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

Rafael Benildo Mafra

Análise e Desenvolvimento de Driver de Comunicação para Sistema SCADA Utilizando o Protocolo EIP: Um Estudo Didático

Blumenau
2024

Rafael Benildo Mafra

Análise e Desenvolvimento de Driver de Comunicação para Sistema SCADA Utilizando o Protocolo EIP: Um Estudo Didático

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. Adão Boava, Dr.

Blumenau
2024

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

Mafra, Rafael Benildo
Análise e Desenvolvimento de Driver de Comunicação para Sistema SCADA Utilizando o Protocolo EIP: Um Estudo Didático / Rafael Benildo Mafra ; orientador, Adão Boava, 2024.
62 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Graduação em Engenharia de Controle e Automação, Blumenau, 2024.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Protocolo de Comunicação. 3. Sistema SCADA. 4. Indústria 4.0. I. Boava, Adão. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

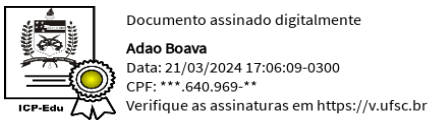
Rafael Benildo Mafra

Análise e Desenvolvimento de Driver de Comunicação para Sistema SCADA Utilizando o Protocolo EIP: Um Estudo Didático

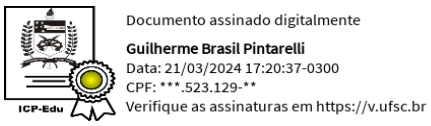
Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Engenheiro de Controle e Automação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 2 de Fevereiro de 2024.

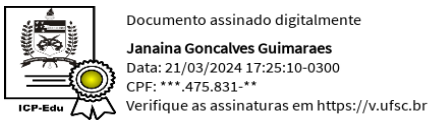
Banca Examinadora:



Prof. Adão Boava, Dr.
Universidade Federal de Santa Catarina



Prof. Guilherme Brasil Pintarelli, Dr.
Universidade Federal de Santa Catarina



Prof. Janaina Gonçalves Guimarães, Dr.
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço aos meus queridos pais, que sem o apoio incondicional nada disso seria possível, e aos meus amigos, que sem o convívio e as experiências compartilhadas eu não chegaria até aqui.

"The engineer's first problem in any design situation is to discover what the problem really is." (Autor Desconhecido)

RESUMO

Frente aos avanços tecnológicos recentes que deram forma à Indústria 4.0, os dados surgem como um recurso de valor inestimável. O paradigma industrial contemporâneo foi profundamente impactado por essas mudanças, integrando-se em um ecossistema centrado na coleta, análise e aplicação de dados. Para impulsionar organizações em direção a modelos orientados por dados na jornada de transformação digital é fundamental expandir e integrar a infraestrutura digital existente, utilizando protocolos de comunicação abertos, interoperáveis, rápidos, leves e escaláveis. A infraestrutura legada, caracterizada por um ecossistema fragmentado e carente de interoperabilidade, precisa ser superada. Este trabalho apresenta em detalhes o procedimento de análise e desenvolvimento de um driver de comunicação fictício, utilizando o protocolo EIP inventado com objetivo didático, para integração em sistema SCADA. Os sistemas SCADA modernos devem ir além e desempenhar o papel de um centro único, flexível e escalável para a coleta e distribuição de dados, integrando de maneira abrangente toda a infraestrutura digital. Proporcionando uma *single source of truth* através do *unified namespace*, permitindo o acesso fácil e confiável dos dados por outros sistemas. Essa abordagem impulsiona as organizações a um novo patamar, disrupcionando o paradigma tradicional. A integração da infraestrutura legada, tema abordado no decorrer deste trabalho, é crucial para atingir essa meta.

Palavras-chave: Indústria 4.0; Protocolo de Comunicação; SCADA

ABSTRACT

In the face of recent technological advancements that have shaped Industry 4.0, data emerges as an invaluable resource. The contemporary industrial paradigm has been profoundly impacted by these changes, integrating into an ecosystem centered around the collection, analysis, and application of data. To propel organizations towards data-driven models in the digital transformation journey, it is essential to expand and integrate existing digital infrastructure using open, interoperable, fast, lightweight, and scalable communication protocols. Legacy infrastructure, characterized by a fragmented ecosystem and lack of interoperability, needs to be overcome. This work presents in detail the procedure for analysis and development of a fictitious communication driver, using the EIP protocol invented for didactic purposes, for integration into a SCADA system. Modern SCADA systems must go beyond and play the role of a single, flexible, and scalable center for data collection and distribution, comprehensively integrating all digital infrastructure. Providing a single source of truth through the unified namespace, enabling easy and reliable data access by other systems. This approach propels organizations to a new level, disrupting the traditional paradigm. The integration of legacy infrastructure, a topic addressed throughout this work, is crucial to achieving this goal.

Keywords: Industry 4.0; Communication Protocol; SCADA.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CAD	Computer-Aided Design
CRC	Cyclic Redundancy Check
DCS	Distributed Control System
DLL	Dynamic Link Library
EIP	EtherNet Industrial Protocol
ERP	Enterprise Resource Planning
FTP	File Transfer Protocol
GE	General Electric
HMI	Human Machine Interface
HTML5	Hypertext Markup Language Version 5
IP	Internet Protocol
IPV4	Internet Protocol Version 4
ISA	International Society of Automation
ISO	International Standards Organization
LAN	Local Area Networks
MQTT	Message Queuing Telemetry Transport
OPC	Object Linking and Embedding for Process Control
OPC UA	OPC Unified Architecture
OSI	Open System Interconnection
PDU	Protocol Data Unit
PID	Proportional Integral Derivative
PLC	Programmable Logic Controllers
PROFIBUS	Process Field Bus
PTC	Parametric Technology Corporation
RTDB	Real-Time Database
SCADA	Supervisory Control and Data Acquisition
TCP	Transport Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network
WPF	Windows Presentation Foundation
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

O surgimento da microeletrônica e a subsequente difusão de diferentes equipamentos microprocessados para automação na indústria geraram a necessidade de comunicação entre eles. Assim, surgiram os protocolos, inicialmente como protocolos analógicos simples, conectando tipicamente dois equipamentos. No entanto, rapidamente evoluíram para protocolos digitais, possibilitando posteriormente a formação de redes de comunicação (AGUIRRE, 2007).

Dada a variedade de convenções, surgiram diversos protocolos digitais no mercado. Inicialmente, destacaram-se os proprietários, cujos parâmetros não eram divulgados publicamente devido a considerações de segredo estratégico, mantendo os usuários vinculados a um fornecedor específico ou a um grupo restrito de fornecedores. Posteriormente, diante das dificuldades impostas por essas restrições, os usuários buscaram a publicação das especificações dos protocolos, dando origem aos protocolos abertos. Isso permitiu o surgimento de vários fornecedores competindo com diferentes níveis de qualidade, custos e serviços. Embora ainda existam vários protocolos no mercado, os abertos estão emergindo como líderes no mercado (AGUIRRE, 2007).

A comunicação digital está agora bem estabelecida em sistemas computadorizados de controle distribuídos, tanto na fabricação discreta quanto nas indústrias de controle de processos. Sistemas de comunicação proprietários dentro de sistemas SCADA foram complementados e parcialmente substituídos por sistemas Fieldbus e barramentos de sensores. A introdução de sistemas Fieldbus tem sido associada a uma mudança de paradigma na implantação de sistemas distribuídos de automação industrial, enfatizando a autonomia dos dispositivos e a tomada de decisões descentralizada, bem como a presença laços de controle descentralizados (PETER NEUMANN, 2009).

Atualmente, os sistemas Fieldbus são padronizados e são os sistemas de comunicação mais importantes usados em instalações de controle comerciais. Ao mesmo tempo, o Ethernet consolidou sua posição como a tecnologia de comunicação mais comumente usada dentro do ambiente de escritório, resultando em preços mais baixos de componentes devido à produção em massa. Isso levou a um aumento do interesse em adaptar o Ethernet para aplicações industriais, e várias abordagens foram propostas, dessa forma, soluções baseadas em Ethernet estão se tornando cada vez mais comuns como tecnologia de integração. (PETER NEUMANN, 2009).

Contudo, para que qualquer automação ou monitoramento remoto seja efetivado, é crucial converter as informações transmitidas para os dispositivos de interface de dados de campo para um protocolo compatível com o SCADA. Dada a vasta diversidade de fabricantes, dispositivos e protocolos presentes no cenário industrial moderno, as empresas desenvolvedoras de sistemas SCADA não conseguem oferecer suporte para todos os protocolos disponíveis. Surge, portanto, a necessidade de uma arquitetura definida para

a implementação de drivers de comunicação no sistema SCADA, permitindo a interação com novos protocolos e garantindo a flexibilidade e expansibilidade do sistema.

1.1 OBJETIVOS

Nas seções abaixo estão descritos o objetivo geral e os objetivos específicos deste trabalho.

1.1.1 Objetivo Geral

O trabalho proposto tem como objetivo elucidar os conceitos de redes de automação industrial com ênfase em drivers de comunicação e sistemas SCADA. O enfoque prático do trabalho consiste em apresentar didaticamente o desenvolvimento de um driver de comunicação, utilizando o protocolo fictício EIP, especialmente direcionado para a integração com um sistema SCADA. Por fim, serão apresentados os resultados dos testes de comunicação realizados em bancada, proporcionando uma avaliação prática da eficácia e desempenho do driver de comunicação fictício implementado com objetivo didático.

1.1.2 Objetivos Específicos

- Apresentar o funcionamento do protocolo EIP, incluindo suas características, estrutura de dados e requisitos de comunicação;
- Desenvolver a versão inicial do driver de comunicação, considerando as especificações e requisitos do protocolo EIP;
- Elucidar em detalhes todo o processo de desenvolvimento e integração do driver no sistema SCADA;
- Realizar testes preliminares do driver em um ambiente de testes controlado, verificando sua capacidade de se comunicar corretamente utilizando o protocolo EIP;

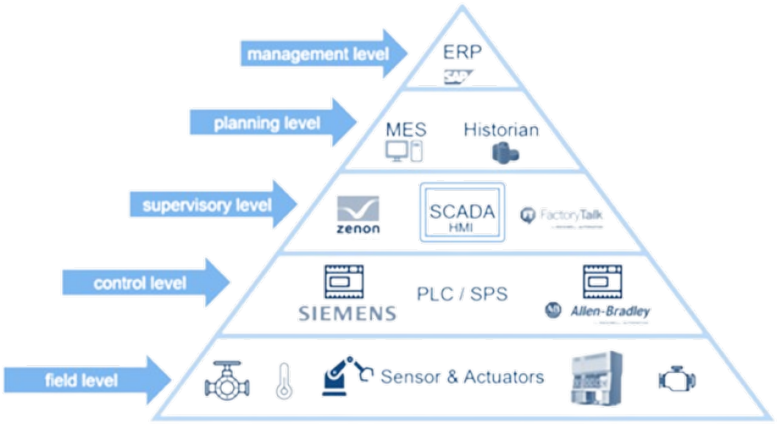
2 REVISÃO DE LITERATURA

Este capítulo apresenta uma revisão teórica sobre os principais conceitos relacionados ao desenvolvimento deste trabalho.

2.1 REDES DE COMUNICAÇÃO INDUSTRIAL

Uma Rede de Comunicação Industrial é um sistema de equipamentos interconectados que desempenha um papel crucial no monitoramento e controle de dispositivos físicos em ambientes industriais. As redes de comunicação industrial facilitam a comunicação e a troca de dados entre os diversos elementos das camadas definidas no padrão ISA-95, o Padrão Internacional para Integração de Sistemas de Automação e Gerenciamento de Produção, o padrão define formalmente as várias camadas que refletem a estrutura hierárquica destes sistemas, buscando assegurar a interoperabilidade estabelecendo uma linguagem consistente que sirva como base para a comunicação entre fornecedores e fabricantes, além de oferecer modelos de informação padronizados. A Figura 1, conhecida como a Pirâmide de Automação, é uma representação visual amplamente vinculada ao ISA-95, delineando as diversas camadas hierárquicas dos sistemas de automação industrial.

Figura 1 – Pirâmide de Automação.

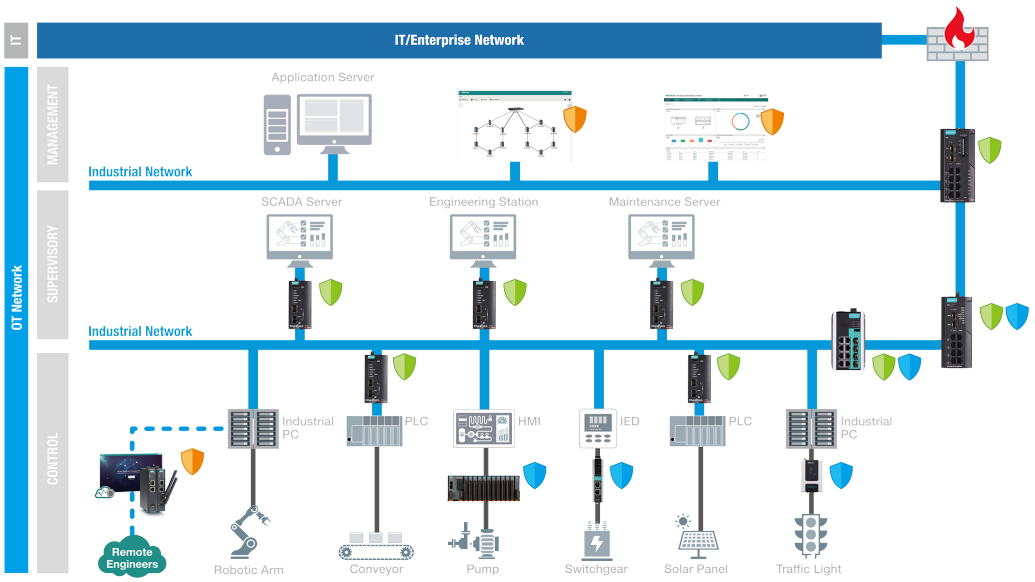


Fonte: Adaptado de (KOZIEROK, 2023).

As redes industriais desempenham um papel fundamental em diversos setores industriais, abrangendo manufatura, geração de eletricidade, processamento de alimentos, bebidas, transporte, distribuição de água e refinamento químico. Praticamente em todas as situações que exigem a supervisão e controle de maquinário, uma rede de controle industrial é instalada de alguma forma. Conforme o exemplo representado na Figura 2, geralmente possuem uma arquitetura mais complexa do que as redes comerciais. Enquanto a rede comercial de uma empresa pode consistir em LANs de filiais ou escritórios interconectadas

por uma rede principal ou WAN, até mesmo redes industriais pequenas tendem a ter uma hierarquia com três ou quatro níveis.

Figura 2 – Exemplo de Rede Industrial.



Fonte: Adaptado de (MOXA, 2024).

Os diferentes protocolos e meios físicos são frequentemente empregados em cada camada do modelo padrão ISA-95, com dispositivos utilizando uma variedade de protocolos, especialmente nas camadas iniciais, que lidam diretamente com dispositivos de campo para controle e monitoramento. Isso geralmente requer o uso de dispositivos de *gateway* para facilitar a comunicação entre as camadas. Embora melhorias em protocolos e tecnologias de redes industriais tenham simplificado as hierarquias típicas, especialmente nas camadas superiores, a arquitetura da rede muitas vezes não é simplificada totalmente para manter a correlação com a hierarquia funcional dos equipamentos controlados. (GERHARD P. HANCKE, 2012).

Como as redes de controle industrial estão conectadas a equipamentos físicos, a falha de um sistema tem impactos muito mais graves do que em sistemas comerciais. Os diversos efeitos da falha em uma rede industrial podem incluir danos aos equipamentos, perda de produção, danos ambientais, perda de reputação e até mesmo perda de vidas. Embora nem sempre causadas por falhas nos sistemas de controle, numerosos desastres industriais, como o acidente nuclear de Fukushima Daiichi em 2011, fornecem exemplos do impacto de uma falha industrial grave (GERHARD P. HANCKE, 2012).

