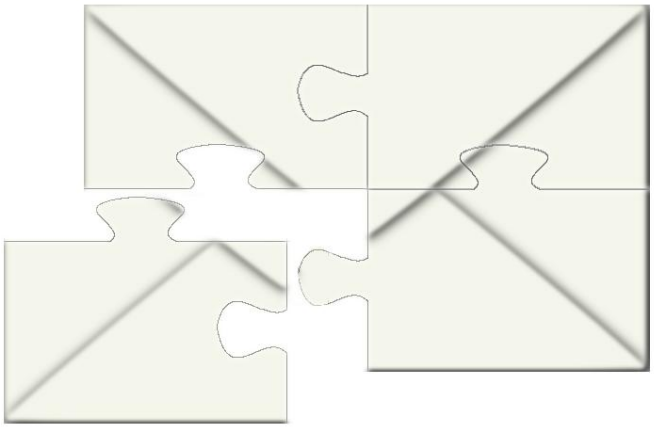
Instituto Superior de Engenharia de Lisboa

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Bus de mensagens



José Carlos Martins Fernandes

(Licenciado)

Trabalho de projecto realizado para a obtenção do grau de Mestre em Engenharia Informática e de Computadores

Júri:

Presidente: Professor-adjunto Pedro Alexandre Seia e Cunha Pereira, ISEL

Vogal (arguente): Mestre Luís Assunção, ISEL

Vogal (orientador): Professor-adjunto José Luís Falcão Cascalheira, ISEL

Vogal (orientador): Professor-adjunto Fernando Miguel Carvalho, ISEL

Novembro de 2010

Resumo

Este trabalho foca a comunicação entre aplicações, em especial o caso em que estas são tecnologicamente diferentes entre si. Pretende-se uma forma de as aplicações trocarem informação em segurança, abstraindo-se das suas diferenças e localização física. Para tal, é necessário um meio transversal às tecnologias/plataformas, capaz de esconder as especificidades de cada interveniente e tornar a comunicação transparente entre os seus interlocutores. Um *bus* de mensagens apresenta-se neste cenário como o meio de alcançar tais necessidades.

O *bus* de mensagens desenvolvido neste projecto dispõe de mecanismos de tolerância a falhas, encaminhamento, transformação e segurança. O encaminhamento suporta comunicação ponto–a–ponto e publicador–subscritor. A transformação de mensagens pode ser feita ao nível dos tipos de dados, do formato e do transporte. Relativamente à segurança, é controlado o acesso a cada aplicação e protegida a informação trocada entre clientes. A solução implementada apresenta ainda vários pontos de extensibilidade ao nível das funcionalidades, dos comandos e dos protocolos de comunicação com os clientes.

O *bus* de mensagens implementado foi testado e avaliado em diferentes cenários de carga, que verificam a conformidade das funcionalidades desenvolvidas e permitiram medir o seu nível de serviço.

Palavras-chave: Integração, *middleware*, *messaging*, modelos e protocolos de comunicação.

Abstract

This work focuses on communication between applications, especially in the case that they are technologically different. The aim is a way for applications to exchange information securely, in abstraction from their differences and physical location. To do this, you need a cross through the technologies / platforms, capable of hiding the specifics of each actor and make transparent communication between the interlocutors. A message *bus* is presented here as a means of achieving those needs.

The message *bus* developed in this project has mechanisms for fault tolerance, routing, transformation and security. The routing supports point-to-point and publish-subscribe communications. The message transformation can be made at the level of the data types, format and transport. Regarding security, is access to every application and secure the information exchanged between customers. The implemented solution provides several extensibility points regarding the features, commands and protocols used to communicate with clients.

The message *bus* implemented was tested and evaluated under different load scenarios that verify compliance of the features developed and allowed to measure their level of service.

Keywords: Integration, middleware, messaging, models and communication protocols

Conteúdo

Índice de figuras

Índice de listagens

Capítulo I

Introdução

Este capítulo começa por apresentar os pontos que motivam a realização deste trabalho (secção 1.1), sendo também enumerados os objectivos gerais pretendidos (secção 1.2). O capítulo inclui ainda secções que orientam o leitor ao longo do documento. São essas secções a "Organização do documento" (secção 1.3) e a lista de acrónimos utilizados (secção 1.4).

1.1 Motivação

Com o surgimento das primeiras aplicações informáticas muitas empresas viram nessa área uma oportunidade de melhorar os seus rendimentos e produtividade. Durante décadas, empresas compraram ou desenvolveram elas próprias aplicações para dar suporte aos seus negócios e informatizar informação que até então era mantida em suporte de papel. Cada aplicação foi desenvolvida a pensar num fim específico e, em muitos dos casos, sem se ter em conta a hipótese de estas aplicações necessitarem de vir a comunicar com outras. Este tipo de desenvolvimento deu origem a que uma empresa possua no seu interior um grande número de sistemas a funcionar de forma independente. Um exemplo típico é o cenário onde cada departamento tem um sistema independente como, por exemplo, um sistema de automatização de vendas, de recursos humanos, de registo de utilizadores, etc. Cada um destes sistemas foi construído utilizando a tecnologia mais comum à data do desenvolvimento e muitos deles utilizando formatos de dados não *standard*.

Com o avançar do tempo, as necessidades das empresas mudam e chega-se a um cenário onde a partilha de informação assume um papel cada vez mais importante. Cada sistema a funcionar de forma isolada passa a necessitar de comunicar com outros sistemas. Tecnologicamente existem diversas formas de transferir (ou partilhar) informação entre sistemas, contudo nem sempre estas tecnologias respondem a necessidades de comunicação como, por exemplo, garantia de fiabilidade ou desacoplamento temporal. Um *bus* de mensagens surge neste contexto como o meio que permite a vários sistemas comunicarem entre si através do envio e recepção de mensagens. O *bus* de mensagens trata-se de uma ferramenta encarregue de entregar cada mensagem no destino certo e capaz de lidar com as diferenças existentes entre cada sistema.

Apesar da grande utilidade como meio de partilha de informação entre sistemas que não foram inicialmente criados para partilhar informação (ou seja, integração de sistemas), a utilização dos sistemas de mensagens não se limita a estes casos. O seu comportamento genérico e a capacidade de adaptação fazem com que sejam adequados para outros cenários onde duas ou mais partes necessitem de trocar dados.

1.2 Objectivos gerais

O objectivo deste trabalho consiste em aprofundar os conceitos relacionados com um *bus* de mensagens, realizando um estudo das principais características e serviços oferecidos por este tipo de aplicações.

Um aspecto com particular relevância nos cenários de aplicações que comunicam por mensagens, e que também será objecto de estudo neste trabalho, são os protocolos utilizados entre o *bus* de mensagens e as aplicações cliente. Estes protocolos estabelecem pré-acordos na forma como as aplicações e o *bus* comunicam entre si. Desta forma, são a chave para que o *bus* de mensagens possa lidar com as diferenças de cada aplicação de modo não intrusivo, isto é, minimizando as alterações necessárias em cada aplicação.

O trabalho prossegue com a implementação de um *bus* de mensagens. Pretende-se que a solução final apresente uma arquitectura robusta e tolerante a falhas, utilizando na sua implementação tecnologias recentes.

1.3 Organização do documento

Este documento está organizado em seis capítulos de acordo com o seguinte:

1. Introdução – Constitui o presente capítulo. Contém a motivação para o desenvolvimento deste trabalho, os objectivos gerais e a lista de acrónimos utilizados.
2. Enquadramento – Trata-se do capítulo onde é feito um estudo em volta do tema do trabalho, englobando conceitos essenciais para a compreensão da solução final.
3. Arquitectura – Contém a arquitectura geral da solução proposta, fazendo uma primeira aproximação na descrição de tecnologias utilizadas.
4. Implementação – Com a descrição dos principais pontos de desenvolvimento. Acompanhado por diagramas de classes e listagens de código sempre que necessário.
5. Resultados – Apresenta os resultados obtidos com a realização do trabalho.
6. Discussão e conclusões – Neste capítulo são apresentadas as conclusões finais seguidas de uma análise crítica. Contém ainda a orientação daquilo que poderá ser o trabalho futuro.

Para além dos seis capítulos, existem ainda três anexos cujo conteúdo será apresentado oportunamente ao longo do documento.

1.4 Lista de acrónimos

Ao longo deste documento serão utilizadas os seguintes acrónimos com o respectivo significado:

AMQP	Advanced Message Queuing Protocol
API	Application Program Interface
DTC	Distributed Transaction Coordinator
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JMS	Java Message Service

JNDI	Java Naming and Directory Interface
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
MOM	Message Oriented Middleware
MSMQ	Microsoft Message Queue
REST	Representational State Transfer
RPC	Remote Procedure Call
TCP	Transport Control Protocol
UDP	User Datagram Protocol
URI	Universal Resource Indicator
WCF	Windows Communication Foundation
XML	Extensible Markup Language
XSLT	XML Stylesheet Language for Transformations

Capítulo II

Enquadramento

Durante décadas, a integração tem sido uma importante necessidade no seio das organizações. A rápida evolução das aplicações e a multiplicidade de tecnologias dificultam o trabalho de integração. O processo de integração de vários sistemas consiste fundamentalmente em duas fases – partilha de informação e partilha de funcionalidades. No contexto de integração, o *middleware* desempenha um papel importante por permitir o fluxo de informação.

Este capítulo inicia-se com uma breve definição do que é a integração (secção 2.1), de que formas pode ser feita (secção 2.1.2) e as características frequentemente encontradas nas aplicações intervenientes (secção 2.1.1). De seguida é apresentado o conceito de *middleware*, focando os vários tipos de *middleware* (secção 2.2.1) existentes e dando especial atenção às características de *middleware* orientado a mensagens. Será feita uma análise das principais diferenças entre ligar duas aplicações directamente ou utilizar um intermediário, como um *bus* de mensagens, para essa ligação (secção 2.5). O capítulo termina com a apresentação de *standards* utilizados na comunicação através de mensagens (secção 2.7.2 e 2.7.3).

2.1 Integração de sistemas

Do ponto de vista histórico, a integração começou por ser um conceito aplicado ao *hardware*. Integrar significava juntar diferentes componentes de *hardware*, para que juntos fornecessem o suporte para uma aplicação.

À medida que o *hardware* evoluiu, a complexidade das aplicações aumentou e a natureza da integração alterou-se. A integração passou então a ser vista como a combinação de *hardware* e *software* que juntos formam um sistema. Actualmente o conceito da integração é dominado pelo *software*. As organizações estão empenhadas em integrar novas aplicações com outras que já utilizavam [RB01]. Hoje em dia, do ponto de vista tecnológico, define-se integração como sendo a fusão numa só arquitectura uniforme entre tecnologias, de dados e aplicações muitas vezes incompatíveis e divergentes [Mye02]. A integração permite que informação e conhecimento sejam simultaneamente partilhados por funcionários de uma empresa, por parceiros de negócio ou até por empresas concorrentes. Permite ainda que várias partes trabalhem simultaneamente na resolução de um problema ou projecto independentemente da sua localização, fuso horário ou da localização da informação [Mye02].

Todas as empresas possuem as suas próprias regras de negócio, isto é, regras que definem o modo como a empresa funciona a nível interno. Uma dessas regras pode ser, por exemplo, no caso de uma instituição bancária, a definição do processo de um pedido de empréstimo. Nesse processo é definido que os dados do empréstimo devem ser inseridos numa base de dados específica para empréstimos e também junto dos restantes dados da conta do cliente. Poderia ainda ser indicado que após a inserção dos dados se verifica se o cliente tem, ou não, dívidas ao banco assim como outras operações como a definição do limite máximo de crédito a conceder àquele cliente.

A necessidade de integração dentro de uma empresa surge pela necessidade de partilhar as regras de negócio da empresa entre várias aplicações. Se essas regras forem especificadas apenas num local e então partilhadas sob a forma de funcionalidades ou serviços a todas as aplicações que delas precisem, torna-se mais simples e eficaz fazer a gestão dessas regras. [RB01]

Para além da partilha de regras de negócio existe ainda a necessidade da partilha de dados acedidos, criados ou modificados pelas aplicações. Permitir o fluxo de informação dentro de uma organização é fundamental para evitar a replicação de dados e permitir às aplicações acederem à informação mais actual [Lin00].

Existem também alguns tipos de aplicações que constituem motivo de atenção especial num processo de integração, pelo facto de não privilegiarem a comunicação

com o exterior. A este tipo de aplicações dá-se o nome de *stovepipes* e a razão de serem aplicações de difícil integração é o facto de terem sido desenvolvidas pensando apenas num fim muito específico e desempenhado de forma isolada [RB01]. Durante o seu desenvolvimento não se teve em conta uma arquitectura geral que abrangesse todas as aplicações existentes no seio de uma organização e os mecanismos de comunicação entre aplicações não foram devidamente considerados.

2.1.1 Tipos de *Stovepipes*

Atendendo a diferentes características de aplicações *stovepipe*, é possível agrupa-las em diferentes tipos. Dos vários tipos de *stovepipe*, destacam-se três por serem aqueles que têm maior importância no contexto deste trabalho e por constituírem possíveis aplicações clientes do *bus* de mensagens desenvolvido. São esses tipos as *aplicações legadas*, *aplicações desktop* e os *pacotes de aplicações*.

2.1.1.1 Aplicações legadas

As aplicações legadas são também conhecidas por *sistemas tradicionais*. As características que definem este tipo de aplicação são o processamento centralizado e o acesso ao sistema ser feito através de terminais [Lin00]. Tanto os dados como a lógica de acesso a dados estão juntos no mesmo sistema. Um exemplo típico de um sistema tradicional é um *mainframe*.

Ao longo dos anos, as aplicações legadas tornaram-se repositórios de dados, contendo informações de grande importância para as organizações. Por outro lado, as aplicações legadas incluem no seu código fonte regras de negócio (por exemplo em que momentos aplicar um determinado desconto). Estas características tornam preferível utilizar as aplicações legadas já existentes em vez de criar réplicas destas aplicações com tecnologias mais actuais para depois as integrar com outras [RB01].

2.1.1.2 Aplicações desktop

As aplicações *desktop* são aplicações instaladas em cada computador de uma dada organização e que são apenas acessíveis no próprio computador. Em termos de integração, este tipo de aplicações constitui um desafio pelo facto de cada aplicação conter localmente as regras de negócio e os dados específicos da aplicação. Possibilitar o acesso aos dados e processos das aplicações pode obrigar à deslocação dos dados e os processos para um servidor central [Lin00].

2.1.1.3 Pacotes de software

Um pacote de *software* (*packaged application*) trata-se de uma aplicação que é comprada em vez de ser desenvolvida pela própria empresa [Lin00]. A empresa deixa de ter a preocupação do desenvolvimento e manutenção, passando essa responsabilidade directamente para o vendedor da aplicação. Contudo, este tipo de aplicações pode não responder à totalidade das necessidades do negócio de toda a empresa [RB01]. Desta forma, os pacotes de *software* têm muitas vezes necessidade de ser integrados, tanto com outros pacotes adquiridos como com aplicações já existentes no seio de uma organização. O facto de serem aplicações proprietárias torna difícil a sua integração.

2.1.2 Modelos de integração

É usual o desenvolvimento de aplicações estar organizado por camadas onde cada camada implementa determinada característica da aplicação. O modelo mais frequente é o modelo de três camadas constituído por uma camada para a apresentação, outra para a lógica de negócio e finalmente uma camada que trata do acesso aos dados. A integração de aplicações tira partido desta característica dando origem a modelos de integração distintos conforme a camada da aplicação onde intervêm. Um modelo de integração, também conhecido por *tipo de integração*, define a forma de integrar as aplicações, fornecendo a natureza e os mecanismos para a integração [RB01]. Neste trabalho destacam-se três modelos de integração:

- Modelo de integração ao nível da apresentação;
- Modelo de integração ao nível dos dados;
- Modelo de integração ao nível das funcionalidades.

2.1.2.1 Modelo de integração ao nível da apresentação

O modelo de integração ao nível da apresentação consiste em combinar várias aplicações utilizando a sua interface gráfica como ponto de integração para a partilha de dados e regras de negócio [Lin00]. As aplicações já existentes são acedidas directamente pela sua interface com o utilizador. O resultado da integração é assim uma nova aplicação que reúne, de uma forma transparente para o utilizador, funcionalidades de aplicações já existentes [RB01].

Este método é tipicamente utilizado para integrar aplicações cuja interface de utilização é baseada em texto (por exemplo, um terminal). Também é utilizado em casos

onde se pretende que o utilizador tenha a percepção de estar a usar uma única aplicação que na verdade é a composição de várias.

Relativamente aos restantes modelos de integração este modelo é simples de implementar visto que, na sua essência, trata-se apenas de aceder à interface gráfica de uma aplicação por intermédio de outra. Contudo os pontos de integração ficam limitados àquilo que as aplicações podem fornecer através das suas interfaces de utilização. Este modelo não é o preferido dos arquitectos de integração e só é utilizado em casos onde a interface de utilização é o único ponto de acesso de uma aplicação.

2.1.2.2 Modelo de integração ao nível dos dados

O modelo de integração ao nível dos dados permite a integração de aplicações através do acesso aos dados por elas criados, geridos ou armazenados [RB01]. A informação é extraída de uma (ou mais) fonte de dados. Durante a extracção poderá ser aplicado processamento e transformação de dados conforme a lógica de negócio da empresa.

De acordo com este modelo, os dados da aplicação são acedidos de forma directa, isto é, sem se utilizar as camadas de acesso a dados e de lógica de negócio. Assim, este modelo tem a vantagem de permitir aumentar o volume de dados acessíveis. Isto porque em vez de se usar a camada de acesso a dados (com a limitação à informação retornada pelos métodos da camada) os dados são acedidos de forma directa [RB01]. Outra vantagem deste tipo de integração é o baixo custo comparativamente aos restantes tipos. Isto porque ao longo da aplicação do processo de integração, não é necessário alterar o código fonte de nenhuma aplicação [Lin00].

No entanto este modelo cria uma relação de compromisso com o formato da fonte de dados. Se o modelo de dados for alterado, a integração pode deixar de funcionar, obrigando à reescrita da solução [RB01].

2.1.2.3 Modelo de integração ao nível de funcionalidades

O modelo de integração de funcionalidades consiste em utilizar as interfaces de uma aplicação para aceder tanto às regras de negócio como aos dados da aplicação [Lin00]. Uma significante parte do orçamento do desenvolvimento de uma aplicação é gasto na criação e manutenção da lógica de negócio. A lógica de negócio trata-se do código da aplicação que implementa as regas de negócio específicas da empresa. Existe

pois interesse em poder partilhar a lógica de negócio já existente numa aplicação evitando assim a sua replicação [RB01].

Um problema comum verifica-se quando se tentam integrar pacotes de software provenientes de mais do que um fornecedor [Lin00]. Isto porque cada fornecedor disponibiliza as suas interfaces de diferentes formas. Durante a integração é necessário extrair a informação (dados), converte-la num formato apropriado para a aplicação destino e finalmente transferir essa informação. A solução mais frequentemente utilizada neste tipo de integração é o recurso a sistemas de mensagens [Lin00].

2.2 Middleware

O *middleware* é a tecnologia que viabiliza a integração pois constitui ele próprio um ponto de ligação entre sistemas. O *middleware* é qualquer tipo de *software* que permite comunicação entre dois ou mais sistemas [Lin00]. Oculta as complexidades de baixo nível de protocolos de rede e API dos sistemas que comunicam entre si. Desta forma, o trabalho do programador passa a estar centrado nas questões relacionadas com a partilha de informação e não na forma de como partilha-la.

2.2.1 Tipos de middleware

Na definição dada para *middleware* referiu-se a comunicação entre sistemas de uma forma vaga e sem especificar que tipo de sistemas. Conforme o tipo de sistema (base de dados, aplicação distribuída, fila de mensagens, etc.) e conforme as características específicas do *middleware*, identificam-se diferentes tipos de *middleware*. Entre os quais se destacam:

- *Remote Procedure Call*;
- *Message-Oriented Middleware*;
- Objectos distribuídos;
- *Database-Oriented Middleware*;
- *Transaction-Oriented Middleware*;
- *Bus* de mensagens.

2.2.1.1 Remote Procedure Call

Os *Remote Procedure Call* (RPC) foram uma das primeiras formas de *middleware* que surgiram. Os RPC permitem invocar uma função num programa como se de um

método local se tratasse e executar essa função noutro programa ou até noutra máquina. Um RPC oculta os pormenores da utilização das camadas de rede e os detalhes do sistema operativo, o que se transforma numa facilidade de utilização para o programador.

Os RCP são síncronos, o que significa que o processo que invocar um RCP ficará bloqueado até que a resposta ao pedido seja retornada.

2.2.1.2 Message-Oriented Middleware

O *Message-Oriented Middleware* (MOM) trata-se tipicamente de *software* que faz uso de mensagens para mover informação de um ponto para outro [Lin00]. O MOM disponibiliza uma API que abstrai o programador dos detalhes de *hardware*, sistema operativo e protocolos de rede.

Os MOM utilizam um de dois modelos de distribuição de mensagens: processo–a–processo ou baseado em filas [Lin00]. No modelo de processo–a–processo o MOM move a mensagem directamente entre dois processos, sendo por isso necessário que ambos os processos estejam em execução. No modelo baseado em filas todas as mensagens trocadas entre dois processos passam primeiro por uma fila que serve de intermediária. A aplicação produtora de informação coloca uma mensagem na fila sendo essa mensagem mais tarde lida pela aplicação consumidora. A consequência directa deste modelo é que os processos participantes na troca de informação não necessitam de estar simultaneamente activos. A grande parte dos produtos MOM existentes no mercado utiliza filas para transportar as mensagens.

Quando são utilizadas filas de mensagens para aplicações comunicarem entre si, é frequente associar a comunicação assíncrona a este tipo de *middleware*. Exemplos de produtos existentes neste tipo de *middleware* são o MSMQ (Microsoft) ou o MQSeries (IBM).

2.2.1.3 Objectos distribuídos

O uso de objectos distribuídos consiste em aplicar conceitos de *object oriented* ao *middleware*. Os objectos distribuídos podem ser considerados *middleware* pelo facto de permitirem a comunicação entre diferentes aplicações. São disponibilizadas interfaces que tornam o funcionamento de uma aplicação semelhante à utilização de um ou mais objectos locais. A aplicação pode depois ser acedida através de uma rede ou da internet utilizando as interfaces disponibilizadas por cada objecto [RB01].

2.2.1.4 Database-Oriented Middleware

O *Middleware* orientado a bases de dados trata-se de qualquer aplicação que facilite a comunicação de uma base de dados com outra aplicação ou com outra base de dados [Lin00]. Este tipo de *middleware* é tipicamente utilizado para extrair informação de bases de dados locais ou remotas.

O *middleware* orientado a bases de dados pode ser de dois tipos: *Call Level Interfaces* (CLI) ou *Native Database*. Os CLI são geralmente utilizados em bases de dados relacionais e caracterizam-se por disponibilizar aceso a um número variado de bases de dados através do uso de interfaces conhecidas. Exemplos de *middleware* orientado a bases de dados do tipo CLI são as implementações de JDBC ou OLE DB.

Por outro lado o *middleware native database* não faz uso de API. Em vez disso, acede a funções e características particulares de uma base de dados em específico utilizando apenas mecanismos nativos. Tem a desvantagem de suportar apenas o sistema de gestão de bases de dados para o qual foi projectado. Podem, no entanto, conseguir melhores desempenhos no acesso aos dados, visto que são especialmente desenhados para uma arquitectura específica.

2.2.1.5 Transaction-Oriented Middleware

O *middleware* orientado a transacções como, por exemplo, monitores transaccionais [Ber90], efectua o trabalho de coordenar o movimento de informação e a partilha de funcionalidades entre diferentes recursos. Um monitor de transacções fornece suporte para regras de negócio críticas preservando a integridade de informação distribuída em recursos como bases de dados, ficheiros ou filas de mensagens [RB01].

2.2.1.6 Bus de mensagens

Ao longo do tempo, vários nomes foram atribuídos ao *middleware* especializado no transporte de mensagens. *Bus* de mensagens, *broker*, sistema de mensagens, servidor de mensagens ou servidor de integração, são apenas alguns exemplos. Neste trabalho, será utilizado o nome *bus* de mensagens para referir este tipo de *middleware*.

Um *bus* de mensagens permite o movimento de informação (sob a forma de mensagens) entre duas ou mais entidades, lidando ainda com diferentes semânticas e plataformas de aplicações [Lin04]. A facilidade que oferecem na transferência de informação entre sistemas, torna um *bus* de mensagens numa ferramenta adequada para integração. Em rigor, um *bus* de mensagens é um MOM, visto que o seu funcionamento

é orientado à mensagem. No entanto o conjunto de funcionalidades oferecido por um *bus* de mensagens (por exemplo encaminhamento ou transformação) pode ser tão vasto que faz com que este se distinga dos restantes sistemas MOM.

Em muitos cenários, para uma eficaz integração, para além de ferramentas que permitam a partilha de informação, são necessárias ferramentas que possibilitem o encaminhamento e a transformação dessa informação. Sendo verdade que é possível delegar estas tarefas em aplicações já existentes ou criadas para o efeito, verifica-se que é boa prática, do ponto de vista arquitectural, concentrá-las num *bus* de mensagens, criando assim um único ponto de integração [Lin04].

Para ser considerado como pilar de integração, o *bus* de mensagens tem de ser capaz de comunicar com vários tipos de aplicações independentemente da tecnologia ou plataforma que utilizam. Os clientes de um *bus* de mensagens são, não só as aplicações comuns, mas também qualquer outra fonte de informação como por exemplo um servidor de base de dados, um servidor *Web*, objectos distribuídos, etc. A esta capacidade do *bus* de mensagens comunicar com qualquer fonte de dados, [Lin00] designa por "*any-to-any*". Outra propriedade importante num *bus* de mensagens é o "*many-to-many*" que significa que, uma vez que uma ou mais aplicações publiquem determinada informação, essa informação ficará disponível para qualquer aplicação que lhe pretenda aceder.

2.3 Serviços de um Message-Oriented Middleware

De acordo com [Mah04], existe um conjunto de serviços que são comuns à maioria do *middleware* orientado a mensagens. De seguida destacam-se os mais relevantes:

- Transacções;
- Garantia de entrega;
- Balanceamento de carga;
- *Clustering*.

2.3.1 Transacções

As transacções permitem que uma aplicação agrupe um conjunto de tarefas e as execute com a garantia de que, ou são todas concluídas com sucesso ou então nenhuma delas é realizada. Para que uma transacção tenha o efeito esperado é necessário que esta cumpra as seguintes propriedades (conhecidas por propriedades ACID):

- Atomicidade – Todas as tarefas são concluídas ou então nenhuma é realizada.
- Consistência – Dado um estado inicial consistente, o estado final terá também de ser consistente independentemente do resultado da transacção (sucesso/falha)
- Isolamento – As transacções são executadas de forma isolada e sem entrar em concorrência com outras transacções.
- Durabilidade – O efeito de uma transacção concluída com sucesso não é perdido.

No contexto de uma transacção, qualquer objecto que seja alvo de uma alteração é designado por *recurso*. É comum um MOM ter a capacidade de realizar o envio ou a recepção de mensagens dentro de uma transacção; podendo as transacções ser de dois tipos – Transacções locais e Transacções globais. Transacções locais envolvem apenas uma entidade como por exemplo um sistema de mensagens que faz ele próprio o controlo dos recursos de forma independente. As transacções globais envolvem vários sistemas de mensagens distribuídos e heterogéneos, cada um controlando os seus recursos. Existe depois uma entidade externa, por exemplo um *Distributed Transaction Coordinator* (DTC) que faz o controlo global da transacção em cada sistema de mensagens. [BN09]

Uma única transacção pode conter o envio de mais do que uma mensagem. Nesse caso, o MOM espera que o cliente faça *commit* da transacção antes de fazer o processamento de cada mensagem.

No caso de a transacção ser iniciada pela aplicação que vai receber as mensagens, então o MOM só elimina as mensagens entregues depois de o cliente ter feito *commit*. Caso seja feito *rollback* o MOM guarda as mensagens como se nunca tivessem sido enviadas.

2.3.2 Garantia de entrega de mensagens

De forma a poder garantir a entrega das mensagens, o MOM tem de guardar todas as mensagens num repositório não volátil – por exemplo um disco rígido. A plataforma envia a mensagem para o destino e aguarda a confirmação de entrega. Isto permite ao remetente da mensagem deixar de ter a preocupação com a entrega da mensagem no destino (*fire-and-forget*), sendo essa responsabilidade assumida pelo MOM.

2.3.3 Balanceamento de carga

O balanceamento de carga é o processo de distribuir a carga de processamento de um sistema por vários servidores. A distribuição correcta deve ser feita dinamicamente

de forma a dar mais processamento aos servidores que se encontram com menos actividade.

Existem duas abordagens principais para o balanceamento de carga – *push* e *pull* [Mah04]. No modelo *push* é utilizado um algoritmo para distribuir a carga pelos vários servidores com base na previsão de qual o servidor mais disponível. Este modelo pode ser imperfeito, visto que o seu desempenho está directamente dependente do erro na estimativa do servidor mais disponível. No modelo *pull* a distribuição de carga é feita colocando mensagens numa fila. Cada servidor encarregue de fazer o processamento das mensagens, recolhe uma da fila assim que estiver livre para a processar. Este modelo é mais eficiente na medida de que os servidores só entram em acção quando estão realmente livres, conduzindo assim a um melhor aproveitamento dos recursos existentes. Como desvantagem, existe o risco dos produtores de mensagens trabalharem a um ritmo muito superior aos dos servidores que recolhem mensagens da fila. Desta forma, criar-se-ia um ponto de congestionamento na fila que poderia levar ao esgotamento dos recursos.

2.3.4 Clustering

Quando os limites de um servidor quer a nível de *software*, quer a nível de *hardware*, são atingidos, torna-se necessário ter mais que uma máquina a funcionar no sistema. O *clustering* consiste na distribuição de uma aplicação por vários servidores com vista a aumentar a escalabilidade do sistema, melhorando assim tanto o desempenho como a fiabilidade. As funcionalidades de um sistema são replicadas ao longo de vários servidores, aumentando assim a tolerância a falhas. Do ponto de vista lógico, o sistema continua a ser visto como uma parte coesa que funciona numa só máquina. O conceito de *clustering* é intrínseco à noção de balanceamento de carga, uma vez que só faz sentido falar em balanceamento de carga depois de se ter um conjunto de servidores a efectuar o mesmo tipo de processamento.

2.4 Serviços de um *bus* de mensagens

É comum um *bus* de mensagens disponibilizar um conjunto de funcionalidades para além das enumeradas na secção anterior para os MOM. Entre as principais funcionalidades oferecidas encontram-se a transformação de dados, processamento de regras e o encaminhamento de mensagens. A existência de outras funcionalidades varia de implementação para implementação.

2.4.1 Transformação de mensagens

A camada de transformação compreende o formato de todas as mensagens que passam no *bus* podendo alterar (transformar) o seu conteúdo. Os dados de uma mensagem são reestruturados resultando numa nova mensagem que é compatível com a(s) aplicações destino. A camada de transformação conhece a forma como cada aplicação comunica com o exterior, assim como que partes de informação têm significado para cada aplicação.

É comum a camada de transformação conter ferramentas de *parsing* e métodos de detecção de padrões para obter a estrutura dos formatos de mensagens suportados. Cada campo da mensagem é então representado individualmente. Uma vez que o corpo da mensagem se encontre decomposto em vários campos, é possível fazer diferentes combinações entre campos formando uma nova mensagem. A Figura 1 ilustra um cenário de transformação onde se altera a estrutura da mensagem (transformação de esquema) quando se retiram os campos SSN1, SSN2 e SSN3 substituindo-os pelo campo SSN. Também ocorre uma transformação de dados ao se substituir o valor "Virginia" por "Va"

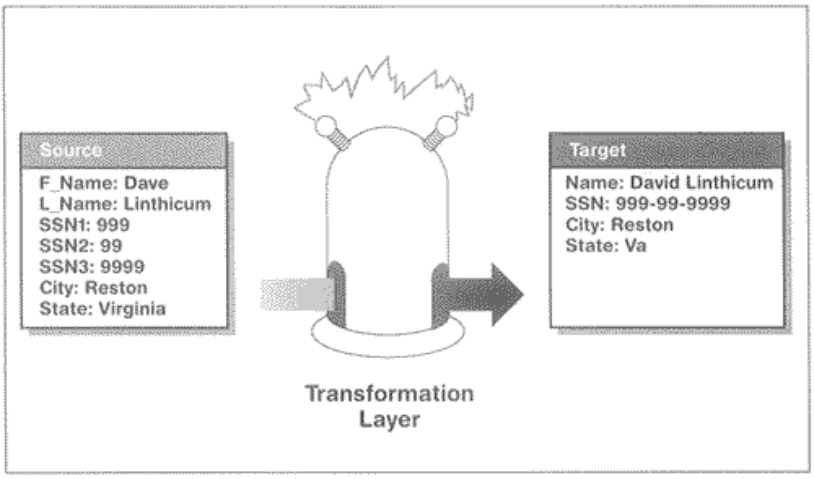


Figura 1: Exemplo de transformação de uma mensagem. Fonte: [Linthicum – 2000]

2.4.1.1 Conversão de esquemas

Uma conversão de esquemas trata-se de um tipo de transformação da qual resulta a alteração da própria estrutura da mensagem com o objectivo de a tornar compatível com a aplicação destino [Lin04]. Por exemplo, considere-se um banco onde existe um conjunto de aplicações que trocam informação entre si. Entre elas, um sistema DB2 que utiliza a seguinte estrutura de dados para representar contas de clientes:

```
Cust_Nr.                Alphanumeric  10
Amt_Due                 Numeric       10
Date_of_Last_Bill       Date
```

E que envia mensagens com a seguinte informação:

```
Cust_Nr                  AB99999999
Amt_Due                  560.50
Date_of_Last_Bill        09/17/2007
```

Suponha-se que estes dados têm de ser enviados para outro sistema onde a mesma informação é armazenada com a seguinte estrutura:

```
Customer_Number          Numeric       20
Money_Due                Numeric       8
Last_Billed              Alphanumeric  10
```

No exemplo dado, o esquema utilizado pelo sistema destino é diferente do esquema utilizado pelo sistema origem. Mover informação entre estes dois sistemas sem uma conversão de esquemas apropriada, resultará em erros provocados por incompatibilidade. Para uma comunicação correcta é necessário converter a informação de "Cust_Nr", cujo formato é alfanumérico, numa cadeia de 20 dígitos. Aqui terá de ser adoptada uma regra de conversão de caracteres em números (por exemplo assumir que A=1, B=2, C=3, etc.). O mesmo terá de ser feito em relação à data.

Muitos sistemas de mensagens utilizam o seu próprio formato interno. Cada mensagem que for recebida é convertida nesse formato, aplicadas as regras necessárias e à saída a mensagem é convertida no formato desejado pelo sistema destino.

2.4.1.2 Conversão de dados

Uma conversão de dados corresponde à alteração do valor de campos da mensagem, sem necessariamente alterar a sua estrutura. Um exemplo de uma conversão de dados consiste em substituir os caracteres "LX" pela palavra "Lisboa" num campo que indique uma cidade. Muitas vezes alterações de estrutura e dados são realizadas em simultâneo (como foi o caso no exemplo da secção anterior, onde se converteram os caracteres "A" e "B" em números).

[Lin04] indica duas formas para realizar uma conversão de dados; recorrer a algoritmos ou utilizar tabelas de *look-up*. O uso de algoritmos consiste em calcular o novo valor de um campo da mensagem com base num procedimento especificado (o algoritmo). Como exemplo de uma conversão por algoritmo tem-se o caso de uma mensagem que contém entre os seus campos o preço de um produto e a quantidade vendida e o sistema destino exige que a mensagem inclua um campo com o valor total facturado. A transformação é então feita utilizando um algoritmo que multiplica o valor de cada produto pelas unidades vendidas.

No caso do uso de tabelas *look-up* a conversão de campos é feita por análise de uma tabela de duas colunas e sem recurso a nenhum algoritmo extra. Por exemplo, para converter os caracteres "LX" em "Lisboa" é acrescentada à tabela de *look-up* uma linha com esta informação.

2.4.2 Encaminhamento

O encaminhamento constitui a camada do *bus* de mensagens capaz de identificar a chegada de uma mensagem e encaminhá-la para a aplicação apropriada.

Quando o *bus* recebe uma mensagem, inspecciona o seu conteúdo decidindo se a mensagem é entregue a uma ou várias aplicações ou se, em vez disso, é descartada. O funcionamento desta camada baseia-se em tabelas de encaminhamento cujas linhas são avaliadas como uma expressão booleana. O resultado de cada avaliação determina se a mensagem é ou não entregue a uma determinada aplicação.

O encaminhamento pode ser de vários tipos, nomeadamente:

- *Channel-based* – Define-se canal (em algumas tecnologias referido por tópico) como um destino de mensagens e é atribuído a cada canal um significado específico. Por exemplo, no contexto de uma entidade bancária, um determinado canal pode representar todas as mensagens com ordens de débito numa conta. Os remetentes enviam as mensagens para os canais conforme o seu significado. As aplicações interessadas em mensagens com um determinado significado registam-se no canal respectivo.
- *Subject-based* – Adiciona-se a cada mensagem o campo *subject* que inclui uma descrição que identifica o tipo do conteúdo da mensagem. As aplicações indicam

quais os padrões de texto que pretendem encontrar nos campos *subject* das mensagens recebidas.

- *Content-based* – O encaminhamento é realizado com base na análise de todo o conteúdo da mensagem. As aplicações especificam qual o conteúdo (ou padrão) que pretendem encontrar no corpo das mensagens recebidas.

O encaminhamento pode revelar-se uma ferramenta útil na distribuição de carga entre várias aplicações que processam pedidos. Por exemplo, num sistema distribuído de processamento de encomendas configura-se o encaminhamento para entregar a uma aplicação mensagens com encomendas entre 0€ e 5€; outra aplicação recebe mensagens com encomendas entre 10€ e 50€; outra recebe mensagens com encomendas a partir de 50€, etc.…

2.4.3 Processamento de regras

O motor de processamento de regras é o componente do *bus* de mensagens que permite a criação de regras para controlar o processamento e a distribuição de mensagens. A utilização de regras de processamento permite, por exemplo, criar novas formas de encaminhamento e transformação.

De um modo geral, as regras são especificadas utilizando linguagens de *script* e interpretadores em vez de linguagens de programação (como Java ou C++) e compiladores. Contudo a forma como cada produto processa as suas regras varia de vendedor para vendedor.

O processador de regras constitui um importante ponto de extensibilidade dentro do *bus* de mensagens. Com o uso de regras é possível, por exemplo, gerar novas mensagens sempre que determinada condição for verificada, aplicar filtros de mensagens, etc.

2.4.4 Armazenamento de mensagens

O armazenamento de mensagens consiste em armazenar num suporte físico persistente (por exemplo uma base de dados) todas as mensagens que passam pelo *bus*. Na maior parte dos casos as mensagens são armazenadas sem alterações, contudo poderão ser armazenadas após uma ou mais transformações, se tal for importante para melhorar a qualidade da informação armazenada [Lin04].

No geral esta funcionalidade é disponibilizada para dar suporte a vários requisitos, nomeadamente: *data mining*, integridade, arquivo e auditoria.

Data mining consiste na análise de grandes quantidades de mensagens com vista à detecção de padrões ou regras que possam auxiliar na tomada de decisões. Por exemplo, se for considerado o caso de uma empresa de vendas onde as várias aplicações existentes comunicam através de um *bus* de mensagens, ao se analisar uma grande quantidade de mensagens com pedidos de compra, é possível determinar padrões e características dos clientes.

O armazenamento de mensagens constitui ele próprio uma forma de persistência (garantia de que as mensagens não são perdidas), dado que estas ficam armazenadas em suporte persistente, garantindo com isso a **integridade** das mensagens. Se por algum motivo o sistema de mensagens falhar, as mensagens que se encontravam a ser processadas no momento da falha serão recuperadas recorrendo ao serviço de armazenamento.

O **arquivo** permite ao *bus* armazenar mensagens durante vários meses ou mesmo anos. Por exemplo, pode ser útil num dado momento do futuro analisar as mensagens trocadas no passado.

As acções de **auditoria** consistem em examinar as mensagens arquivadas e tirar conclusões acerca do desempenho do sistema de mensagens e do estado da comunicação entre aplicações. É possível, por exemplo, verificar o tráfego de mensagens de dias específicos e tirar conclusões acerca de picos de utilização do *bus*. Também é possível analisar situações onde ocorreram erros e tirar as respectivas conclusões.

2.4.5 Autenticação e segurança

Um *bus* de mensagens implementado numa empresa estará acessível a várias aplicações. Tendo em conta o modelo de negócio da empresa, poderá não se pretender que o *bus* de mensagens seja utilizado de igual modo por todas as aplicações. Para solucionar este problema, o *bus* inclui um sistema de autenticação, através do qual verifica a identidade de cada cliente podendo, em função do cliente, limitar o conjunto funcionalidades disponíveis, assim como o conjunto de clientes com os quais é permitido comunicar.

As mensagens trocadas no sistema irão conter informação que pode ser confidencial. É importante evitar que aplicações consigam receber mensagens que não lhes são destinadas. Como garantia extra de protecção dos dados da uma mensagem, o *bus* pode fornecer mecanismos para que o conteúdo de determinadas mensagens seja cifrado durante o transporte no sentido cliente – *bus* e vice-versa. Desta forma, evita-se que mesmo que uma aplicação consiga inadvertidamente tomar posse de uma mensagem não será capaz de interpretar o seu conteúdo.

2.4.6 Interface gráfica

Outra das funcionalidades oferecidas por um *bus* de mensagens é uma interface gráfica que permita criar regras, definir transformações ou visualizar as aplicações cliente activas.

A interface gráfica é ainda útil em tarefas de administração como a monitorização do tráfego de mensagens, desempenho do sistema, estado das aplicações e na detecção de problemas. O administrador tem ainda acesso a estatísticas de utilização do *bus* de mensagens para que, com base nessa informação, possa proceder a melhorias de desempenho ou resolução de problemas.

2.5 Comunicação ponto–a–ponto *versus* *bus* de mensagens

Para que duas ou mais aplicações possam trocar informação entre elas é necessário que ambas se encontrem conectadas. Neste capítulo faz-se a comparação entre duas formas possíveis de interligar várias aplicações – através de ligações ponto–a–ponto ou utilizando um *bus* de mensagens.

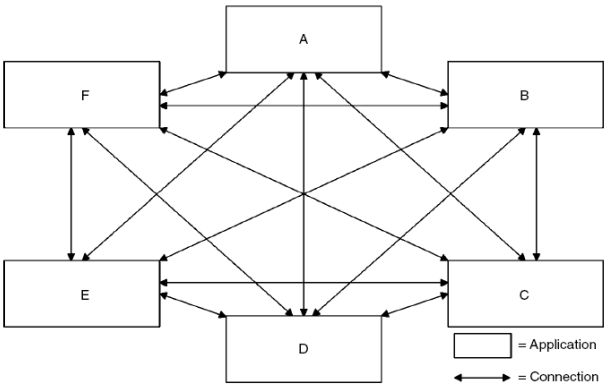


Figura 2: Ligações ponto–a–ponto. Fonte: [Mahmoud – 2004]

Na comunicação ponto–a–ponto, uma aplicação A que pretende comunicar com uma aplicação B, estabelece uma ligação entre A e B. Essa ligação apenas pode ser usada pelas aplicações A e B. Se A pretender comunicar com C, então terá de criar uma nova ligação entre A e C. Desta forma num sistema com *n* aplicações onde seja necessário que cada aplicação comunique com qualquer uma das outras, cada aplicação terá ligações, o que se traduz num total de ligações. A Figura 2 ilustra o cenário atrás descrito. O problema de tal quantidade de ligações está na dificuldade de administrar todas as ligações e no facto deste tipo de ligação introduzir um alto acoplamento (ver secção 2.5.1) entre as aplicações. Existe ainda o problema de como garantir a disponibilidade de cada aplicação para as restantes.

Ao utilizar um *bus* de mensagens (ou de uma forma geral, um MOM) para promover a comunicação entre várias aplicações, introduz-se um intermediário entre as aplicações, tal como se sugere na Figura 3.

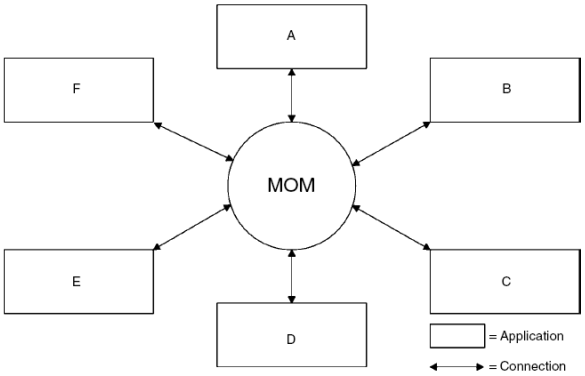


Figura 3: Aplicações que comunicam através de um MOM.Fonte: [Mahmoud – 2004]

Uma das vantagens da presença deste intermediário é o facto de a comunicação passar a ser assíncrona. Cada aplicação que pretender comunicar, só precisa de garantir a ligação ao MOM. A partir desse momento, para o remetente, qualquer destino será visto como disponível, independentemente de estar ou não activo. Outra vantagem é o facto de o MOM assumir a responsabilidade da entrega das mensagens.

