

UNIVERSIDADE FEDERAL FLUMINENSE

LUCAS SEVERIANO DOS SANTOS

**Uma Arquitetura Modular de Middleware para  
Comunicação Eficiente em Sistemas de Gêmeos  
Digitais**

NITERÓI

2026

LUCAS SEVERIANO DOS SANTOS

# **Uma Arquitetura Modular de Middleware para Comunicação Eficiente em Sistemas de Gêmeos Digitais**

Tese de doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Ciência da Computação.

Orientador:

Raphael Carlos Santos Machado

Coorientador:

Luiz Gustavo Cardoso Maria

NITERÓI

2026

Ficha catalográfica automática - SDC/BEE  
Gerada com informações fornecidas pelo autor

S237a Santos, Lucas Severiano dos  
Uma Arquitetura Modular de Middleware para Comunicação  
Eficiente em Sistemas de Gêmeos Digitais / Lucas Severiano  
dos Santos. - 2026.  
168 f.

Orientador: Raphael Carlos Santos Machado.  
Coorientador: Luiz Gustavo Cardoso Maria.  
Tese (doutorado)-Universidade Federal Fluminense, Instituto  
de Computação, Niterói, 2026.

1. Gêmeo Digital. 2. Middleware. 3. Arquitetura de  
Software. 4. Sistemas Ciber-Físicos. 5. Produção  
intelectual. I. Machado, Raphael Carlos Santos, orientador.  
II. Maria, Luiz Gustavo Cardoso, coorientador. III.  
Universidade Federal Fluminense. Instituto de Computação.  
IV. Título.

CDD - XXX

# LUCAS SEVERIANO DOS SANTOS

Uma Arquitetura Modular de Middleware para Comunicação Eficiente em Sistemas de  
Gêmeos Digitais

Tese de doutorado apresentada ao Programa  
de Pós-Graduação em Computação da Uni-  
versidade Federal Fluminense como requisito  
parcial para a obtenção do Grau de Dou-  
tor em Computação. Área de concentração:  
Ciência da Computação.

## BANCA EXAMINADORA

---

Prof. D.Sc. Raphael Carlos Santos Machado - Orientador, UFF

---

Prof. D.Sc. Luiz Gustavo Cardoso Maria - Coorientador, Firjan SENAI

---

Prof. D.Sc. José Viterbo Filho, UFF

---

Prof<sup>a</sup>. D.Sc. Flávia Coimbra Delicato, UFF

---

Prof. D.Sc. Felipe Rebelo Lopes, Firjan SENAI

---

Prof. D.Sc. Daniel Sadoc Menasche, UFRJ

Niterói, 2026

*À minha filha, Laura.*

# Agradecimentos

A Deus, por me sustentar ao longo desta caminhada, concedendo força, sabedoria e perseverança para enfrentar os desafios e seguir em frente até a conclusão desta etapa.

Aos meus orientadores, pelo apoio, pela confiança e pelas contribuições durante o desenvolvimento deste trabalho.

À minha esposa, pelo companheirismo, compreensão, apoio e incentivo em todos os momentos.

Aos meus pais, pelo amor, apoio e incentivo ao longo de toda a minha formação.

Ao Rafael Carvalho, pelo apoio nas solicitações de recursos financeiros para participação em eventos acadêmicos.

À Firjan, pelo apoio financeiro às viagens e à participação em eventos relacionados a esta pesquisa.

Aos meus colegas da Firjan, pelo apoio, pelas trocas de conhecimento e pelo convívio ao longo desta trajetória.

A todos que, de alguma forma, contribuíram para a realização deste trabalho.

# Resumo

Os Gêmeos Digitais [Digital Twins \(DTs\)](#) têm se destacado como um conceito revolucionário para apoiar a Indústria 4.0, ao promover a integração entre sistemas físicos e virtuais. As aplicações dos [DTs](#) podem ser encontrados em diversos como saúde e cidades inteligentes. Contudo, apesar do crescimento de sua adoção em aplicações industriais, sua implementação ainda apresenta desafios relacionados à interoperabilidade, ao acoplamento entre componentes e à dependência de protocolos específicos de comunicação. Esta tese apresenta a proposta, implementação e avaliação experimental de uma arquitetura de middleware modular e agnóstica a protocolos para sistemas de Gêmeos Digitais. A arquitetura foi concebida para desacoplar os mecanismos de comunicação da lógica de negócio dos [DTs](#). A arquitetura proposta foi implementada utilizando linguagem de alto nível (Python) e avaliada por meio de um estudo de caso baseado no Simulador de Treinamento de Imersão dos submarinos da Classe Tupi. Os resultados obtidos demonstraram que a proposta do middleware, baseado na camada de transporte, apresentaram menores latências médias quando comparadas às demais abordagens disponíveis na literatura, em todos os cenários avaliados. Adicionalmente, a arquitetura demonstrou flexibilidade ao permitir a separação entre lógica de negócio, mecanismos de transporte e protocolos utilizados na comunicação com o sistema físico, o que favoreceu a integração e evolução de novos protocolos de comunicação sem a necessidade de modificações significativas na estrutura arquitetural.

**Palavras-chave:** Gêmeo Digital, Middleware, Arquitetura de Software, Sistemas Ciber-Físicos.

# Abstract

Digital Twins (DTs) have emerged as a revolutionary concept to support Industry 4.0 by promoting the integration between physical and virtual systems. Applications of DTs can be found in several domains, such as healthcare and smart cities. However, despite their growing adoption in industrial environments, their implementation still faces challenges related to interoperability, component coupling, and dependency on specific communication protocols. This thesis presents the proposal, implementation, and experimental evaluation of a modular and protocol-agnostic middleware architecture for Digital Twin systems. The architecture was designed to decouple communication mechanisms from the business logic of DTs. The proposed architecture was implemented using a high-level programming language (Python) and evaluated through a case study based on the Immersion Training Simulator of the Tupi-class submarines. The obtained results demonstrated that the proposed transport-layer-based middleware achieved lower average latency when compared to other approaches available in the literature across all evaluated scenarios. Additionally, the architecture demonstrated flexibility by enabling the separation between business logic, transport mechanisms, and protocols used for communication with the physical system, which facilitated the integration and evolution of new communication protocols without requiring significant modifications to the proposed architectural structure.

**Keywords:** Digital Twin, Middleware, Software Architecture, Cyber-Physical Systems.

# Lista de Figuras

|    |                                                                                                                                |    |
|----|--------------------------------------------------------------------------------------------------------------------------------|----|
| 1  | DT de uma Ponte Rolante. Fonte: (Tu <i>et al.</i> , 2021; Santos; Júnior, 2024).                                               | 13 |
| 2  | Ilustração da metodologia para o desenvolvimento deste trabalho.                                                               | 16 |
| 3  | Holonic Manufacturing Systems – HMS. Adaptado de (Van Brussel <i>et al.</i> , 1998).                                           | 21 |
| 4  | Frasco inteligente de molho de espaguete. Adaptado de (Wong <i>et al.</i> , 2002).                                             | 22 |
| 5  | Ilustração de acompanhamento de Produto. Extraído de (Främling <i>et al.</i> , 2003).                                          | 23 |
| 6  | Níveis de integração entre modelo digital, sombra digital e gêmeo digital. Adaptado de (Fuller <i>et al.</i> , 2020).          | 26 |
| 7  | Representação em alto nível do Digital Twin. Extraído de (Prevu3D, 2025).                                                      | 26 |
| 8  | Categorias de comunicação em cinco categorias. Adaptado de (Ala-Laurinaho, 2019).                                              | 28 |
| 9  | Modelo TCP/IP vs OSI.                                                                                                          | 30 |
| 10 | Arquitetura Cliente-Servidor do HTTP. Extraído de (Ross; Kurose, 2012).                                                        | 31 |
| 11 | Estrutura de mensagem de requisição HTTP. Extraído de (Ross; Kurose, 2012).                                                    | 32 |
| 12 | Estrutura de mensagem de requisição HTTP. Extraído de (Ross; Kurose, 2012).                                                    | 33 |
| 13 | Modelo de comunicação <i>publish–subscribe</i> do protocolo MQTT. Extraído de (Durkop; Czybik; Jasperneite, 2015; Naik, 2017). | 34 |
| 14 | Estrutura hierárquica de tópicos no MQTT.                                                                                      | 35 |
| 15 | Estrutura de mensagem do MQTT.                                                                                                 | 35 |
| 16 | Arquitetura do servidor OPC UA(OPC Foundation, 2017b).                                                                         | 38 |

|    |                                                                                                                                            |     |
|----|--------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 17 | <a href="#">DT</a> aplicado à simulação de cirurgia remota com braço robótico.( <a href="#">Laaki; Miche; Tammi, 2019</a> ) . . . . .      | 54  |
| 18 | Arquitetura agnóstica a protocolo. . . . .                                                                                                 | 57  |
| 19 | Diagrama de casos de uso das funcionalidades do Middleware. . . . .                                                                        | 58  |
| 20 | Visão geral da arquitetura proposta do middleware. . . . .                                                                                 | 61  |
| 21 | Escala de níveis de maturidade tecnológica (TRL 1–9). Fonte: ( <a href="#">Abaco Systems, 2025</a> ). . . . .                              | 64  |
| 22 | Diagrama de classes da Implementação do Middleware. . . . .                                                                                | 65  |
| 23 | Middleware em execução como serviço. . . . .                                                                                               | 68  |
| 24 | Estrutura de mensagem. . . . .                                                                                                             | 71  |
| 25 | Estrutura do cache de inicialização. . . . .                                                                                               | 73  |
| 26 | Estrutura de <i>publish/subscribe</i> do Middleware. . . . .                                                                               | 76  |
| 27 | Imagens do <a href="#">TI</a> . . . . .                                                                                                    | 79  |
| 28 | Framework TupiSIM implementado no <a href="#">TI</a> . . . . .                                                                             | 82  |
| 29 | Equipamentos utilizados na bancada de testes. . . . .                                                                                      | 84  |
| 30 | <a href="#">Physical Twin (PT)</a> emulado para no <a href="#">Simulador de Treinamento de Imer-</a><br><a href="#">são (TI)</a> . . . . . | 86  |
| 31 | Lista de tags no software UaExpert. . . . .                                                                                                | 87  |
| 32 | Topologia local para testes. . . . .                                                                                                       | 89  |
| 33 | Topologia distribuída para testes. . . . .                                                                                                 | 90  |
| 34 | Organização dos cenários de testes. . . . .                                                                                                | 92  |
| 35 | Operação de Leitura com Dados Analógicos na Topologia Local. . . . .                                                                       | 93  |
| 36 | Operação de Escrita com Dados Analógicos na Topologia Local. . . . .                                                                       | 94  |
| 37 | Operação de Leitura com Dados Digitais na Topologia Local. . . . .                                                                         | 99  |
| 38 | Operação de Escrita com Dados Digitais na Topologia Local. . . . .                                                                         | 100 |
| 39 | Operação de Leitura com Dados Analógicos na Topologia Distribuída. . . .                                                                   | 104 |
| 40 | Operação de Escrita com Dados Analógicos na Topologia Distribuída. . . .                                                                   | 105 |

|    |                                                                                                   |     |
|----|---------------------------------------------------------------------------------------------------|-----|
| 41 | Operação de Leitura com Dados Digitais na Topologia Distribuída. . . . .                          | 109 |
| 42 | Operação de Escrita com Dados Digitais na Topologia Distribuída. . . . .                          | 110 |
| 43 | Exemplo de captura de pacote UDP utilizando o Wireshark. . . . .                                  | 115 |
| 44 | Operação de leitura variando o tamanho da mensagem requisitadas (local vs <i>setup</i> ). . . . . | 116 |
| 45 | Operação de escrita variando o tamanho da mensagem requisitadas (local vs <i>setup</i> ). . . . . | 118 |
| 46 | Topologia distribuída para testes com gerador de eventos. . . . .                                 | 121 |
| 47 | Execução do cenário experimental no cenário de comunicação assíncrona. .                          | 122 |
| 48 | Throughput médio em função do número de tópicos para TCP e UDP. . .                               | 123 |
| 49 | Cenário experimental com Message Queuing Telemetry Transport (MQTT).                              | 126 |
| 50 | Implementação MQTT. . . . .                                                                       | 127 |
| 51 | Cenário experimental com MQTT. . . . .                                                            | 127 |
| 52 | Evidências de alteração do código. . . . .                                                        | 128 |
| 53 | Diagrama de Classes com os métodos. . . . .                                                       | 144 |
| 54 | Download do <i>Raspberry Pi Imager</i> . . . . .                                                  | 152 |
| 55 | Raspberry Pi Imager. . . . .                                                                      | 153 |
| 56 | Opção de configurações avançadas. . . . .                                                         | 154 |
| 57 | Configurações avançadas. . . . .                                                                  | 155 |

# Lista de Tabelas

|    |                                                                                                                         |     |
|----|-------------------------------------------------------------------------------------------------------------------------|-----|
| 1  | Trabalhos relacionados que aplicam Arquitetura Orientada a Serviço. . . .                                               | 44  |
| 2  | Comparação entre trabalhos relacionados quanto aos mecanismos e estilos de comunicação em middleware para IoT . . . . . | 48  |
| 3  | Comparação dos trabalhos relacionados quanto à modularidade arquitetural e desacoplamento da comunicação. . . . .       | 50  |
| 4  | Rastreabilidade entre requisitos arquiteturais e componentes. . . . .                                                   | 67  |
| 5  | Parâmetros do arquivo de configuração do Middleware. . . . .                                                            | 70  |
| 6  | Distribuição das tags implementadas no <a href="#">TI</a> . . . . .                                                     | 83  |
| 7  | Materiais e equipamentos utilizados na configuração experimental. . . . .                                               | 84  |
| 8  | Variáveis controladas nos experimentos. . . . .                                                                         | 92  |
| 9  | Latência média e desvio padrão para leitura de dados analógicos em topologia local. . . . .                             | 95  |
| 10 | Latência média e desvio padrão para escrita de dados analógicos em topologia local. . . . .                             | 96  |
| 11 | Latência média e desvio padrão para leitura de dados digitais em topologia local. . . . .                               | 101 |
| 12 | Latência média e desvio padrão para escrita de dados digitais em topologia local. . . . .                               | 102 |
| 13 | Latência média e desvio padrão para leitura de dados analógicos em topologia distribuída. . . . .                       | 106 |
| 14 | Latência média e desvio padrão para escrita de dados analógicos em topologia distribuída. . . . .                       | 107 |
| 15 | Latência média e desvio padrão para leitura de dados digitais em topologia distribuída. . . . .                         | 111 |

---

|    |                                                                                                                                                              |     |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 16 | Latência média e desvio padrão para escrita de dados digitais em topologia distribuída. . . . .                                                              | 112 |
| 17 | Estatísticas de latência do middleware para operações de leitura, considerando a variação do tamanho do <i>payload</i> , diferentes protocolos e topologias. | 117 |
| 18 | Estatísticas de latência do middleware para operações de escrita, considerando a variação do tamanho do <i>payload</i> , diferentes protocolos e topologias. | 119 |
| 19 | Variáveis controladas nos experimentos de comunicação assíncrona. . . . .                                                                                    | 123 |
| 20 | Resultados médios de throughput e integridade das mensagens para comunicação <i>publish/subscribe</i> sobre TCP e UDP. . . . .                               | 124 |
| 21 | Evidências do baixo custo arquitetural da extensão do protocolo MQTT. .                                                                                      | 128 |

# Lista de Abreviaturas e Siglas

**ACC** Adaptive Cruise Control

**ADAS** Advanced Driver-Assistance System

**AEB** Autonomous Emergency Braking

**AI** Artificial Intelligence

**API** Application Programming Interface

**BIM** Building Information Modeling

**BLE** Bluetooth Low Energy

**CAD** Computer-Aided Design

**CAM** Computer-Aided Manufacturing

**CIAMA** Centro de Instrução e Adestramento Almirante Átila Monteiro Aché

**CLP** Controlador Lógico Programável

**CMMS** Computerized Maintenance Management System

**CoAP** Constrained Application Protocol

**CPS** Cyber-Physical System

**CPT** Digital Twin Capabilities Periodic Table

**CR** Carriage Return

**CRM** Customer Relationship Management

**DDS** Data Distribution Service

**DT** Digital Twin

**DTC** Digital Twin Consortium

**EAM** Enterprise Asset Management

**EOI** Estação de Operação do Instrutor

**ERP** Enterprise Resource Planning

**GraphQL** Graph Query Language

**GSN** Global Sensor Network

**HATEOAS** Hypermedia as the Engine of Application State

**HMD** Head-Mounted Display

**HMS** Holonic Manufacturing Systems

**HTTP** Hypertext Transfer Protocol

**IBM** International Business Machines

**IIoT** Industrial Internet of Things

**IoT** Internet of Things

**IP** Internet Protocol

**ISI-SVP** Instituto SENAI de Inovação em Sistemas Virtuais de Produção

**IST** Instituto SENAI de Serviço Tecnológico em Automação Industrial

**JSON** JavaScript Object Notation

**M2M** Machine-to-Machine

**MB** Marinha do Brasil

**MCI** MQTT–CoAP Interconnector

**MES** Manufacturing Execution Systems

**MOM** Message Oriented Middleware

**MQTT** Message Queuing Telemetry Transport

**MQTT-SN** MQTT for Sensor Networks

**MR-UDP** Mobile Reliable UDP

**MS** Módulo de Simulação

**MSM** Mirrored Spaces Model

**MTU** Maximum Transmission Unit

**NASA** National Aeronautics and Space Administration

**OASIS** Organization for the Advancement of Structured Information Standards

**OMG** Object Management Group

**OPC** Open Platform Communications

**OPC UA** Open Platform Communications Unified Architecture

**OSI** Open Systems Interconnection

**OT** Operational Technology

**PID** Process Identifier

**PoC** Proof of Concept

**PT** Physical Twin

**Pub/Sub** Publish–Subscribe

**REST** Representational State Transfer

**RFID** Radio Frequency Identification

**ROA** Resource-Oriented Architecture

**RPC** Remote Procedure Calls

**SCADA** Supervisory Control and Data Acquisition

**SDDL** Scalable Data Distribution Layer

**Senaas** Sensor as a Service

**SOA** Service-Oriented Architecture

**SQL** Structured Query Language

**SSL** Secure Sockets Layer

**TCP** Transmission Control Protocol

**TI** Simulador de Treinamento de Imersão

**TIC** Tecnologia da Informação e Comunicações

**TLS** Transport Layer Security

**TRL** Technology Readiness Level

**UDP** User Datagram Protocol

**UML** Unified Modeling Language

**URL** Uniform Resource Locator

**WPF** Windows Presentation Foundation

**XML** Extensible Markup Language

# Sumário

|          |                                                             |           |
|----------|-------------------------------------------------------------|-----------|
| <b>1</b> | <b>Introdução</b>                                           | <b>12</b> |
| 1.1      | Motivação . . . . .                                         | 14        |
| 1.2      | Objetivos . . . . .                                         | 15        |
| 1.3      | Metodologia para o Desenvolvimento deste Trabalho . . . . . | 16        |
| 1.4      | Artigos Associados à Tese . . . . .                         | 18        |
| 1.5      | Organização da Tese . . . . .                               | 18        |
| <b>2</b> | <b>Fundamentação Teórica</b>                                | <b>20</b> |
| 2.1      | Gêmeo Digital . . . . .                                     | 20        |
| 2.1.1    | Contexto histórico . . . . .                                | 20        |
| 2.1.2    | Conceito e Definição . . . . .                              | 24        |
| 2.1.3    | Categorias de comunicação nos Gêmeos Digitais . . . . .     | 27        |
| 2.2      | Arquitetura em Camadas em Redes de Computadores . . . . .   | 29        |
| 2.3      | Protocolos da Camada de Aplicação . . . . .                 | 30        |
| 2.3.1    | HTTP e REST . . . . .                                       | 30        |
| 2.3.2    | MQTT . . . . .                                              | 33        |
| 2.3.3    | GraphQL . . . . .                                           | 36        |
| 2.3.4    | OPC UA . . . . .                                            | 36        |
| <b>3</b> | <b>Trabalhos Relacionados</b>                               | <b>39</b> |
| 3.1      | Arquiteturas Orientadas a Serviço . . . . .                 | 39        |
| 3.2      | Middleware modular para comunicação em IoT . . . . .        | 44        |

|          |                                                              |           |
|----------|--------------------------------------------------------------|-----------|
| 3.3      | Framework para desenvolvimento de simuladores . . . . .      | 48        |
| 3.4      | Considerações Finais . . . . .                               | 50        |
| <b>4</b> | <b>Arquitetura Proposta do Middleware</b>                    | <b>52</b> |
| 4.1      | Requisitos para Sistemas de Gêmeos Digitais . . . . .        | 52        |
| 4.2      | Requisitos Arquiteturais do Middleware . . . . .             | 55        |
| 4.3      | Requisitos Funcionais do Middleware . . . . .                | 57        |
| 4.4      | Visão Geral da Arquitetura . . . . .                         | 59        |
| 4.5      | Prova de Conceito . . . . .                                  | 63        |
| 4.5.1    | Diagrama de classes e Modularidade . . . . .                 | 65        |
| 4.5.2    | Ambiente de Implementação . . . . .                          | 67        |
| 4.5.3    | Arquivo de Configuração . . . . .                            | 68        |
| 4.5.4    | Estrutura de Mensagens e Protocolo de Comunicação . . . . .  | 69        |
| 4.5.5    | Considerações sobre a Implementação . . . . .                | 72        |
| 4.6      | Considerações Finais . . . . .                               | 77        |
| <b>5</b> | <b>Materiais e Métodos</b>                                   | <b>78</b> |
| 5.1      | Descrição do Estudo de Caso . . . . .                        | 78        |
| 5.1.1    | Breve Histórico do Projeto . . . . .                         | 78        |
| 5.1.2    | Descrição Técnica . . . . .                                  | 80        |
| 5.1.3    | Estrutura Física do Simulador . . . . .                      | 82        |
| 5.2      | Reprodução do Cenário Experimental . . . . .                 | 83        |
| 5.2.1    | Materiais e Equipamentos Utilizados . . . . .                | 84        |
| <b>6</b> | <b>Avaliação Experimental</b>                                | <b>88</b> |
| 6.1      | Ambiente Experimental para Avaliação de Desempenho . . . . . | 88        |
| 6.1.1    | Topologia Local . . . . .                                    | 89        |
| 6.1.2    | Topologia Distribuída . . . . .                              | 90        |

|          |                                                                                                   |            |
|----------|---------------------------------------------------------------------------------------------------|------------|
| 6.1.3    | Variáveis Controladas . . . . .                                                                   | 91         |
| 6.1.4    | Avaliação de Desempenho em Topologia Local . . . . .                                              | 92         |
| 6.1.4.1  | Variando a quantidade de mensagens enviadas - leitura e<br>escrita com dados analógicos . . . . . | 92         |
|          | Análise dos Resultados . . . . .                                                                  | 97         |
| 6.1.4.2  | Variando a quantidade de mensagens enviadas – leitura e<br>escrita com dados digitais . . . . .   | 98         |
|          | Análise dos Resultados . . . . .                                                                  | 103        |
| 6.1.5    | Avaliação de Desempenho em Topologia Distribuída . . . . .                                        | 104        |
| 6.1.5.1  | Variando a quantidade de mensagens enviadas - leitura e<br>escrita com dados analógicos . . . . . | 104        |
|          | Análise dos Resultados . . . . .                                                                  | 108        |
| 6.1.5.2  | Variando a quantidade de mensagens enviadas - leitura e<br>escrita com dados digitais . . . . .   | 108        |
|          | Análise dos Resultados . . . . .                                                                  | 113        |
| 6.1.6    | Variando o tamanho da mensagem - Local vs Distribuído . . . . .                                   | 113        |
|          | Análise dos resultados: . . . . .                                                                 | 119        |
| 6.1.7    | Avaliação de Desempenho da Comunicação Assíncrona . . . . .                                       | 120        |
|          | Análise dos Resultados: . . . . .                                                                 | 123        |
| 6.2      | Validação da Arquitetura . . . . .                                                                | 124        |
| 6.2.1    | Cenário Experimental com MQTT . . . . .                                                           | 125        |
| 6.2.2    | Implementação do Cliente MQTT no Middleware . . . . .                                             | 126        |
| <b>7</b> | <b>Conclusão</b>                                                                                  | <b>129</b> |
| 7.1      | Considerações Finais . . . . .                                                                    | 129        |
| 7.2      | Trabalhos Futuros . . . . .                                                                       | 130        |
|          | <b>REFERENCIAS</b>                                                                                | <b>132</b> |

|                                                                                       |            |
|---------------------------------------------------------------------------------------|------------|
| <b>Apêndice A - Diagrama de Classes</b>                                               | <b>143</b> |
| <b>Apêndice B - Especificação das Mensagens da API</b>                                | <b>145</b> |
| B.1 Inicializar comunicação ( <code>init</code> ) . . . . .                           | 145        |
| B.2 Ler valor ( <code>read</code> ) . . . . .                                         | 146        |
| B.3 Escrever valor ( <code>write</code> ) . . . . .                                   | 147        |
| B.4 Assinar tópico ( <code>subscribe</code> ) . . . . .                               | 148        |
| B.5 Cancelar assinatura ( <code>unsubscribe</code> ) . . . . .                        | 149        |
| <b>Apêndice C - Configuração do Physical Twin para o Ambiente Experimental</b>        | <b>151</b> |
| C.1 Procedimento de Instalação do Sistema Operacional no Cartão SD . . . . .          | 151        |
| Passo 1 — Obtenção da Ferramenta de Instalação . . . . .                              | 152        |
| Passo 2 — Preparação do Cartão SD . . . . .                                           | 152        |
| Passo 3 — Seleção do Sistema Operacional . . . . .                                    | 152        |
| Passo 4 — Configurações Avançadas de Inicialização . . . . .                          | 153        |
| Passo 5 — Gravação da imagem no cartão SD . . . . .                                   | 154        |
| Passo 6 — Inicialização do Raspberry Pi . . . . .                                     | 154        |
| <b>Apêndice D - Procedimentos de Instalação e Execução do Middleware Digital Twin</b> | <b>156</b> |
| D.1 Obtenção do Código-Fonte . . . . .                                                | 156        |
| D.2 Estrutura do Projeto . . . . .                                                    | 157        |
| D.3 Requisitos de Software . . . . .                                                  | 157        |
| D.4 Instalação das Dependências . . . . .                                             | 157        |
| D.5 Configuração do Middleware . . . . .                                              | 158        |
| D.6 Execução do Middleware . . . . .                                                  | 158        |

# 1 Introdução

A transformação digital associada à Indústria 4.0 tem fomentado a integração entre sistemas físicos e digitais e promovido novas formas de monitoramento, análise e controle de processos industriais complexos (Lu, 2017). Esse avanço é sustentado pela convergência de tecnologias habilitadoras, como Sistemas Ciberfísicos (Cyber-Physical System (CPS)), Internet das Coisas (Internet of Things (IoT)), computação distribuída e Inteligência Artificial (Artificial Intelligence (AI)) (Mihai *et al.*, 2022). Essa transformação torna os sistemas mais flexíveis, adaptáveis e capazes de operar de forma autônoma e orientada a dados. Adicionalmente, avanços nas Tecnologias da Informação e Comunicações (TICs) têm possibilitado o desenvolvimento de ferramentas computacionais mais complexas e robustas, voltadas à simulação, emulação e ao processamento de dados. De forma geral, busca-se integrar os sistemas físicos industriais a soluções de software que dão suporte à tomada de decisão, à avaliação de desempenho e à implementação de estratégias de otimização. Nesse contexto, os Gêmeos Digitais (Digital Twin (DT)) ocupam uma posição central na Indústria 4.0, ao estabelecerem uma conexão contínua entre um ativo físico, denominado Gêmeo Físico (Physical Twin (PT)), e sua correspondente representação digital.

Um DT é capaz de refletir dinamicamente o estado e o comportamento do sistema por meio da ingestão de dados funcionais e operacionais (Tao *et al.*, 2023). Essa associação, viabiliza o uso do modelo digital não apenas para observação, mas também para análise, simulação e avaliação de cenários futuros e hipotéticos (Rodríguez-Alonso; Pena-Regueiro; García, 2024). Do ponto de vista conceitual e arquitetural, essa relação é formalizada no modelo proposto por Michael Grieves, que define um DT a partir de três elementos fundamentais: os ativos físicos, os ativos virtuais e as conexões de dados e informações responsáveis por integrar esses dois domínios (Liau; Lee; Ryu, 2018). A Figura 1 apresenta uma ilustração de uma aplicação de DT para Ponte Rolante.



Figura 1: DT de uma Ponte Rolante. Fonte: (Tu *et al.*, 2021; Santos; Júnior, 2024).

A importância estratégica dos DTs é evidenciada pelo crescimento expressivo dos investimentos públicos e privados. De acordo com relatório da MarketsandMarkets, o mercado global de Digital Twins foi estimado em aproximadamente 10,1 bilhões de dólares em 2023, com previsão de alcançar 73,5 bilhões até 2028, impulsionado pela digitalização de segmentos como manufatura, energia, saúde e cidades inteligentes (MarketsandMarkets, 2023). Empresas como Siemens, General Electric e IBM direcionam recursos para o desenvolvimento de plataformas baseadas em DT, reconhecendo seu papel fundamental na inovação tecnológica, no aumento da eficiência e na sustentação de processos decisórios orientados a dados (Council, 2024). Além disso, governos de países como Alemanha, China e Estados Unidos passaram a integrar os DTs em suas estratégias nacionais de transformação digital, com destaque para programas voltados à manufatura avançada, automação industrial e infraestrutura urbana inteligente (Ministry of Industry and Information Technology of China, 2023; Gartner, Inc., 2023). No Brasil, empresas como WEG, Tupy e Natura vêm direcionando investimentos à transformação digital por meio de tecnologias como os Gêmeos Digitais, com foco no aumento da produtividade e na modernização industrial (Indústria, 2023).

Apesar do crescimento expressivo de investimentos e do reconhecimento do potencial dos Gêmeos Digitais, sua adoção ainda enfrenta limitações importantes. Estudos apontam a necessidade de investigações sistemáticas sobre aspectos arquiteturais, interoperabilidade, comunicação e desenvolvimento de aplicações de DT (Zheng; Yang; Cheng, 2019; Almeida *et al.*, 2023). No domínio físico, ainda há uma predominância de dispositivos tradicionais (sem capacidades de comunicação e computação) que demandam métodos de coleta de dados, a fim de evitar a substituição integral dos equipamentos existentes e os elevados custos (Pessoa *et al.*, 2018). No contexto virtual, o desenvolvimento de sistemas e a modelagem de máquinas complexas não são tarefas triviais e demandando conhecimen-

tos especializados e a atuação de profissionais qualificados (Zheng; Yang; Cheng, 2019). Ademais, no tocante aos mecanismos de transmissão de dados, manter o sincronismo entre os ativos físico e sua contraparte virtual é uma tarefa crítica, que demanda de métodos e interfaces e protocolos apropriados. Ambientes industriais frequentemente utilizam protocolos especializados, como o [Open Platform Communications Unified Architecture \(OPC UA\)](#) (Mahnke; Leitner; Damm, 2009), o que introduz complexidades relacionadas à modelagem de dados, configuração e interoperabilidade. Ademais, as diferenças entre especificações de protocolos, as restrições de rede e a necessidade de interações com baixa latência adicionam camadas de complexidade ao desenvolvimento de aplicações baseadas em DT (Zhang; Müller; Kim, 2024; Wang; Zhang; Xu, 2024). Diante desse cenário, plataformas de middleware podem atuar como uma camada de abstração intermediária, fornecendo interfaces padronizadas e serviços capazes de reduzir a complexidade de integração e simplificar o processo de desenvolvimento de sistemas de DT (Pereira *et al.*, 2026). Através da abstração dos mecanismos de comunicação e consistência de dados, o middleware possibilita que os desenvolvedores foquem os esforços no desenvolvimento nas implementações da lógicas da aplicação do DT.

## 1.1 Motivação

Diversos estudos na literatura apresentam propostas de arquiteturas e *frameworks* conceituais para a aplicação de Gêmeos Digitais. No entanto, muitas dessas propostas carecem de diretrizes claras para o desenvolvimento e a implementação prática dos sistemas, o que limita sua aplicabilidade em contextos reais. Além disso, a maioria das ferramentas são comerciais, isto é, são soluções fechadas e proprietárias (Rolle; Martucci; Godoy, 2020). Em contrapartida, trabalhos mais recentes têm avançado na construção de arcabouços abertos e metodologias voltadas à implementação prática (Santos; Maria *et al.*, 2024; Alvarenga *et al.*, 2024; Rolle; Martucci; Godoy, 2020). Embora representem avanços importantes na definição de arquiteturas para DTs, esses trabalhos não abordam a modularidade e o desacoplamento entre os mecanismos de comunicação e as regras de negócio<sup>1</sup>. A falta dessa separação compromete a modularidade, aumenta a complexidade de desenvolvimento e manutenção do sistema e sobrecarrega o desenvolvedor com detalhes de rede DT (André; Azzi; Cardin, 2020; Bass; Clements; Kazman, 2021).

Propostas como as de (Schiekofer; Scholz; Weyrich, 2018; Hietala *et al.*, 2020b; Ab-

---

<sup>1</sup>Em engenharia de software, regra de negócio refere-se à lógica que representa os processos, políticas e restrições específicas de um domínio de aplicação. Ela define como os dados devem ser processados e quais decisões a aplicação deve tomar em diferentes cenários.

delgawad; Gausemeier; Trächtler *et al.*, 2017; Pereira *et al.*, 2026) sugerem o uso de arquiteturas de *middleware*, ou *Application Programming Interface (API)*, baseadas em interfaces padronizadas, como *Representational State Transfer (REST)* e *Graph Query Language (GraphQL)*, com o objetivo de reduzir a dependência de protocolos industriais específicos, como o *OPC UA*, e aumentar a flexibilidade na integração entre sistemas heterogêneos. Embora essas abordagens tragam avanços em termos de desacoplamento, interoperabilidade e facilitem o desenvolvimento por parte dos engenheiros e desenvolvedores de *software*, elas geralmente não foram concebidas para atender aos requisitos de comunicação em tempo real. Além disso, esses trabalhos não avançam em propostas de soluções que priorizem o agnosticismo a protocolos, pois mantêm o foco na implementação de padrões industriais específicos, como *OPC UA* e Modbus, o que limita a portabilidade e a adaptabilidade em diferentes contextos de aplicação. Complementarmente, o mapeamento sistemático apresentado em (Almeida *et al.*, 2023) indica que, embora diversos trabalhos abordem a comunicação em tempo real em sistemas de *DT*, ainda há uma considerável carência no detalhamento das implementações para atingir o baixa latência. Além disso, o estudo salienta que a maioria dos estudos mapeados não realizou validações de suas propostas e, em alguns casos, a validação foi conduzida em ambientes simulado.

Requisitos associados aos mecanismos eficientes de comunicação tornam-se ainda mais relevantes em sistemas de *DTs* de alta fidelidade. Esses sistemas utilizam modelos matemáticos (módulos de simulação) para reproduzir as características físicas, operacionais e comportamentais do sistema real (Itani; Sharafeddine, 2024). Nesse contexto, atrasos na propagação (latência) de estados e comandos ampliam o período de desalinhamento entre os componentes físicos e virtuais, podendo comprometer a sincronização e a fidelidade operacional do *DT* (Bellavista *et al.*, 2024; Munoz, 2022). Portanto, torna-se necessário investigar arquiteturas de *middleware* capazes de abstrair os mecanismos de comunicação e reduzir o acoplamento tecnológico sem introduzir *overhead* significativo.

## 1.2 Objetivos

Este trabalho tem por objetivo propor e projetar uma arquitetura modular de *middleware* para viabilizar uma comunicação eficiente, flexível e com baixo *overhead* entre sistemas físicos e Gêmeos Digitais. Além disso, o trabalho propõe uma implementação prática do *middleware* para validar sua aplicabilidade e viabilidade, considerando um estudo de caso real e de relevância. Como objetivos específicos, pretende-se:

- Identificar e analisar os requisitos funcionais e não funcionais, bem como os desafios e limitações relacionados à comunicação entre sistemas físicos e Gêmeos Digitais.
- Realizar o levantamento da literatura sobre protocolos e mecanismos de comunicação utilizados na integração entre sistemas físicos e Gêmeos Digitais.
- Projetar uma arquitetura de middleware modular, flexível e agnóstica a protocolos de comunicação, com suporte a mecanismos síncronos e assíncronos de leitura e escrita.
- Implementar uma **Proof of Concept (PoC)** do middleware com base na arquitetura proposta, utilizando linguagem de alto nível e comunicação na camada de transporte, com a finalidade de minimizar o *overhead*.
- Avaliar experimentalmente o middleware proposto com base em um estudo de caso realista de um **DT** de alta fidelidade.

## 1.3 Metodologia para o Desenvolvimento deste Trabalho

Para alcançar os objetivos propostos neste trabalho, foram definidas atividades organizadas em etapas complementares e sequenciais, conforme ilustrado na Figura 2. Cada retângulo representa uma atividade executada, enquanto que, os círculos preenchidos representam o início e fim das atividades. A seguir cada atividade é descrita:

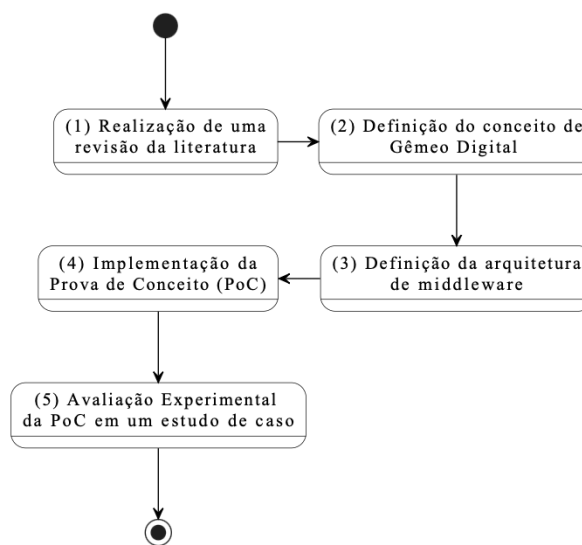


Figura 2: Ilustração da metodologia para o desenvolvimento deste trabalho.

1. **Realização de uma revisão da literatura:** Nesta atividade, foram identificados os fundamentos teóricos e o estado da arte relacionados aos sistemas de DT, com ênfase nos mecanismos de integração entre os componentes físicos e digitais. A revisão incluiu conceitos de Digital Twin, mecanismos síncronos e assíncronos de comunicação, protocolos industriais, arquiteturas de middleware e frameworks para simuladores de alta fidelidade. Esse levantamento permitiu reconhecer limitações relacionadas ao acoplamento, à dependência de protocolos específicos, à extensibilidade e ao *overhead* de comunicação, que fundamentaram a definição da arquitetura proposta nesta Tese.
2. **Definição do Conceito de Gêmeo Digital:** Nesta atividade, foi estabelecida a definição de DT adotada nesta Tese, a partir da análise das principais concepções encontradas na literatura. Para definição do conceito, foi realizado as seguintes atividades:
  - Análise dos conceitos e das definições que antecederam e contribuíram para a consolidação do conceito de DT.
  - Identificação das principais concepções apresentadas na literatura.
  - Análise de aspectos como representação digital, fluxo de dados, sincronização e bidirecionalidade.
  - Delimitação da definição de DT adotada neste trabalho.
3. **Definição da Arquitetura de Middleware:** Nesta atividade, foi definida uma arquitetura modular de middleware para apoiar a comunicação entre o DT e o PT. A arquitetura foi estruturada a partir dos requisitos identificados na literatura e organizada em componentes com responsabilidades específicas.
4. **Implementação da Prova de Conceito:** Nesta atividade, a arquitetura proposta foi materializada por meio do desenvolvimento de uma PoC, com a finalidade de verificar a viabilidade técnica e demonstrar a aplicação dos componentes, interfaces e mecanismos de comunicação definidos na arquitetura.
5. **Avaliação Experimental da PoC em um estudo de caso:** Nesta atividade, a PoC foi avaliada em um estudo de caso baseado em um simulador híbrido de treinamento de alta fidelidade, com a finalidade de analisar seu desempenho e verificar o atendimento aos requisitos definidos para a arquitetura.

## 1.4 Artigos Associados à Tese

Ao longo do desenvolvimento do trabalho apresentado nessa Tese, foram gerados seguintes artigos:

- DOS SANTOS, Lucas S.; MARIA, Luiz C. G.; M. JUNIOR, ERISVALDO F. ; MACHADO, Raphael C. S. Hybrid Simulation Framework: Advanced and Secure Training. In: Seminário Integrado de Software e Hardware, 2024, Brasil. Anais do LI Seminário Integrado de Software e Hardware (SEMISH 2024). p. 13.
- DOS SANTOS, Lucas S.; MARIA, Luiz C. G.; MACHADO, Raphael C. S. Enabling Measurement-Driven Digital Twins through a Modular and Protocol-Agnostic Middleware Framework. In: IEEE International Workshop on Metrology for Industry 4.0 and IoT (MetroInd 2026), 2026, Rome, Italy.
- DOS SANTOS, Lucas S.; MARIA, Luiz C. G.; MACHADO, Raphael C. S. Uma Arquitetura de Middleware Modular e Agnóstica a Protocolos para Sistemas de Gêmeo Digital. In: Seminário Integrado de Software e Hardware (SEMISH 2026), 2026, Brasil.

## 1.5 Organização da Tese

Esta tese está organizada em sete capítulos. O Capítulo 2 apresenta os conceitos e fundamentos que sustentaram o desenvolvimento deste trabalho. São apresentados conceitos relacionados com Gêmeos Digitais, arquitetura em camadas em redes de computadores e os principais protocolos empregados na comunicação entre sistemas físicos e virtuais.

O Capítulo 3 apresenta a revisão dos trabalhos relacionados. O Capítulo contempla arquiteturas orientadas a serviços, middlewares modulares aplicados à Internet das Coisas e frameworks utilizados no desenvolvimento de simuladores.

O Capítulo 4 detalha a arquitetura e funcionalidades do middleware proposta nesta tese.

O Capítulo 5 apresenta os materiais e métodos empregados nesta pesquisa, com foco na validação da proposta apresentada.

O Capítulo 6 apresenta a avaliação experimental da arquitetura proposta. A avaliação contempla tanto a avaliação de desempenho do middleware quanto a validação da

arquitetura proposta.

Capítulo 7 apresenta as considerações finais e conclusões desta tese, destacando as principais contribuições alcançadas, limitações identificadas e direções para trabalhos futuros.

Complementarmente, os apêndices apresentam informações adicionais referentes aos diagramas detalhados, especificação das mensagens da API, configuração do ambiente experimental e procedimentos para instalação e execução do middleware desenvolvido.

## 2 Fundamentação Teórica

Neste Capítulo é discutido o referencial teórico que fundamenta esta pesquisa. A seção 2.1 apresenta os conceitos fundamentais relacionados aos Gêmeos Digitais, abordando seu contexto histórico, definições encontradas na literatura e as categorias de comunicação implementadas. A seção 2.2 introduz os conceitos de arquitetura em camadas em redes de computadores. A seção 2.3 descreve os principais protocolos da camada de aplicação considerados neste trabalho.

### 2.1 Gêmeo Digital

#### 2.1.1 Contexto histórico

O conceito de Gêmeo Digital ([Digital Twin \(DT\)](#)) consolidou-se como um dos pilares da Indústria 4.0, ao promover a integração entre sistemas físicos e virtuais por meio de fluxos contínuos de dados e modelos digitais ([Ala-Laurinaho, 2019](#)). Embora relativamente recente, o termo foi formalmente introduzido em iniciativas da [National Aeronautics and Space Administration \(NASA\)](#) e em trabalhos correlatos ([Shafto et al., 2010, 2012](#); [Schroeder et al., 2016](#)). Embora a terminologia tenha mudado ao longo do tempo, o fundamento do modelo de DT permanece consistente desde as propostas iniciais no início dos anos 2000 ([Grieves; Vickers, 2016](#)).

Em um contexto histórico, diversos conceitos precederam e pavimentaram o caminho para o desenvolvimento do paradigma dos DTs. Em ([Van Brussel et al., 1998](#)), são propostos os “holons”, agentes autônomos que compõem os Sistemas de Manufatura Holônica [Holonic Manufacturing Systems \(HMS\)](#). Os holons podem cooperar entre si, reagir a distúrbios e otimizar processos. O HMS é estruturado em três blocos básicos: (i) *product holon*, que contém informações sobre o produto, como ciclo de vida e lista de materiais; (ii) *resource holon*, voltado à alocação de recursos, formado pela parte física (recurso) e pela parte de processamento de informação; e (iii) *order holon*, uma entidade ativa res-

ponsável por garantir a execução correta e pontual do trabalho. Conforme mostrado na Figura 3, esses três tipos de holons trocam conhecimento sobre o sistema de manufatura. Os *product holons* e os *resource holons* comunicam conhecimento de processo; os *product holons* e os *order holons* trocam conhecimento de produção; e os *resource holons* e os *order holons* compartilham conhecimento de execução do processo.

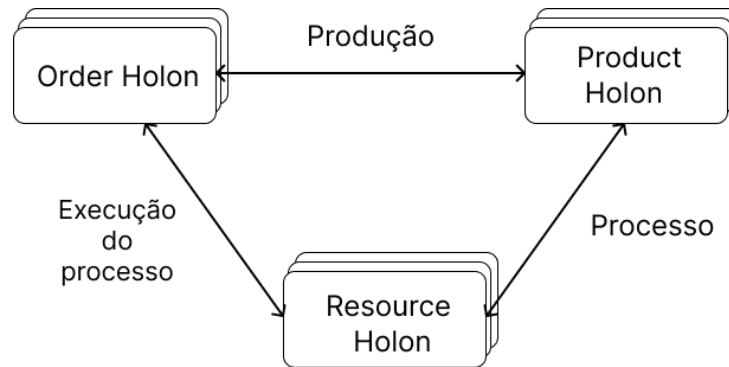


Figura 3: Holonic Manufacturing Systems – HMS. Adaptado de (Van Brussel *et al.*, 1998).

Nessa perspectiva, os autores em (Wong *et al.*, 2002) introduzem o conceito de *Produto Inteligente*. Esse conceito é definido como um ativo que contém pelo menos algumas das seguintes características: (i) identidade única; (ii) capacidade de comunicação com seu ambiente; (iii) habilidade de armazenar informações sobre si; (iv) possibilidade de expressar características ou requisitos de produção; e (v) capacidade de cooperar ou tomar decisões sobre sua própria destinação. Complementarmente, os autores apresentam o conceito de *Agente de Software*. Esse conceito é definido como um processo computacional autônomo, capaz de raciocinar de forma independente, reagir a mudanças de outros agentes ou do ambiente e cooperar com seus pares para alcançar objetivos específicos. Essa integração entre produtos inteligentes e agentes de software está relacionada à noção contemporânea de objetos conectados e autônomos, estabelecendo as bases conceituais da Internet das Coisas (Internet of Things (IoT)) (Ashton *et al.*, 2009; Atzori; Iera; Morabito, 2010).

A Figura 4, apresentada por (Wong *et al.*, 2002), ilustra de forma prática a integração entre o produto físico e sua contraparte digital. No exemplo, é apresentado um frasco de molho (*Spaghetti Sauce*) identificado por uma etiqueta *Radio Frequency Identification (RFID)*. Essa etiqueta permite estabelecer uma conexão, por meio de uma rede local ou remota, entre o produto físico e suas informações, armazenadas em uma base de dados. Além disso, o frasco está associado a um agente de software, responsável por representar digitalmente o produto físico.

Com base na definição de produto inteligente apresentada por (Wong *et al.*, 2002),

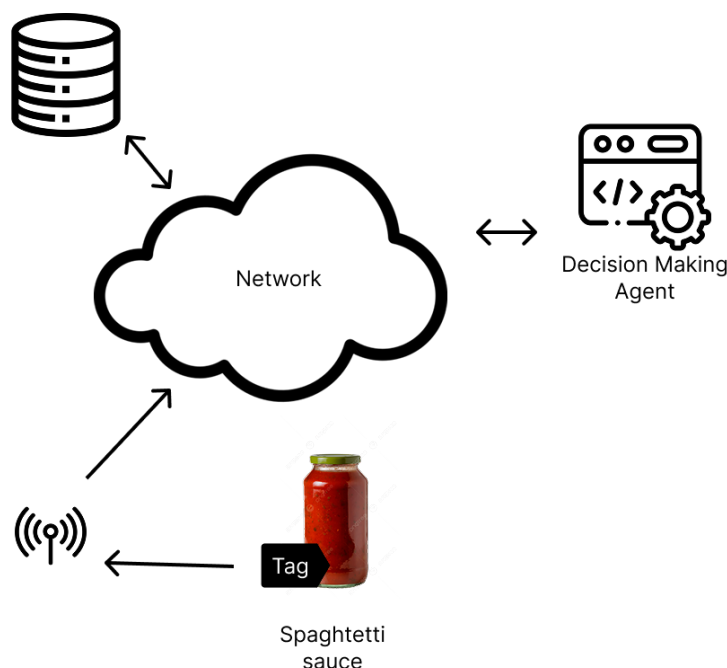


Figura 4: Frasco inteligente de molho de espaguete. Adaptado de (Wong *et al.*, 2002).

