UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Lucas Nicoletti

Desenvolvimento do driver de comunicação MQTT para a plataforma FrameworX

Blumenau
2025

Lucas Nicoletti

Desenvolvimento do driver de comunicação MQTT para a plataforma FrameworX

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheiro de Controle e Automação.
Orientador: Prof. Dr Alex Fabiano Bueno.

Blumenau
2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Nicoletti, Lucas
Desenvolvimento do driver de comunicação MQTT para a plataforma FrameworX / Lucas Nicoletti ; orientador, Alex Fabiano Bueno, 2025.
44 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Graduação em Engenharia de Controle e Automação, Blumenau, 2025.

Inclui referências.

1. Engenharia de Controle e Automação. 2. MQTT. 3. Driver de Comunicação. 4. Sistemas SCADA. 5. FrameworX. I. Bueno, Alex Fabiano. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Lucas Nicoletti

Desenvolvimento do driver de comunicação MQTT para a plataforma FrameworX

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Engenheiro de Controle e Automação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 15 de Dezembro de 2025.

Banca Examinadora:

Documento assinado digitalmente
ALEX FABIANO BUENO
Data: 19/12/2025 18:40:39-0300
CPF: ***.810.780-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Alex Fabiano Bueno
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Ciro Andre Pitz
Data: 17/12/2025 20:52:48-0300
CPF: ***.728.089-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Ciro André Pitz
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 18/12/2025 08:12:35-0300
CPF: ***.999.369-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Carlos Roberto Moratelli
Universidade Federal de Santa Catarina

À minha família,
por tudo que me ensinaram
e por sempre acreditarem em mim.

À minha namorada,
por todo o amor que me dá
e por ser minha inspiração a continuar.

AGRADECIMENTOS

Aos meus pais e irmã, que sempre estiveram ao meu lado e me apoiaram em todas as escolhas que tomei até chegar aqui.

À minha companheira Daniele, por todo seu amor e apoio sempre que precisei.

Ao meu orientador Professor Dr Alex Fabiano Bueno, que além de ser um grande professor, sempre foi um bom amigo.

Aos meus amigos Chen, Eduarda, Lucas, Mateus e Valdir, pela companhia e risadas nas horas intermináveis de estudo que me trouxeram até aqui.

Eu ouço e esqueço. Eu vejo e lembro. Eu faço e entendo (Confúcio)

RESUMO

O presente trabalho aborda o desenvolvimento e a implementação de um *driver* de comunicação em linguagem C#, baseado no protocolo MQTT (*Message Queuing Telemetry Transport*), para integração com a plataforma SCADA (*Supervisory Control And Data Acquisition*) FrameworX. Inserido no contexto da Indústria 4.0 e da Internet das Coisas Industrial (IIoT), o projeto visa superar as limitações de protocolos legados baseados em *polling* ao adotar a arquitetura *publish-subscribe* do MQTT, que oferece maior escalabilidade, eficiência de banda e comunicação orientada a eventos. A solução, integrada ao FrameworX, demonstrou-se funcional e confiável, permitindo a troca de dados em tempo real entre os *tags* do sistema SCADA e clientes MQTT externos, com os testes validando a correta implementação dos mecanismos de *Quality of Service* (QoS) e gerenciamento de sessão. O projeto evidencia a viabilidade de desenvolver *drivers* de comunicação customizados e de alto desempenho para a plataforma FrameworX, reforçando sua flexibilidade para atender às demandas de sistemas de automação modernos.

Palavras-chave: MQTT; Driver de Comunicação; Sistemas SCADA; FrameworX;

ABSTRACT

This work addresses the development and implementation of a communication *driver* in C# language, based on the MQTT (*Message Queuing Telemetry Transport*) protocol, for integration with the SCADA (*Supervisory Control And Data Acquisition*) FrameworX platform. Set in the context of Industry 4.0 and the Industrial Internet of Things (IIoT), the project aims to overcome the limitations of legacy *polling*-based protocols, such as Modbus, by adopting the MQTT's *publish-subscribe* architecture, which offers greater scalability, bandwidth efficiency, and event-driven communication. The solution, integrated into FrameworX, proved to be functional and reliable, allowing for real-time data exchange between the SCADA system's *tags* and external MQTT clients, with tests validating the correct implementation of Quality of Service (QoS) and session management mechanisms. The project highlights the feasibility of developing custom, high-performance communication *drivers* for the FrameworX platform, reinforcing its flexibility to meet the demands of modern automation systems.

Keywords: MQTT; Communication Driver; SCADA; FrameworX.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ADU	Application Data Unit
ASCII	American Standard Code for Information Interchange
CLP	Controlador Lógico Programável
CRC	Cyclic Redundancy Check
DCS	Distributed Control Systems
DLL	Dynamic-Link Library
E/S	Entradas e Saídas
IANA	Internet Assigned Numbers Authority
IHM	Interface Homem-Máquina
IP	Internet Protocol
ISO	International Organization for Standardization
MBAP	Modbus Application Protocol Header
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
QOS	Quality of Service
RTU	Remote Terminal Unit
SCADA	Supervisory Control And Data Acquisition
SMS	Short Message Service
TCP	Transmission Control Protocol
TID	Transaction Identifier
WPF	Windows Presentation Foundation

SUMÁRIO

1 INTRODUÇÃO

A Quarta Revolução Industrial, ou Indústria 4.0, impulsionou a necessidade de protocolos de comunicação mais eficientes, escaláveis e preparados para a *Industrial Internet of Things* (IIoT) (Schlemitz; Mezhuyev, 2024; Sethi; Sarangi, 2017). Neste cenário, o protocolo MQTT (*Message Queuing Telemetry Transport*) se destaca por sua arquitetura *publish-subscribe* e sua leveza, superando as limitações de modelos tradicionais (Al-Sarawi, 2023; Ibitoye, 2025). Este trabalho apresenta a implementação de um *driver* de comunicação MQTT, desenvolvido integralmente em linguagem C#, para a plataforma SCADA FrameworX. O objetivo é criar uma ponte de comunicação moderna entre sistemas supervisórios e a crescente gama de dispositivos inteligentes no chão de fábrica. A escolha por uma implementação a partir do zero, sem o uso de bibliotecas de terceiros, reforça o caráter didático e o aprofundamento técnico do projeto, uma prática validada na literatura recente para resolução de problemas de interoperabilidade industrial (Santos; Gamboa; Rodas, 2022; Tükez; Kaya, 2022).

Historicamente, a automação industrial evoluiu de sistemas centralizados para arquiteturas distribuídas, como descrito por Capelli (Capelli, 2006), em uma busca contínua por maior flexibilidade e integração. Protocolos legados, como o Modbus TCP, baseiam-se em um modelo cliente-servidor onde um mestre realiza um ciclo de *polling*, interrogando cada dispositivo sequencialmente. Embora funcional, essa abordagem se torna ineficiente em redes com milhares de sensores, gerando tráfego excessivo, alta latência e desafios de segurança (Le et al., 2025; Rodrigues, 2022). Em contrapartida, o MQTT inverte essa lógica: os dispositivos publicam dados apenas quando necessário para um intermediário central (*broker*), que os distribui aos sistemas interessados. Essa comunicação orientada a eventos é fundamental para a viabilidade de aplicações IIoT em larga escala (Roldán-Gómez et al., 2022).

O desenvolvimento deste módulo teve como ambiente de implementação a plataforma FrameworX (Tatsoft, 2025), um sistema SCADA moderno que permite a criação de soluções industriais flexíveis e escaláveis. A escolha da linguagem C# e da tecnologia .NET se deu pela integração nativa com a arquitetura da plataforma, além de oferecer um robusto suporte para o desenvolvimento de aplicações de rede seguras e de alto desempenho, como a manipulação de *sockets* e a implementação de protocolos baseados em TCP/IP (Ekren; Sensoy; Akinci, 2025; Qayum, 2024).

O sistema SCADA, como descrito por Boyer (Boyer, 1993), atua como elo entre os dispositivos de campo e os níveis superiores de controle, oferecendo interface gráfica, armazenamento de dados históricos e geração de alarmes. A integração de um *driver* MQTT ao SCADA moderniza suas capacidades de comunicação, permitindo que ele se conecte a ecossistemas IIoT de forma nativa, recebendo e enviando dados de milhares de dispositivos de forma eficiente e em tempo real (Diaba et al., 2023).

Ao longo deste trabalho, serão abordados os fundamentos teóricos do protocolo MQTT, as especificações técnicas da plataforma FrameworX, a arquitetura de desenvolvimento em C# para o cliente e o *broker*, além dos testes realizados para a validação completa da implementação.

1.1 OBJETIVOS

O presente trabalho tem como objetivo principal a implementação de um protocolo de comunicação MQTT, desenvolvido de forma nativa em linguagem C#, e sua subsequente integração como um *driver* de comunicação na plataforma SCADA FrameworX. A proposta visa criar uma solução moderna e funcional que supere as limitações de protocolos baseados em *polling*, permitindo a comunicação orientada a eventos entre o sistema supervisório e dispositivos de IIoT.

1.1.1 Objetivo Geral

Desenvolver, utilizando a linguagem C# e a plataforma .NET, um protocolo de comunicação MQTT funcional, contendo um *broker* e um cliente, e integrá-lo à plataforma FrameworX, visando o estudo aprofundado da arquitetura *publish-subscribe* e sua aplicação prática em sistemas SCADA.

1.1.2 Objetivos Específicos

- Estudar a fundo a especificação técnica do protocolo MQTT (OASIS, 2014), com ênfase na estrutura dos pacotes de controle, nos mecanismos de Qualidade de Serviço (QoS) e no gerenciamento de sessão.
- Desenvolver uma implementação de um *Broker* MQTT em C#, capaz de gerenciar múltiplas conexões de clientes, processar inscrições em tópicos e rotear as mensagens publicadas, considerando aspectos de segurança e autenticação (Bangare; Patil, 2024).
- Desenvolver uma biblioteca de cliente MQTT em C#, sem dependência de pacotes externos, que encapsule as operações de conexão, publicação de mensagens e subscrição a tópicos.
- Criar uma aplicação de testes com interface gráfica em WPF, destinada à validação funcional e depuração da biblioteca cliente, permitindo testar a comunicação de forma isolada e controlada.
- Integrar a biblioteca cliente ao ambiente de *scripts* da plataforma FrameworX, criando um *driver* funcional que vincule os dados das mensagens MQTT aos *tags* do sistema SCADA.

- Validar a solução completa através de testes de comunicação ponta a ponta, utilizando clientes de terceiros e a ferramenta de análise de rede Wireshark para inspecionar o tráfego e confirmar a conformidade com o protocolo.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está organizado em cinco capítulos, buscando apresentar de forma clara e objetiva todas as etapas envolvidas no desenvolvimento e na implementação do protocolo MQTT na plataforma FrameworX.

