A Communications Testbed for Testing Power Electronic Agent Systems

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Benjamin Roy Dean
May 2021

Dedication

To my family, for their unwavering love and support

Acknowledgements

There are many people who I would like to thank for assisting me in the completion of this thesis. First, I would like to thank my advisor, Dr. Leon Tolbert, for offering me a research position at CURENT as a young undergraduate student, and furthermore encouraging me to attend graduate school. I would also like to thank Dr. Michael Starke for supporting me as an undergraduate intern and graduate research assistant at ORNL. Both Dr. Tolbert and Dr. Starke provided a great deal of guidance, support, and encouragement that made this thesis possible.

I would also like to thank my committee members, Dr. Qing Cao and Dr. Fangxing Li, for their helpful insight into this thesis.

I would also like to thank my peers for their support during this thesis. A special thanks goes to my officemates, Peter Pham and Paychuda Kritprajun, whom I surely distracted with my conversations more than they would have liked. I would also like to thank Mitch Smith for providing insight and suggestions about my research. Last, I would like to thank Taylor Short for the many times she has been an excellent lab partner and classmate.

Abstract

As power electronic systems (PESs) continue to incorporate complex intra-system communication, understanding and characterizing this communication has become a complex task. Knowing how a system's communication will behave is vital to ensuring proper operation of these systems. This thesis proposes and outlines a communication testbed that streamlines the development and testing of the communications between the components of a PES, and further presents the characterization of certain communication protocols that are utilized in these multi-agent PESs. These communication protocols include MQTT, Modbus, and User Datagram Protocol (UDP). Understanding the different behaviors of these protocols is paramount for the design of PESs.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The modern power grid shows a large shift in its operation characteristics as more power electronic systems (PESs) are incorporated into this electric network. It is predicted that by 2030, 80% of power generated will flow through some form of power electronics [1]. As these systems see more widespread adoption, their operation and interconnection become increasingly advanced and complex. Traditional communication implementations give way to modern communication designs that take full advantage of the available data and computational power widely available today.

1.1 Power Electronic Systems

A PES can be defined as an electrical system that utilizes power electronic devices. Examples of PESs include energy storage systems (ESSs), energy management systems, power quality systems, etc. Traditionally, the communication utilized by these systems have been industry accepted standards. These standards include the distributed network protocol (DNP) commonly implemented by utility supervisory control and data acquisition (SCADA) systems. This standard is widespread and easily scalable [2]. However, this is a centralized communication protocol, and decentralized communication implementations are of critical importance for the future development of these systems [3].

1.1.1 PES Overview

A PES can incorporate numerous different components. The main component of a PES is the power electronic converter (PEC). A PEC uses a solid-state device to control and convert electric power from one form to another [4]. In general, these converters can be classified into four different categories: DC-DC converter, AC-DC converter (rectifier), DC-AC converter (inverter), and AC-AC converter (cycloconverters). A PES can contain multiple power electronic converters.

The additional components in a PES vary widely depending on the system's purpose and application. These components include renewable energy generation, batteries, SCADA interfaces, intelligence controllers, etc. An example of a PES is a residential ESS. A typical

residential ESS consists of batteries, renewable generation such as solar, and a grid-tied inverter. A block diagram of the electrical connections of this typical residential ESS is shown in Figure 1.1.

Each component, or subsystem, in the ESS has a specified task or set of tasks that it is responsible for accomplishing. The DC-DC converter that connects the solar panels to the DC bus regulates the power that the solar panels produce through *maximum power point tracking* (MPPT). The power produced by the solar panels varies with the voltage across them; therefore, the DC-DC converter is needed to control this voltage, and therefore control the power produced. The battery bank interfaces with the DC bus through a DC-DC converter. This converter is responsible for charging and discharging the batteries and regulating the voltage of the batteries and DC bus. The inverter is needed to interface the DC bus with the AC power grid. It must be rated to handle the necessary amount of power and be compliant with IEEE Std 1547 [5]. This standard states that the inverter must be capable of actively regulating voltage, ride through abnormal voltage and frequency, and be capable of frequency response. A common method for controlling this type of system would a centralized intelligence unit tasked with telling each component how to behave [6].

1.1.2 Systems Become More Complex

The simple system presented in Figure 1.1 requires communication coordination between its components for successful operation. As these systems increase in complexity, more components are incorporated to expand their functionality. These components include HVAC systems, utility interfaces for price signals, electric vehicle (EV) chargers, additional energy storage capacity, etc. Incorporation of these devices requires these PESs to be configurable, expandable, and open source. A single PES becomes a "system of systems," which encourages the development and implementation of decentralized control schemes. Instead of having a single centralized controller, components have individual controllers that communicate with all other PES subsystems to find the component's best operating conditions based on an overall system goal.

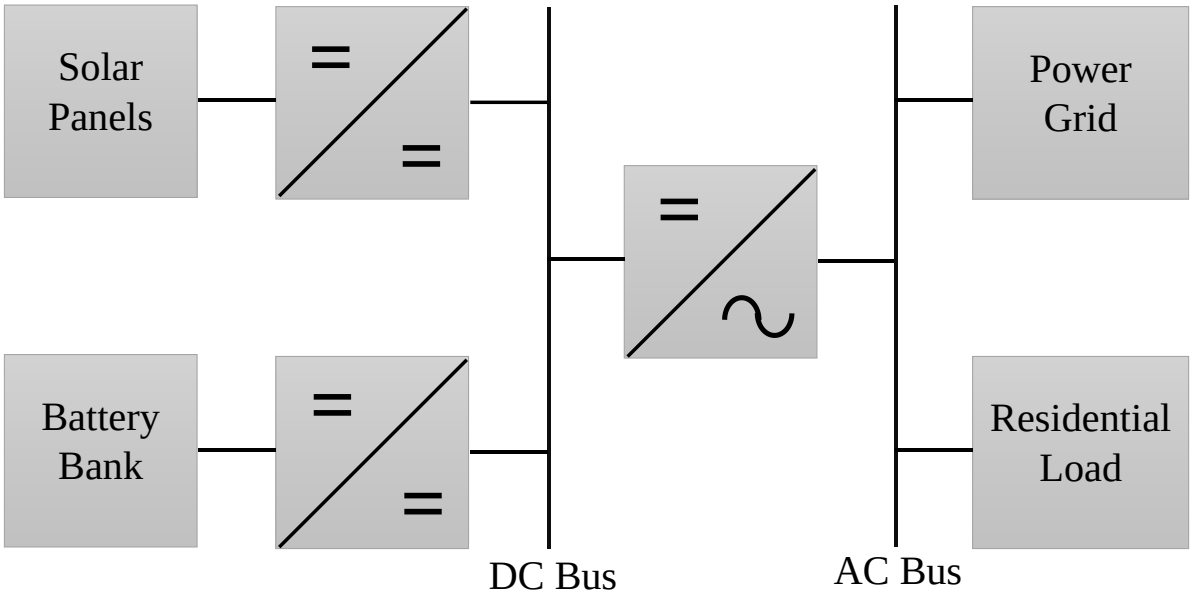


FIGURE 1.1: DIAGRAM OF THE ELECTRICAL INTERCONNECTION OF A TYPICAL RESIDENTIAL ESS

A modular, open source PES structure enables flexible system functionality. From a hardware perspective, it allows components to be replaced or upgraded as the owner sees fit. An inverter or battery management system (BMS) can simply be replaced or upgraded. Therefore, for modular PESs to be expandable and customizable, the communication methods used must be common and open source. PES with company-specific proprietary communication implementations, such as the Tesla Powerwall, have little capacity for expansion or upgrades without the involvement of the original manufacturer [7].

1.1.3 Agent-based Power Electronic Systems

One decentralized control approach for PESs is an agent-based control scheme. Multi-agent systems consist of "agents" that have the capability to be reactive, proactive, and social. To be reactive, agents must demonstrate the ability to perceive the environment around them and react accordingly. To be proactive, they must demonstrate the ability to initialize and exhibit objective-directed behavior. Last, agents must be social by interacting with other agents or entities, such as humans [8]. Agent-based systems allow for multiple levels of software to be used to implement a decentralized architecture. A diagram of an agent-based ESS's communication connection is shown in Figure 1.2.

This control architecture offers many advantages for PESs. Each subsystem can be controlled by an agent and make the optimal decisions for its component. Data can easily be exchanged between the different components, and the reliance on central control is removed. It also allows for the easy addition of other devices that can provide the system additional data and capabilities.

1.1.4 PES Communication Development

Communication development for a PES requires integrating software with multiple layers of communication and translation. When communication is established, this software must be integrated with hardware. This is a time-consuming process, even with tools such as hardware-in-the-loop [2]. In order to expedite the development of PESs, a design tree is shown that proposes four steps from the initial communications to the final system prototype.

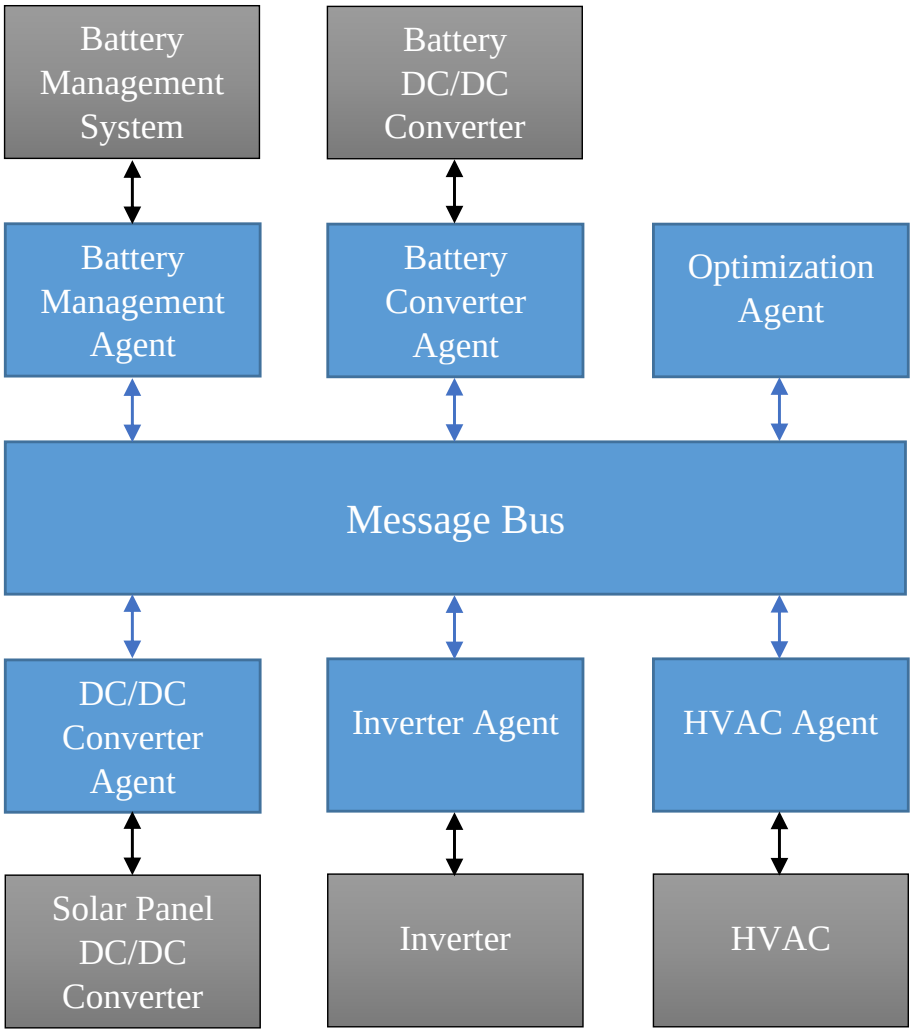


FIGURE 1.2: DIAGRAM OF THE AGENT-BASED COMMUNICATION FLOW OF THE PES PRESENTED IN FIG 1.1 [2]

1.2 Project Description

In this thesis, a testbed for the design and development of modular PES networked communication is presented. The objective of this testbed is to enable the rapid development of intrasystem communication, as well as ease the complication of implementing, testing, and executing different PES communication implementations. This testbed provides an environment to develop and implement new communication strategies while also evaluating its security, latency, throughput, and reliability.

1.3 Project Description

In this thesis, a testbed for the design and development of modular PES networked communication is presented. The objective of this testbed is to enable the rapid development of intrasystem communication, as well as ease the complication of implementing, testing, and executing different PES communication implementations. This testbed provides an environment to develop new communication strategies while also evaluating its security, latency, throughput, and reliability.

1.4 Challenges in Communication Development

Communication development presents numerous challenges that this testbed will address. Figure 1.3 shows a flow chart of the development of a PES. The first step involved is communications devolvement, which presents many factors that must be overcome before further system development. To begin with, the communication environment of an agent-based PES involves multiple communication nodes. Development of multiple pieces of software designed to communicate with one another cannot be tested on a single desktop computer due to the limited communication capability of “localhost,” which limits testing to a single network connection. The intent of this testbed is to provide a communication environment that closely emulates the multiple node environment implemented in a PES.

Furthermore, communication development requires defining a communication schema. A communication schema defines how the data in a communication implementation is organized. The schema allows the devices to know where to publish and access data, such as measurements, setpoints, or operating modes.

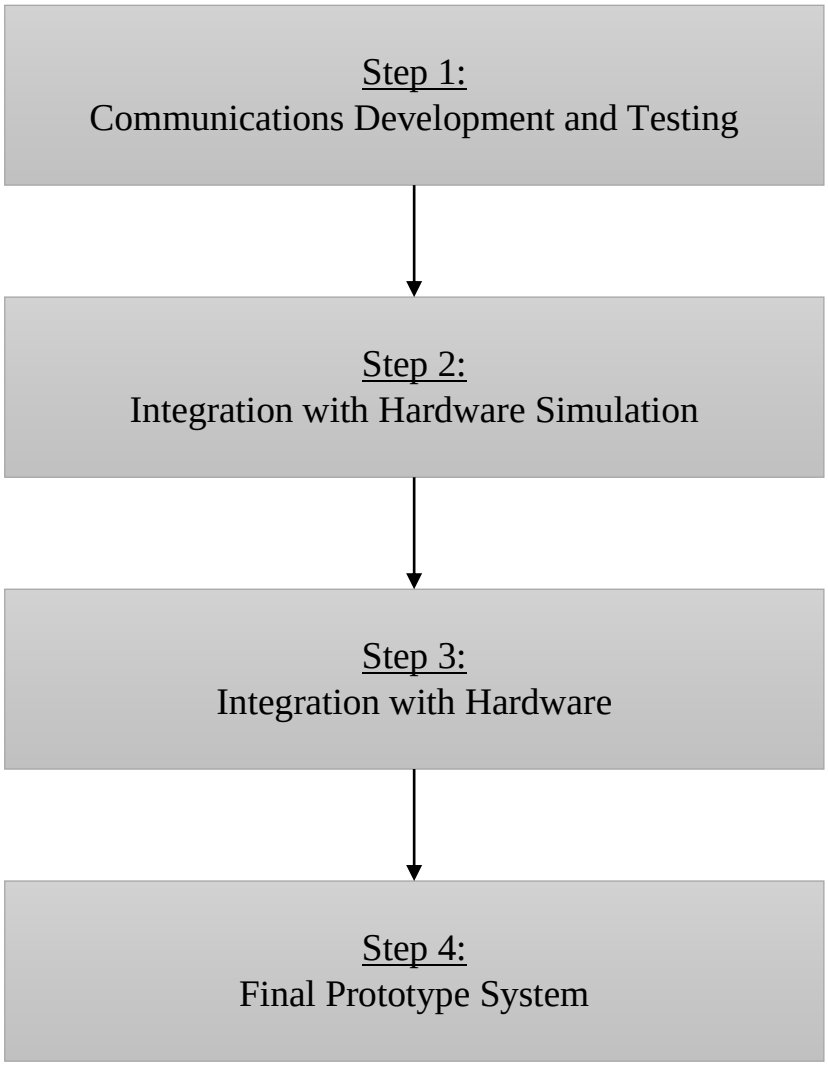


Figure 1.3: FLOW CHART OF THE DEVELOPMENT OF AN ADVANCED PES [2]

PES communication development typically requires multiple iterations of schemas to be defined as the operation of the PES is refined. This testbed will allow for the rapid definition and testing of these schemas.

Last, the testbed will enable quicker adoption of PES hardware. By emulating the PES communication, actual PES components, such as digital signal processors (DSPs), can be connected to the testbed equipment to ensure proper communication is taking place.

1.4.1 Motivation for Communication Testing

The communication protocols chosen for PES communication affect the system's operation. Different protocols have different latencies, messages per second, error rates, and throughputs. These factors are important to have quantified when developing the control for the PES. This testbed enables rapid quantification of communication characteristics.

1.5 Overview

Due to the advancements in PES communications, the development of a testbed environment to develop and test PES communication is the objective of this thesis. Detailed is the background, development, and testing of PES communication

Chapter 2 overviews the communication environment surrounding modern PESs. This includes detailing how systems communicate with their surroundings. It further examines the common communication protocols used within these systems. Last, it discusses other currently developed and implemented communication testbeds.

Chapter 3 discusses the hardware utilized and software developed for the communications testbed. It details the testbed's development and provides an overview of its operation.

Chapter 4 details the communications tests performed on the testbed. The results of these tests are shown. These tests are used to compare different PES communication implementations.

Chapter 5 concludes and summarizes this thesis and discusses future work.

Chapter 2: Literature Review

The push for more advanced PES communication is largely due to the high rate of data utilization and exchange required by these systems. Therefore, in the following literature review, the communication environment of advanced PESs is explored. By understanding how these systems connect to themselves and the world around them, the push for increasingly complex intersystem and intrasystem communication is further understood, as well as the challenges involved in the development of these systems.

The first topic covered are advanced PES's communication with the outside world. The motivation and methodology behind an advanced PES with a high level of intersystem data exchange directly impacts how these systems are designed and implemented. This volume of data collected by these systems impacts their intrasystem operation.

Next, the intrasystem communication of advanced PESs are reviewed. This includes detailing the communication between the components of an advanced PES, such as inverters, DC-DC converters, and intelligence interfaces. Common networked communication protocols are detailed, and their relation to the development of advanced PES is demonstrated.

The last topic explored in this chapter is previous implementations of communication testbeds. The application and methodology of these testbeds are explored, with the purpose of better understanding the functionality of communication testbeds. Specifically, local-area-network (LAN) testbeds with multiple nodes and a focus on communication research are explored.

2.1 Advanced Power Electronic Systems

The trend with modern PES is to utilize the large amount of available data to optimize its functionality and to share its own data with the world around it. This push for high interoperability in PESs further pushes for the development of increasingly advanced PESs, leading to various implementations of intersystem and intrasystem communications that facilitate the sharing of data between devices. These devices include large utility SCADA systems that can control a residential ESS's grid functionality or simply the homeowner's in-house devices, which

give the ESS the ability to make decisions based on a user's energy usage or economic goals. These high interoperability, advanced PES must be purpose built to have a high level of data exchange with the world around it.

2.1.1 Internet of Things

The Internet of Things (IoT) is a networking concept that describes how physical objects communicate over the internet. Figure 2.1 shows some of these objects, or "things," that produce and collect large amounts of data. Therefore, they need a framework for communication with one another [9]. The term "IoT" encompasses numerous different communication protocols and implementations that interconnect these devices. These protocols typically enable devices to publish and subscribe to data within a set environment. This data is routed through a gateway to be made available to all connected and compatible devices [10].

The IoT has played a large role in the development of advanced PESs. An IoT agent-based implementation of a PES allows for easy data distribution between the system and the world around it. Ref. [11] presents an ESS that encompasses the system of systems approach. This ESS is shown in Figure 2.2. The BMS in this ESS receives information from more than 200 industrial components with power meters and must make the appropriate control decision to reduce energy peaks in the industrial site, known as *peak shaving.* Timestamped data is saved using cloud storage, and this is used to forecast the energy needs and determine the system's optimal control strategy. The IoT enables this system to communicate with weather and energy bill data, and therefore make the best decisions for the system to provide the owner with the best economic benefit. This system allowed Schneider Electric to save approximately $1000 a month on their utility bill.

2.1.2 OpenFMB

Open Field Message Bus (OpenFMB) is a framework for communications that uses the IoT publish/subscribe model. This framework is developed and owned by utility companies in the United States. The purpose of OpenFMB is to allow for decentralized communication on the power grid, which removes the need for a centralized control system and gives the utility grid devices a higher level of interoperability.

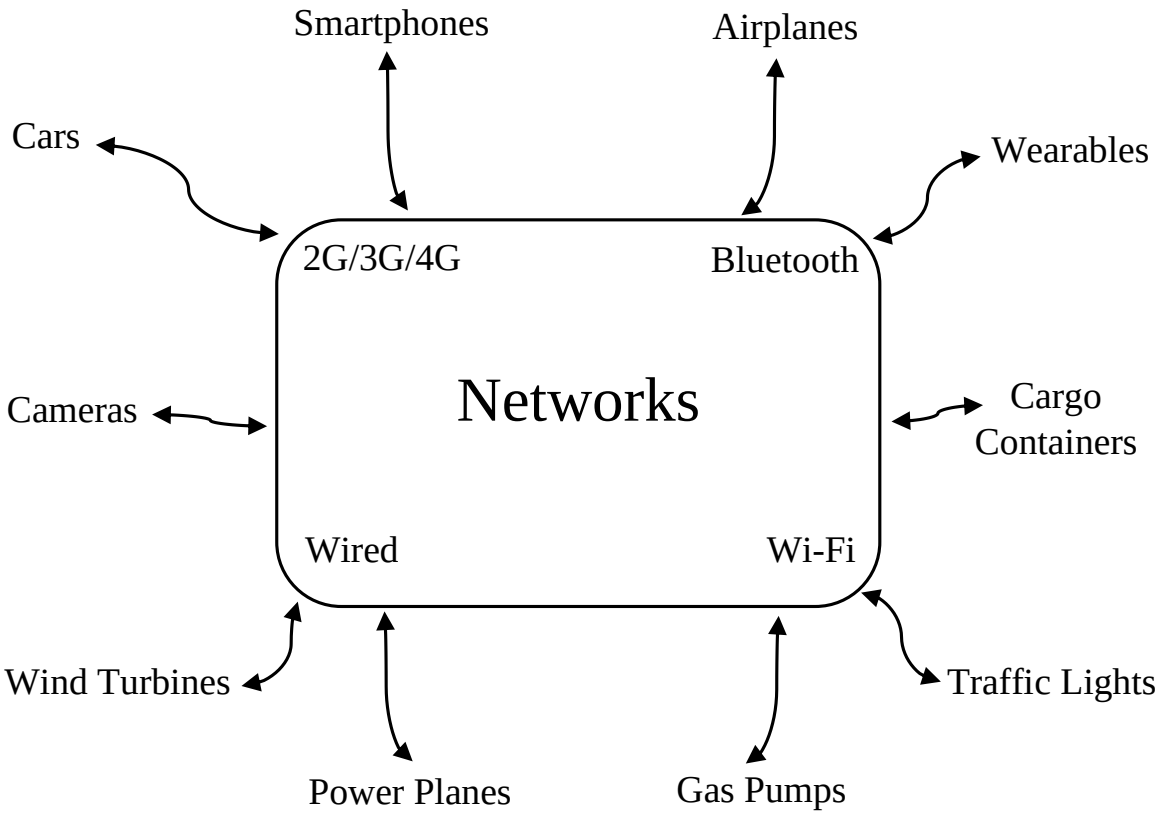


FIGURE 2.1: DIAGRAM SHOWING DEVICES CONNECTED THROUGH THE INTERNET OF THINGS [9]

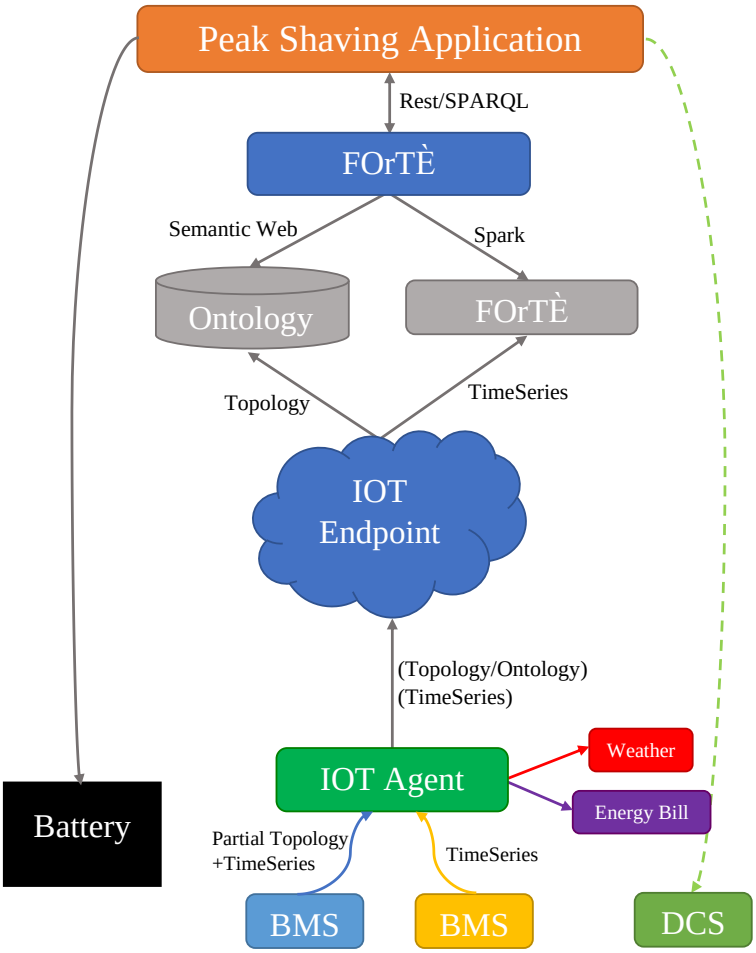


FIGURE 2.2: COMMUNICATION IMPLEMENTATION OF THE ESS PRESENTED IN [11]

Increased levels of data communication improve the grid's level of reliability and resiliency [12]. This framework is typically used in smart grid and microgrid settings, but also has applications in energy storage systems. A control architecture utilizing OpenFMB and incorporating an energy storage system is presented in [12], with the architecture shown in Figure 2.3.

2.1.3 IEC 61850

IEC 61850 is a widely-used standard that defines a communication protocol used by intelligent electrical devices in power grid substations [13]. It defines a standard way to describe and properly access a power device's data with the purpose of increasing interoperability between devices with different manufacturers [14]. It is designed to operate over TCP/IP ethernet and gives a larger degree of control than was previously available over serial based communication [13]. IEC 61850 provides complete, standardized, and self-describing object, as shown in Figure 2.4, that names for all the data it communicates. This makes communication between equipment of different manufacturers possible and facilitates possible plug-and-play functionality.

Utilizing this existing method of communication in PESs provides the opportunity to create an advanced system that is compatible with existing IEC 61850 compatible hardware. This is demonstrated in [15], which models a standard communication protocol based on IEC 61850 for a home energy management system. This model coordinates PV generation, EV charging, and loads of the home energy management system. The network is emulated in Netsim, and two nodes act as the IEC 61850 server and client. The IEC 61850 communication protocol is demonstrated in Figure 2.5. The server and client demonstrated successful messaging over this standard.

Another example of this standard being utilized in a PES is demonstrated in [16]. The purpose of this PES is to slowly provide active and reactive power to support wind farms that are weakly connected to the power grid. Digital controllers emulating the wind turbines and other devices communicate with the IEC 61850 GOOSE Messaging protocol, with the device's interconnections shown in Figure 2.6.

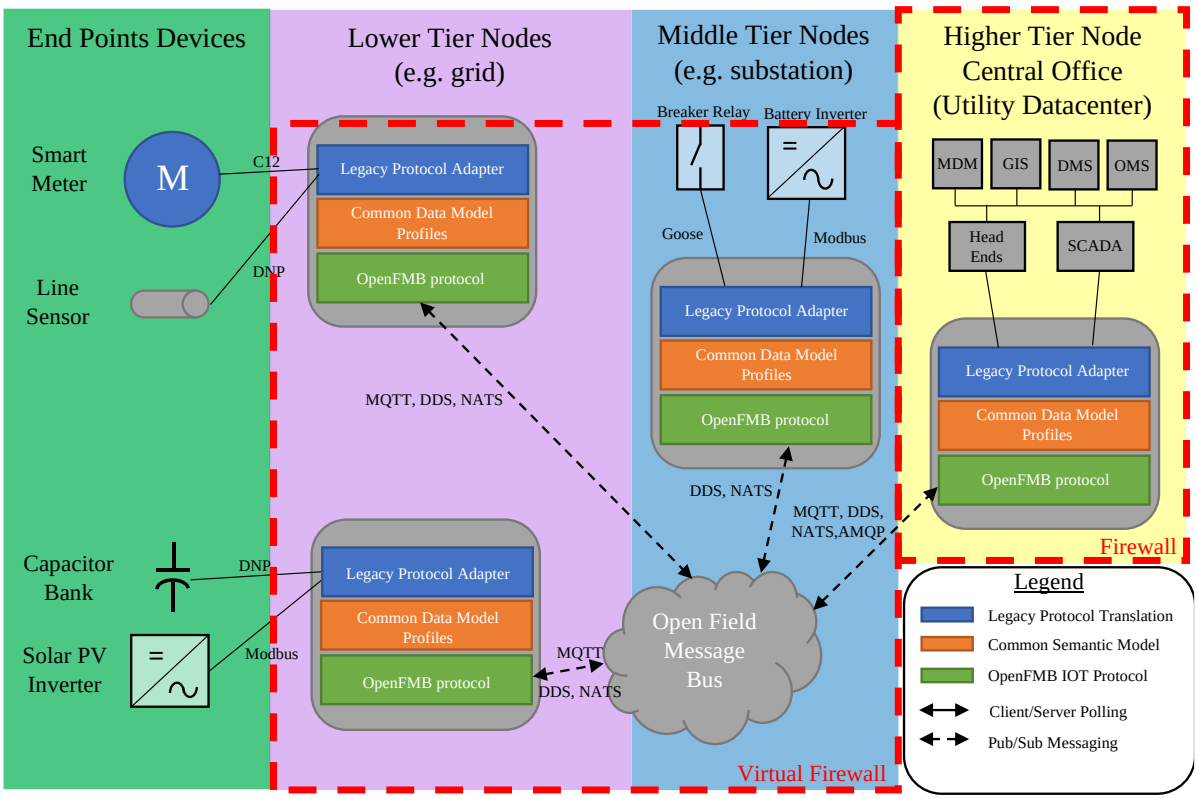


FIGURE 2.3: ARCHITECTURE OF AN OPENFMB CONTROLLED SYSTEM [12]

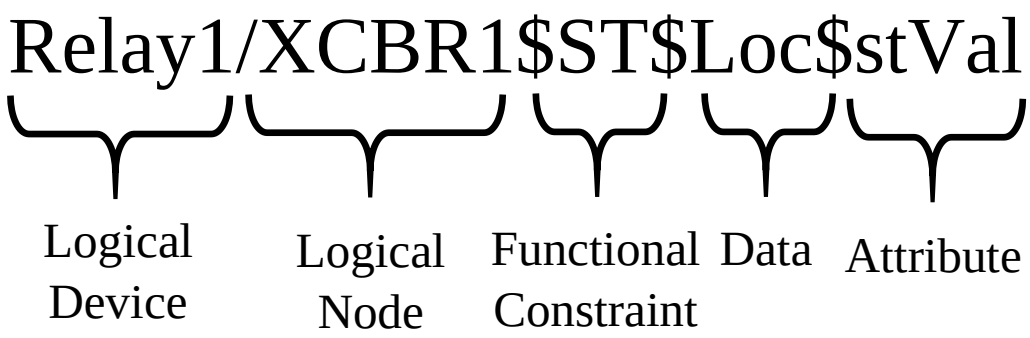


FIGURE 2.4: THE BREAKDOWN OF AN IEC 61850 OBJECT [13]

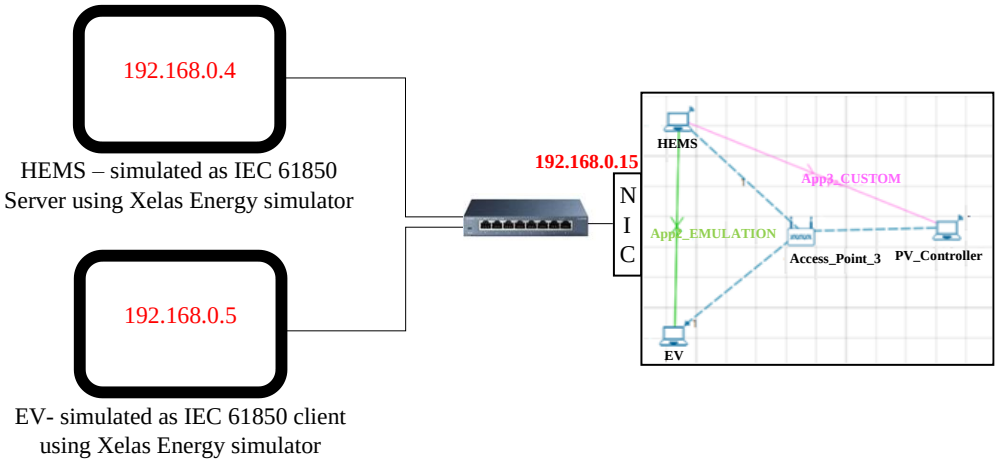


FIGURE 2.5: EMULATION OF THE IEC 61850 SYSTEM IN [15]

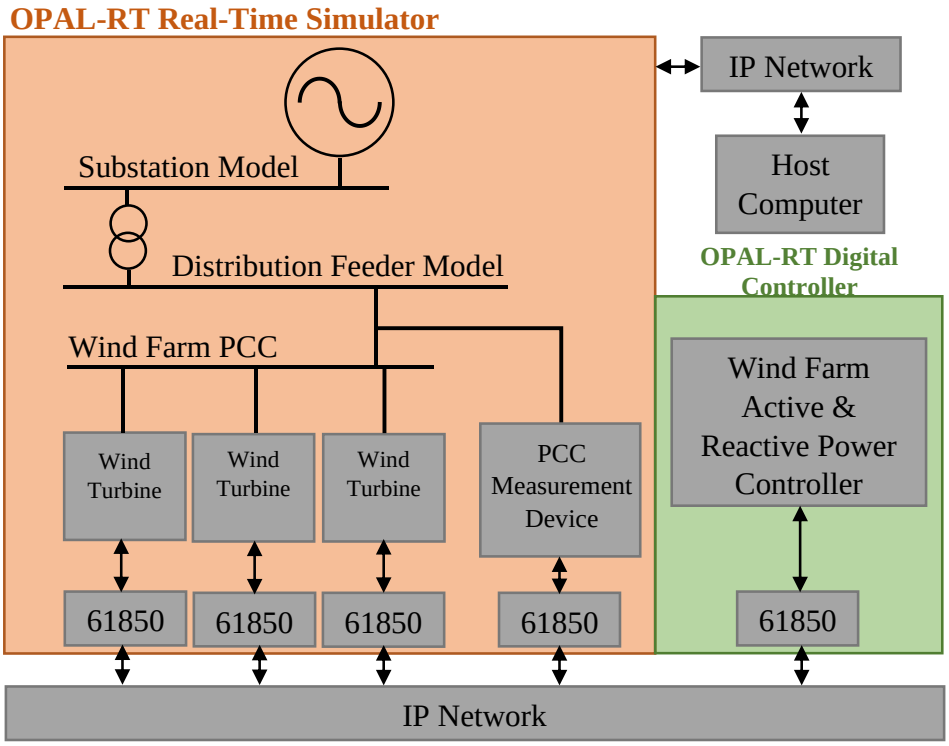


FIGURE 2.6: CONFIGURATION OF HARDWARE-IN-THE-LOOP EMULATION USING IEC 61850 FROM [16]

The data flow of an advanced PES is of importance during their design and development process. For systems to have seamless integration into the world around them, the necessary data exchange must be considered during the design process. Different frameworks have been developed that attempt to establish a generalized approach to the development of an ESS that takes advantage of the data available to it. Ref. [17] presents a method known as *Model Driven Engineering*, a method taken from the computer science domain. This method focuses on the necessary data flow in the operation of a PES. This data flow is visualized through a chart that maps its input, output, and progression. This approach's goal is to design with the intent of high levels of data exchange, instead of designing a system that simply accounts for data exchange. An example of this approach is carried out in [18], where IEC 61850 models are modified to conform to IEC 61499 models. By looking at the necessary data flow, shown in Figure 2.7, from one standard to another, a different communication protocol was easily adapted.

2.1.4 Plug-and-Play Standards

To ease the process of deploying high interoperability PES, plug-and-play standards are required. These allow for manufacturers to incorporate standard, open-source communication implementations with which any PES can comply. They also specify standard schema, typically over a common standard such as Modbus, for the PES to adopt. In communications, a schema outlines how data is organized in a specific communication implementation. For example, one communication protocol, Modbus, uses registers to store data, such a measurement like voltage or current. Each register has an address. Knowing which registers correspond to which values and addresses is fundamental for communications. Plug-and-play standards define these schemas, which allows for other devices to interface with each other without the need for additional setup.

These goals have been identified by many groups, such as Open Automatized Demand Response (OpenADR), who pursue a higher level of distributed energy resources (DER) integration through the use of plug-and-play standards [19]. It defines an OpenADR compliant schema and data type for devices to utilize. Figure 2.8 shows the communication overview of OpenADR.