Nessa perspectiva, na implementação de dispositivos em ambientes industriais, é imperativo satisfazer requisitos específicos, como a necessidade de comunicação em tempo real para operações eficientes, a robustez e resistência contra condições adversas, como poeira, umidade e variações de temperatura. A implementação de redundância é funda-

mental para assegurar a continuidade operacional em situações de falha, desempenhando um papel crucial na confiabilidade dos sistemas industriais. Garantir a conformidade dos dispositivos e da infraestrutura de rede com padrões de comunicação em tempo real, aliada ao uso de certificações de proteção, constitui práticas essenciais para otimizar tanto o desempenho quanto a estabilidade desses ambientes.

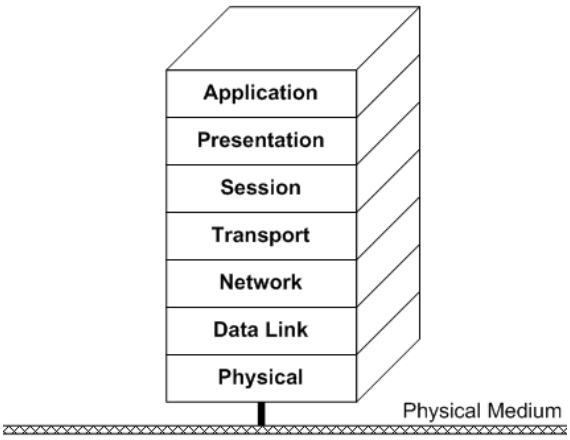
2.1.1 Protocolo de Comunicação

A comunicação bem-sucedida entre duas entidades requer que ambas "falem a mesma linguagem". O conteúdo, a forma e o *timing* da comunicação devem obedecer às convenções mutuamente acordadas entre as partes envolvidas. Essas convenções são conhecidas como protocolo, que pode ser definido como um conjunto de regras que controlam a troca de dados entre duas instalações. Em outras palavras, o protocolo estabelece as diretrizes essenciais para garantir uma comunicação eficaz e padronizada entre as entidades comunicantes (STALLINGS, 2005).

Os protocolos de rede dividem processos mais amplos em funções e tarefas discretas e estritamente definidas em cada nível da rede. No modelo padrão, amplamente conhecido como o modelo OSI, um ou mais protocolos de rede regulam atividades em cada camada na troca de telecomunicações. As camadas mais baixas lidam com o transporte de dados, enquanto as camadas superiores no modelo OSI lidam com software e aplicações.

O Modelo OSI, ilustrado na Figura 3, é fundamentado em uma proposta desenvolvida pela ISO como um marco inicial para a padronização internacional de protocolos em várias camadas. Sua revisão ocorreu em 1995. Denominado Modelo de Referência ISO OSI (TANENBAUM, 2002).

Figura 3 – Modelo de Referência OSI.



Fonte: (VIVIANO, 2023b).

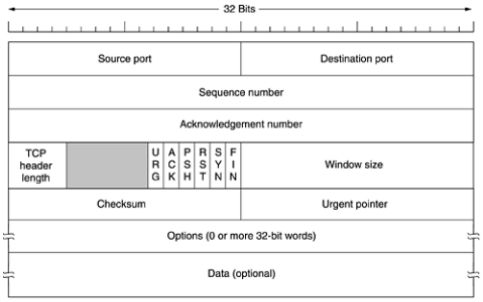
- Camada Física (*Physical*), ocorre a transmissão de sinais físicos entre dispositivos, determinando como os bits são representados e transferidos.
- Camada de Enlace de Dados (*Data Link*) viabiliza a conectividade direta, estabelecendo, gerênciando conexões e corrigindo erros de transmissão;
- Camada de Rede (*Network*) direciona pacotes entre diferentes redes, gerenciando roteamento;
- Camada de Transporte (*Transport*), a transferência de dados é controlada, garantindo confiabilidade com TCP ou simplicidade com UDP;
- Camada de Sessão (*Session*) gerencia o estabelecimento e encerramento de sessões, proporcionando autenticação;
- Camada de Apresentação (*Presentation*) formata dados para serem compreendidos, incluindo criptografia e compressão;
- Camada de Aplicação (*Application*) interage diretamente com o usuário, fornecendo serviços como HTTP e FTP para compartilhamento de arquivos e acesso a aplicativos, completando a estrutura essencial para a comunicação eficiente em redes;

Apesar de ser amplamente reconhecido e servir como referência no estudo e aplicação de protocolos de comunicação, o modelo OSI muitas vezes é utilizado apenas como um modelo teórico distante da implementação real de protocolos de comunicação, e enfrenta críticas consideráveis.

Hoje, compreende-se que o lançamento dos protocolos do padrão OSI foi precipitado. Na época em que esses protocolos foram introduzidos, os concorrentes TCP/IP já estavam sendo amplamente adotados por pesquisadores em universidades. Antes mesmo do início de investimentos bilionários, o mercado acadêmico já era robusto, levando muitos fabricantes a oferecerem produtos baseados em TCP/IP, embora com certa cautela. Quando o modelo foi introduzido, as empresas relutaram em investir em uma segunda pilha de protocolos, a menos que se tornasse uma necessidade imposta pelo mercado. Com todas as empresas aguardando que alguém desse o primeiro passo, o modelo OSI permaneceu, em grande parte, no plano teórico, sem ganhar implementação generalizada (TANENBAUM, 2002).

As mensagens transmitidas entre as entidades comunicantes são denominadas pacotes, um pacote é composto por informações de controle e carga útil, as informações de controle fornecem dados para a entrega e leitura da carga útil, como endereços de rede de origem e destino, tamanho das informações contidas, códigos de detecção de erros ou informações de sequenciamento como *checksum*, CRC e paridade. Normalmente, as informações de controle são encontradas nos cabeçalhos e no final dos pacotes após a carga útil. Tomando como exemplo o pacote do protocolo TCP na Figura 4, os campos do cabeçalho e a carga útil seguem a sequência padrão estabelecida.

Figura 4 – Estrutura do Pacote de Dados TCP.

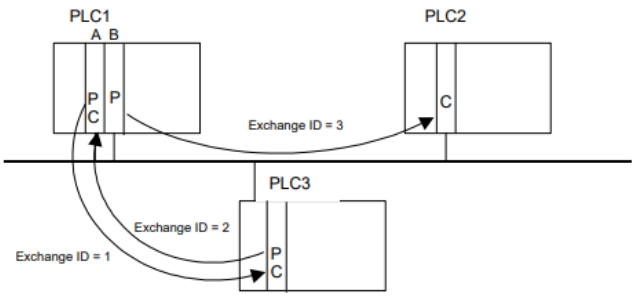


Fonte: (TANENBAUM, 2002).

2.1.1.1 Ethernet Indsutrial Protocol

Para fim didático foi definido que protocolo deveria se basear em protocolos padrão da internet para compartilhar informações entre controladores, trocando amostras de dados, conforme ilustrado na Figura 5. Cada pacote contém uma amostra de dados ou *snap-shot* da memória de um controlador, assim como é comum em protocolos industriais. Essas amostras são enviadas periodicamente para um ou mais controladores pares que armazenam os dados para uso em tarefas de aplicação, os controladores podem tanto ler como escrever na rede. Cada amostra de dados é identificada de forma única para relacioná-la a uma definição que descreve os dados que ela contém, chamada de Exchange. Para este protocolo fictício o controlador que gera a amostra é denominado o Producer, e o controlador que a recebe é denominado Consumer. Cada controlador enviará ou receberá amostras de dados apenas para os Exchanges para os quais foi configurado. Dessa forma, uma rede pode ser configurada para que vários controladores compartilhem informações para realizar funções de controle ou monitoramento.

Figura 5 – Exemplo de Compartilhamento de Memória entre PLCs.



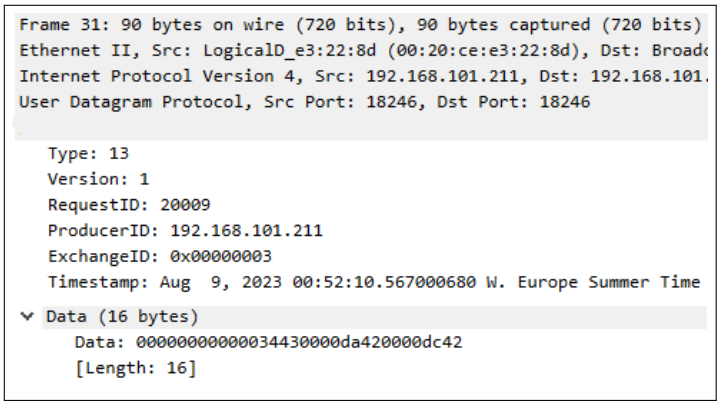
Fonte: (FANUC, 2002).

Para assegurar a confiabilidade da informação transmitida a nível da camada de aplicação, foi definido que o EIP organiza os dados a serem transmitidos em Exchanges, e

várias Exchanges são combinadas para formar *Pages*. Cada *Page* possui um identificador único, que é uma combinação do ID do Produtor (*Producer ID*) e do ID da Exchange (*Exchange ID*). Esse identificador permite que o receptor reconheça os dados e saiba onde armazená-los. Com o EIP, um produtor pode enviar informações para um número ilimitado de consumidores simultaneamente, em uma taxa periódica fixa.

Sua estrutura organizada em Producer ID e Exchange ID, garante a integridade dos dados transmitidos e a sincronização entre produtores e consumidores, a Figura 6 exibe todos os campos do protocolo EIP apresentados neste capítulo.

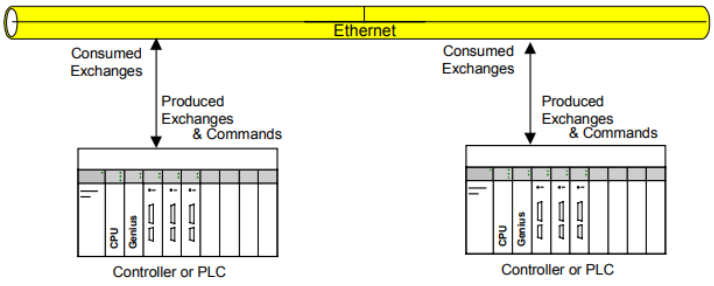
Figura 6 – Exemplo de Protocolo no Wireshark.



Fonte: O Autor.

Na arquitetura de comunicação apresentada na Figura 7, extraída do manual do sistema Cimplicity e adaptado para exibir o protocolo fictício EIP, o sistema Cimplicity é software SCADA e HMI da GE. Na Figura 7 destaca-se a interação entre os controladores assim como o funcionamento definido para o EIP, trazendo paralelos com o funcionamento de um protocolo real. Essa arquitetura utiliza a rede Ethernet como meio de comunicação, assim como o EIP para viabilizar a troca eficiente de informações entre os dispositivos de controle.

Figura 7 – Exemplo de Comunicação Utilizando o meio Ethernet.

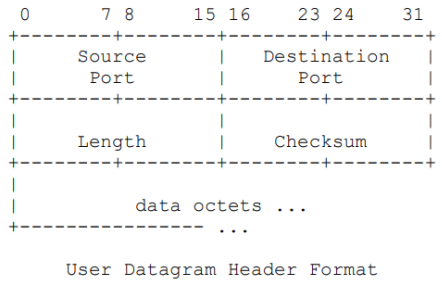


Fonte: Adaptado de (GE, I. S., 2008).

2.1.1.2 User Datagram Protocol

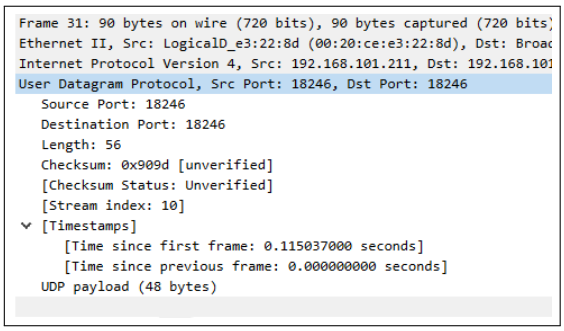
O protocolo UDP é definido para disponibilizar um modo de datagrama na comunicação de computadores em uma rede de computadores interconectada. Esse protocolo pressupõe que o protocolo da internet IP seja utilizado como o protocolo subjacente. Esse protocolo oferece um procedimento para programas de aplicação enviarem mensagens a outros programas com um mínimo de mecanismos de protocolo. O protocolo é orientado a transações, e não são garantidas a entrega nem a proteção contra duplicatas. Aplicações que necessitam de entrega confiável e ordenada de fluxos de dados devem utilizar o TCP (POSTEL, 1980).

Figura 8 – Cabeçalho UDP na Documentação original.



Fonte: (POSTEL, 1980).

Figura 9 – Protocolo UDP no Wireshark.



Fonte: O Autor.

Uma das características mais importantes do UDP é sua baixa sobrecarga em comparação com o TCP, conforme ilustrado nas Figuras 8 e 9, o protocolo UDP possui apenas quatro campos enquanto o pacote TCP conforme a Figura 4 possui dez campos. Ao contrário do TCP, que realiza controle de fluxo, controle de congestionamento e confirmação de recebimento, o UDP não possui esses recursos. Isso torna o UDP mais eficiente em termos de latência e uso de largura de banda, sendo ideal para aplicações em que a velocidade é prioritária e a perda ocasional de dados não é crítica.

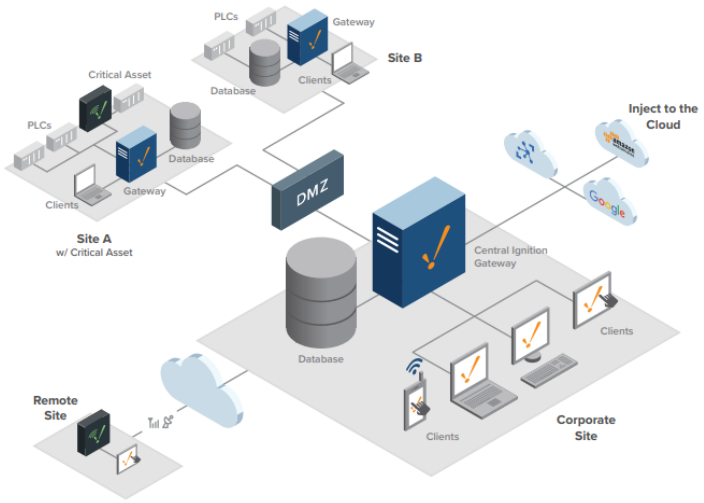
O protocolo EIP emprega o UDP na camada de transporte, conforme apresentado no capítulo anterior. Esse protocolo opera sob a premissa de compartilhar periodicamente a memória dos controladores, geralmente em intervalos inferiores a alguns segundos. A escolha do UDP se justifica pela frequência constante de envio de pacotes na rede, dispensando a necessidade de confirmação de recebimento e garantindo velocidade suficiente no processo.

2.2 SISTEMA SCADA

SCADA é a sigla para Supervisão, Controle e Aquisição de Dados. Como o nome sugere, não se trata de um sistema de controle completo, mas sim concentra-se no nível de supervisão. Sendo assim, é uma camada de software posicionado sobre hardware ao qual é interfaceado, geralmente por meio de PLCs ou outros módulos de hardware comerciais (W. SALTER, 1999).

Um sistema SCADA, conforme o exemplo na Figura 10, coleta informações detalhadas, como a detecção de vazamentos em um gasoduto, transfere esses dados para um local central, emite alertas sobre o vazamento à estação base, conduz análises e controles necessários, como a determinação da criticidade do vazamento, e apresenta as informações de maneira lógica e organizada.

Figura 10 – Exemplo de Arquitetura Externa de sistema SCADA.



Fonte: (IGNITION, 2024).