O Capítulo 1 apresenta a introdução ao tema, contextualizando o projeto no cenário da Indústria 4.0, apresentando a justificativa, os objetivos gerais e específicos, e a estrutura geral do documento.

O Capítulo 2 aborda a fundamentação teórica, descrevendo os conceitos de sistemas SCADA, as arquiteturas de comunicação industrial e um aprofundamento técnico no protocolo MQTT, detalhando o modelo *publish-subscribe*, a função do *broker*, os níveis de QoS e os mecanismos de gerenciamento de estado.

O Capítulo 3 descreve a metodologia de desenvolvimento, detalhando as tecnologias utilizadas, a arquitetura de *software* da solução e as etapas de implementação do *broker* e do cliente MQTT, bem como sua integração com a plataforma FrameworX.

No Capítulo 4, são apresentados os resultados e as discussões, detalhando os testes de validação funcional realizados com a aplicação WPF e no ambiente FrameworX, os testes de confiabilidade e uma análise crítica das limitações da implementação, especialmente no que tange à segurança (Gao, 2024).

O Capítulo 5 apresenta as conclusões do trabalho, sintetizando os resultados obtidos e as contribuições do projeto. Além disso, são propostas direções para trabalhos futuros, como a implementação de recursos avançados do MQTT v5.0, a adição de camadas de segurança com TLS e a evolução para um *driver* nativo utilizando a ProtocolAPI da plataforma.

2 REVISÃO BIBLIOGRÁFICA

2.1 AUTOMAÇÃO INDUSTRIAL E A PIRÂMIDE DA AUTOMAÇÃO

A automação industrial tem evoluído significativamente, transitando de sistemas de controle isolados para arquiteturas de rede complexas e interconectadas, um movimento acelerado pela Quarta Revolução Industrial (Indústria 4.0). Essa transformação, como aponta Capelli (Capelli, 2006), é impulsionada pela busca contínua por maior flexibilidade, eficiência e capacidade de monitoramento e controle em tempo real. A estrutura hierárquica dos sistemas de automação é tradicionalmente representada pela Pirâmide da Automação, um modelo que organiza os sistemas e dispositivos em níveis distintos de funcionalidade (Tempestini, 2025).

Na base da pirâmide (Nível 0), encontram-se os dispositivos de campo, como sensores e atuadores, que interagem diretamente com o processo físico. O Nível 1, de controle, é composto por Controladores Lógicos Programáveis (CLPs) e outros dispositivos que executam a lógica de controle. O Nível 2, de supervisão, é onde operam os sistemas SCADA (*Supervisory Control and Data Acquisition*) e as Interfaces Homem-Máquina (IHMs), que, segundo Boyer (Boyer, 1993), são componentes centrais para a aquisição de dados, monitoramento visual e controle do processo por operadores humanos. Os níveis superiores (3 e 4) englobam os sistemas de gerenciamento da produção (MES) e o planejamento corporativo (ERP), respectivamente (Schlemitz; Mezhuyev, 2024). A comunicação eficaz entre esses níveis é o que viabiliza a operação integrada de uma planta industrial.

2.2 MODELO DE REFERÊNCIA OSI E A PILHA TCP/IP

Para que dispositivos de diferentes fabricantes possam se comunicar de forma padronizada, foram criados modelos de referência de rede. O mais conhecido é o Modelo de Referência OSI (*Open Systems Interconnection*), proposto pela ISO (*International Organization for Standardization*) (Kurose; Ross, 2014). Ele fornece uma estrutura conceitual que divide a complexidade da comunicação em rede em sete camadas abstratas, onde cada camada é responsável por uma função específica, interagindo apenas com as camadas adjacentes (Tanenbaum; Wetherall, 2011). As sete camadas são: Física, Enlace de Dados, Rede, Transporte, Sessão, Apresentação e Aplicação.

Embora o Modelo OSI seja uma referência teórica fundamental, a arquitetura mais utilizada na prática, especialmente em redes modernas e na internet, é a pilha de protocolos TCP/IP (Sethi; Sarangi, 2017). O modelo TCP/IP simplifica a estrutura em quatro camadas: Acesso à Rede (que engloba as camadas Física e de Enlace do OSI), Internet (equivalente à camada de Rede), Transporte e Aplicação (que agrupa as camadas de Sessão, Apresentação e Aplicação do OSI).

O protocolo MQTT, foco deste trabalho, opera na camada de Aplicação, utilizando o TCP (*Transmission Control Protocol*) como seu protocolo na camada de Transporte (Al-Sarawi, 2023). O TCP garante uma comunicação confiável, orientada à conexão, com controle de fluxo e retransmissão de pacotes perdidos, características essenciais para a robustez que o MQTT oferece (Park, 2023). Compreender essa arquitetura é crucial para o desenvolvimento de um protocolo do zero, pois a implementação lidará diretamente com a criação de *sockets* TCP (camada de Transporte) para enviar e receber os pacotes de dados formatados segundo a especificação MQTT (camada de Aplicação).

2.3 A PLATAFORMA SCADA TATSOFT FRAMEWORX

Os sistemas SCADA são componentes centrais na pirâmide da automação, atuando como o cérebro da operação de supervisão (Tükez; Kaya, 2022). A plataforma Tatsoft FrameworX é um exemplo de um sistema SCADA moderno, projetado com base nos princípios de flexibilidade, escalabilidade e abertura (Tatsoft, 2025). Sua arquitetura é construída integralmente sobre a tecnologia .NET da Microsoft, o que a torna um ambiente de desenvolvimento poderoso e extensível (Ekren; Sensoy; Akinci, 2025).

Essa base tecnológica é um diferencial estratégico, pois permite que engenheiros e desenvolvedores estendam as funcionalidades nativas da plataforma de diversas maneiras:

- Scripting Avançado: A plataforma inclui um editor de *scripts* integrado que suporta C# e VB.NET, permitindo a criação de lógicas customizadas, automação de tarefas e manipulação de dados em tempo real (Tatsoft, 2025).
- Integração de Bibliotecas Externas: Por ser uma aplicação .NET, o FrameworX permite a referência e o uso de bibliotecas de classes externas (arquivos .DLL). Essa capacidade é o pilar que viabiliza o presente projeto, pois a biblioteca cliente MQTT desenvolvida em C# pode ser importada e instanciada diretamente nos *scripts* do FrameworX, funcionando como um *driver* de comunicação customizado (Santos; Gamboa; Rodas, 2022; Qayum, 2024).
- Conectividade Nativa e Aberta: Além da extensibilidade, o FrameworX já oferece suporte nativo a uma vasta gama de protocolos industriais, incluindo padrões modernos como OPC UA e o próprio MQTT, demonstrando a relevância do protocolo no cenário atual (Tatsoft, 2025).

Portanto, o FrameworX não é apenas o sistema de destino, mas também um ambiente propício para o desenvolvimento e a validação de soluções de comunicação como a proposta neste trabalho.

2.4 ARQUITETURAS DE PROTOCOLOS DE COMUNICAÇÃO INDUSTRIAL

A eficiência de um sistema de automação é diretamente influenciada pela arquitetura do protocolo de comunicação. Duas arquiteturas predominam no cenário industrial:

o modelo cliente-servidor e o modelo *publish-subscribe*.

2.4.1 O Modelo Cliente-Servidor (Polling): O Caso do Modbus TCP

O modelo cliente-servidor, ou mestre-escravo, é a arquitetura mais tradicional na automação, sendo o Modbus seu exemplo mais emblemático (Le et al., 2025). A comunicação é sempre iniciada por um dispositivo mestre (o cliente), que envia uma requisição a um dispositivo escravo (o servidor) específico. Este ciclo de requisição-resposta é conhecido como *polling*.

Embora simples e determinístico em redes pequenas, o modelo de *polling* apresenta desvantagens significativas em sistemas de grande escala, característicos da IIoT (Roldán-Gómez et al., 2022):

- Uso Ineficiente da Largura de Banda: O mestre precisa interrogar continuamente cada escravo para detectar mudanças, gerando tráfego de rede constante mesmo que os dados não tenham mudado.
- Dificuldade de Escalar: O tempo de um ciclo de varredura aumenta linearmente com o número de escravos. Em redes com milhares de dispositivos, a latência para obter dados pode se tornar inaceitavelmente alta.
- Acoplamento Forte: O mestre precisa conhecer previamente o endereço de cada escravo, tornando a adição ou remoção de dispositivos uma tarefa de reconfiguração manual (Tempestini, 2025).

A ineficiência do *polling* em redes distribuídas foi um fator causal para o desenvolvimento de novas arquiteturas de comunicação, onde os dispositivos comunicam apenas quando necessário.

2.4.2 O Modelo Publish-Subscribe (Event-Driven): O Caso do MQTT

O modelo *publish-subscribe* (pub-sub) representa uma mudança de paradigma (Schlemitz; Mezhuyev, 2024). Nesta arquitetura, a comunicação é mediada por um componente central chamado *broker*. Os clientes, em vez de se comunicarem diretamente, assumem papéis de publicadores ou assinantes (Bangare; Patil, 2024).

- Um publicador envia uma mensagem para o *broker*, associando-a a um "tópico".
- Um assinante informa ao *broker* seu interesse em receber mensagens de um ou mais tópicos.
- O *broker* é responsável por receber as mensagens e encaminhá-las a todos os assinantes interessados no tópico correspondente (Al-Sarawi, 2023).

A principal característica desta arquitetura é o desacoplamento: publicadores e assinantes não precisam se conhecer. Esta abordagem orientada a eventos (*report-by-exception*) oferece vantagens significativas (Pedro, 2022):

- Eficiência de Rede: Os dispositivos só enviam mensagens quando há uma mudança de estado, minimizando o tráfego de rede.
- Escalabilidade Massiva: O *broker* pode gerenciar conexões de milhares ou milhões de clientes. A adição de um novo dispositivo não impacta o desempenho dos outros (Ibitoye, 2025).
- Flexibilidade: A arquitetura desacoplada facilita a adição, remoção ou modificação de componentes do sistema sem afetar os demais.

2.5 O PROTOCOLO MQTT

2.5.1 Histórico e Evolução

O MQTT foi inventado em 1999 por Andy Stanford-Clark (IBM) e Arlen Nipper (Arcom, hoje Cirrus Link) com o objetivo de monitorar oleodutos através de redes de comunicação via satélite, que eram caracterizadas por baixa largura de banda, alta latência e alto custo. A necessidade era de um protocolo extremamente leve, eficiente e confiável. Desde então, o protocolo evoluiu para um padrão aberto mantido pela organização OASIS (*Organization for the Advancement of Structured Information Standards*) (OASIS, 2014). As duas versões principais em uso são a v3.1.1, que se tornou um padrão ISO (ISO/IEC 20922), e a v5.0, lançada em 2019, que introduziu melhorias significativas como melhor reporte de erros, propriedades de mensagem e mecanismos para otimizar o desempenho em sistemas de larga escala (Gao, 2024).