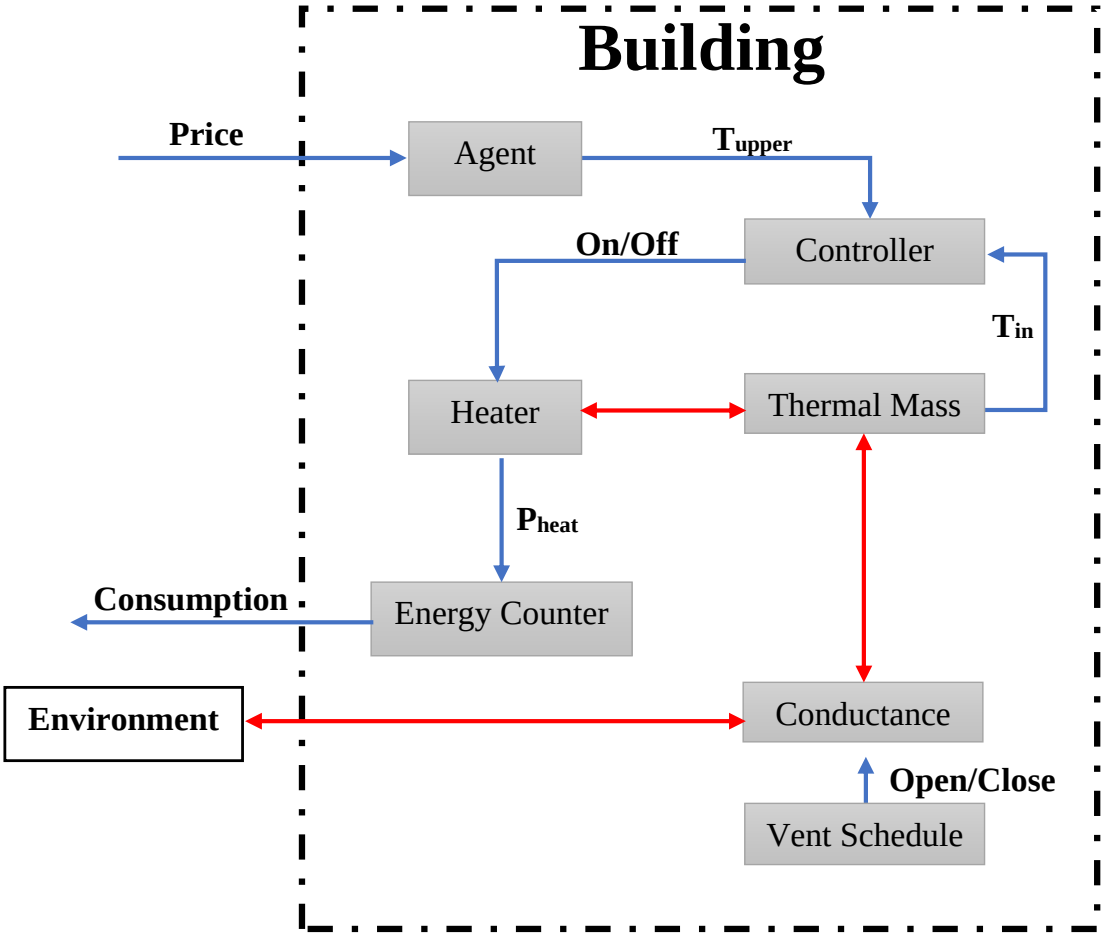


FIGURE 2.7: A DATA FLOW CHART MADE USING MODEL DRIVEN ENGINEERING FROM [18]

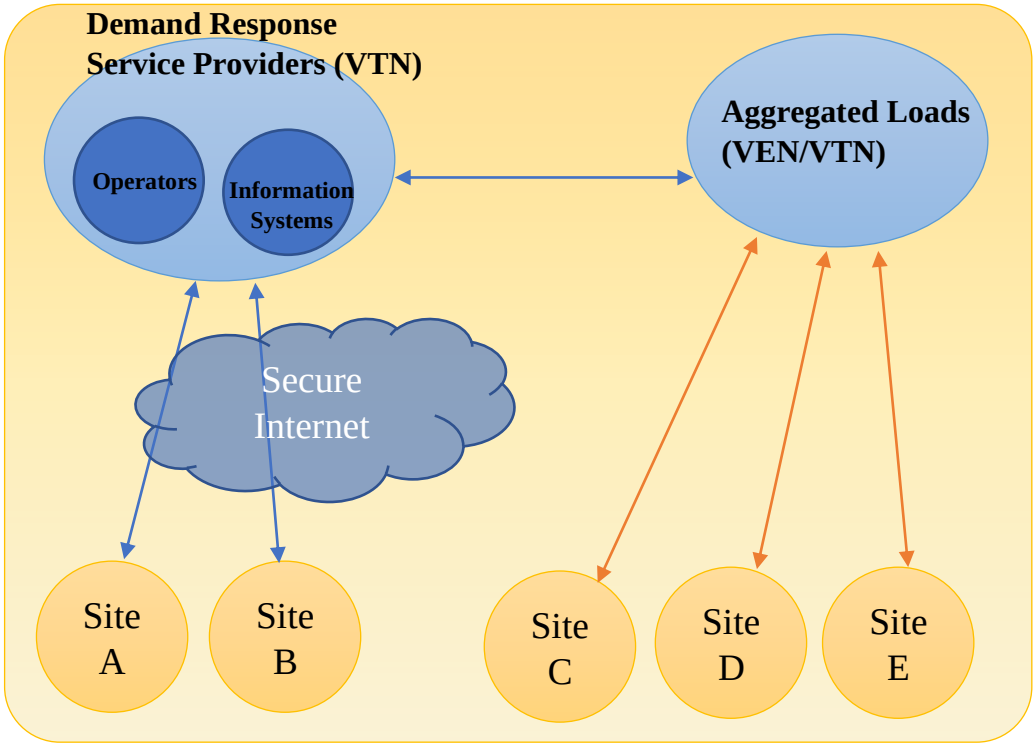


FIGURE 2.8: COMMUNICATION OVERVIEW OF OPENADR [20]

Zigbee Smart Energy Profile 2.0 (SEP) is an application to enable users to provide data to different devices in a home. This standard is the successor to Zigbee Smart Energy 1.0. SEP is designed to operate in a home area network (HAN) [21]. A common application for SEP is allowing homeowners to monitor their electricity consumption from a smart device, such as a cell phone or laptop, with a SEP-compatible smart meter, as shown in the typical layout in Figure 2.9. SEP could be utilized to create a high interoperability system inside a household and enable it to receive a large amount of data concerning the homeowner's power consumption activities. Ref. [22] demonstrates the development of middleware to allow multiple devices using SEP 2.0 to communicate with utilities and third parties about energy use.

Sunspec is an alliance of over 100 industry partners who have the intent of creating plug-and-play standards for DER and reducing system implementation cost. Its communication application is shown in Figure 2.10. It has many standards related to interoperability including Sunspec Modbus, Sunspec for IEEE 1547-2018, and others. Sunspec Modbus specifies Modbus schemas for intrasystem communication for ESSs. An example of an ESS that utilizes this is shown in [24]. This paper looks at the feasibility of Sunspec Modbus as a communication protocol, including the overall interoperability of the standard, and the cybersecurity nature of it. It concluded that Modbus is not a secure protocol, and that connection times could be an issue for this application.

Another group working on the development of plug-and-play standards is the Modular Energy Storage Architecture (MESA) Standards Alliance. This group specifies a standard communication mapping for ESSs to communicate with utility companies through SCADA systems, and specifies Modbus mapping for ESS intrasystem communication [26]. MESA's communication schemas are based on the Sunspec communication standards. MESA-ESS specifies how an ESS should interact with a utility SCADA system for control and data collection. MESA-Device defines how the different components internal to the ESS communicate with one another. MESA-Device has three subcategories with Modbus addressing: MESA-PCS for power conversion systems, MESA-Storage for energy storage devices such as batteries, and MESA-Meter for utility power meters. How these standards interconnect is shown in Figure 2.11.

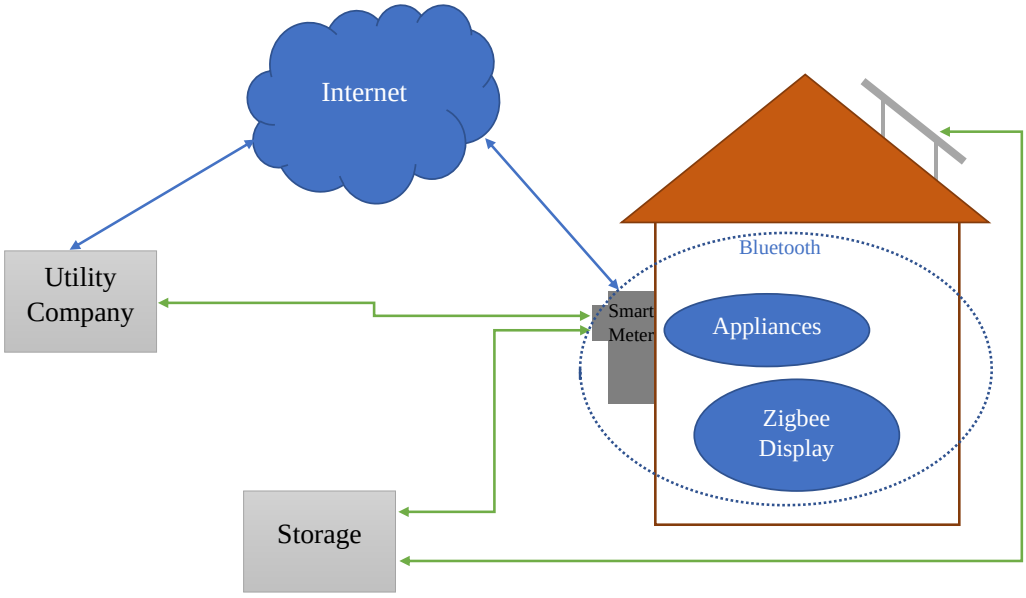


FIGURE 2.9: TYPICAL ZIGBEE SMART ENERGY PROFILE 2.0 IMPLEMENTATION [23]

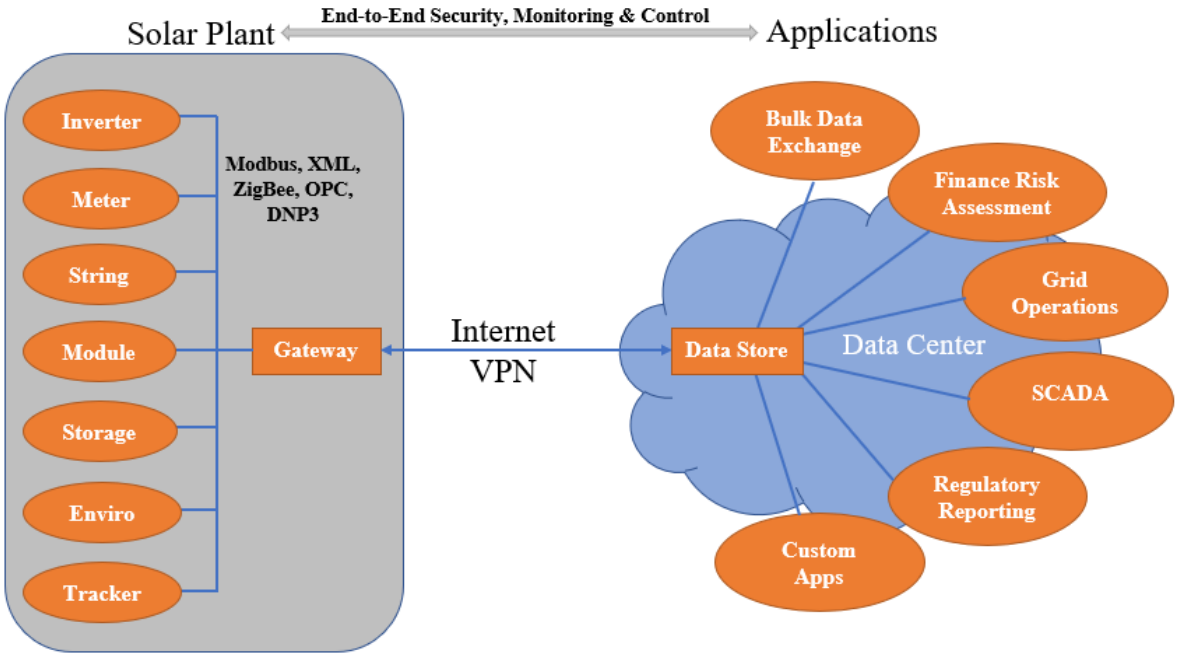


FIGURE 2.10: COMMUNICATION DIAGRAM SHOWING SUNSPEC APPLICATION [25]

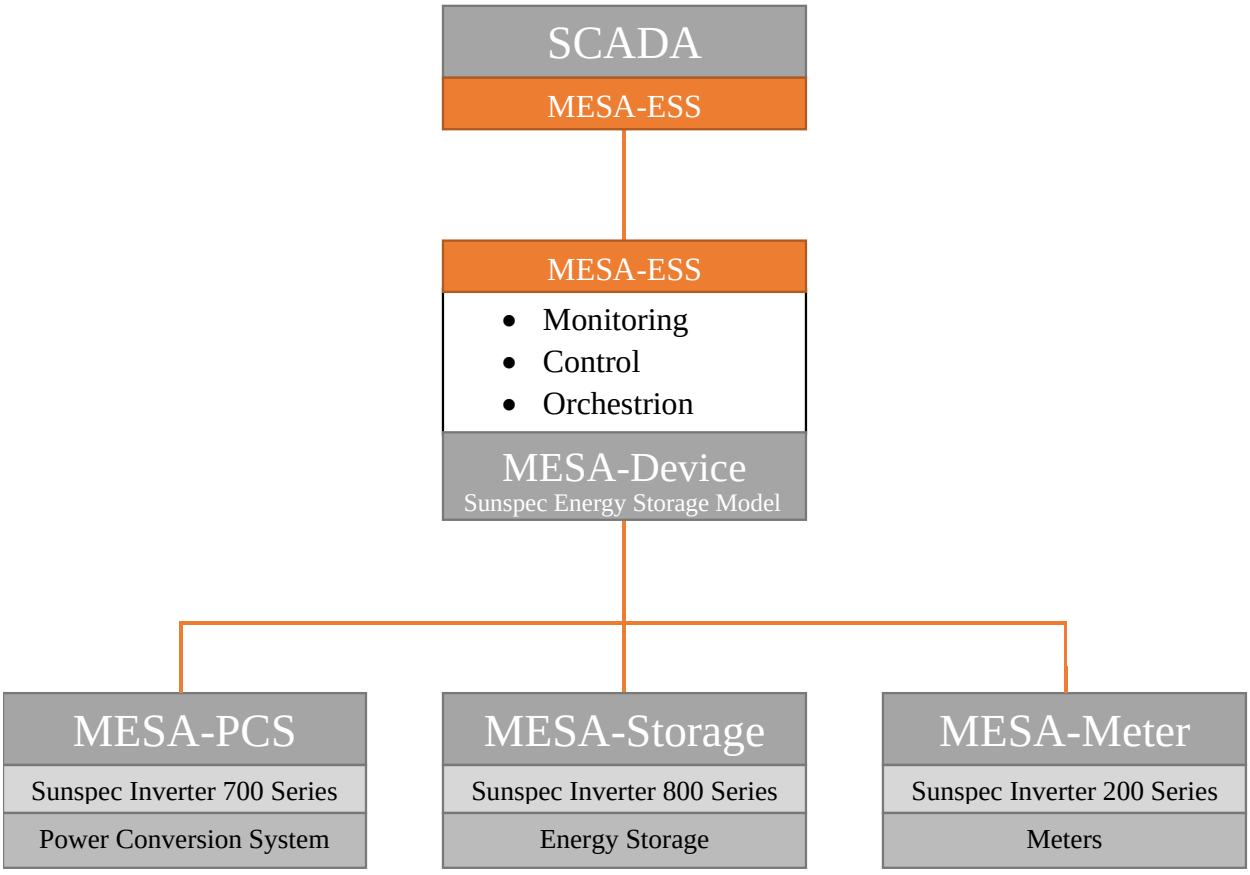


FIGURE 2.11: MESA STANDARDS DIAGRAM [26]

2.2 Power Electronic System Communication

As PESs become more advanced and complex, the communication method utilized by these systems impacts their operation. The latency, data reliability, and message rate of the communications affect the control schemes of each PES subsystem. In the following section, common communication protocols are overviewed. The two common types of communication utilized in agent-based systems are (1) agent-to-agent, also called "message bus," communication and (2) agent-to-device communication. Protocols for both types of communications are presented and overviewed.

The type of communication presented in this section is LAN communication with ethernet as the communication medium. A LAN is defined as a network that is confined to limited physical space, and typically is completely owned and controlled by the entity that utilizes the network [27]. The IEEE standard concerning LANs is IEEE 802.1, which lays out the recommended architecture, security, and management for the network [28].

2.2.1 VOLTTRON

For agent-based systems, a message bus protocol that can be utilized is VOLTTRON. Developed by PNNL, this is an agent-based environment that can be used to integrate data, devices, and systems [29]. A general layout of the architecture is shown in Figure 2.12. The purpose of this platform is to provide an environment to enable communication between devices with differing and proprietary communication protocols. It also encourages a higher level of communication between the power grid and physical devices [30]. This was demonstrated by using VOLTTRON to coordinate household energy use, including coordinating EV charging with HVAC and a maximum energy consumption limit [31]. However, the VOLTTRON platform is not limited to building energy management and has been applied in microgrids and energy storage. Any communication or control issue that can be solved with an agent-based control platform can utilize VOLTTRON.

A multi-chemistry ESS with agent-based control is presented in [32]. Both the utility interface, and the transactive interface are agent-based, and this allows for multiple battery chemistries to be incorporated into a single ESS.

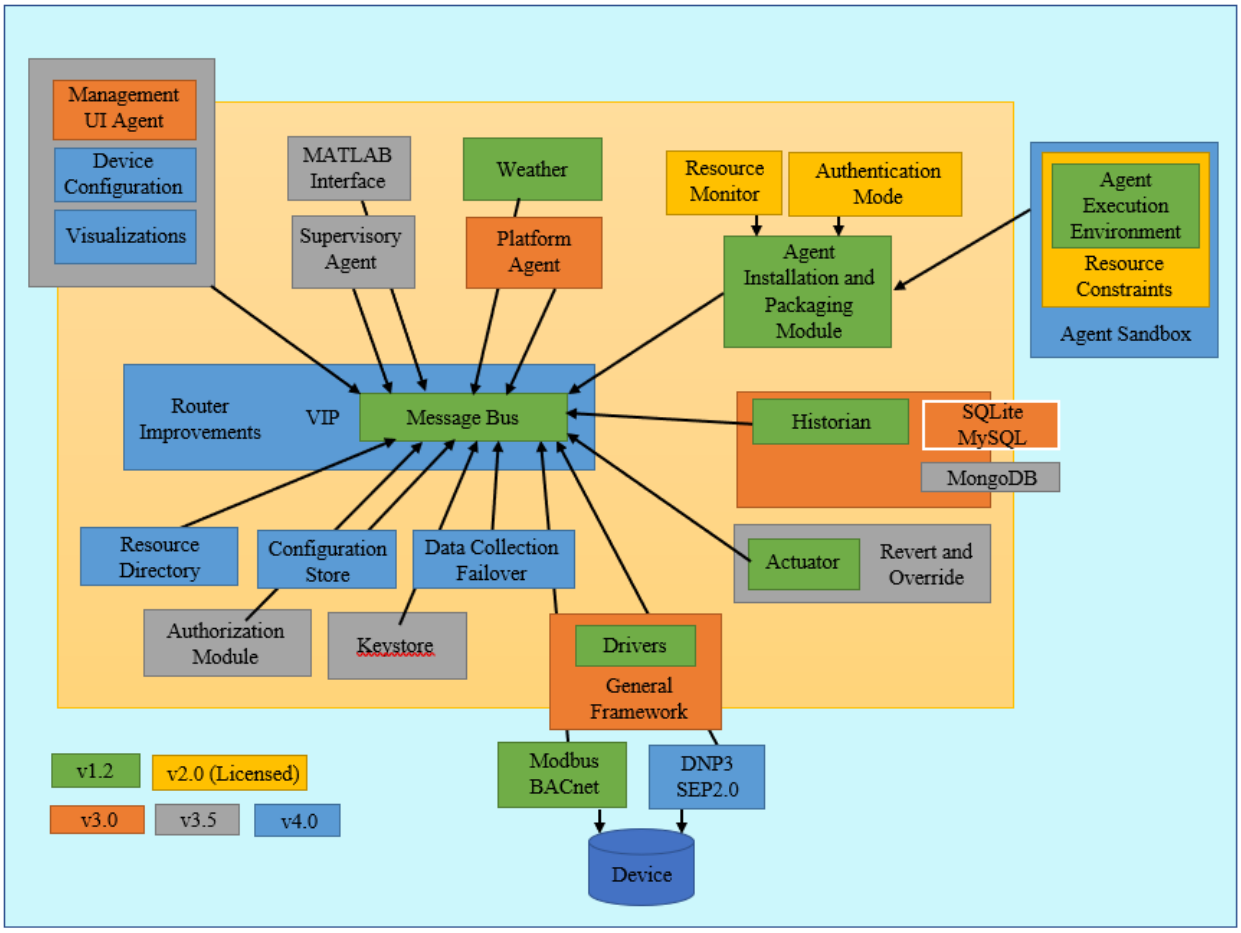


FIGURE 2.12: VOLTTRON PLATFORM [29]

Each battery is controlled by an energy storage module controller (ESMC) that contains five agents that in-turn control the BMS, the DC-DC converter, the optimizer, transactive interface, and the intelligence. The intelligence agent uses the optimal approach determined by the optimizer to make the final decisions for the ESMC's behavior. The central system controller is also agent-based and has agents that control the grid-tied inverter, intelligence, optimizer, transactive interface, and utility interface. This system, shown in Figure 2.13, allowed for each ESMC to evaluate preference and bid for energy based on the main controller's power request.

2.2.2 MQTT

A common communication protocol utilized for IoT functionality is Message Queuing Telemetry Transport (MQTT). MQTT is officially adopted as a standard by OASIS [33]. This communication protocol is a lightweight, publish-subscribe protocol that allows devices to publish messages to a message bus, and categorize these messages by topics . Devices can also subscribe to these topics to receive the published message. MQTT utilizes a broker to handle the movement of the messages. This gives MQTT the advantage of being scalable, as new devices can be easily interfaced with the broker. It also creates a secure environment, as the broker can refuse connection to insecure devices. Because each MQTT device, or "client," connects directly to the broker, the clients are unaware of each other's IP addresses [34].

MQTT has different options that allow it to be as reliable and secure as required by the user. The first security option included is username-password authentication. This requires each MQTT client to have a unique username and password to connect to the MQTT broker. If a device does not have valid credentials, the broker denies the client's connection. The username and password can also be used to limit a client's access to specific topics, allowing data within the system to be secure from different clients. MQTT also supports TSL/SSL encryption. This allows the data that is communicated through the MQTT broker to be encrypted with a "key" and for the clients to decrypt the data with the same key. When MQTT is unencrypted, the standard port utilized is 1883. When TSL/SSL encryption is used, the standard port utilized is 8883 [34]. For MQTT messages, there are three different levels of quality of service (QOS) that can be used [35]. These different levels are outlined in Table 2.1. The different QOS levels allows the PES designer to prioritize data reliability or data transmission speed.

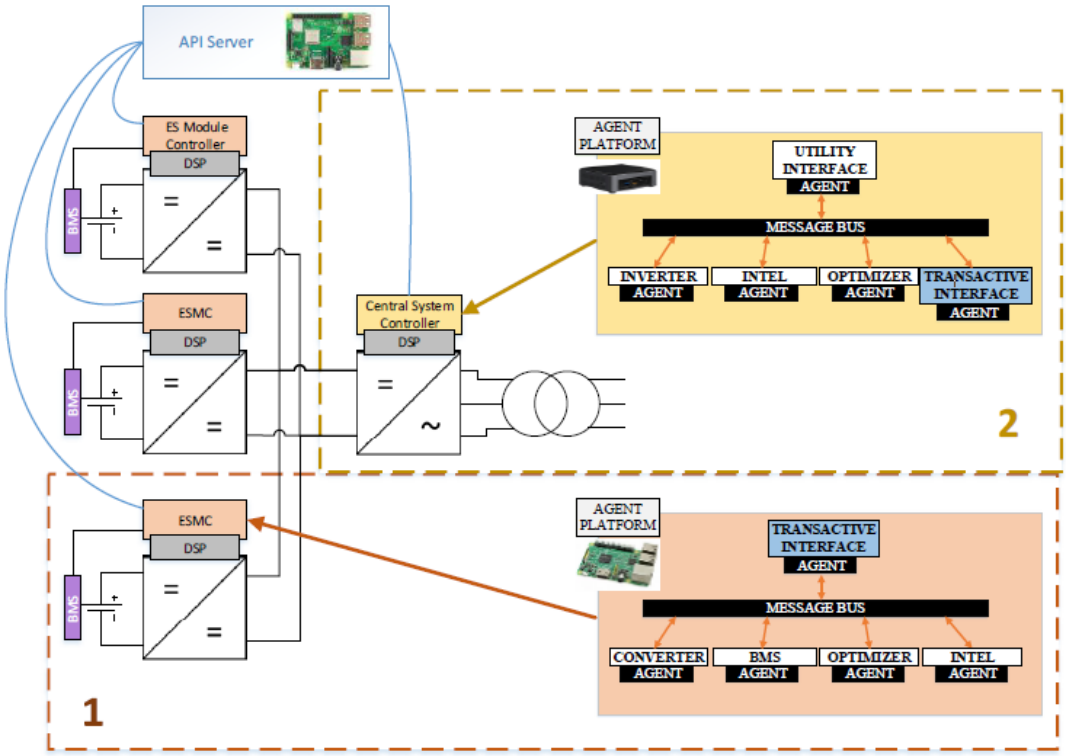


Figure 2.13: COORDINATED COMMUNICATION AND BARTERING SYSTEM FOR ENERGY STORAGE USING AGENTS [32]

TABLE 2.1: THE DIFFERENT MQTT QOS LEVELS [35]

QOS	Description
0	Message is sent, no delivery acknowledgment from broker
1	Message is sent, delivery is acknowledged by broker
2	Message is sent, delivery is acknowledged by broker. If delivery is not acknowledged, message is resent until delivery is acknowledged

2.2.3 ZeroMQ

The final message bus communication protocol covered in this section is ZeroMQ (Zero Message Queuing). This communication protocol is a socket application programming interface (API) that uses multiple different communication patterns for the message flow between devices. These include “request-reply”, and “publish-subscribe.” Request-reply allows devices to request data from another device as the data is required. Publish-subscribe allows a device to receive data that it is subscribed to as the data is published by another connected device. Unlike MQTT, ZeroMQ does not require a dedicated broker, and the devices identify each other with IP addresses and ports [36].

ZeroMQ has been used in multi-agent PES, as shown in [37]. In this system, ZeroMQ is used as the message bus protocol to facilitate communication between the converter, intelligence, battery management system, and interface agents. This allows the PES to be made of multiple different subsystems interacting with one another. Each agent serves as the interface between a subsystem and the message bus. Figure 2.14 shows the agent structure of the PES, along with the hardware and communication infrastructure. Figure 2.15 shown the agent communication.

2.2.4 Modbus

Developed in 1979 by Modicon (now Schneider Electric), Modbus is a free and open source communication protocol, not requiring royalties to be paid to Schneider. Due to this, along with its simple and straightforward format, Modbus has become an industry standard for equipment communication [38]. Modbus functions as a basic server-client communication protocol. A value is stored on the server in a register, which has a specific address. Both the client and the server can read and modify the value on the register. Figure 2.16 shows the frame structure of Modbus TCP/IP and Modbus RTU. Table 2.2 shows the typical register mapping and data types of the registers. While not required, certain ranges of registers are typically set aside for specific functions, such as read-only versus read-write, and data types, such as binary coils. Coils are the Modbus data type that only communicates “0” or “1” values, versus the 16-bit values of input and holding registers. Specifying these ranges allows values to be allocated as protected from changes by the client. For example, a BMS operating as a Modbus server needs to ensure its measurement registers cannot be changed by a client, only read.

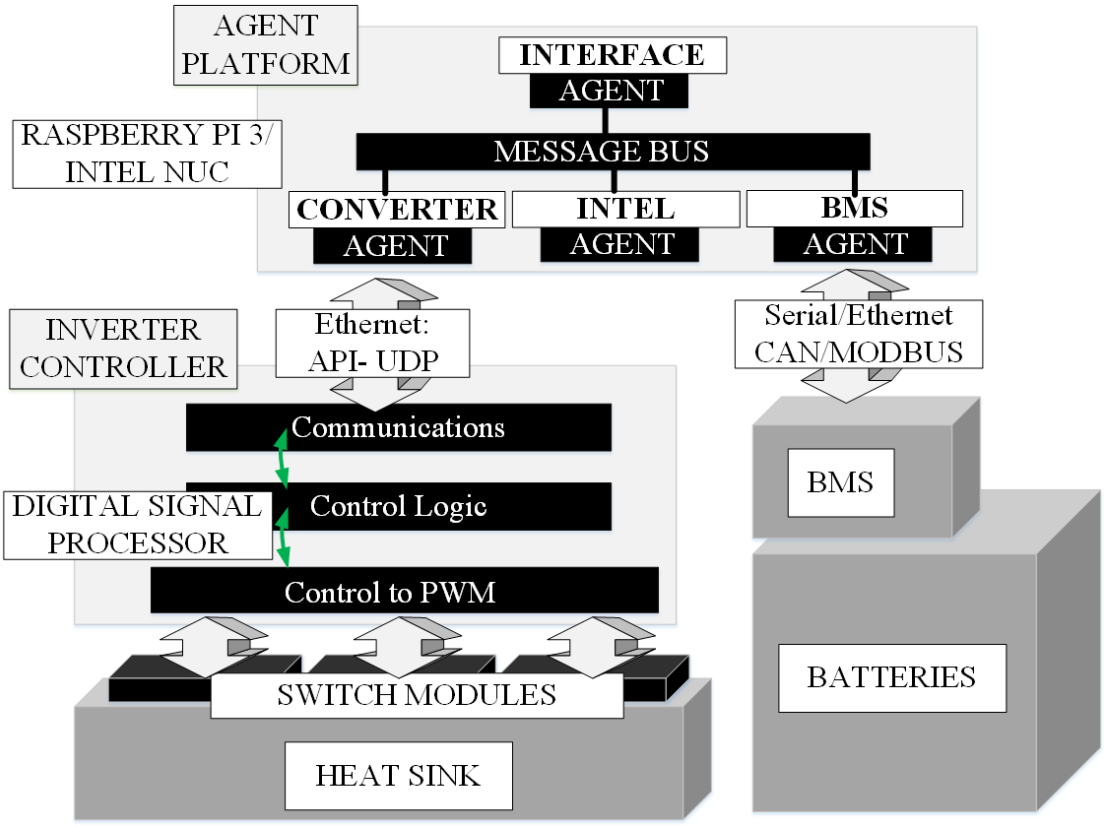


FIGURE 2.14: MULTI-AGENT PES UTILIZING ZEROMQ AS ITS MESSAGE BUS COMMUNICATION PROTOCOL [37]

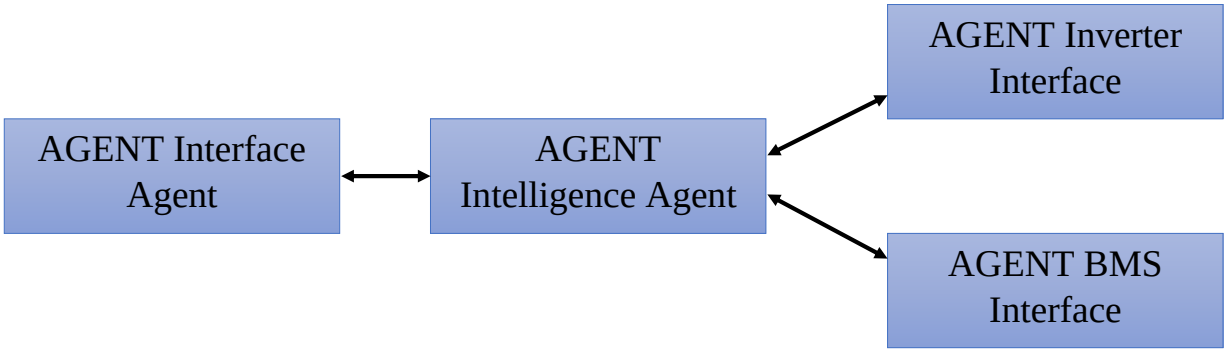


FIGURE 2.15: AGENT DATA FLOW FROM PES PRESENTED IN [37]

TABLE 2.2: THE SPECIFIED ADDRESS RANGES FOR DIFFERENT DATA TYPES IN MODBUS [39]

Coil/Register Number	Type	Table Name
1-9999	Read-Write	Discrete Output Coils
10001-19999	Read-Only	Discrete Input Contacts
30001-39999	Read-Only	Analog Input Registers
40001-49999	Read-Write	Analog Output Holding Registers

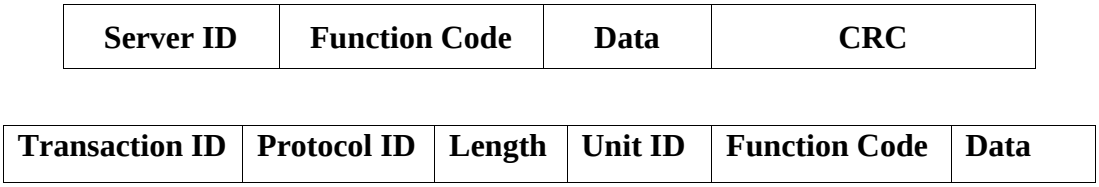


FIGURE 2.16: THE FRAME STRUCTURE FOR MODBUS RTU (TOP) AND MODBUS TCP/IP (BOTTOM) [39]

Modbus has several different implementations. Two common implementations of Modbus are Modbus TCP/IP and Modbus RTU. The basic data structure of the communication is the same, however, the communication protocol is different. Modbus TCP/IP is typically transmitted over Ethernet using the Transmission Control Protocol (TCP) and the Internet Protocol (IP). Modbus RTU typically is transmitted over a serial communication protocol, but can also be tranmitted over the User Datagram Protocol (UDP) of the IP. Modbus over UDP frames are similar to Modbus RTU frames.

For Modbus TCP/IP, an IP address and port number are required to establish a connection. The standard port for Modbus is 502. It should be noted that other Modbus standards can be used over serial communication, including Modbus ASCII. The packets are based on the American Standard Code for Information Interchange, which defines 256 different characters. There are multiple forms of serial communcation over which Modbus RTU and Modbus ASCII can be used, such as RS-232 or RS-485.

2.2.5 User Datagram Protocol

UDP is a fundamental part of the Internet Protocol suite. It is a transport layer, like TCP. The UDP packet structure is shown in Figure 2.17. However, there are numerous differences between UDP and TCP. UDP does not require any handshaking between devices, which means that the delivery of information is not guaranteed. However, this lack of handshaking and information checking enables UDP to be a faster method of intrasystem communication as compared to TCP.

Since UDP is a transport layer, it is open to using numerous different protocols. For example, Modbus can be implemented over UDP, in a form known as *Modbus over UDP*. It also allows custom protocols to be easily implemented with concern for handshaking or data checking. Python has a library called *socket*, which allows the user to create a network socket. The type of socket that is used by UDP is called a datagram socket. These sockets can be used to send UDP packets for peer-to-peer communication. This UDP communication is a stream of bytes, which are encoded on the sending side of the communication and decoded on the receiving side.

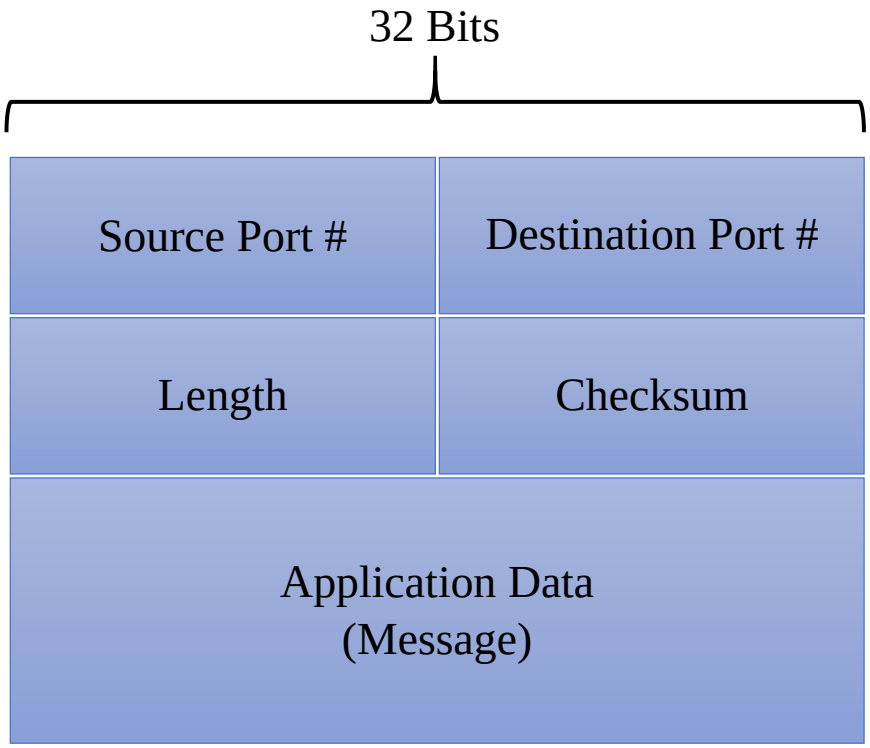


FIGURE 2.17: UDP PACKET STRUCTURE [40]

2.3 Communication Testbeds

This section covers previously implemented communication testbeds. The purpose of this is to explore how these testbeds are implemented, what network architecture they consist of, and the types of communication tested. In general, this review is limited to testbeds with LAN implementations, as opposed to wide-area-network (WAN) or serial communication such as RS-232 or RS-485. LANs are important in PES intrasystem communication, as most of the modern communication protocols are implemented over them. Also, the reviewed testbeds mainly consist of single-board computers with low processing power as nodes. These types of devices are advantageous for the development and deployment of PESs due to their versatility and low cost.

