



UNIVERSIDADE FEDERAL DE MINAS GERAIS

ESCOLA DE ENGENHARIA

SISTEMAS DISTRIBUÍDOS PARA AUTOMAÇÃO

ELT011

Trabalho Prático - OPC e Sockets TCP/IP

Autores:

Vítor Gabriel Reis Caitité

João Pedro Antunes Ferreira

Email:

vcaitite@ufmg.br

jpedroantunes@ufmg.br

11 de outubro de 2020

Sumário

1	Introdução	3
2	Arquitetura do projeto	4
3	Ferramentas e recursos utilizados	6
4	Servidor de <i>Sockets</i> TCP/IP	7
4.1	Requisitos de Projeto	7
4.2	Processo de desenvolvimento	7
5	Cliente OPC	10
5.1	Requisitos de projeto	10
5.2	Processo de desenvolvimento	10
5.2.1	Variáveis e estruturas globais importantes	10
5.2.2	O script principal: <i>main.cpp</i>	11
5.2.3	O script cliente OPC: <i>opcclient.cpp</i>	11
5.2.4	A classe de leitura assíncrona: <i>SOCDataCallback.cpp</i>	13
6	Resultados	14
7	Instruções de Compilação e Execução da Aplicação	20
8	Conclusão	21
9	Rererências	22

Lista de Figuras

1	Arquitetura do projeto.	4
2	Ilustração de um servidor e cliente de <i>sockets</i> no <i>Windows</i> . Fonte: [3]	8
3	Exibição do progresso de inicialização do servidor de <i>sockets</i>	14
4	Exibição do progresso de inicialização do cliente OPC.	15
5	Dados de posição do vagão lidos de forma assíncrona pelo cliente OPC.	15
6	Servidor de <i>sockets</i> - Recebimento mensagem de requisição de dados e a resposta com os dados de posição do vagão.	16
7	Sistema de otimização - Envio de mensagem de requisição de dados e recebimento da resposta com os dados de posição do vagão.	17
8	Servidor de <i>sockets</i> - Recebimento mensagem contendo os parâmetros de carregamento e a resposta de confirmação.	17

9	Sistema de otimização - Envio de mensagem contendo os parâmetros de carregamento e a resposta de confirmação recebida.	17
10	Cliente OPC - Escrita de dados de carregamento no servidor OPC.	18
11	Trecho de funcionamento da aplicação contendo o servidor de <i>sockets</i> e o cliente OPC.	19
12	Sistema de otimização- Trecho de funcionamento da do sistema de otimização.	20

1 Introdução

Esse trabalho visou colocar em prática boa parte do conteúdo aprendido durante as aulas referentes a *Sockets TCP/IP*, *Component Object Model* e OPC.

Atualmente no Brasil o transporte de minério utilizando ferrovias é fundamental. Como exemplo disso podemos citar a Estrada de Ferro Vitória Minas (EFVM), uma das mais produtivas ferrovias brasileiras, operada pela Vale. Com 905 km de extensão, é responsável pelo transporte de cerca de 37% de toda a carga ferroviária nacional. Desse total, 80% é dedicado ao minério de ferro [1] [2].

Normalmente o fluxo de minério de ferro nas ferrovias inicia com o envio de vagões vazios aos pontos de carregamento. Para atingir os altos volumes de transporte, eficiência e rapidez no carregamento é fundamental que esse ocorra de maneira automatizada, de forma a se obter uma maior produtividade dos ativos.

Uma maneira de automatizar essa tarefa é utilizando um sistema distribuído como descrito nesse trabalho, que visa integrar sensores, um sistema de otimização, um software que é responsável por levar os dados dos sensores até o sistema de otimização e levar informações desse para um Controlador Lógico Programável, e obviamente esse CLP para comandar o acionamento da comporta dos silos de carregamento e também a velocidade dos vagões.

O escopo desse trabalho é justamente desenvolver a aplicação capaz de integrar o CLP e o sistema de otimização. Para isso, nossa aplicação funcionará como um servidor de *sockets*, para se comunicar com o sistema de otimização (cliente de *sockets*), e como cliente OPC, para interagir com o servidor OPC clássico que tem o acesso as variáveis de processo do Controlador Lógico Programável.

2 Arquitetura do projeto

O projeto consiste em uma aplicação rodando em um ambiente *Windows*, que visa a integração de um sistema de otimização de processo externo e um servidor OPC com acesso as variáveis de processo do CLP que controla os atuadores responsáveis por realizar a tarefa de carregamento dos vagões. Abaixo, na Fig. 1, segue uma ilustração do processo de *software* executado e descrito acima:

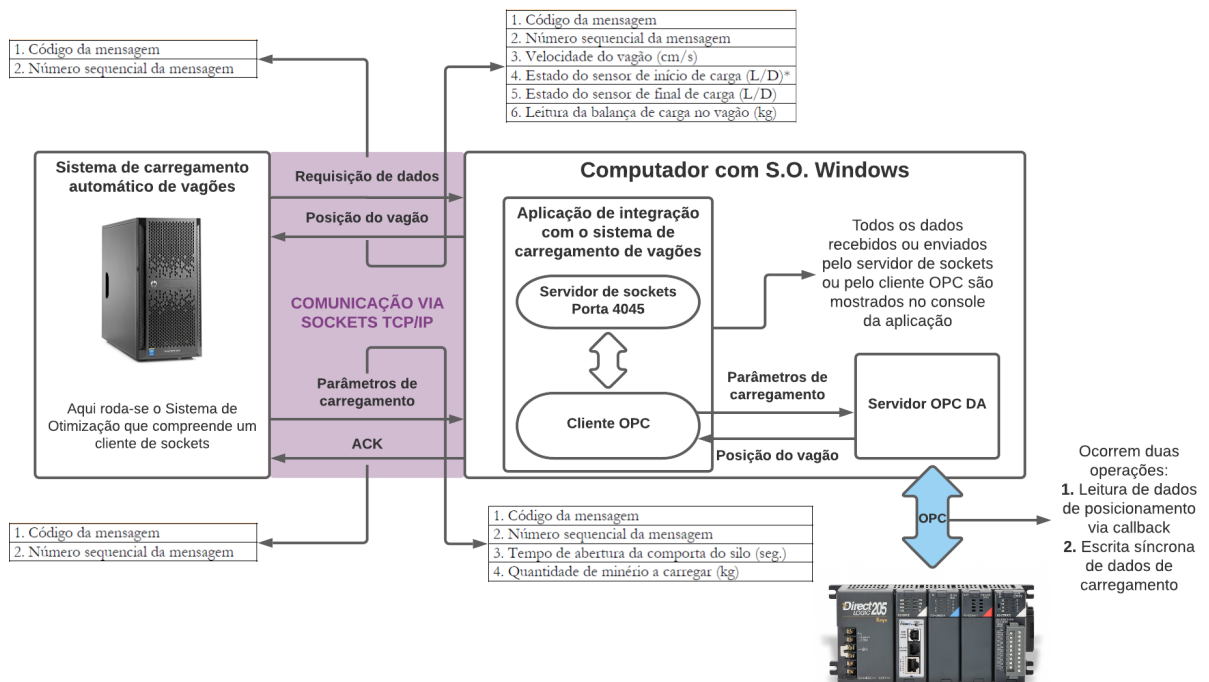


Figura 1: Arquitetura do projeto.

O sistema completo é composto pelos seguintes módulos:

- **Servidor socket:** é o processo responsável por toda a interface de comunicação entre a aplicação e o sistema de otimização externo. Tal servidor será responsável, portanto, por receber e enviar dados das variáveis de processo para o sistema de otimização;
- **Servidor OPC:** este servidor é uma aplicação de simulação utilizada para prover um servidor OPC real. Mais detalhes sobre tal servidor serão dados posteriormente quando necessário. É importante ressaltar que o servidor e o cliente OPC compartilham o mesmo sistema operacional, portanto, não há latência de rede envolvida no processo;

- **Cliente OPC:** é o sistema responsável por ler algumas variáveis de processo do servidor OPC assincronamente e torná-las disponíveis no sistema para que o servidor de *socket* possa retorná-las sempre que solicitado pelo sistema de otimização. O cliente OPC ainda é responsável por escrever, no servidor OPC, parâmetros de carregamento vindos do sistema de otimização;
- **Sistema de otimização:** sistema, que ao receber dados de variáveis do processo, é capaz de gerar os parâmetros de carregamento. Funciona como um cliente de *sockets*.

O projeto consiste em desenvolver, portanto, a aplicação contendo o **servidor de socket** e o **cliente OPC**. A arquitetura definida foi que ambos seriam tratados como processos separados e seriam então iniciados a partir do disparo de *threads* individualizadas para cada uma delas.

A comunicação entre estes serviços será feita indiretamente através de variáveis globais a serem definidas. Uma estrutura global para armazenar os valores de variáveis de processo lidos pelo cliente OPC através do servidor OPC, outra variável global para armazenar os valores das variáveis de processo a serem escritas no servidor OPC pelo cliente OPC e recebidas pelo servidor *socket* e uma terceira variável booleana global que será uma *flag* que forçará ou não o cliente OPC a realizar o processo de escrita dos dados recebidos pelo servidor *socket*. É claro, que todas estas escritas e leituras precisam ser sincronizadas para garantir a consistência do projeto e uma troca de informações segura e confiável entre aplicação e sistema de otimização.

Para essa sincronização foram utilizados mutexes. Essas ferramentas permitem implementar a técnica de exclusão mútua, que é usada em programação concorrente para evitar que dois processos ou *threads* tenham acesso simultaneamente a um recurso compartilhado, acesso esse denominado por seção crítica. Assim, utilizou-se esses mutexes para definir as seções críticas nas quais as variáveis compartilhadas por ambas as *threads* são alteradas ou lidas.

É importante ressaltar que todas as informações de leitura e escrita do cliente OPC e também das mensagens trocadas pelo servidor *socket*, serão exibidas ao usuário através de uma interface de linha de comando (*console*).

3 Ferramentas e recursos utilizados

Para o projeto serão, portanto, utilizadas as seguintes ferramentas básicas:

- **Visual Studio Community Edition:** a IDE utilizada para desenvolvimento foi o Visual Studio da Microsoft configurado com o workload *Desktop development with C++* (vale ressaltar que o projeto foi desenvolvido em C++);
- **Servidor OPC Matrikon de simulação:** foi utilizado um software da empresa Matrikon¹. Este software foi útil para simular um servidor real OPC, já que ele fornece funcionalidades muito úteis para aplicação de testes como, por exemplo, a definição de itens OPC randômicos, como é o caso desejado para esta tarefa;
- **Projeto Simple OPC Client com modificações:** foi utilizado o projeto *Simple OPC Client* já modificado pelo professor Luiz Themystoklitz S. Mendes e fornecido para a turma da disciplina de Sistemas Distribuídos para Automação referente ao semestre 1 do ano letivo de 2020. Seu uso foi tomado como base para desenvolver a lógica de leitura e escrita no servidor OPC (tanto no processo síncrono quanto no assíncrono);
- **Sistema de Otimização:** sistema criado pelo professor para fornecimento de informações a serem escritas no servidor OPC e para solicitações de leitura de variáveis do servidor.