2.5.2 Estrutura do Pacote de Controle MQTT

Toda a comunicação MQTT é realizada através da troca de Pacotes de Controle, que são as unidades básicas de dados transferidas entre cliente e *broker*. Cada pacote é binário e consiste em até três partes, seguindo uma ordem estrita (OASIS, 2014):

1. Cabeçalho Fixo (*Fixed Header*): Presente em todos os pacotes MQTT. É composto por, no mínimo, dois bytes.
    - Primeiro Byte: Contém o tipo de pacote nos 4 bits mais significativos (ex: ‘CONNECT‘, ‘PUBLISH‘, ‘SUBSCRIBE‘), permitindo 16 tipos diferentes de pacotes. Os 4 bits menos significativos são *flags* específicas para cada tipo de pacote (como DUP, QoS e RETAIN para pacotes ‘PUBLISH‘).
    - Segundo Byte (e seguintes): Contém o "Comprimento Restante"(*Remaining Length*), que indica o tamanho em bytes do restante do pacote (Cabeçalho Variável + *Payload*). Utiliza um esquema de codificação de inteiros de byte variável, podendo ocupar de 1 a 4 bytes para representar tamanhos de até 256 MB.

2. Cabeçalho Variável (*Variable Header*): Presente em muitos, mas não em todos os tipos de pacotes. Seu conteúdo depende do tipo de pacote. Por exemplo, em um pacote 'PUBLISH', ele contém o nome do tópico e o Identificador de Pacote (*Packet Identifier*) para QoS > 0. Em um pacote 'SUBSCRIBE', contém apenas o Identificador de Pacote.
3. Carga Útil (*Payload*): É a parte final do pacote e contém os dados da aplicação. Pacotes como 'PUBLISH' contêm a mensagem real a ser transmitida. Pacotes 'SUBSCRIBE' contêm a lista de filtros de tópicos que o cliente deseja assinar. O MQTT é agnóstico ao conteúdo do *payload*, tratando-o como um bloco de bytes, o que lhe confere grande flexibilidade (Hadjkouider, 2023).

2.5.3 Mecanismos Essenciais de Confiabilidade e Estado

O MQTT vai além de uma simples troca de mensagens, incorporando mecanismos que o transformam em um robusto *framework* de gerenciamento de estado, essencial para redes industriais (Boulkaibet, 2023).

2.5.3.1 Qualidade de Serviço (QoS)

O MQTT define três níveis de garantia de entrega, permitindo que o desenvolvedor equilibre confiabilidade com *overhead* de rede (Al-Sarawi, 2023):

- QoS 0 (*At most once*): A mensagem é enviada uma única vez, sem confirmação ("dispare e esqueça"). É o método mais rápido, mas sujeito à perda de mensagens em redes instáveis. Adequado para dados de telemetria não críticos.
- QoS 1 (*At least once*): Garante que a mensagem seja entregue pelo menos uma vez. O remetente armazena a mensagem e a reenvia até receber uma confirmação ('PUBACK') do receptor. Pode ocorrer a entrega de duplicatas se a confirmação for perdida.
- QoS 2 (*Exactly once*): Garante que a mensagem seja entregue exatamente uma vez. Utiliza um *handshake* de quatro vias para evitar perdas e duplicatas. É o nível mais confiável, mas com o maior *overhead*. O fluxo é o seguinte:
    1. O remetente envia a mensagem 'PUBLISH'.
    2. O receptor armazena a mensagem e responde com 'PUBREC' (Publicação Recebida).
    3. O remetente recebe o 'PUBREC', descarta a mensagem original, armazena o 'PUBREC' e responde com 'PUBREL' (Liberação de Publicação).

4. O receptor recebe o ‘PUBREL‘, descarta o estado armazenado e responde com ‘PUBCOMP‘ (Publicação Completa). A transação está finalizada.

2.5.3.2 Sessões Persistentes e Enfileiramento de Mensagens

Quando um cliente se conecta, ele pode especificar se deseja uma "sessão limpa"(*Clean Session*).

- Sessão Limpa (‘true‘): O cliente e o *broker* descartam qualquer estado de sessão anterior. Ao se desconectar, todas as informações sobre o cliente, incluindo suas subscrições, são removidas do *broker*.
- Sessão Persistente (‘false‘): O *broker* armazena o estado da sessão do cliente mesmo após sua desconexão. Esse estado inclui as subscrições do cliente e, crucialmente, enfileira todas as mensagens com QoS 1 e 2 que foram publicadas nos tópicos assinados enquanto o cliente estava offline. Ao se reconectar, o cliente recebe imediatamente todas as mensagens que estavam na fila (Al Hanif, 2023). Este mecanismo é vital para dispositivos em redes instáveis ou que se conectam intermitentemente.

2.5.4 Padronização para a Indústria: A Especificação Sparkplug B

Apesar de sua flexibilidade, o MQTT padrão não define a estrutura do *namespace* de tópicos nem o formato do *payload* (Tatsoft, 2025). Essa liberdade pode levar a problemas de interoperabilidade em ambientes com equipamentos de múltiplos fornecedores. A especificação Sparkplug B, mantida pela Eclipse Foundation, foi criada para resolver exatamente isso, definindo um padrão sobre como usar o MQTT em aplicações de IIoT e SCADA.

A solução do Sparkplug B se baseia em três pilares:

1. Namespace de Tópico Definido: Sparkplug B define uma estrutura de tópico rígida: ‘spBv1.0/<Group ID>/<Message Type>/<EON Node ID>/‘. Cada componente tem um propósito claro, e os Tipos de Mensagem (‘NBIRTH‘, ‘NDEATH‘, ‘DBIRTH‘, ‘DDATA‘, etc.) são usados para gerenciar o ciclo de vida e o estado dos dispositivos na rede.
2. Payload Padronizado: Em vez de *payloads* arbitrários, Sparkplug B utiliza Google Protocol Buffers (Protobuf), um formato de serialização binário, eficiente e fortemente tipado. O *payload* contém não apenas os valores das métricas, mas também *timestamps*, tipos de dados e metadados, garantindo que a informação seja transmitida de forma estruturada e sem ambiguidades.
3. Gerenciamento de Estado: A especificação dita o uso obrigatório de mecanismos como LWT e as mensagens ‘BIRTH‘/‘DEATH‘ para garantir que todos os

participantes da rede, especialmente o sistema SCADA, tenham uma visão consistente e em tempo real do estado (online/offline) de todos os dispositivos.

2.5.5 Segurança em Redes MQTT Industriais

A segurança é uma preocupação primordial em sistemas de controle industrial (Diaba et al., 2023). O MQTT, ao contrário de muitos protocolos legados, foi projetado com a segurança em mente, mas sua implementação segura depende da configuração correta de múltiplos mecanismos (Bangare; Patil, 2024; Gao, 2024).

- Autenticação: É o processo de verificar a identidade de um cliente. A abordagem mais básica é o uso de 'username' e 'password', enviados no pacote 'CONNECT'. Para ambientes industriais, a prática recomendada é a autenticação mútua baseada em certificados X.509, onde tanto o cliente quanto o *broker* validam os certificados um do outro (Aslan et al., 2023).
- Autorização: Uma vez que um cliente é autenticado, a autorização define o que ele tem permissão para fazer. Isso é implementado no *broker* através de Listas de Controle de Acesso (ACLs). As ACLs são regras que especificam, para cada cliente, quais tópicos eles podem publicar e/ou assinar, aplicando o princípio do menor privilégio.
- Criptografia: Para garantir a confidencialidade e a integridade dos dados em trânsito, toda a comunicação MQTT deve ser encapsulada em uma camada de criptografia, utilizando TLS (*Transport Layer Security*) (Gao, 2024). Isso impede que atores maliciosos na rede possam interceptar ou modificar as mensagens.
- Proteção de Rede: A segurança do protocolo deve ser complementada por uma arquitetura de rede segura. O Modelo Purdue para redes de controle industrial preconiza a segmentação da rede em zonas. A rede de controle (TO) deve ser separada da rede corporativa (TI) por meio de uma zona desmilitarizada (DMZ), com *firewalls* configurados para restringir rigorosamente o acesso ao *broker* MQTT.

3 METODOLOGIA E DESENVOLVIMENTO

Este capítulo descreve a abordagem metodológica e as etapas práticas adotadas para o desenvolvimento da solução proposta, detalhando o processo de construção e validação de um protocolo de comunicação MQTT. A metodologia possui um caráter essencialmente aplicado, com foco na implementação de um *broker* e de uma biblioteca cliente em linguagem C#, visando sua integração final como um *driver* de comunicação na plataforma SCADA FrameworX. A abordagem de desenvolvimento foi iterativa e incremental, garantindo a validação contínua da funcionalidade e da conformidade com a especificação do protocolo. São discutidas as estratégias de validação empregadas, incluindo a criação de uma aplicação de testes dedicada em WPF, o uso de clientes e *brokers* de terceiros para testes de interoperabilidade, e a análise de baixo nível dos pacotes de comunicação via Wireshark, uma ferramenta crucial para a depuração de protocolos de rede (Kurose; Ross, 2014).

3.1 VISÃO GERAL DO DESENVOLVIMENTO

O desenvolvimento do projeto foi estruturado em uma abordagem incremental, priorizando a modularidade e a validação em cada fase. O objetivo central foi a criação de uma biblioteca de comunicação reutilizável (compilada como uma DLL, *Dynamic-Link Library*) que encapsulasse toda a lógica do protocolo MQTT. Esta abordagem garante que o núcleo da solução seja agnóstico à aplicação final, permitindo sua utilização tanto em um ambiente de testes controlado quanto na plataforma SCADA de destino, alinhando-se às melhores práticas de arquitetura de *software* para reuso e manutenibilidade (Qayum, 2024).

O processo seguiu um fluxo de trabalho de três fases principais:

1. Desenvolvimento do Núcleo do Protocolo: Implementação simultânea de um *broker* MQTT básico e da biblioteca cliente. O desenvolvimento paralelo permitiu que cada componente servisse como uma contraparte de teste confiável para o outro, acelerando a depuração do fluxo de pacotes e da lógica de estado.
2. Validação em Ambiente Controlado: Criação de uma aplicação de testes com interface gráfica em WPF (*Windows Presentation Foundation*). Esta aplicação, referenciada como "banco de ensaios", consumiu a DLL cliente e permitiu a validação visual e funcional de cada operação do protocolo (conexão, publicação, subscrição) de forma isolada, antes de enfrentar a complexidade do ambiente SCADA.
3. Integração e Validação Final: Integração da DLL cliente ao ambiente de *scripts* da plataforma FrameworX, desenvolvendo a lógica necessária para vincular os dados do protocolo aos *tags* do sistema SCADA e validando a comunicação

ponta a ponta em um cenário de aplicação real (Silva Mognato, 2022).

