mensagem com um *status* de sucesso ou não do envio da mensagem para a fila.

Figura 12 – Fluxo proposto do processo de emissão de venda, com tratamento para eventuais erros no envio de mensagem para fila.



Fonte: O Autor (2023).

Figura 13 – Proposta de solução com uma estrutura de armazenamento de mensagens enviadas para o controlador de filas.



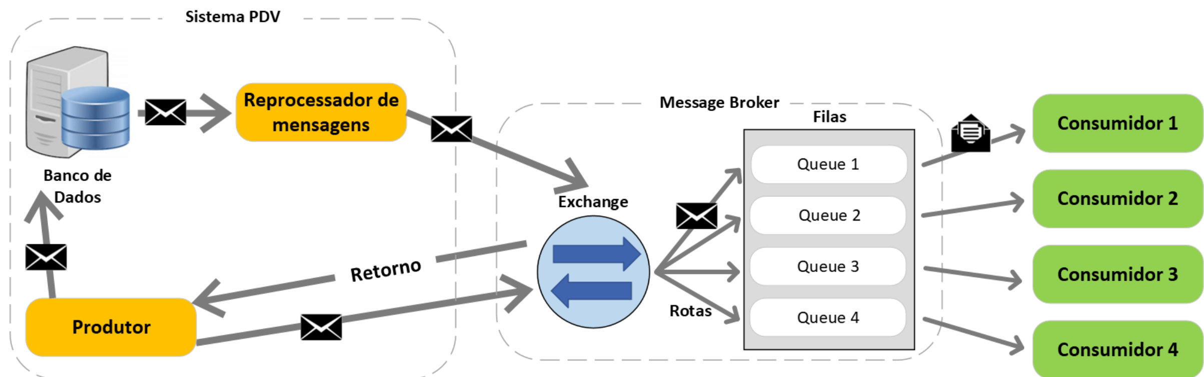
Fonte: O Autor (2024).

Além da estrutura de armazenamento, a solução propõe a criação de outra rotina, à qual deverá verificar as mensagens que não obtiveram sucesso de envio e realizar o reenvio das mesma para o sistema controlador de filas. Esta rotina será implementada num primeiro momento, conjuntamente com as rotinas de fechamento do dia, à qual é a última operação executada pela loja no sistema PDV ao final de seu expediente, porém a mesma pode ser implementada em outros locais do sistema PDV, com o intuito dos reprocessamentos acontecerem com uma maior frequência, a medida que o usuário acessa outras funcionalidades do sistema.

A Figura 14 mostra a proposta da nova arquitetura de envio de mensagens para as filas de mensagens. Nesta nova arquitetura, em relação à arquitetura da Figura 11 apresentada da

Seção 3.1, está se considerando um tratamento mais apropriado para o retorno do envio da mensagem para o *message broker*. Na arquitetura de solução proposta, o sistema PDV irá salvar numa estrutura de armazenamento de um banco de dados local as mensagens enviadas para o *message broker*, com o seu respectivo *status* de envio. Quando a rotina de reprocessamento de mensagens for executada, então irá verificar as mensagens que não tiveram sucesso de enviou para o *message broker* desta foram reenviando as mesmas. O fato de se utilizar um banco de dados local não trará problemas de comunicação.

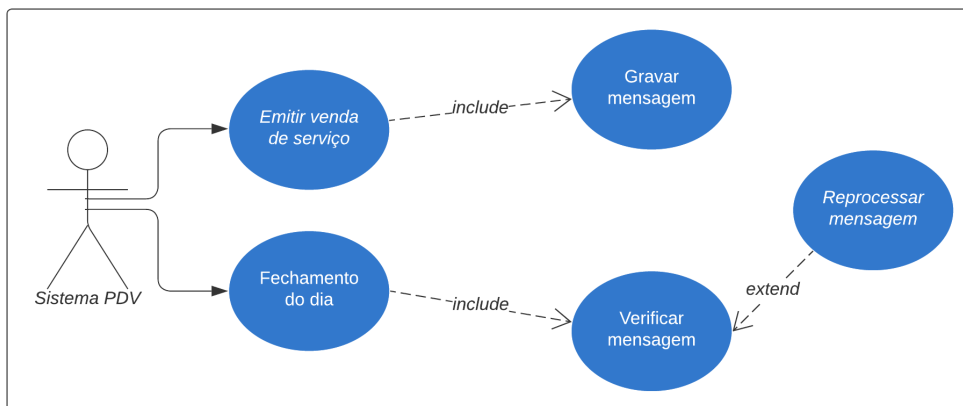
Figura 14 – Nova arquitetura de envio de mensagens para filas.



Fonte: O Autor (2023).

A Figura 15 evidencia as novas funcionalidades que serão agregadas ao sistema PDV, a fim de solucionar o problema de perda de informação nas trocas de mensagens.

Figura 15 – Casos de uso das novas funcionalidades do sistema PDV.

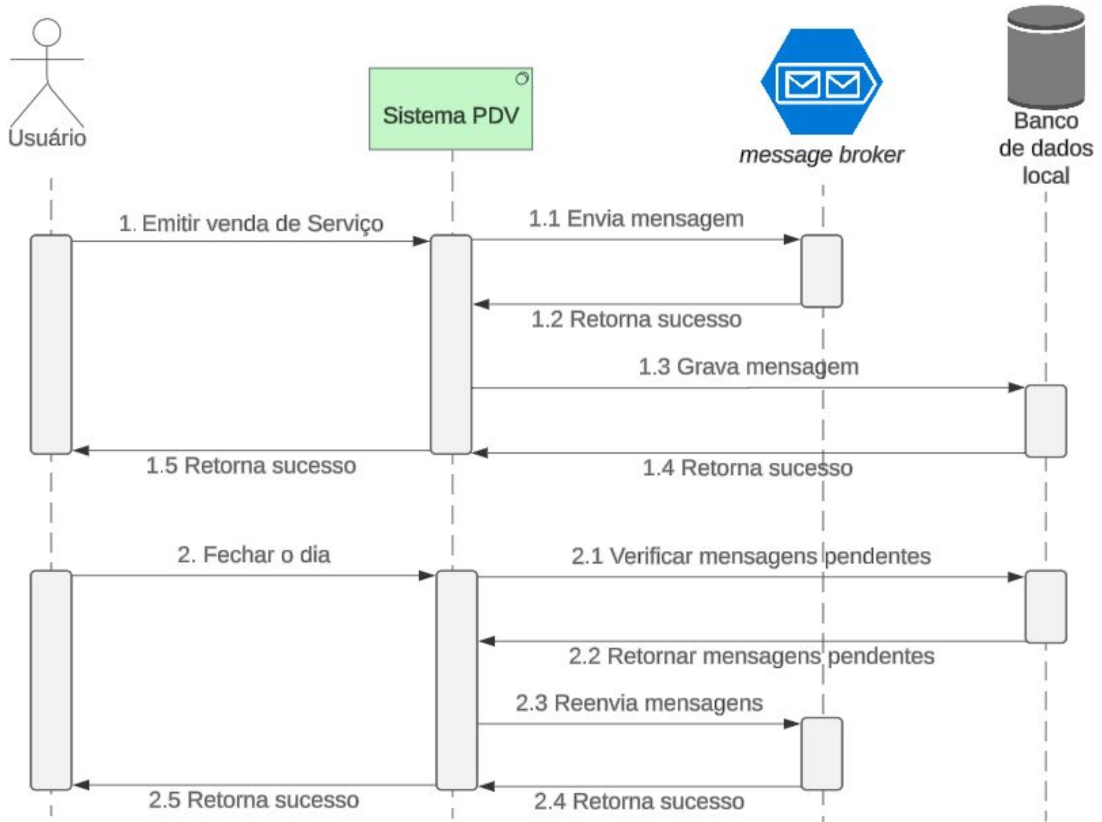


Fonte: O Autor (2023).

