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

Analysis and Development of a SCADA System Communication Driver Using the EIP Protocol: A Didactic Study

Blumenau
2024

Rafael Benildo Mafra

Analysis and Development of a SCADA System Communication Driver Using the EIP Protocol: A Didactic Study

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.
Supervisor:: Prof. Adão Boava, Dr.

Blumenau
2024

Rafael Benildo Mafra

Analysis and Development of a SCADA System Communication Driver Using the EIP Protocol: A Didactic Study

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

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

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.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

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

CONTENTS

1 INTRODUCTION

The emergence of microelectronics and the subsequent diffusion of different microprocessed equipment for automation in the industry created the need for communication between them. Thus, protocols emerged, initially as simple analog protocols, typically connecting two pieces of equipment. However, they quickly evolved into digital protocols, later enabling the formation of communication networks (AGUIRRE, 2007).

Given the variety of conventions, several digital protocols emerged in the market. Initially, proprietary protocols stood out, whose parameters were not publicly disclosed due to strategic secrecy considerations, keeping users tied to a specific supplier or a restricted group of suppliers. Subsequently, faced with the difficulties imposed by these restrictions, users sought the publication of protocol specifications, giving rise to open protocols. This allowed the emergence of several suppliers competing with different levels of quality, costs, and services. Although there are still several protocols in the market, open protocols are emerging as market leaders (AGUIRRE, 2007).

Digital communication is now well established in computerized distributed control systems, both in discrete manufacturing and in process control industries. Proprietary communication systems within SCADA systems have been complemented and partially replaced by Fieldbus systems and sensor buses. The introduction of Fieldbus systems has been associated with a paradigm shift in the deployment of distributed industrial automation systems, emphasizing device autonomy and decentralized decision-making, as well as the presence of decentralized control loops (PETER NEUMANN, 2009).

Currently, Fieldbus systems are standardized and are the most important communication systems used in commercial control installations. At the same time, Ethernet has consolidated its position as the most commonly used communication technology within the office environment, resulting in lower component prices due to mass production. This has led to increased interest in adapting Ethernet for industrial applications, and several approaches have been proposed. Thus, Ethernet-based solutions are becoming increasingly common as integration technology (PETER NEUMANN, 2009).

However, for any automation or remote monitoring to be effective, it is crucial to convert the information transmitted to the field data interface devices into a protocol compatible with SCADA. Given the vast diversity of manufacturers, devices, and protocols present in the modern industrial scenario, SCADA system developers cannot support all available protocols. Therefore, there is a need for a defined architecture for implementing communication drivers in the SCADA system, allowing interaction with new protocols and ensuring the system's flexibility and expandability.

1.1 OBJECTIVES

The sections below describe the general objective and the specific objectives of this work.

1.1.1 General Objective

The proposed work aims to elucidate the concepts of industrial automation networks with an emphasis on communication drivers and SCADA systems. The practical focus of the work consists of didactically presenting the development of a communication driver, using the fictitious EIP protocol, specifically aimed at integration with a SCADA system. Finally, the results of the communication tests carried out on a bench will be presented, providing a practical evaluation of the effectiveness and performance of the fictitious communication driver implemented for didactic purposes.

1.1.2 Specific Objectives

- Present the operation of the EIP protocol, including its characteristics, data structure, and communication requirements;
- Develop the initial version of the communication driver, considering the specifications and requirements of the EIP protocol;
- Elucidate in detail the entire process of developing and integrating the driver into the SCADA system;
- Conduct preliminary tests of the driver in a controlled test environment, verifying its ability to communicate correctly using the EIP protocol;

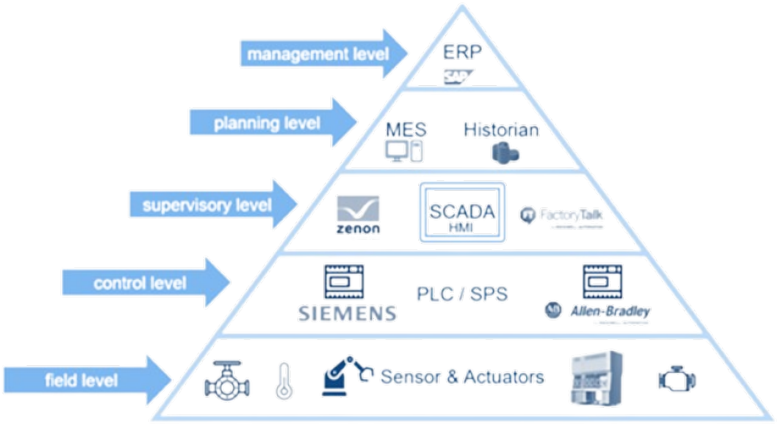
2 LITERATURE REVIEW

This chapter presents a theoretical review of the main concepts related to the development of this work.

2.1 INDUSTRIAL COMMUNICATION NETWORKS

An Industrial Communication Network is a system of interconnected equipment that plays a crucial role in monitoring and controlling physical devices in industrial environments. Industrial communication networks facilitate communication and data exchange between the various elements of the layers defined in the ISA-95 standard, the International Standard for the Integration of Automation and Production Management Systems. The standard formally defines the various layers that reflect the hierarchical structure of these systems, aiming to ensure interoperability by establishing a consistent language that serves as a basis for communication between suppliers and manufacturers, as well as offering standardized information models. Figure 1, known as the Automation Pyramid, is a widely linked visual representation to ISA-95, outlining the various hierarchical layers of industrial automation systems.

Figure 1 – Automation Pyramid.

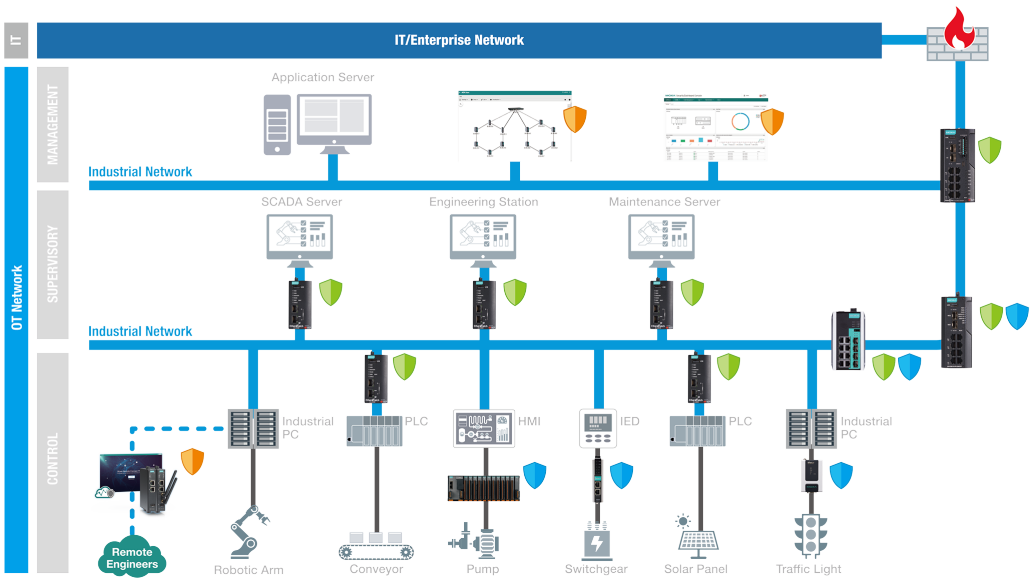


Source: Adaptado de (KOZIEROK, 2023).

Industrial networks play a fundamental role in various industrial sectors, encompassing manufacturing, electricity generation, food and beverage processing, transportation, water distribution, and chemical refining. Practically in all situations that require the supervision and control of machinery, an industrial control network is installed in some form. As exemplified in Figure 2, they generally have a more complex architecture than commercial networks. While a company's commercial network may consist of branch or

office LANs interconnected by a main network or WAN, even small industrial networks tend to have a hierarchy with three or four levels.

Figure 2 – Industrial Network Example.



Source: Adaptado de (MOXA, 2024).

Different protocols and physical media are frequently employed in each layer of the ISA-95 standard model, with devices using a variety of protocols, especially in the initial layers, which deal directly with field devices for control and monitoring. This often requires the use of gateway devices to facilitate communication between the layers. Although improvements in industrial network protocols and technologies have simplified typical hierarchies, especially in the upper layers, the network architecture is often not fully simplified to maintain correlation with the functional hierarchy of the controlled equipment (GERHARD P. HANCKE, 2012).

As industrial control networks are connected to physical equipment, the failure of a system has much more severe impacts than in commercial systems. The various effects of failure in an industrial network can include equipment damage, production loss, environmental damage, loss of reputation, and even loss of life. Although not always caused by control system failures, numerous industrial disasters, such as the Fukushima Daiichi nuclear accident in 2011, provide examples of the impact of a severe industrial failure (GERHARD P. HANCKE, 2012).

From this perspective, in the implementation of devices in industrial environments, it is imperative to meet specific requirements, such as the need for real-time communication for efficient operations, robustness and resistance against adverse conditions such as dust, humidity, and temperature variations. The implementation of redundancy is crucial to ensure operational continuity in failure situations, playing a critical role in the reliabil-

ity of industrial systems. Ensuring the compliance of devices and network infrastructure with real-time communication standards, combined with the use of protection certifications, constitutes essential practices to optimize both performance and stability in these environments.