2.5.1 Acoplamento

As ligações ponto–a–ponto dão origem a um elevado acoplamento (*tightly coupling*) entre sistemas [Mah04]. Isto acontece porque alterações feitas num sistema que impliquem mudanças nas interfaces deste com o exterior, obrigam a alterações nos sistemas que interagem com o sistema inicialmente alterado. À medida que o número de

alterações aumenta também os custos de manutenção de toda a arquitectura distribuída vão aumentar.

O *bus* de mensagens, em contraste a uma ligação ponto-a-ponto, introduz uma nova camada entre a aplicação remetente e a aplicação destino. Desta forma, a relação directa passa a ser entre as aplicações e o *middleware*, contribuindo assim para um forte desacoplamento entre as aplicações que trocam mensagens entre si.

2.5.2 Fiabilidade

Comunicações fiáveis (*reliable communications*) podem ser a maior prioridade numa arquitectura distribuída. Qualquer falha exterior à aplicação – *software*, rede, *hardware* – pode afectar a troca de dados entre sistemas.

Com a existência de um MOM e também no caso concreto de um *bus* de mensagens, a perda de mensagens através da rede ou por falha de sistema é prevenida pelo mecanismo *store and foward*. O facto de o MOM assumir o compromisso de não perder mensagens confere a todo o sistema um elevado nível de fiabilidade [Mah04].

2.5.3 Escalabilidade

A escalabilidade traduz-se na capacidade de um sistema lidar com o aumento de carga. Se o sistema considerado for, não apenas o *bus* de mensagens mas também os seus clientes e a métrica utilizada for o número de mensagens trocadas num intervalo de tempo; então verifica-se que a utilização de um MOM aumenta a escalabilidade do sistema *bus* + clientes.

O facto de a comunicação numa ligação ponto-a-ponto ser síncrona, faz com que uma aplicação que envie um pedido a outra, tenha de esperar pela resposta. Com o aumento do número de pedidos, a aplicação remetente terá de esperar tanto mais tempo quanto o aumento de tempo de processamento na aplicação destino. Considerando várias aplicações a comunicarem entre si num sistema distribuído, o sistema será tão lento quanto o tempo máximo de resposta da aplicação mais lenta [Mah04].

Como consequência do baixo acoplamento entre sistemas oferecido pelo *bus* e o facto de a comunicação ser assíncrona, cada aplicação pode ser alterada de forma a aumentar o número de mensagens enviadas sem consequência directa para as restantes aplicações. A limitação encontra-se apenas no valor máximo de mensagens que o *bus* é capaz de processar por intervalo de tempo.

2.5.4 Disponibilidade

A comunicação ponto-a-ponto exige que ambos os intervenientes estejam simultaneamente activos. Se uma aplicação falhar, nenhuma aplicação vai poder utilizar os serviços disponibilizados pela aplicação que falhou durante todo o tempo em que esta estiver inactiva.

No caso de um MOM, devido à troca de mensagens ser feita segundo o modelo assíncrono, do ponto de vista da aplicação remetente, a aplicação destino está sempre activa. Isto faz com que a disponibilidade num sistema com um MOM esteja apenas limitada pela disponibilidade do próprio MOM.

2.6 Modelos de troca de mensagens

Um modelo de troca de mensagens define a forma como produtores e consumidores comunicam, estabelecendo os critérios de entrega de mensagens. Por determinarem o modo como as mensagens são entregues, os modelos de troca de mensagens são um conceito intrínseco ao encaminhamento (secção 2.4.2). Alguns dos modelos existentes são o ponto-a-ponto, segundo o qual cada mensagem destina-se apenas a um consumidor; o publicador-subscritor, que permite que uma mensagem seja entregue a mais do que um destino; o *ring* [Mah04], no qual existem vários consumidores mas a mensagem é entregue apenas àquele que tiver recebido uma há mais tempo; ou o *broadcast* [RB01], onde cada mensagem é entregue a todos os clientes existentes.

Neste trabalho dá-se destaque ao modelo de troca de mensagens ponto-a-ponto e ao publicador-subscritor, por se entender que estes são os dois modelos mais relevantes num *bus* de mensagens.

2.6.1 Mensagens ponto-a-ponto

No capítulo 2.5 foi apresentado o conceito de ligação ponto-a-ponto, de um ponto de vista físico, entre *hardware*. No entanto, do ponto de vista lógico, existe o conceito de troca de mensagens ponto-a-ponto sem necessariamente significar que os sistemas que trocam mensagens estejam directamente ligados (do ponto de vista físico). Num modelo de mensagens, comunicação ponto-a-ponto significa que o remetente envia a mensagem para um destino específico. A mensagem poderá entretanto passar por um ou mais intermediários (Figura 4) [Cum02].

Figura 4: Comunicação ponto-a-ponto (um-para-um)

Este modo de comunicação tem como características principais o facto da aplicação remetente saber exactamente qual é a aplicação que irá consumir a mensagem. Por outro lado, é garantido que a mensagem terá apenas um destinatário.

Mensagens ponto-a-ponto são o modelo de troca de mensagens mais adequado para comunicações baseadas em *pedido-resposta* (*request-response*). Neste cenário, um servidor recebe pedidos sob a forma de mensagens de várias aplicações. Depois de terminado o processamento, o servidor envia o resultado para quem enviou o pedido [Cum02].

2.6.2 Publicador-subscritor

O modo de comunicação publicador-subscritor é um mecanismo de difusão de mensagens que permite a uma aplicação propagar uma mensagem para um número desconhecido de aplicações, num modo de comunicação *um-para-muitos* ou *muitos-para-muitos* [Cum02]. As mensagens não são enviadas directamente para um destino específico mas sim para um *tópico*. Um tópico é um destino lógico de mensagens e que não está afecto a nenhuma aplicação em específico [Hun03]. Geralmente um tópico está associado a um determinado tema. Por exemplo, numa organização com várias aplicações a comunicar entre si poderão existir os tópicos "vendas", "compras", "*stocks*", etc. O que caracteriza um tópico é o facto de poder ter múltiplos escritores e leitores de mensagens (ver Figura 5). As aplicações interessadas subscrevem um tópico e recebem todas as mensagens dirigidas a esse tópico ou apenas aquelas que obedecerem a determinados critérios de selecção. No contexto deste modo de troca de mensagens, quando uma aplicação envia uma mensagem diz-se que a mensagem foi *publicada*. Cada subscritor é um destino independente de mensagens pelo que as mensagens consumidas por um subscritor poderão também ser consumidas por outros subscritores. No entanto, cada subscritor consome uma única vez cada mensagem publicada [Cum02].

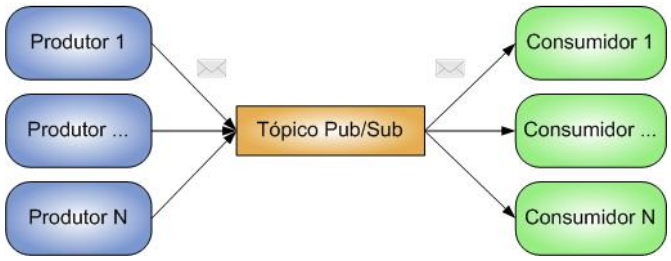


Figura 5: Comunicação Publicador-subscritor (muitos-para-muitos)

O modo publicador-subscritor permite um total desacoplamento entre as aplicações que produzem e consomem mensagens, visto que quem envia a mensagem não tem conhecimento de quem irá recebe-la. Esta característica torna este modo de troca de mensagens adequado para a propagação de eventos enviados sob a forma de mensagens. Um evento constitui um sinal que indica a ocorrência de algo e geralmente a aplicação que gera o evento não tem necessidade de conhecer quais e/ou quantas aplicações estão interessadas nesse evento. Cada aplicação que recebe a *mensagem evento* executa o conjunto de procedimentos definidos para o tratamento do respectivo evento.

2.7 Protocolos de comunicação

Se um *bus* de mensagens se assume como ferramenta de integração, tem de ser capaz de comunicar com clientes (produtores e consumidores) provenientes de diferentes ambientes e plataformas. Mais importante ainda é o *bus* ser capaz de comunicar utilizando as mesmas tecnologias que os clientes. Por exemplo, se um cliente só comunica através de *Web services*, então para integrar esse cliente com outros, o *bus* terá de ser capaz de comunicar por *Web services* também. Se outro cliente suporta apenas ligações TCP, então o *bus*, para além de *Web services*, terá ainda de comunicar por ligações TCP.

Um aspecto importante é que a integração torna-se bastante complexa se cada aplicação criar o seu próprio mecanismo de comunicação e obrigar os outros intervenientes a adaptarem-se a esse mecanismo proprietário. Para ultrapassar este problema, é necessário que exista um acordo quanto às tecnologias utilizadas na comunicação e ao formato dos dados trocados entre ambas as partes. Idealmente, este acordo constitui um *standard* que será conhecido por *bus* de mensagens e pelas aplicações cliente. Desta forma, o *bus* apenas terá de indicar quais o *standards* que

suporta e, caso os clientes suportem algum desses *standards*, então a compatibilidade entre ambos está garantida.

Existem actualmente alguns standards definidos. Uns especificam a tecnologia utilizada num canal de comunicação entre cliente e *bus*, outros definem os dados enviados nesse canal e de que forma devem ser interpretados e outros especificam uma API comum a todas as aplicações cliente. A todos estes standards, ao longo deste documento chamar-se-á *protocolos de comunicação*.

Neste capítulo é feita a análise de três protocolos de comunicação – Java Message Service (JMS), Advanced Message Queueing Protocol (AMQP) e RestMS.

2.7.1 Java Message Service

O *Java Message Service* (JMS) trata-se de uma API *standard* usada para aceder a uma variedade de *bus* de mensagens. Se cada vendedor de um produto de mensagens desenvolver uma API proprietária, o resultado será um vasto número de API distintas, o que reduz (ou mesmo elimina) a portabilidade de código das aplicações clientes e obriga o programador ao esforço adicional de ter de aprender várias API diferentes. [Hun03]

O JMS, que tal como o nome indica, aplica-se apenas ao mundo java, especifica a API que as aplicações cliente de um *bus* de mensagens utilizam. Desta forma, uma aplicação que utilize um *bus* de mensagens do fornecedor A pode passar a usar o *bus* de mensagens do fornecedor B, sem ter alterar o seu código fonte. Basta para isso que ambos os *bus* suportem a API do JMS.

Relativamente à API, esta define, entre outros, objectos que os clientes utilizam para estabelecer a ligação com o *bus*, objectos que mantêm o estado dessa sessão e objectos que representam as mensagens enviadas e recebidas. Suporta vários modelos de troca de mensagens, entre eles, a comunicação ponto-a-ponto e publicador-subscritor.

A Listagem 1 mostra o exemplo da implementação de um cliente JMS que envia uma mensagem no modo publicador-subscritor. A implementação do cliente começa com a criação de uma ligação, da qual se obtém uma sessão. Essa sessão é utilizada para instanciar o objecto que representa o publicador (`QueueSender`) e o objecto que representa a mensagem (`TextMessage`).

```
Context context = new InitialContext(env);
QueueConnectionFactory f =
      (QueueConnectionFactory)context.lookup("ConnectionFactory");
QueueConnection qc = f.createQueueConnection();
QueueSession qs = qc.createQueueSession(false,
                                        Session.AUTO_ACKNOWLEDGE);
Queue q = (Queue)context.lookup("queue/jdt");
QueueSender sender = qs.createQueueSender(q);
TextMessage message = qs.createTextMessage();
message.setText("content");
sender.send(q, message);
qc.close();
```

Listagem 1: Exemplo do envio de uma mensagem em JMS. Fonte: [Hun03]

O objecto de contexto é criado a partir das variáveis de ambiente Java, cuja leitura se omitiu para simplificar o exemplo. Os métodos *lookup* utilizam o JNDI[1] que terá se ser previamente configurado. Os passos da aplicação consumidora são semelhantes, onde se utiliza o objecto de sessão para obter um `QueueReceiver` e no objecto `Queue` se invocam os métodos `start` e `receive`.

Note-se que a limitação do JMS ao mundo Java apenas se aplica aos clientes. Quanto ao servidor, este pode estar implementado numa outra linguagem/plataforma e disponibilizar em Java apenas a implementação da API.

O JMS apenas define as interfaces de uma API sem especificar em concreto como é que essa API deve ser implementada. O resultado é que cada vendedor constrói a sua própria implementação que, apesar de ter uma forma de utilização comum (a API), resulta num novo formato proprietário de troca de mensagens [Vin06]. O JMS tem ainda a particularidade de apenas se aplicar a tecnologias Java, o que significa que não permite estabelecer um *standard* de comunicação entre clientes implementados em plataformas que não sejam Java.

2.7.2 Advanced Message Queuing Protocol

O *Advanced Message Queuing Protocol* (AMQP) [AMQ08] define um protocolo de comunicação entre sistemas através da troca de mensagens utilizando uma combinação de técnicas *store-and-foward*, *publish/subscribe* e *file transfer*.

Foi desenvolvido por um grupo de trabalho formado por membros de várias empresas com o objectivo da criação de um *standard* aberto que permita interoperabilidade na troca de mensagens. Este protocolo está especialmente orientado para as necessidades técnicas em ambientes financeiros [Vin06]. Nestes ambientes estão

[1] Java Naming and Directory Interface. Para mais informações ver: [Hun03]

sistemas de organizações ligadas ao comércio ou entidades bancárias que têm como principais desafios a necessidade de altos níveis de desempenho, débito binário, escalabilidade e fiabilidade.

A especificação é composta pela definição de um protocolo binário de rede e de um conjunto de regras que define o comportamento dos sistemas intervenientes. O AMQP pressupõe a existência de um protocolo de transporte (e.g. TCP), onde se transmitem os dados de forma sequencial e divididos em várias *frames*.

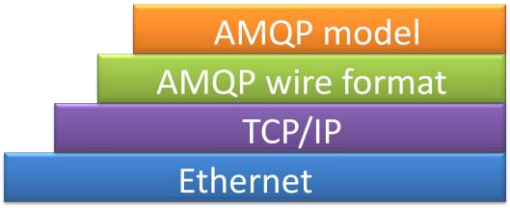


Figura 6: *Stack* do protocolo AMQP. Adaptado de [O'Hara – 2007]

A Figura 6 representa o *stack* de rede do protocolo. Na imagem indicou-se para camada base a *Ethernet*, no entanto, pode ser utilizada qualquer outra que suporte as camadas superiores. Apesar de estar preparado para utilizar protocolos de transporte como o UDP, actualmente a norma apenas reserva o porto 5672 TCP como ponto de acesso ao servidor. O campo de dados das tramas TCP contém a informação AMQP codificada num formato binário especificado pela norma. Ao nível do *modelo AMQP* está definida a *camada funcional* que compreende uma série de comandos possíveis de trocar entre clientes e servidor.

O AMQP é uma especificação completa que cobre áreas importantes para se conseguir a interoperabilidade que outras especificações não cobrem (por exemplo o JMS) [Vin06].

O AMQP é um protocolo neutro em relação à linguagem de programação, tendo sido concebido de modo a ser utilizado em diferentes plataformas, sistemas operativos e dispositivos de *hardware*. O formato das mensagens enviadas é definido sob o nível de transporte (camada *AMQP wire format* da Figura 6), possibilitando assim a correcta leitura das mensagens por qualquer sistema independentemente da tecnologia em que for implementado. As mensagens ficam no entanto com a limitação de não poderem incluir objectos dos tipos nativos da linguagem de programação dos sistemas clientes [O'H07]. Isto é compreensível pelo facto de cada cliente poder estar implementado em diferentes linguagens. Por exemplo, um objecto Java poderá não fazer sentido quando passado a um sistema C++.

2.7.2.1 Modelo AMQP

Para que exista a comunicação entre o cliente e servidor AMQP, é necessário criar uma série de orientações para a implementação das aplicações servidoras. Neste contexto, o modelo AMQP define um conjunto de componentes e regras na forma como esses componentes se relacionam entre si.

Existem três tipos de componentes que são colocados do lado do servidor em diferentes combinações de forma a obter a funcionalidade pretendida (ver Figura 7). São eles:

- *Exchanges*
- Filas (*Queue*)
- *Bindings*

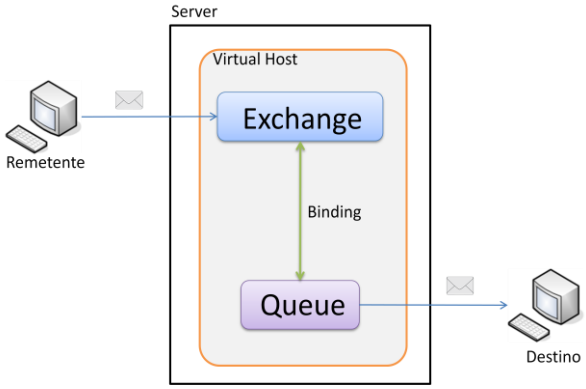


Figura 7: Modelo AMQP. Fonte: [AMQP 0.8 – 2006]

Exchanges, filas e *bindings* constituem objectos geridos pelas aplicações cliente (Origem ou Destino). O conjunto de *exchanges*, filas e *bindings* formam um *virtual host*, que constitui um domínio independente de componentes dentro do servidor. Componentes pertencentes ao mesmo *virtual host* partilham os mesmos dados de autenticação.

Através da combinação dos componentes conseguem-se diferentes modelos de troca de mensagens, tais como o *ponto-a-ponto* ou *publicador-subscritor* e utilizar conceitos como *store-and-forward* ou encaminhamento baseado em conteúdos.

Exchanges

Um e*xchange* é a entidade responsável por receber as mensagens das aplicações clientes e encaminha-las para filas com base em critérios de encaminhamento configuráveis. A aplicação cliente escolhe qual o *exchange* que pretende utilizar para

entregar a sua mensagem. O *exchange* verifica a informação presente nos cabeçalhos da mensagem e decide qual a fila para onde esta será transferida.

Um *exchange* nunca armazena mensagens, a sua função é inspeccionar cada mensagem recebida e procurar correspondências nas tabelas de encaminhamento que lhe foram configuradas.

Para permitir diferentes lógicas de encaminhamento de mensagens, existem vários tipos de e*xchanges*, entre eles destacam-se dois:

- *Direct exchange* – Para modelos de troca de mensagens ponto-a-ponto.
- *Topic exchange* – Para modelos de troca de mensagens publicador-subscritor.

Ambos os tipos tomam a decisão de encaminhamento com base na consulta de apenas um campo da mensagem – o *routing key*. O *routing key* funciona como um endereço virtual que as aplicações preenchem conforme o tipo do *exchange* para o qual enviam a mensagem. No caso de um *direct exchange*, a aplicação remetente deve colocar no *routing key* o nome da fila destino da mensagem. No caso de um *topic exchange*, o nome das filas destino torna-se irrelevante e a aplicação remetente deve colocar no *routing key* o nome do tópico da mensagem.

O AMQP permite uma grande versatilidade na criação de regras de encaminhamento [O'H07]. Exemplo de isso é o tipo *headers exchange*, que examina todos os campos do cabeçalho da mensagem avaliando-os e comparando com predicados fornecidos pelas aplicações interessadas. A mensagem é transferida para a(s) fila(s) sempre que o cabeçalho da mensagem e o predicado corresponderem.

Quando o servidor arranca é automaticamente criado um conjunto de *exchanges*, os quais não poderão ser destruídos. As aplicações poderão instanciar os seus próprios *exchanges* para uso privado.

Filas de mensagens

Uma fila de mensagens (*message queue*) armazena mensagens enquanto estas não são entregues às aplicações consumidoras. As mensagens tanto podem ser guardadas em memória volátil como em memória persistente (por exemplo, discos rígidos). As mensagens são removidas das filas quando são entregues à aplicação destino, podendo a eliminação ocorrer no instante a seguir à entrega ou apenas após a aplicação consumidora confirmar o processamento da mensagem.

Cada fila de mensagens tem associado um conjunto de propriedades: *privada* ou *partilhada*, *nome atribuído pelo servidor* ou *nome atribuído pelo cliente*, *durável* ou *temporária*, são alguns exemplos. Através da combinação de diferentes valores destas propriedades, é possível criar filas com diferentes tipos de comportamento. Por exemplo, uma fila que se destine a conter mensagens consumidas por uma única aplicação para fins privados será *privada* e com *nome atribuído pelo servidor*. Uma fila destinada a armazenar mensagens de um determinado tópico do modelo publicador-subscritor será *partilhada* e com *nome atribuído pelo cliente* (considerando que existem várias aplicações subscritoras).

As filas são criadas pelas aplicações cliente e o seu tempo de vida no servidor depende da propriedade *durável* ou *temporária*. Assim, as filas duráveis são tipicamente utilizadas por mais do que uma aplicação e continuam a existir mesmo não havendo aplicações consumidoras activas. As filas temporárias são normalmente privadas e estão afectas apenas a uma aplicação consumidora. Logo, quando essa aplicação terminar a ligação ao servidor, a fila temporária é eliminada.

Bindings

O *binding* define a relação entre um *exchange* e uma fila de mensagens e fornece os critérios de encaminhamento utilizados pelo *exchange*. O conteúdo de um *binding* varia dependendo do tipo do *exchange* ao qual é associado. Por exemplo, um *binding* para um *direct exchange* requer menos informação que um *binding* para um *header exchange*. Ao receber um novo *binding*, o *exchange* actualiza as suas tabelas de encaminhamento com a informação nele contida. Por exemplo, um *binding* para um *topic exchange* contém (entre outras informações): o nome do *exchange* ao qual se destina, o valor do tópico segundo o padrão publicador-subscritor e o nome da fila para onde devem ser enviadas as mensagens onde exista correspondência entre a *routing key* e o nome do tópico.

Um *binding* é criado pela aplicação cliente (aquela que for proprietária da fila de mensagens envolvida no *binding*).

Modo automático

A utilização combinada de *exchanges*, filas e *bindings* possibilita um elevado nível de personalização do sistema de mensagens. Muitas vezes as organizações não

necessitam de soluções tão personalizadas, casos em que a configuração destes três elementos acrescenta um nível de complexidade desnecessário. O *modo automático* existe para que os cenários mais frequentes de troca de mensagens possam ser configurados de forma mais simples e consiste na existência de:

- Um *Exchange* por omissão para todos os produtores de mensagens;
- Um *binding* por omissão para cada fila de mensagem.

Quando o servidor é iniciado são imediatamente criados dois *exchanges*: um *direct exchange* e um *topic exchange*. Paralelamente é também criado um *binding* que faz a selecção de mensagens através da correspondência entre o nome da fila e o valor do campo *routing key*. Este *binding*, em conjunto com o *direct exchange*, é o utilizado por omissão cada vez que a aplicação produtora realizar o envio de uma mensagem para o servidor sem especificar nenhum *exchange*. A existência do modo automático permite às aplicações produtoras enviar mensagens sem se aperceberem da existência de *exchanges* ou de *bindings*. Do ponto de vista do emissor, a mensagem é directamente enviada para a fila destino implementando assim o modelo de troca de mensagens ponto-a-ponto.

2.7.2.2 Níveis AMQP

O AMQP encontra-se dividido em dois níveis (ou camadas):

- Nível funcional
- Nível de transporte[2]

Ao nível do transporte é definido o que as aplicações clientes e o servidor de mensagens enviam através da rede, enquanto que o nível funcional define as semânticas que uma implementação AMQP tem de obedecer para ser interoperável com outras implementações.

Nível funcional

O nível funcional define um conjunto de comandos (também designados por *métodos*) que se destinam a ser utilizados pelas aplicações cliente e que executam acções no estado do servidor. Para uma melhor organização e também para facilitar a implementação de API para aplicações cliente, os comandos definidos no nível funcional estão organizados em classes (ou categorias). Cada classe cobre um domínio específico de funcionalidades.

[2] Nesta secção utiliza-se "nível de transporte" para designar a camada "AMQP wire format" aprensentada na Figura 6 e não deverá ser estabelecida nenhuma conotação com o modelo OSI.

Entre as classes com mais relevância, destacam-se:

Connection	Para que os clientes comuniquem com o servidor, é necessário estabelecer uma ligação que dura durante toda a comunicação. A classe *Connection* contém os métodos *Open* e *Close* para iniciar e terminar ligações. Contém ainda outros métodos que permitem fazer a autenticação do cliente.
Channel	Na mesma ligação podem ser utilizados vários canais por forma a suportar diferentes contextos de comunicação na mesma ligação. A classe *Channel* contém métodos para abrir e fechar canais (dentro de uma ligação previamente criada).
Access	Os recursos (*exchanges* e filas) estão agrupados em domínios (*realms*) sujeitos a diferentes políticas de controlo de acesso. Para poder utilizar recursos, o cliente tem de pedir autorização ao servidor através do comando *Request* da classe *Access*.
Exchange	A classe *Exchange* contém os métodos que permitem criar (*Declare*) ou destruir (*Delete*) novos *exchanges*. O AMQP não faz uso do nome "*Create*", em vez de isso "*Declare*" significa "criar se não existir, caso contrário, continuar".
Queue	A classe *Queue* contém o método *Declare* para criar novas filas. Não existe comando para destruir filas explicitamente uma vez que o seu tempo de vida é determinado pelas propriedades atribuídas na criação da fila. O método *Bind* é utilizado para associar a fila a um *exchange* e definir as condições para que as mensagens desse *exchange* sejam entregues na respectiva fila.
Basic	A classe *Basic* faz parte de um conjunto de classes que processam conteúdos de mensagens e destina-se a conter comandos para processar o conteúdo de mensagens comuns. Entende-se por mensagem comum aquela cujo conteúdo não se trate de dados de *streaming* ou transferência de ficheiros (o processamento deste tipo mensagens é feito com os comandos das classes *Stream* e *File*). A classe *Basic* contém comandos para enviar mensagens (*Publish*), iniciar e parar o consumo de mensagens (*Consume, Cancel*), obter uma mensagem de forma assíncrona (*Deliver, Return*) ou obter uma mensagem de forma síncrona (*Get*).

A Listagem 2 mostra (em pseudo-código) o exemplo da criação de uma fila para funcionar no modo publicador-subscritor e onde se publica e consome uma mensagem. C1 e C2 são comandos enviados por dois clientes distintos e S são as respostas do servidor.

```
C1: Queue.Declare
      queue = <empty>          //  nome atribuído pelo servidor
      auto_delete = true       //  a fila é destruída quando não houver consumidores
S: Queue.Declare-Ok
      queue = tmp.2            //  resposta do servidor com nome da fila
C1: Queue.Bind                 //  criação de um binding
      queue = tmp.2
      to exchange =  amp.topic
      where routing_key = STOCK.USD.*
C1: Basic.Consume              //  início do consumo de msg na fila "tmp.2"
      queue = tmp.2
C2: Basic.Publish              //  envio de uma msg
      exchange = amq.topic
      routing_key = STOCK.USD.IBM
```

Listagem 2: Exemplo de um publicador-subscritor em AMQP

Apesar do conjunto de comandos definidos no nível funcional, o AMQP não define nenhuma API específica para ser utilizada ao nível das linguagens de programação. Por exemplo, em Java existe o JMS (ver capítulo 2.7.1) e é possível mapear as funcionalidades do AMQP para essa API especifica. Contudo nas restantes linguagens, a sintaxe da API ficará ao critério de quem a implementa [Vin06].

Nível de transporte

O nível de transporte encarrega-se de transportar os comandos da aplicação cliente para o servidor e, quando aplicável, trazer de volta as respostas do servidor ao cliente. Este nível trata de questões como multiplexagem de canais, construção de *frames*, codificação e representação de dados, etc.

Ao nível de transporte, o AMQP é visto como um protocolo binário onde a informação está organizada em *frames* que transportam métodos, conteúdos e outras informações. O tamanho de cada *frame* pode variar, contudo todas as *frames* obedecem ao mesmo formato geral: Cabeçalho – Corpo – Terminador.

O estabelecimento da ligação entre cliente e servidor envolve uma fase de negociação onde é acordado, entre outros aspectos, o limite máximo do tamanho de cada *frame* e o número de canais que será utilizado. Durante a negociação, ambas as

partes apresentam os seus valores e a comunicação é estabelecida utilizando o maior valor comum.

O cabeçalho de uma *frame* é composto por 8 octetos onde está incluída a dimensão do corpo da frame (*payload*). Visto que a mesma ligação pode transportar dados provenientes de vários canais, cada *frame* contém um campo com o número do canal a que os dados se referem (ver Figura 8). Tipicamente, o servidor à medida que for recebendo as *frames*, terá de criar uma máquina de estados para cada canal e processar a informação de diferentes canais de forma isolada.

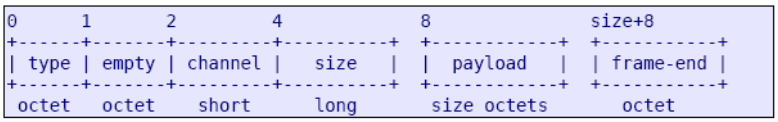


Figura 8: Formato de uma *frame*

O formato do corpo da *frame* varia conforme o tipo da *frame*. Assim, para uma *frame* que transporte um método (comando), o formato será o identificador da classe onde está inserido o método, o identificador do método e finalmente a lista dos argumentos específicos a esse método (ver Figura 9).

Figura 9: Formato do campo de dados de uma *command frame*

Alguns comandos (como por exemplo *Basic.Publish* ou *Basic.Deliver*) envolvem a transferência de conteúdos de dados para além dos parâmetros do método. A transferência de dados é feita através do envio das seguintes *frames*:

[*command frame*] [*header frame*] [*data*]

Os conteúdos de dados vão sempre em *frames* separadas e que precedem o comando. Os dados, propriamente ditos são constituídos por um conjunto de propriedades seguido de conteúdo binário. Esse conjunto de propriedades forma uma *frame* (do tipo *Header frame*) com o formato representado na Figura 10.

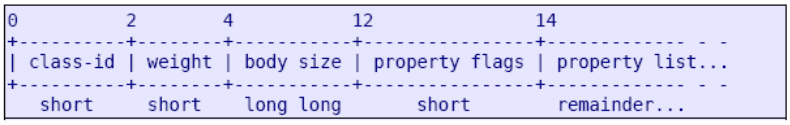


Figura 10: Formato de uma *Header frame*

A seguir às *frames* do tipo *Header* são enviados os dados binários que podem estar repartidos por zero ou mais blocos.

Tipos de dados

Os seguintes tipos de dados são utilizados no nível de transporte do AMQP:

- Inteiro – Com dimensões de 1 a 8 octetos, são utilizados para representar tamanhos, quantidades, limites, etc. Todos os inteiros são considerados sem sinal.
- Bit – Ocupam pelo menos um octeto, podendo o mesmo octeto juntar mais do que um bit com deferentes significados. Servem para representar valores *on/off*.
- *Short string* – Limitadas ao máximo de 255 octetos, servem para representar pequenas propriedades sob a forma de texto.
- *Long sting* – Utilizadas para representar grandes blocos de dados binários.
- *Field table* – Tabelas que armazenam pares nome-valor. Em cada par, o tipo do valor poderá ser uma *string*, um inteiro, etc.

2.7.2.3 Implementações

Existem actualmente várias implementações de servidores de mensagens que utilizam o AMQP como *standard* para comunicar com os clientes, entre elas destacam-se:

- OpenAMQ – desenvolvido pela iMatrix trata-se de uma implementação em C e está a ser utilizado em produção na JPMorgan (um dos parceiros na definição da norma).
- Qpid – Um projecto da Apache onde o servidor está disponível em C++ e Java e as aplicações clientes poderão utilizar C/C++, Java (JMS), Python, Ruby, C#.
- RabbitMQ – Implementação para Erlang.

2.7.3 RestMS

O RestMS [Res09] trata-se de um protocolo que permite implementar um servidor de mensagens cujas interfaces são disponibilizadas através de serviços REST [Fie00].

O RestMS surge como um protocolo de simples implementação, uma vez que utiliza funcionalidades já disponíveis nas plataformas que o implementam, como por exemplo a comunicação via HTTP. O formato dos dados nas mensagens tanto poderá

ser XML como JSON, embora presentemente a norma apenas especifique o uso de XML.

O modelo do RestMS baseia-se em três entidades principais – *Feeds*, *Joins* e *Pipes*. O enquadramento destas entidades no modelo RestMS está ilustrado na Figura 11.

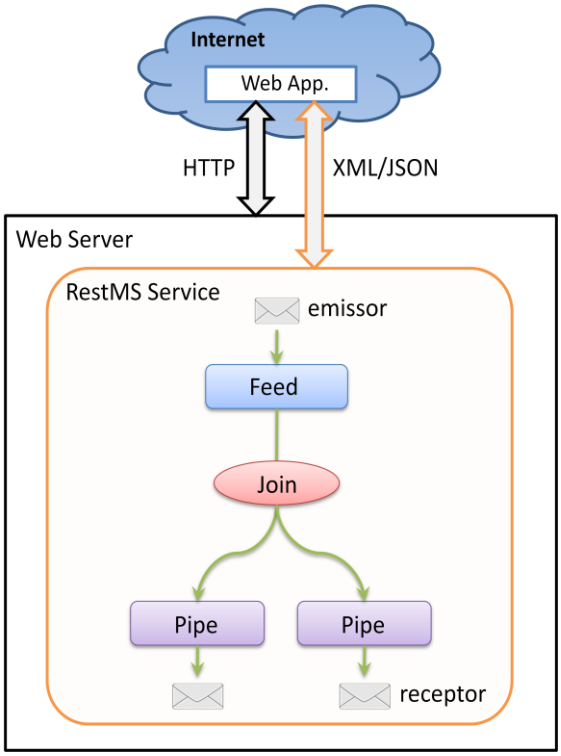


Figura 11: Modelo RestMS. Fonte: [RestMs – 2009]

Um *feed* funciona como o local onde as aplicações publicam as suas mensagens. As aplicações receptoras obtêm as mensagens dos *pipes* e os *joins* estabelecem relações entre *feeds* e *pipes* fornecendo assim os critérios de encaminhamento. O modelo tem algumas semelhanças com o modelo do AMQP (apresentado na Figura 7) onde o papel do *feed* é semelhante ao do exchange, o *join* tem o mesmo papel do *binding* e o *pipe*, tal como as *filas*, são os locais onde os clientes finais obtêm as mensagens.

Sendo um protocolo que se baseia em Rest, as aplicações cliente vêem do lado do servidor um conjunto de recursos aos quais acedem com os verbos HTTP: Get, Put, Post e Delete. Assim sendo, o RestMS define um conjunto de recursos possíveis de ser utilizados pelos clientes. Cada recurso é acedido por um URI formado da seguinte forma:

```
http://{server-name}[:{port}]/restms/{resource-type}/{resource-name}
```

2.7.3.1 Recursos RestMS

No RestMS são definidos um total de sete recursos – *Profiles*, Domínios, *Feeds*, *Pipes*, *Joins*, Mensagens e Conteúdos.

Profile

Um *profile* trata-se de um conjunto de semânticas que o servidor implementa. Na prática, é através da análise de um *profile* que os utilizadores ficam, por exemplo, a conhecer quais os tipos de *feeds* ou *pipes* que podem instanciar.

O *profile* é um recurso que apenas suporta o método GET e o URI correspondente aponta para uma página legível por um humano e onde está a especificação desse *profile*.

Domínio

Um domínio trata-se de uma colecção de *profiles*, *feeds* e *pipes* que permite organizar dentro do servidor vários recursos utilizados por diferentes clientes. Cada domínio funciona para um *feed* ou *pipe* como um espaço de nomes, sendo possível o encaminhamento de mensagens entre domínios diferentes. Recursos pertencentes ao mesmo domínio têm ainda a característica de partilhar os mesmos dados de autenticação para controlo de acesso.

As aplicações cliente e o servidor devem acordar previamente quais os domínios que estarão disponíveis, contudo o servidor deve implementar um domínio por omissão com o nome *default*.

Os domínios são criados de forma estática e os clientes não poderão adicionar novos domínios nem remover os já existentes. Desta forma, o servidor RestMS apenas permite o uso dos seguintes dois métodos no URI de um domínio:

- GET – Obtém a representação em XML do domínio (ver Figura 12), onde constam os *profiles*, *feeds* e *pipes* afectos a esse domínio.
- POST – Cria um novo *feed* ou *pipe* e adiciona-o ao domínio.

```
<?xml version="1.0"?>
<restms xmlns="http://www.restms.org/schema/restms">
  <domain title="{description}">
    [ <profile
        name="{name}"
        title="{description}"
        href="{URI}" /> ] ...
    [ <feed
        name="{name}"
        title="{description}"
        type="{type}"
        license="{license}"
        href="{URI}" /> ] ...
    [ <pipe
        name="{name}"
        title="{description}"
        type="{type}"
        href="{URI}" /> ] ...
  </domain>
</restms>
```

Figura 12: Representação de um domínio RestMS

Na representação do domínio poderão não constar todos os *feeds* e *pipes* existentes nesse domínio, isto porque este tipo de recursos pode ser criado como *privado*.

Feed

O *feed* é o recurso no qual a aplicação remetente deposita as mensagens e pode ser visto como um *stream* de mensagens onde apenas é permitido escrever. A ordem das mensagens dentro do *feed* corresponde à ordem pela qual foram adicionadas, sendo as mensagens de cada *feed* encaminhadas para um ou mais *pipes* de acordo com os *joins* definidos.

É permitido às aplicações cliente criar dinamicamente novos *feeds*, sendo para isso necessário que o cliente envie um POST para o URI um nível acima do URI do novo *feed* (*parent* URI). O conteúdo desse POST deve estar formatado de acordo com a representação XML para um *feed*, apresentada na Figura 13.

```
<?xml version="1.0"?>
<restms xmlns="http://www.restms.org/schema/restms">
  <feed
      name="{feed name}"                    mandatory feed name
    [ type="{feed type}" ]                  optional type
    [ title="{short title}" ]               optional title
    [ license="{license name}" ]            optional license name
    />
</restms>
```

Figura 13: Representação XML de um *feed* RestMS

O tipo do *feed* determina o modo como as mensagens são armazenadas e a forma como é feito o encaminhamento de mensagens. Os tipos de *feeds* permitidos dependem do *profile* implementado pelo servidor. O servidor deve implementar um conjunto de *feeds* públicos que são criados durante o arranque.

O RestMS permite o uso dos seguintes métodos no URI de um *feed*:

- GET – Obtém a representação XML do *feed*;
- PUT – Actualiza campos do *feed*. O nome e tipo do *feed* não podem ser alterados;
- DELETE – Elimina o *feed*;
- POST – Envia uma mensagem para o *feed*.

Pipe

O *pipe* é o recurso a partir do qual as aplicações cliente recebem as mensagens que lhes são destinadas e pode ser visto como um *stream* de mensagens onde apenas é permitida a leitura.

É da responsabilidade das aplicações cliente a criação dos *pipes* que estas necessitem. Para criar um novo *pipe*, o cliente terá de enviar um POST para o URI um nível acima do URI do novo *pipe*. O conteúdo desse POST deve estar formatado de acordo com a representação XML para um *pipe* indicada na Figura 14.

```
<?xml version="1.0"?>
<restms xmlns="http://www.restms.org/schema/restms">
  <pipe
    [ type="{pipe type}" ]                  optional type
    [ title="{short title}" ]               optional title
    />
</restms>
```

Figura 14: Representação XML de um *pipe* RestMS

Cada *pipe* é caracterizado por um tipo e um título. O tipo do *pipe* determina a semântica do encaminhamento de mensagens e os tipos permitidos dependem do *profile* implementado pelo servidor.

Os *pipes* criados são sempre privados pelo que não são partilháveis. Se mais do que um cliente tentar aceder ao mesmo *pipe*, nenhum dos clientes receberá a totalidade das mensagens. O servidor poderá ainda eliminar *pipes* que não estejam em uso.

O RestMS prevê os seguintes métodos para o URI de um *pipe*:

- GET – Obtém a representação XML do *pipe*;

- DELETE – Elimina o *pipe*;
- POST – Cria um novo *join* para o *pipe*.

Join

O *join* especifica os critérios utilizados por um *feed* para encaminhar mensagens para um determinado *pipe*. São sempre criados pelos clientes e são sempre privados. Para criar um *join*, a aplicação cliente envia um POST para o URI do *pipe* ao qual se pretende que o *join* fique associado. O conteúdo do POST deve estar de acordo com a representação XML do *join* apresentada na Figura 15.

```
<?xml version="1.0"?>
<restms xmlns="http://www.restms.org/schema/restms">
  <join
  [ type="{join type}" ]                    optional type
    address="{address pattern}"
    feed="{feed URI}"                       the feed to pull from
    >
    [ <header name="{header name}" value="{header value}" /> ] ...
  </join>
</restms>
```

Figura 15: Representação XML de um *join* RestMS

Um *join* tem como propriedades um tipo, um padrão de endereçamento (que será utilizado no encaminhamento) e um *feed* (aquele no qual são avaliadas as mensagens e tomada a decisão se são ou não encaminhadas para o *pipe* ao qual pertence o *join*). O *join* permite ainda condicionar o encaminhamento de mensagens ao valor de determinados campos presentes no cabeçalho da mensagem.

Se o *feed* ou o *pipe* ao qual o *join* está associado for destruído, então também o *join* será eliminado. Apenas os dois métodos que se seguem são permitidos no URI de um *join*:

- GET – Obtém a representação XML do *join*;
- Delete – Elimina o *join*.

Mensagem e Conteúdo

Mensagens e *conteúdos* são dois tipos de recursos estritamente ligados e servem para enviar e receber mensagens de e para o servidor. Um *conteúdo* trata-se de um bloco de dados com um tipo MIME definido pela aplicação remetente e que pode servir para transportar, anexada à mensagem, informação não textual como por exemplo, uma imagem ou um ficheiro binário. Por outro lado, o recurso *mensagem* trata-se de um "envelope" que se destina a conter informações relevantes para o processamento da

mensagem, nomeadamente o processo de encaminhamento. A cada *mensagem* estão associados zero ou mais *conteúdos* podendo ainda o *conteúdo* estar embebido no corpo da própria *mensagem*, dispensando a utilização de um recurso à parte.

A representação XML de uma *mensagem* (que corresponde igualmente ao conteúdo do método POST para o URI de um *feed*) tem o aspecto da Figura 16.

```
<?xml version="1.0"?>
<restms xmlns="http://www.restms.org/schema/restms">
  <message
    [ address="{address literal}" ]
    [ message_id = "{identifier}" ]
    [ reply_to="{address literal}" ]
    [ <header name="{header name}" value="{header value}" /> ] ...
    [ <content href="{content URI}" ... />
    | <content
        type="{MIME type}"
        encoding="{encoding}">{content value}</content> ]
  </message> ...
</restms>
```

Figura 16: Representação XML de uma *mensagem* RestMS

Publicação de mensagens

No processo de publicação de mensagens, a aplicação remetente começa por enviar para o servidor zero ou mais *conteúdos*. O envio de um *conteúdo* é feito com um POST para o URI do *feed* pretendido, onde o corpo do POST são os dados que formam o *conteúdo*. O formato dos dados que compõem o *conteúdo* é indicado através de um valor MIME-type incluído no campo *content type* do POST. Se não ocorrer nenhum erro, o servidor responde com o código 201 previsto protocolo HTTP (conforme a Figura 17).

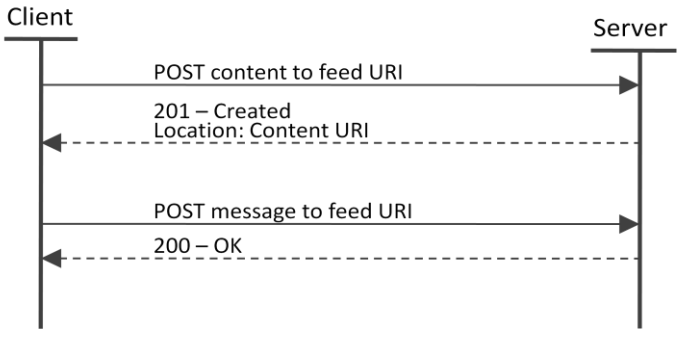


Figura 17: Publicação de mensagens em RestMS. Adaptado de: [RestMs – 2009]

Depois de o cliente transferir zero ou mais *conteúdos*, a publicação da mensagem é concluída com o envio de um POST para o URI do *feed*.

O servidor processa cada mensagem utilizando a semântica de encaminhamento associada ao *feed* correspondente e distribuindo a mensagem por zero ou mais *pipes*. Após o correcto processamento da mensagem, o servidor retorna à aplicação remetente o código 200 (Ok) previsto no protocolo HTTP.

Leitura das mensagens

Para saber se existem mensagens prontas a consumir, um cliente terá de obter o *pipe* (Figura 14). Na descrição do *pipe*, a aplicação cliente encontra a quantidade de mensagens disponíveis no *pipe* bem como os URI para aceder a cada recurso *mensagem*. O cliente envia um GET para o URI da *mensagem* e, caso existam *conteúdos* associados à *mensagem*, envia um GET para o URI de cada *conteúdo*.

Capítulo III

Arquitectura

Neste capítulo são apresentados os aspectos arquitecturais da solução implementada. O capítulo começa por descrever cada uma das funcionalidades do *bus* de mensagens, revelando os traços gerais da sua implementação (secção 3.1). Posto isto, passa-se à apresentação do formato interno utilizado dentro do *bus* (secção 3.2), dos comandos por este suportados (secção 3.4) e da arquitectura de suporte a múltiplos protocolos de comunicação (secção 3.3). No final do capítulo é apresentada a arquitectura geral, explicando como se relacionam todas as partes que compõem o *bus* de mensagens (secção 3.5).