De acordo com o diagrama de sequência da Figura 16, toda a vez que o sistema PDV emitir uma venda de serviço, irá gravar a mensagem enviada ao *message broker*, na estrutura de recuperação, com o seu respectivo *status* de envio. A última operação executada no sistema PDV pelo usuário ao final do expediente, é o fechamento do dia, desta forma conjuntamente com este processo serão verificadas as mensagens criadas na estrutura de recuperação e as mensagens que não obtiveram exito no envio, serão reprocessadas. Logo, o *status* de envio de cada mensagem

enviada será atualizado para o novo *status* de envio, desta forma se o reprocessamento não obteve sucesso, a mensagem será reprocessada novamente numa próxima execução da rotina de reprocessamento de mensagens.

Figura 16 – Diagrama de sequência das novas funcionalidades do sistema PDV.



Fonte: O Autor (2024).

4 DESENVOLVIMENTO DA PROPOSTA DE SOLUÇÃO

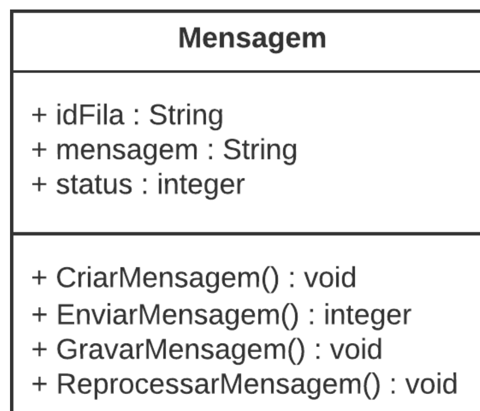
Neste capítulo será abordado o desenvolvimento do aplicativo para simulação da proposta de solução para o problema de pesquisa do presente trabalho. Serão abordados aspectos relacionados à arquitetura de software e ferramentas utilizadas, tanto para o desenvolvimento, quando para os teste de validação da proposta de solução.

4.1 DESCRIÇÃO DO PROTÓTIPO SIMULADOR

Em virtude da complexidade do sistema PDV, para validar a proposta de solução, foi criado um protótipo desenvolvido em *Java* que simula o fluxo de envio de mensagens para o *message broker*, com a mesma lógica que são enviadas as mensagens no processo de venda do sistema PDV no ponto cujo problema está sendo explorado neste trabalho. A cada envio de mensagem realizado através do simulador, o protótipo irá simular o envio de mensagens considerando o novo processo com a estrutura de recuperação, para fins de comparação de resultados, entre processo atual e o proposto neste trabalho. A comparação entre o processo atual e a proposta pode ser feita através das mensagens que serão reprocessadas pela estrutura de recuperação, visto que no processo atual, estas mensagens seriam perdidas.

O protótipo desconsiderou toda a complexidade da operação de venda, desta forma conforme o diagrama de classes da Figura 17, o protótipo possuí de uma única classe de envio de mensagens, que simula através de seus métodos o envio de mensagens pelo novo processo proposto.

Figura 17 – Diagrama de classes.

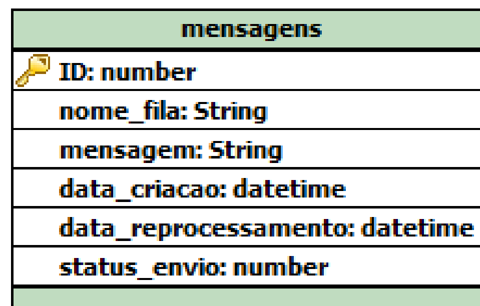


Fonte: O Autor (2023).

A classe **Mensagem** contém os métodos responsáveis por criar e enviar mensagens para o *message broker*, além dos métodos responsáveis pela gravação e reprocessamento das mensagens na estrutura de recuperação de mensagens proposta neste trabalho.

Conforme abordado na Seção 3.2, a proposta de solução do problema descrito na Seção 1.1 contemplou a criação de uma tabela, no banco de dados local da loja de varejo, para armazenamento das mensagens envidadas para o controlador de filas e seus respectivos *status* de envio. A Figura 18 mostra o modelo lógico da referida tabela.

Figura 18 – Modelo lógico da tabela para armazenamento das mensagens enviadas para o controlador de filas.



Fonte: O Autor (2023).

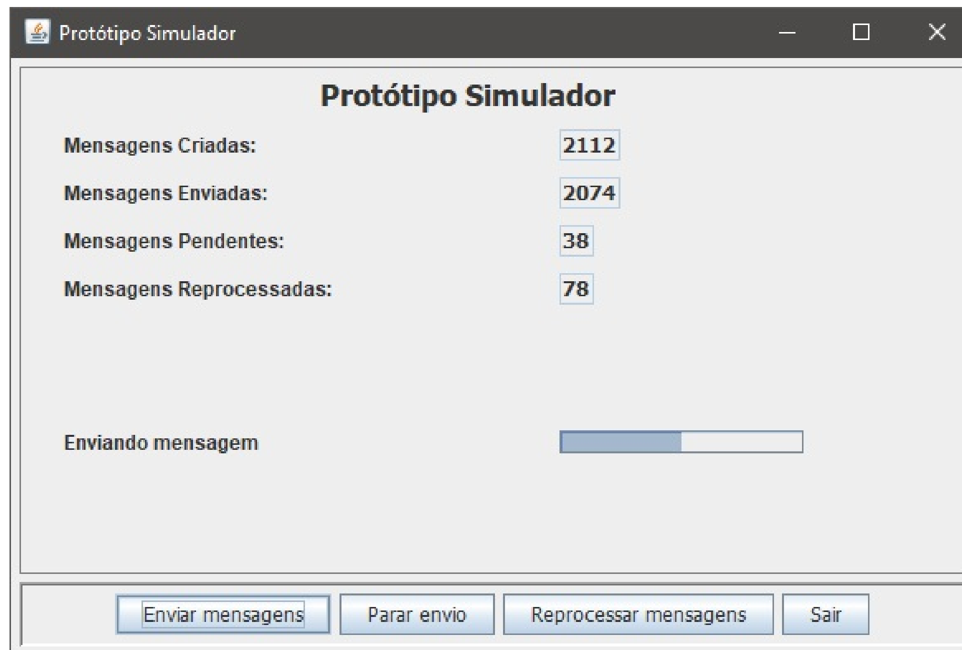
A tabela proposta para a solução é composta por um identificador (ID) do registro inserido na tabela, além do nome da respectiva fila a qual a mensagem pertence, também o *status* de envio da mensagem para o controlador de filas, além da data e hora de criação do registro na tabela e a data e hora do último reprocessamento do registro. O *status* de envio será representado por dois valores numéricos. O valor "1"irá representar sucesso do envio da mensagem para o controlador de fila e o valor "0"o insucesso no envio da mensagem.

O protótipo foi criado para testar a proposta de solução para o problema de perda de mensagens durante o envio para um *Web Service*, conforme discutido neste trabalho. Embora o problema abordado se destaca num sistema com uma estrutura mais complexa, o protótipo foi desenvolvido apenas para simular a solução proposta. Ele inclui funcionalidades para envio, armazenamento e reprocessamento de mensagens não enviadas. No entanto, não abrange toda a complexidade do sistema mencionado, focando apenas na parte específica do problema em análise.

A Figura 19 mostra a tela do protótipo correspondente ao aplicativo cliente composta por quatro botões, o botão **Enviar mensagens** é responsável em criar mensagens e enviar uma após a outra para o aplicativo servidor via uma chamada *Web Service* SOAP e gravar as mensagens e seus *status* de envio numa tabela de histórico de envio de mensagens pertencente ao aplicativo cliente.