O projeto foi inteiramente desenvolvido na linguagem C#, utilizando a plataforma .NET Standard 2.0. A escolha pelo .NET Standard foi estratégica, pois garante a máxima portabilidade da DLL, assegurando que ela possa ser executada tanto em ambientes .NET Framework (como a versão Windows do FrameworX) quanto em ambientes .NET 8+ (para possíveis implantações multiplataforma), garantindo a longevidade e a versatilidade do código desenvolvido.

3.2 TECNOLOGIAS E FERRAMENTAS UTILIZADAS

A seleção de tecnologias e ferramentas foi orientada pela necessidade de compatibilidade com a plataforma SCADA de destino, pela robustez para desenvolvimento de rede e pela adoção de práticas modernas de engenharia de *software*. A viabilidade de desenvolver *drivers* industriais customizados utilizando linguagens de alto nível tem sido demonstrada em trabalhos recentes (Santos; Gamboa; Rodas, 2022; Tükez; Kaya, 2022).

- Linguagem e Framework: A linguagem de programação C# foi escolhida devido à sua robustez, ao seu ecossistema maduro e ao suporte nativo a operações de rede de baixo nível, como a manipulação de *sockets* TCP através do *namespace* “System.Net.Sockets“. O uso do padrão “async/await“ foi extensivo para garantir operações de rede não bloqueantes, essenciais para a responsividade tanto do *broker* quanto do cliente. A biblioteca foi desenvolvida sobre o *framework* .NET Standard 2.0 para garantir a máxima compatibilidade com a plataforma FrameworX, que é construída sobre a tecnologia .NET (Ekren; Sensoy; Akinci, 2025).
- Ambiente de Desenvolvimento: O Microsoft Visual Studio foi utilizado como o Ambiente de Desenvolvimento Integrado (IDE), por oferecer um conjunto completo de ferramentas para codificação, depuração de processos multithread, gerenciamento de projetos e testes unitários.
- Aplicação de Teste: Uma aplicação *desktop* foi desenvolvida utilizando a tecnologia WPF. A WPF foi escolhida por permitir a criação de interfaces de usuário ricas e interativas, ideais para a validação visual e a depuração passo a passo das funcionalidades da API do cliente MQTT.
- Análise de Rede: A ferramenta Wireshark foi indispensável durante todo o processo de desenvolvimento. Ela permitiu a captura e a inspeção de baixo nível dos pacotes de dados trocados entre o cliente e o *broker*. Seu uso foi crucial para verificar a correta formatação binária dos pacotes MQTT (cabeçalhos, *payloads*, e codificação do *Remaining Length*) e para depurar a máquina de estados dos fluxos de comunicação, como o *handshake* de quatro vias do QoS 2, garantindo a conformidade com a especificação oficial do protocolo (OASIS, 2014).

- Plataforma SCADA: A plataforma de destino para a integração do *driver* foi a Tatsoft FrameworX, o sistema SCADA utilizado no ambiente de trabalho do autor (Tatsoft, 2025).
- Ferramentas de Interoperabilidade: Para garantir que a implementação estivesse em conformidade com o padrão MQTT e não apenas funcional consigo mesma, foram utilizados o *broker* de código aberto Mosquitto e o cliente de testes MQTTX. O cliente desenvolvido foi testado contra o *broker* Mosquitto, e o *broker* desenvolvido foi testado com o cliente MQTTX, validando a interoperabilidade da solução.

3.3 ARQUITETURA DA SOLUÇÃO PROPOSTA

A arquitetura do *software* foi projetada com base nos princípios de separação de responsabilidades e modularidade, resultando em um sistema de três camadas distintas, uma abordagem que promove a testabilidade e a reutilização do código (Qayum, 2024).

- Camada de Lógica de Comunicação (Core DLL): Esta é a camada central e o principal artefato do projeto. Consiste em uma biblioteca de classes .NET (compilada como uma DLL) que encapsula toda a lógica de implementação do protocolo MQTT. Esta camada contém as implementações tanto do *broker* MQTT (para testes e validação) quanto da biblioteca cliente MQTT. A biblioteca cliente expõe uma API pública e assíncrona para operações como "ConnectAsync()", "PublishAsync()", "SubscribeAsync()", etc. Esta DLL é totalmente independente de qualquer interface de usuário ou plataforma específica, sendo um componente de *backend* puro.
- Camada de Interface de Validação (Aplicação WPF): Esta camada é uma aplicação *desktop* que consome a Core DLL. Sua única finalidade é servir como um "banco de ensaios"para a API da biblioteca cliente. A interface gráfica permite que um desenvolvedor insira os parâmetros de conexão, envie mensagens, se inscreva em tópicos e visualize as respostas e os *logs* em tempo real, facilitando a depuração interativa do código da DLL.
- Camada de Integração SCADA (Scripts FrameworX): Esta é a camada de aplicação final. *Scripts* escritos em C# dentro do ambiente da plataforma FrameworX referenciam e instanciam a Core DLL. Esses *scripts* utilizam a API do cliente MQTT para estabelecer comunicação com um *broker* externo e criar a lógica de negócio que vincula os dados das mensagens MQTT aos *tags* do sistema SCADA (Tempestini, 2025).

Esta arquitetura em camadas é uma demonstração de uma metodologia de engenharia de *software* madura. A separação da lógica do protocolo em uma DLL desacoplada da sua

aplicação (seja a WPF de teste ou o FrameworX) é um princípio fundamental que garante a manutenibilidade, testabilidade e reutilização do código em diferentes contextos.

3.4 IMPLEMENTAÇÃO DO PROTOCOLO MQTT

A implementação prática do protocolo foi a fase mais complexa do projeto, exigindo uma tradução direta da especificação técnica da OASIS para código C#, com atenção especial à manipulação de dados binários, operações de *bit-shifting* e gerenciamento de estado das conexões.

3.4.1 Desenvolvimento do Broker MQTT

O desenvolvimento de um *broker* próprio, embora não fosse o objetivo final de integração, foi um passo metodológico crucial para aprofundar o entendimento do protocolo e criar um ambiente de testes autocontido. A implementação abordou as seguintes responsabilidades:

- Gerenciamento de Conexões TCP: A classe “System.Net.Sockets.TcpListener“ foi utilizada para escutar por conexões de clientes em uma porta TCP (padrão 1883). Um laço de repetição assíncrono (“while(true)“) aguardava por novas conexões com “await listener.AcceptTcpClientAsync()“. Cada “TcpClient“ aceito era então passado para uma nova tarefa (“Task.Run()“), permitindo que o *broker* gerenciasse múltiplos clientes concorrentemente sem bloquear o *listener* principal.
- Leitura e Desserialização de Pacotes: Dentro da tarefa de cada cliente, um laço de leitura contínuo lia dados do “NetworkStream“. O processo de desserialização seguia estritamente a especificação MQTT: primeiro, lia-se o primeiro *byte* do cabeçalho fixo para determinar o tipo de pacote e as *flags*. Em seguida, implementou-se o algoritmo de decodificação do "Comprimento Restante"(*Remaining Length*), que pode ocupar de 1 a 4 *bytes*, para determinar o tamanho total do pacote atual. Com o tamanho conhecido, o restante do pacote (cabeçalho variável e *payload*) era lido do *stream*. Um bloco “switch“ no tipo de pacote direcionava o *array* de *bytes* para o método de tratamento apropriado (ex: “HandleConnectPacket“, “HandlePublishPacket“).
- Gerenciamento de Estado e Sessões: Foram implementadas estruturas de dados concorrentes para gerenciar o estado global do *broker*. Um “ConcurrentDictionary<string, ClientSession>“ mapeava o “ClientID“ de cada cliente a um objeto de sessão, que continha informações como o estado da conexão, a lista de subscrições (“List<Subscription>“) e a fila de mensagens pendentes para QoS 1 e 2. Outra estrutura, como uma árvore de prefixos (*trie*), foi utilizada

para armazenar as subscrições de forma eficiente, permitindo uma busca rápida de todos os clientes assinantes de um determinado tópico.

- Lógica de Roteamento e QoS: Ao receber um pacote “PUBLISH“, o *broker* realizava a busca na estrutura de subscrições para identificar todos os clientes interessados. Para cada cliente assinante, a mensagem era enfileirada em sua sessão. A implementação dos níveis de QoS foi um ponto crítico: para QoS 1, o *broker* enviava o “PUBLISH“ e aguardava um “PUBACK“; para QoS 2, implementou-se a máquina de estados completa do *handshake* de quatro vias (“PUBLISH“ → “PUBREC“ → “PUBREL“ → “PUBCOMP“), garantindo a entrega "exatamente uma vez".

3.4.2 Desenvolvimento da Biblioteca Cliente MQTT (DLL)

A biblioteca cliente foi projetada para abstrair a complexidade da comunicação de baixo nível, oferecendo uma API de alto nível, simples e intuitiva, seguindo o modelo proposto por Santos et al. (Santos; Gamboa; Rodas, 2022) para drivers industriais.

- API Pública Assíncrona: Foram expostos métodos públicos e assíncronos como “ConnectAsync()“, “PublishAsync()“, “SubscribeAsync()“ e “DisconnectAsync()“. O uso do padrão “async/await“ e de “TaskCompletionSource“ permitiu que a aplicação consumidora pudesse aguardar a conclusão de uma operação de rede (ex: esperar pelo “CONNACK“ após enviar um “CONNECT“) sem bloquear a *thread* principal.
- Serialização de Pacotes: Esta foi a parte mais detalhada da implementação. Foram criados métodos específicos para construir o *array* de *bytes* de cada tipo de pacote. Por exemplo, um método “CreatePublishPacket(string topic, byte payload, QoS qos)“ realizava os seguintes passos:
    1. Montagem do Cabeçalho Fixo: O primeiro *byte* era construído com o tipo de pacote “PUBLISH“ (valor 3) e as *flags* de QoS, DUP e RETAIN através de operações *bitwise*.
    2. Montagem do Cabeçalho Variável: O nome do tópico era codificado em UTF-8 e precedido por 2 *bytes* de comprimento. Se QoS > 0, um Identificador de Pacote (“PacketId“) de 2 *bytes* era adicionado.
    3. Concatenação do *Payload*: Os *bytes* do *payload* eram anexados.
    4. Cálculo do Comprimento Restante: O tamanho total do cabeçalho variável mais o *payload* era calculado e codificado no formato de inteiro de *byte* variável, sendo inserido a partir do segundo *byte* do pacote final.

