INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL MECÂNICA
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

HENRIQUE WOLF

DESENVOLVIMENTO DE UM MIDDLEWARE DE INTEGRAÇÃO ENTRE MODBUS E OPC-UA

FLORIANÓPOLIS, 2023.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL MECÂNICA
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

HENRIQUE WOLF

DESENVOLVIMENTO DE UM MIDDLEWARE DE INTEGRAÇÃO ENTRE MODBUS E OPC-UA

Trabalho de Conclusão de Curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Engenheiro em Mecatrônica.

Orientador:
Prof. Dr. Roberto Alexandre Dias

FLORIANÓPOLIS, 2023.

Ficha de identificação da obra elaborada pelo autor.

Wolf, Henrique
Desenvolvimento de um Middleware de Integração entre Modbus TCP e OPC UA / Henrique Wolf; orientação de Roberto Alexandre Dias. - Florianópolis, SC, 2023.
80 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal de Santa Catarina, Câmpus Florianópolis. Bacharelado em Engenharia Mecatrônica. Departamento Acadêmico de Metal Mecânica.
Inclui Referências.

1. Modbus. 2. OPCUA. 3. Segurança. 4. Gateway. 5. Middleware. I. Alexandre Dias, Roberto. II. Instituto Federal de Santa Catarina. III. Desenvolvimento de um Middleware de Integração entre Modbus TCP e OPC UA.

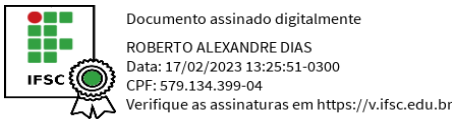
DESENVOLVIMENTO DE UM MIDDLEWARE DE INTEGRAÇÃO ENTRE MODBUS E OPC-UA

HENRIQUE WOLF

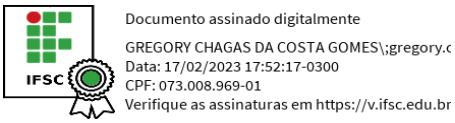
Este trabalho foi julgado adequado para obtenção do título de Engenheiro em Mecatrônica e aprovado na sua forma final pela banca examinadora do Curso Engenharia Mecatrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 9 de Fevereiro, 2023.

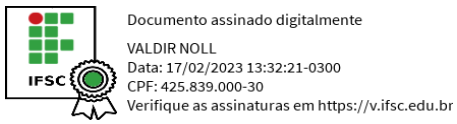
Banca Examinadora:



Dr. Roberto Alexandre Dias.



Me. Gregory Chagas da Costa Gomes.



Dr. Valdir Noll.

Aos meus pais Marcelo e Aline
por todo apoio e incentivo.

RESUMO

Apesar de ter sido publicado várias décadas atrás, o protocolo *Modbus* permanece como um dos mais utilizados no cenário industrial, principalmente em sistemas de automação utilizando CLPs, sensores e sistemas SCADA. Entretanto, o protocolo possui diversas limitações, principalmente no que se refere a segurança, uma vez que não possui elementos nativos que garantam a confidencialidade e a disponibilidade da informação, por exemplo. Novos e mais estruturados protocolos têm surgido ao longo dos anos, principalmente com o advento da indústria 4.0, como é o caso do OPC UA, que apresenta maior robustez e diversas soluções para segurança da informação. Dessa forma, este trabalho objetiva estudar as limitações oriundas do protocolo *Modbus* e as consequências da sua vasta utilização no mercado industrial, bem como apresentar uma solução viável para integrar o OPC UA a sistemas que utilizam *Modbus*. Para isso, foi realizada uma pesquisa acerca dos impactos causados por sistemas legado utilizando *Modbus* na indústria, explorando suas restrições estruturais e questões relacionadas a segurança e integridade da informação. Além disso, foi apresentada a implementação de *middlewares* como solução para essas limitações, de forma que seja possível aproveitar os benefícios de protocolos como o OPC UA sem que seja necessário reorganizar toda a estrutura de um sistema. Sendo assim, foi desenvolvido um protótipo de um *middleware* atuando como *gateway* entre *Modbus* e OPC UA, de forma a facilitar a integração de sistemas e apresentar possíveis soluções de segurança. O protótipo foi capaz de transferir a informação com sucesso em ambos os sentidos, além de contar com a funcionalidade de conversão entre *Modbus* TCP e *Modbus* Serial.

Palavras-chave: Modbus. OPCUA. Segurança. Gateway. Middleware.

ABSTRACT

Despite being published many decades ago, Modbus protocol remains as one of the most popular protocols in the industrial environment, especially in the case of automation systems, which make use of PLCs, sensors and SCADA. However, it holds many limitations, especially when it comes to security issues, once it does not assure, by any means, data confidentiality and availability. Newer and better organized protocols have been published in the recent years, especially with the advent of Industry 4.0, which is the case of OPC UA, once it is more robust and presents several solutions regarding data security issues. Therefore, this work intends to study some of the limitations of the Modbus protocol and the consequences of its vast utilization in the industrial environment, also presenting a practicable solution capable of integrating OPC UA to Modbus predominant systems. Hence, a research about the impact of legacy systems utilizing Modbus has been developed, exploring its structural limitations and data security and integrity issues. Furthermore, the implementation of middlewares as a solution to these limitations is presented, making it possible to take advantage of the benefits of protocols like OPC UA while still maintaining the original structure of the system as a whole. Thus, a gateway prototype between Modbus and OPC UA has been developed, in order to facilitate multi-system integration and present potential security solutions. The gateway prototype was successfully capable of redirect data between protocols in both ways, in addition to a feature that also makes it capable of redirecting data between Modbus TCP and Modbus RTU.

Keywords: Modbus. OPCUA. Security. Gateway. Middleware.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CLP	Controlador Lógico Programável
CRC	*Cyclic Redundancy Check*
CSMS	*Cyber Security Management System*
DB	*Database*
HTTP	*Hypertext Transfer Protocol*
IBM	*International Business Machines*
IFIP	*International Federation for Information Processing*
IFSC	Instituto Federal de Santa Catarina
IO	*Input/Output*
IoT	*Internet of Things*
REST	*Representational State Transfer*
SCADA	*Supervisory Control And Data Acquisition*

SUMÁRIO

1 INTRODUÇÃO

Mesmo sendo publicado em 1979, o *Modbus* segue até hoje como um dos protocolos de comunicação mais populares no que diz respeito a sistemas de automação (LEONARDO; JOHNSON, 2014), se tornando um padrão *de facto* implementado pela indústria de manufatura, ainda que utilizado em vários outros tipos de indústria (DUTERTRE, 2007).

Em 2004, a Schneider Electric, detentora da Modicon, tornou a utilização do protocolo pública e livre de taxas de licenciamento, fator que potencializou o uso do *Modbus* na indústria, se tornando uma das soluções mais baratas e eficientes para redes industriais no ramo de automação (CORDEIRO, 2019). Aliada a este fato, a simplicidade e fácil implementação do *Modbus* tornou-o um protocolo extremamente popular no meio industrial, principalmente considerando que pode ser implementado em diferentes meios físicos, como padrão IA/TIA -232-E, EIA-422, EIA/TIA-485-A, *Ethernet*, fibra ótica e rádio, com possibilidade de integração por meio de *gateways* (MODICON, 2012).

Entretanto, de acordo com Ågren (2020), o *Modbus* carrega muitas limitações, e pelo fato de sistemas industriais terem um longo tempo de vida, muitas mudanças fundamentais de tecnologia não são implementadas rapidamente, demonstrando claramente que protocolos de comunicação legado, como o *Modbus*, precisam se adequar aos atuais e próximos paradigmas da tecnologia.

Além disso, conforme os avanços tecnológicos se desdobram ao longo dos anos, novos e mais modernos protocolos são desenvolvidos pensando em sistemas mais complexos e distribuídos, como o OPC UA que foi idealizado com foco em grande robustez, recursos e funcionalidades de segurança ou o MQTT que, pelo contrário, foi desenvolvido com base na praticidade, agilidade e integração de dispositivos IoT.

1.1 Justificativa

Por mais que a coleta e transmissão de dados sempre tenha feito parte da indústria de automação, nos últimos anos esse tem se tornado um aspecto crucial para o sucesso de um negócio, seja para cumprir regulamentações específicas, gerenciamento de recursos, desenvolvimento de processos de manufatura, ou quaisquer outros motivos (TUNKKARI, 2018). Porém, para coletar e transmitir dados de dispositivos distintos, é necessessário que os mesmos estejam conectados de alguma forma, e o protocolo *Modbus*, especialmente em sua versão TCP, serve bem para integrações mais simples, como CLPs conectados a medidores e sensores, por exemplo. Todavia, segundo Dutertre (2007), o isolamento que existia entre redes de controle industrial e outras redes está desaparecendo, pois a interconectividade de redes traz benefícios que não

eram possíveis antigamente. Tunkkari (2018) afirma que protocolos *Fieldbus* foram desenvolvidos pensando em redes isoladas, e ao tentar integrar esses sistemas a outras redes com diferentes dispositivos, podem ocorrer diversos problemas de tráfego de dados e principalmente de segurança e integridade, fator reforçado também por Parian, Guldimann e Bhatia (2020) e Silva (2019), além de problemas de interpretação de dados, uma vez que a especificação oficial do *Modbus* define apenas registradores de 1 *bit* ou 16 *bits*, deixando a interpretação dos dados transmitidos despadronizada (ÅGREN, 2020). Ademais, Silva (2019) descreve diversas limitações referentes ao *Modbus*, como aplicações envolvendo restrições de banda, serviços em nuvem, dispositivos utilizando DHCP, necessidade de envio de dados sem requisição, entre outras.

1.2 Objetivo Geral

Desenvolver um protótipo de API atuando como *gateway* entre os protocolos *Modbus* TCP e OPC UA, de forma a minimizar os impactos provenientes das limitações do *Modbus*.

1.3 Objetivos Específicos

Os seguintes objetivos específicos foram definidos:

a) Explorar questões de segurança da informação oriundas da utilização do protocolo *Modbus*;

b) Contribuir para as pesquisas a respeito das limitações do *Modbus* no cenário industrial atual.

c) Desenvolver um protótipo de API *gateway* utilizando *Python* capaz de converter requisições entre *Modbus* TCP e OPC UA;

2 FUNDAMENTAÇÃO TEÓRICA

No seguinte capítulo, são introduzidos tópicos a fim de desenvolver uma base teórica essencial para o entendimento do protótipo desenvolvido no capítulo 4. São abordados os conceitos de API, *middleware* e *gateway*, bem como a estrutura dos protocolos *Modbus* e OPC UA de uma maneira geral, uma vez que estes são os pontos chave para o desenvolvimento de um *gateway* entre *Modbus* e OPC UA.

2.1 *Modbus*

Desenvolvido e publicado em 1979 pela *Modicon Industrial Automation Systems*, o *Modbus* é um dos protocolos de comunicação mais antigos do mercado e, apesar disso, permanece até hoje como um dos mais utilizados no meio industrial, principalmente por se tratar de um protocolo aberto e de implementação e manutenção relativamente simples.

Segundo a própria especificação do protocolo (MODICON, 2012), o modelo utilizado define uma comunicação cliente/servidor a qual pode ser estabelecida utilizando diferentes tipos de barramentos e redes, como TCP/IP via *Ethernet*, transmissão serial (EIA/TIA-232-E, EIA-422, EIA/TIA-485-A, fibra, rádio, etc.), entre outros.

2.1.1 Armazenamento de informação

Segundo a Modicon (2012), o modelo de dados do protocolo é definido por uma série de tipos com características distintas, onde os 4 tipos primários estabelecidos por padrão são apresentados na tabela 1.

Tabela 1 – Modelo de dados padrão do *Modbus*

Nome	Tipo de objeto	Tipo de acesso	Descrição
Discrete Input	bit único	*Read*	Tipo de dado utilizado para representar IOs de um dispositivo.
Coil	bit único	*Read/Write*	Dado de armazenamento interno e valor modificável.
Input Register	word de 16-bits	*Read*	Tipo de dado utilizado para representar IOs de um dispositivo.
Holding Register	word de 16-bits	*Read/Write*	Dado de armazenamento interno e valor modificável.

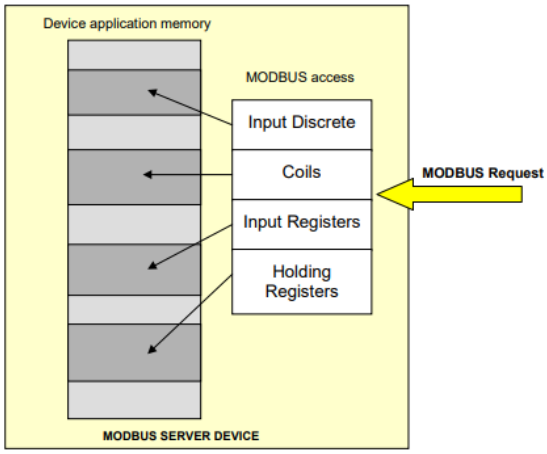
Fonte: Modicon (2012).

O *Modbus* também define que cada um dos quatro tipos de registrador pode ocupar até 65536 endereços. Entretanto, fica a critério de cada dispositivo decidir

quantos endereços serão disponibilizados para cada tipo, o que geralmente varia dependendo da memória total do equipamento e de seu propósito.

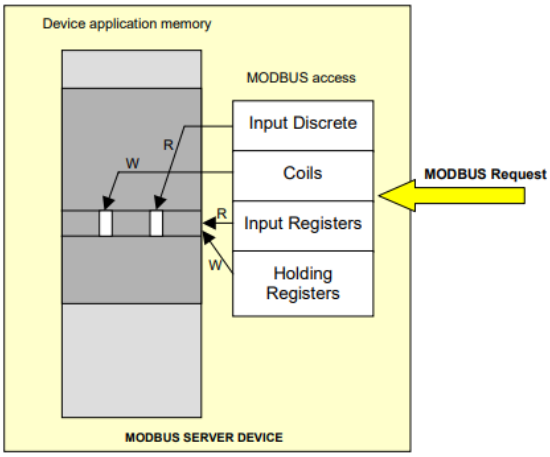
A forma com que os registradores são organizados dentro da memória do equipamento também varia, ficando a critério do fabricante. Por exemplo, um determinado dispositivo pode dedicar um bloco de memória específico para cada tipo de registrador (figura 1), sendo assim, o endereço X de um *input register* estará alocado em um trecho totalmente diferente do endereço X de um *holding register*. Por outro lado, um segundo dispositivo pode utilizar apenas um bloco da memória para todos os tipos de registradores (figura 2), fazendo com que o endereço X de um *input register* seja exclusivo e causando um conflito em transações que tentem acessar esse endereço requisitando outro tipo de registrador. Além desses exemplos, quaisquer outros arranjos de memória são permitidos, desde que respeitem o limite imposto pelo protocolo.

Figura 1 – Alocação de memória em blocos individuais



Fonte: Modicon (2012).

Figura 2 – Alocação de memória em um único bloco



Fonte: Modicon (2012).

2.1.2 Estrutura de mensagem

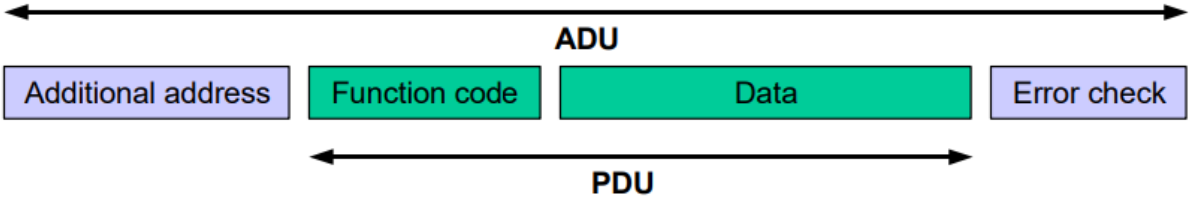
Independente do tipo de barramento ou rede utilizada, o protocolo determina um PDU (*Protocol Data Unit*) padrão para cada transação realizada entre cliente e servidor, entretanto, o PDU se encaixa dentro de uma estrutura maior chamada ADU (*Application Data Unit*), a qual contém campos determinados a depender da rede. A estrutura final da transação envolve 4 campos:

- *Additional address*: Endereços adicionais determinados pela aplicação;
- *Function code*: Código de função *Modbus* que será utilizado na interação com o(s) registrador(es) especificado(s), identificando qual o tipo de registrador (tabela 1), quantidade de registradores acessados em sequência e tipo de operação (leitura ou escrita);

- *Data*: Campo que contém informações a respeito da transação em si, como endereço do registrador acessado e número de registradores lidos em sequência;
- *Error check*: Método de checagem de integridade da transação definido pela aplicação.

Uma ilustração a respeito da ADU pode ser observada na figura 3.

Figura 3 – *Modbus* ADU

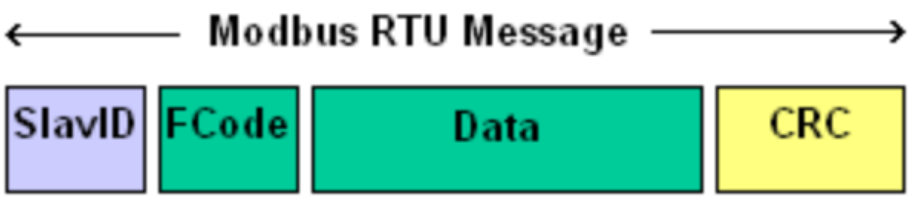


Fonte: Modicon (2012).

O tamanho de um PDU pode variar de acordo com a quantidade de dados requeridos na conexão, sendo 1 byte utilizado para definição do *Function code* e 0 a 252 bytes utilizados para o campo *Data*.

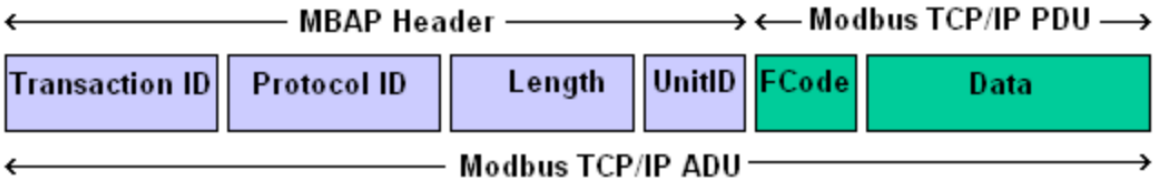
Já os campos adicionais presentes no ADU dependem do tipo de conexão. Para o caso de conexões serial (Figura 4), o campo *Additional address* consiste no ID do escravo que será acessado, definido em 1 byte, enquanto o campo *Error check* utiliza checagem de erro via CRC. Já no caso de conexões TCP (Figura 5), não há campo de checagem de erro, e o campo *Additional address* ocupa 7 bytes e consiste no MBAP *Header*, contendo uma identificação para a transação (2 bytes), uma identificação de protocolo (2 bytes), uma definição de tamanho total da mensagem (2 bytes) e um identificação de servidor que funciona semelhante ao ID de escravo utilizado na conexão serial (1 byte).

Figura 4 – ADU para *Modbus* em conexões serial



Fonte: Modicon (2012).

