Alexandre da Silva

Módulo de Comunicação Abstracto

Dissertação de Mestrado em Engenharia Física no Ramo da Instrumentação apresentado no Departamento de Física da Faculdade de Ciências de Tecnologias da Universidade de Coimbra

2013

Universidade de Coimbra

Módulo de Comunicação Abstracto

Alexandre da Silva
Engenharia Física
Departamento de Física
Faculdade de Ciências e Tecnologias
Universidade de Coimbra

Orientador: Professor Doutor Jorge Afonso Cardoso Landeck
Categoria: Professor Auxiliar
Departamento: Departamento de Física

Júri:

Presidente: José Paulo Pires Domingues
Categoria: Professor Auxiliar
Departamento: Departamento de Física

Nome: Jorge Afonso Cardoso Landeck
Categoria: Professor Auxiliar
Departamento: Departamento de Física

Nome: Custódio Francisco Melo Loureiro
Categoria: Professor Auxiliar
Departamento: Departamento de Física

Nome: Rita Patrícia Dinis Carreira
Categoria: Engª de Sistemas
Instituição: ISA
Morada: Rua D. Manuel I, 30
3020-330 Coimbra

Setembro 2013

Resumo

Nesta dissertação é apresentada uma análise empírica de vários protocolos de comunicação com o intuito de concretizar uma generalização dos mesmos. Optou-se por uma análise empírica dada a inexistência de qualquer generalização deste tipo. São apresentados os vários obstáculos encontrados na procura de tal objectivo e são propostas duas soluções distintas para obter um módulo de comunicação abstracto. Um módulo que se pretende inserir num sistema já existente projectado pela ISA – o iCenterX – permitindo assim interligar sistemas de baixo nível a sistemas de alto nível, que sem esse módulo não teriam interoperabilidade.

Abstract

This thesis presents an empirical analysis of multiple communication protocols in order to achieve a generalization of these protocols. An empirical analysis was chosen due to the inexistence of this type of generalization. It also presents the various obstacles encountered in seeking this goal and two distinct solutions are suggested to obtain an abstract communication module. A module which is intended to be included in a system designed by ISA – the iCenterX – allowing to interconnect low-level systems with high-level systems, which wouldn't interoperate correctly without such module.

Agradecimentos

O trabalho que aqui se apresenta só foi possível graças à colaboração e apoio de algumas pessoas, às quais não posso deixar de expressar o meu agradecimento:

Em primeiro lugar ao professor Jorge Landeck pela possibilidade que me deu de realizar este trabalho, pelo acompanhamento e orientação. À Rita Carreira pela paciência, disponibilidade e conselhos. Ao Pedro Sá pela ajuda e disponibilidade.

Àqueles que me acompanharam ao longo deste percurso académico e que de alguma forma me ajudaram. Em especial ao Tiago Neves, ao Carlos Azevedo, ao João Rodrigues, ao João Perdiz e ao Pierre Barroca pela amizade e pela ajuda.

À Kelly Teixeira pela ajuda e pelos conselhos. Um especial agradecimento a toda a minha família que me ajudou, que me apoiou e que me aturou. Nomeadamente aos meus pais e à minha irmã.

E claro um grande agradecimento à Annabelle Teixeira por acreditar sempre em mim, pelo apoio incondicional, por estar sempre do meu lado e pelas constantes palavras de confiança e de força.

Índice:

Lista de Siglas:

ADU – *Application Data Unit*

ARP – *Address Resolution Protocol*

ASCII – *American Standard Code for Information Interchange*

CR – *Carriage Return*

CRC – *Cyclic Redundancy Code*

DNP3 – *Distributed Network Protocol v.3*

DNS – *Domain Name Service*

FTP – *File Transfer Protocol*

HTTP – *Hypertext Transfer Protocol*

ICMP – *Internet Control Message Protocol*

IEEE - *Institute of Electrical and Electronics Engineers*

IP – *Internet Protocol*

ISA – *Intelligent Sensing Anywhere*

ISO – *International Organization for Standardization*

JPEG – *Joint Photographic Experts Group*

LF – *Line Feed*

MCA – Módulo de Comunicação Abstracto

MPEG – *Moving Pictures Experts Group*

OLE – *Object Linking and Embedding*

OPC – *OLE Process Control*

OSI – *Open Systems Interconnection*

PDU – *Protocol Data Unit*

RPC – *Remote Procedure Call*

RTU – *Remote Terminal Unit*

SCADA – *Supervisory Control And Data Acquisition*

SMTP – *Simple Mail Transfer Protocol*

SQL – *Structured Query Language*

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

UTF-8 – *8-bit Unicode Transformation Format*

XML – *eXtensible Markup Language*

Lista de Figuras:

Lista de Tabelas:

1. Introdução

Nos dias de hoje em que a informática se está a propagar pelas mais diversas áreas da indústria e também das aplicações domésticas tem-se assistido cada vez mais a tentativas de partilha de informação entre diferentes dispositivos e a comunicação entre estes tem vindo a sofrer alterações no sentido de a tornar cada vez mais compatível. Esta compatibilidade ao nível das comunicações de dados permite a interoperabilidade de diferentes dispositivos. Contudo, ainda estamos muito longe de atingir tal objectivo de uma forma universal e, tendo em conta as diferentes necessidades de cada dispositivo, talvez nunca seja atingida nenhuma compatibilidade perfeita e universal.

Assim, a forma de contornar este problema hoje em dia passa pela utilização de *gateways* ou conversores, de forma a fazer a interface entre diferentes protocolos de comunicação. Assim temos um sistema de aquisição de dados de baixo nível (sensores, actuadores, *displays*, sistemas de armazenamento, entre outros) que envia os dados recolhidos para um sistema de alto nível (são os sistemas responsáveis pelo controlo e tratamento dos dados, que podem ser simples mostradores, bases de dados, aplicações, autómatos ou sistemas mais completos e complexos como sistemas SCADA, através de uma interface, seja esta um simples conversor ou um *gateway*. Esta comunicação ocorre, geralmente, nos dois sentidos, pois, normalmente antes de serem enviados os dados tem que existir um pedido dos mesmos.

Existem até *gateways* universais que apesar do nome não são realmente universais, na realidade são apenas dispositivos com diferentes ligações físicas que permitem instalar diferentes *drivers* consoante os protocolos que se pretendam utilizar, possibilitando, normalmente, a interface entre 3 ou 4 protocolos diferentes.

Num ambiente industrial, os sistemas usados actualmente requerem várias interfaces para cada um dos diferentes dispositivos que não sejam compatíveis com o sistema de alto nível. Ou seja, são necessários vários *gateways* para interligar cada um dos sistemas com diferentes protocolos de comunicação, por vezes, pode até ser necessário ter vários *gateways* que façam interface entre os mesmos protocolos por questões de espaço e distâncias na instalação. Tudo isto significa um aumento nos custos da instalação e significa também que uma futura alteração de dispositivos implica a troca por dispositivos com o mesmo protocolo de comunicação, que entretanto até poderão deixar de existir, ou caso contrário implica a alteração não só dos dispositivos em si mas também a alteração das interfaces existentes, aumentando significativamente os custos.

Justifica-se assim a existência de um módulo de comunicação que permita a interface vários tipos de protocolos de comunicação. Um módulo que permita até interligar diferentes sistemas de baixo nível ao mesmo sistema de alto nível ao mesmo tempo. Um sistema destes não necessitaria de ser trocado mesmo que se trocasse os dispositivos, pois poderia comunicar com os novos dispositivos mesmo que o protocolo fosse diferente dos anteriores. Este sistema reduziria claramente a quantidade de *gateways* existentes numa instalação e, consequentemente, o custo da mesma. Para além de que a manutenção da instalação seria mais simples e podem ser adicionados vários dispositivos à instalação sem ser necessário novas interfaces.

1.1. Objectivos

Tendo em conta a crescente necessidade de interligar vários sistemas, cujos protocolos de comunicação não permitem interoperabilidade, é interessante ter apenas uma ferramenta que permita criar e manter várias ligações entre esses diferentes sistemas, isto é, que seja capaz de obter dados de um ou vários dispositivos (fonte), seja qual for o seu protocolo de comunicação, e exportar esses dados para um ou vários sistemas (destino) sem qualquer problema de comunicação.

Assim, uma ferramenta destas é capaz de fazer uma interligação simples entre dois sistemas, recolhendo dados de uma fonte através de qualquer protocolo e exportando-os para um determinado destino:

Figura 1: Recolha de dados de uma fonte e exportação dos mesmos para um determinado destino.

Ou recolher dados de várias fontes, exportando-os para um ou vários destinos:

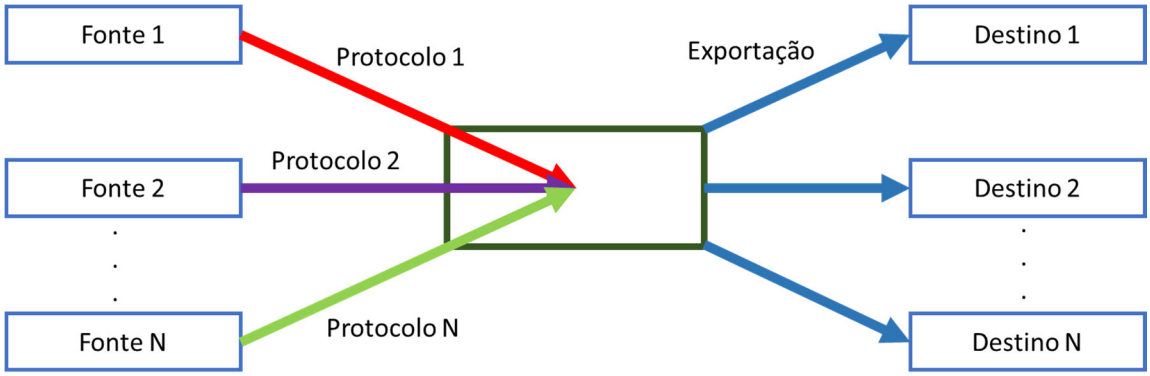

Figura 2: Recolha de dados de várias fontes com diferentes protocolos e exportação desses dados para vários destinos.

Para além de permitir estas interligações simples esta ferramenta deve permitir um controlo mais complexo, sendo possível escolher que dados recolher de qual(quais) fonte(s) e para qual(quais) destino(s) os exportar:

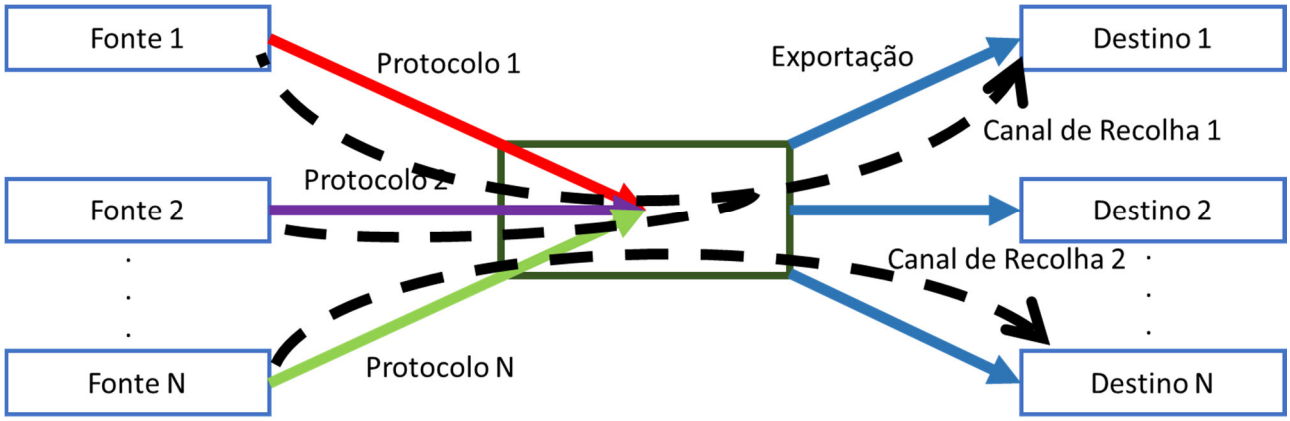

Figura 3: Exemplo de dois canais de recolha de dados

Na Figura 3 temos dois canais de recolha, o primeiro corresponde à recolha de dados das fontes 1 e 2 e à exportação dos mesmos para o destino 1. O segundo corresponde à recolha de dados da fonte N e à exportação dos mesmos para o destino N.

O objectivo desta dissertação passa por demonstrar as fases iniciais de concretização de um Módulo de Comunicação Abstracto (abstracto por não ser um módulo pré-determinado para alguns protocolos em particular, mas ser configurável para qualquer protocolo) e pela idealização das restantes. Este módulo será um software genérico que permitirá implementar diferentes protocolos sem qualquer alteração ao código, sendo apenas alterado um ou vários ficheiros externos para configuração simples de alguns detalhes do protocolo pretendido. O objectivo final é então a inserção deste módulo num sistema já existente, chamado iCenterX, que será explicado mais à frente (capítulo 1.3), para transformar esse sistema nesta ferramenta "ideal" supramencionada.

1.2. Fases do Projecto

Este projecto foi proposto pela ISA com o objectivo de inserir este módulo de comunicação abstracto num sistema já existente, o iCenterX. Este sistema é desenvolvido com base na linguagem de programação C# (C Sharp) e com o Visual Studio 2012 como meio de programação.

Tendo em conta o défice de conhecimentos relativamente ao meio de programação e à linguagem, definiu-se numa fase inicial começar por solidificar os conhecimentos da linguagem e da interface de desenvolvimento seguindo-se depois para a análise e compreensão do programa já existente.

Numa segunda fase pretendia-se começar a ter um melhor conhecimento de como funciona a implementação de um protocolo de comunicação, assim, fez-se a implementação de dois protocolos de comunicação, Modbus RTU e Modbus ASCII, no sistema já existente. Esta fase teve várias intenções: em primeiro lugar aplicar os conhecimentos adquiridos na primeira fase; em segundo perceber como é feita a implementação de um novo protocolo de comunicação no sistema existente; por último, pretendia-se começar a analisar detalhadamente diferentes protocolos para começar a idealizar uma generalização. Isto é, procurar os aspectos comuns e os aspectos característicos e distintivos de cada um dos protocolos, para que estes últimos fossem então colocados num ficheiro de configuração simples, do tipo XML, que fosse lido pelo módulo para que com base na generalização efectuada resultasse um protocolo particular. Em suma, pretende-se criar um ou vários ficheiros de configuração por cada protocolo. Estes ficheiros são escritos no formato XML e são posteriormente lidos pelo módulo por forma a implementar correctamente o protocolo pretendido com as características pretendidas (Figura 4).