  Este processo foi repetido para todos os tipos de pacotes, como “CONNECT“, “SUBSCRIBE“, etc., seguindo rigorosamente a especificação.

- Máquina de Estados e Processamento em Background: Após uma conexão bem-sucedida, uma tarefa em *background* ("Task.Run()") era iniciada. Esta tarefa continha um laço de leitura contínuo que aguardava por dados no "NetworkStream", desserializava os pacotes recebidos do *broker* e os processava. Pacotes "PUBLISH" recebidos disparavam um evento público "MessageReceived". Pacotes de confirmação como "PUBACK" ou "SUBACK" completavam os "TaskCompletionSource" correspondentes, liberando a espera dos métodos assíncronos que iniciaram a operação.

3.5 INTEGRAÇÃO COM A PLATAFORMA FRAMEWORX

A etapa final consistiu em utilizar a biblioteca cliente desenvolvida dentro do ambiente SCADA, validando sua aplicação prática no contexto industrial.

- Referenciamento da DLL: O arquivo DLL compilado foi adicionado como uma referência externa no editor de *scripts* do Tatsoft FrameworX. Este processo, padrão em ambientes .NET, torna as classes e métodos públicos da biblioteca acessíveis aos *scripts* C# da plataforma (Tatsoft, 2025).
- Script de Comunicação e Ciclo de Vida: Um *script* C# foi criado dentro do FrameworX e configurado para ser executado na inicialização do servidor ("Server Startup Task"). Este *script* foi responsável por:
    1. Instanciar o objeto do cliente MQTT a partir da DLL.
    2. Configurar os parâmetros de conexão (endereço do *broker*, porta, credenciais) e as subscrições desejadas.
    3. Implementar um laço de conexão contínuo ("while(true)") com tratamento de exceções ("try-catch") para garantir que, em caso de falha de rede, o cliente tentasse se reconectar automaticamente ao *broker* após um intervalo de tempo.
    4. Assinar o evento "MessageReceived" da biblioteca cliente. O método manipulador deste evento continha a lógica para analisar o tópico da mensagem recebida e atualizar o valor do *tag* correspondente no sistema SCADA (ex: "Tag.Temperatura_Forno = double.Parse(payload);").
    5. Para a publicação de dados, foram utilizados os gatilhos de mudança de valor do FrameworX ("OnDataChange"). Um *script* associado a um *tag* de controle (ex: "Tag.SetPoint_Forno") invocava o método "client.PublishAsync()" sempre que seu valor era alterado pelo operador, enviando o novo comando para o *broker*.

Esta abordagem permitiu uma integração bidirecional e robusta, demonstrando a viabilidade de utilizar bibliotecas customizadas para estender as capacidades de comunicação da plataforma FrameworX.

4 RESULTADOS E DISCUSSÕES

Durante o desenvolvimento e a validação do protocolo de comunicação MQTT, observou-se que a implementação atende de forma satisfatória aos requisitos fundamentais da especificação, tanto em nível de transporte de dados quanto no gerenciamento de estado. A solução demonstrou ser funcional, interoperável e robusta, validando a metodologia de desenvolvimento aplicada. A adequação da implementação será demonstrada ao longo deste capítulo, detalhando os testes funcionais, a análise de pacotes e a integração final com a plataforma SCADA.

4.1 RESULTADOS DOS TESTES COM A PLATAFORMA WPF

A aplicação de testes em WPF foi o "banco de ensaios" fundamental para a validação incremental e a depuração da biblioteca cliente (DLL) e do *broker* desenvolvidos. Cada funcionalidade do protocolo foi testada de forma isolada, com os resultados sendo verificados tanto na interface gráfica quanto através da análise de pacotes com o Wireshark.

4.1.1 Arquitetura de Comunicação e Modularidade

A arquitetura de três camadas, conforme detalhado na Metodologia, foi validada com sucesso. A comunicação entre a camada de interface (WPF) e a biblioteca (DLL) foi realizada por meio de chamadas de métodos assíncronos. A interface gráfica atuou puramente como um "frontend", invocando a API pública da DLL (ex: "await cliente.ConnectAsync(...)") e se inscrevendo em eventos (ex: "cliente.MessageReceived +=...") para receber dados.

Essa modularização permitiu a depuração isolada da lógica do protocolo. O monitoramento da DLL com o Wireshark permitiu verificar o correto funcionamento da serialização e desserialização dos pacotes, sem interferência da camada de interface (Qayum, 2024). Esta separação foi a chave para garantir que a mesma DLL, uma vez validada, pudesse ser integrada ao FrameworX com alta confiança.

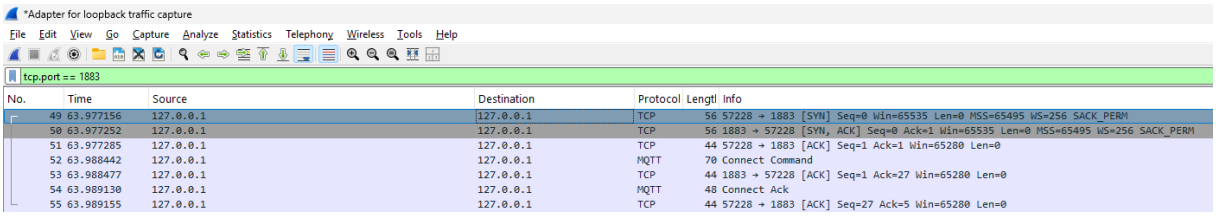
4.1.2 Conexão e Desconexão TCP/MQTT

O primeiro nível de validação consistiu em estabelecer e encerrar a comunicação entre o cliente WPF e o *broker* desenvolvido. A análise no Wireshark permitiu validar não apenas a camada de transporte (TCP), mas, mais importante, a camada de aplicação (MQTT).

Como evidenciado na Figura 1 (captura do Wireshark), a conexão foi estabelecida com sucesso. A captura demonstra o *handshake* TCP de três vias (SYN, SYN-ACK, ACK), seguido imediatamente pelo envio do pacote MQTT "CONNECT" (Tipo 1) do cliente

para o *broker*. O *broker*, após validar o pacote, respondeu com um “CONNACK“ (Tipo 2), confirmando a aceitação da conexão.

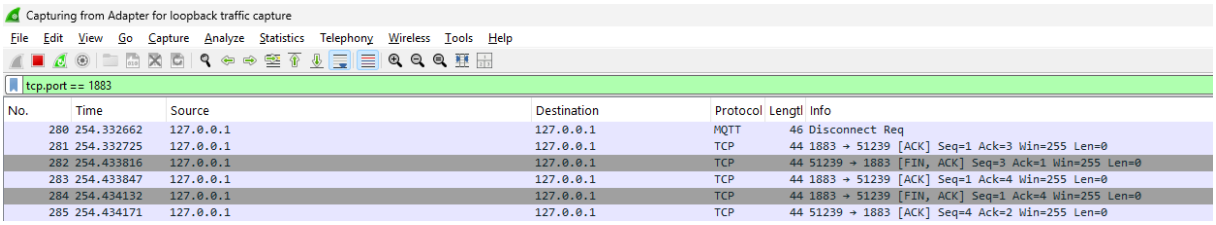
Figura 1 – Handshake TCP seguido da troca de pacotes CONNECT e CONNACK



Fonte: O autor (2025)

O processo de desconexão graciosa também foi validado. Ao acionar a desconexão na aplicação WPF, o cliente enviou o pacote “DISCONNECT“ (Tipo 14) ao *broker*. Conforme ilustrado na Figura 2, o *broker* recebeu o pacote e ambos os lados iniciaram o *handshake* de quatro vias do TCP (FIN, ACK, FIN, ACK) para encerrar a conexão de forma limpa, liberando os recursos de *socket* em ambos os sistemas (Capelli, 2006).

Figura 2 – Pacote MQTT DISCONNECT seguido do encerramento da conexão TCP



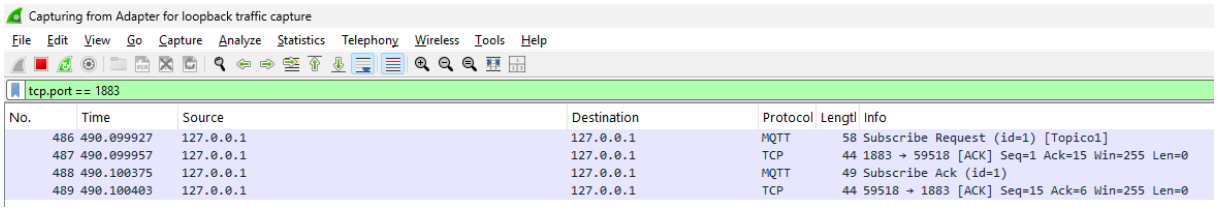
Fonte: O autor (2025)

4.1.3 Publicação, Subscrição e Níveis de QoS

A validação do fluxo de dados foi realizada testando a publicação e a subscrição entre duas instâncias da aplicação WPF (atuando como Cliente A e Cliente B), ambas conectadas ao *broker* desenvolvido.

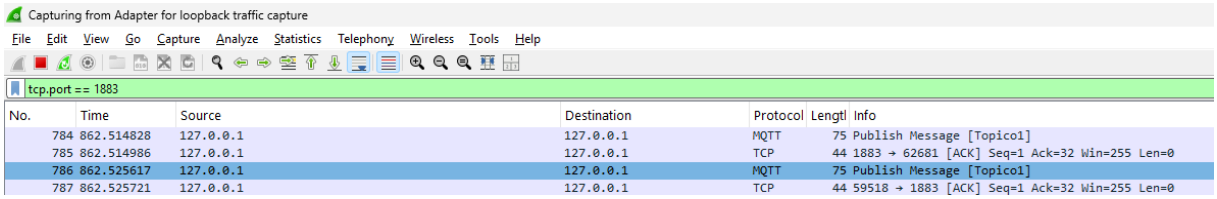
- Subscrição: O Cliente A enviou um pacote “SUBSCRIBE“ (Tipo 8) para o tópico “fabrica/setorA/temperatura“. A Figura 3 mostra o *broker* respondendo corretamente com um “SUBACK“ (Tipo 9), confirmando a inscrição.
- Publicação (QoS 0): O Cliente B publicou uma mensagem no mesmo tópico. A Figura 4 (captura do Wireshark) demonstra o fluxo: o pacote “PUBLISH“ (Tipo 3) foi enviado do Cliente B para o *broker*, que, por sua vez, roteou e encaminhou o pacote “PUBLISH“ para o Cliente A. A interface do Cliente A exibiu o valor recebido, validando a lógica de roteamento do *broker*.