O botão **Parar envio** é responsável em sinalizar a rotina de criação e envio de mensagens para finalizar a simulação de envio de mensagens para o aplicativo servidor. O botão **Reprocessar mensagens** verifica as mensagens que não obtiveram exito no envio e realiza o reenvio das mesmas para o aplicativo servidor. O botão **Sair** finaliza a execução do aplicativo cliente. À medida que as simulações de envio ou reenvio de mensagens vão sendo executa-

Figura 19 – Tela do protótipo simulador desenvolvido.



Fonte: O Autor (2024).

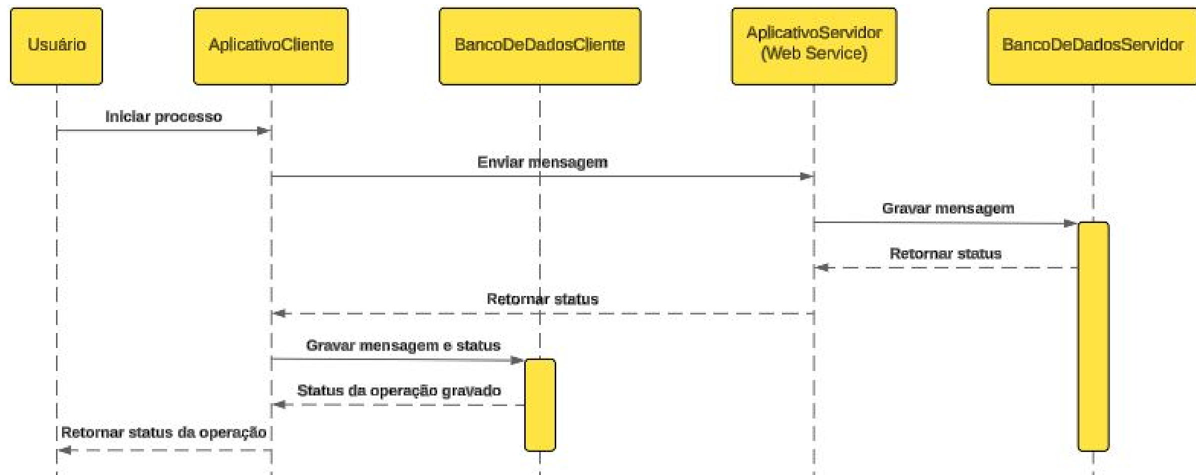
das, o protótipo atualiza os valores referentes as quantidades de mensagens criadas, enviadas, pendente e reprocessadas na tela.

O caminho dos dados, ou seja, das mensagens pode ser descrito por meio de um diagrama de sequência, conforme mostra a Figura 20. Ao ser iniciado o processo pelo usuário através dos botões de envio e reenvio de mensagens, o aplicativo cliente envia as mensagens para o aplicativo servidor por meio do serviço *Web Service*, por sua vez o aplicativo servidor grava a mensagem recebida em seu banco de dados e retorna o *status* da operação para o aplicativo cliente. O aplicativo cliente ao receber o *status* do retorno da chamada *Web Service*, grava a mensagem e o *status* de envio em seu banco de dados, retornando o *status* da operação para o usuário por meio da atualização dos indicadores na tela do aplicativo cliente.

As classes de envio e de reprocessamento de mensagens que são executadas ao acionar os botões **Enviar mensagens** e **Reprocessar mensagens** estendem a classe *Thread* do *Java*, desta forma o envio e reprocessamento de mensagens é executado por meio de *threads*, sendo esta estratégia adotada, para que a tela do simulador (aplicativo cliente) não fique estática e tenha a atualização das informações somente na finalização dos processos em execução, mas sim durante a execução dos mesmos. A atualização da barra de progresso da tela do aplicativo cliente da Figura 19, também utiliza o mesmo conceito de *threads* para ser atualizada de forma paralela aos procedimentos do fluxo principal do aplicativo cliente.

O Algoritmo 1 mostra uma parte da implementação do método *run* da classe no aplicativo cliente responsável em controlar o envio de mensagens para o aplicativo servidor. É possível verificar nas linhas 7 e 16 do trecho de código, a chamada para iniciar uma outra *thread*

Figura 20 – Diagrama de sequência demonstrando o fluxo das mensagens na estrutura que foi criada para o protótipo simulador.



Fonte: O Autor (2024).

diferente da *thread* da que já esta em execução no processo principal, para atualizar a barra de progresso enquanto a *thread* do processo principal cria, envia e grava no banco de dados local a mensagem criada.

Algoritmo 1 – Método *run* em código *Java* da classe do aplicativo cliente responável em enviar mensagens para o aplicativo servidor.

```
1  public void run() {
2      MensagemDao mensagemDao = new MensagemDao();
3      while (executando) {
4      try {
5          atualizaProgressBarThread =
6              new AtualizaProgressBarThread(pFormulario,"Criando␣mensagem");
7          atualizaProgressBarThread.start();
8          mensagem = new Mensagem();
9          mensagem.criarMensagem();
10         atualizaProgressBarThread.join();
11         pFormulario.txtCriadas.setText(String.valueOf(
12             mensagemDao.getQtdMensagensCriadas()));
13
14         atualizaProgressBarThread =
15             new AtualizaProgressBarThread(pFormulario,"Enviando␣mensagem");
16         atualizaProgressBarThread.start();
17         mensagem.setStatus(mensagem.enviarMensagem(mensagem));
18         mensagem.gravarMensagem();
19         atualizaProgressBarThread.join();
```

Fonte: O Autor (2024).

4.2 DESCRIÇÃO DA ARQUITETURA DO PROTÓTIPO

O protótipo foi implementado com o objetivo de realizar testes a fim de verificar a validade da proposta de solução em um ambiente acessível e isolado de todas as outras complexidades do conjunto de sistemas onde o problema de pesquisa se apresenta. Para alcançar esse objetivo, foi necessário implementar duas aplicações. Uma delas é responsável pela criação, envio e reprocessamento de mensagens por meio de requisições SOAP para a outra aplicação. Uma das aplicações é um *Web Service*[1] SOAP encarregado de receber as mensagens e inseri-las em uma estrutura de filas. Esta aplicação foi desenvolvida com o propósito de simular um servidor remoto com o papel de um *message broker*, mas somente com a responsabilidade de receber e armazenar as mensagens em suas respectivas filas de processamento. Os consumidores das filas não foram desenvolvidos, pois não fazem parte do escopo do protótipo da proposta de solução, assim como os mesmos não fazem parte do problema central tratado neste trabalho, visto que a existência ou não destes consumidores, não influenciam no problema discutido.

Em termos de um padrão de distribuição de tarefas e propósito, conforme Figura 21 podemos considerar que o desenvolvimento destas duas aplicações adotou o padrão arquitetural cliente-servidor. A aplicação *desktop*, responsável pela criação, envio e reprocessamento das mensagens, desempenha o papel de cliente, enquanto o *Web Service* atua como servidor. Este modelo arquitetural facilita a separação de responsabilidades e a interação entre os componentes da aplicação, sendo aplicável em vários contextos, deste redes de computadores, aplicações *web* e sistemas distribuídos.

Figura 21 – Representação dos aplicativos utilizando a arquitetura cliente-servidor.