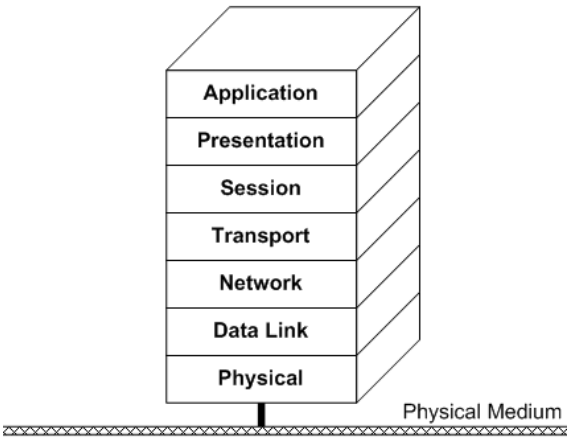
2.1.1 Communication Protocol

Successful communication between two entities requires that both "speak the same language." The content, form, and timing of the communication must adhere to mutually agreed conventions between the involved parties. These conventions are known as a protocol, which can be defined as a set of rules that control the exchange of data between two installations. In other words, the protocol establishes the essential guidelines to ensure effective and standardized communication between the communicating entities (STALLINGS, 2005).

Network protocols divide broader processes into discrete and strictly defined functions and tasks at each network level. In the standard model, widely known as the OSI model, one or more network protocols regulate activities at each layer in the telecommunications exchange. The lower layers deal with data transport, while the upper layers in the OSI model deal with software and applications.

The OSI Model, illustrated in Figure 3, is based on a proposal developed by the ISO as an initial framework for the international standardization of multi-layer protocols. Its review took place in 1995. It is called the ISO OSI Reference Model (TANENBAUM, 2002).

Figure 3 – Reference Model OSI.



Source: (VIVIANO, 2023b).

- *Physical Layer*, where the transmission of physical signals between devices

occurs, determining how bits are represented and transferred.

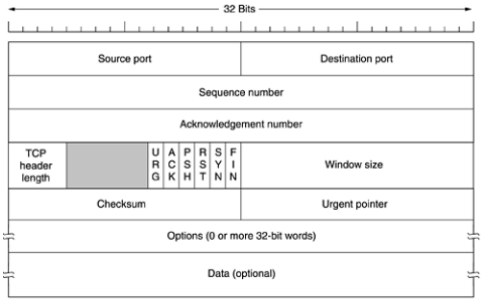
- *Data Link Layer* enables direct connectivity, establishing, managing connections, and correcting transmission errors;
- *Network Layer* directs packets between different networks, managing routing;
- *Transport Layer*, where data transfer is controlled, ensuring reliability with TCP or simplicity with UDP;
- *Session Layer* manages the establishment and termination of sessions, providing authentication;
- *Presentation Layer* formats data to be understood, including encryption and compression;
- *Application Layer* interacts directly with the user, providing services such as HTTP and FTP for file sharing and application access, completing the essential structure for efficient communication in networks;

Despite being widely recognized and serving as a reference in the study and application of communication protocols, the OSI model is often used only as a theoretical model far from the actual implementation of communication protocols and faces considerable criticism.

Today, it is understood that the launch of the OSI standard protocols was rushed. At the time these protocols were introduced, the competing TCP/IP protocols were already being widely adopted by researchers in universities. Even before the start of billion-dollar investments, the academic market was already robust, leading many manufacturers to offer products based on TCP/IP, albeit with some caution. When the model was introduced, companies were reluctant to invest in a second protocol stack unless it became a market-imposed necessity. With all companies waiting for someone to take the first step, the OSI model remained largely theoretical, without gaining widespread implementation (TANENBAUM, 2002).

The messages transmitted between the communicating entities are called packets. A packet is composed of control information and payload, with the control information providing data for the delivery and reading of the payload, such as source and destination network addresses, size of the contained information, error detection codes, or sequencing information such as checksum, CRC, and parity. Typically, control information is found in the headers and at the end of the packets after the payload. Taking the TCP protocol packet as an example in Figure 4, the header fields and payload follow the established standard sequence.

Figure 4 – TCP Data Packet Structure.

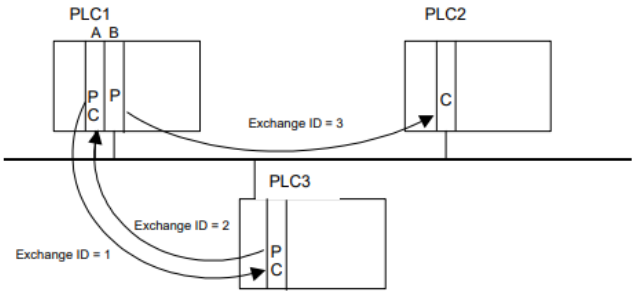


Source: (TANENBAUM, 2002).

2.1.1.1 Ethernet Indsutrial Protocol

For didactic purposes, it was defined that the protocol should be based on standard internet protocols to share information between controllers, exchanging data samples, as illustrated in Figure 5. Each packet contains a data sample or snapshot of a controller's memory, as is common in industrial protocols. These samples are sent periodically to one or more peer controllers that store the data for use in application tasks; the controllers can both read and write on the network. Each data sample is uniquely identified to relate it to a definition that describes the data it contains, called an Exchange. For this fictitious protocol, the controller that generates the sample is called the Producer, and the controller that receives it is called the Consumer. Each controller will send or receive data samples only for the Exchanges for which it has been configured. In this way, a network can be configured so that multiple controllers share information to perform control or monitoring functions.

Figure 5 – Example of Memory Sharing between PLCs.



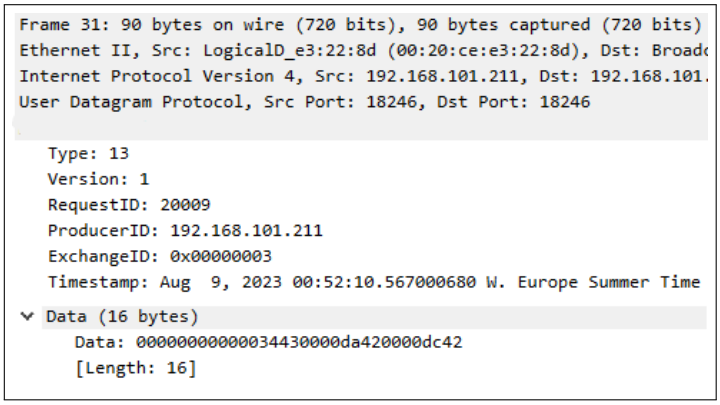
Source: (FANUC, 2002).

To ensure the reliability of the information transmitted at the application layer level, it was defined that EIP organizes the data to be transmitted into Exchanges, and multiple Exchanges are combined to form *Pages*. Each *Page* has a unique identifier, which

is a combination of the Producer ID and the Exchange ID. This identifier allows the receiver to recognize the data and know where to store it. With EIP, a producer can send information to an unlimited number of consumers simultaneously at a fixed periodic rate.

Its structure, organized into Producer ID and Exchange ID, ensures the integrity of the transmitted data and synchronization between producers and consumers. Figure 6 displays all the fields of the EIP protocol presented in this chapter.

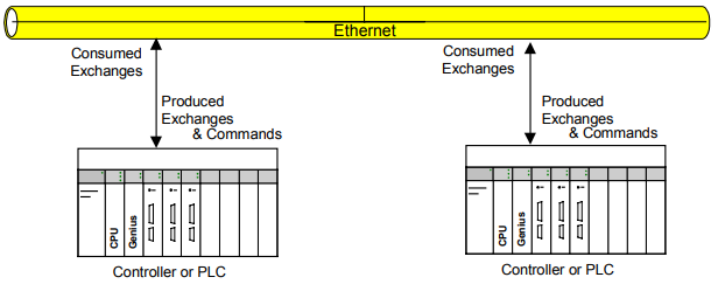
Figure 6 – Protocol example in Wireshark.



Source: O Autor.

In the communication architecture presented in Figure 7, extracted from the Cimplicity system manual and adapted to display the fictitious EIP protocol, the Cimplicity system is SCADA and HMI software from GE. Figure 7 highlights the interaction between the controllers as well as the defined operation for EIP, drawing parallels with the operation of a real protocol. This architecture uses the Ethernet network as a communication medium, just like EIP, to enable efficient information exchange between control devices.

Figure 7 – Example of communication using Ethernet.

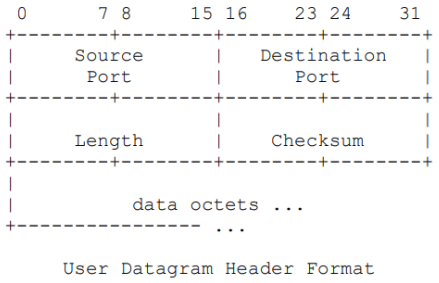


Source: Adaptado de (GE, I. S., 2008).

2.1.1.2 User Datagram Protocol

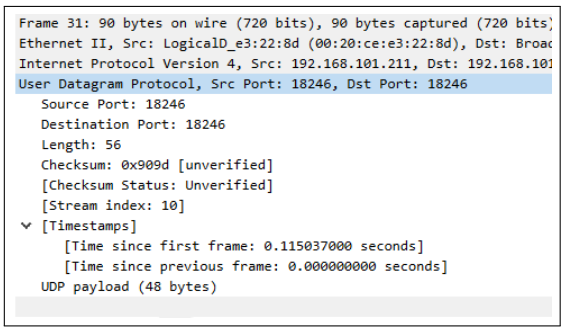
The UDP protocol is defined to provide a datagram mode for communication between computers in an interconnected computer network. This protocol assumes that the internet protocol IP is used as the underlying protocol. This protocol offers a procedure for application programs to send messages to other programs with a minimum of protocol mechanisms. The protocol is transaction-oriented, and delivery and protection against duplicates are not guaranteed. Applications that require reliable and ordered delivery of data streams should use TCP (POSTEL, 1980).

Figure 8 – UDP header in the original documentation.



Source: (POSTEL, 1980).

Figure 9 – UDP protocol in Wireshark.



Source: O Autor.

One of the most important characteristics of UDP is its low overhead compared to TCP. As illustrated in Figures 8 and 9, the UDP protocol has only four fields, whereas the TCP packet, as shown in Figure 4, has ten fields. Unlike TCP, which performs flow control, congestion control, and acknowledgment of receipt, UDP does not have these features. This makes UDP more efficient in terms of latency and bandwidth usage, making it ideal for applications where speed is a priority and occasional data loss is not critical.