2.3.1 Supervisory Control and Data Acquisition Testbed for Research and Education

SCADA systems are used for the control and monitoring of many industrial control systems (ICS). These systems are used extensively in power grid operations. These systems typically utilize network communication, as SCADA systems are intended to be used remotely. These systems have a variety of communication protocols used, including Modbus. Due to SCADA systems use in industrial applications, they are a target for cyber-attacks.

A testbed with a custom SCADA implementation, shown in Figure 2.18, is presented in [38]. The purpose of this testbed is to assess Modbus-based SCADA system cyber vulnerability. Modbus TCP/IP is a common and easy to implement communication protocol due to its simplicity and open-source availability; however, it has numerous inherent security issues. These include a lack of authentication between devices, no privilege management, and no data encryption. Because of this, any malicious device on the SCADA network can read or write any data on the system. This poses a security risk for Modbus-based SCADA systems.

This testbed is designed to emulate a real SCADA system. It contains a Human Machine Interface (HMI) that communicates with a Programable Logic Controller (PLC). These PLCs control two modeled systems, a nuclear power plant and a train. The user can use the HMI to control various functions on the testbed, such as turning on and off the power plant, and controlling the speed and direction of the train.

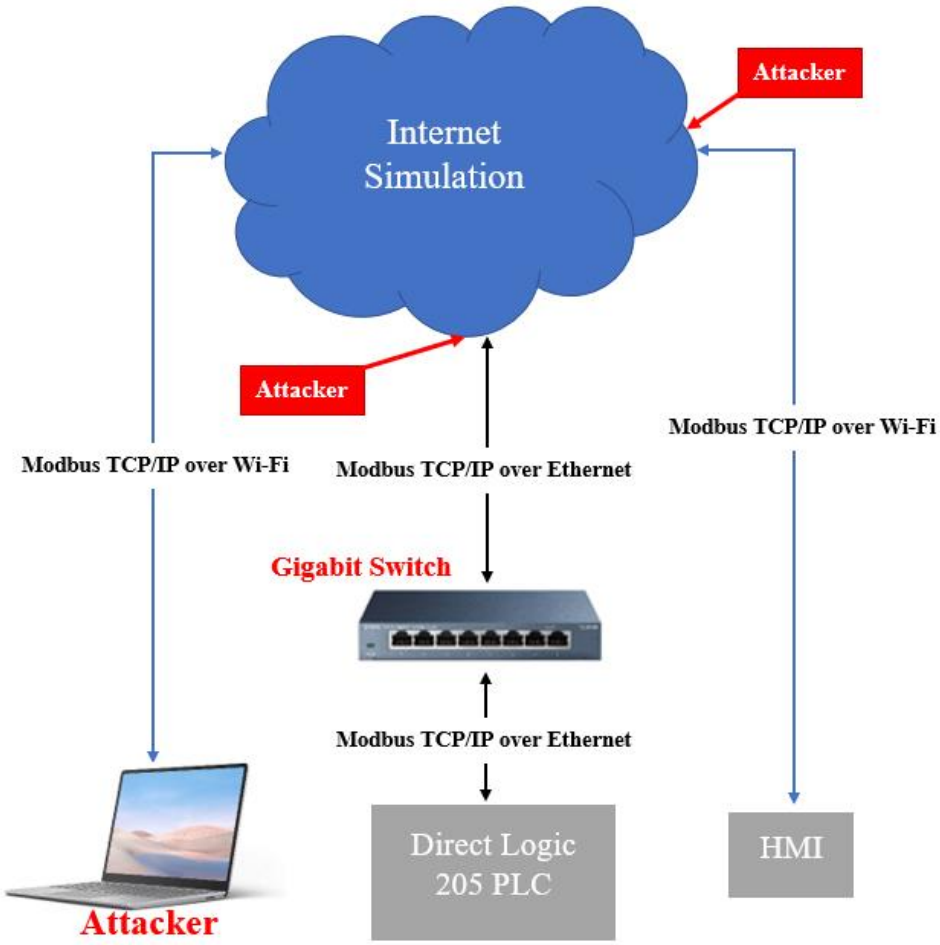


FIGURE 2.18: THE NETWORK ARCHITECTURE OF THE SCADA TEST WITH THE VULNERABLE POINTS INDICATED [38]

The demonstration of cyber-attacks on this testbed are shown in [41]. Different tools are used to analyze and attack the network. *Wireshark* is used to see TCP/IP packets, *Ettercap* is used to execute man-in-the-middle attacks, and *Metasploit* is a penetration tool that allows users to find, exploit, and validate cyber vulnerabilities. In the attacks performed, the IP addresses and MACs are found with network scanning tools, and the data can be read as well. Modbus's lack of encryption allows for the plain text values to be viewed by an attacker.

Three different attacks were executed on this testbed. They were data manipulation, man-in-the-middle, and denial of service attacks. During each of these attacks, the operation of the test was disastrously interrupted, or changed without the user being able to regain control.

2.3.2 Development of a Smart-Grid Cyber-Physical Systems Testbed

As power systems become more complex, their controls become more sophisticated. The devices in these smart grids utilize more and more information to make their control decisions. Understanding these cyber interactions is important for the deployment of these systems. In [42], a testbed, shown in Figure 2.19, is presented to evaluate these different control technologies through the use of real-time simulations and a controller hardware-in-the-loop setup.

This testbed uses a real-time simulator (RTS) to simulate the power grid. This real-time simulator provides a realistic simulation of the power grid, making the testbed simulation of the cyber system more accurate. The testbed also has six Versalogic Corporation "Mamba" board computers, and one Technologic Systems TS-7800 board computer. The Mamba board allows for fast prototyping, while the TS-7800, with its lower computational power, represents the expected processing platform of a typical PEC. All the board computers and the RTS are connected to one another through a LAN using network switches. The RTS could not directly connect with ethernet, so it interfaces with the network through an FPGA.

With this testbed, a microgrid communications architecture was tested. This architecture, called Future Renewable Electric Energy Delivery and Management (FREEDM), has extensive data communication. Some loads and lines have controls, and the generation is typically renewable. This smart grid setup uses a broker that coordinates the smart grid processes of group management, load balancing, fault detection, and state collection.

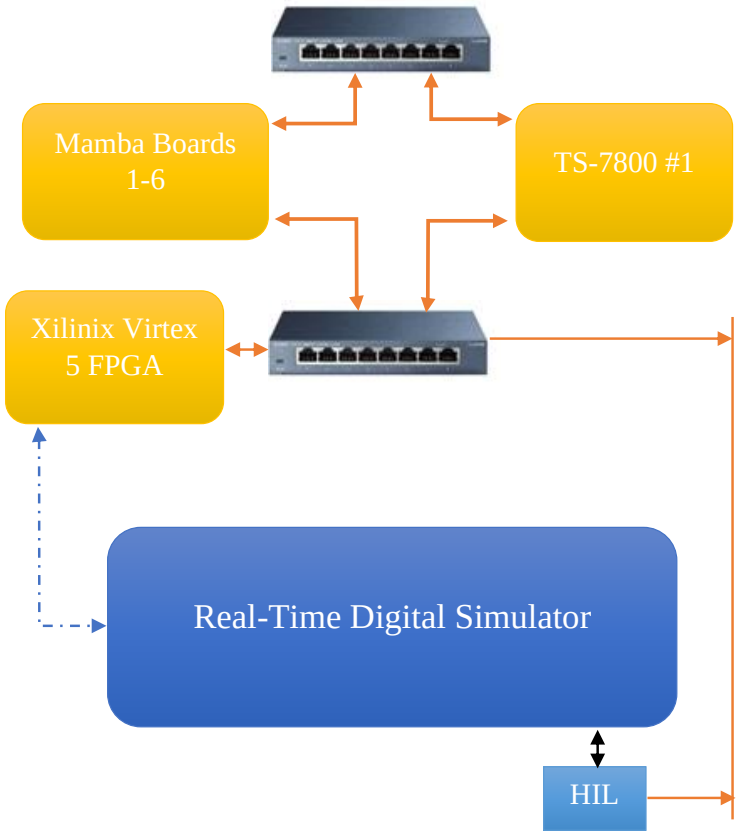


FIGURE 2.19: TESTBED SETUP PRESENTED IN [42] SHOWING THE NETWORKED CONNECTIONS OF THE COMPUTER NODES

The real data communication was able to be observed with this testbed. Real communication obstacles, such as time delay and dropped packets, were able to be incorporated into the smart grid simulation, allowing it to operate closer to the actual smart grid operation. With this testbed, the command delays between the broker and devices were observed. These communication challenges gave insight into the target power values not converging as fast as expected. Future smart grid control developments will be able to account for communication specifications.

2.4 Chapter Summary

In this chapter, the environment of PES communication was explored. Different standards and implementation of advanced, high interoperability ESSs were discussed. Standards for PES intersystem communication, communication with the outside world, were overviewed. Furthermore, intrasystem communication was also reviewed. This included overviewing common network communication protocols. Last, previously built communication testbeds were reviewed. This further showcased how testing communication is important in PES and gave insight to the current communication testing setups.

Chapter 3: PES Communication Testbed

To evaluate and characterize the different communication implementations in agent-based PES, a platform to deploy code from a central controller, emulate the behavior of the PES, and evaluate the performance of the communications was developed. This platform must be able to closely replicate the communication environment of an agent-based PES. Furthermore, the measurement method must not interfere with the communication performance.

To accomplish this, a power electronics communication testbed was developed. This testbed is pictured in Figure 3.1. This testbed utilizes custom hardware and software that closely emulates the communication emulation of an agent-based PES and provides an isolated framework to characterize different communication protocols and schema. The hardware and software are specifically designed to interact with one another to emulate PES communication while collecting data. The hardware connections are shown in Figure 3.2.

To quantify the communication implementations, latency, error rate, and messages per second are determined. This allows for the comparison of a reliable communication protocol, such as TCP/IP, which is slower but ensures message data integrity, to a faster, but more unreliable communication protocol, like UDP, which does not ensure message data integrity. Furthermore, it allows for comparison of different communication standards over these protocols, such as Modbus over UDP verses a custom communication standard.

3.1 Hardware

The hardware used in the power electronics communication testbed is listed below:

- Raspberry Pi 3B (3) – Agent nodes
- Raspberry Pi 3B (3) – DSP Simulator nodes
- Raspberry Pi 3B (1) – MQTT Broker
- Arduino Uno
- Network Switch (2)
- MacBook Pro running Ubuntu 16.04 (Control Computer)
- USB-to-ethernet adapter (12)

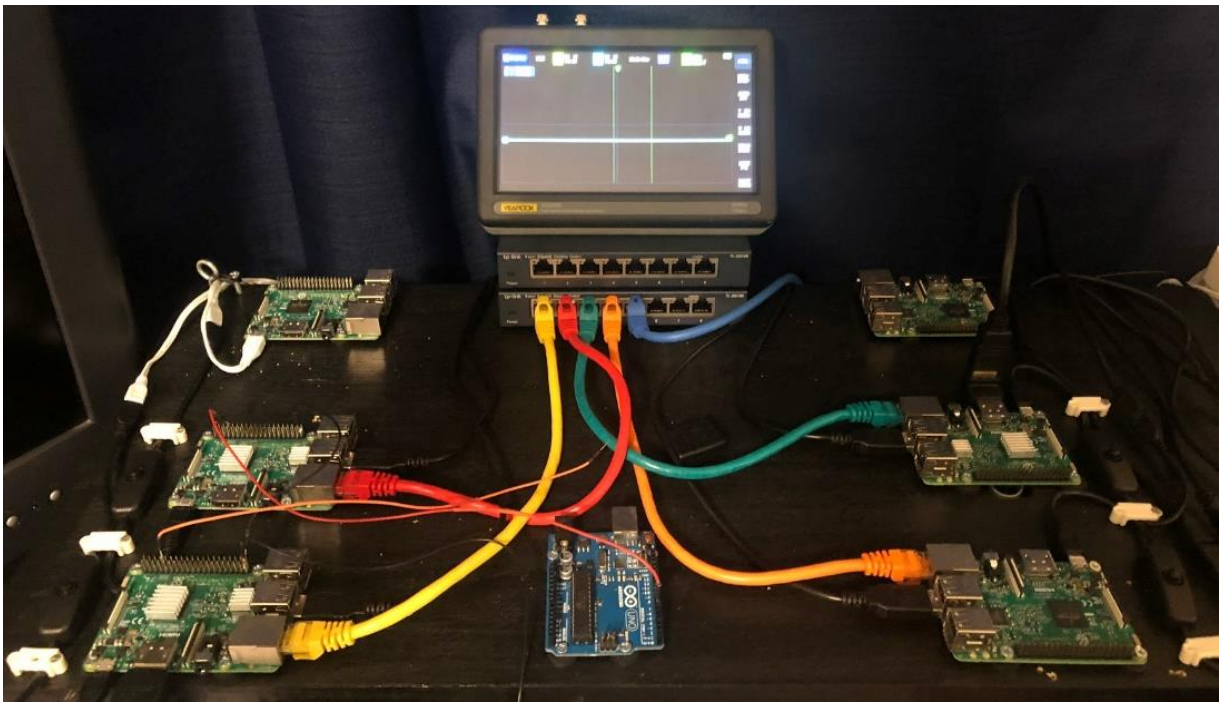


FIGURE 3.1: PHOTO OF THE POWER ELECTRONICS COMMUNICATIONS TESTBED [2]

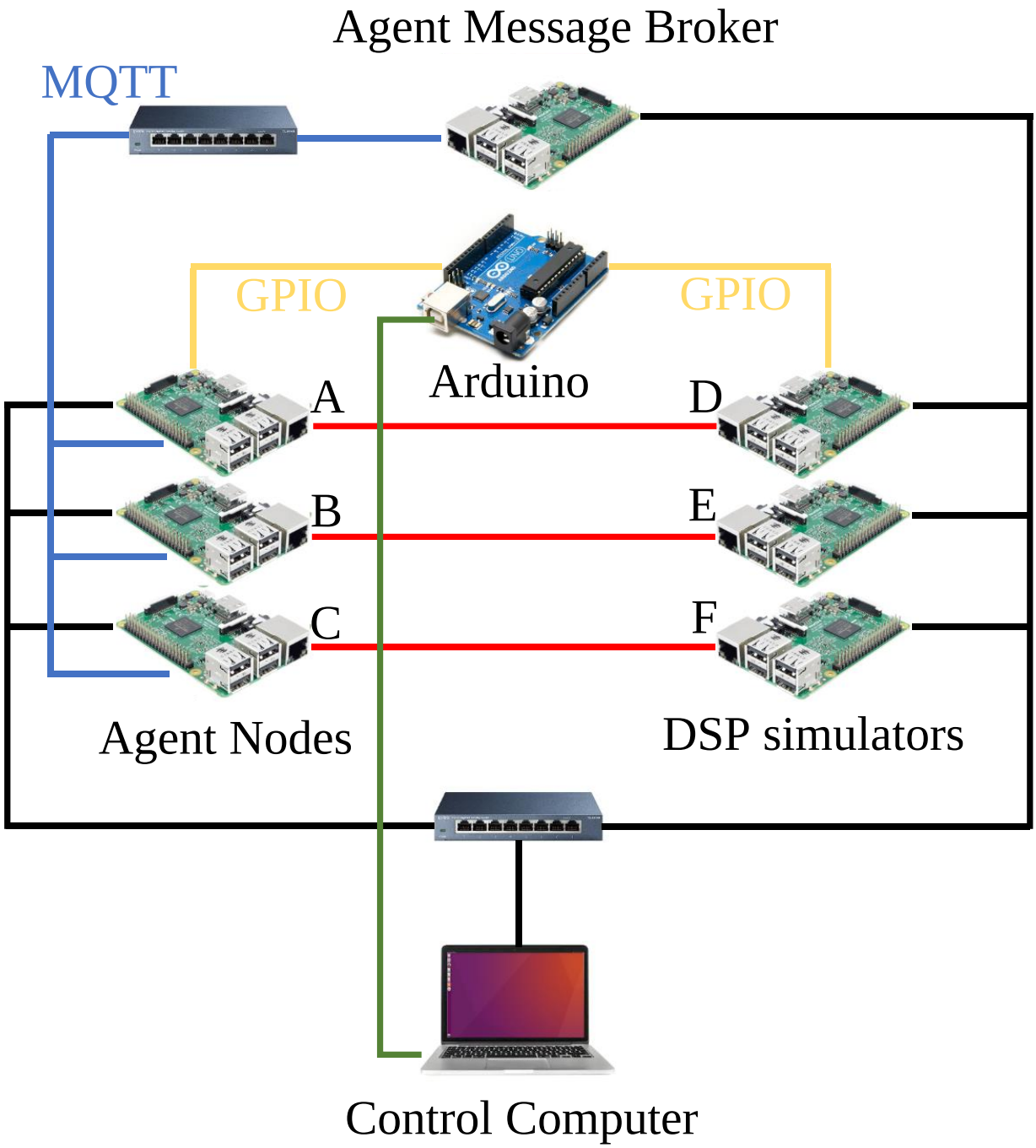


FIGURE 3.2: DIAGRAM OF THE SETUP OF THE COMMUNICATIONS TESTBED [2]

The testbed uses three Raspberry Pi's as agent nodes, labeled in Figure 3.2 as A,B, and C. These nodes interface with each other through the agent network, shown as the blue lines in Figure 3.2. They also individually interface with a DSP simulator node, labeled as D, E, and F in Figure 3.2, through separate, isolated networks called the DSP networks, shown as the red lines in Figure 3.2. All Raspberry Pi's in the testbed connect to the Control Computer through the control network, shown as the black lines in Figure 3.2. All the networks are physically isolated from one another, which allows for faster communication because the networks will not become saturated with traffic that is nonessential to the communication protocols, such as control commands or VNC access.

Additional networking hardware was required to interface all the Raspberry Pi's and Control Computer to one another. The Raspberry Pi 3B only has one onboard ethernet port, so to accommodate the two additional ethernet ports required, each Raspberry Pi used two USB-to-ethernet adapters. For the agent and control networks, which have more than two devices, a network switch was needed to connect multiple devices to the network.

The power electronics communication testbed needed an effective and versatile way to measure network latency. Typically, this would require clock synchronization between the Raspberry Pi's, and reading timestamps of when packets are sent and received. However, a simpler method was developed using the general-purpose input/output (GPIO) pins of the Raspberry Pi's, shown in Figure 3.2 as the yellow lines. Each Raspberry Pi 3B has 40 pins, 28 of which are GPIO pins that can read or write HIGH or LOW states, with the HIGH state corresponding to 3.3V and the LOW state corresponding to 0V. These GPIO pins can be controlled with Python through the GPIO library. Incorporated into the communication code were commands to set a GPIO pin high on one Raspberry Pi when a packet was sent, and to set a GPIO pin high on the receiving Raspberry Pi when the message was received. The time difference in the pulses was measured with an Arduino and written to the Control Computer through serial communication, shown in Figure 3.2 as the green line. To verify the accuracy of the Arduino, a Yearpook ADS1013D oscilloscope was used.

Other hardware used in the testbed included a USB switch with a keyboard and mouse attached that allows for quick control of all Raspberry Pi's, and two computer monitors. These

allow for implementing additional software designed outside the testbed software. The Raspberry Pi's can be used as standard computers running this software and continue to interact with the rest of the system.

3.2 Network Configuration

Utilizing Raspberry Pi's for networking allows for flexible network implementation. Defining static IP addresses and adding multiple networks is easily done through simple startup commands. The Raspberry Pi 3B has a 10/100 Mbit/s ethernet port onboard [43]. As previously mentioned, the testbed uses three different network types for its communications:

- Control Network
- Agent Network
- DSP Networks

By separating these networks, the testbed closely emulates the network implementation of an agent-based PES and allows the different communications to be isolated from one another. This is achieved by using different numerical labeling on the different IP addresses, and using a netmask to ensure the different communication types cannot communicate with one another. There is also a physical separation, as the different networks do not share network switches. Each network has a defined static IP address, and the Raspberry Pi automatically adopts a naming convention of *eth0, eth1, eth2, etc.* for the detected available ethernet ports on the device. The network configuration is shown in Table 3.1.

The control network is responsible for facilitating tasks without interfering with the agent network or DSP network. These tasks include detecting connected Raspberry Pi's, transferring files to the Raspberry Pi's, executing Python scripts on the Raspberry Pi's, and allowing for remote access to the Raspberry Pi's. All these tasks are handled by this network because congestion is not a concern, since none of the tasks done over this network is quantified.

Detecting connected Raspberry Pi's is done by *pinging* a range of IP addresses on the control network and saving the addresses that respond. These saved IP addresses are loaded into the testbed application as active connection points. This ensures that the testbed does not interact with unconnected Raspberry Pi's.

TABLE 3.1: THE NETWORKS' CONFIGURATIONS [2]

Network	Raspberry Pi Name	IP Address	Netmask
DSP Networks	'eth0'	192.168.53.X	255.255.255.0
Control Network	'eth1'	192.168.0.X	255.255.255.0
Agent Network	'eth2'	192.168.99.X	255.255.255.0

The Networked File System (NFS) allows for files to be easily transferred between the different Raspberry Pi's and the Control Computer. The Control Computer acts as the server hosting the files, and each Raspberry Pi is configured to be a client and given access to the Control Computer's file system. By hosting the testbed files in a central location, they can be easily edited from the Control Computer, and each Raspberry Pi always is running the same version of each file.

The execution of Python scripts is done through secure shell (SSH) commands. SSH commands allow one computer to securely access another computer over an unsecure network and execute different commands. The testbed interface sends an SSH command to a Raspberry Pi, telling it which Python script to run, and which command line options need to be used. Scripts can be executed with both Python 2 and 3 and can be run with "super user" (sudo) permission. Having Python 2 capabilities allows for older, legacy code to still be executed through the testbed, however, development of new software in Python 2 is discouraged due to its lack of support from the Python Software Foundation.

The last task handled by the control network is Virtual Network Computing (VNC). This allows the user to remotely access a Raspberry Pi and debug code, access errors, or execute code outside the testbed system. By executing code over VNC through standard terminals, as opposed to executing code through the testbed interface, the terminal errors can be read and assessed. This also allows the file systems of the Raspberry Pi's to be accessed and changed.

3.3 Software

Software was written to handle the different communication protocols, and to coordinate the execution of the communication tests and data collection. The communication software emulates a certain component in an agent-based PES. The testbed interface controls the communication software and enables the user to quickly run communications tests and ensures the network setup of the testbed is running correctly.

The language chosen to write this software was Python 3. This was chosen due to Python 3's quick prototyping capabilities, versatile platform compatibility, and extensive library selection. To enable the user to interact with the software, the Python library Tkinter was used to

create a Graphical User Interface (GUI) for each piece of software, shown in Figure 3.3. It allows for quick implementation of GUI objects such as text entries, buttons, and menus. This user-friendly interface is paramount for quick and effective communication prototyping.

Three types of programs were written for this testbed. The first was the Communications Testbed Interface. This interface controls and coordinates the actions of all the Raspberry Pi's through the control network. It also collects serial data from the Arduino. The second type of software written was DSP emulation software. These programs emulate the DSP side of the communication. They allow the user to quickly specify communication settings and schemas. For the scope of this thesis, two communication protocols had software written, Modbus and a custom UDP protocol. The last type of software written was the Agent interface. This software communicates with the DSP emulators, and publishes data to the message bus. For the scope of this thesis, MQTT was used as the message bus software.

3.3.1 Testbed Interface

The testbed interface is responsible for configuring the hardware in the system, executing communication software, and collecting the measured data. A flow chart of its operation is shown in Figure 3.4. Upon startup, the Control Network is pinged to determine the active Raspberry Pi's on the network. These active Raspberry Pi's are loaded into the GUI. The user then assigns a role to each Raspberry Pi, such as agent, DSP emulator, or MQTT broker. Then for the agents and DSP emulators, a communication profile, which details the settings and schema to be used, is selected. The available profiles are stored in the NFS server directory on the Control Computer. The user can also specify whether to use the GUI. When the profiles have been set for each Raspberry Pi, the user tells the Testbed Interface to execute the files. The Testbed Interface then sends SSH commands to the Raspberry Pi's, telling them which communication software and profile to use. The code is run on the Raspberry Pi through a virtual terminal program, called "LX Terminal." By using a virtual terminal, the user can read the terminal outputs from the communication software by accessing the Raspberry Pi through VNC. This can be done any time after the active Raspberry Pi's are loaded into the testbed interface.

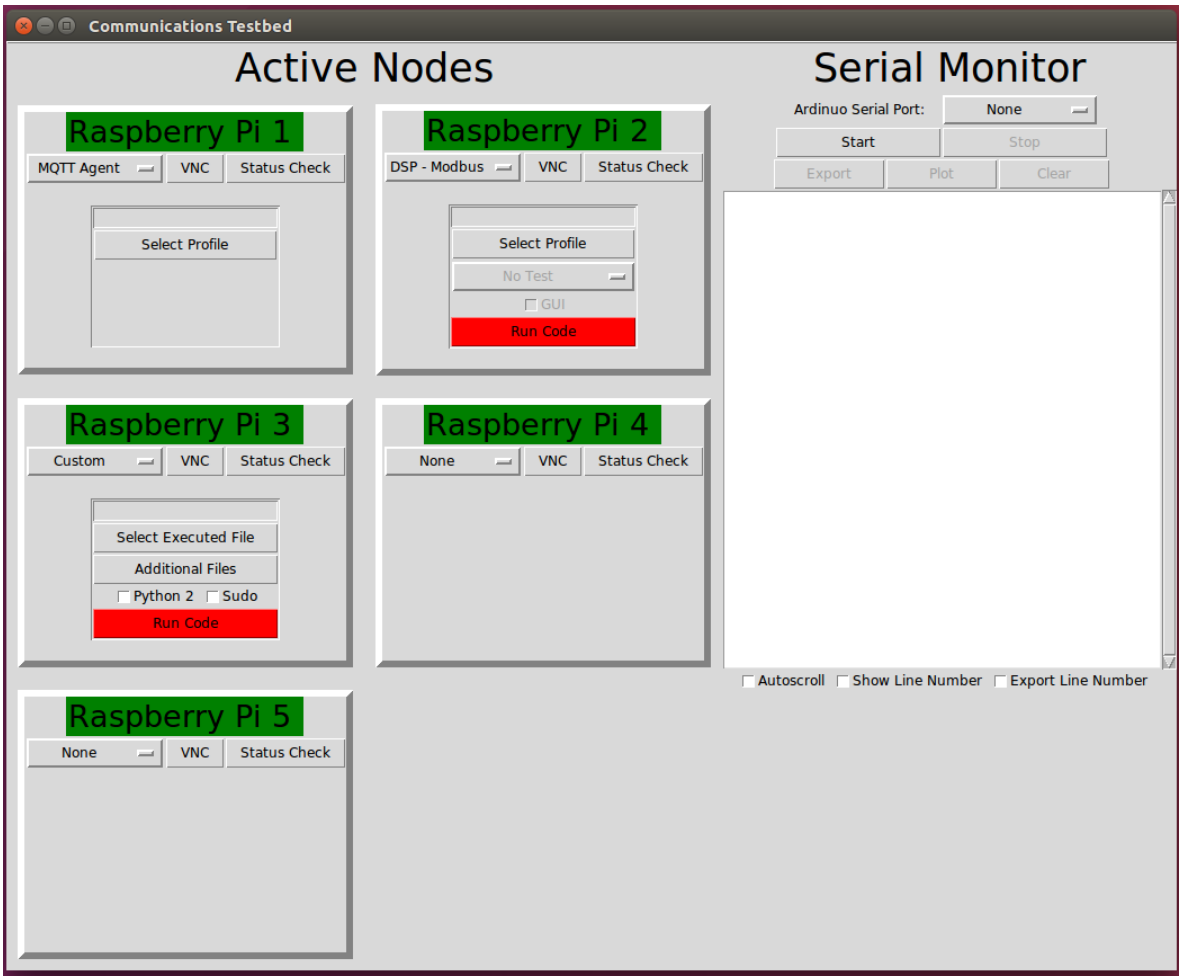


FIGURE 3.3: COMMUNICATIONS TESTBED INTERFACE GUI

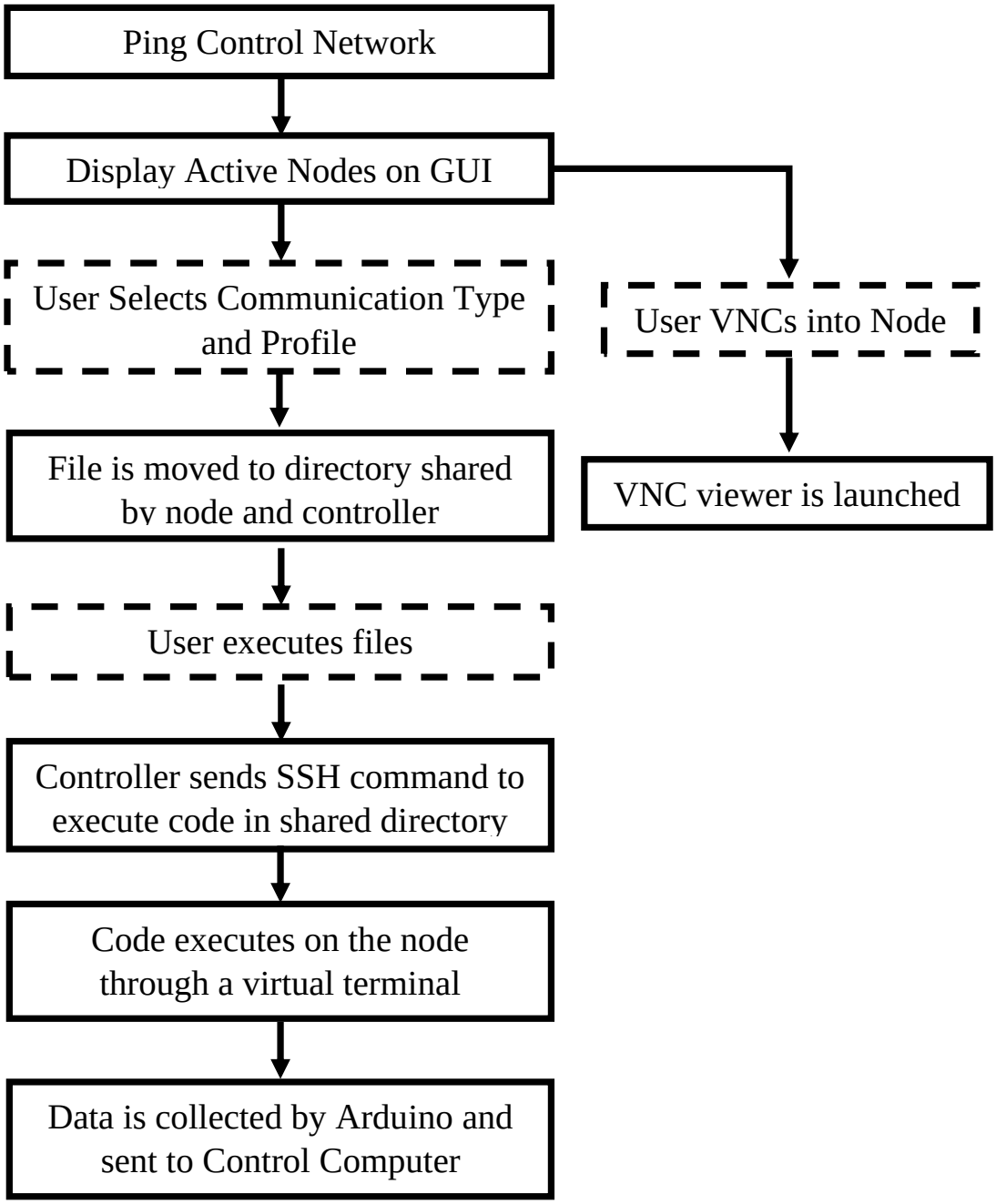


FIGURE 3.4: FLOW DIAGRAM OF THE TESTBED INTERFACE OPERATION

3.3.2 Communications Software

Software was written in Python to emulate the communication implementation of DSPs in PESs. These pieces of software allow the user to define a custom schema, define the protocol options, and run the communication implementations. By using Raspberry Pi's to emulate the DSPs, communications compatibility can be quickly established without slow reprogramming of DSPs.

The communication schemas and settings can be saved and opened as JSON files. Saving profiles is important for auto-launching the DSP communication emulation, as the user is not able to edit communication settings or schemas when the DSP emulator is auto-launched without a GUI. This allows the user to define a schema for testing and save it to the Control Computer NFS server. Then, the schema profile can be run for testing in conjunction with other PES components, and the profile can be characterized based on how it would operate in an agent-based PES. This data flow can be seen in Figure 3.5. The user can also interact with communications software through the terminal command line. The command line interface can be seen in Figure 3.6, where the *help* function is called, showing the user the program's command line options.

Due to the lower computational capabilities of a Raspberry Pi 3B compared to a standard desktop computer, the communication emulation needed to be structured in a computationally efficient manner. The main threat to faster computation is utilizing multiple Python threads and having inefficient data storage. In Python, threading allows multiple functions to be performed simultaneously, however, this slows down Python and results in slower communication.

To ensure computational efficiency, a standardized data structure was incorporated for all communication software. Setting data, GUI objects, and communication functions were stored in the Main class, and schema data and GUI objects were each stored in their own Data class. A list of data objects was declared in the main class, and the communication software can quickly cycle through the data for GUI display or sending and receiving of communication data. This was done to reduce the number of active threads that the emulators required.

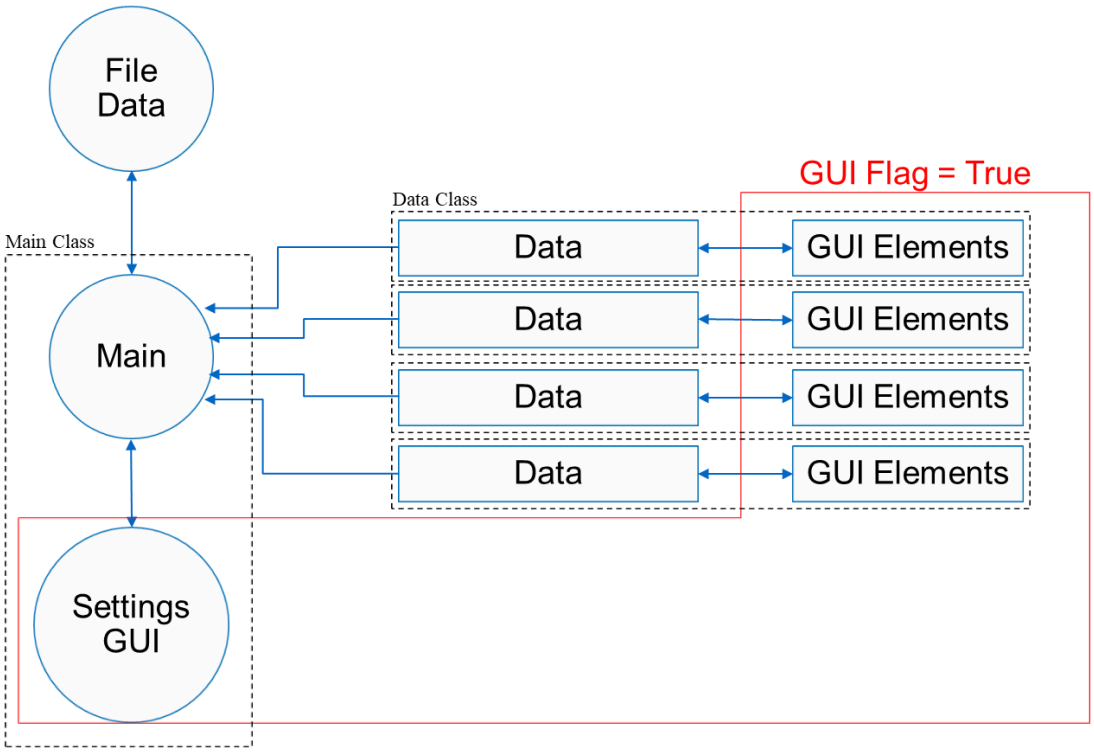


FIGURE 3.5: DATA FLOW OF THE COMMUNICATION SOFTWARE

```
linuxbookpro@linuxbookpro:~/Desktop/MQTTCode7$ python3 ModbusServerGuiV2.py -h
usage: ModbusServerGuiV2.py [-h] [-f FILE] [-g Gui]

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  Filename to open Agent Controller with
  -g Gui, --gui Gui     No to disable GUI
```

FIGURE 3.6: COMMAND LINE HELP FUNCTION OF THE MODBUS EMULATOR

To have communications running and an interactive GUI, multiple threads must be used. To closely emulate actual PES communication, the communication programs are programed to run with or without the GUI being active, as the computational nodes on these systems would not be running GUIs. By doing this, the number of required threads in the communication software is reduced by one. The user can disable the GUI by using a command line argument to set a GUI flag to false. Since the program needs to run with or without a GUI, the data flow does not require the front-end GUI for storing values, and the terminal is used to show pertinent information.