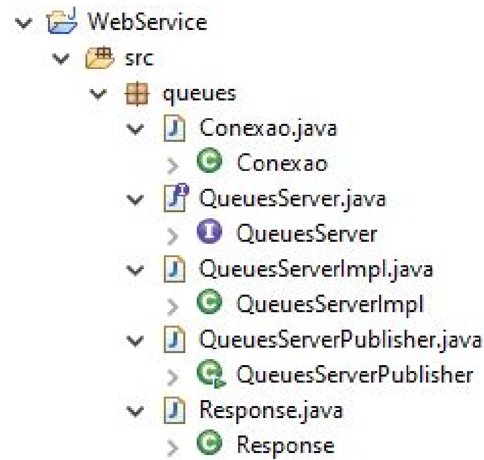
Fonte: O Autor (2024).

Em termos de *design* de *software* relacionado a organização e estrutura interna da aplicação, a aplicação cliente, responsável pelo envio de mensagens, seguiu o padrão de arquitetura *Model-View-Controller* (MVC) para separar as responsabilidades de cada parte da aplicação, assim como também deixar o código mais claro e facilitar a manutenção do mesmo. Segundo Alves (2015), MVC é um padrão de *design* de *software* amplamente reconhecido que separa a lógica de negócios da apresentação dos dados. Fortemente associado ao paradigma da programação orientada a objetos, facilitando a reutilização de código e a construção de *frameworks*, promovendo a organização e manutenção eficiente dos projetos de *software*. O aplicativo servi-

[1] Código fonte disponível em <https://github.com/emenzen/TCC-II-WebService>.

dor, por conter menos partes de código, não seguiu nenhum padrão de arquitetura para agrupar suas classes, ficando conforme mostra a Figura 22 todas as classes num mesmo pacote (*queues*).

Figura 22 – Organização estrutural do código do aplicativo servidor.



Fonte: O Autor (2024).

O aplicativo servidor foi implementado através de quatro classes e uma interface, cada um destes componentes da aplicação, possui funcionalidade específica.

- **QueuesServer**: interface responsável em definir as classes e métodos que serão expostos como operações no serviço do *Web Service*;
- **Conexao**: classe responsável pela conexão com o banco de dados do aplicativo servidor;
- **QueuesServerImpl**: classe responsável pela implementação da interface, sou seja, implementa os métodos dos *Web Service* definidos na interface;
- **QueuesServerPublisher**: classe para publicar e executar o serviço *Web Service*;
- **Response**: classe para representar e encapsular o objeto de retorno referente à mensagem de retorno enviada do aplicativo servidor para o aplicativo cliente após receber uma requisição.

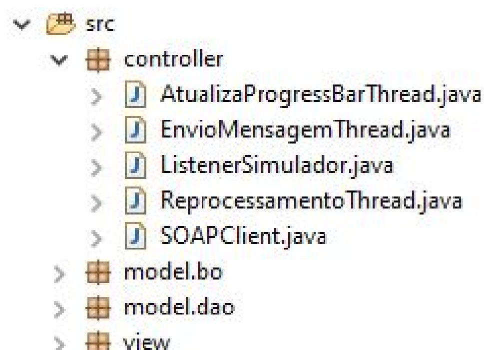
Em relação ao aplicativo cliente[2], conforme já descrito anteriormente a organização e estrutura interna da aplicação seguiu o padrão MVC, ou seja, o código do aplicativo cliente foi estruturado em várias camadas, segundo o propósito de cada parte. Estas camadas serão descritas nos tópicos à seguir.

[2] Código fonte disponível em <https://github.com/emenzen/TCC-II-Simulador>.

4.2.1 Camada de Controle (Controller)

A camada de controle interage com as camadas de visão e modelo recebendo as ações do usuário e coordenando as interações entre estas camadas. Ela interpreta as ações do usuário e atualiza o modelo e a visão. Em algumas situações a camada pode também conter lógica de negócio para garantir a consistência dos dados.

Figura 23 – Camada de controle.



Fonte: O Autor (2024).

A Figura 23 mostra as classes que fazem parte da camada de controle e que são responsáveis pela atualização da tela, assim como também pela comunicação com outros sistemas externos.

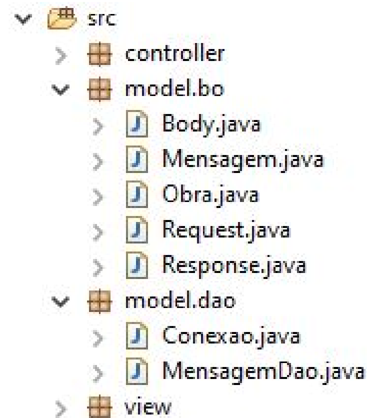
- **AtualizaProgressBarThread**: classe responsável em controlar a atualização da barra de progresso de envio e reprocessamento de mensagens, por meio de *threads* para não deixar a tela da aplicação congelada;
- **EnvioMensagemThread**: classe responsável em controlar a criação e envio de mensagens para o *Web Service* através de *threads*;
- **ListenerSimulador**: classe *listener* responsável em escutar e responder aos eventos da interface de usuário, permitindo que as diferentes partes do aplicativo interajam de forma flexível;
- **ReprocessamentoThread**: classe responsável pelo reprocessamento de envio de mensagens para o *message broker* através de *threads*;
- **SOAPClient**: classe que contém a lógica de comunicação com o *Web Service*, responsável em receber o *payload* da mensagem e envia o mesmo para o *message broker* através de uma requisição utilizando o método *Post* do *Web Service*.

4.2.2 Camada de Modelo (Model)

A camada de modelo é responsável pela representação dos dados da aplicação e das regras de negócio. No presente trabalho esta camada foi implementada sendo composta por

duas subcamadas, a subcamada *Business Objects* (BO) dos objetos de negócio responsável por encapsular a lógica de negócio e a subcamada *Data Access Object* (DAO) responsável em encapsular a lógica de acesso aos dados e estabelecer e gerenciar conexões com banco de dados ou outro sistema de armazenamento de dados, conforme mostra a Figura 24.

Figura 24 – Camada de modelo composta pelas subcamadas BO e *DAO*.



Fonte: O Autor (2024).

Desta forma todas as classes constantes na subcamada BO representam um objeto e encapsula seus atributos e métodos, enquanto que, a classe **Conexao** da camada DAO é responsável em conter a lógica para estabelecer a conexão com o banco de dados. A classe **MensagemDao** possui as operações básicas *Create, Read, Update, Delete* (CRUD) no banco de dados.

O banco de dados escolhido para compor o protótipo da solução foi o *MySQL*[3]. Entre os fatores que levaram à esta escolha para o desenvolvimento do protótipo, está a questão de que o *MySQL* é um software de código aberto distribuído sob a licença *General Public License* (GPL) e sua versão comunitária (*open source*) é gratuita em comparação ao banco de dados utilizado pelo sistema do problema de pesquisa que é o *SQL Server*, produto da *Microsoft*[4], o qual querer licenciamento pago para o seu uso. Outro fator importante é que o *MySQL* é um produto multiplataforma de fácil usabilidade suportando uma variedade de tipos de dados além de possuir uma vasta documentação.

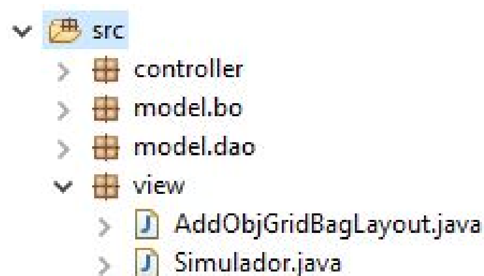
4.2.3 Camada de Visão (View)

A camada de Visão é responsável pela interface do usuário, onde são exibidos os dados e os elementos gráficos do aplicativo, além de interagir com o usuário.

Nesta camada da Figura 25, a criação dos componentes gráficos é de responsabilidade da classe **Simulacao**, a classe **AddObjGridBagLayout**, contém um método estático para adi-

[3] <https://www.mysql.com>
[4] <https://www.microsoft.com>

Figura 25 – Camada de visão.



