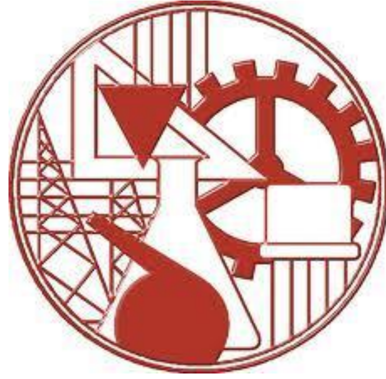


INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia Electrónica e Telecomunicações e de Computadores

Sistemas de Informação

Message Integration Bus

Igor André Gaspar Cândido

(Licenciado)

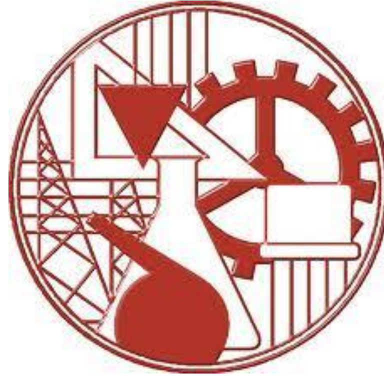
PROJECTO PARA OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA INFORMÁTICA DE COMPUTADORES

Orientador:

Fernando Miguel Carvalho

Setembro

(Página propositadamente deixada em branco)



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia Electrónica e Telecomunicações e de Computadores

Sistemas de Informação

Message Integration Bus

Igor André Gaspar Cândido

(Licenciado)

PROJECTO PARA OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA INFORMÁTICA DE COMPUTADORES

O aluno:

(Igor André Gaspar Cândido)

Orientador:

(Fernando Miguel Carvalho)

(Página propositadamente deixada em branco)

Resumo

O trabalho apresentado por este documento aborda os problemas que advêm da necessidade de integração de aplicações, desenvolvidas em diferentes instantes no tempo, por diferentes equipas de trabalho, que para enriquecer os processos de negócio necessitam de comunicar entre si. A integração das aplicações tem de ser feita de forma opaca para estas, sendo disponibilizada por uma peça de *software* genérica, robusta e sem custos para as equipas desenvolvimento, na altura da integração. Esta integração tem de permitir que as aplicações comuniquem utilizando os protocolos que desejarem. Este trabalho propõe um *middleware* [1] orientado a mensagens como solução para o problema identificado.

A solução apresentada por este trabalho disponibiliza a comunicação entre aplicações que utilizam diferentes protocolos, permite ainda o desacoplamento temporal, espacial e de sincronismo na comunicação das aplicações. A implementação da solução tem base num sistema *publish/subscribe* orientado ao conteúdo [2] e tem de lidar com as maiores exigências computacionais que este tipo de sistema acarta, sendo que a utilização deste se justifica com o enriquecimento da semântica de subscrição de eventos. Esta implementação utiliza uma arquitectura semi-distribuída, com o objectivo de aumentar a escalabilidade do sistema. A utilização da arquitectura semi-distribuída implica que a implementação da solução tem de lidar com o encaminhamento de eventos e divulgação das subscrições, pelos vários servidores de eventos.

A implementação da solução disponibiliza garantias de persistência, processamento transaccional e tolerância a falhas, assim como transformação de eventos entre os diversos protocolos.

A extensibilidade da solução é conseguida à custa de um sistema de *pluggins* que permite a adição de suporte a novos protocolos de comunicação. Os protocolos suportados pela implementação final do trabalho são **RestMS** [3] e **TCP** [4].

Palavras-chave: **Integração, Message-Oriented-Middleware**, **Protocolos**, ***Publish/Subscribe***, **Encaminhamento**, **Transformação**

(Página propositadamente deixada em branco)

Abstract

The work presented in this document addresses the problems in the domain of integrating applications, developed at different instants on time, by different teams, which need to be integrated to enrich the business processes. The integration of application needs to be opaque to the applications, being provided by generic software that is robust and without costs to the developers in the integration moment. This integration needs to enable the use of the desired protocols by the different applications. The current work proposes a message-oriented-middleware as the solutions for the identified problem.

The solution presented by this work provides the communication between applications using different protocols and also provides the decoupling in time, space and synchronism of communication among applications. The solution implementation is based in a publish/subscribe content oriented system and has to handle the computational demands of this type of system, being that the use of this type of system is justified by the enrichment of subscription of events semantics. This implementation uses a semi-distributed architecture with the objective of enabling the scalability of the system. The use of the semi-distributed architecture implies that the solution implementation has to handle the routing of subscriptions and the forwarding of events through the event servers.

The solution implementation provides guaranties of persistence, transactional processing and fault tolerance. The transformation of events between the different protocols types is also provided.

The extensibility of the solution is obtained by the plugins system that enables the addition of the support to new communication protocols. The protocols supported by the final implementation of this work are **RestMS** and **TCP**.

Keywords: **Integration, Message-Oriented-Middleware**, **Protocols**, **Publish/Subscribe**, **Routing**, **Transformation**

(Página propositadamente deixada em branco)

Índice

Índice de figuras xi

1 Introdução 1

1.1 Formulação do problema 1

1.2 Objectivos gerais 3

1.3 Organização do documento 3

1.4 Lista de acrónimos 4

2 Enquadramento 7

2.1 Diferentes paradigmas de interacção 7

2.1.1 Message Passing 8

2.1.2 RPC 8

2.1.3 Notificações 9

2.1.4 Espaços Partilhados 10

2.1.5 Message Queuing 11

2.1.6 Comparação de paradigmas de interacção 12

2.2 Variações do paradigma publish/subscribe 12

2.2.1 Sistemas publish/subscribe baseados em tópicos 12

2.2.2 Sistemas publish/subscribe baseados no conteúdo 13

2.2.3 Sistemas publish/subscribe baseados em tipos 15

2.2.4 Comparação entre os vários tipos de sistemas publish/subscribe 15

2.3 Algoritmos de comunicação multicast 15

2.3.1 Exemplo de definição de um protocolo de comunicação multicast 17

2.3.2 Conclusão 18

2.3.3 Estado da arte 19

2.4 Redes de sobreposição 20

2.4.1 Introdução 20

2.4.2 Trabalho relacionado 22

2.4.3 Estudo de redes de sobreposição e sistemas publish/subscribe orientados ao conteúdo 23

2.4.4 Protocolo Chord 24

2.4.5 Sistema *publish/subscribe* orientado ao conteúdo 24

2.4.6 Sistema *publish/subscribe* orientado ao conteúdo utilizando redes de sobreposição 26

2.4.7 Discussão 31
2.5 Protocolos 32
2.5.1 RestMS 32
2.5.2 AMQP 34
2.6 Conceitos envolvidos na implementação de um Message-Oriented Middleware 35
2.6.1 Eventos 35
2.6.2 Meio de comunicação 37
2.7 Soluções existentes 43
2.7.1 Apache QPID 43
2.7.2 BizTalk 44
2.7.3 Distributed Publish/Subscribe (PubSub) Event System 46
3 Solução 51
3.1 Arquitectura da solução 51
3.2 Aspectos de implementação 53
3.2.1 Comunicação com Blackbox 55
3.2.2 Arquitectura semi-distribuída 58
3.2.3 Pipeline 62
3.2.4 Transacções 64
3.2.5 Persistência 65
3.2.6 Algoritmo de correspondência de eventos 67
3.2.7 Transformadores 71
3.2.8 Loader 72
3.2.9 Adaptadores para clientes 73
3.3 Protocolos 81
3.3.1 TCP 81
3.3.2 RestMS 83
4 Conclusão 90
5 Trabalho futuro 95
6 Bibliografia 97

Índice de figuras

Figura 1 - Espaço de eventos 25

Figura 2 - Figura que demonstra a arquitectura da solução proposta, retirada de [28] .. 27

Figura 3 - Exemplos dos mapeamentos, assim como subscrição e evento exemplo, imagem retirada de [28] 29

Figura 4 - Arquitectura lógica do protocolo RestMS, imagem retirada de [3] 33

Figura 4 - Arquitectura de BizTalk Server, retirada de [50] 44

Figura 5 – Exemplo de arquitectura de servidores de eventos, imagem retirada de [55]47

Figura 6 – Arquitectura de sistema ,imagem retirada de [55] 49

Figura 7 - Arquitectura Lógica MIB 52

Figura 8 - Arquitectura física MIB 52

Figura 9 - Arquitectura física de MIB semi-distribuído 53

Figura 10 - Diagrama UML de classes de Operation, PubSubContract, Event e Subscription 56

Figura 11 - Diagrama UML de classes das classes Operation, Interbroker, Routing e Forwarding 57

Figura 12 - Exemplo de uma rede de instâncias Blackbox 59

Figura 13 - Esquema de comunicação entre instâncias Blackbox 60

Figura 14 - Modelo de dados MIB 66

Figura 15 - Modelo de dados de comunicação entre Blackbox 67

Figura 16 – Exemplo de uma PST 68

Figura 17 - Diagrama UML de classes de SubscriptionTree 69

Figura 18 - Transformação de um evento do protocolo X num do protocolo Y 71

Figura 19 - Diagrama UML de classes da interface a utilizar com entidade Emissor e Receptor, quando o protocolo não fornece formato para as mensagens 74

Figura 20 - Emissão de um evento/subscrição de forma síncrona 75

Figura 21 - Emissão de um evento/subscrição de forma assíncrona 76

Figura 22 - Recepção de eventos, utilizando interacção do tipo Pull de forma síncrona77

Figura 23 - Recepção de eventos, utilizando interacção Push, de forma síncrona 79

Figura 24 - Diagrama UML de classes da hierarquia de classes para interface de adaptador para cliente TCP.. 82
Figura 26 - Modelo relacional da base de dados RestMS 87
Figura 27 - Diagrama UML de classes de conjunto de classes interface com cliente de adaptador RestMS 88

1 Introdução

O cenário empresarial actual define que as aplicações não podem operar isoladamente, sendo que existe a necessidade de ligar várias aplicações, para que possa existir cooperação e desta forma se formarem processos de negócio mais ricos.

As diferentes aplicações podem ser construídas em diferentes momentos no tempo, e por vezes, sobre diferentes plataformas e ainda sem conhecimento mútuo. Sendo que a determinada altura no tempo se determina que duas aplicações, sem qualquer ligação, necessitam de trocar informação. Uma possível solução para este problema de integração seria alterar as aplicações, criar um protocolo de comunicação e depois de realizada a implementação das novas funcionalidades, voltar a testar a aplicação. Esta solução tem o problema de ser demorada e dispendiosa.

A melhor solução, em termos financeiros e temporais, é utilizar-se uma peça de *software* que permita a interligação de aplicações e possa ser utilizada de forma genérica.

Um *Message-Oriented Middleware* [5] (**MOM**) é uma peça de *software* que, utilizando mensagens, permite a comunicação entre aplicações. Este tipo de solução não é desenhada à medida e é suficientemente configurável para servir as necessidades de interligação de diversas aplicações. Normalmente este tipo de solução disponibiliza garantia de entrega de mensagens e processamento transaccional. São também focados aspectos como o desacoplamento no tempo, no espaço e na sincronização.

1.1 Formulação do problema

As soluções **MOM** disponibilizam comunicação entre diversos clientes. Uma correcta implementação de um **MOM** tem de escolher um paradigma de comunicação, tendo como objectivos disponibilizar a comunicação com o melhor grau de semântica no poder de expressão de intenções do cliente e melhor desempenho, possíveis.

Um **MOM**, nos dias de hoje em que a necessidade de integração de aplicações é cada vez maior, tem de ter a capacidade de escalar, ou seja, de ao serem aumentados os seus recursos, suportar a comunicação com um maior número de clientes.

A comunicação *multicast* é um tema de estudo de relevância quando o objectivo é disponibilizar a comunicação entre vários clientes. Este tipo de comunicação permite que um cliente com uma só acção interaja com vários clientes destinatários. Este tipo de comunicação pressupõe que a lógica de propagação de informação do **MOM** é distribuída por vários componentes, permitindo um melhor escalonamento dos custos de desempenho, pelos diversos componentes do **MOM**.

Os clientes dos **MOM** assumem um carácter dinâmico, ou seja, utilizam ligações que podem falhar, os próprios clientes podem falhar, ou ficar indisponíveis durante períodos de tempo. O **MOM** que disponibiliza a comunicação entre os vários tem de suportar estas alterações relativas à disponibilidade dos clientes e ver o seu estado reconfigurado.

As redes de sobreposição figuram uma possível solução para este problema ao dividir os custos da comunicação por cada um dos clientes, portanto a adição de novos clientes, além de aumentar os custos de desempenho, também figura um acréscimo nos recursos disponíveis. As redes de sobreposição também têm a vantagem de suportar bem as alterações na disponibilidade dos clientes, ao terem a capacidade de se configurarem automaticamente, aquando a adição ou remoção de clientes do sistema.