The EIP protocol employs UDP at the transport layer, as presented in the previous chapter. This protocol operates under the premise of periodically sharing the memory of

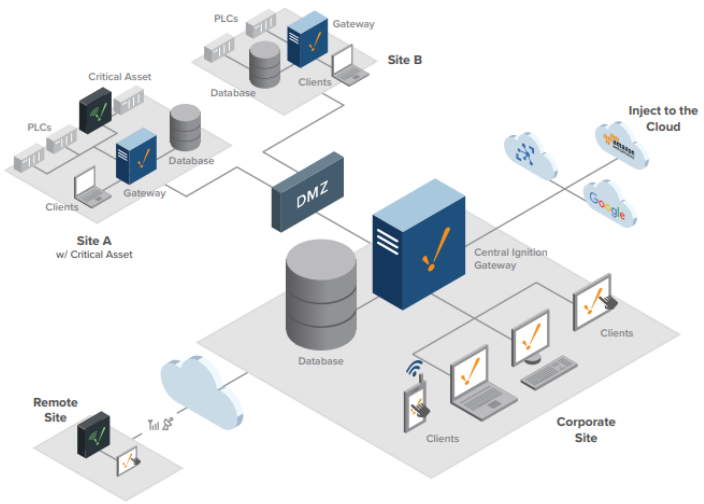
controllers, usually at intervals of less than a few seconds. The choice of UDP is justified by the constant frequency of packet transmission on the network, eliminating the need for acknowledgment of receipt and ensuring sufficient speed in the process.

2.2 SCADA SYSTEM

SCADA stands for Supervisory Control and Data Acquisition. As the name suggests, it is not a complete control system but rather focuses on the supervisory level. Thus, it is a software layer positioned over hardware to which it is interfaced, usually through PLCs or other commercial hardware modules (W. SALTER, 1999).

A SCADA system, as exemplified in Figure 10, collects detailed information, such as leak detection in a pipeline, transfers this data to a central location, issues alerts about the leak to the base station, conducts necessary analyses and controls, such as determining the criticality of the leak, and presents the information in a logical and organized manner.

Figure 10 – Example of External System Architecture SCADA.



Source: (IGNITION, 2024).

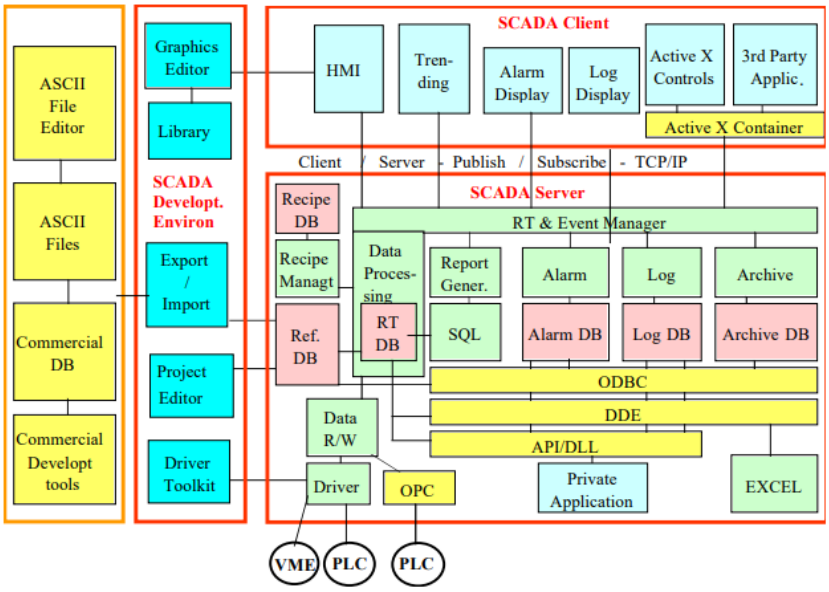
The concept of SCADA was developed to be a universal means of remote access to a variety of local control modules, which can be from different manufacturers and allow access through standard automation protocols. In practice, large SCADA systems have evolved to become very similar to distributed control systems in function, using multiple means of interfacing with the plant.

The focus of a SCADA system is data acquisition and the presentation of a centralized HMI, although it also allows the sending of high-level commands to control hardware, such as instructions to start a motor or change a setpoint. SCADA systems are designed to monitor geographically dispersed control hardware, being especially suitable for industries

such as utility distribution, where plant areas may be located over thousands of square kilometers.

The remote location of devices imposes constraints on the system and is a central aspect of how SCADA systems are designed. Data communication over long distances often involves the use of third-party media, such as telephone lines or cellular telephony. These media are often unreliable or have bandwidth limitations. Therefore, SCADA systems tend to be event-driven rather than process-driven, focusing only on reporting changes in the state of the monitored system, rather than continuously sending a constant stream of process variables. This allows a reduction in the number of communications sent and reduces bandwidth requirements. SCADA software also needs to take into account unreliable communication media and be able to implement features such as logging the last known value of all variables in the system and determining the quality of the data (GERHARD P. HANCKE, 2012).

Figure 11 – Example of internal SCADA system architecture.



Source: (W. SALTER, 1999).

As detailed in Figure 11, SCADA systems are multitasking and are based on a RTDB located on one or more servers. These servers are responsible for data acquisition and manipulation, such as querying controllers, checking alarms, calculations, logging, and archiving, within a set of parameters, usually those to which they are connected.

Communication between server-client and between servers is generally based on a *publish* and *subscribe* model and is event-driven, using the TCP/IP protocol. In other words, a client application subscribes to a parameter managed by a specific server application, and only changes in that parameter are then communicated to the client application.

In accessing devices, data servers perform polling on controllers at a user-defined polling rate. This rate can vary for different parameters. Controllers send the requested parameters to the data servers. The timestamp of process parameters is usually performed in the controllers, and this timestamp is transferred to the data server. If the controller and the communication protocol used support unsolicited data transfer, SCADA also supports this feature.

They also support a variety of communication protocols to ensure efficient integration with widely adopted PLCs and fieldbuses, such as Modbus, PROFIBUS, EtherNet/IP, DeviceNet, and more recently MQTT and OPC UA, depending on the specific needs of the application and the devices used. They generally also provide a toolkit available to create drivers for hardware not supported by SCADA.

Most allow actions to be triggered automatically by events. A scripting language provided by SCADA products enables the definition of these actions. In general terms, it is possible to load a specific screen, send an email, execute a user-defined application or script, and write to the RTDB.

Currently, companies are in a scenario where data plays a crucial role. Allowing control and supervision of data from a wide range of industrial devices elevates the industry to a higher level. However, due to the complexity of SCADA systems, they are expensive, making it difficult for many companies to bear the costs. As an alternative, options with limited functionalities are often offered to meet the needs of smaller companies.

2.3 DISTRIBUTED CONTROL SYSTEM

According to widely recognized nomenclature, DCS, or Distributed Control System, consists of an arrangement of control equipment distributed in the industrial environment. The system, usually equipped with custom processors as controllers, employs redundant networks, which can be proprietary or follow standardized protocols. The controllers receive data from input modules and transmit it to output modules. The input modules capture information from process instruments in the field, while the output modules communicate with field actuators (GARCIA, 2019).

Electrical buses and digital protocols are used to connect the distributed controllers to a control center, which in turn is linked to an HMI, allowing real-time visualization of process data and information. Local operation stations, distributed throughout the plant, are common to take over the functions of the control center in case of failure, ensuring an additional level of redundancy and reliability. DCS components can connect directly to physical equipment, such as switches, pumps, motors, and valves, or operate through an intermediary system, such as SCADA (GARCIA, 2019).

DCS first emerged in large industries with critical and high-value processes, being attractive because manufacturers would provide both the local control level and the central supervision equipment as an integrated package, thus reducing the risk of project

integration, as in the case of the turbine and the MarkVIe, represented in Figure 12, provided by GE. Currently, SCADA and DCS functionalities are very similar, but DCS tends to be used in large continuous process plants, where high reliability and safety are important, and where the control center is not geographically remote (ELORANTA et al., 2014).

DCS are more suitable for continuous processes involving multiple analog signals and complex PID control loops, commonly found in power plants and refineries. The key differentiator of a DCS is its reliability due to the distribution of control processing across nodes in the system. This mitigates a single processor failure. If a processor fails, it will only affect a section of the plant process, unlike a central computer failure, which would affect the entire process. This distribution of local computing power to the field I/O connection racks also ensures fast controller processing times, removing potential network and central processing delays.

Often, DCS only have integrated HMIs, and ready-made SCADA system solutions in DCS are not as advanced, resulting in the need for integration with other systems. In many cases, the DCS supervision system does not replace a SCADA; SCADA systems are specialized to offer centralized supervision and monitoring. As presented in chapter 2.2, they provide specialized tools for process visualization, trend analysis, and data history.

Figure 12 – DCS MarkVIe.

Source: Adaptado de (GE, 2021).

The integration of a SCADA into a DCS provides synergy between these systems. By unifying SCADA, responsible for monitoring and supervision, with DCS, focused on detailed process control, a comprehensive and detailed view of the industrial process is obtained. This is possible due to the data-sharing capability between the systems, allowing for deeper and real-time analysis of the plant's operational status. The integration results in significant improvements in operational efficiency, with more informed and faster decision-making based on detailed process information. Additionally, flexibility and scalability are

enhanced, enabling the smooth incorporation of new devices and systems as the plant's needs evolve.