Fonte: O Autor (2024).

cionar componentes num *JPanel* simplificando a adição de componentes sem a repetição de código.

4.3 AMBIENTE DE DESENVOLVIMENTO

O desenvolvimento do protótipo da proposta de solução foi desenvolvido em linguagem *Java* no ambiente *Windows* através da ferramenta *Eclipse IDE for Java Developers*[5]. A escolha da linguagem *Java* se deu devido à familiaridade do autor do presente trabalho com a linguagem, assim como também pela extensa documentação existente. Os principais pacotes utilizados para o desenvolvimento do protótipo estão descritos a seguir.

- **java.net:**[6] pacote *Java* que fornece classes e interfaces que suportam operações de rede, comunicação por meio de protocolos de rede e acesso a recursos remotos, sendo um pacote para desenvolver aplicativos que precisam interagir através da rede, seja para fazer requisições HTTP, estabelecer conexões *Transmission Control Protocol/Internet Protocol* (TCP/IP) ou enviar e receber dados por meio de *sockets*.
- **java.sql:**[7] pacote *Java* que fornece classes e interfaces para acessar e manipular bancos de dados relacionais usando o *Java Database Connectivity* (JDBC), sendo possível executar consultas, atualizações assim como também outras operações de bando de dados.
- **javax.jws:**[8] pacote *Java* que oferece suporte ao desenvolvimento de serviços da web seguindo os padrões baseados em XML e o protocolo SOAP.
- **javax.swing:**[9] pacote *Java* de componentes gráficos que possibilita a criação de interfaces de usuário interativas em aplicativos *Java* oferecendo uma alternativa mais flexível e moderna em relação ao kit de ferramentas gráficas *Abstract Window Toolkit* (AWT) original do *Java*.

[5] <https://www.eclipse.org/>
[6] <https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html>
[7] <https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html>
[8] <https://docs.oracle.com/javase/8/docs/api/index.html?javax/jws/package-summary.html>
[9] <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>

- **javax.xml:**[10] pacote *Java* para lidar com XML, fornecendo através de suas classes e interfaces um suporte nativo para processamento, validação, manipulação e geração de documentos XML.

- **org.w3c.dom:**[11] este pacote em *Java* faz parte da API *Document Object Model* (DOM) que contém interfaces e classes que permitem o processamento e manipulação de documentos XML ou HTML seguindo os padrões estabelecidos pela *World Wide Web Consortium* (W3C).

4.4 DIAGRAMA DE CLASSES

No diagrama de classes da Figura 26 do aplicativo cliente são apresentados os pacotes *model*, *view* e *controller* e os subpacotes *BO* e *DAO* do pacote *model*, com suas respectivas classes contendo seu métodos e atributos. Devido à aplicação específica de cada classe, não se apresenta com exceção das classes *Body* e *Mensagem* herança que é um conceito fundamental na Programação Orientada a Objetos (POO), que permite a criação de classes baseadas em outras classes existentes. Neste contexto, somente evidenciamos associações unidirecional, como no caso da associação entre as classes *Mensagem* e *Request* do subpacote *bo*, ou seja, nesta situação uma classe tem o conhecimento da outra e com ela interage, mas sem herdar atributos e métodos.

O diagrama de classes da Figura 27 demonstra as classes do aplicativo servidor dispostas num único pacote *queues*. A classe *QueuesServerImpl* herda os atributos e métodos da super classe *Conexao*, que contém os métodos comuns de conexão com o banco da dados, além de implementar a interface *QueuesServer*.

4.5 MODELO DE DADOS

A implementação do protótipo da proposta de solução, conforme descrito anteriormente, fez o uso de duas aplicações cada uma com seu respectivo banco de dados. A aplicação cliente, manteve a proposta inicial em termos de estrutura de banco de dados, contendo ainda apenas uma única tabela para a gravação das mensagens enviadas para o *Web Service* e seus respectivos *status*, conforme mostra a Figura 18 já discutida na Seção 4.1.

Na arquitetura do caso de estudo, as aplicações cliente e servidor rodam com hardware separados não compartilhando recursos de máquina e nem de banco de dados, desta forma cada uma das aplicações mantém uma estrutura de hardware e modelo de dados independente uma da outra. No desenvolvimento do protótipo de solução manteve-se a mesma premissa, desta forma para o aplicativo servidor, foi criada uma nova tabela conforme mostra a Figura 28, a qual grava as mensagens enviadas via *Web Service* pelo aplicativo cliente em suas respectivas filas assim

[10] <https://docs.oracle.com/javase/8/docs/api/javax/xml/package-summary.html>
[11] <https://docs.oracle.com/javase/8/docs/api/index.html?org/w3c/dom/package-summary.html>

Figura 26 – Diagrama de classes do aplicativo cliente.

model

bo

Body
+getsPost(): Mensagem
+setPost(post: Mensagem): void

Obra
-tipo: String
-titulo: String
-autor: String
-ano: int
+getTipo(): String
+setTipo(tipo: String): void
+getTitulo(): String
+setTitulo(titulo: String): void
+getAutor(): String
+setAutor(autor: String): void
+getAno(): int
+setAno(ano: int): void

Mensagem
-id: int
-status: int
-queuename: String
-message: String
+getId(): int
+setId(id: int): void
+getStatus(): int
+setStatus(status: int): void
+getQueuename(): String
+setQueuename(queuename: String): void
+getMessage(): String
+setMessage(message: String): void
-geraMensagem(): String
+criarMensagem(): void
+enviarMensagem(mensagem): int
+gravarMensagem(): void
+reprocessarMensagem(): void

Request
~request: String
+getRequest(): String
+setRequest(mensagem: Mensagem): void

Response
-sucess: int
-message: String
+getSucess(): int
+setSucess(sucess: int): void
+getMessage(): String
+setMessage(message: String): void

dao

Conexao
+conectaBanco(): Connection
+desconectaBanco(c: Connection): void

MensagemDao
~con: Connection
+insert(mensagem: Mensagem): boolean
+update(mensagem: Mensagem): boolean
+getNaoEnviadas(): Mensagem[*]
+getQtdMensagensCriadas(): int
+getQtdMensagensEnviadas(): int
+getQtdMensagensReprocessadas(): int
+getQtdMensagensPendentes(): int

view

AddObjGridBagLayout
+restricoes: GridBagConstraints
+addObjetoGridBagLayout(pnl: JPanel, obj: Component, x: int, y: int, gridwidth: int): void

Simulador
+DPane: JDesktopPane
-pnlMain: JPanel
#pnlCenter: JPanel
+btnEmitirVendas: JButton
+btnPararEmissao: JButton
+btnFechamentoDia: JButton
+btnSair: JButton
+jProgressBar: JProgressBar
-lblMensagensCriadas: JLabel
-lblMensagensEnviadas: JLabel
-lblMensagensReprocessadas: JLabel
-lblMensagensPendentes: JLabel
+lblStatus: JLabel
-lblSimulacao: JLabel
+txtCriadas: JTextField
+txtEnviadas: JTextField
+txtReprocessadas: JTextField
+txtPendentes: JTextField
+main(args: String[*]): void

controller

SOAPClient
-retorno: Response
+post(request: Request): Response

EnvioMensagemThread
-mensagem: Mensagem
-pFormulario: Simulador
+executando: boolean = true
+run(): void