Figura 5 – ADU para *Modbus* em conexões TCP



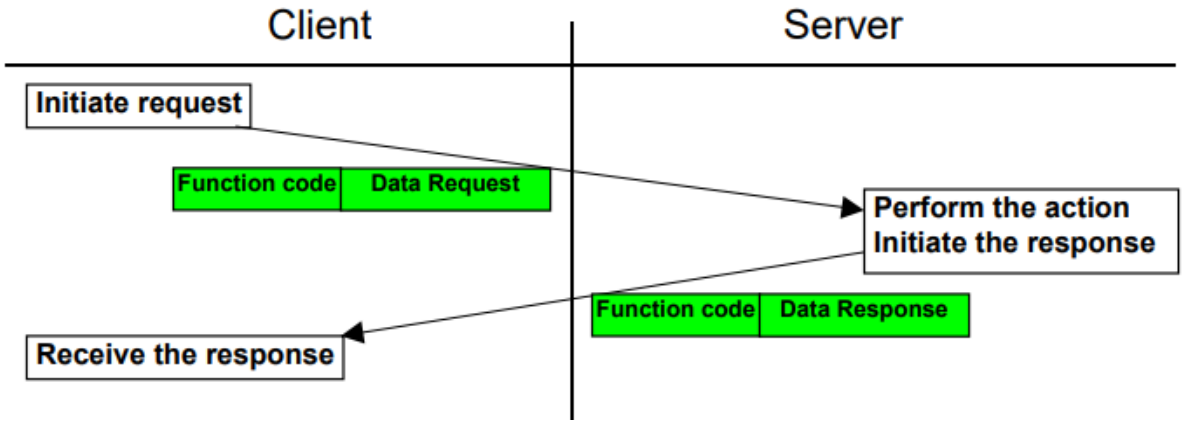
Fonte: Modicon (2012).

Sendo assim, o tamanho total de um ADU em conexão serial pode chegar a um máximo de 256 bytes enquanto em uma conexão TCP o tamanho máximo possui um limite de 260 bytes.

2.1.3 Conexão

A conexão entre dois dispositivos através do protocolo *Modbus* TCP se dá pelo modelo Cliente-Servidor, onde é sempre o cliente que envia uma requisição ao servidor e aguarda por uma resposta. Para o do *Modbus* RTU, o modelo de conexão utilizado é o Mestre-Escravo, que funciona de forma extremamente semelhante. Caso a requisição seja uma leitura, o cliente/mestre receberá os dados requeridos, caso seja uma escrita, receberá uma confirmação de que o dado foi escrito com sucesso. Uma representação gráfica de uma transação *Modbus* bem sucedida pode ser observada na figura 6.

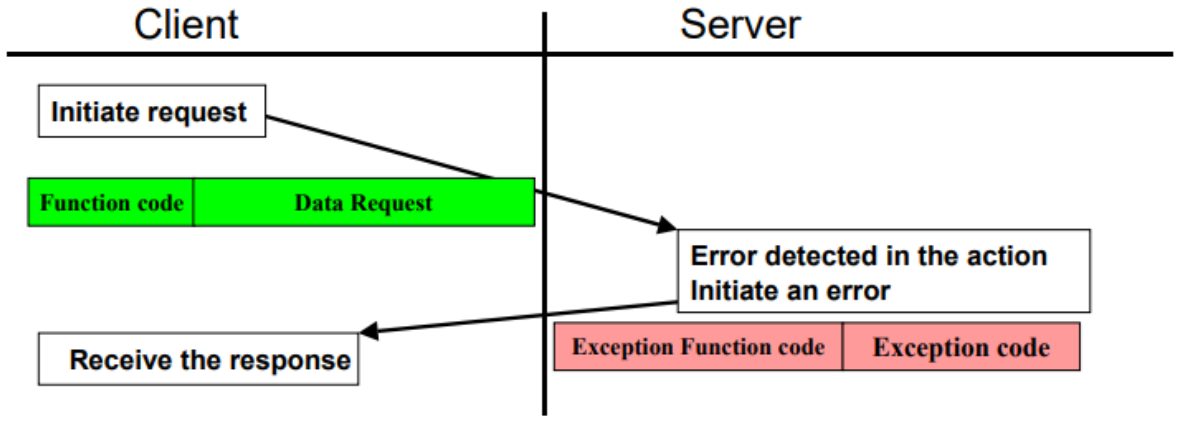
Figura 6 – Transação *Modbus* realizada com sucesso



Fonte: Modicon (2012).

No caso de uma conexão mal sucedida, o servidor enviará, através da mensagem de resposta, um código de erro indicando o problema ocorrido. Entre os erros mais comuns, pode se citar o caso de *function code* ilegal (código 01), endereço ilegal (código 02), dados ilegais (código 03), falha no dispositivo servidor (código 4) e dispositivo servidor ocupado (código 6). Na figura 7, observa-se uma representação gráfica a respeito de transações mal sucedidas.

Figura 7 – Transação *Modbus* mal sucedida



Fonte: Modicon (2012).

Portanto, apesar de simples, o funcionamento correto de uma transação *Modbus* exige que várias condições estejam estabelecidas.

2.1.4 *Function Code*

O *function code* é uma informação de suma importância para uma conexão via *Modbus* pois é ele quem define que tipo de registrador estará sendo acessado bem como a quantidade de registradores em sequência requisitados. Se o *function code* de uma mensagem requisitar um endereço indisponível no servidor, a mensagem irá quebrar e a requisição será mal sucedida.

Como citado, o *function code* ocupa o espaço de 1 byte, portanto, há um *range* de 1-255 a ser selecionado para cada *function code*, uma vez que 0 não é uma seleção válida. Além disso, os espaços de 128-255 são reservados para respostas de exceção.

Existem três tipos de *function code* no protocolo *Modbus*: públicos, definidos por usuário e reservados. Os públicos estão aptos a serem utilizados por padrão em qualquer conexão que utilize o protocolo, os definidos por usuário podem ser programados individualmente de acordo com conexões específicas, já os reservados são códigos utilizados por algumas empresas em equipamentos legado e não estão disponíveis para uso público.

A lista de *function codes* públicos é bastante extensa, entretanto, na maioria das vezes uma conexão *Modbus* utiliza um dos seguintes:

- 01 - *Read Coils*: Função utilizada para ler tanto registradores do tipo *coil* quanto do tipo *discrete input*. Por se tratar de apenas um bit, podem ser lidos até 2000 registradores em sequência dentro de uma única mensagem.

- 02 - *Read Discrete Inputs*: Semelhante ao código 01, podem ser lidos até 2000 registradores em sequência, porém funciona apenas com *discrete inputs* (entradas digitais físicas) e não com *coils*.
- 03 - *Read Holding Registers*: Função utilizada para realizar leitura de *holding registers*. Por se tratarem de registradores 16-bit, podem ser lidos até 125 endereços em sequência.
- 04 - *Read Input Register*: De forma similar ao código 03, porém rege os registradores de entradas analógicas físicas.
- 05 - *Write Single Coil*: Função utilizada para escrita de uma única *coil*, uma vez que *discrete inputs* são somente para leitura.
- 06 - *Write Single Register*: Função utilizada para escrita de um único *holding register*, uma vez que *input registers* são apenas para leitura.
- 15 - *Write Multiple Coils*: Código utilizado para escrita de múltiplas *coils*, desde que estejam em sequência, com um limite de 1968 endereços.
- 16 - *Write Multiple Registers*: Função utilizada para escrita de múltiplos *holding registers*, desde que estejam em sequência, com um limite de 123 endereços.

2.2 Limitações do protocolo *Modbus*

2.2.1 Segurança

Um dos problemas mais discutidos atualmente quando se trata do protocolo *Modbus* é a sua extrema falta de segurança, uma vez que não possui nenhum método de autenticação ou detecção de invasões. Obviamente, este se trata de um problema recente, pois, de acordo com Fovino *et al.* (2009), protocolos como o *Modbus* foram desenvolvidos em uma época em que todas as redes conexão entre dispositivos eram implementadas localmente, o que fez com que problemas como autenticação, integridade e confidencialidade não fossem possibilidades consideradas.

Entretanto, ainda segundo Fovino *et al.* (2009), o advento da *internet* e o uso do *Modbus* via TCP/IP tornou possível a invasão de conexões mal intencionadas, o que coloca em risco a integridade das requisições realizadas na rede e dos sensores, atuadores e demais dispositivos conectados a ela. Martins e Oliveira (2022) dizem que sistemas utilizando *Modbus* podem ser explorados através da interceptação e alteração de pacotes no meio de sua transação (ataque conhecido como *man-in-the-middle*), enviando comandos maliciosos, mal-formados ou repetidos ao sistema.

Os sistemas SCADA estão entre os mais prejudicados por este problema, tendo em vista que, mesmo que recentemente os sistemas estejam sendo desenvolvi-

dos com problemas de segurança em mente, o longo ciclo de vida (de 10 a 20 anos) e alto custo desse tipo de sistema fez com que novas tecnologias fossem implementadas sobre processos já existentes, deixando vários componentes de SCADA vulneráveis a ataques (PARIAN; GULDIMANN; BHATIA, 2020).

Um estudo realizado a respeito da segurança do protocolo *Modbus* TCP (LUSWATA; ZAVARSKY; SWAR, 2018) realizou diversos testes de ataques a um sistema SCADA sob variadas condições de segurança, classificando-os por grau de severidade em uma escala de 0.0 a 10.0. O estudo denota que, uma vez que um dispositivo malicioso já está conectado a rede, ou seja, realizando um ataque interno, é extremamente simples realizar ataques DoS contra os servidores do sistema. De acordo com os resultados, ao utilizar um cliente malicioso para enviar requisições de escrita (especificamente *WriteSingleCoil* e *WriteSingleRegister*) repetidamente e o mais rápido possível a um servidor da rede, os demais clientes ficaram impossibilitados de realizar qualquer operação de escrita nesse mesmo servidor, porém ainda conseguiram realizar operações de leitura, o que prejudicou drasticamente a integridade e a disponibilidade da rede, classificando a severidade do ataque em 10.0 (Crítico). A tabela 2 demonstra os critérios utilizados pelo estudo para avaliar a severidade do ataque.

Tabela 2 – Critérios para classificação de severidade de ataque

Critério	Valor
Vetor de Ataque	Rede
Complexidade	Baixa
Interação de Usuário	Nenhuma
Escopo da Aplicação	Alterado
Impacto na Confidencialidade	Nenhum
Impacto na Integridade	Alto
Impacto na Disponibilidade	Alto

Fonte: Luswata, Zavarsky e Swar (2018).

Segundo Amiri (2021), qualquer dado incorreto transmitido em um sistema tem o potencial de causar um incidente de segurança, e o *Modbus*, apesar de possuir um sistema de checagem de erro que funciona bem em requisições rotineiras, não tem segurança o suficiente para se defender de ataques cibernéticos ou falhas internas de segurança, tornando seu uso via TCP, principalmente quando conectado a uma rede externa (como a própria internet), bastante suscetível a brechas de segurança.

2.2.2 Estrutura

Segundo Urrea, Morales e Kern (2016), o *Modbus* possui diversas limitações no que se refere a sistemas de controle automação atuais, incluindo baixas taxas de transmissão, número limitado de dispositivos conectados à rede e, principalmente, a

impossibilidade de corrigir mensagens corrompidas por erros. Ågren (2020) também discute sobre as limitações de tamanho máximo de dados por cada mensagem *Modbus* em 252 bytes, ainda mais considerando que cada registrador ocupa um máximo de 16 bits.

Além disso, pelo fato de ter sido desenvolvido ao longo da década de 1970, o protocolo *Modbus* se restringiu somente a dois tipos de dados: binários e inteiros de 16 *bits*. Daquela época até hoje, diversos outros tipos de dados foram desenvolvidos, como pontos flutuantes, inteiros de 32 ou até 64 *bits*, *strings*, entre muitos outros, o que fez com que cada fabricante desenvolvesse uma solução "caseira"para todos esses tipos de dados (RINALDI, 2021), prejudicando de forma considerável a padronização do protocolo. Dentre as soluções propostas, muitas vezes são utilizados múltiplos registradores para representar um único dado, diminuindo ainda mais a quantidade de dados que podem ser transmitidos em uma só requisição. Ademais, por utilizar somente registradores definidos em espaços de endereçamento, deve-se considerar que o *Modbus* não define nenhum tipo de objeto e, portanto, nenhuma informação a respeito de cada dado além de seu valor numérico, obrigando os fabricantes de equipamentos a documentar todos os seus registradores, especificando seu endereço, tipo, descrição, escala e unidade de medida.

Também há de se considerar que, por utilizar a metodologia cliente-servidor, o *Modbus* impossibilita a conexão entre dois clientes, bem como não permite que servidores realizem requisições, uma vez que, por definição, podem apenas recebê-las e respondê-las. Para resolver essa limitação, o dispositivo necessita possuir uma porta adicional (seja serial ou *Ethernet*), exigindo mais processamento, mais memória, mais configurações por parte do usuário e mais documentação por parte do fabricante. Além disso, o modelo cliente-servidor implica em conexões ponto a ponto, as quais, no caso do *Modbus* TCP, requerem que cada dispositivo conectado possua um IP fixo, impossibilitando o uso de DHCP, onde o IP varia (SILVA, 2019).

Portanto, se tratando de sistemas tradicionais, os problemas gerados por essas limitações podem ser resolvidos individualmente com o auxílio de soluções de eficácia aceitável, entretanto, considerando a tendência cada vez maior do mercado para a Indústria 4.0 e a Internet das Coisas, com dados cada vez mais complexos, novos protocolos mais estruturados, velocidade, distância e segurança de transmissão de dados cada vez mais exigentes, sensores e equipamentos *wireless* e quantidade cada vez maior de informação, as limitações do protocolo *Modbus* se tornam um problema progressivamente maior.

2.2.3 Integração

A integração ou interoperabilidade entre dispositivos é um problema recorrente quando se trata de sistemas de automação industrial, devido ao fato de muitos equipamentos utilizarem protocolos diferentes. Ågren (2020) afirma que sistemas de controle estão tendendo cada vez mais a uma estrutura descentralizada, com dispositivos capazes de transmitir informação de formas nunca antes imaginadas, fator que deve crescer ainda mais com a Internet das Coisas e a Indústria 4.0. Ågren (2020) também comenta que alguns sensores e outros equipamentos que antes eram medidos puramente por resistência, tensão ou corrente, hoje em dia podem possuir seu próprio poder computacional, capaz de transmitir valores e outras variáveis por conta própria através de protocolos de comunicação.

Dessa forma, a simplicidade estrutural do protocolo *Modbus*, que é sua maior vantagem, pode ser um grande empecílio para a integração de sua rede a protocolos de maior complexidade ou que utilizem modelos demasiadamente diferentes.

2.3 OPC UA

De acordo com a OPC Foundation (2022), OPC UA é uma arquitetura multiplataforma orientada à serviço com base no padrão OPC Classic, que se trata de um padrão estruturado em *Windows* que define 3 especificações diferentes: OPC DA (utilizado para troca de informações como valores de variáveis), OPC AE (utilizado para informações referentes a alarmes e eventos) e OPC HDA (utilizado para troca de dados históricos marcados com datas, como relatórios e base de dados). A chegada do OPC UA unificou todas essas especificações em um só padrão, além de excluir a dependência do sistema *Windows*, se tornando uma opção bastante atrativa para o mercado.

2.3.1 Segurança

Um dos pontos de destaque do OPC UA é o seu modelo de segurança, que permite ao usuário definir arbitrariamente a utilização de proteções para diversos tipos de ameaças. A metodologia definida pelo protocolo para combater potenciais problemas de segurança é identificar pontos de vulnerabilidade de sistemas, mapear possíveis ataques com foco nesses pontos e, em seguida, especificar contramedidas para prevenção.

As vulnerabilidades definidas pela OPC Foundation (2022) são as seguintes:

- Autenticação: A falta de autenticação torna impossível identificar a origem de uma entidade (cliente, servidor, etc) que está realizando um ataque;

- Autorização: A falta de autorização abre brechas para clientes maliciosos obterem acesso a servidores importantes do sistema;
- Confidencialidade: Mensagens importantes sem nenhum tipo de criptografia colocam em risco a sua confidencialidade, tornando esse um problema de segurança da informação;
- Integridade: Problemas de integridade em mensagens podem fazer com que a informação enviada pelo cliente seja diferente da recebida pelo servidor, podendo causar sérios danos ao sistema;
- Não-repúdio: Transações que não estejam protegidas contra repúdio não possuem garantia alguma de que realmente foram enviadas;
- Auditabilidade: Mensagens enviadas em sistemas sem auditabilidade dificultam a produção de evidências, uma vez que, mesmo que haja autenticação, não há gravação de quem enviou a requisição;
- Disponibilidade: A disponibilidade de um sistema pode ser afetada quando alguma ação maliciosa impede que clientes se comuniquem com servidores, causando a perda de controle do sistema.

A OPC Foundation (2022) também define em sua documentação os tipos de ataques que podem ser inferidos com base nessas vulnerabilidades:

- *Denial of Service*: Ataque que objetiva atrasar a operação do sistema através de métodos como *flood* de mensagens, exaustão de recursos e *crash* de aplicações, impactando na disponibilidade do sistema;
- *Eavesdropping*: Consiste em expor informações confidenciais que podem impactar diretamente em brechas de segurança para outros ataques, afetando a confidencialidade do sistema;
- Falsificação de mensagens: Método utilizado para forjar mensagens como se fossem enviadas a partir de uma entidade confiável do sistema, realizando ações não autorizadas e, portanto, atacando a sua integridade e autorização;
- Alteração de mensagens: Consiste em capturar uma mensagem enviada por uma entidade confiável e modificá-la antes que chegue a seu destinatário, permitindo acesso ilegítimo ao sistema e afetando sua integridade, autorização, auditabilidade, não-repúdio e autenticação;
- *Replay* de mensagens: Técnica que objetiva capturar uma mensagem válida e autenticada na rede, como a abertura de uma válvula, e reproduzi-la em um

momento indevido, podendo causar danos a equipamentos e componentes, impactando a autorização e autenticação do sistema;

- Mensagens malformadas: Método em que um cliente malicioso pode enviar mensagens com uma estrutura corrompida, fazendo com que o dispositivo destinatário da mensagem lide com ela de forma incorreta, resultando em diversas possibilidades de falha, como DoS, desligamento ou até *crash* da aplicação, afetando a integridade e a disponibilidade do sistema;
- *Server profiling*: Se baseia na ideia de estudar uma entidade da rede, como um servidor, de modo a entender como a mesma funciona, quais são suas vulnerabilidades, sua entidade e outros metadados a partir de sua mensagem de resposta. Apesar de não ser um ataque em si, essa estratégia pode abrir espaço para qualquer outro tipo de ataque, impactando todos os pontos de segurança indiretamente;
- *Session hijacking*: Ataque em que são utilizadas informações (obtidas por outros métodos ou até por dedução) a respeito de uma conexão estabelecida na rede de forma a introduzir mensagens manipuladas e sequestrar a conexão para si, permitindo realizar operações não autorizadas e, portanto, afetando todos os pontos de segurança;
- *Rogue server*: Se trata de um componente servidor malicioso na rede se passando por uma entidade legítima, arriscando a exposição de informações privilegiadas que podem ser enviadas por clientes. Esse ataque impacta todos os pontos de vulnerabilidade com exceção da integridade e do não-repúdio;
- Credenciais comprometidas: Consiste na obtenção de logins, senhas, certificados e *keys* a partir da obtenção de documentos privilegiados, espionagem ou tentativa e erro através de *softwares*, impactando a autorização, autenticação e confidencialidade;
- Repúdio: Apesar de não se tratar diretamente de um ataque, questões de repúdio afetam bastante a confiança entre clientes e servidores, impactando, logicamente, na vulnerabilidade de não-repúdio.