(Hribernik, K. *et al.*, 2005) propõem o conceito de *Avatar do Produto* (*Product Avatar*). Nesse conceito, o produto físico possui uma contraparte digital, denominada *Avatar* que é representada em um ambiente virtual. Esse avatar é capaz de tomar decisões de forma autônoma e de atuar como uma entidade individual. Em trabalhos posteriores, (Hribernik, K. A. *et al.*, 2006) apresentam uma abordagem voltada à gestão de informações relacionadas ao produto, denominada *Product-Centric Approach*. Nessa abordagem, o próprio produto se torna o elo central de ligação entre os dados que lhe dizem respeito. Diferentemente da abordagem tradicional, em que as informações são armazenadas de forma fragmentada por diferentes partes interessadas, a abordagem centrada no produto consolida e organiza esses dados em torno do próprio ativo.

Os autores em (Främling *et al.*, 2003) utilizam a definição de Agentes apresentada por (Wong *et al.*, 2002) para propor um arquitetura baseada em agentes (*agent-based architecture*). Essa arquitetura é utilizada para gerenciar o produto ao longo de todo processo produtivo. Nessa perspectiva, cada produto possui um contraparte virtual, denominada agente, no qual é capaz de interagir com outros agentes (softwares e usuários, por exemplo), reagir a mudanças no ambiente e atuar de acordo com objetivos específicos. O agente pode ser acessado via internet, de modo a tornar a informação do produto acessível.

A adoção do modelo baseado em agentes pode ser comparada à mudança de para-

digma ocorrida na programação durante a década de 1980, quando o modelo procedural foi substituído pela orientação a objetos. A programação orientada a objetos propõe a organização do software em torno de entidades que encapsulam dados e comportamentos, conhecidas como objetos, os quais interagem por meio de mensagens (Booch *et al.*, 2008). Assim como a programação orientada a objetos passou a concentrar dados e comportamentos em torno do conceito de objeto, o paradigma orientado a agentes estende essa lógica para ambientes distribuídos, em que cada agente representa e gerencia um elemento físico específico do mundo real (Wooldridge; Jennings, 1995). Dessa forma, esses paradigmas resgata o conceito de virtualização e representação de elementos físicos em ambientes computacionais.

No exemplo ilustrado pela Figura 5, o produto físico é acompanhado de uma representação digital que armazena e gerencia suas informações de forma sincronizada. Quando uma determinada informação é necessária — por exemplo, durante etapas de transporte, operação ou manutenção — ela é obtida de forma dinâmica na base de dados compartilhada. Esse mecanismo reduz redundâncias e assegura que o sistema trabalhe sempre com informações atualizadas, reforçando o papel dos agentes como mediadores inteligentes entre o domínio físico e o digital.

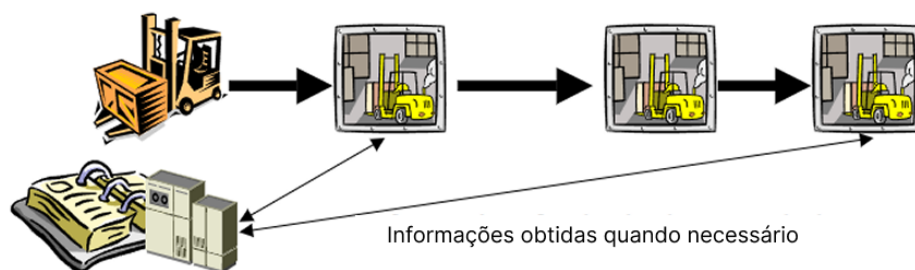


Figura 5: Ilustração de acompanhamento de Produto. Extraído de (Främling *et al.*, 2003).

A consolidação do conceito de Gêmeo Digital ocorreu a partir dos trabalhos de (Grieves, 2006), que introduziram o *Mirrored Spaces Model (MSM)*. Esse modelo é composto por três elementos principais: o *espaço real*, o *espaço virtual* e o *vínculo de comunicação* entre ambos. No *MSM*, cada objeto presente no espaço virtual é conectado ao seu correspondente físico no espaço real, de modo que o primeiro espelha o estado e o comportamento do segundo. Em síntese, o modelo de (Grieves, 2006) estabelece a base conceitual a base teórica do conceito de DT. Embora o termo tenha sido popularizado por (Shafito *et al.*, 2010) no contexto da NASA, (Grieves, 2006) é amplamente reconhecido como o criador do conceito.

A próxima seção aprofunda a discussão sobre os DTs, apresentando suas definições

formais e as respectivas características fundamentais. Esse detalhamento tem como finalidade estabelecer uma compreensão conceitual sólida, que serviu de base para a análise e o desenvolvimento da arquitetura proposta nesta pesquisa.

### 2.1.2 Conceito e Definição

Existem diversas definições de **DT** na literatura, que variam de acordo com o contexto de aplicação e com a perspectiva adotada pelos autores (Ala-Laurinaho, 2019). De modo geral, as definições convergem quanto à ideia central de que o **DT** representa uma contraparte digital de um ativo físico, conectada de forma bidirecional e contínua ao seu equivalente real. A seguir, são apresentadas algumas definições descritas na literatura:

- De acordo com (Shafto *et al.*, 2010, 2012), um **DT** consiste em uma simulação integrada, multifísica, multiescala e probabilística de um veículo ou sistema que utiliza os melhores modelos físicos disponíveis, atualizações provenientes de sensores, histórico da frota e outras informações para espelhar a vida de seu correspondente físico em operação. Além disso, o autor afirma que um **DT** integra dados de sensores, histórico de manutenção e dados de frota. Um **DT** contém, essencialmente, todos os dados disponíveis ao longo do ciclo de vida completo de seu correspondente físico, desde anomalias ocorridas durante a fabricação até os dados gerados durante sua operação. Além de ser capaz de representar com precisão o estado atual de seu correspondente físico, os autores propõem que o Gêmeo Digital também seja utilizado para executar simulações capazes de prever estados futuros do sistema físico correspondente.

Segundo (Shafto *et al.*, 2010, 2012), os **DTs** podem ser empregados em diversas aplicações, dentre as quais destacam-se:

- Simulação de missões antes de sua execução, permitindo avaliar cenários operacionais, riscos e probabilidades de sucesso.
- Monitoramento contínuo do sistema físico durante sua operação, possibilitando a previsão de estados futuros por meio de simulações.
- Análise das causas de falhas e anomalias observadas durante a operação.
- Avaliação dos impactos decorrentes de alterações em parâmetros operacionais, auxiliando a tomada de decisão em situações adversas.
- Certificação de sistemas e veículos por meio de simulações computacionais.

- Para (Grieves; Vickers, 2016), os DTs podem ser compreendidos como uma representação virtual conectada de um produto físico. O ativo virtual mantém uma sincronização bidirecional e contínua com o objeto real ao longo de todo o seu ciclo de vida. Essa definição tem origem no conceito de MSM (conforme visto na seção 2.1.1). Os MSMs estruturam o fluxo de informações entre os espaços físico e virtual, o que permite o gerenciamento integrado de produtos e processos. De acordo com o autor, o DT surge como uma abordagem voltada a mitigar o comportamento imprevisível e emergente característico de sistemas complexos. Esses sistemas são compostos por múltiplos subsistemas interconectados, que interagem de forma dinâmica e não linear, tornando difícil antecipar como pequenas variações locais podem gerar consequências em larga escala. A imprevisibilidade dessas interações pode resultar em falhas sistêmicas, acidentes ou comportamentos indesejáveis. Nesse contexto, o DT atua como uma ferramenta de observação, simulação e análise preditiva, permitindo compreender a interação entre componentes físicos e digitais, antecipar anomalias e validar o desempenho do sistema em condições reais e virtuais. Assim, a concepção de DT está intimamente relacionada à necessidade de compreender e controlar sistemas complexos, oferecendo um meio de projetar, monitorar e evoluir sistemas críticos com maior segurança e confiabilidade.
- Os autores em (Boschert; Rosen, 2016; Gabor et al., 2016) descrevem um DT por uma perspectiva de simulação. Em (Gabor et al., 2016) essa visão é ampliada ao definir um DT como uma simulação de altíssima fidelidade voltada à predição, otimização e apoio à decisão. Para os autores, portanto, um DT é capaz de integrar modelos estruturais desacoplados de modo a possibilitar simulação de um modo abrangente. Em outras palavras, essa integração possibilita representar o comportamento do sistema como um todo, considerando suas diferentes partes e interações. Complementarmente, os autores destacam que um DT pode integrar conhecimento proveniente de diferentes sistemas de DT, utilizando as informações coletadas desses sistemas para aumentar a precisão das simulações.

Complementarmente, em (Fuller et al., 2020) são apresentados três conceitos fundamentais para o entendimento da evolução e do nível de integração entre o mundo físico e o digital: o *Digital Model*, o *Digital Shadow* e o *Digital Twin*. O *Digital Model* refere-se a uma representação digital de um objeto físico existente ou projetado, porém sem qualquer forma de comunicação automática entre os dois domínios. Alterações realizadas no sistema físico não afetam o modelo digital, e vice-versa. O *Digital Shadow* representa um

avanço nesse nível de integração, estabelecendo um fluxo unidirecional de dados: mudanças no estado físico são refletidas no modelo digital, mas não há retroalimentação para o sistema real. Finalmente, o *Digital Twin* caracteriza-se pela integração bidirecional e contínua entre os mundos físico e digital, de modo que alterações em qualquer um dos lados são automaticamente refletidas no outro. Essa reciprocidade estabelece um ciclo de retroalimentação dinâmica, que permite ao DT não apenas espelhar, mas também influenciar o comportamento do ativo físico, constituindo assim a forma mais completa e evoluída de representação virtual de sistemas reais. A Figura 6 ilustra essas três categorias e os respectivos níveis de integração entre o objeto físico e sua contraparte digital, conforme proposto por (Fuller *et al.*, 2020).

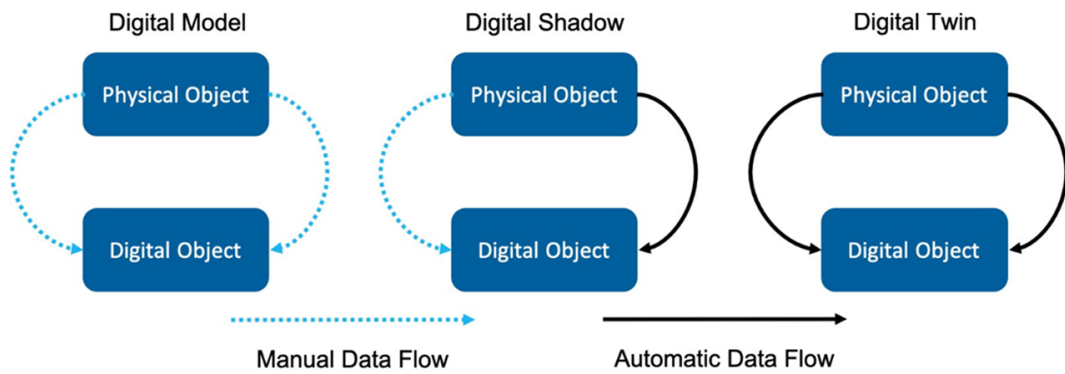


Figura 6: Níveis de integração entre modelo digital, sombra digital e gêmeo digital. Adaptado de (Fuller *et al.*, 2020).

A Figura 7 apresenta uma representação do fluxo de comunicação dentro de um DT com seu respectivo ativo físico.

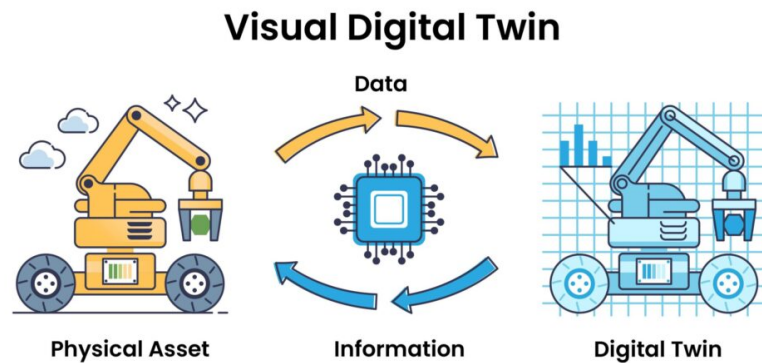


Figura 7: Representação em alto nível do Digital Twin. Extraído de (Prevu3D, 2025).

### 2.1.3 Categorias de comunicação nos Gêmeos Digitais

De acordo com (Ala-Laurinaho, 2019), os tipos de comunicação em sistemas de Gêmeos Digitais são classificados em cinco categorias principais. Essa taxonomia é baseada nos diferentes fluxos de dados e interações entre o Gêmeo Digital, o Gêmeo Físico e outros elementos do ecossistema. A Figura 8 ilustra as diferentes categorias de comunicação. As categorias são descritas a seguir:

1. **Comunicação entre subsistemas internos de um Gêmeo Digital:** ocorre dentro do próprio Gêmeo Digital, entre seus módulos ou componentes, como armazenamento de dados, simulação e tomada de decisão. Essa comunicação é fundamental para a interoperabilidade interna e a coerência das informações trocadas entre subsistemas e, conseqüentemente, para atendimento às regras de negócio.
2. **Comunicação entre diferentes Gêmeos Digitais:** nessa perspectiva, um DT pode ser composto por outros DTs. Portanto, a comunicação pode ser dividida em duas categorias: a comunicação entre os Gêmeos Digitais que compõem um Gêmeo Digital maior, como entre o carro de translação (*trolley*) e a ponte de uma ponte rolante; e a comunicação entre um DT e os DTs de outras máquinas, como entre Gêmeos Digitais de pontes rolantes localizadas no mesmo galpão industrial. Por meio dessa comunicação, os DTs podem compartilhar informações sobre o ambiente em que estão inseridos, bem como sobre seus estados e ações futuras.
3. **Comunicação entre o Gêmeo Digital e sistemas externos:** abrange a integração com sistemas corporativos, como Manufacturing Execution Systems (MES), bancos de simulação, plataformas em nuvem e sistemas de gerenciamento de dados. Esses sistemas fornecem suporte operacional, armazenamento e processamento distribuído.
4. **Comunicação entre o Gêmeo Digital e o Gêmeo Físico:** esse tipo de comunicação representa o núcleo conceitual dos sistemas de Gêmeos Digitais, possibilitando o fluxo bidirecional de dados entre o ativo físico e sua representação digital. Inclui tanto o envio de dados de sensores, controladores e dispositivos de automação — como Controlador Lógico Programável (CLP) — e estados do físico para o digital, quanto o envio de comandos e atualizações do digital para o físico.
5. **Comunicação entre o Gêmeo Digital e o ambiente:** envolve a troca de informações entre o Gêmeo Digital e o contexto físico em que o sistema está inserido,

como sensores ambientais, dispositivos inteligentes ou outros elementos do entorno. Essa comunicação contribui para aumentar a segurança, otimizar a operação e favorecer a cooperação contextual.

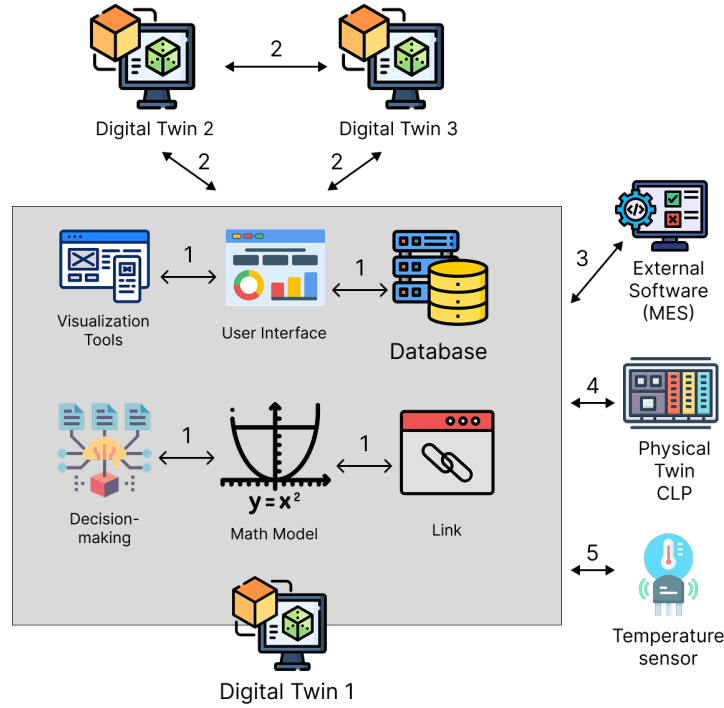


Figura 8: Categorias de comunicação em cinco categorias. Adaptado de (Ala-Laurinaho, 2019).

Embora existam diversas categorias e definições de comunicação dentro do ecossistema dos Gêmeos Digitais, no contexto desta tese, o foco está na comunicação entre o Gêmeo Digital e o Gêmeo Físico (Categorias 4 e 5). Essa escolha se fundamenta no fato de que um DT, pela definição, contém conhecimento contínuo do estado do PT (Ala-Laurinaho, 2019). Os dados coletados por sensores (Message Queuing Telemetry Transport (MQTT), Rabbit e Constrained Application Protocol (CoAP), por exemplo), controladores e dispositivos de automação são utilizados para atualizar constantemente o estado e a representação do modelo digital, de modo a garantir que ele reflita o comportamento e as condições operacionais do sistema real. Além disso, a comunicação do DT com Physical Twin (PT) representa um aspecto crítico da solução, pois os mecanismos de comunicação e decisões arquiteturais adotadas na implementação podem impactar diretamente a escalabilidade, modularidade e o desempenho da solução.

## 2.2 Arquitetura em Camadas em Redes de Computadores

O modelo de referência [Open Systems Interconnection \(OSI\)](#), desenvolvido pela ISO, é amplamente utilizado como base conceitual para descrever o funcionamento das comunicações em rede. Ele é composto por sete camadas funcionais: física, enlace, rede, transporte, sessão, apresentação e aplicação ([Ross; Kurose, 2012](#); [Zimmermann, 1980](#)), conforme ilustrado na Figura 9. As três primeiras são responsáveis pela transmissão dos dados através dos meios físicos e pelo endereçamento e roteamento das informações. A camada de transporte assegura a entrega confiável e ordenada das mensagens entre os sistemas. Já as camadas superiores lidam com a comunicação entre processos: a camada de sessão é responsável por estabelecer, gerenciar e encerrar conexões entre aplicações; enquanto a camada de apresentação trata da tradução, compressão e criptografia dos dados antes de serem entregues à aplicação.

O modelo [TCP/IP](#), também conhecido como Pilha de Protocolos de [TCP/IP](#) (*Internet Protocol Stack*), é uma organização prática do modelo [OSI](#), adotada como padrão na Internet e em sistemas distribuídos. O modelo organiza o processo de comunicação em quatro camadas: Enlace, Rede, Transporte e Aplicação ([Ross; Kurose, 2012](#)), conforme apresentado na Figura 9. As três camadas superiores do modelo [OSI](#) (Aplicação, Apresentação e Sessão) são representadas de forma consolidada pela camada de Aplicação no modelo [TCP/IP](#). As camadas de Enlace e Física do modelo [OSI](#) são agrupadas na camada de Acesso à Rede do modelo [TCP/IP](#). Essa organização evidencia uma arquitetura mais simplificada e pragmática do modelo [TCP/IP](#), voltada à implementação prática dos protocolos que sustentam a arquitetura da Internet. Nas duas arquiteturas, cada camada fornece serviços à camada superior e utiliza os serviços da camada inferior, mantendo a estrutura hierárquica e modular que define o modelo de comunicação.

O modelo [TCP/IP](#) é utilizado como referência no contexto desta tese. A proposta da arquitetura do Middleware, proposta neste trabalho, permeia as camadas de transporte e rede. Diferente de abordagens que utilizam protocolos de aplicação Web — como [HTTP](#) ou [MQTT](#) —, a arquitetura desenvolvida neste trabalho adota uma camada de comunicação direta, implementada por meio de *sockets*. Essa escolha permite o controle sobre os mecanismos de envio e recebimento de dados, reduzindo o overhead associado a protocolos de alto nível e assegurando maior desempenho na troca de informações entre o Gêmeo Digital e o Gêmeo Físico. Além disso, são exploradas decisões arquiteturais, como comunicação assíncrona, que viabilizam a redução do consumo de banda e, conseqüentemente, proporcionam a redução do overhead da rede.

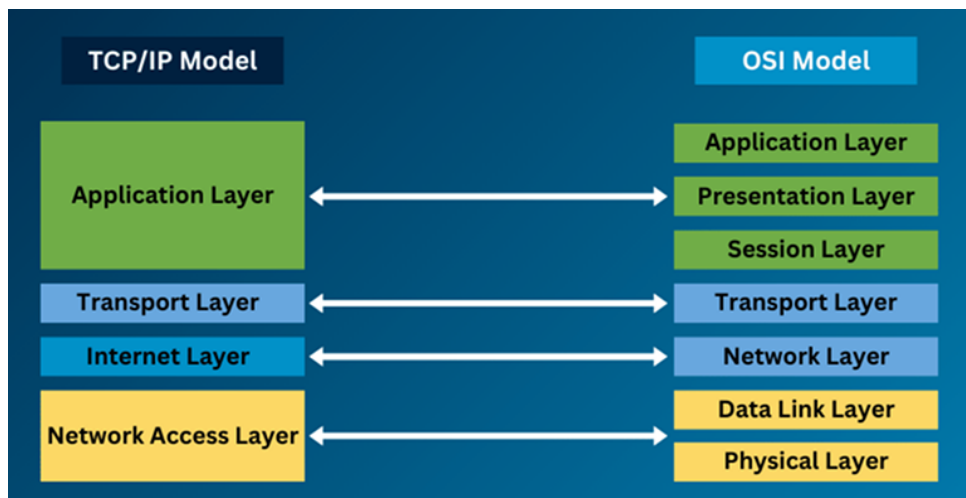


Figura 9: Modelo TCP/IP vs OSI.

## 2.3 Protocolos da Camada de Aplicação

Esta seção descreve os protocolos de comunicação que estão implementados na Camada de Aplicação, seguindo o modelo de referência do TCP/IP. Para cada protocolo, são descritos os conceitos, arquiteturas, vantagens e desvantagens recorrentes de suas respectivas implementações.

### 2.3.1 HTTP e REST

O [Hypertext Transfer Protocol \(HTTP\)](#) é um protocolo da camada de aplicação amplamente utilizado na Internet. O protocolo é o principal responsável pela comunicação entre navegadores e servidores web, na arquitetura cliente-servidor. Originalmente desenvolvido para a transferência de documentos hipermídia, o [HTTP](#) evoluiu para um protocolo genérico de comunicação entre sistemas distribuídos. O protocolo desempenhou papel central em arquiteturas cliente-servidor e em serviços baseados na Web ([Ross; Kurose, 2012](#); [Tanenbaum; Wetherall et al., 2019](#)). A Figura 10 ilustra o comportamento de requisição e resposta implementado pelo [HTTP](#).

O [HTTP](#) implementa o modelo de requisição-resposta (*request-response*), no qual um cliente estabelece uma conexão com o servidor e envia uma requisição especificando o método, o recurso solicitado e a versão do protocolo. Por outro lado, o servidor responde com um código de status e, opcionalmente, com um corpo de mensagem contendo os dados requisitados. Os métodos mais utilizados são *GET*, *POST*, *PUT* e *DELETE*, que correspondem, respectivamente, às operações de leitura, envio, atualização e exclusão de recursos. Cada requisição e resposta contém também um conjunto de cabeçalhos

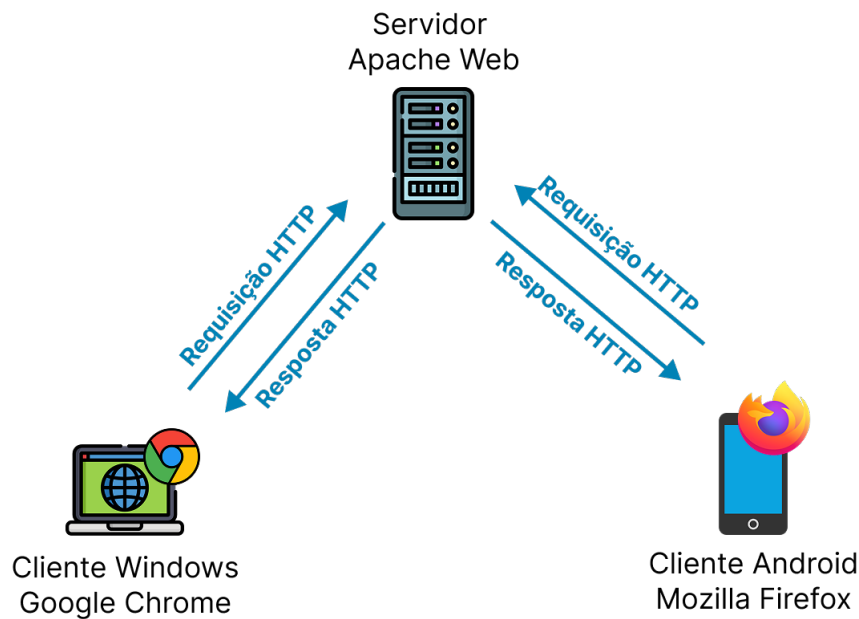


Figura 10: Arquitetura Cliente-Servidor do [HTTP](#). Extraído de ([Ross; Kurose, 2012](#)).

(*headers*) em formato texto, utilizados para transmitir informações adicionais, como tipo de conteúdo, codificação e controle de cache ([Ross; Kurose, 2012](#)).

Na camada de transporte (Figura 9), o [HTTP](#) utiliza o protocolo [TCP](#), que fornece um canal de comunicação confiável e orientado à conexão. Antes de qualquer transferência de dados, o [TCP](#) realiza um processo de *three-way handshake* para estabelecer a conexão, o que implica, no mínimo, dois tempos de ida e volta (*Round-Trip Times* – RTTs) entre cliente e servidor. Esse mecanismo garante a entrega ordenada e livre de erros dos segmentos, mas também adiciona latência à comunicação ([Tanenbaum; Wetherall et al., 2019](#)). O [HTTP/1.1](#) introduziu o conceito de conexões persistentes, permitindo o envio de múltiplas requisições sobre uma mesma conexão [Transmission Control Protocol \(TCP\)](#), reduzindo o custo de abertura de novos canais e melhorando a eficiência da comunicação. Posteriormente, o [HTTP/2](#) aprimorou esse modelo ao introduzir o *multiplexing*, possibilitando o envio simultâneo de várias requisições e respostas sobre um único canal, sem bloqueios ([Ross; Kurose, 2012](#)).

Por ser um protocolo textual, o [HTTP](#) é legível e amplamente compatível. Porém, o protocolo apresenta desvantagens em aplicações que demandam comunicação de baixa latência ou dispositivos com recursos limitados. O uso de mensagens em texto plano aumenta o tamanho dos cabeçalhos e, conseqüentemente, o consumo de banda e o tempo de processamento. Além disso, o [HTTP](#) é um protocolo sem estado (*stateless*), isto é, o servidor não persiste informações sobre o cliente entre requisições consecutivas. Para contornar essa limitação, são utilizados mecanismos de sessão e *cookies*, que permitem a

identificação e o acompanhamento do cliente durante múltiplas interações (Tanenbaum; Wetherall *et al.*, 2019; Ross; Kurose, 2012). As especificações do HTTP [RFC 1945; RFC 2616] (Berners-Lee; Fielding; Frystyk, 1996; Fielding *et al.*, 1999) definem o formato das mensagens do protocolo. No tocante à estrutura, uma mensagem HTTP segue o formato com os campos de linha de requisição, cabeçalhos, linha em branco e corpo da mensagem. Nesse sentido, há dois tipos de mensagens: de requisição e de resposta. A mensagem de requisição.

A Figura 11 apresenta a estrutura da mensagem de requisição do HTTP. A linha de requisição inclui o método HTTP utilizado, o endereço (URL) do servidor e a versão do protocolo. Podem existir diversas linhas de cabeçalho, que são pares “nome–valor” responsáveis por transmitir informações adicionais sobre a requisição. A linha de requisição é separada das linhas de cabeçalho por um caractere de controle (Carriage Return (CR)) e avanço de linha (\r\n), e as linhas de cabeçalho também são separadas entre si pela mesma sequência. Após as linhas de cabeçalho, há um caractere de controle (CR) e avanço de linha adicional, seguido pelos possíveis dados da mensagem.

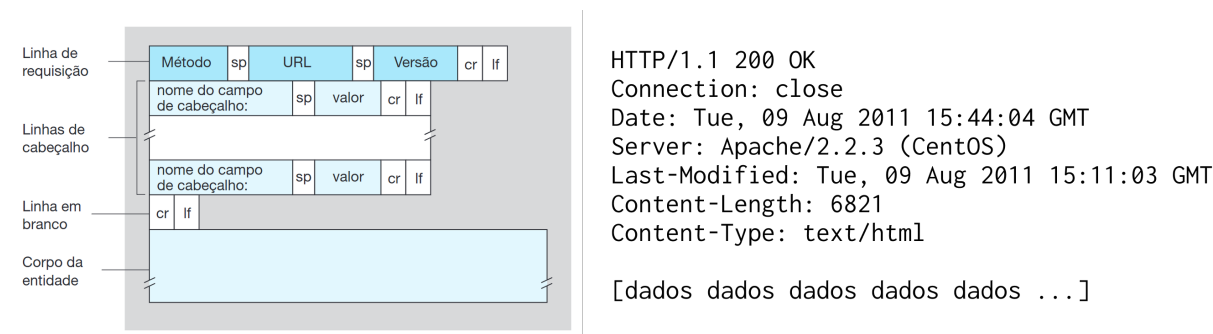


Figura 11: Estrutura de mensagem de requisição HTTP. Extraído de (Ross; Kurose, 2012).

A Figura 12 apresenta a estrutura e um exemplo da mensagem de resposta do [HTTP](#). A resposta do [HTTP](#) segue o mesmo formato da requisição, porém a **linha de estado** é substituída pela **linha de resposta**. Esta linha contém a versão do protocolo, status code, consiste a versão, e uma explicação sobre o status code.

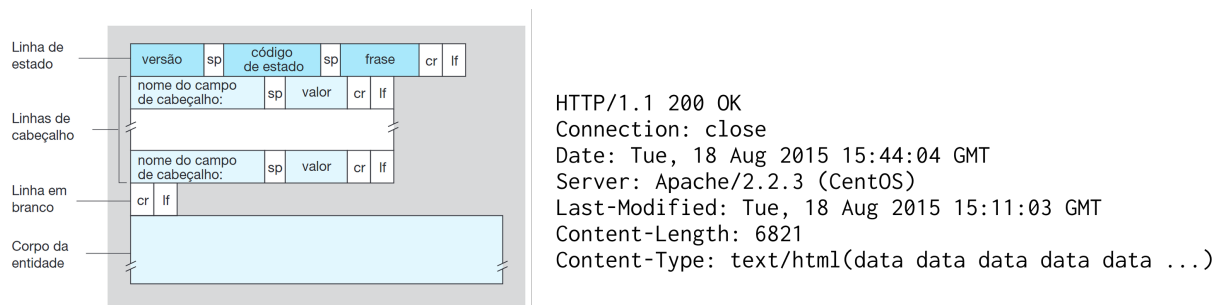


Figura 12: Estrutura de mensagem de requisição [HTTP](#). Extraído de ([Ross; Kurose, 2012](#)).

O [Representational State Transfer \(REST\)](#) é um conjunto de critérios de projeto (*design criteria*) para sistemas hipermídia distribuídos, proposto originalmente por Fielding em sua tese de doutorado ([Fielding, 2000](#)). Esses critérios definem princípios arquiteturais que visam à construção de sistemas escaláveis, interoperáveis e de fácil manutenção. Quando uma aplicação implementa essas restrições de forma consistente, ela é denominada *RESTful* ([Fielding, 2000](#); [Ala-Laurinaho, 2019](#)). De forma prática, serviços “RESTful” utilizam o [HTTP](#) como base, mapeando recursos a [URLs](#) e explorando as semânticas do protocolo (métodos, códigos de status, cache e negociação de conteúdo) ([Ross; Kurose, 2012](#); [Tanenbaum; Wetherall et al., 2019](#)).

Em uma arquitetura [REST](#), o cliente envia requisições a um servidor, que, por sua vez, fornece representações dos recursos solicitados. Cada recurso possui um identificador único, geralmente uma [URL](#), que permite seu acesso e manipulação. Esse paradigma segue o modelo proposto pela [Resource-Oriented Architecture \(ROA\)](#), descrita por Richardson e Ruby ([Richardson; Ruby, 2007](#)), na qual cada recurso é tratado como uma entidade endereçável e manipulável por meio de uma interface uniforme.

### 2.3.2 MQTT

O [MQTT](#) é um protocolo de comunicação da camada de aplicação, amplamente utilizado em sistemas distribuídos e aplicações de [IoT](#), devido à sua simplicidade e eficiência em dispositivos com recursos limitados. O protocolo foi originalmente desenvolvido pela [IBM](#) na década de 1990, com o objetivo de oferecer um mecanismo de troca de mensagens

confiável em redes com alta latência e largura de banda restrita (Naik, 2017). Atualmente, o MQTT é padronizado pela Organization for the Advancement of Structured Information Standards (OASIS).

O MQTT adota uma arquitetura de comunicação publicação–assinatura *Publish–Subscribe* (Pub/Sub), em contraste com o modelo de requisição–resposta do HTTP. Nesse modelo de comunicação, os clientes não estabelecem uma comunicação direta entre si. A comunicação é realizada por meio de um componente intermediário denominado *broker*. Esse componente opera como um servidor central responsável pelo roteamento das mensagens. Os clientes podem publicar (*Publish*) mensagens em tópicos específicos ou se inscrever (*Subscribe*) nesses tópicos para receber atualizações. Quando uma mensagem é publicada em um tópico, o *Broker* distribui a mensagem automaticamente a todos os clientes que se inscreveram no respectivo tópico. Esse modelo proporciona um desacoplamento temporal e espacial entre os emissores e receptores (Santos, 2020; Silva; Moraes; Amorim, 2013). Essa característica o torna altamente escalável e eficiente em aplicações distribuídas (Naik, 2017). A Figura 13 apresenta a arquitetura básica do MQTT.

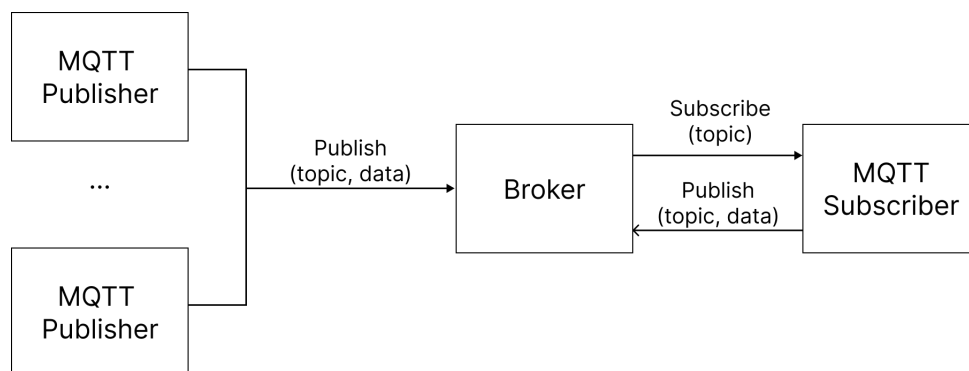


Figura 13: Modelo de comunicação *publish–subscribe* do protocolo MQTT. Extraído de (Durkop; Czybik; Jasperneite, 2015; Naik, 2017).

Na arquitetura do MQTT, um *tópico* é uma cadeia hierárquica de texto que identifica logicamente o canal de comunicação pelo qual as mensagens são trocadas. A hierarquia é definida por níveis separados por barras (Figura 14a), o que permite granularidade na filtragem e organização dos dados. O protocolo suporta o uso de “curingas” (*wildcards*) — o símbolo `+` representa um único nível na hierarquia (Figura 14b), enquanto `#` corresponde a múltiplos níveis —, possibilitando que um cliente receba mensagens de diferentes subárvores de tópicos de forma flexível e eficiente (Figura 14c).

A estrutura da mensagem MQTT é composta por três partes principais: *fixed header* (cabeçalho fixo), *variable header* (cabeçalho variável) e *payload* (corpo da mensagem), conforme ilustrado na Figura 15. O cabeçalho fixo está presente em todas as mensagens

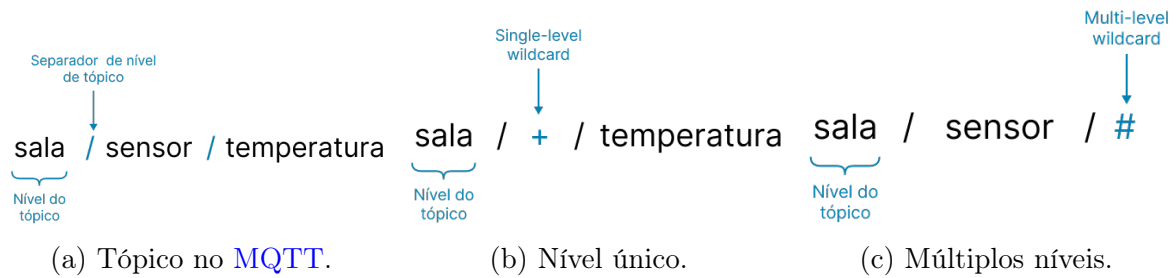


Figura 14: Estrutura hierárquica de tópicos no MQTT.

e contém o tipo de mensagem, um conjunto de *flags* específicos de cada tipo e o campo *Remaining Length*, que indica o tamanho restante da mensagem. Esse campo pode ocupar de um a quatro bytes, dependendo da quantidade de dados a serem transmitidos. Por esse motivo, o tamanho mínimo de uma mensagem MQTT é de dois bytes. O cabeçalho variável é incluído apenas em determinados tipos de mensagens e contém informações específicas relacionadas à função que o pacote desempenha no protocolo. Finalmente, o *payload* transporta o conteúdo da mensagem — por exemplo, os dados de um sensor publicados em um tópico.

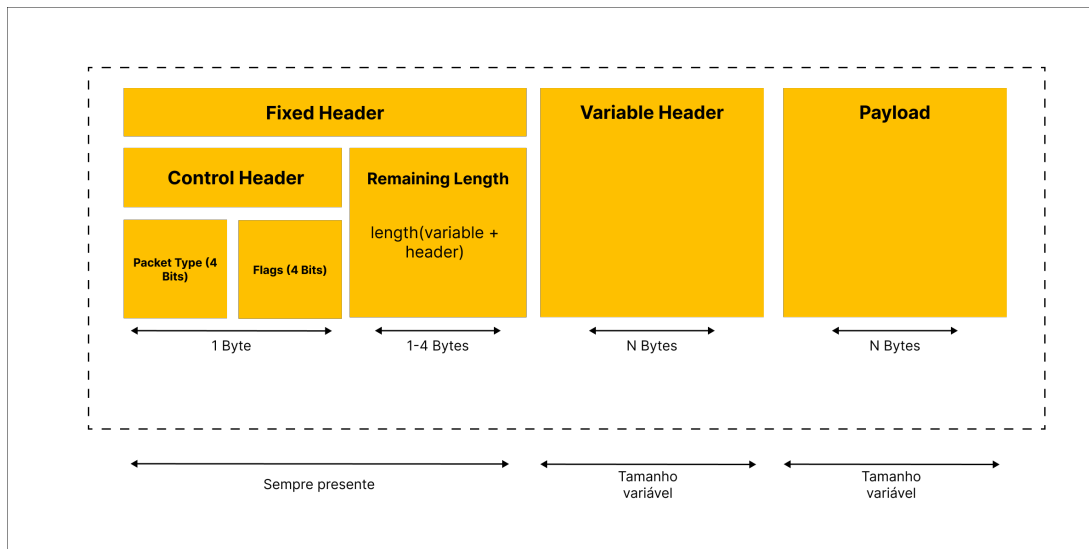


Figura 15: Estrutura de mensagem do MQTT.

Entre as principais vantagens do MQTT, destacam-se o tamanho mínimo de dois bytes (conforme ilustrado na Figura 15), o baixo consumo de *CPU* e memória, além da baixa latência em cenários com baixa taxa de amostragem. Contudo, o protocolo apresenta aumento de latência em aplicações que operam com altas taxas de amostragem, o que pode comprometer o desempenho em ambientes que demandam comunicação em tempo quase real.

### 2.3.3 GraphQL

O [Graph Query Language \(GraphQL\)](#) é uma linguagem de consulta e manipulação de dados para [Application Programming Interfaces \(APIs\)](#), criada originalmente pelo *Facebook* em 2012 e disponibilizada como projeto de código aberto em 2015 ([Facebook, Inc., 2015](#)). Diferentemente do modelo tradicional de comunicação baseado em arquitetura [REST](#), em que múltiplas requisições são necessárias para acessar recursos distintos, o [GraphQL](#) permite que o cliente defina explicitamente quais dados deseja obter em uma única requisição. Essa característica torna as comunicações mais eficientes e reduz a sobrecarga na rede e no processamento de dados, especialmente em aplicações distribuídas e sensíveis à latência ([Hartig; Pérez, 2019](#)).

A estrutura de tipos é baseada em três tipos comuns a todos os servidores [GraphQL](#). Esses tipos também correspondem às operações básicas utilizadas para realizar consultas ao servidor. O tipo *Query* é empregado exclusivamente para operações de leitura de dados. O tipo *Mutation* é utilizado para operações de escrita e modificação de dados no servidor, podendo também retornar informações como resultado da operação executada. Por sua vez, o tipo *Subscription* é destinado ao recebimento de dados em resposta a eventos específicos. A partir desses tipos de operação, as consultas são refinadas por meio de campos e subtipos, permitindo o acesso a informações cada vez mais específicas.

O funcionamento do [GraphQL](#) baseia-se em um único *endpoint* que centraliza todas as operações de consulta (*query*), modificação (*mutation*) e assinatura de eventos (*subscription*). As consultas são estruturadas de forma declarativa, permitindo que o cliente especifique apenas os campos necessários, enquanto o servidor retorna exatamente os dados solicitados, evitando redundância e minimizando o tráfego de informação ([Foundation, 2023](#)). Além disso, o modelo de tipos fortemente definidos do [GraphQL](#) facilita a validação e o autodescobrimento de esquemas, tornando a integração entre diferentes componentes mais previsível e segura.

### 2.3.4 OPC UA

O [Open Platform Communications Unified Architecture \(OPC UA\)](#) é um padrão de comunicação industrial desenvolvido pela *OPC Foundation*, projetado para oferecer interoperabilidade segura e independente de plataforma entre dispositivos, sistemas e aplicações industriais ([OPC Foundation, 2017a](#)). O [OPC UA](#) é amplamente implementado em aplicações de [Industrial Internet of Things \(IIoT\)](#) e em comunicações [Machine-to-Machine](#)

(M2M) (OPC Foundation, 2017a). Diferentemente das versões anteriores, baseadas em *COM/DCOM* (OPC Data Access), o OPC UA adota uma arquitetura orientada a serviços *Service-Oriented Architecture* (SOA), o que permite que dispositivos de automação, sistemas de controle e aplicações corporativas troquem informações de forma padronizada, extensível e independente de fornecedor (Ala-Laurinaho, 2019; Farnham; Barillere, 2011).

O OPC UA suporta dois principais modelos de comunicação: *Client-Server* e *Pub/Sub* (Ala-Laurinaho, 2019; OPC Foundation, 2017a). No modelo *Client-Server*, o servidor disponibiliza um conjunto de serviços agrupados em *Service Sets*, como o *Secure-Channel Service Set* (responsável pelo estabelecimento de canais seguros de comunicação) e o *Session Service Set* (responsável pelo gerenciamento de sessões). Cada servidor especifica aos clientes quais conjuntos de serviços estão disponíveis. Além disso, o OPC UA define mecanismos assíncronos, como *Alarms and Conditions*, usados para notificação de eventos, e *Historical Access*, que permite o acesso a dados históricos de operação.

Por outro lado, Na arquitetura *Pub/Sub*, dois tipos de *Message Oriented Middleware* (MOM) são usados para distribuir mensagens entre publicadores e assinantes: (i) (*broker-less*), onde a infraestrutura de rede roteia diretamente as mensagens entre publicadores e assinantes, utilizando protocolos como UDP multicast; e (ii) (*broker-based*), em que publicadores e assinantes trocam mensagens através de um *broker*, por meio protocolos padronizados como MQTT.

O modelo de informação do OPC UA é orientado a objetos. Os objetos são organizados como nós em um grafo, denominado AddressSpace (OPC Foundation, 2017a; Mahnke; Leitner; Damm, 2009). A arquitetura da OPC UA é exibida na Figura 16.

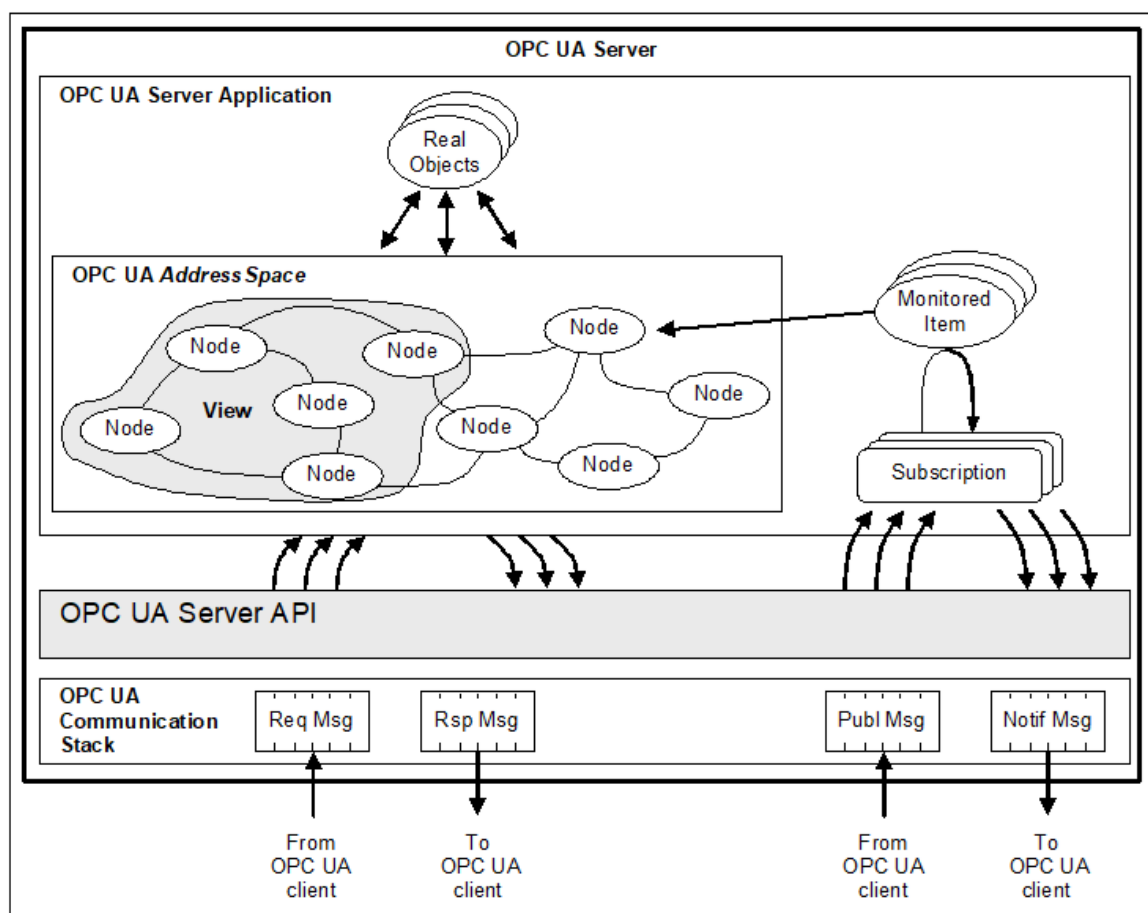


Figura 16: Arquitetura do servidor OPC UA (OPC Foundation, 2017b).

## 3 Trabalhos Relacionados

Este Capítulo apresenta os trabalhos relacionados a esta tese por meio de quatro seções. A Seção 3.1 apresenta estudos que exploram arquiteturas orientadas a serviço, com ênfase no uso do OPC UA e em suas extensões RESTful aplicadas à interoperabilidade em sistemas industriais. A Seção 3.2 discute paradigmas de middleware consolidados no estado da arte, incluindo plataformas Internet of Things (IoT)/Digital Twin (DT) amplamente utilizadas, com destaque em suas contribuições e limitações. A Seção 3.3 aborda frameworks voltados ao desenvolvimento de simuladores, ressaltando sua relevância para a integração entre componentes físicos e virtuais em ambientes de Gêmeo Digital. Finalmente, a Seção 3.4 apresenta as considerações finais, sintetizando os principais pontos discutidos e posicionando a proposta desta Tese em relação às lacunas identificadas na literatura.

### 3.1 Arquiteturas Orientadas a Serviço

Sistemas baseados em arquiteturas orientadas a serviço têm ganhado destaque nos últimos anos em função da crescente demanda por interoperabilidade em ambientes distribuídos. Nesse contexto, os serviços web consolidaram-se como uma solução adequada para a integração de aplicações heterogêneas, distribuídas em diferentes plataformas e dispositivos, ao viabilizarem a troca de dados e funcionalidades por meio da internet, utilizando protocolos padronizados, como o Hypertext Transfer Protocol (HTTP). Essa abordagem favorece características como flexibilidade e escalabilidade, o que tem impulsionado sua adoção em diversos domínios, incluindo o industrial, onde contribui para a conectividade e integração de sistemas associados à IoT e a Cyber-Physical System (CPS).