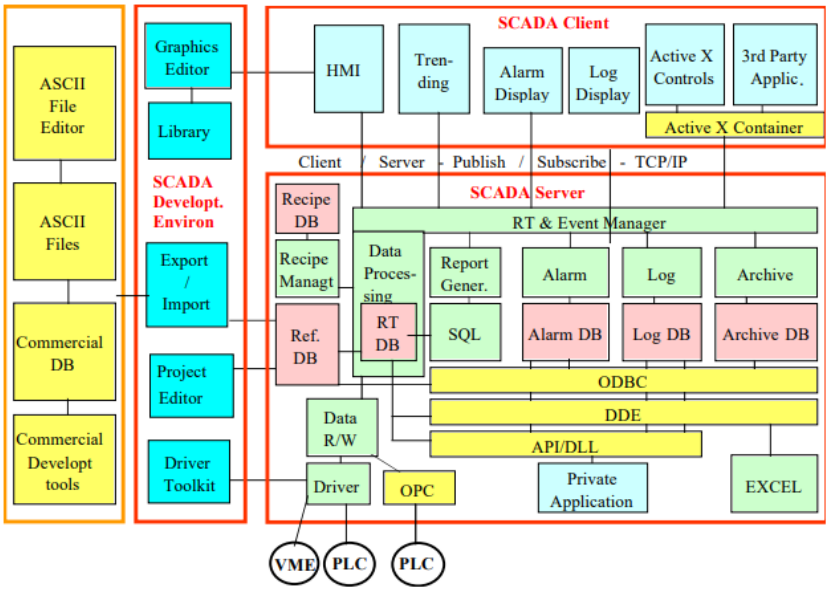
O conceito de SCADA foi desenvolvido para ser um meio universal de acesso remoto a uma variedade de módulos de controle locais, os quais podem ser de diferentes fabricantes e permitem o acesso por meio de protocolos de automação padrão. Na prática, os sistemas

SCADA de grande porte evoluíram para se tornarem muito semelhantes aos sistemas de controle distribuído em função, utilizando múltiplos meios de interface com a planta.

O foco de um SCADA é a aquisição de dados e a apresentação de uma HMI centralizada, embora também permita o envio de comandos de alto nível para controlar hardware, como instruções para iniciar um motor ou alterar um ponto de ajuste. Os sistemas SCADA são projetados para monitorar hardware de controle geograficamente disperso, sendo especialmente adequados para indústrias como distribuição de utilidades, onde as áreas da planta podem estar localizadas em milhares de quilômetros quadrados.

A localização remota dos dispositivos impõe restrições ao sistema e é um aspecto central da forma como os sistemas SCADA são projetados. A comunicação de dados sobre longas distâncias muitas vezes envolve o uso de mídias de terceiros, como linhas telefônicas ou telefonia celular. Essas mídias frequentemente são pouco confiáveis ou têm limitações de largura de banda. Portanto, os sistemas SCADA tendem a ser orientados por eventos, em vez de orientados por processos, com foco apenas em relatar mudanças no estado do sistema monitorado, em vez de enviar continuamente um fluxo constante de variáveis de processo. Isso permite uma redução no número de comunicações enviadas e reduz os requisitos de largura de banda. O software SCADA também precisa levar em consideração as mídias de comunicação pouco confiáveis e ser capaz de implementar recursos, como registrar o último valor conhecido de todas as variáveis no sistema e determinar a qualidade dos dados (GERHARD P. HANCKE, 2012).

Figura 11 – Exemplo de Arquitetura Interna de sistema SCADA.



Fonte: (W. SALTER, 1999).

Conforme detalhado na Figura 11, os sistemas SCADA são multitarefas e fundamentam-

se em um RTDB localizado em um ou mais servidores. Esses servidores são encarregados da aquisição e manipulação de dados, como a consulta de controladores, verificação de alarmes, cálculos, registro e arquivamento, em um conjunto de parâmetros, geralmente aqueles aos quais estão conectados.

A comunicação entre servidor-cliente e entre servidores, em geral, é baseada em um modelo de *publish* e *subscribe* e orientada a eventos, utilizando o protocolo TCP/IP. Em outras palavras, uma aplicação cliente se inscreve em um parâmetro que é gerenciado por uma aplicação de servidor específica, e apenas as alterações nesse parâmetro são então comunicadas à aplicação cliente.

No acesso aos dispositivos, os servidores de dados realizam *polling*, nos controladores em uma *polling rate* definida pelo usuário. Essa taxa pode variar para diferentes parâmetros. Os controladores enviam os parâmetros solicitados para os servidores de dados. O *timestamp* dos parâmetros do processo geralmente é realizado nos controladores, e esse *timestamp* é transferido para o servidor de dados. Se o controlador e o protocolo de comunicação utilizados suportam transferência de dados não solicitados, o SCADA também oferece suporte a esse recurso.

Também oferecem suporte a uma variedade de protocolos de comunicação, para garantir uma integração eficiente com os PLCs e barramentos de campo amplamente adotados, como o Modbus, PROFIBUS, EtherNet/IP, DeviceNet e mais recentemente MQTT e OPC UA, dependendo das necessidades específicas da aplicação e dos dispositivos utilizados, geralmente também fornecem um *Toolkit* disponibilizado para criar drivers destinados a hardware não suportado pelo SCADA.

A maioria permite que ações sejam acionadas automaticamente por eventos. Uma linguagem de script fornecida pelos produtos SCADA possibilita a definição dessas ações. Em termos gerais, é possível carregar uma tela específica, enviar um e-mail, executar uma aplicação ou script definido pelo usuário e escrever no RTDB.

Atualmente, as empresas estão inseridas em um cenário onde os dados desempenham um papel crucial. Permitir o controle e supervisão dos dados provenientes da ampla gama de dispositivos industriais eleva a indústria a um patamar superior. No entanto, devido à complexidade dos sistemas SCADA, eles têm um preço elevado, o que torna difícil para muitas empresas arcar com os custos. Como alternativa, frequentemente são oferecidas opções com funcionalidades limitadas para atender às necessidades de empresas menores.

2.3 SISTEMA DE CONTROLE DISTRIBUIDO

De acordo com a nomenclatura amplamente reconhecida, DCS, ou Sistema de Controle Distribuído, consiste em um arranjo de equipamentos de controle distribuídos no ambiente industrial. O sistema, geralmente equipado com processadores personalizados como controladores, emprega redes redundantes, que podem ser proprietárias ou seguir

protocolos padronizados. Os controladores recebem dados dos módulos de entrada e os transmitem aos módulos de saída. Os módulos de entrada captam informações dos instrumentos de processo no campo, enquanto os módulos de saída se comunicam com os atuadores do campo (GARCIA, 2019).

Barramentos elétricos e protocolos digitais são utilizados para conectar os controladores distribuídos a um centro de controle, que por sua vez é vinculado a uma HMI, permitindo a visualização dos dados e informações do processo em tempo real. Estações de operação locais, distribuídas pela planta, são comuns para assumir as funções da central de controle em caso de falha, garantindo um nível adicional de redundância e confiabilidade. Os componentes do DCS podem se conectar diretamente a equipamentos físicos, como interruptores, bombas, motores e válvulas, ou operar por meio de um sistema intermediário, como o SCADA (GARCIA, 2019).

O DCS surgiu primeiramente em indústrias de grande porte, com processos críticos e de alto valor, sendo atraentes porque os fabricantes forneceriam tanto o nível de controle local quanto o equipamento e a supervisão central como um pacote integrado, reduzindo assim o risco de integração de projeto, como no caso da turbina e do MarkVIe, representado na Figura 12, fornecidos pela GE. Atualmente, as funcionalidades SCADA e do DCS são muito similares, porém o DCS tende a ser utilizado em grandes plantas de processos contínuos, onde alta confiabilidade e segurança são importantes, e onde o centro de controle não está geograficamente remoto (ELORANTA *et al.*, 2014).

DCS são mais adequados para processos contínuos que envolvem múltiplos sinais analógicos e laços de controle PID complexos, comumente encontrados em usinas de energia e refinarias. O diferencial chave de um DCS é sua confiabilidade devido à distribuição do processamento de controle em nós no sistema. Isso atenua uma falha de processador única. Se um processador falhar, afetará apenas uma seção do processo da planta, ao contrário de uma falha de um computador central, que afetaria todo o processo. Essa distribuição do poder de computação local para os racks de conexão de I/O do campo também garante tempos rápidos de processamento do controlador, removendo possíveis atrasos de rede e de processamento central.

Muitas vezes o DCS possue apenas HMIs integradas e as soluções prontas de sistemas SCADA em DCS não são tão avançadas, resultando na necessidade de integração de outros sistemas. Em muitos casos o sistema de supervisão do DCS não substitui um SCADA, os sistemas SCADA são especializados para oferecer supervisão e monitoramento centralizados. Conforme apresentado no capítulo 2.2 eles fornecem ferramentas especializadas para visualização de processos, análise de tendências e histórico de dados.

Figura 12 – DCS MarkVIe.

Fonte: Adaptado de (GE, 2021).

A integração de um SCADA em um DCS proporciona uma sinergia entre esses sistemas. Ao unificar o SCADA, responsável pelo monitoramento e supervisão, com o DCS, focado no controle detalhado dos processos, obtêm-se uma visão abrangente e detalhada do processo industrial. Isso é possível devido à capacidade de compartilhamento de dados entre os sistemas, permitindo uma análise mais profunda e em tempo real do status operacional da planta. A integração resulta em melhorias significativas na eficiência operacional, com uma tomada de decisões mais embasada e rápida, baseada em informações detalhadas do processo. Além disso, a flexibilidade e a escalabilidade são aprimoradas, possibilitando a incorporação fluida de novos dispositivos e sistemas à medida que as necessidades da planta evoluem.

2.4 DRIVER DE COMUNICAÇÃO

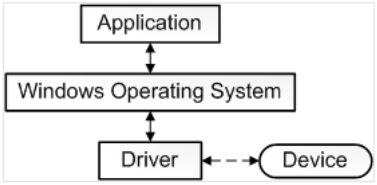
Um driver, de forma geral, pode ser definido como uma interface de software que permite a comunicação entre entidades de hardware distintas. Em um contexto de sistema operacional, conforme a Figura 13, um driver de dispositivo é um componente de software que atua como um tradutor entre o sistema operacional e o dispositivo de hardware específico. O driver de dispositivo compreende os comandos e protocolos específicos do dispositivo, convertendo as solicitações de alto nível do sistema operacional em sinais de baixo nível que o dispositivo pode compreender e executar (VIVIANO, 2023a).

Os driver de dispositivos operam dentro da camada do kernel do sistema operacional, funcionando em um ambiente altamente privilegiado devido à necessidade de acesso de baixo nível às operações de hardware. Sua função principal é facilitar a comunicação entre o SO do computador e o hardware específico para o qual foram desenvolvidos. Essa comunicação ocorre por meio de um barramento de computador que estabelece uma

conexão entre o dispositivo e o computador.

Para acessar e executar as instruções do dispositivo, os drivers dependem das instruções fornecidas pelo sistema operacional. Após a conclusão de suas tarefas, eles transmitem a saída ou mensagens do dispositivo de hardware de volta ao sistema operacional. Dispositivos como modems, roteadores, alto-falantes, teclados e impressoras dependem dos drivers de dispositivo para funcionar corretamente. Esses drivers desempenham um papel crucial ao possibilitar uma interação fluida entre o sistema operacional e o hardware (MALLICK, 2022).

Figura 13 – Exemplo de Driver no Windows.



Fonte: (VIVIANO, 2023a).

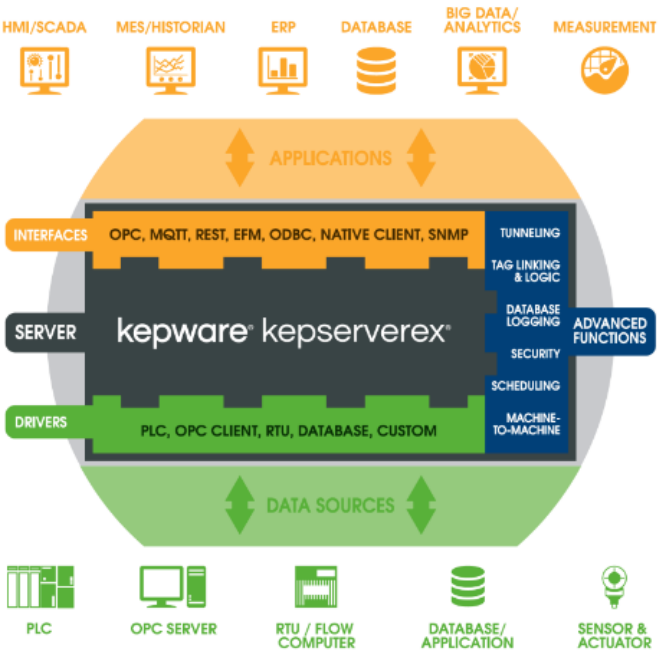
Driver de comunicação são diferentes dos driver de dispositivos, os drivers de comunicação são projetados para a transmissão de dados dos dispositivos de hardware para o software em execução e para a transmissão de comandos do software em execução para os dispositivos, conforme o protocolo de comunicação do dispositivo.

Os drivers de comunicação são frequentemente usados no contexto de automação industrial, sistemas SCADA ou em outros cenários nos quais dispositivos distintos precisam se comunicar, no entanto, os drivers de dispositivo são essenciais para o funcionamento adequado de componentes de hardware, como impressoras, placas gráficas e adaptadores de rede, traduzindo comandos de alto nível do sistema operacional em instruções de baixo nível que o hardware pode executar.

Utilizando como exemplo o software KEPserverEX da empresa Kepware, subsidiaria da americana PTC, pioneira em CAD em 1988. A Kepware concentra-se no desenvolvimento de drivers de comunicação para controladores de automação e dispositivos de campo (KEPWARE, 2023).

KEPServerEX é a principal plataforma de conectividade da indústria, com mais de 150 protocolos de comunicação disponíveis fornecendo uma única fonte de dados de automação industrial para todas as suas aplicações. Os usuários podem conectar, gerenciar, monitorar e controlar diversos dispositivos de automação e aplicativos de software através de uma interface de usuário intuitiva (KEPWARE, 2023).

Figura 14 – Kepware KEPServerEX.



Fonte: (KEPWARE, 2023).

Conforme a Figura 14, a aplicação mais comum do KEPServerEX é para coletar os dados dos mais diversos dispositivos fazendo uso do vaso leque de drivers disponíveis, então enviar os os dados obtidos nos protocolos abertos mais utilizados, como MQTT e OPC, dessa forma, centralizando a coleta de informações e padronizando o envio desses dados, fornecendo um protcolo acessível e permitindo a comunicação com outros sistemas como SCADA, HMI, ERP e banco de dados.

No processo de desenvolvimento e testes de driver de comunicação o KEPserverEX é frequentemente usado, como no caso deste trabalho. Contendo uma enorme variedade de drivers e todos certificados para operaração em aplicações onde é necessário elevado grau de segurança, o software fornece flexibilidade e credibilidade na sua utilização.

3 DESENVOLVIMENTO

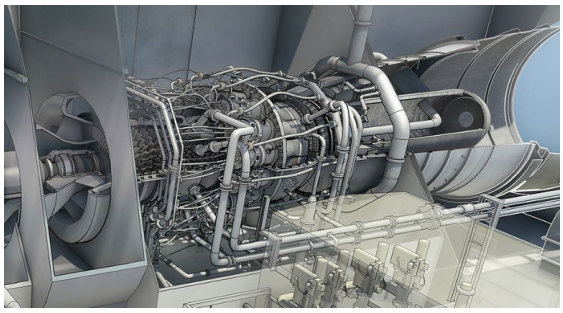
Neste capítulo, é apresentada a série de etapas que culminou no desenvolvimento operacional do driver.