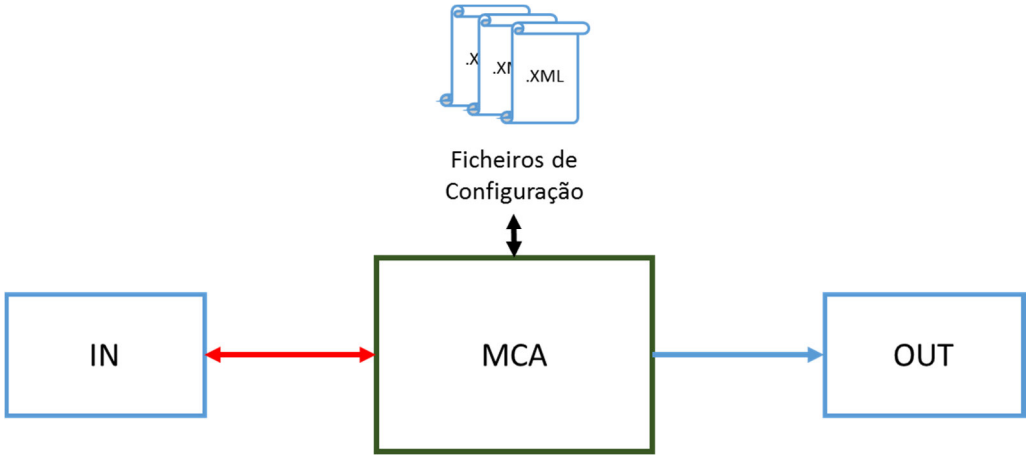

Figura 4: O módulo necessita apenas de uns simples ficheiros de configuração externos por forma a implementar o protocolo.

Concluída esta segunda fase passou-se para o início da concretização do módulo de comunicação abstracto (tendo por base a generalização encontrada após a análise dos dois diferentes protocolos na segunda fase). O objectivo desta fase era conseguir obter um módulo abstracto que funcionasse para os dois protocolos analisados na segunda fase, bem como os ficheiros de configuração XML para cada um deles. Foi também

definido nesta fase que apenas começaríamos com protocolos cuja codificação dos dados fosse em bytes, excluindo assim os protocolos mais complexos a nível de generalização.

Tendo a terceira fase concluída e um módulo abstracto a funcionar para dois protocolos diferentes é então necessário avançar para a seguinte fase e escolher um outro protocolo de comunicação e tentar utilizá-lo com o módulo concebido, por forma a verificar a sua qualidade, isto é, criam-se os ficheiros de configuração necessários para este protocolo e verifica-se se o módulo de comunicação consegue utilizar essa informação para implementar correctamente o protocolo. O protocolo escolhido foi o DNP3. Caso esse protocolo seja bem implementado escolhe-se outro protocolo diferente e repete-se o processo, caso a implementação falhe é então necessário analisar os três protocolos procurando uma nova forma de generalização, voltando de seguida ao início desta quarta fase novamente.

Como se pode perceber, o caminho definido para a concretização deste projecto passa por uma análise empírica, comparando vários protocolos, para tentar obter uma generalização. Convém também referir que os objectivos e o faseamento foram adaptados ao longo do projecto consoante os obstáculos encontrados e as necessidades do mesmo.

1.3. A ISA e o Projecto

A ISA (*Intelligent Sensing Anywhere*) é uma empresa sediada em Coimbra que se especializa em soluções tecnológicas de eficiência, melhoria de processos e telemetria nas áreas de energia e combustíveis.

Uma das soluções que a ISA oferece é o iCenterX. Trata-se de um sistema que permite recolher dados de uma fonte, e exportá-los para um destino – tradicionalmente ficheiros de texto ou *web services*, facilitando processos de monitorização.

A imagem seguinte ilustra o funcionamento do iCenterX:

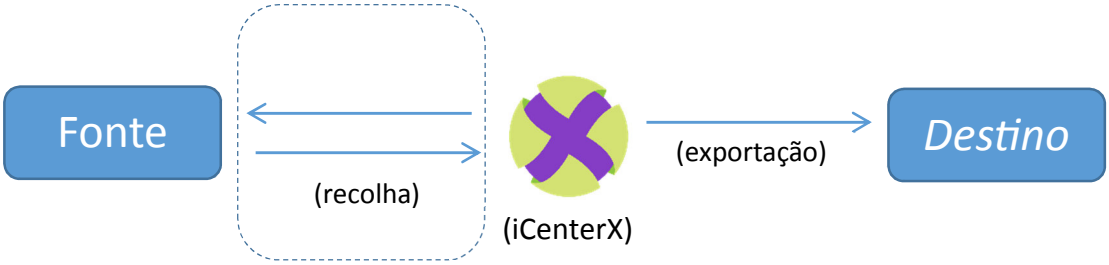

Figura 5: Funcionamento do iCenterX [1].

O iCenterX é uma solução de *software* com funcionamento modular, e apresenta os seguintes módulos:

- *Serviço:*
  - Responsável pelo controlo do funcionamento da aplicação como um todo;
- *Schedule*
  - Responsável pelo controlo e calendarização dos pedidos;
- *Exporter*

- Responsável pela exportação de dados;

- Controller
  - Responsável por recolher dados.

A recolha de dados do iCenterX é efectuada utilizando vários controladores de comunicação geridos pelo serviço. O sistema verifica se existem canais de recolha para efectuar e utiliza esses mesmos controladores para tratar o pedido dos canais de recolha.

Um canal representa uma fonte de dados que é recolhida pelo iCenterX, que é definida por um ficheiro de configuração (Figura 6) que contém todos os parâmetros necessários. Este ficheiro contém:

- Parâmetros de comunicação;
- Parâmetros de recolha do Canal;
- *Devices* do canal – grupo de variáveis;
- Variáveis pretendidas em cada *device*.

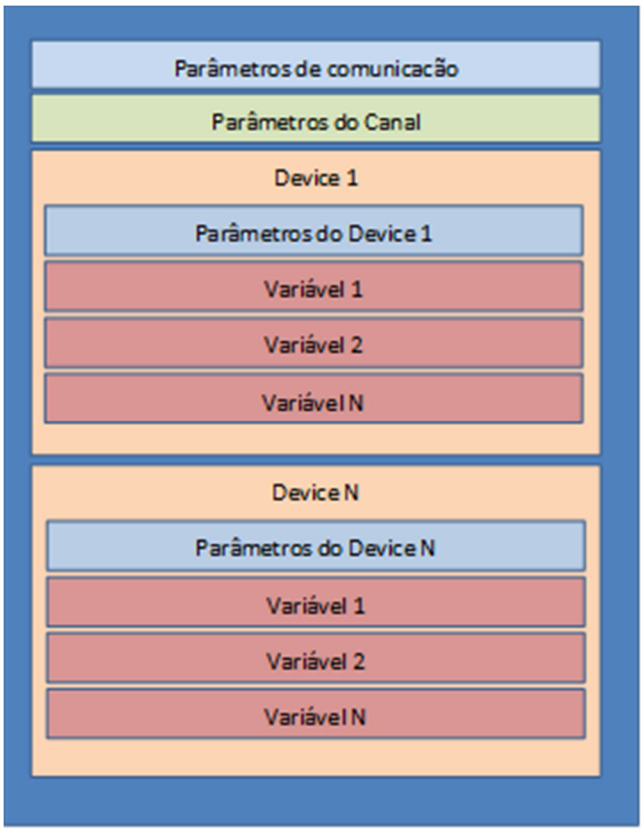

Figura 6: Estrutura geral de um canal [1].

Após a recolha de dados estes são exportados pelo módulo *Exporter* onde se encontram configurados todos os pontos de exportação, podendo estes ser do tipo Ficheiro de Texto ou Webservice.

Cada canal tem um ou mais *templates* de exportação que indicam as variáveis a exportar bem como diversos outros parâmetros necessários à formatação dos dados. Estes *templates* são então associados a pontos de exportação definindo assim o destino dos dados recolhidos.

A configuração do iCenterX é efectuada num interface gráfico, *Monitor*, que permite:

- Configurar canais:
  - Adição/remoção de canais;

- Configurar exportações:
  - Adição/Remoção de novos pontos de exportação;
  - Adição/Remoção de associações entre *templates* e pontos de exportação;

- Configurar controladores de recolha de dados:
  - Adição de novas aplicações de recolha;
  - Remoção de aplicações de recolha existentes;

- Monitorizar o sistema:
  - Permite visualizar o estado dos diferentes componentes;
  - Permite visualizar os consumos de memória;
  - Permite visualizar a utilização do processador;
  - Permite visualizar o conteúdo dos *Logs* da solução;

- Configurar utilizadores.

Tendo em conta que os processos de exportação geralmente utilizados pelo iCenterX não variam muito, decidiu-se aplicar o módulo de comunicação abstracto apenas na importação de dados, ficando a exportação a cargo do iCenterX. Também os meios físicos de ligação já estão implementados de forma bastante completa no iCenterX, assim a definição dos mesmos para qualquer protocolo é feita no sistema, criando-se um objecto do tipo “ADriver” com os métodos necessários para o envio e recepção de mensagens, que é posteriormente usado pelo módulo.

A implementação de um novo protocolo no iCenterX funciona através de um padrão *factory* que fornece instâncias de qualquer tipo de protocolo desde que este siga as regras de implementação necessárias.

Existem duas *factories* (Figura 7), uma para protocolos de importação e outra para protocolos de exportação, que implementam um padrão *singleton*. Estas *factories* vão carregar as bibliotecas, dentro da pasta ‘*bin’* da instalação, que implementem a classe “AImportProtocol” ou “AExportProtocol”.

Os protocolos são diferenciados internamente nas *factories* através do atributo “FactoryKey” que terá de ser implementado em todos os protocolos.

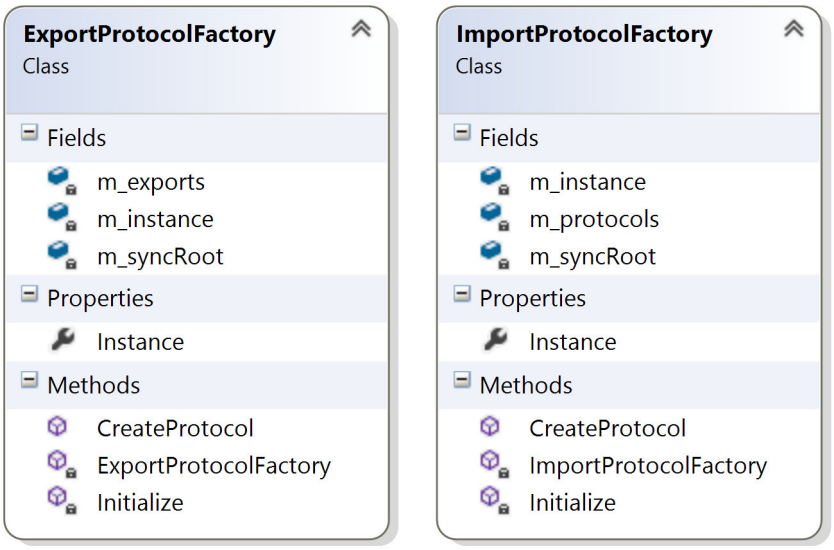

Figura 7: Factories dos protocolos de importação e exportação.

A adaptação do projecto ao iCenterX é então efectuada criando uma classe "GenericProtocol" que implemente a classe "AImportProtocol" (Figura 8). A classe criada terá o módulo de comunicação abstracto implementado, tornando-se assim como que um protocolo de comunicação genérico, que através dos ficheiros de configuração do protocolo requerido irá implementar o mesmo.

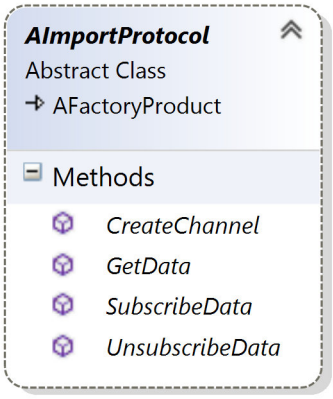

Figura 8: Constituição da classe "AImportProtocol".

Esta adaptação permite que protocolo siga as regras de implementação requeridas pelo iCenterX. Desta forma, o protocolo genérico poderá implementar completamente a classe "AImportProtocol":

- **GetData:** Efectua uma leitura pontual. Este método recebe ainda uma referência para o objecto genérico de dados ChannelData que será populado após uma leitura de dados.
- **CreateChannel:** cria um canal a partir de uma fonte. Nos casos em que existe a capacidade de criar um Channel automaticamente a partir de uma fonte, utiliza-se o método CreateChannel. Este recebe os parâmetros de ligação e preenche uma referência do objecto Channel com a estrutura da fonte.
- **SubscribeData**: efectua uma subscrição a uma fonte e espera por dados. Este método recebe um callback que irá ser chamado (passando um ChannelData) sempre que chegarem novos dados.
- **UnsubscribeData**: termina uma subscrição existente.

Contudo, tendo em conta a já tão grande complexidade do projecto apenas foi implementado o método "GetData". Note-se também que não existe nenhum método que implique o envio de dados pelo módulo, isto é, não é implementado nenhum método do tipo "WriteData", pois, como já foi referido, a exportação dos dados recolhidos fica a cargo do iCenterX.

Em suma, o iCenterX é um sistema que recolhe dados de uma fonte e exporta-os para um destino e esta dissertação pretende idealizar um módulo de comunicação generalizado que seja inserido no iCenterX de forma a permitir aumentar o leque de protocolos de comunicação com os quais o iCenterX possa operar para importar dados, sem ser necessário contínuas alterações ao código-base do mesmo.

2. Protocolos de Comunicação

2.1. Modelo de Camadas OSI

O modelo Open Systems Interconnect (OSI) foi criado pela ISO e foi lançado em 1984. Este modelo possui sete camadas. Este capítulo descreve-as e explica-as, começando com aquela mais abaixo na hierarquia (a física) até chegar à que se encontra mais alto (a aplicação). As camadas encontram-se empilhadas do seguinte modo [9]:

1. Física
2. Ligação de dados
3. Rede
4. Transporte
5. Sessão
6. Apresentação
7. Aplicação

O comportamento do modelo em camadas segue um empacotamento e desempacotamento da mensagem a enviar. Na Figura 9 vemos uma mensagem M ser encapsulada pela fonte ao longo das suas várias camadas. Assim, à mensagem é acrescentado um cabeçalho ao passar da 5ª camada para a 4ª (H4). Na passagem da camada 4 para a 3 a mensagem é dividida em duas partes (H4M1 + M2) e é acrescentado um novo cabeçalho (H3) a cada uma das partes. A segunda camada acrescenta mais um cabeçalho (H2) e um trailer (T2) a cada uma das partes. A camada 1 envia então as duas partes para o destino, onde é feito exactamente o oposto até obtermos a mensagem M. [2]

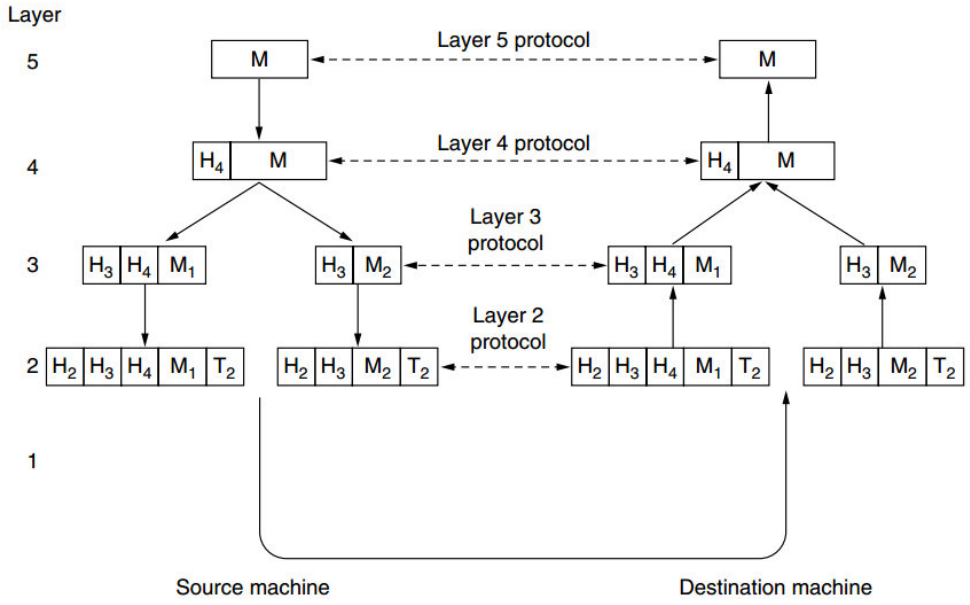

Figura 9: Fluxo de uma mensagem M ao longo de 5 camadas de uma fonte até ao destino [2].