A pluralidade dos protocolos de comunicação é também outro factor relevante na integração de aplicações. Idealmente todas as aplicações deveriam comunicar utilizando os protocolos *standard* utilizados hoje em dia, mas quando se pretende a integração de aplicações desenvolvidas em diferentes instantes no tempo, tem de ser disponibilizado o suporte para os protocolos utilizados. Portanto tem de ser feito um estudo dos protocolos considerados *standard* na comunicação com **MOM**, por forma a permitir a integração de aplicações desenvolvidas actualmente, mas também disponibilizar a adição de suporte de outros protocolos.

A solução a implementar tem de disponibilizar uma semântica no poder de expressão de intenções do cliente elevada, um bom desempenho, ser escalável e suportar falhas de clientes. Portanto escolheu-se o padrão de comunicação *publish/subscribe* orientado ao conteúdo dos eventos, por forma a disponibilizar um elevado grau na semântica no poder de expressão de intenções do cliente. Este tipo de sistema acarta problemas de desempenho, portanto deve ser considerada uma estrutura, que permite um melhor

desempenho, para conter e corresponder as intenções dos clientes com a informação a circular no sistema.

O escalonamento é conseguido ao ser utilizado uma arquitectura semi-distribuída, que define a existência de servidores heterogéneos. A utilização deste tipo de arquitectura pressupõe a utilização de um mecanismo *multicast* para coordenar a comunicação entre os servidores heterogéneos, com o objectivo de fazer chegar a informação aos clientes que tem a intenção de a receber.

1.2 Objectivos gerais

A realização deste trabalho pressupõe a criação de um **MOM** que permita a integração de aplicações, desenvolvidas em diferentes instantes no tempo, por diferentes equipas de desenvolvimento.

Este **MOM** deve ser escalável, ser tolerante a falhas, permitir o desacoplamento no tempo, espaço e sincronismo. Deve também suportar a comunicação com aplicações desenvolvidas em qualquer linguagem de comunicação.

O **MOM** desenvolvido deve suportar os protocolos de comunicação de *middlewares standard*, assim como disponibilizar a adição de suporte de novos protocolos de comunicação.

É também objectivo disponibilizar uma semântica rica no poder de expressão de intenções do cliente.

1.3 Organização do documento

Este documento é organizado em cinco capítulos.

O primeiro capítulo faz uma introdução do trabalho apresentado, assim como exibe os seus objectivos e formula os problemas envolvidos na solução do trabalho.

O segundo capítulo apresenta os conceitos essenciais para a realização do trabalho, assim como os conceitos relacionados com soluções alternativas para os problemas que o trabalho se propõe a resolver.

O terceiro capítulo explica a arquitectura da solução e mostra os aspectos de implementação que se consideram relevantes.

O quarto capítulo apresenta as conclusões retiradas da realização do trabalho.

O quinto capítulo enumera um conjunto de actividades que podem ser realizadas para enriquecer o trabalho já realizado.

1.4 Lista de acrónimos

Os seguintes acrónimos são utilizados ao longo do texto, de seguida são definidos os seus significados:

AMQP	Advanced Message Queuing Protocol
BAM	Business Activity Monitor
CORBA	Common Object Request Broker Architecture
DAC	Distribuited Asynchronous Collections
DSM	Distribuited Shared Memory
DTC	Distributed Transaction Coordinator
FIFO	First In First Out
FTP	File Transfer Protocol
GUID	Globally Unique Identifier
HTTP	Hipertext Transfer Protocol
HTTPS	Hipertext Transfer Protocol Secure
I3	Internet Indirection Infrastructure
JMS	Java Message Service
LAN	Local Area Network
MIB	Message Integration Bus
MIME	Multipurpose Internet Mail Extension
MOM	Message Oriented Middleware
MSMQ	Microsoft Message Queuing
POP3	Post Office Protocol 3
PST	Parallel Search Tree
RAM	Random Access Memory
RPC	Remote Procedure Call

SGBD	Sistema de Gerenciamento de Banco de Dados
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP	Transport Control Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Unique Resource Location
WAN	Wide Area Network
WCF	Windows Communication Foundation
WSE	Web Services Enhancements
WSS	Windows SharePoint Services
XPath	XML Path Language

(Página propositadamente deixada em branco)

2 Enquadramento

O corrente capítulo faz um enquadramento que resulta de um trabalho de pesquisa sobre 2.1 Diferentes paradigmas de interacção, 2.2 Variações do paradigma publish/subscribe, 2.3 Algoritmos de comunicação multicast, 2.6 Conceitos envolvidos na implementação de um Message-Oriented Middleware e 2.4 Redes de sobreposição

O paradigma *publish/subscribe* foi um ponto de foco neste estudo, por ser considerado o paradigma de comunicação de eleição, em termos de comunicações genéricas.

Este capítulo também enquadra diferentes 2.5 Protocolos, uma vez que estes constituem a linguagem de comunicação que os clientes utilizam para comunicar.

É também feito um estudo sobre algumas das 2.7 Soluções existentes de integração de sistemas, que se consideram mais relevantes para a solução a desenvolver.

2.1 Diferentes paradigmas de interacção

Os paradigmas de comunicação *Message Passing*, **RPC**(Remote Procedure Call) [6], notificações, espaços partilhados e filas de mensagens, são alternativos ao *publish/subscribe*. Estes paradigmas situam-se a níveis de abstracções diferente. Esta distinção de paradigmas de comunicação é defendida em [7].

A caracterização destes paradigmas de comunicação é feita segundo desacoplamento temporal, espacial e no sincronismo.

O desacoplamento temporal representa a capacidade de comunicação sem que o emissor e o receptor estejam activos no momento da comunicação, ou seja, é feita a separação entre o acto de emissão e recepção.

O desacoplamento espacial representa a ausência de conhecimento, pelas partes envolvidas na comunicação, da localização específica da outra parte.

O desacoplamento no sincronismo representa a comunicação de forma assíncrona, ou seja, a possibilidade de comunicar sem que o receptor tenha de estar em espera durante o acto da recepção, ou que o emissor realize a emissão sem que tenha de esperar a conclusão da recepção.

2.1.1 Message Passing

O *Message passing* pode ser visto como o antecessor das interacções distribuídas. Este paradigma representa uma comunicação de baixo nível, na qual os intervenientes comunicam enviando e recebendo mensagens.

A passagem de mensagens é assíncrona para o produtor, enquanto geralmente é síncrona para o consumidor. O produtor e o consumidor estão acoplados no tempo e no espaço, uma vez que têm de estar activos ao mesmo tempo. O receptor de uma mensagem tem de conhecer o emissor desta.

2.1.2 RPC

Uma das formas de interacção distribuída mais utilizadas é a remota, sendo esta uma extensão da noção de invocação de uma operação, para um contexto distribuído. Ao expor as interacções remotas da mesma forma que as interacções locais, o modelo de comunicação **RPC** e as suas derivações distribuídas, facilitam a programação distribuída. Contudo a distribuição não pode ser exposta de forma totalmente opaca à aplicação, caso assim fosse, dar-se-ia aso a potenciais falhas, como por exemplo um *timeout* no computador destino ou uma falha no mesmo, que têm de ser tratadas especialmente para este tipo de operações, portanto a aplicação tem de ter noção que está a fazer uma interacção remota.

O paradigma de comunicação **RPC** diferencia-se do *publish/subscribe* em termos de acoplamento, nomeadamente: a natureza síncrona deste paradigma introduz um acoplamento no tempo, sincronização na parte do consumidor, e no espaço, uma vez que os invocadores têm uma referência remota para os invocados.

Foram feitas várias tentativas para remover o acoplamento de sincronização na comunicação remota e evitar o bloqueio do fio de execução invocador, enquanto este espera pela resposta da invocação remota.

A primeira tentativa passou por providenciar um tipo especial de assincronismo em que as invocações remotas não tinham valor de retorno. O **CORBA**(Common Object Request Broker Architecture) [8] disponibiliza um tipo especial de modificador de um só sentido, que pode ser utilizado para especificar este tipo de métodos. Esta tentativa leva a invocações com poucas garantias de fiabilidade, uma vez que o emissor não

recebe notificações de sucesso ou falha, designando-se este tipo de abordagem como *send-and-forget*.

A segunda variação do paradigma **RPC** é menos restritiva pois suporta valores de retorno, mas não os disponibiliza directamente ao fio de execução invocador. Estes valores são disponibilizados através de um mecanismo que é acedido quando os verdadeiros valores de retorno são necessários. Através desta solução do problema, conhecida como *future* [9] ou *future type message*, o fio de execução invocador pode continuar a processar e requisitar o valor de retorno mais tarde, graças ao mecanismo referido anteriormente.

2.1.3 Notificações

O paradigma de notificações consiste na separação das invocações remotas em duas invocações assíncronas, esta separação é feita com o objectivo de conseguir o desacoplamento de sincronização. Neste paradigma as duas invocações assíncronas são sequenciais, sendo que a primeira é enviada pelo cliente para o servidor, transportando os argumentos da invocação e uma referência para o cliente, e a segunda é enviada pelo servidor para o cliente, consistindo numa resposta. Este esquema pode ser facilmente estendido para o servidor passar a fazer várias chamadas de resposta para o cliente.

O paradigma de notificações, onde os subscritores registam o seu interesse directamente nos publicadores, sendo que estes fazem a gestão das subscrições e enviam eventos, corresponde ao padrão de desenho *observer* [10]. Este paradigma é normalmente implementado utilizando invocações assíncronas para conseguir o desacoplamento de sincronização. No entanto, apesar dos publicadores notificarem os subscritores de forma assíncrona, ambos estão acoplados no tempo e no espaço, pois o envio de informação pressupõe que o receptor tem de esperar a emissão e o emissor tem de esperar a conclusão da recepção, e ambos têm de saber a localização do outro.

A gestão de comunicação é feita pelo publicador, podendo sobrecarrega-lo à medida que o sistema cresce em dimensão.

2.1.4 Espaços Partilhados

Um espaço de memória distribuído (**DSM**(Distributed Shared Memory)) providencia alojamento, num sistema distribuído, com a acessibilidade dum espaço comum, sendo que este é partilhado através de espaços de endereçamento disjuntos. Um exemplo de espaço de memória distribuído é um *tuple space* [11].

Um *tuple space* é composto por uma colecção ordenada de *tuples*, que são igualmente acessíveis pelos clientes do sistema distribuído. A comunicação entre os clientes do sistema é feita através da inserção e remoção de *tuples no tuple space*. Este tipo de sistema normalmente disponibiliza três tipos de operações, sendo elas: *out* que coloca um *tuple* no *tuple space*; *in* que lê e remove um *tuple* do *tuple space*; *read* que lê um *tuple* do *tuple space*.

O modelo de interacção providenciado pelo paradigma de espaços partilhados possibilita o desacoplamento no tempo e no espaço, uma vez que os produtores e consumidores de *tuples* permanecem anónimos entre eles.

Uma operação do tipo *in* implementa uma interacção com semântica *one-to-many*, uma vez que apenas um consumidor lê um *tuple*, enquanto uma operação do tipo *read* implementa uma interacção com semântica *one-to-many*, em que um *tuple* pode ser lido por vários consumidores.

Ao contrário do paradigma *publish/subscribe*, o paradigma de espaços partilhados não disponibiliza o desacoplamento da sincronização pois os consumidores recolhem novos *tuples* do *tuple space* de forma síncrona. Esta característica limita a escalabilidade do paradigma, uma vez que os consumidores se sincronizam com os produtores.

Sistemas que seguem este paradigma, como o **JavaSpaces** [12], compensam o acoplamento na sincronização com notificações assíncronas.

O sistema **Internet Indirection Infrastructure** (I3) [13] implementa um conceito semelhante às notificações assíncronas, chamado *rendezvous*. Este conceito implica que em vez de se realizar o envio de pacotes explicitamente para um destino, cada pacote é associado a um identificador. Este é utilizado pelo receptor para obter o pacote. Este

nível de desassociação permite o desacoplamento entre o acto de envio e o acto de recepção.

2.1.5 Message Queuing

Os paradigmas *Message Queuing* e *publish/subscribe* estão ligados, uma vez que sistemas que implementam o paradigma *Message Queuing*, normalmente integram alguma forma de interacção *publish/subscribe*.

As aproximações tão centradas em mensagens são designadas de *message-oriented middleware* (MOM).

O modelo de interacção das *message queues* tem algumas semelhanças com os *tuple spaces*, uma vez que as pilhas podem ser vistas como espaços globais que são preenchidos pelos publicadores. Do ponto de vista funcional, as *message queues* também possibilitam as garantias transaccionais e dão garantias de ordem.

Nos sistemas que implementam o paradigma *message queuing*, as mensagens são recolhidas pelos consumidores de forma concorrente com semântica de *one-of-many*. Este modelo de interacção é também referenciado como *queuing* ponto-a-ponto.

A escolha do elemento a ser recolhido da pilha não é baseada na estrutura do elemento, mas na ordem com que os elementos são colocados na pilha, normalmente seguindo a filosofia FIFO ou baseada num esquema de prioridades.