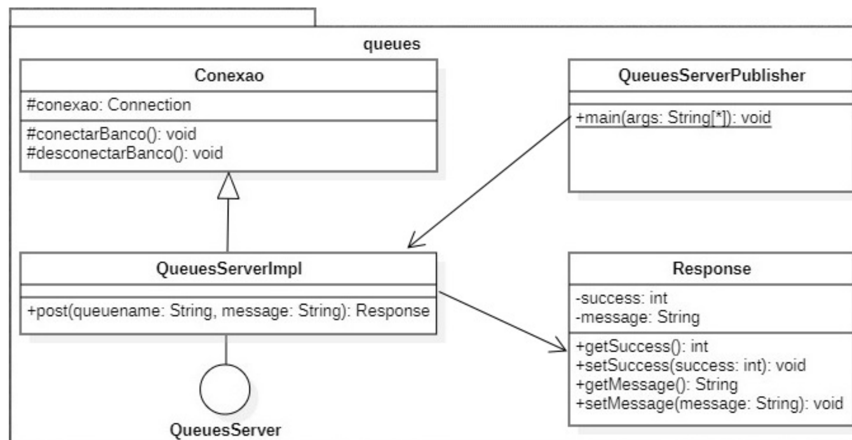
ReprocessamentoThread
-pFormulario: Simulador
+executando: boolean = true
+run(): void

AtualizaProgressBarThread
-pFormulario: Simulador
-texto: String
+run(): void

ListenerSimulador
-pFormulario: Simulador
+actionPerformed(e: ActionEvent): void

Fonte: O Autor (2024).

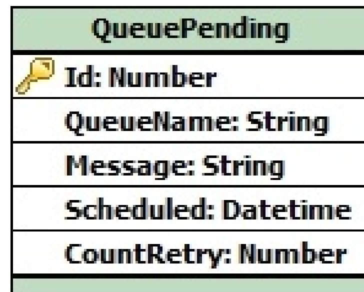
Figura 27 – Diagrama de classes do aplicativo servidor.



Fonte: O Autor (2024).

como outros campos de controle de fila que seriam utilizados pelos consumidores, porém os consumidores não fizeram parte do escopo do desenvolvimento da proposta de solução.

Figura 28 – Tabela de fila de mensagens do aplicativo servidor.



Fonte: O Autor (2024).

Na estrutura da tabela da Figura 28, podemos destacar os seguintes campos:

- **Id**: identificador do registro da tabela, sendo um campo autoincremental;
- **QueueName**: nome da fila de processamento à qual o registro pertence;
- **Message**: mensagem em formato XML ou JSON que deve ser processada pelo consumidor da fila;
- **Scheduled**: data e hora da criação do registro;
- **CountRetry**: número de vezes que o registro foi processado ou reprocessado pelo consumidor da fila.

4.6 CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO

Em termos visuais, pode-se destacar a questão de que a proposta inicial do aplicativo simulador considerava para a parte cliente uma tela com várias barras de progresso para representar as quantidades de mensagens criadas, enviadas, pendentes e reprocessadas. Já nos primeiros testes da versão inicial do aplicativo verificou-se que esta proposta não se mostrou eficiente para mostrar informações ao usuário, além de confusa, então, decidiu-se por um *layout* de tela mais limpo em que as informações de quantidades são mostradas apenas por números, além de conter apenas uma única barra de *status* com o intuito de mostrar a evolução do passo-a-passo da simulação.

Outro aspecto que se pode evidenciar é a questão da simulação de erro no envio para o aplicativo servidor. A primeira versão do *Web Service* foi testada numa máquina virtual Ubuntu[12] *linux* com um banco de dados PostgreSQL[13]. A abordagem neste situação considerou um terceiro aplicativo executando dentro da própria máquina virtual, responsável em enviar

[12] <https://ubuntu.com/>
[13] <https://www.postgresql.org/>

inúmeras requisições para o *Web Service* por meio de laço de repetição e criação de inúmeras *threads*.

Nos primeiros testes com o *Web Service* sem conexão com o banco de dados, a proposta para simulação de erro havia se mostrado satisfatória, porém com a inclusão da lógica de conexão com o banco de dados no *Web Service*, o mesmo começou a apresentar um travamento continuo bloqueando todas as conexões, sem voltar a funcionar verificou-se que no final este bloqueio era causado pelo banco de dados, pois a cada chamada para o *Web Service* se abria uma conexão com o banco de dados, sem fechar a mesma. Este e outros aspectos no código do *Web Service* foram melhorados, desta forma esta abordagem, passou a não mais apresentar as instabilidades desejadas para a simulação de erro.

Visto que a motivação inicial para se manter diferentes sistemas operacionais e gerenciadores de banco de dados entre os aplicativos cliente e servidor não existia mais, visto que não havia nenhuma motivo para se manter tanta heterogeneidade de sistemas operacionais e gerenciadores de banco de dados, passou-se então a considerar que o *Web Service* poderia rodar em um outro computador físico com o mesmo sistema operacional e gerenciador de banco de dados do aplicativo cliente. Desta forma, surgiu uma segunda abordagem para a simulação de erros, a qual consistia numa lógica no *Web Service*, conforme mostra o Algoritmo 2 em gerava um número aleatório entre 0 e 9, e caso o número gerado fosse igual à 5, então, o *Web Service* não se conectava com o banco de dados, forçando erro de conexão com o banco de dados, o qual era retornado ao aplicativo cliente, caso contrário o *Web Service* seguia seu funcionamento normalmente.

Algoritmo 2 – Código para gerar erro de conexão com banco de dados aleatoriamente.

```
1  Random random = new Random();
2  int number = random.nextInt(10);
3
4  if (number != 5)
5      conectarBanco();
```

Fonte: O Autor (2024).

Esta estratégia em forçar erro aleatoriamente através de lógica computacional, foi utilizada durante a finalização do desenvolvimento do aplicativo cliente e melhoria do código desenvolvido. Na versão final do simulado e para os testes apresentado neste trabalho, adotou-se para simular erros os testes de cargas através do software *SoapUI*, conforme já descrito na Seção 5.1. Esta última abordagem para a geração de erro, surgiu como resultado da procura de simular um ambiente similar ao ambiente do problema de pesquisa.

A solução proposta neste trabalho possui algumas limitações em termos de tempo de reprocessamento das mensagens que não tiveram sucesso no primeiro envio para o controlador de filas, visto que o processo de reprocessamento somente será iniciado paralelamente à atividades executadas pelo usuário. Esta espera no reprocessamento, pode ter impacto negativos em outas

atividades que se utilize a estrutura de recuperação proposta neste trabalho. Uma proposta futura, poderia utilizar uma lógica que pudesse reprocessar mensagens não enviadas a cada novo envio de mensagens para o controlador de filas.

5 METODOLOGIA E RESULTADOS DOS TESTES

Neste capítulo serão descritos como foram realizados os testes de validação da proposta de solução com o uso do aplicativo simulador.

5.1 AMBIENTE DE TESTE

Foram realizados testes para explorar a eficácia da solução proposta neste trabalho, testes com e sem sobrecarga no *Web Service*. Para que os testes fossem realizados numa estrutura similar à estrutura onde o problema de pesquisa ocorre, o *Web Service* desenvolvido no aplicativo servidor, foi pulicado em um computador diferente do aplicativo cliente. Em termos de banco de dados, foi configurado um banco de dados local em cada computador, de forma que cada aplicativo da simulação (cliente e servidor) interaja com seu banco de dados local, mantendo também neste quesito a similaridade com o problema de pesquisa.

Todos os cenários de testes tanto o com sobrecarga no *Web Service*, quanto o sem sobrecarregar o *Web Service*, foram realizados com o uso de dois computadores conectados via rede *Wirelles* de 150Mbps. O *Web Service* ficou rodando num computador recém formato com sistema operacional *Microsoft Windows 10 Professional* com processador *Intel Core2 Duo T5750 2.00GHz*, 4GB de memória *RAM* DDR2 e um disco sólido *SSD* de *120GB*. O aplicativo cliente, responsável pela geração e envio de mensagens, rodou num computador com sistema operacional *Microsoft Windows 10 Home* com processador *Intel Core i5 7200U 2.50GHz*, 16GB de memória *RAM* DDR4 e um disco sólido *SSD M.2* de *480GB*.