2.1.1 Camada Física

A camada física, a camada mais abaixo do modelo OSI, diz respeito à transmissão e recepção de sequências de bits não processados nem estruturados sobre um suporte físico. Descreve as interfaces eléctrica/óptica, mecânica e funcional do suporte físico, transportando os sinais para todas as camadas superiores. Proporciona:

- Codificação de dados: Modifica o padrão do sinal digital simples (1 e 0) utilizado pelo sistema para melhor gerir as características do suporte físico, e para auxiliar na sincronização de bits e pacotes. Determina:
  - Qual estado de sinal representa binário 1;
  - Como o receptor conhece quando começa o "tempo do bit";
  - Como o receptor delimita um pacote.
- Ligação ao suporte físico:
  - Quantos pinos possuem os conectores e para que serve cada um dos pinos.
- Técnica de transmissão: determina se os bits codificados serão transmitidos por uma sinalização de banda de base (digital) ou de banda larga (analógica).
- Transmissão do suporte físico: transmite bits como sinais eléctricos ou ópticos apropriados para o suporte físico, e determina:
  - Que opções do suporte físico podem ser utilizadas;
  - Quantos volts/db devem ser utilizados para representar um dado estado de sinal, utilizando um dado suporte físico.

Exemplo de protocolos pertencentes a esta camada: IEEE 802.3, RS-232, RS-485. [10]

2.1.2. Camada de Ligação

A camada de ligação de dados ou enlace proporciona uma transferência com controlo de erros dos pacotes de dados de um nó para outro numa camada física, permitindo que as camadas acima assumam uma transmissão virtualmente sem erros sobre a ligação. Para fazer isto, a camada de ligação de dados disponibiliza:

- Estabelecimento e terminação de ligações: estabelece e termina a ligação lógica entre dois nós.

- Controlo de tráfego de pacotes: avisa o nó de transmissão para "recuar" quando não estiver disponível uma memória intermédia para pacotes.
- Sequenciamento de pacotes: transmite/recebe os pacotes sequencialmente.
- Reconhecimento de pacotes: fornece/espera reconhecimento de pacotes. Detecta e recupera de erros que ocorrem na camada física ao retransmitir pacotes não reconhecidos e gerir recepções de pacotes duplicadas.
- Delimitação de pacotes: cria e reconhece fronteiras de pacotes.
- Verificação de erros de pacotes: verifica a integridade dos pacotes recebidos.
- Gestão de acesso ao meio: determina quando o nó "tem o direito" a usar o suporte físico.

Exemplo de protocolos pertencentes a esta camada: IEEE 802.2, 802.3, 802.5. [10]

2.1.3. Camada de Rede

A camada de rede controla a operação da sub-rede, decidindo qual o caminho físico que os dados devem tomar com base nas condições da rede, a prioridade do serviço e outros factores. Proporciona:

- Encaminhamento: encaminha os pacotes entre as redes.
- Controlo de tráfego na sub-rede: os routers (sistemas intermédios de camada de rede) podem instruir uma estação de envio para que esta reduza a sua transmissão de pacotes quando a memória intermédia do router começar a encher.
- Mapeamento de endereço lógico-físico: traduz endereços lógicos, ou nomes, em endereços físicos.

Exemplo de protocolos pertencentes a esta camada: IP, ICMP, ARP. [10]

2.1.3.1. Sub-rede de comunicações

O software da camada de rede deve criar cabeçalhos de modo a que o software da camada de rede nos sistemas intermédios da sub-rede possam reconhecer e utilizar os mesmos para encaminhar dados até o endereço de destino.

Esta camada faz com que as camadas superiores não precisem de saber tudo sobre as tecnologias de transmissão de dados. Estabelece, mantém e termina ligações ao longo da instalação de comunicação intermediária.

2.1.4. Camada de Transporte

A camada de transporte assegura que as mensagens são transmitidas sem erros, em sequência e sem perdas ou duplicações. Faz com que os protocolos das camadas superiores não precisem de se preocupar com a transferência de dados entres eles e os seus pontos.

A dimensão e complexidade de um protocolo de transporte depende do tipo de serviço que possa obter da camada de rede. É necessária uma rede de transporte mínima para ter uma camada de rede fiável com capacidades de circuito virtual.

A camada de transporte proporciona:

- Segmentação de mensagem: aceita uma mensagem da camada acima, divide a mensagem em unidades mais pequenas (se ainda não forem suficientemente pequenas) e encaminha as unidades mais pequenas pela camada de rede. A camada de transporte na estação de destino volta a montar a mensagem.
- Reconhecimento de mensagem: proporciona uma entrega fiável das mensagens ponto a ponto com reconhecimentos.

- Multiplexagem de sessão: multiplexa várias transmissões de mensagens, ou sessões, numa única ligação lógica e monitoriza quais mensagens pertencem a quais sessões.

Tipicamente, a camada de transporte pode aceitar mensagens relativamente grandes, mas as camadas inferiores possuem limites de dimensão de mensagens rigorosos. Consequentemente, a camada de transporte deve decompor as mensagens em unidades mais pequenas, ou pacotes, prefixando um cabeçalho em cada pacote.

As informações no cabeçalho da camada de transporte devem então incluir informações de controlo, tais como sinais de início e fim da mensagem, para permitir que a camada de transporte no outro ponto reconheça as fronteiras da mensagem. Para além disso, se as camadas inferiores não mantiverem a sequência, o cabeçalho de transporte deve conter informações de sequência para permitir que a camada de transporte no ponto de recepção volte a juntar os elementos na ordem correcta, passando a mensagem recebida para a camada acima.

Exemplo de protocolos pertencentes a esta camada: TCP e UDP. [10]

2.1.5. Camada de Sessão

A camada de sessão permite o estabelecimento de sessões entre processos executados em estações diferentes. Proporciona:

- O estabelecimento, manutenção e terminação da sessão: permite que dois processos da aplicação em máquinas diferentes estabeleçam, utilizem e terminem uma ligação, chamada de sessão.
- Suporte de sessão: desempenha as funções que permitem a estes processos comunicar na rede, efectuando funções de segurança, reconhecimento de nome, registo e outros elementos.

Exemplo de protocolos pertencentes a esta camada: SQL, RPC. [10]

2.1.6. Camada de Apresentação

A camada de apresentação formata os dados a serem apresentados à camada da aplicação. Pode ser vista como a tradutora da rede. Esta camada poderá traduzir dados de um formato utilizado pela camada da aplicação para um formato comum na estação de envio, de seguida traduzindo o formato comum num formato conhecido pela camada da aplicação na estação de recepção.

A camada de apresentação proporciona:

- Tradução de código de caracteres: por exemplo, ASCII para UTF-8.
- Conversão de dados: ordem de bits, CR-CR/LF, número inteiro-vírgula flutuante, entre outros elementos.
- Compressão de dados: reduz o número de bits que precisam de ser transmitidos na rede.
- Encriptação de dados: encriptar dados com finalidades de segurança. Por exemplo, encriptação de palavras-passe.

Exemplo de protocolos pertencentes a esta camada: ASCII, MPEG, JPEG. [10]

2.1.7. Camada da Aplicação

A camada da aplicação serve como a janela na qual os utilizadores e processos da aplicação podem aceder aos serviços da rede.

Exemplo de protocolos pertencentes a esta camada: FTP, HTTP, SMTP, DNS. [10]

2.2. Os Protocolos de Comunicação na Indústria

Hoje em dia são muitos os protocolos de comunicação existentes na indústria, e daí resultam grandes incompatibilidades entre diferentes dispositivos, a idealização deste módulo é assim motivada por esse mesmo facto.

Os protocolos usados na indústria e mesmo em aplicações domésticas são dos mais variados tipos, desde protocolos em meios cablados a protocolos *wireless*, passando por protocolos especializados em controlo e medição de grandes quantidades de dispositivos, ou protocolos especializados em medições de grande precisão. Esta diversidade constituiu um dos maiores obstáculos à concretização deste projecto. Como já foi referido as ligações físicas do MCA são realizadas pelo sistema iCenterX, ou seja, o acesso à 1ª camada do modelo OSI é controlado pelo iCenterX. Os protocolos de comunicação *wireless* não foram incluídos nos objectivos deste módulo precisamente por não terem nenhum controlo implementado no iCenterX. Contudo, implementando o mesmo será possível com o módulo idealizado implementar protocolos *wireless*.

HDLC, Modbus ou DNP3 são alguns dos protocolos mais utlizados nos meios industriais. A maioria dos protocolos usados implementam uma versão simplificada da camada de OSI, geralmente não implementam as camadas 3, 4, 5 e 6. Apesar de alguns, como o DNP3, implementarem uma versão mais simples da camada de transporte, a 4ª camada.

2.2.1. HDLC

HDLC *("High level Data Link Control")* é um protocolo que define um conjunto de regras para a transmissão de dados entres pontes de uma rede, sendo por isso um protocolo pertencente à camada de ligação de dados. É um dos protocolos mais utilizados quer para conexões multiponto quer para conexões ponto a ponto e contem operações para estabelecer conexão, transmitir dados, reiniciar conexão e encerrar conexão. Este é um protocolo baseado em bits e foi um dos antecessores de protocolos com o Ethernet. Tem um modo de operação síncrona e a transferência de dados, é feita através de "*frames"* [6].

2.2.2. Modbus

Este é um protocolo de comunicação que pertence à camada de aplicação e à camada de ligação e utiliza o RS-232, RS-485 ou *Ethernet* como meios físicos. É utilizado em sistemas de automação industrial, sendo uns dos mais antigos protocolos utilizados para aquisição de sinais de instrumentos e comando de actuadores, podendo enviar dados discretos ou numéricos. O mecanismo de controlo é do tipo Cliente-Servidor, permitindo até 274 clientes, em que o servidor envia mensagens aos clientes solicitando o envio dos dados lidos pela instrumentação ou envia sinais para comandar os actuadores [7].

2.2.3. DNP3

O protocolo DNP3 (*"Distributed Network Protocol version 3"*), ou protocolo de rede distribuída define a comunicação entre a unidade terminal mestre e as unidades terminais remotas. Este protocolo é aberto e é um dos mais usados em sistemas SCADA para ambientes industriais nas áreas de energia eléctrica, de água, entre outras. Este protocolo pertence às camadas 1, 2, 4 e 7 (apesar da camada 4 não ser implementada de forma completa). Foi desenvolvido para fazer aquisição de informação, envio de comandos, transmissão de pacotes de dados com grande fiabilidade numa determinada sequência. Serve ainda, para comunicações com

múltiplos clientes, comunicações ponto-a-ponto e comunicações com múltiplos servidores. Faz uso da filosofia de *"polling"* verificando se as ligações com as unidades remotas estão funcionais, pois este protocolo permite que estas enviem informações não solicitadas quando algum dos valores ou estados sofre uma alteração [3].

3. Módulo de Comunicação Abstracto

Como já foi referido, a aproximação feita a este projecto foi muito à base de análise e comparação empírica. Foi também referido no capítulo 1.2. Fases do Projectoque após uma primeira fase de aprendizagem e adaptação às ferramentas de programação, bem como ao código base do iCenterX, se começaria a implementar dois protocolos de forma directa no iCenterX, isto é, da mesma forma que todos os outros protocolos foram implementados até então. Isto é explicado em detalhe no capítulo 3.1.1.

Tendo por base as conclusões aferidas nesse mesmo capítulo passou-se para a primeira generalização dos dois protocolos até então usados. Este processo é explicado no capítulo

Por fim, a última das concretizações a ser realizada foi a implementação do protocolo DNP3 com a generalização realizada. Este passo e os obstáculos encontrados são explicados no capítulo 3.1.3.

O capítulo 3.2. apresenta uma proposta sucinta da idealização de uma nova arquitectura com base nos resultados obtidos nas tentativas anteriores, bem como uma solução para a continuar o desenvolvimento da arquitectura actual.

3.1. Primeiras Fases de Concretização

3.1.1. Implementação Directa dos Protocolos Modbus RTU e Modbus ASCII no iCenterX

Nesta parte pretende-se detalhar a implementação do protocolo Modbus, mais especificamente os protocolos Modbus RTU e Modbus ASCII, no iCenterX, para comunicação entre o mesmo e dispositivos com o protocolo em questão.

A comunicação entre o iCenterX e um dispositivo Modbus é realizada através de diferentes tipos de ligações:

- TCP/IP
- *Asynchronous serial transmission* (cabos: EIA/TIA-232-E, EIA-422, EIA/TIA-485-A; fibra-óptica; radio; etc.)
- Modbus PLUS (a high speed token passing network)

Nesta fase apenas foram consideradas as ligações em série. O meio de ligação é gerido automaticamente pelo iCenterX, que após uma tentativa de ligação bem sucedida, disponibiliza um socket TCP que é utilizado para a troca de mensagens. (Mais tarde o iCenterX sofreu uma alteração neste aspecto, sendo que o meio físico e suas propriedades são escolhidos aquando a criação do controlador e canais, disponibilizando um objecto do tipo "ADriver" com os métodos necessários para o envio e recepção de mensagens).

O protocolo Modbus funciona de acordo com uma comunicação do tipo Cliente/Servidor através da troca de pedidos/respostas. Neste protocolo podem-se considerar dois tipos de dispositivos, Master e Slave, sendo que os dispositivos Master funcionam como clientes que enviam pedidos aos dispositivos Slave que funcionam como servidores, respondendo de acordo com os pedidos efectuados pelo Master. As mensagens trocadas entre os dispositivos são compostas por um "Protocol Data Unit" (PDU), independente do tipo de ligação entre os dispositivos. Os diferentes tipos de ligação Modbus podem acrescentar mais alguns campos à mensagem, resultando daí um "Application Data Unit" (ADU).