A Figura 18 ilustra, do ponto de vista conceptual, o modo como o *bus* de mensagens e as aplicações cliente se relacionam entre si.

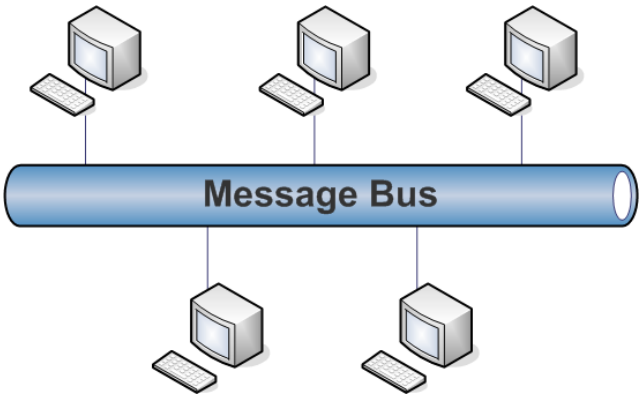


Figura 18: Modelo conceptual

É possível observar-se o papel que o *bus* de mensagens representa como intermediário na comunicação entre os vários sistemas. Utilizando um *bus* de mensagens como *middleware* comum, o desenvolvimento de cada aplicação cliente deixa de considerar aspectos como a localização física das restantes aplicações, assim como a responsabilidade de fazer chegar a essas aplicações a informação pretendida.

Existem três características fundamentais que devem existir num *bus* de mensagens [Hoh04]:

- Uma infra-estrutura comum de comunicação;
- Um formato interno para representação de dados;
- Um conjunto de comandos.

A **infra-estrutura comum de comunicação**, também conhecida por *messaging infrastructure*, trata-se de toda a plataforma que recebe e processa as mensagens, entregando-as no destino final. Esta infra-estrutura representa aquilo que as aplicações cliente vêem do *bus* de mensagens e constitui o ponto de comunicação entre *bus* e clientes. De forma a comunicar com aplicações provenientes de diferentes ambientes, a infra-estrutura proporciona interoperabilidade entre diferentes plataformas e linguagens.

Verificando-se uma variedade tão ampla no tipo de aplicações que podem ser clientes do *bus*, é espectável que cada uma utilize diferentes formatos para representar os seus dados. Para permitir ao *bus* lidar com todos os clientes, é necessário definir um **formato interno** que represente a forma como os dados enviados/recebidos pelo *bus* estão formatados. O formato interno é assim o formato a que todas as mensagens devem obedecer enquanto circulam dentro do *bus*.

As funcionalidades disponibilizadas por um *bus* de mensagens variam de implementação para implementação. Algumas dessas funcionalidades podem ser activadas/configuradas pelas aplicações cliente através do envio de comandos directamente para o *bus*. Tal característica dá origem a que um *bus* de mensagens suporte um **conjunto de comandos** conhecidos pelas aplicações cliente.

3.1 Funcionalidades implementadas

A implementação do *bus* de mensagens realizada no âmbito deste projecto inclui as seguintes funcionalidades:

- Encaminhamento (conforme secção 2.4.2);
- Transformação (conforme secção 2.4.1);
- Autenticação e segurança (conforme secção 2.4.5);
- Extensibilidade (conforme secção 2.4.3).

3.1.1 Encaminhamento

O encaminhamento consiste em calcular, para cada mensagem de dados recebida, o(s) destino(s) final dessa mensagem. De acordo com os modelos de troca de mensagens apresentados no capítulo 2.6, a implementação do *bus* suporta de raiz encaminhamentos baseados nos modelos *ponto a ponto* e *publicador/subscritor*. No entanto a solução é extensível no que diz respeito aos tipos de encaminhamento suportados.

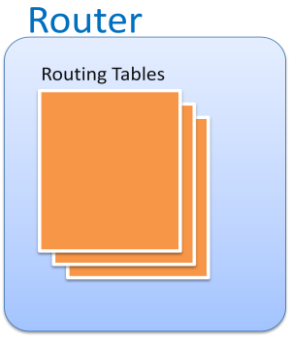


Figura 19: Arquitectura do componente de encaminhamento

O componente do *bus* responsável por realizar o encaminhamento de mensagens, recebeu o nome de *Router* e internamente contém uma ou mais tabelas de encaminhamento (ver Figura 19).

Uma tabela de encaminhamento relaciona o valor de campos específicos da mensagem com o destino que a mensagem irá tomar em função do conteúdo desses campos (para mais informações acerca da composição das tabelas de encaminhamento, consultar o capítulo Implementação na secção 4.5). Protocolos como o AMQP ou o RestMS quebram o modelo tradicional de encaminhamento (segundo o qual existe

apenas uma tabela) ao criarem respectivamente o conceito de *Exchange* e *Feed*. Nestes protocolos, quando um cliente envia uma mensagem, não a envia apenas para o sistema de mensagens mas sim para um *Exchange*/*Feed* específico. A escolha do *Exchange*/*Feed* por parte do remetente da mensagem virá posteriormente a influenciar o processo de encaminhamento, uma vez que a mensagem apenas chegará aos clientes que estabeleceram associações com determinado *Exchange*/*Feed*.

A implementação do encaminhamento realizada neste projecto segue a mesma lógica existente no AMQP e RestMS, permitindo às aplicações remetentes indicarem qual a tabela de encaminhamento que será utilizada no processamento da mensagem. Por esta razão, o componente *Router* pode conter mais que uma tabela de encaminhamento. Contudo, para alguns casos, e também para protocolos de comunicação diferentes, este pode ser um detalhe que não interessa expor às aplicações cliente. Assim, existe uma tabela de encaminhamento por omissão e que é a utilizada sempre que as aplicações não indicarem nenhuma outra.

As tabelas de encaminhamento podem ser de dois tipos; *um para um*, onde cada linha associa o valor de um campo da mensagem a apenas um destino ou *um para muitos*, onde cada linha da tabela associa o valor de um campo da mensagem a vários destinos.

3.1.2 Transformação

A transformação de mensagens é o principal meio para que aplicações baseadas em diferentes plataformas e utilizando diferentes semânticas, possam comunicar entre si.

A implementação realizada do *bus* de mensagens contém o componente *Transformer* (ver Figura 20) cuja função é verificar a necessidade de uma mensagem ser transformada e aplicar essa mesma transformação.

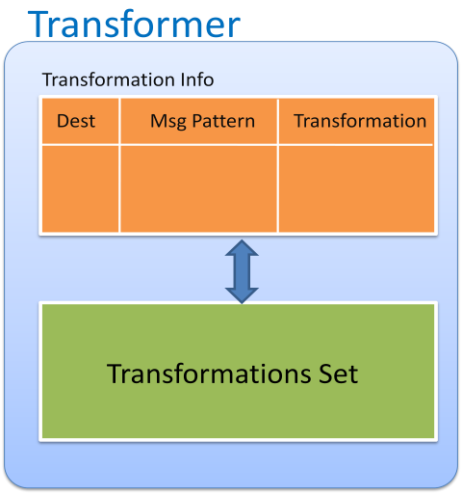


Figura 20: Arquitectura do componente de transformação

Do ponto de vista do remetente da mensagem, a transformação deve ser algo transparente, que ocorre em função do destino e do conteúdo da mensagem. Desta forma, o componente *Transformer* contém internamente uma tabela onde se relaciona o destino de uma mensagem, com uma dada transformação de dados e com um padrão que a mensagem tem de verificar para que a transformação seja aplicada. O padrão consiste num algoritmo que, depois de aplicado à mensagem e juntamente com a informação do destino da mensagem, determina se esta será ou não transformada. A Figura 21 mostra o exemplo de uma mensagem de dados e um padrão de texto. No caso deste padrão, representou-se uma sequência de caracteres por "{*}" e verifica-se correspondência sempre a mensagem tiver conteúdo XML onde o primeiro elemento tenha o nome *order* e o atributo *type* com o valor "*catalog*".

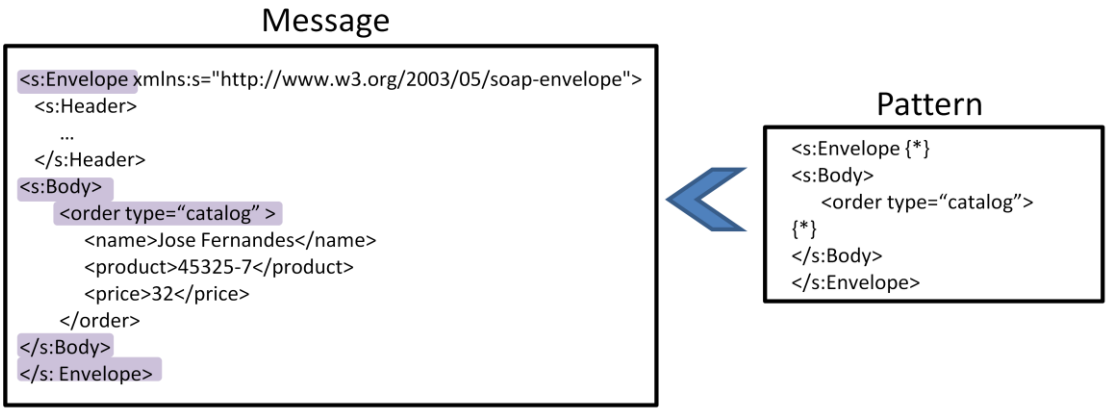


Figura 21: Exemplo de um padrão de texto

Para transformar a mensagem existe um conjunto de algoritmos (transformadores) que estão associados com a tabela atrás referida, que contém a restante informação. Esse conjunto de transformações inclui, na implementação base, três transformadores:

- XSLT
- XML to JSON
- JSON to XML

O primeiro (XSLT) trata-se de um transformador que efectua transformações ao *nível dos dados*. Aplica-se apenas a mensagens recebidas com conteúdo XML e utiliza ficheiros XSLT para alterar o conteúdo das mensagens.

Os restantes transformadores efectuam transformações ao *nível da representação de dados.* Desta forma, altera-se o próprio formato dos dados permitindo conversões de JSON para XML e vice-versa.

Apesar de não ser uma característica específica do componente de transformação, o *bus* de mensagens permite ainda transformações ao *nível do transporte*. Este tipo de transformações é suportado pela utilização de múltiplos protocolos de comunicação (ver capitulo 3.3). Assim, por exemplo, no caso de um cliente AMQP (que utiliza como canal de transporte uma ligação TCP/IP) que envie uma mensagem consumida por um cliente RestMS (que comunica através de HTTP), existe uma conversão implícita entre comunicações TCP e HTTP.

3.1.3 Autenticação e segurança

No que refere às áreas de autenticação e segurança, existem no *bus* de mensagens dois principais requisitos:

- Permitir que as mensagens trocadas entre um determinado cliente e o *bus* contenham o *payload* cifrado;
- Controlar o acesso a determinado cliente por parte dos restantes.

O primeiro requisito define uma espécie de canal seguro entre o *bus* e as aplicações cliente, visto que, apesar de não existir nenhum mecanismo extra para evitar que um atacante consiga “roubar” uma mensagem durante o transporte, caso tal aconteça, é garantido que os dados dessa mensagem permanecerão em segredo. Esta

funcionalidade aplica-se a todas as mensagens enviadas, ou recebidas, pelo cliente assim configurado. O benefício que a utilização de um *bus* de mensagens oferece em termos de desacoplar sistemas, implica que uma aplicação remetente envie uma mensagem sem saber se o destino dessa mensagem requer ou não cifra de dados. Desta forma, terá de ser o *bus* a verificar em que situações é necessário cifrar/decifrar o conteúdo das mensagens mantendo este processo transparente para o utilizador.

O segundo requisito consiste em permitir que um cliente indique explicitamente quais são as aplicações (clientes) que estão autorizadas a enviar mensagens. Tal medida implica criar um mecanismo de autenticação que permita identificar de forma inequívoca a identidade de todos os clientes.

A Figura 22 mostra a arquitectura do componente do *bus* que trata da autenticação de clientes e cifra de dados e ao qual se chamou *Security*.

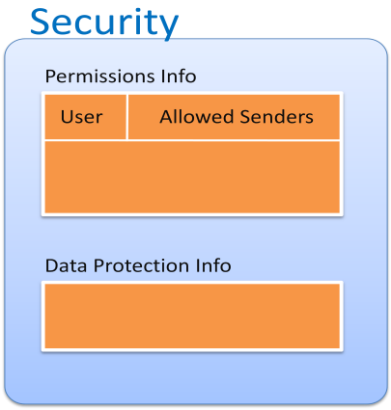


Figura 22: Arquitectura do componente de segurança

Internamente é mantida informação dos clientes para os quais é necessário proteger o *payload* de todas as mensagens enviadas e recebidas (em *Data Protection Info*) e também uma tabela onde consta, para cada utilizador, a lista das aplicações que estão autorizadas a enviar mensagens (em *Permissions Info*).

Os mecanismos de autenticação existem para proteger as aplicações cliente e não o *bus*. Quer isto dizer que os clientes podem, se assim o pretenderem, utilizar o *bus* sem nenhum mecanismo de autenticação/segurança quer para enviar comandos/mensagens de dados ou receber mensagens de outros clientes.

Para detalhes acerca da implementação da autenticação e cifra de dados, consultar a secção 4.7.

3.1.4 Extensibilidade

Uma das funcionalidades oferecidas pelo *bus* de mensagens consiste em estender as próprias funcionalidades do *bus* de mensagens para além daquelas que foram atrás descritas. Assim, será possível alterar o actual modo como é feito o processamento de uma mensagem acrescentando-lhe novos pontos de processamento onde será possível, por exemplo, consultar a informação de uma mensagem, alterar os dados de uma mensagem ou retirar uma mensagem do processamento do *bus*. A extensibilidade é conseguida através de um componente que permite aos administradores acrescentar ao *bus* novos módulos de código. O momento em que esses módulos participam no processamento das mensagens será configurável.

Esta funcionalidade cumpre os mesmos requisitos que um *motor de processamento de regras* apresentado na capítulo 2.4.3.

3.2 Formato interno

De um ponto de vista geral, tanto as mensagens de dados como os comandos enviados para o *bus*, são blocos de informação que se destinam a ser processados. No entanto, analisando com maior detalhe, estes blocos de informação estão estruturados. Podem internamente estar divididos em vários campos ou conter cabeçalhos que ajudam a interpretar o seu conteúdo. Várias formas existem para representar esta informação, desde formatos binários onde a cada byte é atribuído um determinado significado, até formatos de texto, como por exemplo o *Comma-separated Values* (CSV), onde cada campo é codificado com uma *string* e separado dos restantes campos pelo carácter ‘,’.

Tratando-se o *bus* de mensagens de uma ferramenta de integração, é espectável encontrar no conjunto de todos os clientes várias formas distintas de representar mensagens de dados e comandos. Independentemente do formato utilizado pelas aplicações cliente, internamente o *bus* de mensagens faz uso de apenas um formato. Para todos os clientes que utilizem um formato diferente daquele que é utilizado pelo *bus*, terá de ser feita uma conversão na entrada/saída de mensagens.

Na escolha do formato interno do *bus* de mensagens há que ter em conta a interoperabilidade entre diferentes plataformas. Deste ponto de vista, a escolha de um formato baseado em texto será mais facilmente interoperável do que um formato binário

[Res08]. Outra vantagem na escolha de um formato de texto é o facto de as mensagens poderem ser interpretadas por um humano sem ser necessário nenhuma outra máquina.

De entre os formatos de texto, foi dada preferência ao XML pelo facto de já existirem várias bibliotecas que auxiliam a tarefa de leitura e escrita de documentos XML. Em alternativa à criação de um idioma XML de raiz optou-se por adoptar para idioma interno do *bus* o SOAP [Spe07].

> *"O SOAP constitui uma forma simples de trocar informação de forma descentralizada em ambientes distribuídos. O SOAP utiliza tecnologias XML para definir uma plataforma de messaging extensível que fornece criação de mensagens capazes de serem trocadas sob uma variedade de protocolos."* [Spe07]

Uma das principais vantagens da utilização do SOAP é a extensibilidade que este oferece. Com esta característica é possível, por exemplo, acrescentar novos cabeçalhos a uma mensagem SOAP sem que tal implique alterações no código das aplicações que já utilizavam esse formato de mensagem. Quando comparado com um formato binário onde os campos são formados por várias sequências de bytes criteriosamente ordenadas (como por exemplo no caso do AMQP), a mesma tarefa de acrescentar um novo cabeçalho poderá ser bastante mais complexa, caso a especificação binária não tenha previsto nenhuma zona de extensibilidade. No entanto, refira-se que as mensagens de formatos binários ocupam geralmente menos espaço de que mensagens em formatos de texto.

3.3 Protocolos de comunicação

Um protocolo de comunicação trata-se de um acordo com o cliente na forma como este comunica com o sistema de mensagens e vice-versa. O protocolo de comunicação define, não só o formato no qual os dados são transportados, mas também quais as características do canal de transporte. Num sistema de integração, como um *bus* de mensagens, é importante o suporte de protocolos de comunicação *standard*, de forma a minimizar as alterações necessárias nas aplicações sempre que se pretender adicionar uma aplicação ao *bus*. Por outro lado, quanto maior for o número de protocolos de

comunicação suportados por um sistema de mensagens, maior será a capacidade de integração que este consegue oferecer [Res08].

Na implementação feita, são suportados três protocolos de comunicação, dois *standards* e um proprietário. Os protocolos *standard* suportados são o AMQP (secção 2.7.2) e o RestMS (secção 2.7.3). Para além destes dois, foi criado um protocolo especifico para este *bus* de mensagens a que se chamou *MB Protocol*. O *MB Protocol* funciona com base na troca de mensagens SOAP e tira partindo do facto do SOAP ser o formato interno do *bus* para minimizar as conversões entre diferentes formatos.

A tecnologia escolhida para implementar as interfaces de comunicação entre *bus* de mensagens e aplicações cliente foi o *Windows Communication Foundation* (WCF). As funcionalidades do *bus* de mensagens são assim vistas pelos clientes como um conjunto de serviços que processam mensagens SOAP.

Sendo a tecnologia de comunicação com o exterior o WCF, pretendeu-se que todos os protocolos de comunicação comuniquem com o *bus* através de WCF. A Figura 23 mostra de que modo é suportado através de WCF cada um dos três protocolos implementados.

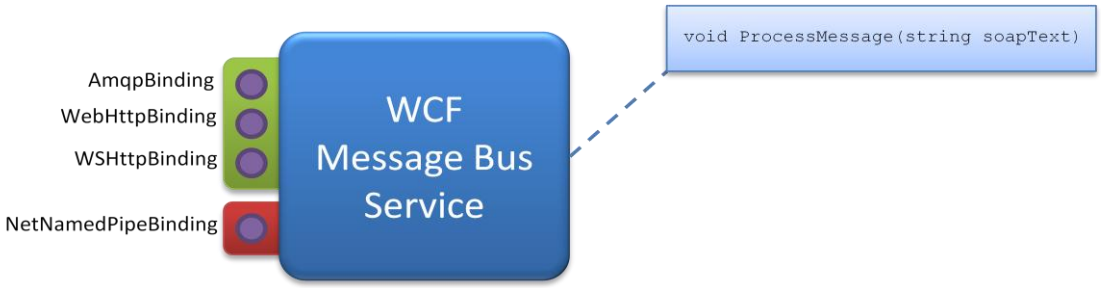


Figura 23: Protocolos de comunicação suportados

O AMQP é um protocolo que utiliza como canal de transporte uma ligação TCP/IP. O *binding* oferecido pelo WCF mais próximo deste tipo de ligação é o *NetTcpBinding*, no entanto, a este *binding* estão associados *encodings* proprietários do WCF. Para ter uma ligação TCP/IP onde os bytes enviados e recebidos sejam interpretados como mensagens AMQP houve necessidade de implementar um *user-defined Binding* com canal de comunicação e *encoding* próprios. A este *binding* criado chamou-se *AmqpBinding*.

No caso do RestMS, sendo um serviço REST, a sua implementação tem de ser feita com base no processamento de pedidos HTTP feitos para um URL específico. O *binding* que melhor se ajusta a estes requisitos é o *WebHttpBinding*. Note-se que o

serviço WCF disponibilizado, apenas contém o método *ProcessMessage* que recebe uma *string* com o conteúdo de uma mensagem SOAP de acordo com o formato interno do *bus*. Desta forma, é da responsabilidade das implementações dos protocolos de comunicação fazer a conversão entre os formatos do protocolo e o formato interno do *bus* de mensagens.

O protocolo de comunicação proprietário (*MB Protocol*) é disponibilizado através do *binding WSHttpBinding*. A escolha deste *binding* está relacionada com o papel actualmente desempenhado pelos *Web Services* como ferramenta suportada por praticamente todas as plataformas. Assim, aplicações que pretendam utilizar este protocolo, poderão estar implementadas numa grande variedade de plataformas e linguagens.

Para além dos três protocolos de comunicação suportados de origem, novos protocolos poderão ser adicionados. Para esse efeito, disponibilizou-se um *binding* extra do tipo *NetNamedPipeBinding*. As novas implementações de protocolos devem disponibilizar o aceso da forma conveniente a esse protocolo, receber as comunicações dos clientes e traduzi-las em mensagens SOAP de acordo com o formato interno do *bus* de mensagens. A mensagem SOAP obtida é então enviada para o *bus* utilizando o *binding NetNamedPipeBinding*. A escolha do tipo deste *binding* prende-se com o facto de este ser o *binding* mais rápido de entre todos os suportados em WCF [Res08]. No entanto, com isto está-se a assumir que a implementação do novo protocolo de comunicação está implementada em *.NET* e está a ser executada na mesma máquina que o servidor WCF do *bus* de mensagens. No caso de algum destes pressupostos não se verificar, os novos protocolos poderão sempre utilizar o *WSHttpBinding* visto que este também recebe as mensagens SOAP já no formato interno do *bus*.

Uma vantagem inerente ao suporte de vários protocolos de comunicação, é o facto de o *bus* de mensagens poder integrar aplicações clientes de diferentes protocolos. Desta forma será possível a um cliente AMQP enviar para o *bus* uma mensagem que será posteriormente consumida por um cliente RestMS ou *MB Protocol* sem que nenhuma das aplicações tenha consciência de estar a comunicar com aplicações de diferentes protocolos. (ver Figura 24)

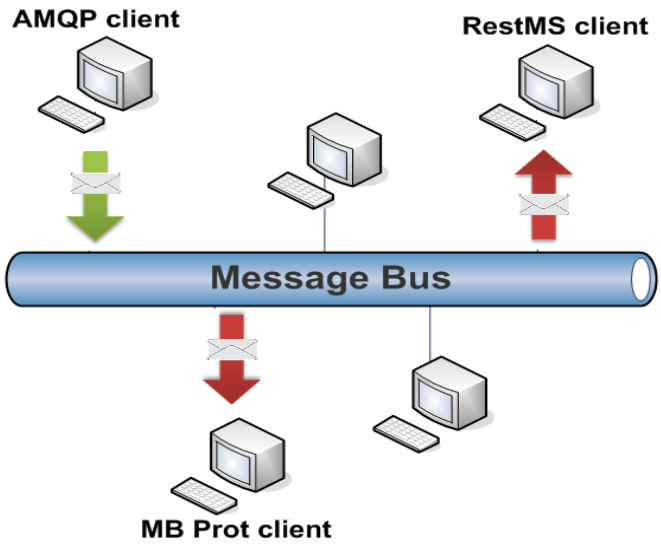


Figura 24: Integração entre vários protocolos

3.4 Comandos

As mensagens enviadas para o *bus* podem ser de dois tipos – mensagens de dados ou mensagens de comandos. No caso de serem mensagens de dados têm como destino outras aplicações clientes do *bus*. No caso de serem mensagens de comandos, o destino é o próprio *bus* e têm como objectivo activar/desactivar ou configurar alguma das funcionalidades apresentadas no capítulo 3.1. A existência de mensagens distintas e com objectivos diferentes das mensagens de dados, corresponde à implementação do padrão *Command Message* descrito em [Hoh04]. A distinção entre os dois tipos de mensagens é feita através do cabeçalho *MessageType* incluído na mensagem SOAP.

De seguida apresenta-se a lista de todos os comandos existentes e uma descrição. A lista foi agrupada segundo os componentes existentes no *bus* de mensagens.

Router

`addRoutingTable`

O comando `addRoutingTable` cria uma nova tabela de encaminhamento. Como parâmetros, contém o nome pelo qual a tabela será conhecida no *bus* e o seu tipo. Os valores aceites para o tipo da tabela são “point-to-point”, caso em que será criada

uma tabela *um para um*, ou poderá ser do tipo "pub-sub", caso em que será criada uma tabela *um para muitos*.

`addRoute`

O comando `addRoute` adiciona uma entrada numa tabela de encaminhamento.

`getRoutingTablesInfo`

O comando `getRoutingTablesInfo` é utilizado para obter uma lista de todas as tabelas de encaminhamento existentes, retornando uma lista de pares *nome, tipo*.

`updateRoutingTable`

Para actualizar o nome ou o tipo de uma tabela de encaminhamento, utiliza-se o comando `updateRoutingTable`.

`removeRoute` e **`removeRoutingTable`**

Os comandos `removeRoute` e `removeRoutingTable` permitem respectivamente eliminar uma entrada de uma tabela de encaminhamento ou eliminar a própria tabela de encaminhamento.

Transformer

`addJsonToXmlTransformation`

O comando `addJsonToXmlTransformation` é utilizado pelas aplicações para configurar uma transformação de dados JSON para XML. Para além do nome do destino que a mensagem tem para que seja aplicada a transformação, é ainda incluída uma expressão à qual a mensagem tem de corresponder.

`addXmlToJsonTransformation`

Semelhante ao comando anterior, o `addXmlToJsonTransformation` tem apenas a diferença de se aplicar a transformações de dados XML para JSON.

`addXsltTransformation`

O comando `addXsltTransformation` destina-se a configurar uma transformação de dados XML utilizando um ficheiro XSLT. Comparativamente aos comandos anteriores só tem a diferença de um parâmetro extra que contém os dados XSLT

É possível ter várias transformações para a mesma mensagem. Porque a ordem pela qual as transformações são aplicadas pode ser importante, todos os comandos para adicionar transformações incluem o parâmetro *priority* que contém um inteiro utilizado para indicar a prioridade que essa transformação tem em relação às outras. Uma transformação com valor de prioridade zero será a primeira a ser executada e só então serão executadas as seguintes por ordem crescente do valor de prioridade. Para transformações com igual nível de prioridade, não é garantido qual delas será executada em primeiro lugar.

Security

`addAllowedSender` e **`removeAllowedSender`**

O par de comandos `addAllowedSender` e `removeAllowedSender` é utilizado para adicionar ou remover nomes de clientes que estão autorizados a enviar mensagens para um determinado destino. Como parâmetros recebem o nome do destino para o qual se pretende efectuar o controlo de recepções e o nome do cliente que está (ou deixa de estar) autorizado. Recebem ainda um conjunto de parâmetros de segurança cujo funcionamento será explicado no capítulo de Implementação.

`setSecureCommunication`

O `setSecureCommunication` é o comando utilizado para uma aplicação cliente indicar que pretende que todas as mensagens trocadas com o *bus* contenham o *payload* cifrado. O mesmo comando serve para desactivar a funcionalidade, através do parâmetro *encriptyData* a receber o valor *"false"* em vez de *"true"*.

Finalmente, existem ainda os seguintes comandos que não estão afectos a nenhum componente em específico.

`addUser` e `removeUser`

Para adicionar e remover utilizadores, as aplicações cliente dispõem respectivamente dos comandos `addUser` e `removeUser`. Cada utilizador é identificado por um nome que está incluído em parâmetros do comando. O *bus* de mensagens garantirá que apenas uma aplicação utiliza um determinado nome de utilizador.

`getUsers`

O comando `getUsers` é utilizado para se obter uma lista de todos os utilizadores registados no *bus* de mensagens.

`setEndpointState`

Uma aplicação pode, num dado momento, estar ou não disponível para receber mensagens. O comando `setEndpointState` faz uso do parâmetro *state*, que assume o valor de *"ready"* ou *"buzy"*, para indicar ao *bus* se uma aplicação cliente está ou não disponível.

`getUserMessagesList`

O comando `getUserMessagesList` contém apenas um parâmetro com o nome de um utilizador e retorna a lista das mensagens pendentes para esse utilizador. Na lista não consta a mensagem inteira mas sim um identificador que deverá ser utilizado para obter uma mensagem específica.

`getMessage`

Para obter uma mensagem, uma aplicação cliente pode enviar o comando `getMessage` incluindo num parâmetro o identificador da mensagem que pretende receber. Contudo, a utilização deste comando não é a única forma que as aplicações têm de receber mensagens.

Os comandos acima descritos são os existentes na versão inicial do *bus* de mensagens. A implementação do *bus* apresenta uma arquitectura que permite que sejam

criados novos comandos caso seja necessário. A estes novos comandos dá-se o nome de *custom commands*.

A Figura 25 ilustra a sequência de comandos utilizada por duas aplicações (*client A* e *Client B*) para se registarem no *bus* e onde o *cliente A* envia uma mensagem que será consumida pelo *cliente B*.

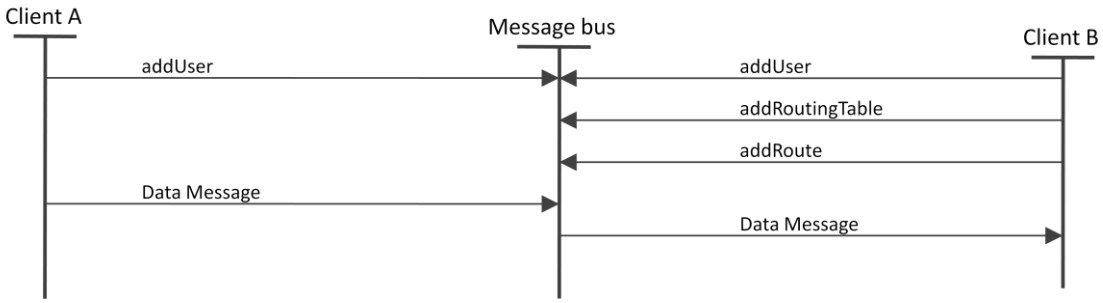


Figura 25: Exemplo de utilização dos comandos

A mensagem *DataMessage* trata-se de uma mensagem de dados e não de um comando. Neste exemplo, o cliente B recebe automaticamente a mensagem, no entanto, dependendo das características do protocolo de comunicação, a entrega pode apenas ser feita a pedido do cliente B.

A existência de um formato interno no *bus* de mensagens implica que os comandos existentes em cada protocolo de comunicação tenham de ser mapeados para os comandos atrás apresentados. O primeiro passo para realizar esta tarefa no caso dos protocolos AMQP e RestMS foi criar correspondências entre as entidades existentes nos modelos do AMQP e RestMS (ver Figura 7 e Figura 11) e os componentes existentes no *bus* de mensagens. Assim, estabeleceu-se que um *Exchange* e um *Feed* desempenham o papel de uma *tabela de encaminhamento*. Os conceitos de *binding* e *join* correspondem a entradas de uma tabela de encaminhamento. Finalmente, os conceitos de *queue* e *pipe* representam os clientes do *bus* e cujos nomes são tratados como *nomes de utilizador*.

Posto isto, estabeleceu-se uma correspondência entre comandos AMQP e comandos do *bus* de mensagens (que são também os comandos do *MB Protocol*). A Tabela 1 mostra essa mesma correspondência[3].

[3] Na Tabela 1apenas constam os comandos invocados no sentido cliente – bus de mensagens.

Comando AMQP	Comando MB Protocol
Exchange.Declare	addRoutingTable
Exchange.Delete	addRoutingTable
Queue.Declare	addUser
Queue.Delete	removeUser
Queue.Bind	addRoute
Queue.Unbind	removeRoute
Basic.Consume	setEndpointStateCmd
Basic.Cancel	setEndpointStateCmd
Basic.Publish	Envio de uma mensagem de dados

Tabela 1: Mapeamento entre comandos AMQP e MB protocol

Num processo semelhante, fez-se o mapeamento entre os "comandos" do RestMS e os comandos do *bus* de mensagens, cujo resultado se encontra na Tabela 2[4].

Recurso	**Verbo HTTP**	**Comando MB Protocol**
Domain	Get	getUsers getRoutingTablesInfo
	Post	addUser ou addRoutingTable
Feed	Get	getRoutingTablesInfo
	Post	Envio de uma mensagem de dados
	Put	updateRoutingTable
	Delete	removeRoutingTable
Pipe	Get	getUserMessagesList
	Post	addRoute
	Delete	removeUser
Join	Delete	removeRoute
Message/Content	Get	getMessage

Tabela 2: Mapeamento entre comandos RestMS e MB Protocol

[4] Os verbos *get* de Join e *remove* de Message não são propagados para o bus (executam-se localmente).

3.5 Arquitectura geral

Neste capítulo apresenta-se a arquitectura geral do *bus* de mensagens implementado. Durante a implementação, houve uma preocupação para que a arquitectura final fosse modular onde cada módulo implemente uma característica específica e apresente um baixo acoplamento com os módulos restantes.

Como se pode verificar na Figura 26, em grande parte dos casos, a cada componente do *bus* apresentado capítulo 3.1 corresponde um módulo na arquitectura geral

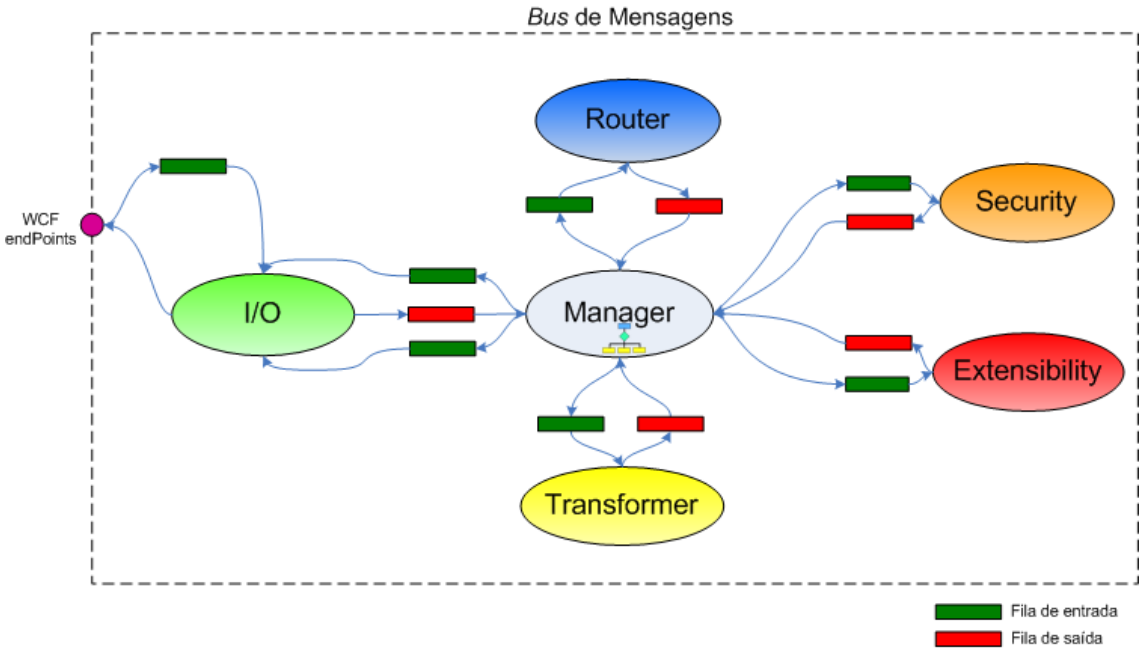


Figura 26: Arquitectura geral

Uma das principais características do modelo da arquitectura geral é o facto que cada módulo se executar num processo dedicado. Desta forma, cada módulo terá os seus recursos (por exemplo, de memória ou *threads* utilizadas da `ThreadPool`) independentes, evitando-se que o congestionamento de um dos módulos interfira com os restantes.

Cada módulo comunica com os restantes utilizando filas MSMQ que podem ser *filas de entrada* ou *filas de saída* de dados. Com excepção do módulo *I/O*, todos os módulos contêm apenas duas filas para comunicar com o exterior – uma fila onde se recolhem dados destinados a serem processados pelo módulo e outra fila onde são depositados esses mesmos dados depois de processados. A utilização de filas MSMQ pressupõe a existência de um "acordo" relativamente ao tipo e formato dos dados

colocados nas filas. Assim, uma vez cumprido esse acordo, para o módulo que processa determinada mensagem, não importa saber qual foi o módulo responsável pela colocação da mensagem, bem como qual será o módulo que irá consumir o resultado do processamento dos dados. Desta forma, a utilização de filas MSMQ como ponto de comunicação entre módulos, permite um baixo acoplamento entre as várias partes do sistema. [Spe02]

Outra funcionalidade oferecida pelas filas MSMQ é o acesso remoto, o que significa que o módulo que coloca dados na fila de entrada de outro módulo não tem necessariamente de estar a ser executado na mesma máquina. Com isto permite-se que o próprio *bus* de mensagens seja uma solução distribuída por várias máquinas. Outra característica é a possibilidade de alguns módulos poderem ter várias instâncias a correr em simultâneo (na mesma máquina ou em máquinas diferentes).

Este tipo de arquitectura onde módulos independentes comunicam através de filas MSMQ constitui uma implementação do padrão *Pipes and Filters* [Hoh04]. Este padrão aplica-se em casos onde a ocorrência de um determinado evento desencadeia uma sequência de passos de processamento; sendo neste caso o evento, a recepção de uma mensagem e os passos de processamento, o encaminhamento, transformação, etc.. Segundo [Hoh04], o padrão *Pipes and Filters* permite que o processamento de uma mensagem seja feito em *pipeline*. Em oposição ao *processamento sequencial* (onde se processa apenas uma mensagem de cada vez), o *processamento em pipeline* consiste em utilizar os vários módulos para processar em simultâneo várias mensagens. Desta forma, enquanto se calculam os destinos de uma mensagem (no módulo de encaminhamento), outra pode estar a ser transformada e enquanto numa terceira mensagem cifra o seu conteúdo. O processamento em *pipeline* tem a vantagem de possibilitar um melhor aproveitamento dos recursos do *bus* de mensagens, contudo, tem a desvantagem de não garantir, no destino final, a entrega das mensagens pela ordem com que foram enviadas. Isto porque, dependendo das características das mensagens e dos próprios clientes, o processamento de uma mensagem não envolve sempre a passagem pelos mesmos módulos. Se a garantia de ordem for um requisito para as aplicações cliente, então terão de ser estas a utilizar um mecanismo externo.

3.5.1 Módulo de entrada/saída

O módulo I/O que se observa na Figura 26 tem o seu nome derivado de "*Input/Output*" e constitui sempre o primeiro e o último módulo envolvido no processamento de uma mensagem (assumindo que a mensagem foi processada sem erros, não sendo descartada por nenhum outro módulo).

Internamente está dividido em duas partes – uma que trata da recepção de mensagens, o *Receiver* e outra que trata do despacho de mensagens, o *Dispatcher* (ver Figura 27).

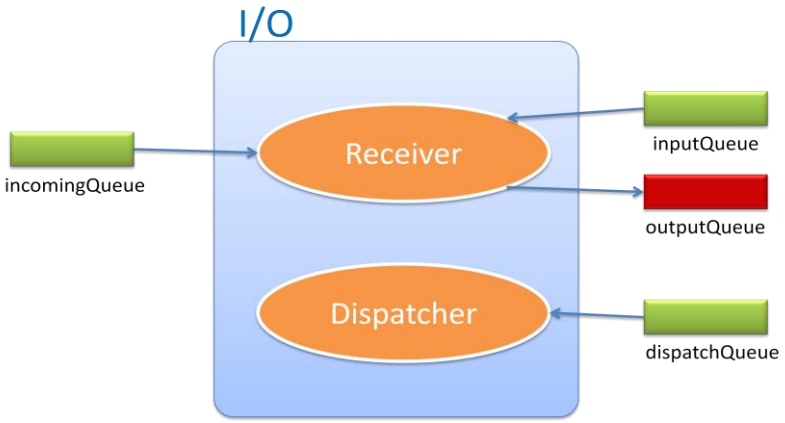


Figura 27: Módulo I/O

Os serviços WCF ilustrados na Figura 23, e que implementam os protocolos de comunicação, fazem a conversão entre o formato utilizado por um dado protocolo e o formato interno em SOAP. A mensagem SOAP daí resultante é colocada na fila MSMQ *incomingQueue*. O *Receiver* recolhe as mensagens da *incomingQueue*, prepara-as para serem processadas e coloca-as na *outputQueue*, onde ficarão disponíveis para funcionalidades como o encaminhamento ou a transformação.

O *Dispatcher* está vocacionado para proceder à entrega de mensagens aos destinos finais. Dispõe de informações acerca de cada protocolo de comunicação e sabe como comunicar como as aplicações cliente dependendo do protocolo que estas utilizem. O *Dispatcher* consome dados da *dispatchQueue* que se trata de uma fila MSMQ dedicada para as mensagens que estão prontas a ser entregues.

A *inputQueue* constitui a fila geral de entrada do módulo e, visto que não recebe nem mensagens vindas dos clientes nem mensagens prontas a entregar (para estes casos existem as filas *incomingQueue* e *dispatchQueue*), as únicas mensagens que dão entrada na *inputQueue* são comandos para serem executados no módulo.

3.5.2 Sequência de processamento

Se cada módulo funciona como um fornecedor de determinada funcionalidade, é necessário que exista algures a lógica que determina quando e de que forma é que cada módulo entra em acção. Existem duas alternativas – ou se distribui essa lógica pelos vários módulos, implementado assim um processo de *coreografia*, ou se cria um ponto central de decisão que decide quando é que cada módulo participa no processamento das mensagens, implementado assim um processo de *orquestração*. [Ros08]

No caso de *coreografia*, todas as mensagens recebidas pelo módulo *I/O* seriam entregues àquele que fosse determinado a ser o primeiro nó de processamento, por exemplo o *Router*. Se fosse estabelecido que, a seguir ao encaminhamento, o próximo passo é a transformação de mensagens, então seria o próprio módulo *Router* a saber que todas as mensagens de saída seriam entregues ao módulo *Transformer*. Por sua vez o *Transformer* passaria as mensagens, por exemplo, ao *Security* e por ai em diante até que tivesse sido concluído o ciclo de processamento da mensagem.

Neste trabalho optou-se por uma técnica de *orquestração* que passou pela criação de um módulo adicional ao qual se chamou *Manager*. Este módulo tem como *input* todas as filas de saída dos restantes módulos e sempre que recebe uma mensagem, analisa-a determinando as suas necessidades e entregando-a ao módulo correspondente. Internamente contém um *Windows Workflow Foundation* que permite configurar graficamente a ordem pela qual os módulos participam no processamento das mensagens. O *workflow* traz a vantagem de permitir a alguém, que não está familiarizado com a linguagem de programação do *bus* de mensagens, fazer alterações ao modo como o *bus* internamente processa as suas mensagens.

Comparativamente à opção de uma técnica de *coreografia* versus *orquestração*, numa *coreografia* é mais fácil acrescentar novos participantes (módulos) no processamento das mensagens [Ros08], contudo considerando o caso de um *bus* de mensagens, não é espectável que o número de módulos cresça em larga escala. Por outro lado, ao se optar por uma *orquestração* torna-se mais simples fazer alterações na ordem de participação de cada módulo, isto porque toda a lógica está localizada num só ponto. Também é importante ter em conta que, utilizando *orquestração*, alterações no *workflow* não obrigam à recompilação dos restantes módulos.

Capítulo IV

Implementação

Neste capítulo é feita uma descrição mais aprofundada sobre a implementação do *bus* de mensagens. No início do capítulo são explicados aspectos gerais à implementação de todos os módulos (secção 4.1). De seguida, isoladamente para cada módulo, é feita a descrição dos detalhes específicos de implementação, começando pelo módulo de entrada/saída que inclui a implementação dos protocolos AMQP e RestMS.

O capítulo termina com a implementação de *custum commands* (secção 4.10) e o suporte de tolerância a falhas (secção 4.11).

4.1 Suporte à implementação de módulos

Os módulos que constituem o *bus* de mensagens e que estão apresentados na Figura 26 partilham em comum vários detalhes de implementação. Por exemplo, todos os módulos são executados no contexto de um processo dedicado e, com excepção do *Manager*, todos contêm pelo menos uma fila de entrada e outra de saída (ver Figura 28).

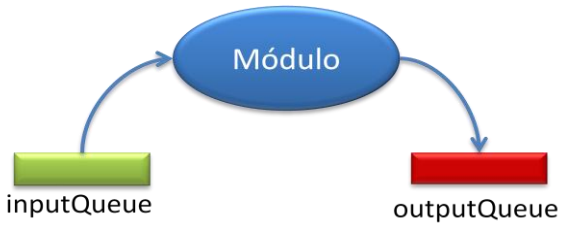


Figura 28: Visão genérica de um módulo

A leitura de dados das filas MSMQ é sempre feita de forma assíncrona e utilizando as *threads* disponíveis na `ThreadPool` permitindo assim a um módulo

processar vários pedidos em simultâneo. Para permitir uma fácil configuração de cada módulo, a localização (endereço) das filas do *bus* de mensagens está reunida na classe `MB_Parameters`.