Os produtores e consumidores estão desacoplados no tempo e no espaço. No entanto, as *message queues* não disponibilizam desacoplamento no sincronismo, uma vez que os consumidores recolhem mensagens de forma síncrona.

Alguns sistemas que implementam o paradigma *message queuing* disponibilizam suporte limitado para entrega de mensagens de forma assíncrona. No entanto, estes mecanismos assíncronos não são escaláveis para grandes números de consumidores, por causa das interacções adicionais necessárias para dar garantias transaccionais e garantias de ordem.

2.1.6 Comparação de paradigmas de interacção

Os paradigmas de interacção tradicionais diferem do *publish/subscribe* pelas suas limitações em suportarem desacoplamento no espaço, no tempo e na sincronização.

A Tabela 1 mostra a caracterização dos paradigmas discutidos nas secções anteriores.

Paradigma	**Desacoplamento Temporal**	**Desacoplamento Espacial**	**Desacoplamento no Sincronismo**
Message Passing	Não	Não	Consumidor: Não
RPC	Não	Produtor: Não	Não
Notificações	Não	Não	Sim
Espaços Partilhados	Sim	Sim	Não
Message Queuing	Sim	Sim	Não

Tabela 1 - Caracterização quanto a desacoplamento temporal, espacial e no sincronismo dos paradigmas Message Passing, RPC, Notificações, Espaços Partilhados, Message Queuing

2.2 Variações do paradigma publish/subscribe

O paradigma *publish/subscribe* faz a separação entre o acto de publicação e subscrição de informação, sendo que a publicação se refere ao acto em que um cliente publica nova informação no sistema, e o acto de subscrição corresponde ao acto em que um cliente expressa a vontade de receber determinada informação.

Os subscritores normalmente estão interessados em eventos particulares ou em padrões de eventos, não estando interessados em todos os eventos. As diferentes formas de especificar quais os eventos em que os subscritores estão interessados levaram a vários esquemas de subscrição.

2.2.1 Sistemas publish/subscribe baseados em tópicos

As primeiras versões do paradigma *publish/subscribe* eram baseadas nas noções de tópico ou assunto e foram implementadas por várias soluções industriais, tais como, **Altherr**, **Talarian** [14], **Skeen** e **TIBCO**. Nestas versões do paradigma, os participantes podem publicar eventos e subscrever tópicos particulares que são identificados por palavras-chave. Subscrever o tópico T pode ser visto como tornar-se

membro do grupo T. Publicar um evento com o tópico T, consiste no *broadcast* desse evento para todos os membros do grupo T.

Na prática, os sistemas *publish/subscribe* baseados em tópicos introduzem uma abstracção programática que mapeia tópicos individuais em canais de comunicação distintos. Cada tópico é visto como um serviço de eventos, sendo identificado por um nome único, tendo uma interface que permite publicar eventos no próprio tópico e subscrever eventos associados ao tópico. A abstracção de tópicos é fácil de entender e força a interoperabilidade entre plataformas ao utilizar apenas conjuntos de caracteres como chaves para dividir o espaço de eventos.

Foram propostos vários melhoramentos ao esquema de interacção orientado ao tópico, sendo que o mais útil consiste no uso de hierarquias para orquestrar tópicos. Quase todos os sistemas baseados em tópicos oferecem uma forma de endereçamento hierárquico, que permite que os programadores organizem os tópicos de acordo com o confinamento de relações. Uma subscrição feita a um nó da hierarquia implica a subscrição de todos os subtópicos daquele nó. Os nomes dos tópicos são geralmente representados utilizando a notação **URL**(Uniform Resource Locator) e introduzem uma hierarquia semelhante às redes de notícias **USENET** [15]. A maior parte dos sistemas permitem que os nomes dos tópicos contenham *wildcards*, esta característica foi introduzida pelo sistema **TIBCO Rendezvous**. A utilização de *wildcards* nos nomes dos tópicos permite subscrever e publicar para vários tópicos cujos nomes correspondam a um conjunto de palavras-chave, tal como uma sub-árvore ou um nível específico da hierarquia.

2.2.2 Sistemas publish/subscribe baseados no conteúdo

Apesar de todas as tentativas de melhoramentos, os sistemas *publish/subscribe* baseados em tópicos representam um esquema estático com expressividade limitada. Os sistemas *publish/subscribe* baseados em conteúdo, ou propriedades, têm vantagens em relação aos esquemas baseados em tópicos por introduzirem um esquema de subscrições baseado no conteúdo dos eventos. Os eventos não são classificados de acordo com um critério previamente definido, como por exemplo o nome de um tópico, mas de acordo com as propriedades do próprio evento. Estas propriedades podem ser atributos internos, das estruturas dos eventos, ou meta-data associada aos eventos.

Neste esquema, os consumidores subscrevem eventos ao especificarem filtros, utilizando uma linguagem de subscrição. Estes filtros definem restrições na forma de pares nome–valor, utilizando operadores de comparação básica para definir quais os eventos a considerar. As restrições podem ser combinadas para formar padrões de subscrição. Os padrões de subscrição são utilizados para identificar os eventos de interesse de um determinado subscritor, e propagá-los. É disponibilizada uma variante da operação subscrição, possibilitando a passagem de um padrão de subscrição. Este padrão pode ser representado das seguintes maneiras:

- *String*: Os padrões de subscrição são muitas vezes especificados utilizando gramáticas textuais, armazenadas em *strings*, tais como **SQL**(Structered Query Language) [16], **OMG Default Filter Constraint Language** [17], **XPath** [18], ou outras gramáticas proprietárias. A descrição textual, na forma de *string*, é descodificada pelo mecanismo apropriado.
- *Template object*: O sistema **JavaSpace** [19], inspirado no sistema de correspondência baseado em *tuples*, adoptou um mecanismo de correspondência também baseado em *tuples*. Utilizando este mecanismo, no acto de subscrição de eventos, um participante fornece um objecto t, sendo que este objecto indica que o participante está interessado em todos os eventos que estão de acordo com o tipo de t e cujos atributos correspondem aos atributos correspondentes de t, excepto aqueles que contêm o *wildcard null*.
- Código executável: Os subscritores disponibilizam um objecto capaz de filtrar eventos em tempo de execução. A implementação deste objecto é deixada a cabo do programador da aplicação. Uma aproximação alternativa ao problema, baseada numa biblioteca de filtros implementados utilizando reflecção, foi descrita por *Eugester e Guerraoui* [20]. Este método de definição de padrões de subscrição não é muito utilizado pois os filtros resultantes são extremamente difíceis de optimizar.

Um sistema baseado no conteúdo dos eventos disponibiliza uma granularidade de subscrição superior a sistemas baseados em tópicos. Para a mesma funcionalidade ser atingida utilizando tópicos, o subscritor teria de filtrar eventos irrelevantes, ou os tópicos teriam de ser divididos em vários subtópicos. A primeira possibilidade leva a um uso ineficiente de largura de banda, enquanto a segunda resulta num elevado número de tópicos e consequentemente um risco acrescido de eventos redundantes.

2.2.3 Sistemas publish/subscribe baseados em tipos

Normalmente os tópicos agrupam eventos que apresentam aspectos em comum, não só no seu conteúdo, mas também na sua estrutura. Esta observação levou à ideia de substituir o modelo de classificação baseado em nomes de tópicos, por um modelo que filtra os eventos de acordo com o seu tipo.

A noção de tipo de evento é directamente correspondida à representação do evento, na linguagem de programação utilizada. Esta correspondência permite uma integração entre a linguagem e o *middleware*. Esta característica permite a verificação de tipo, assegurada em tempo de compilação, uma vez que a abstracção resultante é parametrizada com o tipo dos eventos, resultando em código sem utilização de conversão de tipo. Enquanto as abordagens baseadas em *templates*, como a utilizada no **JavaSpaces**, consideram os tipos dos eventos uma propriedade dinâmica, resultando em interfaces que exigem a conversão explícita de tipos.

Os sistemas *publish/subscribe* baseados em tipo podem ser implementados através de sistemas baseados em conteúdo, que utilizem filtragem por membros públicos, sendo estes considerados o tipo do evento.

2.2.4 Comparação entre os vários tipos de sistemas publish/subscribe

A abordagem baseada em tópicos é estática e primitiva mas pode ser implementada de forma eficiente. Por outro lado, a abordagem baseada em conteúdo é altamente expressiva mas requer protocolos sofisticados com custos de execução elevados. Estes custos adicionais levam a que se devam escolher esquemas estáticos em situações em que as possibilidades de "grupos" de eventos sejam limitadas. A expressividade pode ser atingida ao aplicarem-se filtros baseados no conteúdo, no contexto de tópicos configurados estaticamente. Esta técnica é utilizada em casos em que os valores possíveis, da propriedade em questão, não estão dentro de intervalos discretos.

2.3 Algoritmos de comunicação multicast

O paradigma *publish/subscribe* baseado em conteúdos é o paradigma mais poderoso, no entanto as suas implementações eficientes e escaláveis acatam problemas difíceis de contornar. Estas implementações definem servidores de eventos, como entidades que têm a responsabilidade de implementar a lógica de comunicação entre clientes. A

escalabilidade do sistema é conseguida com a constituição de uma rede de servidores de eventos, dividindo os custos de desempenho da lógica de comunicação, entre clientes, por várias instâncias de servidores de eventos, sendo que cada servidor é responsável por um subconjunto de clientes. Uma implementação eficiente deste paradigma tem de resolver quatro problemas chave:

- A correspondência de um evento com um grande número de subscritores, num único servidor de eventos
- Distribuir de forma eficiente eventos pela rede de servidores de eventos. Este problema toma maior relevância quando:
- O sistema *publish/subscribe* é geograficamente distribuído e os servidores de eventos estão ligados com conexões lentas.
- Existe um grande número de publicadores, subscritores e eventos.

Em ambos os casos deve-se limitar a distribuição dos eventos aos servidores de eventos que tenham subscritores para os eventos a distribuir.

Uma das grandes vantagens dos sistemas baseados no paradigma *publish/subscribe* baseado em tópicos é o facto dos problemas enumerados anteriormente serem facilmente resolvidos. A correspondência é implementada à custa de uma pesquisa numa tabela; enquanto o problema de distribuição de eventos é resolvido formando grupos de distribuição para cada tópico de eventos.

Existem duas soluções simples para o problema de distribuição de eventos no paradigma *publish/subscribe* baseado em conteúdos:

- **Matching-first** – o evento é comparado com todas as subscrições, criando-se uma lista de destinos para o evento, de seguida este é enviado para todas as entradas dessa lista.
- **Flooding** – o evento é enviado para todos os possíveis destinos da rede, em cada um dos destinos é feita a filtragem de forma a rejeitar eventos que não sejam requisitados.

O **Matching-first** funciona melhor em pequenos sistemas, mas em grandes sistemas, com um elevado número de possíveis destinos, o acréscimo à dimensão dos eventos, causado pela adição de informação de encaminhamento aos cabeçalhos dos eventos, pode tornar-se impraticável. Utilizando esta técnica, podem existir várias cópias do mesmo evento a circular no mesmo troço de rede, a caminho de diferentes destinos.

O **Flooding** tem uma menor eficiência quando, num grande sistema, apenas um pequeno número de clientes deseja receber um determinado evento. Esta técnica não consegue explorar a localidade dos subscritores, pois é provável que subscritores numa determinada área geográfica tenham subscrições semelhantes.

O protocolo **Flooding** sobrecarrega uma rede com um número de publicações bastante menor que o **Match First**.

2.3.1 Exemplo de definição de um protocolo de comunicação multicast

O documento [21] caracteriza um protocolo de comunicação *multicast*, que visa melhorar a comunicação *multicast* entre os clientes e servidores de eventos. Este protocolo é constituído por um algoritmo de correspondência de eventos a subscrições e por uma estratégia de distribuição de eventos pela rede de servidores de eventos.

2.3.1.1 Algoritmo de correspondência

O algoritmo de correspondência apresentado, baseia-se na ordenação e organização de subscrições numa estrutura de dados **PST**(Parallel Search Tree) [22]. Nesta estrutura cada subscrição corresponde a um caminho desde o nó raiz até a um nó folha. Cada nó não folha da estrutura contém um atributo e um valor. O processo de correspondência começa no nó raiz e consiste no percorrer de todos os caminhos que o evento satisfaça, até aos nós folha. Esta estrutura escala bem horizontalmente, uma vez que explora os aspectos em comum entre as várias subscrições, ao todas estas corresponderem a caminhos iniciais comuns desde o nó raiz.

2.3.1.2 Link Matching

O **Link Matching** é uma estratégia para distribuir eventos sem usar listas de destino. Aquando a recepção de um evento, cada servidor de eventos apenas faz a correspondência necessária para determinar quais dos seus vizinhos devem receber esse mesmo evento, sendo que estes vizinhos podem ser outros servidores ou subscritores. Isto é, em vez de determinar qual a lista de todos os subscritores do sistema que deve receber o evento, cada servidor apenas determina por quais das suas ligações cada evento deve ser encaminhado. Intuitivamente esta abordagem é mais eficiente, uma vez que o número de ligações de cada servidor é tendencialmente menor que o número de subscritores de todo o sistema.