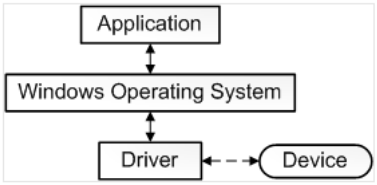
2.4 COMMUNICATION DRIVER

A driver, in general, can be defined as a software interface that allows communication between distinct hardware entities. In an operating system context, as shown in Figure 13, a device driver is a software component that acts as a translator between the operating system and the specific hardware device. The device driver understands the specific commands and protocols of the device, converting high-level requests from the operating system into low-level signals that the device can understand and execute (VIVIANO, 2023a).

Device drivers operate within the kernel layer of the operating system, functioning in a highly privileged environment due to the need for low-level access to hardware operations. Their main function is to facilitate communication between the computer's OS and the specific hardware for which they were developed. This communication occurs through a computer bus that establishes a connection between the device and the computer.

To access and execute the device's instructions, drivers rely on instructions provided by the operating system. After completing their tasks, they transmit the output or messages from the hardware device back to the operating system. Devices such as modems, routers, speakers, keyboards, and printers depend on device drivers to function correctly. These drivers play a crucial role in enabling smooth interaction between the operating system and the hardware (MALLICK, 2022).

Figure 13 – Windows Driver Example.



Source: (VIVIANO, 2023a).

Communication drivers are different from device drivers. Communication drivers are designed for the transmission of data from hardware devices to the running software and for the transmission of commands from the running software to the devices, according to the device's communication protocol.

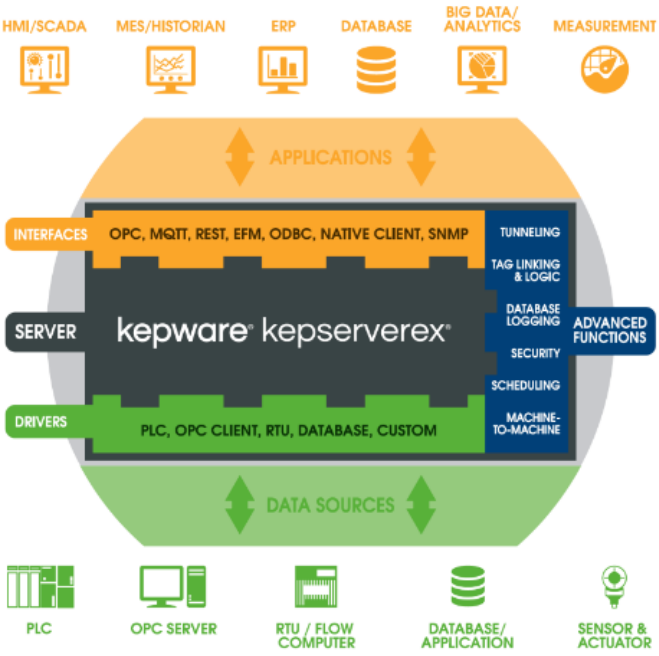
Communication drivers are often used in the context of industrial automation, SCADA systems, or other scenarios where distinct devices need to communicate. However, device drivers are essential for the proper functioning of hardware components, such as

printers, graphics cards, and network adapters, translating high-level commands from the operating system into low-level instructions that the hardware can execute.

Using the KEPserverEX software from Kepware, a subsidiary of the American company PTC, a pioneer in CAD in 1988, as an example. Kepware focuses on developing communication drivers for automation controllers and field devices (KEPWARE, 2023).

KEPServerEX is the industry's leading connectivity platform, with over 150 communication protocols available, providing a single source of industrial automation data for all your applications. Users can connect, manage, monitor, and control various automation devices and software applications through an intuitive user interface (KEPWARE, 2023).

Figure 14 – Kepware KEPServerEX.



Source: (KEPWARE, 2023).

As shown in Figure 14, the most common application of KEPServerEX is to collect data from a wide variety of devices using the vast array of available drivers, and then send the obtained data using the most commonly used open protocols, such as MQTT and OPC. This centralizes data collection and standardizes the transmission of this data, providing an accessible protocol and allowing communication with other systems such as SCADA, HMI, ERP, and databases.

In the process of developing and testing communication drivers, KEPServerEX is frequently used, as in the case of this work. Containing a vast variety of drivers, all certified for operation in applications where a high degree of security is required, the software provides flexibility and credibility in its use.

3 DEVELOPMENT

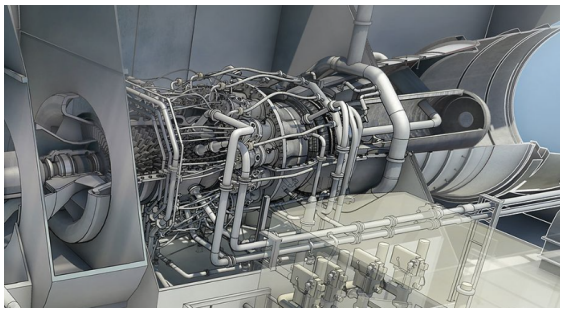
This chapter presents the series of steps that culminated in the operational development of the driver.

3.1 CONTEXT

Assuming a case where support for the EIP protocol in the SCADA system does not exist and the development of a custom driver has been commissioned, the process involves the use of a driver development library, called *Toolkit*, aligned with the architectural standard established for integration with the SCADA system.

Assuming that the initiative arose from the need to establish communication between the SCADA and a power-generating turbine. The turbine, illustrated in Figure **??**, is controlled by a DCS.

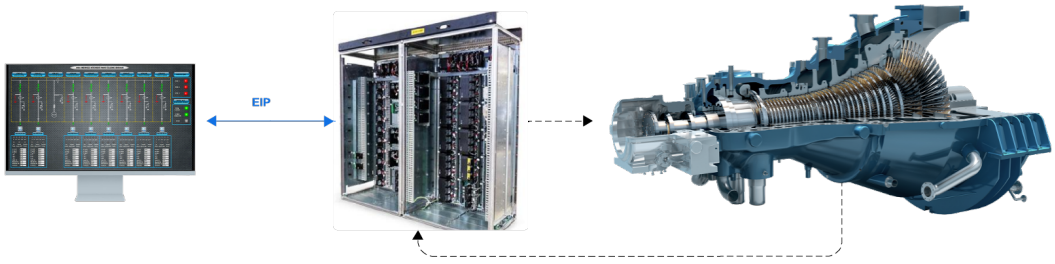
Figure 15 – Internal representation of the turbine.



Source: (GE, 2021).

As illustrated in Figure 16, the SCADA system needs to effectively communicate with the controller, read turbine parameters and display them on a screen for supervision, as well as be able to write commands to the turbine to alter these parameters.

Figure 16 – Overview of the Fictitious Application.



Source: O Autor.

3.2 HEADER IMPLEMENTATION

3.2.1 Protocol Analysis

The process of understanding the operation of a protocol is based on the available documentation about it, thus initiating the process of comprehension and subsequent construction of the header that will enable the identification of the protocol on the network. It is important to note that, due to its wide application in various devices, it is common to encounter specific modifications in the protocol that add or eliminate functionalities to adapt the communication to the desired devices. Therefore, the available documentation does not always provide all the necessary information for the development of the communication driver for the desired application.

Taking as an example the gateways provided by Prosoft Technology, a company globally known for developing industrial communication solutions for control and automation. There are various types of gateways specialized in translating one protocol to other more widely used protocols, and vice versa. For instance, some gateways convert protocols like Modbus to others, while others provide communication interfaces for controllers such as ControlLogix and CompactLogix.

It is important to highlight the fact that it is common to find specific adaptations in the protocol to optimize communication with different equipment, as discussed in the next paragraph. Gateways play a crucial role in the integration of heterogeneous systems, allowing devices and systems operating on different protocols to communicate efficiently. By analyzing the equipment documentation, it is verified that there are specific commands developed by the company. An example of this is the read request command, as illustrated in Figure 17.

It is verified that the packet is encapsulated in UDP. The UDP fields, such as destination, source, and length, are in accordance with the UDP protocol documentation. By analyzing the packet in the read request, the presence of the expected fields is observed. However, additional specific fields for the gateway's read request, such as the Offset field that determines the specific address for reading, are also identified.

Figure 17 – Reading 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 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

Source: Adaptado de (PROSOFT, 2010).

Another example is the *Summary Request* command in Figure 18, which requests information about all Exchanges configured on the corresponding node and receives the *Summary Response* command in Figure 19 as a response, containing the list of information related to each Exchange present on the node.

It is important to emphasize the reason for the implementation of the *Summary Request* and *Response* commands by the engineers at Prosoft. In the Request command, the packet contains few fields, differing only in the PDU Type field from the standard. This element allows the receiving entity to identify it as a *Summary Request*.

The response command, on the other hand, is more complex, with its PDU Type equal to 8, representing the standard header. However, the payload contains information about all the Exchanges configured on this node. Thus, the packet has a dynamic size, depending on the number of Exchanges. It is common to inform the receiver of the payload size that the packet contains to enable data interpretation, and in this case, the Total Cells field plays this role, informing the number of Exchanges. Therefore, the *Summary Response* contains a list of the Exchanges configured on the node with their respective information such as Producer ID and State.

Figure 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

Source: Adaptado de (PROSOFT, 2010).

Figure 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

Source: Adaptado de (PROSOFT, 2010).

These cases demonstrate that protocols, especially proprietary ones, can be modified and often are, in order to adapt to specific applications. This gives engineers and developers the freedom to implement the most suitable solutions, going beyond the protocol specifications. For example, even operating on UDP at the transport layer, it is possible to make use of the upper layers to optimize the reliability of communication in this scenario. The packet is composed of the header and payload. At this stage of development, only the data related to the header were considered.

In Figure 20, a sample command is presented. The header is highlighted in blue, while the rest of the command, the payload highlighted in yellow, represents the value of the variables to be transmitted. Each position in the command is represented by a hexadecimal value equivalent to 1 byte. In this example, the command is small; it is common to have a limit for the command size, assuming that the maximum number of data that can be transmitted in an EIP command is 1000 bytes. By adding the header bytes, the maximum size of the EIP packet is 1024 bytes.