3.1 CONTEXTO

Assumindo um caso em que o suporte ao protocolo EIP no sistema SCADA não existe e foi encomendado o desenvolvimento de um driver personalizado, o processo envolve a utilização de uma biblioteca de desenvolvimento de driver, denominada *Toolkit*, alinhado ao padrão arquitetural estabelecido para a integração com o sistema SCADA.

Assumindo que a iniciativa surgiu da necessidade de estabelecer a comunicação entre o SCADA e uma turbina geradora de energia. A turbina, ilustrada na Figura 15, é controlada por um DCS.

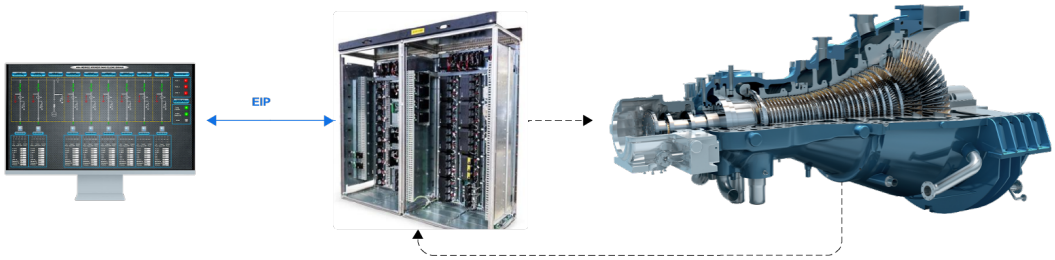
Figura 15 – Representação Interna da Turbina.



Fonte: (GE, 2021).

Conforme ilustrado na Figura 16, o sistema SCADA precisa se comunicar efetivamente com o controlador, ler parâmetros da turbina e exibi-los em um display para supervisão, além de ser capaz de escrever comandos na turbina para alterar esses parâmetros.

Figura 16 – Visão Global da Aplicação Fictícia.



Fonte: O Autor.

3.2 IMPLEMENTAÇÃO DO CABEÇALHO

3.2.1 Análise do Protocolo

O processo de compreender o funcionamento de um protocolo tem como base a documentação disponível acerca do mesmo, então inicia-se o processo de compreensão e posterior construção do cabeçalho que viabilizará a identificação do protocolo na rede. É importante ressaltar que, devido à ampla aplicação em dispositivos diversos, é comum deparar-se com modificações específicas no protocolo que agregam ou eliminam funcionalidades a fim de adequar a comunicação aos dispositivos desejados. Dessa forma, a documentação disponível nem sempre fornece toda a informação necessária para o desenvolvimento do driver de comunicação para a aplicação desejada.

Tendo como exemplo os *gateways* fornecidos pela empresa Prosoft Technology, conhecida mundialmente pelo desenvolvimento de soluções de comunicação industrial para controle e automação. Existem diversos tipos de *gateways* especializados em traduzir um protocolo para outros protocolos mais amplamente utilizados, e vice-versa. Por exemplo, alguns *gateways* convertem protocolos como Modbus para outros, enquanto outros fornecem interfaces de comunicação para controladores como ControlLogix e CompactLogix.

Importante destacar o fato de ser comum encontrar adaptações específicas no protocolo para otimizar a comunicação com diferentes equipamentos, conforme abordado no próximo parágrafo. *Gateways* desempenham um papel crucial na integração de sistemas heterogêneos, permitindo que dispositivos e sistemas que operam em protocolos diferentes se comuniquem de forma eficiente. Ao analisar a documentação do equipamento, é verificado que existem comandos específicos desenvolvidos pela empresa. Um exemplo disso é o comando de solicitação de leitura, conforme ilustrado na Figura 17.

Verifica-se que o pacote é encapsulado no UDP. Os campos do UDP, como destino, origem e comprimento, estão em conformidade com a documentação do protocolo UDP. Ao analisar o pacote na solicitação de leitura, observa-se a presença dos campos esperados. No entanto, também são identificados outros campos adicionais específicos para a solicitação de leitura do *gateway* como o campo Offset que determina o endereço específico para realizar a leitura.

Figura 17 – Solicitação de Leitura.

Packet Info	Description		Data		
UDP	Source Port:		7937 (0x 1F01)		
	Destination Port:		7937 (0x 1F01)		
	Length:		xx bytes (0x 00 XX)		
	Checksum:		xxxxx (0x XX XX)		
			Offset	**Bytes**	**Hex code**
		PDU Type:	0	1	0x 20
		Message Flag:	1	1	0x HH
		Request ID:	2	2	0x HH HH
		PVN1:	4	1	0x 01
		Option Length:	5	1	0x HH
		Message Length:	6	2	0x HH HH
		Configuration Signature:	8	2	0x HH HH
		Address Type:	10	1	0x HH
		Cell Count:	11	1	0x HH
		Producer ID:	12	4	0x HH HH HH HH
		Exchange ID:	16	4	0x HH HH HH HH
		Time Stamp:	20	8	0x HH . . . HH
		Reserved:	28	4	0x HH HH HH HH
		Offset: reads the DB Register specified in the Command. **Length:** is Bytes count, not Words. Multiple Cells may be requested dependent on the Cell Count field specified above.			
		Offset:	32	2	0x HH HH
		Length:	34	2	0x HH HH

Fonte: Adaptado de (PROSOFT, 2010).

Outro exemplo é o comando *Summary Request* na Figura 18, que solicita as informações de todas as Exchanges configuradas no *node* correspondente e recebe como resposta o comando *Summary Response* na Figura 19, contendo a lista das informações referentes a cada Exchange presente no node.

Importante dar ênfase ao motivo da implementação dos comandos *Summary Request* e *Response* pelos engenheiros da Prosoft. No comando de Request o pacote contém poucos campos, divergindo apenas no campo PDU Type do padrão. Esse elemento é o que permite à entidade receptora do comando identificá-lo como um *Summary Request*.

Já o comando de resposta é mais complexo, com o seu PDU Type igual a 8, representando o cabeçalho padrão. No entanto, a carga útil contém informações de todos os Exchanges configurados neste node. Dessa forma, o pacote tem tamanho dinâmico, dependendo da quantidade de Exchanges. É comum informar ao receptor o tamanho da carga útil que o pacote contém para possibilitar a interpretação dos dados, e neste caso, o campo Total Cells desempenha esse papel, informando a quantidade Exchanges, dessa forma o *Summary Response* contém uma lista das Exchanges configuradas no node com suas respectivas informações como Producer ID e State.

Figura 18 – *Summary Request.*

Packet Info	Description		Data		
UDP	Source Port:		7937 (0x 1F01)		
	Destination Port:		7937 (0x 1F01)		
	Length:		xx bytes (0x 00 XX)		
	Checksum:		xxxxx (0x XX XX)		
			Offset	**Bytes**	**Hex code**
		PDU Type:	0	1	0x 07
		Message Flag:	1	1	0x HH
		Request ID:	2	2	0x HH HH
		PVN1:	4	1	0x 01
		Reserved:	5	1	0x HH
		Message Length:	6	2	0x HH HH
		Index:	8	1	0x HH
		Reserved:	9	3	0x 00 00 00

Fonte: Adaptado de (PROSOFT, 2010).

Figura 19 – *Summary Response.*

Packet Info	Description		Data		
UDP	Source Port:		7937 (0x 1F01)		
	Destination Port:		7937 (0x 1F01)		
	Length:		xx bytes (0x 00 XX)		
	Checksum:		xxxxx (0x XX XX)		
			Offset	**Bytes**	**Hex code**
		PDU Type:	0	1	0x 08
		Message Flag:	1	1	0x HH
		Request ID:	2	2	0x HH HH
		PVN1:	4	1	0x 01
		Reserved:	5	1	0x HH
		Message Length:	6	2	0x HH HH
		Index:	8	1	0x HH
		Producer ID:	9	4	0x HH HH HH HH
		Total Cells: (Qty of exchange cells defined for this node)	13	2	0x HH HH
		Cells contained in the message:	15	2	0x HH HH
		Producer ID:	17	4	0x HH HH HH HH
		Exchange ID:	21	4	0x HH HH HH HH
		Mode:	25	1	0x HH
		Reserved:	26	1	0x HH
		State:	27	2	0x HH HH
		Production Period:	29	4	0x HH HH HH HH

Fonte: Adaptado de (PROSOFT, 2010).

Esses casos demonstram que os protocolos, principalmente os proprietários, podem ser modificados e frequentemente o são, a fim de se adaptarem a aplicações específicas. Isso dá liberdade aos engenheiros e desenvolvedores para implementar as soluções mais adequadas, indo além das especificações do protocolo. Por exemplo, mesmo operando em UDP na camada de transporte, é possível fazer uso das camadas superiores para otimizar a confiabilidade da comunicação nesse cenário. O pacote é composto pelo cabeçalho e carga

útil. Nesta etapa do desenvolvimento, foram considerados apenas os dados relacionados ao cabeçalho.

Na Figura 20, é apresentada uma amostra de comando. O cabeçalho é destacado em azul, enquanto o restante do comando, a carga útil destacada em amarelo, representa o valor das variáveis a serem transmitidas. Cada posição no comando é representada por um valor hexadecimal equivalente a 1 byte. Nesse exemplo, o comando é pequeno, é comum existir um limite para o tamanho do comando, assumindo que o número máximo de dados que podem ser transmitidos em um comando EIP é de 1000 bytes, ao somar os bytes do cabeçalho, o tamanho máximo do pacote EIP é de 1024 bytes.

O limite de tamanho é relevante para garantir a transmissão eficiente e evitar problemas de congestionamento ou perda de dados. É necessário considerar essa restrição ao projetar a comunicação a fim de garantir o desempenho adequado e a integridade dos dados transmitidos.

Figura 20 – Amostra de Comando.

```
0xA 0x01 0x91 0x36 0xAC 0x17 0x15 0xB0 0x01 0x00 0x00 0x00 0x73 0x32 0xDD 0x63
0x74 0x16 0x8E 0x0F 0x00
0x07 0x00 0x00 0x00 0x12 0xDE 0xDC 0x63 0x72 0x32 0xDD 0x63 0x69 0x57 0x75 0x5E 0x6C 0x60 0x77
```

Fonte: O Autor.

No protocolo EIP, todos os valores em hexadecimal estão no formato *little-endian*, o que significa que os bytes menos significativos são armazenados primeiro, seguidos pelos bytes mais significativos. Isso implica que, ao representar um valor hexadecimal de vários bytes, o byte de menor valor será armazenado no endereço de memória mais baixo.

Para demonstrar o processo de construção do cabeçalho, é utilizado um exemplo com a amostra de comando da Figura 20. No caso do Producer ID, ele possui uma representação em formato *dotted decimal*, semelhante a um endereço IP. No entanto, ele também pode ser representado em formato decimal padrão. Neste caso, com o Producer ID igual a 172.23.21.176, obtém-se a representação em decimal como 2954172332, que, ao ser convertida para hexadecimal em *little-endian*, resulta em `0xAC 0x17 0x15 0xB0`.

O *timestamp* está representado em formato legível por humanos. Convertendo-o para o formato *UNIX timestamp*, obtém-se o valor 1675437155 em decimal. Em seguida, ao converter esse valor para o formato hexadecimal em formato *little-endian*, obtém-se a sequência: `0x73 0x32 0xDD 0x63 0x74 0x16 0x8E 0x0F`.

Na Tabela 2, estão listados os campos dinâmicos, ou seja, aqueles que mudam de acordo com o comando enviado.

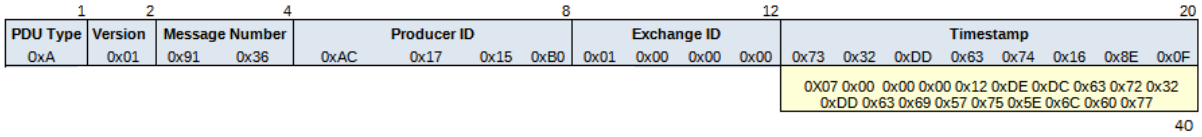
campos	**Decimal**	**Hexadecimal**
Producer ID	295417233	`0xAC 0x17 0x15 0xB0`
Exchange ID	1	0x01
Timestamp	Friday, February 3, 2023 4:12:35 PM.260	`0x73 0x32 0xDD 0x63 0x74 0x16 0x8E 0x0F`
Message Number	13969	`0x91 0x36`

Tabela 1 – Valor dos Campos Dinâmicos do Comando.

Com base nas informações obtidas, é possível determinar a localização dos campos no cabeçalho do comando. A Figura 21 ilustra a posição de cada campo e seu valor correspondente na amostra de comando. demonstrando que o cabeçalho é consistentemente composto por 20 bytes, a carga útil, para esse comando especificamente também contém 20 bytes, totalizando 40 bytes.

Conforme os padões inventados para o protocolo EIP, determina-se o seguinte padrão, o campo PDU, que por sua vez é uma unidade de dados que representa as informações transmitidas por meio do protocolo, ou seja, o receptor dos comandos identifica o protcolo através do PDU, como neste caso, é comum encontrar o PDU na primeira posição do pacote, de modo que o receptor descarta o pacote sem precisar ler os demais campos. Pode ser encontrado com outros nomes, como no caso representado pelo campos Data Type nos *gateways* fornecidos pela Prosoft. Determina-se que todos os comandos contêm o valor `0xA` para o PDU.

Figura 21 – Comando Identificado.

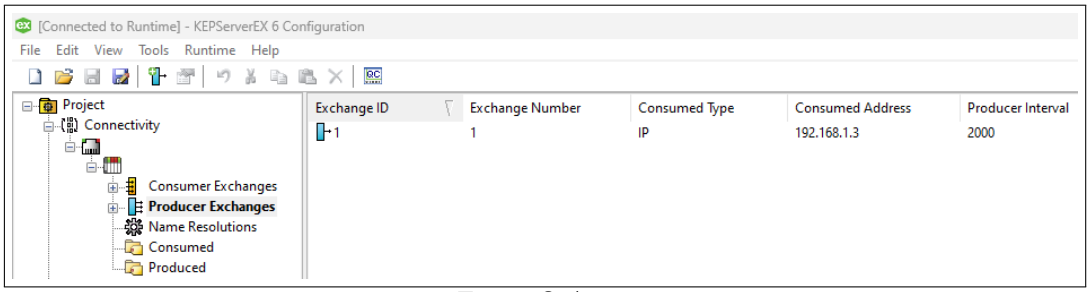


Fonte: O Autor.

3.2.2 Composição do Cabeçalho

Dado que não existe documentação técnica disponível acerca das especificações do protocolo EIP, para assegurar a composição correta do cabeçalho, foi utilizando o KEPServerEX, Trata-se de um servidor de comunicação industrial que oferece suporte a uma ampla variedade de protocolos usados na indústria, e é certificado e adequado para esse propósito específico.

Figura 22 – Exemplo de Configuração de Protocolo no KEPServerEX.



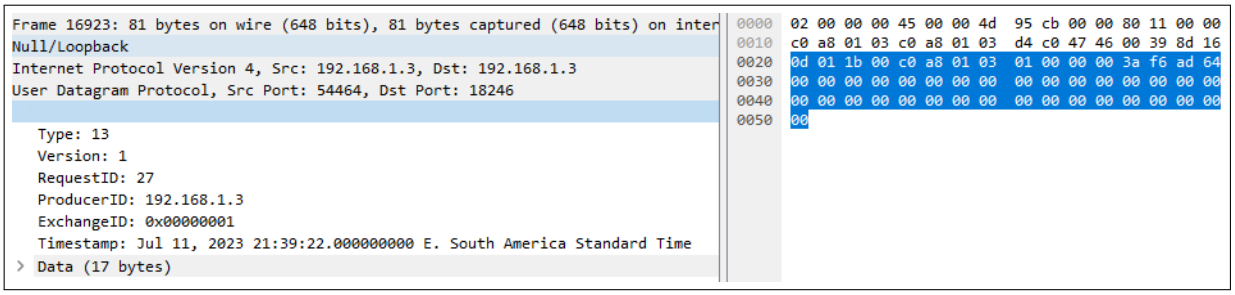
Fonte: O Autor.