¹Servidor de simulação disponível em: <https://www.matrikonopc.com/portal/downloads.aspx>

4 Servidor de *Sockets* TCP/IP

4.1 Requisitos de Projeto

O módulo 1 desse projeto consiste em um servidor de *sockets*, que respeita os seguintes requisitos:

1. Receber mensagens de solicitação de posicionamento dos vagões e a responder imediatamente com uma mensagem englobando os dados de posição do vagão. (Obs.: As mensagens de solicitação são periódicas, a cada dois segundos).
2. Receber mensagens aperiódicas com os parâmetros do vagão e disponibilizar esses dados para o Cliente OPC poder enviá-los ao servidor de *sockets* da Matrikon. E também responder ao cliente de *sockets* com um ACK, ou seja uma confirmação.

OBS.: Todas as mensagens trocadas entre o servidor e cliente de *sockets* devem obedecer integralmente as especificações descritas na especificação desse trabalho.

4.2 Processo de desenvolvimento

Sockets é uma das maneiras mais populares de utilizar as funcionalidades de comunicação TCP/IP. A principal API de *sockets* no ambiente *Windows* é a *WinSock 2* (que foi utilizada nesse trabalho). Utilizando as funções dessa API pode-se modelar um cliente e servidor de *sockets*, como ilustrado na Fig. 2 abaixo.

Assim como ilustrada na Fig. 2, até se estabelecer uma conexão com o cliente os seguintes passos foram executados:

- Inicialização da biblioteca *Winsock 2* (*WSAStartup()*);
- O próximo passo foi criar um *socket* para conexões. Para isso bastou-se invocar a função *socket()*, passando como parâmetros a especificação da família de endereços (foi utilizada a *AF_INET*), a especificação do tipo de *socket* (utilizou-se o *SOCK_STREAM*), e também o protocolo a ser usado (no nosso caso o *IPPROTO_TCP*);
- Uma vez que o *socket* está criado, o próximo passo é associá-lo a uma porta. Para isso pode-se utilizar a função *bind()*. A porta utilizada e que teve seu endereço passado por parâmetro para a função *bind* foi a 4045 (como especificado na descrição do trabalho);
- Após o *socket* ter sido criado e uma porta associada é necessário habilitar o *socket* para receber as conexões. A função *listen()* faz exatamente isso, ou seja, habilita que

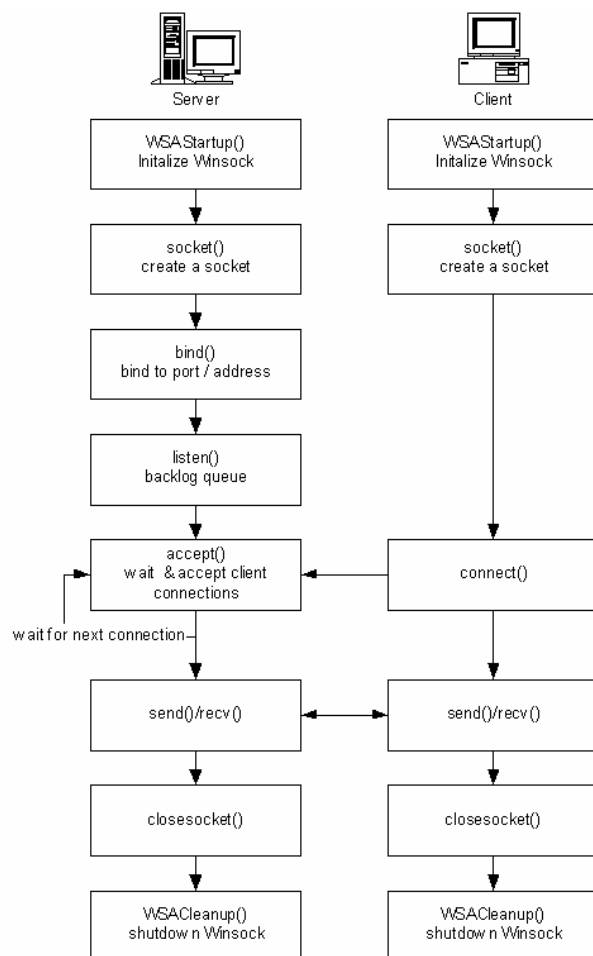


Figura 2: Ilustração de um servidor e cliente de *sockets* no *Windows*. Fonte: [3]

o servidor receba conexões de um programa cliente. Ao chamar essa função passou-se os seguintes parâmetros: o descritor do *socket* aberto e a quantidade máxima de conexões que podem ficar pendentes até que o programa trate todas as conexões anteriores;

- Por fim foi invocada a função *accept()*. Essa é utilizada para aceitar as conexões efetuadas pelos clientes. Foi passado para a função o *socket* aberto.

A partir desse ponto, já temos um servidor capaz de se conectar a um programa cliente. Todos esses passos mostrados acima, e os próximos descritos abaixo, foram executados no arquivo **socket.cpp**.

Uma vez que uma conexão foi estabelecida, entra-se num loop realizando as seguintes atividades:

- Utiliza a função *recv()* para ler uma mensagem do *socket*;
- Caso o recebimento ocorra como esperado, o primeiro passo é checar o número sequencial da mensagem, especificado na descrição do trabalho. Isso é feito invocando a função *check_sequencial_number()*. Essa função foi desenvolvida no arquivo **msg_treatment.cpp** e checa se o valor contido no campo "número sequencial" da mensagem recebida é uma unidade maior que o contador de mensagens do programa. Além disso, incrementa-se o contador de mensagens;
- O próximo passo é tratar as mensagens recebidas. Para isso invoca-se a função *socketMsgTreatment()*. Essa função foi desenvolvida no arquivo **msg_treatment.cpp** e recebe como parâmetros o mensagem e seu respectivo tamanho. Caso essa função identifique que se trata de uma mensagem de solicitação de dados, ela retorna um array com a mensagem de posicionamento preenchida - para preencher os campos dessa mensagem utiliza-se os dados presentes na estrutura global *positionParameters*. Já no caso de se tratar de uma mensagem com os parâmetros de carregamento do vagão, salva-se os dados recebidos na estrutura global *loadingParameters* e monta-se a mensagem de ACK que é retornada;
- Uma vez que se tem a mensagem a ser enviada montada, basta então chamar a função *send()* para enviá-la.