2.3.2 Conclusão

Uma rede de servidores a utilizar o protocolo **Flooding** sobrecarrega, com um número de publicações de eventos significativamente menor, que utilizando o protocolo **Link Matching**, independentemente do número de subscrições. A utilização do protocolo **Link Matching** tem um melhor desempenho, quando comparada com o protocolo **Flooding**, em situações em que um evento tem como destino uma pequena percentagem dos clientes do sistema. No caso em que os eventos têm de ser distribuídos para a maior parte do sistema, a diferença de desempenho entre os dois protocolos discutidos não é muito significativa, uma vez que a maior parte das ligações da rede serão utilizadas para distribuir os eventos no protocolo **Link Matching**. Estes resultados mostram que o protocolo **Link Matching** é mais apropriado para distribuir eventos para partes da rede, sendo este o caso típicos dos sistemas que seguem o paradigma *publish/subscribe* sobre uma **WAN**. Nos casos em que a lista de destinos pode crescer, para incluir centenas ou milhares de destinos, o algoritmo **Match First** torna-se impraticável.

Distribuir um evento por toda a rede e filtrá-lo nos receptores, também tem as suas desvantagens. O algoritmo **Flooding** mostra que este tipo de abordagem de distribuição de eventos tem problemas de escalabilidade e inabilidade de explorar a localidade. O algoritmo **Flooding** é uma boa aproximação da distribuição *broadcast*, uma vez que a maior parte das técnicas de distribuição em **WAN** requerem o uso duma série de *routers* [23] ou *bridges* a ligarem ligações **LAN**.

O **Link Matching** é um algoritmo de distribuição de eventos baseado numa correspondência distribuída, parcial, de eventos publicados. Este algoritmo permite a distribuição de eventos por um grande número de consumidores distribuídos pela **WAN**, sem colocar uma carga exagerada sobre a rede. Este algoritmo também explora a localidade de subscrições.

Demonstra-se que uma rede de servidores de eventos a utilizar o algoritmo **Link Matching** permanece em execução, enquanto que uma rede de servidores de eventos que utilize o algoritmo **Flooding** fica sobrecarregada, este facto deve-se ao maior número de eventos que a rede que usa o **Flooding** tem de processar.

2.3.3 Estado da arte

Os diferentes sistemas *publish/subscribe* diferem entre si pela forma como implementam os mecanismos de encaminhamento de subscrições e informação. Um exemplo destas diferenças é o caso do **TIB/Rendezvous** [24], que implementa o encaminhamento de informação sem ter em conta o conteúdo da informação a ser encaminhada. Neste caso, a informação enviada por um servidor de eventos é distribuída por toda a árvore de servidores do sistema. Cada servidor guarda a informação dos subscritores presentes na sua **LAN**(Local Area Network), desta forma o encaminhamento de subscrições apenas é feito através dos limites das **LAN**s.

O sistema **Gyphon** [25] aplica um algoritmo de correspondência eficiente, em que cada servidor de eventos tem uma tabela com todas as subscrições do sistema. Estas tabelas são assumidas como estáticas. Os eventos são encaminhados, pelos servidores necessários, até chegarem aos seus subscritores, sendo que este encaminhamento é feito com base nas tabelas de subscrições.

O **Siena** [26] incorpora algoritmos de encaminhamento de subscrições, que especificam a forma como as tabelas de subscrição de cada servidor de eventos devem ser mantidas, sendo que se procura evitar a comunicação excessiva de subscrições, assim como a replicação desnecessária das mesmas. Estes algoritmos baseiam-se nas relações de confinamento entre as subscrições.

O factor em comum entre os três sistemas discutidos anteriormente consiste no facto de as operações de transporte de informação serem feitas através de uma topologia fixa de conexões **TCP**. Pode ser utilizada uma topologia dinâmica e auto configurável de ligações **TCP**, com propósito de melhorar o desempenho aplicacional. Esta topologia permitiria reduzir o número de ligações **TCP** necessárias para que a informação atinja o seu destino.

As redes de sobreposição (*overlay network infrastructures*) [27] possibilitam a criação de topologias dinâmicas de redes **TCP**. O principal objectivo destas redes é criar, de forma dinâmica, uma árvore de distribuição, a nível aplicacional, onde a raiz encaminha mensagens para um grupo dinâmico de clientes. Estas redes também procuram atingir alto desempenho, através da construção das árvores com vista à optimização dos

parâmetros de serviço de rede, robustez, através da adaptação da árvore a servidores com falhas, e heterogeneidade, qualquer cliente ligado à internet pode ser um cliente deste tipo de rede.

A noção de rede de sobreposição é facilmente utilizável no paradigma *publish/subscribe* baseado em tópicos, ao considerar-se que cada tópico está associado a uma árvore, e ao implementar-se o encaminhamento de informação utilizando-se o serviço de procura das redes de sobreposição. Esta utilização não pode ser feita no paradigma *publish/subscribe* orientado ao conteúdo, uma vez que cada publicação poderia necessitar de uma diferente árvore de *multicast*.

2.4 Redes de sobreposição

A informação apresentada neste subcapítulo é baseada no documento [28], sendo que a sua autoria é maioritariamente dos autores desse documento. É no entanto apresentada neste subcapítulo por fazer parte do trabalho de pesquisa e revelar a perspectiva do autor do presente documento.

As limitações na configuração automática e na adaptação dinâmica a mudanças são ultrapassadas pela utilização de redes de sobreposição estruturadas. Este feito é conseguido à custa da criação de um estrato mediador entre a rica semântica de subscrição de eventos dos sistemas *publish/subscribe* baseados no conteúdo dos eventos e a lógica *standard* de endereçamento dos esquemas de sobreposição.

Identifica-se que a falta de suporte nativo, das redes de sobreposição, para a comunicação de um para muitos é um dos grandes impedimentos para uma operação de sistema eficiente.

2.4.1 Introdução

Grande parte das aplicações, desde o domínio de integração de aplicações empresariais até às redes de sensores, pode beneficiar da presença de um sistema *publish/subscribe*. A forma mais geral de sistemas *publish/subscribe* suporta subscrições baseadas no conteúdo dos eventos, normalmente em que os subscritores podem expressar interesse nos eventos ao especificarem um conjunto de restrições sobre atributos dos eventos. Esta semântica de subscrição acarreta maiores custos computacionais, no entanto é mais flexível que as subscrições orientadas a tópicos, onde apenas um atributo do tópico

define a relação entre as fontes de informação e fluxos de informação. Contudo, as subscrições baseadas no conteúdo são mais difíceis de implementar, porque a correlação entre as fontes e os fluxos de informação não pode ser determinada *a priori*, tendo que ser computada com base em cada evento. Como consequência, o foco deste estudo são algoritmos eficientes de correspondência e encaminhamento de eventos. Sendo que o problema da divisão de uma implementação escalável, de um sistema *publish/subscribe* baseado no conteúdo, se manter por resolver.

Propõe-se uma arquitectura que beneficia da escalabilidade do encaminhamento de mensagens e da configuração automática adaptativa, das redes de sobreposição, para a implementação de um sistema *publish/subscribe*. Sendo que a arquitectura proposta tem as propriedades tão procuradas pelos esquemas *publish/subscribe* orientados ao conteúdo. Esta arquitectura não necessita de qualquer configuração manual ou de gestão, à parte da configuração da própria rede de sobreposição.

Desenhar uma infra-estrutura *publish/subscribe* baseada no conteúdo sobre o modelo *standard,* de comunicação e programação, disponibilizado pelas redes de sobreposição estruturadas, sendo exemplo destas: **CAN** [29], **Chord** [30], **Pastry** [31] e **Tapestry** [32], requer a abordagem de dois grandes problemas:

- As diferenças entre a linguagem rica que descreve uma subscrição e um evento, e um identificador único que identifica uma mensagem nas redes de sobreposição.
- A falta de eficiência inerente à implementação de comunicações de *one-to-many* sobre primitivas de comunicação ponto-a-ponto, normalmente disponibilizadas por redes de sobreposição.

Como forma de visar o primeiro problema enumerado, introduz-se uma nova classe de mapeamentos de subscrição estática, cuja natureza facilita a adaptação do sistema a mudanças dinâmicas.

A forma de lidar com o segundo problema referido consiste na introdução de uma primitiva de difusão múltipla, ao nível das redes de sobreposição, como forma de eliminar as ineficiências inerentes à utilização de primitivas de difusão ponto-a-ponto, das redes de sobreposição, para implementar semânticas *publish/subscribe* baseadas no conteúdo. Ao nível do sistema *publish/subscribe*, são propostas duas optimizações,

nomeadamente, tornar os mapeamentos discretos e, a recolha e o acumular de eventos, que podem aumentar a escalabilidade e desempenho do sistema.

2.4.2 Trabalho relacionado

Têm sido propostos na literatura vários sistemas *publish/subscribe* com um esquema de endereçamento para as subscrições baseado no conteúdo, tais como: **Gryphon** [33], **Hermes** [34], **JEDI** [35], **LeSubscribe** [36], **SIENA**,. Todos estes sistemas dependem de uma rede de servidores, ao nível aplicacional, que partilham a carga de determinar os receptores de um evento, assim como o encaminhamento dos eventos para os respectivos receptores. Por consequência o estudo sobre sistemas *publish/subscribe* baseados no conteúdo foca-se em algoritmos que se encarreguem das funções, de forma eficiente e eficaz. Nenhum dos sistemas *publish/subscribe* baseados no conteúdo propostos tem uma configuração automática, sendo que requer a intervenção humana para configurar e gerir a rede de servidores, de nível aplicacional. Esta característica limita a instalação destes sistemas em sistemas com configuração em larga escala.

A partição do espaço de eventos é uma aproximação alternativa ao problema. Esta técnica pressupõe que o espaço de eventos é dividido num conjunto de partições, sendo que cada uma é atribuída a um nó. Esta aproximação minimiza o tráfego de eventos ao encaminhar cada evento para apenas um nó. Esta característica simplifica a configuração inicial do sistema e elimina a necessidade de propagar e manter o conhecimento do estado do sistema, como por exemplo as subscrições armazenadas, desta forma torna-se a arquitectura menos dependente de estado global e menos vulnerável a falhas.

A configuração automática e a tolerância a falhas são capacidades que caracterizam as infra-estruturas de redes de sobreposição estruturadas ponto-a-ponto, tais como: **CAN**, **Chord**, **Pastry** e **Tapestry**. Estes sistemas disponibilizam esquemas de endereçamento, independentes dos endereços de rede, que são utilizados para implementar mecanismos de encaminhamento, a nível aplicacional, e são adaptativos à junção e remoção de nós. O **Scribe** [37] e o **Bayeux** [38] são dois sistemas *publish/subscribe* construídos com base no **Pastry** e **Tapestry**, respectivamente, que beneficiam da sua escalabilidade, eficiência e configuração automática. Contudo estes sistemas apenas disponibilizam endereçamento com base em tópicos, limitando a expressividade dos utilizadores.

2.4.3 Estudo de redes de sobreposição e sistemas publish/subscribe orientados ao conteúdo

A ideia comum por detrás da configuração automática e do encaminhamento, na maior parte das redes de sobreposição, é que as mensagens em vez de serem encaminhadas directamente sobre os endereços dos nós físicos de um espaço N, são encaminhadas com base em identificadores lógicos, definidos num espaço K. A rede de sobreposição gere o mapeamento KN: K -> N de chaves para os nós reais, denotados como mapeamentos KN, por outras palavras, cada chave é coberta por algum nó, por exemplo, o que se mapeia para o valor mais próximo da chave. O sistema encaminha, de forma automática, uma mensagem para o nó que cobre a chave da mensagem.

Enquanto as chaves são expostas para a aplicação, os mapeamentos KN são da responsabilidade da rede de sobreposição e são tipicamente escondidos do utilizador. Este tipo de encaminhamento, baseado numa chave, disponibiliza uma conveniente abstracção de alto nível à aplicação. Em adição, este permite que o sistema se adapte rapidamente às alterações dinâmicas, tais como uma falha ou a adição de nós ao sistema.

Virtualmente todos os esquemas de sobreposição disponibilizam uma interface semelhante, às aplicações, esta interface consiste nas seguintes primitivas: *send(m,k)* operação que envia uma mensagem m a um destino determinado pela chave k, *join( )* e *leave( )* operações que permitem que um nó se junte ou se remova do sistema e *deliver(m)* operação que invoca a aplicação informando a entrega da mensagem m.

2.4.4 Protocolo Chord

O protocolo **Chord** é baseado na rápida computação distribuída de uma função *hash*, que mapeia chaves nos nós que as cobrem. Um nó **Chord** apenas mantém informação sobre O(log n) outros nós, numa rede de n nós.