Por fim, a OPC Foundation (2022) estabelece as contra medidas para cada um desses ataques:

- *Denial of Service*: O OPC UA reduz os riscos de ataques DoS ao diminuir a quantidade de processamento utilizada em uma mensagem antes de sua autenticação, além de definir um sistema de autenticação de usuário para prevenir clientes não autorizados de realizar exaustão de recursos;

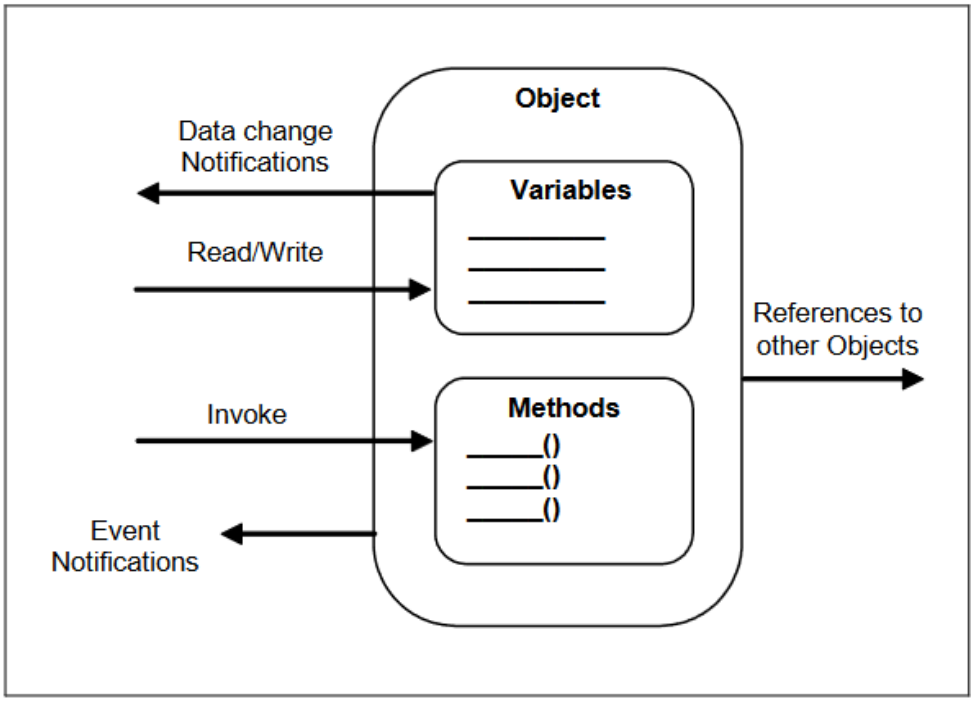
- *Eavesdropping*: O protocolo inibe ataques *eavesdropping* ao permitir criptografia no envio de requisições;
- Falsificação de mensagens: OPC UA abre a possibilidade de assinatura de mensagens, além de definir como necessário um *SessionId*, um *SecureChannelId*, um *RequestId* e um *Timestamp* para cada requisição, reduzindo as possibilidades de falsificação;
- Alteração de mensagens: A assinatura de mensagens disponibilizada pelo OPC UA previne a alteração de mensagens, uma vez que a assinatura sempre pode ser checada e, no caso de alterações, a mensagem pode ser descartada;
- *Replay* de mensagens: Devido ao fato de toda mensagem requerer *SessionId*, *SecureChannelId*, *RequestId* e *Timestamp*, além de assinatura de mensagens, fica muito difícil reproduzir uma mensagem legítima, tornando um trabalho árduo pouco compensador, além de arriscado;
- Mensagens malformadas: O protocolo descarta esse problema ao implementar uma funcionalidade de checagem da forma e dos parâmetros de cada mensagem, descartando as malformadas;
- *Server profiling*: O OPC UA limita a quantidade de informação não identificada que pode ser acessada por clientes, dificultando muito um ataque de *server profiling*;
- *Session hijacking*: Toda conexão estabelecida via OPC UA é feita através de um *secure channel*, requerendo ao usuário malicioso passar pela segurança do canal antes de sequer tentar sequestrar a sessão;
- *Rogue server*: O sistema de segurança utilizando certificados e *private keys* implementado pelo OPC UA torna *rogue servers* inválidos na rede;
- Credenciais comprometidas: Quanto a credenciais transmitidas pela rede, a criptografia disponibilizada pelo protocolo inibe vazamentos, já para o caso de tentativa e erro, é necessário que a planta possua um CSMS;
- Repúdio: Novamente, a assinatura de mensagens previne esse problema.

Portanto, o OPC UA possui um sistema de medidas de segurança bastante robusto, possuindo autenticação de usuário, certificado e *private key*, criptografia e assinatura de mensagens, *secure channel* para sessões e prevenções contra ataques DoS.

2.3.2 Armazenamento de Informação

De acordo com a OPC Foundation (2022), o OPC UA possui um padrão de armazenamento de informações de forma a disponibilizar um espaço denominado *AddressSpace* com objetos em servidores que podem ser acessados por clientes. Objetos OPC UA podem conter diversas variáveis e métodos de vários tipos e hierarquias. O modelo de objeto OPC UA pode ser observado na figura 8.

Figura 8 – Modelo de objeto OPC UA



Fonte: OPC Foundation (2022).

Cada item presente em um objeto OPC UA é chamado de *Node* e cada *Node* pertence a uma *NodeClass*, que define qual o tipo de parâmetro definido. Uma *NodeClass* é constituída por atributos e referências, onde os atributos são os componentes que descrevem suas características enquanto as referências descrevem seu relacionamento com outros *Nodes*. Toda vez que um *Node* é instanciado, ele recebe todos os atributos e referências de sua *NodeClass*, bem como um exclusivo *NodeId*, que serve como um índice de identificação.

As variáveis de cada objeto OPC UA podem ser de dois tipos: *Properties* e *DataVariables*. As variáveis de tipo *Properties* funcionam de maneira semelhante aos atributos dos *Nodes*, porém, enquanto os atributos são metadados que definem todos os componentes de uma *NodeClass*, as *Properties* são definidas por cada servidor e inerentes somente a seu objeto, como uma especificação de nome, unidade de medida, etc. Já as *DataVariables* representam o conteúdo real do objeto, como um *setpoint*, por exemplo. Portanto, um exemplo de objeto representando um sensor de

temperatura pode conter uma *DataVariable* de temperatura em graus Celsius e uma variável *Property* com o nome do sensor.

Cada variável também possui uma definição de *DataType*, responsável por determinar a estrutura do atributo *Value* de cada variável. O OPC UA possui uma grande quantidade de *DataTypes* pré-definidos, dentre os quais podem ser citados:

- *NodeId*: Índice de identificação exclusivo de cada *Node*;
- *Boolean*: variável de dois estados que geralmente é utilizada para condições *true* ou *false*;
- *Byte*: Variável numérica composta por 16 bits, variando em números inteiros de 0 a 255.
- *DateTime*: Variável que representa dia, mês, ano, horas, minutos e segundos;
- *Integer*: Variável que representa um número inteiro com *range* definido pela quantidade de bits utilizados (16, 32 ou 64);
- *Float*: Variável que define valores numéricos não inteiros de acordo com a IEEE 60559:2011;
- *String*: Variável utilizada para representar valores alfanuméricos.
- *Argument*: Variável *array* utilizada em métodos OPC UA, consistindo dos valores Name, DataType, ValueRank, ArrayDimensions e Description.

Outros *DataTypes* podem ser encontrados na documentação oficial do protocolo, disponibilizada pela OPC Foundation (2022).

Métodos podem ser definidos dentro de objetos OPC UA e funcionam de forma semelhante a funções utilizadas em linguagens de programação como C ou Python, por exemplo. Enquanto não possuem um valor específico, podem receber parâmetros, chamados *InputArguments* (variáveis do tipo *Argument*), os quais são utilizados para reproduzir uma saída chamada *OutputArguments* (também do tipo *Argument*).

2.3.3 Conexão

A forma com que clientes acessam e interagem com servidores em OPC UA é definida pela OPC Foundation (2022) através de um conceito chamado serviços, que são métodos estabelecidos para que os clientes possam acessar informações específicas em cada servidor utilizando requisição e resposta. Os serviços são definidos de acordo com sua função em grupos denominados *Service Sets*, ou conjuntos de

serviços, os quais, quando invocados, recebem os parâmetros necessários enviados pelo cliente, processam os dados, repassam ao servidor, recebem a resposta e a redirecionam ao cliente.

Os conjuntos de serviços estabelecidos pela OPC Foundation (2022) são os seguintes:

- *Discovery Service Set*: Serviço utilizado para que o cliente descubra os *endpoints* estabelecidos por um servidor, bem como a configuração de segurança desses *endpoints*. *Endpoints* funcionam como endereços da rede, e, por mais que cada servidor possa definir múltiplos *endpoints*, todos devem ter pelo menos um, chamado *DiscoveryEndpoint*;
- *Secure Channel Service Set*: Serviço que permite ao cliente estabelecer uma comunicação com o servidor desde que atenda a política de segurança definida pelo servidor;
- *Session Service Set*: Esse serviço diz respeito ao estabelecimento de uma sessão entre cliente e servidor, baseada em autenticação de usuário;
- *Node Management Service Set*: Serviço que permite ao cliente adicionar, modificar e deletar *Nodes* do servidor;
- *View Service Set* e *Query Service Set*: Estes dois serviços agem em conjunto. Enquanto o serviço de *View* disponibiliza todos o *AddressSpace* do servidor ao cliente, o serviço de *Query* seleciona um conjunto específico de *Nodes* do *AddressSpace*;
- *Attribute Service Set*: Serviço que permite ao cliente ler/escrever atributos de objetos, variáveis e outros *Nodes* do servidor;
- *Method Service Set*: Serviço que permite ao cliente realizar chamados à métodos de objetos do servidor;

Há de se ressaltar que os servidores não são obrigados a possuir todos os conjuntos de serviços listados.

Portanto, em uma conexão OPC UA o cliente deve sempre se comunicar com quaisquer servidores da rede através destes serviços, a depender de quais foram implementados pelo servidor em questão, uma vez que um servidor pode não contemplar autenticação de usuário, ou um determinado objeto pode não possuir métodos, por exemplo.

2.4 *Middleware*

Segundo a Redhat (2018), *middleware* se trata de um *software* que fornece recursos comuns a aplicações, facilitando assim a integração entre essas aplicações. Funciona como uma camada capaz de integrar diversas aplicações, independentemente de seus protocolos, plataformas, arquiteturas, ambientes e sistemas operacionais (SOFTWAREONE, 2021).

A Redhat (2018) ainda exemplifica vários possíveis tipos de *middleware*, dentre eles a otimização de sistemas já existentes, integração abrangente de aplicações, APIs e transmissão de dados, se provando um conceito de importância considerável para a indústria nos tempos atuais.

2.4.1 Implementação de middlewares na indústria

Historicamente, conforme os sistemas industriais foram ficando cada vez mais exigentes em termos de segurança, disponibilidade e longevidade, centralizar o sistema todo em um só equipamento se tornou cada vez mais inviável e suscetível a falhas, o que acabou por gerar a ascensão de sistemas distribuídos, onde uma rede composta por diversos controladores, sensores, atuadores e outros equipamentos regem o sistema todo simultaneamente (BALADOR; ERICSSON; BAKHSHI, 2017).

Entretanto, ainda de acordo com Balador, Ericsson e Bakhshi (2017), sistemas distribuídos trazem uma nova gama de problemas, como conexões tanto horizontais quanto verticais para todos os dispositivos, uma vez que precisam estar interconectados, e, principalmente, a heterogeneidade da comunicação, tendo em vista que cada equipamento pode possuir configurações de comunicação diferentes, ainda mais quando se mescla dispositivos legado com dispositivos desenvolvidos com base em conceitos da Indústria 4.0.

Todavia, de acordo com Razzaque *et al.* (2015), a grande quantidade de informação transmitida simultaneamente e a heterogeneidade de protocolos de comunicação proporcionada pela IoT traz grandes desafios para o desenvolvimento de aplicações, principalmente no que diz respeito a padronização. Nesse contexto, Razzaque *et al.* (2015) propõe que *middlewares* possam ser apresentados como solução, facilitando a integração e a interoperabilidade do sistema. Tunkkari (2018) também afirma que *gateways*, que são um tipo de *middleware*, servem não apenas para integração entre protocolos, mas também para adicionar uma camada de segurança, sendo essas duas das maiores limitações do protocolo *Modbus*. Ademais, Fersi (2015) denota que, além da interoperabilidade, confiabilidade e homogenização de sistemas, *middlewares* podem contribuir para fatores como escalabilidade, mobilidade e multiplicidade, por exemplo.

2.4.2 *Gateway*

Segundo a Altus (2017), *gateways* são dispositivos capazes de estabelecer a comunicação entre equipamentos que utilizam redes diferentes e padrões distintos, portanto, podem ser considerados um tipo de *middleware*.

Apesar de derivarem de um conceito simples, *gateways* são extremamente úteis no ambiente industrial, pois integram de maneira simples equipamentos que, do contrário, não poderiam estar se comunicando através da mesma rede. Um exemplo de *gateway* extremamente comum na indústria é o de conversão entre *Modbus* TCP e *Modbus* Serial, muito utilizado para integrar equipamentos que suportam apenas comunicação serial a uma rede *Ethernet* mais abrangente. A figura 9 representa o *gateway* entre *Modbus* TCP e Serial GTON-C, da fabricante HI Tecnologia.

Figura 9 – *Gateway* GTON-C

Fonte: Tecnologia (2022).

Além disso, de acordo com a Altus (2017), *gateways* podem ser de grande importância em aplicações IoT, considerando, por exemplo, que podem realizar uma ponte entre serviços em nuvem e equipamentos industriais mais tradicionais, como CLPs, atuadores e sensores.

2.5 API

Uma API consiste em uma interface que permite que um serviço interaja com as funcionalidades e informações de outro serviço agindo como uma interface intermediária, de modo que os desenvolvedores destes serviços não necessitem saber como a API foi desenvolvida ou estruturada, usufruindo apenas de suas funcionalidades (IBM, 2020).

A IBM (2020) também define quatro etapas que classificam uma API:

1. Um cliente realiza o chamado da API (*request*) de modo a requerer informações de um servidor. A API, então, processa esse chamado de acordo com sua

funcionalidade.

2. A API envia o *request* processado para o servidor.
3. O servidor envia uma resposta de volta à API.
4. A API repassa a informação recebida para o cliente.

Segundo Chomko (2010), APIs são de extremo valor para todos os tipos de empresas com foco em serviços de software, pois permitem a integração de sistemas que muitas vezes possuem configurações e interfaces completamente distintas, permitindo um controle muito maior aos desenvolvedores.

2.5.1 REST

REST é um padrão de estilo de arquitetura baseado em HTTP inicialmente apresentado como parte da tese de doutorado de Roy Thomas Fielding. De acordo com Fielding (2000), REST deriva de diversos outros estilos de arquitetura junto de determinadas restrições de forma a desenvolver uma interface de conexão uniforme.

Fielding (2000) estabelece em sua tese algumas características do estilo de arquitetura denominadas restrições, para que um sistema web se encaixe no padrão REST. As restrições definidas pelo autor são as seguintes:

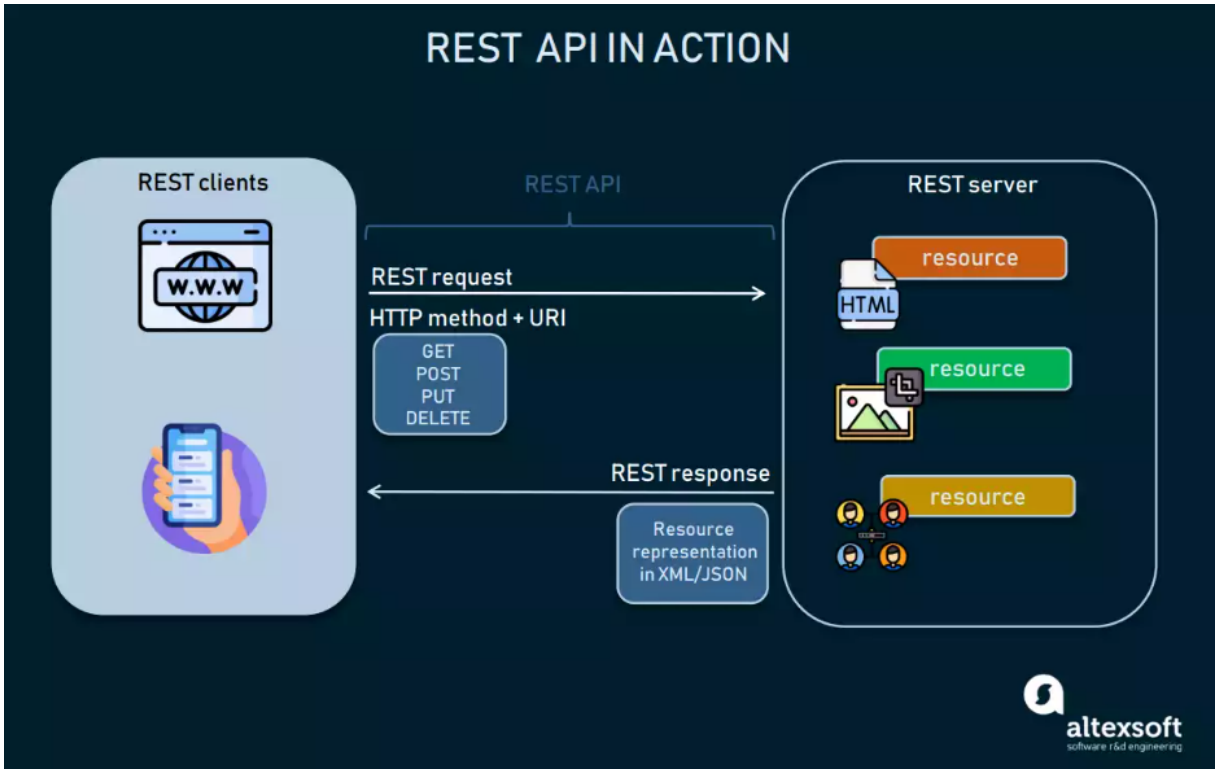
- Cliente-Servidor: A separação entre cliente e servidor tem como princípio desatrelar as responsabilidades geridas pela interface de usuário das responsabilidades submetidas ao armazenamento de dados, melhorando a portabilidade da interface de usuário, uma vez que pode ser dividida em múltiplas plataformas, e a escalabilidade do sistema, simplificando os componentes do servidor;
- *Stateless*: Em uma comunicação *stateless*, toda informação deve necessária para que o servidor interprete uma mensagem corretamente deve estar contida na própria mensagem, não utilizando nenhuma informação já presente no contexto do servidor. Dessa forma, o cliente se torna integralmente responsável pelo estado da mensagem;
- *Cache*: Objetivando melhorar a eficiência da rede, a restrição de *cache* define que toda requisição deve ser marcada como *cacheable* ou *non-cacheable*, pois caso seja marcada como *cacheable*, o cliente pode guardar a mensagem de resposta do servidor para usos posteriores, evitando a necessidade de enviar mensagens redundantes e reduzindo, assim, o fluxo de mensagens e, consequentemente, a latência da rede;

- *Uniform Interface*: Restrição principal do padrão REST, a interface unificada define que aplicações REST devem possuir a mesma interface independente do cliente conectado, permitindo que os usuários interpretem facilmente a lógica por trás da aplicação;
- *Layered System*: A restrição de *layered system* define que o sistema todo deve ser dividido em várias camadas de forma hierárquica, restringindo o acesso de componentes da camada de nível mais baixo a componentes na camada de nível mais alto. Essa restrição reduz a complexidade do sistema e aumenta sua segurança;
- *Code-On-Demand*: Essa restrição consiste em permitir aos clientes de uma rede a aplicação de suas funcionalidades por meio de pequenos *scripts*, de forma a diminuir a necessidade de componentes pré-requisitados para a aplicação. Por reduzir a visibilidade do sistema, essa restrição é dada como opcional;

O padrão REST é bastante utilizado no desenvolvimento de APIs, principalmente envolvendo aplicações com *Web Services*. Quando uma API se encaixa no padrão REST, esta é classificada como RESTful API.