3.2.1.1 Modbus:

The first communications software written was for Modbus, shown in Figure 3.7. The Modbus program sets up a Modbus server on the DSP emulator. It can implement Modbus TCP/IP, Modbus RTU, and Modbus over UDP. To implement the Modbus communication, the python library PyModbus is used. This library allows for a Modbus server or client to be implemented through Python. It includes the necessary functions to read and write to Modbus registers, and can do so with coils (the binary Modbus data type), discrete inputs, input registers, and holding registers. Furthermore, this library includes a function to split 32-bit values into two 16-bit values and read and write them to registers. The library can also re-encode and interpret the two 16-bit values as the original 32-bit value after it is written. This is necessary for communicating 32-bit values through Modbus, as a Modbus register can only hold 16 bits of data.

The GUI allows the user to select the type of Modbus communication. If TCP/IP or UDP is selected, then the user specifies the IP address and port for the Modbus server to use. Because the DSP emulator is configured to be a server, the IP address should be the Raspberry Pi's DSP communication IP address (192.168.53.X). The standard port for Modbus communication is 502, however, other ports can be specified. Another option that can be chosen is the use of a protocol framer. In Modbus, a framer allows for the decoupling of transport and protocol. Therefore, the RTU protocol can be transported over TCP/IP or UDP. This option was not used in the scope of this thesis.

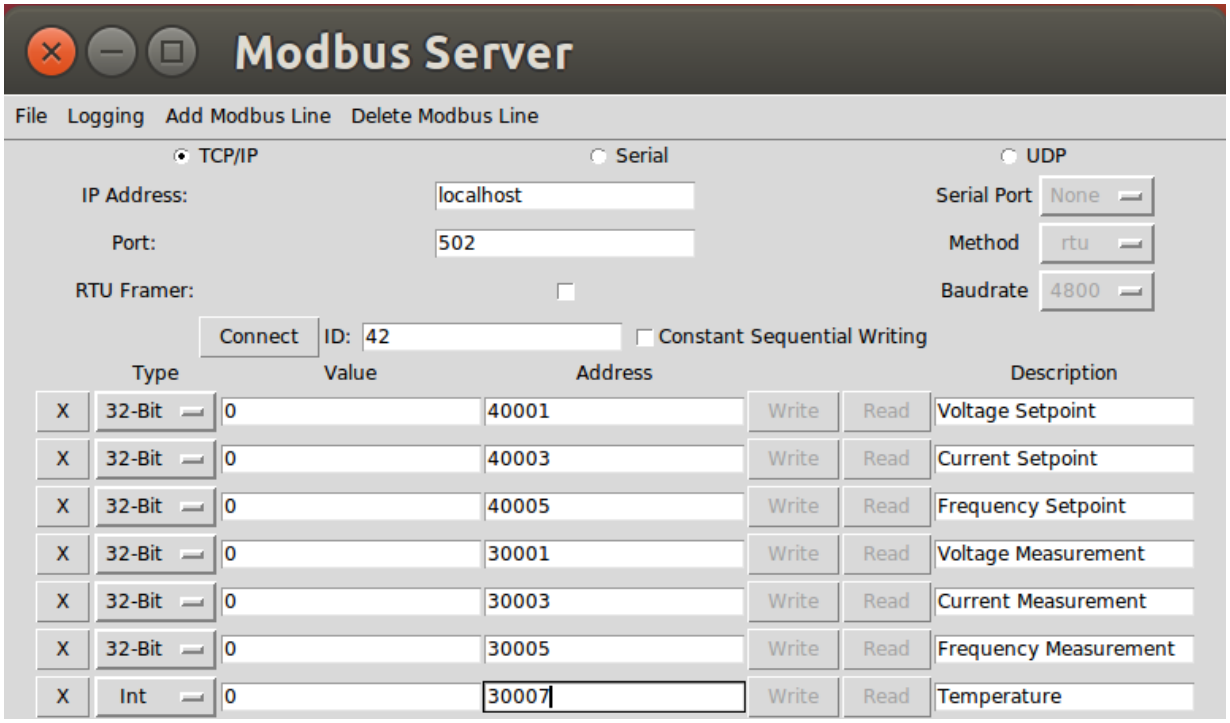


FIGURE 3.7: THE GUI OF THE MODBUS SERVER PROGRAM

The user can specify the schema for the DSP emulator to use. For each Modbus register, an address and value need to be specified. A description of the register can also be added. The "Add Modbus Line" option adds a Modbus entry to the end of the list, and the "Delete Modbus Line" option deletes the last Modbus entry in the list. Specific lines can be deleted by using the "X" button.

After the schema and Modbus settings are input, the server can be connected. Connection of a PyModbus server requires super user permission. When connected, the user can either read or write new values to the registers. The user also has the option of "Constant Sequential Writing." This option has the server cycle through and constantly write the values from the value entries to the Modbus registers.

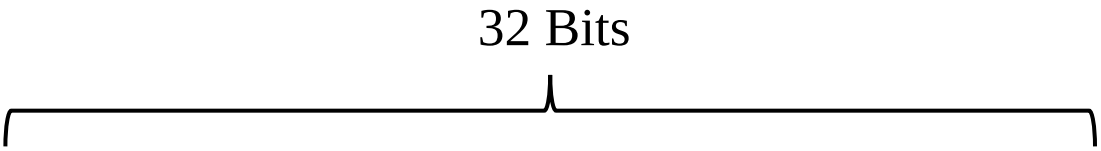
3.2.1.2 UDP

The next communication software written was for a custom UDP protocol. This implementation prioritizes speed of communication and maximum number of messages per second. The length of the data for each UDP packet is 32 bits, giving the overall UDP packet a size of 64 bits. The data is broken down into 4 bytes.

Type H is the high byte for categorization, and Type L is the low byte for categorization, as shown in Figure 3.8. These bytes are used to identify what information is being sent or received. The Type H categories are sub-divided into their communication direction flow, as shown in Table 3.2. The DSP sends the agent its status, measurements, and configuration. The agent sends the DSP its control commands, setpoints, and settings.

Data H refers to the high data byte, and data L refers to the low data byte. This gives the data a length of 16-bits and can send integer values from 0 to 65535. This protocol uses one decimal place for precision, and it uses the leading bit as a sign bit. Therefore, this protocol can effectively transmit values from -3276.7 to +3276.7.

This communication protocol uses the socket library in python to communicate. This library can send encoded bits from one IP address to another through UDP. The socket library is a low-level networking library that can send bits over a network interface. This library has functions for both sending and receiving the network information, however, they must be handled separately, and require two separate threads for sending and receiving.



Type H	Type L	Data H	Data L
1	2	3	4

FIGURE 3.8: BYTE ORDER OF THE CUSTOM UDP PROTOCOL

TABLE 3.2: CATEGORIES AND DATA DIRECTION OF THE CUSTOM UDP PROTOCOL

Direction of message flow	Value	Description
Information from the DSP to the Agent (Tx)	0x01	Status
	0x02	Measurements
	0x03	Configuration
Information from the Agent to the DSP (Rx)	0x04	Control
	0x05	Setpoints
	0x06	Settings

For this implementation, two sockets are opened per node, one for sending and the other for receiving. Each socket has an IP address and port. The receiving IP address is the IP address of the node, and the sending IP address is the address of the node to which the data is being sent. By using two separate ports, the sending and receiving data does not interfere with one another. An example of the communication setup is shown in Figure 3.9.

The GUI gives the user the option to configure the UDP communication protocol. This GUI is shown in Figure 3.10. The different IP addresses and ports can be entered. The buffer size can be specified. This refers to how many bits are processed at once by the socket receiving the data. The GUI also allows the user to specify which communications test to run.

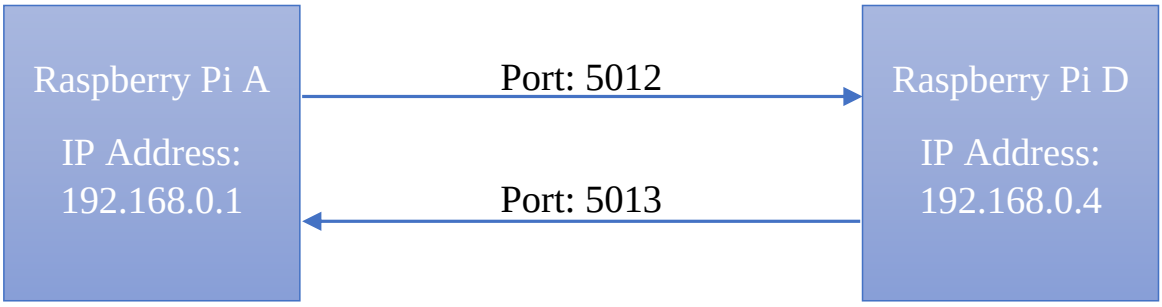
3.2.1.3 MQTT

The last piece of communications software developed for the communications testbed was the message board software, shown in Figure 3.11. The message board communication protocol selected was MQTT (Message Queuing Telemetry Transport). To implement MQTT communications, the Python library Eclipse Paho MQTT Python client library was used. This library allows users to publish and subscribe to different topics.

The MQTT Agent Controller allows the user to specify the client name, the broker IP address and port, and the quality of service (QOS) that the client is going to use. Furthermore, the user can specify the security settings for MQTT. This includes username/password authentication, and TSL/SSL security. If the user specifies these security measures, the necessary credentials can be entered.

For TSL/SSL security, a tool was developed to aid in the generation of the necessary security certifications. This tool is shown in Figure 3.12. This tool allows the user to enter all the required information and filenames for the generation of these security files. Once generated, these files can be used with the MQTT broker to enable secure MQTT communications.

The user then specifies the MQTT schema. The communication settings are used to specified what type of message will be received, custom UDP, Modbus, or manual. The manual communication mode means that the user specifies what the values of the messages are, and when they are published.



	Raspberry Pi A	Raspberry Pi D
Sending IP Address	192.168.0.4	192.168.0.1
Receiving IP Address	192.168.0.1	192.168.0.4
Sending Port	5012	5013
Receiving Port	5013	5012

Figure 3.9: EXAMPLE COMMUNICATIONS CONFIGURATION OF THE CUSTOM UDP PROTOCOL

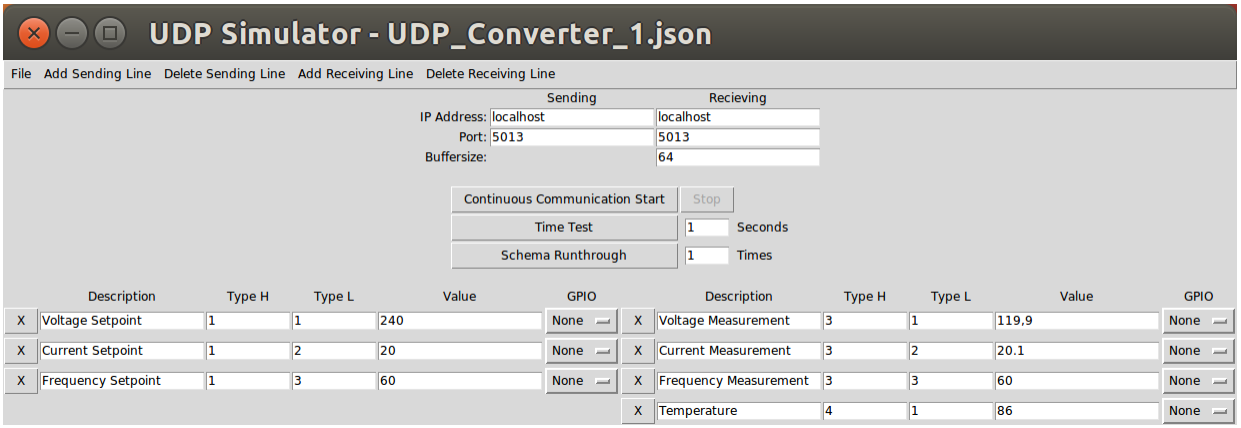


Figure 3.10: CUSTOM UDP PROTOCOL COMMUNICATIONS GUI

FIGURE 3.11: THE MQTT AGENT WITH MANUAL COMMUNICATION

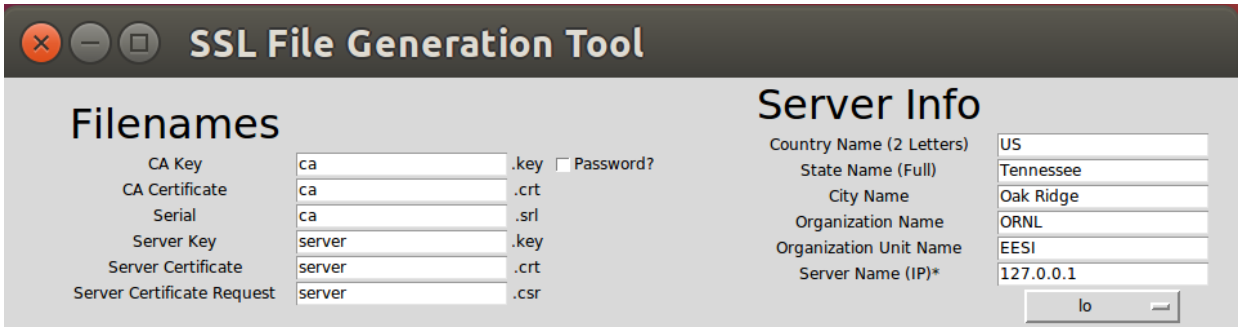


FIGURE 3.12: TSL/SSL CERTIFICATION GENERATION TOOL

When the MQTT communications is unsecure, the standard port to used is 1883. When the MQTT communications is secured, the standard port to use is 8883. This testbed uses the Eclipse Mosquitto MQTT broker to handle the MQTT configurations. It allows for the implementation of the secure MQTT communications and allows for certain topics to be hidden from specific usernames. This allows for an agent-based PES to limit access to certain data which components in the system do not require access to.

3.4 Summary

In this chapter, the hardware and software of the power electronics communications testbed are overviewed. First presented is the hardware setup, which detailed the Raspberry Pis, Arduino, and connecting adapters needed to implement the testbed. As shown is how the testing methodology, specifically latency testing requiring a third-party device, motivated the hardware decisions. Furthermore, the network implementation of the testbed is presented. The motivation for the specific multi-network implementation is described, as it allowed for emulation of a PES communication, as well as enable interaction between the Control Computer and the Raspberry Pi nodes. Finally, the testbed communication emulators for the different communication protocols are overviewed.

In the next chapter, the testbed will be utilized to characterize different PES communication protocols. Each protocol's operation will be detailed. Then, different tests designed to quantify the protocol's behavior will be run, and the results will be presented and described. The communication characterization will give insight to the strengths and limitations of each protocol, and considerations for each protocol in terms of PES utilization will be given.

Chapter 4: PES Communications Testbed Experimental Results

For networked communication, the two important characteristics to quantify are message latency and throughput. Latency describes the speed at which data is transmitted. Throughput describes the amount of data that can be transmitted over a specified length of time. By quantifying these characteristics, the data flow of a PES can be determined. Understanding this data flow is paramount for determining the operation of these systems.

These communication characteristics are limited by two main factors: the network speed and the computational capacity of the computer nodes. The current testbed setup utilizes Raspberry Pi 3Bs as the computational nodes. These devices have Quad Core 1.2 GHz Broadcom BCM2837 64bit CPUs and 1 gigabyte of RAM and a 10/100 Mbit/s ethernet port onboard [43]. Certain actions, such as message encoding and decoding, are limited by the Raspberry Pi's computational capacity. Other actions, such as sending and receiving the protocol messages, are limited by the network speed.

4.1 Latency

In a PES, it is important to know how quickly data can be sent from one device and accessed by another. Therefore, for the purpose of this communication testing, latency is defined as the time delay from the sending PES component being instructed to send a message to the receiving PES component having access to the data contained in the message. Due to the protocols having different relationships between the components, which affects the data flow of the messages, the process of testing latency for different protocols varies. The following sections defines the actions required for each protocol to encode, send, receive, and decode messages, and it presents a methodology to measure these different actions.

4.1.1 Latency Measurement Methodology

Measuring latency requires coordination between the different Raspberry Pi's. Clock synchronization between the Raspberry Pi's would have to be implemented through a network

connection, and thus it introduces error due to a synchronization network's latency. Therefore, a method of measuring latency was implemented that utilizes the Raspberry Pi's general-purpose input/output (GPIO) pins. Using the *RPI.GPIO* library, GPIO pins were set high to indicate a message being sent or received, with the Arduino measuring the time delay between the pulses. To verify the accuracy of the Arduino, a Yeapook ADS1013D oscilloscope was used, shown in Figure 4.1.

To validate this method of communication, the time delay of turning on and off the GPIO pins needed to be determined. This was done by writing a simple Python script that repeatedly cycled the pins on and off without any added time delay. This produced a square wave with a frequency of 241 kHz, with one full square wave cycle taking 4.14 µs, shown in Figure 4.2. This means that the computational time required by the Raspberry Pi 3B to either turn on or off the GPIO pin was approximately 2 µs. Since this was two orders of magnitude lower than the timing measurements, the turn on/off time was ignored during latency testing.

For determining time delays internal to the Raspberry Pi, i.e. encoding and decoding, the python library "time" was used. This library can be used to determine the exact time a process takes in python, and has floating-point number accuracy. To find the time an action takes to execute, the time before the action takes place (time1) is saved. Then the time after the action is completed is finished is saved (time2). By substracting the "time1" from "time2," the time it takes for the action to be completed can be determined.

4.1.2 UDP Latency

Figure 4.3 details the data flow of the custom UDP protocol. "Push button" refers to an action taken by a user or device that initializes the sending of a message. "Encode message" refers to the time it takes for the Type H, Type L, Data H, and Data L to be converted to bytes data type and sent using the Python socket library. The time this action takes is referred to as $t_{encode.}$ The time that the message takes to be sent and received over the network connection is the latency, referred to as $t_{networklatency}$.

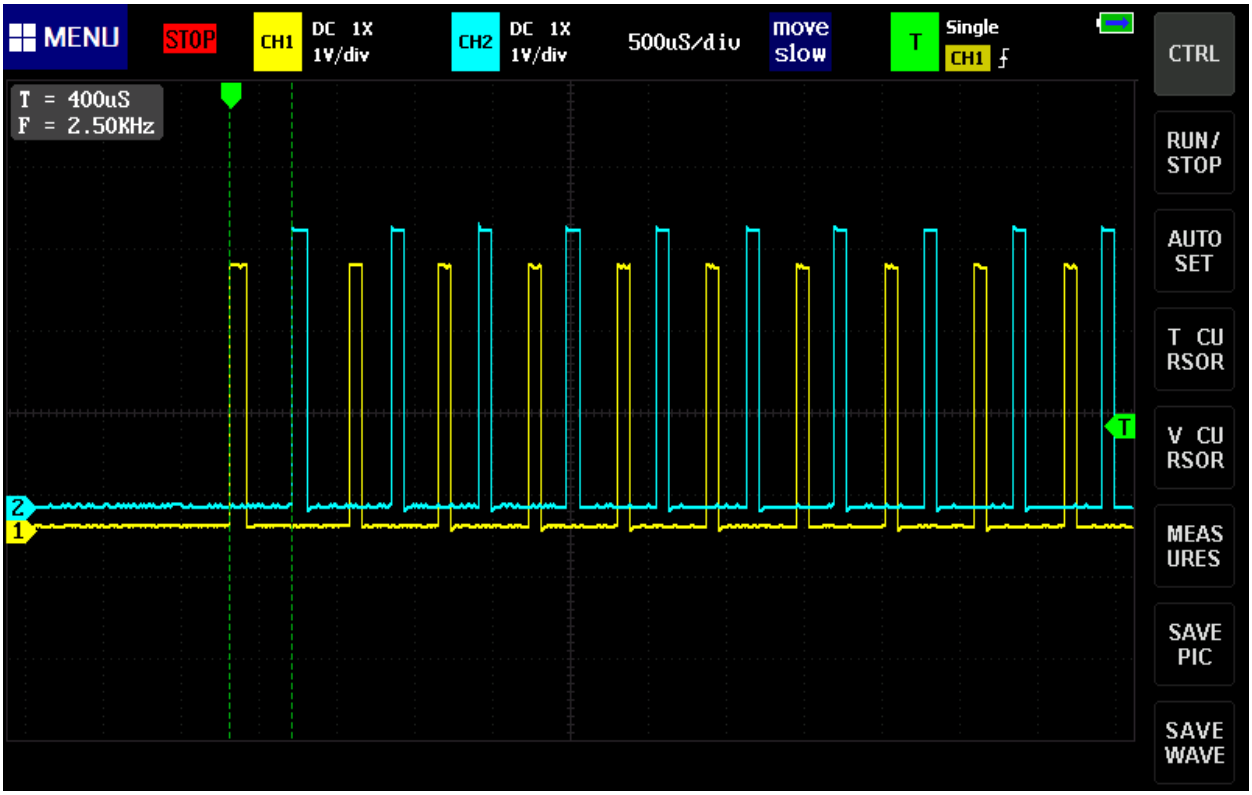


FIGURE 4.1: OSCILLOSCOPE SCREENSHOT SHOWING THE GPIO PULSES CORRESPONDING TO THE SENT (YELLOW) AND RECEIVED (CYAN) MESSAGES

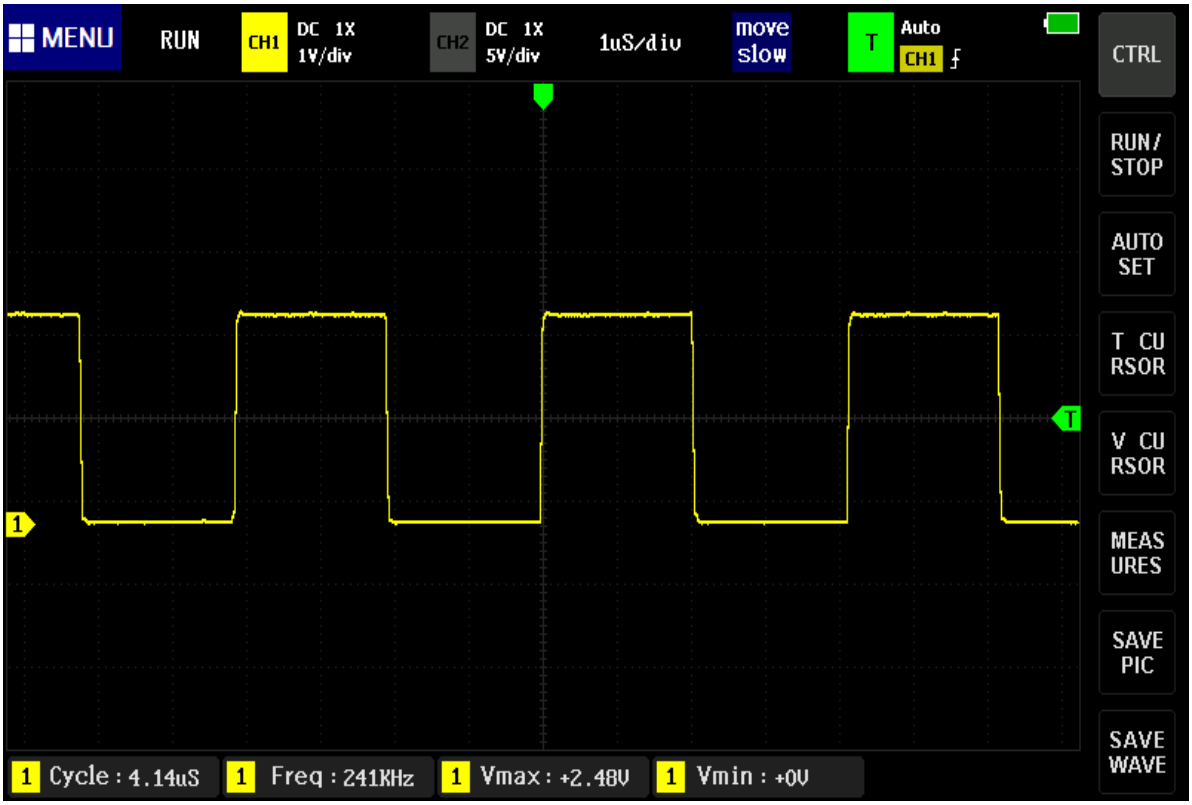


FIGURE 4.2: WAVEFORM OF THE SIGNAL GENERATED BY THE RASPBERRY PI GPIO WHEN CYCLED ON AND OFF SHOWING THE CYCLE TIME OF 4.14 µS

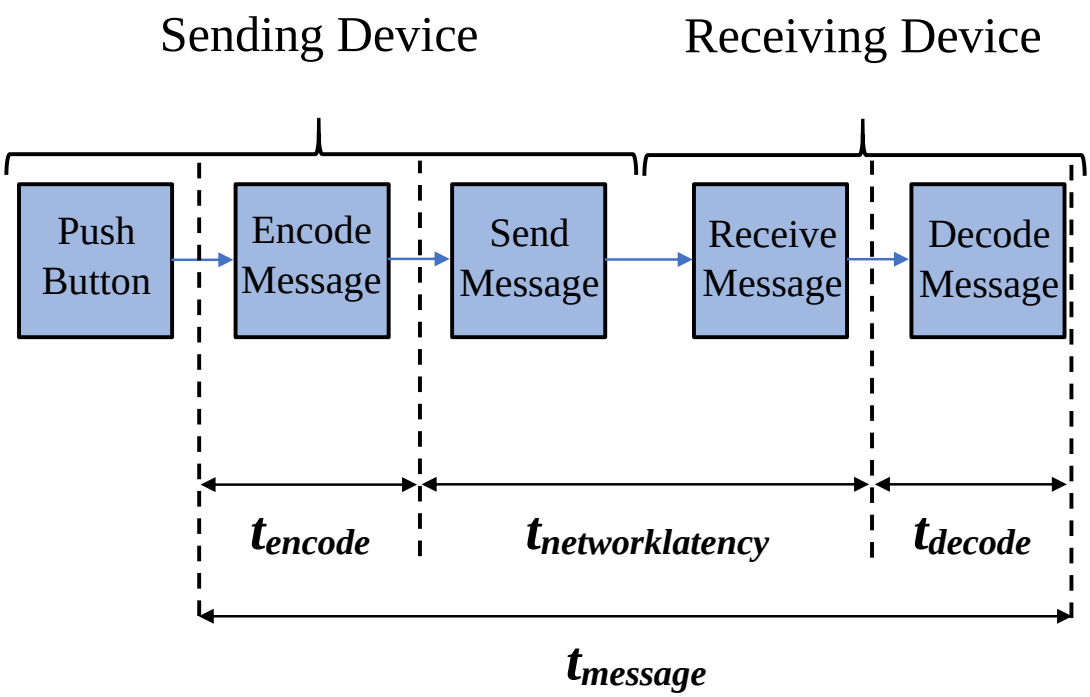


FIGURE 4.3: DATA FLOW OF MESSAGES WITH THE CUSTOM UDP COMMUNICATION PROTOCOL [2]

Last, in “Decode Message” the bytes are decoded into Type H, Type L, and the value by the receiving device. This is referred to as t_{decode}. The overall time that it takes the data to be sent from the sending device to the receiving device and accessed is $t_{message.}$ In this protocol, the times t_{encode} and t_{decode} are subject to the computational capability of the node. The time $t_{networklatency}$ is subject to the network speed, which is 10 mbps for the Raspberry Pi 3B, and its resources. If the network is congested, latency times will be higher.

4.1.3 Modbus Latency

Modbus is a client-server communication protocol; messages are asynchronous in their reading and writing between devices. In Modbus TCP/IP, the servers host the data, and the client must establish a connection to the servers to access and edit the data. This means that clients and servers can read and write data at different speeds. A server’s reading and writing actions are internal to the device, and a client’s reading and writing actions require network communication. To determine the time required to send a message from one device to another, referred to as $t_{message,}$ the reading and writing times, t_{read} and t_{write}, are added together for a theoretical minimal message time. However, it should be noted that due to the asynchronous nature of the server and client, the reading and writing actions are not coordinated, and therefore the actual message time would be longer based on how fast the data was changing and being written to the server, and on how often the client was polling the server. Figure 4.4 shows a diagram of the data flow of Modbus communication. “Modbus Device 1” and “Modbus Device 2” can refer to either the server or client. Regardless, the data will be stored on the server device.

4.1.4 MQTT Latency

The final protocol tested is MQTT, which is a synchronous communication protocol: the data is published to the devices that are subscribed to the data’s topic as the data is published. The data flow is shown in Figure 4.5. Both the publish and subscribe action require communication through the network. Because the subscribed device is actively waiting for the data to be published, the $t_{message}$ value accuracy reflects the time is takes for a message to be communicated through MQTT, with no other delays that need to be accounted.

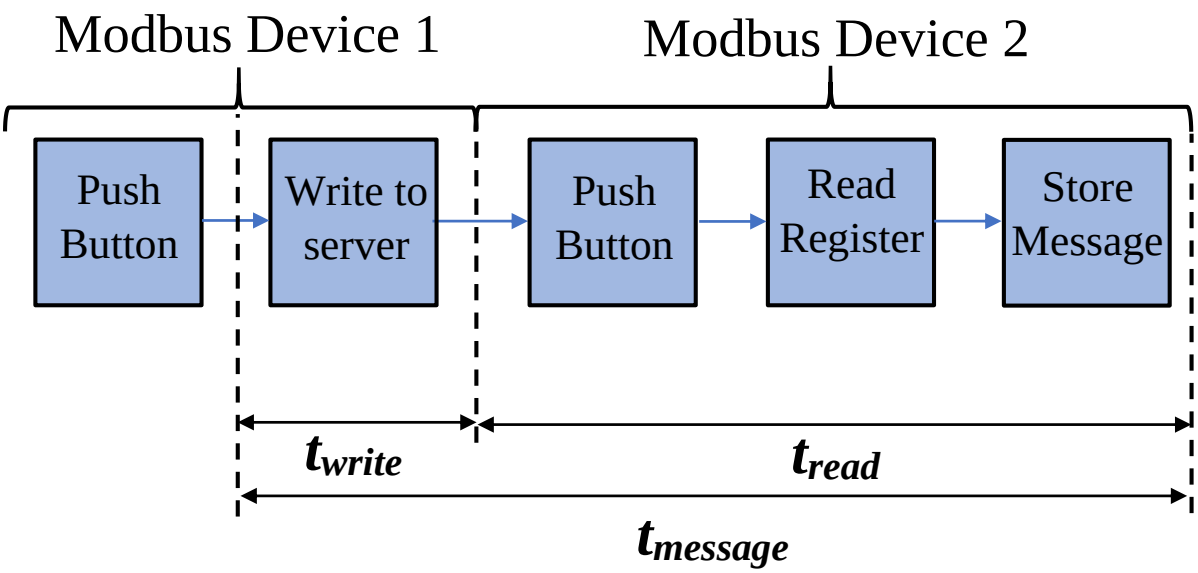


FIGURE 4.4: DATA FLOW OF A CLIENT TO SERVER MESSAGE IN MODBUS

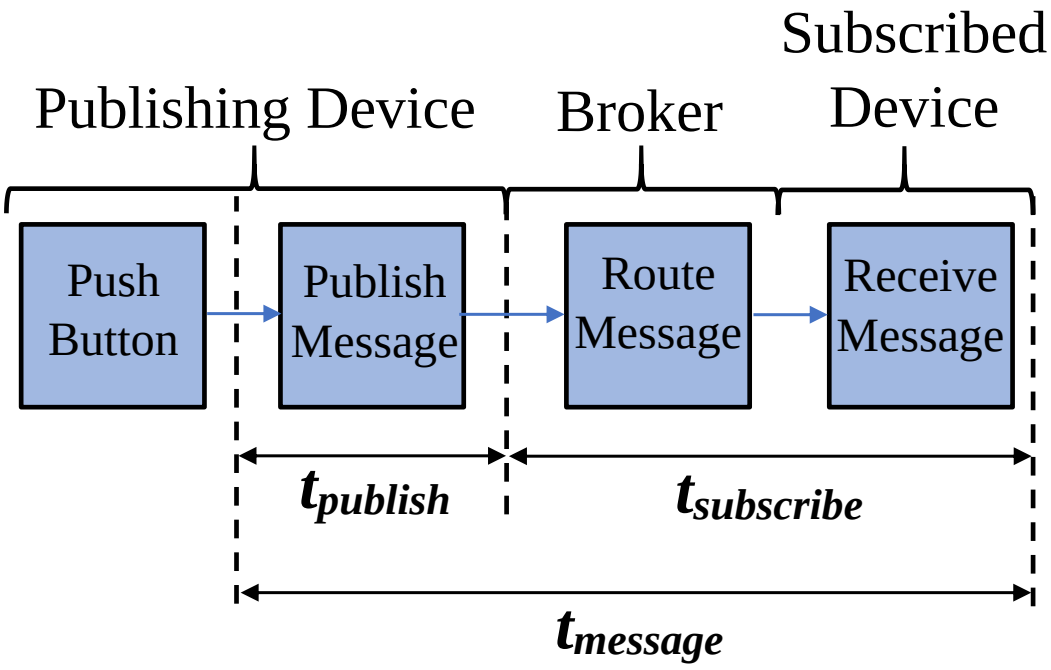


FIGURE 4.5: DATA FLOW FROM A PUBLISHING TO A SUBSCRIBED DEVICE IN MQTT

4.2 Throughput

Throughput is defined as the successful message rate of a communication protocol. For a message to be successful, it must be transmitted without error or being dropped. MQTT and Modbus have built in mechanisms to ensure that information is properly transmitted. For examples, MQTT allows the user to specify if the devices should check with one another that messages are received. However, the custom UDP protocol does not ensure that data is received. A Raspberry Pi is capable of encoding sending messages faster than it is receiving and decoding. This means that the agent running the communication is responsible for ensuring data integrity.

$$error\ rate = \frac{messages\ sent - messages\ received}{messages\ sent} \tag{4.1}$$

Since the purpose of the custom UDP protocol was to transmit data at a faster rate than Modbus, its throughput is compared to Modbus TCP/IP and Modbus over UDP. Furthermore, the throughput of MQTT is determined because it would limit the data flow between different agents in a multi-agent PES.

4.3 UDP Custom Protocol Testing

The first test run on the UDP protocol was to determine its throughput. To do this, data was sent from one Raspberry Pi to another with different message rates, and the error rate was calculated. Since the application is required to ensure data integrity, it incorporated a delay between messages to adjust the message rate. This testing shows how fast the message rate can be while seeing 100% message transmission.

To determine the error rate, comparing the number of messages sent is compared to the number of messages received, as shown in equation 4.1. The testing started at 300 messages per second, and went up to 9000 messages per second, which was the maximum message rate the Raspberry Pi could output. This was well short of the theoretical limit of 104167 messages/second based on the 10 mbps networking used in the testbed. The results of the message rate verses the error rate are shown in Figure 4.6 [2].

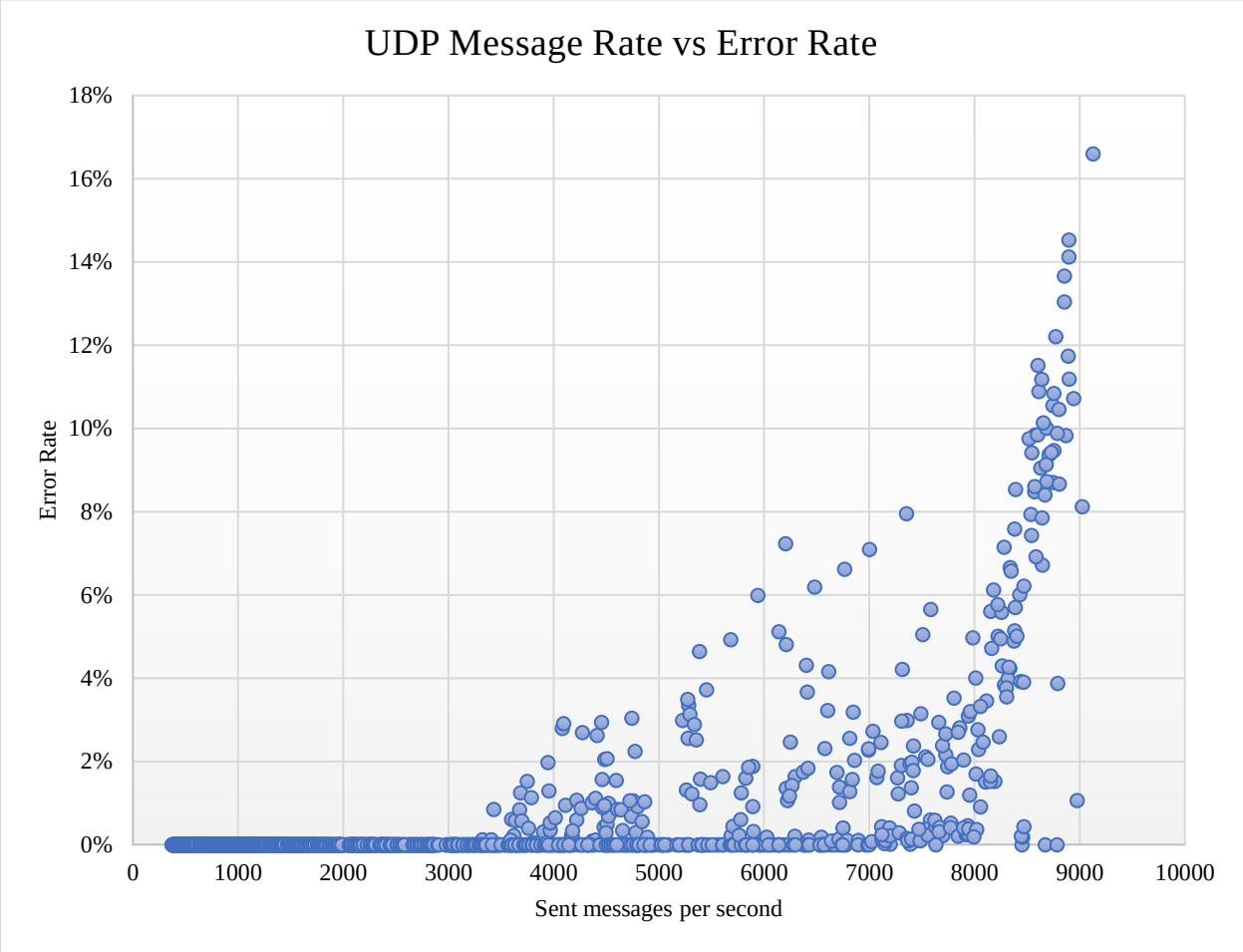


FIGURE 4.6: THE MESSAGE RATE VERSES ERROR RATE FOR THE CUSTOM UDP PROTOCOL [2]

The test showed that the custom UDP protocol had a 0% error rate up to 3325 messages per second. However, when the message rate was higher than 3325 messages per second, the error rate was seen to be above 0%. However, some iterations above 3325 messages per second still had a 0% error rate, up until approximately 7000 messages per second. Above this rate, a 0% error rate was rarely seen. Based on these results, the message rate was limited to 3000 messages per second, giving margin between the message rates with theoretical error rates higher than 0%.