Conforme a Figura 22 foi realizada a configuração do protocolo no KEPServerEX, configurados o Producer, Consumer, número da Exchange, Endereços IP e intervalo de compartilhamento. Feito isso, o servidor do KEPServerEX realiza a comunicação entre o Producer e Consumer executados localmente, para verificar o conteúdo dos pacotes é utilizado o Wireshark capturando os comandos na rede e identificando o fluxo de comandos EIP em UDP. O Wireshark é uma ferramenta de análise de protocolos de rede e captura de pacotes. Ele permite examinar o tráfego de rede em tempo real ou analisar arquivos de captura previamente gravados.

Conforme em azul na Figura 23 o pacote EIP mantém uma estrutura idêntica aquela encontrada nos comandos fornecidos, os dados recebidos estão em conformidade com as informações enviadas pelo Producer configurado no KEPServerEX. Vale ressaltar a importância da consistência na estrutura do pacote recebido, de modo que oo comando é identificado como EIP e todos os campos estão presentes com o valor esperado, assegurando a comunicação eficiente. É importante observar que a carga útil, representada na Figura 23 como o campo Data, contendo 17 bytes, será analisado com mais detalhes no capítulo 3.3.

A captura de pacotes no Wireshark facilita a compreensão da estrutura do cabeçalho, visto que é exibido de forma detalhada desde o encapsulamento IPV4 e UDP até o EIP. Isso permite identificar as portas envolvidas e assegurar o significado de cada byte no pacote. Os campos Type e Version são constantes no Wireshark, sendo representados pelos valores dos campos PDU Type e PDU Version Number do protocolo, enquanto o Request ID se assemelha ao Message Number dos dados fornecidos.

Figura 23 – Pacote EIP no Wireshark Enviado pelo KEPServerEX.

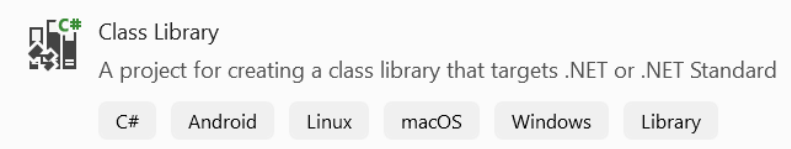


Fonte: O Autor.

3.2.3 Preparação do Ambiente e Início do Desenvolvimento do Driver

Após obter todas as informações necessárias para a implementação do cabeçalho, dá-se início ao projeto de desenvolvimento do driver utilizando o ambiente de desenvolvimento integrado Visual Studio, o ambiente de desenvolvimento integrado fornecido pela Microsoft. O driver será compilado como uma DLL e posteriormente chamado pela DriverAPI do SCADA. Todo o desenvolvimento do projeto é realizado no Framework .NET, garantindo a compatibilidade e integração com o SCADA. Conforme a Figura 24, o projeto é criado como uma Class Library em cima do .NET Framework, assim como o SCADA que é completamente desenvolvido no .NET Framework, tendo acesso a uma documentação extensa, uma comunidade ativa e um ecossistema moderno com suporte da Microsoft.

Figura 24 – Criação do Projeto do Driver no Visual Studio.



Fonte: O Autor.

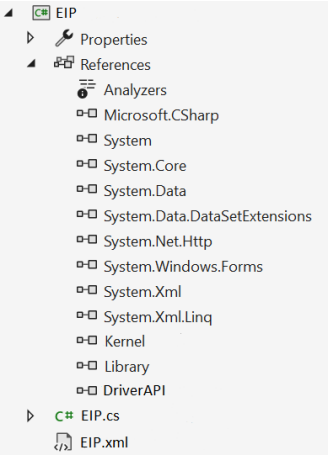
No âmbito deste projeto, são feitas referências diretas ao SCADA para possibilitar a integração com os projetos de engenharia no SCADA. O SCADA é equipado com uma API, DriverAPI, a qual é chamada pelo do módulo Devices, responsável pela configuração dos Canais, nodes e Pontos de comunicação com dispositivos externos.

A DriverAPI assume a responsabilidade de abstrair as complexidades das camadas de transporte e rede durante o processo de desenvolvimento do driver. Ela fornece uma interface consistente e simplificada para que o driver possa se comunicar efetivamente com essas camadas subjacentes. Com isso, o desenvolvedor pode concentrar seus esforços na lógica do driver e na integração com o SCADA, sem se preocupar com os detalhes específicos da implementação da comunicação em nível de transporte e rede.

A configuração da comunicação entre o Driver e a API é possível utilizando referências diretas ao SCADA no projeto do driver, conforme a Figura 25 onde apresenta as

referências diretas as DLLs: Kernel, Library e DriverAPI do SCADA, dessa forma disponibilizando uma variedade de recursos e funcionalidades prontamente disponíveis. Essas referências, conforme a Figura 25, incluem classes, métodos e propriedades especialmente projetados para trabalhar de forma harmoniosa com o SCADA, o driver e a API.

Figura 25 – Exemplo de Referênciais ao SCADA no projeto do Driver.



Fonte: O Autor.

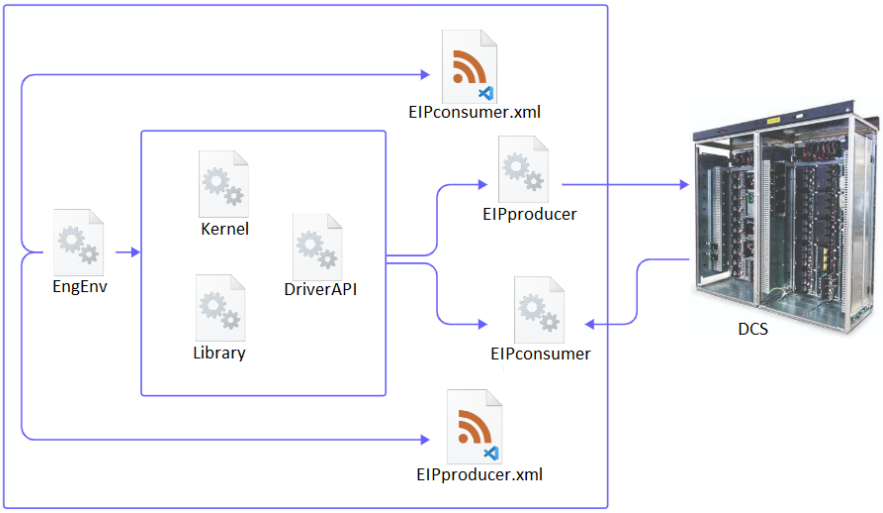
A Figura 26 fornece uma visão simplificada de um exemplo de arquitetura de aplicação, destacando o modelo informal da relação entre as DLLs do SCADA, responsáveis pela comunicação com o DCS. Os arquivos XML de configuração do protocolo são acessados pela DLL EngEnv, que apresenta as informações configuradas na interface gráfica, permitindo ao usuário personalizar características específicas da comunicação, como endereço IP, porta e campos particulares, como no caso do EIP, Producer ID, Exchange.

A DLL EngEnv desempenharia um papel crucial na exibição e controle do ambiente de engenharia do software. Ela seria chamada pelo Kernel durante o início do software. Assim que o ambiente de engenharia é carregado, os arquivos XML dos protocolos são acessados, possibilitando ao usuário realizar configurações. Por outro lado, a DLL DriverAPI, conforme descrita, assumiria a responsabilidade de gerenciar a comunicação com os drivers. Ela acessa as DLLs dos drivers e administra a comunicação, operando de forma isolada.

A Library seria uma biblioteca que contém ferramentas úteis para o desenvolvimento no Framework, proporcionando recursos valiosos para facilitar o processo de criação. Finalmente, o Kernel serve como uma interface para acessar as Tags armazenadas no sistema, fornecendo uma camada de abstração para a interação com os dados do sistema SCADA.

Essa arquitetura parece ser projetada para oferecer uma abordagem modular e flexível, permitindo a personalização das configurações de comunicação e proporcionando ferramentas úteis para o desenvolvimento eficiente no SCADA.

Figura 26 – Exemplo de Arquitetura SCADA.



Fonte: O Autor.

Seria utilizado um arquivo XML para fornecer uma padronização no compartilhamento de informações de configuração do driver que vão ser interpretadas pelo SCADA, contendo detalhes importantes, como o protocolo utilizado na camada de transporte, que pode ser TCP, UDP ou outro específico para o projeto. Essas informações permitem que o sistema SCADA configure adequadamente o driver para se comunicar com as camadas apropriadas.

O arquivo XML conteria também o nome do driver, fornecendo uma identificação única, abrange também um conjunto de informações específicas do driver configuradas pelo usuário na interface gráfica do SCADA. As informações podem variar dependendo das necessidades do driver e das funcionalidades específicas que ele oferece. Esses dados incluem parâmetros como: endereços IP, portas de comunicação, configurações de autenticação, entre outros elementos relevantes para o funcionamento do driver.

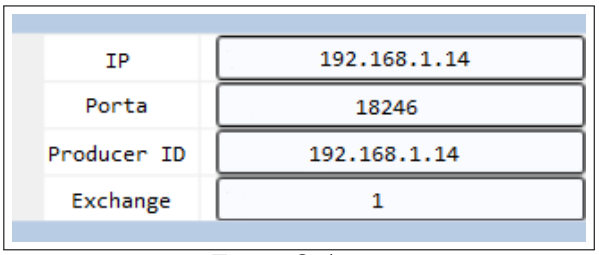
Conforme o XML na Figura 27, o driver é configurado como UDP tendo como porta padrão 18246, porta utilizada pelo protocolo EIP, dentro da estrutura definida pelo UDP na Figura 28, estão os campos específicos e os valores padrão, assim como o tamanho da janela de exibição dessas informações, o SCADA interpreta o XML e gera na interface do usuário a janela de exibição e edição das configurações do driver para o node, conforme a Figura 28.

Figura 27 – XML para Configuração do Driver.

```
<Settings AllowCustomTypes="false">
        <UDP>
            <DefaultValues P1="false" P2="true" P3="18246" P4="1"/>
            <EditModes P1="Hidden" P2="Enable" P3="Enable" P4="Enable" />
        </UDP>
</Settings>
    <Station>
        <UDP>
            <Names P1="ProducerID" P2="ExchangeID"
            <DefaultValues P1 ="192.168.1.1" P2="1" P3="1.1"/>
            <EditModes P1="Enable" P2="Enable" P3="Enable"/>
            <EditTypes P1="string" P2="int" P3="string"/>
            <Hint P1 ="Producer ID Dotted-Decimal Form" P2 ="Specific Data Exchange Number"
            <GridProperties G1="PopUpWidth" G2="PropertyColumnWidth" />
            <GridValues G1="260" G2="90" />
        </UDP>
    </Station>
```

Fonte: O Autor.

Figura 28 – Exemplo de Interface Gráfica de Configuração do Node.



Fonte: O Autor.

Inicializando o desenvolvimento da lógica do driver, Conforme a Figura 29, é implementada a interface IComm e a classe , ambas pertencentes a DriverAPI, a interface define os métodos que devem ser utilizados fornecendo uma sequência lógica do funcionamento do driver, desde a sua inicialização até o envio e recebimento dos comandos.

A classe DriverDef é responsável por substituir os métodos definidos na interface IComm, permitindo o acesso aos objetos e propriedades dos métodos implementados na API. Essa acessibilidade é viabilizada no driver por meio da classe DriverDef.

Dentro do método Initnode, o driver coleta as informações inseridas pelo usuário na interface gráfica do ambiente de engenharia do SCADA e armazena essas informações em variáveis. Essas variáveis são então utilizadas para criar uma instância da classe ConfigNode, que contém todos os dados pertinentes ao node, então o objeto da classe ConfigNode é armazenado na lista de nodes do objeto global protocol, a Figura 30 apresenta a implementação das classes.

Figura 29 – Inicialização do Driver.

```
namespace
{
    0 references
    public class Producer : DriverDef, IComm
    {
        private Protocol protocol;

        0 references
        public override InicializarDriverParams

        {
            try
            {
                protocol = new Protocol (dev, Module);
            }
            catch (Exception ex)
            {
                Exception.Log(ex);
                return eReturn.FAILED;
            }
            return eReturn.SUCCESS;
        }

        0 references
        public override PrepararNode (ConfigNode node)
        {
            try
            {
                string [] nodeParams = node.PrimaryStation.StrValue.Split(';');

                string ProducerID = nodeParams[0];
                int ExchangeID = Convert.To<int>(nodeParams[1]);
                int MessageNumber = 0;


                ConfigNode nodeDef = new ConfigNode (node.Name, ProducerID, ExchangeID,
                MessageNumber);

                protocol.NodeDefinition.Add(nodeDef);

                return Return.SUCCESS;
            }
            catch (Exception ex)
            {
                Exception.Log(ex);
                return eReturn.FAILED;
            }
        }
```

Fonte: O Autor.

A partir da Figura 30, a classe ConfigNode desempenha a função de definir os atributos dos nodes do projeto, conforme especificado no protocolo EIP. Por sua vez, a classe ProtocolDescription desempenha um papel essencial ao armazenar os nodes em uma lista, além disso, utiliza um dicionário para possibilitar o acesso e verificação da quantidade de nodes configurados no driver.

Essa estrutura de classes revela a organização e manipulação dos nodes no protocolo EIP, proporcionando uma representação estruturada para gerenciar as informações necessárias à comunicação e configuração do driver.

Figura 30 – Implementação das Classes.

```
private class Protocol
{
    public int MaxAddressSize = 1000;
    public List< ConfigNode > NodeDefinition;

    //Store multiple Nodes name and IP
    public Dictionary<string, string> Nodes = new Dictionary<string, string

}

6 references
public class ConfigNode
{
    private string nodeName = "";
    private string producerID = "";
    private int exchangeID = 0;
    private int messageNumber = 0;


    1 reference
    public ConfigNode (string nodeName, string producerID, int exchangeID,
    int messageNumber)
    {
        this.nodeName = nodeName;
        this.producerID = producerID;
        this.exchangeID = exchangeID;
    }


    1 reference
    public string NodeName { get { return nodeName; } }
    1 reference
    public string ProducerID { get { return producerID; } }
    4 references
    public int ExchangeID { get { return exchangeID; } }
    1 reference
    public int MessageNumber { get { return messageNumber; } }

    1 reference
    public int UpdateMessage(int CurrentMessage)
    {
        return messageNumber++;
    }

}
```

Fonte: O Autor.

Dando continuidade, após as informações referentes ao node serem armazenadas, o próximo passo consiste em verificar os endereços configurados para aquele node específico. Vale ressaltar que o driver está sendo utilizado pela DriverAPI para realizar a comunicação. A classe ConfigNode, que fornece os nodes, vem da API, assim como a classe ConfigItem, que fornece os itens configurados no projeto. Cada item contém um endereço específico. Conforme o algoritmo na Figura 31, o método ExtractAddress, realiza a operação de Parse do endereço dos itens, onde são retirados o BlockId e o AddressVar. O BlockId representa a posição do byte, enquanto o AddressVar indica o bit do item na carga útil do pacote EIP.

O método OrdenarConfig organiza os Items em ordem com base no byte pelo atributo BlockId e no bit pelo atributo AddressVar, estabelecendo um padrão ao acessar e editar esses Items.