Enquanto houver uma conexão, essa thread permanecerá nesse loop recebendo e respondendo mensagens. E caso a conexão seja perdida, o servidor voltará para o estado de aguardar uma conexão.

5 Cliente OPC

5.1 Requisitos de projeto

O módulo 2 do projeto solicitado e desenvolvido consiste basicamente em criar um cliente OPC que seja responsável por duas operações principais do sistema:

1. Ler do servidor OPC variáveis randômicas (itens) a serem adicionadas pelo próprio cliente em sua inicialização, utilizando uma operação de leitura assíncrona. Será lido, portanto, as informações das variáveis referente a mensagem de posicionamento do vagão. A cada segundo se recebem as mudanças de variáveis ocorridas no servidor da Matrikon. Note que o próprio servidor já é configurado por padrão para atualizar seus dados a cada segundo.
2. Escrever utilizando os métodos de escrita síncrona do cliente OPC, valores recebidos do sistema de otimização referente aos parâmetros de carregamento do vagão.

5.2 Processo de desenvolvimento

Abaixo, serão descritos os principais arquivos criados ou modificados no projeto voltado para o funcionamento do módulo OPC da aplicação, além das variáveis e estruturas mais importantes e suas respectivas definições.

5.2.1 Variáveis e estruturas globais importantes

Abaixo, serão apresentadas as variáveis globais importantes de serem entendidas antes do entendimento dos métodos modificados e adicionados no sistema. Estas variáveis são definidas no arquivo *global_variables.h* e instanciadas no arquivo *global_variables.cpp*:

- ***bool SHOULD_WRITE***: essa variável booleana é uma *flag* global que indica ao processo do cliente OPC, se ele deve ou não atualizar no servidor OPC o valor das variáveis recebidas pelo sistema de otimização e tratadas e armazenadas pelo servidor *socket*;
- ***int ITEMS_QUANTITY***: essa é uma variável auxiliar que armazena a quantidade de itens a serem criados no servidor OPC;
- ***wchar_t array ITEM_IDS***: esse é um vetor de identificadores de itens a serem adicionados no servidor OPC (tais identificadores são baseados na especificação definida do projeto e são compatíveis com os identificadores aceitos pelo servidor OPC de simulação);

- ***position_parameters_t positionParameters***: essa estrutura contém todas as variáveis de posicionamento do vagão (*int wagonSpeed*, *int startSensorStatus*, *int endSensorStatus* e *float loadWeigh*). É esta estrutura global que será preenchida pelo cliente OPC com os valores obtidos e lidos assincronamente do servidor de simulação, garantindo que tais valores estejam sempre atualizados;
- ***loading_parameters_t loadingParameters***: essa estrutura contém todas as variáveis de carregamento do vagão (*int openTime* e *float oreQuantity*). É esta estrutura global que será usada pelo cliente OPC para ler os valores obtidos e preenchidos pelo servidor *socket* e então escrevê-los sincronamente no servidor OPC de simulação.

5.2.2 O script principal: *main.cpp*

Este script foi desenvolvido para se tornar o código principal de inicialização da aplicação (conter o código *main* do sistema).

Ele basicamente tem a responsabilidade de iniciar a aplicação assumindo o controle do console e disparando as duas rotinas principais do nosso sistema em *threads* separadas. Tais rotinas são:

- ***socketServer()***: esta rotina é responsável por trabalhar toda a comunicação *socket* realizada com o sistema de otimização, tratando todas as mensagens e requisições recebidas conforme a especificação definida do projeto;
- ***opcClient()***: esta rotina atua sendo a responsável por iniciar o grupo e os respectivos itens OPC (variáveis de controle do processo dos vagões), além de iniciar o processo de leitura assíncrona de acordo com as mudanças de variáveis detectadas pelo servidor OPC. Ela também é responsável pelo processo de escrita no servidor de acordo com a solicitação do sistema.

5.2.3 O script cliente OPC: *opcclient.cpp*

Este script contém grande parte do código do antigo *SimpleOPCClient_v3.cpp* do projeto *Simple OPC Client com modificações* e, portanto, foi alterado para se tornar o código que descreve a rotina a ser executada pelo cliente OPC.

Abaixo serão listados todos os métodos utilizados no sistema e suas respectivas responsabilidades (sejam eles métodos adicionados, modificados ou já existentes):

- ***opcClient(void)***: como já explicitado na sub-seção *O script principal SimpleOPCClient_v3.cpp*, esta rotina acaba sendo a responsável por iniciar o ambiente, o grupo e os respectivos itens OPC (variáveis de controle do processo dos vagões - tanto aquelas que precisarão ser lidas quanto aquelas que sofrerão escritas do cliente),

além de iniciar o processo de leitura assíncrona (usando a interface *IODataCallback*) de acordo com as mudanças de variáveis detectadas pelo servidor OPC. Ela também é responsável pelo processo de escrita no servidor de acordo com a solicitação do servidor de *sockets* desenvolvido;