A atribuição de chaves a nós é feita através da técnica *consistent hashing*. A função *consistente hash* atribui a cada nó e chave, um identificador com m bits, utilizando **SHA-1** [39]. Estes identificadores são ordenados num círculo de identificadores de módulo 2^m. O círculo de identificadores é chamado anel **Chord**. O termo chave é utilizado para referir tanto a chave original como a sua imagem sobre a função *hash*, sendo que o significado será claro sobre o contexto. Uma chave *k* é atribuída ao nó cujo identificador é igual ou segue-se a *k* no anel, sendo este chamado nó sucessor da chave *k*.

O **Chord** mantém uma tabela chamada *finger table*, com o propósito de disponibilizar uma correspondência mais eficiente. A entrada *i*-ézima, numa tabela no nó *n*, é o nó sucessor *s* do identificador (n + 2^{i-1}) módulo 2^m. O nó *s* é chamado o *i*-ézimo *finger* do nó *n*. Cada entrada também contém o endereço **IP** [40] e a porta do respectivo nó.

2.4.5 Sistema *publish/subscribe* orientado ao conteúdo

Um sistema *publish/subscribe* orientado ao conteúdo contém um conjunto de nós, sendo que cada um pode agir tanto como produtor, como consumidor de informação, tendo o papel de publicador e subscritor, respectivamente. Os publicadores e subscritores trocam informação sobre a forma de eventos e subscrições.

Os eventos são definidos de acordo com um modelo de dados, em que um evento é definido como um conjunto de pares atributo-valor. Cada atributo $e.a_i$ tem um nome e um tipo. O nome é uma *string* de caracteres, e o tipo é normalmente um tipo primitivo da linguagens de programação e interrogação, sendo exemplo, *integer*, *float*, *string*, etc. Desta forma, os eventos são definidos num espaço de eventos de d dimensões, denomidado Ω. As dimensões são os atributos e os eventos localizam-se na posição correspondente aos valores que os seus atributos assumem.

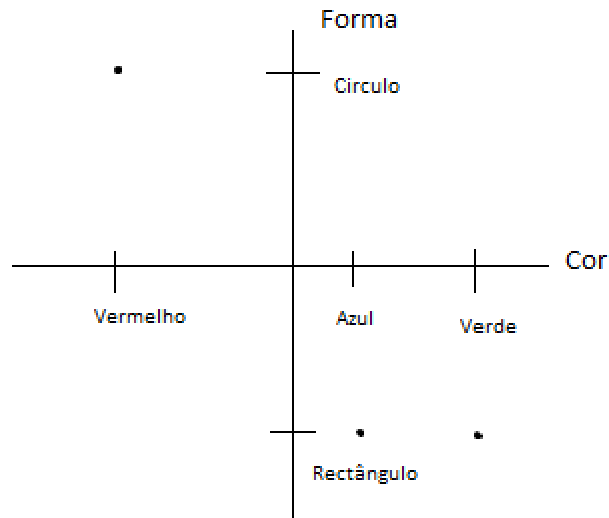


Figura 1 - Espaço de eventos

A Figura 1 mostra três eventos num espaço de eventos com os atributos forma e cor, sendo que os eventos assumem os seguintes valores:

1- Cor: Vermelho, Forma: Circulo
2- Cor: Azul, Forma: Rectângulo
3- Cor: Verde, Forma: Rectângulo

Os subscritores expressão o seu interesse em eventos através de *subscrições*. A *subscrição* σ é uma *interrogação* composta por uma conjunção de *restrições*. Uma *restrição* é indicada por $\sigma.c_i$. As *restrições* dependem do modelo de dados e da linguagem de subscrição utilizados. As *interrogações* são definidas de acordo com a assunção que são elementos do espaço ∑, constituído por todas as *subscrições* possíveis. Portanto, uma *interrogação* σ ∈ ∑ captura um subespaço do espaço de eventos, isto é σ ⊆ Ω. Pode-se dizer que um evento *e* ∈ Ω corresponde a uma subscrição σ ∈ ∑, se *e* só se satisfaz todas as restrições em σ, isto é, se e só se *e* ∈ σ. Quando um evento corresponde a uma subscrição, o subscritor tem de ser notificado quanto a *e*.

Os sistemas *publish/subscribe* baseados no conteúdo distribuem pelos nós do sistema as tarefas de guardar subscrições, corresponder eventos a subscrições e entregar eventos a subscritores, é desta forma que este tipo de sistemas implementa a correspondência de eventos. As subscrições em ∑ e os eventos em Ω são atribuídos a nós através de duas funções de mapeamento, sendo um exemplo destas funções, SN: ∑ -> 2^n e EN: Ω ->

2^n. Dada uma subscrição σ, SN(σ) retorna um conjunto de nós designados nós *rendezvous* de σ, estes nós são responsáveis por armazenar σ e encaminhar eventos que correspondam a σ para todos os subscrições de σ. EN(e) complementa SN ao retornar os nós *rendezvous* de e, que são responsáveis por corresponder e a subscrições registadas no sistema. Estas funções são utilizadas pelos nós da seguinte forma: quando um consumidor emite uma subscrição σ, envia-a aos nós retornados por SN(σ), sendo que estes armazenam σ e o identificador do consumidor. Os produtores enviam os seus eventos para os nós retornados por EN(e), estes correspondem e com as subscrições de que eles são responsáveis. Por cada subscrição que corresponde a e, e é encaminhado para o subscritor correspondente. Este esquema de correspondência funciona e encaminha e para os respectivos consumidores, porque os nós *rendezvous* de e, colectivamente, armazenam todas as subscrições correspondidas a e, isto é, se $e \in \sigma$ para qualquer subscrição σ, então $EN(e) \cap SN(\sigma) \neq \emptyset$. Esta propriedade é designada como a regra de intersecção de mapeamento.

2.4.6 Sistema *publish/subscribe* orientado ao conteúdo utilizando redes de sobreposição

Propõe-se uma arquitectura que é baseada em dois princípios:

- Utilizar e aproveitar as capacidades de configuração automática e encaminhamento escalável providenciados pelas redes de sobreposição, por forma a conseguir sistemas *publish/subscribe* orientados ao conteúdo que são escaláveis e adaptáveis a alterações.
- Utilizar uma forma de mapeamento que não depende das subscrições armazenadas, normalmente designados como mapeamentos sem estado.

Ambos os princípios enumerados acima contribuem para um sistema adaptativo e configurado automaticamente, sendo que, o primeiro torna o sistema adaptativo a falhas de nós e a junções, enquanto o segundo, elimina a necessidade de propagar as subscrições armazenadas.

2.4.6.1 Arquitectura do sistema

A Figura 2 mostra a arquitectura da solução proposta em [28], sendo que esta secção fala da solução proposta neste artigo, assim como aspectos de implementação da mesma. Tal como a figura demonstra, uma aplicação genérica, que utiliza a infra-estrutura *publish/subscribe* orientado ao conteúdo, pode fazer subscrições (sub()) e

publicações (pub()), tal como pode ser notificada da recepção de eventos que correspondam às suas subscrições (notify()). A camada de mais baixo nível da arquitectura é constituída pela rede de sobreposição que implementa o comportamento e providencia as primitivas deste tipo de sistemas.

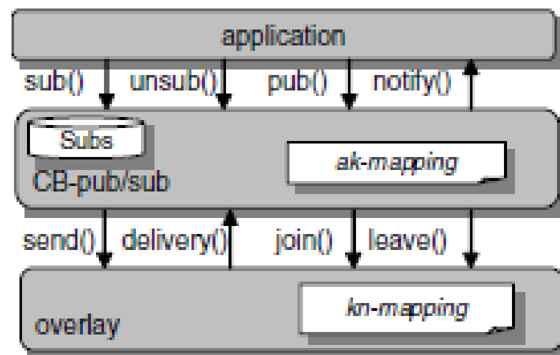


Figura 2 - Figura que demonstra a arquitectura da solução proposta, retirada de [28]

A camada CB-pub/sub mapeia o espaço de eventos ao universo de chaves, ao invés dos nós. Por outras palavras, a implementação tem de disponibilizar as funções de mapeamento SK : $\sum$ -> 2^k e EK : Ω -> 2^k, em vez das SN e EN.

Mais precisamente, a camada CB-pub/sub é responsável por implementar a funcionalidade de um sistema *publish/subscribe* orientado ao conteúdo ao explorar a infra-estrutura de rede de sobreposição sobre a qual opera. Esta camada implementa as seguintes operações:

- Implementa e calcula os mapeamentos SK e EK, representados na Figura 2 como *ak-mappings*.
- Encaminha subscrições σ e eventos *e*, para as chaves em SK(σ) ou EK(*e*), respectivamente. Este encaminhamento é conseguido ao invocar a primitiva ponto-a-ponto *send( )*, disponibilizada pela rede de sobreposição. Quando se envia uma subscrição, a chave do subscritor também é enviada.
- Recebe subscrições e eventos através da chamada de *delivery( )*, efectuada pela rede de sobreposição. As subscrições são armazenadas juntamente com as chaves dos subscritores, enquanto os eventos são correspondidos a subscrições armazenadas.
- Encaminha notificações, quando são encontradas correspondências entre eventos e subscrições. Se um evento corresponde a múltiplas subscrições, as chaves dos subscritores são determinadas e a notificação é enviada aos respectivos subscritores, utilizando a primitiva ponto-a-ponto *send( )*.

- Gere a adição e remoção de nós.

Ao esconder a dinâmica dos mapeamentos KN da aplicação, facilita-se o trabalho da aplicação, no entanto cria um problema às aplicações que dependem de estado, cuja distribuição está dependente da composição dos nós. Por exemplo, quando um novo nó *n* se junta ao sistema, as subscrições, que são mapeadas à sua partição do espaço de chaves, têm de ser movidas para *n* de outros nós. Do mesmo modo, quando um nó é removido do sistema ou falha, as subscrições de que este era responsável devem ser realocadas para os seus vizinhos, no espaço de chaves. À data deste estudo não existiam redes de sobreposição que gerissem a distribuição do estado de uma aplicação, nem expusessem informação acerca dos vizinhos do nó.

Felizmente, cada rede de sobreposição providencia uma maneira proprietária de enviar mensagens para os vizinhos, por exemplo, nó sucessor no **Chord**. Esta funcionalidade permite que um nó que se junte ao sistema recolha o estado dos seus vizinhos. Desta forma as falhas podem ser tratadas por cada nó, ao terem o seu estado replicado num pequeno número de vizinhos.

2.4.6.1.1 Mapeamentos sem estado

Nesta secção são apresentados três mapeamentos *ak*, que satisfazem a regra de intersecção de mapeamento. É utilizada a seguinte notação, para descrever os parâmetros do sistema:

- d, o número de dimensões do espaço de eventos
- Ω_i, projecção de Ω(espaço de eventos) em i dimensões
- K, o espaço de chaves

É importante que o K seja representado por um conjunto de bits, sendo m a dimensão do conjunto, para que seja possível a chave dividir o mapeamento em espaços. Esta característica é comum na maior parte das redes de sobreposição, tais como **Chord** e **Pastry**, é no entanto possível obter uma representação mais genérica, das chaves, noutros sistemas como o **CAN**, em que a chave é um ponto discreto num espaço multidimensional.

Todos os mapeamentos apresentados são baseados numa colecção de funções *hash* h_i : Ω_i -> $[0,1]^l$; h_i mapeia os valores dos atributos em Ω_i num conjunto de bits com

dimensão l. As funções de *hash* e o l fazem parte da definição dos mapeamentos e podem diferir entre mapeamentos. Dado o conjunto de funções h_i, é definido um conjunto de funções de *hash* H_i para as restrições $\sigma.c_i$, tanto para a igualdade como para a diferença, estas retornam conjuntos de bits com dimensão l, tendo a seguinte definição: $H_i(\sigma.c_i)$ = { h_i(x) | x ∈ Ω_i ∧ x satisfaz $\sigma.c_i$ }.

As definições dos mapeamentos apresentados especificam l mas utilizam h_i como um parâmetro externo.

Considera-se um espaço de eventos composto por dois atributos inteiros, a assumir valores num domínio de 0 a 7 cada (| Ω_i | = 8), para que $\mathrm{h}_1 = \mathrm{h}_2$ = h e $H_1 = H_2 = H$. O espaço de chaves deste exemplo coincide com o espaço de atributos, sendo m = 4. É utilizada a subscrição $\sigma = \{\, a_1 < 2{,}3 < a_2 < 7 \,\}$ e o evento e = { $a_1 = 1, a_2 = 6$ }. A Figura 3 mostra o funcionamento dos dois primeiros mapeamentos.