De forma a conseguir uma melhor estruturação do código produzido, foi criada a classe `MBusModule` (Figura 29) da qual derivam as implementações de cada módulo (com excepção do *Manager*).

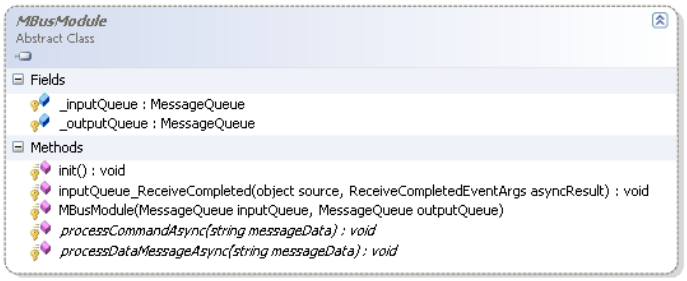


Figura 29: Diagrama de classes de *MBusModule*

A classe `MBusModule` é abstracta e contém dois campos do tipo `MessageQueue` correspondentes às filas de entrada e saída. O método `init` coloca o módulo em funcionamento dando início à leitura assíncrona de mensagens da fila de entrada. Quando uma mensagem der entrada nesta fila, é executado o método `inputQueue_ReceiveCompleted` que analisa a mensagem distinguindo se esta se trata de uma mensagem de dados ou de um comando. Conforme o caso, será invocado o método `processDataMessageAsync` ou `processCommandAsync`, ambos abstractos, cabendo à classe derivada a sua implementação de acordo com a função do módulo.

4.1.1 Módulos *multi-instance*

Para o caso dos módulos que suportam mais do que uma instância foi criada a classe `MBusMultiInstanceModule`, também ela abstracta e que deriva de `MBusModule` conforme representado na Figura 30.

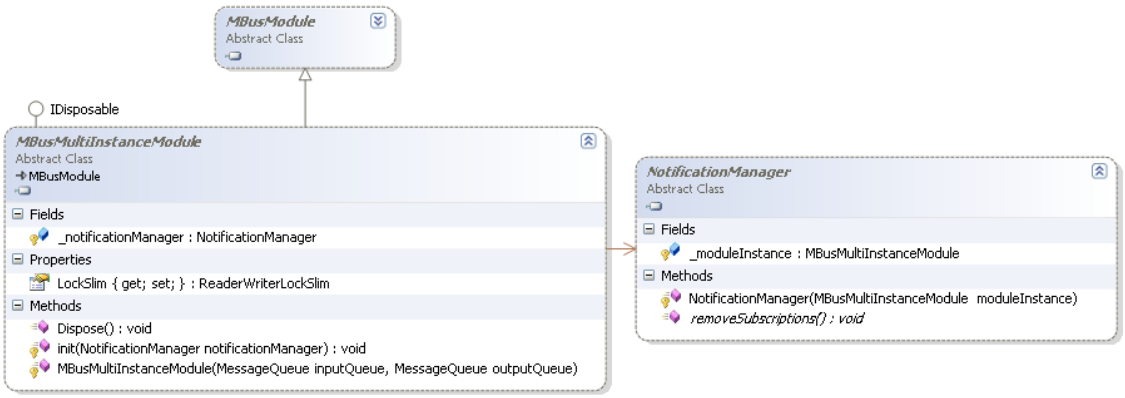


Figura 30: Diagrama de classes do suporte a módulos *multi-instance*

O problema que surge quando se permite que um módulo tenha várias instâncias prende-se com a partilha de estado entre as várias instâncias. É natural que para o funcionamento do módulo seja necessário guardar informação necessária durante processamento para o qual o módulo foi criado (normalmente em campos da classe que implementa o módulo). No entanto essa informação sofre alterações com o funcionamento do *bus*, por exemplo, com a execução de um comando. O problema surge pelo facto de, ao existirem várias instâncias, a informação que compõe o estado do módulo só seria alterada na instância que executasse o comando.

A solução para o problema da partilha de estado entre as várias instâncias dos módulos resolve-se criando uma fonte de dados à qual todas as instâncias têm acesso (que no caso desta implementação é uma base de dados em SQL Server). No entanto, a utilização de uma base de dados traz um problema de desempenho. Como toda a informação está na base dados, cada vez que uma instância precisar de consultar essa informação com garantia de que está a ler os valores actualizados, terá de ir à base de dados. Imagine-se por exemplo o caso do módulo *Router* ter de consultar a base de dados cada vez que é preciso realizar o encaminhamento de uma mensagem.

A solução para este segundo problema passa por manter uma cópia dos dados local a cada instância do módulo e ter um mecanismo de notificações para cada vez que alguma das instâncias alterar os dados. Assim sendo, quando um módulo é iniciado, carrega a informação da base de dados para campos da classe. Se durante a execução de um comando (ou noutra situação qualquer) uma instância alterar os seus dados, então procede à alteração também na base de dados e notifica todas as outras instâncias fornecendo a versão mais recente dos dados alterados.

Como mecanismo de notificação utilizado foi o *Web Solutions Platform* (WSP) [Cod10a]. O WSP trata-se de um sistema publicador de eventos onde as aplicações se

registam para serem notificadas da ocorrência de determinado evento. Os módulos que suportam várias instâncias criam os seus próprios tipos de eventos e registam-se para cada tipo criado.

Voltando à Figura 30, o tipo `MBusMultiInstanceModule` contém uma referência para um `NotificationManager`. O `NotificationManager` trata-se de uma classe abstracta cujas classes derivadas têm como objectivo conter a lógica de processamento de cada evento recebido.

Durante o funcionamento do módulo existirão várias *threads* a consultar os campos da classe que implementa o módulo. Visto que parte dessas threads consulta os campos para leitura e outra parte consulta para escrita, é necessário controlar o acesso aos campos para que a classe seja *thread-safe*. Tal é conseguido através de um `ReaderWriterLockSlim` que cada *thread* deverá adquirir antes de consultar dados partilhados entre *threads*. No caso dos módulos *multi-instance*, o `ReaderWriterLockSlim` é disponibilizado através de uma propriedade para que as instâncias de `NotificationManager` possam alterar campos do módulo sem criar problemas de concorrência com outras *threads*.

4.1.2 Comandos

Os comandos enviados para o *bus* são analisados e entregues a um ou mais módulos responsáveis pelo seu processamento. Todos os módulos que efectuam processamento de comandos apresentam como característica comum a implementação dos padrões `Command` e `Factory` [Eea95], conforme apresentado na Figura 31.

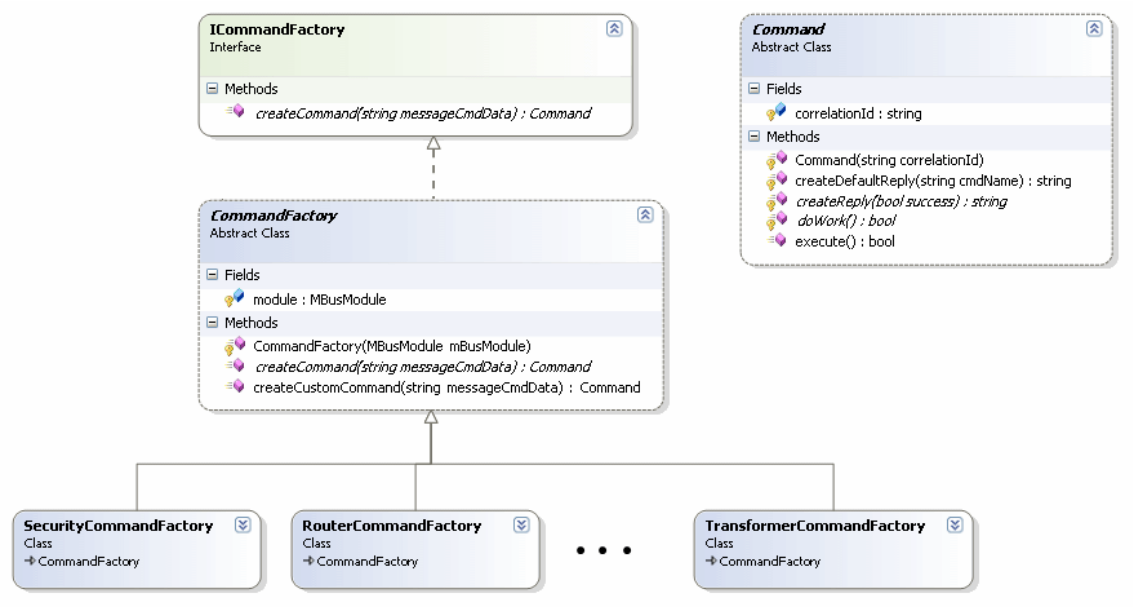


Figura 31: Implementação do processamento de comandos nos módulos

A classe `Command` trata-se de uma classe abstracta que representa que representa a mensagem de um comando, sendo responsável pela execução desse comando. A implementação do método `execute` que consiste em chamar o método `doWork` e, caso a mensagem exija resposta, executa-se o método `createReply`.

A interface `ICommandFactory` representa os objectos que sabem construir instâncias do tipo `Command`. `CommandFactory` é uma classe abstracta destinada a conter alguma lógica para a criação de comandos e dela derivam tantos objectos quanto o número de módulos que processam comandos. As classes derivadas de `CommandFactory` são responsáveis pela instanciação dos comandos associados ao módulo a que pertencem. Para tal, fornecem implementação do método `createCommand` (abstracto na classe base) que recebe a mensagem enviada pelo cliente e analisando a mensagem, retornam o comando adequado.

A implementação do código que suporta a criação de *custom commands* está explicada no capítulo 4.10.

4.2 Arranque do sistema

Sendo o *bus* de mensagens composto por vários módulos que se executam em processos distintos, é necessário um processo extra cuja finalidade é lançar cada um dos módulos. Este processo é o *MB_Core* que está contido no executável que permite lançar o *bus* de mensagens enquanto aplicação.

O *MB_Core* faz uso de um ficheiro de configuração (MBconfig.xml) para saber onde se encontram os executáveis correspondentes a cada módulo. A Listagem 3 exemplifica o conteúdo desse ficheiro

```
<configurations>
  <localProcess>
    <receiver exeFile=".\..\Receiver\bin\Debug\Receiver.exe"/>
    <router exeFile=".\..\Router\bin\Debug\Router.exe"/>
    <router exeFile=".\..\Router\bin\Debug\Router.exe"/>
    <transformer
exeFile=".\..\Transformer\bin\Debug\Transformer.exe"/>
    <manager exeFile=".\..\Manager\bin\Debug\Manager.exe"/>
    <security exeFile=".\..\Security\bin\Debug\Security.exe"/>
    <extensionManager
exeFile=".\..\MbExtensibility\bin\Debug\MbExtensibility.exe"/>
  </localProcess>

  <dataBase connectionString="data source=JC-GIATSI\JCDATABASE;
initial catalog=Message*Bus*DB; integrated security=SSPI; persist
security info=False;  Trusted_Connection=Yes">
  </dataBase>

</configurations>
```

Listagem 3: Exemplo do conteúdo de MBconfig.xml

O *MB_Core* pesquisa o ficheiro procurando os nós existentes em "`/configurations/localProcess/.`" e lançando num novo processo o executável indicado pelo atributo `exeFile`.

Para lançar mais do que uma instância de um módulo (apenas para os módulos que o suportem) basta repetir o elemento correspondente ao módulo. No exemplo exposto, serão lançadas duas instâncias do módulo Router e apenas uma dos restantes módulos. É da responsabilidade do administrador do sistema garantir que pelo menos uma instância de cada módulo é lançada.

O ficheiro *MBconfig* contém ainda a *connection string* que utilizada para aceder à base de dados.

4.3 Módulo Input/Output

O módulo de Input/Output (I/O) é o ponto de entrada e saída de mensagens no *bus*. Conforme exposto na Figura 27, a implementação do módulo I/O divide-se em duas partes – *Receiver* e *Dispatcher*. O *Receiver* processa as mensagens que dão entrada no *bus* através de um dos múltiplos protocolos de comunicação suportados. Um aspecto

comum a todos os protocolos de comunicação é o de, directa ou indirectamente, assentarem a sua comunicação em WCF.

O *bus* de mensagens define duas interfaces distintas às quais correspondem dois serviços, cabendo às implementações de protocolos de comunicação optar por usar um deles. As funcionalidades oferecidas por cada um dos serviços são semelhantes, apenas diferindo no modo de comunicação (*channel shape*) – *One Way* num dos serviços e *Duplex* no outro. A Figura 32 representa o diagrama de classes das interfaces e classes que compõem ambos os serviços.

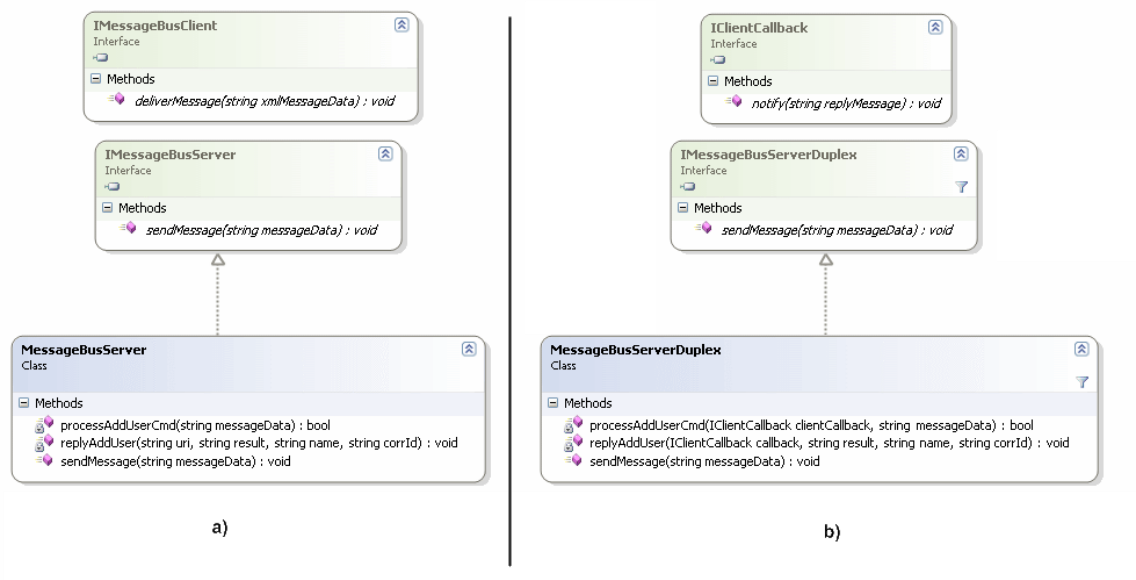


Figura 32: Diagrama de classes do serviço WCF

Na Figura 32 a) todas as comunicações entre cliente e *bus* de mensagens são *one-way*. O cliente terá ele próprio de implementar uma componente servidora que implemente a interface `IMessageBusClient`. Nessa interface existe o método `deliverMessage` que servirá para o servidor comunicar com o cliente, tanto para entregar mensagens de dados, como repostas a comandos que o cliente tenha executado.

A Figura 32 b) representa as classes que implementam a mesma funcionalidade mas com comunicação *duplex*. Através de atributo `ServiceContract` aplicado à interface `IMessageBusDuplex`, indica-se que `IClientCallBack` é a interface que o servidor dispõe para comunicar com o cliente. Desta forma, em qualquer ponto na implementação do servidor podem ser invocados os métodos de `IClientCallBack`.

Em ambos os casos, sempre que o cliente pretender enviar uma mensagem (de dados ou um comando) para o servidor, é utilizado o método `sendMessage`. O método `sendMessage` verifica se a mensagem é um comando do tipo *AddUser* e apenas em

caso negativo é que a mensagem é colocada na fila *incomingQueue*, ficando então disponível para ser processada pelo *bus*. No caso de se tratar de um comando *AddUser*, a sua execução é feita logo no momento por ser necessário travar o processamento caso o nome de utilizador pedido já existir.

4.3.1 Receiver

A função do *Receiver* é recolher as mensagens da *incomingQueue*, formata-las e coloca-las na fila *outputQueue*. As mensagens que entrarem em *incomingQueue* já se encontram formatadas no formato interno do *bus*. Ao longo da passagem pelos vários módulos do sistema, serão acrescentados vários cabeçalhos SOAP à mensagem. A formatação referida por parte de *Receiver* consiste em limpar da mensagem todos esses cabeçalhos, evitando assim que algum cliente tente voluntariamente interferir com o normal funcionamento do *bus*.

O Receiver constituiu o núcleo do módulo I/O e é implementado pela classe `MessageBusReceiver` que estende de `MBusModule`. Não se permitem várias instâncias deste módulo pelo facto do processo que executa o módulo I/O incluir também a implementação dos protocolos de comunicação. Vários tipos de ligação (por exemplo TCP/IP utilizado pelo AMQP) não permitem que exista mais do que um servidor na mesma máquina.

4.3.2 Dispatcher

Quando as mensagens estão prontas para entregar ao destino final são colocadas na fila *dispatchQueue*. O *dispatcher* é responsável por ler as mensagens dessa fila, fazer o mapeamento entre o endereço virtual e físico do destino e finalmente entregar a mensagem ao cliente final de acordo com o protocolo que este utilizar (tendo em conta se o protocolo do cliente utiliza um serviço *duplex* ou *one-way*).

Na Figura 33 observa-se o diagrama de classes do tipo `Dispatcher`, que realiza os passos atrás descritos.

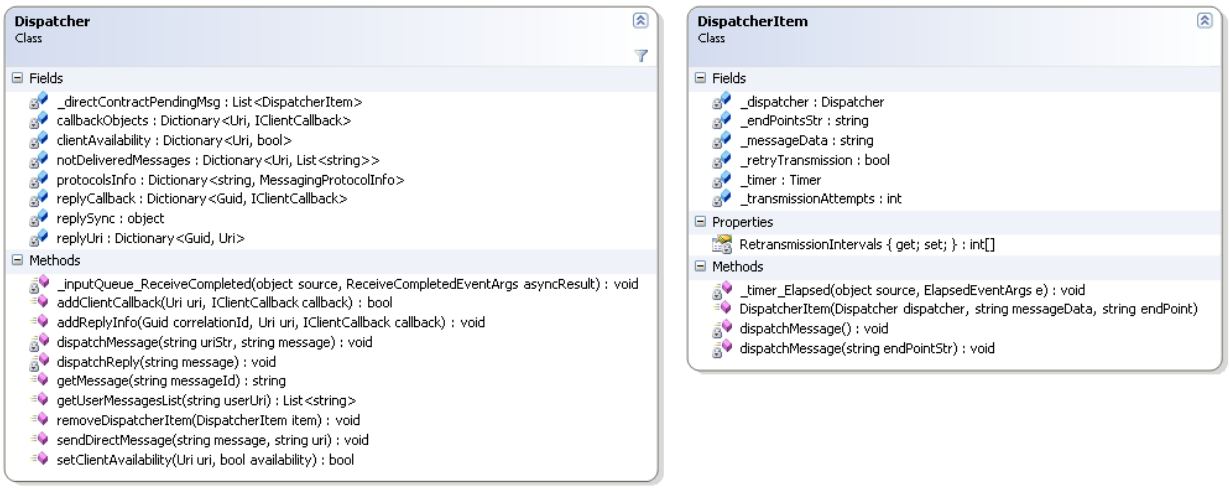


Figura 33: Diagrama de classes da implementação do *dispatcher*

O `Dispatcher` contém internamente:

- Informação sobre o modo de funcionamento de cada protocolo de comunicação (campo `protocolsInfo`);
- Referências para os objectos *callback* de clientes que utilizem o serviço *duplex* (`callbackObjects`);
- Informação da disponibilidade de cada cliente para receber mensagens (`clientAvailability`);
- Conjunto de mensagens que ainda não estão entregues ou porque a primeira tentativa de envio falhou ou porque o cliente não estava receptivo (`notDeliveredMessages` e `_directContractPendingMsg`);
- Mecanismos que permitem saber a que cliente deve ser entregue determinada resposta a um comando (`replyUri` e `replyCallback`).

As entregas a clientes com protocolos baseados no serviço *one-way* são suportadas por objectos do tipo `DispatcherItem`. Este objecto estabelece uma ligação com a aplicação cliente (na qual é o *bus* de mensagens que se comporta como cliente) e procede ao envio da mensagem. Caso o envio falhe, existe um campo do tipo `Timer` que dispara eventos em intervalos de tempo configuráveis para que seja tentado o reenvio.

4.3.3 Comandos

O módulo I/O é responsável pela execução de três tipos de comandos:

- *GetMessage*;
- *GetUserMessagesList*;
- *SetEndPointState*.

Para todos os comandos, o responsável pela sua execução é o *Receiver* (embora durante a execução invoque métodos do *Dispatcher*).

4.4 Protocolos de comunicação

Na implementação fornecida do *bus* de mensagens são suportados três protocolos de comunicação – *MB Protocol*, AMQP e RestMs. Novos protocolos podem ser implementados enviando para o *binding netNamedPipeBinding* mensagens com o formato de *MB Protocol*. Este *binding* está associado à interface do serviço *duplex* atrás exposta.

O *bus* de mensagens necessita de um conjunto de informações acerca do modo de funcionamento de cada protocolo de comunicação. Essa informação é fornecida através do ficheiro de configuração MessagingProtocolsDescriptor.xml que é disponibilizado na mesma máquina onde for executado o módulo I/O. A Listagem 4 mostra um exemplo do conteúdo deste ficheiro.

```
<MessagingProtocols>
  <amqp
    clientInteraction="CallbackContract"
    clientDeliveryMode="Passive"
    protocolScheme="amqp"/>

  <RestMS
    clientInteraction="CallbackContract"
    clientDeliveryMode="Active"
    protocolScheme="rm"/>

  <Message*Bus*Protocol
    clientInteraction="directContract"
    clientDeliveryMode="passive"
    protocolScheme="mb"/>

</MessagingProtocols>
```

Listagem 4: Exemplo do conteúdo de MessagingProtocolsDescriptor.xml

São três as informações necessárias acerca de cada protocolo de comunicação:

- *ProtocolSchema* – consiste num nome único que permite ao *bus* de mensagens identificar um protocolo de comunicação.

- *ClientInterection* – especifica o modo de interacção com o cliente. São aceites dois valores. "CallBackContract" no caso de o protocolo utilizar a interface *duplex* do serviço WCF, "directContract" caso o protocolo utilize a interface *one-way*. No caso de novas implementações, este valor será sempre "CallBackContract", visto que o *binding netNamedPipeBinding* está associado à interface *duplex*.
- *ClientDeliveryMode* – indica de que forma é que as mensagens são entregues aos clientes finais. Se atribuído o valor "Active", cabe às aplicações cliente verificar junto do *bus* se existem mensagens para elas. Por outro lado, o valor "passive" indica que as aplicações cliente são de natureza passiva, isto é, cabe ao *bus* a iniciativa de entregar as mensagens.

As implementações de novos protocolos, antes de enviarem dados para o *binding netNamedPipeBinding* têm de acrescentar uma nova entrada no ficheiro MessagingProtocolsDescriptor.xml, tendo em atenção que não poderão repetir nenhum dos *protocolSheme* já existentes.

4.4.1 MB Protocol

O *MB Protocol* é o protocolo de comunicação criado no contexto deste trabalho e utiliza como formato de dados o SOAP (do qual deriva o formato interno do *bus* de mensagens). O canal de comunicação utilizado por este protocolo são *Web Services*, pelo que a comunicação com o *bus* é feita através de um *binding wsHttpBinding* associado à interface *one-way* do serviço. Os clientes *MB Protocol* têm natureza passiva no que diz respeito à recepção de mensagens.

O *MB Protocol* prevê a existência de três tipos de mensagens:

- Mensagens de dados;
- Comandos;
- Respostas.

As mensagens de dados contêm a informação que as aplicações cliente pretendem trocar entre elas. As mensagens de comandos consistem os comandos referidos no capítulo 3.4 e que permitem configurar o funcionamento do *bus*. A alguns comandos

estão associadas respostas que são enviadas ao utilizador e que compõem o ultimo tipo de mensagens.

Todas as mensagens contêm o elemento `<mb:MessageType>` no *Header* do SOAP. Este elemento permite identificar o tipo de mensagem conforme o texto encontrado dentro no seu interior e que pode ser "*data*", "*command*" ou "*reply*".

A informação nas mensagens de dados é transportada no campo *Body* da mensagem SOAP. Dentro desse campo o formato é livre, podendo ser XML ou qualquer outro formato baseado em texto. Se for necessário transportar dados binários, as aplicações deverão incluir o elemento `<mb:attachment>` dentro do *Body* onde podem colocar conteúdos codificados em Base64.

O formato (e tipo) dos comandos definidos no *MB Protocol* está no Anexo 1. Um comando só dará origem a uma resposta caso o cliente inclua no *Header* o elemento `<mb:CorrelationID>` cujo texto deverá ter a representação textual de um objecto `Guid` da plataforma .Net. A mensagem de resposta repetirá o elemento `<mb:CorrelationID>` para que a aplicação cliente saiba qual o comando a que a resposta se refere.

4.4.2 Advanced Message Queuing Protocol

A implementação do protocolo AMQP foi feita através da criação de um *user-defined binding* para WCF. Os *user-defined binding* são utilizados quando nenhum dos *bindings* fornecidos pelo WCF preenche os requisitos do serviço. Outra alternativa seria criar um *custom binding*, no entanto, os *user-defined bindings* têm a vantagem de ser mais facilmente reutilizáveis em vários projectos distintos. [Res08] Desta forma, permite-se que o *binding* criado seja utilizado por outras implementações de servidores AMQP realizadas fora deste projecto.

Antes de se criar um *binding* é necessário conhecer o *stack* do WCF, representado na Figura 34.

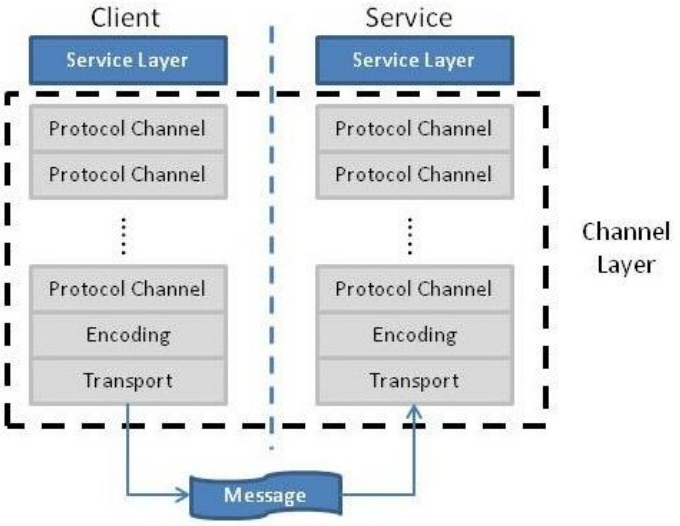


Figura 34: *Stack* de comunicação WCF. Fonte: [JR09]

O *stack* apresentado na Figura 34 divide-se em duas camadas – *channel layer* e *service layer*. A *channel layer* é responsável pelo envio e recepção de mensagens WCF enquanto a *service layer* tem a responsabilidade de traduzir essas mensagens nas chamadas aos métodos que implementam determinado serviço. [Sho05]

O *binding* desenvolvido centra-se exclusivamente na *channel layer*, pelo que as mensagens WCF ao atingirem a *service layer* já levam o conteúdo necessário para que seja invocado o serviço disponibilizado pelo *bus* de mensagens (no caso desta implementação, o serviço `Message`*`Bus`*`ServerDuplex`).

Internamente a *channel layer* é composta por um elemento de transporte, um elemento de codificação e zero ou mais elementos de protocolo. O elemento de transporte lida com os mecanismos utilizados para mover a mensagem entre o cliente e o servidor (e vice-versa). O elemento de codificação é responsável por codificar/descodificar os dados no formato utilizado pelo transporte em informação interpretável dentro do *stack* do WCF. Finalmente, os elementos de protocolo destinam-se implementar necessidades específicas de um determinado *binding*. Essas necessidades incluem aspectos de segurança, gestão de um determinado protocolo, etc.

Os elementos utilizam *canais* para comunicar com os elementos imediatamente acima e abaixo no *stack* WCF na Figura 34 podendo esses canais ser de dois tipos – *canais de transporte* e *canais de protocolo*. [All07]

Um *binding* não é mais do que uma colecção de *BindingElements* onde cada `BindingElement` consiste num objecto que representa um elemento de transporte,

codificação ou protocolo. O *runtime* do WCF inspecciona os *BindingElements* presentes num *binding* de forma a determinar qual o elemento utilizado para transporte, codificação, etc. Cada `BindingElement` dispõe de métodos que permitem instanciar o objecto responsável pelas tarefas de codificar, transportar, etc.

A Listagem 5 mostra os passos na criação de um servidor utilizando o objecto `ServiceHost`. Este objecto automatiza os passos atrás descritos durante a análise de um *binding* e criação do *stack* de comunicação para um dado serviço.

```
ServiceHost host = new ServiceHost(typeof(Service), new Uri("amqp://localhost/"));
host.AddServiceEndpoint(typeof(IMessageBusServerDuplex),        new        AmqpBinding(),
"MessageBus");
host.Open();
```

Listagem 5: WCF - Passos na instanciação de um serviço

No entanto, internamente o *runtime* do WCF ao executar o código da Listagem 5, executará uma sequência de passos semelhante à representada na Listagem 6. [Smi07]

```
     Binding binding = new AmqpBinding();
     Uri uri = new Uri("amqp://localhost/");
     IChannelListener<IDuplexSessionChannel> listener =
binding.BuildChannelListener<IDuplexSessionChannel>(uri, new
BindingParameterCollection());
     listener.Open();
     IDuplexChannel channel = listener.AcceptChannel();
     channel.Open();

     //...
     //durante a comunicação
     Message m = channel.Receive();
     channel.Send(m);
```

Listagem 6: Passos no *runtime* durante a instanciação de um serviço

A implementação do *binding* consistiu na criação da classe `AmqpBinding` que deriva de `Binding` e na implementação de um conjunto de classes que permite ter os elementos *transport*, *encoding* e *protocol* presentes na Figura 34.

A Figura 35 mostra o diagrama de interacção entre as principais classes da implementação de cada um dos três elementos.

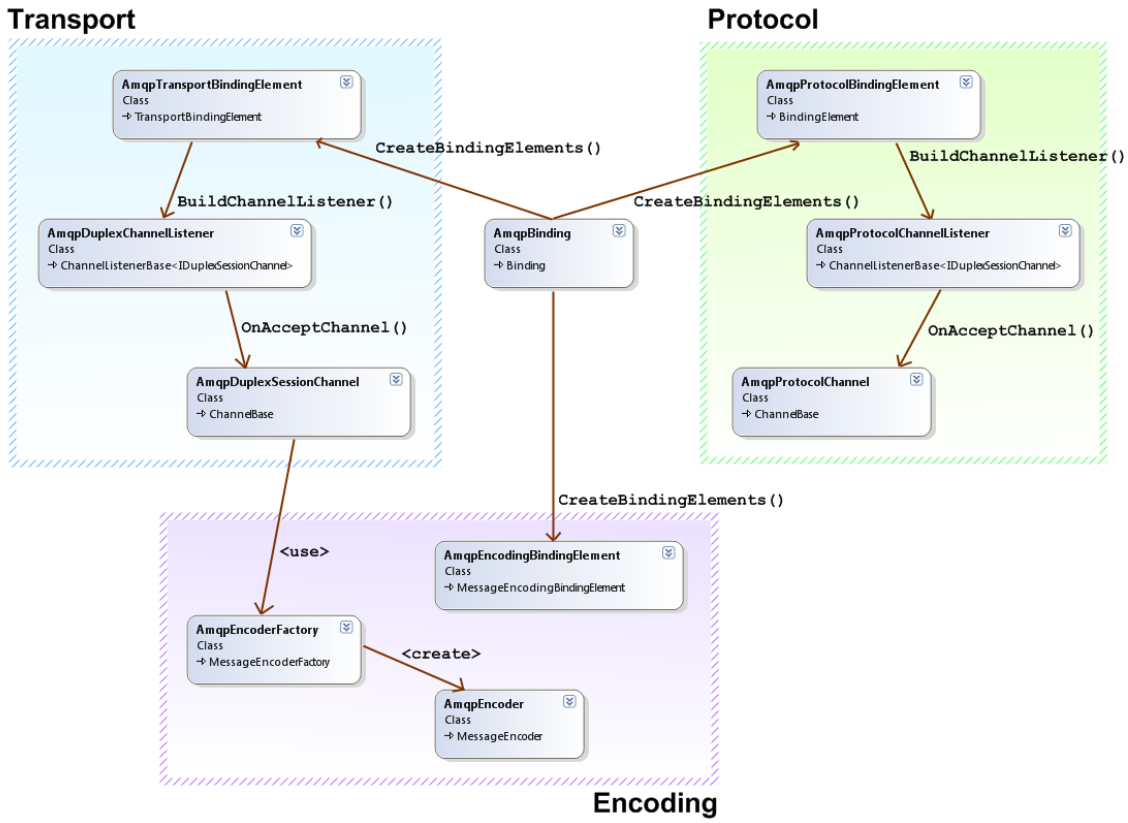


Figura 35: Diagrama de interacção entre os objectos de cada elemento do `AmqpBinding`

De um modo geral, cada elemento contém três objectos principais:

- Um *BindingElement*;
- Um objecto responsável pela criação do canal de comunicação;
- Um objecto que efectivamente implementa a lógica do canal.

O *BindingElement* identifica, através do tipo base, o tipo de elemento em questão (elemento de protocolo, elemento de transporte, etc.). É através do *BindingElement* que o *runtime* do WCF obtém uma instância de um *ChannelListener*. O *ChannelListener* é o tipo utilizado do lado do servidor para esperar por uma ligação do lado do cliente. Quando essa ligação é recebida, o *ChannelListener* instancia o terceiro objecto, responsável por tratar da comunicação com esse cliente.

Os elementos de codificação não seguem exactamente as mesmas regras que os de transporte e protocolo. A principal característica que os distingue é o facto de não possuírem um canal e terem uma relação de dependência directa com o elemento de transporte. Esta é a razão pela qual possuem um *factory* em vez de um *channel listener*.

A utilização dos métodos de AmqpEncoder é controlada pelas instâncias de `AmqpDuplexSessionChannel`.

Nos capítulos que se seguem será explicada de forma mais detalhada como cada um dos elementos foi desenvolvido.

4.4.2.1 Elemento de transporte

O elemento de transporte implementa os três objectos comuns a todos os elementos e utiliza como canal um *canal de transporte*. O canal de transporte tanto pode ser uma ligação entre dois computadores numa rede, como uma zona de memória partilhada ou um ficheiro. No caso do elemento de transporte para o *binding* AMQP, o canal de transporte é uma ligação TCP/IP.

A implementação do transporte consiste assim na criação dos tipos `AmqpTransportBindingElement`, `AmqpDuplexChannelListener` e `AmqpDuplexSessionChannel` cujo diagrama de classes se encontra na Figura 36[5].

[5] Os diagramas de classes neste documento contêm apenas os campos e métodos relevantes para a sua explicação.

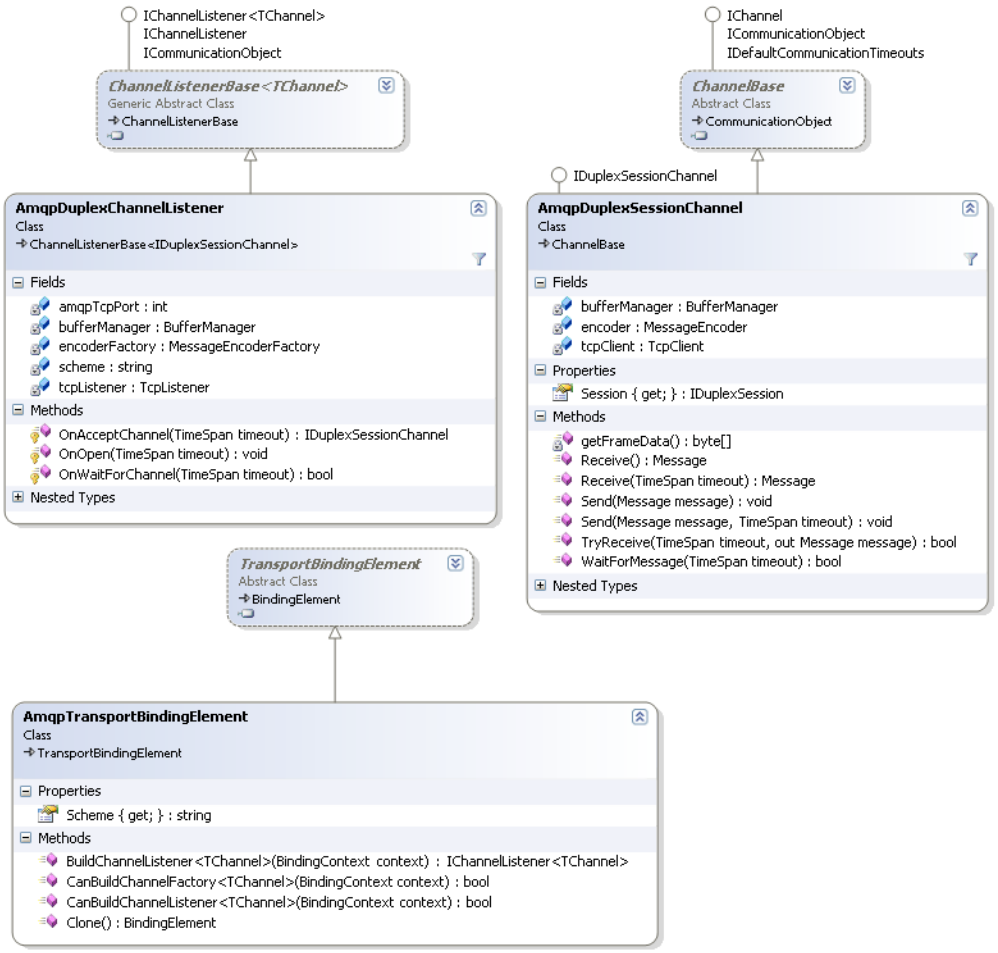


Figura 36: Diagrama de classes da implementação do canal de transporte

O WCF define o conceito de *channel shape* [Res08] para especificar o modo de interacção entre o cliente e servidor. São suportados três *channel shapes* – *One-way*, *Resquest-Replay* e *Duplex*. Em comunicações *One-way* o fluxo de informação dá-se apenas no sentido cliente-servidor. No *Resquest-Replay* existe um pedido inicial do cliente ao qual o servidor envia uma resposta no contexto desse pedido. Finalmente, o *channel shape Duplex* permite a comunicação nos dois sentidos onde cliente e servidor poderão em qualquer instante enviar dados. No AMQP a comunicação é bidireccional, o que significa que o canal de transporte implementado tem de ser *Duplex*. A consequência mais visível da comunicação *Duplex* é a existência dos métodos `Send` e `Receive` na classe `AmqpDuplexSessionChannel`.

Os clientes AMQP estabelecem com o servidor uma ligação TCP/IP baseada em *sockets*. Isto implica que o servidor tenha para cada cliente um *socket* independente, o que no contexto do WCF significa dizer que o canal utilizado pelo servidor tem de suportar sessões. Nos canais com sessão, o servidor utiliza uma métrica que permite

distinguir quando é que os dados recebidos devem ser associados a um novo cliente (implicando a criação de um canal dedicado a esse cliente) ou a um cliente com quem já existe um canal aberto. Desta forma, o servidor manterá tantos objectos `AmqpDuplexSessionChannel` quantos os clientes activos que existirem. (ver Figura 37)

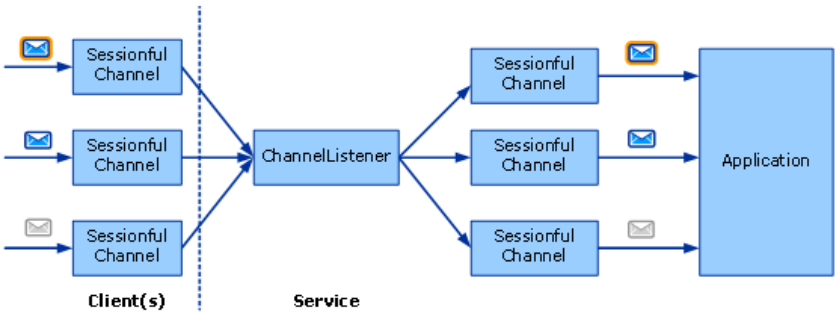


Figura 37: Canal WCF com sessão. Fonte: [MSD10a]

No caso do `AmqpDuplexSessionChannel` a métrica utilizada para associar os dados recebidos com os canais já existentes é uma *string* com a forma "IP:porto" com o IP e porto origem dos dados enviados.

Cada instância do canal de transporte contém um campo do tipo `MessageEncoder` utilizado para codificar os dados no sentido descendente do *stack* e descodificar os dados no sentido ascendente do *stack*.

4.4.2.2 Elemento de codificação

A principal diferença entre a implementação do elemento de transporte e de codificação é o facto de a codificação não utilizar nenhum canal. Esta característica leva à substituição do objecto *ChannelListener* por um *MessageEncoderFactory*. De resto, como se pode verificar no diagrama de classes da Figura 38, mantém-se a existência de um *BindingElement* e de uma classe que implementa o *encoder* propriamente dito.

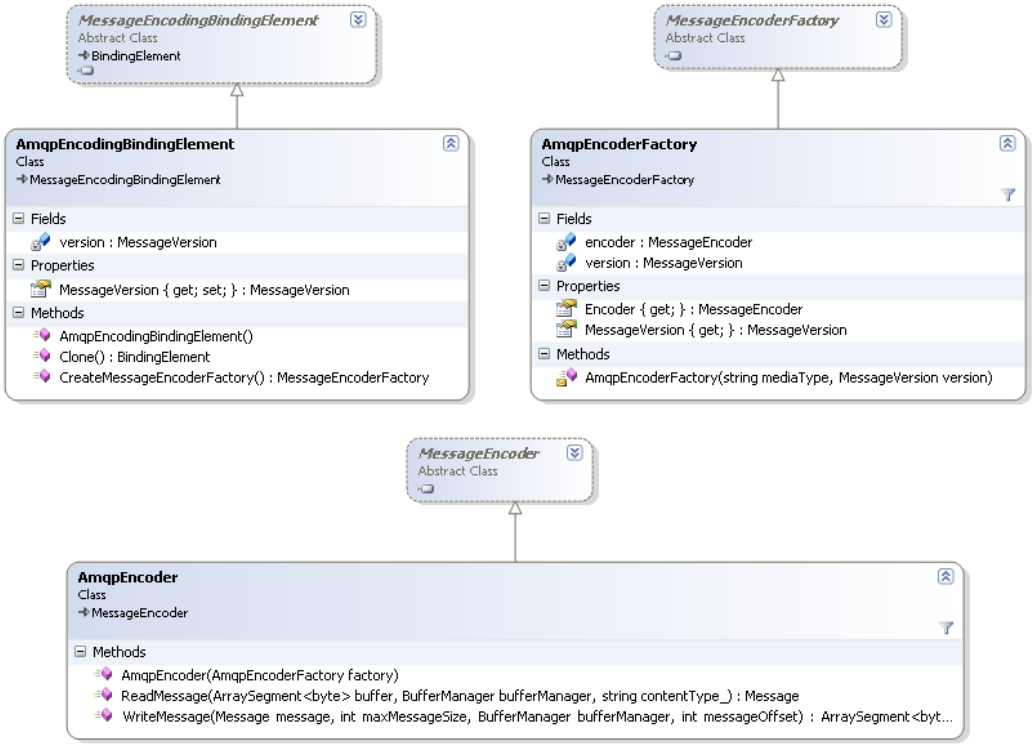


Figura 38: Diagrama de classes da implementação do *encoder*

O canal de transporte contém uma referência para um `AmqpEncoder`, que é utilizada nos métodos `Send` e `Receive`. O método `Send` utiliza o método `WriteMessage` de `AmqpEncoder` para converter um objecto `ServiceModel.Channels.Message` numa sequência de *bytes* a enviar pela rede. No processo inverso, o método `Receive` do canal de transporte, chama o método `ReadMessage` de `AmqpEncoder` para converter uma sequência de *bytes* num objecto `ServiceModel.Channels.Message`.

Implementou-se um conjunto de objectos que auxilia a codificação e execução de cada um dos comandos existentes no AMQP. Desse conjunto de objectos, os relevantes para o elemento de codificação são os `AmqpFrame` e `AmqpFrameDecoder` e cujo diagrama de classes se representa na Figura 39.