Figura 31 – Obtenção e Ordenação dos Items.

```
public override ExtractAddress (string Address, ConfigItem Item)
{
    try {

        if (Regex.IsMatch(Address, @"^\d+\.\d+$"))
        {
            Item.BlockId = Convert.To<int>(Item.Address.Split('.')[0]);
            Item.AddressVar = Convert.To<int>(Item.Address.Split('.')[1]);
        }
        else
        {
          Error("Invalid Address Format: " + Address);
        }

        return eReturn.SUCCESS;
    }
    catch (Exception ex)
    {
        TException.Log(ex);
        return eReturn.FAILED;
    }
}

0 references
public override int OrdenarConfig ( ConfigItem itemMain, ConfigItem itemNext)
{
    try
    {
        //Operand
        if (itemMain.BlockId > itemNext.BlockId)
            return 1; //Main > Next
        if (itemMain.BlockId < itemNext.BlockId)
            return -1; //Main < Next

        //Address Offset
        if (itemMain.AddressVar > itemNext.AddressVar)
            return 1; //Main > Next
        if (itemMain.AddressVar < itemNext.AddressVar)
            return -1; //Main < Next
    }
    catch (Exception ex)
    {
        Exception.Log(ex);
        return 1;
    }
    return 0; // equal
}
```

Fonte: O Autor.

Após a conclusão das operações nos nodes e nos Items, a etapa subsequente envolve a organização dos Items em blocos. Os blocos representam uma maneira de agrupar e segmentar os Items de acordo com as especificações definidas pelo desenvolvedor. Similar aos procedimentos anteriores, esse método é invocado pela API, e todos os Items configurados para o Canal são fornecidos, ou seja, todos os Items de todos os nodes.

Dado que um Canal pode possuir varios nodes os Blocos são delineados com base no node ao qual o Item está associado. Desta forma, como os Items já estão ordenados pelo seus respectivos bytes e bits pelo método OrdenarConfig, agora serão agrupados em blocos determinados pelo node ao qual pertencem. A partir do algoritmo na Figura 32, o primeiro parâmetro do método, denominado first, representa o primeiro Item configurado pelo usuário, enquanto o parâmetro item representa o próximo Item. A API utiliza esse método passando todos os Items, um de cada vez, para os parâmetros item e first, realizando assim comparações entre todos os Items. Se a diferença entre o byte do último item recebido pelo método e o byte do primeiro item for maior que o tamanho máximo suportado pelo protocolo EIP, ou seja, 1000 bytes, a inconsistência é informada ao usuário. Quando o

método retorna *true*, a API agrupa os Items no mesmo Bloco; caso retorne *false*, os Items não pertencem ao mesmo Bloco.

Figura 32 – Agrupamento de Blocos.

```
public override bool ComparaBlocos(ConfigItem first, ConfigItem item)
{
    try
    {
        int dataBufferSize = (item.BlockId - first.BlockId + 1);

        if (dataBufferSize > protocol.MaxAddressSize)
        {
            CommAPI.Trace("Data Buffer Size Exceeded");
            dataBufferSize = protocol.MaxAddressSize;
        }

        if (first.NodeName != item.NodeName)
        {
            return false; //Different Block
        }

        return true; // Same Block
    }
    catch (Exception ex)
    {
        Exception.Log(ex);

        return false;

    }
}
```

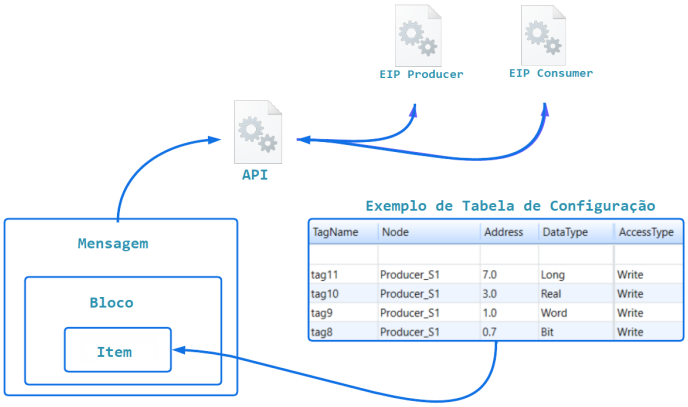
Fonte: O Autor.

A Figura 33 representa um modelo informal que torna visível a origem dos objetos discutidos e a relação entre eles. No modelo, a API realiza chamadas aos Drivers, e a Mensagem flui entre a API e os Drivers, transportando o pacote de dados a ser enviado e informações sobre a configuração do Driver.

Cada Item pertence a um Bloco, onde as informações configuradas na interface gráfica do módulo Devices são encapsuladas. Cada linha configurada na interface do Devices representa um Ponto de Comunicação, incluindo uma Tag associada, node, Endereço e tipo de dado, os tipos de dados dos Pontos de comunicação vão ser tratados mais a frente. Cada node, por sua vez, pertence a um Canal. No contexto do EIP, cada node pode ser um Producer ou Consumer, identificado por um ID e Exchange específicos. O endereço é composto por um byte seguido de um bit, determinando a posição da carga útil para escrita (no caso do Producer) ou leitura (no caso do Consumer). As informações referentes ao Endereço são armazenadas no Item, e esses Items são segregados em Blocos, separados no Driver de acordo com o node ao qual pertencem.

Os Blocos, por sua vez, são armazenados na mensagem. Cada comando recebido ou enviado pelo canal cria uma instância da classe Mensagem, sendo tratada pela API em conjunto com os Drivers. A API gerencia diversos nodes e Canais, contendo informações confidenciais que não podem ser detalhadas neste trabalho.

Figura 33 – Modelo Relacional.



Fonte: O Autor.

A próxima etapa consiste em construir o pacote utilizando o método PrepComando responsável por construir a parte do pacote correspondente ao protocolo EIP, como essa parte do driver se refere exclusivamente ao Producer, apenas comandos de escrita são enviados. Ao iniciar a construção do cabeçalho primeiramente é limpado o buffer como medida de segurança, os sistemas normalmente possuem os buffers Tx e Rx, o buffer Tx, conhecido como buffer de transmissão, é uma área de armazenamento em um sistema onde os dados de saída são temporariamente armazenados antes de serem transmitidos por um Canal de comunicação.

O buffer Rx, ou buffer de recepção, é onde os dados recebidos são temporariamente armazenados após terem sido recebidos pelo Canal de comunicação. Ele serve como uma área de armazenamento temporário até que o sistema esteja pronto para processar os dados recebidos. Os buffers são desenvolvidos internamente, constituem um vetor de bytes pertencente a mensagem. A mensagem, definida como *msg*, é um objeto da classe Mensagem, esta classe é de importância fundamental porque contém os Blocos e os Buffers, ela é utilizada pela API tanto para enviar como receber os bytes das camadas inferiores, dependendo se a operação é de escrita ou leitura respectivamente.

Conforme as Figuras 34 e 35, primeiramente é verificado se é um comando de escrita sendo enviado para o Driver, caso sim, é encontrado o node especifico que enviou o comando com base nos Items contidos no bloco da mensagem, conforme explicado anteriormente. No algoritmo é apresentado cada campo do cabeçalho do protocolo, sendo as duas primeiras posições do TxBuffer fixas e as demais preenchidas com as informações do node específico, são utilizadas operações de bishift e bitwise para obter os bytes dos campos em *little-endian* e armazenar no buffer Tx. Importante ressaltar que alguns campos são dinâmicos, mesmo com quantidade fixa de bytes, tendo o valor conforme inserido na configuração do node, e outros são estáticos com valor fixo, totalizando os 20 bytes do cabeçalho EIP.

Figura 34 – Implementação do Cabeçalho EIP - Parte 1.

```
public override PrepComando (Mensagem msg)
{
    try
    {
        if (msg.Block.CmdType == Write)
        {
            msg.TxBuffer.Clear();
            msg.RxBuffer.Clear();

            //Blocks are built by node names
        ConfigNode CurrentNode = protocol.NodeDefinition.Find(node => node.NodeName ==
            msg.Block.FirstItem.NodeName);

            ////Little-Endian
            msg.TxBuffer.Clear();

            //PDU type
            msg.TxBuffer += 0x0D; //00

            //PDU version number
            msg.TxBuffer += 0x01; //01

            //Request ID
            int MessageNumber = CurrentNode.UpdateMessage(CurrentNode.MessageNumber);

            msg.TxBuffer += Convert.To<byte>(MessageNumber & 0xFF); //02
            msg.TxBuffer += Convert.To<byte>((MessageNumber >> 8) & 0xFF); //03

            //Producer ID
            string[] octets = CurrentNode.ProducerID.Split('.');

            msg.TxBuffer += Convert.To<byte>( Convert.To<int>(octets[0])); //04
            msg.TxBuffer += Convert.To<byte>( Convert.To<int>(octets[1])); //05
            msg.TxBuffer += Convert.To<byte>( Convert.To<int>(octets[2])); //06
            msg.TxBuffer += Convert.To<byte>( Convert.To<int>(octets[3])); //07
```

Fonte: O Autor.

Figura 35 – Implementação do Cabeçalho EIP - Parte 2.

```
//Exchange ID
msg.TxBuffer += Convert.To<byte>(CurrentNode.ExchangeID & 0xFF); //08
msg.TxBuffer += Convert.To<byte>((CurrentNode.ExchangeID >> 8) & 0xFF); //09
msg.TxBuffer += Convert.To<byte>((CurrentNode.ExchangeID >> 16) & 0xFF); //10
msg.TxBuffer += Convert.To<byte>((CurrentNode.ExchangeID >> 24) & 0xFF); //11

//Timestamp
byte[] byteTime = null;
byteTime = BitConverter.GetBytes(DateTimeOffset.UtcNow.ToUnixTimeSeconds());

msg.TxBuffer += byteTime[0]; //12
msg.TxBuffer += byteTime[1]; //13
msg.TxBuffer += byteTime[2]; //14
msg.TxBuffer += byteTime[3]; //15
msg.TxBuffer += byteTime[4]; //16
msg.TxBuffer += byteTime[5]; //17
msg.TxBuffer += byteTime[6]; //18
msg.TxBuffer += byteTime[7]; //19
```

Fonte: O Autor.

3.3 IMPLEMENTAÇÃO DA CARGA ÚTIL

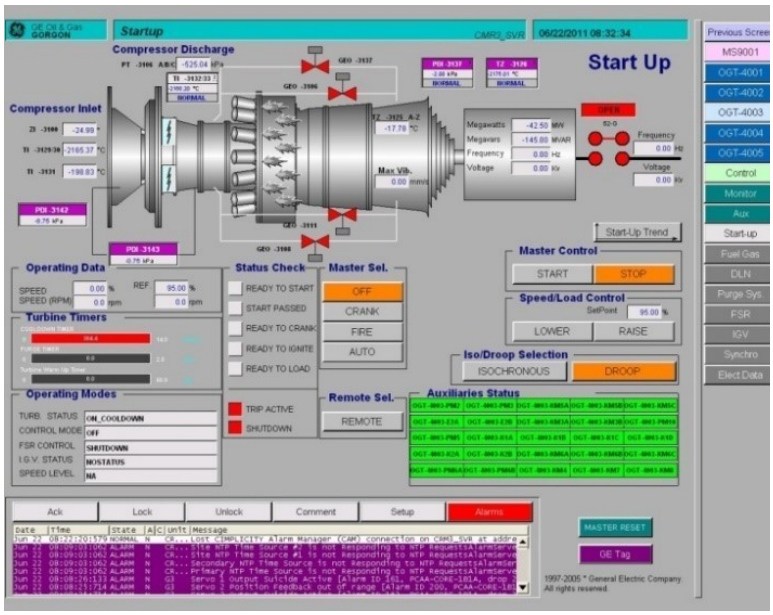
A carga útil, também conhecida como *payload*, é uma parte essencial de um pacote de dados em protocolos de comunicação. Em um contexto de transmissão de dados entre dispositivos ou sistemas, o pacote, como discutido anteriormente, geralmente é composto por duas partes: o cabeçalho e a carga útil.

Na seção anterior, a implementação do cabeçalho foi detalhadamente demonstrada. Nesta seção, serão apresentados os passos tomados para o desenvolvimento do algoritmo que implementa a carga útil do pacote no projeto do driver. Neste driver a carga útil é composta pelo conjunto de valores que representam as variáveis físicas da turbina sendo

controlada. Esses valores são utilizados para a troca de informações entre o driver e o DCS, possibilitando a comunicação bidirecional entre os dispositivos. Ao se comunicar com o DCS, o Driver recebe e envia os parâmetros de controle do equipamento, que podem incluir dados como pressão, velocidade, temperatura, referência e Ponto de ajuste, entre outros.

No SCADA os valores são armazenados em variáveis internas denominadas Tags, estas por sua vez são organizadas em uma estrutura hierárquica no sistema SCADA, facilitando a categorização e o acesso rápido aos dados. Os operadores do sistema podem visualizar as informações das Tags em telas de supervisão, realizar ajustes nos parâmetros de controle e monitorar o desempenho dos equipamentos.

Figura 36 – Exemplo de Projeto de Visualização de Turbina.



Fonte: (SSE, 2020).

A Figura 36 exibe a tela de supervisão, contendo um projeto exemplo de controle e supervisão de uma turbina, os valores exibidos são os valores das Tags do projeto, que são configuradas para ler e escrever os dados na rede, no caso do Producer as Tags preenchem a carga útil do pacote EIP e o comando é enviado pela rede, esse processo ocorre periodicamente conforme especifica o protocolo.

Para o Consumer, como será demonstrando posteriormente, as Tags armazenam o valor dos dados contidos na carga útil do pacote recebido, e então o valor é exibido na tela de supervisão. Normalmente a tela de supervisão do SCADA é compatível tanto com a tecnologia HTML5 quanto com a plataforma WPF, ou seja, pode ser exibida na rede pelo navegador ou em janela no Windows.

O procedimento de elaboração da carga útil consiste na inclusão dos valores das Tags no pacote. Para concretizar esse processo, é essencial determinar o tipo de dado, o qual determina a quantidade de bits necessária para representar integralmente cada

valor. A diversidade de tipos de dados respaldados pelo protocolo encontra-se ilustrada na Tabela 2.

Tipo	**Tamanho**	**Range de Valor**
Bit	1 bit	0 ou 1
Word	2 bytes ou 16 bits	-32768 até 32768
Real	4 bytes ou 32 bits	-9.99×10^{37} até 9.99×10^{37}
Long	8 bytes ou 64 bits	-9.22×10^{18} até 9.22×10^{18}

Tabela 2 – Tipos de Dados Suportados pelo Protocolo.

As Tags são configuradas para realizar operações de leitura e escrita nos Pontos de comunicação associados ao node dentro do Canal de protocolo correspondente, a Figura 37 apresenta um exemplo de configuração dos Pontos do protocolo EIP. Conforme explicado anteriormente cada Ponto possui atributos específicos, como endereço, tipo de dado e tipo de acesso. No contexto do protocolo EIP, o endereço é fundamental para localizar o valor da tag na carga útil. Esse endereço é formado por duas partes: a posição do byte e, se o tipo de dado for Bit, a posição do bit, separadas por um Ponto. No caso de um tipo de dado que não seja Bit, o valor do Bit é zero.

O tipo de dado desempenha um papel crucial ao determinar quantos bits o valor da Tag ocupará no pacote. Isso resulta em um *offset* em relação ao endereço original. Por exemplo, se uma tag está no endereço 126.0 e possui o tipo de dado Word, ela representa a posição 126 da carga útil. Devido ao tipo de dado Word, acrescentam-se mais 16 bits ou 2 bytes a esse endereço, abrangendo as posições 126 até 127 da carga útil. Consequentemente, o próximo valor deve começar na posição 128.

O tipo de acesso, por sua vez, determina se a operação envolve a leitura ou a escrita de um valor. No Consumer, todos os Pontos são configurados exclusivamente para leitura, enquanto no Producer, os Pontos são destinados à escrita. Na estrutura XML da Figura 27, é possível identificar o formato definido para a configuração dos Pontos.

Figura 37 – Exemplo de Configuração dos Pontos de Comunicação.