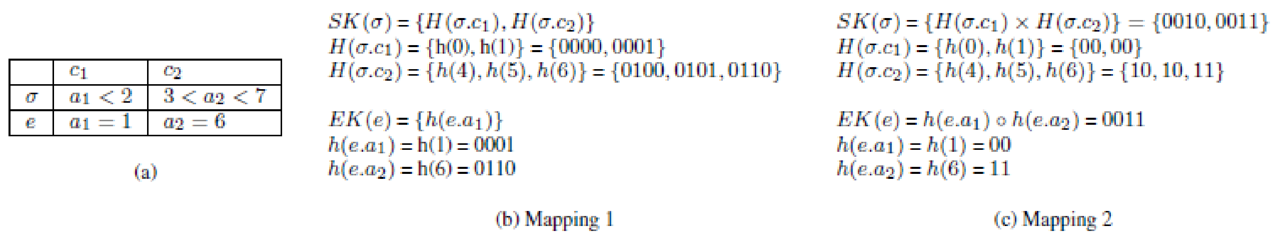


Figura 3 - Exemplos dos mapeamentos, assim como subscrição e evento exemplo, imagem retirada de [28]

Primeiro mapeamento: Divisão do espaço por atributos

A dimensão da sequência de bits retornada pelas funções h_i é igual ao número de bits da chave, isto é, l = m, para que as funções h_i e H_i retornem chaves e conjuntos de chaves simples. A ideia por detrás deste mapeamento consiste no cálculo do *hash* de cada restrição $\sigma.c_i$ de uma subscrição σ, independentemente do conjunto de chaves $\mathrm{H}_i(\sigma.c_i)$, e envio da subscrição para a união de todos os conjuntos. Este mapeamento satisfaz a regra de intersecção de mapeamento ao escolher-se uma chave de *rendezvous* para um evento, através do cálculo do *hash* de apenas um dos atributos do evento. A função SK é definida da seguinte forma: SK (σ) = $\cup_i\ H_i(\sigma.c_i)$ e a função EK é definida da seguinte forma: EK(e) = { $h_i(e.a_i)$, para algum i, 1 ≤ i ≤ d }. A Figura 3 mostra um exemplo deste mapeamento em que são utilizadas sequências de 4 bits retornadas por

SK e EK, quando invocadas sobre σ e e, respectivamente. Como são retornados dois valores por c1 e três por c2, SK(σ) retorna um total de cinco chaves.

Desta forma a função EK retorna apenas uma chave, mas a função SK retorna um conjunto de até O($\sum_{i=1}^{d} [ri.2^{m} / | \Omega_i |] = O$ (d+$2^{m}.\sum_{i=1}^{d} (ri / | \Omega_i |)$) chaves diferentes. Esta característica pode ser utilizada para aumentar a disponibilidade das subscrições. Contudo, se d é um valor elevado, a subscrição pode ser mapeada para um grande número de chaves, tendo como consequência a redução da escalabilidade do sistema. Os dois mapeamentos apresentados de seguida têm o objectivo de reduzir o número de chaves a que uma subscrição é mapeada.

Segundo Mapeamento: Divisão do espaço pela chave

Este mapeamento é baseado na ideia de partir os m bits do espaço da chave pelos atributos, para que [m / d] bits sejam atribuídos a cada atributo. Para que seja possível atingir a ideia enunciada, tem-se l = [m / d], neste mapeamento. A função SK retorna todas as concatenações possíveis das sequências de bits: SK(σ) = { $s_i \circ \ldots \circ s_d \mid s_i \in [0,1]^{l} \wedge s_i \in H_i(\sigma.c_i)$}. Consegue-se satisfazer a regra de intersecção de mapeamento ao se definir EK como: EK (e) = $h_1(e.a_1) \circ \ldots \circ h_d(e.a_d)$, isto é, retorna uma única chave rendezvous. A Figura 3 mostra um exemplo deste mapeamento. A função h retorna uma sequência de dois bits, para cada valor (l = m / d = 2).

O número de concatenações diferentes que SK retorna é $\Pi_{i=1}^{d} [ri . 2^{[m/d]} / | \Omega_i |]$.

Terceiro Mapeamento: Selectivo quanto a um atributo

Este mapeamento é baseado na observação, de que na maior parte dos casos, as subscrições podem exibir uma forte selectividade num determinado atributo, isto é, apenas não filtram uma pequena porção de todos os valores possíveis para este atributo, sendo um atributo selectivo. Estas restrições selectivas ocorrem frequentemente em espaços de eventos reais, por exemplo, restrições de igualdade em atributos como tipo ou tópico. A ideia por detrás deste mapeamento consiste em mapear uma subscrição σ apenas pela sua restrição selectiva $\sigma.c_s$, $r_s / | \Omega_i | = min_{i=1}^{d}(r_i / | \Omega_i |)$. $l = m$, como no primeiro mapeamento. A função SK é definida como: SK(σ) = $H_s(\sigma.c_s)$. Contudo,

cada evento tem de ser mapeado por cada um dos atributos em separado, portanto EK é definido como: EK(e) = $\cup_{i=1}^{d} \{h_i(e.a_i)\}$.

Neste mapeamento uma subscrição é mapeada em $[2^m.min_{i=1}^{d}(r_i/|\ \Omega_i\ |)]$ chaves. Este mapeamento é pelo menos d vezes melhor que o primeiro mapeamento, mesmo que não existam restrições selectivas. Contudo, a comparação com o segundo mapeamento não é tão clara. Se todas as restrições não forem selectivas, sendo que $\forall i, r_i. 2^{[m\,/\,d]}/|\ \Omega_i\ | > 1$, então o segundo mapeamento tem sempre um melhor desempenho que este mapeamento. Contudo, se existir pelo menos uma restrição selectiva, por exemplo, uma restrição de igualdade, este mapeamento mapeia a subscrição só para uma chave. Considere-se também que o segundo mapeamento também pode retornar um grande número de todas as combinações se todas as restrições não forem selectivas. Este mapeamento é menos sensível a subscrições parciais, ou seja, subscrições que especificam restrições para apenas alguns atributos.

Este mapeamento, ao contrário dos outros dois, mapeia os eventos para d chaves, no pior dos casos. Esta desvantagem pode ser significante, se grande parte do trabalho do sistema consiste no encaminhamento de eventos.

2.4.7 Discussão

Criar funções SK e EK não é uma tarefa trivial, mas mesmo assim é mais simples que criar mapeamentos SN e EN. Esta facilidade deve-se ao facto do universo de chaves ser conhecido, *a priori*, por todos os nós. Outra vantagem de se utilizar estes mapeamentos estáticos consiste no facto dos nós não necessitarem de coordenar a sua computação de mapeamentos, nem no arrancar do sistema, nem na presença de alterações dinâmicas na configuração dos nós. A rede de sobreposição utilizada trata, de forma opaca, dos ajustes nos mapeamentos KN e encaminhamentos, de forma apropriada. Em adição a esta facilidade, na configuração automática, a arquitectura apresentada também torna o sistema menos dependente da configuração dos nós. Sendo ainda que a maior parte das redes de sobreposição são simétricas no sentido de não existirem nós com funções específicas, tal como a existência da raiz de uma árvore. Estes dois factores tornam a arquitectura resistente a falhas, porque em caso de falha de um nó, pouca informação é perdida, sendo que esta informação pode ser facilmente replicada por um conjunto de nós.

2.5 Protocolos

Nesta secção do documento são apresentados os protocolos que são frequentemente utilizados em soluções **Message-Oriented Middleware**.

2.5.1 RestMS

O **RestMS** é um serviço de mensagens **RESTful** [41] que disponibiliza uma comunicação assíncrona através duma interface **REST**, utilizando **HTTP**(HyperText Transfer Protocol)/**HTTPS**(HyperText Transfer Protocol Secure). O **RestMS** assenta sobre **RestTL** e é extensível através de especificações de perfis.

Domains

Os *domains* são *namespaces* para *pipes* e *feeds*. Um servidor pode ter vários *domains* e cada um deles pode ter credenciais de segurança diferentes. Um *domain* contém os *profiles* que nele podem ser utilizados

Profiles

Contém um conjunto de semânticas que o servidor implementa, estas semânticas podem definir o tipo de *feeds*, *pipes*, códigos de resposta, utilizações dos objectos definidos.

Feeds

As *feeds* são o ponto de entrada de mensagens no sistema. Vários clientes podem escrever mensagens numa *feed*. A ordem das mensagens das *feeds* deve ser mantida, relativamente a cada cliente. As *feeds* podem ser criadas dinamicamente por clientes.

Pipes

Os *pipes* consistem na forma de recepção de mensagens dos clientes. Sobre os *pipes* apenas se podem realizar leituras e são ordenados por leitor. São criados de forma dinâmica e são privados para os clientes que os criam. O servidor pode limpar *pipes* que não sejam utilizados ou que estejam a ficar sobrecarregados de mensagens.

Joins

Os *joins* definem o encaminhamento que deve ser feito das mensagens recebidas nas *feeds*, para os *pipes* subscritos por clientes. São criados de forma dinâmica e devem sempre ser privados. A criação de *joins* deve ser feita sempre para o URI do *pipe* respectivo. Se a *feed* ou o *pipe*, a que o *join* diz respeito, forem destruídos, o *join* também o será.

Mensagens

As mensagens e conteúdos estão ligados, sendo que uma mensagem é um envelope que pode conter conteúdos, nela embebidos, ou endereços para conteúdos sobre a forma de recursos. Estes recursos são especificados com um tipo **MIME**(Multipurpose Internet Mail Extensions) [42] e são tratados como *blobs*.

A Figura 4 mostra as relações entre entidades acima mencionadas.

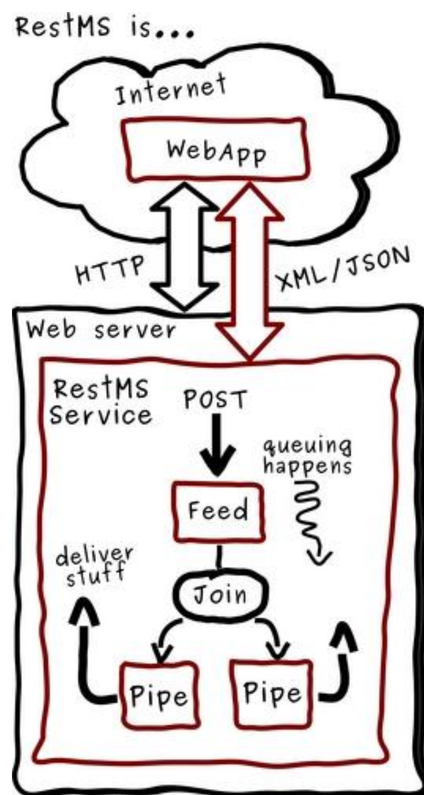


Figura 4 - Arquitectura lógica do protocolo RestMS, imagem retirada de [3]

Publicação:

Os recursos são primeiro publicados para uma *feed*, sendo enviados através de pedidos POST, em que o corpo do POST contém a mensagem e o Content-Type, sendo este um tipo MIME. O servidor responde o endereço **URI**(Uniform Resource Identifier) que deve ser utilizado para referenciar o recurso criado.

Os tipos MIME de conteúdos têm de ser estruturados seguindo a forma: "application/restms+xml", ou "text/xml". Caso assim não aconteça o recurso será reconhecido como uma mensagem.

Atributos adicionais sobre o recurso devem ser especificados nas mensagens que os referenciam.

As mensagens são criadas após todos os recursos, que esta utiliza, estarem criados. A mensagem será enviada para o **URI** da *feed* e deverá conter os recursos que deseje embebidos ou referenciados. Os recursos referenciados têm de ser primeiro criados, da forma que foi descrita anteriormente.

2.5.2 AMQP

Os sistemas de mensagens das empresas permitem que os programas comuniquem através da troca de mensagens, tal como as pessoas comunicam ao trocarem *emails*. Mas ao contrário dos *emails*, os sistemas de mensagens das empresas providenciam garantias de entrega, velocidade, segurança e liberdade de spam. Até ao surgimento do **AMQP**(Advanced Message Queuing Protocol), não havia nenhum *standard* aberto para este tipo de sistemas, portanto os programadores tinham de escrever os seus próprios, ou utilizar sistemas proprietários que são dispendiosos.

O **AMQP** é o primeiro sistema aberto para mensagens de empresas. Foi desenhado para suportar trocas de mensagens para qualquer aplicação distribuída. O encaminhamento pode ser flexivelmente configurável e suporta facilmente paradigmas de mensagens comuns, tais como: ponto-a-ponto, *fanout*, *publish-subscribe* e pedido-resposta.

O **AMQP** define não só a interface, mas também o comportamento dos clientes e servidores, por forma a possibilitar uma verdadeira interoperabilidade entre diferentes fabricantes.

Define o formato das mensagens a serem enviadas através do canal de comunicação.

Opera segundo três ideias:

- Message Queues: Podem dar diferentes tipos de qualidade de serviço, exemplo: garantia de persistência de dados, ou armazenamento apenas em **RAM**(Random Access Memory). Um cliente pode especificar qual a qualidade de serviço que deseja, no acto da criação da *queue*.
- Exchanges: Encaminham as mensagens para *queues*. Têm semânticas: *one-to-one*, *one-to-many*, *one-to-one-of-many*.
- Bindings: Os Bindings são conjuntos de regras que definem o comportamento dos Exchanges, podendo definir que todas as mensagens são encaminhadas para uma *queue*, ou, que o encaminhamento é baseado no conteúdo das mensagens.

As mensagens circulam entre servidores de eventos e clientes, através de métodos assíncronos e síncronos.