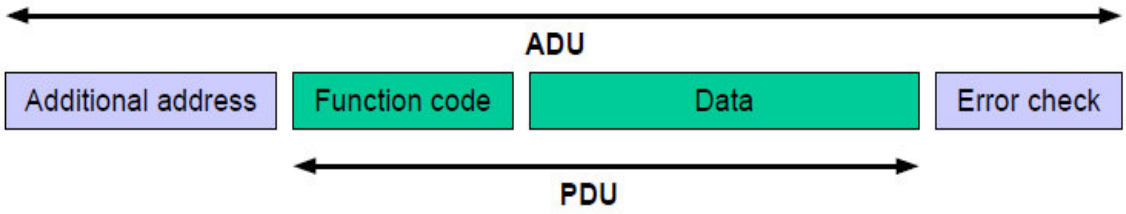

Figura 10: Frame geral das mensagens Modbus [4]

O endereço inicial da mensagem corresponde ao endereço físico do dispositivo Slave para o qual a mensagem se destina. O "Function Code" especifica o tipo de acção a realizar, seguindo-se o bloco "Data" que pode conter vários dados como o endereço inicial do grupo de registos ou coils que se pretende obter ou alterar informações, a quantidade desses registos ou coils, o valor a colocar nos mesmos, entre outros. Por último, coloca-se um "Error Check" para controlo de erros.

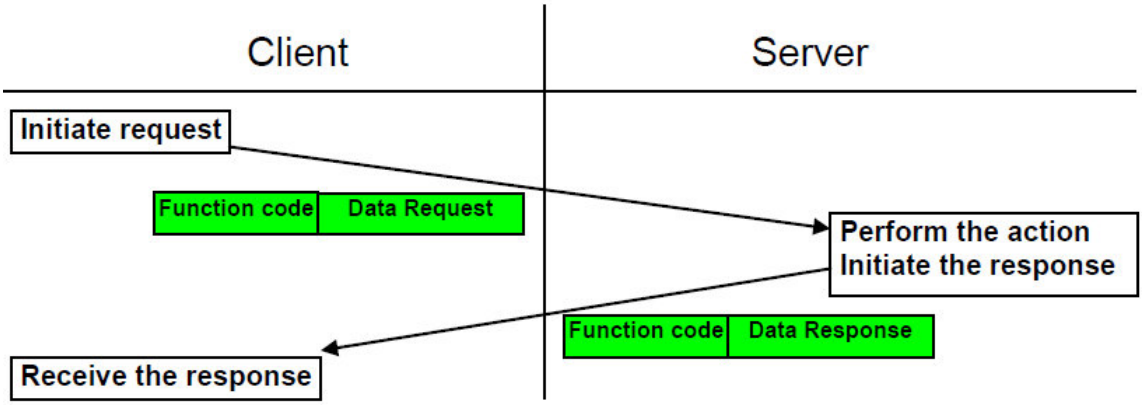

Figura 11: Troca de mensagens sem erros [4]

A troca de mensagens através do protocolo Modbus é, normalmente, efectuada como descrito na Figura 11. Contudo, caso ocorra algum problema no lado do servidor antes do envio da resposta é enviada uma resposta com um código de erro, como descrito na Figura 12. O "Exception Function Code" é igual ao "Function Code" do pedido somado de 0x80 (128).

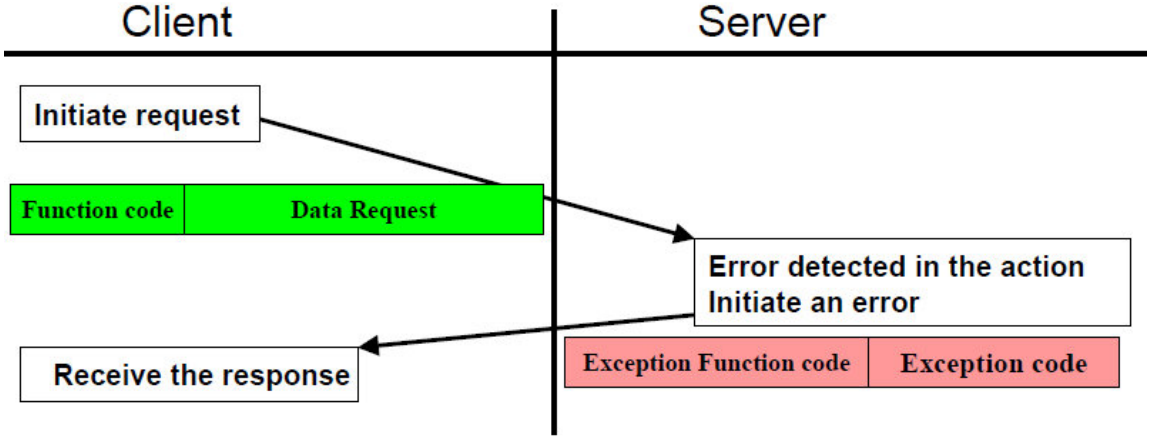

Figura 12: Troca de mensagens com erro [4].

Neste caso específico, apenas foram implementadas as funções de leitura com os códigos 1, 2, 3 e 4, uma vez que o objectivo do sistema se prende com a leitura de informação.

Request

Function code	1 Byte	**0x01**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of coils	2 Bytes	1 to 2000 (0x7D0)

Response

Function code	1 Byte	**0x01**
Byte count	1 Byte	**N***
Coil Status	**n** Byte	n = N or N+1

***N** = Quantity of Outputs / 8, if the remainder is different of 0 ⇒ N = N+1

Error

Function code	1 Byte	**Function code + 0x80**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read discrete outputs 20–38:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**01**	Function	**01**
Starting Address Hi	**00**	Byte Count	**03**
Starting Address Lo	**13**	Outputs status 27-20	**CD**
Quantity of Outputs Hi	**00**	Outputs status 35-28	**6B**
Quantity of Outputs Lo	**13**	Outputs status 38-36	**05**

Figura 13: Exemplo de um pedido e consequente resposta do protocolo Modbus [4].

A implementação deste protocolo Modbus permite o uso do Modbus RTU, assim como o Modbus ASCII. A diferenciação entre estas duas variações do protocolo é efectuada através das classes "SerialRTUTransport" e "SerialASCIITransport" (as várias classes desta implementação estão ilustradas no anexo D). Assim, antes de se proceder ao transporte das mensagens é criada a forma de empacotamento das mesmas, bem como a verificação de erros que difere também nas duas variações do protocolo. É ainda definida a forma de leitura dos dados provenientes da porta de série.

Como já foi referido, todos os protocolos de importação devem implementar a classe abstracta “AImportProtocol” definida na biblioteca Factories do projecto iCenterX.

Como já foi explicado, para este protocolo apenas será implementado o método “GetData”.

Método “GetData”:

Antes de proceder à leitura propriamente dita dos dados começa-se por verificar qual o tipo de protocolo a usar, se RTU ou ASCII, esta informação é fornecida através de um parâmetro "protocolType" no canal, este parâmetro é necessário e obrigatório. Nos anexos A.1 e A.2 temos um exemplo de canal para os protocolos Modbus RTU e ASCII, respectivamente.

De seguida é instanciado um meio de transporte que será usado posteriormente para a troca de mensagens. Esta instância é obtida através das classes já referidas SerialASCIITransport ou SerialRTUTransport, sendo que ambas implementam a classe abstracta ASerialTransport, que trata do envio das mensagens e da verificação das respostas e que por sua vez implementa a classe abstracta ATransport, responsável pelo controlo da troca de mensagens, bem como o controlo dos erros resultantes desta troca.

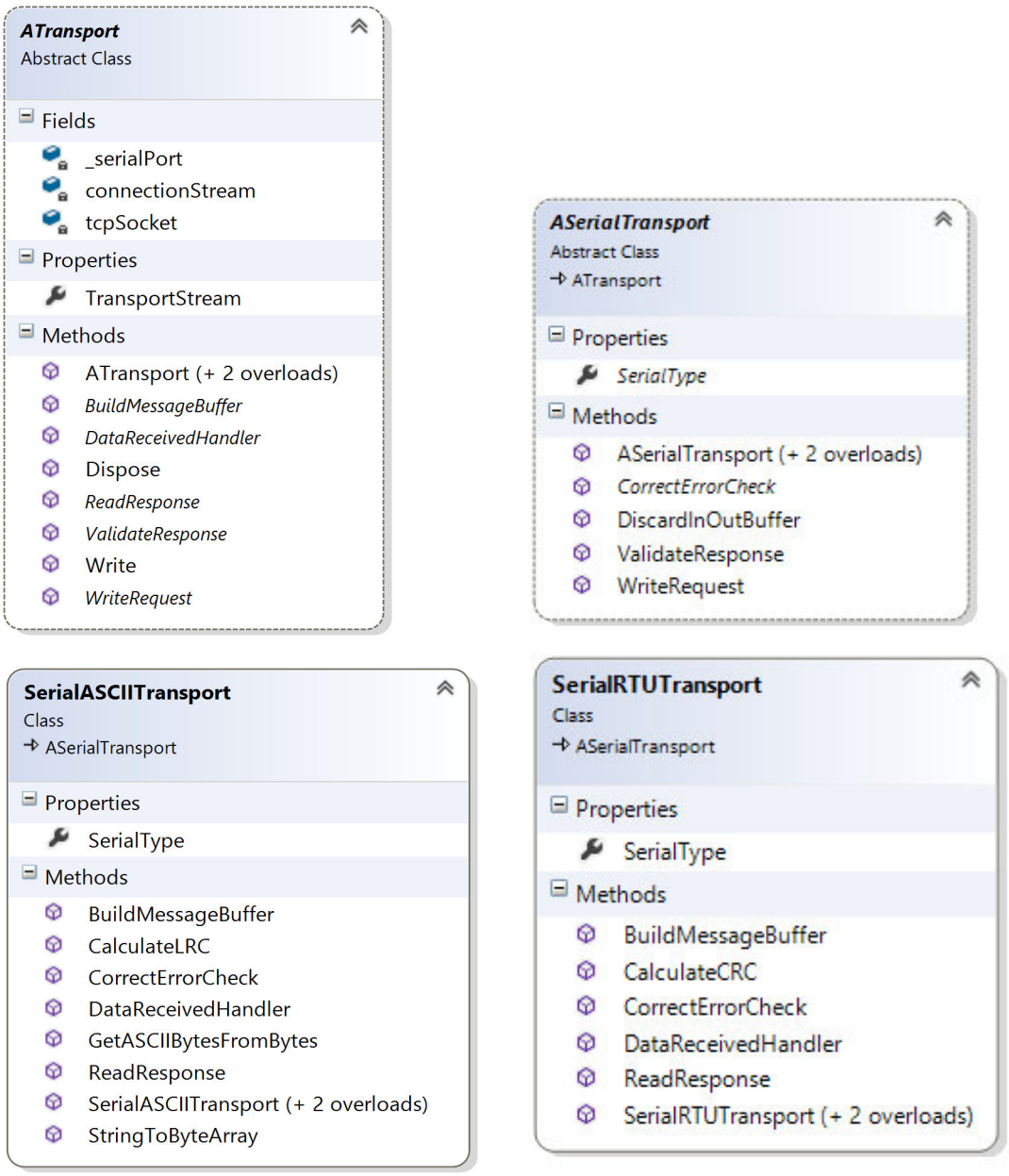

Figura 14: Constituição das Classes "ATransport", "ASerialTransport", "SerialASCIITransport" e "SerialRTUTransport".

As mensagens trocadas podem ser de três tipos diferentes: mensagens de pedidos definidas pela classe ModbusRequestMessage; mensagens de resposta definidas pela classe ModbusResponseMesage; ou mensagens de excepção definidas pela classe ModbusExceptionMessage. Todas estas classes implementam a classe abstracta AModbusMessage que contém as diferentes propriedades existentes em cada tipo de mensagem (entenda-se por propriedades da mensagem os diferentes blocos que a compõem, ex.: “Initial Address”, “Function Code”, “Data” ou “Error Check”), a MessageFrame, o PDU e o tipo de mensagem. Contém ainda os métodos de inicialização “Init” e “SpecificInit” para inicializar e construir os diferentes tipos de mensagens.

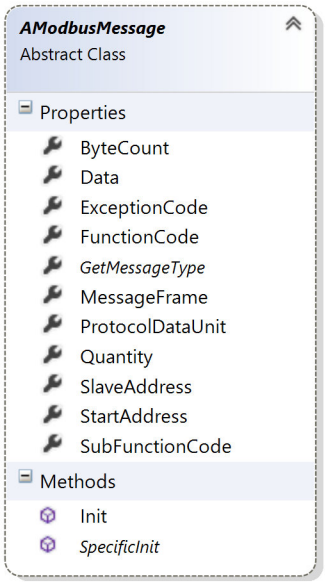

Figura 15: Constituição da classe "AModbusMessage".

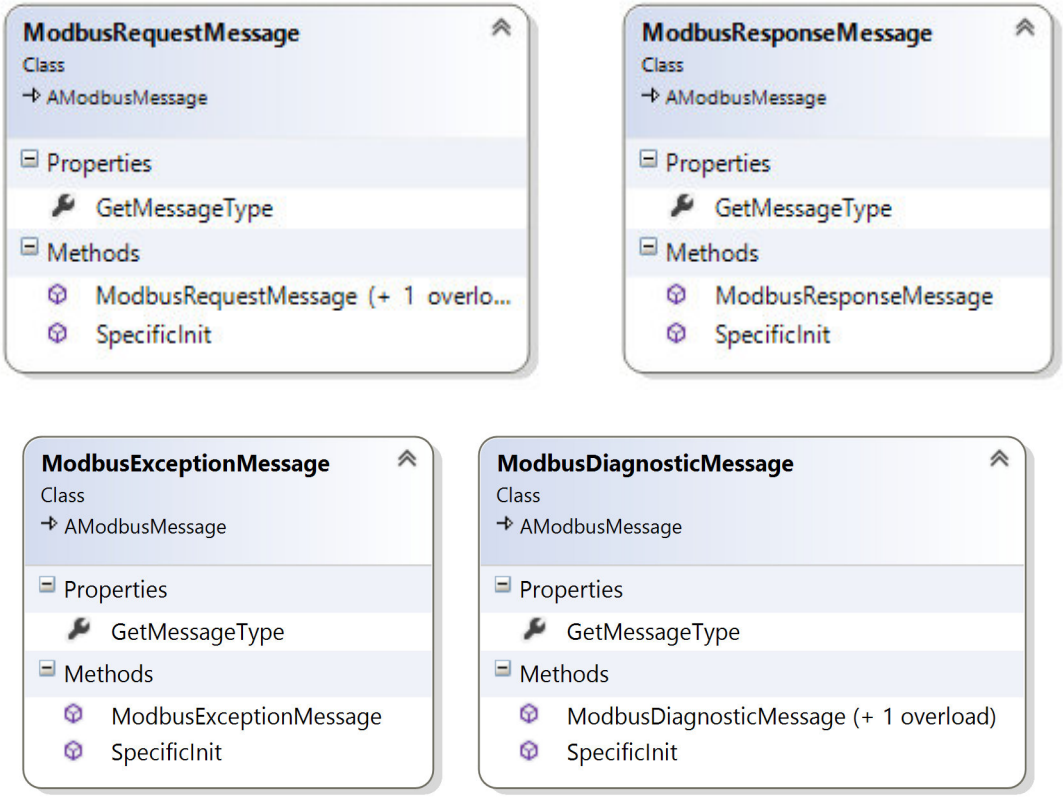

Figura 16: Constituição das classes dos diferentes tipos de mensagem.