Figura 3 – Troca de pacotes SUBSCRIBE e SUBACK entre cliente e broker



Fonte: O autor (2025)

Figura 4 – Fluxo de um pacote PUBLISH (QoS 0) do Cliente B para o Cliente A, mediado pelo Broker



Fonte: O autor (2025)

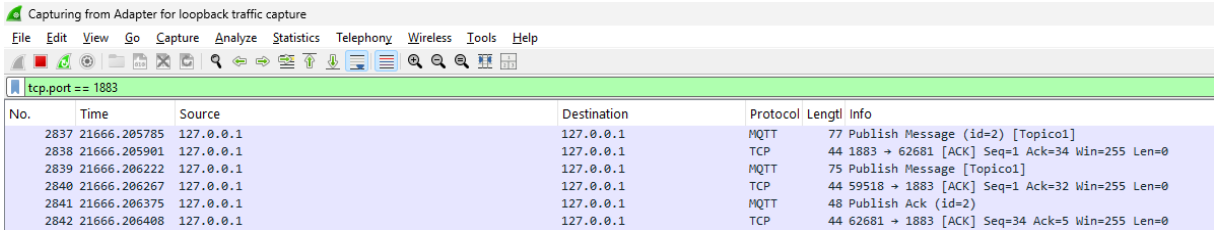
4.1.3.1 Validação Específica dos Níveis de Qualidade de Serviço (QoS)

Um dos resultados mais importantes foi a validação dos complexos fluxos de confirmação de QoS, essenciais para garantir a integridade dos dados em redes industriais (Al-Sarawi, 2023).

- QoS 1 (At Least Once): Um teste foi realizado publicando uma mensagem com QoS 1. A Figura 5 mostra a captura do Wireshark, onde o cliente envia o "PUBLISH" e o receptor (neste caso, o *broker*) responde com um "PUBACK" (Tipo 4). Isso confirma a implementação correta do *handshake* de duas vias que garante a entrega "pelo menos uma vez".
- QoS 2 (Exactly Once): A validação do QoS 2, o nível mais complexo, foi um marco do projeto. A Figura 6 ilustra o *handshake* completo de quatro vias capturado no Wireshark. O fluxo ocorreu exatamente como especificado:
    1. O publicador enviou o pacote "PUBLISH" (Tipo 3).
    2. O receptor respondeu com "PUBREC" (Tipo 5, "Publicação Recebida").
    3. O publicador, ao receber o "PUBREC", enviou o "PUBREL" (Tipo 6, "Liberação de Publicação").
    4. O receptor, ao receber o "PUBREL", finalizou a transação com "PUBCOMP" (Tipo 7, "Publicação Completa").

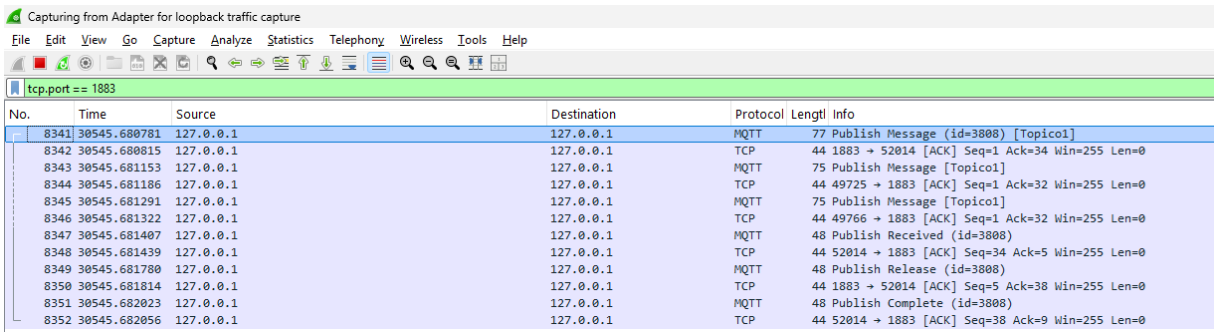
A observação deste fluxo confirmou que a máquina de estados para a garantia de entrega “exatamente uma vez“ foi implementada corretamente.

Figura 5 – Captura do handshake de duas vias do QoS 1 (PUBLISH → PUBACK)



Fonte: O autor (2025)

Figura 6 – Captura do handshake de quatro vias do QoS 2 (PUBLISH → PUBREC → PUBREL → PUBCOMP)



Fonte: O autor (2025)

4.2 RESULTADOS DA INTEGRAÇÃO COM O FRAMEWORX

Após a validação da biblioteca cliente (DLL) no ambiente controlado do WPF, a etapa seguinte foi a sua integração no ambiente SCADA FrameworX. A DLL foi importada como uma referência no editor de *scripts* C# da plataforma, conforme descrito na metodologia (Tempestini, 2025). Foram criados *scripts* para atuar como publicador e como assinante, vinculando *tags* do FrameworX a tópicos MQTT.

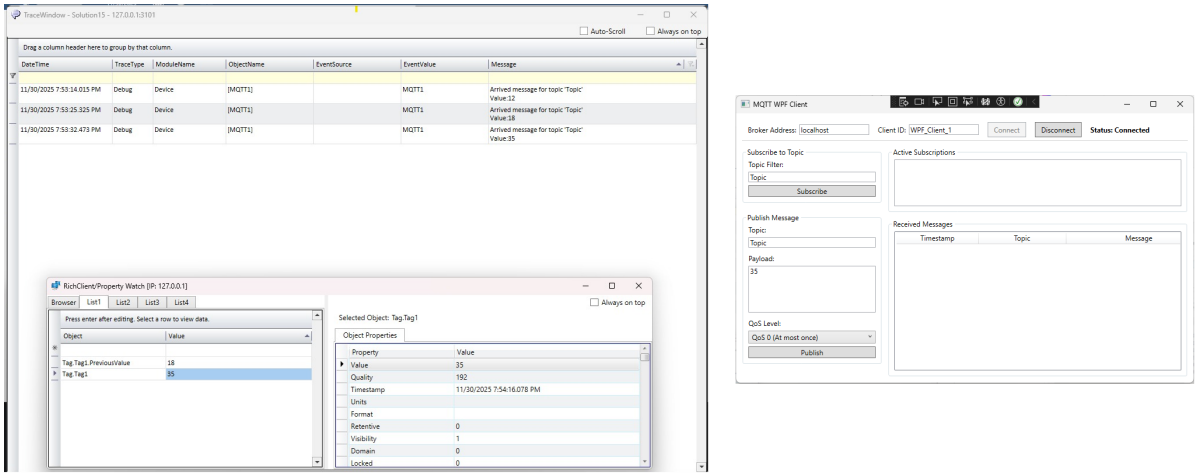
4.2.1 FrameworX como Assinante (Recebimento de Dados)

Para validar o recebimento de dados, um *script* no FrameworX instanciou o cliente MQTT da DLL e se inscreveu no tópico “scada/teste/recebimento“. Um cliente externo (MQTTX) foi utilizado para publicar valores neste tópico.

Como demonstrado na Figura 7, os valores publicados no cliente MQTTX foram recebidos pelo *script* no FrameworX e atualizados nos *tags* correspondentes, que por sua vez atualizaram os componentes visuais na tela de supervisão. O teste validou a

comunicação ponta a ponta, desde um dispositivo de campo simulado (MQTTX) até a interface do operador (FrameworX), demonstrando a separação clara entre lógica de backend e interface de usuário (Silva Mognato, 2022).

Figura 7 – Tela do FrameworX (esquerda) recebendo dados em tempo real de um cliente MQTT externo (direita)



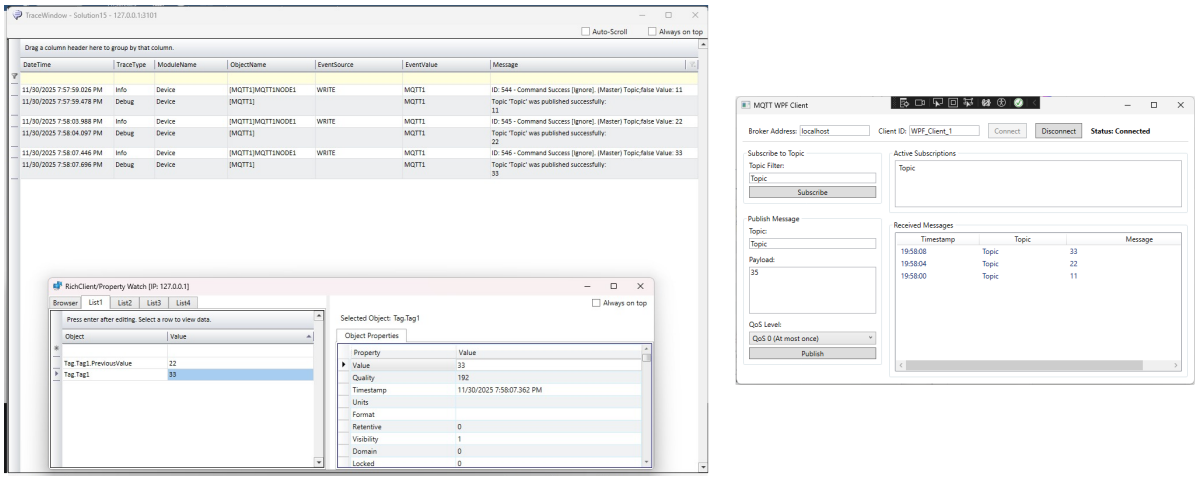
Fonte: O autor (2025)

4.2.2 FrameworX como Publicador (Envio de Dados)

Para validar o envio de dados, foi configurado um *script* no FrameworX para ser disparado pelo evento "OnDataChange" de um *tag* de *setpoint* (ex: "@Tag.Setpoint_Forno"). Quando o valor desse *tag* era alterado na tela do SCADA, o *script* invocava o método "cliente.PublishAsync()" da DLL.

Conforme ilustrado na Figura 8, a alteração do valor no *display* do FrameworX (simulando um comando do operador) resultou na publicação imediata da mensagem, que foi recebida pelo cliente MQTTX, que estava inscrito no tópico "scada/teste/envio". Este teste confirmou o sucesso da comunicação bidirecional.