Foram realizados testes com o protótipo da solução, em 2 cenários distintos. Um cenário sem sobrecarregar o *Wev Service* e um segundo cenário com o uso do aplicativo *SoapUI*[1] para sobrecarregar requisições no *Web Service*, pois este aplicativo possui uma funcionalidade de testes de carga, no qual é possível simular diferentes tipos de carga. Para este trabalho, utilizou-se a estrategia *variance*. Nesta estratégia, a simulação pelo aplicativo *SoapUI* irá iniciar a simulação de carga com o número de *threads* inicialmente configurado e rodar a simulação durante um determinado intervalo de tempo também configurado. Dentro de cada intervalo de 60 segundos o número de *threads* irá aumentar e diminuir de acordo com o percentual configurado para a *variance* em relação ao valor inicial, retornando ao seu valor inicial ao final dos 60 segundos. Este comportamento pode ser verificado para um número inicial de 20 *threads* no Quadro 1 para vários valores de variância.

Para a realização dos testes com carga, de acordo com a Figura 29 foi configurado para o parâmetro *variance* o valor 0,5 que corresponde à uma variância de 50%, 20 para o número inicial de *threads* e um tempo de 120 segundos de simulação (parâmetros *Limit*). Na Figura 30

[1] <https://www.soapui.org/>

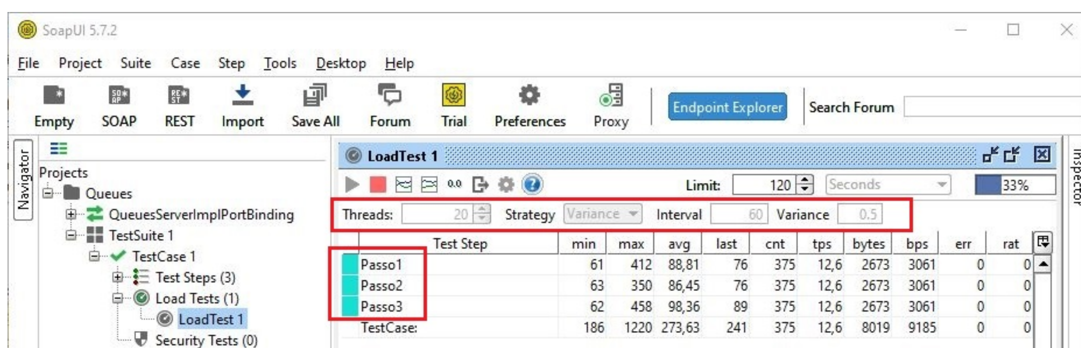
Quadro 1 – Número de *threads* ao longo do tempo, para vários valores de variância.

Nº threads	**Tempo (s)** / **Variância(%)**	0	15	30	45	60	75	90	105	120
	10	20	22	20	18	20	22	20	18	20
	25	20	25	20	15	20	25	20	15	20
	50	20	30	20	10	20	30	20	10	20
	80	20	36	20	4	20	36	20	4	20

Fonte: O Autor (2024).

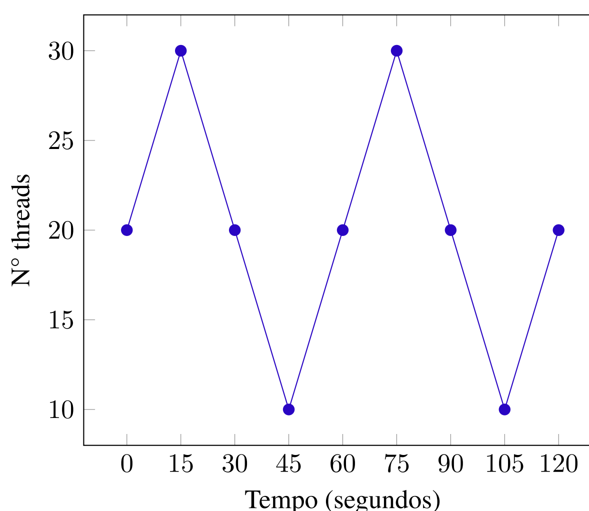
é possível evidenciar o comportamento da aplicação *SoapUI* para os respectivos parâmetros, visto que o número de *threads* aumenta de 20 para 30 *threads* nos primeiros 15 segundos, nos próximos 15 segundos retorna para 20 e continua à diminuir até 10 *threads* quando o tempo de simulação chega aos 45 segundos, retornando novamente para o valor inicial de 20 *threads* aos 60 segundos, então, este ciclo se repete até atingir os 120 segundos de simulação, o que pode também ser evidenciado no Quadro 1 gerando assim um gráfico similar aos dentes de uma serra.

Figura 29 – Configuração do *SoapUI*, considerando uma estratégia de carga variável.



Fonte: O Autor (2024).

Figura 30 – Número de *threads* criadas na simulação pelo *SoapUI* ao longo do tempo.



Fonte: O Autor (2024).

No primeiro cenário de testes utilizando-se somente o protótipo da solução, sem sobrecarregar o *Web Service*, foi acionada a funcionalidade de geração e envio de mensagens no aplicativo cliente por um tempo de 2 minutos e anotadas as quantidades de mensagens criadas e enviadas. Conforme o Quadro 2, este procedimento foi repetido por 5 vezes e em todas as vezes o aplicativo cliente criou e enviou 50 mensagens para o aplicativo servidor sem que nenhum dos envios desse erro, ou seja, todas as mensagens criadas nesta situação chegaram em seu destino.

Quadro 2 – Resultados do aplicativo cliente nas simulações sem sobrecarga no *Web Service*.

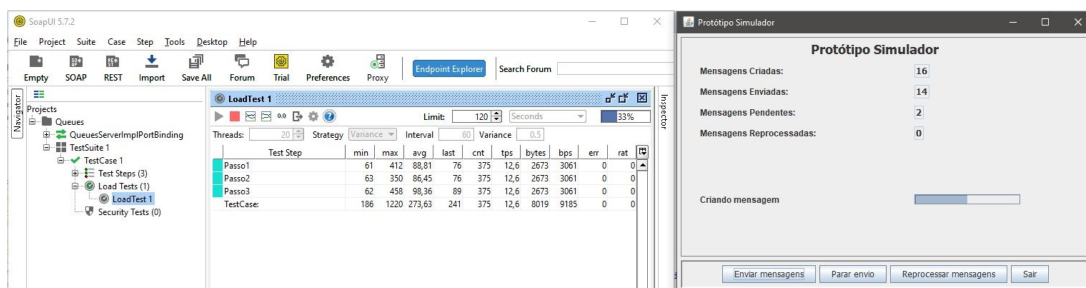
Mensagens	Simulação					Média
	1	2	3	4	5	
Criadas	50	50	50	50	50	50
Enviadas	50	50	50	50	50	50
Pendentes	0	0	0	0	0	0

Fonte: O Autor (2024).

O segundo cenário de testes, considerou testar a aplicação numa situação com sobrecarga de requisições para o *Web Service*. Para sobrecarregar o *Web Service* no aplicativo servidor, utilizou-se a funcionalidade de testes de carga do aplicativo *SoapUI*. Sendo o objetivo do desenvolvimento do simulador a validação da proposta de solução numa situação adversa, considerou-se conforme já mencionado anteriormente em se utilizar a estratégia de carga variável do aplicativo *SoapUI*, configurando-se um número inicial de 20 *threads*, intervalo de 60 segundos e variância igual a 0,5 (50%), para um limite de 120 segundos de simulação.