3.1.2. Primeira Generalização

Após a implementação directa dos protocolos Modbus RTU e ASCII no iCenterX chegou-se à conclusão que um dos aspectos importantes na especificação de um protocolo trata-se da descrição dos vários tipos de mensagens que o mesmo contém. Isto é, para um determinado protocolo existe um certo número de tipos de mensagens que podem transitar entre dispositivos. Podemos ter mensagens de pedidos, mensagens de resposta, mensagens de erro, mensagens de diagnóstico, entre outras. Sendo que cada um destes tipos

obedece sempre à mesma forma de encapsulamento que lhe está associado, podendo haver até aspectos comuns ao encapsulamento dos vários tipos. Isto significa também que existe um número finito e concreto de propriedades que as mensagens podem incluir e que estas propriedades têm sempre as mesmas características (limites, tamanho, tipo, etc.) seja qual for o tipo de mensagem. Isto acontece, pois geralmente este tipo de protocolos pertence à camada 2 e/ou 7 do modelo OSI (neste caso, os protocolos Modbus apenas pertence à camada 7, a de aplicação)

Assim, para fazer esta primeira generalização decidiu-se abstrair e generalizar apenas as camadas 2 e 7 do modelo OSI. Optou-se então pela criação de um documento XML com a especificação de todas as propriedades que as mensagens podem ter, bem como as suas características e condições, e dos vários templates de mensagens correspondentes aos vários tipos de mensagens que o protocolo permite, assim como as suas regras de empacotamento e restrições para posterior validação, visto que a validação de uma mensagem está relacionada com as propriedades que esta inclui e com o tipo ou template da mensagem. No anexo C temos um exemplo de um ficheiro deste tipo com as especificações para os protocolos Modbus RTU e ASCII. E no anexo B temos um exemplo de um canal para o protocolo Modbus RTU que juntamente com o ficheiro do anexo C permite que o módulo recolha os dados indicados nesse mesmo canal.

Definiu-se também que, tendo em conta os protocolos já implementados, uma resposta estará sempre associada a um determinado pedido.

A definição dos vários tipos de mensagens e propriedades é realizada através da classe “MessagesConfiguration”, com o auxílio de um documento XML. Este documento terá todas as informações relativas às mensagens do protocolo, bem como a sua construção e identificação.

"MessagesConfiguration":

A classe “MessagesConfiguration” é constituída como apresentado na Figura 17:

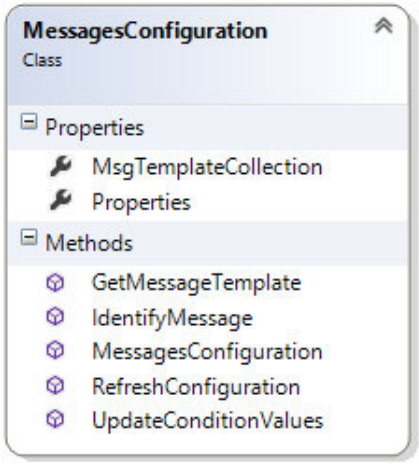

Figura 17: Constituição da Classe "MessagesConfiguration"

Este objecto é responsável pela leitura do documento XML com a especificação dos protocolos e contém duas propriedades. A primeira, ‘MsgTemplateCollection’ é um objecto de classe “MessageTemplateCollection” que contém todos os templates de mensagens definidos para o protocolo em questão no documento XML (“messages.mcf”) distribuídos por diferentes grupos: o grupo

“OutMessageTemplates” contém todos os templates de mensagens de saída, isto é, mensagens que serão enviadas pelo sistema; o grupo “InMessageTemplates” contém todos os templates de entrada, ou seja, as mensagens que serão recebidas pelo sistema. A segunda propriedade, ‘Properties’, contém todos os componentes que podem ser usados no protocolo, esses componentes são denominados “Property”. Podemos então ter, por exemplo no caso do protocolo Modbus ASCII, uma “Property” chamada ‘MessageStart’ com o valor ‘:’, outra com o nome ‘MessageEnd’ com o valor “/r/n”, etc. Estes componentes contêm ainda um ID, o tamanho (caso este seja fixo) e o tipo do valor (char, string, byte, etc.). Podem ainda ter condições de restrição, contudo isto será explicado mais detalhadamente na secção sobre a classe "Property".

A classe “MessagesConfiguration” contém ainda vários métodos associados. O método ‘GetMessageTemplate’ serve para obter um determinado template a partir do seu nome para o associar a uma mensagem, feito geralmente, aquando a leitura das variáveis do canal para criar um pedido inicial. O nome do template deve ser definido no documento XML do canal através do atributo “template-name” do elemento “variable”. Como por exemplo:

```
<variable id="ReadHoldingRegisters" template-name="GlobalRequest">
```

O método ‘RefreshConfiguration’ actualiza os valores das condições dos vários componentes de “Properties” e ”DataPackets”, uma vez que no processo de leitura não é possível saber o tipo da propriedade ou datapacket em questão. Este método deve ser sempre chamado após a criação de “MessagesConfiguration”.

O método ‘IdentifyMessage’ é usado para identificar um determinado array de bytes que tenha sido recebido. Este método tenta associar os dados recebidos a um template incluído no grupo “InMessageTemplates” e verificar a validade dos dados de acordo com o mesmo e com o pedido efectuado. Se o template for validado com sucesso significa que corresponde à mensagem recebida e é então devolvido por referência a mensagem de resposta já criada com todas as propriedades e com os valores provenientes do buffer de bytes que foi recebido. Caso o template não seja validado passa-se ao próximo até que algum seja validado. Se nenhum for validado é criada uma excepção com a mensagem “Couldn't identify message!”.

"MessageTemplateCollection":

A classe “MessageTemplateCollection” (Figura 18), como já foi referido, contém todos os templates de mensagens definidos para o protocolo em questão no documento XML (“messages.mcf”) distribuídos por diferentes grupos: o grupo “OutMessageTemplates” contém todos os templates de mensagens de saída, isto é, mensagens que serão enviadas pelo sistema; o grupo “InMessageTemplates” contém todos os templates de entrada, ou seja, as mensagens que serão recebidas pelo sistema. Contém ainda uma lista de “Frames” chamada ‘CommonFrames’. Esta é constituída por várias frames que são comuns a diferentes templates, como o próprio nome indica. Serve apenas para facilitar a escrita da descrição dos templates, uma vez que podemos definir estas frames comuns e depois associá-las a um determinado template através do nome.

Exemplo de utilização das ‘CommonFrames’:

```
<commonframes>
    <frame name="CommonHeader">
        <datapacket name="SlaveAddress" property-id="2" />
        <datapacket name="FunctionCode" property-id="3">
            <condition name="GlobalRequestFCCondition" affects="value" type="updown">
                <min>0</min>
                <max>127</max>
            </condition>
        </datapacket>
    </frame>
</commonframes>

<messagetemplate name="GlobalRequest" coding-type="byteEncoding">
    <frame name="CommonHeader" is-common="true" />
</messagetemplate>
```

Para definir uma frame como sendo comum é necessário colocar o nome igual ao da frame que se pretende e colocar o atributo “is-common” como verdadeiro.

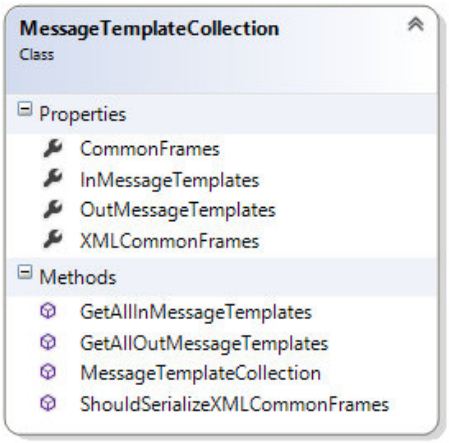

Figura 18: Constituição da Classe "MessageTemplateCollection"

Esta classe tem ainda dois métodos que servem apenas para devolver uma lista com todos os templates de mensagens de entrada ou saída conforme se use o método “GetAllInMessageTemplates” ou o método “GetAllOutMessageTemplates”.

"MessageTemplateGroup":

A classe “MessageTemplateGroup” (Figura 19) corresponde a um grupo de templates com a direcção em comum, isto é, ou correspondem todos a mensagens de saída ou a mensagens de entrada. Este grupo está então dividido em listas de templates globais, específicos ou de erro.

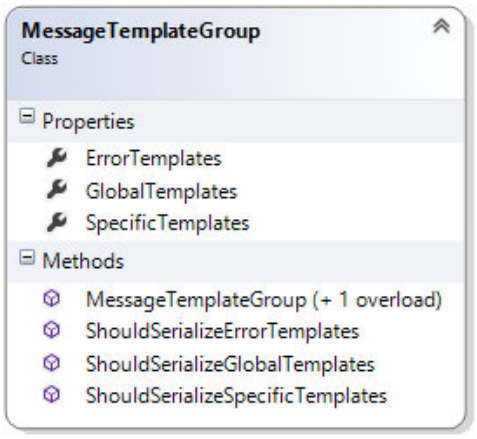

Figura 19: Constituição da Classe "MessageTemplateGroup"

"MessageTemplate"

A classe “MessageTemplate” (Figura 20) corresponde ao template de uma mensagem e contém uma lista com todas as frames da mensagem. Contém ainda os tipos de envio ou recepção da mensagem, bem como o tipo de codificação da mesma.

Os tipos de codificação idealizados para já são apenas a codificação em bytes (‘byteEncoding’) e a codificação em bytes ASCII (‘asciiEncoding’). Contudo a construção do código foi feita o mais possível de forma a ser compatível com outros tipos de codificação.

Um template é constituído da seguinte forma:

```
<messagetemplate>
    <frame>
        <datapacket />
        <datapacket />
        <datapacket />
        ...
    </frame>
    <frame>
        <datapacket />
        <datapacket />
        ...
    </frame>
    ...
</messagetemplate>
```

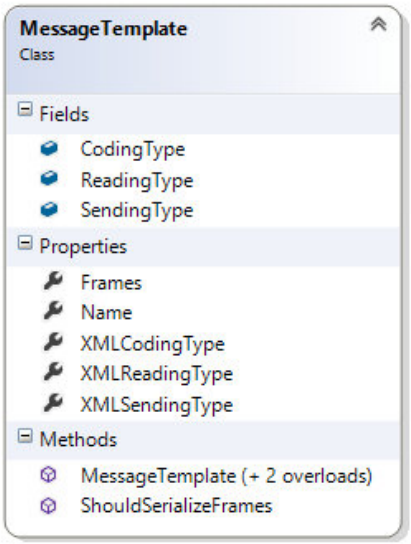

Figura 20: Constituição da Classe "MessageTemplate"

"Frame"

A classe “Frame” (Figura 21) contém uma lista de “DataPackets” chamada ‘Packets’ onde se encontram todas os componentes desta parte do template. Contém a propriedade ‘IsCommon’ anteriormente referida, bem como outras propriedades do tipo booleano que por enquanto ainda não estão a ser aplicadas mas poderão vir a sê-lo, como é o caso de ‘FixedSize’, ‘FrameSent’ e ‘WaitForAck’.

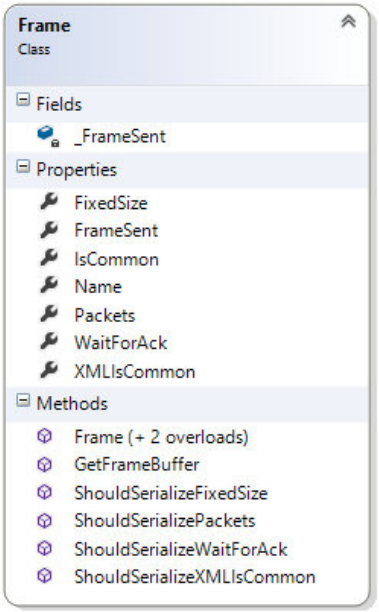

Figura 21: Constituição da Classe "Frame"

Esta classe tem ainda o método ‘GetFrameBuffer’ que serve para devolver um buffer de bytes com a frame toda codificada de acordo com o definido no template.

"DataPacket"

A classe “DataPacket” (Figura 22) contém uma string ‘PropertyID’ com o ID da “Property” à qual este “DataPacket” corresponde. Esta é a forma de definir quais as “Properties” que um determinado template contém de forma simples, sem estar a repetir novamente o que é definido em ‘<properties>’. Permite também a possibilidade de acrescentar restrições que servirão para validar os dados das mensagens. Estas restrições são as mesmas aplicadas às propriedades aquando a sua definição, mas têm a vantagem adicional de permitir estabelecer condições de comparação com outras propriedades da mesma mensagem ou da mensagem de pedido correspondente. A lista ‘PacketConditions’ contém precisamente estas restrições referidas. As propriedades ‘IsIdentifier’ e ‘Size’ não são usadas.

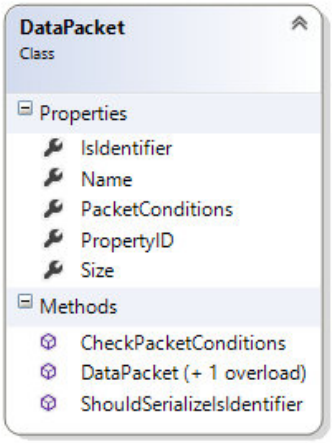

Figura 22: Constituição da Classe "DataPacket"

O método ‘CheckPacketConditions’ definido nesta classe serve para verificar a validade de um determinado valor tendo em conta as condições deste “DataPacket” em questão, devolvendo verdadeiro caso a validação tenha sucesso e falso caso contrário. Este método é normalmente usado aquando a identificação de um buffer de bytes recebido.

"Property"

Esta classe corresponde às propriedades que são definidos para as diferentes mensagens do protocolo. Contém o nome da propriedade, o ID, o tamanho (se for 0 ou não definido significa que o tamanho é variável), o tipo do valor e o valor da propriedade em si. As propriedades que tenham valores por defeito devem tê-los definidos nesta fase, caso o ‘Value’ não seja definido significa que a propriedade não tem valores por defeito (Figura 23).

A propriedade pode ainda ter um conjunto de condições de restrição definidas pela lista de “Conditions” chamada ‘Constraints’.

O ‘Delimiter’ é usado para definir qual o separador usado nos casos de valores que sejam arrays.