Entidades relevantes, do ponto de vista de comunicação:

- Servidor de eventos: Entidade a que os utilizadores se conectam. Pode ser distribuído por várias réplicas organizadas em *cluster* (rede de servidores de eventos).
- Utilizador: Autenticações de acesso ao *middleware*
- Conexão: Ligação física, exemplo: **TCP/IP**
- Canal: Ligação lógica associada a uma conexão, através desta consegue-se manter estado numa comunicação.

2.6 Conceitos envolvidos na implementação de um Message-Oriented Middleware

Nesta secção são discutidos alguns aspectos da implementação de um sistema de integração de aplicações. Entre estes aspectos estão: os eventos, o meio e a qualidade de serviços. Esta discussão é guiada pelas preocupações causadas pela flexibilidade, confiabilidade, escalabilidade e desempenho.

2.6.1 Eventos

Os eventos podem assumir duas formas, sendo estas mensagens ou invocações. As mensagens são eventos entregues a subscritores através de uma operação genérica, por exemplo notificação, enquanto as invocações são eventos que despoletam a execução de operações específicas no subscritor.

2.6.1.1 Mensagens

Ao mais baixo nível, quaisquer dados que circulem pela rede são encapsulados em mensagens. A maior parte dos sistemas implementam as notificações através de mensagens, sendo estas explicitamente criadas pelas aplicações.

As mensagens são compostas por um cabeçalho num formato genérico, que contém informação específica à mensagem, e dados de conteúdo, que contêm informação específica aos utilizadores. O cabeçalho das mensagens tem como campos comuns os seguintes: identificador da mensagem, emissor, prioridade e data de expiração; estes campos podem ser interpretados pelo sistema, ou serem simplesmente informação para os consumidores. O tratamento dos dados de conteúdo duma mensagem varia de sistema para sistema, sendo que existem três tipos de tratamento comuns. O primeiro não faz qualquer interpretação destes dados, considerando-os como um conjunto de bytes. O segundo tratamento encontrado disponibiliza um conjunto de tipos de mensagem, que permite identificar o formato dos dados de conteúdo contidos nesta. Finalmente, o terceiro tratamento encontrado, permite que as mensagens sejam auto descritivas quanto ao seu tipo.

O sistema **TIBCO Rendezvous** define um formato de mensagem que não tem cabeçalho, no entanto, permite que o programador crie a sua própria estrutura de mensagem baseada num conjunto de tipos básicos, que podem ser estruturados hierarquicamente. Sendo que o tipo de mensagem pode ser interrogado em tempo de execução.

Os sistemas **DAC** [43] e **JMS** [44] suportam objectos mensagem, em que o evento pode ser qualquer objecto **Java** seriável.

Muitas vezes as mensagens são vistas como registos com vários campos.

2.6.1.2 Invocações

As invocações são direccionadas a um tipo específico de objecto e têm uma semântica bem definida. O sistema assegura que todos os consumidores têm uma interface compatível para processar a invocação. A interface tem o mesmo propósito que um contracto entre o invocador e os invocados.

Os sistemas que disponibilizam interacções baseadas em invocações, disponibilizando múltiplas semânticas e esquemas de endereçamento, são normalmente chamados sistemas de mensagens. Estes sistemas incorporam lógica adicional aos esquemas *publish/subscribe* ou filas de mensagens, por forma a transformar as mensagens de baixo nível em invocações a métodos dos subscritores, sendo que todos têm de ser do mesmo tipo.

Enquanto alguns sistemas tomam em consideração os valores de retorno das invocações, os modelos *publish/subscribe* baseados em tipos do **COM**+(Component Object Model) [45] e o **CORBA Event Service**, tipicamente apenas consideram invocações num sentido, não existindo resposta do chamado.

Os produtores invocam operações num objecto intermediário, por exemplo um canal de eventos, sendo que este exibe a mesma interface que os consumidores. O objecto intermediário reencaminha os eventos para todos os consumidores registados.

O **COM**+ ainda disponibiliza uma forma de filtragem baseada no conteúdo dos eventos, ao possibilitar que sejam especificados valores para os argumentos de invocação, para restringir as possíveis invocações.

2.6.2 Meio de comunicação

A transmissão de dados entre os produtores e os consumidores é da responsabilidade do meio do *middleware*. O meio pode ser classificado de acordo com características como: arquitectura ou garantias que providencia para os dados, tais como persistência e confiabilidade.

2.6.2.1 Arquitecturas

A arquitectura de um sistema de eventos caracteriza a topologia e a delegação de responsabilidades das diferentes entidades do sistema.

2.6.2.1.1 Centralizada

O papel dos sistemas *publish/subscribe* consiste na disponibilização da troca de eventos entre produtores e consumidores de eventos, de forma assíncrona. O assincronismo pode ser implementado à custa dos produtores enviarem mensagens para uma entidade específica, que os armazena e os encaminha para os consumidores. Esta abordagem é

chamada arquitectura centralizada, por existir uma entidade central que armazena e encaminha as mensagens. As aplicações baseadas neste tipo de sistemas têm fortes requisitos em termos de confiabilidade, consistência de dados e suporte transaccional, não necessitando de um elevado desempenho do meio de comunicação.

2.6.2.1.2 Distribuída

O assincronismo também pode ser implementado ao serem utilizadas primitivas de comunicação inteligentes que implementem mecanismos de armazenamento e encaminhamento, tanto nos processos produtores como nos consumidores, para que a comunicação seja assíncrona e opaca para as aplicações, sem a necessidade da utilização de uma entidade intermediária. Esta abordagem é conhecida como arquitectura distribuída, uma vez que não existe uma entidade central no sistema. Estas arquitecturas estão bem adaptadas para entregas rápidas e eficientes, de dados.

2.6.2.1.3 Servidores distribuídos

Existe uma abordagem intermédia, adoptada por diversos sistemas actuais, que consiste na implementação do serviço de notificações de eventos, como uma rede distribuída de servidores. Em contraste aos sistemas completamente descentralizados, esta abordagem alivia a carga dos processos participantes ao utilizar servidores dedicados a executar os protocolos complexos para a persistência, confiabilidade e alta disponibilidade, tais como a filtragem baseada no conteúdo dos eventos e para o encaminhamento.

No sistema **Gryphon** existe um grafo que sumariza os interesses dos subscritores. Este é imposto ao grafo de servidores de eventos, por forma a evitar comparações redundantes. O sistema **Siena** utiliza encaminhamento das subscrições e da publicação dos eventos para fixar os caminhos das notificações.

Os servidores de eventos mantêm registo de informação relevante para fazer a correspondência de eventos com subscrições de uma forma eficiente.

2.6.2.2 Disseminação

A disseminação caracteriza a forma como a informação é distribuída. Sendo que os dados podem ser enviados utilizando primitivas de comunicação ponto-a-ponto, ou utilizando mecanismos *multicast*, tais como **IP** *multicast*. A escolha do mecanismo de

comunicação a utilizar depende de vários factores, tais como: ambiente de utilização ou arquitectura do sistema.

As abordagens centralizadas, tais como alguns sistemas de filas de mensagens, utilizam primitivas de comunicação ponto-a-ponto entre os produtores/consumidores e o servidor de eventos centralizado. Estes sistemas focam-se em dar garantias, além do alto desempenho de comunicação e escalabilidade.

Os sistemas *publish/subscribe* baseados em tópicos podem beneficiar dos vários estudos feitos sobre comunicação de grupo e dos protocolos resultantes para disseminar eventos para os subscritores.

É garantido um bom desempenho relativo ao número de mensagens da comunicação, ao serem utilizados os protocolos **IP** *multicast* ou um dos muitos protocolos de *multicast*.

A utilização eficiente de eventos *multicast*, em sistemas *publish/subscribe* baseados em conteúdo, é ainda um problema. O desempenho destes sistemas baseados em disseminação é afectado pelo custo da filtragem dos eventos em cada um dos servidores, que depende do número de subscrições no sistema.

Independentemente das técnicas de filtragem empregues, o encaminhamento selectivo de eventos dos sistemas *publish/subscribe* baseados em conteúdo dificulta a exploração das primitivas de *multicast*, ao nível de rede.

2.6.2.3 Qualidade de serviço

As garantias que o meio disponibiliza a todas as mensagens variam entre os diferentes sistemas. As características da qualidade de serviço mais consideradas são: persistência, garantias transaccionais e prioridades.

2.6.2.3.1 Persistência

As mensagens podem ser enviadas sem gerar qualquer resposta, assim como podem ser processadas horas após terem sido enviadas. Os intervenientes da comunicação não controlam como as mensagens são transmitidas, nem como são processadas. Assim, o sistema de mensagens tem de providenciar garantias, não só ao nível de confiabilidade, mas também ao nível de durabilidade da informação.

Não é suficiente saber que uma mensagem chegou ao sistema de mensagens entre os produtores e consumidores, também é necessário ter garantias que a mensagem não será perdida em caso de falhas do sistema de mensagens.

A persistência é normalmente implementada nos sistemas *publish/subscribe* que têm uma arquitectura centralizada e guardam as mensagens até que os consumidores as possam processar.

Os sistemas de filas, tais como **IBM MQSeries** [46] e o **Oracle Advanced Queuing** [47], oferecem persistência utilizando uma base de dados.

Os sistemas *publish/subscribe* distribuídos não costumam oferecer persistência, uma vez que as mensagens são directamente enviadas pelo produtor para todos os subscritores. Apenas nos casos em que os produtores guardam uma cópia de cada mensagem, os subscritores com falhas podem receber mensagens perdidas, após a recuperação.

O sistema **TIBCO Rendezvous** oferece uma aproximação mista, em que um processo está receptivo a um conjunto de tópicos, este processo armazena de forma persistente as mensagens destes tópicos e reenvia mensagens perdidas a subscritores em recuperação de falhas.

O sistema **Cambridge Event Architecture** [48] disponibiliza um repositório de eventos, potencialmente distribuído, para armazenamento e recolha de eventos, eficiente. Este repositório fornece mecanismos que facilitam a procura de eventos simples e compostos, permitindo a resposta de sequências de eventos armazenados.

2.6.2.3.2 Prioridades

Pode ser desejável ordenar, pela sua prioridade, as mensagens à espera para serem processadas pelo consumidor. Por exemplo, um evento real-time pode requerer uma resposta imediata, como por exemplo notificação de falha, e deve ser processado antes das outras mensagens.

As prioridades afectam as mensagens que estão em trânsito, ou seja, não estão a ser processadas. As prioridades de execução são geridas pelo *scheduler* da aplicação e não pelo sistema de mensagens. Este facto leva a que dois subscritores dos mesmos tópicos

possam processar mensagens por ordens diferentes, uma vez que processam mensagens com ritmos diferentes, apesar dos canais de comunicação serem FIFO(First In First Out).

As prioridades apenas devem ser consideradas em ultimo caso, ao contrário da persistência.

A maioria dos sistemas de mensagens *publish/subscribe*, centralizados ou distribuídos, disponibilizam mecanismos de prioridades, apesar de disponibilizarem números diferentes de prioridades e diferirem na forma como as aplicam.

2.6.2.3.3 Transacções

As transacções são geralmente utilizadas para agrupar múltiplas operações em blocos atómicos, que ou são executados na totalidade, ou não são executados de todo.

Em sistemas de mensagens, as transacções são utilizadas para agrupar mensagens em grupos atómicos, em que, ou a totalidade do grupo de mensagens é enviada, ou nenhuma mensagem é enviada. Um caso de utilização de transacções é o seguinte, um produtor publica várias mensagens semanticamente relacionadas e não deseja que os consumidores vejam parcialmente a sequência de mensagens, formando um conjunto inconsistente de mensagens, no caso de falha no envio do mesmo. Semelhantemente, uma aplicação vital para o negócio pode desejar consumir uma ou várias mensagens, processá-las e apenas depois fazer *commit* da transacção. Caso o consumidor falhe antes de fazer o *commit* da transacção, todas as mensagens envolvidas estarão disponíveis para serem reprocessadas, após a recuperação da aplicação.

A forte integração com bases de dados permite que os sistemas **IBM MQSeries** e **Oracle Advanced Queuing** disponibilizem um leque alargado de mecanismos transaccionais. Os sistemas **JMS**(Java Message System) e **TIBCO Rendezvous** também disponibilizam suporte para agregação transaccional de mensagens no contexto de uma sessão. O sistema **JavaSpace**s disponibiliza mecanismos transaccionais simples para garantir a atomicidade da produção e consumo de eventos. Um evento publicado num **JavaSpace**, no contexto de uma transacção, não é visível fora da transacção, até que se faça *commit* da transacção. De igual modo, um evento consumido não é

removido do **JavaSpace**, até que se faça *commit* da transacção que engloba a operação. Vários eventos podem ser produzidos e consumidos no contexto da mesma transacção.

2.6.2.3.4 Confiabilidade

O desacoplamento da sincronização entre produtores e consumidores de informação torna a implementação de propagação confiável de eventos, ou seja, entrega garantida, uma tarefa desafiadora.