Next, the t_{encode}, $t_{networklatency}$, t_{decode}, and $t_{message}$ values were measured for the custom UDP protocol. Each test consists of 3000 messages. Due to the testing setup needing to be reconfigured to measure different values, four separate tests were required to measure the four separate values. The first test was overall message time, shown in Figure 4.7. The custom UDP protocol had an average message time of 522 µs. For the 3000-message sample below, it can be seen there is fluctuation in the message time. To determine the cause behind the fluctuations, further examination of the components that make up the message time, t_{encode}, $t_{networklatency}$, and t_{decode} will take place.

The next test run was the encoding time, t_{encode}. This is the time that the sending Raspberry Pi takes to encode the message value before it is sent over the network to the receiving Raspberry Pi. A test of 3000 messages is shown in Figure 4.8. Encoding time is entirely dependent on the computational capacity of the Raspberry Pi. The average time of encoding was 114 µs. A large majority of the samples measured were close to this average value. However, a small sample of the times were outliers. Due to the encoding time being almost entirely dependence on the computational capacity of the Raspberry Pi, these outliers can be attributed to the Raspberry Pi's process variation. Other processes are constantly running in the background of the Raspberry Pi, using computational resources.

After the encoding test, the latency time, $t_{networklatency}$, test was run. This test shows how fast the UDP packets are transferred from the sending Raspberry Pi to the receiving Raspberry Pi. A test of 3000 messages is shown in Figure 4.9. The average latency time for the UDP packet was 306 µs. Most latency samples measured were close to this average time, however, significant outlying values were also measured. Latency variation for the UDP packets was expected, as previous research into UDP packet latency has shown variation is inherent to their behavior [44].

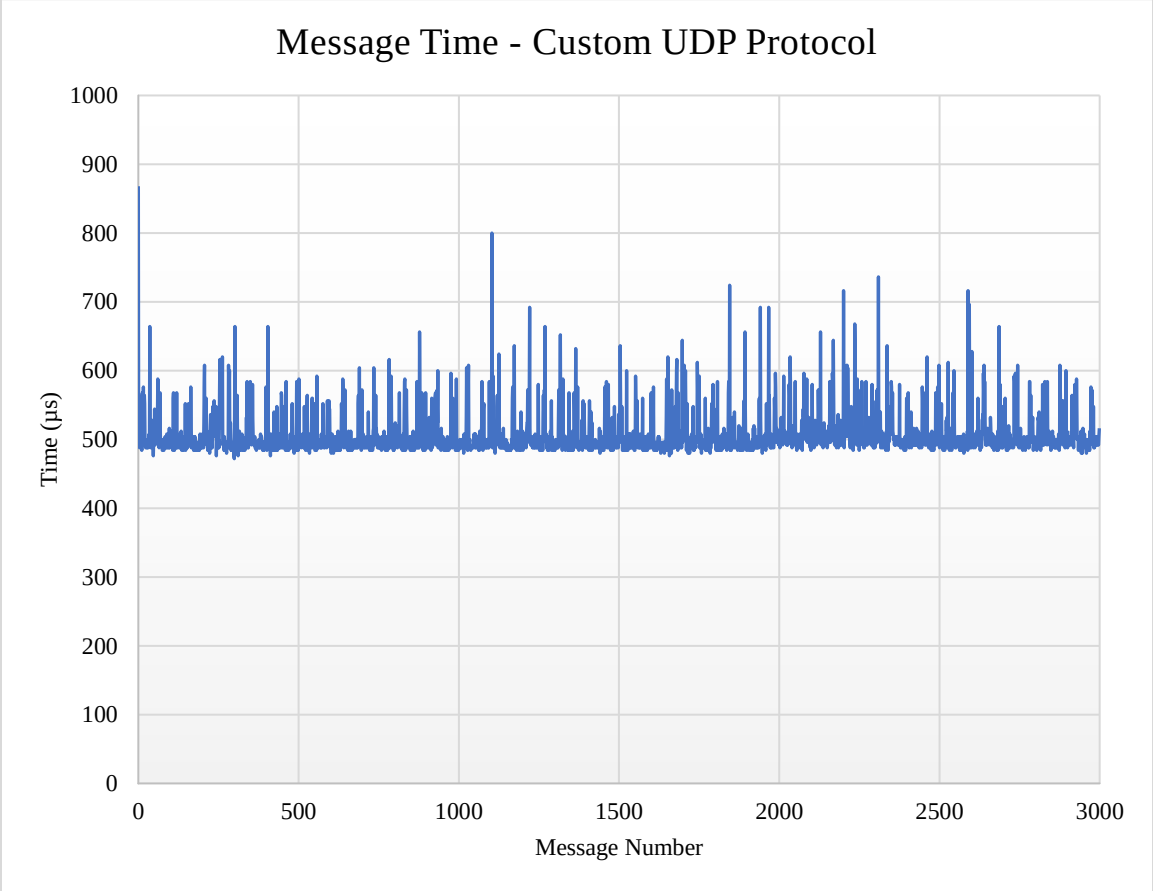


FIGURE 4.7: OVERALL MESSAGE TIME FOR THE CUSTOM UDP PROTOCOL [2]

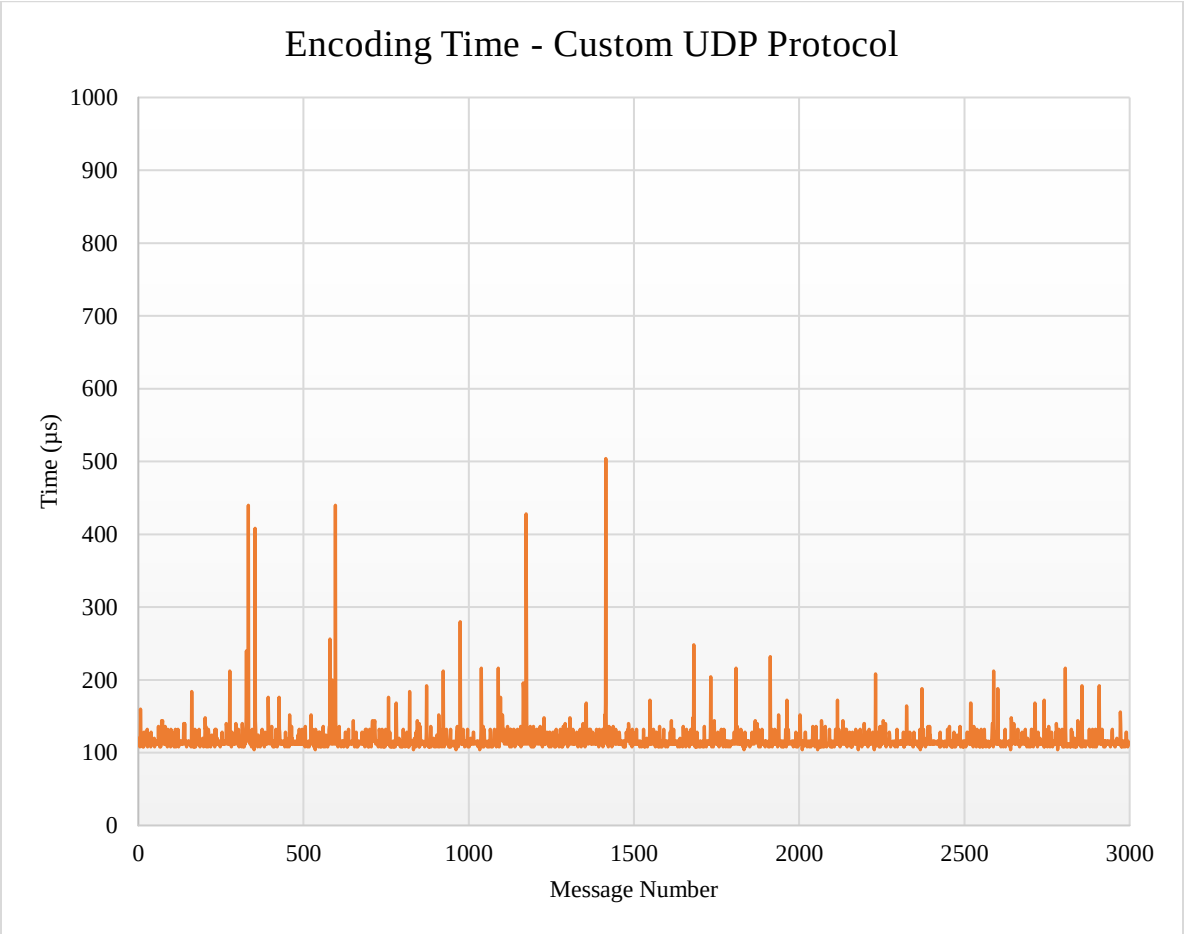


FIGURE 4.8: RASPBERRY PI ENCODING TIME FOR THE CUSTOM UDP PROTOCOL

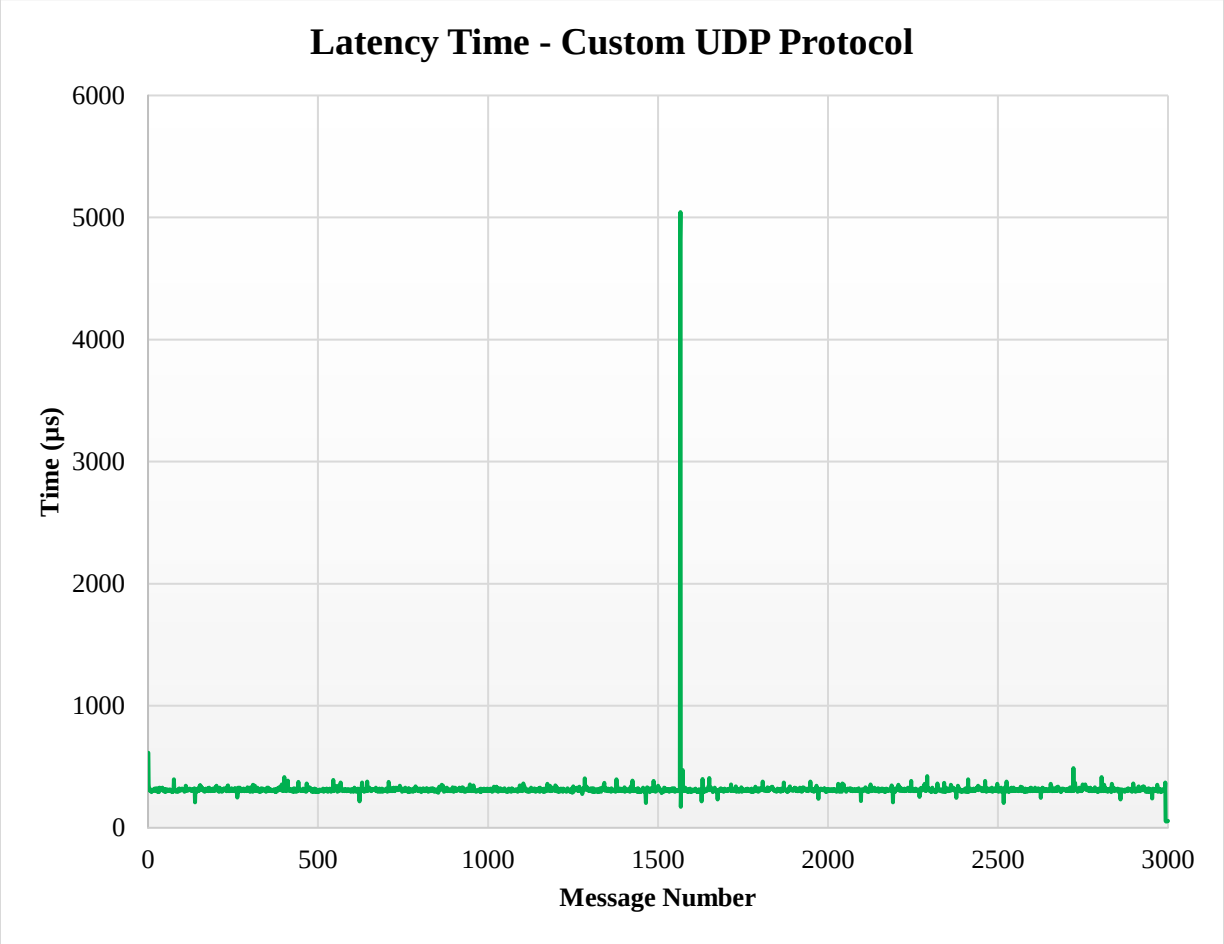


FIGURE 4.9: NETWORK LATENCY TIME FOR THE CUSTOM UDP PROTOCOL

The last custom UDP protocol timing test run was the decoding time, t_{decode}. This test shows how fast the UDP packets are decoded by the receiving Raspberry Pi. A test of 3000 messages is shown in Figure 4.10. While Figure 4.10 does not show any extreme outliers in decoding time, the 9000-message test shown in Figure 4.11 did show some outlying values. The encoding and decoding average times were close to one another, with average times of 114 µs and 102 µs, respectively. This is excepted, as both processes are dependent on the Raspberry Pi's available computational capacity and are functionally the inverse of one another.

All the timing tests exhibited similar behavior with most samples being close to the average value. However, in each test, there were multiple values that were much higher than the average value. This behavior can be observed in Figure 4.11. In this figure, each time measurement is measured 9000 times. Of the individual timing values (t_{encode}, $t_{networklatency}$, t_{decode}), the encoding time diverged the least from its average value, and the latency time diverged the most. This overall divergence from the average time is culminated in the message time. By slowing down the message rate, the receiving Raspberry Pi has time to process all the messages, even the one with a much higher message time. However, as the message rate increases, the receiving Raspberry Pi no longer as time to process a message before another arrives. This causes packets to be dropped, and the error rate to rise above 0%.

The average measured times are shown in Table 4.1. These average values are from 9000 timing samples. Based on the measured values, the limiting factors of the maximum message rate and maximum message rate with 0% error rate can be determined. . The maximum message rate is limited by the time required to encode the UDP message, as:

$$\frac{1\ message}{t_{encode}} = \frac{1\ message}{114\ \mu s} = 8772\ messages/second \tag{4.2}$$

and network latency as:

$$\frac{1\ message}{t_{latency}} = \frac{1\ message}{306\ \mu s} = 3268\ messages/second \tag{4.3}$$

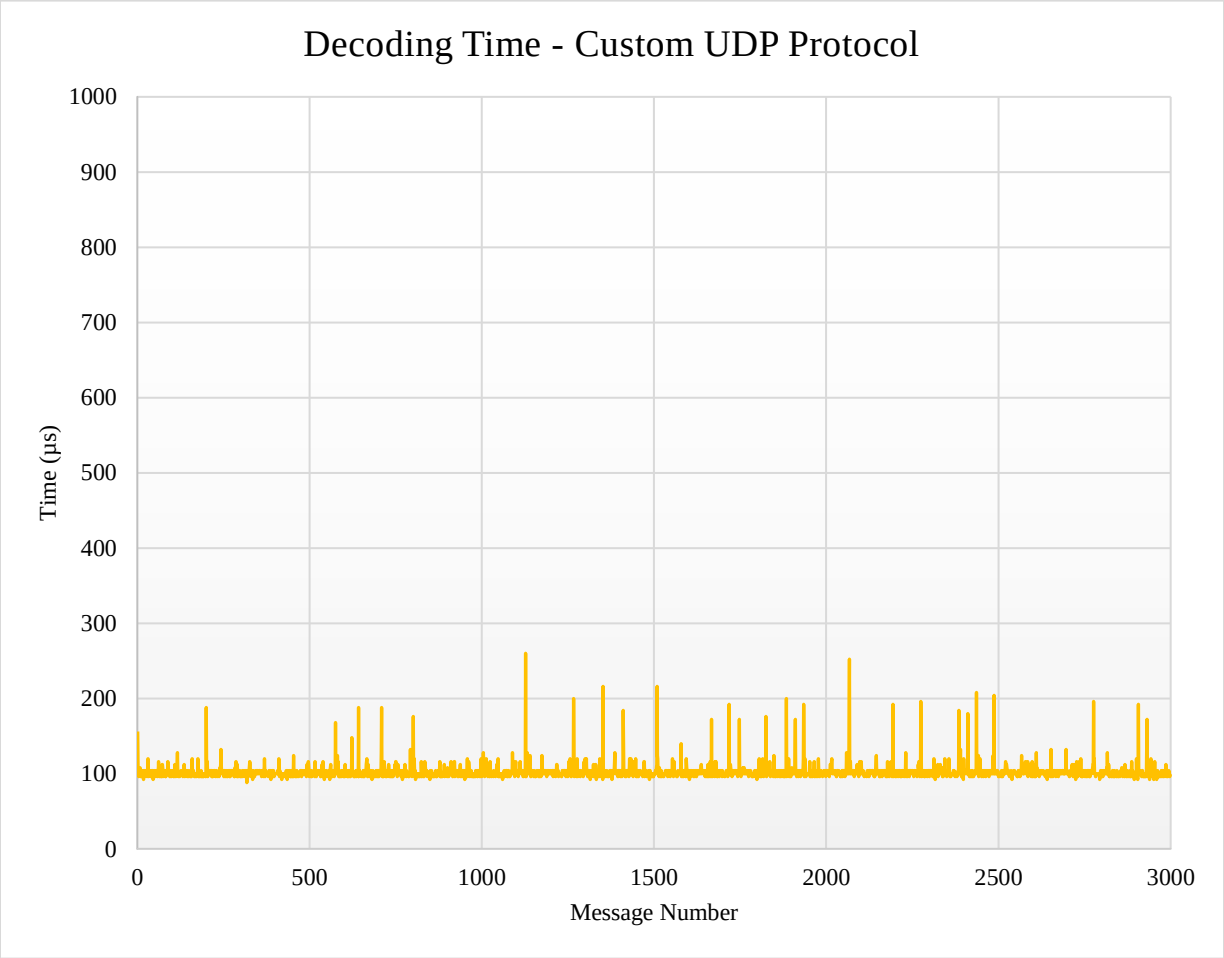


FIGURE 4.10: RASPBERRY PI DECODING TIME FOR THE CUSTOM UDP PROTOCOL

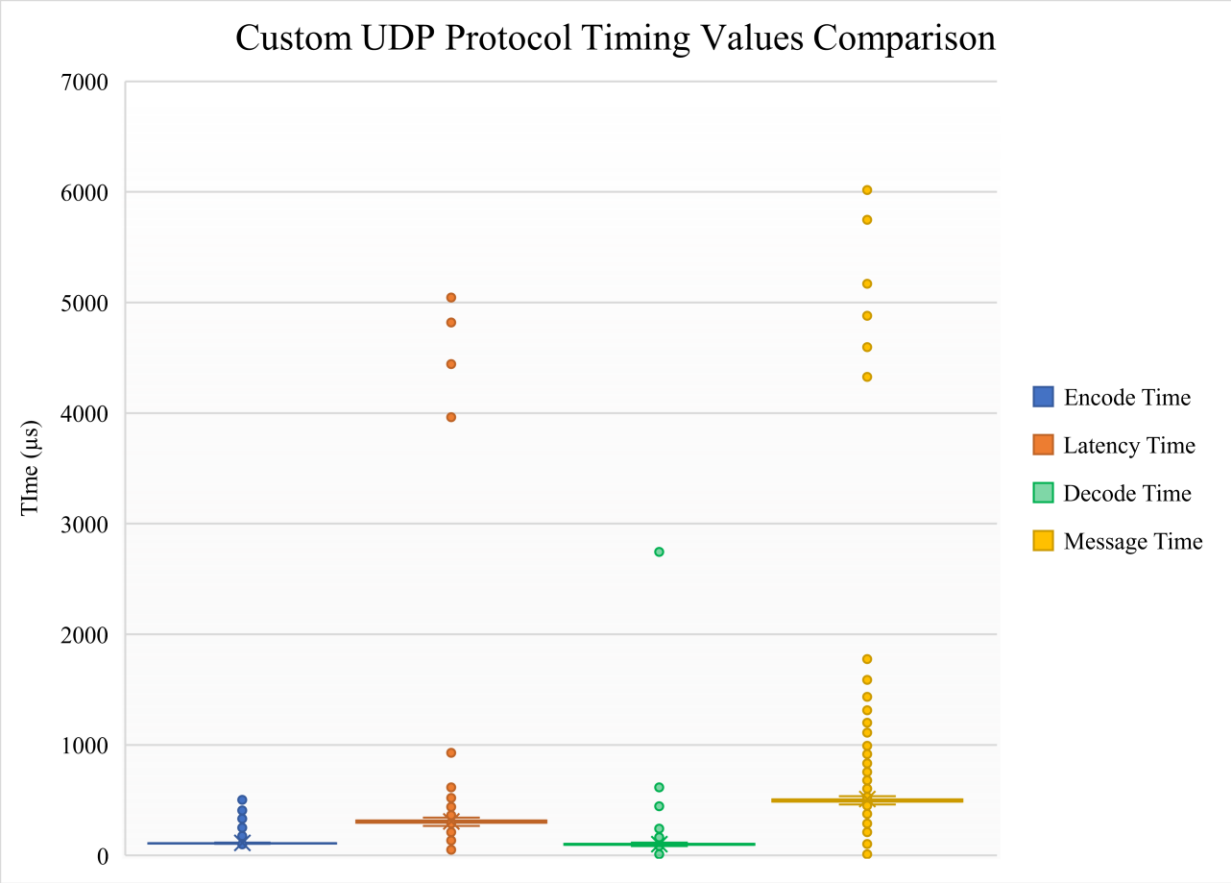


Figure 4.11: RASPBERRY PI TIMING VALUES FOR THE CUSTOM UDP PROTOCOL

TABLE 4.1: MEASURED AVERAGE TIMES FOR THE CUSTOM UDP PROTOCOL [2]

t_{encode}	114 µs
$t_{latency}$	306 µs
t_{decode}	102 µs
$\boldsymbol{t_{message}}$	**522 µs**

The 8772 message per second rate corresponds with the maximum messages per second seen in the error rate test. The 3268 messages per second corresponds to the 3325 messages per second that was determined to be the maximum message rate without errors. The network can only send packets fast enough to support this message rate. With faster message rates, packets must be dropped. Therefore, this setup's largest limiting factor for throughput is the Raspberry Pi's network capability (which is limited by the ethernet port on the Raspberry Pi).

4.4 Modbus Testing

The timing test for Modbus consists of finding t_{read} and t_{write} for both the client and the server. The two Modbus protocols tested were Modbus TCP/IP and Modbus over UDP. Figure 4.12 shows the Modbus TCP/IP register reading times of a Modbus client over 3000 messages. The times were constantly in the 3000 µs to 5000 µs range, with the average time being 3608 µs. However, some outliers were observed at the beginning of the test.

The time required to read Modbus registers from the client is determined by both the available computational capacity of the Raspberry Pi and the network speed. In the previous test, reading times were higher during the beginning of the test. This was believed to be partially caused by the TCP slow start, which is a congestion control technique that involves sending less data at first, then increasing data amount of data as the receiver acknowledges the data is received.

Furthermore, due to Modbus reading and writing times being dependent on the Raspberry Pi's available computational resources, the CPU utilization was measured during an additional Modbus client reading test. From the results shown in Figure 4.13, during the start of the communication test, the CPU usage increases, which contributes to the lower reading times.

For PES, this slight rise in reading times would only come into effect during the startup of the system. After startup, a PES is expected to be operational for hours. This slight increase in reading times only lasts milliseconds. For the remaining timing tests, this increase will be noted, but overall, it is considered inconsequential for PES operation.

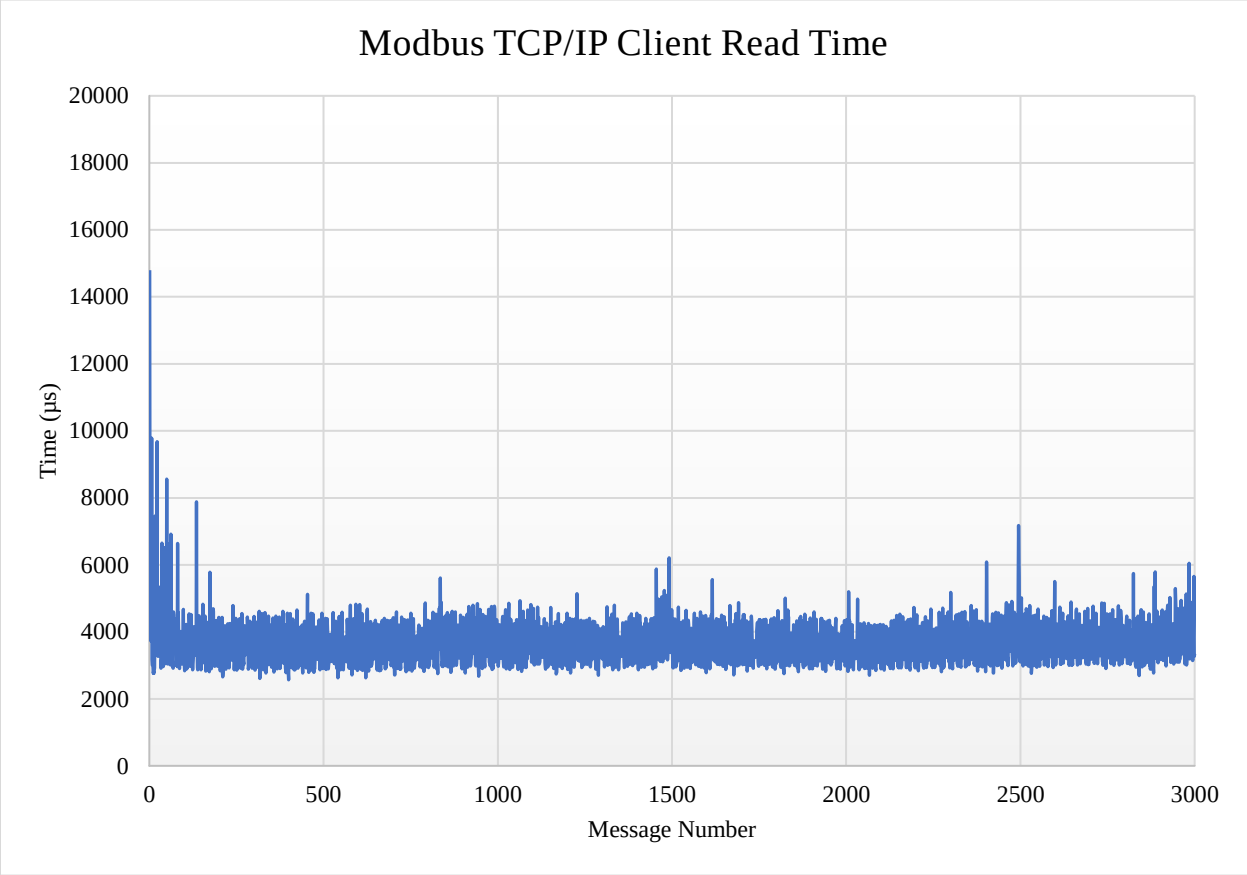


FIGURE 4.12: MODBUS TCP/IP REGISTER READING TIME FOR CLIENT WITH A 3000 MESSAGE SAMPLE

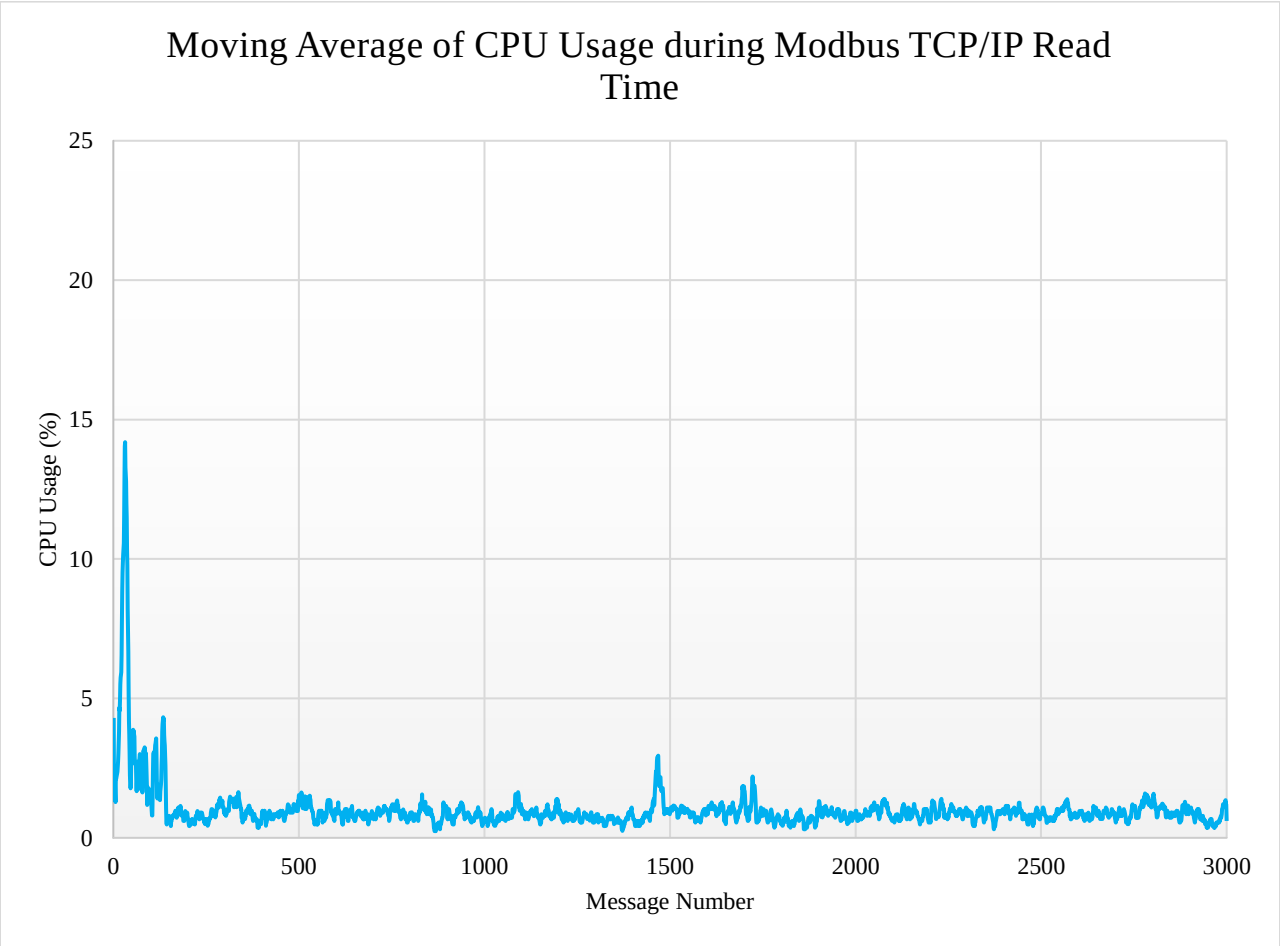


FIGURE 4.13: MOVING AVERAGE OF THE CPU USAGE DURING MODBUS CLIENT READING TIME TEST

Figure 4.14 shows the Modbus TCP/IP register writing times of a Modbus client over a test of 3000 messages. As previously seen with the reading times, the writing times were constantly in the 3000 µs to 5000 µs range with some outliers observed, an average time of 3481 µs. The beginning outliers can be attributed to the startup CPU usage. Other outliers appeared infrequency throughout the test. Modbus writing time is subject to both CPU usage and network speed. Due to the use of TCP/IP, which prioritizes data integrity, and incorporates error checking and correction, random spikes in network timing tests can be expected when it is in use, as the error checking and correction provision could be taking place.

Figure 4.15 shows the Modbus TCP/IP register writing times of a Modbus server over 3000 messages. These reading times were much faster than the client reading times, with most values approximately around the average value of 47 µs. However, there were many outliers from this average. A Modbus server reading registers is completely dependent on computational capacity, as the values are stored on the server. The startup CPU usage affected the beginning of the reading time test. The random spikes seen in the server reading times will affect the overall message time in a PES.

Figure 4.16 shows the Modbus TCP/IP register writing times of a Modbus server over 3000 messages. The server writing times had an average of 528 µs, but there were many outliers from this average. These server writing times also were not as consistent as the reading times, presenting many more spikes in time across the test. These spikes increased greatly after 1500 messages. Since Modbus server writing times are dependent on the computational resources, not the network resources, these spikes can be attributed to the Raspberry Pi's resources being utilized. If minimizing message time is a goal for a specific PES design, Modbus is not an appropriate protocol to utilize in said design.

The timing data for Modbus TCP/IP is presented in Figure 4.17. From the figure, the server reading and writing times are substantially lower than the client reading and writing times. This is to be expected, as the data is stored on the server, and the client must communicate over the network to read or write registers. Furthermore, the spikes from the client reading, client writing, and server writing times are observed.