O trabalho apresentado em (Melik-Merkumians *et al.*, 2012) analisa a viabilidade do protocolo Open Platform Communications Unified Architecture (OPC UA) como tecnologia base para a implementação de Arquiteturas Orientadas a Serviços (Service-Oriented Architecture (SOA)) em sistemas de automação industrial. O estudo demonstra que o OPC UA atende a requisitos fundamentais de portabilidade, segurança, eficiência e

compatibilidade. Nesse sentido, a solução se posiciona como um *middleware* para integração entre softwares de controle e aplicações externas, como [Enterprise Resource Planning \(ERP\)](#) ([Davenport, 2000](#)), [Manufacturing Execution Systems \(MES\)](#) ([Kletti, 2007](#)) e [Supervisory Control and Data Acquisition \(SCADA\)](#) ([Gaushell; Darlington, 1987](#)). A principal contribuição do trabalho está em evidenciar o [OPC UA](#) como um mecanismo viável para suportar a integração orientada a serviços em ambientes de automação. Contudo, a proposta permanece fortemente acoplada a um único protocolo, restringindo sua aplicação a cenários nos quais o [OPC UA](#) é utilizado. Nesse sentido, a proposta apresentada nesta Tese diferencia-se por adotar um *design* modular e *protocol-agnostic*, capaz de integrar múltiplos padrões de comunicação além do [OPC UA](#).

Em ([Grüner; Pfrommer; Palm, 2016](#)) é apresentada uma proposta de extensão da arquitetura [OPC UA](#) para integrar funcionalidades RESTful, com a finalidade de melhorar a interoperabilidade em sistemas industriais, oferecendo uma abordagem simplificada para a comunicação entre máquinas e sistemas baseados em tecnologias web. As modificações propostas envolveram: 1) a ampliação do [OPC UA](#) para permitir requisições de serviço sem estado (*stateless*); 2) Implementação de um mecanismo de controle de cache; e 3) o estudo do transporte opcional de mensagens [OPC UA](#) via o Protocolo [User Datagram Protocol \(UDP\)](#). As extensões propostas foram implementadas na pilha de comunicação do próprio [OPC UA](#) de código aberto chamada *open62541*<sup>1</sup> e testadas em dois sistemas embarcados. As melhorias, no entanto, implicaram em restrições nos casos de uso do [OPC UA](#), como a diminuição do número de serviços disponíveis e do comprimento máximo das mensagens.

No trabalho apresentado em ([Schiekofer; Scholz; Weyrich, 2018](#)), o autor também propõe uma arquitetura RESTful para o [OPC UA](#). No entanto, algumas das limitações dos estudos anteriores são abordadas. Essa nova abordagem foca em resolver o problema relacionado ao *Namespace* e *ServerArray*, que anteriormente impunha restrições aos servidores que adotavam a extensão RESTful. O conceito introduzido permite que qualquer servidor [OPC UA](#) utilize a arquitetura RESTful sem a necessidade de restrições adicionais e ainda incorpora o princípio [Hypermedia as the Engine of Application State \(HATEOAS\)](#), proporcionando uma navegação mais completa e alinhada ao padrão arquitetural [Representational State Transfer \(REST\)](#).

Embora esses trabalhos não utilizem explicitamente o termo “*Digital Twin*” em suas abordagens, eles estão diretamente relacionados à temática desta Tese, pois exploram

---

<sup>1</sup>Disponível em <https://www.open62541.org/>

protocolos e arquiteturas comumente empregados em soluções de sistemas [DT](#). O foco principal dos estudos foi a extensão e modificação da pilha do protocolo [OPC UA](#) para implementar funcionalidades da arquitetura RESTful. No entanto, essa abordagem limita as soluções desenvolvidas à utilização exclusiva da pilha de protocolos [OPC UA](#). Além disso, esses trabalhos não apresentaram a aplicação prática em um estudo de caso específico, o que restringe a avaliação de seu impacto em cenários reais.

Em ([Cavaliere; Salafia; Scroppo, 2019](#)), os autores introduzem uma plataforma denominada [OPC UA Web Platform](#), baseada em uma arquitetura [REST](#), projetada para oferecer acesso a servidores [OPC UA](#). Diferentes das abordagens apresentadas, onde os autores modificaram a pilha de protocolo [OPC UA](#), esse trabalho implementa uma interface RESTful para [OPC UA](#). Nesse sentido, a plataforma permite que aplicações comuns, que não precisam ser compatíveis com a especificação [OPC UA](#) nem implementar sua pilha de comunicação, acessem o conjunto completo de informações de um servidor [OPC UA](#) de forma simplificada. O código da plataforma foi implementado e disponibilizado no GitHub<sup>2</sup>. Esse trabalho se assemelha à proposta desta Tese ao apresentar uma interface que abstrai a complexidade de implementação de protocolos de comunicação. Essa abordagem permite que os participantes do projeto acessem e manipulem informações sem a necessidade de um conhecimento especializado sobre protocolo, o que torna o processo de desenvolvimento mais simples.

Ainda na sob a ótica de abstração da complexidade de implementação dos protocolos, o trabalho apresentado por ([Hietala et al., 2020a,b](#)) propõe uma [Application Programming Interface \(API\)](#), com a finalidade de simplificar a transferência de dados com servidores [OPC UA](#) e reduzir a curva de aprendizado para os desenvolvedores. A [API](#) foi desenvolvida utilizando o [Graph Query Language \(GraphQL\)](#)<sup>3</sup>, o que tornou a solução acessível a qualquer dispositivo compatível com a web, tornando os dados disponíveis para uma ampla gama de linguagens de programação e plataformas. O estudo de caso do trabalho envolveu o uso da [API](#) em uma aplicação de controle de Ponte Rolante. Os resultados indicaram que a [API](#) foi suficientemente rápida para entradas do usuário e aplicações que exigem sincronização rápida de valores de diferentes fontes de dados. No entanto, a [API](#) introduziu uma latência considerável em comparação com o uso direto do servidor [OPC UA](#), o que representa um problema em aplicações sensíveis ao tempo. O estudo ([Tu et al., 2021](#)) utilizou o [GraphQL wrapper](#) proposto por Hietala como base para a comunicação entre o ambiente de realidade mista e o servidor [OPC UA](#). A solução integrou o módulo de

<sup>2</sup><https://github.com/OPCUAUniCT/OPCUAWebPlatformUniCT>

<sup>3</sup><https://graphql.org/>

controle e visualização do Gêmeo Digital ao sistema físico da Ponte Rolante<sup>4</sup>, permitindo a leitura e escrita de variáveis de processo em tempo real a partir de dispositivos de realidade mista (HoloLens). Os experimentos indicaram que a interface obteve precisão adequada, com erros inferiores a 10 cm, validando a viabilidade da abordagem. Contudo, o trabalho também evidenciou limitações relacionadas à latência de comunicação e à estabilidade do rastreamento espacial do HoloLens.

(Ala-Laurinaho *et al.*, 2022) realizou uma análise comparativa entre as abordagens **REST** e **GraphQL** aplicadas à comunicação com servidores **OPC UA**, avaliando métricas de desempenho, latência e consumo de recursos. Os experimentos demonstraram que o **GraphQL** superou o **REST** em operações envolvendo múltiplas leituras ou escritas, devido à sua capacidade de agrupar requisições em uma única consulta, reduzindo o número de chamadas à **API**. No entanto, para operações unitárias, o **REST** apresentou desempenho ligeiramente superior. Em todos os cenários, o acesso direto ao servidor **OPC UA** permaneceu como a opção mais eficiente em termos de latência. Os resultados reforçaram que a escolha entre **REST** e **GraphQL** depende do equilíbrio entre desempenho e flexibilidade, sendo o **GraphQL** mais adequado para aplicações com alto volume de dados e o **REST** preferível em contextos sensíveis ao tempo.

Diversos estudos têm se concentrado na avanço da modularidade e no *design* agnóstico a protocolos em arquiteturas de middleware. A literatura recente aponta para a necessidade de middlewares capazes de lidar com a crescente heterogeneidade dos ambientes industriais, tanto no que se refere a protocolos de comunicação quanto à diversidade de dispositivos conectados. Nesse contexto, surgem propostas de middlewares modulares e multiprotocolo, concebidos para oferecer maior flexibilidade na integração de Sistemas Ciberfísicos e Digital Twins.

Em (André; Azzi; Cardin, 2020), os autores propõem um middleware de comunicação heterogêneo para sistemas de manufatura ciberfísicos baseados em Digital Twin. O trabalho aborda a separação das responsabilidades de comunicação dos demais aspectos da aplicação, visando melhorar a evolução do sistema e permitir sua adaptação a diferentes contextos industriais. A solução proposta pelos autores é uma abordagem multiprotocolo que simplifica a interoperabilidade entre diferentes sistemas e dispositivos. A pesquisa destaca a importância da modularidade e da “reconfigurabilidade” para garantir flexibilidade e eficiência na comunicação. Os resultados indicam que a implementação do middleware contribui para a modernização da manufatura, tornando os sistemas mais

---

<sup>4</sup>Ponte rolante é uma máquina de elevação do tipo guindaste de ponte, utilizada para movimentação de cargas em ambientes industriais.

adaptáveis e escaláveis. Esse trabalho está alinhado com a proposta desta Tese por fornecer um abordagem de modularização dos protocolos de comunicação e por se tratar de um sistema multiprotocolo. Entretanto, a presente Tese avança no estado da arte ao propor não apenas uma arquitetura, mas também um conjunto de operações (leitura, escrita, inscrição e cancelamento de inscrição) e decisões arquiteturais voltadas à eficiência, como o uso de cache de inicialização e mecanismos de baixa latência. Além disso, a solução foi formalizada, implementada e validada em um estudo de caso realista, o que consolida a contribuição prática e amplia a aplicabilidade para além do domínio manufatura.

Em (Künzel *et al.*, 2021) propõe o desenvolvimento de um middleware modular e flexível para automação industrial, utilizando LabVIEW (Travis; Kring, 2006) e OPC UA, com a finalidade de prover comunicação entre sistemas e dispositivos de diferentes fabricantes. De acordo com a pesquisa conduzida, o padrão arquitetural mais indicado para o desenvolvimento foi o SOA. Outro ponto importante do trabalho está na descrição dos requisitos do middleware, tais como manutenibilidade, reuso, flexibilidade e confiabilidade. A conclusão do trabalho aponta que a combinação dessas tecnologias resulta em uma plataforma eficiente e adaptável, capaz de melhorar a flexibilidade, reduzir custos e facilitar a modernização dos processos de automação, além de ser uma solução viável para melhorar a comunicação e a manutenção de sistemas industriais.

No trabalho de (Hajnoruzi, 2023) é proposto uma arquitetura baseada em microserviço para Digital Twins, com a finalidade de melhorar a escalabilidade e a confiabilidade desses sistemas. O estudo define sete microserviços principais, com foco em cinco essenciais: aquisição de dados, gerenciamento do DT, simulação, comando e controle, e visualização. A solução é validada com um caso de uso de turbinas eólicas, validando a integração em tempo real, desafios e *trade-offs* do projeto e implementação. Por fim, o trabalho destaca a modularidade e a interoperabilidade proporcionadas pela abordagem de microserviço, essenciais para aplicações complexas de Digital Twins. De forma semelhante, (Rodríguez-Alonso; Pena-Regueiro; García, 2024) apresenta uma plataforma de DT para estações de tratamento de água, baseada em uma arquitetura de microserviços. Porém, a proposta do trabalho é direcionado para uma solução de Computação na Borda (*Edge Computing*). A plataforma visa melhorar a eficiência operacional e a manutenção preditiva, utilizando técnicas de aprendizado de máquina, modelagem de processos e simulação. A plataforma integra modelos *Building Information Modeling* (BIM) para apoiar a tomada de decisões.

A Tabela 1 sintetiza os trabalhos relacionados no contexto de arquitetura orientada a serviço.

| Trabalho         | Ano         | Paradigma          | Multiprotocolo | Baixa Latência |
|------------------|-------------|--------------------|----------------|----------------|
| Melik et al.     | 2012        | SOA                | Não            | Parcial        |
| Grüner et al.    | 2016        | REST               | Não            | Parcial        |
| Schiekofer       | 2018        | REST               | Não            | Não            |
| Cavalieri et al. | 2019        | REST               | Não            | Não            |
| Hietala et al.   | 2020        | GraphQL            | Não            | Não            |
| Ala et al.       | 2022        | REST/GraphQL       | Não            | Não            |
| André et al.     | 2020        | SOA                | Sim            | Não            |
| Künzel et al.    | 2021        | SOA                | Não            | Não            |
| Hajnoruzi et al. | 2023        | Microserviços      | Sim            | Parcial        |
| <b>Esta Tese</b> | <b>2026</b> | <b>Baixo nível</b> | <b>Sim</b>     | <b>Sim</b>     |

Tabela 1: Trabalhos relacionados que aplicam Arquitetura Orientada a Serviço.

### 3.2 Middleware modular para comunicação em IoT

Os avanços no desenvolvimento de middlewares para sistemas [IoT](#) evidenciam a importância dessa camada como elemento estruturante na integração de sistemas heterogêneos. Embora grande parte dessas propostas tenha como foco principal a interoperabilidade entre dispositivos [IoT](#), seus princípios arquiteturais — como modularidade, abstração da comunicação e suporte a múltiplos protocolos — são diretamente relevantes para sistemas de [DTs](#). Nesse contexto, esta subseção discute trabalhos de middleware para [IoT](#) como base conceitual e arquitetural, estabelecendo um paralelo com a proposta desta Tese.

O desenvolvimento de *middleware* para sistemas de [IoT](#) tem evoluído para dar suporte a ambientes complexos e heterogêneos, como *Smart Campuses* ([Nagowah; Sta; Gobin-Rahimbux, 2023](#)). Um dos principais desafios das redes de dispositivos de [IoT](#) é assegurar a interoperabilidade entre dispositivos que operam diversos protocolos ([Lamnaour et al., 2024](#)). O trabalho apresentado em ([Gutierrez; Villegas-Ch; Govea, 2025](#)) propõe um *middleware* modular voltado a ambientes [IoT](#), tendo como estudo de caso um *Smart Campus*. A arquitetura apresentada adota uma abordagem baseada em microserviços, com o objetivo de promover escalabilidade, interoperabilidade e eficiência energética em cenários compostos por múltiplos protocolos de comunicação e diferentes classes de dispositivos. O middleware proposto implementa mecanismos de tradução de protocolos, descoberta dinâmica de dispositivos e orquestração de serviços, permitindo a integração de

tecnologias como [Message Queuing Telemetry Transport \(MQTT\)](#), [CoAP](#), [HTTP](#) e [BLE](#) em um ambiente compartilhado. A validação experimental considera tanto a implantação real, com aproximadamente 120 dispositivos físicos, quanto cenários simulados que estendem a avaliação para até 500 nós, analisando métricas como latência, interoperabilidade e consumo de energia. A validação experimental em um ambiente controlado demonstrou uma redução de 26,7% no consumo de energia para os nós otimizados, alcançando uma média de 60 W, em comparação com 80 W para os nós não otimizados. Os tempos de resposta apresentaram média de 130 ms nos nós otimizados, superando o desempenho do LoRaWAN e alcançando uma taxa de sucesso de interoperabilidade de 94%.

Em ([SOARES, 2021](#)) é proposta uma arquitetura orientada a serviços (SOA) para apoiar a criação e a execução de ambientes experimentais voltados a sistemas ubíquos e aplicações IoT. Os autores utilizam o paradigma [Sensor as a Service \(SenaaS\)](#) para abstrair sensores e atuadores. Desse modo, esses recursos são disponibilizados como serviços por aplicações em nuvem. Dessa forma, o desenvolvedor pode acessar dados e comandos de dispositivos físicos sem lidar diretamente com detalhes de hardware, conectividade ou configuração dos dispositivos. O trabalho também incorpora conceitos de virtualização, containerização e automação de infraestrutura, com foco em escalabilidade, reprodutibilidade experimental e gerenciamento de recursos em ambientes distribuídos. Embora a arquitetura apresente características importantes, como desacoplamento e uso de modelos [Publish–Subscribe \(Pub/Sub\)](#), a solução permanece fortemente acoplada a tecnologias específicas da pilha de serviços web e não aborda de forma explícita requisitos de baixo overhead e avaliação sistemática de latência, aspectos críticos em aplicações industriais sensíveis ao desempenho. Além disso, o trabalho não explora a integração multiprotocolo nem a substituição transparente de mecanismos de comunicação.

O trabalho apresentado por ([Dave; Doshi; Arolkar, 2020](#)) propõe uma solução de interoperabilidade em nível de camada de aplicação para ambientes de Internet das Coisas, denominada [MQTT–CoAP Interconnector \(MCI\)](#). A abordagem consiste em um *middleware bridge* responsável pela tradução bidirecional de mensagens entre os protocolos [MQTT](#) e [Constrained Application Protocol \(CoAP\)](#), permitindo a comunicação entre dispositivos heterogêneos que adotam diferentes modelos de interação, como *publish/subscribe* e *request/response*. De forma análoga, ao trabalho apresentado em ([Cruz, 2021](#)) um middleware denominado *In.IoT*. O trabalho é focado em desafios como segurança, usabilidade e desempenho. O estudo também investiga o impacto da linguagem de programação no desempenho global do middleware, comparando implementações em Java, Python e JavaScript, e identifica o Java como a opção mais robusta. O *In.IoT* está alinhado a pro-

posta desta Tese ao oferecer um mecanismo de desacoplamento entre protocolos distintos, como [MQTT](#) e [HTTP](#). Porém, o trabalho ainda não apresenta mecanismos de baixa latência em cenários [DT](#) onde a aplicação é sensível ao tempo.

A dissertação de mestrado apresentada em ([Carbonel Martínez; Gallego Martínez; Tolosana Calasanz, 2020](#)) segue na investigação do problema da interoperabilidade entre dispositivos de [IoT](#) que utilizam protocolos de comunicação heterogêneos, como [MQTT](#), [MQTT for Sensor Networks \(MQTT-SN\)](#), [CoAP](#) e [HTTP](#). Os autores destacam que as dificuldades de integração não se limitam ao formato das mensagens, mas também aos diferentes paradigmas/modelos de comunicação adotados pelos protocolos, como *request/response* e *publish/subscribe*, o que impõe barreiras técnicas significativas, especialmente para pequenas e médias empresas. Nesse contexto, é proposto e implementado um middleware capaz de intermediar de forma transparente a comunicação entre dispositivos [IoT](#) baseados em diferentes protocolos. A validação experimental considera distintos cenários de execução, incluindo dispositivos com e sem restrições de recursos, avaliando o desempenho do middleware principalmente em termos de latência em ambientes distribuídos.

([Khansari; Sharifian, 2024](#)) propõe uma arquitetura baseada em microserviços no qual desacopla o gerenciamento dos dispositivos do gerenciamento do protocolo para proporcionar flexibilidade e modularidade de integração de novos dispositivos na rede [IoT](#). O problema central é a alocação eficiente em ambientes dinâmicos e heterogêneos, onde limitações de recursos e requisitos de qualidade de serviço são desafios significativos.

Além das propostas mais recentes, torna-se igualmente relevante posicionar o presente trabalho em relação a soluções tradicionais e consolidadas na literatura, que servem como referência conceitual e arquitetural para o desenvolvimento de middleware em sistemas [IoT](#). Nesse sentido, o [Global Sensor Network \(GSN\)](#)<sup>5</sup> ([Aberer; Hauswirth; Salehi, 2006](#)) é um middleware projetado com o objetivo de facilitar o desenvolvimento e a programação de redes de sensores. A plataforma foi concebida para operar como uma camada unificada de integração, permitindo a conexão e o gerenciamento de um grande número de sensores heterogêneos em uma única infraestrutura. Uma de suas principais características é a abstração dos sensores por meio do uso de descritores baseados em [Extensible Markup Language \(XML\)](#), aliados à utilização de consultas [Structured Query Language \(SQL\)](#) para a recuperação e o processamento das informações coletadas. No [GSN](#), cada sensor é encapsulado por um *wrapper*, responsável por definir os protocolos de comunicação, as configurações de rede, os procedimentos de aquisição dos dados e as ações a

---

<sup>5</sup><https://gsn.epfl.ch>

serem executadas sobre as informações obtidas. Adicionalmente, o middleware oferece mecanismos para publicação dos dados por meio de *Web Services*, possibilitando o acesso e a disseminação das informações geradas pelos sensores. O GSN também incorpora funcionalidades de processamento de fluxos de dados (*data stream processing*), permitindo a detecção de eventos complexos e a geração de contexto a partir das informações coletadas, por meio da execução de consultas SQL sobre os fluxos de dados em tempo real.

O ContextNet (Endler *et al.*, 2011) é um middleware destinado ao fornecimento de dados de contexto em ambientes compostos por redes estacionárias e móveis. A solução foi projetada para atender aplicações que demandam mobilidade e comunicação em grupo, apoiando-se em uma camada de distribuição de dados escalável, denominada Scalable Data Distribution Layer (SDDL). Essa camada permite que o sistema suporte o crescimento da rede e a disseminação eficiente de informações entre um grande número de dispositivos. Do ponto de vista arquitetural, o ContextNet adota o modelo de comunicação Pub/Sub. A transferência de dados é realizada por meio de dois protocolos distintos: o Mobile Reliable UDP (MR-UDP), responsável pela troca de mensagens entre clientes e *gateways*, e o Data Distribution Service (DDS), que atua no núcleo da rede, gerenciando a distribuição eficiente dos dados entre os nós participantes. Essa abordagem híbrida permite alcançar elevados níveis de escalabilidade, assegurando a disseminação de conteúdo em redes de grande porte e altamente dinâmicas.

O trabalho apresentado em (Ostermaier; Schlup; Römer, 2010) propõe o WebPlug, um framework voltado ao paradigma da *Web of Things*, cujo objetivo é integrar sensores e atuadores ao ambiente Web por meio de recursos acessíveis via interfaces REST. A abordagem modela dispositivos físicos como recursos Web, permitindo sua composição em aplicações distribuídas, denominadas *physical mash-ups*. O framework introduz mecanismos para versionamento de recursos, observação de mudanças de estado e manutenção de histórico, possibilitando o acompanhamento de estados passados de sensores e a geração de eventos a partir de alterações nos dados. Além disso, o WebPlug adota um modelo de comunicação orientado a recursos e baseado em *callbacks* via Uniform Resource Locators (URLs), favorecendo a construção modular de aplicações e a integração com serviços existentes na Web. A proposta é validada por meio de um protótipo funcional e de um estudo de caso em um cenário real, evidenciando o potencial do framework para a composição distribuída de aplicações em ambientes heterogêneos. Outras propostas semelhantes exploram a abordagem de Web Service para integração de dispositivos como o middleware e Tiny Web Service (Priyantha *et al.*, 2008) e SOCRADES (De Souza *et al.*, 2008).

A Tabela 2 apresenta o resumo dos trabalhos apresentados nesta subseção.

| Trabalho          | Ano         | Mecanismo de Co-<br>municação   | Estilo de Comu-<br>nicação                         |
|-------------------|-------------|---------------------------------|----------------------------------------------------|
| GSN               | 2006        | Web Services (HTTP)             | Request/Reply                                      |
| Tiny Web Services | 2008        | Web Services (HTTP)             | Request/Reply                                      |
| SOCRADES          | 2008        | Web Services (HTTP)             | Request/Reply                                      |
| WebPlug           | 2010        | REST / Web of Things            | Request/Reply                                      |
| ContextNet        | 2011        | DDS + MR-UDP                    | Publish/Subscribe                                  |
| Soares et al.     | 2021        | Web Services (HTTP)             | Publish/Subscribe                                  |
| Dave et al. (MCI) | 2020        | Bridge/Gateway de<br>protocolos | Publish/Subscribe<br>+ Request/Reply               |
| Carbonell         | 2020        | MQTT, CoAP e HTTP               | Publish/Subscribe<br>+ Request/Reply               |
| Da Cruz (In.IoT)  | 2021        | Web Services (HTTP)             | Request/Reply                                      |
| Khansari et al.   | 2024        | Web Services (HTTP)             | Publish/Subscribe                                  |
| Gutierrez et al.  | 2025        | Web Services (HTTP)             | Publish/Subscribe<br>+ Request/Reply               |
| <b>Esta Tese</b>  | <b>2026</b> | <b>Sockets<br/>(TCP/UDP)</b>    | <b>Request/Reply<br/>e Publish/Subs-<br/>cribe</b> |

Tabela 2: Comparação entre trabalhos relacionados quanto aos mecanismos e estilos de comunicação em middleware para IoT

3.3 Framework para desenvolvimento de simuladores

Embora esta Tese tenha como foco os sistemas de Digital Twin, há uma interseção significativa com o desenvolvimento de simuladores, especialmente os que combinam componentes de hardware e software, e hápticos. Esses simuladores desempenham um papel essencial no treinamento de pessoal, modelagem e validação de sistemas ciberfísicos, possibilitando a interação dinâmica e em tempo real entre componentes físicos e virtuais. Assim, nesta subseção, são apresentadas as propostas de framework que serviram como referência para o desenvolvimento dos módulos da arquitetura do middleware proposto neste trabalho.

Os trabalhos de ([Abdelgawad; Abdelkarim et al., 2014](#); [Abdelgawad; Hassan et al., 2015](#); [Abdelgawad; Gausemeier; Stöcklein et al., 2017](#); [Abdelgawad; Gausemeier; Trächler et al., 2017](#)) apresentam avanços nas metodologias e abordagens da simulação e treinamento em Sistemas Avançados de Assistência ao Motorista [Advanced Driver-Assistance System \(ADAS\)](#). Esses trabalhos apresentam soluções arquiteturais que enfatizam a escalabilidade e modularidade dos sistemas. ([Abdelgawad; Abdelkarim et al., 2014](#)) introduz uma arquitetura escalável para protótipos virtuais de [ADAS](#) implementados em *MATLAB/Simulink*. A arquitetura proposta descreve os módulos funcionais, as relações de entrada e saída dos modelos, e a integração da estrutura em um simulador de direção para demonstração e treinamento de [ADAS](#). Esse estudo é expandido em ([Abdelgawad; Hassan et al., 2015](#)), que detalha a arquitetura modular interativa, com foco na assistência de frenagem e direção de emergência. Em sequência, ([Abdelgawad; Gausemeier; Stöcklein et al., 2017](#)) propõem uma plataforma de treinamento com [Head-Mounted Display \(HMD\)](#)s, com a finalidade de prover simulações imersivas e monitoramento de condutores em rede, aplicada a escolas de condução modernas. Finalmente, ([Abdelgawad; Gausemeier; Trächler et al., 2017](#)) os autores apresentam um método de projeto orientado à aplicação para simulação de condução em rede.

Ainda no contexto de [ADAS](#), o trabalho ([Ataman; Biberici; Celik, 2024](#)) explora a simulação de [ADAS](#) em um modelo dinâmico de veículo. Embora a proposta do trabalho não seja focada na construção de framework ou arquitetura específica, o estudo apresenta uma arquitetura do sistema de modelagem e analisa o impacto das funções do [ADAS](#) no consumo de energia durante o ciclo de condução em um ambiente virtual. O projeto utiliza um modelo dinâmico em *MATLAB/Simulink*. Além disso, destaca a fusão de sensores para melhorar a detecção de obstáculos e avalia o impacto de sistemas como Frenagem Autônoma de Emergência ( [Autonomous Emergency Braking \(AEB\)](#)) e Controle de Cruzeiro Adaptativo ( [Adaptive Cruise Control \(ACC\)](#)) no desempenho energético de veículos elétricos.

([Santos; Maria et al., 2024](#)) propõem um framework para o desenvolvimento de simuladores híbridos, aplicado na modernização do simulador de submarinos da Classe Tupi, com a integração dos sistemas físicos e virtuais. O framework é composto por dois sistemas principais: a [Estação de Operação do Instrutor \(EOI\)](#) e o [Módulo de Simulação \(MS\)](#), com uma arquitetura modular baseada em camadas, que facilita a manutenção e a comunicação eficiente entre os sistemas. O [MS](#) suporta dinâmica da simulação, modelagem matemática e comunicação com o sistema de automação, permitindo a verificação e validação do tempo de simulação e resposta. Em termos de implementado, o

(EOI) foi implementado em C# e a (MS) em C++, o que validou a sua modularidade. Além disso, o projeto foi validado com profissionais experientes em termos de navegação, comprovando que a arquitetura proposta e sua respectiva implementação atenderam os requisitos funcionais e não-funcionais do [Simulador de Treinamento de Imersão \(TI\)](#). Embora esse trabalho seja voltado para procedimentos de simulação, o aspecto do projeto está intimamente relacionado com as definições de [DT](#) em termos de comunicação e interação entre dispositivos físicos (atuadores e sensores), sistema de [Open Platform Communications \(OPC\)](#) e sistema virtual.

A Tabela 3 sintetiza os trabalhos relacionados no contexto de simulação e treinamento comparando-os sob a ótica da arquitetura proposta, do nível de modularidade e da integração entre componentes físicos e virtuais.

| Trabalho          | Ano         | Arq. Modular | Des. da Comunicação |
|-------------------|-------------|--------------|---------------------|
| Abdelgawad et al. | 2014        | Sim          | Não                 |
| Abdelgawad et al. | 2015        | Sim          | Não                 |
| Abdelgawad et al. | 2017        | Sim          | Não                 |
| Abdelgawad et al. | 2017        | Parcial      | Não                 |
| Ataman et al.     | 2024        | Parcial      | Não                 |
| Dos Santos et al. | 2024        | Sim          | Parcial             |
| <b>Esta tese</b>  | <b>2026</b> | <b>Sim</b>   | <b>Sim</b>          |

Tabela 3: Comparação dos trabalhos relacionados quanto à modularidade arquitetural e desacoplamento da comunicação.

## 3.4 Considerações Finais

Este capítulo apresentou uma revisão dos principais trabalhos relacionados a esta tese, organizados em três eixos: (i) arquiteturas orientadas a serviço aplicadas à interoperabilidade industrial, com ênfase no [OPC UA](#) e em extensões RESTful; (ii) middlewares para [IoT](#), incluindo propostas recentes voltadas à modularidade e multiprotocolo, bem como soluções tradicionais consolidadas; e (iii) frameworks voltados ao desenvolvimento de simuladores, relevantes para cenários em que componentes físicos e virtuais precisam operar de forma integrada.

No contexto de Arquiteturas Orientadas a Serviço, observou-se que as abordagens analisadas contribuem para reduzir a complexidade de integração por meio da padronização de interfaces e do uso de tecnologias amplamente difundidas no ecossistema Web. Entretanto, tais soluções tendem a permanecer acopladas ao [OPC UA](#) ou a pilhas específicas de serviços, além de introduzirem sobrecarga inerente a camadas adicionais de abstração,

o que pode limitar sua adequação em aplicações sensíveis ao tempo. Esse cenário reforça a importância de investigar alternativas arquiteturais que preservem interoperabilidade, porém com menor *overhead* e maior flexibilidade na adoção de diferentes mecanismos de comunicação.

Em relação aos middlewares para sistemas **IoT**, os trabalhos recentes evidenciam avanços significativos em termos de modularidade, escalabilidade, segurança e interoperabilidade multiprotocolo, especialmente por meio de arquiteturas baseadas em microserviços. Ainda assim, parte relevante dessas propostas é orientada predominantemente a tecnologias baseadas em **HTTP** e serviços web, ou foca na integração funcional de dispositivos, sem tratar de forma sistemática a comunicação como uma preocupação arquitetural central para cenários com requisitos rigorosos de latência. Adicionalmente, as soluções clássicas discutidas fornecem fundamentos importantes sobre abstração de sensores, processamento de fluxos e distribuição escalável de dados, que nortearam a estruturação e proposta da arquitetura do middleware.

Por fim, os frameworks voltados ao desenvolvimento de simuladores reforçam a relevância de arquiteturas modulares e do desacoplamento entre componentes físicos e virtuais, especialmente em aplicações em que a sincronização temporal e a integração entre subsistemas são requisitos críticos. Esses trabalhos contribuem para contextualizar a necessidade de soluções arquiteturais capazes de organizar a comunicação e a integração entre módulos de forma consistente, reusável e evolutiva.

Em síntese, a literatura analisada evidencia lacunas relacionadas (i) ao suporte flexível e extensível a múltiplos protocolos, (ii) ao desacoplamento explícito entre mecanismos de comunicação e regras de negócio, e (iii) ao atendimento de requisitos de baixa latência em arquiteturas voltadas a **DTs**. Essas lacunas motivam e fundamentam a proposta desta Tese, que é detalhada nos capítulos subsequentes.

## 4 Arquitetura Proposta do Middleware

Neste capítulo é apresentada a arquitetura do *middleware* proposta para viabilizar a integração eficiente entre o [Physical Twin \(PT\)](#) e o [Digital Twin \(DT\)](#). A proposta tem por objetivo preencher as lacunas identificadas no Capítulo 3. Entre as limitações observadas, destacam-se a ausência de soluções efetivamente *protocol-agnostic*, a elevada complexidade de configuração e a dependência de plataformas em nuvem e tecnologias Web, que introduzem sobrecarga adicional na transmissão de informações, maior consumo de largura de banda e riscos associados à latência. Somam-se ainda as restrições de segurança e *compliance*, que impõem a execução de sistemas [DT](#) diretamente na borda, assegurando que o processamento ocorra dentro da rede local, com maior previsibilidade de desempenho, redução do tráfego externo e preservação da confidencialidade dos dados.

A Seção 4.1 apresenta os requisitos e capacidades associados aos sistemas dos [DTs](#). Complementarmente, a Seção 4.2 formaliza os requisitos que orientaram a elaboração da arquitetura (modularidade, extensibilidade, agnosticismo a protocolos, eficiência com baixo *overhead*, separação de responsabilidades e portabilidade). A Seção 4.3 apresenta os requisitos funcionais para implementação do middleware. A Seção 4.4 apresenta a visão geral da arquitetura proposta. A Seção 4.5 descreve a implementação da Prova de Conceito ([Proof of Concept \(PoC\)](#)). Por fim, a Seção 4.6 sintetiza as contribuições do capítulo e posiciona a solução quanto à sua aplicabilidade em cenários industriais críticos e à maturidade tecnológica.

### 4.1 Requisitos para Sistemas de Gêmeos Digitais

O [Digital Twin Consortium \(DTC\)](#) é uma iniciativa colaborativa que reúne representantes da indústria, academia e governo com o objetivo de fomentar o desenvolvimento, a adoção e a consolidação de tecnologias relacionadas aos Gêmeos Digitais. A organização integra o [Object Management Group \(OMG\)](#) e conta com a participação de empresas e

instituições de destaque no ecossistema de transformação digital, incluindo organizações como Microsoft, Dell Technologies e Dassault Systèmes (Object Management Group, Inc., 2024). Além de contribuir para a consolidação conceitual dos DTs, o DTC disponibiliza diretrizes, modelos de referência e estruturas para apoiar o projeto e o desenvolvimento desses sistemas. Entre essas iniciativas, destaca-se a Digital Twin Capabilities Periodic Table (CPT). O CPT é um framework agnóstico à tecnologia, desenvolvido para apoiar a definição de requisitos de sistemas de Gêmeos Digitais a partir dos requisitos necessários a cada caso de uso. Nesse sentido, o CPT propõe seis categorias para classificar as capacidades de um DT. Essas categorias abarcam 62 capacidades recomendadas pelo DTC (Object Management Group, Inc., 2024). A seguir são descritos as categorias (Schalkwyk, 2022):

- **Data Services:** Essa categoria abarca as capacidades relacionadas ao acesso, aquisição, ingestão, processamento, gerenciamento e armazenamento dos dados utilizados pelos sistemas de DT. De acordo com o DTC, essa categoria estabelece a conexão entre os domínios físico e virtual, permitindo que informações provenientes de sensores, dispositivos e sistemas de controle sejam adquiridas, processadas e posteriormente disponibilizadas aos demais componentes do sistema. Dessa forma, os serviços de dados constituem um elemento fundamental para que a representação digital possa ser continuamente alimentada com informações provenientes de sua contraparte física.
- **Integration:** contempla as capacidades necessárias para integrar o Gêmeo Digital aos diferentes sistemas, aplicações e plataformas que compõem seu ecossistema (conforme ilustrado na Figura 8 do Capítulo 2). Essa categoria inclui a integração com sistemas corporativos, como Enterprise Resource Planning (ERP), Enterprise Asset Management (EAM), Customer Relationship Management (CRM) e Computerized Maintenance Management System (CMMS); sistemas de engenharia, como Computer-Aided Design (CAD), Computer-Aided Manufacturing (CAM), Building Information Modeling (BIM); e sistemas operacionais e dispositivos de Internet of Things (IoT)/Operational Technology (OT), como sensores, sistemas de controle e Supervisory Control and Data Acquisition (SCADA). Essa categoria também abrange a integração entre diferentes instâncias de DTs, permitindo a troca de informações entre aplicações de DT, bem como a disponibilização de dados e serviços por meio de Application Programming Interfaces (APIs).
- **Intelligence:** reúne as capacidades relacionadas ao desenvolvimento e à execução dos processos que conferem comportamento analítico e inteligente aos Gêmeos Di-

gitais. Inclui integração e análise de dados, inteligência artificial, aprendizado de máquina, predição, simulação, orquestração, regras de negócio e mecanismos de comando e controle.

- **User Experience (UX):** reúne as capacidades relacionadas à visualização, interpretação e interação dos usuários com os DTs. Essa categoria contempla mecanismos de visualização básica e avançada, incluindo gráficos, tabelas, *dashboards*, modelos tridimensionais e animações, além de recursos para monitoramento de informações continuamente atualizadas em tempo real. Além disso, abrange a visualização das relações entre entidades que compõem o sistema, permitindo a navegação em estruturas hierárquicas ou baseadas em grafos. Adicionalmente, são consideradas formas de interação imersiva, como realidade aumentada e realidade virtual, bem como mecanismos de inteligência contínua, visualização por motores gráficos e gerenciamento de fluxos de processos. De forma geral, essas capacidades têm como finalidade facilitar a compreensão do estado e do comportamento do DT e apoiar a interação dos usuários com a respectiva contraparte física. A Figura 17 ilustra uma aplicação de representação tridimensional utilizada em um sistema de Gêmeo Digital voltado à simulação de cirurgia remota.

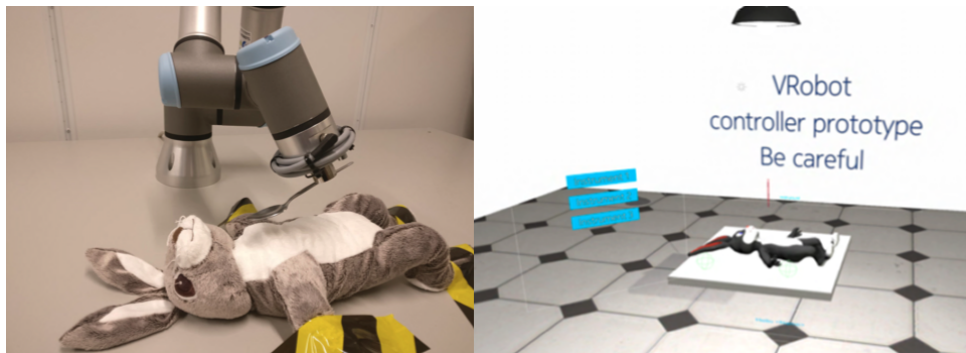


Figura 17: DT aplicado à simulação de cirurgia remota com braço robótico. (Laaki; Miche; Tammi, 2019).

- **Management:** abrange as capacidades necessárias ao gerenciamento e monitoramento dos sistemas e do ecossistema de Gêmeos Digitais. Inclui o gerenciamento de dispositivos, o monitoramento de sistemas e aplicações, o registro de eventos (*logging*) e a governança dos dados utilizados ao longo do ciclo de vida do sistema.
- **Trustworthiness:** reúne as capacidades relacionadas à segurança, privacidade, segurança operacional, confiabilidade e resiliência. Essa categoria contempla mecanismos de criptografia e proteção dos dados, autenticação e autorização de dispositivos, controle de acesso e preservação da confidencialidade, integridade e disponibilidade

das informações. Também abrange aspectos relacionados à operação segura do sistema, à manutenção de níveis adequados de serviço e à capacidade de recuperação diante de falhas, interrupções ou outras condições adversas.

Além das capacidades apresentadas pelo DTC, a norma ISO 23247-1:2021 ([International Organization for Standardization, 2021](#)) estabelece requisitos para o desenvolvimento de sistemas de Gêmeos Digitais aplicados ao contexto de manufatura. Entre eles, destacam-se a aquisição de dados, identificação das entidades, comunicação, sincronização, gerenciamento de recursos, análise de dados, simulação, suporte ao ciclo de vida e modelagem hierárquica. Complementarmente, ([Bellavista et al., 2024](#)) apresentam um conjunto de requisitos voltados a ecossistemas de DT. Os autores destacam seis requisitos principais: mobilidade entre a borda e a nuvem (*edge-to-cloud mobility*), resiliência a cargas variáveis (*variable load resilience*), consciência do entrelaçamento ciberfísico (*entanglement awareness*), gerenciamento do ciclo de vida (*life cycle*), descrição declarativa da aplicação (*declarative application description*) e responsabilização e rastreabilidade operacional (*accountability*). Dentre esses requisitos, a consciência do entrelaçamento ciberfísico apresenta relação direta com o problema de comunicação investigado nesta Tese, uma vez que a manutenção da correspondência entre os estados físico e digital depende da obtenção e transmissão contínua das informações entre o PT e o DT.

## 4.2 Requisitos Arquiteturais do Middleware

Esta Seção apresenta os requisitos arquiteturais adotados, complementarmente aos requisitos do domínio, para nortear o desenvolvimento do middleware proposto. Segundo Bass, Clements e Kazman ([Bass; Clements; Kazman, 2021](#)), a definição explícita de requisitos arquiteturais é essencial para garantir que os atributos de qualidade — como modularidade, extensibilidade, portabilidade e eficiência — sejam sistematicamente incorporados desde a concepção da arquitetura. Além disso, normas como a ISO/IEC/IEEE 42010:2011 ([ISO/IEC/IEEE. . . , 2011](#)) reforçam a importância da formalização como forma de alinhar arquitetura e requisitos do projeto. Nesse sentido, de acordo com os componentes arquiteturais apresentados na Seção 4.4, torna-se necessária a formalização dos requisitos que nortearam a concepção da arquitetura. Esses requisitos funcionam como princípios fundamentais que orientam as decisões de projeto e asseguram que a arquitetura proposta atenda não apenas às necessidades funcionais do *middleware*, mas também a atributos de qualidade reconhecidos na literatura de Engenharia e Arquitetura de Software.

Nesta Seção, cada requisito é discutido, incluindo sua fundamentação teórica, relação com os componentes da arquitetura proposta e a definição de cenários de qualidade que permitem avaliar sua efetiva implementação. Portanto, a seguir, cada requisito é descrito individualmente, relacionando-o aos componentes do *middleware* que materializam sua implementação.

- **Modularidade:** a arquitetura deve permitir a substituição de componentes sem impactar outros módulos do sistema. Esse requisito é atendido principalmente pelo **Orchestrator**, que atua como coordenador central, garantindo que módulos especializados — como **Protocol Bridge** e **Protocol Service** — possam ser substituídos ou atualizados de forma transparente. A modularidade é um aspecto essencial para manutenção evolutiva e suporte a novos casos de uso
- **Extensibilidade:** é entendida como a capacidade do *middleware* de suportar a inclusão de novas funcionalidades e comportamentos e, principalmente, protocolos. A separação entre interpretação semântica e execução concreta das operações contribui diretamente para a extensibilidade do protocolo. Enquanto o *Protocol Service* pode ser estendido para reconhecer novos comandos ou variações semânticas, o *Protocol Bridge* pode ser ampliado para suportar novos protocolos com o **PT**, mantendo uma interface uniforme para o restante do *middleware*. Dessa forma, a evolução do protocolo ocorre de maneira controlada e localizada, evitando impactos em cascata na arquitetura.
- **Agnosticismo a Protocolos:** um dos pilares da arquitetura proposta neste trabalho é o design agnóstico a protocolos. Esse é um princípio estratégico para garantir flexibilidade, modularidade e interoperabilidade em ambientes industriais complexos, principalmente em sistemas **DT**. Conforme especificado anteriormente, o princípio é materializado principalmente por meio do componente **Protocol Bridge**. Este componente funciona como uma fachada que expõe uma interface de comunicação genérica para os demais componentes do Middleware, principalmente para o **Orchestrator**, enquanto encapsula internamente a complexidade de interagir com protocolos específicos. Para cada protocolo suportado — como **Open Platform Communications Unified Architecture (OPC UA)**, **Message Queuing Telemetry Transport (MQTT)**, **Serial** ou **Modbus** — uma implementação concreta (ou *adaptador*) é desenvolvida, conforme ilustrado na Figura 18. Essa implementação é responsável por traduzir as requisições genéricas do Middleware para os comandos específicos do protocolo do *Physical Twin*, e vice-versa. Portanto, a arquitetura estabelece um

desacoplamento bem definido entre a ação (ex: "ler a temperatura do motor") e o mecanismo (ex: "realizar uma chamada OPC UA ao nó  $ns=2;s=Motor.Temp$ "). A Figura 18 expande o componente **Protocol Bridge**.

- **Separação de Responsabilidades:** a lógica de negócio do **DT** deve estar desacoplada dos mecanismos de comunicação implementados pelo *Middleware*. Esse requisito é assegurado pela própria estrutura em três camadas principais: **Digital Twin**, **Middleware** e **Physical Twin**. Assim, alterações na lógica do **DT** não impactam a camada de comunicação, preservando a independência e clareza de responsabilidades.
- **Eficiência (Baixo Overhead):** a arquitetura deve introduzir sobrecarga mínima em tempo de execução, garantindo comunicação eficiente em tempo real. Esse requisito se materializa na **InterfaceIO**, que prioriza camadas de transporte leves (como **Transmission Control Protocol (TCP)/User Datagram Protocol (UDP)**) para reduzir latência, e no uso de **cache interno** para otimizar operações de leitura e escrita, evitando inicializações redundantes.

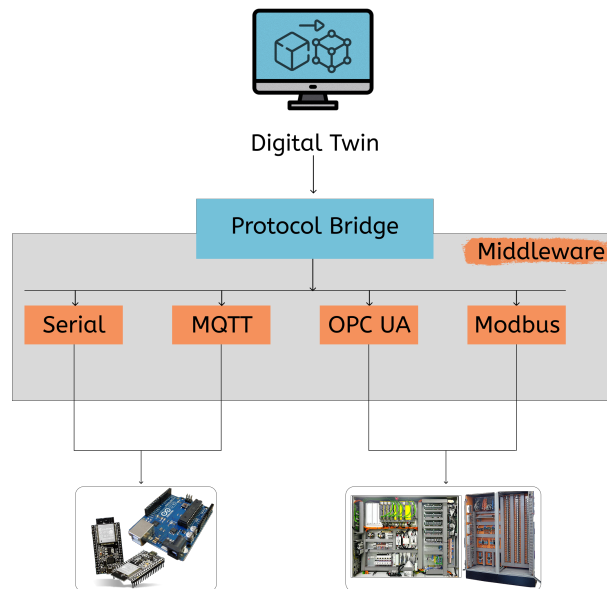


Figura 18: Arquitetura agnóstica a protocolo.

## 4.3 Requisitos Funcionais do Middleware

Embora existam capacidades e requisitos recorrentes em sistemas de Gêmeos Digitais, os requisitos funcionais podem variar significativamente de acordo com o domínio de

aplicação, do contexto do sistema e dos objetivos estabelecidos para o DT. Assim, a partir dos requisitos e capacidades discutidos anteriormente, este trabalho realiza um recorte direcionado ao problema de comunicação entre o DT e sua contraparte física.

Para o desenvolvimento da PoC proposta nesse trabalho foram consideradas as funcionalidades fundamentais que o *Middleware* deve prover para suportar a comunicação e interação entre o DT e PT. A Figura 53 ilustra o diagrama de caso de uso, com as funcionalidades mapeadas.

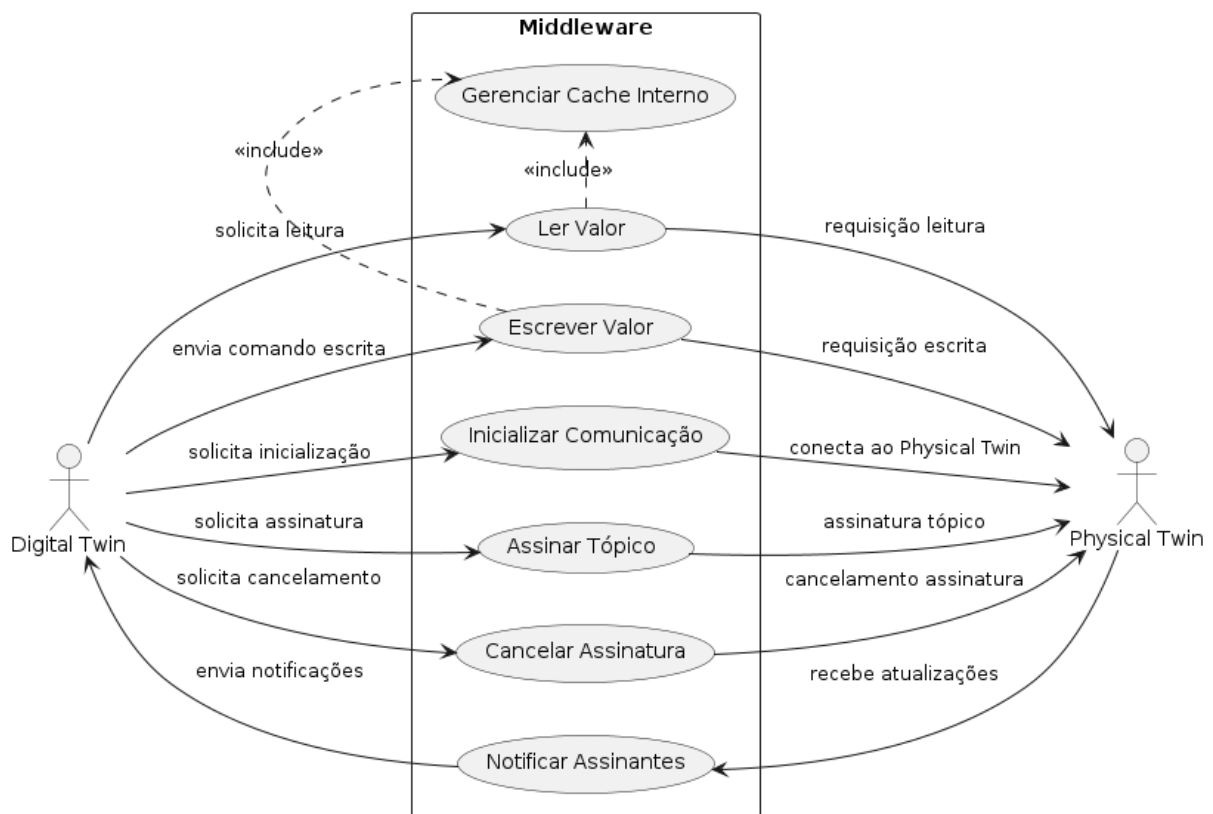


Figura 19: Diagrama de casos de uso das funcionalidades do Middleware.