Segundo a Altexsoft (2021), a estrutura de uma API RESTful define um servidor contendo diversos recursos, os quais podem ser acessados por clientes por meio de requisições HTTP. A figura 10 representa a conexão entre um cliente e uma API RESTful.

Figura 10 – Conexão entre cliente e RESTful API



Fonte: Altexsoft (2021).

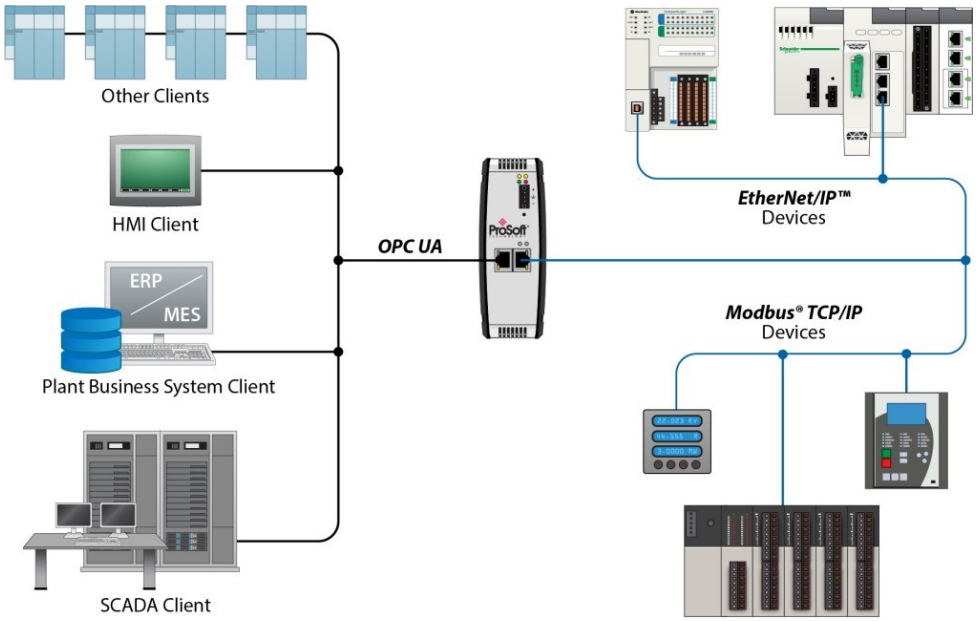
3 DEFINIÇÃO DO PROBLEMA

Considerando as diversas limitações do protocolo *Modbus*, se faz necessária uma solução capaz de realizar a integração de dispositivos *Modbus* a protocolos mais familiarizados com os conceitos da Indústria 4.0.

Essa integração deve ser capaz de estabalecer ao menos uma camada de segurança para a conexão, além de propor um modelo de dados padronizado e estruturado de forma que sua configuração seja realizada da maneira mais intuitiva possível, uma vez que o *Modbus* prevê apenas registradores de 16 *bits* enquanto protocolos mais atuais possuem tipos de dados e estruturas consideravelmente mais complexas. Dessa forma, sistemas de supervisão e controle conectados a equipamentos via *Modbus* poderão usufruir de uma maior escalabilidade, segurança e facilidade de integração.

Considerando essas características, vale ressaltar que já existem algumas soluções no mercado, como por exemplo o *gateway* entre *Modbus* TCP e OPC UA PLX32-EIP-MBTCP-UA da Prosoft (Figura 11). Segundo a Prosoft, o equipamento é capaz de estabelecer uma comunicação estável e segura com até 10 dispositivos conectados na rede.

Figura 11 – *Gateway* PLX32-EIP-MBTCP-UA



Fonte: Prosoft.

4 PROPOSTA DE SOLUÇÃO

Considerando a forma como se estruturam as conexões entre dispositivos, máquinas e sensores no cenário industrial atual, além da perspectiva de futuro que se tem a respeito desse tema, fica claro que a tendência é, cada vez mais, a necessidade de integrar diferentes camadas de software e protocolos de comunicação. Entretanto, diversos equipamentos do mercado ainda possuem uma capacidade extremamente limitada no que se refere a variedade de conexões, suportando somente comunicação via *Modbus* TCP e serial ou, em alguns casos mais extremos, somente conexão *Modbus* serial. Uma das maneiras de solucionar este problema é através de um dispositivo *gateway*, isto é, um conversor entre protocolos que funciona de maneira a intermediar as requisições enviadas por diferentes dispositivos utilizando diferentes protocolos de comunicação.

Para este trabalho, foi proposto o desenvolvimento de um *gateway* que fosse capaz de converter requisições *Modbus* TCP para *Modbus* serial e vice versa, além de requisições *Modbus* TCP para OPC UA e vice versa. Dessa forma, é possível nivelar e integrar dois protocolos de comunicação extremamente distintos, unindo a disponibilidade do protocolo *Modbus* entre os dispotivios do mercado à segurança, robustez e estrutura do OPC UA sem que seja necessário alterar o sistema por completo.

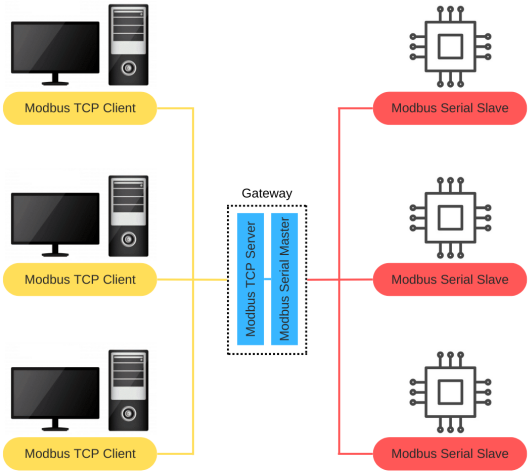
O *hardware* utilizado para executar o *gateway* foi um computador convencional. Considerando que a aplicação exige uma capacidade de processamento significativamente baixa, a mesma poderia ser embarcada em um hardware menor e mais portátil, como uma Raspberry PI, por exemplo, necessitando apenas algumas adaptações de *software*.

O protótipo completo pode ser encontrado em um repositório no *Github*, através do seguinte link: https://github.com/worfero/gateway.

4.1 Arquitetura

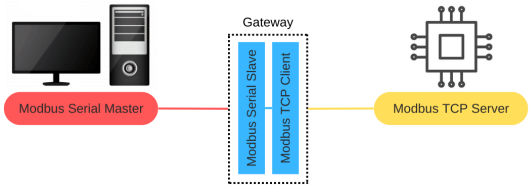
A ideia inicial seria operar tanto a conversão entre *Modbus* TCP e *Modbus* serial quanto a conversão entre *Modbus* TCP e OPC UA de forma simultânea, porém, por limitações de *hardware* (quantidade de portas *Ethernet* disponíveis no computador) e de tempo de desenvolvimento, a ideia se reduziu a operar em apenas um modo por vez. Dessa forma, o *gateway* possui 4 modos de conversão no total, conforme demonstram as figuras 12, 13, 14 e 15.

Figura 12 – *Modbus* TCP para Serial



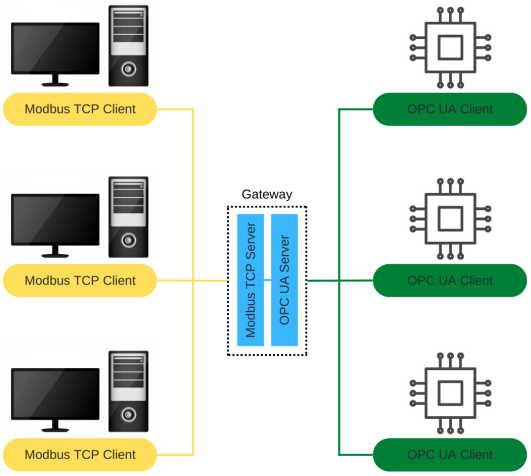
Fonte: Do Autor (2022).

Figura 13 – *Modbus* Serial para TCP



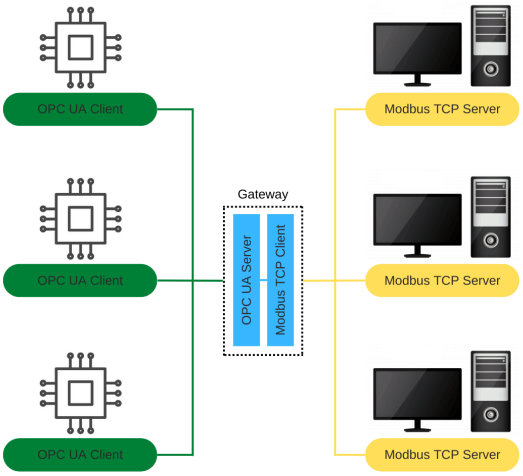
Fonte: Do Autor (2022).

Figura 14 – *Modbus* TCP para OPC UA



Fonte: Do Autor (2022).

Figura 15 – OPC UA para *Modbus* TCP



Fonte: Do Autor (2022).

Com exceção da conversão de *Modbus* serial para *Modbus* TCP, que permite somente um mestre por conexão, todos os modos permitem múltiplos mestres e escravos simultaneamente. A base estrutural de cada modo de operação é a seguinte:

- *Modbus* TCP para *Modbus* serial: O *gateway* recebe a mensagem enviada via *Modbus* TCP através de um servidor local e, imediatamente, redireciona a mensagem por um mestre *Modbus* serial para os escravos especificados, utilizando a porta e as configurações (*baudrate*, paridade, etc.) determinadas pelo usuário através da página de configuração na *web*;

- *Modbus* serial para *Modbus* TCP: O *gateway* recebe a mensagem enviada via *Modbus* serial através de um servo local e, imediatamente, redireciona a mensagem por um cliente *Modbus* TCP para o servidor especificado, utilizando a porta e as configurações (*baudrate*, paridade, etc.) determinadas pelo usuário através da página de configuração na *web*;
- *Modbus* TCP para OPC UA: O usuário define, através da página de configuração na *web*, uma relação entre as *tags* OPC UA, seu tipo (inteira, *booleana*, etc.) e um determinado registrador *Modbus*, assim o *gateway* recebe a mensagem enviada via *Modbus* TCP através de um servidor local e cria também um servidor OPC UA com as *tags* determinadas pelo usuário, associando a mensagem à *tag* especificada, permitindo acesso de leitura a quaisquer clientes OPC UA interessados na informação armazenada na *tag*.;
- OPC UA para *Modbus* TCP: O usuário define, através da página de configuração na *web*, uma relação entre as *tags* OPC UA, seu tipo (inteira, *booleana*, etc.) e um determinado servidor *Modbus*, informando seu IP, ID e registrador no qual o valor deve ser escrito, assim o *gateway* recebe a mensagem enviada via OPC UA através de um servidor local e redireciona os valores das *tags* aos servidores *Modbus* determinados pelo usuário.

4.2 Ferramentas

O funcionamento da aplicação foi programado, integralmente, utilizando *Python*, devido a sua grande gama de bibliotecas, flexibilidade e facilidade para desenvolvimento de pequenos *softwares* de automação e aplicações *Web*.

Dentre as bibliotecas utilizadas, destacam-se a *pymodbus* e a *Free OPC-UA*, possuindo grande contribuição para o funcionamento geral do sistema, uma vez que possuem os métodos responsáveis por inicializar e configurar os clientes e servidores *Modbus* e OPC UA. Também são importantes a biblioteca *Flask* juntamente de suas extensões *flask_restful* e *flask_sqlalchemy*, utilizadas para incorporar o *microframework Flask* a aplicação, facilitando imensamente a implementação das páginas *Web*, além de permitir o desenvolvimento da API *RESTful*, onde se encontram os principais métodos do dispositivo, e de bancos de dados, para armazenar as *tags* do dispositivo.

O *layout* das páginas *web* foi desenvolvido utilizando HTML e CSS simples, pois a aplicação não necessita de estruturas de página complexas.

Os arquivos de configuração foram armazenados em formato JSON devido a sua baixa complexidade e facilidade de acesso utilizando *Python*.

Por fim, foi utilizado *SQLAlchemy*, como mencionado anteriormente, para armazenamento das *tags* utilizadas nas conversões envolvendo OPC UA.

4.3 Interface de Configuração

A aplicação possui, ao total, seis páginas *web* distintas: a *home page*, as quatro páginas de configuração (uma para cada modo de operação), e duas para criação de novas *tags* (uma para cada modo de conversão entre *Modbus* TCP e OPC UA).

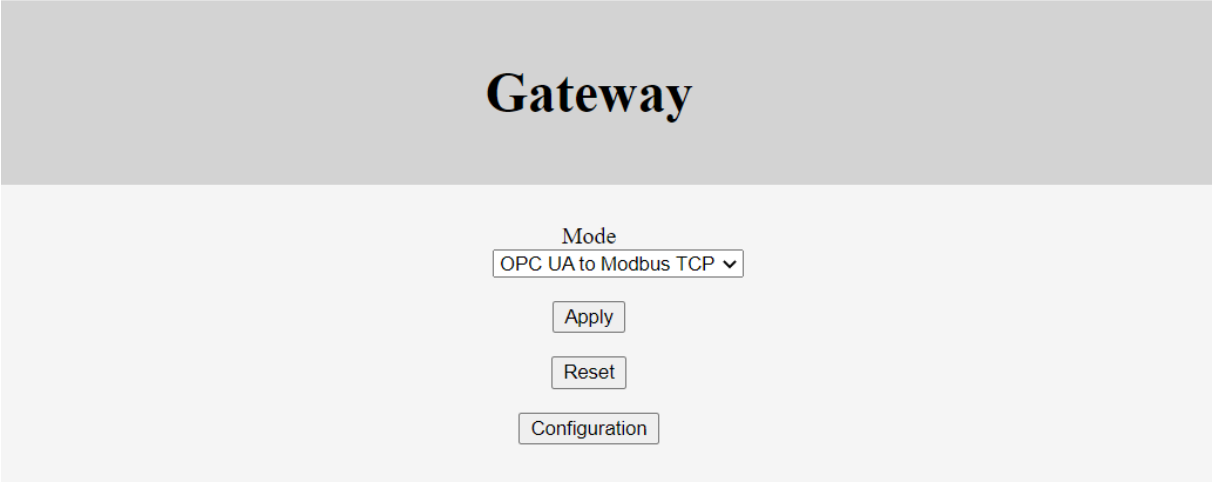
4.3.1 *Home page*

A *home page* se encontra no endereço raíz da aplicação *web*, que no caso da aplicação em modo teste, se trata do domínio *localhost:5000*, ou seja, o IP local do computador que está rodando o programa e utilizando a porta 5000. Caso a aplicação estivesse rodando em um sistema embarcado, a página *web* seria acessada através do IP do dispositivo.

Nessa página, o usuário pode selecionar o modo de operação que será utilizado pela aplicação, dentre as quatro opções citadas anteriormente. Após selecionar o modo desejado, deve-se pressionar o botão *Apply* para que as novas condições sejam salvas e utilizadas na próxima inicialização do dispositivo, ou seja, para que a alteração de modo comece a funcionar, é necessário pressionar o botão *Reset*. Além disso, a página inicial possui um botão chamado *Configuration* que, ao ser pressionado, redireciona o usuário para uma tela de configuração, a depender do modo de operação selecionado.

O layout completo da página inicial pode ser observado na figura 16.

Figura 16 – *Home page*



Fonte: Do Autor (2022).

4.3.2 Configuração de *Modbus* TCP para *Modbus* serial

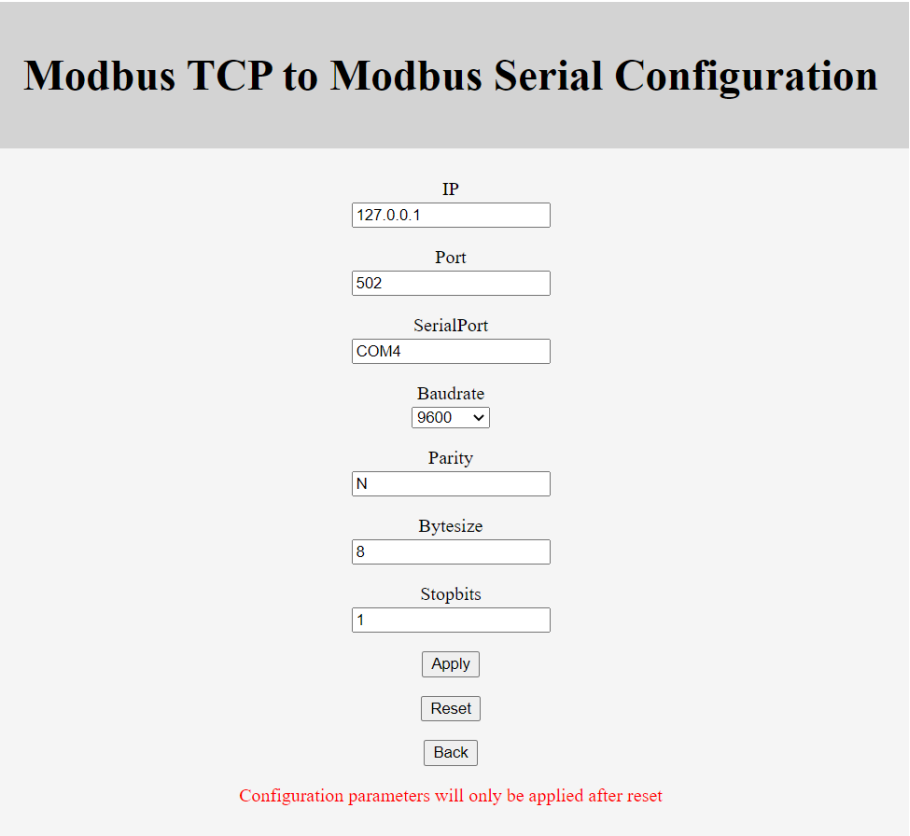
Dentre as páginas de configuração do dispositivo, há a configuração do modo de conversão *Modbus* TCP para *Modbus* serial, a qual pode ser acessada através do

endereço /tcp_to_serial ou através do botão *Configuration* na *home page*, quando este modo de operação estiver selecionado.

Para este modo, é necessário configurar as condições de entrada, isto é, o IP do dispositivo (sendo, nesse caso, o IP local do computador) e também a sua porta, dessa forma permitindo o acesso de qualquer cliente *Modbus* TCP interessado. De forma similar, deve-se configurar as condições de saída, ou seja, a porta serial, sua *baudrate*, paridade, *stopbits* e *bytesize*. As configurações são salvas pressionando o botão *Apply*, porém somente começam a surtir efeito após reiniciar o dispositivo, pressionando o botão *Reset*.

A página em questão pode ser observada na figura 17.

Figura 17 – Configuração *Modbus* TCP para Serial



Fonte: Do Autor (2022).