- **WriteItem(..)**: este método foi criado para conter o processo de leitura síncrona do sistema. Ele é chamado pela rotina *opcClient(void)* quando existe a necessidade de realizar alguma escrita no servidor OPC. Ele, portanto, cria e recupera uma referência para a interface *IOPCSyncIOInterface* e realiza a escrita desejada. Tal método recebe como parâmetro o grupo para escrita, a quantidade de itens a serem escritos, os *OPC server handle items* e os respectivos valores de inserção. Checa-se se o resultado de escrita foi bem-sucedido e, então faz o *release* da interface recuperada;
- **InstantiateServer(..)**: este método não foi modificado e basicamente instancia o servidor de simulação do lado do cliente;
- **AddTheGroup(..)**: este método não foi modificado e basicamente adiciona o grupo OPC no servidor de simulação e recupera sua referência para o sistema (a única alteração realizada foi a mudança do nome do grupo por desejo dos integrantes, porém tal mudança é irrelevante para o projeto);
- **AddTheItem(..)**: este método foi alterado pelo grupo apenas para adaptá-lo para conseguir receber dinamicamente por parâmetro os handles e a posição em um array dos itens que precisariam ser adicionados no servidor. Com estas informações ele basicamente consegue montar o comando de adição no servidor definindo *szItemID* (recebe o identificador do item a ser criado, tal identificador se encontra em um vetor global no script), o *hClient* (recebe o identificador do lado do cliente para aquela variável. Tal identificador foi definido como o índice daquela variável no vetor definido) e o *vtRequestedDataType* (recebe o tipo da variável a ser criada. Tais tipos também são definidos em um vetor compatível com o vetor de itens). Feito isso os itens são adicionados ao servidor com o auxílio da interface *IOPCItemMgt*;
- **AddInitialItems(..)**: este método foi criado e basicamente é responsável por encapsular as chamadas para o método *AddTheItem(..)*. Ou seja, ele itera por um vetor que contém todos os itens que desejamos que seja cadastrado e, portanto, prepara a chamada para o método que de fato fará esta adição;
- **ReadItems(..)**: este método não é utilizado em nosso sistema, porém optou-se por não removê-lo do sistema, já que faz parte do projeto base *Simple OPC client com modificações*;
- **RemoveItem(..)**: este método não foi modificado e basicamente remove um dado item do servidor OPC de simulação;

- ***RemoveGroup(..)***: este método não foi modificado e basicamente remove um dado grupo do servidor OPC de simulação.

5.2.4 A classe de leitura assíncrona: *SOCDataCallback.cpp*

Esta classe foi modificada para realizar a leitura assíncrona a partir dos valores atualizados de variáveis recebidos pelo servidor de simulação.

O único método modificado comparado ao projeto *Simple OPC Client com modificações*, foi o seguinte:

- ***OnDataChange(..)***: este método anteriormente recebia uma única variável atualizada do servidor OPC, e printava seu valor no console concatenada com a qualidade identificada e a *timestamp* recebida.

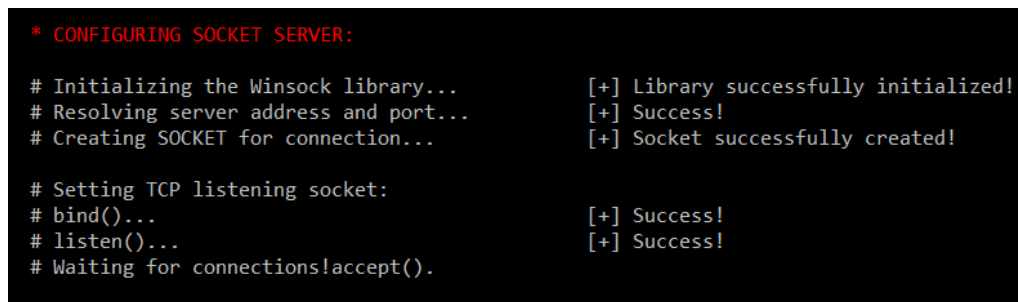
Foi necessário ser feito uma mudança para que tal método recebesse e tratasse todas as variáveis alteradas. Portanto, de acordo com a quantidade de variáveis (itens) recebidos pelo cliente através do servidor, o sistema irá iterar nestas variáveis para que consiga atualizá-las em uma estrutura de dados global criada chamada de ***positionParameters*** já explicada na primeira sub-sessão desta sessão. Além desta atualização de variáveis, este método também fica responsável por printá-las no console do sistema informando, portanto, o identificador do item, o seu valor lido, a qualidade da leitura e o *timestamp* da leitura.

6 Resultados

Como explorado nas seções anteriores de desenvolvimento do projeto é necessário inicializar um servidor de *sockets* TCP/IP e um cliente OPC. No início da *thread* relativa ao servidor de *sockets* realiza-se os passos:

1. Inicializar a biblioteca Winsock 2;
2. Criar um *socket*;
3. Associá-lo a uma porta.
4. Habilitar o *socket* para receber conexões.

Todos esses passos têm seu progresso mostrado no console de execução ao se executar o programa. Isso pode ser visto na Fig. 3 abaixo.



```
* CONFIGURING SOCKET SERVER:

# Initializing the Winsock library...      [+] Library successfully initialized!
# Resolving server address and port...     [+] Success!
# Creating SOCKET for connection...        [+] Socket successfully created!

# Setting TCP listening socket:
# bind()...                               [+] Success!
# listen()...                             [+] Success!
# Waiting for connections!accept().
```

Figura 3: Exibição do progresso de inicialização do servidor de sockets.

Ao se iniciar a rotina do cliente OPC *opcClient()*, inicia-se o ambiente COM, instancia-se o servidor, adiciona-se o grupo dos itens a serem lidos e então se adiciona de fato estes itens de acordo com os identificadores definidos (é importante ressaltar que durante esta adição, armazenamos os *handles* de cada item).