Conforme ilustrado na Figura 53, o caso de uso **Inicializar comunicação** representa o processo pelo qual o DT requisita o estabelecimento de conexão com PT por meio do *Middleware*. Antes da execução desse caso de uso, é necessário que o *Middleware* esteja configurado e em operação. Com isso, o DT inicia a interação solicitando a inicialização da comunicação. Em seguida, o *Middleware* valida a requisição e as configurações, e estabelece a conexão com PT. Uma vez estabelecida a conexão, o *Middleware* confirma ao DT que a comunicação foi iniciada com sucesso, permitindo que o fluxo de troca de mensagens seja realizada. Caso ocorra algum problema na conexão ou recebimento da solicitação, o *Middleware* notifica o DT, de modo a permitir que o sistema possa tratar o erro de forma adequada.

Após o estabelecimento da comunicação, é possível realizar as operações de leitura e escrita de valores no sistema do **PT**. Essas operações podem ser síncronas ou assíncronas. Nesse sentido, o caso de uso **Ler Valor** representa o processo de solicitação de leitura síncrona. Como o *Middleware* é um sistema externo, ele não mantém informações persistentes sobre o estado das variáveis instanciadas no espaço de endereçamento do **PT**. Em alguns protocolos, como o **OPC UA**, é necessário inicializar a variável de leitura antes de realizar operações de leitura ou escrita, o que aumenta o custo computacional. Para evitar esse processamento de inicialização, o *Middleware* implementa um mecanismo de cache interno de variáveis, representado pelo caso de uso **Gerenciar Cache Interno**. Dessa forma, sempre que ocorre uma operação síncrona de leitura ou escrita, o *Middleware* verifica primeiro o cache interno em busca da variável correspondente, o que evita inicializações repetidas e, consequentemente, reduz o custo computacional. Após os devidos tratamentos, o *Middleware* encaminha a solicitação ao **PT** e, uma vez recebida a resposta, retorna o valor ao **DT**. Caso ocorra alguma falha na leitura ou na comunicação, o *Middleware* notifica o **DT**, permitindo o tratamento adequado do erro. O mesmo procedimento ocorre para a operação de escrita, representada pelo caso de uso **Escrever Valor**.

O *Middleware* prevê um mecanismo para operações assíncronas. Para isso é utilizado o padrão de comunicação *Publish/Subscriber* (Tanenbaum; Wetherall *et al.*, 2019). O caso de uso **Assinar Tópico** representa a funcionalidade de inscrição, do **DT**, aos tópicos específicos de interesse para receber atualizações do **PT**. Deste modo, o **DT** solicita a assinatura, o *Middleware* registra a inscrição e estabelece o fluxo de notificações para o **DT**. Caso haja falha na assinatura, o *Middleware* informa o **DT**. De modo semelhante, o caso de uso **Cancelar Assinatura** descreve a ação de remover um tópico da lista de interesses.

Finalmente, o caso de uso **Notificar Assinantes** define a funcionalidade de distribuição de atualizações provenientes do **PT** para todos os tópicos registrados no **DT**. Esse caso de uso evidencia a capacidade do *Middleware* de desacoplar a responsabilidade de gerenciamento das operações de leitura e escrita, permitindo que o **DT** receba atualizações de forma eficiente, sem necessitar gerenciar diretamente a comunicação com o **DT**.

## 4.4 Visão Geral da Arquitetura

Esta Tese propõe e projeta uma arquitetura de middleware voltada para apoiar o desenvolvimento de sistemas Digital Twin, com foco nos aspectos de comunicação. A arquitetura atua como uma camada intermediária de software, responsável por implementar mecanismos de comunicação e orquestração entre os componentes do sistema. Sua concepção foi guiada por princípios de modularidade, flexibilidade e agnosticismo a protocolos, com a finalidade de permitir a reutilização de componentes, a escalabilidade e a evolução contínua em diferentes cenários e tecnologias. A arquitetura assegura a separação clara de responsabilidades entre os mecanismos de comunicação e a regra de negócio do DT, mantendo o sistema desacoplado. Com isso, a lógica central do DT pode evoluir sem afetar diretamente a camada de comunicação, e vice-versa. Essa separação, em conjunto com o caráter modular e agnóstico a protocolos, possibilita a substituição ou atualização de tecnologias sem demandar alterações estruturais significativas, garantindo adaptabilidade e robustez mesmo em ambientes complexos e dinâmicos.

Entre os objetivos da solução, destaca-se a finalidade de fornecer um mecanismo de comunicação com baixo *overhead*, essencial para aplicações industriais de DT que demandam troca eficiente de dados em tempo real. Esse objetivo é explorado por meio de decisões tecnológicas e arquiteturais que conduziram à implementação de uma PoC, na qual são avaliadas métricas de desempenho — como latência, taxa de transferência e validação da arquitetura — para validar a viabilidade e aplicabilidade da proposta em um cenário real. A PoC é discutida em maiores detalhes na Seção 4.5.

A Figura 20 apresenta a estrutura conceitual da arquitetura proposta. Conforme ilustrado, a arquitetura está posicionada como uma camada intermediária entre o DT e o PT. Nesse contexto, a arquitetura é conceitualmente dividida em três principais componentes desacoplados: o Digital Twin, o Middleware e o Physical Twin. Essa separação constitui um princípio fundamental da proposta, pois distribui de forma clara as responsabilidades e abstrai as complexidades inerentes a cada domínio. Além disso, essa característica favorece a execução de testes isolados, simplifica atividades de manutenção e possibilita a análise funcional e de desempenho do Middleware, do DT e do PT de forma independente.

O componente **Digital Twin** representa a contraparte virtual do sistema físico. Esse componente modela o comportamento, estados e iterações de forma digital. Dentro do contexto da arquitetura, o DT realiza ações — como operações de leitura e escrita — e processa as respostas provenientes do sistema físico. Como o componente Digital Twin é desacoplado, ele pode ser implementado utilizando diferentes tecnologias, conforme os requisitos da aplicação. Por exemplo, o DT pode ser desenvolvido em ambientes gráficos

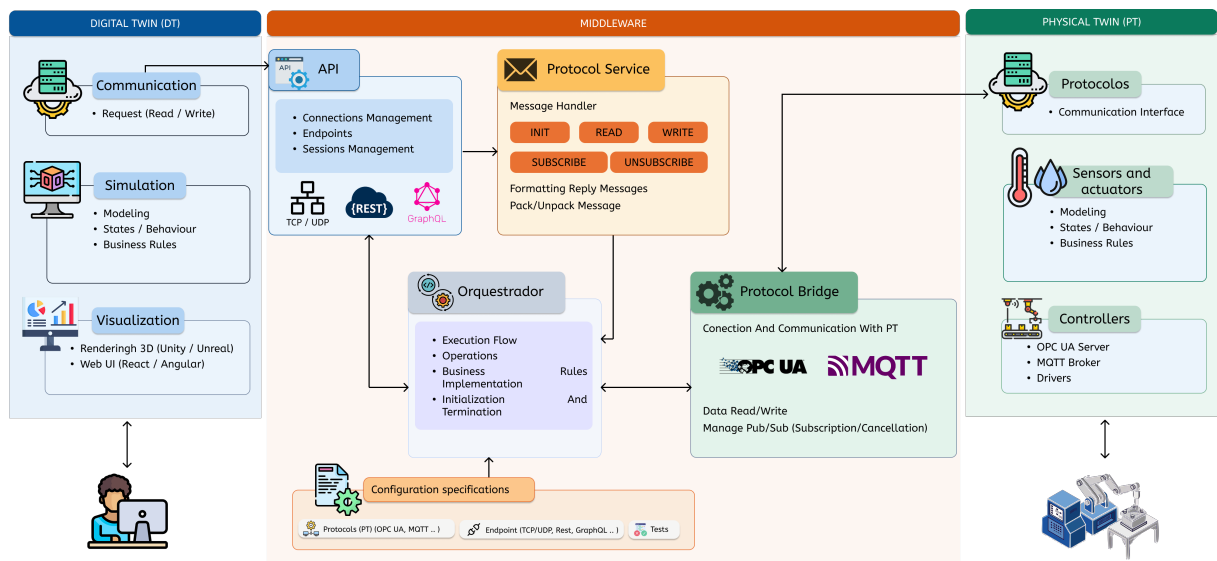


Figura 20: Visão geral da arquitetura proposta do middleware.

como Unity<sup>1</sup> ou Unreal Engine<sup>2</sup>, que oferecem recursos avançados para representação tridimensional e simulação visual do sistema físico. Alternativamente, o Digital Twin pode ser construído como um painel interativo utilizando frameworks web modernos como React<sup>3</sup>, Angular<sup>4</sup> ou Vue.js<sup>5</sup>, permitindo visualizações dinâmicas e interfaces customizadas para monitoramento em tempo real. Essa flexibilidade arquitetural permite que o Digital Twin seja adaptado a diferentes contextos, integrando visualização, simulação e análise de forma modular. Além disso, possibilita a criação, o teste e a reprodução isolada de diferentes partes do Digital Twin, facilitando o desenvolvimento incremental e a validação de seus componentes de maneira independente.

O **Physical Twin** corresponde ao ativo físico real, dotado de sensores, atuadores e controladores responsáveis pela aquisição e execução de comandos no ambiente físico. Esse componente fornece dados em tempo real e recebe instruções de controle, normalmente por meio de protocolos industriais padronizados, como **OPC UA**. Além de sua função operacional, o Physical Twin é a fonte primária de dados para o Digital Twin, viabilizando o monitoramento, a análise e a sincronização contínua entre os domínios físico e virtual.

No tocante aos componentes do **Middleware**, a **API** atua primordialmente como um "ponto de entrada" (*endpoint*) entre o **DT** e a infraestrutura interna do middleware. Sua responsabilidade central é estabelecer e manter o canal de comunicação entre **DT** e

<sup>1</sup>Unity: <https://unity.com/>

<sup>2</sup>Unreal Engine: <https://www.unrealengine.com/>

<sup>3</sup>React: <https://reactjs.org/>

<sup>4</sup>Angular: <https://angular.io/>

<sup>5</sup>Vue.js: <https://vuejs.org/>

o middleware, incluindo a inicialização do ponto de acesso (configuração de *host* e porta, por exemplo), a aceitação de conexões e o gerenciamento do ciclo de vida da sessão de comunicação. A **API** concentra, portanto, os aspectos relacionados ao gerenciamento da conexão, tais como abertura e encerramento de sessões, envio e recebimento de mensagens e tratamento de eventos de desconexão. Ao restringir a **API** a esse papel de mediação e gerenciamento da comunicação, a arquitetura estabelece uma separação clara entre o mecanismo de transporte e a lógica de processamento do middleware. Essa decisão arquitetural permite que o ponto de acesso permaneça simples, estável e independente de mudanças na semântica dos protocolos ou nas regras de negócio, contribuindo para a modularidade, extensibilidade e manutenibilidade da solução proposta.

O componente **Protocol Service** é responsável por definir e implementar o protocolo de comunicação lógico do middleware. Seu papel central consiste em interpretar semanticamente as mensagens recebidas do **DT**, identificar os comandos nelas contidos e mapear cada comando para a operação interna correspondente. Dessa forma, o Protocol Service atua como a camada semântica, estabelecendo a ponte entre o fluxo de dado recebido pela **API** e os serviços funcionais do middleware. A interpretação das mensagens é realizada a partir da extração de campos estruturados do protocolo (por exemplo, cabeçalhos e dados), permitindo identificar comandos como inicialização da comunicação, leitura, escrita, inscrição e cancelamento de inscrição. Cada comando é associado explicitamente a um método responsável por tratá-lo, por meio de um mecanismo de mapeamento entre identificadores de comandos e funções internas. Essa estratégia torna o protocolo extensível, possibilitando a inclusão de novos comandos ou a modificação dos existentes sem impactos estruturais sobre os demais componentes da arquitetura. Além do despacho das operações, o Protocol Service define o formato das mensagens de resposta e notificação geradas pelo middleware. Esse componente é responsável por construir respostas síncronas às requisições do **DT**, incluindo metadados como status de execução, timestamp para instrumentação de desempenho, bem como por estruturar mensagens assíncronas de publicação de eventos associadas ao modelo *publish/subscribe*. Dessa forma, o protocolo mantém consistência tanto em interações do tipo *request/reply* quanto em comunicações orientadas a eventos. É importante ressaltar que o Protocol Service não implementa diretamente o acesso ao **PT** nem gerencia o mecanismo de transporte da comunicação. Em vez disso, ele delega a execução efetiva das operações a outros componentes do middleware, mantendo-se desacoplado tanto do nível de transporte quanto da lógica específica de interação com os dispositivos físicos. Essa separação de responsabilidades permite que o protocolo de comunicação evolua de forma independente, reforçando a modularidade, a

manutenibilidade e a adaptabilidade da arquitetura proposta.

O componente **Protocol Bridge** é responsável por encapsular e implementar a comunicação direta entre o *middleware* e o PT. Do ponto de vista arquitetural, ele atua como um adaptador especializado (*wrapper*), cujo papel é traduzir as operações abstratas definidas pelo *middleware* em operações concretas com o domínio físico, de acordo com o protocolo e a tecnologia utilizados pelo sistema físico. O componente implementa o papel de cliente do PT, concentrando todas as responsabilidades relacionadas à conexão, leitura de valores, escrita de valores, inscrição e cancelamento de inscrições em atualizações de dados. Dessa forma, o restante da arquitetura interage com o PT de maneira homogênea, sem conhecimento explícito sobre detalhes de endereçamento, temporização, modelo de interação ou formato de mensagens.

Finalmente, o **Orchestrator** atua como o coordenador central da arquitetura do *Middleware*, responsável por gerenciar o fluxo principal de execução do sistema. Do ponto de vista funcional, o Orchestrator atua como o elo central entre a API, o *Protocol Service* e o *Protocol Bridge*. As mensagens recebidas pela API são encaminhadas ao *Protocol Service*, que identifica os comandos e delega sua execução ao Orchestrator. Este, por sua vez, invoca as operações apropriadas no *Protocol Bridge*, responsável pela interação efetiva com o PT. Essa organização estabelece uma separação clara entre acesso, interpretação semântica, coordenação e execução, reforçando a modularidade da arquitetura. O Orchestrator utiliza o **Configuration Specification** que define os parâmetros fundamentais para a inicialização e operação do *Middleware*. Essas configurações incluem o protocolo de comunicação utilizado para interação com o PT e o tipo de API utilizada pelo *Middleware*, por exemplo.

## 4.5 Prova de Conceito

Esta Seção apresenta o desenvolvimento da prova de conceito (PoC), que possui como principal objetivo a demonstração da implementação prática da arquitetura proposta para o *Middleware*. A PoC foi desenvolvida com base nos princípios arquiteturais fundamentais e nas funcionalidades essenciais a um sistema de DT, com a finalidade de validar a viabilidade técnica e operacional da arquitetura proposta. A implementação da prova de conceito foi orientada pelos requisitos e pelo modelo de DT definidos no Capítulo 5, servindo como uma instância representativa de aplicação da arquitetura proposta.

Uma tecnologia pode ser caracterizada em diferentes estágios de maturidade por meio

do conceito de [Technology Readiness Level \(TRL\)](#) ([Mankins, 1995](#)). O TRL organiza o ciclo de evolução tecnológica em nove níveis contínuos. Nos estágios iniciais (TRL 1 e TRL 2), concentram-se a observação de princípios básicos e a formulação do conceito tecnológico, nesse estágio não há implementação e comprovação prática. O TRL 3 corresponde à prova de conceito experimental, em que simulações e protótipos preliminares permitem verificar a viabilidade dos princípios. O TRL 4 marca a validação em ambiente de laboratório, com protótipos funcionais avaliados sob condições controladas. Em seguida, o TRL 5 e o TRL 6 ampliam a confiança ao validar e demonstrar a tecnologia em ambientes relevantes, mais próximos do contexto operacional. O TRL 7 situa a demonstração em condições reais de operação, já com sistemas quase finais. O TRL 8 corresponde ao sistema concluído e qualificado após testes formais. Finalmente, o TRL 9 representa a adoção completa, quando a tecnologia está em uso comprovado em ambiente operacional. Nesse sentido, a escala de TRL estabelece uma referência para avaliar a evolução da pesquisa desde sua fase de ideação até a implementação em ambiente de produção. A Figura 21 apresenta a escala de TRL e seus respectivos níveis, desde a pesquisa básica até a operação de sistemas reais.

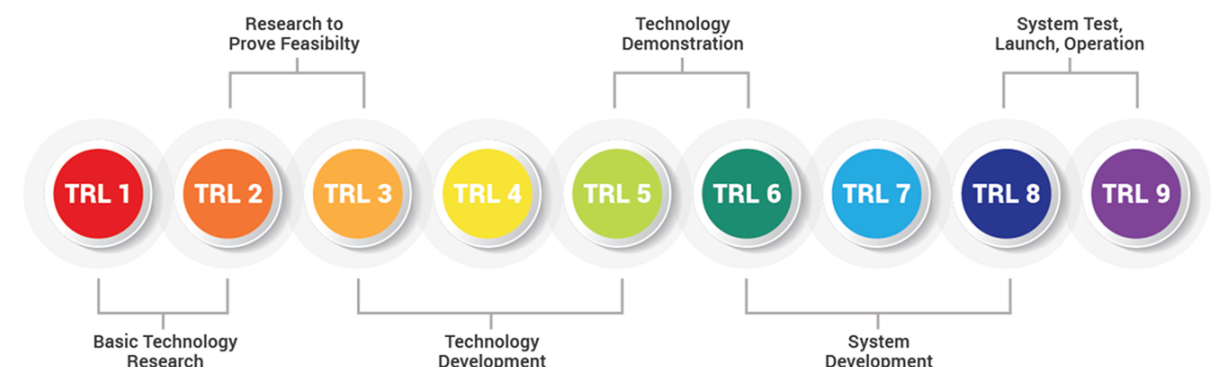


Figura 21: Escala de níveis de maturidade tecnológica (TRL 1–9). Fonte: ([Abaco Systems, 2025](#)).

A implementação do middleware desenvolvida nesta Tese pode ser caracterizada como uma [PoC](#), visto que consolida os princípios arquiteturais e valida suas funcionalidades fundamentais em um protótipo funcional. Nesse sentido, a [PoC](#) é consistente com os níveis intermediários da escala [TRL](#), situando-se na escala [TRL 4](#), no qual a tecnologia é validada em ambiente de laboratório sob condições controladas. Embora a [PoC](#) já permita a avaliação da viabilidade técnica e operacional da arquitetura, trata-se de um estágio exploratório, que antecede a aplicação em produção (nível de produção em escala e industrial). Nesse sentido, a [PoC](#) estabelece a base para evoluções futuras em direção ao [TRL 5](#), no qual a tecnologia pode ser validada em ambientes relevantes, e ao [TRL 6](#),

caracterizado pela demonstração em cenários operacionais mais próximos do contexto de aplicação.

A PoC está disponível em [https://gitlab.com/lseveriano/middleware\\_dt](https://gitlab.com/lseveriano/middleware_dt).

### 4.5.1 Diagrama de classes e Modularidade

Essa Seção descreve a implementação da arquitetura do middleware sob a perspectiva arquitetural do sistema. É destacado como os padrões de projeto e decisões arquiteturais foram empregados para assegurar modularidade, flexibilidade e manutenibilidade. Entre os padrões utilizados, destacam-se o *Factory Method* (Gamma *et al.*, 1994), aplicado na criação de objetos de comunicação desacoplados de implementações específicas; e o *Observer* (Gamma *et al.*, 1994), responsável pelo gerenciamento de eventos e notificações (*Publisher/Subscriber*).

A Figura 22 apresenta o diagrama Unified Modeling Language (UML) de classes que modela a estrutura interna do *Middleware*. O Apêndice A apresenta uma versão com mais detalhes do diagrama UML da arquitetura, com os métodos e atributos associados à respectivas classes. O diagrama está organizado em pacotes — *InterfaceIO*, *InterfaceProtocols* e *Functionalities* —, de modo a evidenciar as diferentes camadas funcionais e a distribuição de responsabilidades entre os diferentes módulos e classes, bem como as relações de dependência, agregação e composição. Dessa forma, demonstra como a camada de abstração proposta originalmente na arquitetura contribui para a evolução futura do sistema, minimizando o impacto na aplicação como um todo.

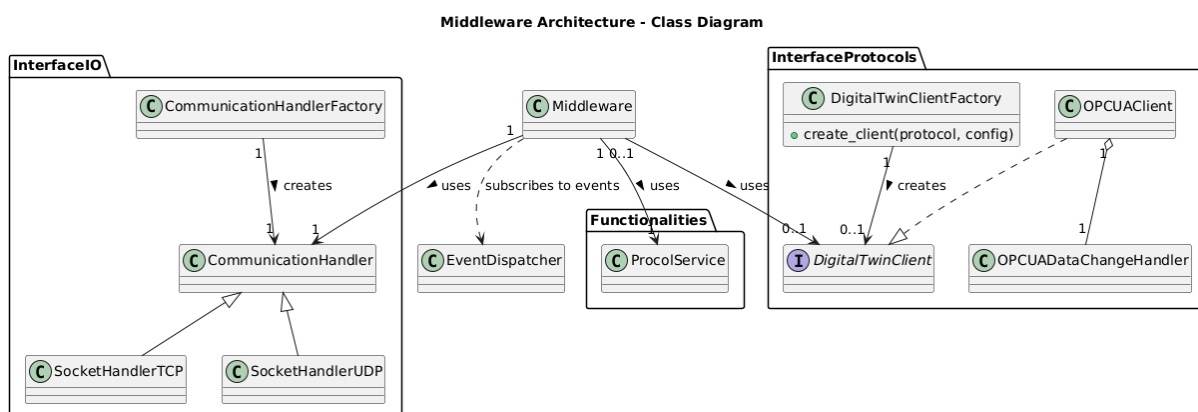


Figura 22: Diagrama de classes da Implementação do Middleware.

No pacote *InterfaceIO*, a interface *CommunicationHandler* define o contrato que abstrai a camada de transporte responsável pelo envio e recebimento de mensagens. A partir

dessa definição, duas implementações concretas — *SocketHandlerTCP* e *SocketHandlerUDP* — materializam, respectivamente, a comunicação por meio dos protocolos [TCP](#) e [UDP](#). Essa organização permite a seleção do mecanismo de transporte de forma transparente para os demais componentes do Middleware, que permanecem independentes dos detalhes de implementação subjacentes. A instanciação desses manipuladores é realizada pela classe *CommunicationHandlerFactory*, a qual centraliza o processo de criação e assegura flexibilidade arquitetural: a introdução de novos mecanismos de comunicação demanda apenas a implementação de classes aderentes ao contrato estabelecido pela interface, seguidas de seu registro na *factory*.

O pacote *InterfaceProtocols* abstrai a comunicação com o [PT](#). A interface *DigitalTwinClient* define as interfaces de conexão, leitura, escrita e gerenciamento de assinaturas, funcionando como uma fachada agnóstica ao protocolo do [PT](#). Uma implementação concreta da interface, *OPCUAClient*, encapsula a interação via [OPC UA](#) e utiliza o *OPCUADataChangeHandler* para tratar notificações de mudança de valor (*data change*). Tal arranjo separa a lógica de alto nível do *Middleware* das particularidades do protocolo adotado, preservando a possibilidade de incorporar, no futuro, outros clientes (por exemplo, Modbus ou MQTT) sem alterações na camada de aplicação. De forma análoga ao transporte, a *DigitalTwinClientFactory* assume a responsabilidade de instanciar o cliente apropriado a partir da configuração especificada.

O pacote *Functionalities* concentram os componentes que estruturam a lógica de aplicação do *Middleware*. Dentro desse pacote, a classe *ProtocolService* define o vocabulário de comandos e de suas semânticas. Em termos conceituais, o serviço atua como um roteador semântico: valida as requisições, aciona leituras e escritas por meio do cliente de protocolo, ao final, padroniza as respostas a serem devolvidas ao [DT](#).

Finalmente, a classe *Middleware* integra os pacotes anteriores e provê o ciclo de vida do sistema. Sua função é orquestrar a inicialização do *CommunicationHandler*, estabelecer o serviço de recepção de mensagens, acoplar o *ProtocolService* como interpretador dos comandos e manter a referência ao *DigitalTwinClient* conforme o protocolo selecionado. Essa orquestração é realizada por meio de interfaces e “fábricas”, o que assegura baixo acoplamento entre os módulos e permite substituições orientadas por configuração, sem a necessidade de alterações estruturais. Em conformidade com o princípio da inversão de dependência, o *Middleware* interage com abstrações, e não com implementações concretas, o que facilita a evolução tecnológica, a realização de testes e a simulação de componentes.

Para reforçar a rastreabilidade entre requisitos e arquitetura, a Tabela [4](#) apresenta o

mapeamento entre os requisitos e os componentes responsáveis por atendê-los.

| Requisito                      | Componente(s) Relacionado(s)            |
|--------------------------------|-----------------------------------------|
| Modularidade                   | Orchestrator, API                       |
| Extensibilidade                | Protocol Service, Protocol Bridge, API  |
| Protocol-agnostic              | Protocol Bridge                         |
| Eficiência (Baixo Overhead)    | InterfaceIO, Cache Interno              |
| Separação de Responsabilidades | Digital Twin, Middleware, Physical Twin |

Tabela 4: Rastreabilidade entre requisitos arquiteturais e componentes.

### 4.5.2 Ambiente de Implementação

A implementação da Prova de Conceito foi realizada em Python 3.13.2, linguagem escolhida pela ampla disponibilidade de bibliotecas, comunidade ativa e agilidade no desenvolvimento de protótipos. A utilização de uma linguagem de alto nível, com suporte nativo a múltiplos paradigmas de programação e bibliotecas maduras, favoreceu a implementação dos conceitos arquiteturais definidos, reduzindo o esforço necessário para integração entre componentes de comunicação, protocolos industriais e estruturas de dados.

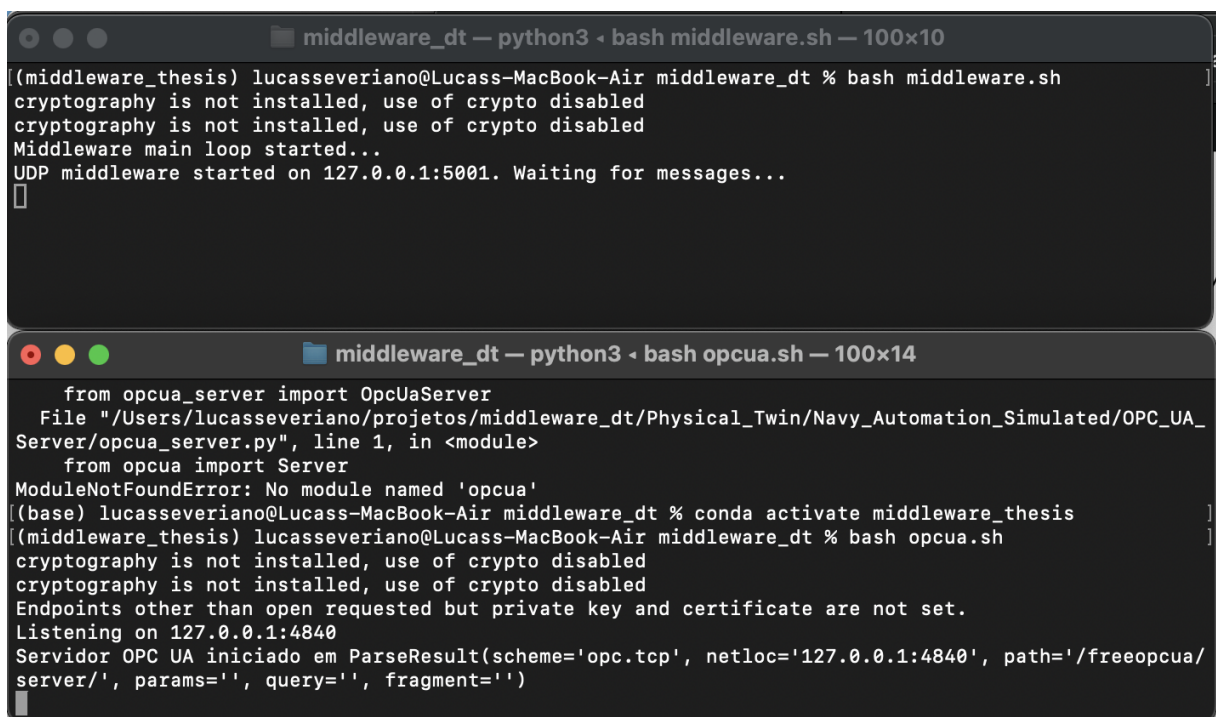
É importante ressaltar que a escolha por Python ([Van Rossum; Drake, 1995](#)) atende ao propósito exploratório da PoC, mas não reflete necessariamente a linguagem mais apropriada para uma versão de produção do Middleware. Trabalhos relacionados ([Bass; Clements; Kazman, 2021](#); [Tanenbaum; Bos, 2014](#)) destacam que requisitos não-funcionais, como eficiência em tempo de execução, consumo reduzido de memória e segurança de concorrência, são fatores fundamentais em sistemas críticos. Nesse sentido, linguagens orientadas a sistemas, como Go ([Donovan; Kernighan, 2015](#)) ou Rust ([Klabnik; Nichols, 2021](#)), apresentam vantagens significativas em cenários industriais devido ao gerenciamento eficiente de recursos, suporte nativo a concorrência segura e melhor adequação para implantação em ambientes de produção. Assim, embora a PoC apresentada tenha sido implementada em Python, a arquitetura proposta mantém-se independente da linguagem de programação, podendo ser reimplementada em linguagens de mais baixo nível em fases futuras do projeto.

O ambiente de software foi composto por um conjunto de bibliotecas de apoio, entre as quais se destacam: *opcua* (0.98.13)<sup>6</sup>, responsável pela comunicação com servidores OPC

<sup>6</sup><https://github.com/FreeOpcUa/python-opcua>

UA; *socket*, para implementação do mecanismo de transporte TCP e UDP; e *threading*, utilizada para controle concorrente de execução.

Outro aspecto relevante do ambiente de implementação é sua natureza **multiplataforma**. A PoC foi projetada para operar em sistemas Linux, Windows e macOS sem a necessidade de modificações significativas, explorando a portabilidade oferecida pela linguagem Python e pelas bibliotecas utilizadas. Essa característica reforça o requisito arquitetural de independência de plataforma, permitindo que o Middleware seja implantado em diferentes contextos industriais com esforço reduzido de adaptação. Dessa forma, a solução pode ser facilmente executada em estações de desenvolvimento, servidores dedicados ou dispositivos embarcados, preservando consistência funcional e comportamental entre ambientes distintos. A Figura 23 ilustra o middleware em execução como serviço, em sistema macOS, utilizando socket UDP como mecanismo de transporte.



```
middleware_dt — python3 • bash middleware.sh — 100x10
((middleware_thesis) lucasseveriano@Lucass-MacBook-Air middleware_dt % bash middleware.sh
cryptography is not installed, use of crypto disabled
cryptography is not installed, use of crypto disabled
Middleware main loop started...
UDP middleware started on 127.0.0.1:5001. Waiting for messages...
█

middleware_dt — python3 • bash opcua.sh — 100x14
from opcua_server import OpcUaServer
File "/Users/lucasseveriano/projetos/middleware_dt/Physical_Twin/Navy_Automation_Simulated/OPC_UA_Server/opcua_server.py", line 1, in <module>
  from opcua import Server
ModuleNotFoundError: No module named 'opcua'
((base) lucasseveriano@Lucass-MacBook-Air middleware_dt % conda activate middleware_thesis
((middleware_thesis) lucasseveriano@Lucass-MacBook-Air middleware_dt % bash opcua.sh
cryptography is not installed, use of crypto disabled
cryptography is not installed, use of crypto disabled
Endpoints other than open requested but private key and certificate are not set.
Listening on 127.0.0.1:4840
Servidor OPC UA iniciado em ParseResult(scheme='opc.tcp', netloc='127.0.0.1:4840', path='/freeopcua/server/', params='', query='', fragment='')
█
```

Figura 23: Middleware em execução como serviço.

### 4.5.3 Arquivo de Configuração

A inicialização do Middleware é realizada com base na leitura de um arquivo de configuração. Esse arquivo centraliza os parâmetros essenciais para a execução do sistema, permitindo que diferentes protocolos industriais, mecanismos de comunicação e ajustes de experimentação sejam definidos de maneira externa ao código-fonte. Essa abordagem contribui para a flexibilidade arquitetural e para o baixo acoplamento, pois possibilita

adaptar o comportamento do Middleware por meio de ajustes no arquivo de configuração, sem a necessidade de modificações estruturais na implementação. O trecho 4.1 apresenta um exemplo de arquivo de configuração utilizado na PoC.

```
1 {  
2   "protocol_physical_twin" : "OPC-UA",  
3   "protocol_middleware_communication_handler" : "socket_udp",  
4   "socket_host_middleware": "127.0.0.1",  
5   "socket_port_middleware": 5001,  
6   "ENABLE_TEST_PADDING": false,  
7   "MIN_RESPONSE_SIZE_BYTES": 1000  
8 }
```

Listing 4.1: Exemplo de arquivo de configuração do Middleware

A Tabela 5 descreve cada parâmetro, suas opções e sua finalidade.

#### 4.5.4 Estrutura de Mensagens e Protocolo de Comunicação

A interação entre o DT e o Middleware é mediada por um protocolo de comunicação leve, cujo objetivo é garantir uniformidade semântica, simplicidade de parsing e interoperabilidade tecnológica. Para atender a esses requisitos, optou-se pela utilização do formato JSON, amplamente reconhecido pela legibilidade, suporte em múltiplas linguagens e facilidade de integração com diferentes plataformas. Embora alternativas baseadas em formatos binários, como o *Protocol Buffers* (Google, 2025), ou frameworks de RPC, como o RPC sobre HTTP/3 (Google, 2016; Iyengar; Thomson, 2022), possam oferecer ganhos de eficiência em cenários de produção, o uso de JavaScript Object Notation (JSON) (Ecma International, 2017) mostrou-se mais adequado ao caráter exploratório desta PoC. Sua simplicidade favorece a depuração, reduz barreiras de integração em prototipação rápida e permite a participação de clientes heterogêneos sem necessidade de bibliotecas específicas.

As mensagens transmitidas seguem um padrão com dois blocos — **header** e **data**, conforme ilustrado na Figura 24. Portanto, no bloco **header** são concentradas os metadados para identificação, correlação, validação, roteamento, auditoria e controle. Nesse sentido, o **timestamp** registra o instante de emissão (permitindo ordenação e cálculo de latência fim a fim), **client address** identifica a origem da requisição e **command** especifica a operação solicitada (inicialização, leitura, escrita ou assinatura, por exemplo). Por outro lado, o bloco **data** carrega os parâmetros específicos de cada operação. Por exemplo, para operações de leitura, o **bloco data** contém apenas o Tópico (*topic*) necessário para realizar

| Parâmetro                            | Opções                                            | Descrição                                                                                                                                                                                                   |
|--------------------------------------|---------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>protocol_physical_twin</code>  | <code>OPC-UA</code> , <code>MQTT</code> ..        | Define o protocolo utilizado para comunicação com o <a href="#">PT</a> . No estado atual, a <a href="#">PoC</a> suporta apenas esse protocolo, porém a arquitetura permite a inclusão de novos adaptadores. |
| <code>protocol_middleware</code>     | <code>socket_tcp</code> , <code>socket_udp</code> | Especifica o mecanismo de comunicação da API do Middleware com os clientes ( <a href="#">DT</a> ).                                                                                                          |
| <code>socket_host_middleware</code>  | Endereço IP (ex.: 127.0.0.1)                      | Define o endereço de rede no qual o Middleware ficará disponível. Esse parâmetro é utilizado apenas quando o mecanismo de comunicação é baseado em <i>sockets</i> .                                         |
| <code>socket_port_middleware</code>  | Inteiro (ex.: 5001)                               | Porta de rede utilizada para estabelecer a comunicação com o Middleware. Relevante apenas para <i>handlers</i> baseados em <i>sockets</i> .                                                                 |
| <code>ENABLE_TEST_PADDING</code>     | <code>true</code> , <code>false</code>            | Habilita a inserção de dados adicionais no corpo das mensagens para experimentos de desempenho. Quando ativado, permite simular cenários com mensagens maiores.                                             |
| <code>MIN_RESPONSE_SIZE_BYTES</code> | Inteiro (ex.: 1000)                               | Define o tamanho mínimo, em bytes, das respostas enviadas pelo Middleware. Esse parâmetro é empregado nos experimentos para padronizar a carga de resposta.                                                 |

Tabela 5: Parâmetros do arquivo de configuração do Middleware.

a leitura. Em contrapartida, para operações de escrita é necessário indicar o tópico e valor. Embora a Figura 24 destaque a requisição, as mensagens de resposta seguem o mesmo padrão, incluindo **status** e timestamps no cabeçalho e o resultado em **data**. Os formatos completos de requisição e resposta de todas as funcionalidades e operações implementadas na PoC estão mapeados no Apêndice B.

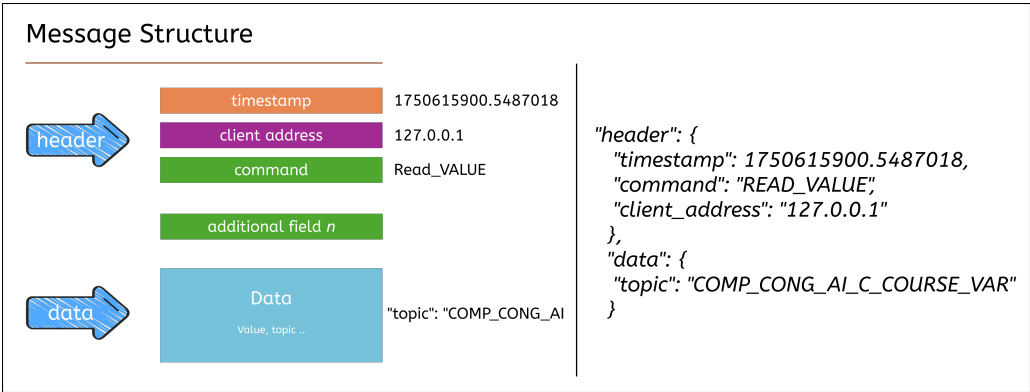


Figura 24: Estrutura de mensagem.

### 4.5.5 Considerações sobre a Implementação

Durante a implementação do Middleware, algumas decisões técnicas foram adotadas para garantir aderência aos requisitos arquiteturais definidos na Seção 4.2, em especial no que se refere à eficiência e ao baixo *overhead* na comunicação.

No contexto do protocolo OPC UA, cada variável (ou *tag*) deve ser resolvida no servidor antes de ser utilizada, operação que implica a obtenção do respectivo *NodeId* e a criação de um objeto do tipo *Node* no cliente. Segundo Mahnke, Leitner e Damm (Mahnke; Leitner; Damm, 2009), o acesso a dados em OPC UA pressupõe um processo de identificação semântica da variável, envolvendo etapas de resolução de *namespace* e validação de tipo. Esse procedimento, quando executado de forma repetida em leituras e escritas sucessivas, acarreta sobrecarga considerável no processo de comunicação. Em implementações baseadas em Python, como a biblioteca *opcua*, essa sobrecarga torna-se ainda mais perceptível devido à natureza interpretada da linguagem, que adiciona custos relacionados à alocação dinâmica de objetos e à validação em tempo de execução. Conforme destacado por (Zhang; Wang; Zhang, 2020), o modelo cliente-servidor do OPC UA envolve múltiplas trocas de mensagens de serviço para cada operação de leitura ou escrita, de modo que a resolução redundante de nós amplifica a latência total observada.

Sem o uso de mecanismos de cache (especificado na Seção 4.3), o custo de cada operação de leitura ou escrita pode ser modelado como:

$$T_{\text{operação}} = T_{\text{init}} + T_{\text{exec}},$$

em que  $T_{\text{init}}$  corresponde ao tempo de resolução e inicialização do nó (*tag*) do PT, e  $T_{\text{exec}}$  representa o tempo efetivo da operação de leitura ou escrita. Em cenários em que as mesmas variáveis são acessadas de forma repetitiva e em alta frequência, esse custo adicional impacta diretamente a latência do sistema. Para mitigar esse problema, o *Middleware* implementa um mecanismo de cache de inicialização (conforme apresentado na Seção 4.3), para armazenar na região de memória do *Middleware* os nós já resolvidos. Com isso, operações seguintes sobre a mesma variável utilizam diretamente o objeto armazenado, reduzindo o tempo total para:

$$T_{\text{operação}} = T_{\text{exec}} \quad (\text{quando há acerto no cache}).$$

O mecanismo de cache de inicialização fornece uma estrutura semântica entre as

operações síncronas (leitura e escrita) e operações assíncronas (assinatura de eventos), pois o mesmo objeto *Node* é reutilizado em diferentes contextos. A Figura 25 ilustra o funcionamento do cache de inicialização implementado no Middleware. No lado do servidor **OPC UA**, os objetos reais (*Real Objects*) são representados virtualmente no *AddressSpace* por meio de diferentes nós (*Node 1*, *Node 2*, *Node 3*, ..., *Node 6*). Quando o *Middleware* acessa, por exemplo, o Node 5, o identificador correspondente é resolvido e armazenado no cache interno, conforme representado pela entrada "Node 5": `Node(StringNodeId(s=Node 5))`. A partir desse momento, operações de leitura, escrita ou assinatura sobre o mesmo nó passam a utilizar diretamente o objeto armazenado no espaçamento de memória do *Middleware*, o que elimina a necessidade de resoluções repetidas no servidor.

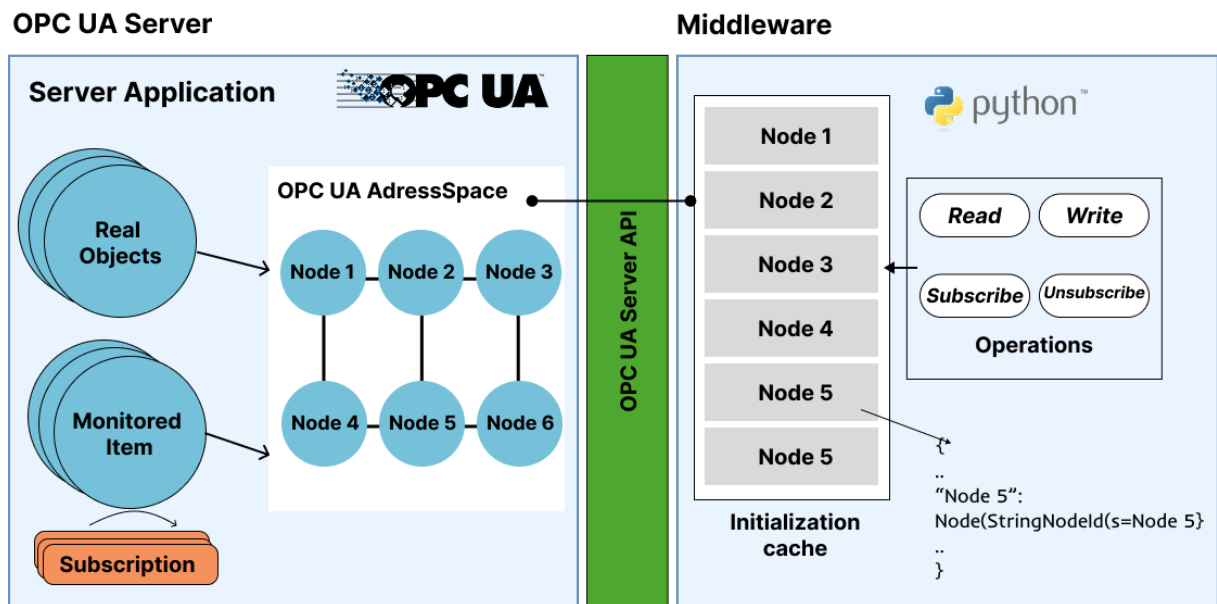


Figura 25: Estrutura do cache de inicialização.

O Algoritmo 1 descreve o procedimento de inicialização de um recurso no *Physical Twin*. Inicialmente, o identificador lógico do recurso é utilizado para obter a referência correspondente no sistema físico. Em seguida, de acordo com a operação solicitada, o algoritmo executa uma leitura ou escrita para validar a disponibilidade do recurso. Uma vez validado, o recurso é armazenado em um cache interno do middleware, permitindo que acessos subsequentes sejam realizados de forma mais eficiente, sem a necessidade de nova inicialização.

---

**Algoritmo 1** Inicialização de Recurso no Physical Twin

---

**Require:**  $tag$  (identificador do recurso),  $action \in \{get\_value, set\_value\}$ ,  $new\_value$  (opcional)

**Ensure:**  $result$  (valor lido ou confirmação de escrita) ou  $false$  em caso de falha

```

1:  $node \leftarrow GETNODEFROMPT(tag)$ 
2: if  $node = \emptyset$  then
3:   return  $false$ 
4: end if
5: if  $action = get\_value$  then
6:    $value \leftarrow READVALUE(node)$ 
7:    $Cache[tag] \leftarrow node$ 
8:   return  $value$ 
9: end if
10: if  $action = set\_value$  then
11:   if  $new\_value = \emptyset$  then
12:     return  $false$ 
13:   end if
14:    $WRITEVALUE(node, new\_value)$ 
15:    $Cache[tag] \leftarrow node$ 
16:   return  $true$ 
17: end if
18: return  $false$ 

```

---

O Algoritmo 2 descreve o procedimento para realizar a leitura, incorporando um mecanismo de inicialização. Inicialmente, o algoritmo verifica se o recurso identificado por  $tag$  já se encontra previamente inicializado e armazenado na estrutura interna de *cache* (*Nodes*). Caso positivo, a leitura é realizada diretamente, reduzindo o custo de acesso e o tempo de resposta. Caso contrário, o algoritmo executa a etapa de inicialização do recurso (Algoritmo 1), seguido da operação de leitura. De forma semelhante, o Algoritmo 3 descreve o procedimento de escrita, seguindo o mesmo princípio.

---

**Algoritmo 2** Leitura de Valor com Inicialização

---

**Require:** *tag* (identificador lógico do recurso)**Ensure:** *value* (valor lido do recurso)

```

1: if tag ∈ Nodes then
2:   value ← READFROMNODE(tag)
3: else
4:   value ← INITIALIZENODE(tag, get_value())
5: end if
6: return value

```

---



---

**Algoritmo 3** Escrita de Valor no Physical Twin

---

**Require:** *tag* (identificador do recurso), *new\_value* (valor a ser escrito)**Ensure:** *success* ∈ {true, false}

```

1: if tag ∈ Nodes then
2:   SETVALUE(Nodes[tag], new_value)
3:   success ← true
4: else
5:   success ← INITIALIZENODE(tag, set_value(new_value))
6: end if
7: return success

```

---

Conforme discutido nas Seções anteriores, o *Middleware* adota o padrão [Publish–Subscribe](#) (Pub/Sub) como mecanismo para notificação sobre alterações em nós (variáveis) do PT. No contexto desta implementação, tal mecanismo foi implementado sobre o protocolo [OPC UA](#). O *Middleware*, por sua vez, atua como difusor, redistribuindo as atualizações recebidas para todos os clientes previamente inscritos no tópico correspondente, de forma uniforme e padronizada. A Figura 26 exibe a implementação do mecanismo de Pub/Sub do *Middleware*. Essas notificações percorrem a [API](#) do servidor (*OPC UA Server API*) e chegam ao *Middleware*, onde o nó correspondente já se encontra registrado no cache de inicialização. O *Middleware*, então, associa esse nó a um *handle* interno, que organiza e difunde os eventos recebidos para os clientes inscritos. Dessa forma, as operações de *Subscribe* e *Unsubscribe* permitem ao *Middleware* controlar dinamicamente quais variáveis devem ser monitoradas, enquanto as mensagens de *Notification* asseguram a atualização dos valores. Nesse sentido, as implementações de *subscribe* e *unsubscribe* dependem diretamente do modelo oferecido pela biblioteca do protocolo adotado no PT. No caso específico do [OPC UA](#) em Python, a criação e o encerramento de *subscriptions* exigem

chamadas próprias (`create_subscription`, `subscribe_data_change`, `unsubscribe`) e a instalação de um *handler* de eventos de alteração de dados.