4.3.3 Configuração de *Modbus* serial para *Modbus* TCP

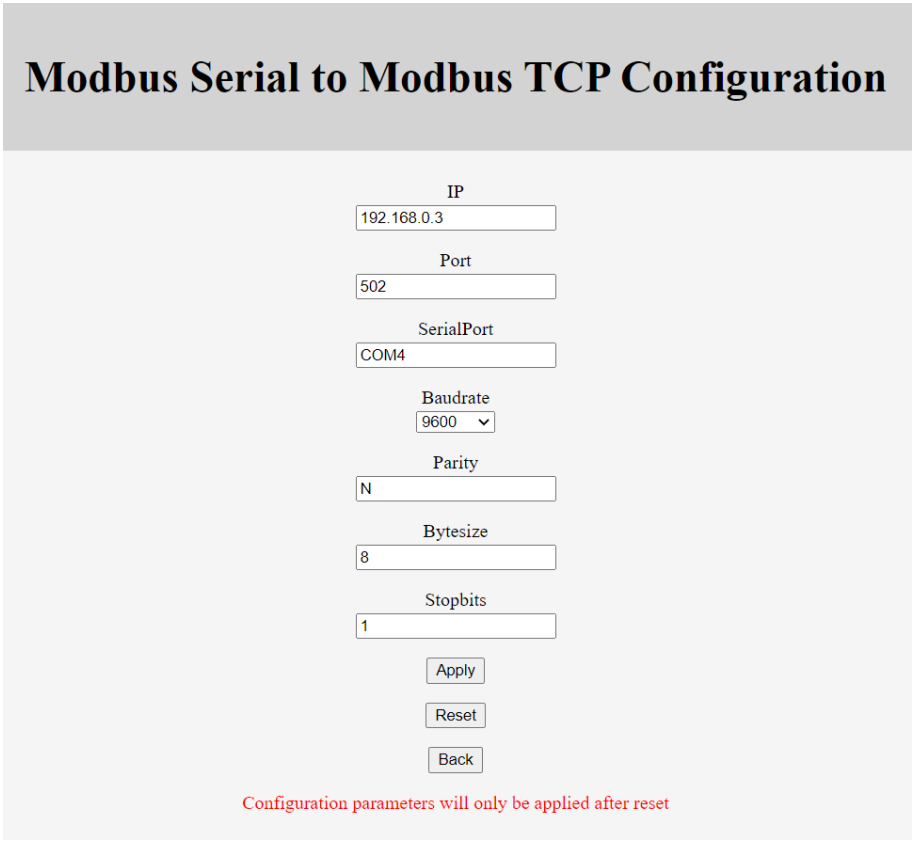
Nessa página são aplicadas as configurações do modo de conversão *Modbus* serial para *Modbus* TCP, podendo ser acessada através do endereço /serial_to_tcp ou através do botão *Configuration* na *home page*, quando este modo de operação estiver selecionado.

De forma similar a página descrita na seção 4.3.2, aqui são configuradas as condições de entrada e de saída. Entretanto, para esse caso a porta serial representa

as condições de entrada, permitindo acesso de qualquer mestre *Modbus* serial que possua as mesmas especificações, já na saída deve ser especificado o IP e a porta do servidor *Modbus* TCP. Novamente, as condições são salvas ao pressionar o botão *Apply* e efetivadas ao reiniciar o dispositivo, pressionando o botão *Reset*.

A página pode ser observada na figura 18.

Figura 18 – Configuração *Modbus* Serial para TCP



Fonte: Do Autor (2022).

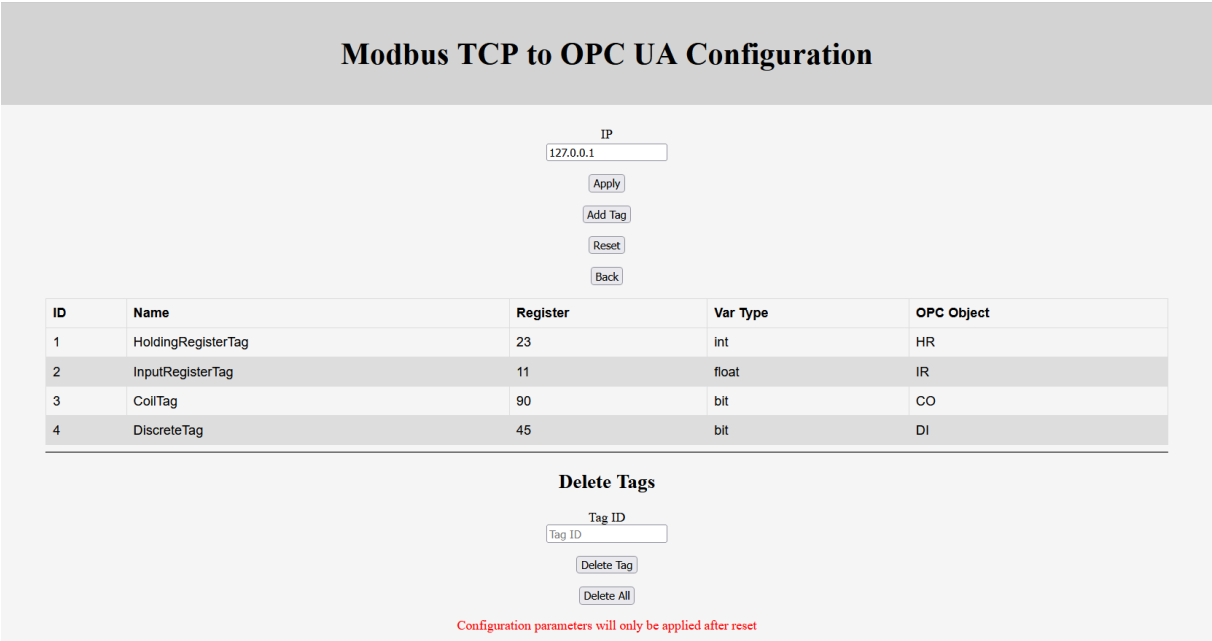
4.3.4 Configuração de *Modbus* TCP para OPC UA

Nessa página são configurados os parâmetros para a conversão de requisições *Modbus* TCP para *tags* OPC UA, sendo acessada através do endereço /tcp_to_opc ou através do botão *Configuration* na *home page*, quando este modo de operação estiver selecionado.

Devido ao fato do *Modbus* e do OPC UA se tratarem de protocolos consideravelmente diferentes, a configuração é um pouco mais complexa. Nesse caso, além de determinar o IP do dispositivo (nesse caso, o IP local do computador), devem-se criar objetos OPC UA com *tags* associadas a um registrador *Modbus*, onde os objetos representam o tipo de registrador. As *tags* podem ser criadas pressionando o botão *Add Tag*, que redireciona à página de criação de *tags* específica para este método, descrita na seção 4.3.6.

A figura 19 representa o layout desenvolvido para a página.

Figura 19 – Configuração *Modbus* TCP para OPC UA



Fonte: Do Autor (2022).

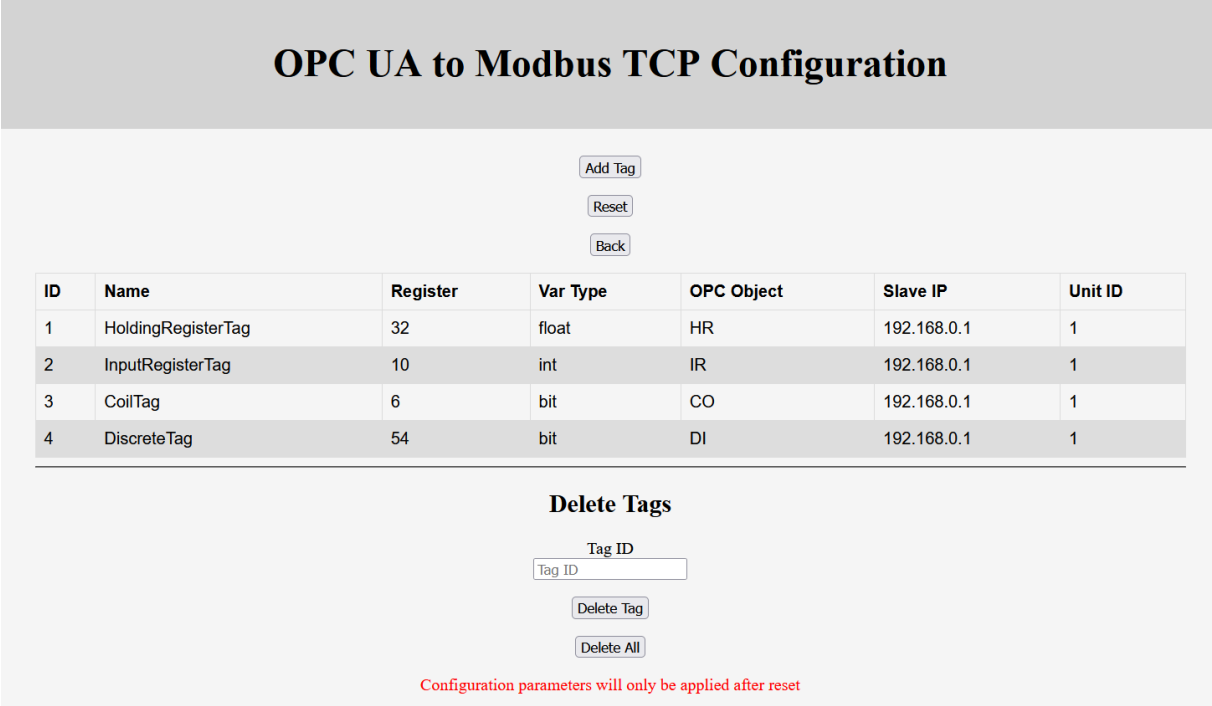
4.3.5 Configuração de OPC UA para *Modbus* TCP

De forma similar às anteriores, para acessar a página de configuração do modo de conversão OPC UA para *Modbus* TCP, pode-se clicar no botão *Configuration* na *home page* ou acessar o endereço /opc_to_tcp.

O método utilizado para associar as *tags* aos registradores funciona de forma similar ao da seção 4.3.4, com a diferença de que, desta vez, deve-se também especificar o IP e a porta dos servidores *Modbus* que cada tag deseja alcançar.

A página pode ser observada na figura 20.

Figura 20 – Configuração OPC UA para *Modbus* TCP



Fonte: Do Autor (2022).

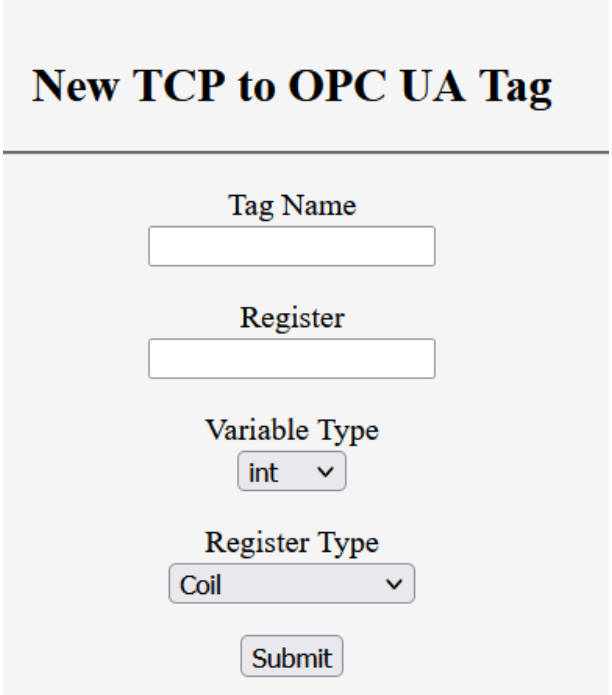
4.3.6 Nova *tag* de *Modbus* TCP para OPC UA

Essa página é acessada ao apertar o botão *New tag* presente na página de configuração de *Modbus TCP* para OPC UA, observada na figura 19, ou através do endereço /new_tcp_to_opc_tag.

É solicitado um formulário com informações necessárias para o cadastro da *tag*, como nome da *tag*, tipo de variável, endereço *Modbus* associado e objeto OPC UA (tipo de registrador). Para finalizar o cadastro, basta pressionar o botão *Submit*.

O layout da página está representado na figura 21.

Figura 21 – Nova Tag de *Modbus* TCP para OPC UA



Fonte: Do Autor (2022).

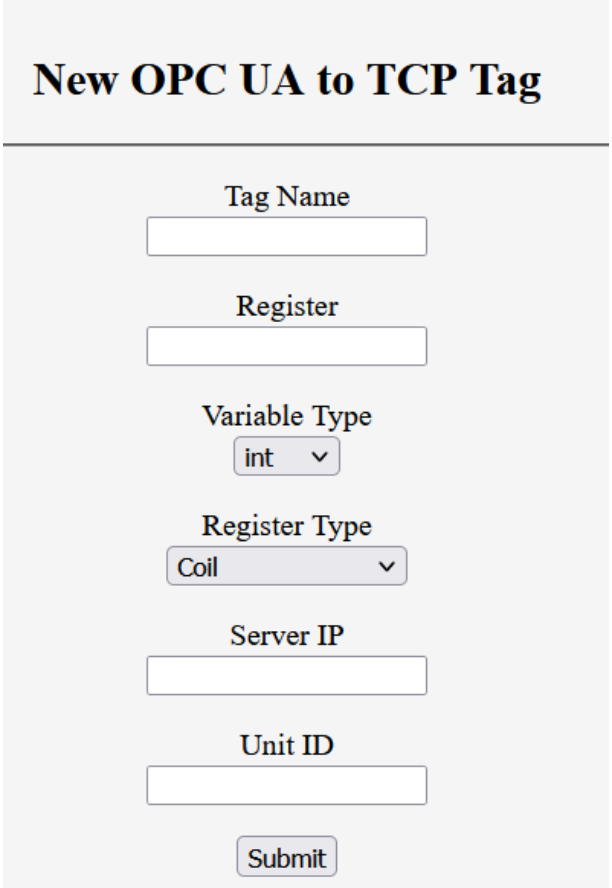
4.3.7 Nova *tag* de OPC UA para *Modbus* TCP

Essa página é acessada ao apertar o botão *New tag* presente na página de configuração de OPC UA para *Modbus TCP*, observada na figura 20, ou através do endereço /new_opc_to_tcp_tag.

De forma semelhante a página do capítulo 4.3.6, é apresentado um formulário para cadastro de uma nova *tag*, porém, nesse caso, também é requisitado o IP do servidor *Modbus* que receberá a informação da *tag*.

A página pode ser observada na figura 22.

Figura 22 – Nova Tag de *Modbus* TCP para OPC UA



Fonte: Do Autor (2022).

4.4 Código e Funcionamento

A aplicação é estruturada envolvendo 4 seguintes grupos de componentes:

- Código Principal: É o coração da aplicação, onde todos os outros componentes se interconectam e são processados. Consiste em uma aplicação *Flask* de múltiplas páginas integrada a uma API *RESTful* com recursos para realizar as conversões;
- Arquivos de Configuração: São arquivos em formato JSON lidos eventualmente pelo código. Incluem seleção de modo de operação e configurações de parâmetros salvas para cada modo;
- Bancos de Dados: Correspondem arquivos de DBs que contém as tags OPC UA cadastradas pelo usuário na conversão bem como seu endereço *Modbus* associado. Além disso, há um banco de dados com usuários cadastrados para autenticação;
- Templates: São arquivos HTML que correspondem aos *layouts* de cada página do programa.

O código principal, portanto, tem o trabalho de integrar todas as componentes restantes, bem como assegurar o funcionamento de cada modo de operação.

4.4.1 Bancos de Dados

Por se tratarem de bancos de dados pequenos que priorizam a eficiência e o tempo de resposta, todos os bancos de dados utilizados no desenvolvimento da aplicação são baseados na *engine* SQLite, sendo este ideal para aplicações pequenas, rápidas, independentes e confiáveis (SQLITE, 2022). Para a criação dos bancos de dados no código, foi utilizada a biblioteca *Flask-SQLAlchemy*. A figura 23 demonstra como é declarado o banco de dados para *tags* utilizadas na conversão de *Modbus* TCP para OPC UA.

Figura 23 – Declaração de Banco de Dados

```
588  class TCP_to_OPC_Tags(db.Model):
589      id = db.Column('tag_id', db.Integer, primary_key = True)
590      name = db.Column(db.String(100))
591      register = db.Column(db.String(10))
592      var_type = db.Column(db.String(10))
593      opc_object = db.Column(db.String(100))
594
595      def __init__(self, name, register, var_type, opc_object):
596          self.name = name
597          self.register = register
598          self.var_type = var_type
599          self.opc_object = opc_object
```

Fonte: Do Autor (2022).

Como pode-se observar, os bancos de dados são definidos como classes, onde pode-se definir qual o tipo de dado de cada coluna, qual a *primary key* (nesse caso, uma contagem numérica simples) e qual a quantidade máxima de caracteres disponibilizados para cada coluna. Esse processo se repete para a criação dos demais bancos de dados.

Por fim, são instanciados junto da aplicação *Flask*, como na figura 24.

Figura 24 – Instância dos Banco de Dados

```
568  # databases instance
569  app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db/tcp_to_opc.sqlite3'
570  app.config['SQLALCHEMY_BINDS'] = {
571      "opc_to_tcp": "sqlite:///db/opc_to_tcp.sqlite3",
572      "users": "sqlite:///db/userdb.sqlite3"
573      }
574  db = SQLAlchemy(app)
```

Fonte: Do Autor (2022).

4.4.2 Formulários

A aplicação utiliza diversos formulários para a configuração de parâmetros e seleção de modo, como visto na seção 4.3. Para a criação destes formulários, foi usada a biblioteca *WTForms*, que transforma formulários em classes e facilita a integração dos elementos HTML com os elementos em *Python*.

A figura 25 representa a definição de uma classe de formulários para configuração de parâmetros da conversão entre *Modbus* TCP e *Modbus* Serial.

Figura 25 – Declaração de formulário para configuração de parâmetros

```
52  # modbus TCP/modbus serial conversion settings form
53  class ModbusConfigs(Form):
54      ip = StringField('IP', [validators.Length(min=0, max=15)])
55      port = StringField('Port', [validators.Length(min=0, max=6)])
56      serial_port = StringField('SerialPort', [validators.Length(min=0, max=8)])
57      baudrate = SelectField(u'Baudrate', coerce=int, choices=[(4800, "4800"), (9600, "9600"),
58          (19200, "19200"), (57600, "57600"), (115200, "115200")])
59      parity = StringField('Parity', [validators.Length(min=0, max=4)])
60      bytesize = StringField('Bytesize', [validators.Length(min=0, max=1)])
61      stopbits = StringField('Stopbits', [validators.Length(min=0, max=1)])
```

Fonte: Do Autor (2022).

Em suma, o cadastro de um campo do formulário contempla tipo de variável (string, int, float, etc.), nome do campo e tamanho mínimo e máximo do valor inserido em caracteres. Especificamente para o caso de *SelectField*, é necessário explicitar suas opções e quais valores cada opção representa.

Todos os outros formulários utilizados na aplicação (configuração de parâmetros para conversão entre *Modbus* TCP e OPC UA, seleção de modo e cadastro de *tags*) foram definidos de forma semelhante.

4.4.3 Arquivos de Configuração

As configurações de parâmetros e seleção de modo são armazenadas em arquivos JSON, dessa forma o sistema sempre terá salvas as últimas configurações

utilizadas pela aplicação toda vez que o processo for finalizado ou reiniciado. A figura 26 demonstra o arquivo de configuração de parâmetros para conversão entre *Modbus* TCP e *Modbus* Serial.

Figura 26 – Arquivo de configuração em JSON

```
1  {"ip": "127.0.0.1", "port": 502, "serial_port": "COM4", "baudrate": 9600, "bytesize": 8, "parity": "N", "stopbits": 1}
```

Fonte: Do Autor (2022).

4.4.4 Páginas

Todas as páginas descritas na seção 4.3 foram programadas utilizando *Flask* para a dinâmica de operação e HTML/CSS para design dos templates, os quais se relacionam com o código em *Python* através da função *@app.route()* disponibilizada pelo *Flask*.

4.4.4.1 Home page e configuração de Modbus TCP e Modbus Serial

A figura 27 representa a *home page* em *Python* enquanto a figura 28 demonstra a *home page* em HTML e CSS.