Semelhante a figura anterior, na Fig. 4 temos o progresso de inicialização do cliente OPC mostrado.

```
* CONFIGURING OPC CLIENT SERVER:

# Initializing the COM environment...      [+] Success!
# Instantiating the MATRIKON OPC Server...  [+] Success!

# Adding a group in the INACTIVE state for the moment:
# [OPCCCLIENT] ITEM 0 - Adding the item Random.UInt1
# [OPCCCLIENT] ITEM 1 - Adding the item Random.UInt2
# [OPCCCLIENT] ITEM 2 - Adding the item Random.UInt4
# [OPCCCLIENT] ITEM 3 - Adding the item Saw-toothed Waves.Real4
# [OPCCCLIENT] ITEM 4 - Adding the item Bucket Brigade.UInt1
# [OPCCCLIENT] ITEM 5 - Adding the item Bucket Brigade.Real4

# [OPCCCLIENT] Setting up the IConnectionPoint callback connection.
# [OPCCCLIENT] Changing the group state to ACTIVE.

***** Finished the initialization *****
```

Figura 4: Exibição do progresso de inicialização do cliente OPC.

Uma vez que a inicialização terminou, podemos verificar no console todas as leituras assíncronas do cliente OPC ocorrendo para as quatro variáveis desejadas (referentes aos dados de posição do vagão). Como é um processo de leitura por subscrição, o servidor OPC envia os dados para o cliente OPC sempre que esses sofrem alterações (isso ocorre de um em um segundo).

Na Fig. 5 é possível visualizar o console com tais leituras:

```
# [OPCCCLIENT] Reading items values asynchronously
# [OPCCCLIENT] READ ITEM 0 (Wagon Speed): Value = 8 - Quality: good - Time: 11/10/2020 02:21:25
# [OPCCCLIENT] READ ITEM 1 (Start Sensor Status): Value = 14932 - Quality: good - Time: 11/10/2020 02:21:25
# [OPCCCLIENT] READ ITEM 2 (End Sensor Status): Value = 28112 - Quality: good - Time: 11/10/2020 02:21:25
# [OPCCCLIENT] READ ITEM 3 (Load Weight): Value = 43.18 - Quality: good - Time: 11/10/2020 02:21:25
```

Figura 5: Dados de posição do vagão lidos de forma assíncrona pelo cliente OPC.

As informações de cada leitura assíncrona realizada são:

- **Item ID:** o identificador do item sendo lido;
- **Value:** o valor lido pelo cliente OPC no servidor OPC;
- **Quality:** a qualidade da leitura de acordo com o servidor OPC (boa ou não);
- **Timestamp:** data e hora da leitura executada;

Abaixo, é possível visualizar a tabela que mostra a correspondência do identificador do item com a variável de processo sendo lida:

ID	Nome variável de processo	Identificador variável de processo	Tipo de operação
0	Velocidade do vagão (cm/s)	<i>Random.UInt1</i>	LEITURA
1	Estado sensor início de carga	<i>Random.UInt2</i>	LEITURA
2	Estado sensor fim de carga	<i>Random.UInt4</i>	LEITURA
3	Balança de carga no vagão (kg)	<i>Saw-toothed Waves.Real4</i>	LEITURA
4	Tempo abertura do silo (seg.)	<i>Bucket Brigade.UInt1</i>	ESCRITA
5	Quantidade de minério (kg)	<i>Bucket Brigade.Real4</i>	ESCRITA

Da mesma forma, caso uma conexão via *sockets* TCP/IP com o sistema de otimização tenha ocorrido, aparecerá no console todas as mensagens recebidas pelo servidor de *sockets* e sua respectiva resposta.

De dois em dois segundos o sistema de otimização envia uma requisição de dados de posicionamento do vagão para o servidor de *sockets* e esse responde com uma mensagem contendo os dados como especificado na descrição do trabalho. Um exemplo disso pode ser visto na Fig. 6

```
[SOCKETSERVER] -> Data request message received...      [+] Bytes received: 8
# Received: 55|00041

[SOCKETSERVER] <- Sending wagon positioning message...    [+] Success!Bytes sent 34
# Message sent: 55|00042|00221|00001|00001|0063.50
```

Figura 6: Servidor de *sockets* - Recebimento mensagem de requisição de dados e a resposta com os dados de posição do vagão.

OBS: Na especificação do trabalho mostra que os estado dos sensores de início de carga e de fim de carga devem ser enviados ao sistema de otimização como LIGADO (00001) ou DESLIGADO (00000). Porém, também segundo a especificação esses dados, no servidor OPC, são do tipo *Random.UInt2* (para o sensor de início de carga) e *Random.UInt4* (para o sensor de fim de carga). Dessa maneira o cliente OPC pode ler qualquer valor randômico especificado por esse tipo, contudo, por convenção do grupo, consideramos qualquer valor maior que zero como LIGADO, e igual a zero como DESLIGADO. Assim no momento de envio da mensagem de posicionamento do vagão para o cliente de *sockets* enviamos 00001 ou 00000 nos campos "Estado do sensor de início de carga" e "Estado do sensor de fim de carga".

Na Fig. 7, pode-se observar o funcionamento do programa de otimização (desenvolvido pelo professor Luiz Themystokliz Santos Mendes) referente a parte mostrada na Fig 6.

Observando essas duas figuras acima, pode-se notar que uma mensagem de solicitação de dados é disparada pelo sistema de otimização. O servidor de *sockets*, então, recebe essa mensagem e responde com uma mensagem contendo os dados de posicionamento do vagão. E, por fim, o sistema de otimização recebe a mensagem enviada pelo servidor.

```
Msg de requisicao de dados enviada ao sistema de controle [192.168.68.141]:  
55|00041  
  
Mensagem de dados recebida do sistema de controle [192.168.68.141]:  
55|00042|00221|00001|00001|0063.50
```

Figura 7: Sistema de otimização - Envio de mensagem de requisição de dados e recebimento da resposta com os dados de posição do vagão.