The size limit is relevant to ensure efficient transmission and avoid congestion or data loss issues. It is necessary to consider this restriction when designing the communication to ensure adequate performance and the integrity of the transmitted data.

Figure 20 – Command Sample.

```
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
```

Source: O Autor.

In the EIP protocol, all hexadecimal values are in *little-endian* format, which means that the least significant bytes are stored first, followed by the most significant bytes. This implies that when representing a multi-byte hexadecimal value, the byte with the lowest value will be stored at the lowest memory address.

To demonstrate the header construction process, an example is used with the sample command from Figure 20. In the case of the Producer ID, it has a representation in *dotted decimal* format, similar to an IP address. However, it can also be represented in standard decimal format. In this case, with the Producer ID equal to 172.23.21.176, the decimal representation is 2954172332, which, when converted to hexadecimal in *little-endian* format, results in `0xAC 0x17 0x15 0xB0`.

The *timestamp* is represented in a human-readable format. Converting it to the *UNIX timestamp* format, the value obtained is 1675437155 in decimal. Then, converting this value to hexadecimal in *little-endian* format, the sequence obtained is: `0x73 0x32 0xDD 0x63 0x74 0x16 0x8E 0x0F`.

In Table 2, the dynamic fields are listed, that is, those that change according to the command sent.

Fields	**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`

Table 1 – Value of Dynamic Command Fields.

Based on the obtained information, it is possible to determine the location of the fields in the command header. Figure 21 illustrates the position of each field and its corresponding value in the sample command, demonstrating that the header is consistently composed of 20 bytes. The payload, for this specific command, also contains 20 bytes, totaling 40 bytes.

According to the standards invented for the EIP protocol, the following pattern is determined: the PDU field, which in turn is a data unit that represents the information transmitted through the protocol. In other words, the receiver of the commands identifies the protocol through the PDU. As in this case, it is common to find the PDU in the first position of the packet, so the receiver can discard the packet without needing to read the other fields. It can be found under other names, such as the Data Type field in the gateways provided by Prosoft. It is determined that all commands contain the value `0xA` for the PDU.

Figure 21 – Command Identified.

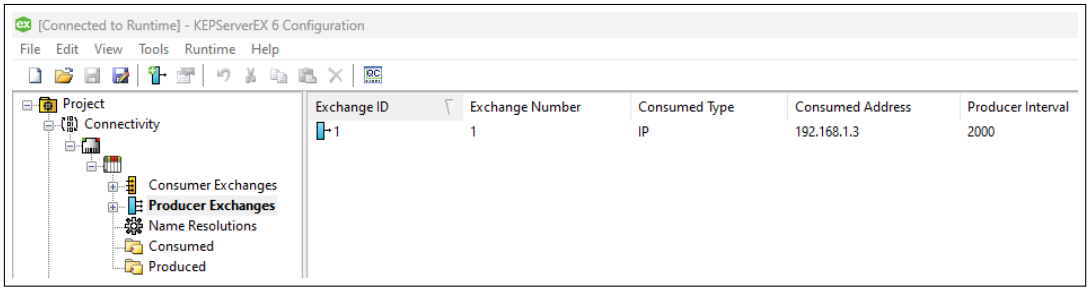
1	2	4		8				12				20							
PDU Type	**Version**	**Message Number**		**Producer ID**				**Exchange ID**				**Timestamp**							
0xA	0x01	0x91	0x36	0xAC	0x17	0x15	0xB0	0x01	0x00	0x00	0x00	0x73	0x32	0xDD	0x63	0x74	0x16	0x8E	0x0F
												0X07 0x00 0x00 0x00 0x12 0xDE 0xDC 0x63 0x72 0x32 0xDD 0x63 0x69 0x57 0x75 0x5E 0x6C 0x60 0x77							
																			40

Source: O Autor.

3.2.2 Heading Composition

Given that there is no technical documentation available regarding the specifications of the EIP protocol, to ensure the correct composition of the header, KEPServerEX was used. It is an industrial communication server that supports a wide variety of protocols used in the industry and is certified and suitable for this specific purpose.

Figure 22 – Example of Protocol Configuration in KEPServerEX.



Source: O Autor.

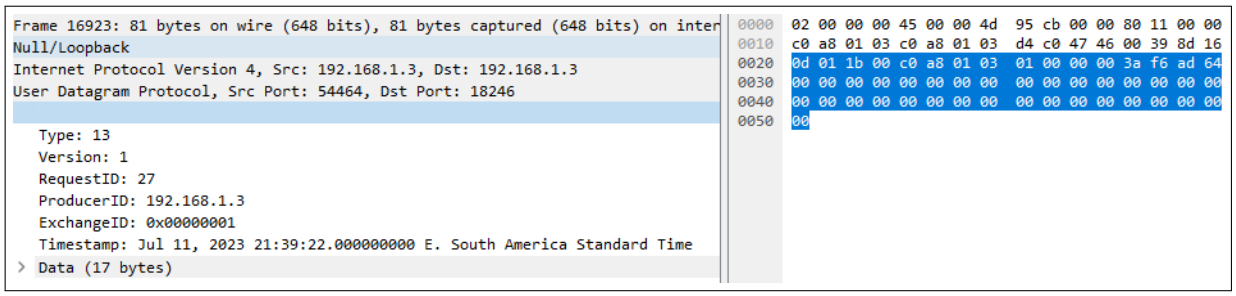
As shown in Figure 22, the protocol configuration was performed in KEPServerEX, configuring the Producer, Consumer, Exchange number, IP Addresses, and sharing interval.

Once this was done, the KEPServerEX server facilitated communication between the locally executed Producer and Consumer. To verify the content of the packets, Wireshark was used to capture the commands on the network and identify the flow of EIP commands in UDP. Wireshark is a network protocol analysis and packet capture tool. It allows examining network traffic in real-time or analyzing previously recorded capture files.

As highlighted in blue in Figure 23, the EIP packet maintains an identical structure to that found in the provided commands. The received data is consistent with the information sent by the Producer configured in KEPServerEX. It is worth noting the importance of consistency in the structure of the received packet, ensuring that the command is identified as EIP and all fields are present with the expected value, ensuring efficient communication. It is important to note that the payload, represented in Figure 23 as the Data field containing 17 bytes, will be analyzed in more detail in Chapter 3.3.

Packet capture in Wireshark facilitates the understanding of the header structure, as it is displayed in detail from IPV4 and UDP encapsulation to EIP. This allows identifying the involved ports and ensuring the meaning of each byte in the packet. The Type and Version fields are constants in Wireshark, represented by the PDU Type and PDU Version Number fields of the protocol, while the Request ID resembles the Message Number of the provided data.

Figure 23 – EIP packet in Wireshark Sent by KEPServerEX.

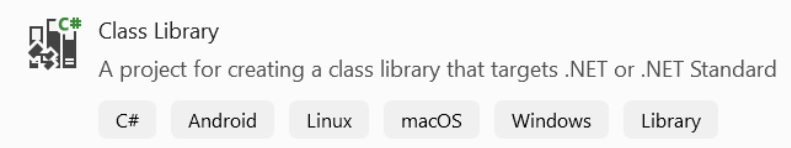


Source: O Autor.

3.2.3 Environment Preparation and Start of Driver Development

After obtaining all the necessary information for the header implementation, the driver development project begins using the integrated development environment Visual Studio, the integrated development environment provided by Microsoft. The driver will be compiled as a DLL and subsequently called by the SCADA DriverAPI. The entire project development is carried out in the .NET Framework, ensuring compatibility and integration with SCADA. As shown in Figure 24, the project is created as a Class Library on top of the .NET Framework, just like the SCADA, which is entirely developed in the .NET Framework, having access to extensive documentation, an active community, and a modern ecosystem supported by Microsoft.

Figure 24 – Creating the Driver Project in Visual Studio.



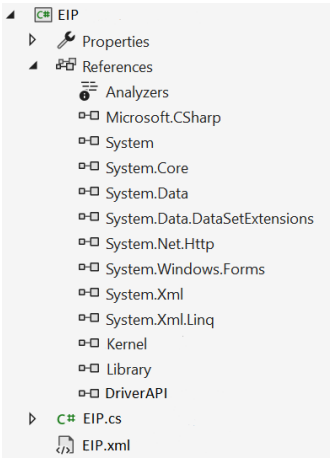
Source: O Autor.

In the scope of this project, direct references are made to SCADA to enable integration with engineering projects in SCADA. The SCADA is equipped with an API, DriverAPI, which is called by the Devices module, responsible for configuring Channels, nodes, and communication Points with external devices.

The DriverAPI assumes the responsibility of abstracting the complexities of the transport and network layers during the driver development process. It provides a consistent and simplified interface so that the driver can effectively communicate with these underlying layers. This allows the developer to focus their efforts on the driver's logic and integration with SCADA, without worrying about the specific details of transport and network communication implementation.

The communication configuration between the Driver and the API is possible using direct references to SCADA in the driver project, as shown in Figure 25, which presents the direct references to the DLLs: Kernel, Library, and DriverAPI of SCADA, thus providing a variety of readily available resources and functionalities. These references, as shown in Figure 25, include classes, methods, and properties specifically designed to work harmoniously with SCADA, the driver, and the API.

Figure 25 – Example of references to SCADA in the Driver project.



Source: O Autor.