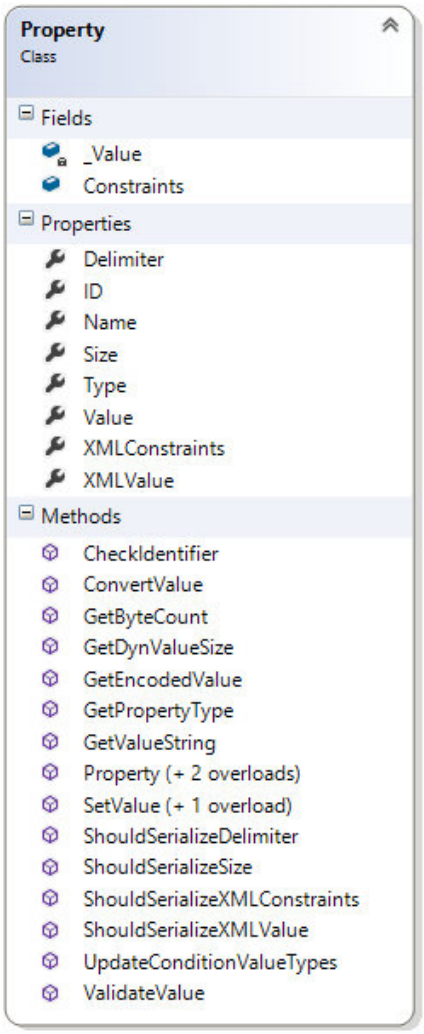

Figura 23: Constituição da Classe "Property"

Os métodos desta classe que estão actualmente a ser usados são:

- 'ConvertValue': converte o valor lido do ficheiro XML para o tipo definido em 'Type';
- 'GetByteCount': devolve a quantidade de bytes que o 'Value' tem. (Ex.: O tipo *ushort* tem 2 *bytes*);
- 'GetDynValueSize': devolve um inteiro com o tamanho de 'Value';
- 'GetEncodedValue': devolve o 'Value' codificado de acordo com um determinado *codingType*;
- 'GetPropertyType': devolve o "Type" da propriedade no formato *System.Type* em vez de *string*;
- 'SetValue': tenta colocar um determinado valor em 'Value' e devolve um booleano com o resultado;
- 'UpdateConditionValueTypes': actualiza os tipos dos valores das condições desta propriedade de acordo com o seu 'Type';
- 'ValidateValue': verifica se um determinado valor é valido de acordo com as condições da propriedade. (Este método é também usado no 'SetValue')

"Condition"

A classe "Condition" (Figura 24) representa uma restrição a ser aplicada ao valor ou tamanho da "Property" à qual está associada. Não esquecer que apesar de se poder atribuir uma condição a um

"DataPacket", este por sua vez está associado a uma "Property", logo essa condição será aplicada à propriedade associada a esse datapacket.

Uma condição começa por se definir a que é que se refere, isto é, se se refere ao valor da propriedade ou ao tamanho. Para tal usa-se o campo 'Affects', que pode ter os valores "value" ou "size". De seguida define-se o tipo de condição a implementar. Os tipos de condições existentes até ao momento são:

Tabela 1: Descrição dos tipos de condições que podem ser usados.

Tipo	Descrição
updown	Representa um limite superior e inferior
up	Representa um limite superior
down	Representa um limite inferior
list	Representa uma lista de valores ou tamanhos permitidos
exceptionlist	Representa uma lista de valores ou tamanhos não permitidos (esta condição ainda não foi implementada)
equalizer	Representa uma comparação com o valor de outra propriedade somada de uma constante

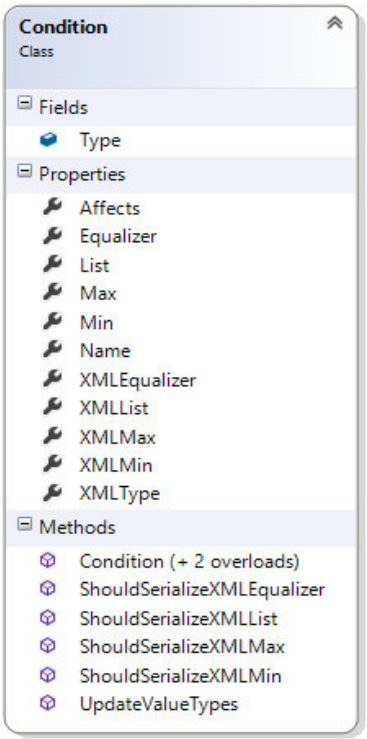

Figura 24: Constituição da Classe "Condition"

"Equalizer"

Esta classe representa uma condição de comparação entre o valor ou tamanho de uma propriedade e o valor de outra propriedade somada de uma constante. Para tal, esta classe tem duas propriedades a ter em consideração, o 'ReqPropID' que representa o ID de uma propriedade da mensagem de pedido associada a

esta mensagem e o 'RespPropID' que representa o ID de uma propriedade da mesma mensagem que a propriedade que tem esta condição. Apenas um destes Ids pode ser definido de cada vez, uma vez que se pretende comparar apenas com uma propriedade. Existe ainda a 'Constant' que é somada ao valor da propriedade com o ID definido.

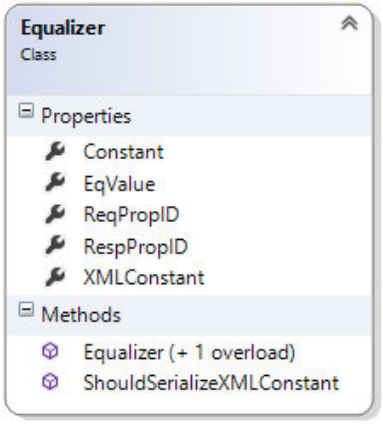

Figura 25: Constituição da Classe "Equalizer"

Funcionamento:

O valor ou o tamanho da "Property" à qual esta condição está associada deve ser:

- Se estiver definido 'ReqPropID':
  - Deve ser igual a: [(Valor da "Property" do pedido com ID = 'ReqPropID') + 'Constant']

- Se estiver definido 'RespPropID':
  - Deve ser igual a: [(Valor da "Property" da resposta com ID = 'RespPropID') + 'Constant']

Conforme a condição afecte o valor ou o tamanho (Propriedade "Affects" da "Condition").

Exemplo:

Se fizermos um pedido com as seguintes propriedades:

SlaveAddress (id=0) -> 5

FunctionCode (id=1) -> 1

InitialAddress (id=2) -> 0

Quantity (id=3) -> 2

Devemos obter uma resposta com:

SlaveAddress (id=0) -> valor igual a [(valor de ReqPropID=0) + 0]

FunctionCode (id=1) -> valor igual a [(valor de ReqPropID=1) + 0]

ByteCount (id=4) -> 2

Data (id=5) -> tamanho igual a [(valor de RespPropID=4) + 0]

Se tivéssemos uma resposta de erro teríamos:

SlaveAddress (id=0) -> valor igual a [(valor de ReqPropID=0) + 0]

ExceptionCode (id=6) -> valor igual a [(valor de ReqPropID=1) + 128]

Verificação de Erros:

Geralmente todos os protocolos utilizam algum método de verificação de erros, seja ele CRC16 ou CRC32. Assim foi criado um conjunto de classes (Figura 26) que permite generalizar o error check, sendo que este código permite usar os métodos mais conhecidos como CRC16-IBM-Normal, CRC16-IBM - Reversed, CRC16-CCITT - Normal, CRC16-CCITT – Reversed, CRC32-Normal, CRC32-Reversed, mas permite também escolher um modo “custom” que permite personalizar o error check.

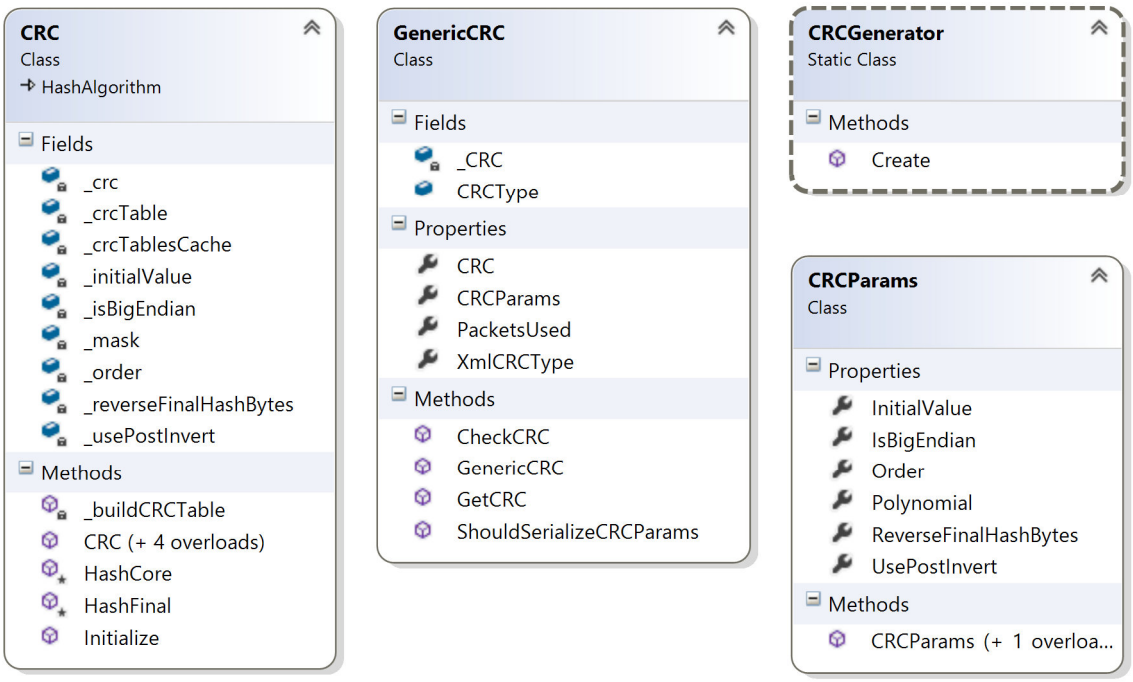

Figura 26: Constituição das classes de generalização do *error check*.

3.1.3. Implementação do Protocolo DNP3 na primeira Generalização

Terminada a primeira generalização chegou a altura de tentar usá-la para implementar outro protocolo, o escolhido foi o DNP3.

Como já foi referido o DNP3 permite que os dispositivos enviem informações não solicitadas quando algum dos valores ou estados sofre uma alteração [3]. Ora isto entra em conflito com uma das considerações que tínhamos feito na generalização, a que ditava que uma resposta estava sempre associada a um determinado pedido.

Outra das considerações que entra em conflito com o protocolo DNP3 foi a ideia de que as propriedades das mensagens têm sempre as mesmas características e que um determinado tipo de mensagem tem sempre o mesmo template. Tal não acontece no DNP3, por exemplo, uma das características que este protocolo tem é que as camadas têm limites máximos de tamanho e quando as mensagens que vêm do nível superior ultrapassam esse nível são dividas sendo que cada parte tem depois o acrescento do cabeçalho dessa camada [3] (Figura 27Figura 27: Fluxo de empacotamento do protocolo DNP3 [5].). Este controlo é realizado no DNP3 através de uma camada de pseudo-transporte, uma camada que implementa uma forma simplificada da 4ª camada do modelo OSI. Contudo na nossa primeira generalização não foi feita nenhuma abstracção desta camada.

Apesar de esta ser uma situação algo simples de ultrapassar, pois a descrição dos templates das mensagens na generalização já contempla os campos de tamanho máximo de cada parte da mensagem, faltando apenas alterar a implementação dessa característica no código de criação das mensagens, não conseguimos ultrapassar facilmente o facto de um cabeçalho de uma determinada camada para um determinado tipo de mensagem não ser sempre igual, pois no protocolo DNP3 o facto de as mensagens serem divididas de acordo com os tamanhos máximos requer alterações no cabeçalho de cada uma das divisões da mensagem inicial.

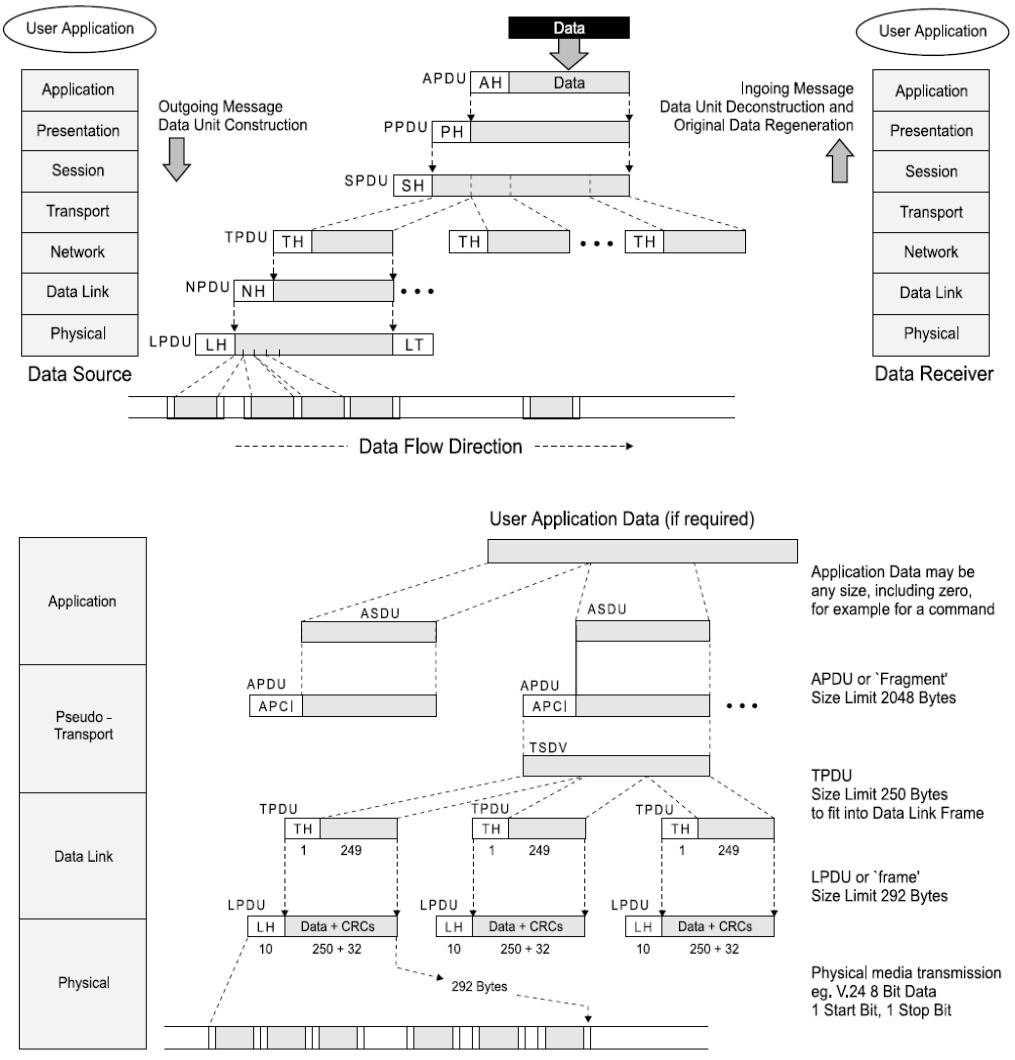

Figura 27: Fluxo de empacotamento do protocolo DNP3 [5].

Para além disso estas alterações não ocorrem apenas numa codificação de bytes, podendo haver alterações binárias, algo que não foi contemplado na primeira generalização (Figura 28).