A Figura 31 mostra a criação e envio de mensagens para o servidor através do aplicativo cliente, em paralelo com a execução de teste de carga no aplicativo *SoapUI*, verifica-se que nesta situação não são todas as mensagens criadas que são enviadas com sucesso para o destino.

Figura 31 – Simulação de criação e envio de mensagens com sobrecarga no *Web Service*.



Fonte: O Autor (2024).

Este cenário de teste, também foi repetido por 5 vezes obtendo-se os resultados descritos no Quadro 3, referentes às mensagens criadas e enviadas com sucesso através do aplicativo cliente.

Devido à simulação de teste de carga realizada em paralelo com o aplicativo cliente através do *SoapUI*, foram recebidas pelo aplicativo servidor uma média de 6.973 mensagens em cada uma das simulações, conforme mostra o Quadro 4.

Quadro 3 – Resultados do aplicativo cliente nas simulações em paralelo com o aplicativo de teste de carga.

Mensagens	Simulação					Média
	1	2	3	4	5	
Criadas	51	51	51	51	51	51
Enviadas	45	44	43	45	45	44
Pendentes	6	7	8	6	6	7

Fonte: O Autor (2024).

Quadro 4 – Mensagens recebidas pelo aplicativo servidor durantes as simulações.

Mensagens	Simulação					Média
	1	2	3	4	5	
Recebidas	6.929	7.015	7.028	6.948	6.945	6.973

Fonte: O Autor (2024).

No término dos 120 segundos, o botão **Parar envio** era acionado e posteriormente o botão **Reprocessar mensagens**, desta forma o protótipo parava a criação e envio de mensagens e reprocessava o envio das mensagens pendentes, nesta situação conforme mostra a Figura 32 todas as mensagens pendentes de envio eram enviadas ao servidor, mostrando a eficacia da solução.

Figura 32 – Tela do aplicativo cliente após o reprocessamento das mensagens pendentes.

Fonte: O Autor (2024).

6 CONCLUSÃO

O sistema PDV da loja *STORE* envia mensagens para um *message broker* mas sem tratamento adequado de falhas de envio, o que causa perda irreparável de mensagens. Desta forma a loja *STORE* quer uma solução de fácil implementação e manutenção, a qual não tenha a necessidade de instalação de novos softwares ou ainda rotinas desconexas com os processos já realizados pelos usuários do sistema PDV.

Este trabalho consistiu em explorar uma proposta de solução para o problema de pesquisa, considerando uma estrutura de armazenamento das mensagens num banco de dados local, neste caso o mesmo banco de dados do sistema PDV, com o *status* de envio da mensagem, além de uma rotina de reprocessamento das mensagens não enviadas. Para explorar esta proposta, foi idealizado um protótipo da solução com as funções de enviar, armazenar e reprocessar mensagens.

Ao longo deste trabalho, exploramos o desenvolvimento do protótipo da proposta de solução, destacando os elementos fundamentais da arquitetura do software, ferramentas utilizadas e a descrição detalhada do simulador. Desde a escolha das linguagens e ferramentas até a modelagem dos dados e os cenários de teste, cada etapa foi crucial para garantir a eficácia e a robustez do protótipo, visto que ao longo do desenvolvimento de cada etapa, foram sendo aperfeiçoados os métodos e decições de projeto.

A implementação do simulador permitiu não apenas validar a proposta de solução, mas também identificar ajustes necessários para melhorar sua eficiência e usabilidade. Através dos testes realizados em diferentes cenários, foi possível avaliar a capacidade do sistema em lidar com situações adversas, como sobrecarga no *Web Service*, e verificar sua capacidade de recuperação diante destas falhas.

Os testes realizados com o protótipo da solução foram cruciais para comprovar que existe uma resposta de solução para o problema de pesquisa tratado neste trabalho e também mostraram a eficacia da solução proposta. Podemos também concluir que a proposta de solução abordada representa uma das possíveis propostas de solução, visto que a partir desta é possível realizar abordagens diferentes de solução e mitigação de problemas levando em consideração a arquitetura do sistema abordado.

REFERÊNCIAS

ALVES, W. P. **Java para Web: Desenvolvimento de Aplicações**. 1. ed. Sao Paulo: Editora Érica, 2015.

BIRRELL, A. D.; NELSON, B. J. **Implementing Remote Procedure Calls**. ACM Transactions on Computer Systems, v. 2, n. 1, p. 39–59, 1984. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/2080.357392>. Acesso em: 15 nov. 2023.

CIELO. **API Link de Pagamento Cielo**. Cielo Developers, 2023. Disponível em: <https://developercielo.github.io/>. Acesso em: 09 set. 2023.

COULOURIS, G. *et al.* **Sistemas Distribuídos: Conceitos e Projeto**. 5. ed. Porto Alegre: Bookman Editora, 2013.

ELMASRI R. A.; NAVATHE, S. B. **Sistemas de Banco de Dados**. 7. ed. São Paulo: Pearson Education do Brasil, 2018.

ERL, T. **SOA: Princípios de design de serviços**. São Paulo: Pearson Prentice Hall, 2009.

FERREIRA, A. G. **Interface de programação de aplicações (API) e web services**. 1. ed. São Paulo: Editora Saraiva, 2021.

FILHO, W. d. P. P. **Engenharia de Software - Projetos e Processos**. 4. ed. Rio de Janeiro: Grupo Gen-LTC, 2019. v. 2.

FOWLER, M. **Padrões de arquitetura de aplicações corporativas**. 1. ed. Porto Alegre: Bookman Editora, 2007.

IBM. **Reprocessamento de Mensagens**. Produtos da Documentação IBM, 2021. Disponível em: <https://www.ibm.com/docs/pt-br/maximo-manage/8.0.0?topic=management-message-reprocessing>. Acesso em: 07 set. 2023.

MAZIERO, C. **Sistemas Operacionais: Conceitos e Mecanismos**. Curitiba : DINF - UFPR: [s.n.], 2020. ISBN 978-85-7335-340-2. Disponível em: <https://www.researchgate.net/publication/343921399_Sistemas_Operacionais_Conceitos_e_Mecanismos>. Acesso em: 18 aug. 2023.

MONTEIRO, E. R. *et al.* **Sistemas Distribuídos**. Porto Alegre: SAGAH, 2020.

RODRIGUES, T. N. *et al.* **Integração de Aplicações**. Porto Alegre: SAGAH, 2020.

ROY, J.; RAMANUJAN, A. **Understanding web services. IEEE Internet Computing**. v. 3, n. 6, p. 69–73, 2001.

SAWAYA, M. R. **Dicionário de Informática e Internet**. São Paulo: Nobel, 1999.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Fundamentos de Sistemas Operacionais**. 9. ed. Rio de Janeiro: Grupo Gen-LTC, 2015.

STALLINGS, W. **Operating Systems: internals and design principles**. 7. ed. [S.l.]: Pearson Prentice Hall, 2012.

TAMAE, R. Y. **SISPRODIMEX - Sistema de Processamento Distribuído de Imagens**. Dissertação (Mestrado em Ciência da Computação) — Fundação de Ensino Eurípides Soares da Rocha, Marília, 2004. Disponível em: <https://livros01.livrosgratis.com.br/cp005630.pdf>. Acesso em: 01 nov. 2023.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas distribuídos: princípios e paradigmos**. 2. ed. São Paulo: Pearson Prentice Hall, 2007.

TIHOMIROVS, J.; GRABIS, J. **Comparison of SOAP and REST Based Web Services Using Software Evaluation Matrics**. Information Technology and Management Science, Riga, v. 19, n. 1, p. 92–97, 2016. Disponível em: <https://itms-journals.rtu.lv/article/view/itms-2016-0017/0>. Acesso em: 23 aug. 2023.