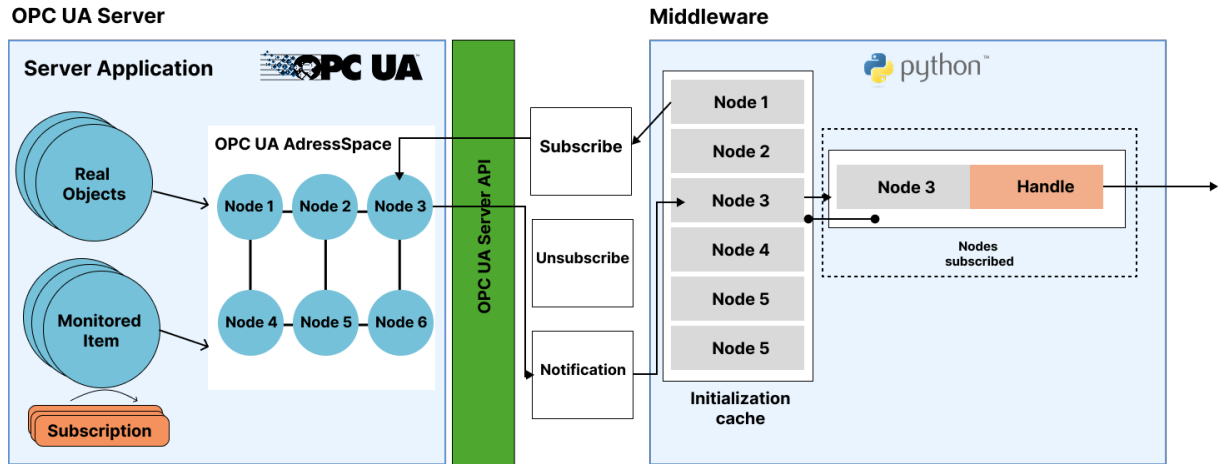


Figura 26: Estrutura de *publish/subscribe* do Middleware.

O Algoritmo 4 descreve o procedimento de inscrição de um cliente em um tópico de interesse no *Middleware*. Inicialmente, o algoritmo solicita ao *Physical Twin* a inscrição no tópico requerido, garantindo que a origem dos dados esteja habilitada a produzir atualizações. Caso essa etapa falhe, o processo é interrompido e a operação é considerada malsucedida. Em seguida, o algoritmo realiza a atualização da estrutura interna de assinaturas do *Middleware*, adicionando o identificador do cliente ao conjunto de assinantes do tópico, operação que é protegida por um mecanismo de sincronização para assegurar consistência em ambientes concorrentes. Por fim, a inscrição é confirmada, permitindo que o *DT* passe a receber notificações sempre que ocorrerem atualizações associadas ao tópico assinado.

---

**Algoritmo 4** Inscrição de Cliente em um Tópico

---

**Require:** *client\_address* (identificador do cliente), *topic* (tópico a ser assinado)

**Ensure:** *success*  $\in \{\text{true}, \text{false}\}$

- 1: *success\_pt*  $\leftarrow$  PT\_SUBSCRIBE(*topic*)
  - 2: **if** *success\_pt* = *false* **then**
  - 3:     **return** *false*
  - 4: **end if**
  - 5: LOCK(Subscriptions)
  - 6: *Subscribers[topic]*  $\leftarrow$  *Subscribers[topic]*  $\cup$  {*client\_address*}
  - 7: UNLOCK(Subscriptions)
  - 8: **return** *true*
-

A implementação dos métodos `subscribe`, `unsubscribe` e `notify_subscribers` no Middleware segue o mesmo princípio já descrito para o cliente OPC UA: delegar ao servidor a criação e remoção das *subscriptions*, enquanto o Middleware mantém uma tabela local de inscrições que associa cada tópico ao conjunto de clientes interessados.

## 4.6 Considerações Finais

Este capítulo apresentou a arquitetura proposta do *Middleware* e sua implementação por meio de uma prova de conceito, enfatizando a separação de responsabilidades entre DT, Middleware e PT. A arquitetura foi concebida para reduzir acoplamentos entre lógica de aplicação e mecanismos de comunicação, de modo a preservar a possibilidade de evolução independente tanto do protocolo quanto dos mecanismos de transporte e de acesso ao domínio físico. Como resultado, a solução organiza o fluxo de comunicação em camadas bem definidas — acesso, interpretação semântica, orquestração e execução — favorecendo modularidade, manutenibilidade e substituição de componentes com impacto localizado.

A discussão dos requisitos arquiteturais evidenciou como modularidade, extensibilidade, agnosticismo a protocolos e eficiência foram incorporados ao projeto. Em particular, a extensibilidade é viabilizada pelo mapeamento explícito entre comandos e rotinas internas no *Protocol Service*, enquanto o agnosticismo a protocolos é sustentado pelo *Protocol Bridge*, que uniformiza as operações de interação com o PT. A eficiência na comunicação foi tratada na PoC por meio do uso de mecanismos leves de transporte (camada de comunicação de baixo nível) e pela introdução de um cache de inicialização, que reduz a recorrência de etapas custosas em acessos repetitivos. Adicionalmente, a adoção do padrão Pub/Sub permitiu suportar comunicação assíncrona, mantendo o *Middleware* como difusor de eventos e preservando o desacoplamento entre produtores e consumidores.

A PoC consolida, portanto, os elementos essenciais da proposta, posicionando a solução como um protótipo funcional de nível intermediário de maturidade tecnológica, e estabelecendo uma base concreta para evolução futura, incluindo: ampliação do conjunto de adaptadores no *Protocol Bridge*, refinamento do protocolo de mensagens, endurecimento de mecanismos de tolerância a falhas e validações, e integração com cenários operacionais mais próximos do ambiente industrial real. Dessa forma, este capítulo fornece o arcabouço arquitetural necessário para sustentar, no restante da Tese, a avaliação experimental e a discussão sobre aplicabilidade e desempenho do *Middleware*.

# 5 Materiais e Métodos

Este capítulo apresenta os materiais, ferramentas, equipamentos e métodos empregados na validação experimental da arquitetura do middleware proposta neste trabalho. O ambiente experimental de validação foi implementado a partir do [Simulador de Treinamento de Imersão \(TI\)](#) para submarinos da classe Tupi<sup>1</sup> ([Santos, 2020](#)), selecionado como estudo de caso em razão da sua relevância e da complexidade inerente à integração entre sistemas físicos e virtuais. O TI foi escolhido por reunir características típicas de um [Digital Twin \(DT\)](#), tais como comunicação em tempo real, utilização de protocolos industriais padronizados ([OPC](#)), integração entre elementos físicos e simulados, e implementação de modelos matemáticos capazes de reproduzir o comportamento dinâmico do sistema real. Nesse contexto, a aplicação do middleware neste ambiente teve como objetivo demonstrar sua aplicabilidade, modularidade e desempenho em um cenário representativo de sistemas industriais complexos. Além disso, este capítulo detalha a estrutura lógica e física do simulador, o processo de reprodução experimental do ambiente de testes e os procedimentos metodológicos empregados para avaliar o desempenho e a eficiência da comunicação do middleware no contexto do [DT](#).

## 5.1 Descrição do Estudo de Caso

### 5.1.1 Breve Histórico do Projeto

O TI foi originalmente desenvolvido na década de 1980 para o [CIAMA](#), da [Marinha do Brasil \(MB\)](#), com a finalidade de treinar o corpo de militares em procedimentos de imersão e operação de submarinos da classe Tupi. O sistema foi concebido de forma modular, combinando componentes fabricados a partir de diferentes empresas, como a Congop (painéis e instrumentação física), uma empresa alemã responsável pelo software de controle e uma inglesa responsável pelos sistemas auxiliares de simulação. Portanto, ao passar dos anos a infraestrutura física tornou-se obsoleta, e a maior parte dos componentes eletrônicos,

---

<sup>1</sup><https://www.webcitation.org/6XZ9kwieT>

cablagens e sistemas de controle não dispunham mais de suporte técnico ou reposição comercial. Além disso, ao longo das décadas, observou-se uma perda significativa do conhecimento técnico sobre a arquitetura de software do simulador, incluindo a documentação, decisões de projeto e, em parte, o próprio código-fonte e artefatos de manutenção, o que dificultou a compreensão do funcionamento interno e a realização de evoluções ou correções. Como consequência, o TI acabou tornando-se inoperante, impossibilitando sua utilização contínua para fins de treinamento.

Em 2020, o Instituto SENAI de Inovação em Sistemas Virtuais de Produção (ISI-SVP) e o Instituto SENAI de Serviço Tecnológico em Automação Industrial (IST) iniciaram o projeto de reestruturação completa do simulador, englobando a substituição de todo o sistema de automação, a reconstrução digital do software de controle e a criação de uma nova arquitetura de comunicação baseada em tecnologias modernas. Essa modernização deu origem à versão utilizada como ambiente de validação experimental desta tese (Santos; Maria *et al.*, 2024). A Figura 27 apresenta a estrutura física do TI.



Figura 27: Imagens do TI.

É importante destacar que, embora o projeto tenha sido originalmente concebido como um simulador de treinamento, ele apresenta características essenciais associadas a sistemas Digital Twin. Especificamente, estabelece um fluxo contínuo e bidirecional de dados entre ativos físicos e uma camada virtual de software responsável pelo monitoramento,

controle e visualização do estado. Essa integração entre entidades físicas e digitais, com sincronização em tempo real e *loops* de *feedback*, está alinhada com a definição central de sistemas *Digital Twin* descrita na literatura (Tao *et al.*, 2023; Fuller *et al.*, 2020; Singh *et al.*, 2022) e discutidas na Seção 2.1 do Capítulo 2.

### 5.1.2 Descrição Técnica

Do ponto de vista lógico e arquitetural de software, o TI segue a estrutura conceitual do framework TupiSIM proposto em (Santos; Maria *et al.*, 2024). A estrutura proposta divide o software em dois módulos principais: o **Estação de Operação do Instrutor (EOI)** e o **Módulo de Simulação (Módulo de Simulação (MS))**.

O EOI é responsável por prover a base estrutural de software do sistema de simulação. Ele concentra os recursos necessários para o gerenciamento e a operação do ambiente de treinamento. Esse módulo engloba funcionalidades essenciais, tais como a inicialização do sistema, o gerenciamento de usuários e perfis de instrutor, o controle e a condução dos exercícios de simulação, a geração de relatórios operacionais e de desempenho, bem como o monitoramento contínuo do estado da simulação. Do ponto de vista arquitetural, o EOI implementa a estação de operação do instrutor, atuando como o principal ponto de interação entre o usuário e o sistema de simulação. Sua função envolve a coordenação das operações de alto nível, incluindo a orquestração dos módulos de simulação e a aplicação das regras de negócio associadas ao treinamento. A Figura 28 apresenta a arquitetura proposta para o EOI. A estrutura de software do EOI foi concebida com base em uma arquitetura em camadas, na qual as responsabilidades são organizadas segundo níveis distintos de abstração. A camada de apresentação é responsável por fornecer uma interface gráfica capaz de capturar as ações do instrutor e, além disso, é capaz de apresentar o estado geral do processo de simulação e do TI em geral. A camada de aplicação coordena a lógica de negócio do sistema, em outras palavras, essa camada realiza a orquestração dos serviços necessários para executar uma ação específica. A camada de serviço concentra as regras de negócio e realiza as devidas chamadas às operações específicas da simulação. A camada de infraestrutura encapsula o acesso a recursos externos, como sistemas de banco de dados, bibliotecas auxiliares, mecanismos de comunicação e operações de entrada e saída.

O MS, apresentado na Figura 28 é um módulo autocontido responsável por concentrar todas as responsabilidades e regras de negócio associadas ao processo de simulação. Esse

módulo executa a dinâmica da simulação propriamente dita, abrangendo a implementação dos modelos matemáticos, o processamento dos sinais de entrada e saída, a atualização dos estados internos do sistema e a comunicação com o sistema de automação responsável pela interface com os equipamentos físicos. Dessa forma, o MS representa o núcleo funcional da simulação, no qual são definidos e executados os comportamentos dinâmicos que caracterizam o sistema modelado.

Uma característica fundamental do MS é a possibilidade de execução independente da estação do instrutor. Essa propriedade permite que todo o processo de simulação seja reproduzido de forma isolada, favorecendo atividades de Verificação, Validação e Testes (VV&T). Em particular, esse isolamento possibilita a avaliação de aspectos críticos, como a otimização do tempo de simulação, o desempenho dos modelos matemáticos, o tempo de resposta da comunicação com o sistema físico e a coerência das dinâmicas implementadas, sem a sobrecarga computacional introduzida por módulos adicionais de interface gráfica, gerenciamento de usuários ou serviços auxiliares.

Do ponto de vista arquitetural, o MS provê uma infraestrutura de software para o suporte à modelagem matemática, à comunicação com o sistema de automação e ao gerenciamento de subsistemas complementares, como sinais visuais e sonoros. A lógica de modelagem é organizada de forma modular, de modo que os modelos matemáticos e as regras associadas à dinâmica da simulação sejam desacoplados das demais preocupações de engenharia de software. Os dados de entrada e saída da simulação são estruturados por meio de componentes específicos, que refletem continuamente o estado corrente do sistema simulado e funcionam como uma interface programática para acesso externo.

A interação entre o MS e o sistema físico ocorre por meio de um componente dedicado à comunicação, responsável por realizar operações síncronas e assíncronas. Na comunicação síncrona, a troca de dados ocorre a cada iteração do ciclo de simulação, assegurando a atualização em tempo real dos estados do sistema. Na comunicação assíncrona, por sua vez, a transmissão de informações é acionada sob demanda, apenas quando há alterações relevantes nos estados monitorados. Essa flexibilidade permite adaptar o comportamento do simulador a diferentes requisitos operacionais, equilibrando fidelidade da simulação, consumo de recursos computacionais e desempenho global do sistema.

Em termos de implementação, o Módulo de Simulação foi desenvolvido utilizando a linguagem C++, escolhida por seu alto desempenho e controle eficiente de recursos, o que a torna adequada para aplicações que exigem processamento em tempo real e operações numéricas intensivas. Por outro lado, o EOI foi implementada em C#, com o apoio

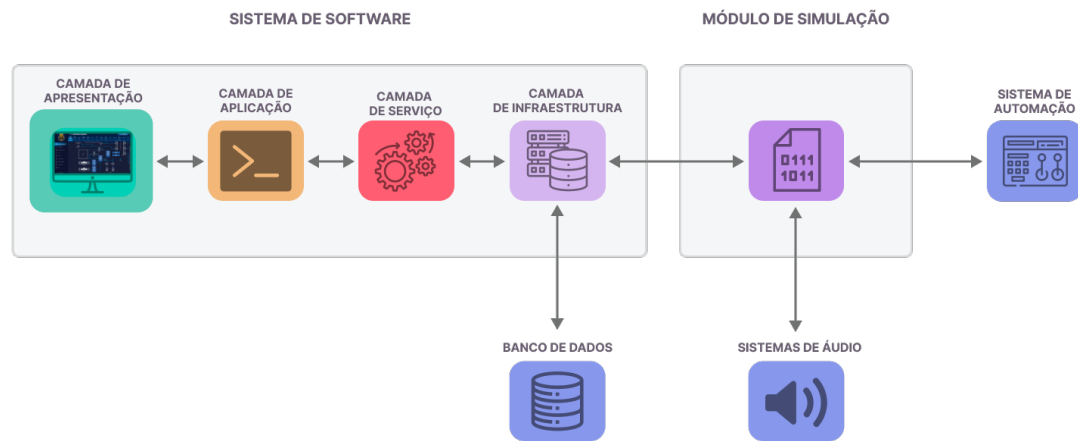


Figura 28: Framework TupiSIM implementado no TI.

do [Windows Presentation Foundation \(WPF\)](#), que proporcionou uma arquitetura moderna e flexível para a construção de interfaces gráficas responsivas, seguras e integradas aos demais serviços do sistema. Essa combinação de tecnologias favorece a modularidade e a interoperabilidade entre os componentes, garantindo a separação clara das responsabilidades entre as atividades de modelagem e simulação e as funções de engenharia de software, como gestão de banco de dados, controle de usuários e registro de logs de exercícios.

Apesar da modularidade estrutural do TupiSIM, a arquitetura apresenta um acoplamento lógico em relação à comunicação com o sistema de automação, uma vez que a camada de regras de negócio do Módulo de Simulação encontra-se diretamente vinculada às tecnologias de comunicação empregadas. Essa integração dificulta a manutenção e a substituição de protocolos, além de comprometer a independência entre a modelagem e a infraestrutura de comunicação. Nesse contexto, reforça-se a necessidade de introdução de um middleware como camada intermediária de abstração, responsável por desacoplar as regras de negócio das tecnologias de comunicação, assegurando maior flexibilidade, escalabilidade e portabilidade à arquitetura, sem incorrer em aumento significativo de *overhead* de comunicação.

### 5.1.3 Estrutura Física do Simulador

A estrutura física do TI é composta por um conjunto de controladores, sensores, atuadores e painéis de comando interligados a um [Controlador Lógico Programável \(CLP\)](#). O CLP é responsável por gerenciar os sinais de entrada e saída entre os elementos físicos e o ambiente virtual da simulação. Ele garante a coerência entre os estados lógicos do sistema físico e os parâmetros reproduzidos no ambiente digital.

A comunicação entre o Módulo de Simulação e o Sistema de Automação é realizada sobre a arquitetura [TCP/IP](#), utilizando o protocolo OPC DA Classic<sup>2</sup>. Essa camada de comunicação é responsável por disponibilizar os dados lógicos e analógicos utilizados durante a simulação, estabelecendo a troca bidirecional de informações entre o simulador virtual e o sistema físico.

A Tabela 6 apresenta o conjunto de *tags* implementado no [TI](#):

Tabela 6: Distribuição das tags implementadas no [TI](#).

| Operação           | Digitais | Analógicas | Total |
|--------------------|----------|------------|-------|
| Leitura            | 143      | 85         | 228   |
| Escrita            | 177      | 78         | 255   |
| <b>Total Geral</b> | 320      | 163        | 483   |

Essas *tags* representam estados e variáveis de instrumentos físicos, como válvulas, tanques, motores e alarmes, que são continuamente atualizados durante o ciclo de simulação. O processo de leitura e escrita é executado de forma cíclica, garantindo a sincronização entre o comportamento físico e a representação digital do sistema.

Para a versão experimental desenvolvida nesta tese, a comunicação foi modernizada para o protocolo [OPC UA](#). Essa migração proporcionou maior alinhamento com as práticas contemporâneas adotadas em sistemas de [DT](#) e em arquiteturas de [Industrial Internet of Things \(IIoT\)](#) ([OPC Foundation, 2017b](#)).

## 5.2 Reprodução do Cenário Experimental

Com a finalidade validar a arquitetura de middleware proposta, o ambiente do [TI](#) foi reproduzido experimentalmente. Essa reprodução envolveu a recriação da rede, das *tags* e das interações do simulador real, o que permitiu a realização de testes em ambiente controlado de laboratório controlados.

Para fins de coerência com a literatura e compatibilidade com a arquitetura de middleware, adotou-se o protocolo [OPC UA](#) como interface de comunicação principal, substituindo o OPC DA Classic do projeto original.

<sup>2</sup><https://opcfoundation.org/about/opc-technologies/opc-classic/>

5.2.1 Materiais e Equipamentos Utilizados

A Tabela 7 e Figura 29 apresenta a os equipamentos que foram utilizados para compor a configuração experimental.

| Equipamento                                                              | Descrição e Função                                                                                                 |
|--------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| Desktop Switch TP-Link LS1005G Litewave (5 portas Gigabit)               | Empregado na interconexão dos dispositivos de rede do ambiente experimental.                                       |
| Raspberry Pi 4 Model B (4 GB RAM)                                        | Configurado como servidor <a href="#">OPC UA</a> e responsável por emular o comportamento do <a href="#">CLP</a> . |
| Adaptador UGREEN USB-C para Ethernet (Thunderbolt 3 → RJ-45 Gigabit LAN) | Utilizado para possibilitar a conexão do RJ-45 com conector USB sem impactos negativos na comunicação.             |
| Cabo de Rede Categoria 6 (2 m)                                           | Utilizado para garantir baixa latência e mínima perda de sinal na comunicação cabeada.                             |
| MacBook Air M4 (16 GB RAM, macOS Sequoia 15.2)                           | Utilizado como estação principal para execução do <i>Middleware</i> e monitoramento do ambiente experimental.      |

Tabela 7: Materiais e equipamentos utilizados na configuração experimental.

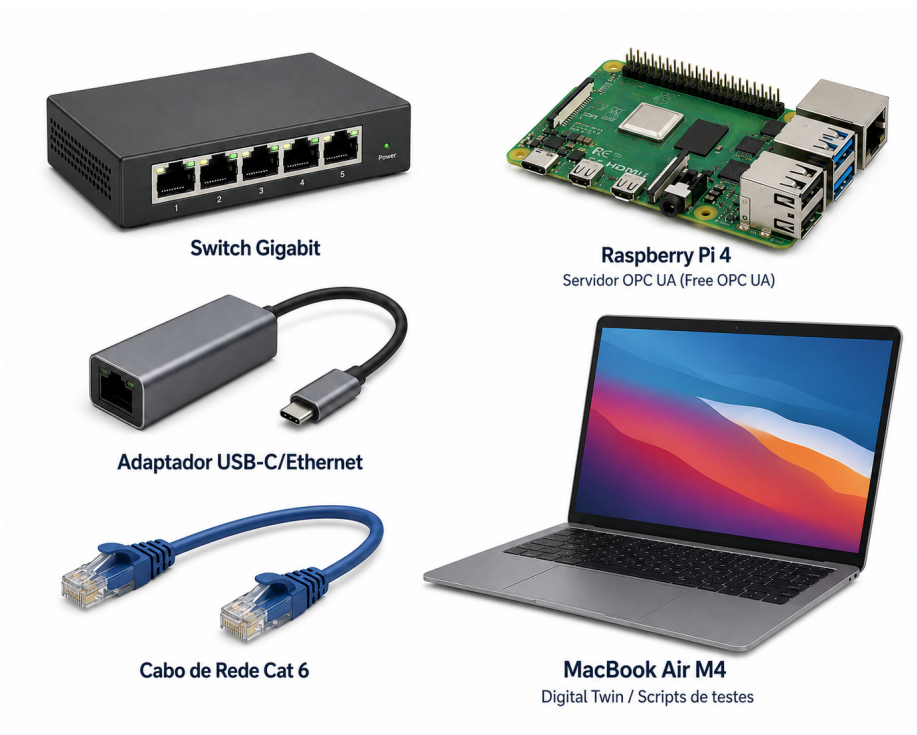


Figura 29: Equipamentos utilizados na bancada de testes.

Essa infraestrutura compacta possibilitou reproduzir de forma realista o fluxo de comunicação entre o simulador virtual e o ambiente físico, respeitando as restrições de tempo e sincronização necessárias à validação do sistema. O arranjo de componentes foi projetado de modo a permitir medições controladas de desempenho, latência e interoperabilidade entre o middleware, o servidor OPC UA e os dispositivos conectados à rede.

Com o objetivo de validar a comunicação entre o middleware e um ambiente físico representativo, foi implementado um servidor OPC UA que emulou o comportamento do PT, no contexto do TI. Esse servidor atuou como ponto de integração responsável por disponibilizar, em tempo real, as variáveis e estados simulados do sistema físico (PT).

A emulação do servidor Open Platform Communications Unified Architecture (OPC UA) foi desenvolvida na linguagem Python 3.13, utilizando a biblioteca *FreeOPCUA*<sup>3</sup> — um conjunto de ferramentas de código aberto amplamente adotado para criação de servidores e clientes OPC UA compatíveis com o padrão IEC 62541. Essa biblioteca permitiu a definição de um espaço de endereçamento estruturado (*Address Space*) contendo nós correspondentes às *tags* analógicas e digitais do sistema. Essas *tags* foram modeladas seguindo o mesmo padrão de nomenclatura e estrutura utilizado no sistema real do TI, de modo a representar um cenário experimental equivalente ao empregado no ambiente operacional. As variáveis analógicas foram associadas a grandezas contínuas (por exemplo, nível de tanque e rotação de motor), enquanto as variáveis digitais representaram estados binários (como acionamentos e indicadores de falha). A Figura 30 exemplifica como as *tags* são representadas no sistema emulado.

Essa abordagem possibilitou reproduzir, de forma controlada, o comportamento do sistema físico dentro de um ambiente emulado de testes, permitindo a mensuração da latência de comunicação das operações de leitura e escrita, e a avaliação da interoperabilidade entre o middleware e o servidor OPC UA. Embora o ambiente físico, por exemplo, a válvula, não esteja fisicamente implementado (potenciômetro, cabos e afins), o formato de comunicação física entre o PT e DT segue o mesmo mecanismo, desde o formato de envio da mensagem até o mecanismo físico de transmissão da mensagem (protocolos, switch e cabos de rede, por exemplo).

Para visualização e acompanhamento das alterações e atualizações das tags, foi utilizado um cliente OPC UA UaExpert<sup>4</sup>. A Figura 31 exibe a lista de tags dentro do cliente.

---

<sup>3</sup><https://github.com/FreeOpcUa/python-opcua>

<sup>4</sup><https://www.unified-automation.com/products/development-tools/uaexpert.html>

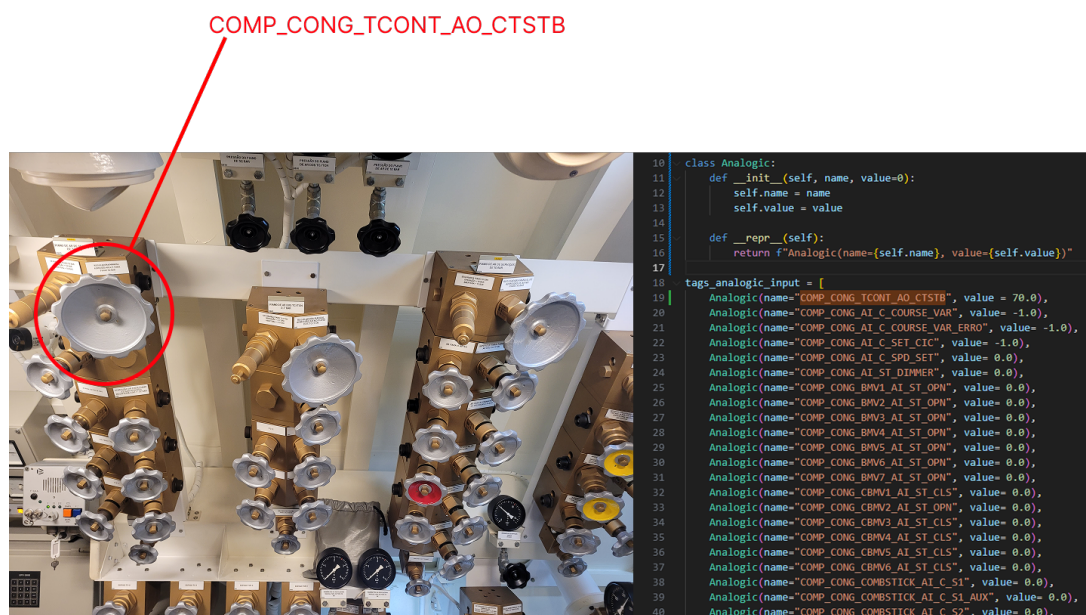


Figura 30: Physical Twin (PT) emulado para o TI.

Para monitoramento e depuração da transmissão das mensagens do *middleware*, foi utilizado o software Wireshark<sup>5</sup>, uma ferramenta de código aberto amplamente empregada para captura e análise de pacotes em redes TCP/IP. Essa inspeção permitiu examinar, de forma detalhada, o conteúdo dos pacotes transmitidos, seus tamanhos, a ordem de envio.

Para a implementação do servidor OPC UA responsável pela emulação do PT, foi utilizado um Raspberry Pi 4 Model B (4 GB RAM) (Figura 29), executando o sistema operacional *Raspberry Pi OS (64-bit) - Raspbian GNU/Linux 12 (bookworm)*. Essa plataforma foi escolhida por oferecer suporte à biblioteca FreeOPCUA e capacidade de processamento suficiente para executar, de forma estável, a lógica de publicação e atualização das *tags* em tempo real. O processo completo de preparação, instalação do sistema operacional, configuração de rede e implantação do servidor OPC UA está detalhado no Apêndice C.

<sup>5</sup><https://www.wireshark.org/>

The screenshot displays the UaExpert software interface. The 'Data Access View' pane shows a list of tags with the following columns: #, Server, Node Id, Display Name, Value, Datatype, Source Timestamp, and Server Time. The tags are organized by server (OPC UA Server@127.0.0.1) and node ID.

| #  | Server                  | Node Id        | Display Name   | Value | Datatype | Source Timestamp | Server Time |
|----|-------------------------|----------------|----------------|-------|----------|------------------|-------------|
| 1  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_ASP_DO... | false | Boolean  | 16:20:17.221     | 21:00:00    |
| 2  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.194     | 21:00:00    |
| 3  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.195     | 21:00:00    |
| 4  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.194     | 21:00:00    |
| 5  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.221     | 21:00:00    |
| 6  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.221     | 21:00:00    |
| 7  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.222     | 21:00:00    |
| 8  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.222     | 21:00:00    |
| 9  | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.222     | 21:00:00    |
| 10 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.222     | 21:00:00    |
| 11 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.223     | 21:00:00    |
| 12 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.223     | 21:00:00    |
| 13 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.223     | 21:00:00    |
| 14 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.223     | 21:00:00    |
| 15 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.222     | 21:00:00    |
| 16 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_APANEL... | false | Boolean  | 16:20:17.222     | 21:00:00    |
| 17 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_Automa... | false | Boolean  | 16:20:17.216     | 21:00:00    |
| 18 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_BAG_DI... | false | Boolean  | 16:20:17.195     | 21:00:00    |
| 19 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONGO...  | false | Boolean  | 16:20:17.259     | 21:00:00    |
| 20 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONGO...  | false | Boolean  | 16:20:17.215     | 21:00:00    |
| 21 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.223     | 21:00:00    |
| 22 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.223     | 21:00:00    |
| 23 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.224     | 21:00:00    |
| 24 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.224     | 21:00:00    |
| 25 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.224     | 21:00:00    |
| 26 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.226     | 21:00:00    |
| 27 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | 0     | Double   | 16:20:17.194     | 21:00:00    |
| 28 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | -1    | Double   | 16:20:17.174     | 21:00:00    |
| 29 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | -1    | Double   | 16:20:17.174     | 21:00:00    |
| 30 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | -1    | Double   | 16:20:17.174     | 21:00:00    |
| 31 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | 0     | Double   | 16:20:17.174     | 21:00:00    |
| 32 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | 0     | Double   | 16:20:17.175     | 21:00:00    |
| 33 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.195     | 21:00:00    |
| 34 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.195     | 21:00:00    |
| 35 | OPC UA Server@127.0.0.1 | NS0\$String... | COMP_CONG...   | false | Boolean  | 16:20:17.226     | 21:00:00    |

The right-hand pane shows the 'Attributes' section for the selected tag, displaying various properties such as NameSpaceIndex, IdentifierType, NodeClass, BrowseName, DisplayName, and Value. The 'References' section at the bottom shows the tag's relationships with other tags in the system.

Figura 31: Lista de tags no software UaExpert.

## 6 Avaliação Experimental

Neste Capítulo é apresentada a avaliação experimental da arquitetura de middleware proposta neste trabalho. A avaliação foi conduzida por meio da execução de testes controlados, considerando diferentes topologias de implantação, arquiteturas de interface e configurações de comunicação. O objetivo, portanto, foi analisar o comportamento do middleware sob a perspectiva de desempenho, bem como discutir a viabilidade de aplicação em um cenário representativo de sistema de [Digital Twin \(DT\)](#), apresentado na [Capítulo 5](#).

Em relação à avaliação de desempenho, os experimentos foram organizados em duas topologias distintas. Na primeira, referente a um ambiente de execução local, o [PT](#), o middleware e o [DT](#) foram executados no mesmo *host*. Na segunda, correspondente a um ambiente distribuído em rede, o [PT](#) foi executado em uma nó dedicado, enquanto o middleware e o [DT](#) permaneceram alocados no mesmo *host*. Essa organização permitiu avaliar tanto o *overhead* introduzido pelo middleware quanto os efeitos da comunicação em rede sobre o desempenho do sistema, conforme discutido em trabalhos correlatos ([Rizano; Abeni; Palopoli, 2013](#); [Borsatti et al., 2020](#); [Liang; Yuan; Lin, 2023](#); [Hietala et al., 2020b](#)).

Em relação à avaliação da arquitetura proposta, foi definido um cenário experimental de alteração do protocolo de comunicação utilizado pelo [DT](#), com a finalidade de representar uma modificação significativa na estrutura de comunicação. Essa análise foi conduzida com o objetivo de avaliar o impacto dessa alteração sobre o middleware e, consequentemente, verificar em que medida a arquitetura proposta é capaz de acomodar mudanças tecnológicas no [DT](#) sem exigir modificações substanciais nos demais elementos do sistema.

### 6.1 Ambiente Experimental para Avaliação de Desempenho

Esta Seção descreve a metodologia e a configuração do ambiente para a análise de desempenho. Portanto, são detalhadas as topologias implementadas, quais variáveis foram utilizadas para análise de desempenho e quais métricas foram coletadas.

### 6.1.1 Topologia Local

Na topologia local, o **PT**, o middleware e o **DT** são implantados e executados no mesmo *host*. Nesse sentido, o fluxo de comunicação entre os componentes ocorre sem a influência de atrasos introduzidos pela rede física. Essa configuração foi selecionada com a finalidade de isolar o *overhead* associados aos mecanismos de rede (conforme apresentado na Seção 2.2) do *overhead* introduzido pelo middleware.

A Figura 32 ilustra a topologia local adotada nos experimentos. Nessa configuração, todos os componentes do sistema — **DT**, **Physical Twin (PT)** e middleware — são executados no mesmo *host* (Macbook Air M4), utilizando o endereço de *loopback* (127.0.0.1). Cada componente é executado em um processo distinto, identificado por um **Process Identifier (PID)** próprio. O processo correspondente ao servidor OPC UA, responsável por emular as *tags* de leitura e escrita, permanece disponível por meio do *endpoint* 0.0.0.0:4048. O middleware, por sua vez, foi disponibilizado por meio das interfaces TCP (127.0.0.1:5001), UDP (127.0.0.1:5001), REST (<http://127.0.0.1:8000>) e GraphQL (<http://127.0.0.1:8001/graphql>). Por fim, o processo correspondente ao Digital Twin, implementado por meio dos scripts de teste, executa a bateria de testes definida para avaliar o desempenho do middleware.

O Apêndice C, detalha o procedimento de configuração e inicialização do middleware.

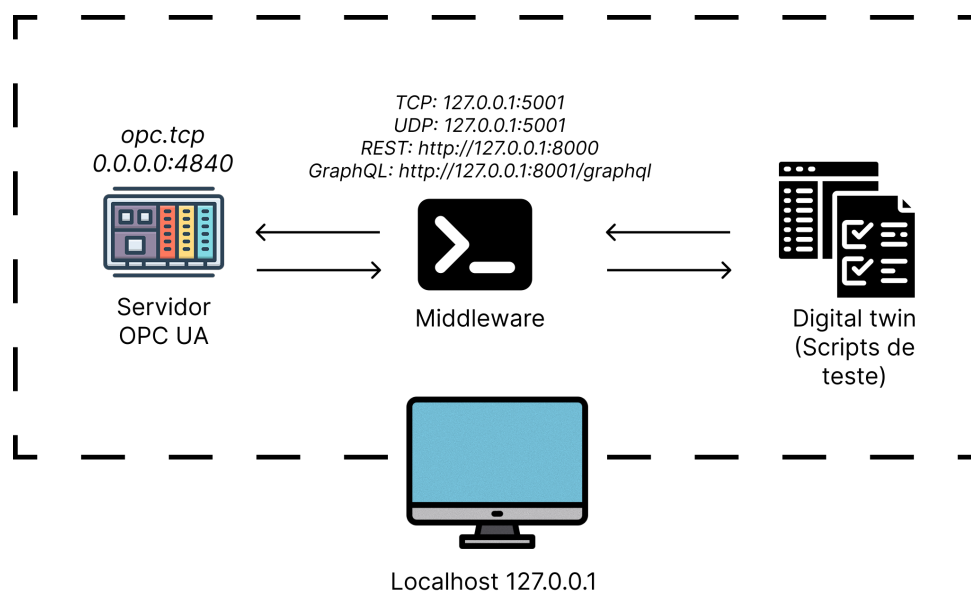


Figura 32: Topologia local para testes.

### 6.1.2 Topologia Distribuída

Na topologia distribuída, o processo *Servidor OPC UA* é executado em um nó dedicado, enquanto o processos *Middleware* e o *Digital Twin (script de testes)* são executados em um segundo nó, ambos interconectados por meio de uma rede local cabeada, utilizando um Switch Gigabit. A finalidade da implementação dessa topologia foi aproximar a avaliação experimental, realizada neste trabalho, ao estudo de caso apresentado no Capítulo 5.

A Figura 33 ilustra a topologia distribuída. Nessa configuração, o processo *Servidor OPC UA* é executado em um dispositivo embarcado (Raspberry Pi), responsável por disponibilizar as variáveis do processo físico (dados analógicos e digitais) por meio do protocolo de comunicação adotado. Os processos *Middleware* e *Digital Twin (Scripts de Testes)* são executados no mesmo *host* (Macbook Air M4). De forma semelhante à topologia local, o middleware foi disponibilizado por meio das interfaces TCP (127.0.0.1:5001), UDP (127.0.0.1:5001), REST (<http://127.0.0.1:8000>) e GraphQL (<http://127.0.0.1:8001/graphql>).

A utilização de um switch Gigabit como único elemento de interconexão teve como finalidade estabelecer um ambiente de testes controlado, no qual os atrasos introduzidos pela rede são restritos à latência da rede local e ao processamento dos próprios dispositivos envolvidos. Dessa forma, foi reduzida a influência de fatores externos, como roteamento dinâmico, congestionamento em redes externas e variações associadas à comunicação via internet.

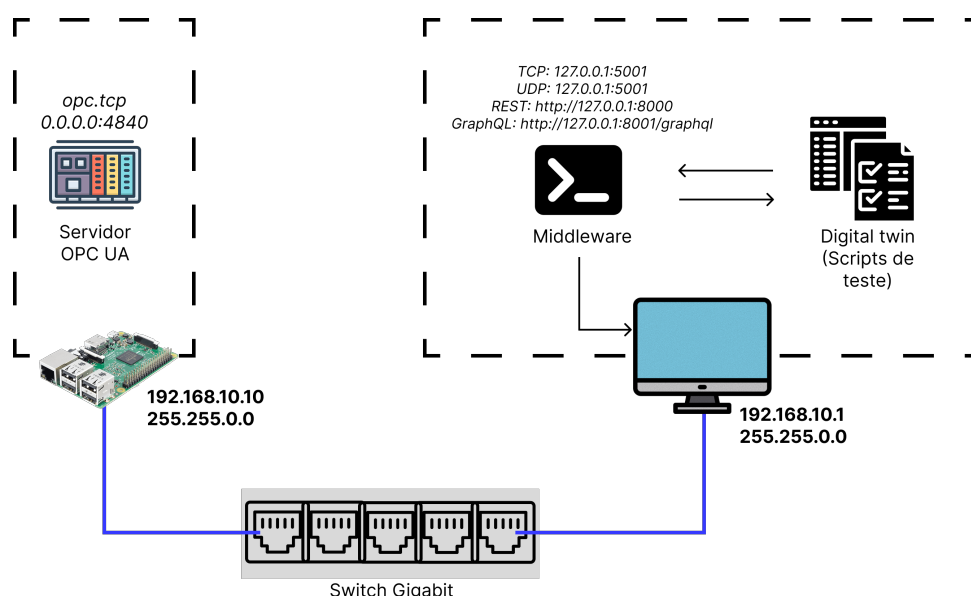


Figura 33: Topologia distribuída para testes.

### 6.1.3 Variáveis Controladas

Com o objetivo de garantir a reprodutibilidade dos experimentos e permitir uma comparação consistente entre os diferentes cenários avaliados, os testes de desempenho foram planejados a partir do controle sistemático de variáveis experimentais, com base nos princípios de Design de Experimentos ([<empty citation>](#)). Complementarmente, os testes foram desenhados com base nos requisitos estabelecidos pelo [Simulador de Treinamento de Imersão \(TI\)](#).

Em cada uma das topologias (local e distribuída), foram realizados diferentes cenários de testes para dados analógicos e digitais, considerando operações de leitura e escrita. Para ambos os tipos de dados, a avaliação contemplou variações na quantidade de mensagens trocadas e tamanho das mensagens transmitidas. A definição da quantidade de mensagens avaliadas foi orientada pela quantidade total – analógicas/digitais de escrita e leitura – utilizadas pelo [TI](#), conforme exibido na Tabela 6 do Capítulo 5. Dessa forma, os valores considerados nos experimentos buscaram representar cargas compatíveis com o cenário real da aplicação. Em relação ao tamanho das mensagens, foram considerados pacotes de 300 *bytes*, 700 *bytes* e 1000 *bytes*. Esses tamanhos foram definidos de modo a permanecer abaixo do limite definido pelo [Maximum Transmission Unit \(MTU\)](#) em redes Ethernet, evitando a fragmentação dos pacotes no nível de rede. Com essa escolha, buscou-se avaliar o impacto do volume de dados transmitidos sobre o desempenho da comunicação sem introduzir interferências adicionais decorrentes de segmentação ou remontagem de pacotes.

Para cada um dos cenários avaliados foram considerados diferentes arquiteturas de interface para acesso ao middleware ([Transmission Control Protocol \(TCP\)](#), [User Datagram Protocol \(UDP\)](#), [Representational State Transfer \(REST\)](#) e [Graph Query Language \(GraphQL\)](#)). No entanto, no que se refere à comunicação entre o [PT](#) e o middleware, a análise foi concentrada no protocolo [Open Platform Communications Unified Architecture \(OPC UA\)](#), visto que este foi o protocolo utilizado pelo [TI](#).

A Tabela 8 apresenta o conjunto de variáveis controladas nos experimentos e os respectivos valores avaliados.

Para melhor entendimento, a Figura 34 ilustra a organização utilizada para os testes.

Para cada configuração experimental, os testes de desempenho foram realizados com 100 execuções independentes, com a finalidade de obter métricas estatísticas representativas, como média e desvio padrão.

Tabela 8: Variáveis controladas nos experimentos.

| Variável                                 | Valores Avaliados       |
|------------------------------------------|-------------------------|
| Tipo de dado                             | Analógico, Digital      |
| Operação                                 | Read, Write             |
| Quantidade de mensagens                  | 1, 10, 100, 300, 500    |
| Tamanho da mensagem                      | 300 B, 700 B, 1000 B    |
| Arquitetura de interface                 | REST, GraphQL, TCP, UDP |
| Protocolo <a href="#">PT</a> -Middleware | OPC UA                  |

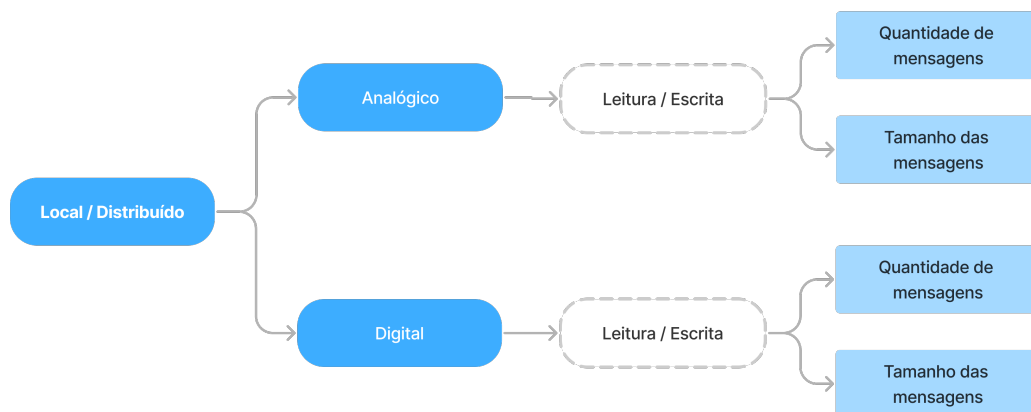


Figura 34: Organização dos cenários de testes.

### 6.1.4 Avaliação de Desempenho em Topologia Local

Essa Seção apresenta os resultados e discussão sobre os resultados dos experimentos realizados na Topologia local.

#### 6.1.4.1 Variando a quantidade de mensagens enviadas - leitura e escrita com dados analógicos

A seguir são apresentados os resultados experimentais referentes às operações de leitura de dados analógicos em topologia local, considerando diferentes arquiteturas de interface para acesso ao middleware, incluindo [TCP](#), [UDP](#), [REST](#) e [GraphQL](#).

A Figura [35](#) e Figura [36](#) apresentam os resultados das operações de leitura e escrita dos dados analógicos. Os resultados apresentam o impacto da variação da quantidade de mensagens requisitadas pelo [DT](#) (1, 10, 100, 300 e 500 *tags*).

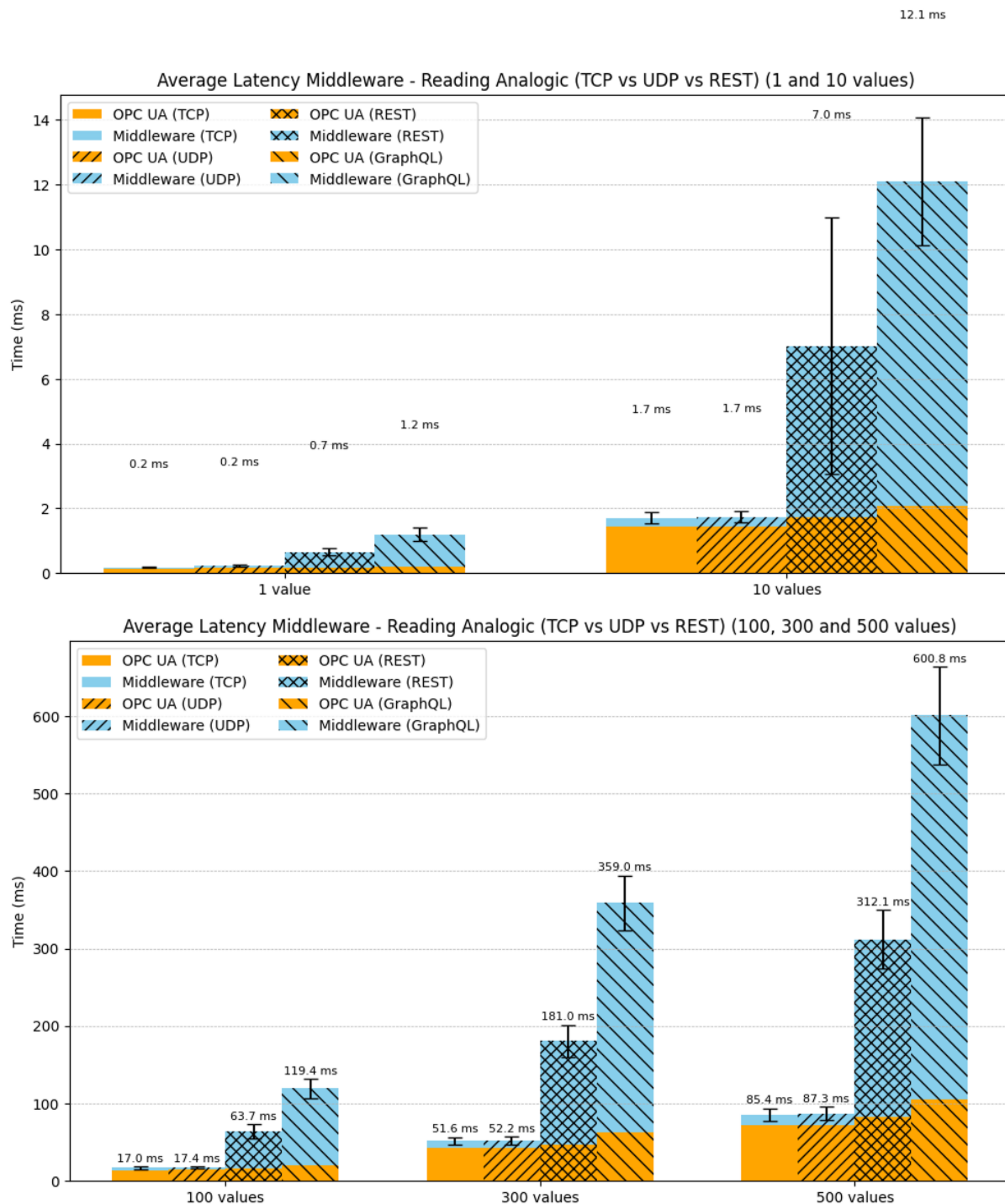


Figura 35: Operação de Leitura com Dados Analógicos na Topologia Local.