Figura 27 – *Home page* em *Python*

```python
@app.route("/", methods=['GET', 'POST'])
def hello():
    global restart
    with open('config/Mode.json', 'r') as openfile:
        lastConfig = json.load(openfile)
    form = Mode(request.form, mode=lastConfig['mode'])
    if request.method == 'POST' and form.validate():
        mode = form.mode.data

        payload = {'mode': mode}
        mode_config = json.dumps(payload)
        if request.form['action'] == 'Apply':
            with open("config/Mode.json", "w") as outfile:
                outfile.write(mode_config)
            return redirect(url_for('hello'))
        elif request.form['action'] == 'Reset':
            restart = 1
            return redirect(url_for('hello'))
        elif request.form['action'] == 'Configuration':
            if mode == TCP_TO_SERIAL:
                return redirect(url_for('tcp_to_serial'))
            elif mode == SERIAL_TO_TCP:
                return redirect(url_for('serial_to_tcp'))
            elif mode == TCP_TO_OPC:
                return redirect(url_for('tcp_to_opc'))
            elif mode == OPC_TO_TCP:
                return redirect(url_for('opc_to_tcp'))
    return render_template('index.html', form=form)
```

Fonte: Do Autor (2022).

Assim como em todas as páginas que possuem um botão de *reset*, a variável global *restart* deve ser declarada, pois é necessária para disparar o *event* descrito na seção 4.4.5. Após isso, é aberto o arquivo JSON de seleção de modo para que seja mostrada na tela a última seleção, junto das outras opções de modo em um *SelectField*, puxado da classe de formulários *Mode*. Ao clicar em qualquer botão da página, o sistema recebe uma requisição HTTP POST, e então interpreta qual botão foi pressionado. Ao clicar em *Apply*, o modo selecionado é aplicado ao arquivo JSON, ao clicar no botão *Configuration*, a aplicação redireciona o usuário para a tela de configuração específica do modo selecionado, e por fim, ao clicar no botão *Reset*, a aplicação é reiniciada.

Figura 28 – *Home page* em HTML

```
{% from "_formhelpers.html" import render_field %}

<style>
  /* Style the header */
  header {
    background-color: #D3D3D3;
    padding: 10px;
    padding-left: 50px;
    padding-right: 50px;
    text-align: center;
    font-size: 25px;
    color: black;
  }
</style>

<header>
  <h2>Gateway</h2>
</header>
<div style="text-align: center; background-color: #F5F5F5; padding: 10px;">
  <form method=post action="">
    <dl display="inline-block;">
      {{ render_field(form.mode) }}
    </dl>
    <p><input type=submit name="action" value="Apply"></p>
    <p><input type=submit name="action" value="Reset"></p>
    <p><input type=submit name="action" value="Configuration"></p>
  </form>
  <p style="color: red;">Changes will only be applied after reset</p>
</div>
```

Fonte: Do Autor (2022).

Pelo lado HTML da página, inicialmente são definidas as configurações CSS e, em seguida, os elementos disponíveis, como textos, formulário e botões.

Além disso, na primeira linha é incluída uma macro utilizada para facilitar a declaração de formulários em HTML. A macro pode ser observada na figura 29.

Figura 29 – Macro para criação de formulários

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd style="padding-right: 40px; padding-top: 2px;">{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
{% endmacro %}
```

Fonte: Do Autor (2022).

As páginas de configuração para conversão entre *Modbus* TCP e *Modbus* Serial são programadas utilizando essa mesma metodologia.

4.4.4.2 Configuração de Modbus TCP e OPC UA

Para as páginas de configuração para conversão entre *Modbus* TCP e OPC UA, além da metodologia descrita na seção 4.4.4.1, também é necessário incluir as interações com o banco de dados de *tags*, como demonstra a figura 30.

Figura 30 – Interação com o banco de dados de tags

```
762         elif request.form['action'] == 'Add Tag':
763             return redirect(url_for('new_tcp_to_opc_tag'))
764
765         elif request.form['action'] == 'Delete Tag':
766             tag_id = request.form['tagID']
767             obj = TCP_to_OPC_Tags.query.filter_by(id=int(tag_id)).one()
768             db.session.delete(obj)
769             db.session.commit()
770             time.sleep(1)
771             return redirect(url_for('tcp_to_opc'))
772
773         elif request.form['action'] == 'Delete All':
774             db.session.query(TCP_to_OPC_Tags).delete()
775             db.session.commit()
776             time.sleep(1)
777             return redirect(url_for('tcp_to_opc'))
```

Fonte: Do Autor (2022).

Nesse caso, para a adição de *tags*, a aplicação redireciona o usuário para uma página com um formulário para cadastro de *tags* (seção 4.4.4.3). Para a remoção de uma *tag* específica, primeiramente é realizada uma *query* em busca da *tag* em

questão com base em sua *primary key* e, em seguida, essa *tag* é excluída do banco de dados pela função *session.delete(obj)*. Já para a remoção de todas as *tags*, todos os dados do banco são deletados pela função *session.delete()*.

Todas as funções de interação com os bancos de dados são disponibilizadas pela biblioteca *SQLAlchemy*.

Para o funcionamento do lado HTML da página, é necessário que uma *query* do banco de dados seja passada como argumento pela função *render_template()*, demonstrado pela figura 31.

Figura 31 – *Tags* do banco de dados enviadas para o *template*

```
785        return render_template('tcp_to_opc.html', tags = TCP_to_OPC_Tags.query.all(), form=form)
```

Fonte: Do Autor (2022).

No código HTML da página em si, além da metodologia utilizada na seção 4.4.4.1, é adicionada a tabela com as *tags* presentes no banco de dados, como demonstra a figura 32.

Figura 32 – Declaração da tabela de *tags* em HTML

```
50              <table>
51                  <thead>
52                      <tr>
53                          <th>ID</th>
54                          <th>Name</th>
55                          <th>Register</th>
56                          <th>Var Type</th>
57                          <th>OPC Object</th>
58                      </tr>
59                  </thead>
60
61                  <tbody>
62                      {% for tag in tags %}
63                          <tr>
64                              <td>{{ tag.id }}</td>
65                              <td>{{ tag.name }}</td>
66                              <td>{{ tag.register }}</td>
67                              <td>{{ tag.var_type }}</td>
68                              <td>{{ tag.opc_object }}</td>
69                          </tr>
70                      {% endfor %}
71                  </tbody>
72              </table>
```

Fonte: Do Autor (2022).

4.4.4.3 Adição de tags

Para as páginas de adição de *tags* na conversão entre *Modbus* TCP e OPC UA, além dos métodos utilizados nas seções 4.4.4.1 e 4.4.4.2, também é necessária a utilização da função *session.add()*, como exemplificado na figura 33.

Figura 33 – Código para formulário e adição de novas *tags*

```
819  # ------------------------------------------------------------------------ #
820  # modbus TCP to OPC UA tag creation page
821  # ------------------------------------------------------------------------ #
822  @app.route('/new_tcp_to_opc_tag', methods = ['GET', 'POST'])
823  def new_tcp_to_opc_tag():
824      form = OPC_to_TCP_Configs(request.form, name="", register="", var_type="", opc_object="")
825      if request.method == 'POST' and form.validate():
826          name = form.name.data
827          register = form.register.data
828          var_type = form.var_type.data
829          opc_object = form.opc_object.data
830
831          tag = TCP_to_OPC_Tags(name, register, var_type, opc_object)
832
833          db.session.add(tag)
834          db.session.commit()
835          return redirect(url_for('tcp_to_opc'))
836      return render_template('new_tcp_to_opc_tag.html', form=form)
```

Fonte: Do Autor (2022).

No que diz respeito ao HTML dessas páginas, é realizado um formulário simples de forma semelhante às seções anteriores.

4.4.5 Inicialização

Ao ser inicializado, primeiramente o programa interpreta o código disponibilizado na figura 34.

Figura 34 – Inicialização do Processo

```
918  if __name__ == "__main__":
919      db.create_all()
920      #user = UserDB("admin", "admin")
921      #db.session.add(user)
922      #db.session.commit()
923      event = multiprocessing.Event()
924      p = Process(target=boot, args=(event,))
925      p.start()
926      while True:
927          if event.is_set():
928              p.terminate()
929              args = [sys.executable] + [sys.argv[0]]
930              subprocess.call(args)
```

Fonte: Do Autor (2022).

A função *db.create_all()* tem o papel de criar os arquivos dos bancos de dados instanciados na figura 24 caso os mesmos ainda não existam.

Também é utilizada a biblioteca *multiprocessing* para criar processos e subprocessos, ferramenta recorrida para reiniciar o programa sem que seja necessário fechar a aplicação. Dessa forma, é criado um processo apontando para a função *boot()*, junto de um *event*, que consiste em uma condição a qual dispara a função *is_set()* que, quando realizada, encerra o processo e inicia um novo.

Figura 35 – Função *boot()*

```
900  def boot(event):
901      # sets the event for the api to restart with new config
902      def reset():
903          global restart
904          while True:
905              if restart == 1:
906                  time.sleep(0.5)
907                  event.set()
908      # Assures the reset check loop will only start after application start
909      reset_check = threading.Thread(target=reset)
910      reset_check.start()
911      # starts the application
912      start_runner()
913      app.run()
```

Fonte: Do Autor (2022).

A função *boot()* (Figura 35), então, cria uma *thread* para checar frequentemente se a variável global *restart* foi acionada, disparando o *event* responsável por reiniciar o programa. Após dar início à *thread*, são invocadas as funções *run()*, para dar início à aplicação, e *start_runner()*, que leva para a interpretação do modo de operação selecionado.

4.4.6 Seleção dos Modos de Operação

A função *start_runner()* (Figura 36) consiste em uma *thread* que, a depender do modo selecionado no arquivo JSON *Mode*, aponta para uma determinada função da classe *Starters* responsável por criar os clientes e/ou servidores *Modbus* TCP/Serial e OPC UA. É necessário que esse processo seja repetido através de uma *thread* até que a conexão esteja estabelecida com sucesso, condição estabelecida pela variável *not_started*, que é variada através do *return* das funções da classe *Starters*.

Figura 36 – Função *start_runner()*

```
862  def start_runner():
863      def start_loop():
864          not_started = True
865          while not_started:
866              print('In start loop')
867
868              # reads selected mode from mode selection file
869              with open('config/Mode.json', 'r') as openfile:
870                  mode = json.load(openfile)
871
872              # modbus TCP to modbus serial conversion mode
873              if mode['mode'] == TCP_TO_SERIAL:
874                  not_started = Starters.startTCPtoSerial()
875
876              # modbus serial to modbus TCP conversion mode
877              elif mode['mode'] == SERIAL_TO_TCP:
878                  not_started = Starters.startSerialToTCP()
879
880              # modbus TCP to OPC UA conversion mode
881              elif mode['mode'] == TCP_TO_OPC:
882                  not_started = Starters.startTCPtoOPC()
883
884              # OPC UA to modbus TCP conversion mode
885              elif mode['mode'] == OPC_TO_TCP:
886                  not_started = Starters.startOPCtoTCP()
887              # ------------------------------------------------------------------------ #
888              # exception
889              # ------------------------------------------------------------------------ #
890              else:
891                  print('Invalid mode selected.')
892
893      print('Started runner')
894      start_api = threading.Thread(target=start_loop)
895      start_api.start()
```

Fonte: Do Autor (2022).

A classe *Starters* (Figura 37) possui 4 funções principais: *startTCPtoSerial()*, *startSerialToTCP()*, *startTCPtoOPC()* e *startOPCtoTCP*, ou seja, uma para iniciar cada modo de operação. Além disso, possui outras 6 funções auxiliares para realizar a requisição dos recursos da API RESTful dentro da aplicação (seção 4.4.7), que são classes especiais responsáveis tanto pela criação de clientes e servidores *Modbus* e OPC UA quanto pela conversão das mensagens.

Figura 37 – Classe *Starters*

```
82  class Starters():
83      # modbus TCP to modbus serial conversion mode
84  >   def startTCPtoSerial():···
100
101     # modbus serial to modbus TCP conversion mode
102 >   def startSerialToTCP():···
118
119     # modbus TCP to OPC UA conversion mode
120 >   def startTCPtoOPC():···
138
139     # OPC UA to modbus TCP conversion mode
140 >   def startOPCtoTCP():···
153
154 >   def modbusTCPServerStart(pay, head):···
157
158 >   def TcpClientOPCServerStart(head):···
161
162 >   def opcClientStart(head):···
165
166 >   def opcServerStart(head):···
169
170 >   def serialForwarderStart(pay, head):···
173
174 >   def tcpForwarderStart(pay, head):···
```

Fonte: Do Autor (2022).

As 4 funções principais da classe *Starters* funcionam de maneira semelhante, variando somente alguns aspectos. Como exemplo, pode-se observar a função *startTCPtoSerial()* na figura 38.

Figura 38 – Função *startTCPtoSerial()*

```
84      def startTCPtoSerial():
85          with open('config/TCP_to_Serial.json', 'r') as openfile:
86              payload = json.load(openfile)
87          try:
88              r = requests.get('http://127.0.0.1:5000/')
89              if r.status_code == 200:
90                  print('Starting server...')
91                  pay = payload
92                  head = {'content-type': 'application/json'}
93                  startForwarder = threading.Thread(target=Starters.serialForwarderStart, args=(pay, head,))
94                  startForwarder.start()
95                  not_started = False
96          except:
97              print('Server not yet started')
98              not_started = True
99          return not_started
```

Fonte: Do Autor (2022).

Primeiramente, é carregado o arquivo de configuração JSON competente a esse modo, nesse caso, *TCP_to_Serial.json*. Em seguida, é realizado um *request* HTTP para assegurar que a aplicação já foi iniciada (como visto na seção 4.4.5). Caso a *request* obtenha sucesso, é iniciada uma *thread* apontando para a função *serialForwarderStart()* junto dos parâmetros lidos pelo arquivo JSON (Figura 39), responsável por acionar o recurso de API para conversão de *Modbus* TCP para *Modbus* Serial (seção 4.4.7.1).

Figura 39 – Função auxiliar *tcpForwarderStart()*

```
174     def tcpForwarderStart(pay, head):
175         req = requests.post('http://localhost:5000/modbus-explorer/api/mbusTCPforwarder', data=json.dumps(pay),
176                             headers = head)
```

Fonte: Do Autor (2022).

Para as funções principais *startSerialToTCP()*, a estrutura utilizada é a mesma, variando somente o arquivo de configuração JSON, a função auxiliar chamada e, portanto, o recurso de API utilizado.

4.4.7 Recursos da API

Os recursos da API *RESTful* integrados à aplicação se tratam da parte mais importante do sistema, afinal é através deles que as conversões são realizadas.

No total, há 5 recursos disponíveis, sendo eles:

- *ModbusSerialForwarder*: Recebe requisições *Modbus* através de um servidor via TCP e redireciona a mensagem para o servidor desejado através de um cliente serial;

- *ModbusTcpForwarder*: Recebe requisições *Modbus* através de um servidor via serial e redireciona a mensagem para o servidor desejado através de um cliente TCP;
- *ModbusTCPServer*: Servidor *Modbus* TCP simples que receberá as mensagens que serão redirecionadas posteriormente via OPC UA;
- *OPCUAServer*: Servidor OPC UA que recebe os valores de registradores estipulados por suas *tags*.
- *OPCUAClient*: Cliente OPC UA com um pequeno cliente *Modbus* TCP imbutido que age como pivô ao ler os registradores do recurso *ModbusTCPServer* e enviá-los para o recurso *OPCUAServer*.
- *TcpClientOPCServer*: Servidor OPC UA com um cliente *Modbus* TCP embutido que envia os valores de suas *tags* para os registradores e endereços de IP associados a elas.

A tabela 3 demonstra a relação entre modo selecionado e recursos utilizadas.

Tabela 3 – Relação entre modo selecionado e recursos de API utilizados

Modo	Recursos
Modbus TCP para *Modbus* Serial	*ModbusSerialForwarder*
Modbus Serial para *Modbus* TCP	*ModbusTcpForwarder*
Modbus TCP para OPC UA	*ModbusTCPServer*, *OPCUAServer* e *OPCUAClient*
OPC UA para *Modbus* TCP	*TcpClientOPCServer*

Fonte: Do Autor (2022).

4.4.7.1 ModbusSerialForwarder

Primeiramente, o recurso inicializa uma função de *RequestParser()*, responsável por receber via *request* HTTP os parâmetros de configuração do arquivo JSON enviados por sua respectiva função na classe *Starters*, como exemplifica a figura 40.

Figura 40 – Função *RequestParser()* do recurso *ModbusTcpForwarder*

```
230     def __init__(self):
231         # ------------------------------------------------------------------------ #
232         # argument parsing
233         # ------------------------------------------------------------------------ #
234         self.reqparse = reqparse.RequestParser()
235         self.reqparse.add_argument('ip', type=str, required=True, location='json',
236                                    help='No IP address provided')
237         self.reqparse.add_argument('port', type=int, required=True, location='json',
238                                    help='No port provided')
239         self.reqparse.add_argument('serial_port', type=str, required=True, location='json',
240                                    help='No serial port provided')
241         self.reqparse.add_argument('baudrate', type=int, required=True, location='json',
242                                    help='No baudrate provided')
243         self.reqparse.add_argument('bytesize', type=int, required=True, location='json',
244                                    help='No bytesize provided')
245         self.reqparse.add_argument('parity', type=str, required=True, location='json',
246                                    help='No parity provided')
247         self.reqparse.add_argument('stopbits', type=int, required=True, location='json',
248                                    help='No stopbits provided')
249         super(ModbusSerialForwarder, self).__init__()
```

Fonte: Do Autor (2022).

Como pode-se observar, o recurso recebe as configurações de IP e porta, que são competentes ao próprio *gateway*, e as de porta serial, *baudrate*, *byte size*, paridade e *stop bits*, competentes aos servidores serial que se deseja alcançar. O ID do servidor desejado é inserido diretamente na mensagem enviada.

Após isso, é utilizada a biblioteca *pymodbus* para instanciar um servidor *Modbus* TCP com um cliente *Modbus* serial inserido na sua variável *store*. Dessa forma, todas as mensagens recebidas pelo servidor em TCP são imediatamente redirecionadas pelo cliente em serial utilizando os parâmetros selecionados. A figura 41 ilustra a programação desse processo.

Figura 41 – Programação do recurso de conversão *Modbus* TCP para *Modbus* Serial

```
251     def post(self):
252         # ------------------------------------------------------------------------ #
253         # argument parsing query
254         # ------------------------------------------------------------------------ #
255         query = self.reqparse.parse_args()
256 
257         client = ModbusSerialClient(method='rtu', port=query['serial_port'], baudrate=query['baudrate'],
258                                     bytesize=query['bytesize'], parity=query['parity'], stopbits=query['stopbits'])
259 
260         store = {unit_number: RemoteSlaveContext(client, unit=unit_number)
261                     for unit_number in range(1, 248)}
262         context = ModbusServerContext(slaves=store, single=False)
263 
264         # ------------------------------------------------------------------------ #
265         # start redirecting process
266         # ------------------------------------------------------------------------ #
267         StartTcpServer(context, address=(query['ip'], query['port']))
```

Fonte: Do Autor (2022).

4.4.7.2 ModbusTcpForwarder

Esse recurso funciona de forma extremamente semelhante ao da seção 4.4.7.1, contudo, dessa vez os parâmetros recebidos através da função *RequestParser()* realizam seus papéis de forma invertida, com a porta serial, *baudrate*, *byte size*, paridade e *stop bits* sendo utilizadas para configurar o *gateway* e o IP e porta TCP sendo usados para encontrar o servidor desejado. Por limitações do protocolo, esse modo suporta apenas um servidor por vez.

De forma análoga, nesse caso é instanciado um servidor *Modbus* Serial com um cliente *Modbus* TCP inserido em sua variável *store*, de modo que as mensagens recebidas pelo servidor serial são imediatamente redirecionadas pelo cliente TCP utilizando os parâmetros selecionados. A figura 42 ilustra a programação desse processo.

Figura 42 – Programação do recurso de conversão *Modbus* Serial para *Modbus* TCP