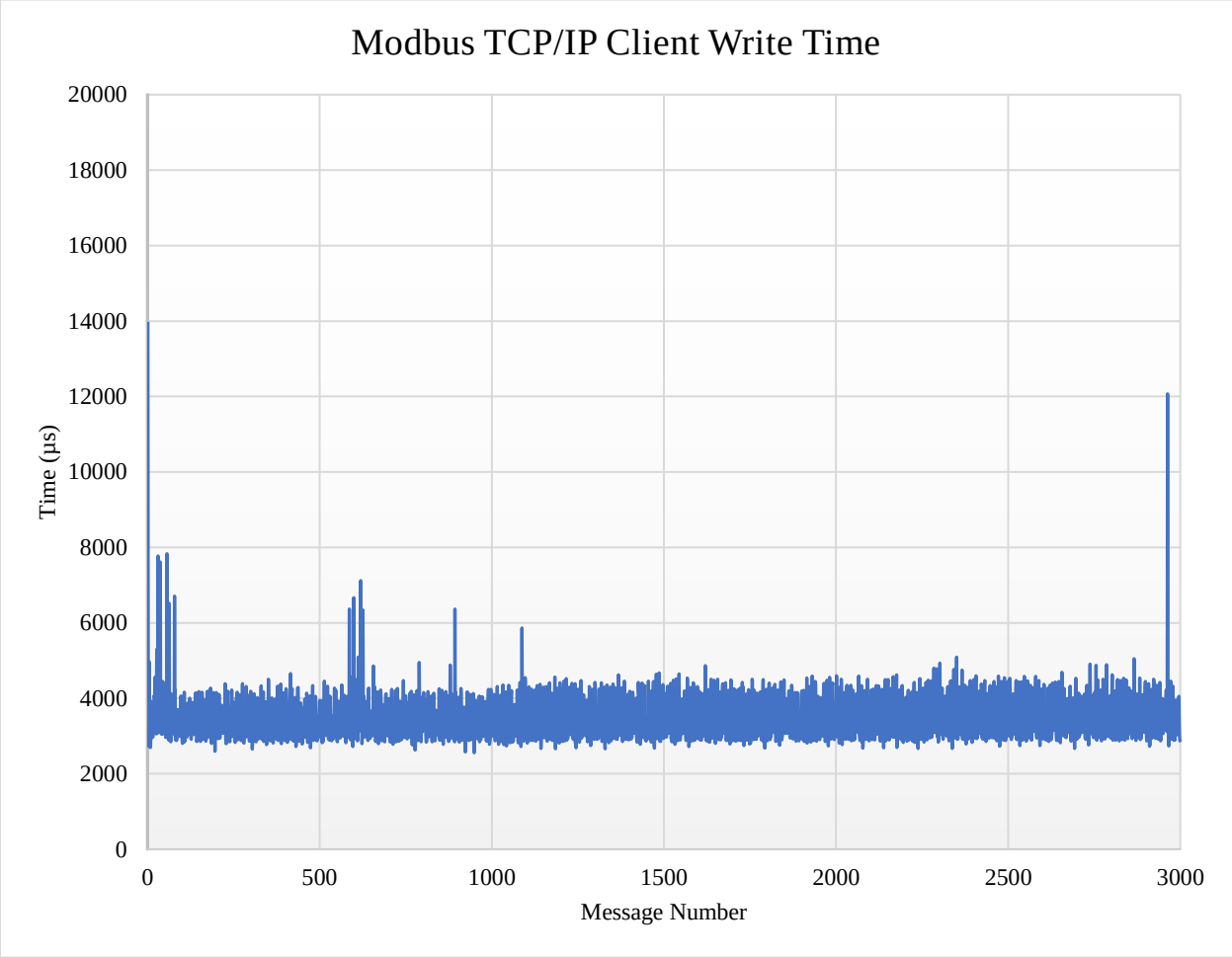


FIGURE 4.14: MODBUS TCP/IP REGISTER WRITING TIME FOR CLIENT WITH A 3000 MESSAGE SAMPLE

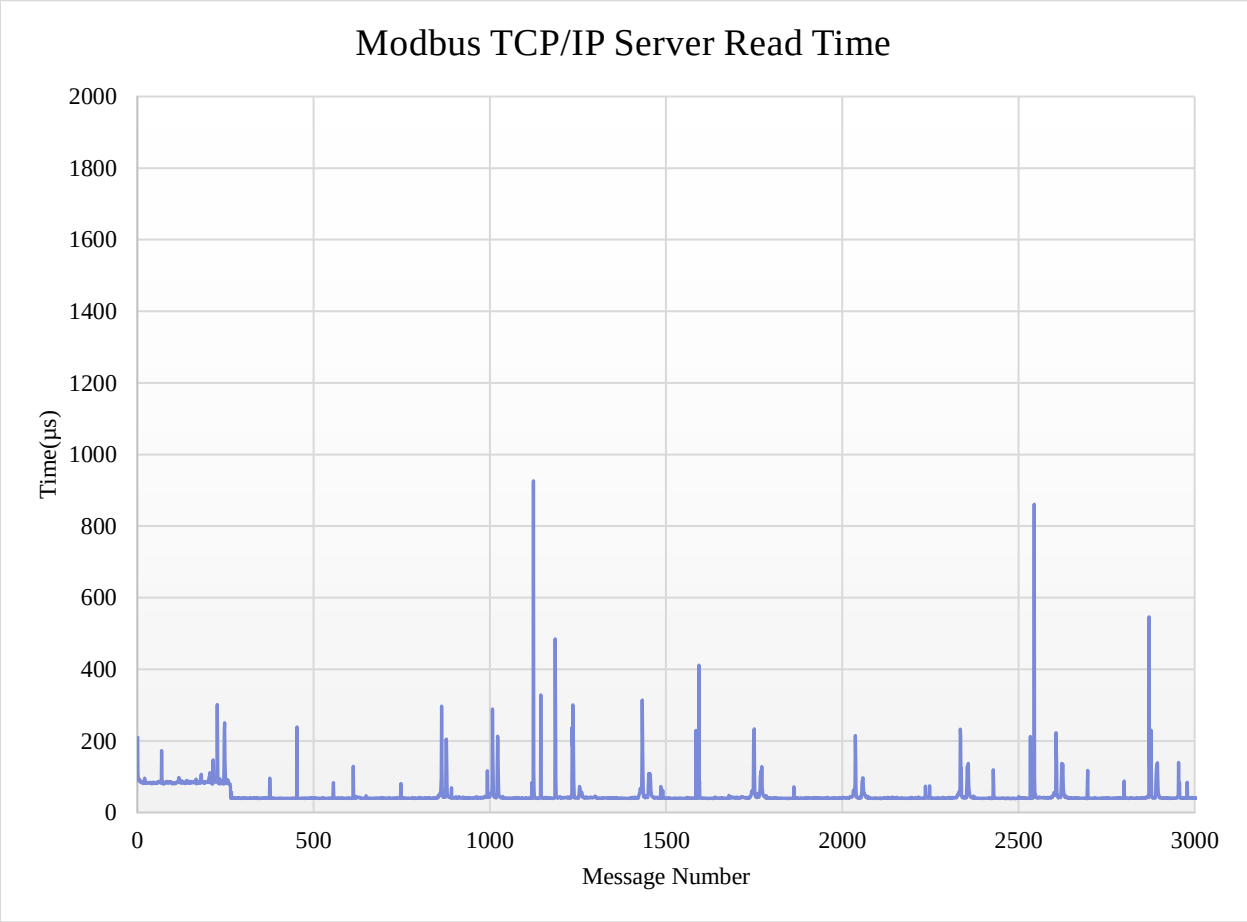


FIGURE 4.15: MODBUS TCP/IP REGISTER READING TIME FOR SERVER WITH A 3000 MESSAGE SAMPLE

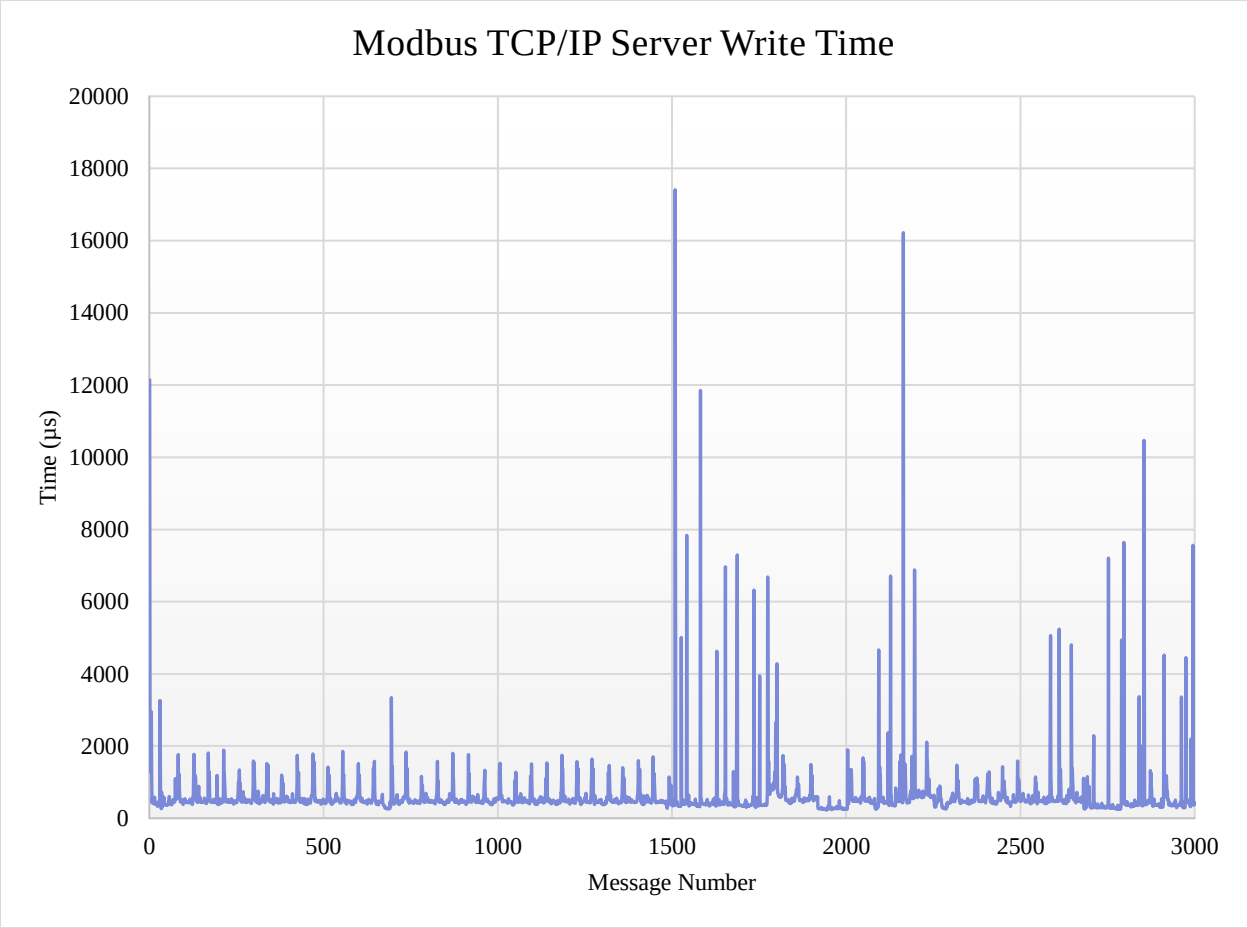


FIGURE 4.16: MODBUS TCP/IP REGISTER READING TIME FOR SERVER WITH A 3000 MESSAGE SAMPLE

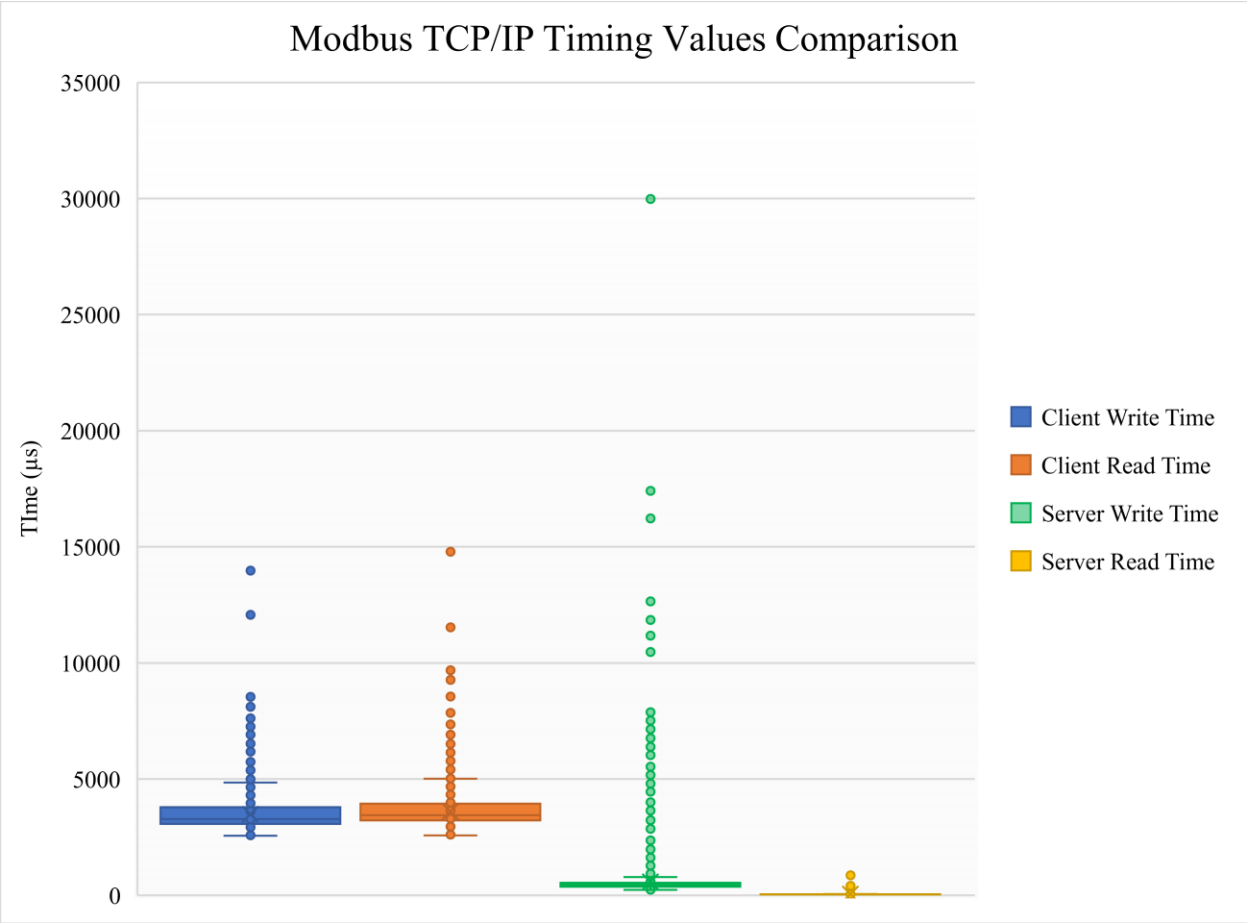


FIGURE 4.17: MODBUS TCP/IP TIMING DATA

The next tests were with Modbus over UDP. The same timing tests, client's and server's reading and writing times were run. The Modbus over UDP client reading times are shown in Figure 4.18. The values were typically between 4000 µs to 6000 µs, the average time being 4713 µs. Outliers were observed. As was Modbus TCP/IP, a Modbus over UDP client's reading time is affected by both computational resources and network speed. Also, as previously stated, UDP is subject to latency variation. Therefore, these outliers are expected.

Figure 4.19 shows the Modbus over UDP register writing times of a Modbus client over a test of 3000 messages. As previously seen with the reading times, the writing times were constantly in the 4000 µs to 6000 µs range. This test shows many outliers, including groups of messages all with higher writing times. The average time was 4607 µs. This action is affected by both computational resources and network speed. Therefore, similar to reading times, these outliers are expected.

Figure 4.20 shows the Modbus over UDP register writing times of a Modbus server over a test of 3000 messages. Again, as seen with Modbus TCP/IP, the server reading times were much faster than the client reading times, with most values approximately around the average value of 47 µs. Some outliers were observed, with all outliers being less than 400 µs. The beginning messages taking longer is the same behavior observed with Modbus TCP/IP and is due to more computational resources being used during the start of the test.

Figure 4.21 shows the Modbus over UDP register writing times of a Modbus server over a test of 3000 messages. The server writing times had an average of 575 µs. As also seen in the server writing times of Modbus TCP/IP, there were many outliers from this average. These Modbus server writing times are dependent on the computational resources, not network speed, As previously found with the Modbus TCP/IP times, if minimizing message time is a goal for a PES design, Modbus over UDP is not an ideal protocol to utilize.

All of the timing data for Modbus over UDP is presented in Figure 4.22. The server reading and writing times are substantially lower than the client reading and writing times, which was the same trend observed with Modbus TCP/IP. This is to be expected, as the data is stored on the server, and the client must communicate over the network to read or write the data.

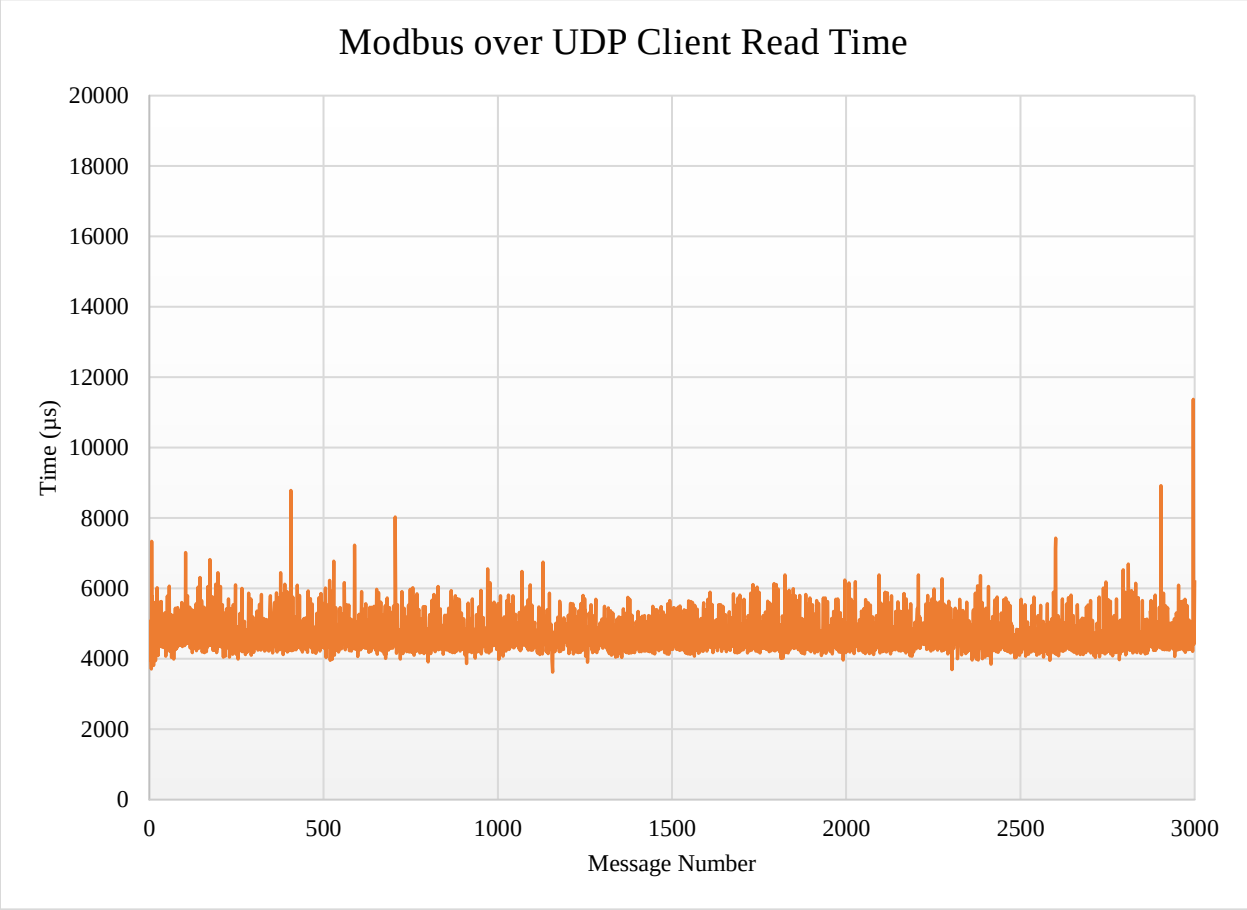


FIGURE 4.18: MODBUS OVER UDP REGISTER READING TIME FOR CLIENT WITH A 3000 MESSAGE SAMPLE

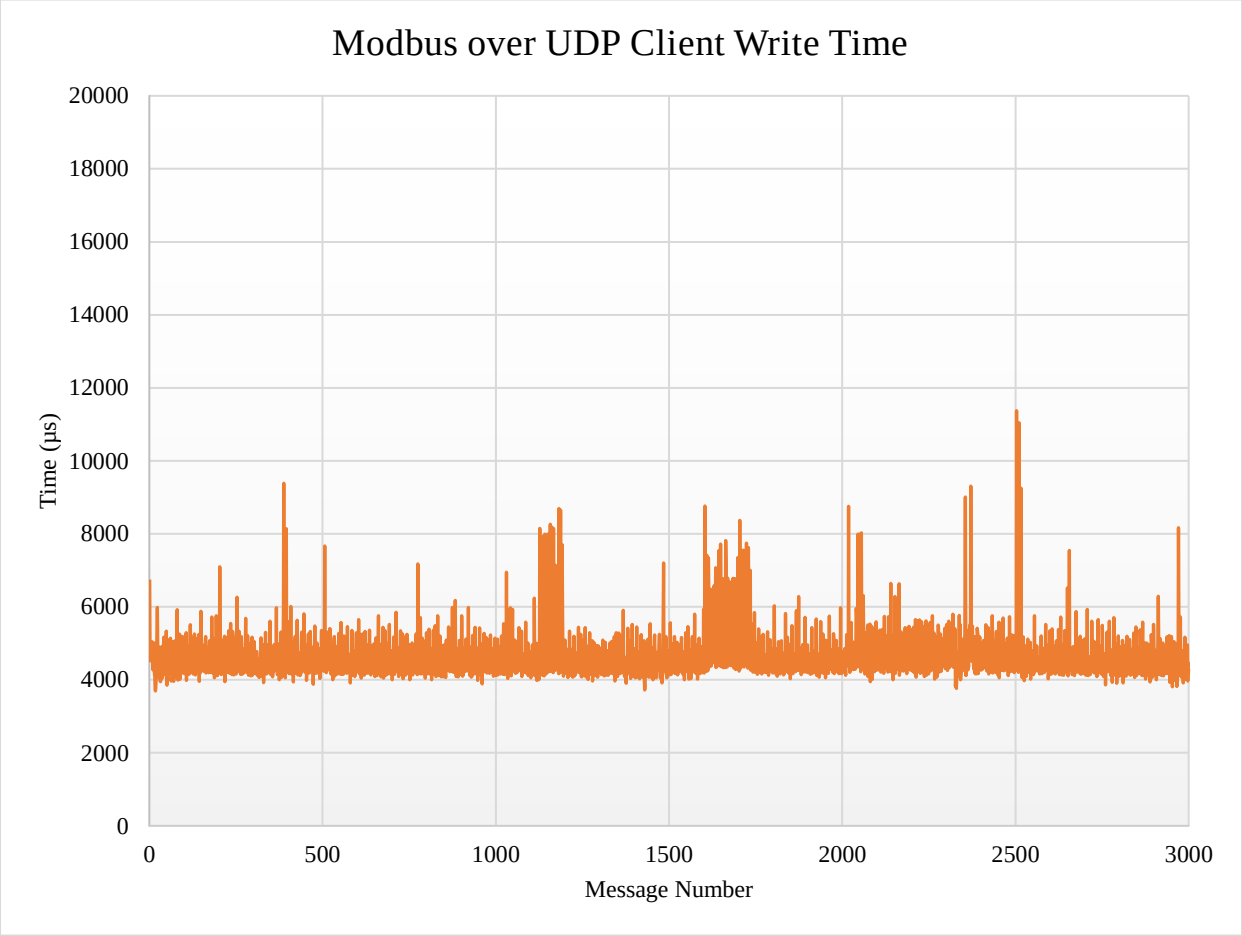


FIGURE 4.19: MODBUS UDP REGISTER WRITING TIME FOR CLIENT WITH A 3000 MESSAGE SAMPLE

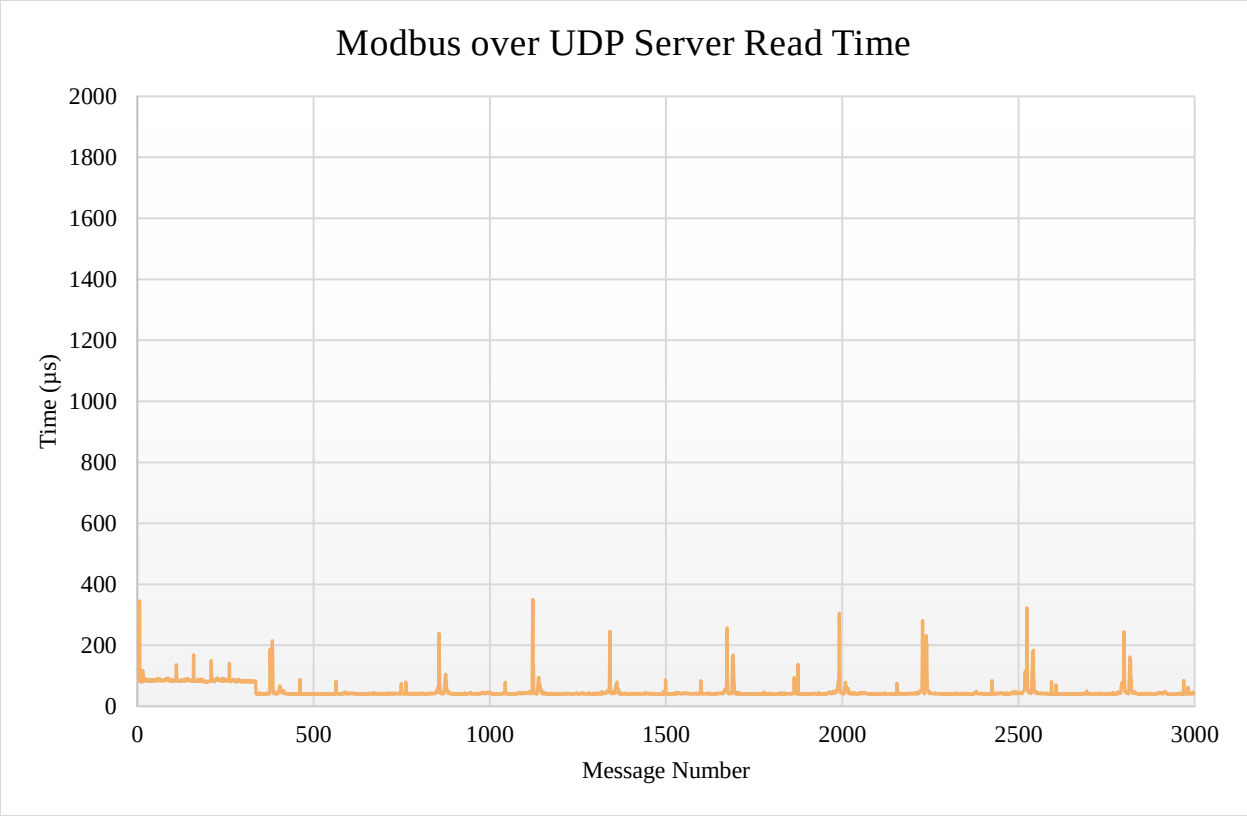


FIGURE 4.20: MODBUS UDP REGISTER READING TIME FOR SERVER WITH A 3000 MESSAGE SAMPLE

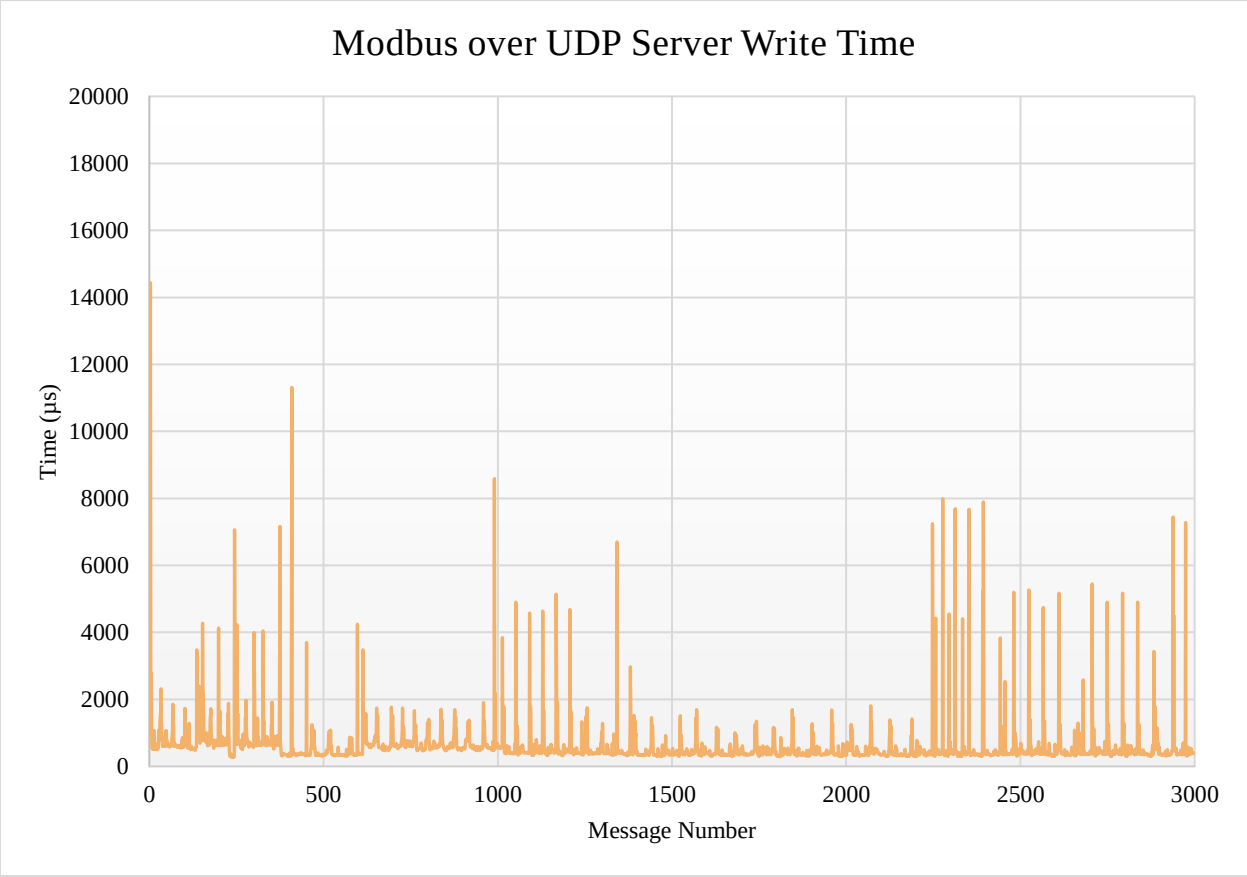


FIGURE 4.21: MODBUS UDP REGISTER WRITING TIME FOR SERVER WITH A 3000 MESSAGE SAMPLE

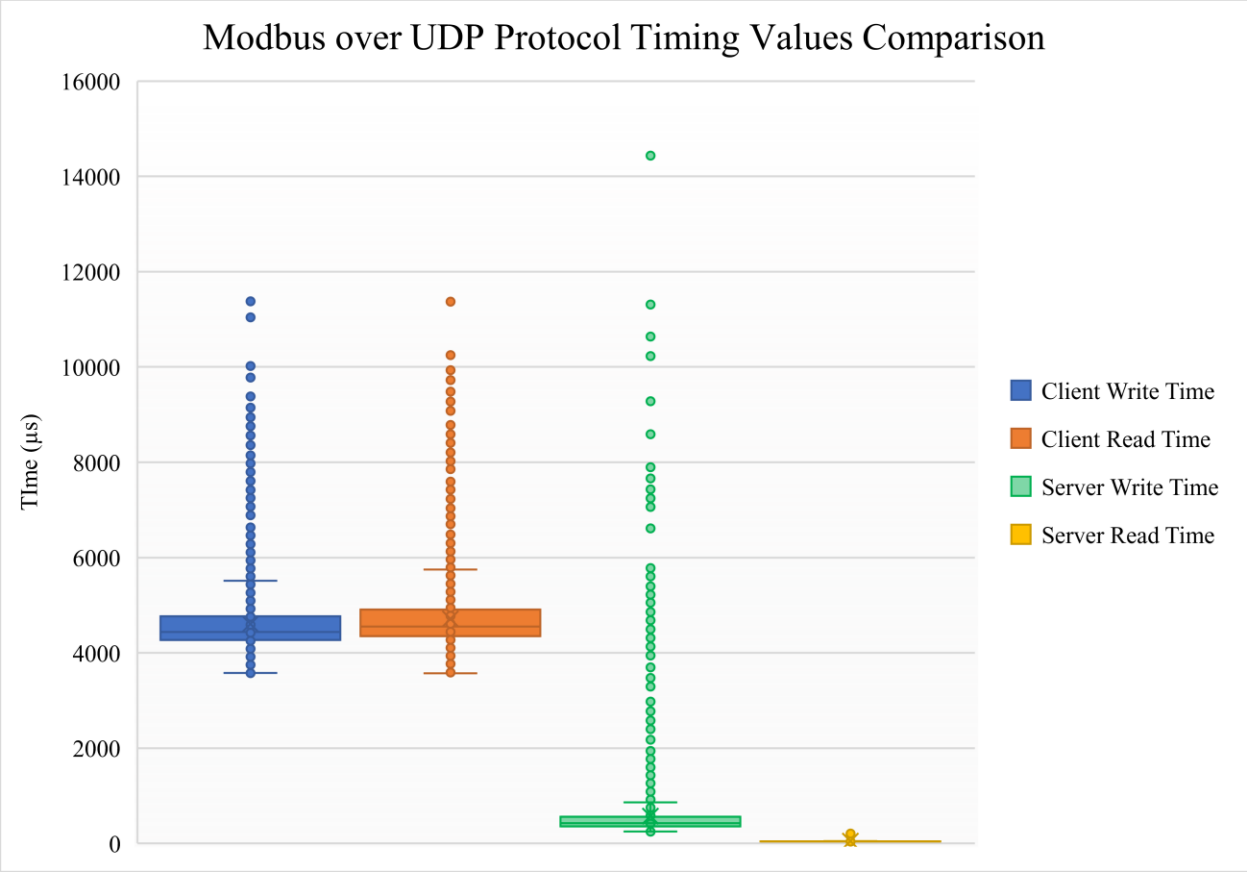


FIGURE 4.22: MODBUS OVER UDP TIMING DATA

Furthermore, the spikes from the client reading, client writing, and server writing times are observed. The client's reading and writing times had similar average times, and a similar distribution of outlying, higher latency times. For the server, the read times had no notable outliers, compared to the server's write times, which had a notable number of outliers. Due to the client and server outliers, using Modbus over UDP for a low-latency dependent PES would not be recommended.

Table 4.2 shows the average reading and writing times for the client and server for both Modbus TCP/IP and Modbus over UDP. Modbus TCP/IP showed faster times than Modbus over UDP for every timing metric except for the server reading speed, where both protocols, averaged 47 µs.

Next, the throughput for the Modbus protocols needed to be determined. To find the throughput of Modbus, the amount of times per second that both the server and client could read and write a register needed to be determined. The lowest read/write value of the server or client limits the throughput of Modbus.

Figure 4.23 shows how many times a single Modbus client could be polled in one second for both Modbus TCP/IP and Modbus over UDP. From the figure, Modbus TCP/IP has a faster reading and writing. Modbus TCP/IP averaged 384 messages per second for reading and 391 messages per second for writing, while Modbus over UDP averaged 272 messages per second for reading and 275 messages per second for writing.

Next, the server reading and writing rates were determined. These values are shown in Figure 4.24. The server reading and writing message rates were much greater than the client reading and writing message rates due to the server hosting the data, and therefore no network communication is required for the server to read or write values. The Modbus TCP/IP server had an average of 26543 messages per second for reading and 25258 messages per second for writing, while the Modbus over UDP server had an average of 26872 messages per second for reading and 25109 messages per second for writing. Due to the server values being much higher than the client values, the throughput of Modbus is limited by the client's reading and writing rates. All Modbus timing averages are shown in Table 4.3. The theoretical minimum latency times are shown in Table 4.4. The directional throughputs are shown in Table 4.5.