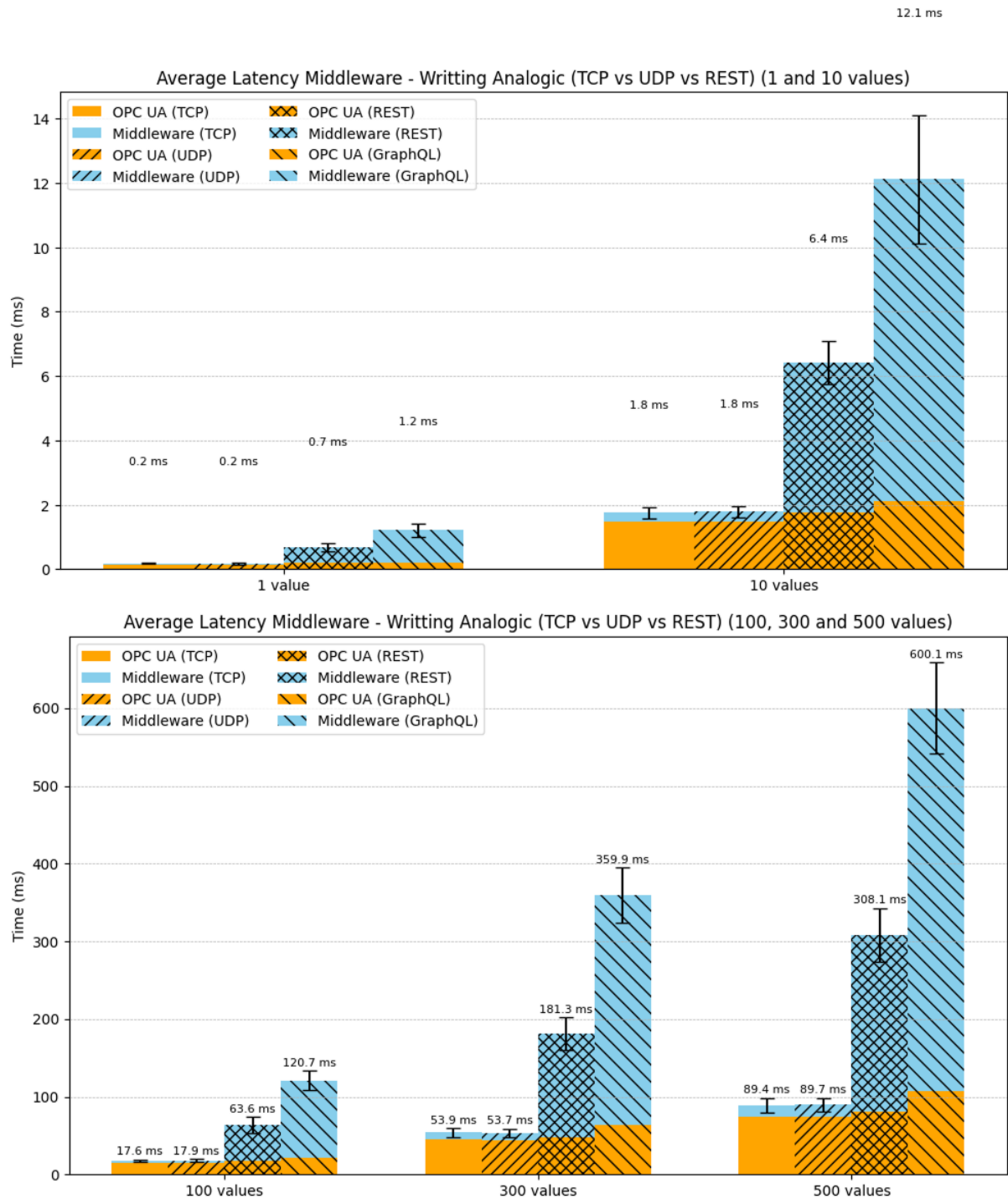


Figura 36: Operação de Escrita com Dados Analógicos na Topologia Local.

Tabela 9: Latência média e desvio padrão para leitura de dados analógicos em topologia local.

| <b>Protocolo</b> | <b>Qtd. Valores</b> | <b>Média (ms)</b> | <b>Desvio Padrão (ms)</b> | <b>OPC UA (ms)</b> |
|------------------|---------------------|-------------------|---------------------------|--------------------|
| TCP              | 1                   | 0.17              | 0.02                      | 0.15               |
| TCP              | 10                  | 1.71              | 0.17                      | 1.44               |
| TCP              | 100                 | 17.03             | 1.67                      | 14.30              |
| TCP              | 300                 | 51.61             | 5.07                      | 43.33              |
| TCP              | 500                 | 85.39             | 8.39                      | 71.81              |
| UDP              | 1                   | 0.24              | 0.03                      | 0.17               |
| UDP              | 10                  | 1.74              | 0.17                      | 1.44               |
| UDP              | 100                 | 17.41             | 1.71                      | 14.40              |
| UDP              | 300                 | 52.21             | 5.13                      | 43.20              |
| UDP              | 500                 | 87.26             | 8.58                      | 72.23              |
| REST             | 1                   | 0.66              | 0.10                      | 0.17               |
| REST             | 10                  | 7.02              | 3.96                      | 1.74               |
| REST             | 100                 | 63.75             | 9.48                      | 16.81              |
| REST             | 300                 | 181.02            | 20.75                     | 47.25              |
| REST             | 500                 | 312.12            | 38.26                     | 82.49              |
| GraphQL          | 1                   | 1.19              | 0.20                      | 0.21               |
| GraphQL          | 10                  | 12.09             | 1.98                      | 2.07               |
| GraphQL          | 100                 | 119.44            | 12.39                     | 20.81              |
| GraphQL          | 300                 | 358.98            | 35.37                     | 62.51              |
| GraphQL          | 500                 | 600.85            | 62.74                     | 105.11             |

Tabela 10: Latência média e desvio padrão para escrita de dados analógicos em topologia local.

| Protocolo | Qtd. Valores | Média (ms) | Desvio Padrão (ms) | OPC UA (ms) |
|-----------|--------------|------------|--------------------|-------------|
| TCP       | 1            | 0.18       | 0.02               | 0.15        |
| TCP       | 10           | 1.76       | 0.17               | 1.47        |
| TCP       | 100          | 17.62      | 1.73               | 14.74       |
| TCP       | 300          | 53.86      | 5.95               | 44.92       |
| TCP       | 500          | 89.38      | 9.25               | 74.69       |
| UDP       | 1            | 0.17       | 0.02               | 0.14        |
| UDP       | 10           | 1.79       | 0.18               | 1.47        |
| UDP       | 100          | 17.90      | 1.76               | 14.74       |
| UDP       | 300          | 53.72      | 5.28               | 44.21       |
| UDP       | 500          | 89.74      | 8.83               | 73.92       |
| REST      | 1            | 0.68       | 0.11               | 0.19        |
| REST      | 10           | 6.42       | 0.67               | 1.78        |
| REST      | 100          | 63.61      | 10.66              | 16.91       |
| REST      | 300          | 181.35     | 20.88              | 47.86       |
| REST      | 500          | 308.11     | 34.89              | 81.34       |
| GraphQL   | 1            | 1.21       | 0.22               | 0.21        |
| GraphQL   | 10           | 12.12      | 2.00               | 2.10        |
| GraphQL   | 100          | 120.70     | 12.49              | 21.28       |
| GraphQL   | 300          | 359.90     | 35.45              | 64.31       |
| GraphQL   | 500          | 600.13     | 58.90              | 106.87      |

**Análise dos Resultados** Com base na análise dos resultados, é possível observar que as operações de leitura e escrita apresentam comportamentos equivalentes em termos de latência média para todas as arquiteturas avaliadas. Essa similaridade evidencia que, em topologia local, o desempenho é predominantemente determinado pelo custo de comunicação e serialização dos dados, sendo pouco influenciado pela semântica específica da operação, isto é, se foi uma operação de leitura ou escrita.

Os resultados mostram que todas as arquiteturas avaliadas apresentam aumento da latência com o crescimento da quantidade de valores analógicos processados. Porém, houve diferença na intensidade do crescimento da latência. Essa observação permitiu identificar diferenças importantes de escalabilidade entre as abordagens consideradas.

Para quantidade de operações menores, correspondentes a 1 e 10 valores, as arquiteturas baseadas em [TCP](#) e [UDP](#) apresentam latências muito próximas às obtidas na comunicação direta via [OPC UA](#), com valores inferiores a 2 ms tanto para leitura quanto para escrita. Nesses cenários, o *overhead* introduzido pelo middleware ficou na faixa de 15% a 20% (tanto para leitura quanto escrita), indicando que o custo adicional da camada intermediária é praticamente constante e desprezível para pequenas quantidades de dados.

À medida que a quantidade de valores aumenta para 100, 300 e 500, observa-se um crescimento aproximadamente linear da latência para todas as arquiteturas. Esse comportamento é evidente nas abordagens baseadas em sockets (baixo nível), [TCP](#) e [UDP](#), nas quais a latência cresce de forma aproximadamente proporcional ao volume de dados processados. No caso da leitura, por exemplo, para 500 valores, as latências médias atingem aproximadamente 85,39 ms em [TCP](#) e 87,26 ms em [UDP](#), enquanto a comunicação direta via [OPC UA](#) apresenta cerca de 71,81 ms. De forma análoga, na operação de escrita, os valores observados foram de 89,38 ms para [TCP](#), 89,74 ms para [UDP](#) e 74,69 ms para [OPC UA](#). Esses resultados correspondem a um *overhead* adicional da ordem de 15% a 20%, demonstrando que o middleware proposto apresenta comportamento escalável, previsível e sem indícios de saturação no intervalo avaliado.

Em contraste, as arquiteturas baseadas em Web, [REST](#) e [GraphQL](#), apresentam crescimento significativamente mais acentuado da latência com o aumento da carga. Para 500 valores analógicos, na operação de leitura, a latência média atinge aproximadamente 312,12 ms para [REST](#) e 600,85 ms para [GraphQL](#), enquanto, na escrita, são observados aproximadamente 308,11 ms e 600,13 ms, respectivamente. Em relação à comunicação direta via [OPC UA](#), esses valores correspondem, na leitura, a aumentos de aproximadamente 278,4% para [REST](#) e 471,7% para [GraphQL](#); na escrita, os aumentos observados

são de aproximadamente 278,8% e 461,6%, respectivamente. Em outras palavras, essa análise indica que a latência do [REST](#) é cerca de 3,8 vezes a do [OPC UA](#), enquanto a do [GraphQL](#) é aproximadamente 5,7 vezes maior. Esse comportamento evidencia que o custo associado à serialização textual, ao encapsulamento via [HTTP](#) e ao processamento adicional dessas arquiteturas torna-se dominante em cenários com maior volume de dados.

Destaca-se que a arquitetura baseada em [GraphQL](#) apresenta consistentemente os maiores valores de latência entre todas as abordagens analisadas. Esse resultado pode ser atribuído ao custo adicional relacionado ao processamento das consultas e à construção dinâmica das respostas. A arquitetura [REST](#), embora mais eficiente que [GraphQL](#), ainda apresenta *overhead* significativo quando comparada às abordagens baseadas na proposta desta tese.

#### **6.1.4.2 Variando a quantidade de mensagens enviadas – leitura e escrita com dados digitais**

A seguir são apresentados os resultados experimentais referentes às operações de leitura e escrita de dados digitais em topologia local, considerando diferentes arquiteturas de interface para acesso ao middleware, incluindo [TCP](#), [UDP](#), [REST](#) e [GraphQL](#). A Figura 37 apresenta os resultados da latência média das operações de leitura dos dados digitais. Por outro lado, a Figura 38 as operações de escrita dos dados digitais.

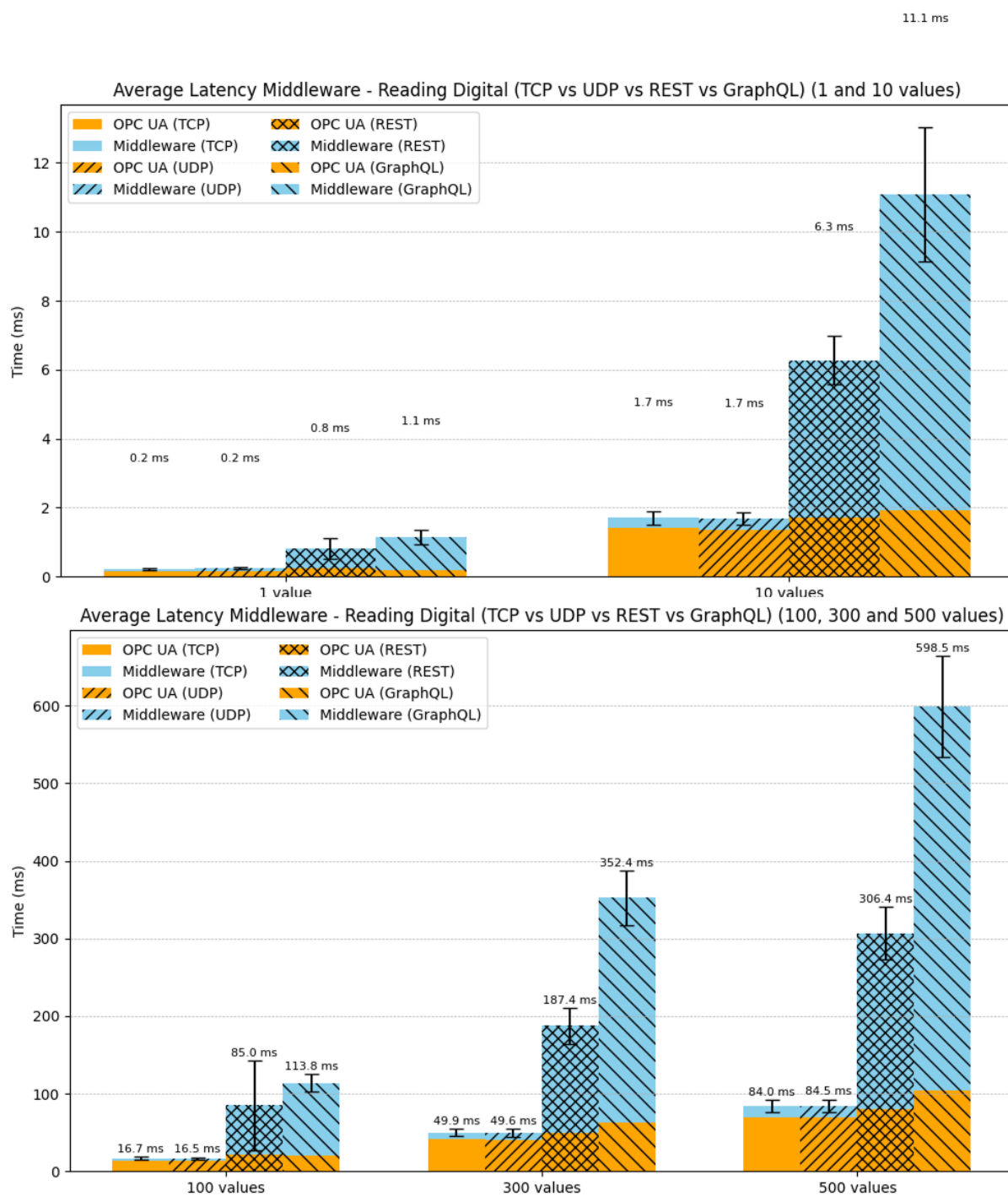


Figura 37: Operação de Leitura com Dados Digitais na Topologia Local.

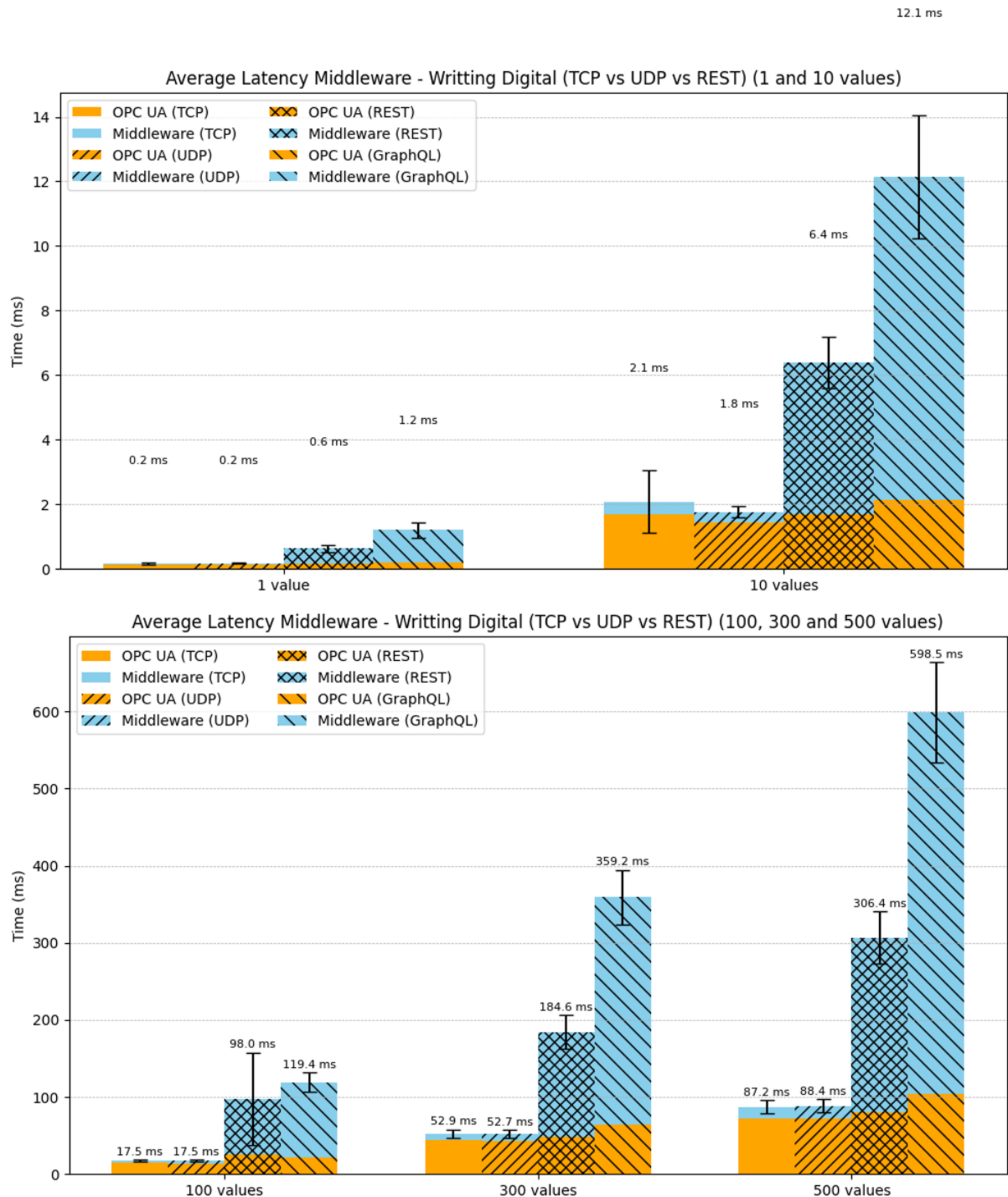


Figura 38: Operação de Escrita com Dados Digitais na Topologia Local.

Tabela 11: Latência média e desvio padrão para leitura de dados digitais em topologia local.

| <b>Protocolo</b> | <b>Qtd. Valores</b> | <b>Média (ms)</b> | <b>Desvio Padrão (ms)</b> | <b>OPC UA (ms)</b> |
|------------------|---------------------|-------------------|---------------------------|--------------------|
| TCP              | 1                   | 0.23              | 0.03                      | 0.17               |
| TCP              | 10                  | 1.71              | 0.19                      | 1.41               |
| TCP              | 100                 | 16.70             | 1.64                      | 13.85              |
| TCP              | 300                 | 49.93             | 4.90                      | 41.45              |
| TCP              | 500                 | 84.04             | 8.25                      | 69.99              |
| UDP              | 1                   | 0.24              | 0.03                      | 0.17               |
| UDP              | 10                  | 1.68              | 0.18                      | 1.37               |
| UDP              | 100                 | 16.50             | 1.62                      | 13.43              |
| UDP              | 300                 | 49.60             | 4.87                      | 40.48              |
| UDP              | 500                 | 84.46             | 8.29                      | 69.51              |
| REST             | 1                   | 0.83              | 0.30                      | 0.24               |
| REST             | 10                  | 6.27              | 0.71                      | 1.70               |
| REST             | 100                 | 85.01             | 57.40                     | 21.94              |
| REST             | 300                 | 187.35            | 23.36                     | 49.38              |
| REST             | 500                 | 306.44            | 33.95                     | 80.46              |
| GraphQL          | 1                   | 1.14              | 0.21                      | 0.20               |
| GraphQL          | 10                  | 11.09             | 1.94                      | 1.92               |
| GraphQL          | 100                 | 113.84            | 11.60                     | 20.31              |
| GraphQL          | 300                 | 352.41            | 35.29                     | 62.87              |
| GraphQL          | 500                 | 598.52            | 65.30                     | 104.03             |

Tabela 12: Latência média e desvio padrão para escrita de dados digitais em topologia local.

| Protocolo | Qtd. Valores | Média (ms) | Desvio Padrão (ms) | OPC UA (ms) |
|-----------|--------------|------------|--------------------|-------------|
| TCP       | 1            | 0.18       | 0.02               | 0.15        |
| TCP       | 10           | 2.08       | 0.97               | 1.70        |
| TCP       | 100          | 17.49      | 1.72               | 14.53       |
| TCP       | 300          | 52.87      | 5.33               | 43.88       |
| TCP       | 500          | 87.24      | 8.57               | 72.59       |
| UDP       | 1            | 0.17       | 0.02               | 0.14        |
| UDP       | 10           | 1.76       | 0.17               | 1.44        |
| UDP       | 100          | 17.50      | 1.72               | 14.31       |
| UDP       | 300          | 52.71      | 5.18               | 43.20       |
| UDP       | 500          | 88.41      | 8.69               | 72.58       |
| REST      | 1            | 0.64       | 0.11               | 0.17        |
| REST      | 10           | 6.39       | 0.80               | 1.68        |
| REST      | 100          | 98.04      | 59.85              | 26.01       |
| REST      | 300          | 184.59     | 21.90              | 48.87       |
| REST      | 500          | 306.44     | 33.95              | 80.46       |
| GraphQL   | 1            | 1.20       | 0.24               | 0.21        |
| GraphQL   | 10           | 12.14      | 1.91               | 2.13        |
| GraphQL   | 100          | 119.38     | 12.07              | 21.30       |
| GraphQL   | 300          | 359.15     | 35.37              | 63.76       |
| GraphQL   | 500          | 598.52     | 65.30              | 104.03      |

**Análise dos Resultados** Os resultados obtidos para leitura de dados digitais em topologia local apresentam comportamento consistente com aquele observado para dados analógicos, evidenciando tendências semelhantes entre as diferentes arquiteturas avaliadas.

De forma geral, observa-se que as arquiteturas baseadas em [TCP](#) e [UDP](#) apresentam latências muito próximas à comunicação direta via [OPC UA](#) para todas as quantidades de valores analisadas. Para cargas reduzidas (1 e 10 valores), as latências permanecem inferiores a 2 ms, com *overhead* tipicamente na faixa de 15% a 20%. Esse resultado indica que o custo adicional introduzido pelo middleware é baixo e aproximadamente constante nesses cenários.

À medida que a quantidade de valores aumenta, observa-se um crescimento aproximadamente linear da latência para as arquiteturas baseadas em sockets. Para 500 valores, por exemplo, a latência média atinge aproximadamente 84,04 ms para [TCP](#) e 84,46 ms para [UDP](#), enquanto a comunicação direta via [OPC UA](#) apresenta cerca de 69,99 ms. Esse comportamento corresponde a um *overhead* adicional da ordem de 20%, indicando que o middleware mantém boa escalabilidade e previsibilidade mesmo com o aumento da carga.

Em contraste, as arquiteturas baseadas em Web, [REST](#) e [GraphQL](#), apresentam crescimento significativamente mais acentuado da latência com o aumento da quantidade de valores. Para 500 valores, a latência média observada é de aproximadamente 306,44 ms para [REST](#) e 598,52 ms para [GraphQL](#), em comparação com 80,46 ms e 104,03 ms da comunicação via [OPC UA](#), respectivamente. Esses valores correspondem a aumentos de aproximadamente 280% para [REST](#) e entre 470% e 480% para [GraphQL](#), evidenciando que o custo associado à serialização textual e ao processamento das requisições torna-se dominante em cenários com maior volume de dados.

A arquitetura baseada em [GraphQL](#) apresenta consistentemente os maiores valores de latência entre todas as abordagens analisadas, refletindo o custo adicional associado ao processamento das consultas e à construção dinâmica das respostas. Embora ofereça maior flexibilidade na definição das requisições, esse modelo impõe penalidades significativas de desempenho em cenários com alta carga de dados.

### 6.1.5 Avaliação de Desempenho em Topologia Distribuída

Essa Seção apresenta os resultados e discussão sobre os resultados dos experimentos realizados na Topologia Distribuída, conforme ilustrado na Figura 33.

#### 6.1.5.1 Variando a quantidade de mensagens enviadas - leitura e escrita com dados analógicos

A Figura 39 apresenta os resultados das operações de leitura das *tags* analógicas. Por outro lado, a Figura 40 apresenta os dados das operações de escrita.

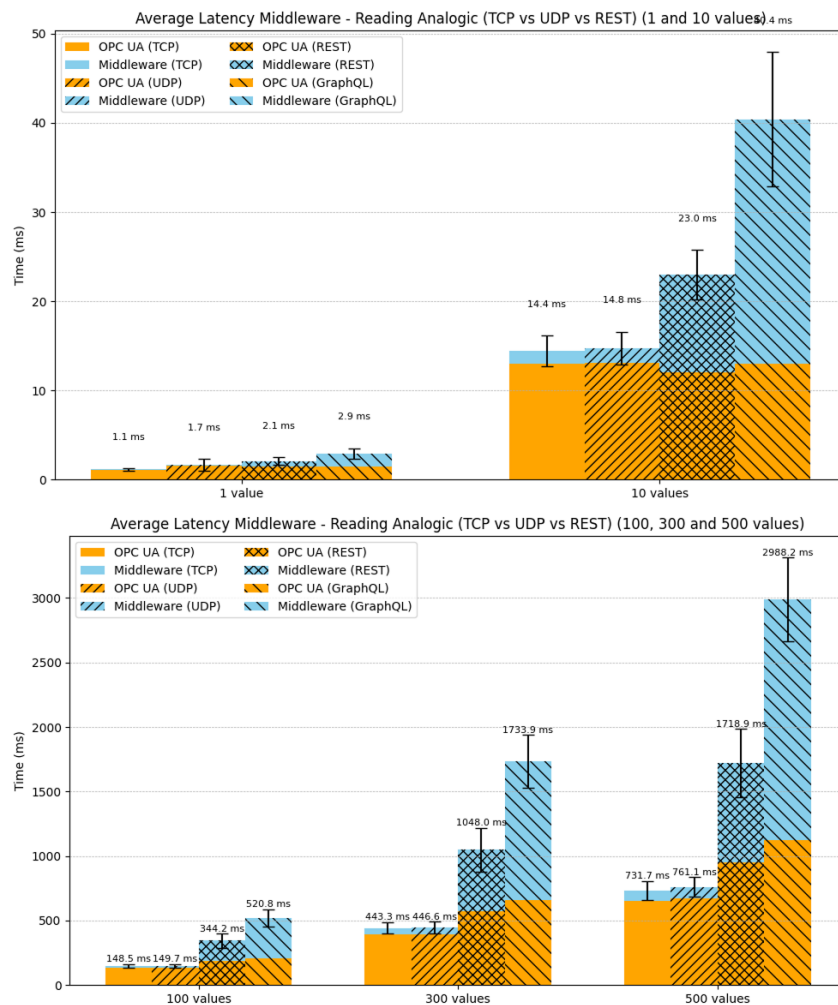


Figura 39: Operação de Leitura com Dados Analógicos na Topologia Distribuída.

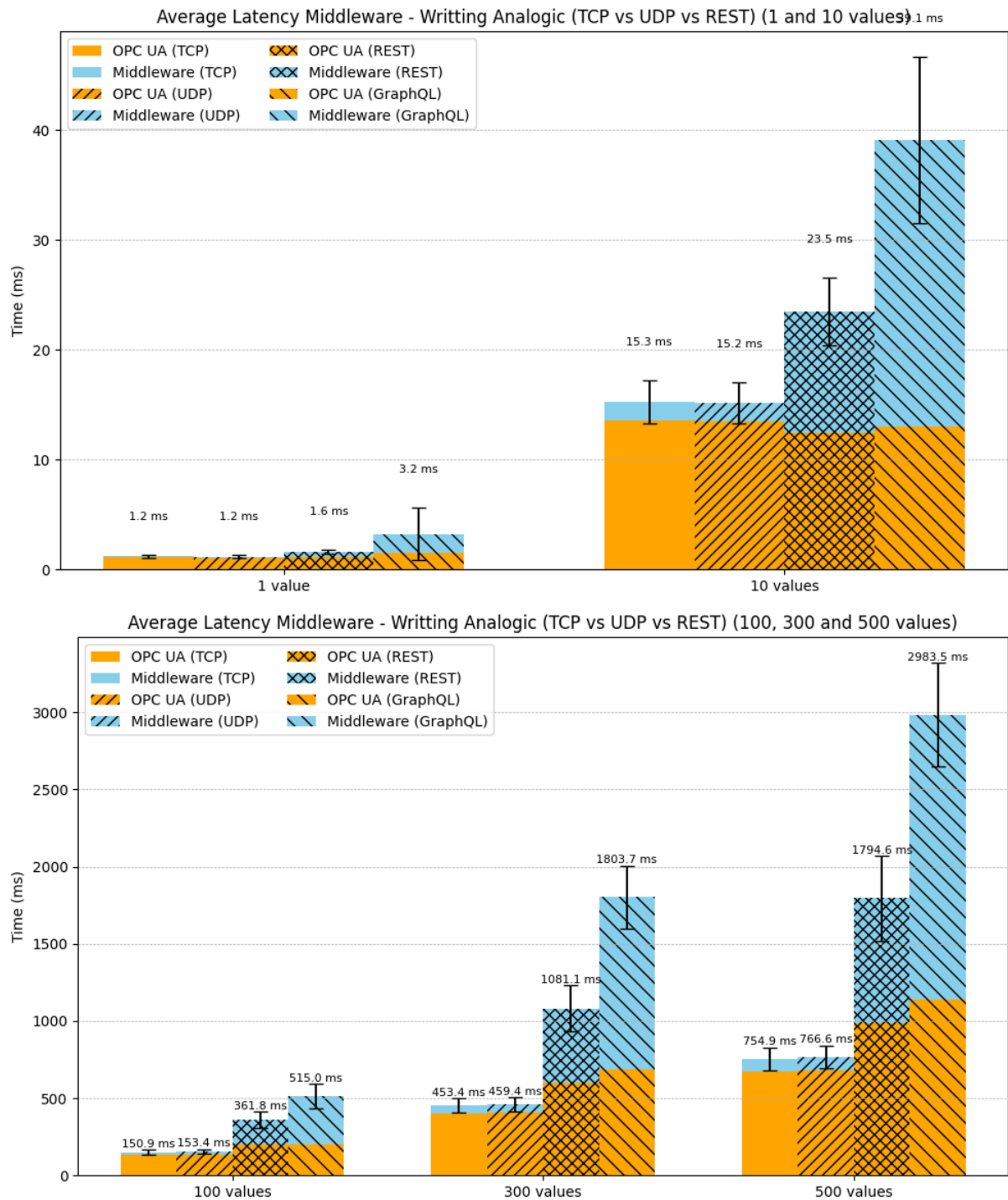


Figura 40: Operação de Escrita com Dados Analógicos na Topologia Distribuída.

Tabela 13: Latência média e desvio padrão para leitura de dados analógicos em topologia distribuída.

| Protocolo | Qtd. Valores | Média (ms) | Desvio Padrão (ms) | OPC UA (ms) |
|-----------|--------------|------------|--------------------|-------------|
| TCP       | 1            | 1.15       | 0.13               | 1.07        |
| TCP       | 10           | 14.44      | 1.75               | 12.96       |
| TCP       | 100          | 148.54     | 14.69              | 132.73      |
| TCP       | 300          | 443.30     | 43.80              | 396.56      |
| TCP       | 500          | 731.74     | 75.73              | 654.21      |
| UDP       | 1            | 1.66       | 0.66               | 1.54        |
| UDP       | 10           | 14.76      | 1.84               | 13.13       |
| UDP       | 100          | 149.67     | 14.69              | 132.98      |
| UDP       | 300          | 446.62     | 43.93              | 396.84      |
| UDP       | 500          | 761.11     | 77.45              | 673.98      |
| REST      | 1            | 2.06       | 0.42               | 1.47        |
| REST      | 10           | 23.01      | 2.76               | 12.04       |
| REST      | 100          | 344.18     | 56.24              | 190.17      |
| REST      | 300          | 1048.02    | 170.80             | 574.83      |
| REST      | 500          | 1718.88    | 265.80             | 949.05      |
| GraphQL   | 1            | 2.94       | 0.58               | 1.48        |
| GraphQL   | 10           | 40.42      | 7.52               | 12.95       |
| GraphQL   | 100          | 520.77     | 68.16              | 206.08      |
| GraphQL   | 300          | 1733.87    | 204.29             | 660.62      |
| GraphQL   | 500          | 2988.16    | 324.86             | 1123.80     |

Tabela 14: Latência média e desvio padrão para escrita de dados analógicos em topologia distribuída.

| Protocolo | Qtd. Valores | Média (ms) | Desvio Padrão (ms) | OPC UA (ms) |
|-----------|--------------|------------|--------------------|-------------|
| TCP       | 1            | 1.22       | 0.15               | 1.13        |
| TCP       | 10           | 15.27      | 1.96               | 13.58       |
| TCP       | 100          | 150.90     | 15.55              | 134.58      |
| TCP       | 300          | 453.42     | 44.83              | 404.07      |
| TCP       | 500          | 754.91     | 74.24              | 672.93      |
| UDP       | 1            | 1.18       | 0.14               | 1.10        |
| UDP       | 10           | 15.17      | 1.90               | 13.44       |
| UDP       | 100          | 153.36     | 15.17              | 135.81      |
| UDP       | 300          | 459.40     | 45.20              | 407.08      |
| UDP       | 500          | 766.60     | 75.73              | 678.23      |
| REST      | 1            | 1.62       | 0.17               | 1.11        |
| REST      | 10           | 23.46      | 3.09               | 12.42       |
| REST      | 100          | 361.82     | 52.21              | 200.41      |
| REST      | 300          | 1081.07    | 147.83             | 598.64      |
| REST      | 500          | 1794.63    | 273.44             | 988.12      |
| GraphQL   | 1            | 3.24       | 2.36               | 1.50        |
| GraphQL   | 10           | 39.08      | 7.56               | 13.00       |
| GraphQL   | 100          | 514.98     | 78.15              | 202.06      |
| GraphQL   | 300          | 1803.70    | 202.53             | 686.97      |
| GraphQL   | 500          | 2983.51    | 335.10             | 1140.07     |

**Análise dos Resultados** Na topologia distribuída, é possível observar um aumento expressivo das latências em relação à topologia local, evidenciando o impacto direto da comunicação em rede sobre o desempenho do sistema. Para cargas reduzidas, correspondentes a 1 e 10 valores analógicos, as latências médias passam da ordem de décimos de milissegundo para valores na faixa de 1 a 3 ms, indicando a presença de um custo adicional associado à transmissão dos dados entre o *raspberrypi* e computador. À medida que a quantidade de valores analógicos aumenta para 100, 300 e 500, a influência da rede torna-se ainda mais evidente. Para 500 valores, as latências médias atingem aproximadamente 730–760 ms nas arquiteturas baseadas em [TCP](#) e [UDP](#), cerca de 1,7 s para [REST](#) e valores próximos a 3 s para [GraphQL](#). Esses resultados representam um aumento de ordem de grandeza em relação à topologia local, o que indica que a latência de rede e o custo de transmissão de grandes volumes de dados passam a ser fatores dominantes em relação ao tempo total das operações.

De forma semelhante ao observado na topologia local, as operações de leitura e escrita apresentam comportamentos praticamente equivalentes em termos de latência média, com diferenças situadas na variabilidade experimental. Esse comportamento reforça que, em topologia distribuída, o desempenho é predominantemente influenciado pelo custo de comunicação em rede e pelo mecanismo de comunicação empregado, e não pela semântica específica da operação (leitura e escrita) realizada pelo Digital Twin.

Apesar do aumento significativo nos valores absolutos de latência, mantém-se o mesmo padrão relativo entre as arquiteturas avaliadas: as abordagens baseadas em comunicação por sockets apresentam melhor desempenho, enquanto as arquiteturas baseadas em [REST](#) e [GraphQL](#) apresentam custos substanciais, especialmente em cenários com grandes volumes de dados.

#### 6.1.5.2 Variando a quantidade de mensagens enviadas - leitura e escrita com dados digitais

A Figura 41 e Figura 42 apresentam os resultados para leitura e escrita das *tags* digitais em topologia distribuída (*setup*).

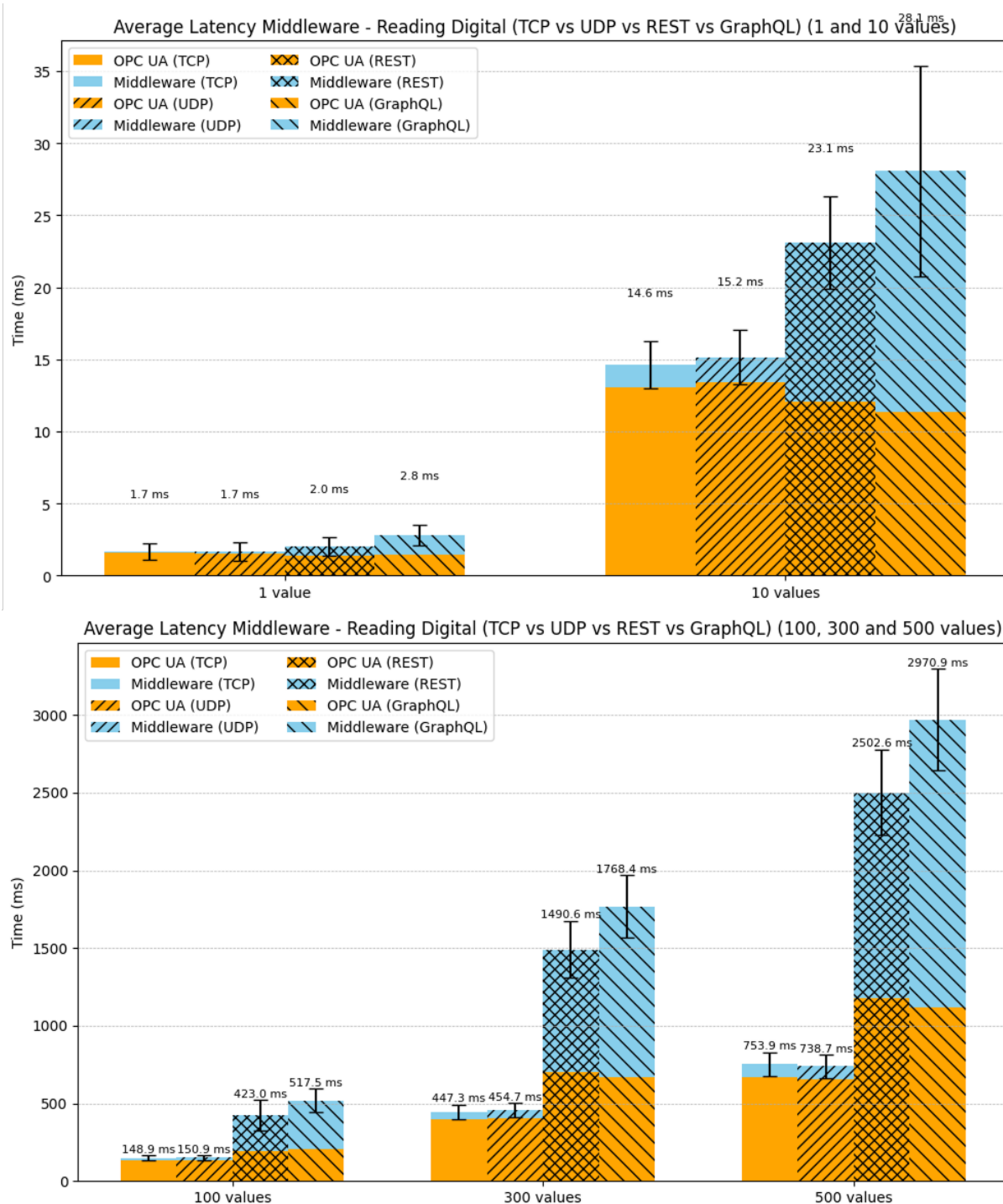


Figura 41: Operação de Leitura com Dados Digitais na Topologia Distribuída.

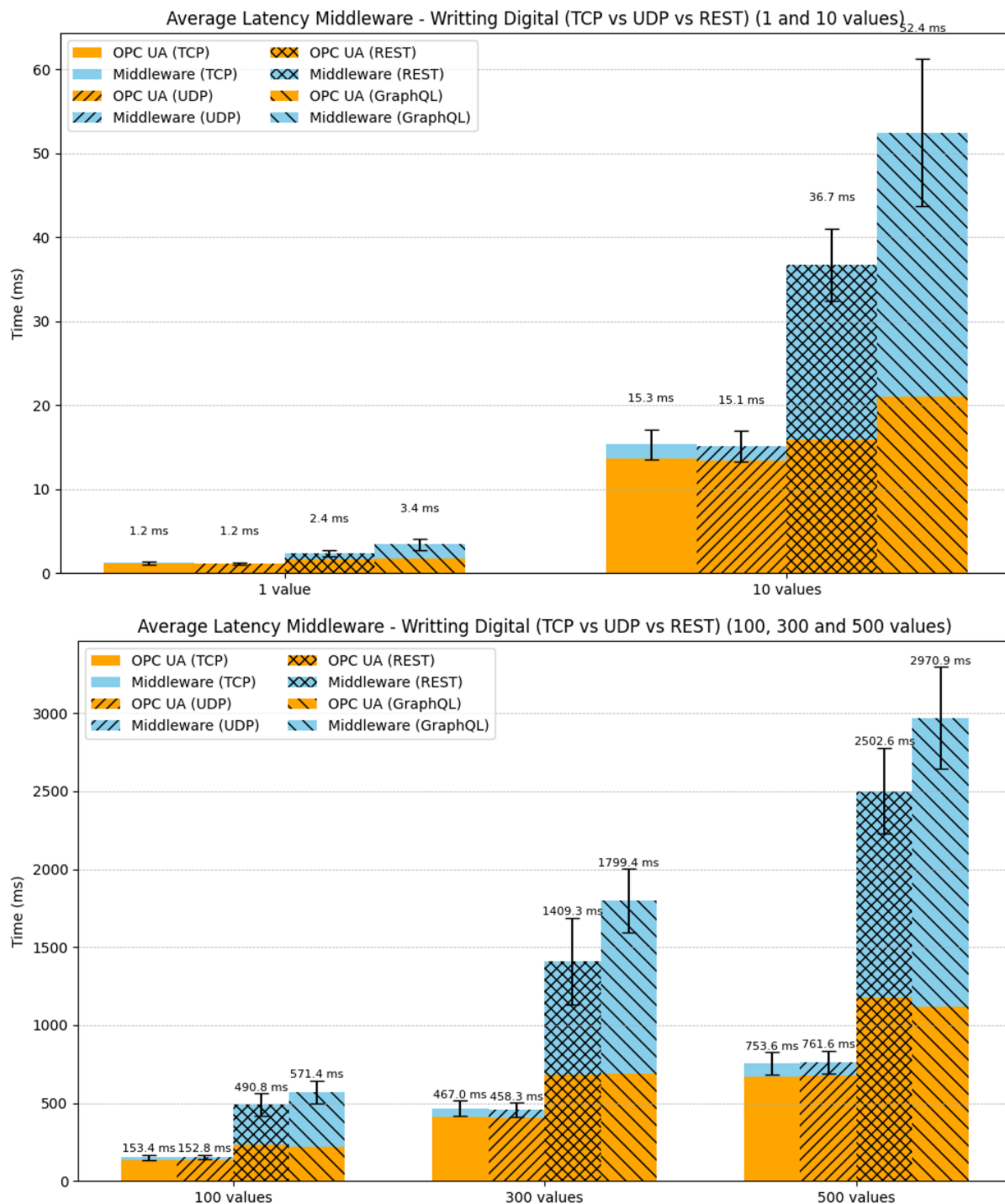


Figura 42: Operação de Escrita com Dados Digitais na Topologia Distribuída.

Tabela 15: Latência média e desvio padrão para leitura de dados digitais em topologia distribuída.

| Protocolo | Qtd. Valores | Média (ms) | Desvio Padrão (ms) | OPC UA (ms) |
|-----------|--------------|------------|--------------------|-------------|
| TCP       | 1            | 1.68       | 0.58               | 1.57        |
| TCP       | 10           | 14.60      | 1.64               | 13.04       |
| TCP       | 100          | 148.87     | 16.02              | 132.45      |
| TCP       | 300          | 447.28     | 45.99              | 398.15      |
| TCP       | 500          | 753.88     | 75.58              | 671.17      |
| UDP       | 1            | 1.66       | 0.66               | 1.54        |
| UDP       | 10           | 15.15      | 1.90               | 13.39       |
| UDP       | 100          | 150.94     | 15.91              | 133.28      |
| UDP       | 300          | 454.70     | 46.26              | 402.38      |
| UDP       | 500          | 738.68     | 72.90              | 656.65      |
| REST      | 1            | 2.00       | 0.66               | 1.40        |
| REST      | 10           | 23.08      | 3.20               | 12.03       |
| REST      | 100          | 423.05     | 99.85              | 196.16      |
| REST      | 300          | 1490.60    | 183.41             | 699.75      |
| REST      | 500          | 2502.64    | 276.60             | 1175.02     |
| GraphQL   | 1            | 2.81       | 0.73               | 1.45        |
| GraphQL   | 10           | 28.06      | 7.28               | 11.33       |
| GraphQL   | 100          | 517.51     | 76.04              | 205.45      |
| GraphQL   | 300          | 1768.42    | 199.68             | 669.68      |
| GraphQL   | 500          | 2970.88    | 326.33             | 1116.91     |

Tabela 16: Latência média e desvio padrão para escrita de dados digitais em topologia distribuída.

| Protocolo | Qtd. Valores | Média (ms) | Desvio Padrão (ms) | OPC UA (ms) |
|-----------|--------------|------------|--------------------|-------------|
| TCP       | 1            | 1.20       | 0.14               | 1.12        |
| TCP       | 10           | 15.33      | 1.78               | 13.62       |
| TCP       | 100          | 153.37     | 16.10              | 136.00      |
| TCP       | 300          | 467.05     | 47.60              | 414.13      |
| TCP       | 500          | 753.64     | 74.20              | 671.70      |
| UDP       | 1            | 1.18       | 0.14               | 1.10        |
| UDP       | 10           | 15.08      | 1.86               | 13.37       |
| UDP       | 100          | 152.79     | 15.10              | 135.20      |
| UDP       | 300          | 458.29     | 45.17              | 406.37      |
| UDP       | 500          | 761.62     | 75.07              | 675.26      |
| REST      | 1            | 2.40       | 0.35               | 1.62        |
| REST      | 10           | 36.73      | 4.32               | 15.89       |
| REST      | 100          | 490.79     | 71.52              | 234.12      |
| REST      | 300          | 1409.29    | 280.68             | 680.71      |
| REST      | 500          | 2502.64    | 276.60             | 1175.02     |
| GraphQL   | 1            | 3.42       | 0.63               | 1.77        |
| GraphQL   | 10           | 52.45      | 8.78               | 20.97       |
| GraphQL   | 100          | 571.42     | 71.84              | 222.67      |
| GraphQL   | 300          | 1799.39    | 204.17             | 687.41      |
| GraphQL   | 500          | 2970.88    | 326.33             | 1116.91     |

**Análise dos Resultados** Os resultados obtidos para operações de leitura e escrita com dados digitais corroboram o comportamento já observado nos cenários anteriores. De forma semelhantes aos cenários experimentais anteriores, não foram identificadas diferenças relevantes entre as duas operações, indicando que a semântica de leitura ou escrita não introduziu influência significativa sobre a latência total quando comparada ao custo de comunicação.

A introdução da comunicação em rede (Figura 33) resulta em um deslocamento significativo dos valores de latência em todas as arquiteturas avaliadas, mantendo, entretanto, o mesmo comportamento relativo de desempenho. As arquiteturas baseadas em TCP e UDP permanecem próximas à comunicação direta via OPC UA, mesmo para cargas mais elevadas, apresentando um crescimento aproximadamente linear à medida que o número de valores aumenta. Por outro lado, as arquiteturas baseadas em REST e GraphQL apresentaram uma degradação acentuada de desempenho em cenários com maior volume de dados, efeito que é amplificado pela presença da rede.

Assim, os resultados em ambiente distribuído reforçam as conclusões obtidas na topologia local, evidenciando que o middleware proposto nesta tese, baseado em comunicação de baixo nível por sockets (TCP/UDP), mantém melhor eficiência e escalabilidade. Além disso, foi confirmado que as diferenças observadas entre as arquiteturas decorrem principalmente dos mecanismos de comunicação utilizados, e não do tipo de dado ou da operação realizada.

### 6.1.6 Variando o tamanho da mensagem - Local vs Distribuído

Os testes foram conduzidos utilizando a leitura e escrita de uma única variável analógica `SIM_Roll_Velocidade`. Essa escolha foi motivada pelos resultados obtidos nos experimentos apresentados nas seções anteriores, os quais permitiram avaliar o impacto do middleware proposto e comparar o comportamento do middleware em função do tipo de dado transferido (analógico e digital). Esses experimentos prévios demonstraram que, nos cenários experimentais considerados, a variação do tipo de dado não introduziu diferenças significativas no comportamento temporal (latência) do middleware. Portanto, a adoção de uma única variável analógica neste conjunto de testes não representou uma simplificação arbitrária, mas uma decisão baseada nas evidências experimentais apresentadas na Seção anterior.