Além da mensagem de solicitação de dados de posicionamento do vagão, o sistema de otimização também pode enviar (sempre que o usuário tecla "p") uma mensagem contendo os parâmetros de carregamento do vagão. Nesse caso o servidor de *sockets* lê esses dados, salvando-os para serem enviados do cliente OPC para o servidor OPC. Além disso, ele retorna uma mensagem de confirmação ao sistema de otimização. Na Fig. 8 e na Fig. 9 abaixo pode-se observar o funcionamento do servidor e do cliente (sistema de otimização) de *sockets* respectivamente.

```
[SOCKETSERVER] -> Charging parameters received...  
* Opening time of silo gate: 300 sec.  
* Amount of ore to be loaded : 1250.00 kg.          [+] Bytes received: 22  
# Received: 99|00043|00300|1250.00  
  
[SOCKETSERVER] <- Sending ACK msg to TCP / IP client...  [+] Success! Bytes sent 8  
# Message sent: 00|00044
```

Figura 8: Servidor de *sockets* - Recebimento mensagem contendo os parâmetros de carregamento e a resposta de confirmação.

```
Mensagem de parametros de carregamento do vagao enviada ao sistema de controle [192.168.68.141]:  
99|00043|00300|1250.00  
  
Mensagem de ACK recebida do sistema de controle [192.168.68.141]:  
00|00044
```

Figura 9: Sistema de otimização - Envio de mensagem contendo os parâmetros de carregamento e a resposta de confirmação recebida.

Como falado acima, uma vez que parâmetros de carregamento são recebidos, o cliente OPC deve escrevê-los no servidor OPC, como é mostrado na Fig. 10. Vale ressaltar que essa escrita ocorre usando o método de escrita síncrona da *IOPCSyncIOInterface*.

```
# [OPCCCLIENT] Writing synchronously LOADING PARAMETERS on OPC Server:  
# [OPCCCLIENT] WRITE ITEM 4 (Open time of the silo gate): Value = 300  
# [OPCCCLIENT] WRITE ITEM 5 (Ore quantity): Value = 1250.00
```

Figura 10: Cliente OPC - Escrita de dados de carregamento no servidor OPC.

As informações de cada escrita realizada são:

- **Item ID:** o identificador do item sendo escrito;
- **Value:** o valor escrito pelo cliente OPC no servidor OPC.

Pode-se notar que os valores escritos, mostrados na Fig. 10, são referentes aos enviados na Fig. 9 pelo sistema de otimização.

Na Fig. 11 pode-se observar um trecho do funcionamento completo do sistema. Nessa figura, observamos os seguintes passos:

1. Ocorre uma leitura assíncrona dos dados de posicionamento do vagão, pelo cliente OPC. O servidor OPC envia esses dados para o cliente sempre que ocorre uma atualização desses (isso ocorre de um em um segundo).
2. Ocorre outra leitura assíncrona dos dados de posicionamento do vagão, pelo cliente OPC.
3. O servidor de *sockets* recebe, do sistema de otimização, uma mensagem de solicitação de dados de posicionamento do vagão. Essa requisição de dados é enviada de maneira periódica (a cada 2 segundos), pelo sistema de otimização.
4. O servidor de *sockets* responde, então, com uma mensagem contendo os dados atualizados em 2.
5. O servidor de *sockets* recebe, do sistema de otimização, uma mensagem contendo os parâmetros de carregamento. Esses dados são enviados sempre que se tecla "p" no sistema de otimização. O servidor de *sockets* salva os parâmetros recebidos em variáveis globais, que a *thread* relativa ao cliente OPC possa acessar e setar a flag `SHOULD_WRITE` indicando que deve-se escrever esses dados no servidor OPC.
6. O servidor de *sockets* responde com uma mensagem de confirmação.
7. Ocorre outra leitura assíncrona dos dados de posicionamento do vagão, pelo cliente OPC.
8. O cliente OPC escreve os dados, recebidos em 5, no servidor da Matrikon.
9. Ocorre outra leitura assíncrona dos dados de posicionamento do vagão, pelo cliente OPC.