TABLE 4.2: MODBUS AVERAGE TIMES TAKEN OVER 9000 MESSAGES

	TCP/IP		UDP	
Device	**Server**	**Client**	**Server**	**Client**
t_{read}	47 µs	3608 µs	47 µs	4713 µs
t_{write}	528 µs	3481 µs	575 µs	4607 µs

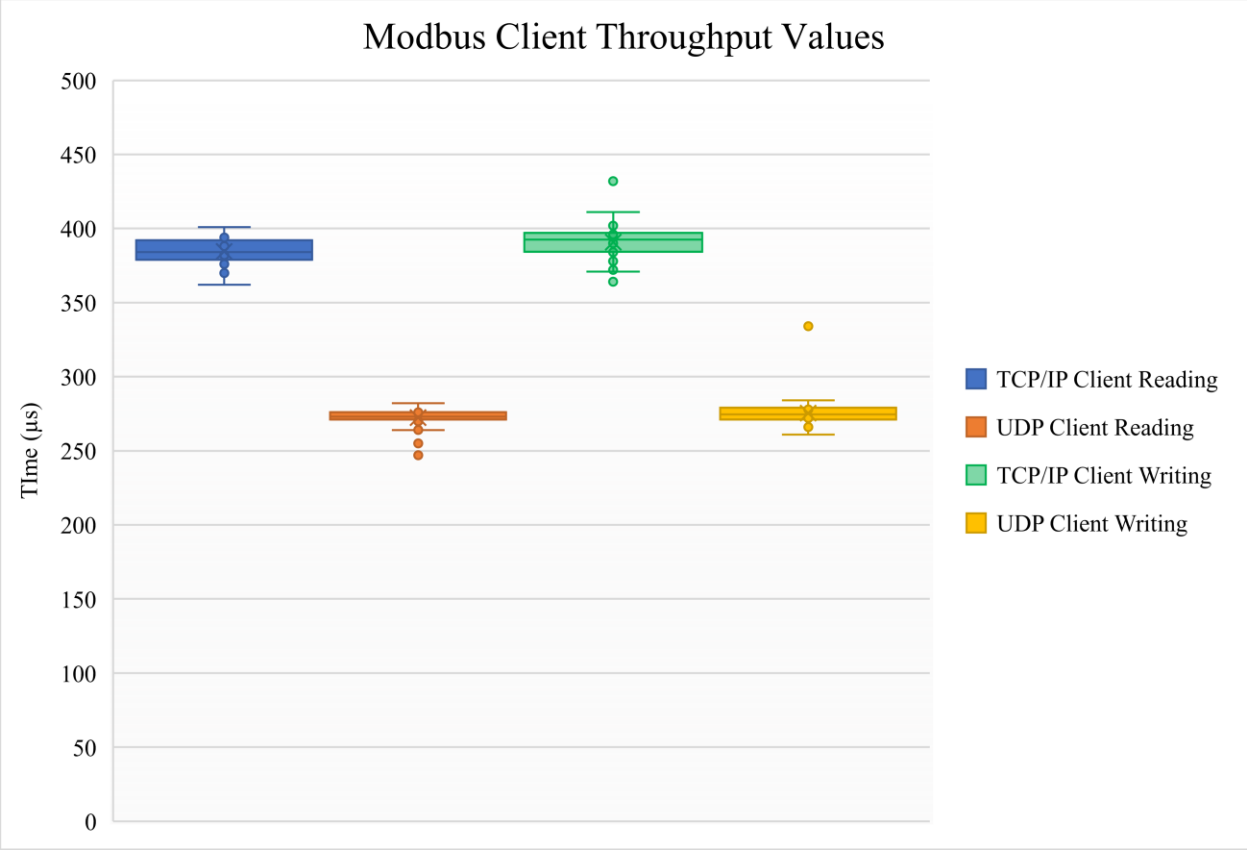


FIGURE 4.23: MODBUS CLIENT'S MAXIMUM REGISTERS READ IN ONE SECOND OVER TCP/IP AND UDP

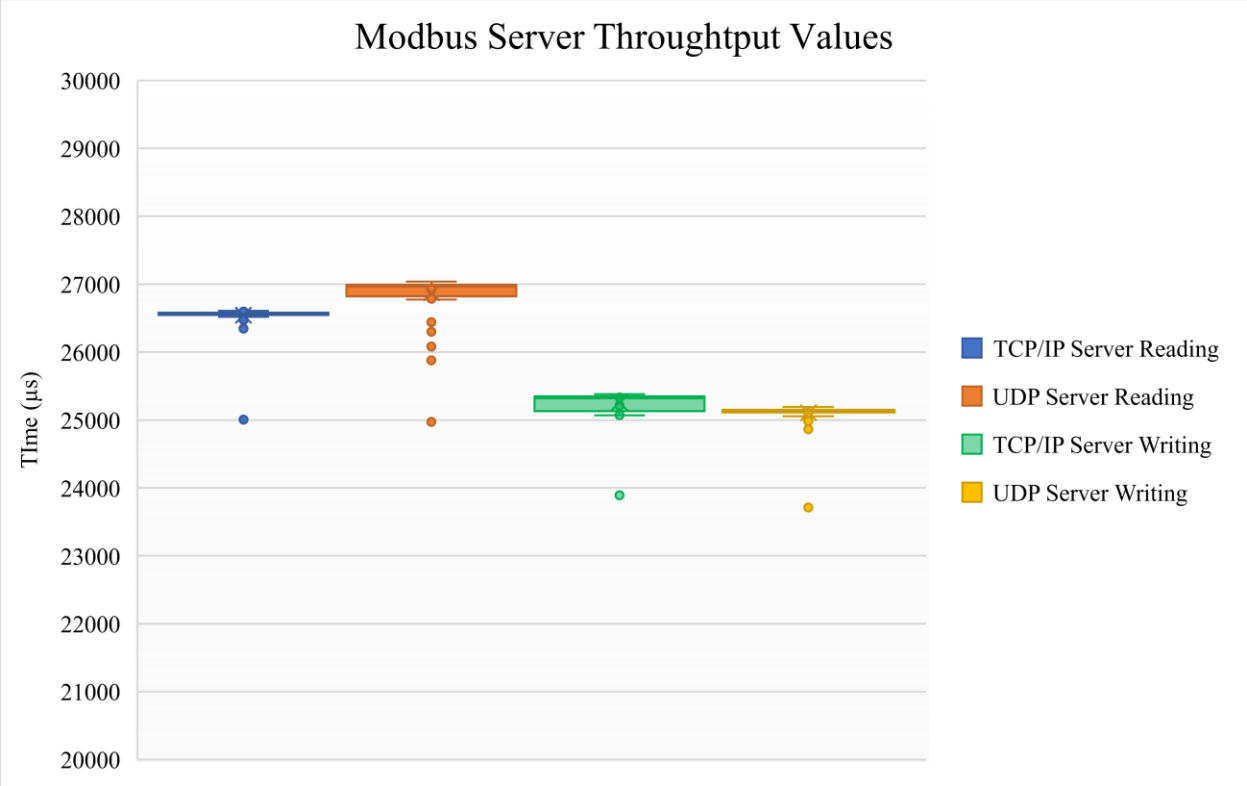


FIGURE 4.24: MODBUS CLIENT'S MAXIMUM REGISTERS WRITTEN TO IN ONE SECOND OVER TCP/IP AND UDP

TABLE 4.3: MODBUS AVERAGE READING AND WRITING MESSAGES PER SECOND OVER 9000 MESSAGES

	TCP/IP		**UDP**	
Device	**Server**	**Client**	**Server**	**Client**
Reading	26543 messages/sec	384 messages/sec	26872 messages/sec	272 messages/sec
Writing	25258 messages/sec	391 message/sec	25109 messages/sec	275 messages/sec

TABLE 4.4: MODBUS AVERAGE MESSAGE TIMES TAKEN OVER 9000 MESSAGES

	TCP/IP		**UDP**	
Data Flow Direction	**Server → Client**	**Client → Server**	**Server → Client**	**Client → Server**
$t_{message}$	4136 µs	3528 µs	5288 µs	4654 µs

TABLE 4.5: MAXIMUM MODBUS THROUGHPUT BASED ON DATA FLOW DIRECTION AND COMMUNICATION PROTOCOL

	TCP/IP		**UDP**	
Data Flow Direction	**Server → Client**	**Client → Server**	**Server → Client**	**Client → Server**
Throughput	384 messages/sec	391 message/sec	272 messages/sec	275 messages/sec

4.5 MQTT Testing

The last protocol examined in this chapter is MQTT. This message bus communication protocol has three levels of quality of service and security that gives nine different implementations to examine. Therefore, for each latency and throughput, nine different tests were run to quantify these characteristics of the protocol. The security measures that can be implemented within MQTT are username/password authentication and TSL encryption. MQTT can also have no security implemented.

The first tests run were MQTT latency with no security. These tests were run with a 1000 message sample, with the QOS being set to 0, 1, or 2. Figure 4.25 shows the latency times of the 1000 message test with no security and a QOS of 0. The average latency test observed was 3542 µs.

Figure 4.26 shows the latency times of the 1000 message test with no security and a QOS of 1. The average latency test observed was 3597 µs. This average was close to the latency measured with no security. This was expected, as the username/password security measure does not affect the data being communicated over the protocol.

Figure 4.27 shows the message times of the 1000 message test with no security and a QOS of 2. The average message time observed was 5788 µs. This average was higher than that of QOS 0 and QOS 1. This higher message time is due to the protocol verifying that the data is transmitted between the publishing and subscribed device, which is a quality of QOS 2.

The next tests run were with the MQTT username/password security. This security measure prevents a device from connecting to the MQTT broker without credentials, however, the messages are not protected once the broker is accessed. Figure 4.28 shows the latency times of 1000 messages with username/password security and a QOS of 0. The average message time was 3459 µs, which was very close to the average message time of MQTT with no security.

Figure 4.29 shows the latency times of the 1000 message test with username/password security and a QOS of 1. The average latency test observed was 3566 µs. Again, this average value was close to the no security message time average value.

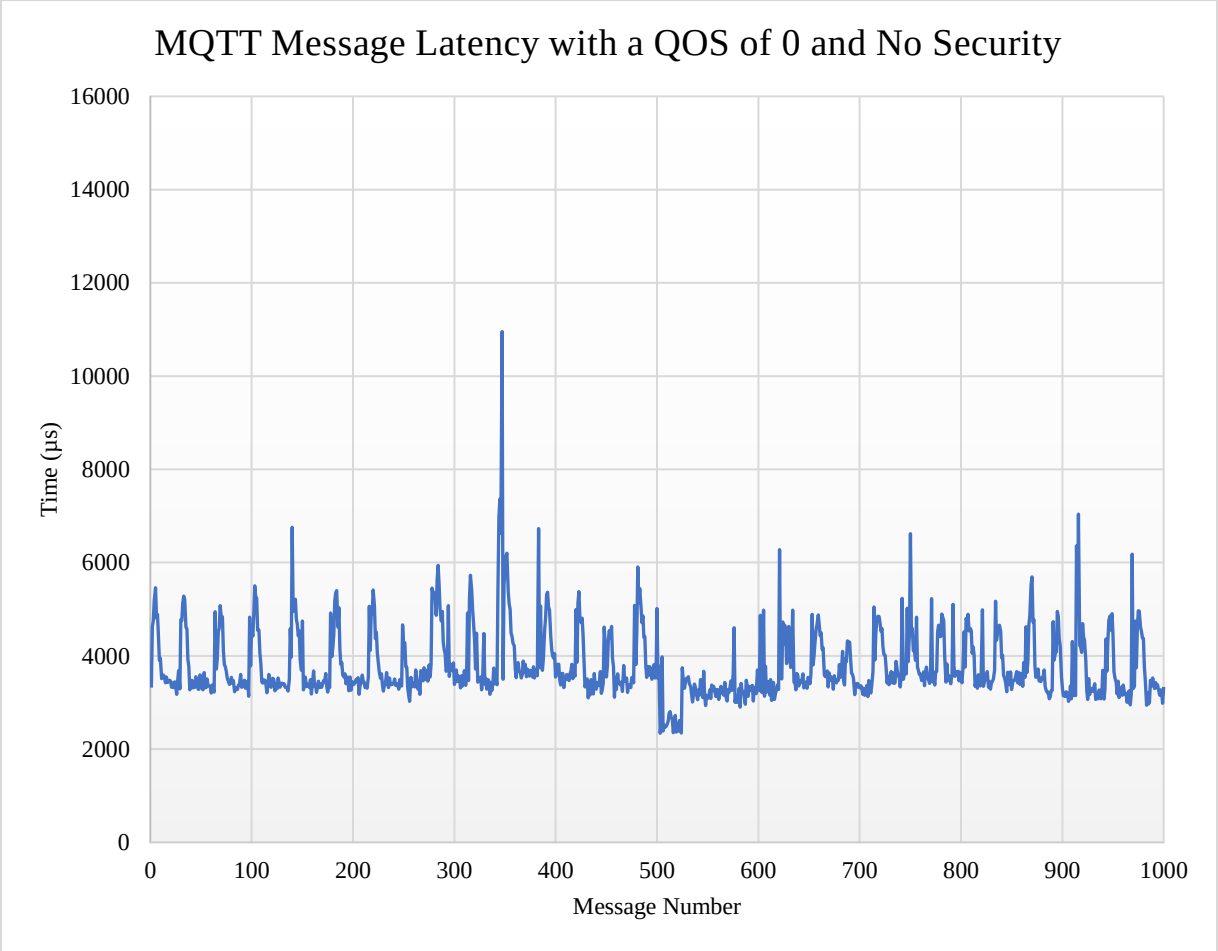


FIGURE 4.25: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 0

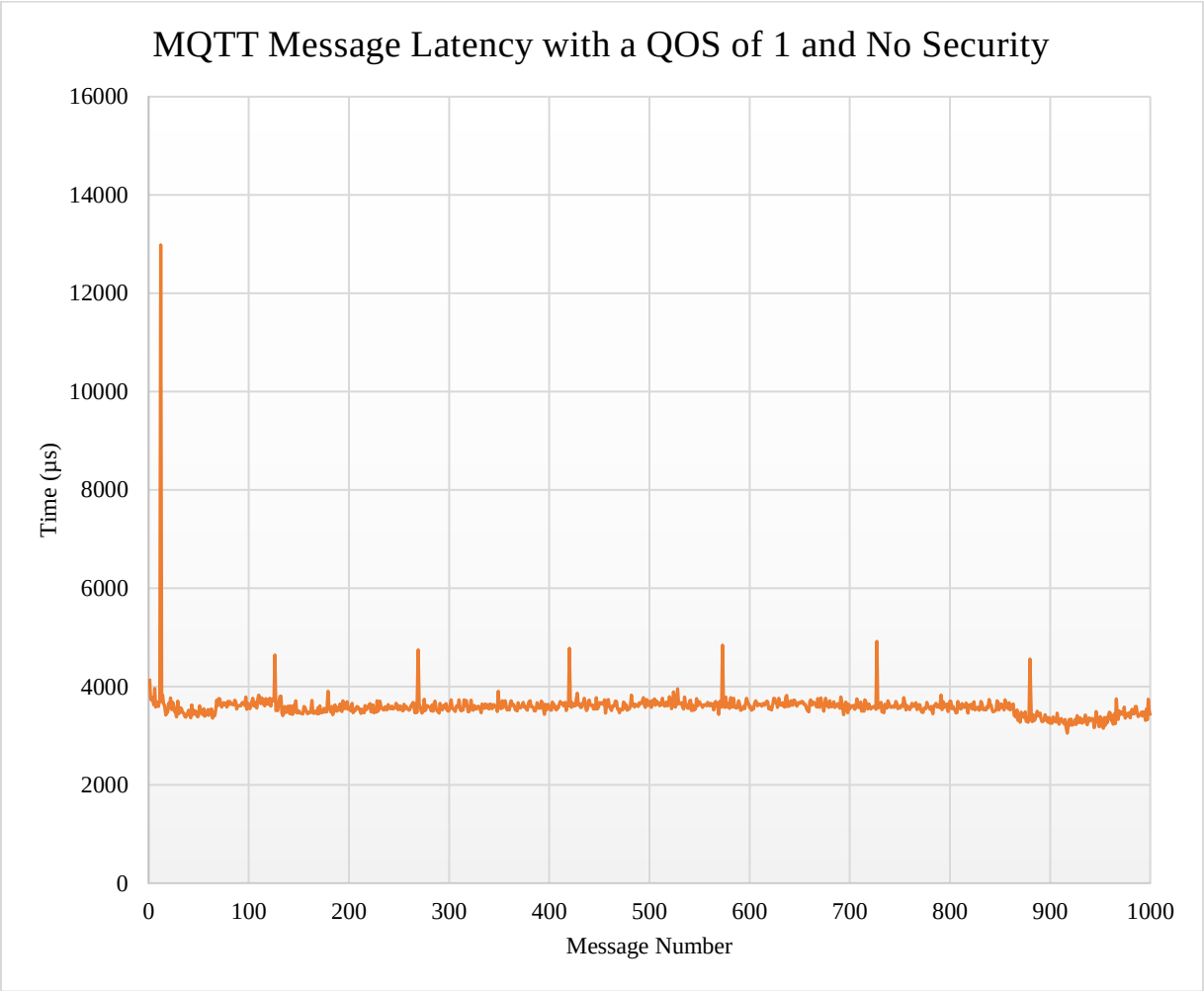


FIGURE 4.26: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 1

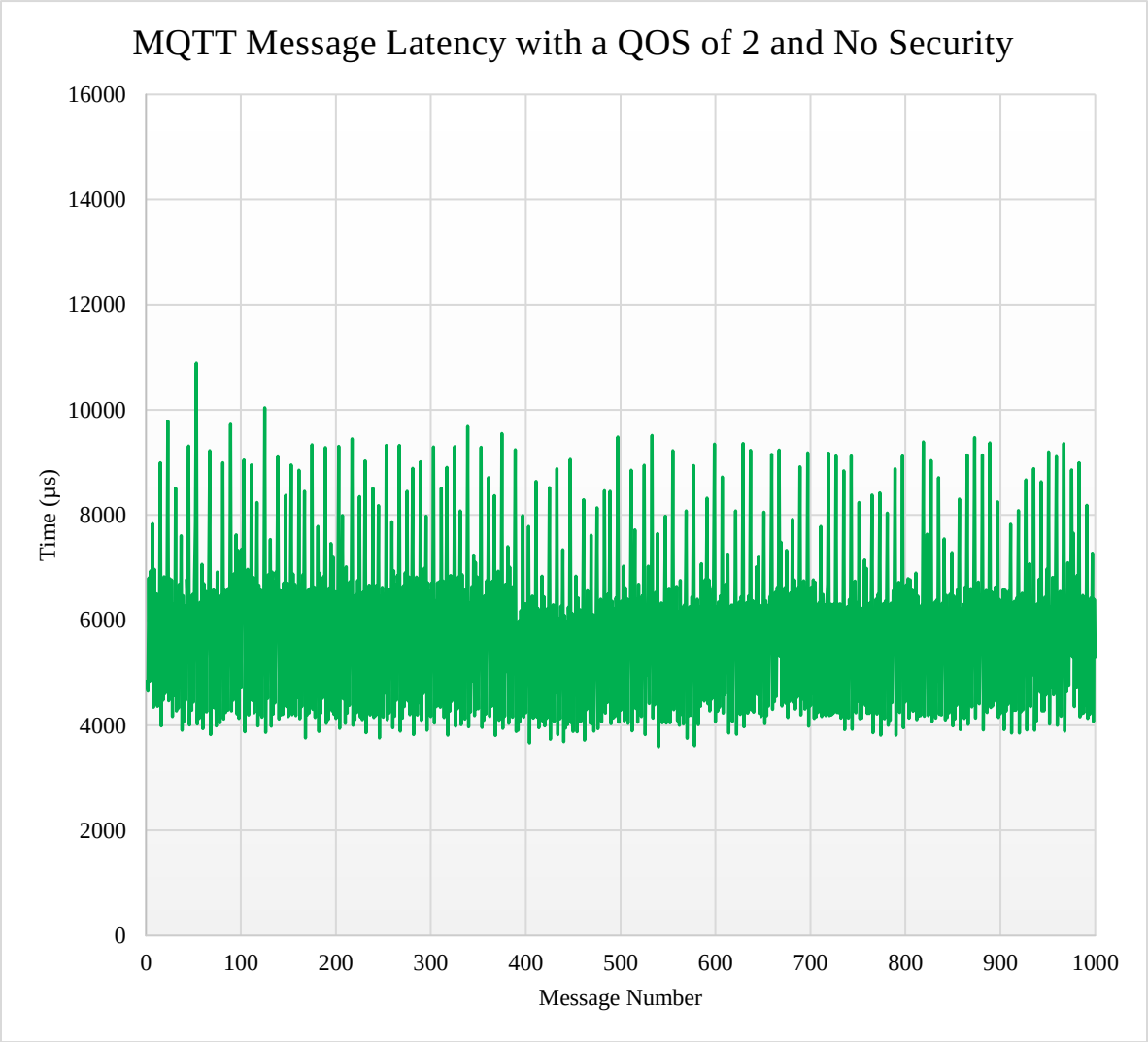


FIGURE 4.27: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 2

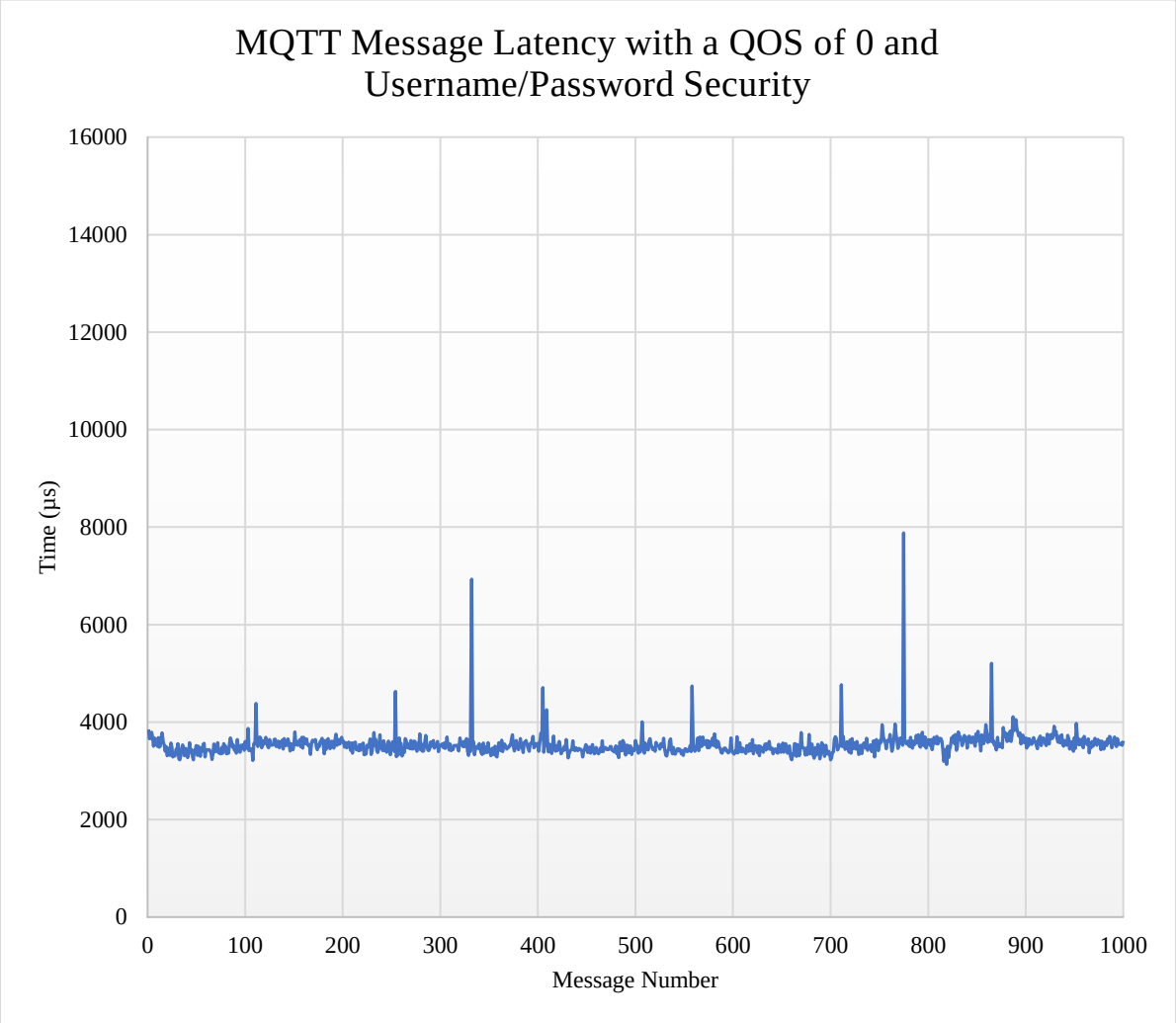


FIGURE 4.28: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 0 AND USERNAME/PASSWORD SECURITY

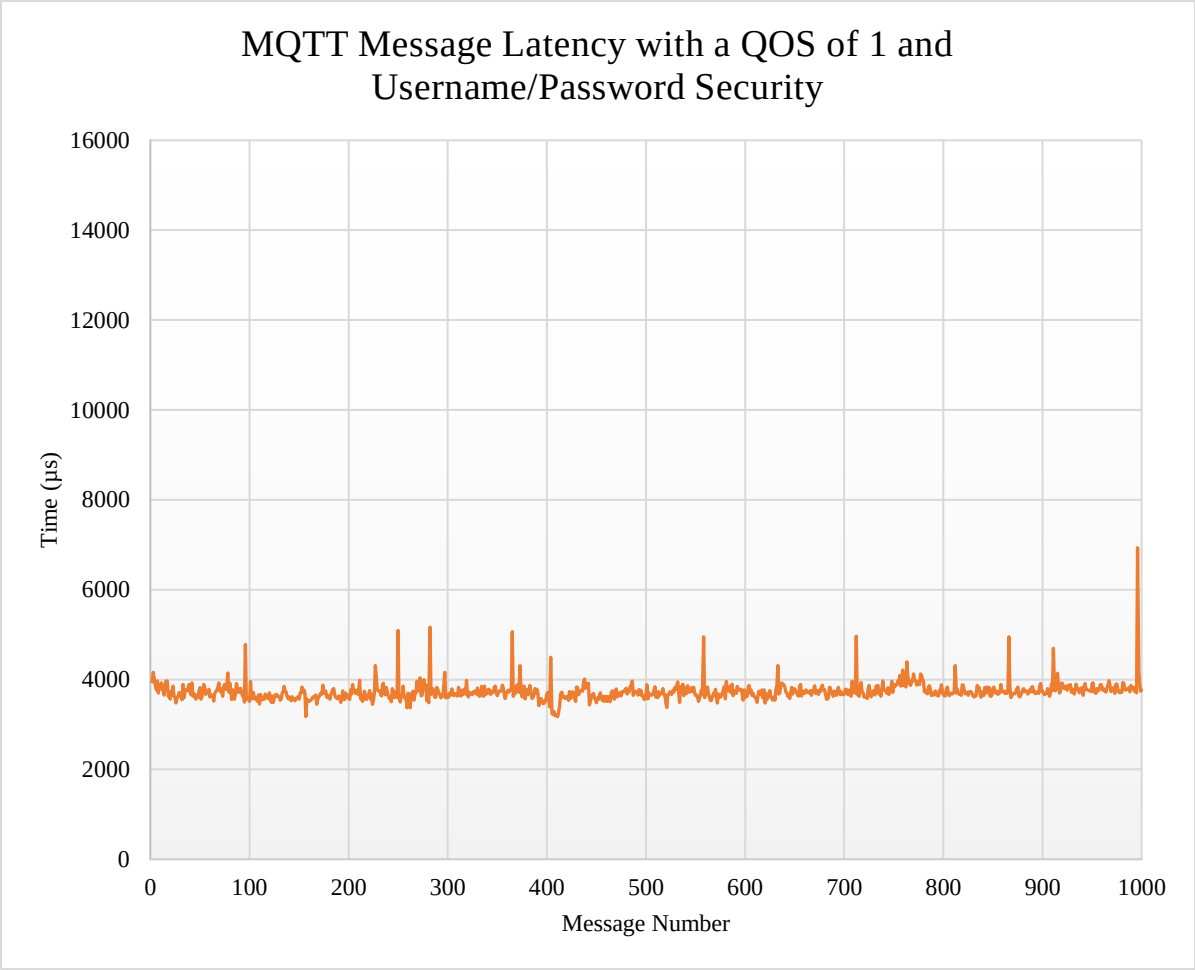


FIGURE 4.29: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 1 AND USERNAME/PASSWORD SECURITY

Figure 4.30 shows the latency times of the 1000 message test with username/password security and a QOS of 2. The average latency test observed was 5974 µs. Again, this average value was close to the no security message time average value. This test shows the message time remaining close to the average message time but had some extreme outliers that were an order of magnitude larger than the average value.

The last MQTT security implementation tested was TSL encryption. This security is important, as it encrypts the data before it is communicated across the network. Only devices with the correct security key can decrypt the data and read the message. Therefore, due to the time required to encrypt and decrypt the data, the message times are expected to be longer than the previous security implementation.

Figure 4.31 shows the message times with TSL security and a QOS of 0. The average message time was 3653µs. While this is higher than the previous average message time observed, this increase in message time is not substantial.

Figure 4.32 shows the message times of with TSL security and a QOS of 1. The average message time was 3764 µs. Again, the increase in message time is not substantial as compared to the other security implementations. An outlier around 25000 µs was observed, but most of the message time samples were close to the average message time.

Figure 4.33 shows the message times of TSL security with a QOS of 2. The average message time was 6410 µs. This message time was notably higher than previous MQTT message times with a QOS of 2 and other security implementations.

Figure 4.34 compares all the message times across the different MQTT security implementations and QOS levels. For this figure, each category has 9000 samples. From the figure, it can be noted that having a QOS of 2 affects the message time the greatest. Overall, the security implementation did not have a large effect on the MQTT message time.

The next test run with MQTT was throughput. For this test, it was measured how many times a message could be received from the publishing device in one second. For each MQTT security and QOS implementation, a test for one second was run 100 times. The results are shown in Figure 4.35. All latency and throughput averages are shown in Table 4.6.

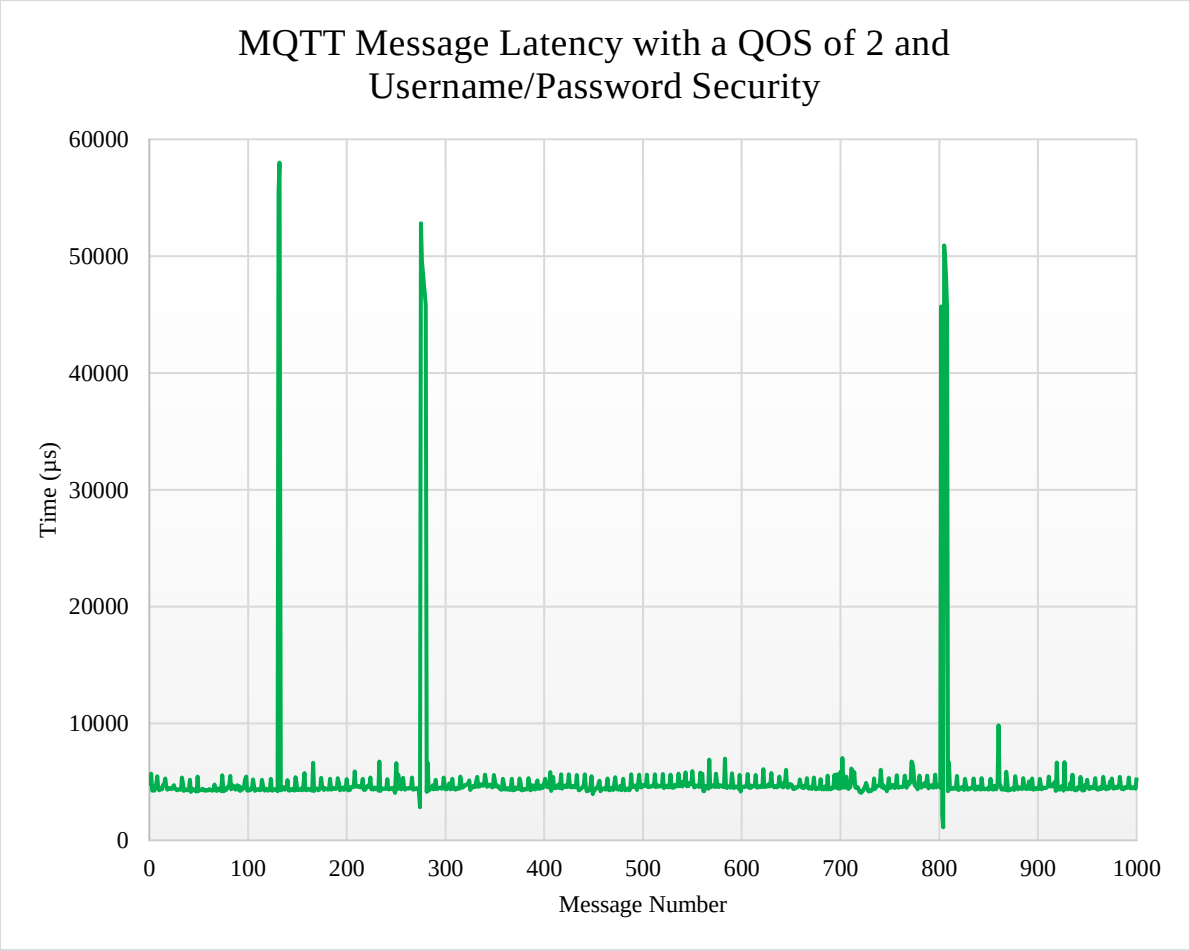


FIGURE 4.30: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 2 AND USERNAME/PASSWORD SECURITY

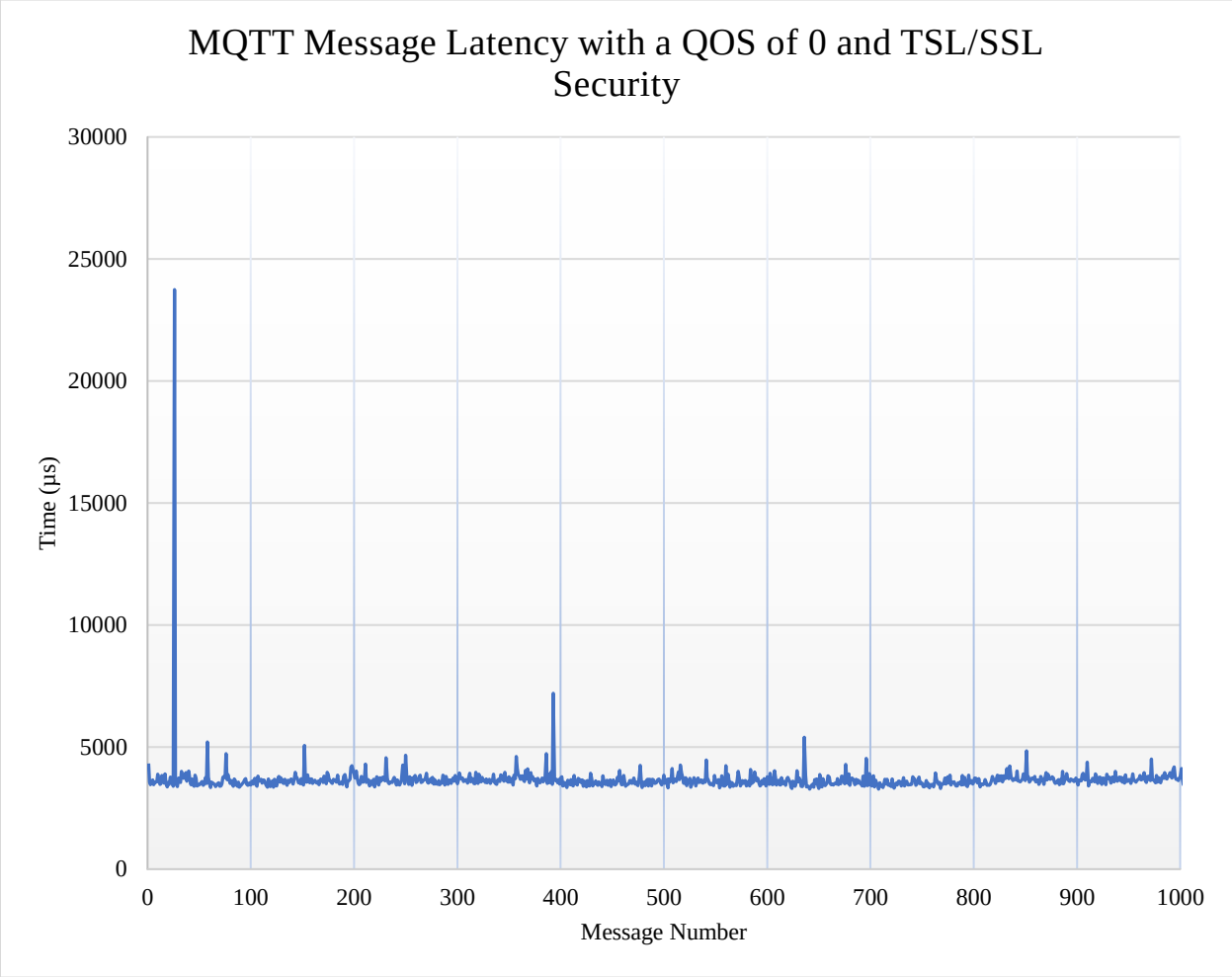


FIGURE 4.31: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 0 AND TSL/SSL SECURITY

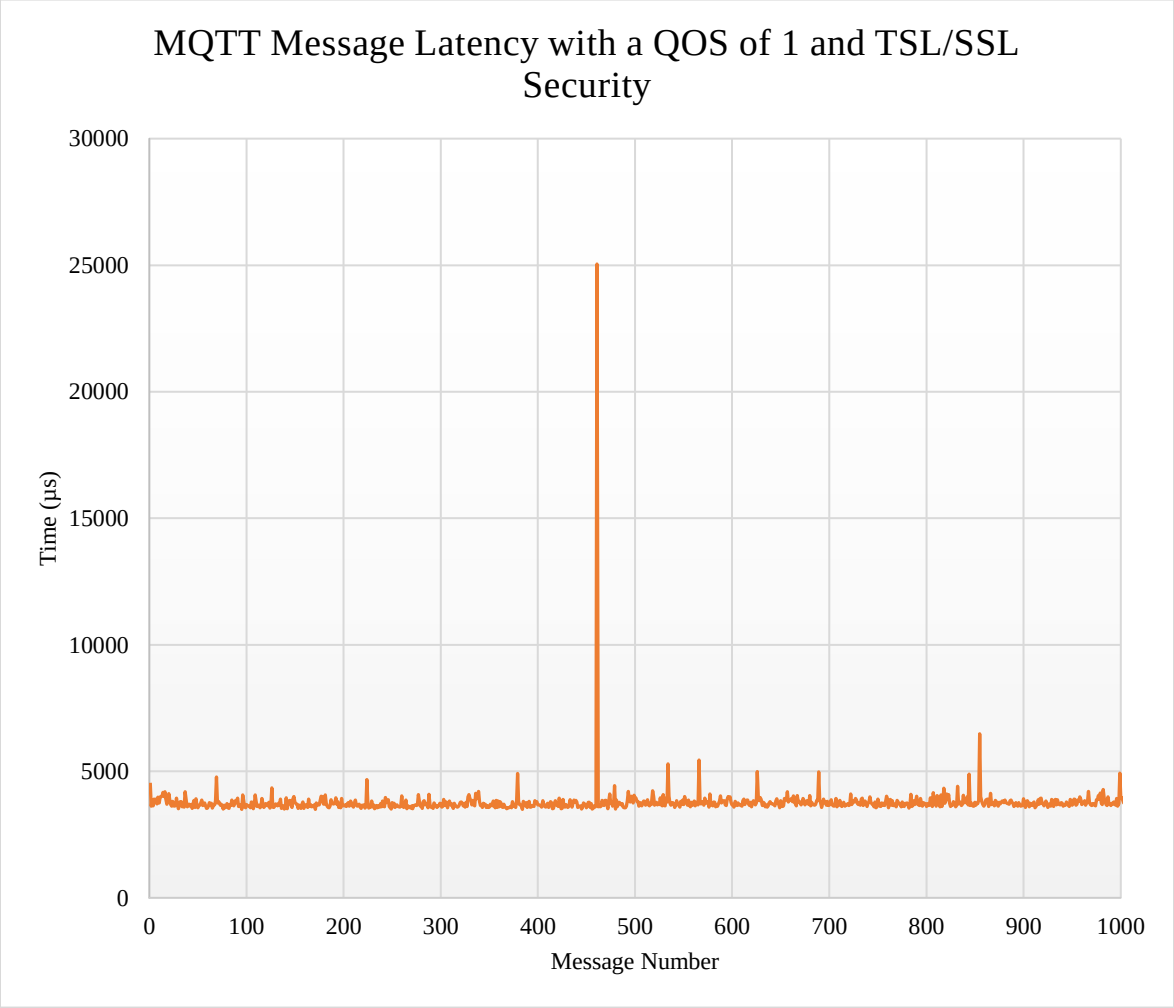


FIGURE 4.32: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 1 AND TSL/SSL SECURITY