Figura 8 – Cliente MQTT externo (direita) recebendo dados publicados a partir da alteração de um tag na tela do FrameworX (esquerda)



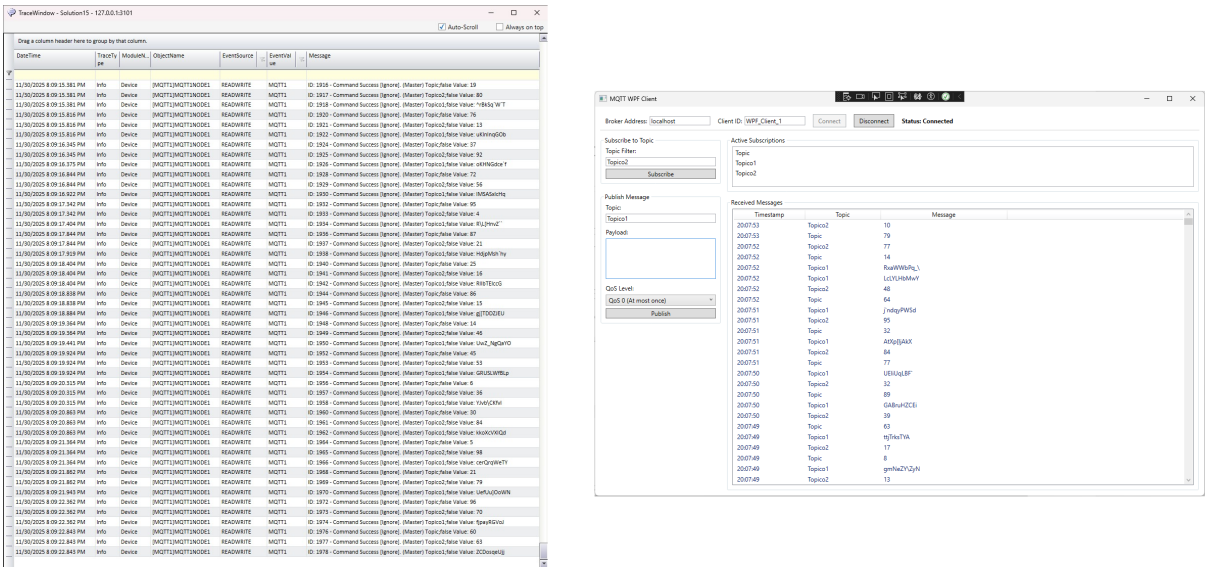
Fonte: O autor (2025)

4.3 TESTES DE ESTRESSE E CONFIABILIDADE

Para garantir que a implementação fosse robusta para um ambiente industrial, foi realizado um teste de confiabilidade com duração de quatro horas contínuas. Durante este período, múltiplos clientes, incluindo a instância do FrameworX e clientes MQTTExplorer, publicaram e receberam mensagens em alta frequência (aproximadamente 4 mensagens por segundo em diversos tópicos).

O *broker* desenvolvido e a biblioteca cliente mantiveram a comunicação estável durante todo o período, sem desconexões inesperadas, falhas ou perda de pacotes em QoS 1 e 2. O monitoramento de recursos do sistema indicou um uso de memória estável, sem evidências de *memory leaks*. A Figura 9 exibe um momento do teste, mostrando o alto volume de mensagens sendo processado pelo *broker* e recebido pelo FrameworX, confirmando a confiabilidade da solução para operação contínua, um requisito crítico em ambientes SCADA (Tükez; Kaya, 2022).

Figura 9 – Monitoramento do FrameworX e do Broker durante o teste de estresse de 4 horas



Fonte: O autor (2025)

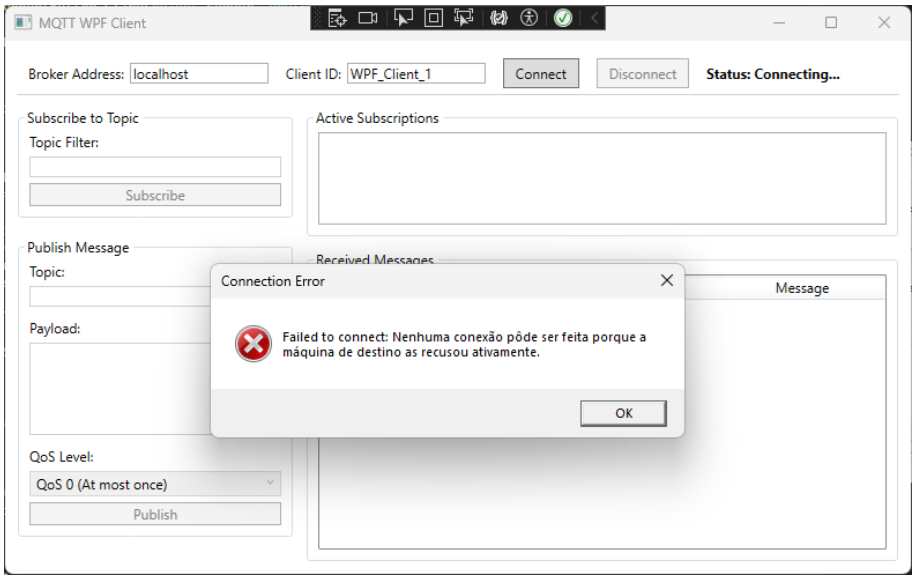
4.4 COMPORTAMENTO EM SITUAÇÕES DE FALHAS

Uma das principais vantagens do MQTT é seu conjunto de mecanismos para lidar com falhas de rede. Os testes a seguir validaram a correta implementação desses recursos críticos.

- Validação do Last Will and Testament (LWT): Um cliente WPF foi conectado ao *broker* com uma mensagem LWT configurada para o tópico “dispositivos/clienteWPF/status“ com a mensagem “OFFLINE“. Outro cliente estava inscrito neste tópico. O processo do cliente WPF foi encerrado abruptamente (simulando uma falha de energia ou de rede). Conforme esperado, o *broker* detectou a desconexão não graciosa e publicou imediatamente a mensagem “OFFLINE“ no tópico LWT, que foi recebida pelo segundo cliente, validando este mecanismo essencial de diagnóstico de falhas.
- Validação de Sessão Persistente (Persistent Session): Um cliente WPF (Cliente A) conectou-se com a *flag* “CleanSession“ definida como “false“ e se inscreveu no tópico “dados/criticos“ com QoS 1. Em seguida, o Cliente A foi desconectado. Enquanto estava offline, um segundo cliente (Cliente B) publicou 10 mensagens nesse tópico. Ao reconectar o Cliente A, ele recebeu imediatamente todas as 10 mensagens que haviam sido enfileiradas pelo *broker* durante sua ausência. Este teste validou o mecanismo de persistência de sessão, crucial para dispositivos em redes instáveis (Al Hanif, 2023).

- Tratamento de Exceções na Aplicação: A aplicação WPF foi testada em cenários de falha, como tentar conectar a um *broker* com endereço IP incorreto. A Figura 10 mostra que a aplicação capturou a exceção de *socket* e exibiu uma mensagem de erro clara para o usuário, em vez de travar, demonstrando um tratamento de erros robusto na camada de interface.

Figura 10 – Aplicação WPF exibindo uma mensagem de erro tratada após falha na tentativa de conexão



Fonte: O autor (2025)

4.5 ANÁLISE CRÍTICA E LIMITAÇÕES DA IMPLEMENTAÇÃO

Embora a implementação tenha se mostrado funcional, robusta e em conformidade com os aspectos centrais do protocolo MQTT, uma análise crítica revela limitações importantes que a diferenciam de uma solução de nível de produção.

- Padronização de Tópicos e Payloads: A maior limitação da implementação é a mesma que o protocolo MQTT base oferece: flexibilidade total. A solução não impõe uma estrutura de tópicos (namespace) nem um formato de *payload* (como JSON ou Protobuf). Isso significa que, para a interoperabilidade, todos os dispositivos na rede devem concordar previamente com uma estrutura de dados customizada. Isso impede a interoperabilidade “plug-and-play“ com dispositivos de outros fabricantes que aderem a padrões como o Sparkplug B (Capelli, 2006).
- Ausência de Segurança Robusta: A implementação focou na mecânica do protocolo sobre TCP (porta 1883). Não foi implementada a camada de criptografia de transporte (TLS/SSL, porta 8883), o que significa que todos os dados, incluindo credenciais de usuário (se usadas), trafegam em texto claro. Gao et al. (Gao, 2024) destacam que a falta de TLS expõe o ecossistema MQTT a

vulnerabilidades críticas, como ataques *Man-in-the-Middle* e interceptação de dados, tornando esta uma adição mandatória para um ambiente de produção real.

- Foco nos Recursos Essenciais (v3.1.1): A implementação concentrou-se nos recursos fundamentais do MQTT, alinhados principalmente com a especificação v3.1.1. Recursos avançados da v5.0, como Propriedades de Mensagem (para metadados), Tópicos de Resposta (para implementar padrões de requisição-resposta), e Expiração de Sessão, não foram implementados.

5 CONCLUSÃO

O presente trabalho teve como objetivo central o desenvolvimento e a implementação de um protocolo de comunicação MQTT completo, em linguagem C#, e sua subsequente integração como um *driver* de comunicação funcional na plataforma SCADA Tatsoft FrameworX. O projeto foi concebido para ir além da simples utilização de bibliotecas de terceiros, propondo a construção do protocolo a partir de seus fundamentos — um desafio que englobou a criação de um *broker* funcional e uma biblioteca cliente modular (DLL).

Com base nos objetivos específicos traçados no início deste projeto, pode-se afirmar que as metas foram integralmente cumpridas. A biblioteca C# foi implementada com sucesso, encapsulando a lógica de comunicação e aderindo à especificação do protocolo MQTT v3.1.1 (OASIS, 2014). A aplicação de testes em WPF mostrou-se uma ferramenta metodológica indispensável, permitindo a depuração e a validação de cada pacote de controle e fluxo de comunicação de forma isolada. Por fim, os testes de integração com a plataforma FrameworX, validados por ferramentas de análise de rede como o Wireshark e por clientes de terceiros, evidenciaram a conformidade, a robustez e o correto funcionamento da solução proposta.

Durante as etapas iniciais do projeto, a maior dificuldade encontrada foi a complexidade inerente à própria especificação do protocolo. Diferentemente de um protocolo de requisição-resposta como o Modbus (Capelli, 2006; Le et al., 2025), a implementação do MQTT exigiu um aprofundamento significativo em conceitos de programação de rede assíncrona, manipulação binária de dados e gerenciamento de estado. A correta implementação da serialização dos pacotes, a codificação do "Comprimento Restante"(*Remaining Length*) e, principalmente, o desenvolvimento das máquinas de estado para os fluxos de *QoS* — em particular o *handshake* de quatro vias do *QoS* 2 — representaram os desafios técnicos mais significativos. A construção de um *broker* próprio, embora complexa, foi crucial para solidificar o entendimento desses mecanismos.