Figure 26 provides a simplified view of an example application architecture, highlighting the informal model of the relationship between the DLLs of SCADA, responsible for communication with the DCS. The protocol configuration XML files are accessed by

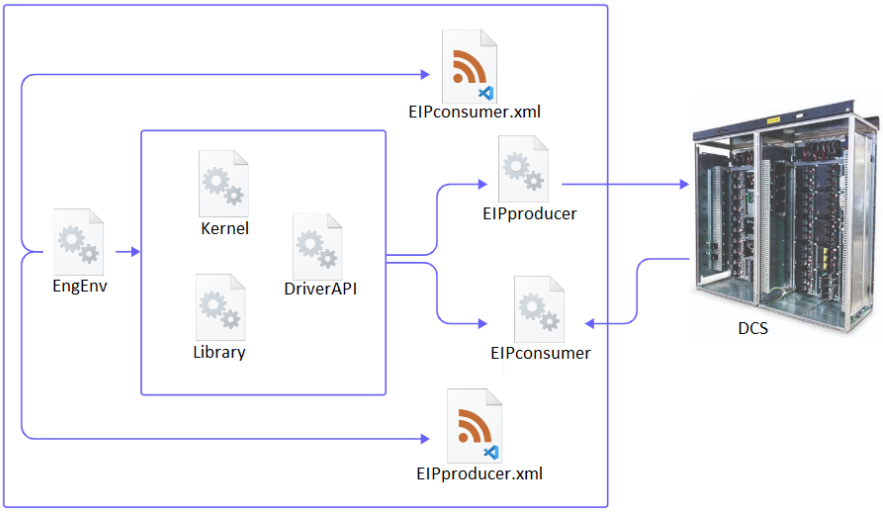
the EngEnv DLL, which presents the configured information in the graphical interface, allowing the user to customize specific communication characteristics, such as IP address, port, and particular fields, as in the case of EIP, Producer ID, Exchange.

The EngEnv DLL would play a crucial role in displaying and controlling the software engineering environment. It would be called by the Kernel during the software startup. Once the engineering environment is loaded, the protocol XML files are accessed, enabling the user to perform configurations. On the other hand, the DriverAPI DLL, as described, would assume the responsibility of managing communication with the drivers. It accesses the driver DLLs and manages the communication, operating in isolation.

The Library would be a library containing useful tools for development in the Framework, providing valuable resources to facilitate the creation process. Finally, the Kernel serves as an interface to access the Tags stored in the system, providing an abstraction layer for interacting with the SCADA system data.

This architecture seems to be designed to offer a modular and flexible approach, allowing the customization of communication settings and providing useful tools for efficient development in SCADA.

Figure 26 – SCADA Architecture Example.



Source: O Autor.

An XML file would be used to provide standardization in sharing driver configuration information that will be interpreted by SCADA, containing important details such as the protocol used in the transport layer, which can be TCP, UDP, or another specific to the project. This information allows the SCADA system to properly configure the driver to communicate with the appropriate layers.

The XML file would also contain the driver's name, providing a unique identification, and it also encompasses a set of specific driver information configured by the user in the SCADA graphical interface. The information can vary depending on the driver's

needs and the specific functionalities it offers. These data include parameters such as IP addresses, communication ports, authentication settings, among other elements relevant to the driver's operation.

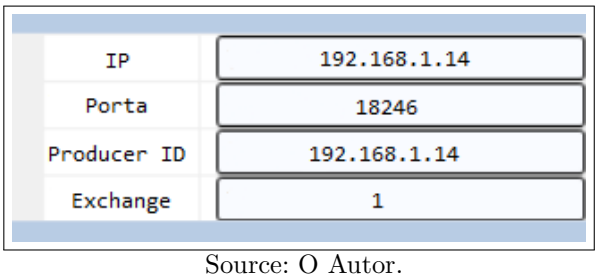
As shown in the XML in Figure 27, the driver is configured as UDP with a default port of 18246, a port used by the EIP protocol. Within the structure defined by UDP in Figure 28, there are specific fields and default values, as well as the display window size for this information. The SCADA interprets the XML and generates in the user interface the display and editing window for the driver's node configurations, as shown in Figure 28.

Figure 27 – XML for Driver Configuration.

```
<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>
```

Source: O Autor.

Figure 28 – Example of Node's Graphical Configuration Interface.



Source: O Autor.

Initializing the development of the driver's logic, as shown in Figure 29, the IComm interface and the DriverDef class are implemented, both belonging to DriverAPI. The interface defines the methods that must be used, providing a logical sequence of the driver's operation, from its initialization to the sending and receiving of commands.

The DriverDef class is responsible for overriding the methods defined in the IComm interface, allowing access to the objects and properties of the methods implemented in the API. This accessibility is enabled in the driver through the DriverDef class.

Within the Initnode method, the driver collects the information entered by the user in the graphical interface of the SCADA engineering environment and stores this information in variables. These variables are then used to create an instance of the

ConfigNode class, which contains all the pertinent data for the node. The ConfigNode class object is then stored in the global protocol object's node list. Figure 30 presents the implementation of the classes.

Figure 29 – Driver Initialisation.

```
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;
            }
        }
```

Source: O Autor.

From Figure 30, the ConfigNode class performs the function of defining the attributes of the project's nodes, as specified in the EIP protocol. In turn, the ProtocolDescription class plays an essential role by storing the nodes in a list. Additionally, it uses a dictionary to enable access and verification of the number of nodes configured in the driver.

This class structure reveals the organization and manipulation of nodes in the EIP protocol, providing a structured representation to manage the information necessary for the driver's communication and configuration.

Figure 30 – Class Implementation.

```
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++;
    }

}
```

Source: O Autor.

Continuing, after the information regarding the node is stored, the next step is to verify the addresses configured for that specific node. It is worth noting that the driver is being used by the DriverAPI to perform the communication. The ConfigNode class, which provides the nodes, comes from the API, as does the ConfigItem class, which provides the items configured in the project. Each item contains a specific address. According to the algorithm in Figure 31, the ExtractAddress method performs the parsing operation of the items' addresses, where the BlockId and AddressVar are extracted. The BlockId represents the byte position, while the AddressVar indicates the item's bit in the EIP packet payload.

The OrdenarConfig method organizes the Items in order based on the byte by the BlockId attribute and the bit by the AddressVar attribute, establishing a standard for accessing and editing these Items.

Figure 31 – Obtaining and Sorting 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
}
```

Source: O Autor.

After completing the operations on the nodes and Items, the subsequent step involves organizing the Items into blocks. The blocks represent a way to group and segment the Items according to the specifications defined by the developer. Similar to previous procedures, this method is invoked by the API, and all the Items configured for the Channel are provided, that is, all the Items from all the nodes.

Given that a Channel can have multiple nodes, the Blocks are delineated based on the node to which the Item is associated. Thus, as the Items are already ordered by their respective bytes and bits by the OrdenarConfig method, they will now be grouped into blocks determined by the node to which they belong. According to the algorithm in Figure 32, the first parameter of the method, named first, represents the first Item configured by the user, while the parameter item represents the next Item. The API uses this method by passing all the Items, one at a time, to the item and first parameters, thus performing comparisons between all the Items. If the difference between the byte of the last item received by the method and the byte of the first item is greater than the maximum size supported by the EIP protocol, that is, 1000 bytes, the inconsistency is reported to the

user. When the method returns *true*, the API groups the Items in the same Block; if it returns *false*, the Items do not belong to the same Block.

Figure 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;

    }
}
```

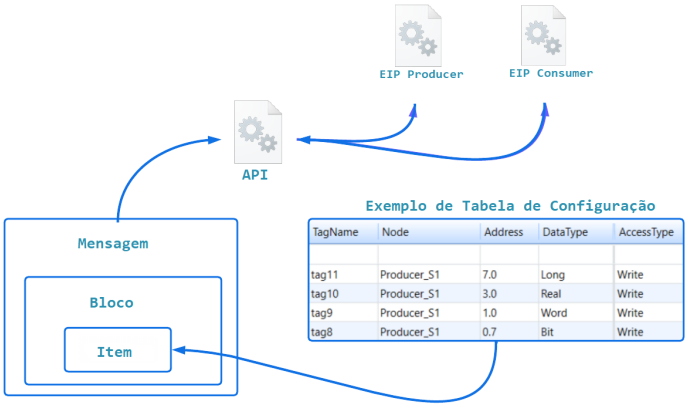
Source: O Autor.

Figure 33 represents an informal model that makes visible the origin of the discussed objects and the relationship between them. In the model, the API makes calls to the Drivers, and the Message flows between the API and the Drivers, carrying the data packet to be sent and information about the Driver configuration.

Each Item belongs to a Block, where the information configured in the graphical interface of the Devices module is encapsulated. Each line configured in the Devices interface represents a Communication Point, including an associated Tag, node, Address, and data type; the data types of the Communication Points will be addressed later. Each node, in turn, belongs to a Channel. In the context of EIP, each node can be a Producer or Consumer, identified by a specific ID and Exchange. The address consists of a byte followed by a bit, determining the payload position for writing (in the case of the Producer) or reading (in the case of the Consumer). The information regarding the Address is stored in the Item, and these Items are segregated into Blocks, separated in the Driver according to the node to which they belong.

The Blocks, in turn, are stored in the message. Each command received or sent by the channel creates an instance of the Message class, being handled by the API in conjunction with the Drivers. The API manages various nodes and Channels, containing confidential information that cannot be detailed in this work.

Figure 33 – Relational Model.



Source: O Autor.

The next step consists of building the packet using the PrepComando method, which is responsible for constructing the part of the packet corresponding to the EIP protocol. Since this part of the driver refers exclusively to the Producer, only write commands are sent. When starting the construction of the header, the buffer is first cleared as a safety measure. Systems typically have Tx and Rx buffers. The Tx buffer, known as the transmission buffer, is a storage area in a system where outgoing data is temporarily stored before being transmitted through a communication Channel.

The Rx buffer, or reception buffer, is where received data is temporarily stored after being received by the communication Channel. It serves as a temporary storage area until the system is ready to process the received data. The buffers are developed internally and constitute a byte array belonging to the message. The message, defined as *msg*, is an object of the Message class. This class is of fundamental importance because it contains the Blocks and Buffers. It is used by the API both to send and receive bytes from the lower layers, depending on whether the operation is a write or read, respectively.