```
207     def post(self):
208         # ------------------------------------------------------------------------ #
209         # argument parsing query
210         # ------------------------------------------------------------------------ #
211         query = self.reqparse.parse_args()
212 
213         client = ModbusTcpClient(port=query['ip'], baudrate=query['port'])
214 
215         store = {unit_number: RemoteSlaveContext(client, unit=unit_number)
216                     for unit_number in range(1, 248)}
217         context = ModbusServerContext(slaves=store, single=False)
218 
219         # ------------------------------------------------------------------------ #
220         # start redirecting process
221         # ------------------------------------------------------------------------ #
222         StartSerialServer(context, port=query['serial_port'], framer=ModbusRtuFramer, baudrate=query['baudrate'],
223                         bytesize=query['bytesize'], parity=query['parity'], stopbits=query['stopbits'])
```

Fonte: Do Autor (2022).

4.4.7.3 ModbusTcpServer

Esse recurso é o mais simples dentre os 6, pois funciona em conjunto com o recurso *OPCUAServer*. Nesse caso, também há um *RequestParser()*, porém, é necessário apenas o IP e a porta utilizadas para que o *gateway* seja descobertos por clientes *Modbus* TCP.

Dessa forma, é instanciado um servidor *Modbus* TCP simples que será requerido posteriormente pelo recurso *OPCUAServer* para associar as *tags* OPC UA aos registradores *Modbus* desejados.

Na figura 43, pode-se observar a instanciação de um servidor *Modbus* TCP com 20000 registradores de cada tipo.

Figura 43 – Programação do recurso de servidor *Modbus* TCP

```
285     def post(self):
286         # -------------------------------------------------------------------------- #
287         # argument parsing query
288         # -------------------------------------------------------------------------- #
289         query = self.reqparse.parse_args()
290 
291         initval = 0
292         max_regs = 20000
293 
294         store = ModbusSlaveContext(
295             di=ModbusSequentialDataBlock(0, [initval]*max_regs),
296             co=ModbusSequentialDataBlock(0, [initval]*max_regs),
297             hr=ModbusSequentialDataBlock(0, [initval]*max_regs),
298             ir=ModbusSequentialDataBlock(0, [initval]*max_regs)
299         )
300         context = ModbusServerContext(slaves=store, single=True)
```

Fonte: Do Autor (2022).

4.4.7.4 OPCUAServer

Para este recurso, primeiramente é declarada uma função para autenticação de usuários, conforme o banco de dados de usuários criado conforme a seção 4.4.1. Por questões de tempo, não foi criada uma página de cadastro de usuários, necessitando que os usuários sejam inseridos manualmente no banco de dados. O controle de usuários do recurso pode ser observado na figura 44.

Figura 44 – Controle de autenticação de usuários OPC UA

```
393     def user_manager(isession, username, password):
394         isession.user = UserManager.User
395         users = UserDB.query.all()
396         for user in users:
397             if username == user.name and password == user.password:
398                 return True
399         return False
```

Fonte: Do Autor (2022).

Dessa forma, se o usuário inserido durante o início da sessão OPC UA com o *gateway* não estiver habilitado, a sessão não é iniciada.

A configuração do servidor, observada na figura 45, inicia-se declarando o seu *endpoint* através da função *set_endpoint()*, que para esse caso foi selecionado *opc.tcp://127.0.0.1:4880*. Após isso, é selecionada sua URI (*Gateway*) e declarado seu

namespace pela função *register_namespace()*. Em seguida, é definida sua política de segurança através da função *set_security_IDs*, que, nesse caso, contempla apenas a autenticação de usuários. Por fim, é criada a variável *objects* para cadastro de objetos OPC UA.

O trecho seguinte corresponde a um pequeno loop desenvolvido para realizar uma *query* no banco de dados de *tags* disponível e, então, adicionar tanto seus objetos quanto as *tags* em si, através das funções *add_object* e *add_variable*.

Figura 45 – Programação do recurso de servidor OPC UA

```
511     def post(self):
512         server = Server()
513         server.set_endpoint("opc.tcp://127.0.0.1:4880")
514         uri = "Gateway"
515         idx = server.register_namespace(uri)
516 
517         policyIDs = ["Username"]
518         server.set_security_IDs(policyIDs)
519         server.user_manager.set_user_manager(OPCUAServer.user_manager)
520 
521         objects = server.get_objects_node()
522 
523         tags = TCP_to_OPC_Tags.query.all()
524         distinct_objs = TCP_to_OPC_Tags.query.with_entities(TCP_to_OPC_Tags.opc_object).distinct()
525         index = 0
526         obj_index = 0
527         n_tags = len(tags)
528         n_objects = distinct_objs.count()
529         tag_name = [None] * n_tags
530         object_name = [None] * n_objects
531         opc_objects = [None] * n_objects
532         # populating our address space
533         for tag in distinct_objs:
534             object_name[obj_index] = tag.opc_object
535             opc_objects[obj_index] = objects.add_object(idx, str(tag.opc_object))
536             obj_index += 1
537         obj_index = 0
```

Fonte: Do Autor (2022).

Com todos os objetos e *tags* cadastrados, a função *start()* é invocada para iniciar o servidor OPC UA. Além disso, é chamado o recurso *OPCUAClient* através de uma *thread*, pois não há como alterar os valores das *tags* do servidor de forma direta, necessitando de um cliente OPC UA. Esse processo pode ser visto na figura 46.

Figura 46 – Instanciação do servidor OPC UA e chamado do cliente OPC UA

```
551         server.start()
552         head = {'content-type': 'application/json'}
553         startOPCUA = threading.Thread(target=Starters.opcClientStart, args=(head,))
554         startOPCUA.start()
555         try:
556             while True:
557                 pass
558         finally:
559             #close connection, remove subcsriptions, etc
560             server.stop()
```

Fonte: Do Autor (2022).

Todas as funções utilizadas para configurar o servidor OPC UA são disponibilizadas pela biblioteca *FreeOPCUA*.

4.4.7.5 OPCUAClient

Para esse recurso, como observa-se na figura 47, é declarado um cliente OPC UA conectado diretamente ao *endpoint* configurado no servidor OPC UA do recurso descrito na seção 4.4.7.4 e, então, iniciado através da função *connect()*.

O trecho seguinte corresponde a um pequeno loop desenvolvido para preparar o redirecionamento das *tags* e objetos utilizados no servidor, juntando tanto as *tags* quanto os objetos em *arrays*.

Figura 47 – Programação do recurso de cliente OPC UA

```
312     def post(self):
313         opc_client = Client("opc.tcp://127.0.0.1:4880")
314         try:
315             # connecting!
316             opc_client.connect()
317             # Client has a few methods to get proxy to UA nodes that should always be in address space
318             root = opc_client.get_root_node()
319             tags = TCP_to_OPC_Tags.query.all()
320             distinct_objs = TCP_to_OPC_Tags.query.with_entities(TCP_to_OPC_Tags.opc_object).distinct()
321             index = 0
322             n_tags = len(tags)
323             n_objects = distinct_objs.count()
324             tag_name = [None] * n_tags
325             tag_object = [None] * n_tags
326             myData = [None] * n_tags
327             obj = [None] * n_tags
328             tag_register = [None] * n_tags
329             check = [0] * n_tags
330             data = [None] * n_tags
331             for tag in tags:
332                 tag_name[index] = "2:" + str(tag.name)
333                 tag_object[index] = "2:" + str(tag.opc_object)
334                 myData[index] = root.get_child(["0:Objects", tag_object[index], tag_name[index]])
335                 obj[index] = root.get_child(["0:Objects", tag_object[index]])
336                 tag_register[index] = int(tag.register)
337                 index += 1
```

Fonte: Do Autor (2022).

Em seguida, é declarado um cliente *Modbus* TCP através, novamente, da biblioteca *pymodbus*. Por fim, é iniciado um loop contendo as funções de leitura e escrita, de modo que uma não interfira na outra. A função de escrita (Figura 48) lê os registradores do servidor *Modbus* TCP disponibilizado pelo *gateway* (onde o cliente irá escrever seus dados) e redireciona para as *tags* OPC UA correspondentes, enquanto a função de leitura (Figura 49) redireciona os dados das *tags* OPC UA para seus respectivos registradores no servidor *Modbus* TCP, onde o cliente poderá lê-los. Para que não haja interferência entre as funções, uma variável *check* é criada para que se identifique qual função foi utilizada.

Figura 48 – Escrita de mensagens *Modbus* TCP para OPC UA

```
mbus_client = ModbusTcpClient("127.0.0.1", 502)
mbus_client.connect()
while True:
    index = 0
    register = None
    for tag in tags:
        # write function
        if tag.var_type == "int":
            count = 1
            if tag.opc_object == "HR":
                data[index] = mbus_client.read_holding_registers(tag_register[index], count, unit=1)
            elif tag.opc_object == "IR":
                data[index] = mbus_client.read_input_registers(tag_register[index], count, unit=1)
            register = data[index].registers[0]
        elif tag.var_type == "bit":
            count = 1
            if tag.opc_object == "CO":
                data[index] = mbus_client.read_coils(tag_register[index], count, unit=1)
            elif tag.opc_object == "DI":
                data[index] = mbus_client.read_discrete_inputs(tag_register[index], count, unit=1)
            register = data[index].bits[0]
        elif tag.var_type == "float":
            count = 2
            if tag.opc_object == "HR":
                data[index] = mbus_client.read_holding_registers(tag_register[index], count, unit=1)
            elif tag.opc_object == "IR":
                data[index] = mbus_client.read_input_registers(tag_register[index], count, unit=1)
            decoder = BinaryPayloadDecoder.fromRegisters(data[index].registers, Endian.Big, wordorder=Endian.Little)
            register = decoder.decode 32bit float()
```

Fonte: Do Autor (2022).

Figura 49 – Leitura de mensagens *Modbus* TCP para OPC UA

```
        # read function
        if myData[index].get_value() != check[index]:
            if tag.var_type == "int":
                if tag.opc_object == "HR":
                    mbus_client.write_registers(int(tag.register), int(myData[index].get_value()), unit=1)
            elif tag.var_type == "bit":
                if tag.opc_object == "CO":
                    mbus_client.write_coil(int(tag.register), int(myData[index].get_value()), unit=1)
            elif tag.var_type == "float":
                builder = BinaryPayloadBuilder(byteorder=Endian.Big, wordorder=Endian.Little)
                builder.add_32bit_float(float(myData[index].get_value()))
                payload = builder.to_registers()
                if tag.opc_object == "HR":
                    mbus_client.write_registers(int(tag.register), payload)
        elif register != check[index]:
            myData[index].set_value(register)
        check[index] = register
```

Fonte: Do Autor (2022).

Todas as funções utilizadas para configurar o servidor OPC UA são disponibilizadas pela biblioteca *FreeOPCUA*.

4.4.7.6 TcpClientOPCServer

Esse recurso funciona de forma semelhante ao *OPCUAServer*, possuindo autenticação de usuários, *endpoint*, URI, *namespace* e política de segurança exatamente da mesma maneira. A diferença, para esse caso, é que ao invés de necessitar declarar um cliente OPC UA, é iniciado um loop dentro do próprio servidor, o qual realiza as funções de leitura (Figura 50) e escrita (Figura 51) de maneira inversa ao método anterior, afinal o modo de operação é exatamente o oposto. Nesse caso, a variável *check* também é utilizada para que uma função não interfira com a outra.

Figura 50 – Leitura de mensagens OPC UA para *Modbus* TCP

```
446             while True:
447                 index = 0
448                 register = None
449                 for tag in tags:
450                     myData = tag_name[index].get_value()
451                     mbus_client = ModbusTcpClient(tag.slave_ip, 502)
452                     mbus_client.connect()
453                     # read function
454                     if tag.var_type == "int":
455                         count = 1
456                         if tag.opc_object == "HR":
457                             data[index] = mbus_client.read_holding_registers(int(tag.register), count, unit=1)
458                         elif tag.opc_object == "IR":
459                             data[index] = mbus_client.read_input_registers(int(tag.register), count, unit=1)
460                         register = data[index].registers[0]
461                     elif tag.var_type == "bit":
462                         count = 1
463                         if tag.opc_object == "CO":
464                             data[index] = mbus_client.read_coils(int(tag.register), count, unit=1)
465                         elif tag.opc_object == "DI":
466                             data[index] = mbus_client.read_discrete_inputs(int(tag.register), count, unit=1)
467                         register = data[index].bits[0]
468                     elif tag.var_type == "float":
469                         count = 2
470                         if tag.opc_object == "HR":
471                             data[index] = mbus_client.read_holding_registers(int(tag.register), count, unit=1)
472                         elif tag.opc_object == "IR":
473                             data[index] = mbus_client.read_holding_registers(int(tag.register), count, unit=1)
474                         decoder = BinaryPayloadDecoder.fromRegisters(data[index].registers, Endian.Big, wordorder=Endian.Little)
475                         register = decoder.decode_32bit_float()
476 
477                     # check variable
478                     if register != check[index]:
479                         tag_name[index].set_value(register)
```

Fonte: Do Autor (2022).

Figura 51 – Escrita de mensagens OPC UA para *Modbus* TCP