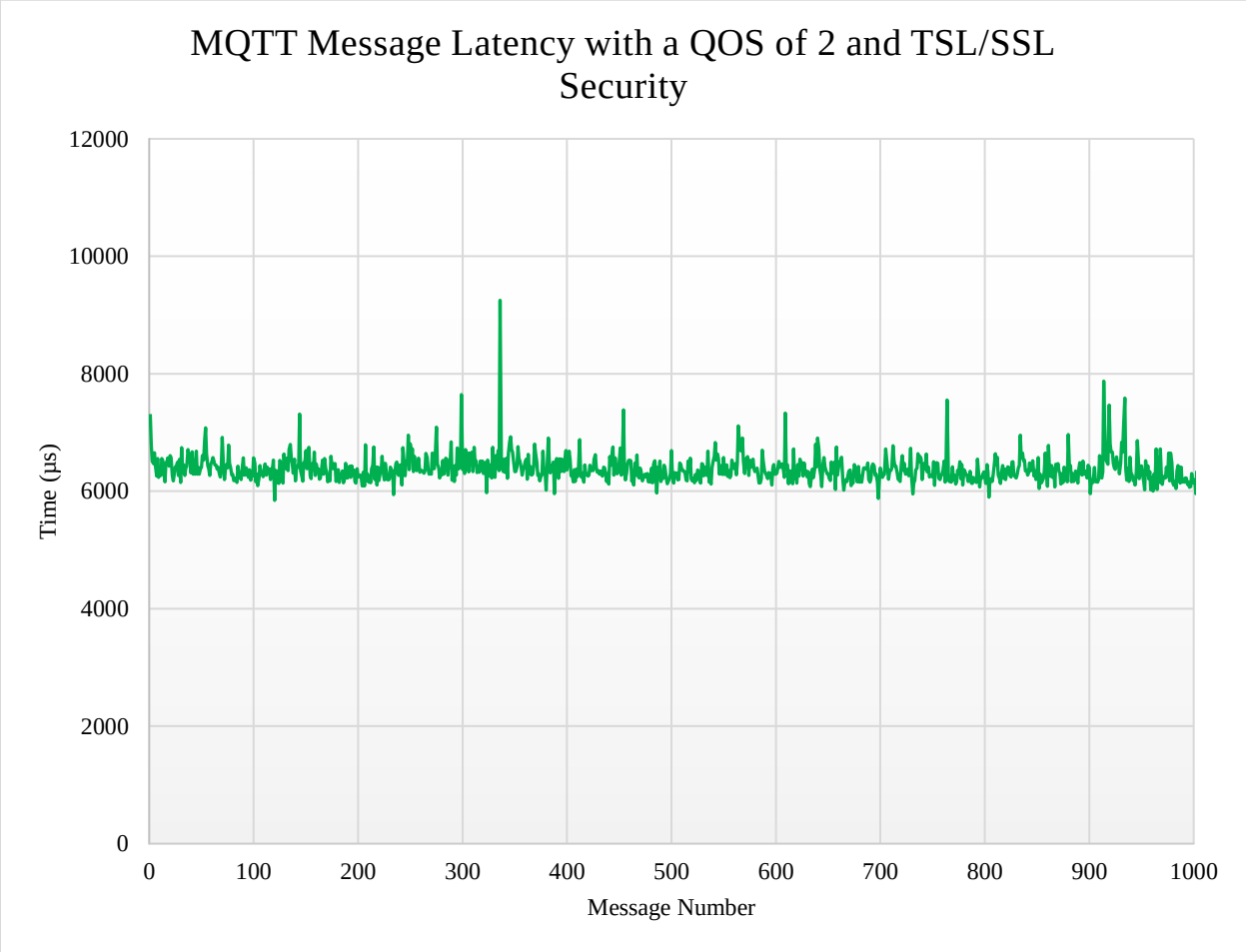


Figure 4.33: 1000 MESSAGE SAMPLE OF MQTT LATENCY WITH A QOS OF 2 AND TSL/SSL SECURITY

Figure 4.34: COMPARISON OF MQTT MESSAGE LATENCY ACROSS DIFFERENT SECURITY MEASURES AND QOS LEVELS

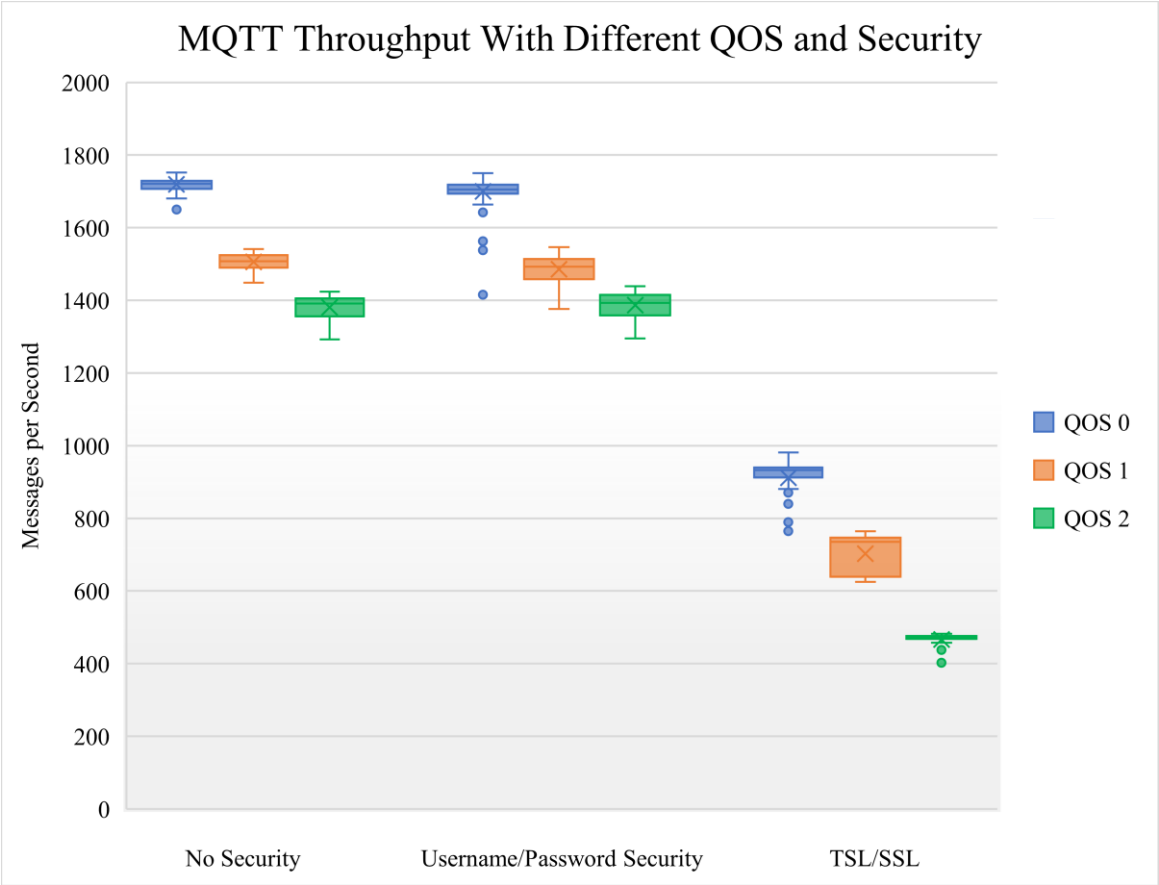


FIGURE 4.35: COMPARISON OF MQTT THROUGHPUT ACROSS DIFFERENT SECURITY MEASURES AND QOS LEVELS

TABLE 4.6: AVERAGE LATENCY AND THROUGHPUT FOR DIFFERENT MQTT QOS LEVELS AND SECURITY IMPLEMENTATIONS

	No Security			**Username/Password**			**TSL/SSL**		
QOS	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**
Latency	3542 µs	3597 µs	5788 µs	3459 µs	3566 µs	5974 µs	3653 µs	3764 µs	6410 µs
Throughput (mps)	1719	1506	1380	1700	1486	1387	912	703	466

Username/password security had little to no effect on MQTT's throughput, as it can be seen that the average values, are very close to one another. As previously mentioned, username/password security does not affect the communication traffic between devices, so it was expected that throughput would not change between these no security and username/password security. TSL security, however, greatly affected the throughput. For QOS 0, the average throughput dropped by 47% from no security to TSL security. For QOS 1, the throughput dropped by 53%, and for QOS 2, it dropped by 66%.

Further examination also shows that QOS 0 has the highest throughput, while QOS 1 has a lower throughput, and QOS 2 has the lowest. These are expected results, as QOS 1 requires delivery to be acknowledged by the broker, and QOS 2 requires delivery to be acknowledged by the broker as well as resends the messages if delivery is not acknowledged.

Changing the MQTT implementation causes a much greater change in throughput than it did in latency. The average variation on throughput observed with the changes in MQTT implementation was much larger than with latency. This impacts PES communication, as all data must travel through the message bus. When multiple subsystems are connected to the message bus and trying to communicate, the throughput will be greatly limited if TSL security and QOS 2 are implemented. Guaranteed, secure communication comes at the expense of throughput.

4.6 Plug-and-Play Considerations

When designing the custom UDP protocol, one important feature was "plug-and-play" functionality between devices. Any PES device, such as an inverter or DC-DC converter, that is compatible with this protocol can be connected to an agent and begin operation with no other setup. For a plug-and-play protocol to have maximum effectiveness, different versions should be forward and backwards compatible with one another. For example, if a power converter is a completed product with the "1.0" version of a plug-and-play protocol, it should be fully compatible with an agent system that has been updated to the "2.0" version of the protocol.

As the versions of a plug-and-play standard progress, additional values are added to be schema. These allow the functionally of the standard to expand. These functionalities could be features like control modes, measurements, etc. However, while older devices with an older standard version may not be able to take advantage of these new features, this newer standard should still be compatible with the older versions such that older versions would continue to operate.

Updated standards will still categorize data, and new values could be incorporated all throughout the updated schema. A consequence of this is the need to read values individually. The custom UDP protocol handles this by sending single messages, and receiver identifies the data by its “Type High” and “Type Low” values. If the receiver does not have a data designation for the “Type High” and “Type Low” values it receives, the data is simply ignored. Furthermore, this allows for the schema of the custom UDP protocol to not be organized in a sequential manner.

Because of this property of plug-and-play communication protocols, the Modbus registers were polled individually. However, Modbus has the capability to read multiple registers at once. By polling multiple registers, the throughput of the protocol is increased. Future work with plug-and-play protocols may transition to using Modbus polling multiple registers to increase throughput. Therefore, a determination is needed for how fast registers must be polled to have a higher throughput than the custom UDP protocol. Figure 4.37 shows the average time to poll multiple registers at once in Modbus TCP/IP and Modbus over UDP. For each number of registers read, the test was repeated 1000 times, and then the values averaged. Figure 4.38 compared the Modbus multiple register reading speed with the message rate of the custom UDP protocol. For Modbus TCP/IP, five registers could be polled in the same time that the custom UDP protocol sends five messages. This is the “breakeven point” for throughput between the custom UDP protocol and Modbus TCP/IP. For Modbus over UDP, this breakeven point was ten registers.

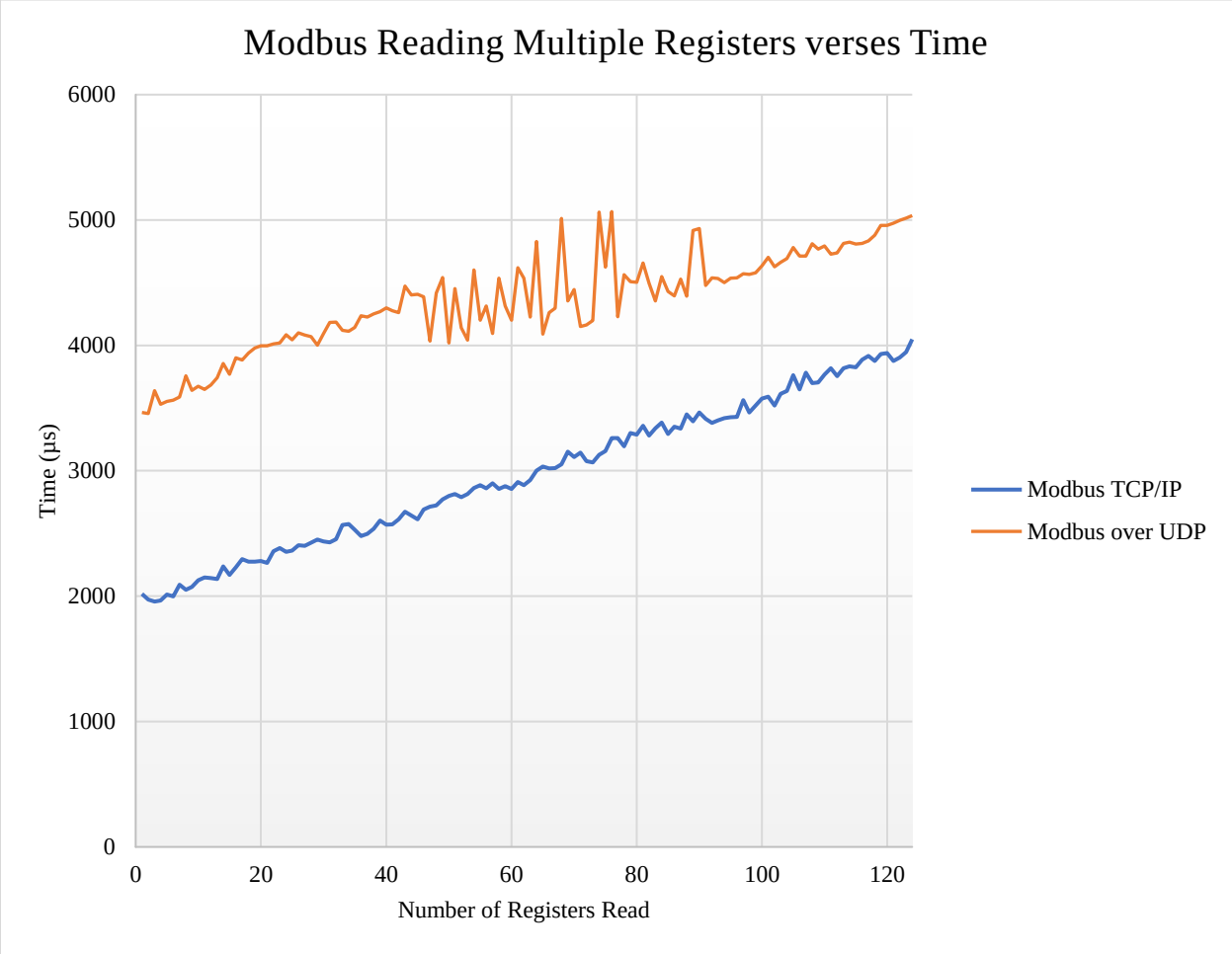


FIGURE 4.36: READING MULTIPLE REGISTERS AT ONCE WITH A MODBUS CLIENT

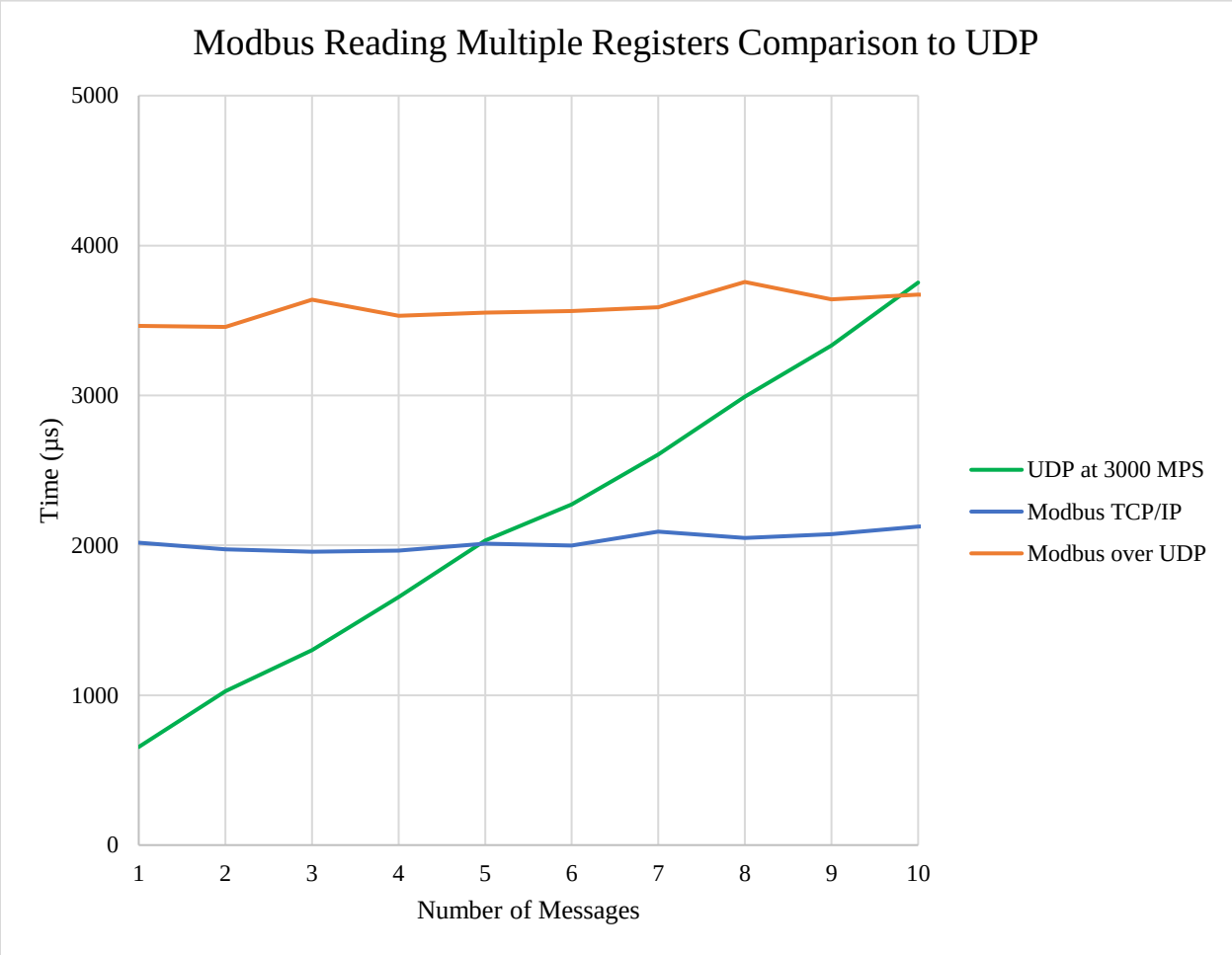


FIGURE 4.37: READING MULTIPLE REGISTERS AT ONCE WITH A MODBUS CLIENT COMPARED WITH THE CUSTOM UDP PROTOCOL

4.7 Chapter Summary

The results presented in this chapter show the latency and throughput of three communication protocols utilized in PES. It demonstrated the superior speed and throughput of the custom UDP protocol in comparison to the established, industry stand protocol of Modbus. Furthermore, it shows the latency and throughput of MQTT with different QOS levels and security measures. These higher QOS levels and security measures resulted in slightly slower latencies. However, they had a large impact on the throughput of MQTT. This is important to consider, as all messages in an agent-based PES are subject to the throughput of the message bus software.

Furthermore, plug-and-play considerations were given to the communication protocols. The need to individually poll messages was explained. Consideration to Modbus's capability to poll multiple registers was explored. The breakeven point for Modbus's throughput to surpass the custom UDP protocols' throughput was found.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

As the power grid continues to adopt increasingly modern power electronic technologies, exploring and understanding their behavior and operation is increasingly more important. An example of this modern power electronic technology is the agent-based PES, which relies on the decentralized control of multiple “agents” in a controlled environment. Therefore, a testbed was created to emulate this controlled environment and allow for the development, testing, and characterization of PES intrasystem communication. This testbed consists or multiple Raspberry Pi computers networked together, with a central controller that provides for communication code to be executed and data to collected about the behavior of the communication.

The previous chapter demonstrates how different communication protocols greatly vary in their operation inside a PES. Therefore, it is of the utmost importance to properly quantify the characteristics of the communication protocol if it is incorporated in a PES. The tests performed measured latency and throughput of these communication protocols. For DSP to agent communication, it was demonstrated how the custom UDP protocol outperformed Modbus TCP/IP and Modbus over UDP in terms of latency and throughput. The custom UDP protocol was able to reach a message rate of 3000 messages per second, which was a large improvement over the Modbus TCP/IP rate of 384 messages per second or the Modbus over UDP rate of 272 messages per second when reading messages individually. Furthermore, the latency of the custom UDP protocol was found to be faster than that of Modbus TCP/IP and Modbus over UDP. By using the custom UDP protocol, data can be communicated faster and in greater volumes than with Modbus.

Furthermore, the message bus protocol MQTT was fully quantified by finding the latency and throughput with different QOS levels and security measures. It was found that QOS levels 0 and 1 do not have a meaningful difference in latency, while a QOS level 2 was found to raise the latency time. It was also found that a QOS level of 2 caused a small percentage of messages to take much longer to be delivered. Therefore, if message integrity is a higher priory in a PES

design than message speed, QOS 2 should be used. If message speed is a higher priority, QOS 1 should be used.

Also demonstrated was the different security measures available with MQTT. The username/password security measure did not have a notable impact on latency or throughput as compared to the MQTT implementation with no security. This is due to be fact that username/password security restricts what MQTT clients can connect to the broker; it does not alter the MQTT message's network traffic. However, TSL/SSL security encrypts and decrypts the messages, and therefore latency times were higher, and throughput was lower. The increase in latency times was not significant, except when QOS 2 was used. QOS 2 with TSL security increased the latency by approximately 1000 µs, which could impact PES operation.

The throughput of MQTT is an important consideration due to all agent-based PES communication being facilitated through it. As messages are published, they are queued in a "first in, first out" manner. Therefore, if messages are being published at a faster rate than the MQTT implementation's throughput, messages will take longer to send and run the risk of being dropped. Therefore, the message rates of all agents need to be considered to avoid a message backlog on the broker.

The last subject covered was plug-and-play considerations in PES. This section covered why individual message polling was used in testing, as it allows for multiple versions of plug-and-play schemas to be compatible with one another. However, considerations for multiple register were discussed, and the time required to poll multiple registers in Modbus TCP/IP and Modbus over UDP were found. This opens Modbus to being used in future plug-and-play communication implementations, as the potential to have a higher message throughput when multiple registers are read at once is available.

5.2 Future Work

In this thesis, the Raspberry Pi 3B was utilized. It is recommended in the future to repeats these tests on a Raspberry Pi 4, which has superior computational capabilities. Other future work recommended includes testing additional communication protocols, such as Zero Message Queuing (ZMQ). UDP has a similar implementation that encrypts messages in the same manner

as TCP/IP TSL/SSL security. It is recommended to implement this secure UDP transport layer with the custom UDP protocol to test secure DSP to Agent communication. Further recommendations include a greater bit size on the custom UDP protocol to increase data resolution. While this should have the effect of lowering the throughput, this would be allowable as it could match with the throughput of MQTT and allow for more data transfer.

Further communications research for PES systems involves examining different communication mediums. Legacy communication protocols, such as RS-232, could be used as an alternate to ethernet based communication. Also, wireless communication could be characterized, which would allow for more flexibility in designing future PESs.

References

[1] L. M. Tolbert, T. J.King, B.Ozpineci, J. B.Campbell, G.Muralidharan, D. T.Rizy, A. S.Sabau, H.Zhang, W.Zhang, X.Yu, H. F.Huq, H.Liu, "Power Electronics for Distributed Energy Systems and Transmission and Distribution Applications," *ORNL/TM-2005/230*, no. December, pp. 65–91, 2005.

[2] B. Dean, M. Starke, M. Smith, M. Chinthavli, and L. Tolbert, "A Communication Testbed for Testing Power Electronic Agent Systems," *IEEE Power Energy Soc. Innov. Smart Grid Technol. Conf.*, pp. 1–5, 2021.

[3] A. Rufer, "Today's and tomorrow's meaning of power electronics within the grid interconnection," in *2007 European Conference on Power Electronics and Applications*, 2007, pp. 1–11, doi: 10.1109/EPE.2007.4417785.

[4] "IEEE Power Electronics." https://ewh.ieee.org/soc/pels/home/Control-Theory.php (accessed Jan. 04, 2021).

[5] W. Group, *IEEE Application Guide for IEEE Std 1547, IEEE Standard for Interconnecting Distributed Resources with Electric Power Systems*, no. April. 2008, pp. 1–207.

[6] W. Xiong, J. Zeng, L. Wu, and H. Cheng, "Power management of a residential hybrid photovoltaic inverter with battery energy storage system," *PEDG 2019 - 2019 IEEE 10th Int. Symp. Power Electron. Distrib. Gener. Syst.*, pp. 450–454, 2019, doi: 10.1109/PEDG.2019.8807638.

[7] "Combining Systems | Powerwall Support." https://www.tesla.com/support/energy/powerwall/learn/combining-systems (accessed May 28, 2020).

[8] N. A. Z. Musa, M. Z. M. Yusoff, R. Ismail, and Y. Yusoff, "Issues and challenges of forensics analysis of agents' behavior in multi-Agent systems: A critical review," *2015 Int. Symp. Agents, Multi-Agent Syst. Robot. ISAMSR 2015*, pp. 122–125, 2016, doi: 10.1109/ISAMSR.2015.7379782.

[9] S. Singh and N. Singh, "Internet of Things (IoT): Security challenges, business opportunities & reference architecture for E-commerce," in *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, Oct. 2015, pp. 1577–1581, doi: 10.1109/ICGCIoT.2015.7380718.

[10] M. Rouse, "What is IoT (Internet of Things) and How Does It Work?," *Cyber Resil. Syst. Networks*, pp. 1–150, 2009, Accessed: Dec. 28, 2020. [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

[11] D. Couloumb, C. El Kaed, A. Garg, C. Healey, J. Healey, and S. Sheehan, "Energy efficiency driven by a storage model and analytics on a multi-system semantic integration," *Proc. - 2017 IEEE Int. Conf. Big Data, Big Data 2017*, vol. 2018-Janua, pp. 4555–4561, 2017, doi: 10.1109/BigData.2017.8258498.

[12] K. P. Schneider, S. Laval, J. Hansen, R. B. Melton, L. Ponder, L. Fox, J. Hart, J.

Hambrick, M. Buckner, M. Baggu, K. Prabakar, M. Manjrekar, S. Essakiappan, L. M. Tolbert, Y. Liu, J. Dong, L. Zhu, A. Smallwood, A. Jayantilal, C. Irwin, G. Yuan,, *al* "A distributed power system control architecture for improved distribution system resiliency," *IEEE Access*, vol. 7, pp. 9957–9970, 2019, doi: 10.1109/ACCESS.2019.2891368.

[13] R. E. Mackiewicz, "Overview of IEC 61850 and Benefits," in *2006 IEEE PES Power Systems Conference and Exposition*, 2006, pp. 623–630, doi: 10.1109/PSCE.2006.296392.

[14] W. Huang, "Learn IEC 61850 configuration in 30 minutes," *71st Annu. Conf. Prot. Relay Eng. CPRE 2018*, vol. 2018-Janua, pp. 1–5, 2018, doi: 10.1109/CPRE.2018.8349803.

[15] T. S. Ustun and S. M. S. Hussain, "Implementation and performance evaluation of IEC 61850 based home energy management system," *2019 IEEE 8th Glob. Conf. Consum. Electron. GCCE 2019*, pp. 24–25, 2019, doi: 10.1109/GCCE46687.2019.9015222.

[16] D. Mascarella, M. Chlela, G. Joos, and P. Venne, "Real-time testing of power control implemented with IEC 61850 GOOSE messaging in wind farms featuring energy storage," *2015 IEEE Energy Convers. Congr. Expo. ECCE 2015*, pp. 6710–6715, 2015, doi: 10.1109/ECCE.2015.7310599.

[17] C. Zanabria, F. P. Andren, J. Kathan, and T. Strasser, "Towards an integrated development of control applications for multi-functional energy storages," *IEEE Int. Conf. Emerg. Technol. Fact. Autom. ETFA*, vol. 2016-Novem, 2016, doi: 10.1109/ETFA.2016.7733617.

[18] P. Palensky, E. Widl, and A. Elsheikh, "Simulating Cyber-Physical Energy Systems: Challenges, Tools and Methods," *IEEE Trans. Syst. Man, Cybern. Syst.*, vol. 44, no. 3, pp. 318–326, 2014, doi: 10.1109/TSMCC.2013.2265739.

[19] K. Matsumura, M. Marmiroli, and Y. Tsukamoto, "Smart grid for distributed resources and demand response integration," *IET Conf. Publ.*, vol. 2018, no. CP757, 2018, doi: 10.1049/cp.2018.1749.

[20] "OpenADR." https://sites.google.com/a/nestfield.co.kr/nestfield/home_e/openadr (accessed Dec. 30, 2020).

[21] Robby Simpson, "An Overview of Smart Energy Profile 2 . 0," *GE Digit. Energy*, pp. 1–22, 2013, [Online]. Available: hes-standards.org/doc/SC25_WG1_N1612.pdf.

[22] Y. Lu, Y. Ding, X. Li, Q. Duan, and Y. C. Tian, "Upper-middleware development of smart energy profile 2.0 for demand-side communications in smart grid," *Proc. IECON 2018 - 44th Annu. Conf. IEEE Ind. Electron. Soc.*, pp. 306–310, 2018, doi: 10.1109/IECON.2018.8591771.

[23] "An introduction to ZigBee Smart Energy Profile 2.0." https://archive.eetasia.com/www.eetasia.com/ART_8800685052_590626_TA_a3e1e20e.HTM (accessed Dec. 30, 2020).

[24] M. K. Ferst, H. F. M. De Figueiredo, and G. W. Denardin, "Connection Time in Modbus/TLS for Secure Communications on Photovoltaic Systems," *2019 IEEE 15th*

Brazilian Power Electron. Conf. 5th IEEE South. Power Electron. Conf. COBEP/SPEC 2019, 2019, doi: 10.1109/COBEP/SPEC44138.2019.9065406.

[25] I. Specification, "SunSpec Technology Overview," p. 8, 2015, [Online]. Available: https://sunspec.org/wp-content/uploads/2015/06/SunSpec-Techonology-Overview-12040.pdf?pdf=%22technology-overview%22.

[26] "MESA Standards | MESA Standards." http://mesastandards.org/mesa-standards/ (accessed Aug. 09, 2020).

[27] G. A. Donahue, *Network Warrior, Second Edition*. 2007.

[28] "IEEE 802.1 Working Group." https://1.ieee802.org/ (accessed Dec. 29, 2020).

[29] S. Katipamula, J. Haack, G. Hernandez, B. Akyol, and J. Hagerman, "VOLTTRON: An Open-Source Software Platform of the Future," *IEEE Electrif. Mag.*, vol. 4, no. 4, pp. 15–22, 2016, doi: 10.1109/MELE.2016.2614178.

[30] N. Singh, I. Elamvazuthi, P. Nallagownden, G. Ramasamy, and A. Jangra, "VOLTTRON based Multi-agent System for Residential Micro-grid," *2018 IEEE 4th Int. Symp. Robot. Manuf. Autom. ROMA 2018*, pp. 1–6, 2018, doi: 10.1109/ROMA46407.2018.8986726.

[31] J. Haack, B. Akyol, N. Tenney, B. Carpenter, R. Pratt, and T. Carroll, "VOLTTRON™: An agent platform for integrating electric vehicles and Smart Grid," *2013 Int. Conf. Connect. Veh. Expo, ICCVE 2013 - Proc.*, pp. 81–86, 2013, doi: 10.1109/ICCVE.2013.6799774.

[32] M. T. Smith, M. R. Starke, M. Chinthavali, and L. M. Tolbert, "Architecture for Utility-Scale Multi-Chemistry Battery Energy Storage," *2019 IEEE Energy Convers. Congr. Expo. ECCE 2019*, pp. 5386–5392, 2019, doi: 10.1109/ECCE.2019.8912309.

[33] "MQTT Version 5.0," 2018.

[34] O. Sadio, I. Ngom, and C. Lishou, "Lightweight Security Scheme for MQTT/MQTT-SN Protocol," *2019 6th Int. Conf. Internet Things Syst. Manag. Secur. IOTSMS 2019*, pp. 119–123, 2019, doi: 10.1109/IOTSMS48152.2019.8939177.

[35] "Quality of service and connection management." https://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/bc62020_.htm (accessed Dec. 29, 2020).

[36] Y. Yang, L. Zhang, and X. Wang, "Time delay performance analysis of distributed communication platform based on ZeroMQ," *Proc. - 2019 Int. Conf. Commun. Inf. Syst. Comput. Eng. CISCE 2019*, pp. 319–323, 2019, doi: 10.1109/CISCE.2019.00078.

[37] M. Starke, R. Zeng, S. Zheng, M. Smith, M. Chinthavali, Z. Wang, B. Dean, L. M. Tolbert, "A Multi-Agent System Concept for Rapid Energy Storage Development," *2019 IEEE Power Energy Soc. Innov. Smart Grid Technol. Conf. ISGT 2019*, pp. 1–5, 2019, doi: 10.1109/ISGT.2019.8791563.

[38] J. Stranahan, T. Soni, and V. Heydari, "Supervisory control and data acquisition testbed

for research and education," *2019 IEEE 9th Annu. Comput. Commun. Work. Conf. CCWC 2019*, pp. 85–89, 2019, doi: 10.1109/CCWC.2019.8666482.

[39] M. de Sousa and P. Portugal, "Modbus Application Protocol," *Ind. Commun. Syst.*, p. 36, 2016, doi: 10.1201/9781420041682.ch18.

[40] C. G. Bell, A. N. Habermann, J. McCredie, R. Rutledge, and W. Wulf, *Computer Networking: A Top-Down Approach*, vol. 3, no. 5. 2013.

[41] J. Stranahan, T. Soni, and V. Heydari, "Supervisory Control and Data Acquisition Testbed Vulnerabilities and Attacks," *Conf. Proc. - IEEE SOUTHEASTCON*, vol. 2019-April, 2019, doi: 10.1109/SoutheastCon42311.2019.9020436.

[42] M. J. Stanovich, I. Leonard, K. Sanjeev, M. Steurer, T.S. Jackson, M. Bruce, "Development of a smart-grid cyber-physical systems testbed," *2013 IEEE PES Innov. Smart Grid Technol. Conf. ISGT 2013*, pp. 6–11, 2013, doi: 10.1109/ISGT.2013.6497874.

[43] "Raspberry Pi 3 Model B." https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (accessed Jan. 26, 2021).

[44] M. Smith, "Evaluation of IEEE 802.1 Time Sensitive Networking Performance for Microgrid and Smart Grid Power System Applications," *Masters Theses*, p. 128, 2018, [Online].

[45] M. Starke, N. Sawyer, B. Dean, M. Smith, G. Liu, D. Spiers, B. Schultz, H. Harman, "Development and analysis of an energy storage sizing tool for residential deployment," 2017 IEEE Power & Energy Society General Meeting, Chicago, IL, USA, 2017, pp. 1-5.

[46] M. Starke, M. Chinthavali, S. Zheng, S. Campbell, R. Zeng, M. Smith, B. Dean, "Residential (Secondary-Use) Energy Storage System with Modular Software and Hardware Power Electronic Interfaces," 2019 IEEE Energy Conversion Congress and Exposition (ECCE), Baltimore, MD, USA, 2019, pp. 2445-2451.

[47] M. Starke, M. Chinthavali, S. Zheng, S. Campbell, R. Zeng, M. Smith, T. Kuruganti, B. Dean, "Agent-Based Framework for Supporting Behind the Meter Transactive Power Electronic Systems," 2020 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), Washington, DC, USA, 2020, pp. 1-5.

[48] M. Starke, P. Bhowmik, S. Campbell, M. Chinthavali, B. Xiao, R. Moorthy, B. Dean, J. Choi"A Plug-and-Play Design Suite of Converters for the Electric Grid," 2020 IEEE Energy Conversion Congress and Exposition (ECCE), Detroit, MI, USA, 2020, pp. 2314-2321, doi: 10.1109/ECCE44975.2020.9235812.

[49] P. Bhowmik, M. Starke, B. Dean, M. Chinthavali, "Vulnerability Detection Methodology for a Digital Signal Processor Micro-Controller at Edge Level Distributed Energy Resource Controller," 2020 IEEE CyberPELS (CyberPELS), Miami, FL, USA, 2020, pp. 1-5.

Vita

Benjamin Roy Dean graduated from Cookeville High School in 2013 and chose to attend the University of Tennessee, Knoxville to study Electrical Engineering. During his time as an undergraduate, he completed a co-op at Southern Company, an internship at ORNL, and was an undergraduate research assistant at CURENT. Upon graduation with his bachelors, he decided to continue his education at UTK, and completed his master's degree in the spring of 2021. His future plans are to pursue a Ph.D. at UTK, and to continue contributing to the field of Electrical Engineering.