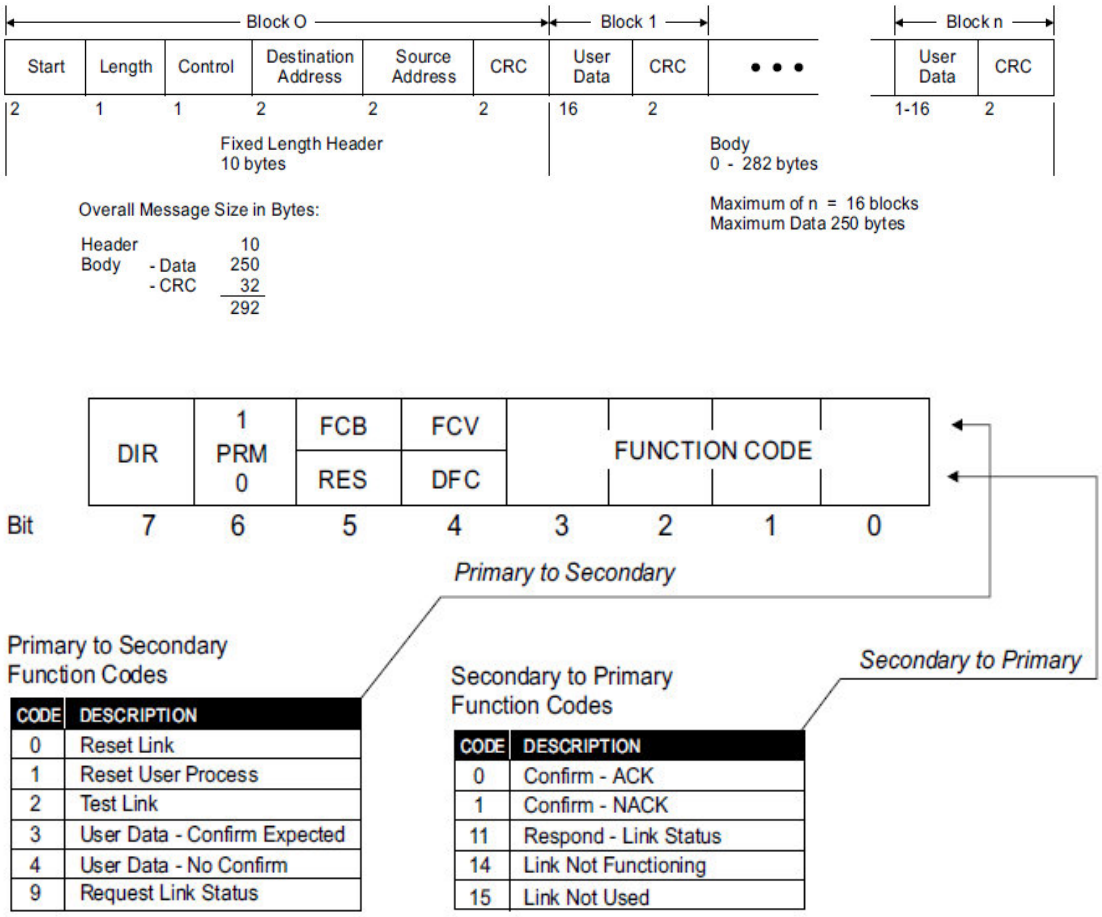

Figura 28: Exemplo do empacotamento de uma mensagem, bem como da codificação binária do byte de controlo[5].

Todos estes obstáculos aliados há limitação temporal não permitiram avanços desta fase para a frente, ficando a concretização do projecto por realizar.

Contudo, estes obstáculos permitiram trazer novas ideias para uma próxima generalização, que serão discutidas no próximo capítulo.

3.2. Arquitectura Proposta

Tendo em conta os obstáculos encontrados aquando a tentativa de implementação do protocolo DNP3, poderíamos enveredar por dois caminhos distintos.

Um deles passaria pela continuação da generalização existente melhorando a adaptação por forma a tornar compatível o DNP3 e outros protocolos de igual complexidade. O segundo seria tomar uma nova perspectiva, deixando de procurar aspectos comuns nos diferentes tipos de mensagens ou nos seus

empacotamento, passando a olhar com mais atenção para as diferentes camadas e para os seus comportamentos com as camadas adjacentes.

3.2.1. Continuação da Generalização

Para continuar a desenvolver a generalização existente, dever-se-ia começar por implementar de forma completa a restrição no tamanhos das mensagens e consequente divisão das mesmas, acrescentando novas características aos templates que indicassem o comportamento a tomar pelo mesmo quando as restrições fossem atingidas. Seria algo semelhante à classe "Conditions" que já existe e, por isso, deve ser possível de implementar.

Outra das situações a que é necessário dar a volta tem a ver com a possibilidade de codificação binária. Uma possível solução para este problema seria a criação de um novo tipo de dados (por exemplo "RawData <T>", onde T seria o verdadeiro tipo da propriedade, como byte ou ushort). Assim, todas as propriedades das mensagens teriam este tipo, que seria um tipo global que permitisse operações entre todas e quaisquer propriedades. Para além disso, seria necessário um outro tipo de dados ("Binary") que fosse utilizado pelas propriedades binárias, mas que também pudesse ser usado pelo "RawData<Binary>" para permitir operações entre todas as propriedades.

Convém referir que esta solução já foi começada a ser implementada, sendo apenas necessário refiná-la e testá-la para verificar se é mesmo alcançável (Figura 29).

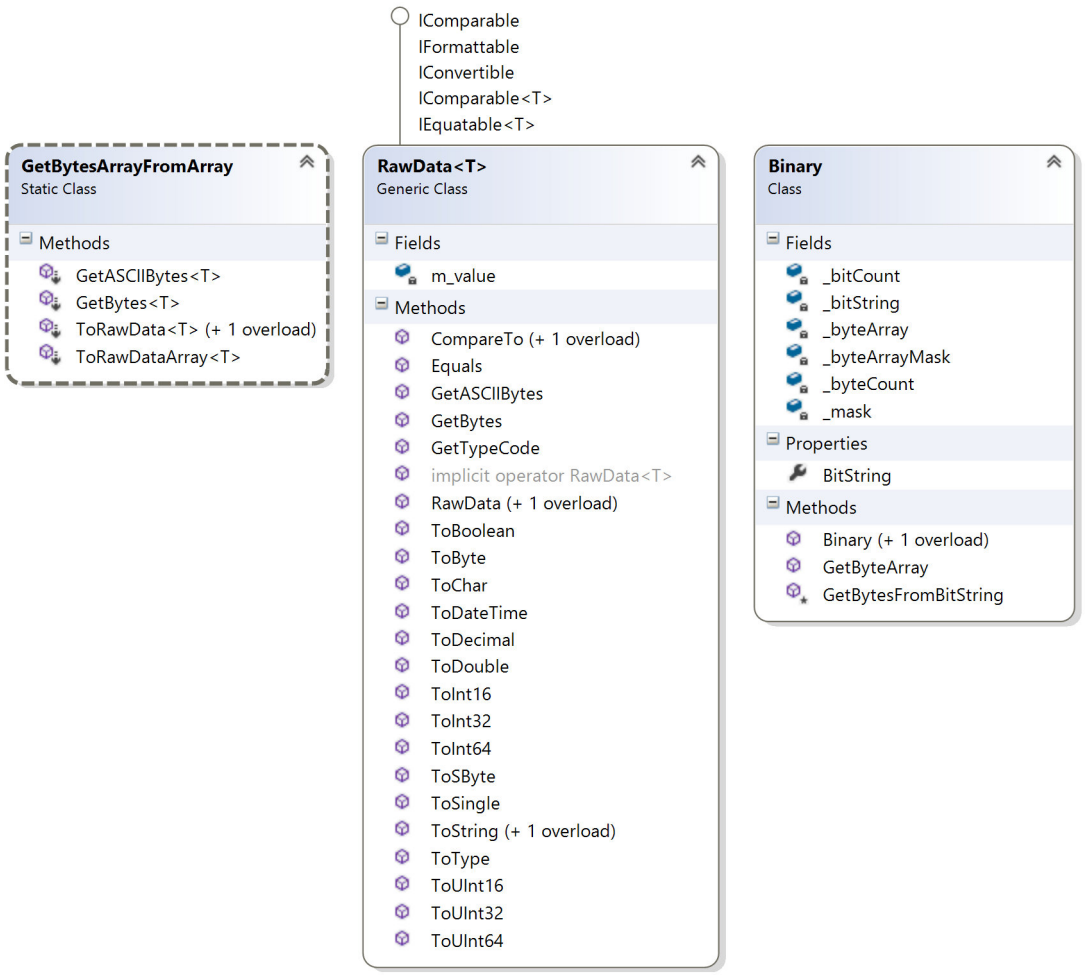

Figura 29: Constituição das classes "RawData" e "Binary", bem como outros métodos úteis.

Conseguindo ultrapassar estes obstáculos já seria mais fácil lidar com o facto da mutabilidade das mensagens do protocolo DNP3 ou outros de igual complexidade.

3.2.2. Nova Generalização

Esta nova aproximação consiste numa análise detalhada do comportamento de cada protocolo entre cada camada e criar um sistema de camadas com os diferentes métodos analisados nos vários protocolos, sendo então apenas necessário um ficheiro de configuração que indique quais os métodos a utilizar e de que forma.

Esta aproximação poderia ter uma perspectiva mais virada para eventos, isto é, poderia ser *event driven*. A ideia seria pensar em cada camada individualmente e definir eventos, como a chegada de uma mensagem da camada superior, que despoletassem os comportamentos definidos no ficheiro de configuração (Figura 30).

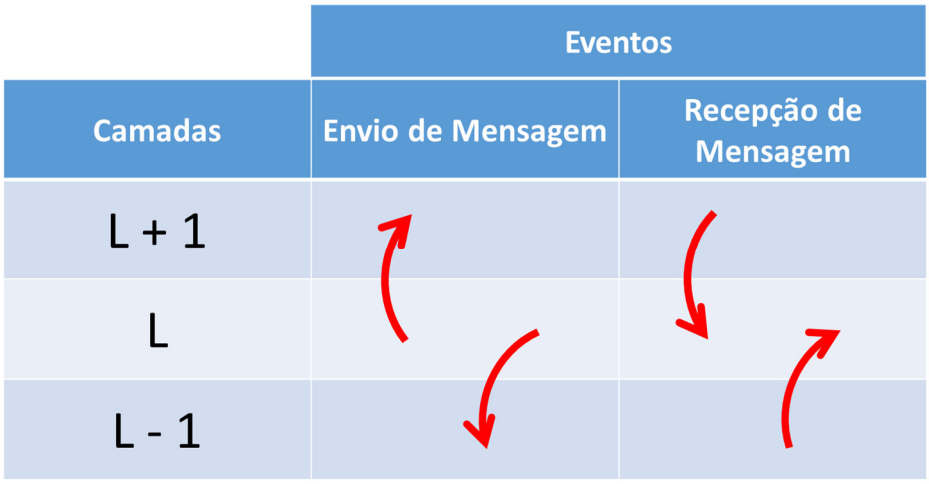

Figura 30: Generalização baseada em eventos.

Com esta aproximação pode-se abstrair e generalizar mais camadas do modelo OSI, podendo até fazer-se uma generalização de todas as camadas se necessário, aumentando-se assim o leque de protocolos que se poderia implementar.

4. Conclusões e Trabalho Futuro

4.1. Conclusões

Apesar do projecto não ter sido concretizado na sua plenitude devo afirmar que os resultados finais foram bastante positivos. Em primeiro lugar pelo grande salto que foi dado a nível de novos conhecimentos de programação, resultantes da interacção directa com o código fonte de um software completo e bastante complexo que é o iCenterX. Também os conhecimentos adquiridos relativamente aos processos de trabalho existentes numa empresa com a ISA são de louvar.

Outra questão que devo referir foi o facto de um projecto complexo como este me ter obrigado a explorar outras áreas diferentes do que estava habituado mesmo não estando directamente relacionadas com o projecto. Uma das situações foi a necessidade de não só aprender a linguagem de desenvolvimento do projecto em si (C#) mas também aprender as bases de uma outra linguagem de programação muito interessante, a C++. Tal foi necessário pois muitas das implementações de protocolos são feitas nessa linguagem e ao pesquisar deparei-me várias vezes com a mesma e optei por adquirir as bases necessárias para perceber aquilo com que me deparava.

Num ponto menos pessoal, continuo a achar que há coisas positivas a tirar deste projecto, pois foi dado um passo grande numa direcção pouco explorada que há partida já se sabia que iria ter vários obstáculos. O facto de termos ultrapassado vários desses obstáculos e encontrado outros de seguida permite deixar bases bastante fortes para uma continuação do projecto, bem como para um reinício do mesmo. Para além disso, julgo que apesar do módulo com a implementação que tem actualmente ainda não estar compatível com o protocolo DNP3, poderá já dar resultados com outros protocolos como Profibus ou Fieldbus. Será mais uma questão a verificar numa continuação do projecto.

4.2. Trabalho Futuro

Após ser concretizada uma generalização de protocolos satisfatória é necessários melhorar o controlo do fluxo do protocolo, que com a generalização actual está um pouco confuso, mas a generalização baseada em eventos referida em 3.2.2 é mais propícia para implementar uma máquina de estados de forma simples e completa.

Outro passo importante no futuro será a criação de uma interface gráfica para a especificação de protocolos de acordo com a generalização concretizada de forma simples e rápida.

Por último, dever-se-ia começar a estender o módulo a outros tipos de protocolos que não sejam baseados apenas em codificação por bytes para assim atingir plenamente a definição do próprio módulo.