TagName	Node	Address	DataType	AccessType
TAG1	Exchange2	119.5	Bit	Read
TAG2	Exchange2	119.6	Bit	Read
TAG3	Exchange2	120.0	Bit	Read
TAG4	Exchange2	120.1	Bit	Read
SISTEMA1OK	Exchange2	120.4	Bit	Read
SISTEMA2OK	Exchange2	120.6	Bit	Read
PRESSAO_OK	Exchange2	120.7	Bit	Read
TEMP_OK	Exchange2	121.1	Bit	Read
OIL_OK	Exchange2	121.2	Bit	Read
SPEED_OK	Exchange2	121.3	Bit	Read
FLOW_OK	Exchange2	121.4	Bit	Read
CONTROLE_OK	Exchange2	121.7	Bit	Read
PRESSAO	Exchange2	126.0	Word	Read
TEMPERATURA	Exchange2	134.0	Word	Read
VELOCIDADE	Exchange2	138.0	Word	Read
PRESSAO_TURBINA1	Exchange2	140.0	Word	Read
VELOCIDADE_AB	Exchange2	160.0	Word	Read
CORRENTE_ELT	Exchange2	166.0	Word	Read
TENSAO	Exchange2	208.0	Real	Read
POTENCIA_UTIL	Exchange2	228.0	Real	Read
CALOR_DISS	Exchange2	288.0	Real	Read
MOMENTO	Exchange2	300.0	Real	Read
TORQUE	Exchange2	340.0	Real	Read
VACUO	Exchange2	356.0	Real	Read
DENSIDADE_CPO	Exchange2	368.0	Real	Read

Fonte: O Autor.

Analisando o código referente a implementação da carga útil na Figura 38 e Figura 39, é possível visualizar a relação entre o projeto configurado e o desenvolvimento do driver, especialmente no método PrepComando utilizado para preencher o cabeçalho e a carga útil.

Dentro desse método, um algoritmo é empregado para iterar através dos Items contidos na lista. Cada Item nessa lista representa um Ponto configurado. Durante a iteração, o índice do byte e do bit correspondente ao Item configurado no endereço é registrado. Em seguida, uma verificação do tipo de dado associado é realizada para preencher o buffer apropriado. Quando o tipo de dado é Word, que ocupa 2 bytes, cada byte é extraído do valor e armazenado no buffer em uma posição determinada pelo índice.

No caso dos bits, é adotada uma abordagem ligeiramente diferente. Para cada valor do bit, verifica-se se é igual a 1 ou 0, e então é inserido no vetor de bits. Esse processo é repetido até que o último bit correspondente ao byte em questão seja alcançado. Ao atingir o último bit configurado para o byte, o vetor de bits é inserido no buffer na posição indicada pelo índice do byte.

Figura 38 – Preenchimento da Carga Útil - Parte 1.

```
//Data Buffer
int headerBuffer = 32;
BitArray bitArray = new BitArray(8);
Dictionary< DataType, int[]> dynamicOffset = new Dictionary<DataType, int[]>();


foreach ( ConfigItem item in msg.Block.ListItems)
{
    int bitsCountInCurrentByte = msg.Block.ListItems.Count(items =>
    items.BlockId == item.BlockId);

    int byteIndex = item.BlockId;
    int bitIndex = item.AddressVar;

    switch (item.ConfigDataType)
    {
        case DataType.Word:

            msg.TxBuffer[headerBuffer + byteIndex] =
            Convert.To<byte>(item.ItemValue.Integer & 0xFF);

            msg.TxBuffer[headerBuffer + byteIndex + 1] =
            Convert.To<byte>((item.ItemValue.Integer >> 8) & 0xFF);

            break;

        case DataType.Bit:

            byte[] bytes = new byte[1];

            bitArray[bitIndex] = (item.ItemValue.Integer != 0) ? true : false;

            if (bitIndex == bitsCountInCurrentByte - 1)
            {
                bitArray.CopyTo(bytes, 0);
                msg.TxBuffer[headerBuffer + byteIndex] = bytes[0];
                bitArray.SetAll(false);
            }

            break;
```

Fonte: O Autor.

Figura 39 – Preenchimento da Carga Útil - Parte 2.

```
case DataType.Long:

    msg.TxBuffer[headerBuffer + byteIndex] =
    Convert.To<byte>(item.ItemValue.Integer & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 1] = Convert.To<byte>((item.ItemValue.Integer >> 8) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 2] = Convert.To<byte>((item.ItemValue.Integer >> 16) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 3] = Convert.To<byte>((item.ItemValue.Integer >> 24) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 4] = Convert.To<byte>((item.ItemValue.Integer >> 32) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 5] = Convert.To<byte>((item.ItemValue.Integer >> 40) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 6] = Convert.To<byte>((item.ItemValue.Integer >> 48) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 7] = Convert.To<byte>((item.ItemValue.Integer >> 54) & 0xFF);

    break;

default: //Treat all not specified DataTypes as 4 bytes

    msg.TxBuffer[headerBuffer + byteIndex] = Convert.To<byte>(item.ItemValue.Integer & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 1] = Convert.To<byte>((item.ItemValue.Integer >> 8) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 2] = Convert.To<byte>((item.ItemValue.Integer >> 16) & 0xFF);
    msg.TxBuffer[headerBuffer + byteIndex + 3] = Convert.To<byte>((item.ItemValue.Integer >> 24) & 0xFF);

    break;
```

Fonte: O Autor.

3.4 TESTE PRODUCER

Finalizado o desenvolvimento do Producer, o Driver passa por testes para assegurar seu funcionamento e a eficácia da comunicação com o Consumer. Esses testes envolvem a transmissão de pacotes na rede local, abrangendo todos os tipos de dados. Os valores

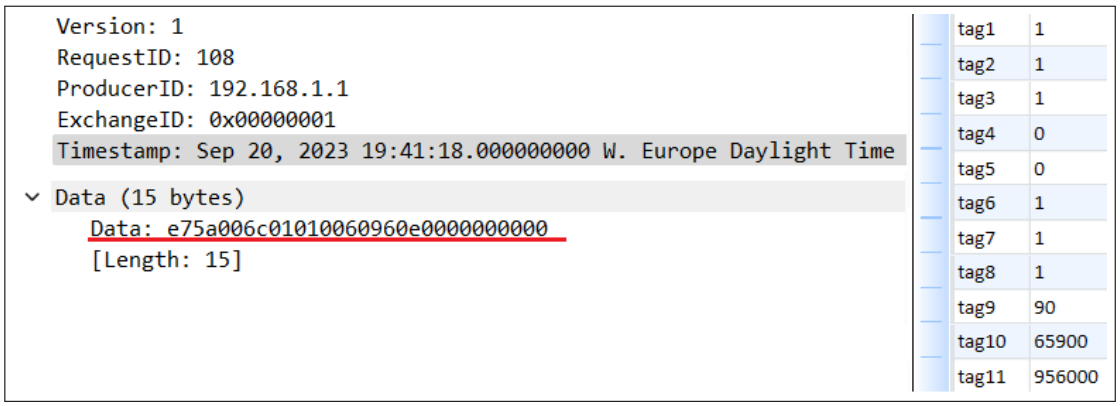
são escritos nas Tags previamente configuradas dentro do próprio SCADA, e o resultado é então verificado na carga útil do pacote por meio do uso do Wireshark, como ilustrado na Figura 40 e Figura 41.

Figura 40 – Configuração dos Pontos de Comunicação para Teste.

TagName	Node	Address	DataType	AccessType
tag11	Producer_S1	7.0	Long	Write
tag10	Producer_S1	3.0	Real	Write
tag9	Producer_S1	1.0	Word	Write
tag8	Producer_S1	0.7	Bit	Write
tag7	Producer_S1	0.6	Bit	Write
tag6	Producer_S1	0.5	Bit	Write
tag5	Producer_S1	0.4	Bit	Write
tag4	Producer_S1	0.3	Bit	Write
tag3	Producer_S1	0.2	Bit	Write
tag2	Producer_S1	0.1	Bit	Write
tag1	Producer_S1	0.0	Bit	Write

Fonte: O Autor.

Figura 41 – Resultado Obtido do Envio do Pacote.



Fonte: O Autor.

A análise dos resultados revela que o Driver opera conforme as expectativas estabelecidas. A rede identificou com precisão o pacote como EIP, e todas as informações configuradas são apresentadas com exatidão. A carga útil também está em conformidade com os requisitos previamente definidos.

A Figura 42 ilustra os resultados obtidos, representando os valores das oito primeiras Tags configuradas em formato hexadecimal, onde cada valor é composto por 1 byte. Em binário, esses valores se traduzem em 11100111, equivalendo a `0xE7` em hexadecimal no formato *little-endian*, conforme demonstrado.

Os próximos 2 bytes representam o valor da Tag9, do tipo Word, com 90 equivalente a `0X5A` em hexadecimal, enquanto o segundo byte permanece nulo, pois não foi utilizado.

O resultado da Tag10 é 65900, que, quando convertido no formato *little-endian*, torna-se `0x6C 0x01 0x01 0x00`.

Por fim, o valor 956000 equivale em formato *little-endian* `0x60 0x96 0x0E`, como demonstrado. Esses resultados destacam a precisão das conversões.

Figura 42 – Carga útil do Pacote Enviado pelo Driver e Recebido no Wiresahrk.

11100111	0	90	65900				956000							
Tag1 – Tag8 (Bit)	**Tag9 (Word)**		**Tag10 (Real)**				**Tag11 (Long)**							
0xE7	0x5A	0X00x	0x6C	0x01	0x01	0x00	0x60	0x96	0x0E	0x00	0x00	0x00	0x00	0x00

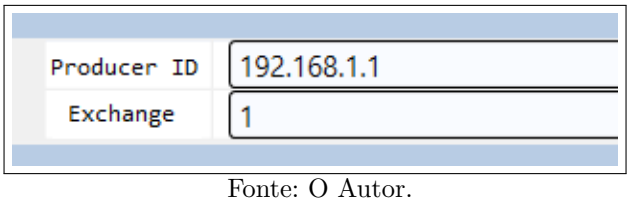
Fonte: O Autor.

3.5 DESENVOLVIMENTO DO CONSUMER

A inicialização do Consumer é semelhante a do Producer porque para a leitura ser realizada é preciso que as configurações permitam a troca de informação, ou seja, conforme estipula o protocolo EIP os campos: Producer ID, Exchange ID, precisam permitir essa comunicação.

Conforme ilustrado na figura 43, o Producer difere do Consumer apenas pelo IP e a porta que precisam ser configurados para determinar o endereço de escrita dos pacotes, sendo assim o Consumer assume a porta padrão do EIP e possui o IP do dispositivo em que o driver esta sendo executado.

Figura 43 – Configuração dos nodes.



Fonte: O Autor.

No que diz respeito ao desenvolvimento do Consumer, o método PrepComando, responsável por construir e escrever os comandos na rede é implementado apenas porque a interface IComm da DriverAPI atribuida a classe obriga a implementação do método.

Figura 44 – Método PrepComando no Consumer.

```
public override Return PrepComando (Mensagem msg)
{
    try
    {
        return Return.SUCCESS;
    }
    catch (Exception ex)
    {
        Exception.Log(ex);
        Trace("Exception:" + ex.Message);
        msg.ErrorCode = 1;

        return Return.FAILED;
    }
}
```

Fonte: O Autor.

3.5.1 Verificação da Mensagem

A implementação das operações com os nodes, Items e Blocos é semelhante ao Producer, no entanto, o Consumer implementa um método adicional denominado Verifica-Protocolo que desempenha um papel fundamental no funcionamento do Consumer. Esse método opera como uma máquina de estados, sendo responsável por receber a mensagem do Producer e determinar se a leitura é necessária. Para isso, ele deve passar por uma série de estados em sequência.

O primeiro estado, Figura 45, realiza a verificação inicial da mensagem recebida, analisando se o byte inicial corresponde ao campo PDU, semelhante ao protocolo EIP. Todos os pacotes EIP têm o valor 13 ou o equivalente hexadecimal `0xA` em sua posição inicial. Portanto, esse estado descarta mensagens que não possuem esse valor desejado, prevenindo que mensagens indesejadas sobrecarreguem a máquina de estados do Driver, eliminando esses pacotes indesejados desde o início do processo.

Figura 45 – Primeiro Estado.

```
public override void  VerificaProtocolo (Mensagem msg, byte byteRx)
{
    try
    {
        switch (msg.Estado )
        {
            case 0:
                if (byteRx == 0x0D)
                {
                    msg.StateRx++;
                    msg.CmdBuffer += byteRx;
                }
                break;
```

Fonte: O Autor.

Caso o primeiro estágio seja satisfeito, o contador aumenta uma unidade o byte é armazenado no buffer, e segue para o próximo estágio, 46, onde novamente é verificado um siples byte estático do cabeçalho EIP.

Figura 46 – Segundo Estado.

```
case 1:
    if (byteRx == 0x01)
    {
        msg.Estado ++;
        msg.CmdBuffer += byteRx;
    }
    else
    {
        RefreshBuffer(msg);
    }
    break;
```

Fonte: O Autor.

O terceiro estágio, Figura 47, verifica o Producer ID, a verificação é realizada armazenando 4 bytes, referentes a cada um dos valores do Producer ID, então, é verificado se o Producer ID do mensagem recebida tem algum node com Producer ID semelhante, caso sim, segue para o prócimo estado.

Figura 47 – Terceiro Estado.

```
case 2:  //Producer identifier
    msg.CmdBuffer += byteRx;
    if (msg.CmdBuffer.BufIndex == 8)
    {
        for (int i = 0; i < 4; i++)
        {
            protocol.byteProducerID[i] = msg.CmdBuffer[4 + i];
        }

        if (protocol.NodeDefinition.Any(node => node.ProducerID.SequenceEqual(protocol.byteProducerID)))
        {
            msg.Estado ++;
        }
        else
        {
            Trace("Invalid Producer");
            RefreshBuffer(msg);
        }
    }
    break;
```

Fonte: O Autor.

O quarto estado, Figura 48, verifica a Exchange ID da mensagem recebida comparando os bytes do pacote a partir da posição 12 até 16, ou seja, a Exchange conforme determina o protocolo, ocupa 4 bytes, a localização esperada destes bytes é verificada na carga útil, e assim como o procedimento realizado para o Producer ID, o algoritmo verifica se a Exchange está presente em algum node, caso esteja, o algoritmo segue para o próximo estado, caso contrário descarta a mensagem e atualzia o *buffer*.

Figura 48 – Quarto Estado.

```
case 3: //Exchange identifier
    msg.CmdBuffer += byteRx;
    if (msg.CmdBuffer.BufIndex == 12)
    {
        byte[] byteExchange = new byte[4];

        for (int i = 0; i < 4; i++)
        {
            byteExchange[i] = msg.CmdBuffer[8 + i];
        }

        protocol.exchangeID = BitConverter.ToInt32(byteExchange, 0);

        if (protocol.NodeDefinition.Any(node => node.ExchangeID == protocol.exchangeID))
        {
            msg.Estado ++;
        }
        else
        {
            Trace("Invalid Exchange");
            RefreshBuffer(msg);
        }
    }
    break;
```

Fonte: O Autor.

Finalmente no quinto estado, Figura 49, realiza-se a verificação da carga útil, onde o node correspondente é identificado. Em outras palavras, cada um dos campos armazenados é verificado para encontrar o node que possui os mesmos valores. Esse node é então armazenado no atributo CurrentNode para uso posterior. Em sequência, os bytes do pacote são armazenados no DataBuffer da mensagem, sendo também preservados para usos subsequentes.

Figura 49 – Quinto Estado.

```
case 5: //Data verification
    msg.CmdBuffer += byteRx;

    if (msg.Status == MsgStatus.ReceivingLastRX)
    {
        try
        {
          ConfigNode currentNode = protocol.NodeDefinition.Find(node => (node.MinorSignature == protocol.minorSignature
            && node.MajorSignature == protocol.majorSignature &&
            node.ExchangeID == protocol.exchangeID && node.ProducerID.SequenceEqual(protocol.byteProducerID)));

            protocol.CurrentNode = currentNode.NodeName;

            for (int i = 32; i < msg.CmdBuffer.BufIndex; i++)
            {
                msg.DataBuffer += msg.CmdBuffer[i];

                if (msg.CmdBuffer.BufIndex == protocol.MaxAddressSize)
                {
                    break;
                }

            }

            msg.ConcludedRx = true;

        }

        catch (Exception ex)
        {
            Exception.Log(ex);
            TraceError("Invalid Header");
            RefreshBuffer(msg);

        }

        break;
    }
    break;
```

Fonte: O Autor.