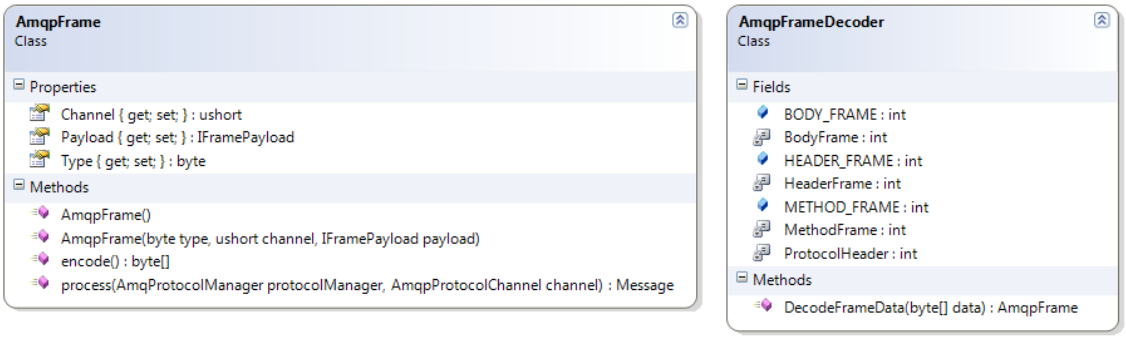


Figura 39: Diagrama de classes dos tipos `AmqpFrame` e `AmqpFrameDecoder`

A classe `AmqpFrame` representa qualquer *frame* de dados entre aquelas que estão definidas no AMQP. Através do atributo `Type` é possível conhecer o tipo de *frame* (i.e. *method frame*, *body frame*, *etc.*) e o seu conteúdo obtém-se com a propriedade `Payload` cujo tipo não se encontra representado na Figura 39. Os objectos do tipo `AmqpFrame` contêm dois métodos – `encode` e `processAmqpProtocolManager` – o primeiro é utilizado pelo método `WriteMessage` de `AmqpEncoder` para obter a sequência de *bytes* que codifica a *frame* representada pelo objecto AmqpFrame no qual se invocou o método. O segundo método (`processAmqpProtocolManager`) é utilizado pela camada no *stack* WCF acima do *Encoder* (neste caso, o *Protocol Channel*).

Note-se que os dados depois de descodificados pelo *Encoder* (nas mensagens que circulam no sentido ascendente do *stack*) são representados por um objecto `ServiceModel.Channels.Message` (porque o WCF assim o impõe). Este objecto, que representa ele próprio uma mensagem SOAP, contém no *body* desse SOAP um objecto `AmqpFrame` seriado.

Os objectos `AmqpFrame` e `AmqpFrameDecoder` constituem o *front end* de um extenso conjunto de classes que fornecem suporte às suas funcionalidades. Na implementação dessas classes, cuja explicação se encontra no Anexo 2, foram utilizadas tecnologias como por exemplo *Platform Invoke* (P/Invoke).

4.4.2.3 Elemento de protocolo

O elemento de protocolo trata-se de um elemento do *stack* WCF e como tal implementa os três objectos comuns a elementos referidos no capítulo 4.4.2. O diagrama de classes destes três tipos apresenta-se na Figura 40.

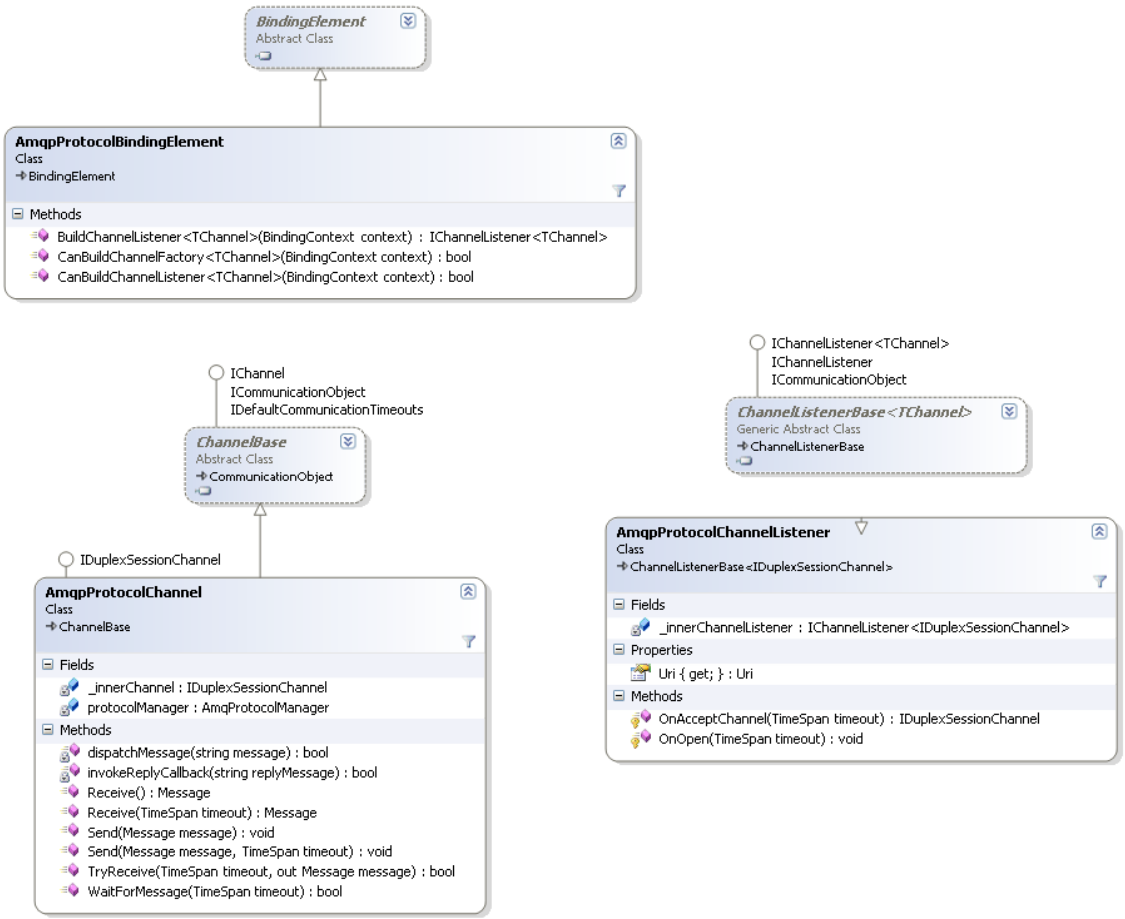


Figura 40: Diagrama de classes da implementação do canal de protocolo

No *stack* WCF representado na Figura 34 cada elemento (correspondente a uma camada no esquema da figura) controla as acções do elemento que lhe estiver imediatamente abaixo. Por este motivo, os objectos `AmqpProtocolChannel` e `AmqpProtocolChannelListener` contêm o campo `_innerChannel` e `_innerChannelListener`. O objectivo deste campo é que, para todos os métodos referentes à implementação de um canal, o canal superior invoque o método homólogo no canal inferior. Tomando como exemplo o método `Receive`, o *runtime* do WCF invocará este método no canal do elemento que estiver no topo do *stack* (o `AmqpProtocolChannel`, neste caso). O canal do protocolo realiza as acções que

fizerem sentido ao nível do protocolo e de seguida invoca o mesmo método no objecto `_innerChannel`. [Jud07]

O papel do elemento de protocolo na implementação do *binding* para AMQP é o de gerir o processo de estabelecimento da ligação com o cliente, gerir o estado de parâmetros específicos ao AMQP que não têm equivalência no *MB Protocol* e decidir que mensagens é que são convertidas no formato *MB Protocol* encaminhando-as para a *service layer*. Parte destes requisitos são suportados pelo campo `protocolManager` do tipo `AmqpProtocolManager` e cujo diagrama de classes se encontra na Figura 41.

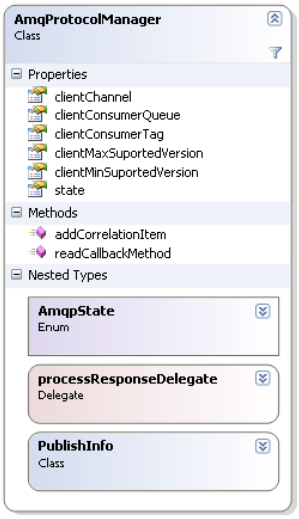


Figura 41: Diagrama de classes do tipo `AmqpProtocolManager`

Sendo a ligação AMQP implementada através de um canal com sessão, então existirá um objecto do tipo `AmqpProtocolManager` por cada cliente AMQP. Neste objecto são armazenadas características do cliente como por exemplo, a máxima e mínima versão do protocolo suportada, o canal AMQP pretendido pelo cliente, etc.

Uma característica das ligações AMQP é a existência de estado. Esse estado é mantido e controlado pelo `AmqpProtocolManager` de acordo com o diagrama que se mostra na Figura 42.

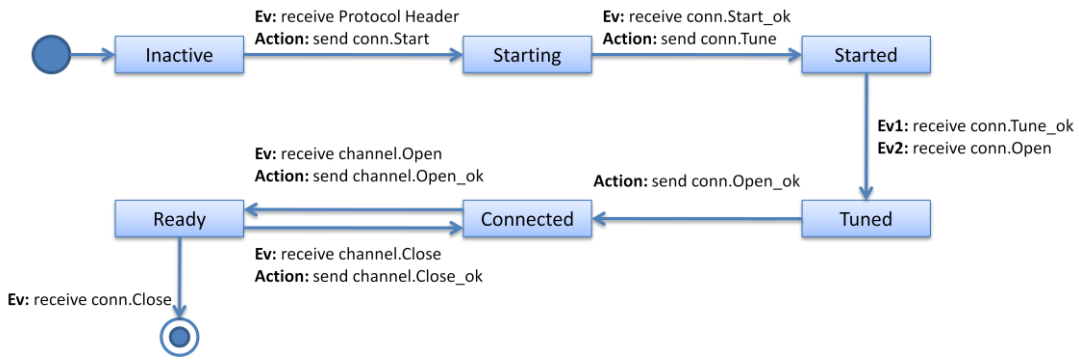


Figura 42: Diagrama de estados de uma ligação AMQP

O estado da ligação é consultado na execução de cada comando, onde se verifica se a ligação está no estado necessário para que seja possível executar o comando pedido.

Tal como já foi referido, alguns comandos AMQP são convertidos em comandos *MB Protocol* que seguem para a *service layer* sendo posteriormente processados no *bus* de mensagens. No caso de alguns destes comandos, é necessário enviar uma resposta ao cliente AMQP. Essa reposta, virá do *bus* sob a forma de uma chamada ao método `send` de `AmqpProtocolChannel` e onde os dados estarão formatados de acordo com o *MB Protocol*. A responsabilidade da conversão entre *MB Protocol* e AMQP fica a cargo do objecto `AmqpFrame` que representava o comando original. Coloca-se então o problema de como identificar esse objecto durante a execução do método `send`. A solução adoptada consiste em ter no `AmqpProtocolManager` a tabela `responseCorrelationTable` composta por pares de identificadores únicos (objectos do tipo `Guid`) e *delegates*, onde cada *delegate* é definido com a seguinte assinatura:

```
    public delegate void processResponseDelegate(string response,
AmqProtocolManager protocolManager, AmqpProtocolChannel channel);
```

Listagem 7: *Delegate* de `AmqpProtocolManager`

As implementações dos comandos que necessitem de enviar uma resposta ao cliente implementam um método que verifique a assinatura do *delegate* e invocam o método `addCorrelationItem` de `AmqpProtocolManager` para adicionar uma nova entrada na tabela `responseCorrelationTable`. Desta forma, o método `send` de `AmqpProtocolChannel` verificará se a mensagem se trata de uma *reply message* e, caso afirmativo, extrai-se o `Guid` nela incluído e utiliza-se a tabela

`responseCorrelationTable` para identificar e invocar o *delegate* que saberá converter os dados e encaminha-los para o cliente final.

4.4.3 RestMS

Implementar um servidor RestMS implica, numa visão mais abstracta, implementar um serviço REST. No WCF existe o *binding* `WebHttpBinding` que é indicado para implementar serviços que recebem chamadas através de protocolos *web* como pedidos HTTP e que utilizam padrões REST/POX[6]/JSON. [Low08] Posto isto, a implementação do protocolo RestMS consiste na criação de um serviço WCF que utiliza um *binding* `WebHttpBinding`.

Em REST, as operações de um serviço são identificadas com base em URI que podem inclusive conter parâmetros para passar à operação. O `WebHttpBinding` efectua o trabalho de *parsing* do URI e identifica o método que implementa determinada operação, extraindo também os parâmetros que esse método recebe. Assim, por exemplo, a definição da operação associada a um *post* para um *feed* será:

```
[OperationContract]
[WebInvoke(Method = "POST", UriTemplate = "/feed/{feedName}")]
Stream postFeed(string feedName);
```

Listagem 8: Definição de uma operação REST

Desta forma, o programador apenas terá de se focar na escrita da lógica que implementa o serviço. A implementação do serviço é feita pelo objecto `RestMsServer` que contém um método por cada verbo HTTP suportado para cada recurso (consultar Tabela 2 – capítulo 3.4 para uma lista completa). Para processar cada um dos pedidos, foram criadas seis classes auxiliares correspondentes aos seis recursos definidos no RestMS. Cada uma destas classes tem métodos chamados *get*, *put*, etc. correspondentes aos verbos HTTP suportados pelo recurso. Estes métodos processam as operações solicitadas pelo cliente e efectuando sempre que necessário a tradução em comandos *MB Protocol* que são enviados para o *bus*. Têm ainda que aguardar pela resposta do *bus* e retornar a informação adequada ao cliente.

Tal como na implementação do protocolo AMQP, também no RestMS existem dados que não podem ser guardados no *bus* de mensagens por serem específicos ao

[6] Plain Old XML

protocolo de comunicação. Para conter essa informação, foi criada a classe `ServerState` que contém, entre outros, informação dos *pipes*, *feeds* e *joins* existentes, os *id* utilizados na atribuição automática de nomes a cada um dos recursos, ou mensagens que já saíram do *bus* de mensagens mas ainda estão em *cache* (note-se que no *bus* as mensagens são eliminadas assim que entregues ao cliente final, no entanto, em RestMS uma mensagem só eliminada do servidor quando o cliente executar um *delete* sobre o recurso da *mensagem*).

Como forma de demonstração, a implementação do RestMS faz uso do mecanismo disponibilizado pelo *bus* de mensagens para extensão de protocolos de comunicação. Quer isto dizer que a implementação RestMS apresenta por um lado uma componente servidora (aquela que foi até aqui exposta) e por outro, uma componente cliente (cliente do *bus* de mensagens através do *binding* `NetNamedPipeBinding`).

Para realizar a ponte entre componente servidora e componente cliente, foi criado o objecto `Message`*`Bus`*`ProtocolExtention`. Este objecto contém o método `sendMessage` que permite enviar mensagens (de dados ou comandos) para o *bus* de mensagens. Contém ainda uma tabela que permite associar objectos `Guid` a *delegates*, de forma a determinar que objectos notificar quando são recebidas do *bus* respostas a comandos. O *binding* `NetNamedPipeBinding` utiliza comunicação bidireccional (*duplex communication*), o que em WCF implica que exista uma segunda interface que especifique o tipo de interacção no sentido servidor – cliente. O objecto `CallbackHandler` implementa essa interface e sempre que existir comunicação do lado do servidor (*bus* de mensagens), notifica o `Message`*`Bus`*`ProtocolExtention`.

O Figura 43 representa um diagrama de interacção entre os vários componentes descritos até ao momento e que constituem a implementação do protocolo de comunicação RestMS.

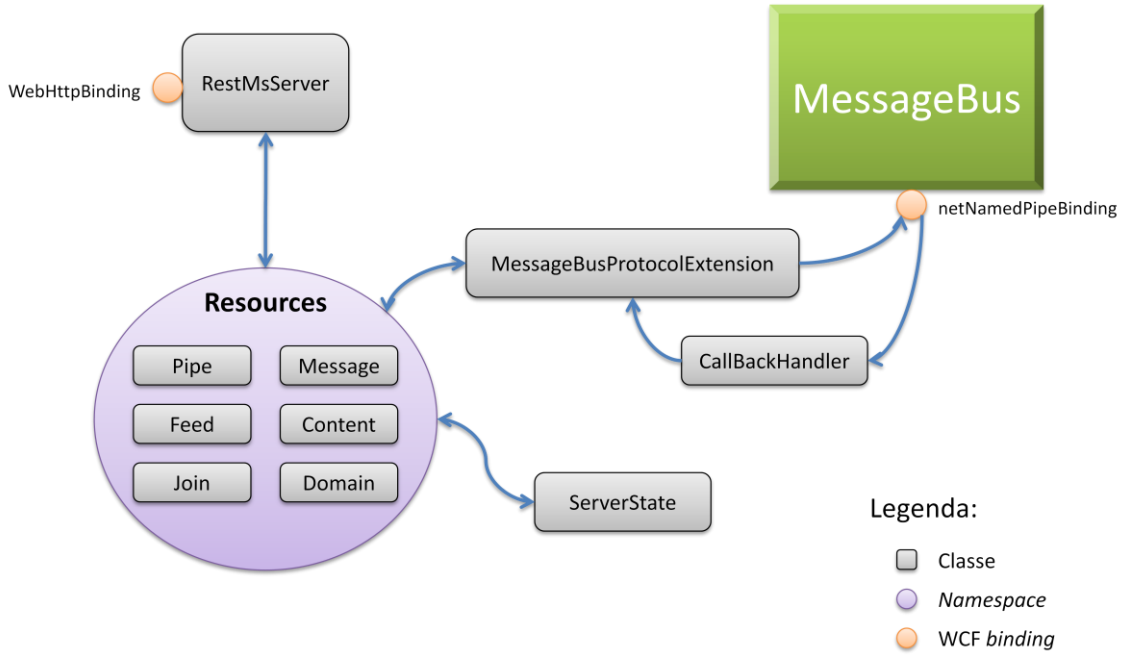


Figura 43: Diagrama de interacção entre os componentes da implementação RestMS.

Contrariamente ao que acontece no AMQP, onde o *payload* das mensagens consiste numa sequência de *bytes*, no RestMS as mensagens de dados apresentam *payloads* formatados (ver Figura 16). É importante permitir que uma mensagem produzida por um cliente RestMS possa ser consumida por clientes de outros protocolos (e vice-versa). Desta forma, tal como os comandos RestMS são convertidos em comandos *MB Protocol*, também o conteúdo das mensagens de dados terá de ser convertido para uma representação neutra interna ao *bus*.

A Figura 44 exemplifica a forma como essa conversão é realizada.

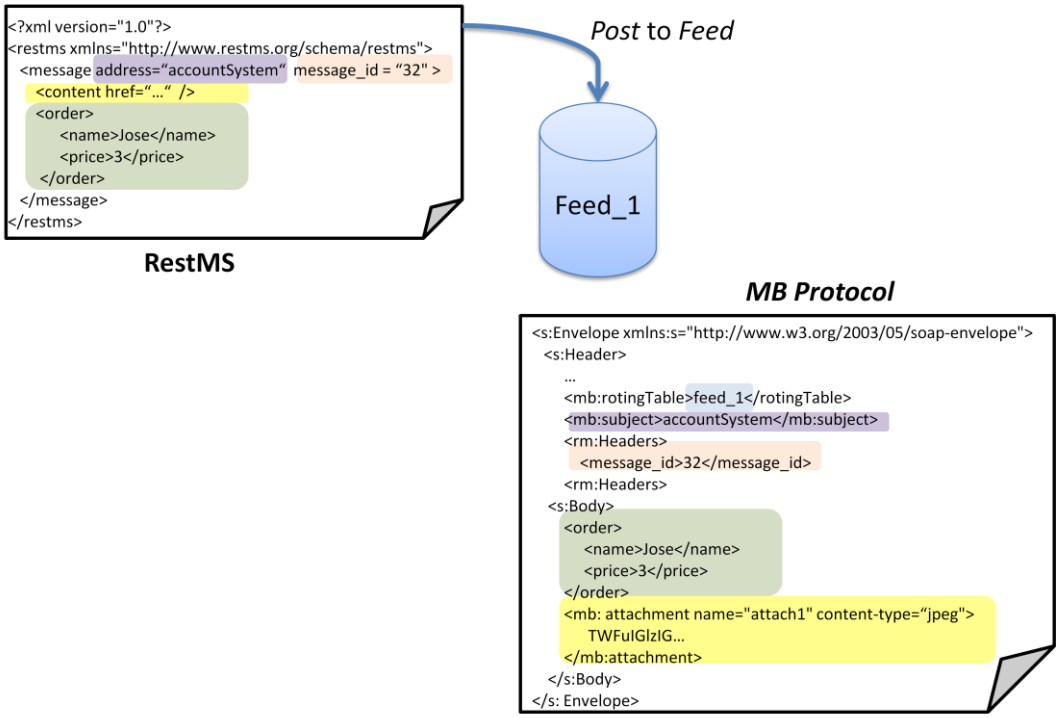


Figura 44: Conversão de mensagens de dados RestMS para *MB Protocol*

Apesar da norma RestMS referir que os dados das mensagens podem estar formatados em XML ou JSON, apenas está documentada a formatação em XML. Após o elemento raiz `<restms>`, as mensagens de dados RestMS contêm o elemento `<message>`. Este elemento é especifico ao protocolo RestMS não podendo por isso constar no *payload* das mensagens *MB Protocol*. No entanto, nele está contida informação que poderá ser relevante para os clientes RestMS. Se o destino final da mensagem for um cliente RestMS, então a mensagem tem de ser reconstruída de acordo com a original. Caso o destino seja um cliente de um protocolo diferente, então o elemento `<message>` não é relevante. Para permitir a reconstrução da mensagem original, os atributos do elemento `<message>` são guardados na mensagem *MB Protocol* em `/s:Envelope/s:Header/rm:Headers`. Coloca-se o problema de como gerar estes atributos no caso do remetente da mensagem não ser um cliente RestMS. Esse problema terá de ser a implementação do protocolo a resolver e no caso específico do *message_id* é gerado automaticamente um identificador único. No entanto, este pode ser um factor que gere dificuldades na implementação de determinado protocolo de comunicação.

Outro aspecto onde os formatos RestMS e *MB Protocol* diferem é na inclusão de anexos na mensagem. No *MB Protocol* os anexos consistem em elementos `<mb:attachment>` incluídos no elemento `<s:Body>`, onde o conteúdo do anexo circula codificado em base64 e o atributo *content-type* permite dar uma orientação à aplicação destino de como interpretar o conteúdo do anexo. Em RestMS os anexos são incluídos através do elemento `<content>` que poderá ter uma referência para um *conteúdo* (recurso) previamente enviado ou então incluir os dados do conteúdo no próprio elemento.

Depois de extraído o elemento `<message>` e convertidos todos os elementos `<content>`, tudo o que sobrar (representado a verde) é interpretado como dados para incluir no elemento `<s:Body>`.

As restantes informações, como o nome da tabela de encaminhamento a ser utilizada ou o *subject* são dados que existem nos dois protocolos, pelo que a sua conversão não foi problema.

4.5 Módulo de encaminhamento

A implementação do módulo de encaminhamento recebeu o nome de *Router*. Trata-se da implementação de um *router dinâmico* [Hoh04], isto é, as tabelas de encaminhamento sofrem alterações ao longo da execução do programa (o *bus*, neste caso). O *Router*, enquanto módulo, permite mais do que uma instância estando as suas principais funcionalidades implementadas na classe `DinamicRouter` que implementa `MBusMultiInstanceModule` e cujo diagrama de classes se apresenta na Figura 45.

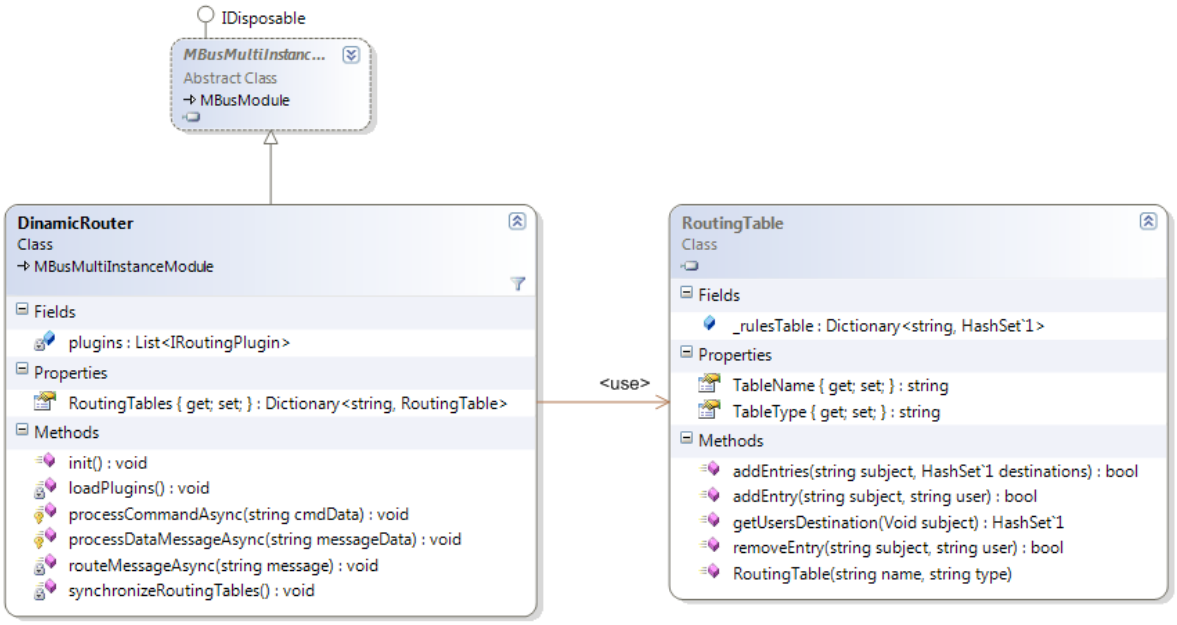


Figura 45: Diagrama de classes do módulo de encaminhamento

As tabelas de encaminhamento são fornecidas pela propriedade `RoutingTables` que consiste num dicionário de pares `string`, `RoutingTable`. A `string` identifica a tabela através de um nome único enquanto o objecto `RoutingTable` representa a tabela de encaminhamento propriamente dita. Um `RoutingTable` representa tanto tabelas de encaminhamento *um para um* como tabelas de encaminhamento *um para muitos* sendo o tipo da tabela identificado pela propriedade `TableType`.

O algoritmo de encaminhamento é implementado no método `processDataMessageAsync` e tem em conta o valor de dois campos presentes no cabeçalho SOAP das mensagens – o `<mb:subject>` e o `<mb:routingTasble>`. O primeiro funciona como uma "chave" utilizada para encaminhamento e o segundo contém o nome da tabela de encaminhamento a utilizar. O significado atribuído ao elemento *subject* varia conforme o tipo da tabela de encaminhamento utilizada pelo *Router*. No caso de uma tabela *um para um* então o *subject* corresponde ao nome da aplicação destino da mensagem. Caso a tabela de encaminhamento seja do tipo *um para muitos*, então o *subject* corresponde ao nome do tópico utilizado no modo *publicador-subscritor*.

Após executado o algoritmo de encaminhamento, o *Router* escreve na mensagem o nome dos destinos finais, utilizando para isso o campo `<mb:destination>` no cabeçalho SOAP.

Se por algum motivo, for necessário um algoritmo de encaminhamento diferente, por exemplo, um que tenha em conta outros campos da mensagem, então o administrador do sistema poderá instalar algoritmos feitos à sua medida. Os novos algoritmos são implementados de acordo com o padrão *plugin* [Fow03] (ver secção 4.8 para detalhes acerca deste padrão). As implementações dos *plugins* têm acesso à mensagem SOAP recebida pelo módulo *Router* e também às tabelas de encaminhamento. Cada *plugin* deve adicionar destinos ao elemento `<mb:destination>` tendo em conta que o conteúdo final deste elemento inclui o resultado do processamento de todos os *plugins* bem como o resultado do algoritmo base fornecido pelo *bus* de mensagens.

O módulo de encaminhamento faz uso da base de dados para partilhar informação entre as várias instâncias. A informação que é necessário manter em base de dados é o conteúdo de cada uma das tabelas de encaminhamento e cujo modelo relacional se encontra na Figura 46.

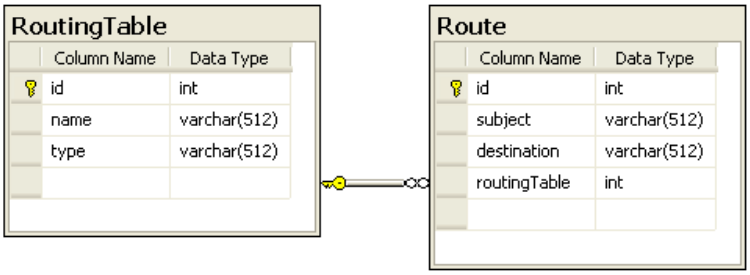


Figura 46: Modelo de relacional referente ao *Router*

A entidade *RoutingTable* armazena a informação das tabelas de encaminhamento (nome e tipo). Cada linha da entidade *Route* representa uma entrada de uma tabela de encaminhamento. A tabela de encaminhamento à qual a entrada se refere é indicada através de uma chave estrangeira para *RoutingTable*.

A informação presente na base de dados é consultada apenas uma vez durante todo o tempo de execução do *bus* de mensagens. No carregamento do módulo, durante o método `init`, é invocado o método `synchronizeRoutingTables` que preenche o campo `RoutingTables` de acordo com a informação na base de dados.

4.5.1 Comandos

Ao módulo de encaminhamento são entregues os comandos para consultar, adicionar, actualizar ou remover tabelas de encaminhamento e adicionar ou remover entradas de tabelas de encaminhamento.

Cada comando é implementado por uma classe que estende de `Command` (ver Figura 47). Para tornar a explicação mais simples, até ao fim deste capítulo será apenas exposta a implementação do comando para adicionar tabelas de encaminhamento (*addRoutingTable*)

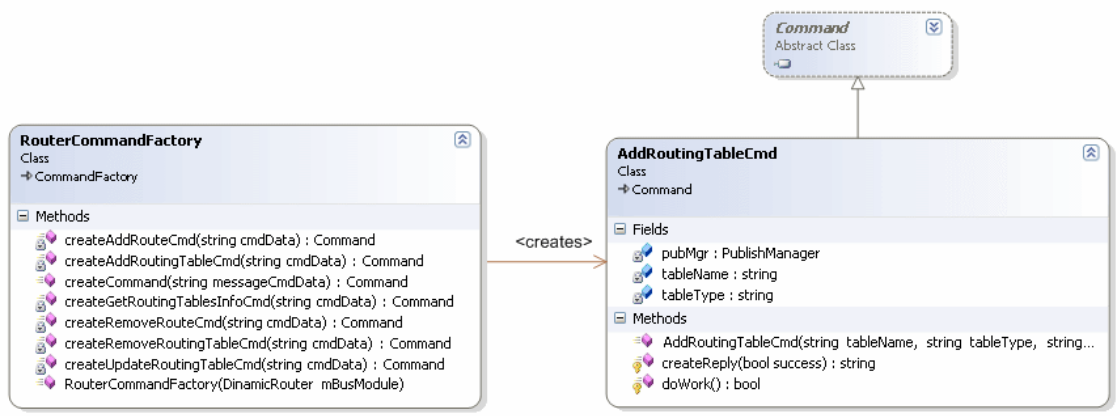


Figura 47: Diagrama de classes da implementação de comandos no *Router*

O método `createCommand` de `RouterCommandFactory` inspecciona o nome do primeiro elemento existente no *Body* da mensagem SOAP e com base nele constrói o objecto Command correspondente.

A implementação do comando faz uso do sistema de notificações WSP e cujo diagrama de classes dos objectos envolvidos se apresenta na Figura 48.

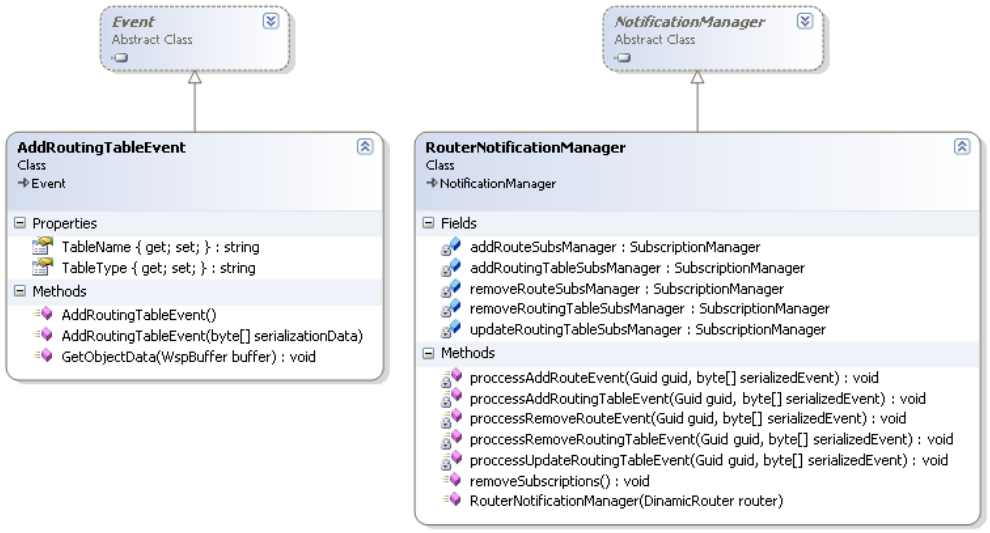


Figura 48: Diagrama de classes do sistema de notificações do *Router*

Por cada comando que envolva alteração do estado no módulo, é criada uma classe que deriva de `Event` (tipo pertencente à API do WSP). Essa classe contém em propriedades os dados que serão acrescentados, alterados ou removidos.

Posto isto, a classe `AddRoutingTableCmd` contém um campo do tipo `PublishManager` (pertencente à API do WSP). A implementação de `doWork` cria um `AddRoutingTableEvent` e publica-o, utilizando o `PublishManager`, a todas as aplicações interessadas.

Os subscritores de eventos (cada uma das instancias de *Router*) possuem um objecto do tipo `RouterNotificationManager` que recebe os eventos e realiza as alterações necessárias no estado da instância do módulo.

4.6 Módulo de transformação

Transformer foi o nome dado ao módulo responsável pela transformação de dados. Definiu-se um conjunto de tipos onde cada um é responsável por realizar uma transformação específica. Todos esses tipos (transformadores) implementam uma interface comum, conforme se pode verificar no diagrama de classes da Figura 49.

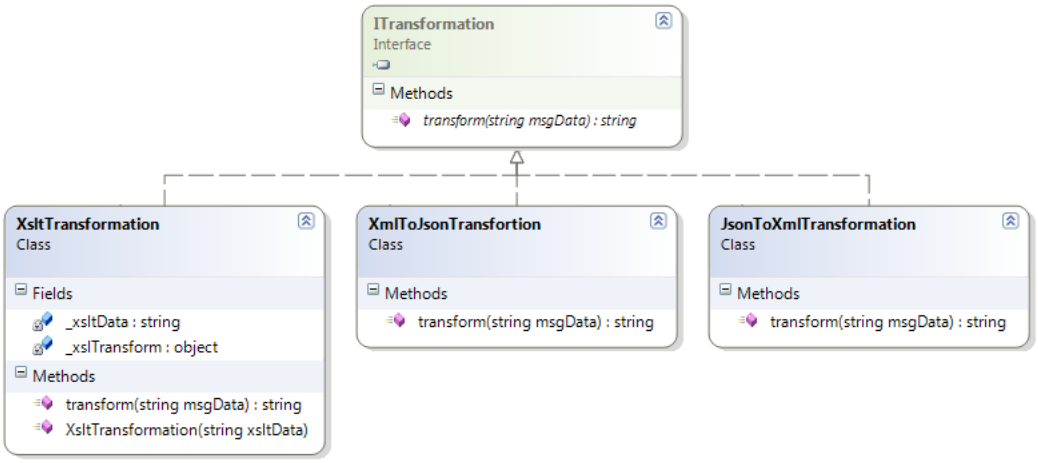


Figura 49: Diagrama de classes dos transformadores

A interface `ITransformation` define que todos os transformadores implementam o método `transform`. Este método recebe a mensagem de dados SOAP recebida pelo módulo de transformação, efectua a transformação e retorna a mensagem alterada.

Relativamente à implementação de cada um dos transformadores, o `XsltTransformation` utiliza objectos `XslCompiledTransform` e *strings* com o

conteúdo XSLT para modificar mensagens XML (a campo `_xslTransform` destina-se a melhorar o desempenho do transformador evitando criar um `XslCompiledTransform` por cada transformação).

Os transformadores `XmlToJsonTransformation` e `JsonToXmlTransformation` utilizam a biblioteca *Json.Net* externa à plataforma .NET e que está disponível em [Cod10b].

Posto isto, resta criar forma de identificar quando é que uma mensagem precisa de ser transformada. A transformação de uma mensagem é uma acção transparente para a aplicação remetente da mensagem. Por outro lado, surge com uma necessidade da aplicação destino, fruto da forma como esta espera encontrar os dados na mensagem. Assim sendo, terá de ser com base no destino final da mensagem que se decide quando se aplica uma transformação. No entanto, as mensagens não podem ser todas transformadas apenas por se destinarem a determinada aplicação. É necessário um mecanismo que permita identificar padrões na mensagem que, caso sejam verificados, é transformada a mensagem. Visto que todas as mensagens estão em formato SOAP, que é um formato baseado em texto, a detecção de padrões nas mensagens é feita utilizando expressões regulares. Em síntese, o critério de decisão de aplicar uma transformação ou não, é tomado com base no destino da mensagem e numa expressão regular que indica como é que os dados na mensagem de origem têm de se encontrar para que seja decidido aplicar determinada transformação.

A principal classe no módulo de transformação é o `Translator` cujo diagrama de classes se mostra na Figura 50.

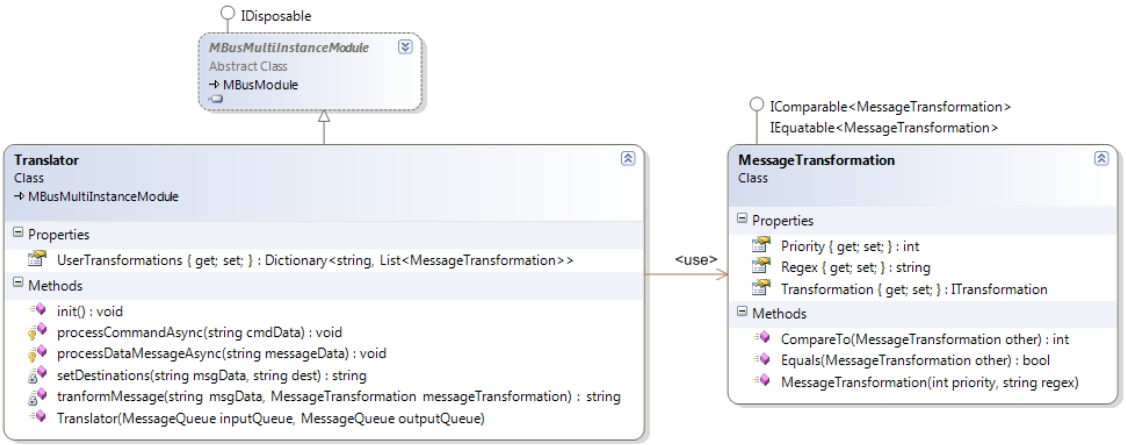


Figura 50: Diagrama de classes do módulo de transformação

A classe `Translator` deriva de `MBusMultiInstanceModule`, permitindo assim executar em simultâneo mais do que uma instânciadeste módulo. Internamente contém um dicionário que associa destinos de mensagens (nomes de clientes) a objectos `MessageTransformation`. Um `MessageTransformation` contém o objecto `ITransformation` que realiza a transformação da mensagem e a expressão regular. O método `processDataMessageAsync` obtém da mensagem os destinos finais que esta terá e, para cada destino, verifica se existe uma entrada no dicionário. Caso exista, aplica-se a expressão regular e se for verificada correspondência, realiza-se a transformação. A Listagem 9 mostra o exemplo de uma expressão regular que verifica se a mensagem está no formato interno do *bus* e se o campo de dados se trata de informação XML onde o elemento raiz tem o nome "OrderInfo".

```
public static string regexSample = "<s:Envelope.*[\\w]*.*>.*[\\w]*.*<s:Header.*[\\w]*.*>.*[\\w]*.*<s:Body.*[\\w]*.*><OrderInfo>";
```

Listagem 9: Exemplo de expressão regular

É possível aplicar à mesma mensagem mais do que uma transformação. Dependendo da situação, a ordem pela qual as transformações são aplicadas pode não ser indiferente. Por exemplo, numa mensagem XML onde seja necessário remover um elemento e converter o resultado em JSON, a transformação a ser aplicada em primeiro lugar terá de ser o XSLT. Assim sendo, criou-se um mecanismo que permite às aplicações indicar a ordem de execução das transformações. Esse mecanismo consiste num valor inteiro que indica a prioridade da transformação e onde zero é o valor máximo de prioridade. Assim, no caso de existirem três transformações para a mesma mensagem, uma com prioridade zero, outra com um e outra com seis. Será aplicada primeiro a de zero, seguida pela de um e finalmente a de seis. Os objectos `MessageTransformation` existentes na lista incluída no dicionário estão ordenados por índice de prioridade. Para transformações com igual nível de prioridade não existe garantia de qual será executada em primeiro lugar.

4.6.1 Comandos

Definiu-se um comando por cada transformação existente. Cada comando contém os parâmetros com a informação necessária à criação do objecto `ITransformation`

correspondente, sendo que todos eles têm pelo menos de incluir o nome do cliente destino, a expressão regular e o nível de prioridade.

Sendo este um módulo que suporta várias instâncias, a implementação de cada comando utiliza o WSP para notificar as restantes instâncias, existindo ainda o objecto `TranslatorNotificationManager` que trata de processamento dos eventos recebidos.

4.7 Módulo de segurança

O módulo de segurança recebeu o nome de *Security* e trata-se de um módulo que suporta mais do que uma instância. A Figura 51 representa o diagrama de classes da classe que implementa o módulo.

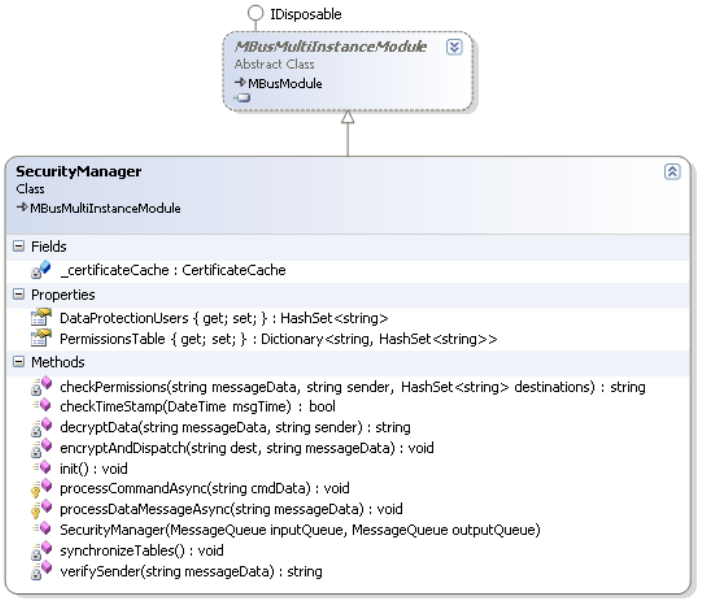


Figura 51: Diagrama de classes do módulo *Security*

A lista de aplicações cliente que requereram transporte de dados cifrados entre o *bus* e o cliente é guardada no campo acessível através da propriedade `DataProtectionUsers`. A propriedade `PermissionsTable` disponibiliza um dicionário onde se guarda, para cada aplicação cliente, o nome dos clientes que estão autorizados a enviar-lhe mensagens.

O método `processDataMessageAsync` começa por verificar a identidade do remetente da mensagem. Depois disso, é obtida uma lista dos destinos da mensagem, verificando-se para cada destino se existem políticas de envio e, caso existam, se o remetente está na lista de aplicações autorizadas a enviar mensagens para esse destino.

Finalmente verifica-se se os dados da mensagem estão cifrados (caso em que se efectua a decifra) e se os destinos da mensagem exigem dados cifrados (realizando-se a cifra).

À semelhança do módulo de encaminhamento, também neste módulo se utiliza a base de dados para persistência e partilha de informação entre as várias instâncias. A Figura 52 mostra o modelo relacional das tabelas criadas.

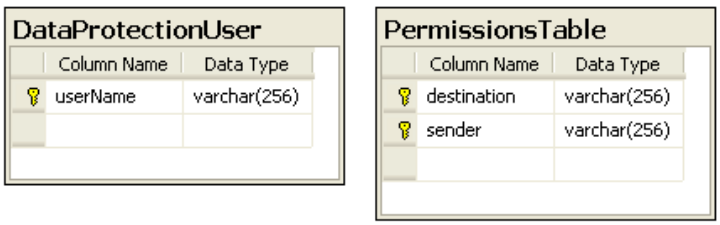


Figura 52: Modelo relacional referente ao módulo de segurança

Durante o carregamento de cada instância do módulo, o método `synchronizeTables` realiza a leitura das tabelas *DataProtectionUser* e *PermissionsTable* preenchendo os campos homónimos do objecto `SecurityManager`.

4.7.1 Autenticação

Para realizar as funcionalidades deste módulo, é necessário um mecanismo que permita autenticar as aplicações cliente. O mecanismo implementado consiste na *prova de chave privada*. Algoritmos de assinatura assimétricos utilizam um par de chaves criptográficas composto por uma chave privada que apenas uma das partes tem acesso e uma chave pública, à qual têm acesso todos os outros participantes.