As shown in Figures 34 and 35, it is first checked whether a write command is being sent to the Driver. If so, the specific node that sent the command is found based on the Items contained in the message block, as explained earlier. The algorithm presents each field of the protocol header, with the first two positions of the TxBuffer being fixed and the others filled with information from the specific node. Bitshift and bitwise operations are used to obtain the bytes of the fields in *little-endian* and store them in the Tx buffer. It is important to note that some fields are dynamic, even with a fixed number of bytes, having the value as inserted in the node configuration, while others are static with a fixed value, totaling the 20 bytes of the EIP header.

Figure 34 – Implementing the EIP Header - Part 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
```

Source: O Autor.

Figure 35 – Implementing the EIP Header - Part 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
```

Source: O Autor.

3.3 PAYLOAD IMPLEMENTATION

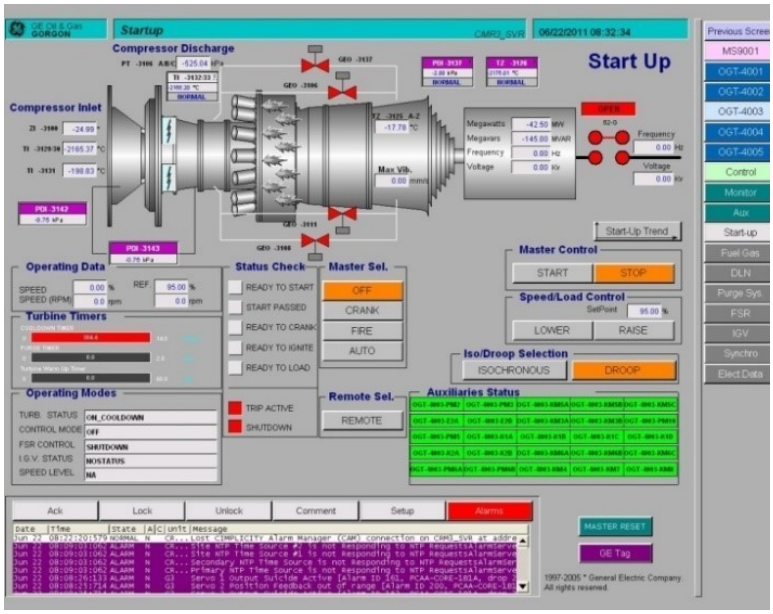
The payload is an essential part of a data packet in communication protocols. In the context of data transmission between devices or systems, the packet, as previously discussed, is generally composed of two parts: the header and the payload.

In the previous section, the implementation of the header was demonstrated in detail. In this section, the steps taken for the development of the algorithm that implements the payload of the packet in the driver project will be presented. In this driver, the payload is composed of the set of values that represent the physical variables of the turbine being controlled. These values are used for the exchange of information between the driver and

the DCS, enabling bidirectional communication between the devices. When communicating with the DCS, the Driver receives and sends the equipment control parameters, which may include data such as pressure, speed, temperature, reference, and setpoint, among others.

In the SCADA, the values are stored in internal variables called Tags, which are organized in a hierarchical structure within the SCADA system, facilitating categorization and quick access to the data. System operators can view the Tag information on supervisory screens, make adjustments to control parameters, and monitor the performance of the equipment.

Figure 36 – Turbine Visualisation Project Example.



Source: (SSE, 2020).

Figure 36 shows the supervisory screen, containing a sample project for the control and supervision of a turbine. The displayed values are the values of the project's Tags, which are configured to read and write data on the network. In the case of the Producer, the Tags fill the payload of the EIP packet, and the command is sent over the network. This process occurs periodically as specified by the protocol.

For the Consumer, as will be demonstrated later, the Tags store the value of the data contained in the payload of the received packet, and then the value is displayed on the supervisory screen. Typically, the SCADA supervisory screen is compatible with both HTML5 technology and the WPF platform, meaning it can be displayed on the network via a browser or in a window on Windows.

The procedure for creating the payload consists of including the values of the Tags in the packet. To accomplish this process, it is essential to determine the data type, which dictates the number of bits required to fully represent each value. The variety of data types supported by the protocol is illustrated in Table 2.

Type	**Size**	**Value Range**
Bit	1 bit	0 or 1
Word	2 bytes or 16 bits	-32768 to 32768
Real	4 bytes or 32 bits	-9.99×10^{37} to 9.99×10^{37}
Long	8 bytes or 64 bits	-9.22×10^{18} to 9.22×10^{18}

Table 2 – Data Types Supported by the Protocol.

The Tags are configured to perform read and write operations on the Communication Points associated with the node within the corresponding protocol Channel. Figure 37 presents an example of the configuration of the EIP protocol Points. As previously explained, each Point has specific attributes, such as address, data type, and access type. In the context of the EIP protocol, the address is fundamental for locating the tag value in the payload. This address is composed of two parts: the byte position and, if the data type is Bit, the bit position, separated by a dot. In the case of a data type that is not Bit, the bit value is zero.

The data type plays a crucial role in determining how many bits the Tag value will occupy in the packet. This results in an offset relative to the original address. For example, if a tag is at address 126.0 and has the data type Word, it represents position 126 of the payload. Due to the Word data type, an additional 16 bits or 2 bytes are added to this address, covering positions 126 to 127 of the payload. Consequently, the next value must start at position 128.

The access type, in turn, determines whether the operation involves reading or writing a value. In the Consumer, all Points are configured exclusively for reading, while in the Producer, the Points are intended for writing. In the XML structure of Figure 27, it is possible to identify the defined format for the configuration of the Points.

Figure 37 – Example of Communication Point Configuration.

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

Source: O Autor.

AAnalyzing the code related to the implementation of the payload in Figure 38 and Figure 39, it is possible to visualize the relationship between the configured project and the driver development, especially in the PrepComando method used to fill the header and the payload.

Within this method, an algorithm is employed to iterate through the Items contained in the list. Each Item in this list represents a configured Point. During the iteration, the byte and bit index corresponding to the Item configured at the address is recorded. Next, a check of the associated data type is performed to fill the appropriate buffer. When the data type is Word, which occupies 2 bytes, each byte is extracted from the value and stored in the buffer at a position determined by the index.

In the case of bits, a slightly different approach is adopted. For each bit value, it is checked whether it is equal to 1 or 0, and then it is inserted into the bit vector. This process is repeated until the last bit corresponding to the byte in question is reached. Upon reaching the last bit configured for the byte, the bit vector is inserted into the buffer at the position indicated by the byte index.

Figure 38 – Filling the Payload - Part 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;
```

Source: O Autor.

Figure 39 – Filling the Payload - Part 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;
```

Source: O Autor.

3.4 PRODUCER TEST

Upon completion of the Producer development, the Driver undergoes tests to ensure its functionality and the effectiveness of communication with the Consumer. These tests involve the transmission of packets over the local network, covering all data types. The

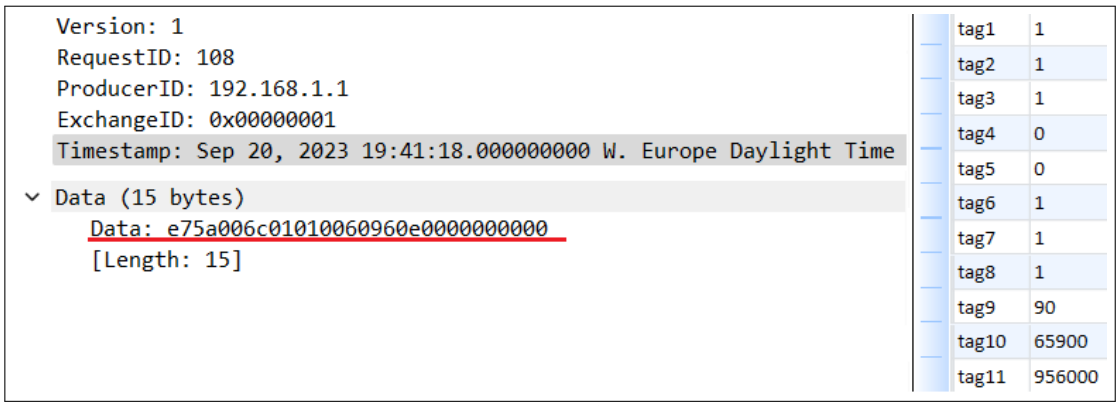
values are written to the Tags previously configured within the SCADA itself, and the result is then verified in the packet payload using Wireshark, as illustrated in Figure 40 and Figure 41.

Figure 40 – Configuration of Communication Points for Testing.

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

Source: O Autor.

Figure 41 – Result obtained from sending the parcel.



Source: O Autor.

The analysis of the results reveals that the Driver operates according to the established expectations. The network accurately identified the packet as EIP, and all configured information is presented accurately. The payload also complies with the previously defined requirements.

Figure 42 illustrates the obtained results, representing the values of the first eight configured Tags in hexadecimal format, where each value is composed of 1 byte. In binary, these values translate to 11100111, equivalent to `0xE7` in hexadecimal in little-endian format, as demonstrated.

The next 2 bytes represent the value of Tag9, of type Word, with 90 equivalent to `0X5A` in hexadecimal, while the second byte remains null as it was not used. The result of

Tag10 is 65900, which, when converted to little-endian format, becomes `0x6C 0x01 0x01 0x00`.

Finally, the value 956000 is equivalent in little-endian format to `0x60 0x96 0x0E`, as demonstrated. These results highlight the precision of the conversions.

Figure 42 – Payload of the Packet Sent by the Driver and Received at the 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

Source: O Autor.

3.5 CONSUMER DEVELOPMENT

Regarding the development of the Consumer, the PrepComando method, responsible for building and writing commands on the network, is implemented only because the IComm interface of the DriverAPI assigned to the class requires the implementation of the method.