Referências

[1] iCenterX – Especificação Técnica do Projecto, ISA

[2] Tanenbaum, Wetherall, “Computer networks – Fifth Edition”, 2011, [33]

[3] G. Clarke, D. Reynders, “Pratical Modern SCADA Protocols”, 2004, [78-142]

[4] “Modbus Application Protocol Specification V1.1b”, 28/12/2006, URL: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf

[5] G. Clarke, D. Reynders, “Pratical Modern SCADA Protocols”, 2004, [75, 79]

[6] Wikipédia – “High-Level Data Link Control”, URL: http://en.wikipedia.org/wiki/High-Level_Data_Link_Control [último acesso: 12/09/2013]

[7] “Modbus Application Protocol Specification V1.1b”, 28/12/2006, URL: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf

[8] R. Kumar, S. Nelvagal, S. Marcus – “A Discrete Event Systems Aproach for Protocol Conversion”, 1997, URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.6198&rep=rep1&type=pdf [último acesso: 30/08/2013]

[9] Wikipédia – “OSI Model”, URL: http://en.wikipedia.org/wiki/OSI_model [último acesso: 12/09/2013]

[10] “7 Layers of OSI”, URL: http://www.sis.pitt.edu/~icucart/networking_basics/7layersofOSI.htm [último acesso: 15/09/2013]

[11] Gregor, Bochmann – “Deriving Protocol Converters for Communications Gateways”, 1990, URL: http://site.uottawa.ca/~bochmann/ELG7187C/CourseNotes/Literature/Boch90xx%20-%20from%20IEEE%20Library%20-%20deriving%20protocol%20converters.pdf [último acesso: 05/09/2013]

[12] D. Brand, P. Zafiropulo – “On Communicating Finite-State Machines”, 1983, URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.478&rep=rep1&type=pdf [último acesso: 30/01/2013]

[13] Li Layuan – “A New Formal Method for Communication Protocol Specification”, 1989

Anexos

Anexo A – Canais Modbus

Anexo A.1 – Canal Modbus RTU para ioLogikR2110

```
<?xml version="1.0" encoding="utf-8"?>
<Channel id="ioLogikR2110_Test" usesSpecificController="true" specificController="1"
defaultUpdatePeriod="900000" defaultRequestSplitSpan="3600000">
    <driver type="modbus" acquisitionType="read" />
    <parameters>
        <parameter name="protocolType" value="rtu" />
        <parameter name="requestWriteRetries" value="5" /><!-- opcional -->
        <parameter name="responseReadRetries" value="3" /><!-- opcional -->
        <parameter name="rtuDelay" value="50" /><!-- opcional -->
        <parameter name="readTimeout" value="5" /><!-- opcional -->
        <parameter name="writeTimeout" value="5" /><!-- opcional -->
        <parameter name="portReadAttempts" value="3" /><!-- opcional -->
    </parameters>
    <connectionSettings>
        <serialConnectionSettings comName="COM5" baudRate="19200" parity="even"
dataBits="8" stopBits="one" handshake="None" />
    </connectionSettings>
    <device id="ioLogikR2110-1" lastDataTime="13-11-2012 12:00:00">
        <parameters>
            <parameter name="slaveAddress" value="5" />
        </parameters>
        <variables acquisitionPeriod="900000">
            <variable id="ReadCoils">
                <parameters>
                    <parameter name="functionCode" value="1" />
                    <parameter name="startAddress" value="0" />
                    <parameter name="quantity" value="8" />
                </parameters>
            </variable>
            <variable id="ReadHoldingRegisters">
                <parameters>
                    <parameter name="functionCode" value="3" />
                    <parameter name="startAddress" value="32" />
                    <parameter name="quantity" value="8" />
                </parameters>
            </variable>
        </variables>
    </device>
</Channel>
```

O código acima apresentado representa um exemplo de um canal para recolha de dados do dispositivo de entradas e saídas ioLogikR2110 da Moxa usando o protocolo Modbus RTU. Este canal foi criado aquando a implementação dos protocolos Modbus RTU e ASCII no sistema iCenterX existente. Serviu para testar a implementação do protocolo Modbus RTU. Neste canal são definidas as características do protocolo nos parâmetros do '*Channel*' e são definidos parâmetros necessários para efectuar as leituras das variáveis pretendidas.

Anexo A.2 – Canal Modbus ASCII para SIGMA950

```
<?xml version="1.0" encoding="utf-8"?>
<Channel id="SIGMA950_Test" usesSpecificController="true" specificController="1"
defaultUpdatePeriod="900000" defaultRequestSplitSpan="3600000">
    <driver type="modbus" acquisitionType="read" />
    <parameters>
        <parameter name="protocolType" value="ascii" />
        <parameter name="requestWriteRetries" value="5" /><!-- opcional -->
        <parameter name="responseReadRetries" value="3" /><!-- opcional -->
        <parameter name="readTimeout" value="3" /><!-- opcional -->
        <parameter name="writeTimeout" value="5" /><!-- opcional -->
        <parameter name="portReadAttempts" value="10" /><!-- opcional -->
    </parameters>
    <connectionSettings>
        <serialConnectionSettings comName="COM4" baudRate="9600" parity="even"
dataBits="7" stopBits="one" handshake="None" />
    </connectionSettings>
    <device id="Sigma950-1" lastDataTime="13-11-2012 12:00:00">
        <parameters>
            <parameter name="slaveAddress" value="1" />
        </parameters>
        <variables acquisitionPeriod="900000">
            <variable id="ReadPower">
                <parameters>
                    <parameter name="functionCode" value="3" />
                    <parameter name="startAddress" value="38" />
                    <parameter name="quantity" value="2" />
                </parameters>
            </variable>
            <variable id="ReadTemperatureUnit">
                <parameters>
                    <parameter name="functionCode" value="3" />
                    <parameter name="startAddress" value="49" />
                    <parameter name="quantity" value="1" />
                </parameters>
            </variable>
            <variable id="ReadWrongAddress">
                <parameters>
                    <parameter name="functionCode" value="1" />
                    <parameter name="startAddress" value="0" />
                    <parameter name="quantity" value="2" />
                </parameters>
            </variable>
        </variables>
    </device>
</Channel>
```

O código acima apresentado representa um exemplo de um canal para recolha de dados do medidor de fluxo SIGMA 950 da Hach usando o protocolo Modbus ASCII. Este canal foi criado aquando a implementação dos protocolos Modbus RTU e ASCII no sistema iCenterX existente. Serviu para testar a implementação do protocolo Modbus ASCII. Neste canal são definidas as características do protocolo nos parâmetros do '*Channel*' e são definidos parâmetros necessários para efectuar as leituras das variáveis pretendidas.

Anexo B – Canal Genérico

```
<?xml version="1.0" encoding="utf-8"?>
<Channel id="GenericProtocolChannel" defaultUpdatePeriod="900000"
defaultRequestSplitSpan="3600000">
    <driver type="modbus" acquisitionType="read" />
    <device id="ioLogikR2110-1" lastDataTime="13-11-2012 12:00:00">
        <variables acquisitionPeriod="900000">
            <variable id="ReadCoils" template-name="GlobalASCIIRequest">
                <parameters>
                    <parameter name="SlaveAddress" value="5" />
                    <parameter name="FunctionCode" value="1" />
                    <parameter name="InitialAddress" value="0" />
                    <parameter name="Quantity" value="8" />
                </parameters>
            </variable>
            <variable id="ReadHoldingRegisters" template-name="GlobalRequest">
                <parameters>
                    <parameter name="SlaveAddress" value="5" />
                    <parameter name="FunctionCode" value="3" />
                    <parameter name="InitialAddress" value="32" />
                    <parameter name="Quantity" value="8" />
                </parameters>
            </variable>
        </variables>
    </device>
</Channel>
```

O código acima apresentado representa um exemplo de um canal para recolha de dados do dispositivo de entradas e saídas ioLogikR2110 da Moxa usando o protocolo generalizado. Este canal foi criado aquando a implementação da primeira generalização no sistema iCenterX existente. Serviu para testar a implementação do protocolo Modbus RTU através do protocolo genérico e do ficheiro de configuração "MessagesConfiguration.mcf" apresentado no anexo C. Neste canal são definidas as características das variáveis que se pretende ler bem como as mensagens usadas para a leitura das mesmas.

Anexo C – Messages Configuration File

```
<?xml version="1.0" encoding="utf-8"?>
<MessagesConfiguration>
    <properties>
        <property id="0" name="MessageStart" type="char" size="1" value=":" />
        <property id="1" name="MessageEnd" type="string" size="2" value="\r\n" />
        <property id="2" name="SlaveAddress" type="byte" size="1">
            <constraints>
                <!-- condition: Represents an upper and/or down limit to size or value or represents
a list of permited sizes or values -->
                <!-- type: "up/down", "up", "down", "list", "exceptionlist" -->
                <condition name="SlaveAddressCondition" affects="value" type="updown">
                    <min>0</min>
                    <max>242</max>
                </condition>
                <condition name="SlaveAddressCondition1" affects="value" type="updown">
                    <min>0</min>
                    <max>242</max>
                </condition>
            </constraints>
        </property>
        <property id="3" name="FunctionCode" type="byte" size="1">
            <constraints>
                <!-- condition: Represents an upper and/or down limit to size or value or represents
a list of permited sizes or values -->
                <condition name="FunctionCodeCondition" affects="value" type="list">
                    <listitem>1</listitem>
                    <listitem>2</listitem>
                    <listitem>3</listitem>
                    <listitem>4</listitem>
                    <listitem>8</listitem>
                </condition>
            </constraints>
        </property>
        <property id="4" name="InitialAddress" type="ushort" size="1" />
        <property id="5" name="Quantity" type="ushort" size="1" />
        <property id="6" name="ByteCount" type="byte" size="1" />
        <property id="7" name="Data" type="byte[]" delimiter="," />
        <property id="8" name="ExceptionCode" type="byte" size="1">
            <constraints>
                <!-- condition: Represents an upper and/or down limit to size or value or represents
a list of permited sizes or values -->
                <condition name="ExceptionCodeCondition" affects="value" type="list">
                    <listitem>129</listitem>
                    <listitem>130</listitem>
                    <listitem>131</listitem>
                    <listitem>132</listitem>
                    <listitem>136</listitem>
                </condition>
            </constraints>
        </property>
        <property id="9" name="e" type="byte[]" size="5" value="1,255,3,4,5" delimiter="," />
        <property id="10" name="CRC" type="CRCTypes" value="CRC16_IBM_reversed"  />
        <property id="11" name="CRC16" type="byte[]" size="2" delimiter="," is-check="true" check-
type="crc">
            <crc type="CRC16_IBM_reversed" packets-used="2,3,4,5" />
        </property>
        <property id="12" name="ReceivedCRC16" type="byte[]" size="2" delimiter="," is-check="true"
check-type="crc">
            <crc type="CRC16_IBM_reversed" packets-used="2,3,6,7,12" />
        </property>
    </properties>
    <messagetemplates>
        <commonframes>
            <frame name="CommonHeader" fixed-size="true">
                <datapacket name="SlaveAddress" property-id="2"></datapacket>
                <datapacket name="FunctionCode" property-id="3" is-identifier="true">
                    <condition name="GlobalRequestFCCondition" affects="value" type="updown">
                        <min>0</min>
                        <max>127</max>
                    </condition>
                </datapacket>
            </frame>
```

```
        </commonframes>
        <outmsg>
            <global>
                <messagetemplate name="GlobalRequest" coding-type="byteEncoding">
                    <frame name="CommonHeader" is-common="true" />
                    <frame name="Body" fixed-size="true">
                        <datapacket name="InitialAddress" property-id="4" />
                        <datapacket name="Quantity" property-id="5" />
                    </frame>
                    <frame name="Footer" fixed-size="true">
                        <datapacket name="CRC16" property-id="11" />
                    </frame>
                </messagetemplate>
                <messagetemplate name="GlobalASCIIRequest" coding-type="asciiEncoding">
                    <frame name="Header" fixed-size="true">
                        <datapacket name="MessageStart" property-id="0" />
                        <datapacket name="SlaveAddress" property-id="2" />
                        <datapacket name="FunctionCode" property-id="3" is-identifier="true">
                            <condition name="GlobalRequestFCCondition" affects="value" type="updown">
                                <min>0</min>
                                <max>127</max>
                            </condition>
                        </datapacket>
                    </frame>
                    <frame name="Body" fixed-size="true">
                        <datapacket name="InitialAddress" property-id="4" />
                        <datapacket name="Quantity" property-id="5" />
                    </frame>
                    <frame name="Footer" fixed-size="true">
                        <datapacket name="CRC16" property-id="11" />
                        <datapacket name="MessageEnd" property-id="1" />
                    </frame>
                </messagetemplate>
            </global>
            <specific>
            </specific>
            <error>
            </error>
        </outmsg>
        <inmsg>
            <global>
                <messagetemplate name="GlobalResponse" coding-type="byteEncoding">
                    <frame name="Header" wait-for-ack="false" fixed-size="true">
                        <datapacket name="SlaveAddress" property-id="2">
                            <condition name="GlobalResponse_SlaveAddress_Condition" affects="value"
type="equalizer">
                                <equalizer request-property-id="2" constant="0" />
                            </condition>
                        </datapacket>
                        <datapacket name="FunctionCode" property-id="3" is-identifier="true">
                            <condition name="GlobalResponseFCCondition" affects="value"
type="updown">
                                <min>0</min>
                                <max>127</max>
                            </condition>
                        </datapacket>
                    </frame>
                    <frame name="Body" wait-for-ack="false" fixed-size="false">
                        <datapacket name="ByteCount" property-id="6" />
                        <datapacket name="Data" property-id="7">
                            <condition name="GlobalResponseDataSizeCondition" affects="size"
type="equalizer">
                                <!-- Tamanho de "Data" tem de ser igual ao valor de "ByteCount" -->
                                <equalizer response-property-id="6" constant="0" />
                            </condition>
                        </datapacket>
                    </frame>
                    <frame name="Footer" wait-for-ack="false" fixed-size="true">
                    </frame>
                </messagetemplate>
                <messagetemplate name="GlobalASCIIResponse" coding-type="asciiEncoding">
                    <frame name="Header" wait-for-ack="false" fixed-size="true">
                        <datapacket name="MessageStart" property-id="0" />
                        <datapacket name="SlaveAddress" property-id="2">
                            <condition name="GlobalResponse_SlaveAddress_Condition" affects="value"
type="equalizer">
                                <equalizer request-property-id="2" constant="0" />
```

```
                        </condition>
                    </datapacket>
                    <datapacket name="FunctionCode" property-id="3" is-identifier="true">
                        <condition name="GlobalResponseFCCondition" affects="value"
type="updown">
                            <min>0</min>
                            <max>127</max>
                        </condition>
                    </datapacket>
                </frame>
                <frame name="Body" wait-for-ack="false" fixed-size="false">
                    <datapacket name="ByteCount" property-id="6" />
                    <datapacket name="Data" property-id="7">
                        <condition name="GlobalResponseDataSizeCondition" affects="size"
type="equalizer">
                            <!-- Tamanho de "Data" tem de ser igual ao valor de "ByteCount" -->
                            <equalizer response-property-id="6" constant="0" />
                        </condition>
                    </datapacket>
                </frame>
                <frame name="Footer" wait-for-ack="false" fixed-size="true">
                    <datapacket name="ReceivedCRC16" property-id="12" />
                    <datapacket name="MessageEnd" property-id="1" />
                </frame>
            </messagetemplate>
        </global>
        <specific>
        </specific>
        <error>
            <messagetemplate name="ExceptionResponse" coding-type="byteEncoding" sending-
type="block">
                <frame name="Header" wait-for-ack="false" fixed-size="true">
                    <datapacket name="SlaveAddress" property-id="2">
                        <condition name="ExceptionResponse1_SlaveAddress_Condition"
affects="value" type="equalizer">
                            <equalizer request-property-id="2" constant="0" />
                        </condition>
                    </datapacket>
                    <datapacket name="ExceptionCode" property-id="8" is-identifier="true">
                        <condition name="ExceptionResponse1_ExceptionCode_Condition"
affects="value" type="equalizer">
                            <!--
                                Neste caso terá que ser igual ao valor da propriedade c/ ID="3" do
pedido mais 128
                                Ex.: ExceptionCode == FunctionCode + 128 ???
                            -->
                            <equalizer request-property-id="3" constant="128" />
                        </condition>
                    </datapacket>
                </frame>
            </messagetemplate>
        </error>
    </inmsg>
  </messagetemplates>
</MessagesConfiguration>
```

O código acima apresentado representa um exemplo de um ficheiro de configuração que contém a especificação de todas as propriedades que as mensagens podem ter, bem como as suas características e condições, e dos vários templates de mensagens correspondentes aos vários tipos de mensagens que os protocolos Modbus RTU e ASCII permitem, assim como as suas regras de empacotamento e restrições para posterior validação. Este ficheiro corresponde à primeira generalização feita para estes dois protocolos.

Este ficheiro é lido pelo MCA para que este possa implementar o protocolo pretendido e, também, para permitir a leitura do canal, como por exemplo o canal do anexo B, e construir as mensagens necessárias para recolher os dados desejados.