O mecanismo de autenticação desenvolvido consiste em fornecer ao *bus* de mensagens uma chave pública associada ao nome de uma aplicação cliente. De seguida, nas comunicações em que for necessário o cliente autenticar-se, será incluído pelo cliente um cabeçalho onde consta a informação do seu nome assinada com a chave privada. Quando o *bus* verificar a validade da assinatura, prova-se que quem emitiu a mensagem tem em sua posse a chave privada que está associada à chave pública fornecida ao *bus* para aquele cliente [Gol06] As chaves públicas são armazenadas em certificados X509 localizados no *certificate store* do Windows e não é da responsabilidade do *bus* o transporte e instalação dos certificados. Os certificados funcionam apenas como meio de associar uma chave pública a um nome de utilizador (*subject*) não sendo feita verificação da cadeia do certificado.

O método de autenticação atrás descrito tem a fragilidade de um atacante poder capturar uma mensagem assinada e mais tarde repeti-la. O problema pode ser minimizado incluindo nos dados assinados da mensagem a data e hora em que a mensagem foi produzida. Desta forma, define-se um tempo máximo durante o qual é aceitável que a mensagem demore no transporte (por exemplo, 5 minutos) e todas as mensagens recebidas que tenham sido criadas há mais tempo, são descartadas. Contudo, esta solução obriga a que o *bus* e os clientes tenham um relógio sincronizado. Considerou-se que o peso de tal solução não se justifica e optou-se por uma solução em meio-termo. A autenticação inclui data e hora da criação da mensagem mas a tolerância dada pelo servidor são 13h. Assim, por um lado consegue-se minimizar os efeitos de um ataque por repetição de mensagens e pelo outro, apenas é exigido às aplicações cliente que tenham a data correcta.

A Listagem 10 mostra um excerto de uma mensagem de dados autenticada.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>data</mb:MessageType>
    ...
    <mb:security encryptedPayload = "false">
      <sender>SalesSystem</sender>
      <timeStamp>26-06-2010 19:57:44</timeStamp>
    </mb:security>
    <mb:securitySignature>TWFuIGbIG...</mb:securitySignature>
    ...
  </s:Header>
  <s:Body>
     ...
  </s:Body>
</s:Envelope>
```

Listagem 10: Exemplo de mensagem autenticada

O elemento `mb:security` contém o nome do cliente remetente da mensagem acompanhado da marca temporal. O conteúdo desse elemento é assinado e a assinatura convertida em Base64 é colocada no elemento `mb:securitySignature`. Ao validar a assinatura, fica a garantia de que quem criou a mensagem está em posse da chave privada associada à chave pública fornecida ao *bus*, para o cliente indicado.

A assinatura é produzida utilizando o algoritmo RSA, com chaves de 1024 bits e algoritmo de *hash* SHA-1.

4.7.2 Cifra de dados

A protecção dos dados de uma mensagem é feita cifrando o conteúdo do elemento *body* da mensagem. Este elemento é cifrado com recurso a um algoritmo simétrico, o que implica que a chave utilizada para cifrar é a mesma que faz a decifra. Essa mesma chave é incluída na mensagem e protegida por um algoritmo de cifra assimétrico (ver Listagem 11).

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>data</mb:MessageType>
    ...
    <mb:security encryptedPayload = "true">
      <sender>sales</sender>
      <timeStamp>26-06-2010 19:57:44</timeStamp>
    </mb:security>
    <mb:AES>
      <key>AHD64Sgs8...</key>
      <iv>LYH6ghsASjh...</iv>
    </mb:AES>
    ...
  </s:Header>
  <s:Body>
     AhjGT546jHG...
  </s:Body>
</s:Envelope>
```

Listagem 11: Exemplo de mensagem cifrada

Sempre que o atributo `encryptedPayload` do elemento `mb:security` tiver o valor *true*, significa que os dados presentes em `<s:Body>` estão cifrados. O elemento `<mb:AES>` contém a chave simétrica utilizada nessa cifra (acompanhada pelo *initial vector*). Essa chave está ela própria protegida por um algoritmo de cifra assimétrico. Para reaproveitar os recursos já existentes no *bus*, o algoritmo de cifra assimétrico é o RSA, onde são utilizadas chaves de 1024 bit e *padding* PKCS#1. É garantido que apenas o destino final da mensagem terá a chave privada que permite decifrar a chave simétrica para posteriormente decifrar o conteúdo de `<s:Body>`.

O algoritmo simétrico utilizado foi o AES com chaves de 128 bits, em modo de operação CBC e *padding* PKCS#7. Note-se que na Listagem 11, todos os campos que contêm cifras, estão codificados em Base64.

Como classe auxiliar do módulo de segurança, implementou-se o `CryptoUtils` que contém métodos de criptografia para cifra e assinatura de dados e cujo diagrama de classes se encontra na Figura 53.

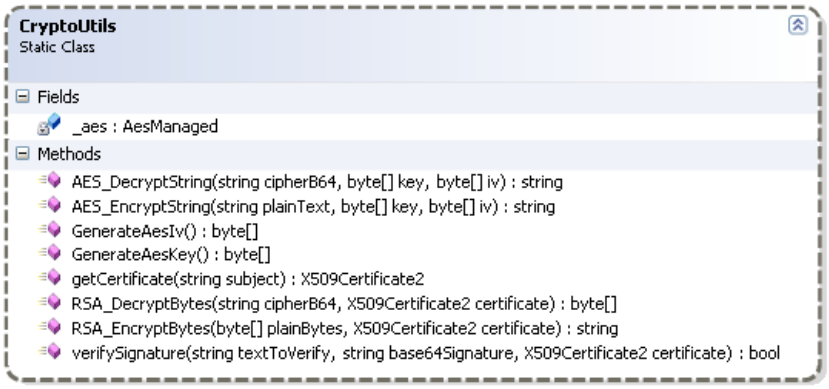


Figura 53: Diagrama de classes de `CryptoUtils`

4.7.3 Cache de certificados

O funcionamento deste módulo exige que durante o processamento de uma mensagem de dados sejam obtidos vários certificados do *certificate store*. O *certificate store* é um recurso disponibilizado pelo sistema operativo e cuja utilização envolve acessos ao sistema de ficheiros, o que do ponto de vista do tempo de processamento de uma mensagem, é uma acção demorada.

Para melhorar o desempenho do módulo de segurança, foi criado um sistema de cache de certificados X509. O sistema armazena os últimos certificados utilizados. Os certificados são solicitados através do seu *subject*. Quando se solicita um certificado, o sistema verifica se esse certificado existe em *cache*, caso não exista, é obtido do *certificate store* e adicionado à *cache*. A cache tem uma dimensão máxima e se estiver cheia no momento em que é obtido um certificado do *certificate store*, então o certificado que estiver há mais tempo sem ser utilizado é removido, dando lugar ao mais recente.

A Figura 54 ilustra o diagrama de classes da classe que implementa a *cache* de certificados.

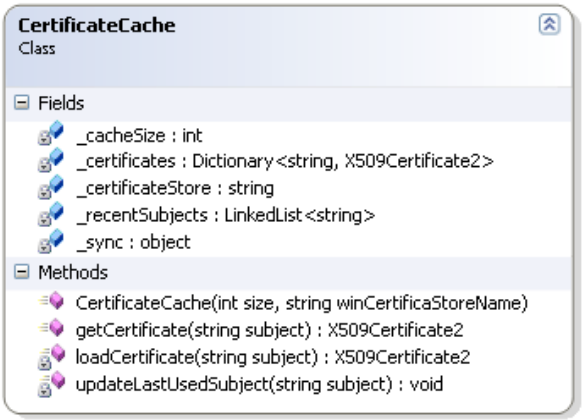


Figura 54: Diagrama de classes de `CertificateCache`

O método `getCertificate` é o utilizado pelo módulo *Security* para obter os certificados. Internamente a classe faz uso de um dicionário para guardar cada certificado associado ao seu nome (*subject*) e de uma lista que contém os nomes de cada *subject* ordenados por último acesso.

Como cada mensagem de dados é processada numa *thread* independente, poderá haver concorrência no acesso ao objecto `CertificateCache`. Por esta razão, a implementação da classe `CertificateCache` é *thread safe*.

4.7.4 Comandos

O módulo de segurança é responsável pelo processamento dos comandos:

- setSecureCommunication;
- addAllowedSender;
- removeAllowedSender.

Por este ser um módulo que permite várias instâncias, a implementação do processamento destes comandos segue a mesma lógica do *Router* detalhada no capítulo 4.5.1.

No geral, não existe verificação da entidade que envia os comandos para o *bus*. No entanto, no caso dos três comandos do módulo de segurança, não faz sentido permitir que qualquer aplicação os utilize. Repare-se por exemplo, se tal acontecesse, uma aplicação α que não estivesse autorizada a enviar dados para a aplicação β, poderia contornar o sistema de segurança simplesmente enviando um comando

addAllowedSender. Para evitar isto, estes três comandos têm de incluir uma assinatura com a chave associada ao cliente visado pelo comando.

4.8 Módulo de extensibilidade

O módulo de extensibilidade recebeu o nome de *Extensibility* e permite estender as próprias funcionalidades do *bus* de mensagens. Existiram dois requisitos na implementação deste módulo:

- Ter forma de acrescentar novas funcionalidades sem recompilar o código fonte do *bus*;
- Poder configurar em que momento do processamento de uma mensagem é que determinada nova funcionalidade é executada.

O primeiro requisito é alcançado através da implementação do padrão *plugin* [Fow03]. A Figura 55 pretende dar uma visão ilustrada do padrão *plugin*.

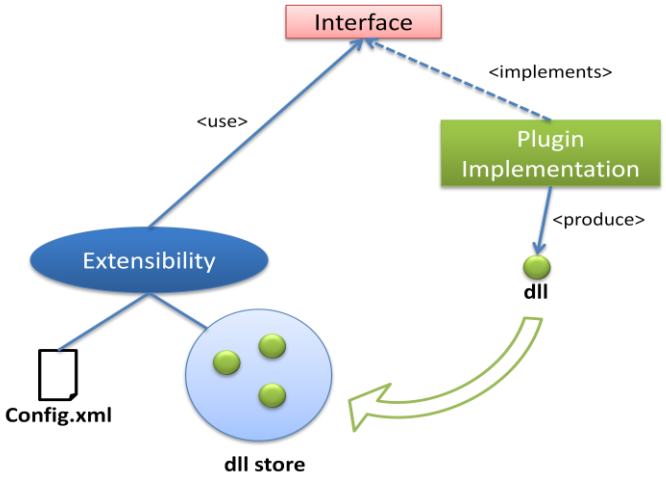


Figura 55: Implementação do padrão *plugin*

No padrão *plugin* existem dois participantes – a entidade que utiliza código externo (neste caso o módulo *Extensibility*) e a entidade que implementa esse código. Uma interface conhecida por ambas as entidades define o modo de interacção entre o *Extensibility* e o código externo. A implementação do código externo implementa essa interface e distribui o resultado através de uma biblioteca (dll). A entidade que utiliza o código externo dispõe de um repositório onde estão armazenadas as várias implementações e, fazendo uso de um ficheiro de configuração, instancia cada uma das implementações.

No caso do *bus* de mensagens, o ficheiro de configuração chama-se plugins.xml e contém, entre outras informações, um nome que identifica a implementação da nova funcionalidade, a localização da biblioteca (dll) correspondente e o nome completo do *assembly* existente na biblioteca.

A principal classe que implementa o módulo *Extensibility* recebeu o nome de `ExtensibilityManager` e tem o seu diagrama de classes representado na Figura 56.

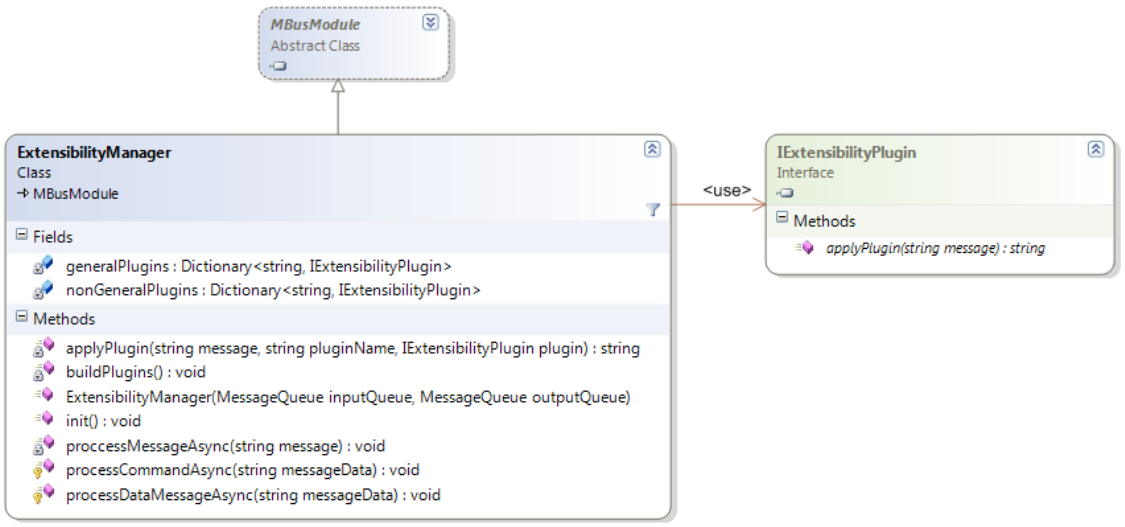


Figura 56: Diagrama de classes da implementação do módulo de extensibilidade

A interface `IExtensibilityPlugin` é incluída num projecto separado e é onde se define o contrato que cada implementação externa tem de respeitar. Neste caso, trata-se apenas de um método que recebe uma mensagem SOAP e tem o mesmo retorno, podendo a mensagem retornada ser igual ou não à recebida.

Indo de encontro ao segundo requisito referido no início deste capítulo, no *workflow* que define a ordem de participação de cada módulo, podem existir actividades que consistem na execução de um *plugin* específico ou uma actividade geral que envia a mensagem para o módulo *Extensibility* e onde são executados todos os *plugins* não específicos. No momento da recepção de uma mensagem na fila de entrada, a forma que o módulo tem de distinguir se é para executar um *plugin* específico ou não, é através de um cabeçalho SOAP que indica o nome do plugin a ser executado.

A implementação de `processMessageAsync` começa por procurar o cabeçalho que dá indicação de um *plugin* específico. Se for encontrado, executa esse plugin e coloca o resultado na fila de saída. Se não for encontrado então executa todos os restantes plugins colocando o resultado final na fila de saída.

No ficheiro de configuração, os *plugins* que são invocados no workflow por actividades dedicadas têm o atributo `dedicatedWfActivity="true"`. Durante o carregamento do módulo, o ficheiro de configuração é lido e constrói-se por introspecção uma instância de cada plugin. As instâncias criadas são armazenadas (para todo o tempo de vida do módulo) nos contentores `generalPlugins` ou `nonGeneralPlugins` conforme tenham indicação de que são ou não invocados por actividades dedicadas.

Para exemplificar a utilização deste módulo foi adicionada uma extensão que realiza o registo de todas as mensagens que passam no *bus*.

4.9 Módulo de gestão

Até aqui cada módulo foi apresentado como uma unidade autónoma que recolhe mensagens de uma fila de entrada, faz determinado processamento e coloca-as numa fila de saída. A autonomia de cada módulo faz com nenhum módulo se preocupe com o momento em que participa no processamento da mensagem.

O módulo de gestão, ao qual se chamou *Manager*, é um módulo especial por não participar directamente no processamento das mensagens e ser o módulo que decide quando é que todos os outros entram em acção. O *Manager* não tem apenas uma fila de entrada e outra de saída. Todas as filas de saída dos restantes módulos são filas de entrada para o módulo *Manager* e as filas de saída do *Manager* são filas de entrada de outros módulos.

Como já foi referido, a configuração do processamento de cada mensagem é feito através de um *workflow*. Cada módulo do sistema, ao efectuar o seu processamento, deixa na mensagem uma marca específica que consiste num cabeçalho SOAP. No *workflow* as mensagens são inspeccionadas, determinando-se o seu tipo (dados ou comando) e verificando-se por que sítios já passaram. Esta informação serve depois para tomar a decisão de onde colocar a mensagem.

A Figura 57 exemplifica um processo de *workflow* utilizado no processamento das mensagens.

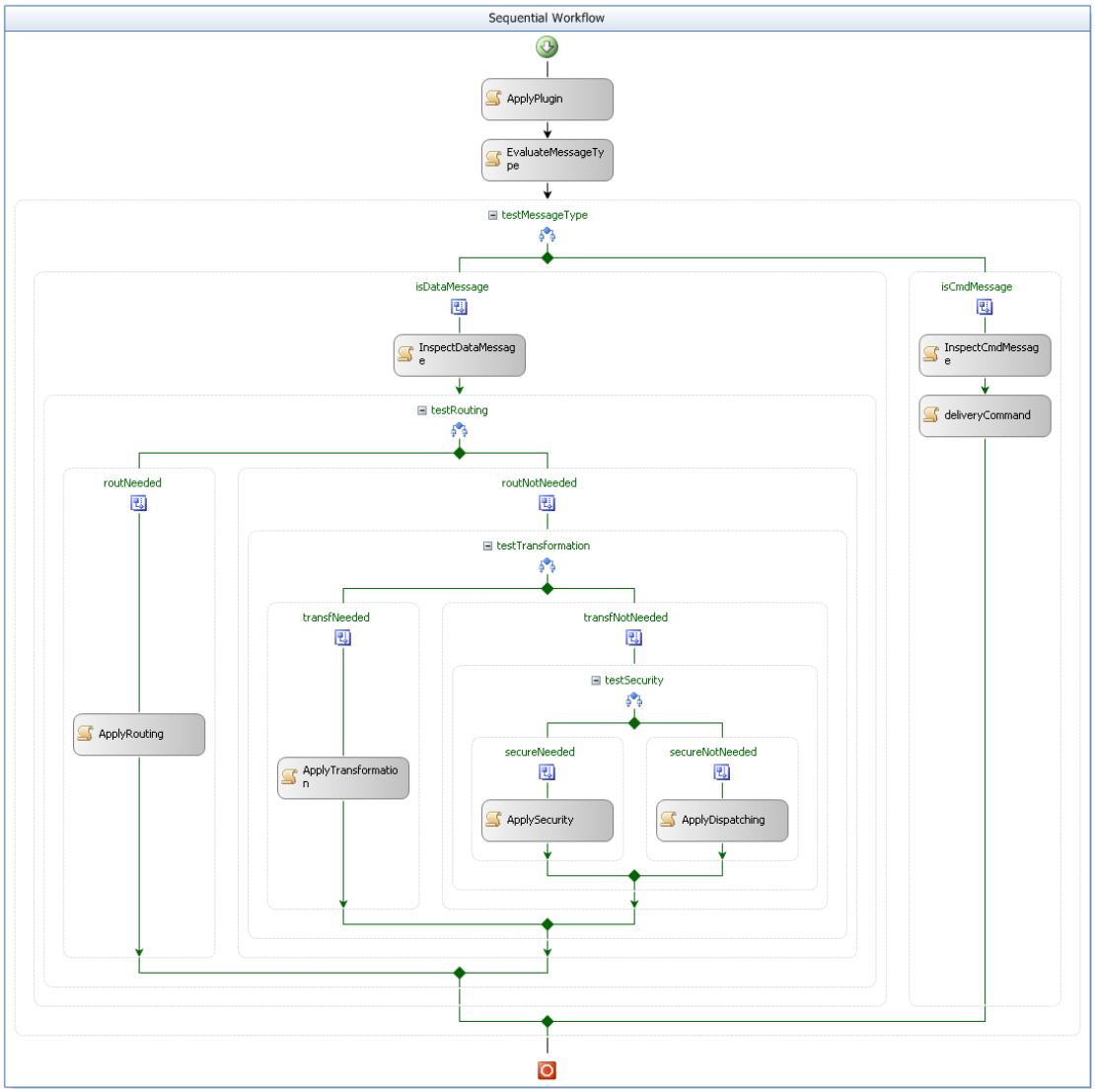


Figura 57: Exemplo de *workflow*

Para optimizar o processamento das mensagens, o *bus* só envia as mensagens para o módulo de extensibilidade caso existam *plugins* instalados. Por este motivo, o módulo *Manager* tem de ter acesso ao ficheiro de configuração plugins.xml. Caso o módulo de extensibilidade e o *Manager* não estejam a ser executados na mesma máquina, então este ficheiro tem de ser replicado.

4.10 Extensão de comandos

No capítulo 3.4 foi apresentada a lista de comandos suportados de raiz pelo *bus* de mensagens. Para além destes comandos de raiz, é importante ter uma forma de

implementar novos comandos, idealmente sem ser necessário recompilar ou alterar o código original do *bus* de mensagens. Um exemplo da utilidade de um novo comando é a implementação de um novo protocolo de comunicação que contenha uma especificidade não prevista à data da realização deste trabalho e, por isso, não suportada pelos comandos existentes. Outro exemplo onde é necessário implementar novos comandos é na inclusão de novos tipos de transformações (a transformação `xpto` necessita do comando `AddXptoTransformation`).

Aos novos comandos, definidos pelo utilizador, dá-se o nome de *custom commands*. Para que sejam reconhecidos pelo *bus* como tal, todos os c*ustom commands* têm de obedecer a algumas regras no que diz respeito ao seu formato. A Listagem 12 mostra o formato geral de um c*ustom command*.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType> command </mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD…</mb:CorrelationID>
    <!--outros headers reconhecidos pelo bus-->
  </s:Header>
  <s:Body>
    <mb:customCommand id ="1" target = "Transformer">
      <!-- parâmetros do comando (se houver)-->
    </mb:customCommand>
  </s:Body>
</s:Envelope>
```

Listagem 12: Formato genérico de um *custom command*

Todos os c*ustom commands* apresentam no *soap body* um único elemento chamando *customCommand*. Esse elemento contém como atributos um *id* que identifica o comando e um atributo que indica o módulo do *bus* responsável pelo processamento do comando. O corpo do elemento `<customCommand>` é livre, destinando-se a conter informação relevante para a sua execução.

A ideia na extensibilidade de comandos é que seja o utilizador a fornecer o código (sob a forma de bibliotecas) que sabe processar cada um dos novos comandos. Desta forma, sempre que for recebido um comando no formato da Listagem 12, o *bus* identifica qual a biblioteca responsável por processar o comando e passa-lhe na íntegra a mensagem SOAP que representa o comando. Para estabelecer a correspondência entre os novos comandos (identificados pelo *id*) e a biblioteca responsável pela sua execução, é utilizado um ficheiro de configuração. Neste caso, utilizou-se o ficheiro MBconfig.xml cujo excerto se apresenta na Listagem 13.

```
<configurations>
  ...
  <customCommands>
    <addXptoTransformation id="1"
                           targetModule="Transformer"
                           typeName="CustCmd.AddXptoTransformationCmd"
                           filePath="C:\MB\cmds\AddXptoTransf.dll" />
  </customCommands>
  ...
</configurations>
```

Listagem 13: Configuração de custom commands em MBconfig.xml

As implementações de cada comando implementam a interface `ICustomCommand`, cujo diagrama de classes se mostra na Figura 58.

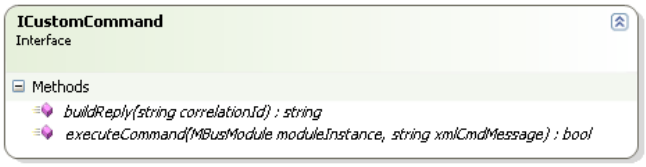


Figura 58: Diagrama de classes da interface ICustomCommand

Seguindo a lógica dos objectos `Command` já existentes no *bus*, as implementações dos novos comandos são decompostas em dois métodos. O método `executeCommand`, que efectivamente realiza as acções que caracterizam o comando e o método `buildReply` que constrói a mensagem de resposta a enviar ao cliente, caso seja requerida. Se a implementação de `buildReply` retornar o valor `null`, então o *bus* de mensagens gera a mensagem de resposta por omissão, que apenas contém o nome do comando (neste caso "customCommand" concatenado com o *id*) e o *correlationID* que permite ao cliente identificar a que comando pertence a resposta.

A classe `CommandFactory` representada na Figura 31 implementa o método `createCustomCommand` que deverá ser utilizado pelas classes base sempre que for necessário construir um *custom command*. Este método retorna um objecto do tipo `CustomCommandExecuter` que estende a classe Command (ver Figura 31). A execução de um *custom command* varia conforme o módulo ao qual o comando se destina seja um módulo que permite várias instâncias ou não. No caso de se tratar de um módulo de apenas uma instância, então utiliza-se a instância do objecto que implementa ICustomCommand para realizar de imediato os passos do comando. No caso de se tratar

de um módulo de várias instâncias, então utiliza-se o sistema de notificações [Cod10a] já anteriormente apresentado para notificar todas as instâncias do módulo, permitindo assim que o comando seja executado em todas as instâncias.

Cada módulo, ao ser iniciado, verifica quais os *custom commands* que lhe estão associados e instancia por introspecção os objectos que os executam, armazenando-os no dicionário `CustomCmds`. Note-se que durante todo o tempo de execução do *bus* de mensagens é criada apenas uma instância de cada objecto `ICustomCommand`, permitindo assim às implementações dos comandos armazenar estado entre as várias chamadas.

4.11 Tolerância a falhas

A tolerância a falhas reflecte a capacidade de um sistema em recuperar de situações anómalas como, por exemplo, uma falha de *hardware*. No caso de um *bus* de mensagens, "recuperar" significa voltar ao correcto funcionamento sem perder nenhuma mensagem e mantendo o estado existente antes da ocorrência da falha. O *bus* de mensagens implementado inclui algumas características que lhe permitem a recuperar face a alguns tipos de falhas.

As mensagens (sejam de dados ou comandos) são enviadas para o *bus* segundo a lógica *send and forget* [Hoh04], significando isto que, uma vez que o *bus* tenha confirmado a recepção da mensagem, assume a responsabilidade de processar essa mensagem sem a perder. Desta forma, a necessidade de persistência das mensagens começa no momento em que for confirmado, através do mecanismo definido pelo protocolo de comunicação, à aplicação remetente a recepção de uma mensagem.

O padrão *pipes and filters* [Hoh04] utilizado na arquitectura interna do *bus* oferece vantagens na implementação da persistência, uma vez que as filas MSMQ podem funcionar como pontos intermédios onde se garante que uma mensagem não é perdida. Desta forma, a implementação da persistência de mensagens, passou por construir filas MSMQ transaccionais. Para optimizar o desempenho, o MSMQ pode nalguns casos manter as mensagens de uma fila apenas em memória RAM. No entanto, no caso de uma fila ser transaccional, as mensagens são sempre armazenadas em disco rígido [Red04]. Mantendo as mensagens em disco e utilizando transacções na leitura e

escrita garante-se a persistência das mesmas à medida que vão transitando entre os vários módulos do *bus*.

Os acessos às filas MSMQ num contexto transaccional são feitos utilizando transacções internas (ver Listagem 14) ou transacções externas (ver Listagem 15). As transacções internas são utilizadas quando se tem um conjunto de acções que envolvem acesso a uma ou mais filas MSMQ e onde se pretende que todas as acções sejam concluídas com sucesso ou, caso algo não corra como esperado, as acções sejam interrompidas e as filas envolvidas recuperem o estado que tinham antes do início da transacção.

```
MessageQueue queue;
//...
MessageQueueTransaction transaction = new MessageQueueTransaction();
try{
      transaction.Begin();
      queue.Send(someData, transaction);
      transaction.Commit();
}
catch{ transaction.Abort(); }
```

Listagem 14: Acesso a fila MSMQ envolvendo transacções internas

Porém, existem situações onde se tem mais do que uma entidade transaccional envolvida. Por exemplo, uma mensagem é recolhida de uma fila MSMQ dentro de uma transacção e o seu processamento envolve alterações a uma base de dados. Neste caso, se a transacção for interrompida, pretende-se que a mensagem volte à fila MSMQ e também que a base de dados fique no estado que estava antes do início da transacção. Filas MSMQ e servidores de bases de dados utilizam cada um o seu próprio ambiente transaccional. Nestas situações uma entidade externa conhecida por *Distributed Transaction Coordinator* (DTC) encarrega-se de receber cada uma das transacções (MSMQ e Sql Server) e coordena-las de modo a que ambas sejam concluídas ou abortadas.

```
MessageQueue queue;
//...
using (TransactionScope transaction = new TransactionScope())
{
      queue.Send(someData, MessageQueueTransactionType.Automatic);
      // ... do things such as database access
      transaction.Complete();
}
```

Listagem 15: Acesso a fila MSMQ envolvendo transacções externas

As transacções detectadas pelo MS-DTC (DTC da Microsoft) são representadas pelo objecto `TransactionScope` e do ponto de vista do MSMQ são designadas por transacções externas. As transacções externas envolvem maior utilização de recursos e implicam acessos às filas mais demorados [Mac02], pelo que é preferível o uso de transacções internas sempre que a única entidade transaccional envolvida forem filas MSMQ.

No caso dos módulos *Security* e *Router*, a execução de alguns comandos implica alterações na base de dados. Desta forma, leituras de mensagens recolhidas das filas de entrada destes módulos têm de ser feitas recorrendo a transacções externas. Os métodos da camada de acesso a dados destes módulos, ao serem invocados, verificarão junto do DTC se existe alguma transacção no contexto da chamada e, caso exista, alinham nela de forma automática. De forma a agilizar o desenvolvimento de novos módulos, a classe `MBusModule` (classe base de todos os módulos) contém um campo que indica se os acessos de leitura da fila de entrada são feitos com recurso a transacções internas ou externas.

As leituras de mensagens das filas MSMQ são realizadas de forma assíncrona. Contudo, o método `BeginReceive` não suporta transacções [MSD]. De forma a contornar esta limitação, nos módulos com transacções, os métodos beginReceive foram substituídos por métodos beginPeek (ver Listagem 16). Quando uma mensagem dá entrada na fila, o evento de *peekCompleted* é disparado e no seu processamento realiza-se uma leitura síncrona.