Adicionalmente, os testes iniciais de leitura e escrita de variáveis digitais e analógicas — realizados com diferentes quantidades de mensagens enviadas (1, 10, 100, 300 e 500) —

evidenciaram uma variação significativa no tamanho dos pacotes trafegados. Essa variação está diretamente associada ao tamanho do *payload* das mensagens, isto é, ao conteúdo efetivamente transportado. Embora os cabeçalhos dos protocolos de transporte e de rede permaneçam constantes, o aumento do *payload* resulta em um crescimento do tamanho total dos pacotes. Esse efeito ocorre, por exemplo, em função da estrutura interna das mensagens, do tipo de comando solicitado (inicialização, leitura ou escrita, por exemplo), do formato da requisição, bem como das características da *tag* requisitada, incluindo o tamanho de seu identificador e dos metadados associados. A Figura 43 ilustra esse comportamento, evidenciando que o tamanho do *payload* da camada de aplicação constitui o principal fator de variação no tamanho dos pacotes trafegados nos experimentos<sup>1</sup>.

Com o objetivo de analisar de forma mais precisa o impacto do tamanho do *payload* na comunicação, esta Subseção propõe experimentos adicionais com tamanho fixo da mensagem.

---

<sup>1</sup>Na versão atual do middleware proposta nesta tese, não foi aplicada criptografia aos dados trafegados. Essa decisão teve como objetivo isolar e avaliar de forma controlada o impacto das escolhas arquiteturais, dos mecanismos de comunicação e da serialização de dados sobre o desempenho do middleware. Entretanto a arquitetura proposta é compatível com a incorporação de camadas de segurança, como criptografia de transporte e autenticação, as quais são previstas como extensão natural em trabalhos futuros.

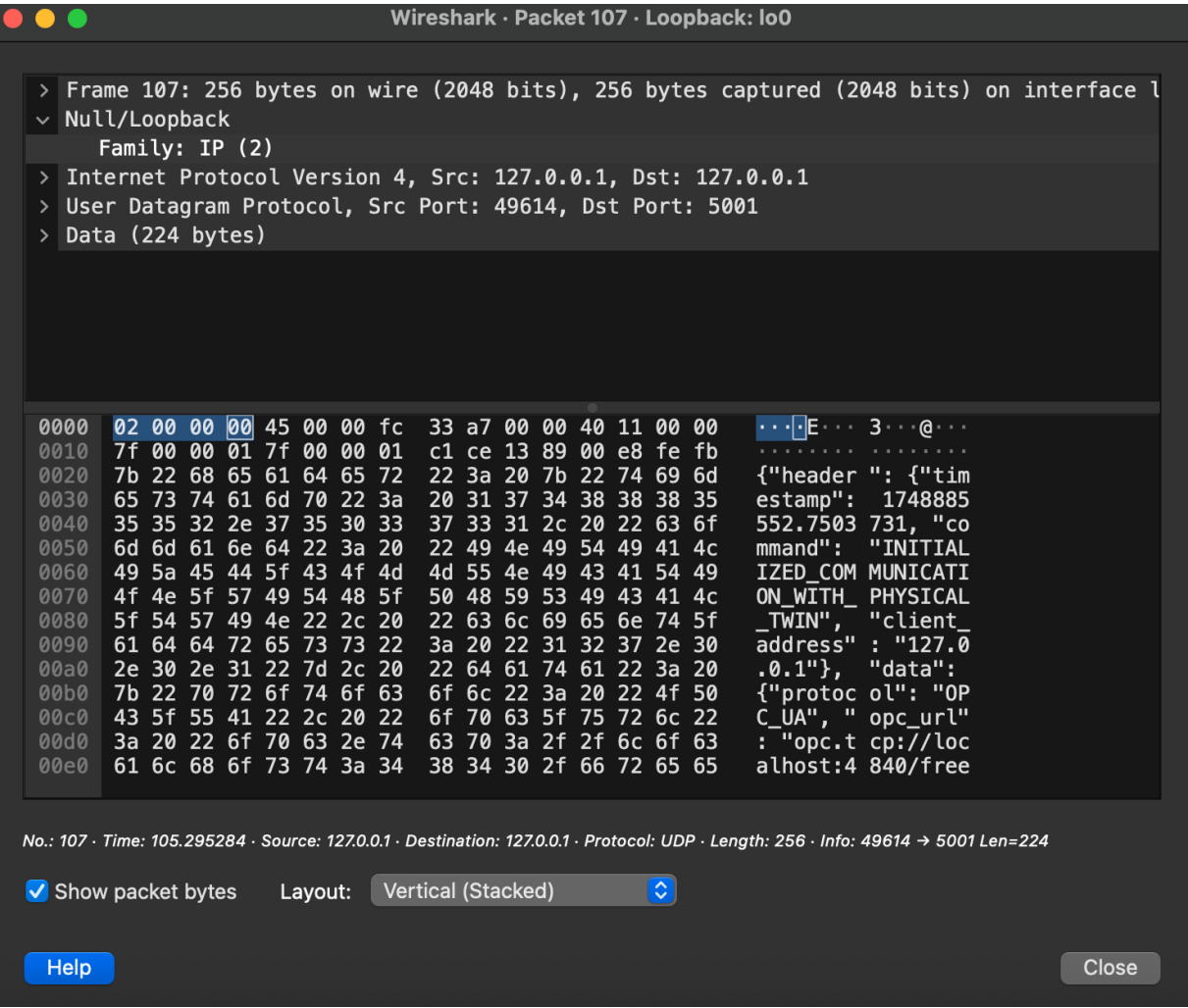


Figura 43: Exemplo de captura de pacote UDP utilizando o Wireshark.

A Figura 44 apresenta os resultados obtidos nos experimentos de leitura. A Tabela 18 consolida as estatísticas de latência para operações de leitura em topologia local e distribuída.

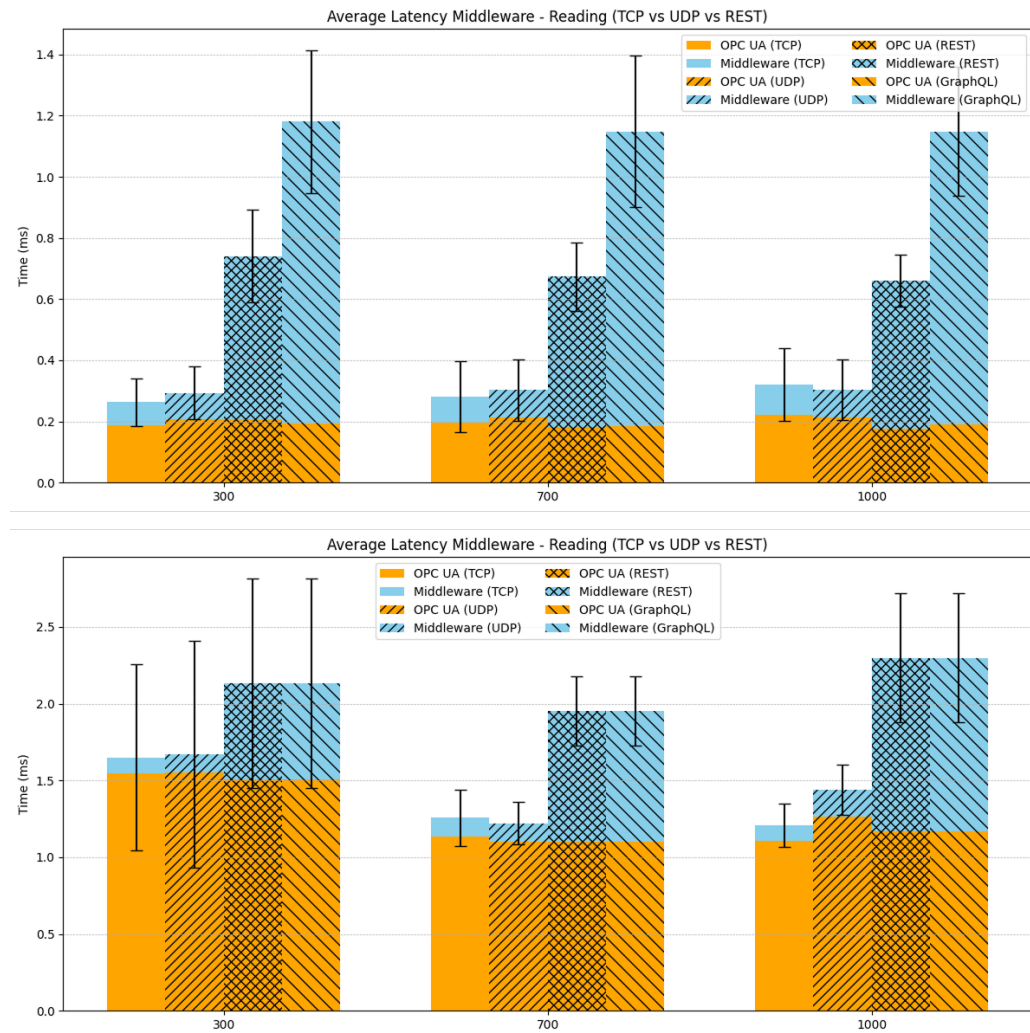


Figura 44: Operação de leitura variando o tamanho da mensagem requisitadas (local vs *setup*).

Tabela 17: Estatísticas de latência do middleware para operações de leitura, considerando a variação do tamanho do *payload*, diferentes protocolos e topologias.

| Topologia    | Protocolo | Payload | Média (ms) | Std (ms) | Mín (ms) | Máx (ms) |
|--------------|-----------|---------|------------|----------|----------|----------|
| Local        | TCP       | 300 B   | 0.263      | 0.078    | 0.060    | 0.738    |
| Local        | TCP       | 700 B   | 0.281      | 0.116    | 0.095    | 0.959    |
| Local        | TCP       | 1000 B  | 0.320      | 0.119    | 0.115    | 0.641    |
| Local        | UDP       | 300 B   | 0.293      | 0.086    | 0.080    | 0.574    |
| Local        | UDP       | 700 B   | 0.303      | 0.100    | 0.096    | 0.596    |
| Local        | UDP       | 1000 B  | 0.304      | 0.099    | 0.095    | 0.596    |
| Local        | REST      | 300 B   | 0.740      | 0.152    | 0.119    | 1.511    |
| Local        | REST      | 700 B   | 0.674      | 0.111    | 0.086    | 1.122    |
| Local        | REST      | 1000 B  | 0.661      | 0.084    | 0.056    | 0.889    |
| Local        | GraphQL   | 300 B   | 1.180      | 0.233    | 0.195    | 2.115    |
| Local        | GraphQL   | 700 B   | 1.149      | 0.247    | 0.195    | 2.438    |
| Local        | GraphQL   | 1000 B  | 1.148      | 0.211    | 0.171    | 2.032    |
| <i>Setup</i> | TCP       | 300 B   | 1.649      | 0.606    | 0.569    | 3.769    |
| <i>Setup</i> | TCP       | 700 B   | 1.257      | 0.182    | 0.132    | 1.898    |
| <i>Setup</i> | TCP       | 1000 B  | 1.209      | 0.139    | 0.072    | 1.652    |
| <i>Setup</i> | UDP       | 300 B   | 1.670      | 0.738    | 0.697    | 4.326    |
| <i>Setup</i> | UDP       | 700 B   | 1.221      | 0.139    | 0.074    | 1.582    |
| <i>Setup</i> | UDP       | 1000 B  | 1.440      | 0.163    | 0.093    | 1.631    |
| <i>Setup</i> | REST      | 300 B   | 2.132      | 0.681    | 0.642    | 4.327    |
| <i>Setup</i> | REST      | 700 B   | 1.954      | 0.224    | 0.131    | 2.293    |
| <i>Setup</i> | REST      | 1000 B  | 2.299      | 0.418    | 0.334    | 4.086    |
| <i>Setup</i> | GraphQL   | 300 B   | 2.676      | 0.652    | 0.612    | 4.234    |
| <i>Setup</i> | GraphQL   | 700 B   | 2.585      | 0.375    | 0.264    | 4.071    |
| <i>Setup</i> | GraphQL   | 1000 B  | 2.865      | 2.052    | 1.508    | 17.327   |

A Figura 45 apresenta os resultados experimentais referentes às operações de escrita em topologia local e distribuída, considerando a variação do tamanho do *payload*. A Tabela 18 apresenta as estatísticas de latência das operações de escrita.

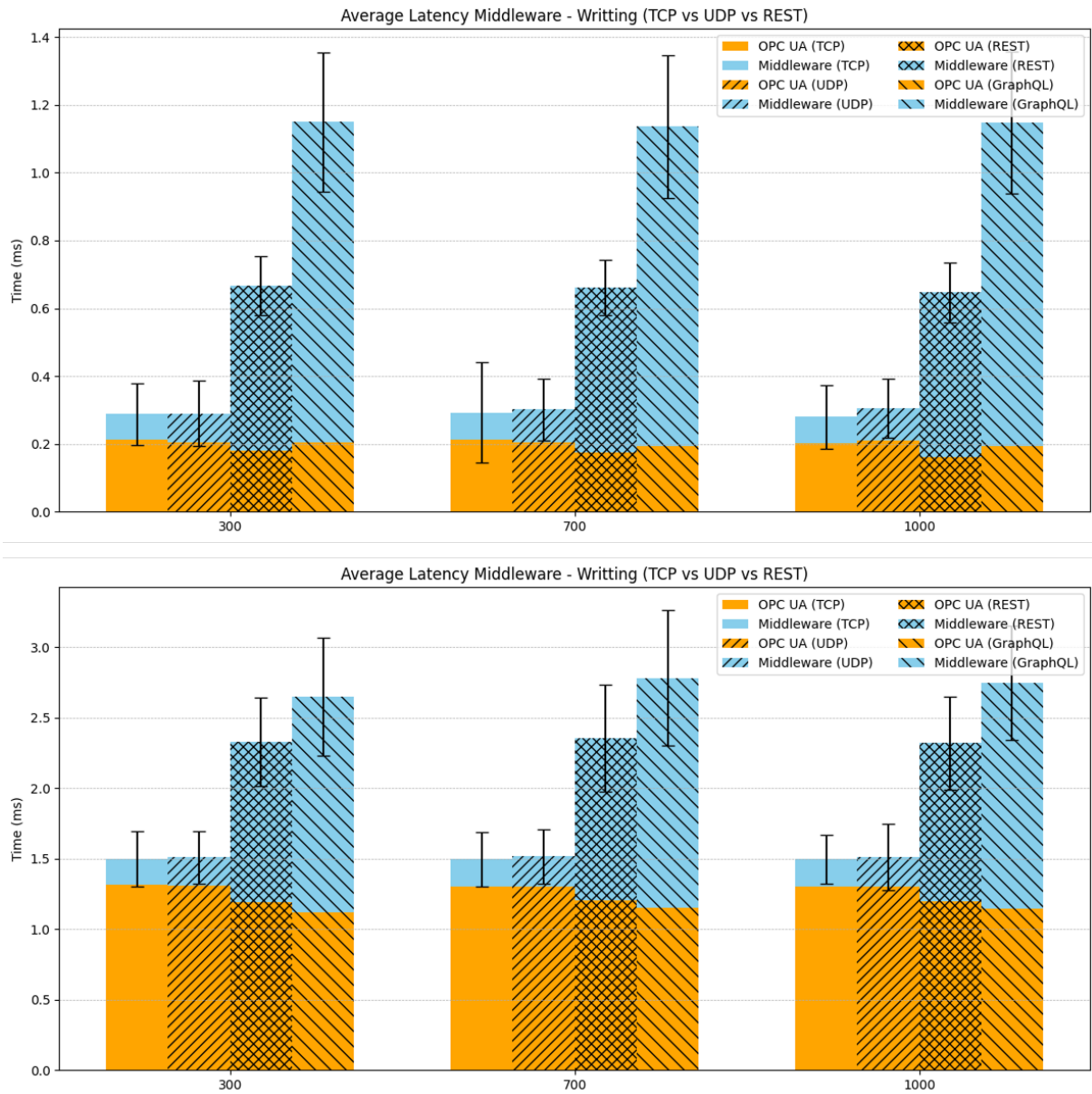


Figura 45: Operação de escrita variando o tamanho da mensagem requisitadas (local vs *setup*).

Tabela 18: Estatísticas de latência do middleware para operações de escrita, considerando a variação do tamanho do *payload*, diferentes protocolos e topologias.

| Topologia | Protocolo | Payload | Média (ms) | Std (ms) | Mín (ms) | Máx (ms) |
|-----------|-----------|---------|------------|----------|----------|----------|
| Local     | TCP       | 300 B   | 0.288      | 0.092    | 0.086    | 0.596    |
| Local     | TCP       | 700 B   | 0.293      | 0.149    | 0.126    | 1.122    |
| Local     | TCP       | 1000 B  | 0.280      | 0.093    | 0.088    | 0.601    |
| Local     | UDP       | 300 B   | 0.290      | 0.098    | 0.094    | 0.560    |
| Local     | UDP       | 700 B   | 0.301      | 0.092    | 0.085    | 0.635    |
| Local     | UDP       | 1000 B  | 0.304      | 0.087    | 0.080    | 0.610    |
| Local     | REST      | 300 B   | 0.667      | 0.086    | 0.055    | 0.981    |
| Local     | REST      | 700 B   | 0.662      | 0.081    | 0.052    | 0.857    |
| Local     | REST      | 1000 B  | 0.647      | 0.088    | 0.061    | 0.955    |
| Local     | GraphQL   | 300 B   | 1.150      | 0.205    | 0.158    | 2.097    |
| Local     | GraphQL   | 700 B   | 1.136      | 0.210    | 0.171    | 2.013    |
| Local     | GraphQL   | 1000 B  | 1.148      | 0.208    | 0.167    | 2.028    |
| Setup     | TCP       | 300 B   | 1.497      | 0.197    | 0.136    | 1.989    |
| Setup     | TCP       | 700 B   | 1.498      | 0.194    | 0.133    | 1.953    |
| Setup     | TCP       | 1000 B  | 1.498      | 0.174    | 0.100    | 1.797    |
| Setup     | UDP       | 300 B   | 1.510      | 0.188    | 0.122    | 1.921    |
| Setup     | UDP       | 700 B   | 1.517      | 0.193    | 0.128    | 1.967    |
| Setup     | UDP       | 1000 B  | 1.511      | 0.237    | 0.194    | 2.011    |
| Setup     | REST      | 300 B   | 2.330      | 0.312    | 0.213    | 3.378    |
| Setup     | REST      | 700 B   | 2.355      | 0.377    | 0.306    | 3.466    |
| Setup     | REST      | 1000 B  | 2.320      | 0.332    | 0.237    | 3.536    |
| Setup     | GraphQL   | 300 B   | 2.649      | 0.421    | 0.313    | 4.440    |
| Setup     | GraphQL   | 700 B   | 2.783      | 0.479    | 0.376    | 4.777    |
| Setup     | GraphQL   | 1000 B  | 2.747      | 0.405    | 0.286    | 4.418    |

**Análise dos resultados:** Em topologia local, foi observado que tanto as operações de leitura quanto de escrita apresentam comportamento praticamente constante para os tamanhos de mensagem avaliados (300 B, 700 B e 1000 B), independentemente da arquitetura de interface utilizada. Esse comportamento indica que, no cenário experimental considerado, o custo total da operação é dominado por componentes fixos, tais como chamadas de sistema associadas à comunicação entre processos, serialização e desserialização das mensagens e o encaminhamento interno das requisições no middleware. Como o volume

de dados transmitidos permanece fixo (leitura e escrita de uma única *tag*), o aumento do *payload* não foi suficiente para alterar de forma significativa esses custos.

No ambiente distribuído, os resultados de leitura e escrita apresentam um deslocamento dos valores absolutos de latência em relação ao cenário local, decorrente da introdução da comunicação em rede, conforme foi observado nas análises anteriores. Esse deslocamento está associado ao tempo de transmissão no enlace físico, bem como a variações introduzidas por mecanismos de enfileiramento, *buffers* de comunicação e escalonamento do sistema operacional nos diferentes nós envolvidos. No entanto, assim como observado no cenário local, não foi verificado uma dependência sistemática da latência em relação ao tamanho do *payload*, o que indica que, não ocorrem efeitos de fragmentação de pacotes, congestionamento do enlace ou saturação de *buffers* capazes de alterar a ordem de grandeza do tempo de resposta.

A variabilidade de latências das operações, expressa pelo jitter, reflete a natureza não determinística desses mesmos componentes. Em topologia local, o jitter observado define um limite inferior associado ao escalonamento do sistema operacional e às variações do ambiente de execução. No ambiente distribuído, foi observado um aumento dessa variabilidade, especialmente para arquiteturas de interface de nível mais alto, como [REST](#) e [GraphQL](#), em função da maior sensibilidade a atrasos variáveis introduzidos pela pilha de comunicação e pelo processamento adicional das camadas relacionadas ao protocolo.

A comparação direta entre operações de leitura e escrita, considerando o por tamanho de *payload*, evidencia que ambas apresentam comportamento equivalente. As diferenças observadas nos valores de latência e *jitter* são pontuais e atribuídas principalmente à fatores estatísticos e à introdução da comunicação em rede no cenário distribuído.

A comparação direta entre operações de leitura e escrita, considerando o tamanho de *payload*, evidenciou que ambas apresentam comportamento equivalente. As diferenças observadas nos valores de latência e *jitter* são pontuais e atribuídas principalmente à fatores estatísticos e à introdução da comunicação em rede no cenário distribuído. .

### 6.1.7 Avaliação de Desempenho da Comunicação Assíncrona

Os experimentos de comunicação assíncrona foram conduzidos exclusivamente considerando a topologia distribuída (Figura 33). Essa decisão foi fundamentada no fato de que a topologia local foi previamente explorada de forma exaustiva nos experimentos de comunicação síncrona apresentados na Seção 6.1.5, os quais permitiram isolar e analisar o

impacto do middleware desconsiderando os efeitos introduzidos pela rede. Dessa forma, a repetição dos experimentos em ambiente local para o mecanismo assíncrono não agregaria evidências relevantes para o cenário assíncrono.

Nesse cenário de teste, apenas os protocolos **TCP** e **UDP** foram considerados por oferecer suporte nativo à comunicação bidirecional e não demandarem implementações adicionais.

Foi desenvolvido um gerador de eventos no servidor **OPC UA**, com a finalidade de emular o comportamento de atualização das variáveis. Algo semelhante ao que ocorre em ambiente de produção, quando os participantes do treinamento no **TI** realizam as interações com os componentes físicos, como válvulas, botões, timão e etc. A Figura 46 apresenta a topologia distribuída com a implementação do gerador de eventos. A Figura 47 apresenta o cenário de teste assíncrono em execução.

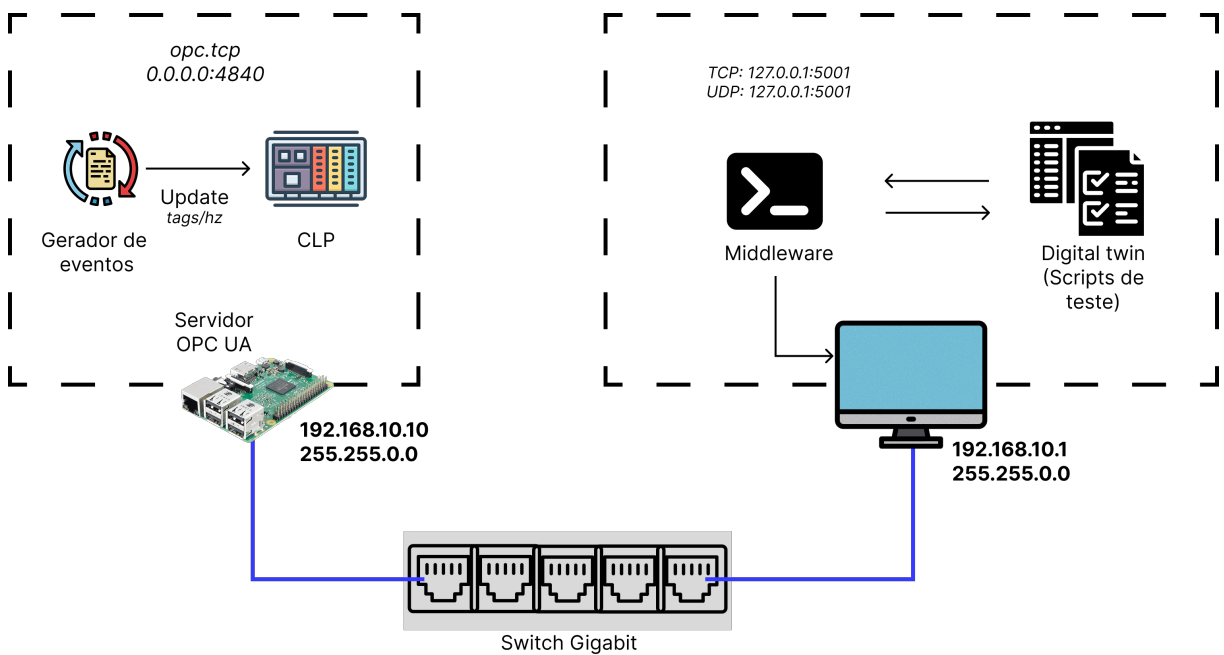


Figura 46: Topologia distribuída para testes com gerador de eventos.

Os experimentos foram organizados a partir da variação do número de tópicos ativos, definidos como  $N \in \{1, 10, 100, 200, 300, 500\}$ . Para cada valor de  $N$ , o **DT** assinante realiza a inscrição simultânea em  $N$  tópicos distintos, passando a receber eventos publicados pelo *middleware*.

A coleta de métricas foi realizada dentro de uma janela temporal fixa de observação de 10 segundos. Durante esse intervalo, todas as mensagens recebidas foram armazenadas, processadas e analisadas. Cada cenário experimental foi repetido 100 vezes de forma independente, com o objetivo de reduzir o impacto de flutuações pontuais do sistema

Script de testes simulando o comportamento do Digital Twin no host

Servidor OPC UA com gerador no Raspberry

Middleware no host

Figura 47: Execução do cenário experimental no cenário de comunicação assíncrona.

operacional, da pilha de protocolos e do canal de comunicação.

Formalizando, para os testes realizados foi considerada uma taxa de atualização constante por tópico  $f$ , uma janela de observação de duração  $T$  e um total de  $N$  tópicos ativos. Portanto, o número total de mensagens geradas durante o experimento pode ser expresso por:

$$M = f \cdot T \cdot N \quad (6.1)$$

Para os experimentos, adotou-se  $f = 10$  Hz e  $T = 10$  s, resultando em:

$$M = 100 \cdot N \quad (6.2)$$

Assim, o aumento do número de tópicos implica diretamente no crescimento linear da carga total de mensagens trafegadas no sistema.

Os testes foram realizados separadamente para os protocolos **TCP** e **UDP**, utilizando a mesma configuração experimental, o que possibilitou uma comparação direta entre os dois mecanismos de transporte no contexto de comunicação assíncrona baseada em eventos.

A Tabela 19 apresenta as variáveis utilizadas nos experimentos.

A Figura 48 apresenta os resultados dos experimentos, considerando o *throughput*.

Tabela 19: Variáveis controladas nos experimentos de comunicação assíncrona.

| Variável                         | Valores avaliados         |
|----------------------------------|---------------------------|
| Tipo de dado                     | Digital de saída          |
| Modelo de comunicação            | Publish/Subscribe         |
| Protocolos de transporte         | TCP, UDP                  |
| Quantidade de tópicos ( $N$ )    | 1, 10, 100, 200, 300, 500 |
| Janela de observação             | 10 segundos               |
| Taxa de atualização por tópico   | 10 Hz                     |
| Número de repetições por cenário | 100 execuções             |
| Protocolo PT-Middleware          | OPC UA                    |

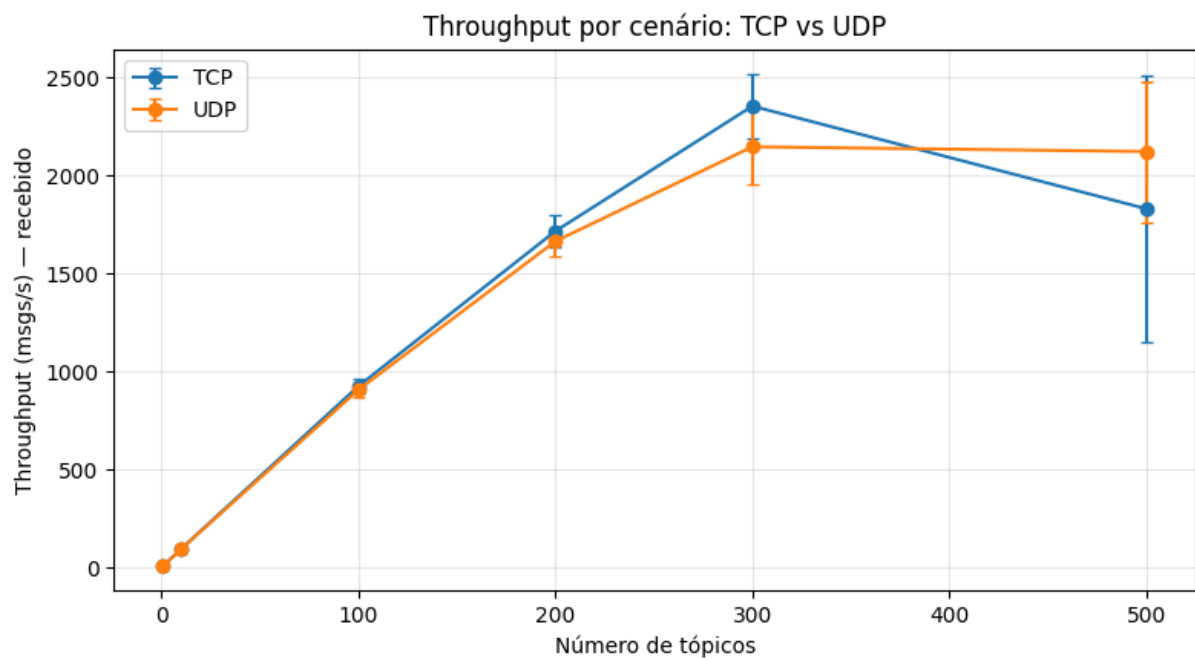


Figura 48: Throughput médio em função do número de tópicos para TCP e UDP.

**Análise dos Resultados:** É possível observar um crescimento aproximadamente linear do *throughput* até 300 tópicos, tanto para TCP quanto para UDP, o que evidencia boa escalabilidade do mecanismo assíncrono (Pub/Sub) implementado no middleware. Nesse intervalo, as diferenças entre os protocolos são pouco expressivas, indicando que o custo adicional dos mecanismos do TCP ainda não foi um fator dominante na latência.

Entretanto, no cenário com 500 tópicos, foi observada uma degradação do desempenho para o protocolo TCP, acompanhada por um aumento significativo do desvio padrão. Esse comportamento sugere a influência de mecanismos internos do TCP, como controle de congestionamento e retransmissões, que possam ter impactado negativamente o *throughput* sob cargas elevadas. Em contrapartida, o protocolo UDP mantém valores médios mais elevados e menor variabilidade relativa, evidenciando maior estabilidade. Vale res-

saltar que, em todos os cenários avaliados, não foram observadas perdas, duplicações ou reordenação de mensagens. A Tabela 20 apresenta os dados consolidados dos resultados dos testes.

Tabela 20: Resultados médios de throughput e integridade das mensagens para comunicação *publish/subscribe* sobre TCP e UDP.

| Protocolo | Tópicos | Throughput<br>(msgs/s) | Desvio<br>(msgs/s) | Perda | Duplicação | Fora de ordem |
|-----------|---------|------------------------|--------------------|-------|------------|---------------|
| TCP       | 1       | 9.952                  | 0.050              | 0.0   | 0.0        | 0.0           |
| TCP       | 10      | 95.625                 | 3.149              | 0.0   | 0.0        | 0.0           |
| TCP       | 100     | 925.339                | 34.259             | 0.0   | 0.0        | 0.0           |
| TCP       | 200     | 1715.951               | 80.897             | 0.0   | 0.0        | 0.0           |
| TCP       | 300     | 2353.192               | 164.111            | 0.0   | 0.0        | 0.0           |
| TCP       | 500     | 1829.704               | 679.507            | 0.0   | 0.0        | 0.0           |
| UDP       | 1       | 9.324                  | 0.043              | 0.0   | 0.0        | 0.0           |
| UDP       | 10      | 95.319                 | 2.967              | 0.0   | 0.0        | 0.0           |
| UDP       | 100     | 906.638                | 33.679             | 0.0   | 0.0        | 0.0           |
| UDP       | 200     | 1664.206               | 73.059             | 0.0   | 0.0        | 0.0           |
| UDP       | 300     | 2145.940               | 193.663            | 0.0   | 0.0        | 0.0           |
| UDP       | 500     | 2121.637               | 359.382            | 0.0   | 0.0        | 0.0           |

## 6.2 Validação da Arquitetura

A modularidade e o agnosticismo a protocolos de comunicação foram os pilares fundamentais no desenvolvimento da arquitetura do middleware proposto. Nesse contexto, esta Seção tem como objetivo validar a extensibilidade arquitetural da solução, demonstrando que a inclusão de um novo protocolo de comunicação pode ser realizada com baixo custo de implementação e impacto mínimo nos componentes centrais do middleware.

O estudo de caso descrito no Capítulo 5 apresentou o **OPC UA** como protocolo de comunicação utilizado pelo **PT**. Para esta avaliação, o mesmo estudo de caso é mantido, contudo, é considerado a necessidade hipotética de substituição do protocolo de comunicação do **PT**. Isto é, parte-se do pressuposto de que, por alterações nos requisitos técnicos e funcionais, tornou-se necessária a substituição do protocolo originalmente adotado no **TI**.

Para este cenário experimental foi selecionado o protocolo **MQTT**. A escolha do protocolo justifica-se pelo seu modelo de comunicação orientado a mensagens no padrão **Pub/Sub**, mediado por um *broker*, representando um modelo de comunicação estruturalmente diferente do utilizado no **OPC UA**. Dessa forma, a adoção do **MQTT** permitiu avaliar a capacidade da arquitetura em absorver mudanças significativas no mecanismo

de comunicação.

### 6.2.1 Cenário Experimental com MQTT

Nesta Subseção é apresentado o cenário experimental adotado para integrar o protocolo [MQTT](#) ao estudo de caso. Foi mantido o mesmo conjunto de variáveis (*tags*) e o mesmo padrão de operações de leitura e escrita originalmente utilizado com [OPC UA](#). A Figura 49 ilustra como o cenário foi elaborado para o teste. Foi criado um repositório de *tags* (*TagRepository*). Esse repositório armazena as variáveis digitais e analógicas, tanto de leitura quanto de escrita, que foram inicializadas a partir das listas de *tags* já utilizadas no estudo de caso do [TI](#). Foi implementado um gerador de eventos (*PTEventGeneratorMQTT*) responsável por atualizar periodicamente um conjunto de *tags* e publicar seus valores nos respectivos tópicos de saída. O gerador recebe como parâmetros: (i) o subconjunto de *tags* a serem atualizadas; e (ii) a taxa de atualização em Hertz (*rate\_hz*).

A comunicação foi estruturada por tópicos direcionais, codificando explicitamente (i) o domínio da variável (*Digital* ou *Analogic*) e (ii) a direção do fluxo (*in* ou *out*). O formato geral dos tópicos é dado por:

`{base}/{Domain}/{Direction}/{TagName}`

onde *base* representa o prefixo configurável do experimento (por exemplo, *pt*), *Domain*  $\in \{\text{Digital}, \text{Analogic}\}$ , *Direction*  $\in \{\text{in}, \text{out}\}$  e *TagName* corresponde ao identificador textual da variável.

Com base nessa convenção, foram definidos dois conjuntos de tópicos:

- **Tópicos de entrada (in):** utilizados para comandos de escrita (*write*), aos quais o [PT](#) se inscreve. Ex.: *pt/Digital/in/<TagName>* e *pt/Analogic/in/<TagName>*.
- **Tópicos de saída (out):** utilizados para publicação do estado/telemetria, viabilizando assinatura (*subscribe*) e leitura do último valor publicado. Ex.: *pt/Digital/out/<TagName>* e *pt/Analogic/out/<TagName>*.

Em relação ao formato As mensagens trafegadas no [Message Queuing Telemetry Transport \(MQTT\)](#) foram codificadas em [JavaScript Object Notation \(JSON\)](#), preservando um formato simples e uniforme entre domínios. No caso de publicações de estado (*out*), o [PT](#) publica estruturas do tipo:

```
{"tag": «TagName>", "value": <V>, "ts": <timestamp>}
```

e, no caso de comandos de escrita (in), o middleware publica:

```
{"value": <V>, "ts": <timestamp>, "source": "Middleware"}
```

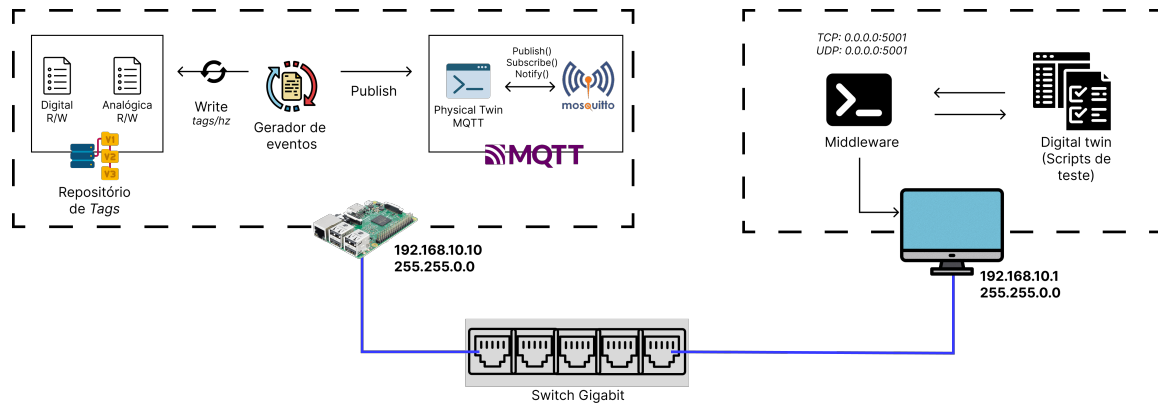


Figura 49: Cenário experimental com MQTT.

### 6.2.2 Implementação do Cliente MQTT no Middleware

O objetivo dessa etapa foi incorporar um novo mecanismo de comunicação sem introduzir novas dependências ou modificar os componentes fundamentais do sistema.

Do ponto de vista arquitetural, a inclusão do MQTT exigiu apenas duas alterações principais: (i) a implementação de um novo cliente de comunicação, responsável por encapsular as implementações do protocolo MQTT; e (ii) a extensão pontual do mecanismo de criação de clientes, seguindo o padrão (*Factory*).

A Figura 50 apresenta o diagrama de classes atualizado com a implementação do cliente MQTT. O novo adaptador, denominado `MQTTClient`, foi desenvolvido como uma implementação concreta da interface abstrata `PhysicalTwinClient`, previamente utilizada pelo adaptador OPC UA. Dessa forma, embora o MQTT adote um modelo de comunicação orientado a mensagens e mediado por broker, enquanto o OPC UA utiliza um modelo cliente-servidor fortemente tipado, essas particularidades são tratadas exclusivamente no interior do adaptador. Adicionalmente, a classe responsável pela criação de instâncias de clientes de comunicação (`DigitalTwinClientFactory`) foi estendida para reconhecer o protocolo MQTT como uma nova opção válida de protocolo de comunicação. Complementarmente, a Figura 51 apresenta o cenário de teste em execução.

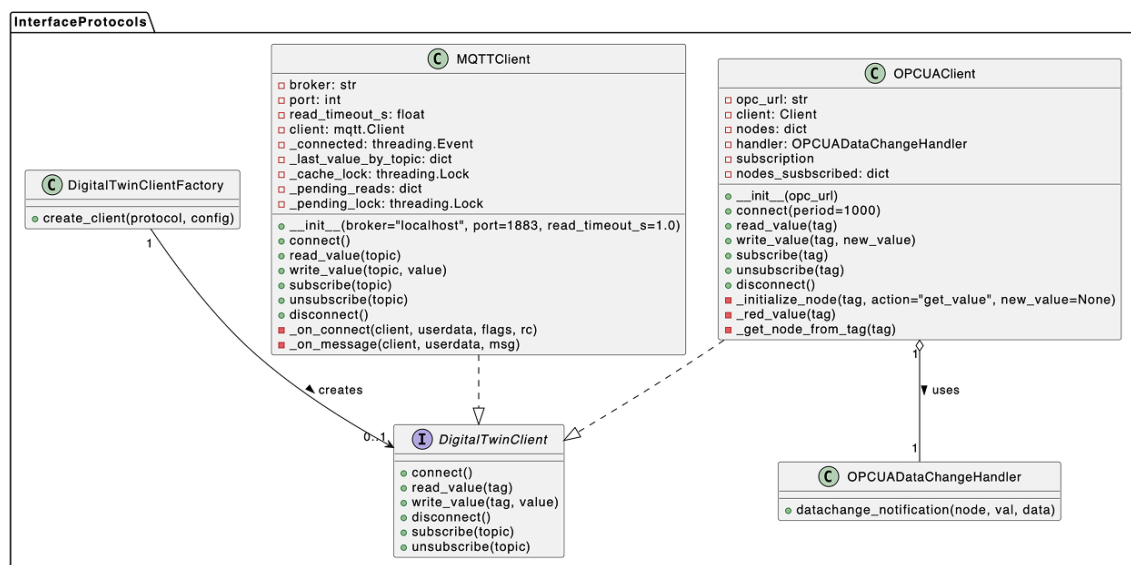


Figura 50: Implementação MQTT.

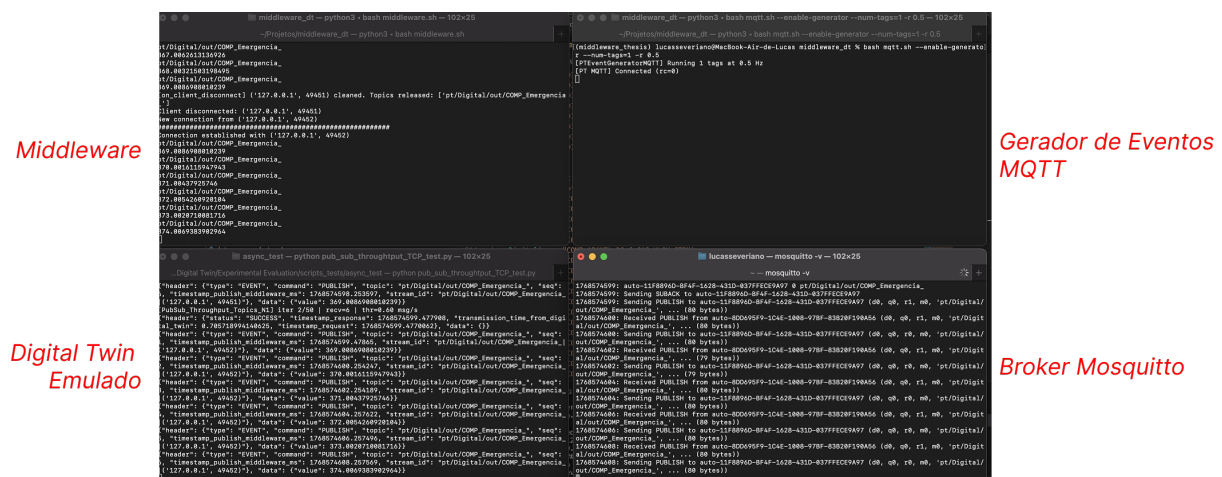


Figura 51: Cenário experimental com MQTT.

Como evidência dos testes, a Tabela 21 apresenta um resumo das alterações realizadas no middleware para suportar o protocolo MQTT. É possível Observar que a extensão demandou exclusivamente a criação de um novo cliente no pacote *InterfaceProtocols* e uma modificação pontual na fábrica de clientes (*DigitalTwinClientFactory*), sem alterações no núcleo do middleware. Esse resultado reforça que a arquitetura suporta a substituição de protocolos por meio de composição e configuração, e não por modificação estrutural profunda.

Tabela 21: Evidências do baixo custo arquitetural da extensão do protocolo MQTT.

| Aspecto avaliado                   | Resultado observado           |
|------------------------------------|-------------------------------|
| Protocolo adicionado               | MQTT                          |
| Novo adaptador implementado        | MQTTClient                    |
| Pacotes centrais alterados         | Nenhum                        |
| Alterações no núcleo do middleware | Não                           |
| Alterações na API pública          | Não                           |
| Modificação na fábrica de clientes | Sim (pontual)                 |
| Estratégia de integração           | Substituição por configuração |
| Impacto arquitetural               | Mínimo                        |

De forma complementar, a análise do histórico de versionamento do código-fonte corrobora essa constatação. A inspeção do comando `git diff -stat` entre a versão base do middleware, utilizando [OPC UA](#), e a versão estendida com suporte a [MQTT](#), evidencia que as alterações concentram-se exclusivamente nos arquivos associados ao adaptador MQTT e na fábrica de clientes. Não foram observadas modificações propagadas para outros módulos ou componentes centrais da arquitetura. Esse resultado confirma, de maneira empírica, que a arquitetura proposta atende ao requisito de extensibilidade com baixo acoplamento, validando o princípio de *middleware protocol-agnostic* adotado como um dos pilares fundamentais desta Tese.

```
(middleware_thesis) lucasseveriano@MacBook-Air-de-Lucas middleware_dt % git diff --stat
Middleware/InterfaceProtocols/digital_twin_client_factory.py | 9 ++++++--
Middleware/config.json | 2 +-
2 files changed, 8 insertions(+), 3 deletions(-)
(middleware_thesis) lucasseveriano@MacBook-Air-de-Lucas middleware_dt %
```

Figura 52: Evidências de alteração do código.

# 7 Conclusão

Este capítulo apresenta a conclusão deste trabalho. A Seção 7.1 apresenta as considerações finais e conclusões. A Seção 7.2 aponta os direcionamentos para as próximas etapas do projeto.

## 7.1 Considerações Finais

Nesta tese de doutorado foi proposta e implementada uma arquitetura de *middleware* modular e agnóstica a protocolos para a comunicação eficiente entre sistemas de Digital Twin (DT) e Physical Twin (PT). A arquitetura foi projetada como um mecanismo flexível de troca de dados, capaz de abstrair as complexidades relacionadas aos mecanismos de comunicação. A arquitetura promove a separação de responsabilidades entre a lógica de negócio e a infraestrutura de rede. Além disso, a proposta consistiu em implementar a arquitetura considerando escolhas arquiteturais orientadas à eficiência na comunicação. Tais escolhas envolveram a adoção de mecanismos de comunicação de baixo *overhead*, o uso de interfaces baseadas em *sockets* (Transmission Control Protocol (TCP) e User Datagram Protocol (UDP)) e a definição de uma estrutura modular capaz de reduzir o acoplamento entre os componentes do sistema.

Para a validação da proposta de arquitetura apresentada nesta tese, foi definido um cenário experimental fundamentado em um estudo de caso real, correspondente ao simulador de imersão (Simulador de Treinamento de Imersão (TI)). Embora seja classificado como um simulador, o TI apresenta características que permitem caracterizá-lo como um DT. Essas características incluem a representação virtual de um sistema físico, a comunicação bidirecional entre os domínios físico e digital e a sincronização contínua de estados em tempo real. Esse ambiente experimental foi selecionado por representar um sistema que requer comunicação eficiente e de baixa latência.

A avaliação experimental foi conduzida de forma sistemática e controlada, considerando diferentes configurações de topologia (local e distribuída), tipos de dados (análo-

gicos e digitais), operações de leitura e escrita, variações na quantidade de mensagens e diferentes tamanhos de *payload*. Esses cenários experimentais foram definidos com base nos requisitos estabelecidos para operação do TI. Finalmente, foi proposto um cenário de substituição hipotética do protocolo [Open Platform Communications Unified Architecture \(OPC UA\)](#) por [Message Queuing Telemetry Transport \(MQTT\)](#), com a finalidade de avaliar as propriedades de agnosticismo a protocolo, modularidade e capacidade de evolução da arquitetura,

Os resultados obtidos dos experimentos evidenciaram que a arquitetura proposta nesta tese atende aos requisitos arquiteturais definidos para sistemas de DT, especialmente no que se refere à eficiência da comunicação e ao baixo *overhead*. Em comparação com as abordagens baseadas em tecnologias *Web*, como [Representational State Transfer \(REST\)](#) ([Schiekofer; Scholz; Weyrich, 2018](#); [Grüner; Pfrommer; Palm, 2016](#); [Cavalieri; Salafia; Scropo, 2019](#); [Künzel et al., 2021](#)) e [Graph Query Language \(GraphQL\)](#) ([Hietala et al., 2020b](#); [Ala-Laurinaho et al., 2022](#)), a proposta abordada nessa tese apresentou desempenho superior, com menores latências médias e maior previsibilidade sob diferentes níveis de carga. Adicionalmente, a avaliação dos mecanismos de comunicação assíncrona evidenciou um comportamento escalável do *middleware*, com crescimento aproximadamente linear do *throughput* até 300 tópicos. Esses resultados demonstraram a eficiência do modelo *publish/subscribe* implementado, bem como sua capacidade de suportar o aumento da carga sem degradação significativa de desempenho nesse intervalo. Porém, foi identificado degradação do desempenho para cargas 500 tópicos. Para todos cenários experimentais para comunicação assíncrona não foram observadas perdas, duplicações ou reordenação de mensagens, o que reforça a confiabilidade do mecanismo assíncrono proposto.

## 7.2 Trabalhos Futuros

Apesar das contribuições apresentadas, esta tese possui algumas limitações relacionadas ao desenvolvimento e à implementação da solução proposta. Tais limitações representam oportunidades de extensão e melhoria do trabalho. Nesse contexto, pretende-se realizar a implementação dos mecanismos de serialização de mensagens mais eficientes, como *Protocol Buffers* e outros formatos binários, com a finalidade de reduzir o custo de serialização e o volume de dados trafegados. Adicionalmente, pretende-se investigar o impacto dessas abordagens na latência e no *throughput* do *middleware*.