```
481                 # write function
482                 elif myData != check[index]:
483                     if tag.var_type == "int":
484                         if tag.opc_object == "HR":
485                             mbus_client.write_registers(int(tag.register), int(myData), unit=int(tag.unit_id))
486                     elif tag.var_type == "bit":
487                         if tag.opc_object == "CO":
488                             mbus_client.write_coil(int(tag.register), bool(myData), unit=int(tag.unit_id))
489                     elif tag.var_type == "float":
490                         builder = BinaryPayloadBuilder(byteorder=Endian.Big, wordorder=Endian.Little)
491                         builder.add_32bit_float(float(myData))
492                         payload = builder.to_registers()
493                         if tag.opc_object == "HR":
494                             mbus_client.write_registers(int(tag.register), payload, unit=int(tag.unit_id))
495                 check[index] = register
496                 mbus_client.close()
497                 #index += 1
```

Fonte: Do Autor (2022).

4.5 Resultados

A conversão da informação entre protocolos foi realizada com sucesso em todos os quatro modos desenvolvidos. Entretanto, não foi possível que todos os modos operassem simultaneamente devido à falta de portas *Ethernet* disponíveis. Além disso, como a aplicação foi executada em um computador convencional, foi necessário utilizar um conversor de USB para RS-485, uma vez que o equipamento não possui meio físico RS-485 nativo. Também deve-se ressaltar que toda alteração realizada nos arquivos de configuração JSON ou nos bancos de dados somente surtem efeito após a reinicialização do sistema, a qual pode ser realizada pelos botões *Reset* presentes em quase todas as páginas da aplicação e leva de 6 a 8 segundos para ser efetuada.

4.5.1 Conversão de *Modbus* TCP para *Modubs* Serial

Para os testes realizados nesse modo de operação, foi utilizado o software *Modbus Doctor*, da *Lamesoft*, como cliente TCP e o CLP TPW-04, da WEG, como servidor serial.

Primeiramente, foi realizado um teste simples utilizando somente uma instância do cliente e um CLP como servidor. Foram enviadas mensagens de leitura e escrita, tanto para registradores de *coil* quanto de *holding register*, *input register* e *discrete input*. A aplicação funcionou de forma eficaz em todas as *baudrates* disponíveis pelo CLP, tanto em comandos de leitura quanto em comandos de escrita.

Por fim, os testes foram repetidos com até 5 instâncias de cliente *Modbus* TCP, onde a conexão se manteve estável para todos os clientes.

4.5.2 Conversão de *Modbus* Serial para *Modubs* TCP

Para os testes realizados nesse modo de operação, foi utilizado o software *Modbus Doctor*, da *Lamesoft*, como cliente Serial e o CLP TPW-04, da WEG, como servidor TCP.

Primeiramente, foi realizado um teste simples utilizando somente uma instância do cliente e um CLP como servidor, utilizando IP 192.168.0.1. Foram enviadas mensagens de leitura e escrita, tanto para registradores de *coil* quanto de *holding register*, *input register* e *discrete input*. A aplicação funcionou de forma eficaz em todos os casos.

Por limitações do protocolo, esse modo só permite um cliente conectado por vez.

4.5.3 Conversão de *Modbus* TCP para OPC UA

Para o caso de conversões de *Modbus* TCP para OPC UA, foram utilizados somente softwares devido a falta de equipamentos compatíveis com o OPC UA. O software *Modbus Doctor*, da *Lamesoft*, foi novamente utilizado como cliente TCP enquanto o software UaExpert, da *Unified Automation*, foi utilizado como cliente OPC UA.

Primeiramente, foram cadastradas 4 tags com 4 objetos distintos, um para cada tipo de registrador *Modbus* (*coil*, *discrete input*, *holding register* e *input register*), sendo duas delas bit, uma int e uma float, como demonstra a figura 52.

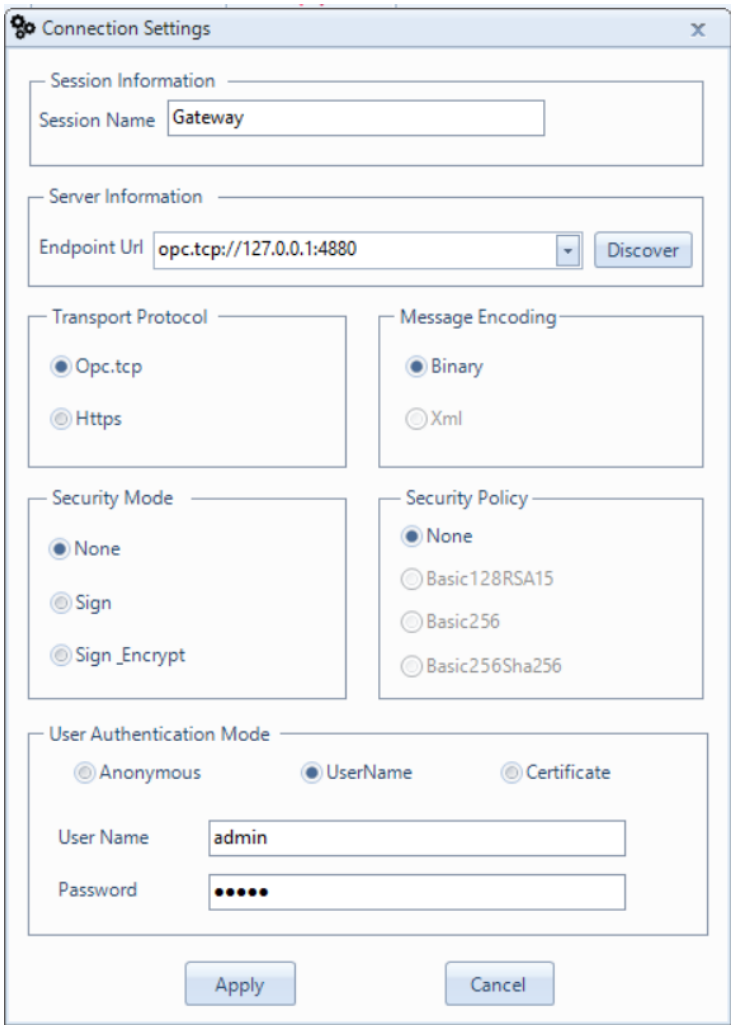
Figura 52 – *Tags* utilizadas na conversão de *Modbus* TCP para OPC UA

ID	Name	Register	Var Type	OPC Object
1	CoilTag	10	bit	CO
2	DiscreteTag	20	bit	DI
3	HoldingRegisterTag	30	float	HR
4	InputRegisterTag	40	int	IR

Fonte: Do Autor (2022).

Pelo lado do *UaExpert*, o servidor foi cadastrado como na figura 53, sendo que único usuário cadastrado na autenticação está com *login admin* e senha *admin*.

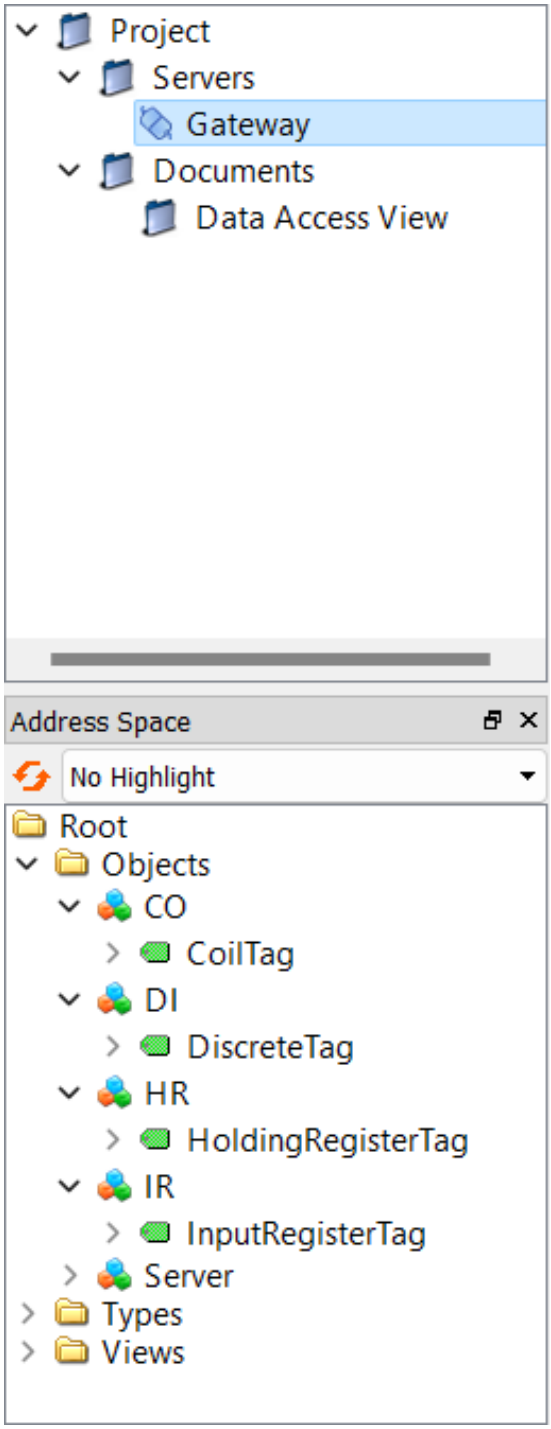
Figura 53 – Cadastro do servidor no *UaExpert*



Fonte: Do Autor (2022).

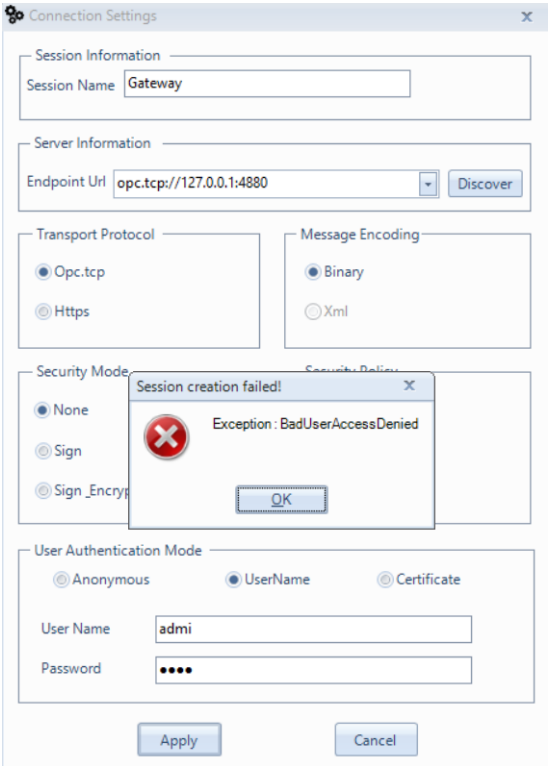
Como pode-se observar na figura 54, a conexão foi estabelecida com sucesso. Se o usuário e senha estivessem incorretos, a situação se encontraria como na figura 55.

Figura 54 – Conexão com o servidor OPC UA realizada com sucesso



Fonte: Do Autor (2022).

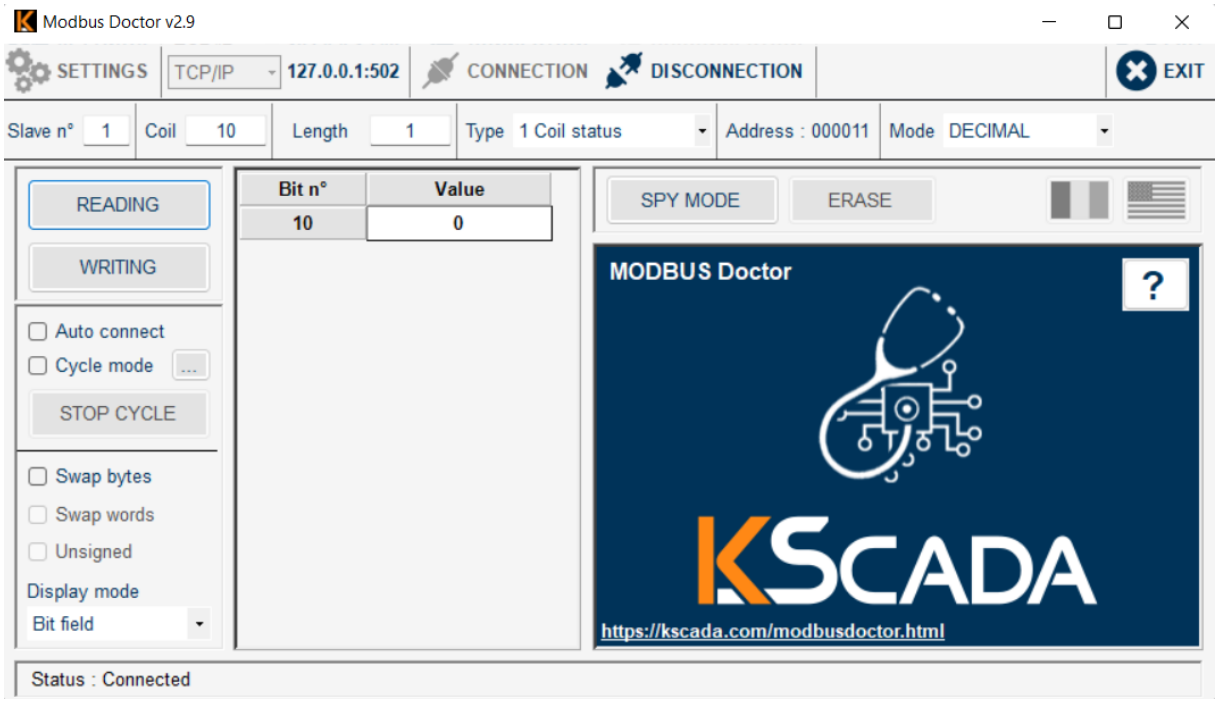
Figura 55 – Conexão com o servidor OPC UA mal sucedida



Fonte: Do Autor (2022).

Em seguida, foi estabelecida a conexão do cliente *Modbus* com o *gateway*, como na figura 56.

Figura 56 – Conexão com o servidor *Modbus* TCP do *gateway*



Fonte: Do Autor (2022).

Ademais, a *coil* foi alterada para 1 e o *holding register* para 44,21, enquanto os registradores de *discrete input* e *input register* são apenas para leitura, portanto permanecerão em 0. A figura 57 denota essa interação pelo *Modbus Doctor*.

Figura 57 – Alteração de registradores *Modbus*

Bit n°	Value	Bit n°	Value
10	1	20	0
Register n°	**Value**	**Register n°**	**Value**
30	44,21	40	0

Fonte: Do Autor (2022).

Por fim, os valores são imediatamente alterados no servidor OPC UA, que pode ser conferido através do UaExpert, como na figura 58.

Figura 58 – Verificação das alterações no servidor OPC UA

DisplayName	CoilTag	DisplayName	DiscreteTag
Description	CoilTag	Description	DiscreteTag
WriteMask	0	WriteMask	0
UserWriteMask	0	UserWriteMask	0
Value	True	Value	False
DisplayName	HoldingRegisterT...	DisplayName	InputRegisterTag
Description	HoldingRegisterT...	Description	InputRegisterTag
WriteMask	0	WriteMask	0
UserWriteMask	0	UserWriteMask	0
Value	44.2099990844727	Value	0

Fonte: Do Autor (2022).

Nota-se que somente a variável float não ficou exatamente igual, porém isso se deve a um erro de interpretação e não de conversão.

Fazendo o caminho contrário, a aplicação funcionou corretamente ao variar as *tags* OPC UA através do *UaExpert* e lê-las pelos seus registradores associados através do *Modbus Doctor*.

Foram testadas até 5 instâncias de cliente *Modbus* TCP conectadas e enviando requisições simultaneamente e também 2 instâncias de cliente OPC UA. A conexão permaneceu estável e eficaz para todos os clientes.

4.5.4 Conversão de OPC UA para *Modbus* TCP

Para esse modo de operação, foi utilizado um servidor *Modbus* TCP em software disponibilizado pela própria biblioteca *pymodbus* em sua documentação, e o software *UaExpert*, da *Unified Automation*, como cliente OPC UA.

Primeiramente, foram cadastradas 4 *tags* com 4 objetos distintos associados ao IP do CLP (192.168.0.1), um para cada tipo de registrador *Modbus* (*coil*, *discrete input*, *holding register* e *input register*), sendo duas delas bit e outras duas int, como demonstra a figura 59. Por conta de um bug não detectado, a conversão de variáveis float nesse modo não funcionou corretamente.

Figura 59 – *Tags* utilizadas na conversão de OPC UA para *Modbus* TCP

ID	Name	Register	Var Type	OPC Object	Slave IP	Unit ID
2	InputRegisterTag	10	int	IR	192.168.0.1	1
3	CoilTag	6	bit	CO	192.168.0.1	1
4	DiscreteTag	54	bit	DI	192.168.0.1	1
5	HoldingRegisterTag	10	int	HR	192.168.0.1	1

Fonte: Do Autor (2022).

Pelo lado do *UaExpert*, o servidor foi cadastrado e a conexão foi estabelecida seguindo os mesmos passos da seção 4.5.3.

Ademais, a *coil* foi alterada para 1 e o *holding register* para 98. Por se tratar de um servidor em *software* programável, foi possível forçar um valor de *input register*, nesse caso, 17, enquanto o registrador de *discrete input* permaneceu em 0. A figura 60 denota essa interação pelo *UaExpert*.

Figura 60 – Alterações no servidor OPC UA

DisplayName	"", "CoilTag"
Description	"", "CoilTag"
Value	
SourceTimestamp	31/12/1600 21:00:00.000
SourcePicoseconds	0
ServerTimestamp	31/12/1600 21:00:00.000
ServerPicoseconds	0
StatusCode	Good (0x00000000)
Value	1

DisplayName	"", "DiscreteTag"
Description	"", "DiscreteTag"
Value	
SourceTimestamp	12/12/2022 20:11:50.539
SourcePicoseconds	0
ServerTimestamp	31/12/1600 21:00:00.000
ServerPicoseconds	0
StatusCode	Good (0x00000000)
Value	0

DisplayName	"", "HoldingRegisterTag"
Description	"", "HoldingRegisterTag"
Value	
SourceTimestamp	31/12/1600 21:00:00.000
SourcePicoseconds	0
ServerTimestamp	31/12/1600 21:00:00.000
ServerPicoseconds	0
StatusCode	Good (0x00000000)
Value	98

DisplayName	"", "InputRegisterTag"
Description	"", "InputRegisterTag"
Value	
SourceTimestamp	12/12/2022 20:31:47.255
SourcePicoseconds	0
ServerTimestamp	31/12/1600 21:00:00.000
ServerPicoseconds	0
StatusCode	Good (0x00000000)
Value	17

Fonte: Do Autor (2022).

Por fim, os valores alterados no CLP podem ser visualizados conectando-se a ele através do *Modbus Doctor*, como denota a figura 61.

Figura 61 – Alterações visualizadas no servidor *Modbus* TCP

Bit n°	Value	Bit n°	Value
6	1	54	0
Register n°	**Value**	**Register n°**	**Value**
10	98	10	17

Fonte: Do Autor (2022).

Com exceção da variável float, todos os dados foram convertidos com sucesso.

Fazendo o caminho contrário, a aplicação funcionou corretamente ao variar os valores dos registradores no servidor *Modbus* TCP, onde o *UaExpert* as leu corretamente.

Finalmente, ainda foram realizados testes com 3 servidores diferentes, os quais possuiam, obviamente, IPs diferentes. A comunicação funcionou perfeitamente em todos os servidores.

5 CONSIDERAÇÕES FINAIS

Conclui-se, portanto, que o protocolo *Modbus*, apesar de ainda ser extremamente utilizado na indústria, se torna cada vez mais limitado, principalmente no que se refere a questões de segurança, uma vez que não possui nenhum método nativo para garantia de autenticação, criptografia, não-repúdio e confidencialidade. Além disso, fica claro que limitações como tamanho de pacote de dados, velocidade de transmissão, número de dispositivos conectados simultaneamente e dificuldade na integração entre sistemas legado e atuais tende a tornar o *Modbus* uma opção cada vez menos atrativa para o mercado, apesar de sua simplicidade.

Além disso, é notável que o uso de *middlewares* na indústria já é uma opção não apenas atrativa como também muito utilizada, uma vez que facilitam de forma considerável a integração de sistemas com diferentes camadas e protocolos além de apresentarem soluções para segurança, integridade e confidencialidade da informação.

Entretanto, apesar de ter sido bastante abordada durante a pesquisa bibliográfica, as questões de segurança foram pouco exploradas durante o desenvolvimento do protótipo.

5.1 Sugestões para Trabalhos Futuros

No que se refere a aplicação desenvolvida, um *gateway* capaz de converter mensagens entre *Modbus* TCP e OPC UA, fica como sugestão a implementação de criptografia e autenticação por certificado, além de uma tela protegida para cadastro de usuários. Também pode-se citar a implementação de um sistema embarcado carregando a aplicação, de forma a tornar o protótipo ainda mais dinâmico.

REFERÊNCIAS

ALTEXSOFT. *REST API: Key Concepts, Best Practices, and Benefits*. 2021. Disponível em: https://www.altexsoft.com/blog/rest-api-design/. 33, 34

ALTUS. *O que é um gateway e por que devo utilizá-lo em minhas aplicações IoT?* 2017. Disponível em: https://www.altus.com.br/post/386/o-que-e-um-gateway-e-por-que-devo-utiliza-lo-em-minhas-aplicacoes-iot. 31

AMIRI, H. *Investigation on Modbus protocol to develop an interoperable IIoT platform using Node-RED with MQTT gateway*. Tese (*Master of Telecommunication Engineering*) — *Scuola di Ingegneria Industriale e dell'Informazione*, Itália, 2021. 21

BALADOR, A.; ERICSSON, N.; BAKHSHI, Z. Communication middleware technologies for industrial distributed control systems: A literature review. **IEEE**, 2017. 30

CHOMKO, R. What's an api and why is it important? ZDNET, 2010. Disponível em: https://www.zdnet.com/article/whats-an-api-and-why-is-it-important/. 32

CORDEIRO, J. A. F. *Estudo dos Principais Protocolos de Redes Iindustriais Utilizadas no Brasil: AS-I, Modbus e Profibus*. 2019. Monografia (Graduação em Engenharia de Controle de Automação), Universidade Federal de Ouro Preto, Ouro Preto. 13

DUTERTRE, B. Formal modeling and analysis of the modbus protocol. **IFIP**, 2007. 13

FERSI, G. Middleware for internet of things: a study. **International Conference on Distributed Computing in Sensor Systems**, 2015. 30

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (*PhD in Information and Computer Science*) — *University of California*, Estados Unidos, 2000. 32

FOVINO, I. N. *et al.* Design and implementation of a secure modbus protocol. **IFIP**, 2009. 20

IBM. Application programming interface (api). **IBM Cloud Education**, 2020. Disponível em: https://www.ibm.com/cloud/learn/api. 31

LEONARDO, C.; JOHNSON, D. Modbus covert channel. **Rochester Institute of Technology,**, 2014. 13

LUSWATA, J.; ZAVARSKY, P.; SWAR, B. Analysis of scada security using penetration testing: A case study on modbus tcp protocol. **Concordia University of Edmonton**, 2018. 21

MARTINS, T.; OLIVEIRA, S. V. G. Enhanced modbus/tcp security protocol: Authentication and authorization functions supported. **Sensors**, 2022. 20

MODICON. *MODBUS Appication Protocol Specification*. [S.l.], 2012. Disponível em: https://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf. 13, 15, 16, 17, 18, 19

OPC FOUNDATION. *OPC Unified Architecture Specification*. [S.l.], 2022. Disponível em: https://opcfoundation.org/. 23, 24, 25, 27, 28, 29

PARIAN, C.; GULDIMANN, T.; BHATIA, S. Fooling the master: Exploiting weaknesses in the modbus protocol. **Procedia Computer Science**, 2020. 14, 21

RAZZAQUE, M. A. *et al.* Middleware for internet of things: a survey. **IEEE**, 2015. 30

REDHAT. Middleware. **Redhat**, 2018. Disponível em: https://www.redhat.com/pt-br/topics/middleware/what-is-middleware. 30

RINALDI, J. What modbus doesn't do very well. **RTA Automation**, 2021. Disponível em: https://www.rtautomation.com/rtas-blog/what-modbus-doesnt-do-very-well/. 22

SILVA, F. C. de S. Modelos cliente-servidor do modbus tcp e publish-subscribe do mqtt: quando utilizar cada um deles e quais suas vantagens e desvantagens? **Novus**, 2019. 14, 22

SOFTWAREONE. Middleware: O que É e quais são as vantagens de usar? **Softwareone**, 2021. Disponível em: https://www.softwareone.com/pt-br/blog/artigos/2020/03/02/middleware-o-que-e-e-quais-sao-as-vantagens-de-usar. 30

SQLITE. What is sqlite? **SQLite**, 2022. 46

TECNOLOGIA, H. *Especificações do GTON-C*. 2022. Disponível em: https://materiais.hitecnologia.com.br/produtos/equipamentos/conversores-e-gateways/conversor-serial-ethernet-modbus-gton-c/. 31

TUNKKARI, J. *Mapping Modbus to OPC Unified Architecture*. Tese (*Master of Science in Technology*) — *Aalto University School of Electrical Engineering*, Finlândia, 2018. 13, 14, 30

URREA, C.; MORALES, C.; KERN, J. Implementation of error detection and correction in the modbus-rtu serial protocol. **International Journal of Critical Infrastructure Protection**, 2016. 21

ÅGREN, D. *Improving system integration by standardizing and automating the Modbus protocol*. Tese (*Bachelor's degree in System Sciences*) — *Luleå University of Technology*, Suécia, 2020. 13, 14, 22, 23