```
private void inputQueue_PeekCompleted(...)
{
  MessageQueue queue;
  using (TransactionScope transaction = new TransactionScope())
  {
  try
  {
   Message msg = queue.Receive(TimeSpan.Zero,
                               MessageQueueTransactionType.Automatic);
  }catch (MessageQueueException) {transaction.Complete();  return; }
   // do some work
   transaction.Complete();
  }
  inputQueue.BeginPeek();
}
```

Listagem 16: Leitura assíncrona de mensagens em contexto transaccional

A desvantagem desta solução é que se um módulo tiver varias instâncias, todas elas serão notificadas assim que chegar uma mensagem à fila mas apenas uma delas é que a irá processar. Para minimizar este problema, introduziu-se um *timeout* no método de leitura síncrona. Desta forma, o tempo de duração da thread que processa o evento *peekComplected* (que pertence à *thread pool*) é bastante reduzido.

A versão 3.0 do MSMQ suporta escritas transaccionais em filas remotas mas não suporta transacções em leituras (mesmo como as da Listagem 16) caso a fila seja remota [MSD10b]. Se o administrador do *bus* optar por instalar cada módulo em máquinas distintas, tem de ter o cuidado de colocar as filas de entrada na mesma máquina onde está o módulo correspondente.

Ao longo deste capítulo utilizou-se o termo persistência para designar a garantia de que as mensagens MSMQ não são perdidas. No entanto, a persistência também se aplica ao estado de cada módulo do *bus*. À medida que comandos vão sendo executados, o estado (entenda-se, o valor dos campos) dos módulos sofre alterações. Idealmente, no caso de uma falha que obrigue o *bus* a ser reiniciado, pretende-se que cada módulo mantenha o estado que tinha antes da ocorrência da falha. Módulos como o *Security* ou o *Router* mantêm em base de dados uma cópia do seu estado. Quando estes módulos arrancam, o seu estado é sincronizado com a informação da base de dados pelo que a sua persistência está assegurada.

Contudo os restantes três módulos (*I/O*, *Transformer* e *Extensibility*) não usam base de dados. O estado do módulo Extensibility não sofre alterações ao longo do funcionamento do *bus*, pelo que a sua persistência não é problema. Relativamente aos módulos *Transformer* e *I/O* houve necessidade de recorrer a outros mecanismos para garantir a persistência de estado.

Persistência no módulo *Transformer*

O único campo do *Transformer* que sofre alterações com o funcionamento do *bus* é a tabela de transformações. Cada vez que se executa um comando para adicionar uma transformação acrescenta-se informação nesta tabela. A solução adoptada para manter a tabela persistente foi seriar o objecto sempre que este é alterado. No carregamento do

módulo, verifica-se se existe algum objecto seriado e em caso afirmativo, constrói-se a tabela com o resultado da deseriação.

Esta solução garante a persistência da tabela caso o módulo seja reiniciado mas não é totalmente fiável. Isto porque o acesso ao sistema de ficheiros (aquando a seriação) não participa em transacções. Quer isto dizer que se existir uma falha no preciso momento em que se faz a seriação, a mensagem MSMQ que representa o comando volta a ser colocada na fila mas o sistema de ficheiros pode ficar num estado inconsistente.

As versões do Windows posteriores ao Windows Vista já suportam acessos a sistemas de ficheiros em contextos transaccionais (através do TxF) [MSD10c] mas este tema não foi objecto de estudo neste trabalho.

Persistência no módulo *I/O*

A manutenção de estado no módulo *I/O* concentra-se nos campos da classe `Dispatcher`. Também aqui, para garantir persistência, recorreu-se à seriação de objectos. Os campos com informações sobre os protocolos de comunicação não são seriados porque a informação neles contida pode ser novamente obtida a partir de ficheiros de configuração. Relativamente aos campos com as referências para os clientes que utilizam o serviço *duplex* também não são seriados. Quer isto dizer que se o *bus* for reiniciado, estes clientes perdem a ligação. A razão para esta decisão está relacionada com protocolos, como por exemplo o AMQP, onde a comunicação é baseada em *sockets*, sendo impossível reiniciar a parte servidora mantendo a cliente.

Em caso de falha, após a recuperação do sistema, pode acontecer que um cliente receba duas vezes a mesma mensagem. Isto porque o processamento das mensagens da fila *dispatchQueue* envolve um bloco transaccional onde são realizadas várias acções. A entrega da mensagem no destino final é apenas uma dessas acções. Se ocorrer uma falha já depois da entrega da mensagem mas antes do bloco transaccional ter terminado, a mensagem volta para a fila *dispatchQueue* tornando a ser processada quando o sistema reiniciar. Para resolver este problema seria necessário que o cliente participasse também na transacção.

O uso de transacções para aceder a filas MSMQ penaliza o desempenho destas acções, em especial no caso de filas remotas e com transacções externas [Jon04]. Trata-se de uma contrapartida inevitável para situações onde a tolerância a falhas seja mais importante do que o desempenho do *bus*. No entanto, na implementação desta funcionalidade existiu o cuidado de permitir ao administrador decidir prescindir da tolerância a falhas, dando prioridade ao desempenho. Desta forma, incluiu-se no ficheiro de configuração MBconfig.xml um atributo que activa ou desactiva o modo de tolerância a falhas. Caso este modo esteja desactivo, o *bus* funciona sem nenhuma das características enumeradas ao longo deste capítulo, nomeadamente as transacções e a seriação/deseriação de objectos.

As técnicas apresentadas para tolerância a falhas recorrem essencialmente a suportes de memória não volátil (seja pelo uso de bases de dados, seja pela utilização directa do sistema de ficheiros). Assim sendo, não estão abrangidas falhas nesses suportes como por exemplo, erros num disco rígido.

Capítulo V

Resultados

Ao longo deste capítulo serão apresentados os resultados obtidos com a solução implementada. O capítulo está dividido em duas fases – uma fase onde se expõem os resultados alcançados e a forma como a sua demonstração foi feita (secção 5.1) e uma fase onde se realizam testes de carga que mostram o comportamento do *bus* quando sujeito a uma grande quantidade de mensagens (secção 5.2).

5.1 Demonstrações

As demonstrações funcionam como prova do alcance de determinada funcionalidade ou característica para a qual a solução implementada se propõe resolver. Na maioria, as demonstrações são compostas por aplicações que funcionam como clientes do *bus*.

As primeiras demonstrações serão as funcionalidades do bus assim como a implementação dos protocolos de comunicação (secção 5.1.1). De seguida será demonstrado o funcionamento do *bus* como ferramenta de integração (secção 5.1.2). Finalmente, será explicado como se testou cada ponto de extensibilidade do *bus* de mensagens (secção 5.1.3).

5.1.1 Funcionalidades e protocolos

A implementação de cada um dos três protocolos de comunicação (AMQP, RestMS e *MB Protocol*) foi testada através de aplicações cliente que enviam e consomem mensagens do *bus*.

O protocolo AMQP foi testado com a aplicação de teste fornecida com o OpenAMQ [Ope] (implementação *opensource* de um *bus* de mensagens desenvolvido pela iMatrix). Esta aplicação destina-se a testar o próprio OpenAMQ e faz uso de uma API em C++ (designada por WireAPI) que disponibiliza as funcionalidades do protocolo AMQP, encarregando-se da construção e envio das tramas pela rede. O cenário de teste consiste em criar um *Exchange* e uma fila no servidor, associando de seguida estes dois elementos através de um *Binding*. O teste termina com o envio e consumo de uma mensagem. Este teste tem especial relevância por apenas incluir código desenvolvido por terceiras entidades, comprovando assim o correcto funcionamento da implementação AMQP deste projecto de acordo com a norma.

Relativamente ao RestMS, não se encontrou nenhuma API desenvolvida por terceiras entidades que fosse suficientemente estável. Desta forma, a alternativa passou por desenvolver uma aplicação em C# que envia pedidos http para um url configurado. Através desta aplicação cria-se no servidor um *feed* e um *pipe* associados através de um *join*. O teste segue os mesmos passos do teste AMQP, sendo enviada uma mensagem para o *bus* e consumida de seguida. Para provar a conformidade da aplicação de teste com a norma, utilizou-se previamente esta aplicação com o Zyre [Zyr] (um *bus* de mensagens RestMS)

O protocolo *MB protocol* foi testado nos mesmos moldes dos anteriores mas com uma aplicação que utiliza *web services* para enviar e receber documentos XML (de acordo com o formato apresentado no Anexo 1). Através desta aplicação é ainda testada cada uma das funcionalidades suportadas por cada módulo. O teste destas funcionalidades consiste em enviar para o *bus* o comando que activa/desactiva a funcionalidade seguido de uma mensagem de dados que force o uso da funcionalidade em teste.

5.1.2 Integração

Logo no início deste documento, o *bus* de mensagens foi apresentado como uma ferramenta que potencia a integração entre ambientes heterogéneos, derivados da existência de diferentes plataformas e linguagens de programação. Para demonstrar esta capacidade de integração foi criado um cenário fictício onde várias aplicações comunicam entre si utilizando o *bus* de mensagens desenvolvido neste projecto.

Com este cenário de teste (representado na Figura 59) pretende-se demonstrar, não só a integração entre diferentes ambientes, como também a interoperabilidade entre os vários protocolos de comunicação.

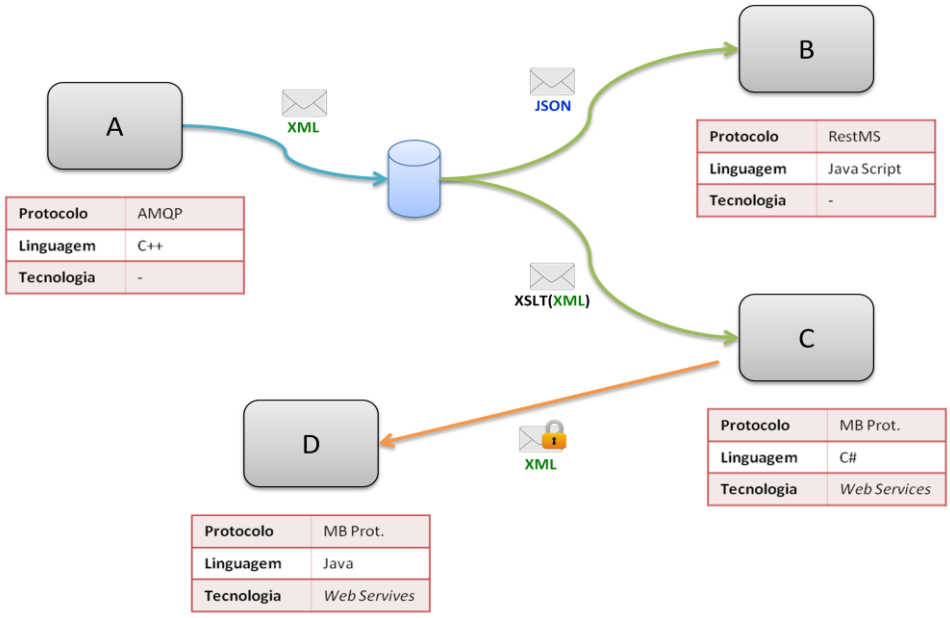


Figura 59: Demonstração da integração de sistemas

Tentando dar algum significado ao cenário de teste, imagine-se que uma empresa num ramo de vendas possui quatro aplicações aqui designadas por A, B, C e D. A aplicação A permite inserir dados de novos produtos, está implementada em C++ e é cliente do *bus* de mensagens através do protocolo AMQP (utiliza a WireAPI). As aplicações B e C estão interessadas em ser notificadas cada vez que um novo produto for introduzido no sistema. Desta forma, as aplicações A, B e C comunicam no modo *publicador-subscritor* onde a aplicação A é produtora de mensagens e as aplicações B e C são consumidoras.

A aplicação B trata-se de uma aplicação *Web* que comunica com o *bus* utilizando o protocolo RestMS e dispõe de funções em Java Script que enviam pedidos http para o *bus*. Por se tratar de uma aplicação Web, é mais fácil processar dados formatados em JSON. No entanto a aplicação A utiliza XML para representar a informação dos novos

produtos. Assim, foi criada uma regra de transformação que indica que o conteúdo de todas as mensagens entregues à aplicação B, que esteja formatado em XML com informação de novos produtos, deve ser convertido para JSON.

A aplicação C, implementada em C#, comunica com o *bus* utilizando *Web services* e o *MB Protocol*. Esta aplicação destina-se a dar suporte interno à empresa e insere o novo produto numa base de dados (não implementada). As aplicações A e C tratam-se de *packaged applications* desenvolvidas por diferentes fornecedores. Como consequência, a aplicação A utiliza dois campos (id + nome) para representar um produto mas a aplicação C utiliza apenas um campo formatado na forma "*id:nome*". Houve então necessidade de aplicar uma regra de transformação XSLT para que as mensagens destinadas à aplicação C sejam entregues no formato esperado.

Ocasionalmente a aplicação C envia relatórios sobre clientes para a aplicação D, implementada em Java e cliente do *bus* através do *MB Protocol*. Estas duas aplicações comunicam no modo *ponto-a-ponto* e é importante garantir que apenas a aplicação C envia dados para a D. Também é importante que o conteúdo das mensagens trocadas entre C e D, não seja revelado caso as mensagens sejam intersectadas durante o transporte. Assim, foram definidas regras de controlo de acesso à aplicação D e configurou-se o *bus* para que todas as mensagens entregues a D circulem de conteúdo cifrado.

Por razões de conveniência, todas as aplicações foram executadas em ambiente Windows. No entanto, aplicações como a A ou D poderiam estar a correr num ambiente diferente como, por exemplo, Linux.

Com a implementação destas quatro aplicações demonstrou-se a utilização do *bus* de mensagens como ferramenta de integração. Note-se que esta integração só foi possível devido ao uso de tecnologias *standard* (como *Web services*) e de protocolos de comunicação que estabelecem o comportamento das aplicações.

5.1.3 Pontos de extensibilidade

O *bus* de mensagens possui quatro pontos onde é possível estender as suas funcionalidades para além daquelas originalmente implementadas:

- Adição de protocolos de comunicação;
- Adição de novos passos na cadeia de processamento de uma mensagem;

- Adição de novas regras de encaminhamento;
- Adição de novos comandos.

A adição de novos protocolos de comunicação foi demonstrada com a implementação do protocolo RestMS que utiliza o *binding* WCF *NetNamedPipeBinding* disponibilizado para o efeito.

Relativamente aos restantes três pontos de extensibilidade, foram implementadas soluções que permitem demonstrar o seu correcto funcionamento.

Adição de novos passos na cadeia de processamento de uma mensagem

Para adicionar um passo extra na cadeia de processamento das mensagens, é necessário desenvolver uma biblioteca que implemente a interface `IExtensibilityPlugin` de acordo com o padrão *plugin* apresentado no capítulo 4.8.

Assim sendo, criou-se um novo projecto .Net onde se adicionou o *assembly MbPluginInterface* resultante da compilação do código do *bus* de mensagens. O exemplo implementado de um novo passo na cadeia de processamento, consiste em fazer um registo de todas as mensagens que passam no *bus*. Desta forma, sempre que o método `applyPlugin` é invocado, adiciona-se no ficheiro MB_log.txt o conteúdo da mensagem presente.

Adição de novas regras de encaminhamento

A implementação de uma biblioteca com uma nova regra de encaminhamento foi feita seguindo os mesmos passos do exemplo anterior mas desta vez implementando a interface `IRoutingPlugin` disponível no mesmo assembly. No corpo do método `RouteMessage` não foi definido nenhum encaminhamento em específico mas apenas escrita uma mensagem na consola indicando que o *plugin* está em execução.

Adição de novos comandos

Com a adição de comandos pretendeu-se simultaneamente demonstrar a possibilidade de adicionar comandos que desempenham acções em módulos específicos

e também demonstrar como se acrescenta uma nova transformação para além das suportadas de raiz.

Para implementar o novo comando foi criada (num projecto dedicado) uma classe que deriva de `ICustomCommand`. Tal como explicado no capítulo 4.10, esta interface contém apenas o método `executeCommand` que recebe a instância de um módulo do *bus* (neste caso, o *Transformer*) e o conteúdo do comando. Na sua implementação, inspecciona-se o conteúdo do comando e adiciona-se uma entrada no dicionário `UserTransformations`. Nesta nova entrada é necessário incluir a instância do objecto que realiza a transformação. O tipo dessa instância deriva de `ITransformation` e também foi definido no projecto de teste.

No cenário de teste, a transformação implementada não altera o conteúdo da mensagem, limitando-se a escrever uma mensagem na consola para que se verifique que o código foi executado.

5.2 Testes de carga

Os testes de carga consistem em submeter o *bus* a um grande número de mensagens em simultâneo e analisar o seu comportamento. Para avaliar este comportamento é necessário encontrar métricas que permitam perceber se o número de mensagens enviadas está a conduzir o sistema a uma situação de sobrecarga (caso em que este deixa de ser capaz de processar as mensagens a um ritmo superior àquele com que são enviadas).

A arquitectura do *bus* de mensagens baseia-se num conjunto de processos a trabalhar de forma independente, obtendo tarefas a partir de uma fila MSMQ e depositando noutra fila o resultado do processamento. Desta forma, a métrica necessária para avaliar o desempenho será medir o número de mensagens existentes na fila de entrada de cada módulo e perceber se as mensagens estão a chegar à fila de entrada a um ritmo superior àquele a que o módulo consegue processá-las.

Os testes de carga efectuados consistiram numa primeira fase em testar cada um dos módulos de forma isolada para conhecer a quantidade de mensagens que cada módulo é capaz de processar por segundo. Numa segunda fase, colocou-se o *bus* de mensagens a funcionar na totalidade, analisando o desempenho de todos os módulos em conjunto.

Para testar os módulos de forma isolada, criou-se uma aplicação que envia determinado número de mensagens para uma fila MSMQ pré-configurada. De seguida lança-se o processo correspondente ao módulo em estudo e através do *monitor de desempenho* do *Windows* monitorizou-se a evolução da quantidade de mensagens em cada fila.

Os testes foram realizados com cliente e servidor a correr na mesma máquina. A máquina em questão é um computador marca *Acer*, modelo *Aspire 5738Z* e cujas principais características de *hardware* são:

- Processador *Pentium Dual-Core* T4300 (2.1 GHz, 800MHz FSB)
- RAM: 4 Gb DDR3

Relativamente ao *software*, o sistema operativo utilizado foi o *Windows Vista* (MSMQ v4.0) com plataforma .NET v3.5.

5.2.1 Módulo de encaminhamento

Para testar o módulo de encaminhamento, foram enviadas dez mil mensagens de dados para a fila de entrada correspondente. O conteúdo das mensagens enviadas foi seleccionado de forma a exigir do módulo um processamento considerado médio, face ao espectável durante o normal funcionamento do *bus*. Desta forma, o módulo de encaminhamento foi previamente configurado para que, do processamento das mensagens de teste, resulte o encaminhamento para três destinos *ponto-a-ponto*.

A Figura 60 mostra a variação do número de mensagens MSMQ ao longo do tempo nas filas de entrada e saída do módulo.

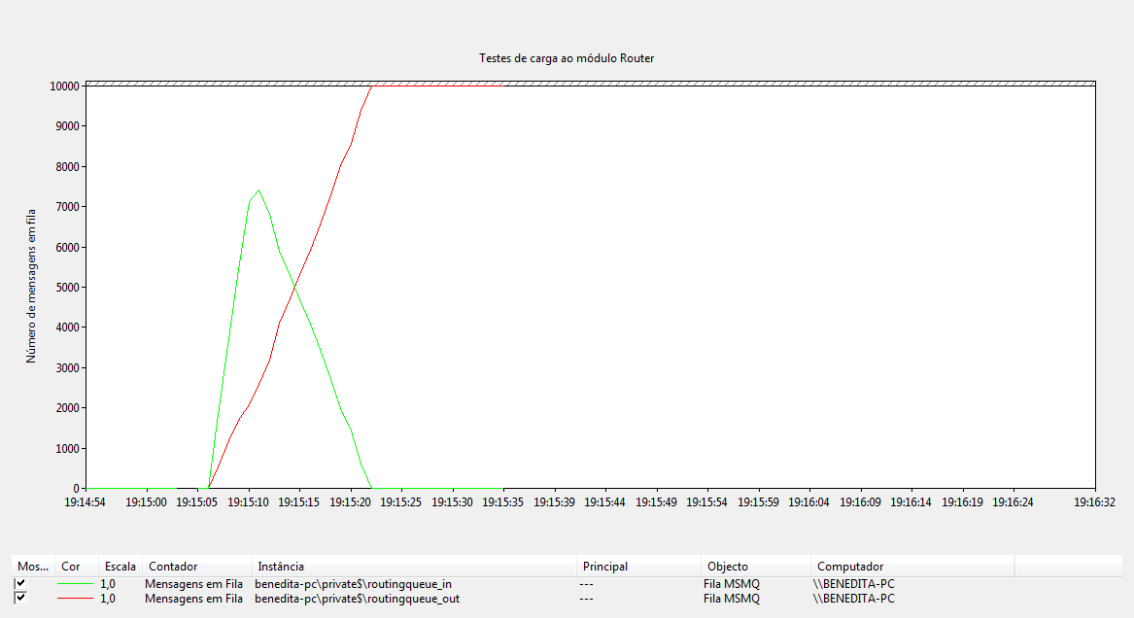


Figura 60: Testes de carga ao módulo *Router*

Neste e nos próximos gráficos de testes de carga a módulos isolados, a quantidade de mensagens na fila de entrada está representada pela curva verde e a quantidade de mensagens na fila de saída está representada pela curva vermelha. O eixo das abcissas corresponde à evolução do tempo representado na forma "hh:mm:ss". O intervalo mínimo de tempo entre recolha de amostras suportado pelo *monitor de desempenho* é um segundo. Durante um segundo a variação do número de mensagens nas filas é por vezes na ordem de milhares, contudo o grande número de mensagens enviadas permite obter resultados conclusivos.

A carga máxima de mensagens suportada pelo módulo é calculada com base na quantidade de mensagens existente na fila de saída (que corresponde ao trabalho terminado). No caso do *Router*, o processamento das dez mil mensagens levou um total de 16 segundos, o que dá uma média de 625 mensagens/s. No entanto, durante o pico máximo de variação de mensagens na fila de saída, registou-se um acréscimo de 915 mensagens num só segundo.

No Anexo 3 está o registo sob a forma de tabela dos valores presentes no gráfico da Figura 60, assim como todos os restantes registos dos testes de carga apresentados a partir deste momento.

5.2.2 Módulo de transformação

Os testes de carga ao módulo de transformação consistiram em enviar dez mil mensagens de dados para a fila de entrada do módulo e registar a variação de mensagens na fila de saída.

O módulo foi pré-configurado para que o processamento de cada mensagem consista em converter o campo de dados XML de 152 caracteres para o formato JSON.

A Figura 61 mostra a variação de ambas as filas.

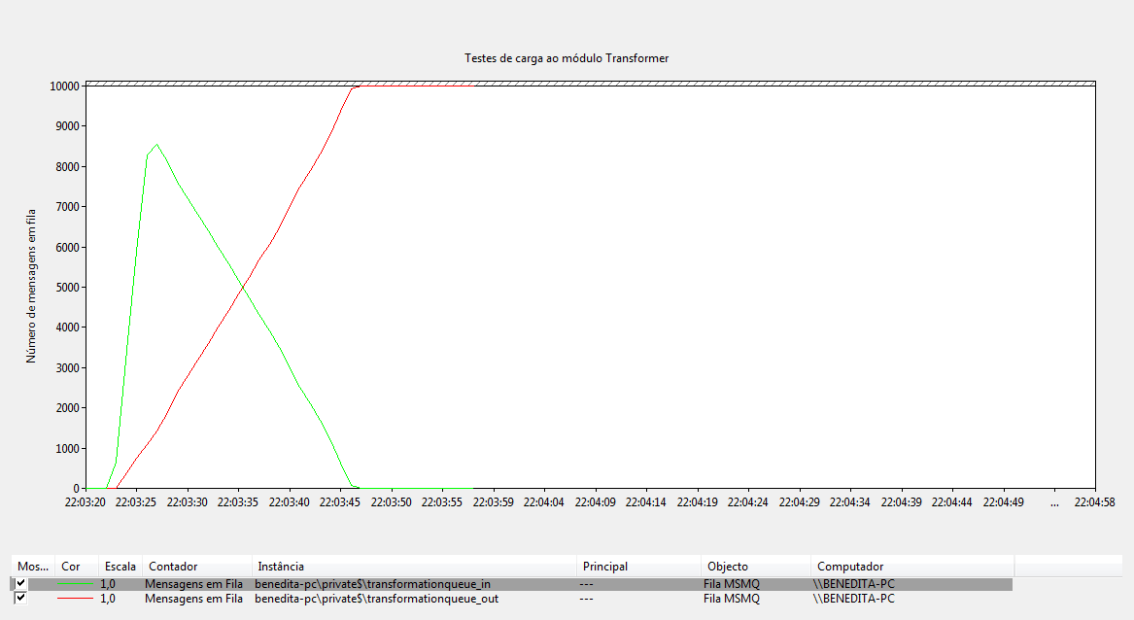


Figura 61: Testes de carga ao módulo *Transformer*

Da observação dos resultados verifica-se que o módulo processou as dez mil mensagens num total de 25 segundos. Em média o módulo mostrou capacidade de processar 431 mensagens/s (neste calculo não foi incluído o valor da última amostra por se verificar que o módulo só não processou mais mensagens nesse segundo por serem as últimas existentes). O pico máximo de mensagens processadas num segundo foi de 549.

5.2.3 Módulo de segurança

Para testar o módulo de segurança, foram enviadas dez mil mensagens de dados para a fila de entrada deste módulo. As mensagens de dados contêm um destino para o qual o módulo de segurança foi previamente configurado como sendo um destino de acesso controlado. Esse destino exige ainda que o *payload* das mensagens entregues esteja cifrado.

A Figura 62 mostra a quantidade de mensagens existentes em cada uma das filas do módulo.

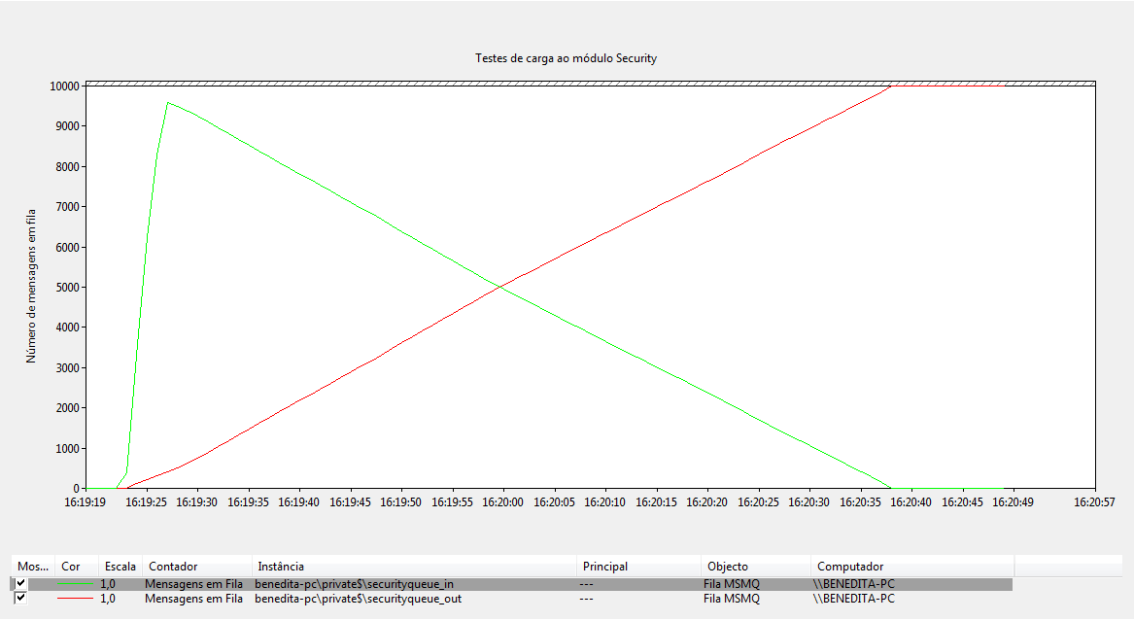


Figura 62: Testes de carga ao módulo *Security*

O módulo necessitou de um minuto e dezassete segundos para processar as dez mil mensagens. O ritmo médio de processamento foi de 133 mensagens/s (não incluindo a primeira e última amostra) tendo um pico de processamento de 150 mensagens/s.

5.2.4 Módulo de entrada/saída

O módulo de entrada/saída foi testado enquanto ponto de entrada das mensagens no *bus*, através do envio de dez mil mensagens de dados para a fila *receivingIncomingQueue*. O conteúdo das mensagens não é relevante, visto que o processamento aplicado é a inserção de um campo *id* e a remoção de todos os cabeçalhos específicos do *bus* de mensagens (caso existam).

A Figura 63 mostra os resultados obtidos para este módulo.

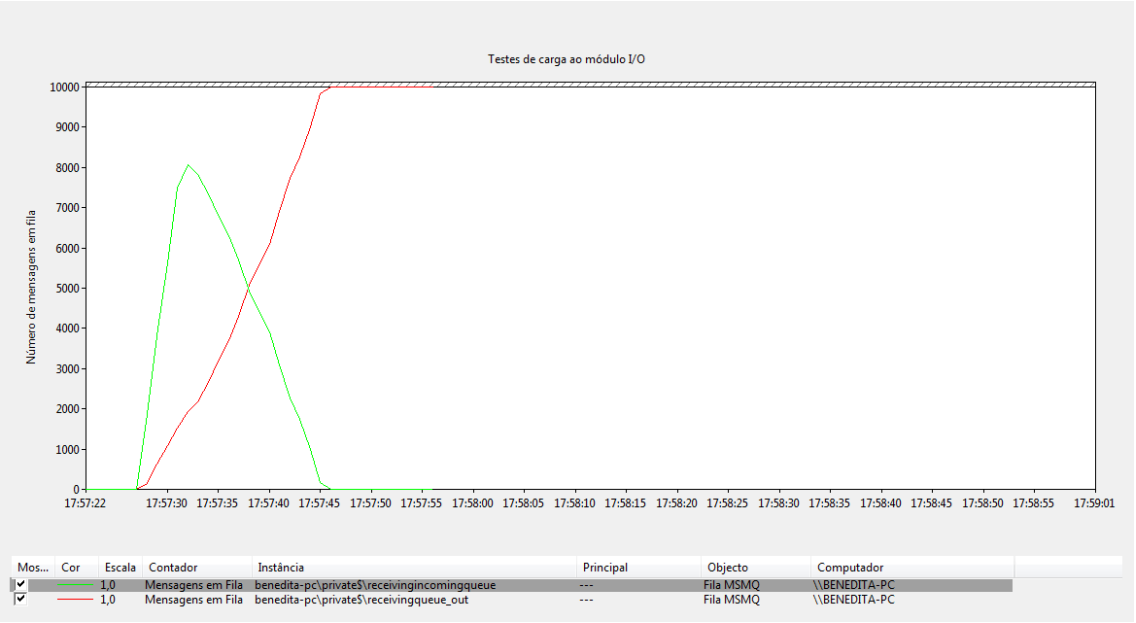


Figura 63: Testes de carga ao módulo I/O

O tempo total para processar as dez mil mensagens foi de 19 segundos. O número de mensagens processadas em cada segundo foi um pouco irregular (como aliás se pode verificar pelo aspecto pouco recto da linha vermelha). Ainda assim, e excluindo a primeira e última amostra, o valor médio de mensagens processadas por segundo é de 569 mensagens/s. O pico de processamento atingido foi de 828 mensagens num segundo.

5.2.5 Módulo de gestão

O teste de carga ao módulo de gestão difere um pouco dos anteriores pelo facto de não existir apenas uma fila de entrada e outra de saída. Do ponto de vista do módulo de gestão, são filas de entrada todas as filas de saída dos restantes módulos e são filas de saída todas as filas de entrada dos restantes módulos.

Durante o teste, o módulo de gestão executou o *workflow* apresentado na Figura 57. Nos primeiros ensaios, verificou-se que a capacidade de processamento do módulo de gestão é inferior à dos módulos até aqui testados. De forma a manter a mesma escala de tempo utilizada nos testes anteriores, reduziu-se o número de mensagens para mil. As mil mensagens foram repartidas por todas as filas que funcionam como filas de entrada para o módulo de gestão. Assim, as mensagens de dados utilizadas no teste foram repartidas da seguinte forma:

- 250 mensagens na fila de saída do módulo *I/O* (que serão encaminhadas para a fila de entrada do *Router*);
- 250 mensagens na fila de saída do *Router* (que serão encaminhadas para a fila de entrada do *Transformer*);
- 250 mensagens na fila de saída do *Transformer* (que serão encaminhadas para a fila de entrada do *Security*);
- 250 mensagens na fila de saída do *Security* (que serão encaminhadas para a fila de entrada do módulo *I/O*).

A Figura 64 mostra a evolução do número de mensagens em todas as filas atrás referidas.

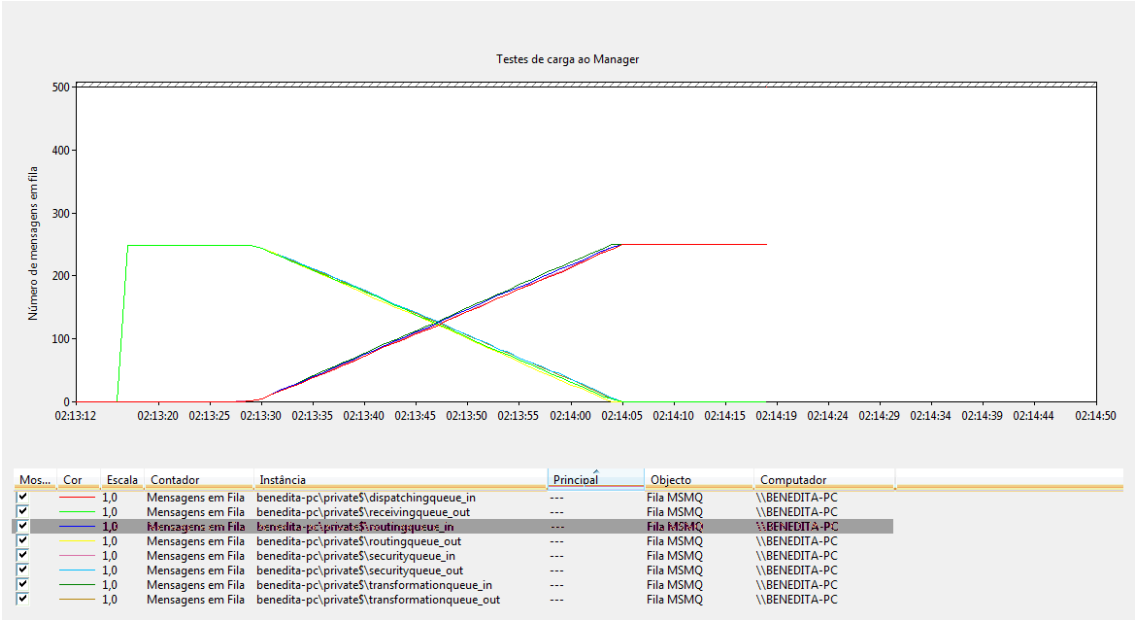


Figura 64: Testes de carga ao módulo *Manager*

O módulo de gestão precisou de 50 segundos para processar as mil mensagens. Para calcular o valor médio de mensagens processadas por segundo, somou-se o número de mensagens de todas as filas de entrada e analisou-se a variação desse valor ao longo do tempo. Desta forma, determinou-se que o módulo de gestão é capaz de processar em média 28 mensagens/s tendo o pico máximo atingido as 37 mensagens/s.

5.2.6 *Bus* de mensagens

Depois de feitos os testes de carga isoladamente a cada um dos módulos, procedeu-se aos testes de carga com a totalidade do *bus* em funcionamento. Para estes testes foram utilizados clientes RestMS que enviaram mensagens destinadas a um

cliente específico (comunicação ponto-a-aponto). As mensagens foram recebidas sem o conteúdo cifrado e sem transformações.

Para o teste foram utilizados 10 clientes (a correr na mesma máquina que o *bus*) onde cada um enviou 50 mensagens consecutivas, resultando num total de 500 mensagens recebidas pelo *bus* no espaço de segundos.

Durante os testes monitorizaram-se todas as filas existentes no *bus*, tendo o resultado obtido o aspecto da Figura 65.

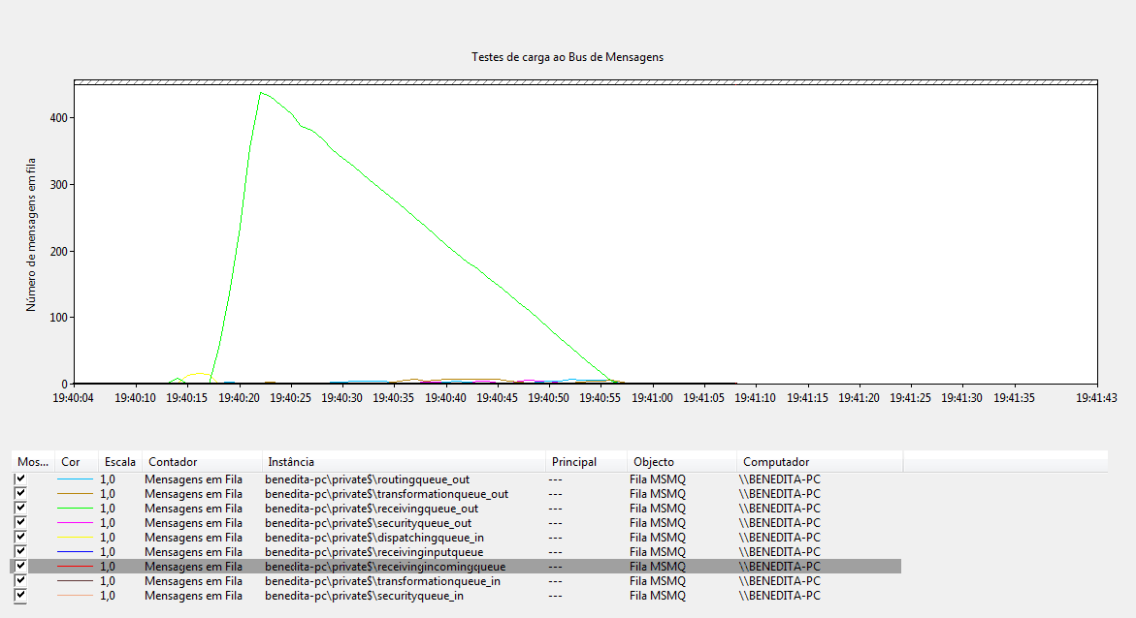


Figura 65: Testes de carga ao *Bus* de Mensagens

Da observação da Figura 65 verifica-se a existência de uma acentuada acumulação de mensagens na fila de saída do módulo I/O, mensagens estas que aguardam o processamento pelo módulo de gestão. Trata-se de um resultado esperado pois, conforme verificado atrás, o módulo de gestão é entre todos, o que processa menor número de mensagens por segundo.

A Figura 66 representa a mesma informação da Figura 65 mas com diferente escala no eixo das ordenadas, permitindo assim observar melhor a evolução de mensagens nas filas além da fila de saída do módulo I/O.

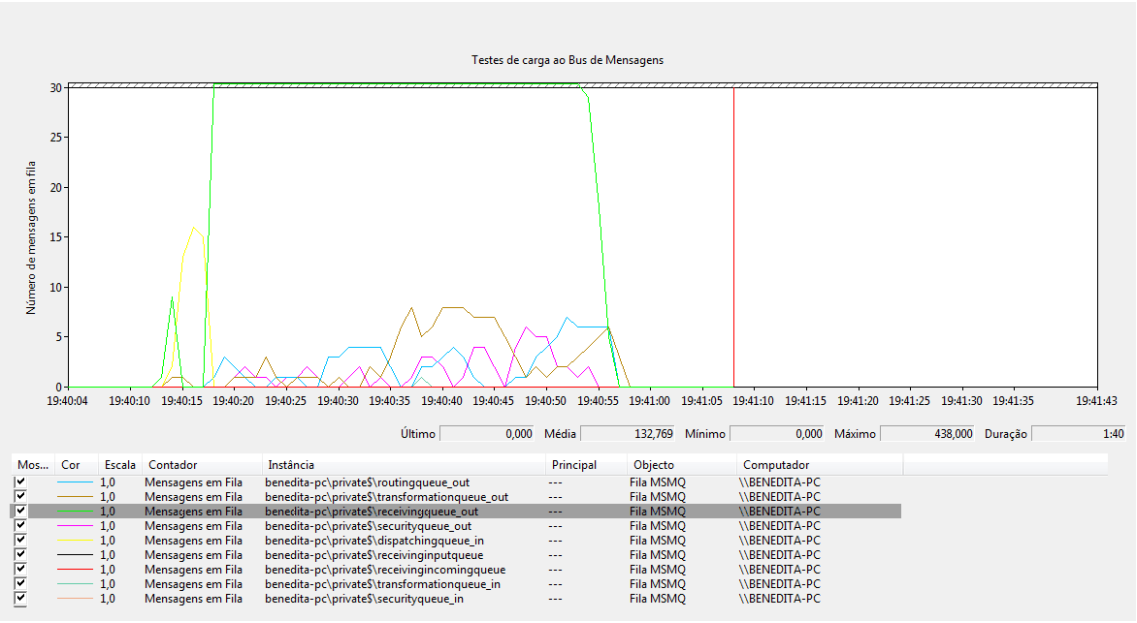


Figura 66: Detalhe do teste de carga ao *Bus* de Mensagens

O *bus* processou as 500 mensagens em 44 segundos. Dado que houve acumulação de mensagens em várias filas, conclui-se que o *bus* funcionou ao máximo da sua capacidade. Assim sendo, afirma-se que, em média, o *bus* de mensagens desenvolvido neste projecto é capaz de processar um máximo de 11,4 mensagens/s.

5.2.7 *Bus* de mensagens em modo tolerância a falhas

Os testes de carga feitos ao *bus* de mensagens na secção 5.2.6, foram repetidos com o *bus* a funcionar em modo de tolerância a falhas. As condições foram as mesmas, onde dez aplicações cliente enviaram 50 mensagens cada.

O registo de mensagens em cada uma das filas do *bus* está representado na Figura 67.

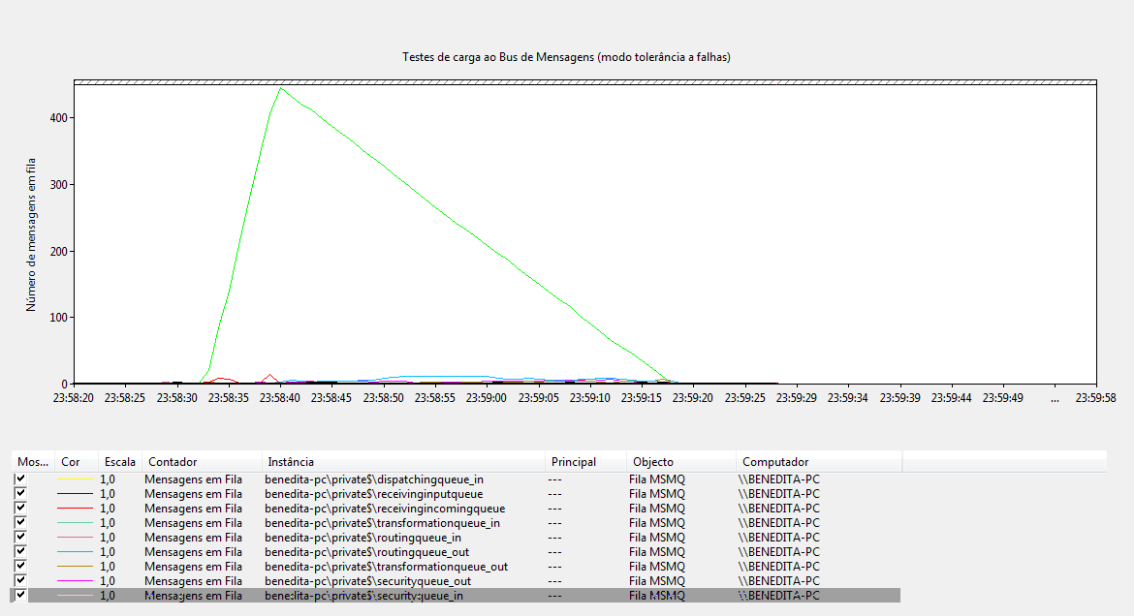


Figura 67: Testes de carga ao *Bus* de Mensagens em modo tolerância a falhas

Comparando com os resultados obtidos na Figura 66, conclui-se que se continua a verificar uma acumulação de mensagens na fila de saída do módulo I/O. O processamento das 500 mensagens demorou 50 segundos (mais 6 que no modo normal de funcionamento). Assim, em média, o *bus* de mensagens consegue processar um máximo de 10 mensagens/s em modo de tolerância a falhas.

Capítulo VI

Discussão e conclusões

Neste capítulo são apresentados os resultados finais obtidos com a realização deste trabalho. O capítulo começa com conclusões gerais obtidas a partir da análise do Capítulo V (secção 6.1). De seguida é apresentada uma análise crítica, não só dos resultados, mas também de vários aspectos tratados ao longo deste trabalho (secção 6.2). O capítulo termina com a indicação do trabalho que poderá ser realizado no seguimento deste projecto (secção 6.3).

6.1 Conclusão

O objectivo deste trabalho foi o desenvolvimento de um *bus* de mensagens. Um *bus* capaz de funcionar como ferramenta de integração entre aplicações a operar em diferentes ambientes/plataformas. Exigiu-se uma arquitectura modular, capaz de suportar as funcionalidades de encaminhamento, segurança e transformação de mensagens. A solução final é ainda extensível nas funcionalidades suportadas, nos algoritmos de encaminhamento, nos comandos interpretados e nos protocolos de comunicação.

No capítulo 5.1 ficou demonstrada a implementação de um *bus* de mensagens com as funcionalidades enumeradas. Demonstrou-se ainda a interoperabilidade entre diferentes protocolos de comunicação.

O resultado final, tal como resumido atrás, cumpre os objectivos propostos.

No que respeita aos resultados dos testes de carga, verificou-se que o *bus* desenvolvido é capaz de processar em média uma carga máxima de 11,4 mensagens/s. Diz-se "em média" pelo facto de cada mensagem poder exigir do *bus* diferentes tipos de processamento, sendo que no teste de carga contemplou-se o caso considerado mais comum, onde as mensagens são apenas encaminhadas para o destino final. Quando repetido o mesmo teste com o modo de tolerância a falhas activo, verificou-se que o valor médio de carga máxima suportada passou de 11,4 para 10 mensagens por segundo. Com base nestes resultados, conclui-se que o modo de tolerância a falhas penaliza o desempenho do *bus* na ordem dos 12,3%, penalização esta proveniente do uso de transacções e da imposição das mensagens MSMQ serem armazenadas em disco rígido.

Individualmente, o módulo que revelou melhor desempenho foi o de encaminhamento com uma capacidade de processamento de 625 mensagens/s. Seguiu-se o módulo de entrada/saída (569 msg/s), transformação (431 msg/s), segurança (133 msg/s) e finalmente o módulo de gestão (37 msg/s).

6.2 Análise Crítica

A principal conclusão retirada dos resultados dos testes de carga é que a solução implementada não é adequada para cenários onde o alto débito de entrega de mensagens seja um requisito. Em vez disso, é indicado para cenários de volume médio de troca de mensagens e onde a integração entre diferentes aplicações seja a principal preocupação. Este facto evidencia os custos de uma arquitectura de integração. Se o principal objectivo fosse a rápida troca de mensagens, então a solução passaria por desenvolver um sistema que apenas realizasse encaminhamento de mensagens e impondo aos clientes um modo de comunicação optimizado para esse sistema. Ao se pretender que a troca de mensagens possibilite a integração de aplicações, introduzem-se outras necessidades como transformar os dados ou suportar vários modos de comunicar com os clientes.

Em termos de desempenho, o módulo que pior se destacou foi o de gestão, ficando muito aquém dos resultados apresentados pelos restantes módulos. Isto faz repensar as vantagens da utilização do *Windows Workflow* como meio de permitir a um indivíduo sem conhecimentos de programação controlar a sequência de processamento

das mensagens. Certamente que este módulo teria obtido melhores resultados se a lógica do processamento de mensagens tivesse sido programada estaticamente. Além disso, existiam outras alternativas ao *workflow* e que também permitiam que um não programador configurasse a sequência de processamento. Por exemplo, um ficheiro de configuração em texto que seria carregado no arranque no módulo de gestão.

Durante a fase de testes de carga, verificou-se que o processador da máquina de teste atingiu a taxa de utilização de 100%. Este facto inviabilizou a repetição dos testes de carga ao *bus* com mais do que uma instância do módulo de gestão a correr; alteração que poderia conduzir a melhores resultados. Ainda na sequência da taxa de utilização do CPU, fica a ideia de que os resultados dos testes de carga obtidos poderão ser melhorados, por exemplo, separando a execução dos vários módulos por diferentes máquinas (assumindo que o tempo de comunicação entre as máquinas não é significativo). A prova de vantagens na separação dos módulos é ainda fundamentada pelo facto do desempenho de cada módulo durante o teste de carga ao *bus* ter sido inferior ao registado durante os testes realizados individualmente a cada módulo.

Relativamente aos protocolos de comunicação estudados, verifica-se que protocolos como o JMS e o AMQP complementam-se. O JMS define apenas a API que as aplicações cliente utilizam para aceder ao *bus*, deixando por definir o formato dos dados. Este pormenor permite que um dado cliente e servidor, ambos suportando JMS, possam não ser compatíveis. Por outro lado, o AMQP (e também o RestMS) especifica o conjunto de comandos e o formato dos dados enviados na rede, garantido desta forma a interoperabilidade entre qualquer par de cliente/servidor que cumpra o protocolo. No entanto o AMQP deixa a definição da API ao critério do fornecedor do *bus*, dificultando assim a substituição do fornecedor do *bus* sem alterar o código das aplicações cliente.

O AMQP prevê uma ligação permanente entre cliente e servidor através de um *socket*. Este facto potencia problemas de escalabilidade por obrigar o servidor a manter em simultâneo todas as ligações. Do ponto de vista de escalabilidade, os protocolos RestMS e *MB Protocol* oferecem vantagens por não serem orientados à ligação.

Os três protocolos aqui implementados apresentam diferenças no modo de entrega das mensagens. No AMQP e *MB Protocol*, a iniciativa da entrega parte do lado do servidor (característica classificada neste trabalho como, *cliente passivo*) ao passo que no RestMS a entrega das mensagens é feita apenas a pedido do cliente (*cliente activo*).

Se por um lado, os clientes passivos têm a vantagem de receber as mensagens no preciso momento em que estas ficam disponíveis, por outro, clientes activos utilizando RestMS têm a vantagem de poder mudar de localização (entenda-se de endereço IP) sem que isso constitua problema para o *bus* de mensagens. No caso do *MB Protocol*, os clientes também podem mudar de localização, mas existe sempre o compromisso com o URL do *web service* para entrega de mensagens fornecido ao *bus*.

6.3 Trabalho futuro

Apesar do *bus* de mensagens implementado ser considerado estável e cumprir os objectos para os quais foi proposto, há ainda alguns pontos que podem ser trabalhados, nomeadamente, o desempenho do módulo de gestão. O próximo passo mais imediato seria substituir a utilização do *workflow* por outro meio de configuração da sequência de processamento e verificar se seriam obtidas vantagens.

Um pouco à medida de cada cenário de integração, a implementação de mais protocolos de comunicação poderá ser uma mais-valia para o *bus* de mensagens.

Ao nível das funcionalidades implementadas, o módulo de segurança talvez seja aquele onde mais trabalho pode ser feito. Por exemplo, a integridade das mensagens (garantia de que o conteúdo não foi alterado por ninguém que não o emissor) poderá ser um aspecto a ter em conta.

Referências

[All07] Allen, Nicholas. **Protocol channels**. *MSDN Blogs - http://blogs.msdn.com/b/drnick/archive/2007/02/21/protocol-channels.aspx*, Feb 2007.

[AMQ08] **Amqp specification 0.9.1**. *http://www.amqp.org/confluence/download/attachments/720900/amqp0-9-1.pdf?version=1&modificationDate=1227526523000*, Nov 08.

[Ber90] Bernstein, Philip. **Transaction processing monitors**. *Communications of the ACM*, 33(11):pp. 75–86, Nov 1990.

[BN09] Bernstein, Philip and Eric Newcomer. ***Principles of Transaction Processing***. Morgan Kaufmann, 2nd edition, 2009.

[Cla03] Clark, Jason. **Calling win32 dlls in c-sharp with p/invoke** - http://msdn.microsoft.com/en-us/magazine/cc164123.aspx. *MSDN Magazine*, Jul 2003.

[Cod10a] CodePlex. ***Distributed Publish/Subscribe (Pub/Sub) Event System***. http://pubsub.codeplex.com/, 2010.

[Cod10b] CodePlex. ***Json.NET***. http://json.codeplex.com/, 2010.

[Cum02] Cummins, Fred A. ***Enterprise Integration: An Architecture For Enterprise Application And Systems Integration***. Wiley, 2002.

[Duf06] Duffy, Joe. ***Professional .NET Framework 2.0***. Wiley, 2006.

[Eea95] Erich, Gamma et al. ***Design Patterns - Elements of Reusable Object-Oriented Software***. Addison-Wesley, 1995.

[Fie00] Fielding, Roy. **Architectural styles and the design of network-based software architectures**. 2000.

[Fow03] Fowler, Martin. ***Patterns of Enterprise Application Architecture***. Addison-Wesley, 2003.

[Gol06] Gollmann, Dieter. ***Computer Security***. Wiley, 2nd edition, 2006.

[Hoh04] Hohpe, Gregor et al. ***Enterprise Integration Patterns***. Addison-Wesley, 2004.

[Hun03] Hunt, Chris and John Loftus. ***Guide to J2EE: Enterprise Java***. Springer, 2003.

[Jon04] Jones, Michael. **Nine tips to enterprise-proof msmq**. *http://www.devx.com/enterprise/Article/22314/0/page/1*, Nov 2004.

[JR09] Rodriguez, Jesus and Pablo Cibraro . **Wcf extensibility guidance - extending the wcf channel model**. *MSDN* - , Oct 2009.

[Jud07] Juday, Jeffrey. **Building wcf channels and bindings**. *http://www.developer.com/net/net/article.php/11087_3676161_2/Building-WCF-Channels-and-Bindings.htm*, May 2007.

[Lin00] Linthicum, David S. ***Enterprise Application Integration***. Addison-Wesley, 2000.

[Lin04] Linthicum, David S. ***Next Generation Application Integration***. Addison-Wesley, 2004.

[Low08] Lowy, Juval. ***Programming WCF Services***. O'Reilly, 2nd edition, 2008.

[Mac02] Mackenzie, Duncan. **Reliable messaging with msmq and .net**. *http://msdn.microsoft.com/en-us/library/ms978430.aspx*, Feb 2002.

[Mah04] Mahmoud, Qusay. ***Middleware for Communications***. Wiley, 2004.

[MSD] MSDN. **Messagequeue.beginreceive method**. *http://msdn.microsoft.com/en-us/library/43h44x53.aspx*.

[MSD10a] MSDN. **Choosing a message exchange pattern**. *http://msdn.microsoft.com/en-us/library/aa751829.aspx*, 2010.

[MSD10b] MSDN. **Reading messages from remote queues**. *http://msdn.microsoft.com/en-us/library/ms699854(VS.85).aspx*, 2010.

[MSD10c] MSDN. **Transactional ntfs (txf)**. *http://msdn.microsoft.com/en-us/library/bb968806(v=VS.85).aspx*, 2010.

[Mye02] Myerson, Judith M. ***Enterprise Systems Integration***. Auerbach, 2 nd edition, 2002.

[O'H07] O'Hara, J. **Toward a commodity enterprise middleware**. *Acm Queue*, vol. 5:pp. 48–55, 2007.

[Ope] **OpenAMQ**. http://win.openamq.org/.

[RB01] Ruh, Francis et al. ***Enterprise Application Integration***. Wiley, 2001.

[Red04] Redkar, Arohi et al. ***Pro MSMQ***. Apress, 2004.

[Res08] Resnick, Steve et al. ***Essential Windows Communication Foundation***. Addison-Wesley, 2008.

[Res09] RestMS. **Restms - a restful messaging service**. *http://www.restms.org/spec:2*, 2009.

[Ros08] Rosen, Mike. **Orchestration or choreography?** http://www.bptrends.com/publicationfiles/04-08-col-bpmandsoa-orchestrationorchoreography-%200804-rosen%20v01%20_mr_final.doc.pdf. *BPTrends*, April 2008.

[Sho05] Shohoud, Yasser. **Meet the wcf channel model - part 1**. *MSDN Blogs - http://blogs.msdn.com/b/yassers/archive/2005/10/12/480175.aspx*, Oct 2005.

[Smi07] Smith, Justin. ***Inside Microsoft Windows Communication Foundation***. Microsoft Press, 2007.

[Spe02] Spencer, Ken. **Using msmq with visual basic .net** – http://msdn.microsoft.com/en-us/magazine/cc188921.aspx. *MSDN Magazine*, Nov - 2002.

[Spe07] **Soap version 1.2 part 1: Messaging framework** – http://www.w3.org/tr/2007/rec-soap12-part1-20070427/. *W3C Recommendation*, 2007.

[Vin06] Vinoski, S. **Advanced message queuing protocol**. *IEEE Internet Computing*, vol. 10(no. 6):pp. 87–89, 2006.

[Zyr] **Zyre**. http://www.zyre.com/.

Nota: Todos os links de internet indicados estão disponíveis à data da impressão deste documento.

Anexo 1

Neste anexo apresenta-se o formato e estrutura de cada comando e mensagem de dados de acordo com o definido no *MB Protocol*.

Mensagem de dados:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>data</mb:MessageType>
    <mb:subject>SalesSubject</mb:subject>
    <mb:rotingTable>RT</mb:rotingTable>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>

    <!-- adicionado pelo receiver -->
    <mb:MessageId>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:MessageId>

    <!--Acrescentados pelo router-->
    <mb:destination>
      <user>amqp://queue_1</user>
      <user>rm://pipe_1</user>
      <user>mb://app1</user>
    </mb:destination>

    <!--Acrescentado pelo Manager numa actividade de plugin personalizada-->
    <mb:ApplyExtensionPlugin>Logger</mb:ApplyExtensionPlugin>


    <!--acrescentado pelo MbExtensility à medida que cada plugin é aplicado-->
    <mb:performedExtensions>
      <extension>Logger</extension>
      <extension>other</extension>
    </mb:performedExtensions>

    <!-- Acrecentado pelo translator.-->
    <mb:performedTransformations>
      <transformation>JsonToXml</transformation>
      <transformation>Xslt</transformation>
    </mb:performedTransformations>

    <!-- adicionado pelo cliente-->
    <mb:security encryptedPayload = "true">
      <sender>client_a</sender>
      <timeStamp>26-06-2010 19:57:44</timeStamp>
    </mb:security>

    <!--assinatura do elem <mb:security> (em base64)-->
    <mb:securitySignature >TWFuIGlzIGRpc3RpbmdlQsIG...</mb:securitySignature>
```

```
    <!-- Marca deixada pelo módulo Security-->
    <mb:securityTag/>

  </s:Header>
  <s:Body>
    <OrderInfo info="order nessage test">
      <FirstName>Jose</FirstName>
      <LastName>Fernandes</LastName>
      <ProductId>32</ProductId>
      <Quantity>3</Quantity>
    </OrderInfo>

    <mb:attachment name="attach1" content-type="binary">
      TAsd7620A ... WFuIGlzIGRpc3RpbmdlaXNoZW=
    </mb:attachment>

  </s:Body>
</s:Envelope>
```

Comando addAllowedSender

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <addAllowedSender timeStamp="26-06-2010 19:57:44" destination="client1"
sender="client2" xmlns="http://www.isel.deetc/MessageBus"/>
    <!-- o elem signature contem a assinatura (em b64) do elem
<addAllowedSender..> -->
    <mb:signature>AHgdyAH675AGd=...</mb:signature>
  </s:Body>
</s:Envelope>
```

Comando addJsonToXmlTransformation

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:addJsonToXmlTransformation
              user="userName"
              regex="GFaft54Ggt6..."
              priority="0"
       />
  </s:Body>
</s:Envelope>
```

Comando addRoute

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:addRoute routingTable = "rt" subjectText="um subj">
      <mb:destination>queue.0</mb:destination>
      <mb:destination>http://localhost:8000/topico2</mb:destination>
    </mb:addRoute>
  </s:Body>
</s:Envelope>
```

Reply de addRoute

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>reply</mb:MessageType>
    <mb:OriginalCommand>addRoute</mb:OriginalCommand>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
  </s:Header>
  <s:Body>
    <mb:state>Success</mb:state>
    <mb:Route table="Routing Table 1" subject="um subj"/>
  </s:Body>
</s:Envelope>
```