Superada a fase de implementação do protocolo, o desenvolvimento avançou de forma mais fluida para a integração com o FrameworX. Conforme validado em trabalhos recentes sobre o uso de .NET em automação (Ekren; Sensoy; Akinci, 2025; Tükez; Kaya, 2022), a arquitetura aberta da plataforma facilitou a integração da DLL cliente. A solução desenvolvida mostrou-se funcional e confiável, permitindo a comunicação bidirecional de dados em tempo real entre os *tags* do sistema SCADA e o ecossistema MQTT, validando a arquitetura *publish-subscribe* como uma alternativa moderna e escalável ao *polling* tradicional (Al-Sarawi, 2023).

A principal contribuição deste trabalho reside no profundo valor acadêmico de dissecar e reconstruir um protocolo de comunicação moderno, demonstrando o domínio sobre conceitos de redes, serialização binária e programação assíncrona. O sucesso da

integração reforça a viabilidade de se desenvolver soluções de comunicação customizadas e de alto desempenho dentro do FrameworX, capacitando o engenheiro de automação a ir além das ferramentas nativas para criar soluções otimizadas e perfeitamente adaptadas às demandas da Indústria 4.0, alinhando-se às práticas de desenvolvimento de *drivers* industriais descritas na literatura (Santos; Gamboa; Rodas, 2022).

5.1 TRABALHOS FUTUROS

Embora a implementação atual seja funcional e atenda aos objetivos propostos, ela serve como uma fundação robusta sobre a qual diversas expansões podem ser construídas para elevar a solução de um protótipo funcional para um *driver* de nível industrial. As seguintes direções são sugeridas para trabalhos futuros:

- Implementação da Especificação Sparkplug B: O passo mais significativo seria evoluir a implementação para aderir totalmente à especificação Sparkplug B, um padrão industrial sobre MQTT. Isso envolveria a adoção do *namespace* de tópico padronizado, a implementação da serialização e desserialização de *payloads* utilizando Google Protocol Buffers, e a implementação da máquina de estados para o gerenciamento do ciclo de vida dos dispositivos (‘NBIRTH‘/‘NDEATH‘). Tal implementação transformaria o *driver* em uma solução verdadeiramente interoperável no ecossistema IIoT (Schlemitz; Mezhuyev, 2024).
- Adição de uma Camada de Segurança Robusta: A solução atual focou na funcionalidade do protocolo sobre TCP. Um trabalho futuro essencial seria a implementação de mecanismos de segurança em conformidade com as melhores práticas de TO (Tecnologia da Operação). Conforme alertam Gao et al. (Gao, 2024) e Roldán-Gómez et al. (Roldán-Gómez et al., 2022), a falta de criptografia expõe a rede a riscos críticos. Portanto, a próxima etapa deve incluir a integração de criptografia de transporte (TLS/SSL), o suporte à autenticação mútua baseada em certificados X.509 e o desenvolvimento de um sistema de Listas de Controle de Acesso (ACLs) no *broker* para autorização granular (Bangare; Patil, 2024).
- Suporte a Recursos Avançados do MQTT v5.0: A implementação atual baseou-se primariamente nas especificações essenciais do MQTT, alinhadas com a versão 3.1.1. Uma evolução natural seria adicionar suporte aos recursos avançados da versão 5.0, como *Response Topics* (para implementar padrões de requisição-resposta), *Session Expiry Interval* (para um controle mais refinado de sessões persistentes) e *Message Properties* (para adicionar metadados às mensagens).
- Integração Nativa via ProtocolAPI: Uma evolução de arquitetura seria migrar a solução de um *driver* baseado em *script* para um protocolo nativo da plataforma FrameworX, utilizando o "driver toolkit"ou a "ProtocolAPI"oficial da Tatsoft.

Isso permitiria que o *driver* operasse como um componente nativo, com suporte pleno aos recursos internos de *logs*, diagnóstico, configuração via interface gráfica e gerenciamento de ciclo de vida automático pelo *software* (Tatsoft, 2025). No entanto, é importante notar que a documentação para essa API pode ser escassa, o que justificou a abordagem metodológica via *scripts* neste projeto como uma alternativa viável e robusta.

REFERÊNCIAS

TEMPESTINI, Vitor Sartori. **Desenvolvimento do driver de comunicação modbus para a plataforma FrameworX**. [S.l.*:* s.n.], 2025. Trabalho de Conclusão de Curso (Graduação em Engenharia de Controle e Automação). Universidade Federal de Santa Catarina, Blumenau.

RODRIGUES, Lucas Costa Ximenes. **Desenvolvimento de sistema SCADA baseado no padrão de IHM de alta performance aplicado à subestação de geração eólica**. [S.l.*:* s.n.], 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica). Universidade Federal do Ceará, Fortaleza.

MAFRA, Rafael Benildo. **Análise e Desenvolvimento de Driver de Comunicação para Sistema SCADA**. [S.l.*:* s.n.], 2024. Trabalho de Conclusão de Curso (Graduação em Engenharia de Controle e Automação). Universidade Federal de Santa Catarina, Blumenau.

SILVA MOGNATO, Fernanda da. **Desenvolvimento de uma tela no supervisório IFIX para um sistema de combate a incêndio**. [S.l.*:* s.n.], 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia de Controle e Automação). Instituto Federal do Espirito Santo, Vitória.

PEDRO, Rui. **Self-Immunity in Industrial Cyber-Physical Systems**. 2022. Dissertação de Mestrado – Universidade do Porto.

SANTOS, Gabriel; GAMBOA, Silvana; RODAS, Ana. Development of an Industrial Communication Driver for Profinet Devices. In*:* LATEST Advances in Electrical Engineering, and Electronics. [S.l.]: Springer, 2022. p. 168–184.

SCHLEMITZ, Alexandra; MEZHUYEV, Vitaliy. Approaches for data collection and process standardization in smart manufacturing: Systematic literature review. **Journal of Industrial Information Integration**, Elsevier, v. 38, p. 100578, 2024.

BANGARE, Pallavi S.; PATIL, Kishor P. Enhancing MQTT security for internet of things: Lightweight two-way authorization and authentication with advanced security measures. **Measurement: Sensors**, Elsevier, v. 33, p. 101212, 2024.

EKREN, Nazmi; SENSOY, Mehmet; AKINCI, Tahir Cetin. Smart Buildings Using Web of Things with.NET Core: A Framework for Inter-Device Connectivity and Secure Data Transfer. **Information**, MDPI, v. 16, n. 2, p. 123, 2025.

ROLDÁN-GÓMEZ, José; CARRILLO-MONDÉJAR, Javier;
GÓMEZ, Juan Manuel Castelo; RUIZ-VILLAFRANCA, Sergio. Security Analysis of the MQTT-SN Protocol for the Internet of Things. **Applied Sciences**, MDPI, v. 12, n. 21, p. 10991, 2022.

TÜKEZ, Enes Talha; KAYA, Adnan. SCADA System for Next-Generation Smart Factory Environments. **ICONTECH International Journal of Surveys, Engineering, Technology**, v. 6, n. 1, p. 48–52, 2022.

DIABA, Sayawu Yakubu; ANAFO, Theophilus; TETTEH, Lord Anertei; OYIBO, Michael Alewo; ALOLA, Andrew Adewale; SHAFIE-KHAH, Miadreza; ELMUSRATI, Mohammed. SCADA securing system using deep learning to prevent cyber infiltration. **Neural Networks**, Elsevier, v. 165, p. 321–332, 2023.

LE, Xiang; LI, Ji; ZHAO, Yong; FAN, Zhaohong. A Security-Enhanced Scheme for ModBus TCP Protocol Based on Lightweight Cryptographic Algorithm. **Electronics**, MDPI, v. 14, n. 18, p. 3674, 2025.

ASLAN, Ömer; AKTUĞ, Semih Serkant; OZKAN-OKAY, Merve; YILMAZ, Abdullah Asim; AKIN, Erdal. A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions. **Electronics**, MDPI, v. 12, n. 6, p. 1333, 2023.

CHOUDHURY, Bikramjit; HOTA, Ashlesha; KARMAKAR, Mithun; SAHA, Sanchita; NAG, Amitava; NANDI, Sukumar. A Comprehensive Survey on Pre Versus Post Quantum Security Schemes for 5G-Enabled IoT Applications. **IEEE Access**, IEEE, v. 13, p. 1–1, 2025.

AL-SARAWI, S. et al. Towards Smart City IoT: A Survey of Protocols and Architectures. **Smart Cities**, v. 8, 2023.

SETHI, P.; SARANGI, S. R. Internet of Things: Architectures, Protocols, and Applications. **Journal of Electrical and Computer Engineering**, 2017.

BOULKAIBET, I. et al. Review of Cyber-Physical Systems Modeling With UML, SysML, and MARTE. **IEEE Access**, 2023.

QAYUM, F. et al. A Framework and Taxonomy for Characterizing the Applicability of Software Architecture. **Software: Practice and Experience**, 2024.

GAO, H. et al. **Securing MQTT Ecosystem: Exploring Vulnerabilities, Mitigations, and Future Trajectories**. [S.l.*:* s.n.], 2024.

IBITOYE, et al. Smart Waste, Smarter World: Exploring Waste Types, Trends, and Tech- Driven Valorization Through Artificial Intelligence, Internet of Things, and Blockchain. **Sustainable Development**, 2025.

PARK, K. et al. TechnologytrendsandchallengesinSDNandserviceassurancefor end-to-end networkslicing. **Elsevier**, 2023.

HADJKOUIDER, A. et al. Stackelberg Game Approach for Service Selection in UAVNetworks. **Sensors**, 2023.

AL HANIF, A. et al. Effective Feature Engineering Framework for Securing MQTT Protocol in IoT Environments. **Sensors**, 2023.

ZEYDAN, E. et al. A Marketplace Solution for Distributed Network Management... [**Journal**], 2023.

CAPELLI, Sandra. **Redes industriais: comunicação, controle e automação**. São Paulo: Érica, 2006.

BOYER, Stuart A. **SCADA: Supervisory Control and Data Acquisition**. Research Triangle Park: ISA – The Instrumentation, Systems, e Automation Society, 1993.

KUROSE, James F.; ROSS, Keith W. **Redes de computadores e a Internet: uma abordagem top-down**. 6. ed. São Paulo: Pearson, 2014.

TANENBAUM, Andrew S.; WETHERALL, David J. **Redes de Computadores**. 5. ed. São Paulo: Pearson, 2011.

OASIS. **MQTT Version 3.1.1**. [S.l.], 2014. Acesso em: 07 nov. 2025. Disponível em: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

TATSOFT. **FrameworX Documentation - Scripting and Extensibility**. 2025. Disponível em: https://docs.tatsoft.com/. Acesso em: 7 nov. 2025.