10. O servidor de *sockets* recebe, do sistema de otimização, uma mensagem de solicitação de dados de posicionamento do vagão.
11. O servidor de *sockets* responde, então, com uma mensagem contendo os dados atualizados em 9.

```
# [OPCCLIENT] Reading items values asynchronously
# [OPCCLIENT] READ ITEM 0 (Wagon Speed): Value = 6 - Quality: good - Time: 11/10/2020 10:00:04
# [OPCCLIENT] READ ITEM 1 (Start Sensor Status): Value = 28357 - Quality: good - Time: 11/10/2020 10:00:04
# [OPCCLIENT] READ ITEM 2 (End Sensor Status): Value = 4474 - Quality: good - Time: 11/10/2020 10:00:04
# [OPCCLIENT] READ ITEM 3 (Load Weight): Value = 78.74 - Quality: good - Time: 11/10/2020 10:00:04

# [OPCCLIENT] Reading items values asynchronously
# [OPCCLIENT] READ ITEM 0 (Wagon Speed): Value = 157 - Quality: good - Time: 11/10/2020 10:00:05
# [OPCCLIENT] READ ITEM 1 (Start Sensor Status): Value = 22586 - Quality: good - Time: 11/10/2020 10:00:05
# [OPCCLIENT] READ ITEM 2 (End Sensor Status): Value = 7738 - Quality: good - Time: 11/10/2020 10:00:05
# [OPCCLIENT] READ ITEM 3 (Load Weight): Value = 81.28 - Quality: good - Time: 11/10/2020 10:00:05

[SOCKETSERVER] -> Data request message received...          [+] Bytes received: 8
# Received: 55|00007

[SOCKETSERVER] <- Sending wagon positioning message...      [+] Success!Bytes sent 34
# Message sent: 55|00008|00157|00001|00001|0081.28

[SOCKETSERVER] -> Charging parameters received...
* Opening time of silo gate: 300 sec.
* Amount of ore to be loaded : 1250.00 kg.                  [+] Bytes received: 22
# Received: 99|00009|00300|1250.00

[SOCKETSERVER] <- Sending ACK msg to TCP / IP client...    [+] Success!Bytes sent 8
# Message sent: 00|00010

# [OPCCLIENT] Reading items values asynchronously
# [OPCCLIENT] READ ITEM 0 (Wagon Speed): Value = 227 - Quality: good - Time: 11/10/2020 10:00:06
# [OPCCLIENT] READ ITEM 1 (Start Sensor Status): Value = 26151 - Quality: good - Time: 11/10/2020 10:00:06
# [OPCCLIENT] READ ITEM 2 (End Sensor Status): Value = 22857 - Quality: good - Time: 11/10/2020 10:00:06
# [OPCCLIENT] READ ITEM 3 (Load Weight): Value = 83.82 - Quality: good - Time: 11/10/2020 10:00:06

# [OPCCLIENT] Writing synchronously LOADING PARAMETERS on OPC Server:
# [OPCCLIENT] WRITE ITEM 4 (Open time of the silo gate): Value = 300
# [OPCCLIENT] WRITE ITEM 5 (Ore quantity): Value = 1250.00

# [OPCCLIENT] Reading items values asynchronously
# [OPCCLIENT] READ ITEM 0 (Wagon Speed): Value = 237 - Quality: good - Time: 11/10/2020 10:00:07
# [OPCCLIENT] READ ITEM 1 (Start Sensor Status): Value = 17713 - Quality: good - Time: 11/10/2020 10:00:07
# [OPCCLIENT] READ ITEM 2 (End Sensor Status): Value = 31707 - Quality: good - Time: 11/10/2020 10:00:07
# [OPCCLIENT] READ ITEM 3 (Load Weight): Value = 86.36 - Quality: good - Time: 11/10/2020 10:00:07

[SOCKETSERVER] -> Data request message received...          [+] Bytes received: 8
# Received: 55|00011

[SOCKETSERVER] <- Sending wagon positioning message...      [+] Success!Bytes sent 34
```

Figura 11: Trecho de funcionamento da aplicação contendo o servidor de *sockets* e o cliente OPC.

Na Fig. 12, pode-se observar o trecho referente a Fig. 11, porém da perspectiva do sistema de otimização.

```
Sistema de Otimizacao de Carregamento de Vagoes: data/hora local = 11-10-2020 19:00:05
Msg de requisicao de dados enviada ao sistema de controle [192.168.68.141]:
55|00007

Mensagem de dados recebida do sistema de controle [192.168.68.141]:
55|00008|00157|00001|00001|0081.28

Mensagem de parametros de carregamento do vagao enviada ao sistema de controle [192.168.68.141]:
99|00009|00300|1250.00

Mensagem de ACK recebida do sistema de controle [192.168.68.141]:
00|00010

Sistema de Otimizacao de Carregamento de Vagoes: data/hora local = 11-10-2020 19:00:07
Msg de requisicao de dados enviada ao sistema de controle [192.168.68.141]:
55|00011

Mensagem de dados recebida do sistema de controle [192.168.68.141]:
55|00012|00237|00001|00001|0086.36
```

Figura 12: Sistema de otimização- Trecho de funcionamento da do sistema de otimização.

7 Instruções de Compilação e Execução da Aplicação

Pode-se compilar o projeto, abrindo-o a partir do arquivo de projeto *tp1_sistemas_distribuidos.sln* no Visual Studio e pressionando o ícone de compilação.

Pode-se também rodar o programa executável:

`\tp_sistemas_distribuidos\Debug\tp1_sistemas_distribuidos.exe`.

8 Conclusão

Com esse trabalho foi possível entender mais sobre o conteúdo estudado até o momento na disciplina Sistemas Distribuídos para Automação. Foi possível observar na prática o funcionamento de clientes e servidores de *sockets* TCP/IP e OPC, consolidando ainda mais todo conteúdo aprendido.

Foi ainda mais interessante perceber como pode ser desenvolvido um sistema distribuído capaz de automatizar tarefas industriais, como por exemplo, o carregamento de vagões mostrado nesse trabalho.

Todas as tarefas obrigatórias requisitadas foram desenvolvidas e assim obtivemos um programa com duas *threads* capaz de receber e enviar mensagens via TCP/IP e ainda escrever e ler dados em um servidor OPC clássico. Com isso, temos um sistema capaz de automatizar o carregamento de vagões como proposto no enunciado do trabalho.

9 Rererências

[1] BARROS, A. Distribuição horária de lotes de vagões GDE para carregamento de minério na EFVM-. Monografia-Curso de Especialização de Transportes Ferroviário de Carga-Instituto Militar de Engenharia, 2008.

[2] Vale – Nossos Negócios > Logística > Transporte Ferroviário.
Disponível em: www.vale.com >. Acesso em: 30 set. 2020.

[3] X. Yang, Xiang Wu, Zhenpeng Zhao and Yingjie Li, "Hand tele-rehabilitation in haptic virtual environment,"2007 IEEE International Conference on Robotics and Biomimetics (ROBIO), Sanya, 2007, pp. 145-149, doi: 10.1109/ROBIO.2007.4522150.