Comando addRoutingTable

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:addRoutingTable name="Routing Table 1" type="point-to-point"/>
  </s:Body>
</s:Envelope>
```

Reply de addRoutingTable

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType
xmlns="http://www.isel.deetc/MessageBus">reply</mb:MessageType>
    <mb:OriginalCommand
xmlns="http://www.isel.deetc/MessageBus">addRoutingTable</mb:OriginalCommand>
    <mb:CorrelationID xmlns="http://www.isel.deetc/MessageBus">936DA01F-9ABD-
4d9d-80C7-02AF85C822A8</mb:CorrelationID>
  </s:Header>
  <s:Body>
    <mb:state xmlns="http://www.isel.deetc/MessageBus">Success</mb:state>
    <mb:RoutingTable xmlns="http://www.isel.deetc/MessageBus" name="Routing
Table 1" type="point-to-point"/>
  </s:Body>
</s:Envelope>
```

Comando addUser

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:addUser>
      <userName>Jc</userName>
      <endPoint>http://localhost:8000/serv3</endPoint>
    </mb:addUser>
  </s:Body>
</s:Envelope>
```

Reply de addUser

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>reply</mb:MessageType>
    <mb:OriginalCommand>addUser</mb:OriginalCommand>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>

  </s:Header>
  <s:Body>
    <mb:state>Success</mb:state>
    <mb:user name="queue.0" />
  </s:Body>
</s:Envelope>
```

Comando addXmlToJsonTransformation

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:addXmlToJsonTransformation
              user="userName"
              regex="Af54gh..."
              priority="0"
       />
  </s:Body>
</s:Envelope>
```

Comando addXsltTransformation

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:addXsltTransformation
              user="userName"
              regex="GdstGFFDFA54..."
              xsltData="Hg6FG..."
              priority="2"
       />
  </s:Body>
</s:Envelope>
```

Comando getMessage

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:getMessage messageId="4H4YA01F-9ABD-ft54-80C7-02AF85H46EHA5"/>
  </s:Body>
</s:Envelope>
```

Reply de getMessage

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>reply</mb:MessageType>
    <mb:OriginalCommand >getMessage</mb:OriginalCommand>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:subject>SalesSubject</mb:subject>
    <mb:rotingTable>feed_1</mb:rotingTable>
    <mb:MessageID>4H4YA01F-9ABD-ft54-80C7-02AF85H46EHA5</mb:MessageID>
    <mb:Result>success</mb:Result>
  </s:Header>
  <s:Body>
    <OrderInfo info="order nessage test">
      <FirstName>José</FirstName>
      <LastName>Fernandes</LastName>
      <ProductId>32</ProductId>
      <Quantity>3</Quantity>
    </OrderInfo>
    <attachment name="attach1" content-type="binary">
      TWFuIGlzIGRpc3R...
    </attachment>
  </s:Body>
</s:Envelope>
```

Commando getRoutingTablesInfo

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:getRoutingTablesInfo/>
  </s:Body>
</s:Envelope>
```

Reply de getRoutingTablesInfo

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>reply</mb:MessageType>
    <mb:OriginalCommand>getRoutingTablesInfo</mb:OriginalCommand>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
  </s:Header>
  <s:Body>
    <mb:table name="exchange.0" type="point-to-point"/>
    <mb:table name="feed_0" type="point-to-point"/>
    <mb:table name="feed_1" type="pub-sub"/>
  </s:Body>
</s:Envelope>
```

Comando getUserMessagesList

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:getUserMessagesList userName="pipe_0"/>
  </s:Body>
</s:Envelope>
```

Reply de getUserMessagesList

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>reply</mb:MessageType>
    <mb:OriginalCommand>getUserMessagesList</mb:OriginalCommand>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
  </s:Header>
  <s:Body>
    <mb:message messageId="4H4YA01F-9ABD-ft54-80C7-02AF85H46EHA5"
subject="SalesSubject"/>
    <mb:message messageId="8FGdH74U-5D6j-567d-fg36-FYT74hr7HR73J"
subject="queue_0"/>
  </s:Body>
</s:Envelope>
```

Comando getUsers

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:getUsers/>
  </s:Body>
</s:Envelope>
```

Reply de getUsers

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <mb:MessageType>reply</mb:MessageType>
    <mb:OriginalCommand>getUsers</mb:OriginalCommand>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
  </s:Header>
  <s:Body>
    <mb:user name="queue.0" />
    <mb:user name="pipe_0" />
    <mb:user name="pipe_1" />
  </s:Body>
</s:Envelope>
```

Comando removeAllowedSender

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:removeAllowedSender timeStamp="26-06-2010 19:57:44"
destination="client1" sender="client2"/>
    <mb:signature
xmlns="http://www.isel.deetc/MessageBus">AHgdyAH675AGd=...</mb:signature>
  </s:Body>
</s:Envelope>
```

Comando removeRoute

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:removeRoute routingTable = "rt" subjectText="um subj"
destination="queue.0" />
  </s:Body>
</s:Envelope>
```

Comando removeRoutingTable

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:removeRoutingTable name="Routing Table 1" />
  </s:Body>
</s:Envelope>
```

Comando removeUser

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:removeUser userName="MB://Jc" />
  </s:Body>
</s:Envelope>
```

Comando setEndpointStateCmd

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:setEndpointState endPoint = "endPointStr" state="ready" />
  </s:Body>
</s:Envelope>
```

Comando setSecureCommunication

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:setSecureCommunication timeStamp="26-06-2010 19:57:44"
userName="queue.0" encriptyData="true"/>
    <mb:signature>AHgdyAH675AGd=...</mb:signature>
  </s:Body>
</s:Envelope>
```

Comando updateRoutingTable

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:mb="http://www.isel.deetc/MessageBus">
  <s:Header>
    <Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">MyAction</Acti
on>
    <mb:MessageType>command</mb:MessageType>
    <mb:CorrelationID>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</mb:CorrelationID>
    <mb:ReplyUri>http://localhost:8000/serv3</mb:ReplyUri>
    <mb:ProtocolSchema>amqp</mb:ProtocolSchema>
  </s:Header>
  <s:Body>
    <mb:updateRoutingTable originalName="Routing Table 1" newName="RTable 1"
type="point-to-point"/>
  </s:Body>
</s:Envelope>
```

Anexo 2

Neste anexo apresenta-se em detalhe a hierarquia de classes que suporta a implementação da classe `AmqpFrame`, presente na Figura 39. Essa classe representa uma *frame* de dados AMQP, conforme apresentada na secção "Nível de transporte" do capítulo 2.7.2.2 e é responsável pela execução e codificação[7] de cada *frame*.

A Figura 68 contém o diagrama de classes de todos os objectos utilizados pela classe `AmqpFrame`.

[7] Neste capítulo, entende-se por codificação, a tradução de cada objecto `AmqpFrame` na sequência de bytes correspondente, definidos no nível de transporte do AMQP e que são enviados na rede. Descodificação, será o processo inverso.

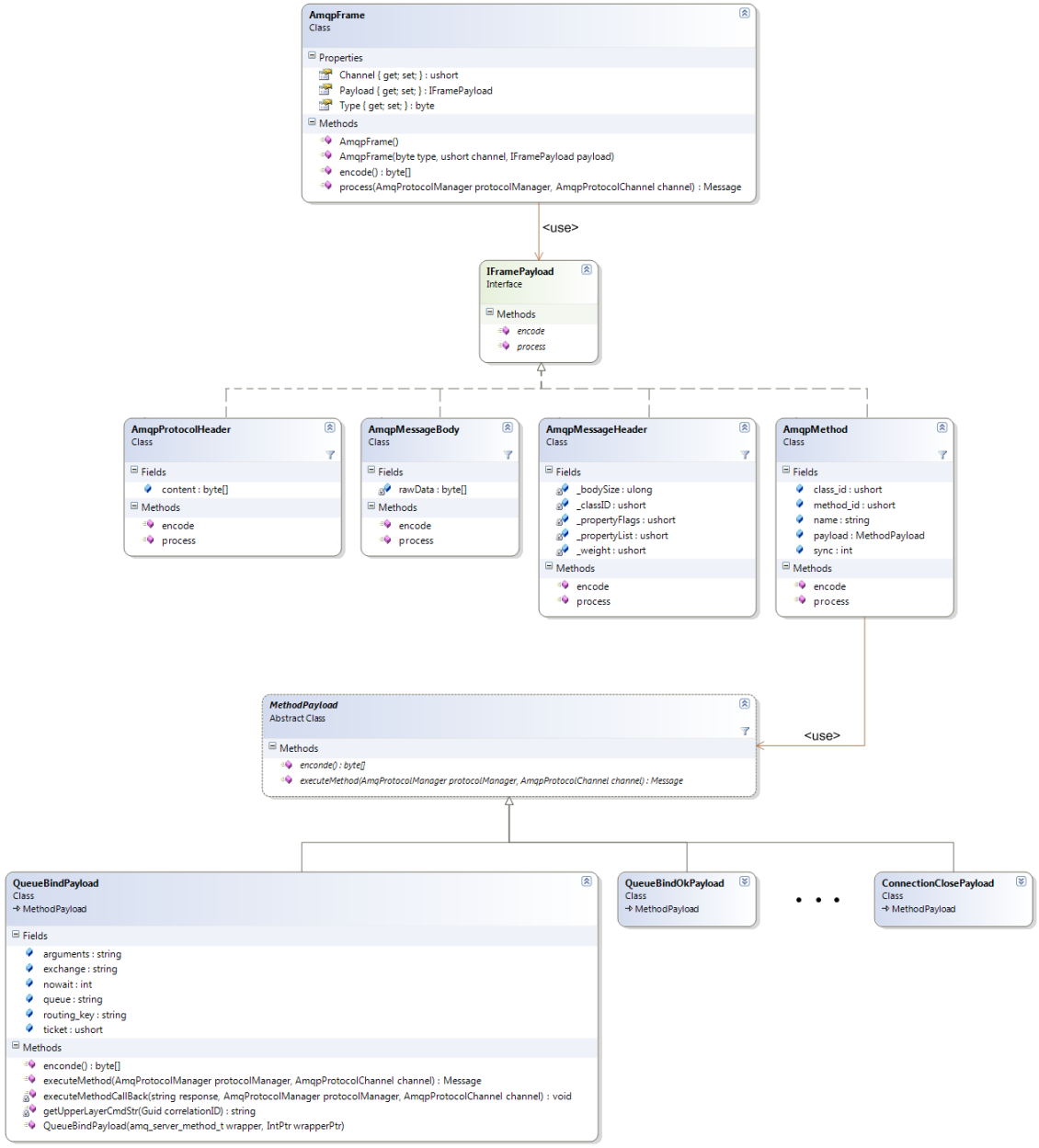


Figura 68: Diagrama de classes da implementação de AmqpFrame

Para cada um dos quatro tipos de *frames* definidos no AMQP, foi criada uma classe que representa o *payload* da respectiva *frame*. Todas essas classes implementam a interface `IFramePayload` de onde herdam o método para codificar esse *payload* no formato de rede e o método para executar no servidor as acções associadas à *frame* em questão.

No caso do *payload* das *frames* do tipo *method* (*frames* que representam os comandos AMQP), existe um campo do tipo `MethodPayload`. O `MethodPayload` é uma classe abstracta que serve de classe base a tantos objectos quantos os comandos AMQP existentes. Na Figura 68, por questões práticas, foram apenas representados três

objectos que derivam de `MethodPayload` mas na implementação são um total de 49, que incluem todos os comandos (e respostas) que circulam no sentido servidor – cliente e cliente – servidor. É nestes objectos que reside a lógica dos passos a realizar no servidor aquando a execução de cada comando. Se no futuro a norma vier a incluir novos tipos de comandos, na implementação do protocolo AMQP bastará criar um novo objecto que deriva de `MethodPayload` e actualizar a biblioteca que codifica e descodifica os comandos (ver em baixo).

Para além do processamento das *frames*, a classe `AmqpFrame` implementa a funcionalidade de codificar cada frame na sequência de *bytes* correspondente. A codificação em AMQP está desenhada por camadas, à semelhança do que se encontra em outros protocolos de rede. Por exemplo, uma mensagem http poderá estar encapsulada no campo de dados de um datagrama TCP, que por sua vez está encapsulado no campo de dados de um pacote IP. Também as *frames* AMQP contêm um campo de dados cujo conteúdo poderá representar, por exemplo, um método, que por sua vez também terá um campo de dados contendo os parâmetros do método, etc. Desta forma, a implementação do método `encode` de `AmqpFrame` obtém apenas os campos relativos a uma *frame* e para preencher o *payload* invoca no campo do tipo `IFramePayload` o método homónimo. A lógica da codificação está assim distribuída por cada objecto que deriva de `IFramePayload`. Excepção para o tipo `AmqpMethod` que delega a codificação dos métodos na classe `AmqpMethodEncoding` (ver Figura 69).

Quando o servidor recebe informação (composta por uma sequência de bytes) do cliente, é necessário converter essa informação num objecto `AmqpFrame`. Este processo, que consiste na descodificação, é implementado pelos objectos cujo diagrama de classes se mostra na Figura 69.

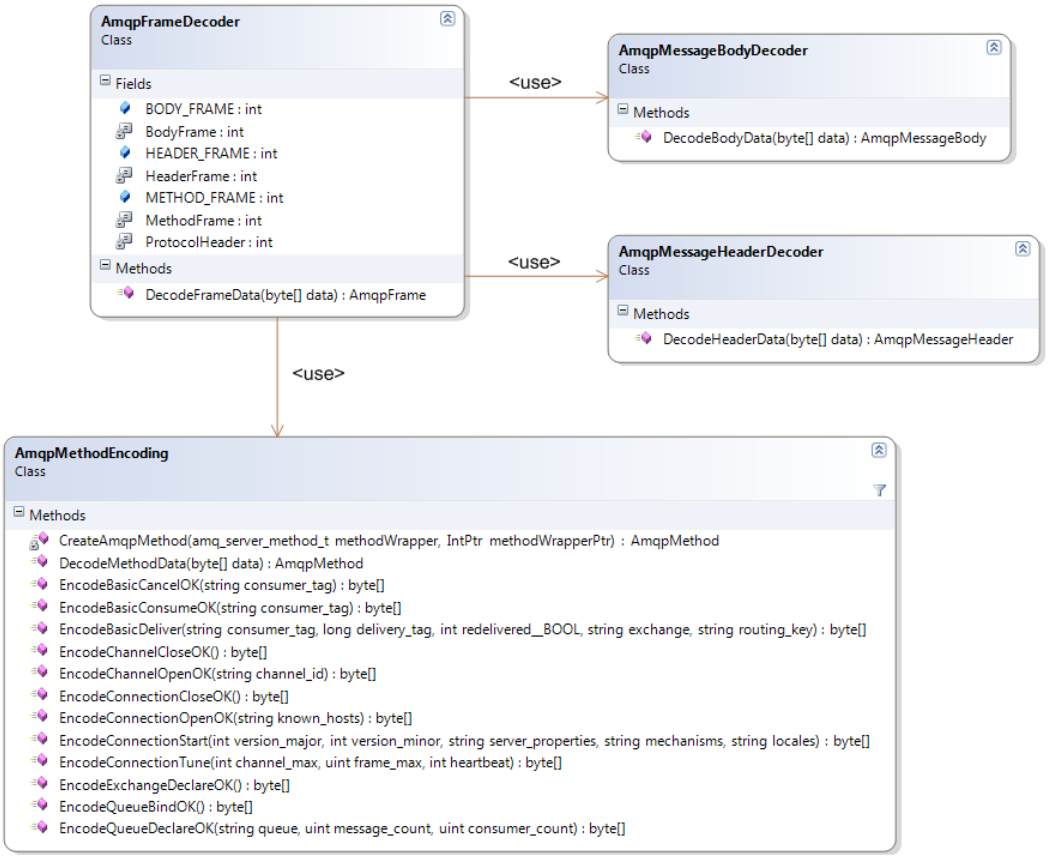


Figura 69: Diagrama de classes dos objectos que suportam a descodificação AMQP

No processo de descodificação, o objecto `AmqpFrameDecoder` assume o papel principal e é aquele que é utilizado pela camada de codificação do *binding* AMQP desenvolvido. Quando o servidor recebe do cliente uma sequência de *bytes*, invoca o método estático `decodeFrameData` da classe `AmqpFrameDecoder`. Este método descodifica os campos relativos a uma *frame* genérica e, depois de verificar o tipo de *frame*, invoca o método *decode…* da classe correspondente (`AmqpMessageBodyEncoder`, `AmqpMessageHeaderEncoder` ou `AmqpMethodEncoding`)

De entre todos os tipos de *frames* as *frames* do tipo *method* são as que apresentam codificação/descodificação mais complexa. Isto deve-se em parte ao grande número de comandos definidos pela norma mas também ao facto de cada comando ter um diferente número (e tipo) de parâmetros. Para facilitar a implementação da classe `AmqpMethodEncoding`, reaproveitou-se código já existente de soluções *open source* de servidores AMQP. Mais especificamente, foi utilizado código do *bus* de mensagens OpenAmq [Ope]. Este *bus* de mensagens está implementado em linguagem C e contém

funções que permitem codificar e descodificar comandos utilizando estruturas definidas em C. A Listagem 17 contém excertos do código de OpenAMQ utilizado.

```
//  função para codificar comandos
ipr_bucket_t* amq_server_method_encode (amq_server_method_t* self);

//  função para descodificar comandos
amq_server_method_t* amq_server_method_decode (
                                          ipr_bucket_t* bucket,
                                          char* strerror);
//  estruturas utilizadas pelas funções:
struct _ipr_bucket_t {
        //  ...
        size_t cur_size;
        size_t max_size;
        byte *data;
};

struct _amq_server_method_t {
      dbyte class_id, method_id
      void *content;
      char *name;
      Bool sync;
      union {
         amq_server_connection_start_t connection_start;
         amq_server_connection_start_ok_t connection_start_ok;
         amq_server_connection_secure_t connection_secure;
         //  ...
      } payload;
};

//  a função amq_server_method_encode recebe como parametro um
amq_server_method_t. Para obter essa estrutura, encontra-se definida
uma função por cada comando existente. Por exemplo, para criar um
comando Connection.Tune:
amq_server_method_t* amq_server_method_new_connection_tune (
                              int channel_max,
                              qbyte frame_max
                              int heartbeat);
```

Listagem 17: Excertos do código fonte do OpenAMQ

A estrutura `ipr_bucket_t` representa um comando codificado e pronto a enviar pela rede (a sequência de *bytes* a transmitir está em `data`). A estrutura `amq_server_method_t` representa igualmente um comando mas sob a forma descodificada, isto é, trata-se de uma estrutura de dados onde o comando já não é visto como uma sequência de *bytes* mas sim como um conjunto de campos utilizados pelo *bus* para processar o comando. `amq_server_method_t` faz uso de um `union` para incluir os parâmetros de cada comando.

Posto isso, o desafio que se coloca é encontrar um mecanismo que permita invocar funções C existentes numa dll *unmanaged* a partir de código C# (*managed*).

Para além da invocação, é ainda necessário lidar com as diferenças que as duas plataformas têm no modo como representam os dados em memória.

O *Platform Invoke* (P/Invoke) surge como uma tecnologia capaz de resolver estes problemas. O primeiro passo para a utilização de P/Invoke é escrever em código *managed* a assinatura das funções de código *unmanaged* (ver Listagem 18). Este será o ponto de "interface" entre as duas plataformas.

```
[DllImport(DLL_PATH, EntryPoint = "EncodeConnectionTune")]
public static extern int EncodeConnectionTune(
                                IntPtr dest,
                                out Int32 dataDim,
                                Int32 channel_max,
                                UInt32 frame_max,
                                Int32 heartbeat
                              );
```

Listagem 18: Assinatura da função *unmanaged* para codificar o comando Connection.Tune

A palavra-chave `extern` indica ao compilador que a implementação do método é externa, isto é, o método não se encontra implementado em .Net. Quando combinado com o atributo `DllImport`, então o compilador saberá que o método está implementado numa dll *unmanaged*.

O segundo aspecto a ter em conta na invocação de código *unmanaged* é a passagem e recepção de parâmetros. As linguagens C# e C têm variáveis primitivas com características semelhantes mas a forma como armazenam essas variáveis em memória pode ser diferente. Assim, passar uma variável de determinado tipo em C# para uma função C, poderá não corresponder apenas a uma simples cópia de memória.

Os tipos cuja disposição em memória é idêntica nas duas plataformas são chamados *tipos blittable*. No caso destes tipos, a passagem de variáveis de uma plataforma para outra é directa. No entanto, nos casos de *tipos não blittable* é necessário indicar explicitamente como deve ser feito o *marshaling* dos parâmetros. O *marshaling* consiste numa transformação de nível binário que permite que variáveis existentes numa plataforma sejam utilizadas em outra. [Duf06] O *marshaling* tanto pode ser uma simples cópia bit-a-bit como implicar uma reestruturação completa dos dados.

A Figura 70 contém uma tabela onde se estabelece a correspondência entre os tipos C utilizados pelo OpenAMQ e os tipos .Net utilizados na invocação dessas funcionalidades. O *marshaling* destes tipos é automático.

***Unmanaged* C/C++**	**CLR (CTS)**
`int`	`Int32`
`unsigned int (qbyte)`	`UInt32`
`unsigned char (byte)`	`Byte`
`unsigned short (dbyte)`	`UInt16`
`__int64`	`Int64`

Figura 70: Mapeamento entre tipos C e C#. Fonte: [Cla03]

Para além dos tipos presentes na Figura 70, as funções do OpenAMQ utilizadas recebem parâmetros cujo *marshaling* tem de ser explicitamente indicado. Exemplos desses parâmetros são as *strings*. Uma *string* em C é formada por um *array* de caracteres terminado por um carácter nulo. Do lado .Net é simplesmente um objecto do tipo `String`. Assim, funções como a da Listagem 18 que recebem *strings* como parâmetros, contêm campos do tipo `String` e esses campos têm o atributo `MarshalAs`, conforme se exemplifica na Listagem 19.

```
[DllImport(DLL_PATH, EntryPoint = "EncodeConnectionOpenOK")]
public static extern int EncodeConnectionOpenOK(
                                    IntPtr dest,
                                    out Int32 dataDim,
                                    [MarshalAs(UnmanagedType.LPStr)]
                                    string known_hosts
                                );
```

Listagem 19: Exemplo de função onde se indica o tipo de *marshaling*.

Ao utilizar o atributo `MarshalAs` com o valor `UnmanagedType.LPStr`, está-se a indicar que do lado C estará um array de caracteres de um byte e terminado por um carácter nulo.

Para além dos parâmetros string, utilizou-se mais um parâmetro onde foi necessário indicar explicitamente o *marshaling*. Esse parâmetro é um *array* de *bytes* com dimensão variável. Neste caso, do lado .Net a função contém um parâmetro do tipo `byte[]` com o atributo `MarshalAs`, desta vez com o valor `UnmanagedType.LPArray`. Neste caso os serviços *interop* fazem a transformação

(*marshal*) dos dados e passam para o lado *unmanaged* um ponteiro para o primeiro elemento do array já em *C-style*.

A passagem de parâmetros torna-se mais complexa quando estes não são tipos primitivos da linguagem (e consequentemente *não blittable*). Exemplo disso é o caso da estrutura `amq_server_method_t` (ver Listagem 17), criada pela função que descodifica comandos e que é necessário passar do lado *unmanaged* para o *managed*. Para trazer este tipo para o lado *managed* define-se desse lado uma estrutura de dados que representa o objecto *unmanaged*. Quando todos os campos existentes na estrutura são de tipos *blittable*, o *marshal* dos parâmetros é realizado de forma automática. No entanto, em situações especiais, como é o caso, o marshal destas estruturas terá de ser manual. O que torna o caso da estrutura `amq_server_method_t` uma situação especial é o facto de esta conter campos que são ponteiros de memória e a existência de blocos *union*. Quando em C uma estrutura contém um bloco *union*, significa que naquela posição de memória poderá existir qualquer um dos tipos presentes dentro do bloco. Esta situação dificulta a definição da estrutura *managed* pelo facto de, na mesma estrutura poderem existir campos de diferentes tipos. Como só pode ser definida uma estrutura para mapear o lado *unmanaged*, então a solução é utilizar o atributo `FieldOffset` (ver Listagem 20).

```
[StructLayout(LayoutKind.Explicit, Size = 812)]
class amq_server_method_t
{
    //... (campos omitidos) ...
    [FieldOffset(10)]
    public UInt16 class_id;
    [FieldOffset(12)]
    public UInt16 method_id;
    [FieldOffset(14)]
    public IntPtr content;
     [FieldOffset(20)]
    [MarshalAs(UnmanagedType.LPStr)]
    public string name;
    //... (campos omitidos) ...
    //offset do UNION
    [FieldOffset(32)]
    public amq_server_connection_start_t connection_start;
    [FieldOffset(32)]
    public amq_server_connection_start_ok_t connection_start_ok;
}
```

Listagem 20: Estrutura do tipo managed que representa um comando

A Listagem 20 corresponde à estrutura *managed* que mapeia a estrutura `amq_server_method_t` apresentada na Listagem 17. A presença do atributo `StructLayout` com a opção `LayoutKind.Explicit` informa os serviços interop que, durante o *marshaling* desta estrutura, o *offset* de cada campo do lado *unmanaged* é indicado de forma explícita. Depois cada campo tem o atributo `FieldOffset` onde consta o número de bytes existentes na estrutura *unmanaged* entre a posição de memória onde começa o objecto e a posição de memória onde está o campo. O que se fez foi colocar o mesmo offset para todos os campos do bloco *union*. Ao se utilizar a opção `LayoutKind.Explicit` é necessário conhecer a forma como os campos são organizados dentro de uma estrutura em C/C++. Mais especificamente, para além de conhecer o espaço em memória ocupado por cada variável, é necessário ter em atenção duas regras:

- Campos contínuos do mesmo tipo têm *packing* contínuo;
- Campos de tipos diferentes do campo anterior estão alinhados em endereços múltiplos da dimensão do campo.

Para cada um dos tipos presentes no bloco *union* foi criada uma estrutura que mapeia o seu conteúdo, estrutura essa que corresponde aos parâmetros de cada comando. As estruturas utilizadas para mapear dados *unmanaged* pertencem ao espaço de nomes `AmqpServer.WireLevel.WrapperTypes`.

Relativamente ao problema das estruturas *unmanaged* que contêm ponteiros, não há no P/Invoke nenhum suporte para esta situação, pelo que a solução foi trazer para o lado *managed* o valor do ponteiro (através de campos do tipo `IntPtr`) e mais tarde, através dos mecanismos apropriados, fazer a leitura dessa zona de memória para uma estrutura apropriada.

As funcionalidades pretendidas do código do OpenAMQ são a codificação e descodificação de comandos. O ideal seria que estas funcionalidades fossem utilizadas invocando apenas uma função. Contudo, ao analisar a Listagem 17 verifica-se que, por exemplo, para codificar um comando *connection.Tune* é necessário primeiro executar a função `amq_server_method_new_connection_tune` para obter um `amq_server_method_t*`. Depois chamar a função `amq_server_method_encode` passando o resultado obtido na primeira chamada. Finalmente, extrair de

`ipr_bucket_t*` os *bytes* correspondentes ao comando codificado e retorna-los. De forma a agilizar este e outros processos e também a facilitar a utilização do P/Invoke, escreveu-se em C uma biblioteca que realiza estes passos e exporta apenas as funções para codificar cada um dos comandos e uma função para descodificar comandos. Esta biblioteca, a que se chamou Amqp_Wire_Dll, é a biblioteca à qual o *bus* de mensagens acede via P/Invoke.

A solução final da codificação/descodificação de comandos AMQP está assim implementada por camadas, como se ilustra na Figura 71.

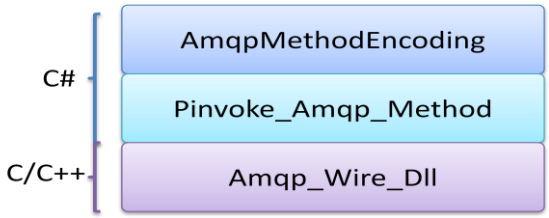


Figura 71: Arquitectura da codificação/descodificação de comandos AMQP

A camada Amqp_Wire_Dll corresponde à biblioteca C/C++ que exporta as funcionalidades do OpenAMQ. A camada PInvoke_Amqp_Method contém a declaração de cada método do lado *managed* (o conteúdo da Listagem 18 e Listagem 19 é um excerto desta camada). Finalmente a camada AmqpMethodEncoding corresponde à classe com o mesmo nome presente na Figura 69.

A lógica da utilização do P/Invoke está contida na classe `AmqpMethodEncoding`. Dado que as funcionalidades pretendidas envolvem transferência de dados entre memória *unmanaged* memória *managed*, existem ainda alguns detalhes a ter em conta. O CLR utiliza zonas distintas de memória caso esta seja *managed* ou *unmanaged* de tal forma que memória *unmanaged* não pode ser acedida a partir de código *managed*. A Listagem 21 mostra o método de `AmqpMethodEncoding` utilizado para codificar um comando *Connection.Tune* (sendo semelhantes os restantes métodos de codificação).

```
public static byte[] EncodeConnectionTune(int channel_max, UInt32 frame_max,
int heartbeat){
        IntPtr ptr = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(Byte)) * 256);
        int dim = 0;
        PInvoke_AMQP_Method.EncodeConnectionTune(ptr, out dim, channel_max,
frame_max, heartbeat);
        byte[] encodedBytes = new byte[dim];
        Marshal.Copy(ptr, encodedBytes, 0, dim);
        Marshal.FreeHGlobal(ptr);
        return encodedBytes;
}
```

Listagem 21: Método EncodeConnectionTune em AmqpMethodEncoding

O método `Marshal.AllocHGlobal` aloca em memória *unmanaged* a quantidade pedida. O método `EncodeConnectionTune` corresponde à chamada da função na biblioteca C. Função essa que escreve o comando codificado no endereço recebido no primeiro parâmetro. Como esse endereço corresponde à memória *unmanaged* alocada no inicio do método, não haverá violação de memória *managed*. Através de `Marshal.Copy` copia-se o resultado da memória *unmanaged* para *managed* libertando-se finalmente a memória *unmanaged* através de `Marshal.FreeHGlobal`. Note-se que toda a memória alocada durante a execução da função de Amqp_Wire_Dll é libertada antes do retorno, não restando assim memória por libertar.

O mesmo procedimento foi aplicado ao método `DecodeMethodData` de `AmqpMethodEncoding` cujo conteúdo se mostra na Listagem 22.

```
public static AmqpMethod DecodeMethodData(byte[] data)
{
    IntPtr ptr =
Marshal.AllocHGlobal(Marshal.SizeOf(typeof(amq_server_method_t)));
    PInvoke_AMQP_Method.DecodeMethodData(ptr, data, data.Length);
    amq_server_method_t methodWrapper =
(amq_server_method_t)Marshal.PtrToStructure(ptr, typeof(amq_server_method_t));
    AmqpMethod method = CreateAmqpMethod(methodWrapper, ptr);
    Marshal.FreeHGlobal(ptr);
    return method;
}
```

Listagem 22: Método DecodeMethodDatae em AmqpMethodEncoding

Relativamente ao código da Listagem 21, a diferença é que o resultado que agora se pretende é uma estrutura que representa o comando. Após a invocação do código nativo através de `PInvoke_AMQP_Method.DecodeMethodData`, essa estrutura é copiada para o endereço indicado no primeiro parâmetro. A estrutura só pode ser utilizada em C# após a execução de `Marshal.PtrToStructure`. Este método converte numa estrutura C# a zona de memória com início no endereço indicado. O *marshaling* dessa estrutura é realizado com base nos atributos utilizados na definição de

`amq_server_method_t` (ver Listagem 20). Durante este processo é necessário conhecer o tamanho de `amq_server_method_t`. No entanto, como esta estrutura contém vários campos com *offsets* sobrepostos, o *runtime* não tem forma de saber essa dimensão. Nestes casos é utilizado o valor de `Size` presente no atributo de `StructLayout` (Listagem 20). A este valor corresponde o maior tamanho possível de um comando tendo em conta a dimensão dos parâmetros presentes no *payload* do maior dos comandos.

Visto que a estrutura *managed* `amq_server_method_t` contém campos `IntPtr` cujo valor só é lido após a execução de `PInvoke_AMQP_Method.DecodeMethodData`, o código nativo não pode libertar toda a memória alocada antes do retorno (como acontecia nos métodos de codificação). Esta memória é libertada através de uma nova chamada a código nativo que é realizada no decorrer do método auxiliar `CreateAmqpMethod`.

Anexo 3

O Anexo 3 contém as tabelas com os registos feitos durante os testes de carga. São apresentados primeiro os resultados relativos aos testes aos módulos de forma isolada e no final estão os resultados dos testes com todos os módulos em conjunto.

Teste de carga ao módulo *Router*

tempo [s]:	nº msg (in)	Variação (in)	nº msg (out)	Variação (out)
0	0		0	
1	1834	1834	521	521
2	3720	1886	1181	660
3	5439	1719	1689	508
4	7116	1677	2078	389
5	7409	293	2590	512
6	6819	-590	3180	590
7	5904	-915	4095	915
8	5287	-617	4712	617
9	4683	-604	5316	604
10	4074	-609	5925	609
11	3461	-613	6539	614
12	2712	-749	7288	749
13	1990	-722	8009	721
14	1435	-555	8564	555
15	605	-830	9394	830
16	0	-605	10000	606

Teste de carga ao módulo *Transformer*

tempo [s]:	nº msg (in)	Variação (in)	nº msg (out)	Variação (out)
0	0		0	
1	647	647	0	0
2	3429	2782	393	393
3	5896	2467	748	355
4	8279	2383	1094	346
5	8562	283	1437	343

6	8122	-440	1877	440
7	7598	-524	2401	524
8	7188	-410	2812	411
9	6810	-378	3190	378
10	6399	-411	3601	411
11	5987	-412	4012	411
12	5578	-409	4421	409
13	5161	-417	4838	417
14	4746	-415	5253	415
15	4335	-411	5665	412
16	3925	-410	6074	409
17	3514	-411	6485	411
18	2999	-515	7000	515
19	2500	-499	7499	499
20	2091	-409	7908	409
21	1679	-412	8321	413
22	1174	-505	8825	504
23	626	-548	9373	548
24	77	-549	9922	549
25	0	-77	10000	78

Teste de carga ao módulo *Security*

tempo [s]:	nº msg (in)	Variação (in)	nº msg (out)	Variação (out)
1	0		0	
2	389	389	1	1
3	3312	2923	109	108
4	6240	2928	215	106
5	8291	2051	306	91
6	9593	1302	406	100
7	9499	-94	500	94
8	9374	-125	625	125
9	9244	-130	755	130
10	9103	-141	896	141
11	8957	-146	1042	146
12	8813	-144	1186	144
13	8668	-145	1331	145
14	8521	-147	1478	147
15	8378	-143	1621	143
16	8241	-137	1758	137
17	8095	-146	1904	146
18	7946	-149	2053	149

19	7808	-138	2191	138
20	7676	-132	2323	132
21	7544	-132	2455	132
22	7397	-147	2602	147
23	7249	-148	2750	148
24	7102	-147	2897	147
25	6955	-147	3044	147
26	6828	-127	3171	127
27	6680	-148	3319	148
28	6530	-150	3469	150
29	6380	-150	3619	150
30	6233	-147	3766	147
31	6085	-148	3914	148
32	5937	-148	4062	148
33	5796	-141	4203	141
34	5656	-140	4343	140
35	5509	-147	4490	147
36	5369	-140	4630	140
37	5224	-145	4775	145
38	5080	-144	4919	144
39	4947	-133	5052	133
40	4817	-130	5182	130
41	4685	-132	5314	132
42	4554	-131	5445	131
43	4423	-131	5576	131
44	4292	-131	5707	131
45	4161	-131	5838	131
46	4030	-131	5969	131
47	3904	-126	6095	126
48	3774	-130	6225	130
49	3642	-132	6357	132
50	3514	-128	6485	128
51	3388	-126	6611	126
52	3260	-128	6740	129
53	3132	-128	6868	128
54	3003	-129	6996	128
55	2875	-128	7124	128
56	2746	-129	7254	130
57	2623	-123	7376	122
58	2495	-128	7504	128
59	2365	-130	7634	130
60	2236	-129	7763	129
61	2108	-128	7891	128

62	1979	-129	8020	129
63	1843	-136	8156	136
64	1698	-145	8301	145
65	1563	-135	8436	135
66	1433	-130	8566	130
67	1304	-129	8695	129
68	1180	-124	8819	124
69	1051	-129	8948	129
70	923	-128	9076	128
71	794	-129	9205	129
72	666	-128	9333	128
73	538	-128	9461	128
74	409	-129	9590	129
75	280	-129	9719	129
76	148	-132	9851	132
77	0	-148	9999	148
78	0	0	10000	1

Teste de carga ao módulo I/O

tempo [s]:	nº msg (in)	Variação (in)	nº msg (out)	Variação (out)
0	0		0	
1	1852	1852	140	140
2	3806	1954	619	479
3	5623	1817	1085	466
4	7485	1862	1533	448
5	8075	590	1924	391
6	7825	-250	2174	250
7	7344	-481	2655	481
8	6815	-529	3184	529
9	6314	-501	3686	502
10	5704	-610	4295	609
11	4932	-772	5067	772
12	4417	-515	5582	515
13	3908	-509	6091	509
14	3093	-815	6906	815
15	2265	-828	7734	828
16	1731	-534	8268	534
17	987	-744	9012	744
18	177	-810	9822	810
19	0	-177	10000	178

Testes de carga ao módulo *Manager*

tempo [s]:	I/O out	Router In	Router Out	Transf In	Transf Out	Security In	Security Out	Disp In
1	0	0	0	0	0	0	0	0
2	249	0	249	0	249	0	249	0
3	249	0	249	0	249	0	249	0
4	249	0	249	0	249	0	249	0
5	249	0	249	0	249	0	249	0
6	249	0	249	0	249	0	249	0
7	249	0	249	0	249	0	249	0
8	249	0	249	0	249	0	249	0
9	249	0	249	0	249	0	249	0
10	249	0	249	0	249	0	249	0
11	249	0	249	0	249	0	249	0
12	249	0	249	0	249	0	249	0
13	249	0	249	1	249	1	249	0
14	249	1	249	1	249	1	249	1
15	244	5	244	5	244	5	244	5
16	236	13	237	13	237	12	237	12
17	230	20	230	19	231	19	231	18
18	222	27	222	27	223	26	225	25
19	216	33	215	34	217	32	218	31
20	209	40	208	41	210	39	212	38
21	202	47	200	49	203	46	205	44
22	195	54	194	55	197	52	198	51
23	188	61	186	63	190	60	191	58
24	182	68	179	70	183	66	184	65
25	175	75	172	77	176	73	177	72
26	168	82	164	85	169	81	169	80
27	160	89	157	92	161	89	162	88
28	153	97	150	99	154	95	155	94
29	146	103	144	106	147	102	148	101
30	138	111	137	113	141	109	142	108
31	131	118	128	120	133	116	135	114
32	124	125	122	127	126	124	128	121
33	117	133	115	134	119	130	121	129
34	109	140	108	142	112	137	113	136
35	102	147	100	149	105	144	106	143
36	94	155	93	157	98	151	99	150
37	87	162	85	164	91	158	92	157
38	81	169	78	171	84	165	84	165
39	74	176	71	178	77	172	78	172
40	67	182	63	186	70	169	70	179

41	60	189	56	193	64	186	64	185
42	52	197	48	201	56	193	57	193
43	45	204	42	208	49	200	50	199
44	38	211	34	215	42	207	44	206
45	31	218	27	222	35	215	36	213
46	25	224	21	229	28	222	29	220
47	17	232	14	236	20	229	21	228
48	10	239	6	243	12	237	14	236
49	4	246	0	250	4	245	6	243
50	0	250	0	250	0	250	0	250

Teste de carga ao *Bus* de Mensagens

tempo [s]:	I/O In	I/O out	Router In	Router Out	Transf In	Transf Out	Security In	Security Out	Disp In	I/O Incoming
1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0
3	0	9	0	0	0	1	0	1	2	0
4	0	0	0	0	0	1	0	1	13	0
5	0	0	0	0	0	0	0	0	16	0
6	0	0	0	0	0	0	0	0	15	0
7	0	54	0	1	0	0	0	0	0	1
8	0	135	0	3	1	0	0	0	0	0
9	0	236	0	2	0	1	1	1	0	0
10	0	356	1	1	0	1	0	2	0	0
11	0	438	0	0	0	1	0	1	0	0
12	0	432	0	0	0	3	0	1	0	0
13	0	419	0	1	0	1	0	0	0	0
14	0	406	0	1	0	0	0	1	0	0
15	0	388	0	1	0	1	1	1	0	0
16	0	381	0	0	0	1	0	2	0	0
17	0	368	0	0	0	1	0	1	0	0
18	0	353	0	3	0	0	0	0	0	0
19	0	340	0	3	0	1	0	0	0	0
20	0	327	0	4	0	0	0	1	0	0
21	0	315	0	4	0	0	0	2	0	0
22	0	301	0	4	0	2	0	0	0	0
23	0	288	0	4	0	1	0	1	0	0
24	0	276	0	2	0	3	0	0	0	0
25	0	263	0	0	0	6	0	0	0	1
26	0	249	0	0	0	8	0	1	0	0
27	0	237	0	2	0	5	1	3	0	0

28	0	223	0	2	1	6	0	3	0	0
29	0	208	0	3	0	8	0	2	0	0
30	0	196	0	4	0	8	0	0	0	0
31	0	184	0	3	0	8	0	1	0	0
32	0	173	0	1	0	7	0	4	0	0
33	0	160	0	0	0	7	0	4	0	0
34	0	148	0	0	0	7	0	2	0	0
35	0	136	0	0	0	5	0	0	0	0
36	0	123	0	1	0	3	0	4	0	0
37	0	110	0	1	0	1	0	6	0	0
38	0	96	0	3	0	2	0	5	0	0
39	0	83	0	4	0	1	0	5	0	0
40	0	69	0	5	0	2	0	2	0	0
41	0	55	0	7	0	2	0	2	0	0
42	0	42	0	6	0	3	0	1	0	0
43	0	29	0	6	0	4	0	2	0	0
44	0	18	0	6	0	5	0	0	0	0
45	0	5	0	6	0	6	0	0	0	0
46	0	0	0	0	0	3	0	0	0	0
47	0	0	0	0	0	0	0	0	0	0

Teste de carga ao *Bus* de Mensagens (em modo tolerância a falhas)

tempo:	I/O In	I/O out	Router In	Router Out	Transf In	Transf Out	Security In	Security Out	Disp In	I/O Incoming
1	0	0	0	0	0	0	0	0	0	0
2	0	0	2	0	0	0	0	0	0	0
3	0	0	3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	0	0	1	1
7	0	21	0	1	0	3	0	0	0	3
8	0	85	0	2	1	1	0	1	0	9
9	0	140	1	0	0	2	0	1	0	7
10	0	215	1	1	0	1	0	1	0	1
11	0	280	0	1	0	2	0	1	0	2
12	0	347	0	2	0	1	0	3	0	1
13	0	406	0	2	0	1	0	2	0	15
14	0	446	1	3	0	2	0	1	0	0
15	0	433	0	6	0	1	0	3	1	0
16	0	421	0	5	0	2	0	3	0	0
17	0	412	0	3	0	2	1	4	0	0
18	0	399	0	5	0	2	1	2	0	0

19	0	388	0	4	0	2	1	5	0	0
20	0	376	0	5	0	1	0	5	0	0
21	0	364	0	5	0	2	0	5	0	0
22	0	351	0	6	0	1	0	5	1	0
23	0	339	0	6	0	2	0	3	0	0
24	0	327	0	9	0	1	0	5	0	0
25	0	314	0	10	0	1	0	4	0	0
26	0	303	1	11	0	1	0	4	0	0
27	0	290	0	11	0	2	0	2	0	0
28	0	278	0	11	0	3	0	2	0	0
29	0	265	0	12	0	3	0	1	0	0
30	0	254	0	12	0	1	0	3	1	0
31	0	242	0	12	0	1	0	3	0	0
32	0	231	0	11	0	3	0	3	0	0
33	0	220	0	11	1	3	0	3	0	0
34	0	208	0	11	0	2	0	4	0	0
35	0	197	0	9	0	3	1	4	0	0
36	0	186	0	7	0	3	0	5	0	0
37	0	174	0	7	0	3	0	5	0	0
38	0	162	1	9	0	4	0	5	0	0
39	0	150	0	8	0	3	0	5	0	0
40	0	138	0	6	1	4	0	6	0	0
41	0	127	0	6	0	6	0	4	0	0
42	0	116	1	4	0	5	1	6	0	0
43	0	102	0	8	0	4	0	6	0	0
44	0	90	0	7	0	5	0	7	0	0
45	0	78	0	9	0	3	0	6	0	0
46	0	66	1	9	0	2	0	7	0	0
47	0	55	0	7	0	4	0	5	0	0
48	0	46	0	6	1	5	0	5	0	0
49	0	35	0	5	0	5	1	2	0	0
50	0	23	1	5	0	5	0	2	0	0
51	0	12	0	3	0	7	1	2	0	0
52	0	0	0	4	0	4	0	5	0	0
53	0	0	0	0	0	0	0	0	0	0