3.5.2 Leitura da Carga Útil

Realizada a verificação da mensagem e armazenado nos buffers Rx e DataBuffer, o procedimento seguinte consiste em ler os valores dos buffers e armazenar nas Tags correspondentes.

Conforme a Figura 50, o método TratarMensagem é responsável por interpretar mensagens não solicitadas. Em drivers de comunicação, é comum enviar um comando de solicitação de leitura. Quando a entidade comunicante recebe, verifica e interpreta esse comando com sucesso, um comando de resposta correspondente é enviado estabelecendo um Canal de comunicação no qual a confirmação de sucesso é aguardada por meio da recepção da mensagem de resposta, no entanto, no caso em questão, ao utilizar o protocolo UDP, as mensagens são distribuídas pela rede sem uma solicitação prévia e sem confirmação de recebimento.

O funcionamento do algoritmo consiste em iterar por todos os Items da mensagem e verificar se o Item pertence ao node que foi identificado no método VerificaProtocolo, conforme explicado no método ExtractAddress do Producer, semelhante ao Consumer, os atributos BlockId e AddressVar armazenam o byte e o bit respectivamente, esses valores são armazenados localmente pelas varáveis index.

O *switch* implementado no código verifica o tipo de dado configurado para o Item, e então, busca dentro do DataBuffer contendo a carga útil do pacote EIP, o valor para atribuir ao Item com base no seu index, ou seja, o bit para o caso DataType.Bit e o byte para os demais.

Sendo assim, conforme o endereço definido para o Item, o Item sendo uma Tag configurada como um Ponto de comunicação no SCADA e o endereço do respectivo Item composto pelo byte e pelo bit, sendo estes a localização do Item dentro da carga útil. Para o caso Word existe um *offset* de 1 byte em relação ao byte do Item, totalizando 2 bytes, dessa forma é buscado dentro do buffer a localização do endereço e são acessados dois bytes em *little-endian* que retornam o valor em decimal para ser armazenado pela Tag configurada para aquele Ponto de comunicação.

Figura 50 – Procedimento de Parse da Carga Útil.

```
public override Return TratarMensagem(Mensagem msg)
{
    try
    {
        foreach (ConfigItem item in msg.Block.ListItems)
        {
            if (item.NodeName != protocol.CurrentNode)
                continue;

            int byteIndex = item.BlockId;
            int bitIndex = item.AddressVar;

            switch (item.ConfigDataType)
            {
                case DataType.Word:

                    byte[] byteWord = new byte[] { msg.DataBuffer[byteIndex],
                    msg.DataBuffer[byteIndex + 1], 0, 0 };

                    item.TagRef.SetValue(BitConverter.ToInt32(byteWord, 0));

                    break;

                case DataType.Bit:

                    BitArray bitArray = new BitArray(new byte[] { msg.DataBuffer[byteIndex] });

                    item.TagRef.SetValue((bitArray[bitIndex] == true ? 1 : 0));

                    break;

                case DataType.Long:

                    byte[] byteLong = new byte[] { msg.DataBuffer[byteIndex],
                    msg.DataBuffer[byteIndex + 1],
                    msg.DataBuffer[byteIndex  + 2], msg.DataBuffer[byteIndex + 3],
                    msg.DataBuffer[byteIndex + 4], msg.DataBuffer[byteIndex + 5],
                    msg.DataBuffer[byteIndex + 6], msg.DataBuffer[byteIndex + 7] };

                    item.TagRef.SetValue(BitConverter.ToInt64(byteLong, 0));

                    break;

                default:

                    byte[] byteReal = new byte[] { msg.DataBuffer[byteIndex],
                    msg.DataBuffer[byteIndex + 1],
                    msg.DataBuffer[byteIndex  + 2], msg.DataBuffer[byteIndex  + 3] };

                    item.TagRef.SetValue(BitConverter.ToInt32(byteReal, 0));

                    break;
            }
        }
```

Fonte: O Autor.

O procedimento se repete para os demais tipos de dados, exceto para o bit. No caso de uma Tag armazenar um valor *boolean*, o processo envolve acessar o byte do endereço do Item do tipo bit (*boolean*). Dentro dos oito bits que compõem o byte, verifica-se, através do índice do bit, se aquela posição é 1 ou 0. O valor é então armazenado no respectivo Item e, consequentemente, na Tag configurada para aquele Ponto.

3.6 TESTE CONSUMER

Para concluir a implementação do Consumer, o próximo passo envolve a execução de uma série de testes no driver, a fim de garantir seu funcionamento adequado, tanto em ambiente local quanto, posteriormente, em produção.

Nesse estágio, é crucial configurar o Canal, o node e os Pontos de comunicação do Consumer. Além disso, devem ser criadas novas Tags para capturar os valores gerados pelo Producer, e essas Tags devem ser configuradas como os Pontos de comunicação no

SCADA, conforme ilustrado na figura 51. Isso permitirá não apenas a leitura dos valores produzidos, mas também o armazenamento desses valores recebidos de forma eficaz e segura.

Figura 51 – Lista de Pontos do Projeto no SCADA.

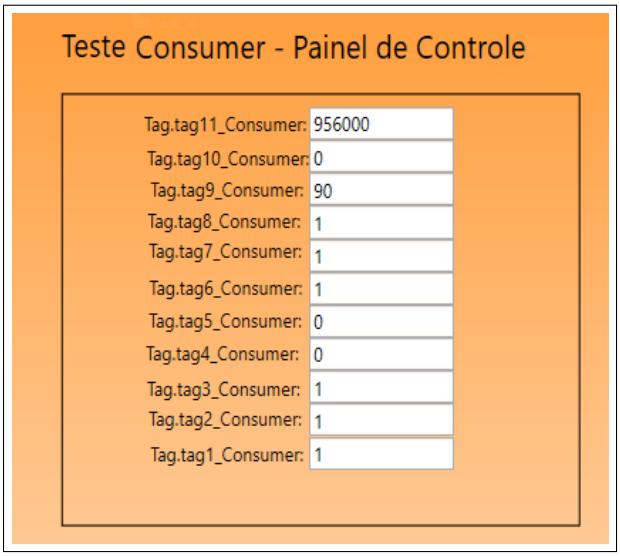
TagName	Node	Address	DataType	AccessType
tag11_Consumer	Consumer	7.0	Long	Read
tag10_Consumer	Consumer	3.0	Real	Read
tag9_Consumer	Consumer	1.0	Word	Read
tag8_Consumer	Consumer	0.7	Bit	Read
tag7_Consumer	Consumer	0.6	Bit	Read
tag6_Consumer	Consumer	0.5	Bit	Read
tag5_Consumer	Consumer	0.4	Bit	Read
tag4_Consumer	Consumer	0.3	Bit	Read
tag3_Consumer	Consumer	0.2	Bit	Read
tag2_Consumer	Consumer	0.1	Bit	Read
tag1_Consumer	Consumer	0.0	Bit	Read
tag11	Producer	7.0	Long	Write
tag10	Producer	3.0	Real	Write
tag9	Producer	1.0	Word	Write
tag8	Producer	0.7	Bit	Write
tag7	Producer	0.6	Bit	Write
tag6	Producer	0.5	Bit	Write
tag5	Producer	0.4	Bit	Write
tag4	Producer	0.3	Bit	Write
tag3	Producer	0.2	Bit	Write
tag2	Producer	0.1	Bit	Write
tag1	Producer	0.0	Bit	Write

Fonte: O Autor.

No projeto, os comandos são transmitidos pelo Producer semelhante ao valores enviados no teste do Producer apresentado na Figura 41, através da rede local e posteriormente lidos pelo Consumer. O resultado dessas operações é então apresentado em um display WPF, conforme ilustrado na Figura 52. É importante observar que, durante esse teste, os valores utilizados são arbitrários e destinam-se exclusivamente a fins de verificação. No entanto, esse processo demonstra a funcionalidade do sistema SCADA em termos de monitoramento e controle dos valores recebidos e enviados por outros dispositivos na rede.

O resultado obtido na Figura 52, destaca o desempenho dos Drivers. Cada valor apresentado reflete o êxito do algoritmo desenvolvido para o Producer, que preenche tanto o cabeçalho quanto a carga útil. De maneira análoga, o Consumer é capaz de receber o pacote de forma precisa por meio da verificação do cabeçalho, extrair os valores da carga útil e armazená-los nas Tags, posteriormente exibidas no Display de um cliente WPF no SCADA. Essa demonstração evidencia de forma notável a funcionalidade efetiva de ambos os Drivers no sistema SCADA, ilustrando a interação harmoniosa entre esses elementos e proporcionando uma solução incrivelmente útil.

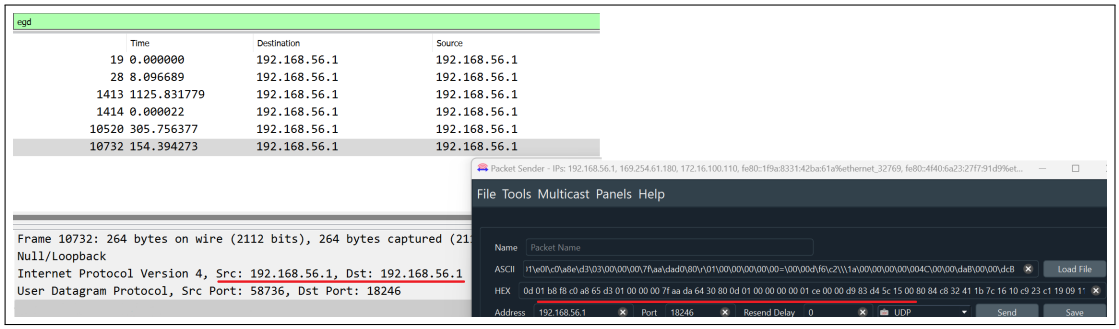
Figura 52 – Visualização de Resultado.



Fonte: O Autor.

Dando continuidade, o próximo teste foi realizado utilizando o software Packet Sender, uma ferramenta de código aberto amplamente empregada por profissionais de rede e desenvolvedores. Este software é usado para enviar e receber pacotes de rede em diversos protocolos, sendo útil para verificar a conectividade da rede, depurar problemas e executar tarefas relacionadas à rede. Foram enviados comandos com o cabeçalho EIP através da rede local utilizando o protocolo UDP. Esses comandos foram direcionados à porta específica 18246, onde o Consumer foi configurado para aguardar a chegada dos pacotes, conforme ilustra a Figura 53.

Figura 53 – Teste Utilizando Packet Sender.



Fonte: O Autor.

4 CONCLUSÃO

Este trabalho alcançou os objetivos estabelecidos, através de um exemplo fictício de implementação de driver para sistema SCADA espera-se contribuir para a base de conhecimento em sistemas SCADA e protocolos de comunicação industrial. Essas áreas, frequentemente inacessíveis devido à confidencialidade em ambientes altamente competitivos, foram exploradas de forma didática ao longo do trabalho, desde a análise do protocolo até o desenvolvimento do driver e a execução de testes em bancada.

Como o protocolo *Ethernet Industrial Protocol* é fictício, foi um desafio de determinar as especificações do mesmo, seguindo os padrões básicos dos protocolo industriais e buscando apresentar um protocolo simples e didático, sendo assim espera-se que este trabalho forneça detalhes abrangentes sobre o protocolo para o conhecimento de engenheiros e desenvolvedores, visando também facilitar o entendimento e o desenvolvimento de drivers de comunicação, suprindo a carência de informações disponíveis até o momento.

A escassez de informações sobre o processo de integração de um driver de comunicação em um sistema SCADA foi outro ponto focal deste trabalho. Com a apresentação detalhada do procedimento de desenvolvimento, busca-se não apenas preencher uma lacuna existente, mas também oferecer uma referência valiosa para profissionais que buscam *insights* práticos.

A validação bem-sucedida do funcionamento dos drivers, conforme demonstrado nos testes em bancada, confirma sua eficácia na comunicação utilizando o protocolo EIP e agrega a base de conhecimento relacionada a testes e depuração da comunicação entre dispositivos.

REFERENCES

AGUIRRE, Luis Antônio. **Enciclopédia de Automática: Controle e Automação. Vol. 2**. [*S.l.*]: Blucher, 2007.

ELORANTA, Veli-Pekka; KOSKINEN, Johannes; LEPPÄNEN, Marko; REIJONEN, Ville. **Designing Distributed Control Systems: A Pattern Language Approach**. [*S.l.*]: John Wiley & Sons, jun. 2014. P. 512. ISBN 978-1-118-69415-2.

FANUC, GE. **TCP/IP Ethernet Communications for the Series 90™ PLC User's Manual GFK-1541B**. [*S.l.*], 2002.

GARCIA, E. **Introdução a sistemas de supervisão, controle e aquisição de dados - SCADA**. [*S.l.*]: Alta Books, 2019. ISBN 9788550804644.

GE. **Mark VIe Dsitributed Control System (DCS)**. 2021. Disponível em: https://www.ge.com/gas-power/products/digital-and-controls/mark-vie-ecosystem.

GE, Industrial Systems. **Service GEI-100504**. [*S.l.*], 2008.

GERHARD P. HANCKE, Brendan Galloway e. Introduction to Industrial Control Networks. **IEEE Communications Surveys, Tutorials Volume: 15, Issue: 2, Second Quarter 2013**, 2012.

IGNITION. **SCADA System Design Made Simple**. 2024. Disponível em: https://inductiveautomation.com/ignition/architectures.

KEPWARE. **What Is KEPServerEX?** 2023. Disponível em: https://www.ptc.com/en/products/kepware/kepserverex.

KOZIEROK, Charles M. **Binary Information and Representation: Bits, Bytes, Nibbles, Octets and Characters**. 2023. Disponível em: https://ispe.org/pharmaceutical-engineering/ispeak/new-trends-requirements-digitalization-annex-1-continuous.

MALLICK, Chiradeep Basu. **What Is a Device Driver? Definition, Types, and Applications**. 2022. Disponível em: https://www.spiceworks.com/tech/devops/articles/what-is-device-driver/.

MOXA. **Moxa Secures Your OT Networks With OT/IT Integrated Security**. 2024. Disponível em: https://www.moxa.com/en/spotlight/portfolio/industrial-network-security/index.

PETER NEUMANN, Carlos E. Pereira e. Industrial Communication Protocols. *In*: SPRINGER Handbook of Automation. [*S.l.*]: Springer, 2009. P. 981–982.

POSTEL, J. **User Datagram Protocol**. 1980. Disponível em: https://www.ietf.org/rfc/rfc768.txt.

PROSOFT. **ProLinx Gateway**. [*S.l.*], 2010. Available at https://www.prosoft-technology.com/prosoft/download/731/6619/file/protocol_manual.pdf.

SSE. **GERAÇÃO DE ENERGIA**. 2020. Disponível em: https://ssebrasil.com.br/estudo/geracao-de-energia/.

STALLINGS, William. **Redes e sistemas de comunicação de dados: teoria e aplicações corporativas.** Tradução de Business data communications. 5. ed. [*S.l.*]: Elsevier, 2005.

TANENBAUM, Andrew S. **Computer networks.** 4. ed. [*S.l.*]: Prentice Hall, 2002.

VIVIANO, Amy. **What is a driver?** 2023. Disponível em: https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-.

VIVIANO, Amy. **Windows network architecture and the OSI model**. 2023. Disponível em: https://learn.microsoft.com/en-us/windows-hardware/drivers/network/windows-network-architecture-and-the-osi-model.

W. SALTER, A. Daneels e. WHAT IS SCADA? **International Conference on Accelerator and Large Experimental Physics Control Systems, 1999, Trieste, Italy**, 1999.