No contexto de segurança, é previsto a incorporação de mecanismos de segurança na comunicação baseada em *sockets*. Esses mecanismos incluem o uso de protocolos seguros na camada de transporte, como [Transport Layer Security \(TLS\)](#)/[Secure Sockets Layer \(SSL\)](#), bem como a confidencialidade e integridade dos dados trafegados entre o DT e o PT. No que se refere à tolerância a falhas, pretende-se investigar estratégias para eliminação de pontos únicos de falha (*Single Point of Failure* – SPOF), por meio da containerização do middleware e sua orquestração em ambientes como Kubernetes. Como o foco deste trabalho foi a redução da latência, não foi considerado o uso de tecnologias de containerização, em virtude do potencial impacto no desempenho da solução. A adoção de contêineres pode introduzir sobrecargas adicionais relacionadas à virtualização, principalmente em ambientes baseados em *Windows* ou *MacOS*, à rede e ao gerenciamento de recursos, o que poderia influenciar diretamente as métricas avaliadas.

No que se refere às propriedades de desacoplamento, o *middleware* proposto já contempla o desacoplamento espacial e de sincronização, o que permite que DT e PT operem de forma independente, sem necessidade de conhecimento mútuo ou bloqueio na comunicação. Entretanto, o desacoplamento temporal ainda não é plenamente suportado, uma vez que a troca de mensagens depende da disponibilidade simultânea dos componentes. Nesse contexto, para trabalhos futuros pretende-se a introdução de uma camada de persistência de dados, com o objetivo de viabilizar o desacoplamento temporal entre o DT e o PT. Essa camada permitirá o armazenamento intermediário das mensagens, possibilitando seu consumo posterior e contribuindo para maior resiliência, flexibilidade e robustez do sistema em cenários distribuídos.

Finalmente, é prevista a evolução da solução para um modelo de produto, por meio da implementação de um serviço em execução contínua (*daemon*), inspirado em sistemas como *message brokers*. Além disso, é previsto o desenvolvimento de uma biblioteca cliente para facilitar sua integração e utilização por aplicações externas.

# REFERENCIAS

ABACO SYSTEMS. **Technology Readiness Level (TRL)**. Acesso em: 30 ago. 2025. 2025. Disponível em: <https://abaco.com/technology-readiness-level>.

ABDELGAWAD, Kareem; ABDELKARIM, Mohamed *et al.* A scalable framework for advanced driver assistance systems simulation. *In: CITESEER. PROCEEDINGS of the 6th International Conference on Advances in System Simulation (SIMUL2014), Nice, France. [S. l.: s. n.], 2014. p. 12–16.*

ABDELGAWAD, Kareem; GAUSEMEIER, Jürgen; STÖCKLEIN, Jörg *et al.* A platform with multiple head-mounted displays for advanced training in modern driving schools. **Designs**, MDPI, v. 1, n. 2, p. 8, 2017.

ABDELGAWAD, Kareem; GAUSEMEIER, Jürgen; TRÄCHTLER, Ansgar *et al.* An Application-Oriented Design Method for Networked Driving Simulation. **Designs**, MDPI, v. 1, n. 1, p. 6, 2017.

ABDELGAWAD, Kareem; HASSAN, Bassem *et al.* A modular architecture of an interactive simulation and training environment for advanced driver assistance systems. **Int. J. Adv. Softw. IARIA**, v. 8, p. 247–261, 2015.

ABERER, Karl; HAUSWIRTH, Manfred; SALEHI, Ali. A middleware for fast and flexible sensor network deployment. *In: September. VLDB. [S. l.: s. n.], 2006. v. 6, p. 1215.*

ALA-LAURINAHO, Riku. of thesis Sensor Data Transmission from a Physical Twin to a Digital Twin, 2019.

ALA-LAURINAHO, Riku *et al.* Comparison of REST and GraphQL Interfaces for OPC UA. **Computers**, MDPI, v. 11, n. 5, p. 65, 2022.

ALMEIDA, Andre *et al.* Middleware for digital twins: A systematic mapping study. *In: PROCEEDINGS of the 1st International Workshop on Middleware for Digital Twin. [S. l.: s. n.], 2023. p. 19–24.*

ALVARENGA, Cristiane do Bom Conselho Sales *et al.* Framework para Digital Twins: integração entre o BIM, IoT e plataformas em nuvem para projetos de estruturas de edifícios industriais. Universidade Federal de Minas Gerais, 2024.

ANDRÉ, Pascal; AZZI, Fawzi; CARDIN, Olivier. Heterogeneous communication middleware for digital twin based cyber manufacturing systems. *In: SPRINGER. SERVICE Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future: Proceedings of SOHOMA 2019 9*. [S. l.: s. n.], 2020. p. 146–157.

ASHTON, Kevin *et al.* That ‘internet of things’ thing. **RFID journal**, Hauppauge, New York, v. 22, n. 7, p. 97–114, 2009.

ATAMAN, Tevfik; BIBERCI, Mehmet Ali; CELIK, Mustafa Bahattin. Simulation of Advanced Driving Assistance Systems for a Dynamic Vehicle Model. **Engineering, Technology & Applied Science Research**, v. 14, n. 5, p. 16553–16558, 2024.

ATZORI, Luigi; IERA, Antonio; MORABITO, Giacomo. The internet of things: A survey. **Computer networks**, Elsevier, v. 54, n. 15, p. 2787–2805, 2010.

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software Architecture in Practice**. 4th. [S. l.]: Addison-Wesley Professional, 2021. ISBN 9780136886099.

BELLAVISTA, Paolo *et al.* An entanglement-aware middleware for digital twins. **ACM Transactions on Internet of Things**, ACM New York, NY, v. 5, n. 4, p. 1–25, 2024.

BERNERS-LEE, Tim; FIELDING, Roy; FRYSTYK, Henrik. **RFC1945: Hypertext Transfer Protocol–HTTP/1.0**. [S. l.]: RFC Editor, 1996.

BOOCH, Grady *et al.* Object-oriented analysis and design with applications. **ACM SIGSOFT software engineering notes**, ACM New York, NY, USA, v. 33, n. 5, p. 29–29, 2008.

BORSATTI, Davide *et al.* From IoT to cloud: applications and performance of the MQTT protocol. *In: IEEE. 2020 22nd international conference on transparent optical networks (ICTON)*. [S. l.: s. n.], 2020. p. 1–4.

BOSCHERT, Stefan; ROSEN, Roland. Digital twin—the simulation aspect. *In: MECHATRONIC futures: Challenges and solutions for mechatronic systems and their designers*. [S. l.]: Springer, 2016. p. 59–74.

CARBONEL MARTÍNEZ, Asier; GALLEGO MARTÍNEZ, José Ramón; TOLOSANA CALASANZ, Rafael. Diseño e implementación de un middleware CoAP-MQTT-HTTP para la mejora de la interoperabilidad de los protocolos de aplicación en redes IoT, 2020.

CAVALIERI, Salvatore; SALAFIA, Marco Giuseppe; SCROPPO, Marco Stefano. Integrating OPC UA with web technologies to enhance interoperability. **Computer Standards & Interfaces**, Elsevier, v. 61, p. 45–64, 2019.

COUNCIL, Forbes Technology. **How Enterprises Are Using Digital Twins To Drive Innovation**. Accessed: 2026-01-21. 2024. Disponível em: <https://www.forbes.com/sites/forbestechcouncil/2024/01/10/>.

CRUZ, Mauro Amaro da. **An enhanced multi-protocol middleware solution for Internet of things**. 2021. Tese (Doutorado) – Université de Haute Alsace-Mulhouse.

DAVE, Madhavi; DOSHI, Jyotika; AROLKAR, Harshal. MQTT-CoAP interconnector: IoT interoperability solution for application layer protocols. *In: IEEE. 2020 fourth international conference on I-SMAC (IoT in social, mobile, analytics and cloud)(I-SMAC)*. [S. l.: s. n.], 2020. p. 122–127.

DAVENPORT, Thomas H. **Mission Critical: Realizing the Promise of Enterprise Systems**. [S. l.]: Harvard Business Press, 2000.

DE SOUZA, Luciana Moreira Sá *et al.* Socrates: A web service based shop floor integration infrastructure. *In: SPRINGER. THE Internet of Things: First International Conference, IOT 2008, Zurich, Switzerland, March 26-28, 2008. Proceedings*. [S. l.: s. n.], 2008. p. 50–67.

DONOVAN, Alan A. A.; KERNIGHAN, Brian W. **The Go Programming Language**. [S. l.]: Addison-Wesley Professional, 2015. ISBN 9780134190440.

DURKOP, Lars; CZYBIK, Bjorn; JASPERNEITE, Jurgen. Performance evaluation of M2M protocols over cellular networks in a lab environment. *In: IEEE. 2015 18th international conference on Intelligence in Next Generation Networks*. [S. l.: s. n.], 2015. p. 70–75.

ECMA INTERNATIONAL. **The JSON Data Interchange Standard (ECMA-404)**. [S. l.: s. n.], 2017. Disponível em: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.

ENDLER, M *et al.* Contextnet: Context reasoning and sharing middleware for large-scale pervasive collaboration and social networking. *In: PROCEEDINGS of the workshop on posters and demos track*. [S. l.: s. n.], 2011. p. 1–2.

FACEBOOK, INC. **GraphQL: A Query Language for APIs**. [S. l.: s. n.], 2015. Accessed: 2025-11-07. Disponível em: <https://graphql.org/>.

FARNHAM, B; BARILLERE, R. **MIGRATION FROM OPC-DA TO OPC-UA**. [S. l.], 2011.

FIELDING, Roy *et al.* **RFC2616: Hypertext transfer protocol–HTTP/1.1**. [S. l.]: RFC Editor, 1999.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. Tese (Doutorado) – University of California, Irvine.

FOUNDATION, GraphQL. **GraphQL Specification**. [S. l.: s. n.], 2023. Accessed: 2025-11-07. Disponível em: <https://spec.graphql.org/>.

FRÄMLING, Kary *et al.* Product agents for handling information about physical objects. **Report of Laboratory of information processing science series B, TKO-B**, v. 153, n. 03, 2003.

FULLER, Adam *et al.* Digital twin: Enabling technologies, challenges and open research. **IEEE Access**, IEEE, v. 8, p. 108952–108971, 2020.

GABOR, Thomas *et al.* A simulation-based architecture for smart cyber-physical systems. *In: IEEE. 2016 IEEE international conference on autonomic computing (ICAC)*. [S. l.: s. n.], 2016. p. 374–379.

GAMMA, Erich *et al.* **Design Patterns: Elements of Reusable Object-Oriented Software**. [S. l.]: Addison-Wesley, 1994. ISBN 9780201633610.

GARTNER, INC. **Emerging Technologies and Trends Impact Radar 2023**. [S. l.: s. n.], 2023. Accessed: 2026-01-21. Disponível em: <https://www.gartner.com>.

GAUSHELL, Dennis J; DARLINGTON, Henry T. Supervisory control and data acquisition. **Proceedings of the IEEE**, IEEE, v. 75, n. 12, p. 1645–1658, 1987.

GOOGLE. gRPC: A high performance, open-source universal RPC framework. *In: PROCEEDINGS of the IEEE International Conference on Cloud Computing*. [S. l.: s. n.], 2016. Disponível em: <https://grpc.io/>.

GOOGLE. **Protocol Buffers: Developer Guide**. [S. l.: s. n.], 2025. Disponível em: <https://developers.google.com/protocol-buffers>.

GRIEVES, Michael. Product lifecycle management: driving the next generation of lean thinking. **(No Title)**, 2006.

GRIEVES, Michael; VICKERS, John. Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. *In: TRANSDISCIPLINARY perspectives on complex systems: New findings and approaches*. [S. l.]: Springer, 2016. p. 85–113.

GRÜNER, Sten; PFROMMER, Julius; PALM, Florian. RESTful industrial communication with OPC UA. **IEEE Transactions on Industrial Informatics**, IEEE, v. 12, n. 5, p. 1832–1841, 2016.

GUTIERREZ, Rommel; VILLEGAS-CH, William; GOVEA, Jaime. Modular middleware for IoT: scalability, interoperability and energy efficiency in smart campus. **Frontiers in Communications and Networks**, Frontiers Media SA, v. 6, p. 1672617, 2025.

HAJNORUZI, Farhad. **Design and Implementation of a Microservices-Based Digital Twin Architecture**. 2023. Diss. (Mestrado) – NTNU.

HARTIG, Olaf; PÉREZ, Jorge. Semantics and Complexity of GraphQL. *In: PROCEEDINGS of the 2019 World Wide Web Conference (WWW)*. [S. l.: s. n.], 2019. p. 1156–1167. DOI: [10.1145/3308558.3313616](https://doi.org/10.1145/3308558.3313616).

HIETALA, Jani *et al.* **Real-time two-way data transfer with a digital twin via web interface**. 2020. Diss. (Mestrado).

HIETALA, Jani *et al.* GraphQL interface for OPC UA. *In: IEEE. 2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. [S. l.: s. n.], 2020. v. 1, p. 149–155.

HRIBERNIK, Karl *et al.* A concept for product-instance-centric information management. *In: IEEE. 2005 IEEE International Technology Management Conference (ICE)*. [S. l.: s. n.], 2005. p. 1–8.

HRIBERNIK, Karl A *et al.* The product avatar as a product-instance-centric information management concept. **International Journal of Product Lifecycle Management**, Inderscience Publishers, v. 1, n. 4, p. 367–379, 2006.

INDÚSTRIA, Confederação Nacional da. **Saiba como empresas brasileiras investem na transformação digital para aumentar a produtividade**. Accessed: 2026-01-21. 2023. Disponível em: <https://noticias.portaldaindustria.com.br/noticias/inovacao-e-tecnologia/saiba-como-empresas-brasileiras-investem-na-transformacao-digital-para-aumentar-a-produtividade/>.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO 23247-1:2021 – Automation Systems and Integration – Digital Twin Framework for Manufacturing – Part 1: Overview and General Principles.**

Geneva, Switzerland: [s. n.], out. 2021. Acesso em: 11 ago. 2026. Disponível em: <https://www.iso.org/standard/75066.html>.

ISO/IEC/IEEE 42010:2011 - Systems and software engineering — Architecture description. Geneva, Switzerland: [s. n.], 2011. Available at: <https://www.iso.org/standard/50508.html>.

ITANI, May; SHARAFEDDINE, Sanaa. Toward industrial metaverse: a high fidelity digital twin for IIoT with optimized age-accuracy tradeoff. **IEEE Access**, IEEE, v. 12, p. 104242–104253, 2024.

IYENGAR, J.; THOMSON, M. **HTTP/3**. [S. l.], 2022. Disponível em: <https://www.rfc-editor.org/rfc/rfc9114.html>.

KHANSARI, Mina Emami; SHARIFIAN, Saeed. A scalable modified deep reinforcement learning algorithm for serverless IoT microservice composition infrastructure in fog layer. **Future Generation Computer Systems**, Elsevier, v. 153, p. 206–221, 2024.

KLABNIK, Steve; NICHOLS, Carol. **The Rust Programming Language**. 2. ed. [S. l.]: No Starch Press, 2021. ISBN 9781718503106.

KLETTI, Jürgen. **Manufacturing Execution Systems—MES**. [S. l.]: Springer, 2007.

KÜNZEL, Anne *et al.* Modular and flexible Automation Middleware based on LabVIEW and OPC UA. *In*: IOP PUBLISHING, 1. IOP Conference Series: Materials Science and Engineering. [S. l.: s. n.], 2021. v. 1193, p. 012109.

LAAKI, Heikki; MICHE, Yoan; TAMMI, Kari. Prototyping a digital twin for real time remote control over mobile networks: Application of remote surgery. **Ieee Access**, IEEE, v. 7, p. 20325–20336, 2019.

LAMNAOUR, Mohammed *et al.* A semantic-based middleware for supporting heterogeneity and context-awareness within IoT applications. **J. Commun**, v. 19, n. 1, p. 19–27, 2024.

LIANG, Wen-Yew; YUAN, Yuyuan; LIN, Hsiang-Jui. A performance study on the throughput and latency of zenoh, mqtt, kafka, and dds. **arXiv preprint arXiv:2303.09419**, 2023.

- LIAU, Y; LEE, H; RYU, K. Digital Twin concept for smart injection molding. *In: IOP PUBLISHING. IOP Conference Series: Materials Science and Engineering*. [S. l.: s. n.], 2018. v. 324, p. 012077.
- LU, Yang. Industry 4.0: A survey on technologies, applications and open research issues. **Journal of industrial information integration**, Elsevier, v. 6, p. 1–10, 2017.
- MAHNKE, Wolfgang; LEITNER, Stefan-Helmut; DAMM, Matthias. **OPC unified architecture**. [S. l.]: Springer Science & Business Media, 2009.
- MANKINS, John C. **Technology Readiness Levels: A White Paper**. [S. l.], 1995. Available at: <https://ntrs.nasa.gov/citations/19950022575>.
- MARKETSANDMARKETS. **Digital Twin Market by Application, Industry, and Region - Global Forecast to 2028**. [S. l.: s. n.], 2023. Accessed: 2026-01-21. Disponível em: <https://www.marketsandmarkets.com/Market-Reports/digital-twin-market-225269522.html>.
- MELIK-MERKUMIANS, Martin *et al.* Towards OPC UA as portable SOA middleware between control software and external added value applications. *In: IEEE. PROCEEDINGS of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. [S. l.: s. n.], 2012. p. 1–8.
- MIHAI, Stefan *et al.* Digital twins: A survey on enabling technologies, challenges, trends and future prospects. **IEEE Communications Surveys & Tutorials**, IEEE, v. 24, n. 4, p. 2255–2291, 2022.
- MINISTRY OF INDUSTRY AND INFORMATION TECHNOLOGY OF CHINA. **China's Digital Twin Strategy for Smart Manufacturing**. [S. l.: s. n.], 2023. Accessed: 2026-01-21. Disponível em: <https://www.miit.gov.cn>.
- MUNOZ, Paula. Measuring the fidelity of digital twin systems. *In: PROCEEDINGS of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. [S. l.: s. n.], 2022. p. 182–188.
- NAGOWAH, Soulakshmee D; STA, Hatem Ben; GOBIN-RAHIMBUX, Baby. Modelling Sustainability for an IoT-enabled Smart Green Campus using an Ontology-based Approach. *In: WOP@ ISWC*. [S. l.: s. n.], 2023. p. 62–73.
- NAIK, Nitin. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *In: IEEE. 2017 IEEE international systems engineering symposium (ISSE)*. [S. l.: s. n.], 2017. p. 1–7.

OBJECT MANAGEMENT GROUP, INC. **Digital Twin Consortium Home**. [S. l.: s. n.], 2024. <https://www.digitaltwinconsortium.org/>. Acesso em: 11 ago. 2026.

OPC FOUNDATION. **OPC Unified Architecture Specification**. [S. l.], 2017. Version 1.04. Acessado em: 2025-11-08. Disponível em: <https://opcfoundation.org>.

OPC FOUNDATION. **OPC Unified Architecture Specification**. [S. l.], 2017. Version 1.04. Acessado em: 2025-11-08. Disponível em: <https://reference.opcfoundation.org/Core/Part1/v104/docs/6.3.1>.

OSTERMAIER, Benedikt; SCHLUP, Fabian; RÖMER, Kay. Webplug: A framework for the web of things. *In: IEEE. 2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. [S. l.: s. n.], 2010. p. 690–695.

PEREIRA, Lucas Silva *et al.* Enabling 6G digital twin ecosystems through middleware-based synchronization and management. **Computer Communications**, Elsevier, p. 108554, 2026.

PESSOA, Marcosiris AO *et al.* Industry 4.0, how to integrate legacy devices: a cloud IoT approach. *In: IEEE. IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. [S. l.: s. n.], 2018. p. 2902–2907.

PREVU3D. **Understanding Digital Twins: How They Work and Why They Matter**. Blog & News, Prevu3D. Jan. 2025. Disponível em: <https://www.prevu3d.com/news/understanding-digital-twins/>. Acesso em: 12 out. 2025.

PRIYANTHA, Bodhi *et al.* Tiny web services for sensor device interoperability. *In: IEEE. 2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*. [S. l.: s. n.], 2008. p. 567–568.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services**. [S. l.]: O'Reilly Media, 2007.

RIZANO, Tizar; ABENI, Luca; PALOPOLI, Luigi. Experimental evaluation of the real-time performance of publish-subscribe middlewares. Universidad Carlos III de Madrid, 2013.

RODRÍGUEZ-ALONSO, Carlos; PENA-REGUEIRO, Iván; GARCÍA, Óscar. Digital twin platform for water treatment plants using microservices architecture. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 24, n. 5, p. 1568, 2024.

ROLLE, R; MARTUCCI, V; GODOY, E. Architecture for Digital Twin implementation focusing on Industry 4.0. **IEEE Latin America Transactions**, IEEE, v. 18, n. 05, p. 889–898, 2020.

ROSS, Keith W; KUROSE, James F. **Computer networking**. [S. l.]: Pearson Education London, UK, 2012.

SANTOS, Lucas dos; JÚNIOR, Erisvaldo. Simulador híbrido de ponte de rolante. *In*: I SPDE Seminário de Pesquisa e Desenvolvimento: 1º Encontro dos Institutos SENAI e do Centro de Inovação SESI. Rio de Janeiro: Firjan SENAI SESI, 2024. (Estudos e Pesquisas – Anais Eletrônicos), p. 45–47. Artigos-resumo apresentados no I SPDE, realizado entre os dias 16 e 22 de outubro de 2023.

SANTOS, Lucas S dos; MARIA, Luiz CG *et al.* Hybrid Simulation Framework: Advanced and Secure Training, p. 13–24, 2024.

SANTOS, Lucas Severiano dos. **Monitoramento de Ambiente Através de uma Rede Oportunística Centrada em Interesse**. Fev. 2020. Dissertação de Mestrado – Universidade Federal do Rio de Janeiro (UFRJ), COPPE, Rio de Janeiro.

SCHALKWYK, Pieter van. **Digital Twin Capabilities Periodic Table: A Digital Twin Consortium User Guide**. [S. l.: s. n.], 2022. Acesso em: 11 ago. 2026. Disponível em: [https://github.com/digitaltwinconsortium/capabilities-toolkit/blob/main/digital-twin-cpt/user\\_guide.pdf](https://github.com/digitaltwinconsortium/capabilities-toolkit/blob/main/digital-twin-cpt/user_guide.pdf).

SCHIEKOFER, Rainer; SCHOLZ, Andreas; WEYRICH, Michael. REST based OPC UA for the IIoT. *In*: IEEE. 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA). [S. l.: s. n.], 2018. v. 1, p. 274–281.

SCHROEDER, Georg *et al.* Digital Twin Data Modeling with AutomationML and a Communication Methodology for Data Exchange. **IFAC-PapersOnLine**, Elsevier, v. 49, n. 30, p. 12–17, 2016.

SHAFTO, Michael *et al.* The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles. *In*: 53RD AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference. [S. l.: s. n.], 2012.

SHAFTO, Michael *et al.* **Modeling, Simulation, Information Technology & Processing Roadmap**. [S. l.], 2010. NASA Technology Roadmaps TA11.

SILVA, Michael DB; MORAES, Heberte F; AMORIM, Claudio L. Persistência de Dados em Redes Móveis Ad-hoc Centradas em Interesses. *In*: SBC. SIMPÓSIO Brasileiro de Sistemas Multimídia e Web (WebMedia). [S. l.: s. n.], 2013. p. 77–80.

SINGH, Maulshree *et al.* Digital twin: Origin to future. **Applied Sciences**, MDPI, v. 12, n. 19, p. 9798, 2022.

SOARES, HEDER DORNELES. UMA ARQUITETURA ORIENTADA A SERVIÇOS DE NUVEM PARA APOIAR TESTES EXPERIMENTAIS DE SISTEMAS UBÍQUOS, 2021.

TANENBAUM, A; WETHERALL, D *et al.* Computer networks title: Computer networking: A top-down approach. **Instructor**, v. 201901, 2019.

TANENBAUM, Andrew S.; BOS, Herbert. **Modern Operating Systems**. 4. ed. [S. l.]: Pearson, 2014. ISBN 9780133591620.

TAO, Fei *et al.* Digital twin and its applications in the context of Industry 4.0. **IEEE Transactions on Industrial Informatics**, IEEE, v. 19, n. 1, p. 585–595, 2023.

TRAVIS, Jeffrey; KRING, Jim. **LabVIEW for everyone: Graphical programming made easy and fun (National instruments virtual instrumentation series)**. [S. l.]: Prentice Hall PTR, 2006.

TU, Xinyi *et al.* A mixed reality interface for a digital twin based crane. **Applied Sciences**, MDPI, v. 11, n. 20, p. 9480, 2021.

VAN BRUSSEL, Hendrik *et al.* Reference architecture for holonic manufacturing systems: PROSA. **Computers in industry**, Elsevier, v. 37, n. 3, p. 255–274, 1998.

VAN ROSSUM, Guido; DRAKE, Fred L. **Python Tutorial**. [S. l.]: Centrum voor Wiskunde en Informatica (CWI), 1995.

WANG, Bo; ZHANG, Lei; XU, Jian. A Joint Communication and Computation Framework for Digital Twin over Wireless Networks. **IEEE Transactions on Industrial Informatics**, v. 20, n. 5, p. 4123–4134, 2024.

WONG, Chien Yaw *et al.* The intelligent product driven supply chain. *In*: IEEE. IEEE international conference on systems, man and cybernetics. [S. l.: s. n.], 2002. v. 4, 6–pp.

WOOLDRIDGE, Michael; JENNINGS, Nicholas R. Intelligent agents: Theory and practice. **The knowledge engineering review**, Cambridge University Press, v. 10, n. 2, p. 115–152, 1995.

ZHANG, Shun; WANG, Qiang; ZHANG, Min. OPC UA-based communication framework for industrial internet of things. **IEEE Access**, IEEE, v. 8, p. 206639–206651, 2020.

ZHANG, Wei; MÜLLER, Thomas; KIM, Yuna. How to synchronize Digital Twins? A Communication Performance Analysis. **Journal of Industrial Information Integration**, Elsevier, v. 30, p. 100395, 2024.

ZHENG, Yu; YANG, Sen; CHENG, Huanchong. An application framework of digital twin and its case study. **Journal of ambient intelligence and humanized computing**, Springer, v. 10, n. 3, p. 1141–1153, 2019.

ZIMMERMANN, Hubert. OSI reference model-the ISO model of architecture for open systems interconnection. **IEEE Transactions on communications**, IEEE, v. 28, n. 4, p. 425–432, 1980.

## **APÊNDICE A - Diagrama de Classes**

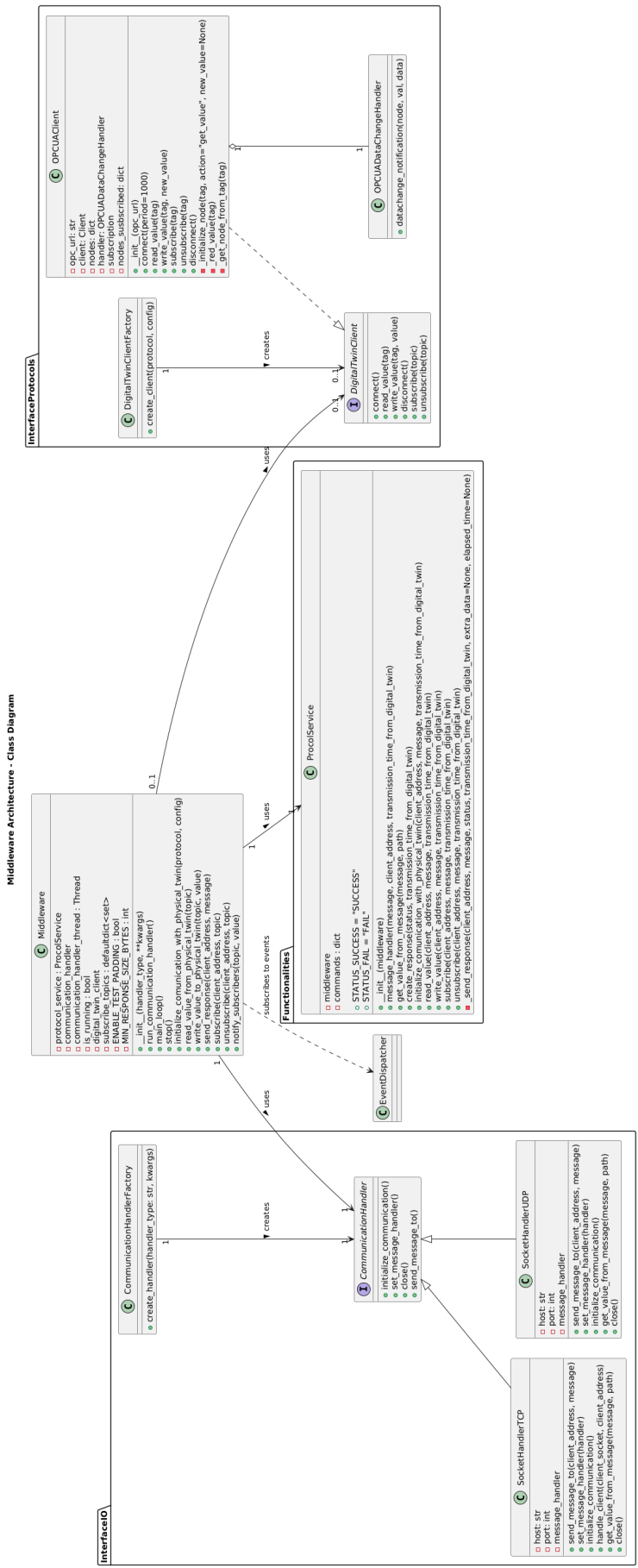


Figura 53: Diagrama de Classes com os métodos.

## APÊNDICE B - Especificação das Mensagens da API

Este apêndice apresenta a especificação detalhada das mensagens utilizadas pelo middleware. Para cada operação, são descritos os campos obrigatórios e opcionais da requisição, os campos retornados na resposta e exemplos completos em formato JSON. Salvo indicação em contrário, todas as mensagens seguem a estrutura padrão de envelope, composta por cabeçalho (*header*) e corpo da mensagem (*body*).

### B.1 Inicializar comunicação (init)

**Requisição (mínima):** header{request\_id, command=init, timestamp, protocol}, body{endpoint[, auth]}.

```

1 {
2   "header": {
3     "timestamp": 1758395433.673463,
4     "client_address": "127.0.0.1",
5     "command": "INITIALIZED_COMMUNICATION_WITH_PHYSICAL_TWIN"
6   },
7   "data": {
8     "protocol": "OPC-UA",
9     "opc_url": "opc.tcp://localhost:4840/freeopcua/server/"
10  }
11 }
```

Listing B.1: init: resposta de inicialização

**Resposta (típica):** header{request\_id, status, timestamp\_response}, body{data.conne

```

1 {
```

```
2  "header": {
3    "status": "SUCCESS",
4    "timestamp_response": 1758395742.990788,
5    "transmission_time_from_digital_twin": 1.1599063873291016,
6    "timestamp_request": 1758395433.673463
7  },
8  "data": {}
9 }
```

Listing B.2: response: resposta de sucesso

## B.2 Ler valor (read)

**Requisição:** body{variable}.

```
1  {
2    "header": {
3      "timestamp": 1758395829.108751,
4      "client_address": "127.0.0.1",
5      "command": "READ_VALUE"
6    },
7    "data": {
8      "topic": "SIM_Roll_Velocidade"
9    }
10 }
```

Listing B.3: read: requisição de leitura de valor

**Resposta:** body{variable, data.value[, data.unit]}.

```
1  {
2    "header": {
3      "status": "SUCCESS",
4      "timestamp_response": 1758395878.531502,
5      "transmission_time_from_digital_twin": 45649.89709854126,
6      "timestamp_request": 1758395829.108751,
7      "elapsed_time_physical_twin": 5.380868911743164,
8      "command": "READ_VALUE"
9    },
```

```
10  "data": {
11    "topic": "SIM_Roll_Velocidade",
12    "value": 10
13  }
14 }
```

Listing B.4: read: resposta de leitura de valor

## B.3 Escrever valor (write)

**Requisição:** body{variable, value}.

```
1  {
2    "header": {
3      "timestamp": 1758395977.2144058,
4      "client_address": "127.0.0.1",
5      "command": "WRITE_VALUE"
6    },
7    "data": {
8      "topic": "SIM_Roll_Velocidade",
9      "value": 50
10   }
11 }
```

Listing B.5: write: requisição de escrita de valor

**Resposta:** body{variable, data.acknowledged}.

```
1  {
2    "header": {
3      "status": "SUCCESS",
4      "timestamp_response": 1758396044.603211,
5      "transmission_time_from_digital_twin": 8.355140686035156,
6      "timestamp_request": 1758395977.2144058,
7      "elapsed_time_physical_twin": 0.0,
8      "command": "WRITE_VALUE"
9    },
10   },
11   "data": {
```

```
12     "topic": "SIM_Roll_Velocidade",
13     "value": 50
14 }
15 }
```

Listing B.6: write: resposta de escrita de valor

## B.4 Assinar tópico (subscribe)

Requisição: body{topic}.

```
1 {
2   "header": {
3     "timestamp": 1758488117.088886,
4     "client_address": "127.0.0.1",
5     "command": "SUBSCRIBE"
6   },
7   "data": {
8     "topic": "SIM_Roll_Velocidade"
9   }
10 }
```

Listing B.7: subscribe: requisição de inscrição em tópico

```
1 {
2   "header": {
3     "status": "SUCCESS",
4     "timestamp_response": 1758488219.036683,
5     "transmission_time_from_digital_twin": null,
6     "command": "SUBSCRIBE"
7   },
8   "data": {
9     "topic": "SIM_Roll_Velocidade"
10  }
11 }
```

Listing B.8: subscribe: resposta de inscrição em tópico

## B.5 Cancelar assinatura (unsubscribe)

**Requisição:** body{topic}.

```
1 {  
2   "header": {  
3     "timestamp": 1758488443.7187111,  
4     "client_address": "127.0.0.1",  
5     "command": "UNSUBSCRIBE"  
6   },  
7   "data": {  
8     "topic": "SIM_Roll_Velocidade"  
9   }  
10 }
```

Listing B.9: unsubscribe: requisição de cancelamento de inscrição em tópico

```
1 {  
2   "header": {  
3     "status": "SUCCESS",  
4     "timestamp_response": 1758488219.036683,  
5     "transmission_time_from_digital_twin": null,  
6     "command": "UNSUBSCRIBE"  
7   },  
8   "data": {  
9     "topic": "SIM_Roll_Velocidade"  
10  }  
11 }
```

Listing B.10: subscribe: resposta de cancelamento de inscrição em tópico

```
1 {  
2   "header": {  
3     "status": "SUCCESS",  
4     "timestamp_response": 1758489500.536115,  
5     "transmission_time_from_digital_twin": null,  
6     "command": "Notify"  
7   },  
8   "data": {  
9     "topic": "SIM_Roll_Velocidade",  
10  }  
11 }
```

```
10     "value": 10
11   }
12 }
```

Listing B.11: `notify`: mensagem de notificação de valor

## APÊNDICE C – Configuração do Physical Twin para o Ambiente Experimental

O objetivo deste apêndice é descrever o procedimento de instalação e configuração do ambiente experimental utilizado neste trabalho, com ênfase na emulação do PT e na preparação da infraestrutura necessária para a execução do *Middleware*. O ambiente foi projetado com o propósito de reproduzir, em condições controladas de laboratório, um cenário representativo de operação de sistemas DT, permitindo a avaliação funcional e de desempenho da arquitetura proposta.

Em particular, este apêndice detalha o processo de instalação do sistema operacional no dispositivo *Raspberry Pi*. O *Raspberry Pi* foi configurado para atuar como servidor do OPC UA. Essa abordagem possibilita a simulação de diferentes padrões de interação entre o *Middleware* e o sistema físico, sem a necessidade de equipamentos industriais dedicados.

Além disso, são apresentados os procedimentos básicos para a instalação e execução do *Middleware* na plataforma de testes, incluindo dependências de software, organização do ambiente de execução e inicialização dos serviços necessários. O detalhamento dessas etapas visa garantir a reprodutibilidade dos experimentos conduzidos neste trabalho, permitindo que o ambiente experimental seja reconstruído por outros pesquisadores ou adaptado a cenários semelhantes.

### C.1 Procedimento de Instalação do Sistema Operacional no Cartão SD

Esta seção descreve, de forma sistemática, o procedimento adotado para instalação do sistema operacional no cartão SD utilizado no *Raspberry Pi*, considerando as versões mais recentes do ecossistema oficial da plataforma. O procedimento foi definido de modo a garantir reprodutibilidade do ambiente experimental e padronização da configuração inicial do dispositivo.

## Passo 1 — Obtenção da Ferramenta de Instalação

O primeiro passo consiste no download da ferramenta oficial *Raspberry Pi Imager*, disponibilizada pela *Raspberry Pi Foundation*. A ferramenta pode ser obtida no endereço:

<https://www.raspberrypi.com/software/>

O *Raspberry Pi Imager* está disponível para os sistemas operacionais Linux, Windows e macOS. Após o download, a ferramenta deve ser instalada seguindo o procedimento padrão do sistema operacional hospedeiro. A Figura ?? apresenta o site para download do *Raspberry Pi Imager*.

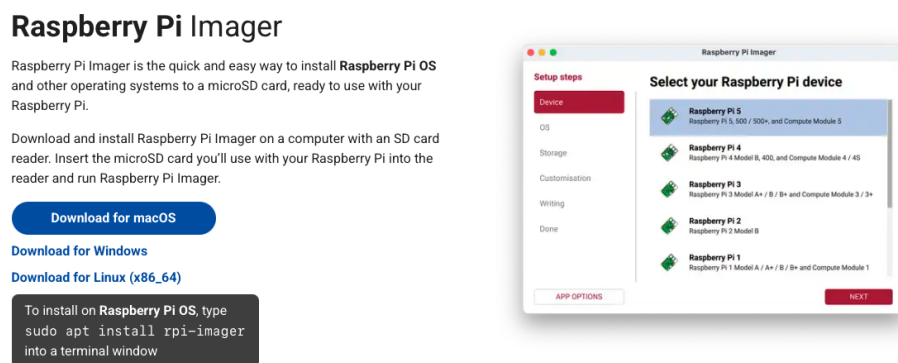


Figura 54: Download do *Raspberry Pi Imager*.

## Passo 2 — Preparação do Cartão SD

Um cartão de memória do tipo SD ou microSD, com capacidade mínima recomendada de 16 GB, deve ser inserido no computador hospedeiro por meio de um adaptador compatível. Recomenda-se que o cartão esteja previamente formatado ou sem dados relevantes, uma vez que o processo de instalação sobrescreve completamente seu conteúdo.

## Passo 3 — Seleção do Sistema Operacional

Com o *Raspberry Pi Imager* em execução, devem ser realizadas as seguintes seleções:

1. Selecionar o dispositivo alvo (*Raspberry Pi 4*).
2. Selecionar o sistema operacional *Raspberry Pi OS (64-bit)*, versão estável mais recente.

3. Selecionar o cartão SD como dispositivo de destino.

A escolha da versão 64 bits do sistema operacional visa melhor aproveitamento dos recursos de hardware e maior compatibilidade com aplicações modernas. A Figura 55 exibe o *Raspberry Pi Imager* em execução com as opções selecionadas.



Figura 55: Raspberry Pi Imager.

## Passo 4 — Configurações Avançadas de Inicialização

Antes de iniciar a gravação da imagem, deve-se acessar o menu de configurações avançadas do *Raspberry Pi Imager*. Essa opção fica disponível após clicar em "NEXT", conforme ilustrado na Figura 56.

Após selecionar esse opção será exibida a interface de configurações avançadas, conforme ilustrada na Figura 57.

Nesse menu, são definidos os seguintes parâmetros:

- Nome do dispositivo (*hostname*);
- Criação do usuário padrão e definição de senha;
- Configuração da rede sem fio (SSID e senha), quando aplicável;

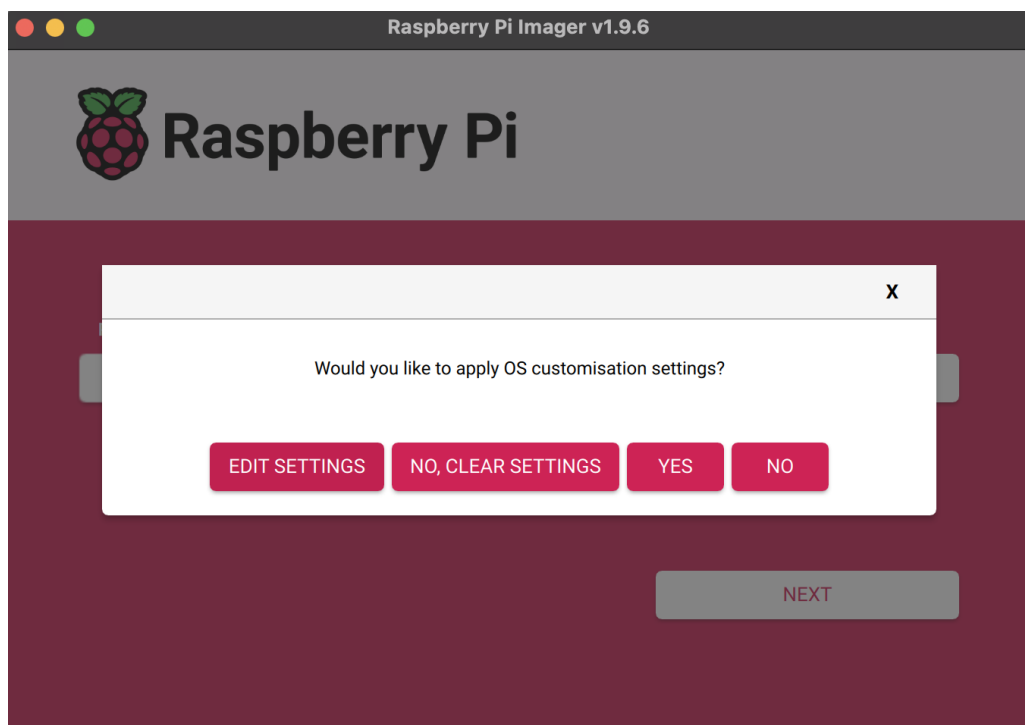


Figura 56: Opção de configurações avançadas.

- Habilitação do acesso remoto via *SSH*.

A definição prévia dessas configurações permite que o dispositivo seja acessado remotamente após o primeiro *boot*, sem a necessidade de monitor, teclado ou mouse.

## Passo 5 — Gravação da imagem no cartão SD

Após a configuração dos parâmetros, o processo de gravação da imagem é iniciado. O *Raspberry Pi Imager* realiza automaticamente:

1. O download da imagem do sistema operacional selecionado;
2. A gravação da imagem no cartão SD;
3. A verificação de integridade dos dados gravados.

## Passo 6 — Inicialização do Raspberry Pi

O cartão SD configurado deve ser inserido no *Raspberry Pi*. Em seguida, o dispositivo deve ser energizado. Durante a primeira inicialização, o sistema operacional realiza procedimentos automáticos de expansão do sistema de arquivos e aplicação das configurações previamente definidas.

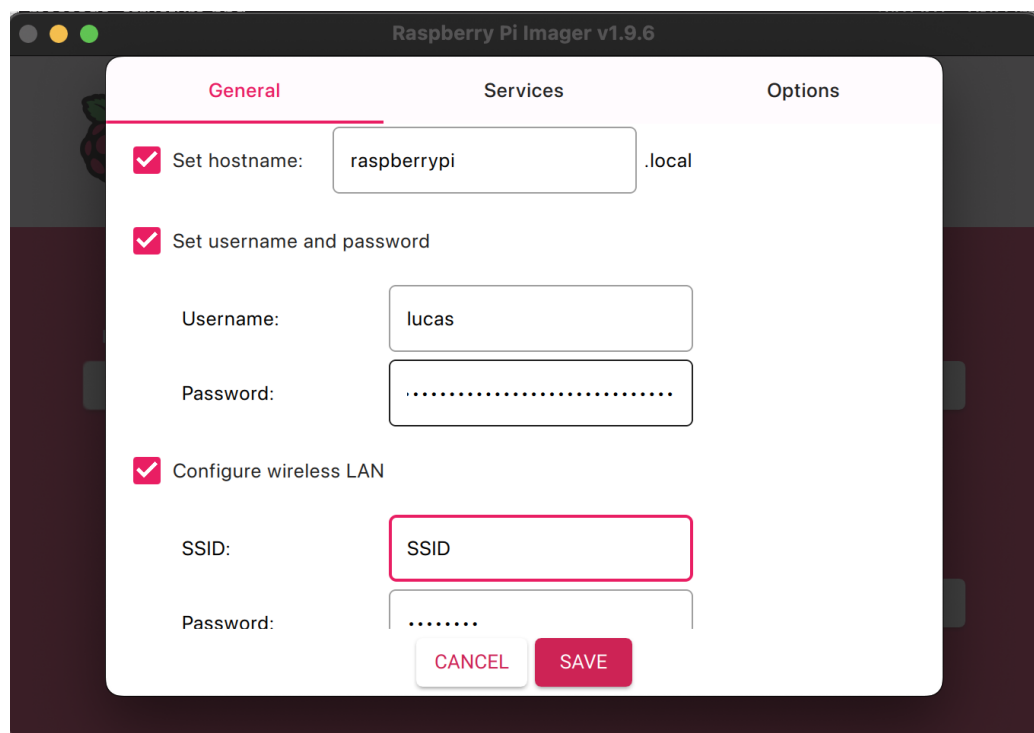


Figura 57: Configurações avançadas.

Após a conclusão do processo de inicialização, o sistema operacional encontra-se pronto para uso, possibilitando a execução das etapas subsequentes de configuração do ambiente experimental, incluindo a instalação dos serviços de comunicação industrial e do *Middleware*.

## APÊNDICE D - Procedimentos de Instalação e Execução do Middleware Digital Twin

O objetivo deste apêndice é descrever, de forma sistemática e reproduzível, os procedimentos de obtenção, instalação, configuração e execução do *Middleware Digital Twin* utilizado na prova de conceito apresentada neste trabalho. Diferentemente dos capítulos principais da tese, que abordam aspectos conceituais, arquiteturais e metodológicos, este apêndice tem caráter operacional, servindo como material de apoio à reprodução do ambiente experimental.

O middleware descrito neste apêndice atua como a aplicação central responsável por gerenciar a comunicação entre o **PT** e o **DT**, viabilizando a troca de dados e comandos de forma desacoplada, conforme a arquitetura proposta no Capítulo 4. Os procedimentos apresentados a seguir contemplam desde a obtenção do código-fonte até a execução do sistema em ambiente local ou embarcado, como o Raspberry Pi.

### D.1 Obtenção do Código-Fonte

O código-fonte do middleware encontra-se disponível em um repositório público hospedado na plataforma GitLab. Para obtê-lo, deve-se executar o comando de clonagem do repositório:

```
1 git clone https://gitlab.com/lseveriano/middleware_dt
```

Listing D.1: Clonagem do repositório do Middleware Digital Twin

Após a execução desse comando, será criado localmente um diretório contendo a estrutura completa do projeto, incluindo os módulos do *Physical Twin*, do *Middleware* e do *Digital Twin*.

## D.2 Estrutura do Projeto

O projeto está organizado de forma modular, refletindo diretamente os princípios arquiteturais adotados neste trabalho. Os principais módulos são:

- **Physical Twin:** contém a simulação dos componentes físicos do sistema e implementações de protocolos industriais utilizados na comunicação com o middleware;
- **Middleware:** aplicação central responsável por gerenciar o fluxo de comunicação entre o [PT](#) e o [DT](#);
- **Digital Twin:** módulo utilizado para simular chamadas ao middleware, permitindo testes funcionais e validação da comunicação.

## D.3 Requisitos de Software

Para a execução do middleware, é necessário que o ambiente possua previamente instalados:

- Python versão 3.13.2;
- Gerenciador de pacotes `pip`.

Recomenda-se a utilização de um ambiente virtual para isolamento das dependências do projeto, especialmente em ambientes compartilhados ou embarcados.

## D.4 Instalação das Dependências

As bibliotecas necessárias para a execução do middleware estão listadas no arquivo `requirements.txt` localizado no diretório raiz do projeto. A instalação das dependências deve ser realizada por meio do comando:

```
1 pip install -r requirements.txt
```

Listing D.2: Instalação das dependências do Middleware

Esse procedimento assegura que todas as bibliotecas utilizadas na implementação estejam disponíveis no ambiente de execução.

## D.5 Configuração do Middleware

O middleware utiliza um arquivo de configuração no formato JSON para definir seus parâmetros de inicialização e operação. O arquivo `config.json` deve ser ajustado conforme o ambiente experimental. Um exemplo de configuração é apresentado a seguir:

```
1 {  
2     "protocol_middleware_communication_handler": "socket",  
3     "socket_host_middleware": "127.0.0.1",  
4     "socket_port_middleware": 5000,  
5     "memory_size": 1024  
6 }
```

Listing D.3: Exemplo de arquivo de configuração do Middleware

Esses parâmetros definem o mecanismo de comunicação utilizado pela API do middleware, o endereço de rede e a porta de escuta, além de configurações internas relacionadas ao gerenciamento de memória.

## D.6 Execução do Middleware

Após a instalação das dependências e a configuração do arquivo `config.json`, o middleware pode ser iniciado por meio da execução do arquivo principal do projeto:

```
1 python main.py
```

Listing D.4: Execução do Middleware

Para apoiar no processo de inicialização foram desenvolvidos os *scripts* de execução:

- **middleware.sh**: Responsável por iniciar o middleware.
- **opcua.sh**: Responsável por inicializar o emulador do servidor [OPC UA](#).
- **mqtt.sh**: Responsável por inicializar o emulador do servidor [MQTT](#).