Figure 43 – PrepCommand method in 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;
    }
}
```

Source: The Author.

3.5.1 Message Verification

The implementation of operations with nodes, Items, and Blocks is similar to the Producer. However, the Consumer implements an additional method called VerificaProtocolo, which plays a fundamental role in the functioning of the Consumer. This method operates as a state machine, responsible for receiving the message from the Producer and determining if the reading is necessary. For this, it must go through a series of states in sequence.

The first state, Figure 44, performs the initial verification of the received message, analyzing if the initial byte corresponds to the PDU field, similar to the EIP protocol. All EIP packets have the value 13 or the hexadecimal equivalent `0xA` in their initial position.

Therefore, this state discards messages that do not have this desired value, preventing unwanted messages from overloading the Driver's state machine by eliminating these unwanted packets from the beginning of the process.

Figure 44 – First State.

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

Source: The Author.

If the first stage is satisfied, the counter increases by one unit, the byte is stored in the buffer, and it moves to the next stage, 45, where a simple static byte of the EIP header is verified again.

Figure 45 – Second State.

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

Source: The Author.

The third stage, Figure 46, verifies the Producer ID. The verification is performed by storing 4 bytes, referring to each of the values of the Producer ID. Then, it is checked if the Producer ID of the received message has any node with a similar Producer ID. If so, it moves to the next state.

Figure 46 – Third State.

```
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;
```

Source: The Author.

The fourth state, Figure 47, verifies the Exchange ID of the received message by comparing the bytes of the packet from position 12 to 16. In other words, the Exchange, as determined by the protocol, occupies 4 bytes. The expected location of these bytes is verified in the payload, and just like the procedure performed for the Producer ID, the algorithm checks if the Exchange is present in any node. If so, the algorithm moves to the next state; otherwise, it discards the message and updates the buffer.

Figure 47 – Fourth State.

```
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;
```

Source: The Author.

Finally, in the fifth state, Figure 48, the payload is verified, where the corresponding node is identified. In other words, each of the stored fields is verified to find the node that has the same values. This node is then stored in the CurrentNode attribute for later use. Subsequently, the bytes of the packet are stored in the DataBuffer of the message, also being preserved for subsequent uses.

Figure 48 – Fifth State.

```
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;
```

Source: The Author.

3.5.2 Payload Reading

After verifying the message and storing it in the Rx and DataBuffer buffers, the next procedure consists of reading the buffer values and storing them in the corresponding Tags.

As shown in Figure 49, the TratarMensagem method is responsible for interpreting unsolicited messages. In communication drivers, it is common to send a read request command. When the communicating entity receives, verifies, and successfully interprets this command, a corresponding response command is sent, establishing a communication channel in which the success confirmation is awaited through the reception of the response message. However, in this case, by using the UDP protocol, the messages are distributed over the network without a prior request and without receipt confirmation.

The algorithm works by iterating through all the Items in the message and verifying if the Item belongs to the node identified in the VerificaProtocolo method. As explained in the ExtractAddress method of the Producer, similar to the Consumer, the BlockId and AddressVar attributes store the byte and the bit, respectively. These values are stored locally by the index variables.

The switch implemented in the code checks the data type configured for the Item and then searches within the DataBuffer containing the EIP packet payload for the value to assign to the Item based on its index, i.e., the bit for the DataType.Bit case and the

byte for the others.

Thus, according to the address defined for the Item, the Item being a Tag configured as a communication Point in the SCADA, and the address of the respective Item composed of the byte and the bit, these being the location of the Item within the payload. For the Word case, there is an offset of 1 byte in relation to the Item byte, totaling 2 bytes. In this way, the address location is searched within the buffer, and two bytes in little-endian are accessed, which return the decimal value to be stored by the Tag configured for that communication Point.

Figure 49 – Payload Parse Procedure.

```
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;
            }
        }
```

Source: The Author.

The procedure is repeated for the other data types, except for the bit. In the case of a Tag storing a boolean value, the process involves accessing the byte of the Item address of the bit type (boolean). Within the eight bits that make up the byte, it is checked, through the bit index, if that position is 1 or 0. The value is then stored in the respective Item and, consequently, in the Tag configured for that Point.

3.6 CONSUMER TEST

To conclude the implementation of the Consumer, the next step involves performing a series of tests on the driver to ensure its proper functioning, both in a local environment and later in production.

At this stage, it is crucial to configure the Channel, the node, and the Consumer communication Points. Additionally, new Tags must be created to capture the values generated by the Producer, and these Tags must be configured as communication Points in the SCADA, as illustrated in Figure 50. This will allow not only the reading of the produced values but also the effective and secure storage of these received values.

Figure 50 – Project Points List in 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

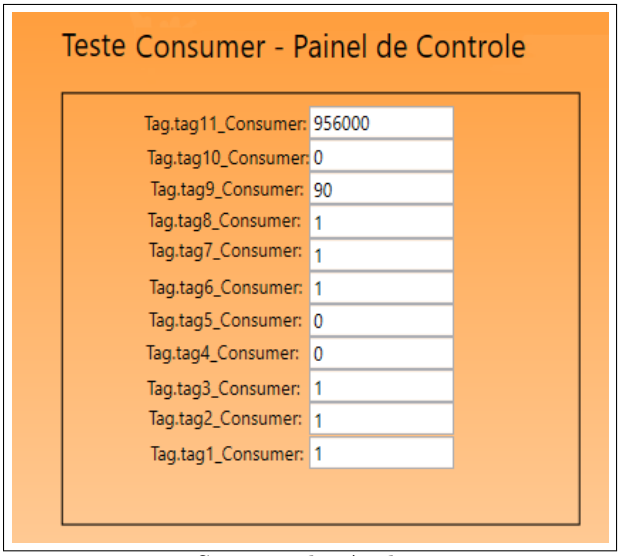
Source: The Author.

In the project, the commands are transmitted by the Producer similar to the values sent in the Producer test presented in Figure 41, through the local network and later read by the Consumer. The result of these operations is then presented on a WPF display, as illustrated in Figure 51. It is important to note that during this test, the values used are arbitrary and intended exclusively for verification purposes. However, this process demonstrates the functionality of the SCADA system in terms of monitoring and controlling the values received and sent by other devices on the network.

The result obtained in Figure 51 highlights the performance of the Drivers. Each presented value reflects the success of the algorithm developed for the Producer, which fills both the header and the payload. Similarly, the Consumer is able to accurately receive the packet by verifying the header, extracting the payload values, and storing them in

the Tags, later displayed on a WPF client display in the SCADA. This demonstration remarkably evidences the effective functionality of both Drivers in the SCADA system, illustrating the harmonious interaction between these elements and providing an incredibly useful solution.

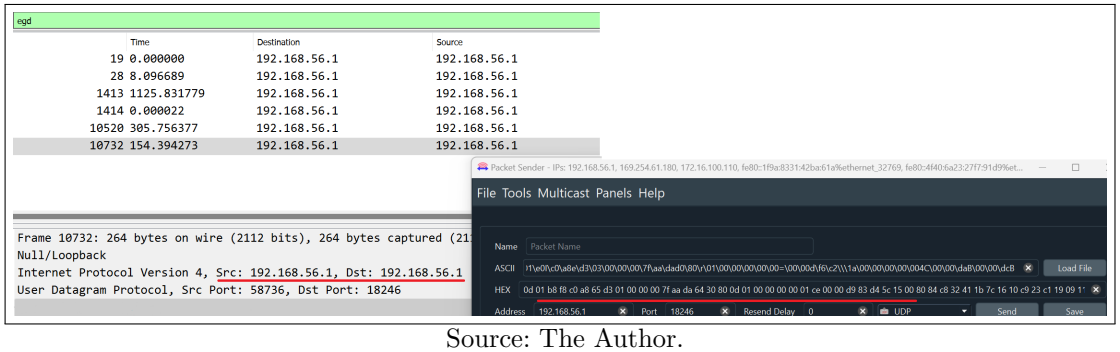
Figure 51 – Result Visualization.



Source: The Author.

Continuing, the next test was performed using the Packet Sender software, an open-source tool widely used by network professionals and developers. This software is used to send and receive network packets in various protocols, being useful for verifying network connectivity, debugging issues, and performing network-related tasks. Commands with the EIP header were sent through the local network using the UDP protocol. These commands were directed to the specific port 18246, where the Consumer was configured to await the arrival of the packets, as illustrated in Figure 52.

Figure 52 – Test Using Packet Sender.



Source: The Author.

4 CONCLUSION

This work achieved the established objectives. Through a fictitious example of implementing a driver for a SCADA system, it is hoped to contribute to the knowledge base in SCADA systems and industrial communication protocols. These areas, often inaccessible due to confidentiality in highly competitive environments, were explored didactically throughout the work, from protocol analysis to driver development and bench testing.

Since the *Ethernet Industrial Protocol* is fictitious, it was a challenge to determine its specifications, following the basic standards of industrial protocols and aiming to present a simple and didactic protocol. Thus, it is hoped that this work provides comprehensive details about the protocol for the knowledge of engineers and developers, also aiming to facilitate the understanding and development of communication drivers, addressing the lack of available information so far.

The scarcity of information about the process of integrating a communication driver into a SCADA system was another focal point of this work. With the detailed presentation of the development procedure, it seeks not only to fill an existing gap but also to offer a valuable reference for professionals seeking practical insights.

The successful validation of the drivers' functionality, as demonstrated in the bench tests, confirms their effectiveness in communication using the EIP protocol and adds to the knowledge base related to testing and debugging communication between devices.

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, June 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. Available from: 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. Available from: https://inductiveautomation.com/ignition/architectures.

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

KOZIEROK, Charles M. **Binary Information and Representation: Bits, Bytes, Nibbles, Octets and Characters**. 2023. Available from: 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. Available from: https://www.spiceworks.com/tech/devops/articles/what-is-device-driver/.

MOXA. **Moxa Secures Your OT Networks With OT/IT Integrated Security**. 2024. Available from: 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. Available from: 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. Available from: 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. Available from: https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-.

VIVIANO, Amy. **Windows network architecture and the OSI model**. 2023. Available from: 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.