Os sistemas *publish/subscribe* centralizados normalmente utilizam canais ponto-a-ponto confiáveis para comunicar com os publicadores e os consumidores, e mantêm cópias dos eventos em locais de armazenamento, estáveis. Sendo que os eventos são entregues de forma confiável a todos os subscritores. Uma falha do servidor centralizado de eventos apenas causa um atraso na entrega.

Sistemas baseados numa rede de sobreposição de servidores de eventos normalmente utilizam protocolos confiáveis para propagar eventos para todos os servidores ou para um subconjunto deles. Podem ser utilizados protocolos baseados em comunicação em grupo e mecanismos de *multicast* confiáveis, de nível aplicacional, em sistemas baseados numa rede de sobreposição de servidores de eventos, uma vez que estes são resistentes a falhas de alguns dos servidores da rede.

Os publicadores e subscritores comunicam com o servidor de eventos mais próximo, geralmente utilizando canais de comunicação ponto-a-ponto.

Sistemas que permitem que os publicadores e os subscritores comuniquem directamente, tais como **TIBCO Rendezvous**, também utilizam protocolos *multicast* confiáveis simples. Uma vez que os eventos ficam residuais no sistema para subscritores que tenham falhado ou se tenham atrasado, existindo a necessidade de desacoplamento temporal, a entrega garantida tem de ser implementada por um processo dedicado a guardar eventos e reenvia-los para os subscritores que os requisitem.

2.7 Soluções existentes

Nesta secção é feita uma descrição de algumas soluções, para integração de *software* empresarial, a serem utilizadas no mercado.

2.7.1 Apache QPID

O **Apache Qpid** é um sistema de mensagens de empresas multiplataforma que implementa o protocolo **AMQP**, providenciando servidores de eventos escritos em **C++** e **Java**, juntamente com clientes para **C++, JMS**, **.Net**, **Python** e **Ruby**.

Ao implementar a última especificação **AMQP**, o **Qpid** providencia gestão de transacções, filas de mensagens, distribuição de mensagens, segurança, *clustering*, federação e suporte heterogéneo para multiplataforma.

O **Apache Qpid** é um esforço conjunto da **Red Hat**, **Iona** e outros para construir *software* que utilize **AMQP**.

O **Apache Qpid** é constituído pelos seguintes componentes: **SocketIo Thread Pool**, **Event Processor Thread Pool** e **Message Pump Thread Pool**, sendo as suas funções as seguintes.

SocketIo Thread Pool:

- Ler informação dos **Sockets** [49]
- **Asynchronous Read Filter** faz passar as mensagens pelo **FilterChain** para serem processadas como eventos, num outro protocolo.

Event Processor Thread Pool:

- Processar eventos de filas associadas a sessões
- Escrever eventos em *buffers* associados a sessões
- Descodificar eventos
- Encaminhar eventos através de **Exchange**
- Criar **Frame** e cabeçalho de conteúdo
- Escrever evento no *buffer* de sessão (envio directo) (Pode ser feita uma conversão de protocolo)
- Colocar evento numa fila para ser posteriormente entregue

Message Pump Thread Pool:

- Armazenamento de eventos, enquanto a fila está em modo de persistência

2.7.2 BizTalk

Nenhuma aplicação é uma "ilha". Ligar sistemas tornou-se a norma, chegando ao ponto de a ligação de várias peças de *software* se tornar mais do que uma simples trocas de bytes. À medida que as organizações se movem na direcção de uma realidade orientada a serviços, o verdadeiro objectivo, criar processos de negócio eficientes que consigam unir diferentes sistemas num todo coerente, torna-se mais atingível.

O **BizTalk** suporta este objectivo, ao permitir a conexão entre diferentes peças de *software*, permitindo ainda a criação e modificação gráfica, da lógica de processamento desse *software*. O **BizTalk** também permite que os processos em execução sejam monitorizados, se interaja com os parceiros de negócio e se realizem outras tarefas orientadas ao negócio

O **BizTalk Server** é a solução de integração e conectividade da **Microsoft**. Providencia uma solução que permite que organizações se conectem, mais facilmente, a diferentes sistemas. Inclui mais de 25 adaptadores para diferentes plataformas e uma infra-estrutura de mensagens robusta.

2.7.2.1 Arquitectura

O **BizTalk** integra uma gama de tecnologias para cumprir as suas funções, sendo que a Figura 5 mostra os principais componentes deste.

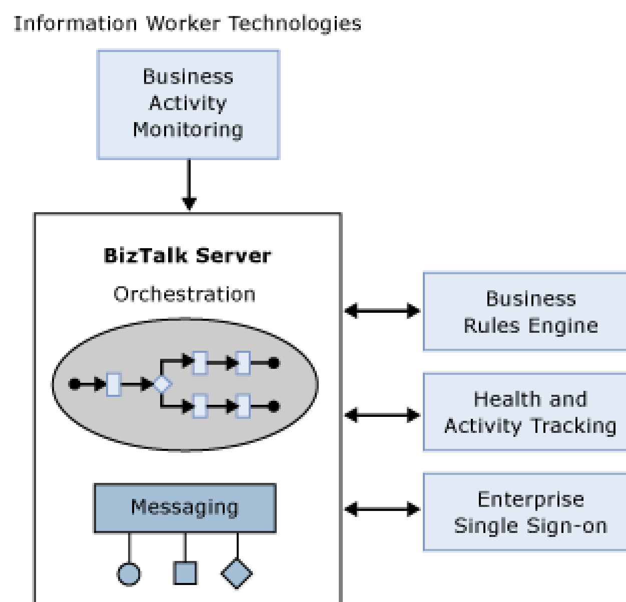


Figura 5 - Arquitectura de BizTalk Server, retirada de [50]

Tal como a figura sugere, o coração do **BizTalk** é o **BizTalk Server Engine**. Este é dividido em dois componentes:

- Um componente de mensagens que disponibiliza a habilidade de comunicar com diversos componentes de *software*. Ao utilizar adaptadores, para os diferentes tipos de comunicação, o **BizTalk Server Engine** consegue suportar vários protocolos e formatos de dados, incluindo *Web Services* [51].
- Um suporte para criar e executar orquestrações, que são processos definidos graficamente. As orquestrações são construídas sobre os componentes de mensagens do **BizTalk Server Engine** e implementam a lógica que coordena toda, ou parte, da lógica de negócio.

Além do **BizTalk Server Engine**, podem ser utilizados outros componentes do **BizTalk**, tais como:

- Um motor de regras de negócio que avalia conjuntos de regras complexas
- Um ponto central que permite aos programadores e administradores monitorizarem e gerirem o **BizTalk Server Engine**, assim como as orquestrações que este executa.
- Um mecanismo de identificação único à empresa, que disponibiliza a possibilidade de mapear a informação de autenticação entre sistemas Windows e não Windows.

O **BizTalk Server** ainda inclui uma ferramenta de monitorização de actividades de negócio (**BAM**(Business Activity Monitoring)), que permite a monitorização de processos de negócio, sendo que a informação é disponibilizada contextualizada ao negócio, em vez de em termos técnicos.

2.7.2.1.1 Comunicação

O **BizTalk** é construído sobre uma arquitectura *publish/subscribe*, baseada no conteúdo dos eventos. As mensagens são publicadas no sistema e são recebidas por um ou mais subscritores activos.

O **BizTalk** torna o processamento de mensagens seguro, ao serializá-las para uma base de dados, enquanto espera por eventos externos, prevenindo-se assim a perda de dados. Esta arquitectura liga o **BizTalk** ao **Microsoft SQL Server**.

2.7.2.2 Adaptadores

O **BizTalk** utiliza adaptadores para comunicações com diferentes protocolos e produtos de *software* específicos, como o **SharePoint** [52]. Alguns dos adaptares incluídos na versão Server 2006 são: **Base EDI**(Covast), **File**, **HTTP**, **FTP**(File Transfer Protocol), **SMTP**(Simple Mail Transfer Protocol), **POP3**(Post Office Protocol 3), **SOAP**(Simple Access Protocol), **SQL**, **MSMQ**(Microsoft Message Queue), **WSE** 2.2 e os adaptadores **WSS**. O adaptador **WCF** [53] foi adicionado na versão 2006 R2.

2.7.3 Distributed Publish/Subscribe (PubSub) Event System

O sistema de eventos **Distributed Publish/Subscribe (PubSub) Event System** é um sistema de eventos *publish/subscribe* distribuído. É distribuído, porque a publicação e subscrição de eventos pode ser intra-máquina ou inter-máquina. Os publicadores e subscritores não necessitam de saber da existência uns dos outros, sendo contudo possível que um publicador possa escutar os eventos de subscrição, para determinar se existem subscritores interessados nos seus eventos, antes da sua publicação.

2.7.3.1 Subscrições

Os eventos são tipificados através da propriedade **EventType**. Esta propriedade é um *GUID*(Globally Unique Identifier) [54] que permite que sejam definidos tipos eventos, sem que exista a preocupação de conflitos com tipos de evento criados por outras utilizações. As aplicações subscrevem tipos de evento, sendo que existe a opção de subscrição de múltiplos tipos de eventos.

As subscrições têm associados identificadores. Na criação de uma subscrição é criado um evento de subscrição, que é publicado pela rede, contendo o respectivo identificador e tipo de evento.

2.7.3.2 Topologia

As máquinas na rede de eventos estão organizadas em hierarquia. Cada máquina tem conhecimento do seu pai e cada pai tem conhecimento dos seus filhos, à medida que estes se conectam. As comunicações entre máquinas são feitas através de **TCP**. A Figura 6 mostra um exemplo desta organização.

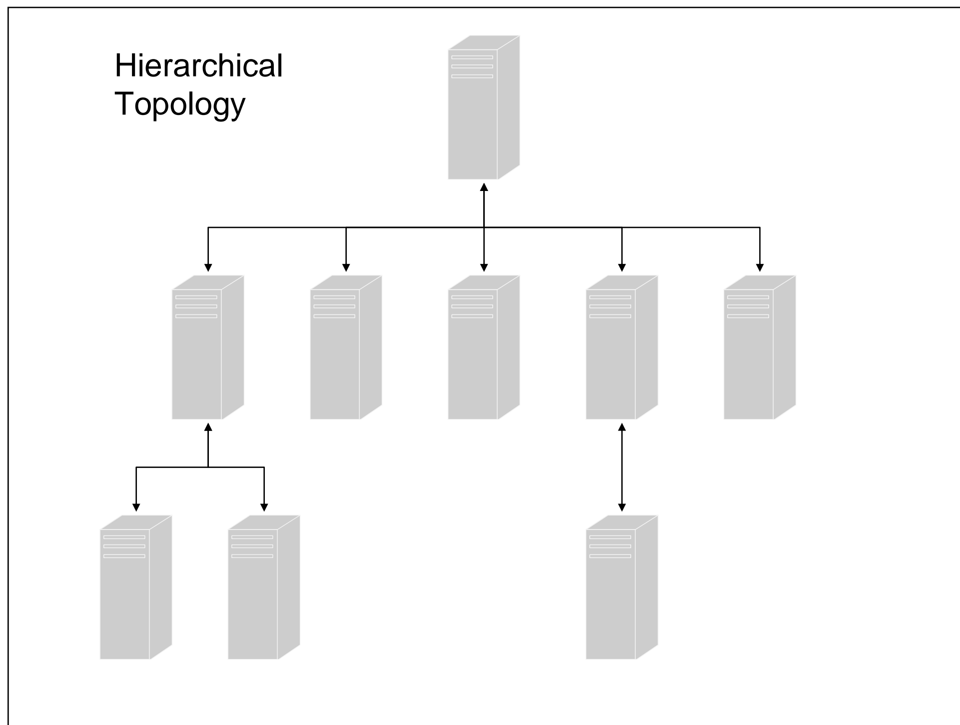


Figura 6 – Exemplo de arquitectura de servidores de eventos, imagem retirada de [55]

Caso os eventos sejam enviados entre irmãos, o encaminhamento é feito através da máquina pai, até ao nó destino. Independentemente do número de subscritores atingíveis através do nó pai, um evento será enviado apenas uma vez da máquina filho para a máquina pai, ou de um pai para um filho. O pai encaminha o evento para o seu pai e/ou seus filhos, caso seja apropriado.

Os servidores de eventos do sistema *Distributed Publish/Subscribe Event System* estão organizados numa estrutura hierárquica, onde os clientes podem-se ligar a qualquer nó.

As subscrições são propagadas no sentido ascendente da árvore de servidores. Esta topologia hierárquica tende a causar uma elevada carga de trabalho nos servidores raiz, e uma falha de um servidor pode desligar toda uma subárvore.

2.7.3.3 Encaminhamento de eventos

As tabelas de encaminhamento são construídas e mantidas dinamicamente, de forma semelhante a como os *routers* de rede mantêm as suas tabelas para *multicast* **IP**. As subscrições são tratadas ao serem enviados eventos através da rede. Cada nó utiliza os eventos de subscrição para manter as suas tabelas de encaminhamento. Quando são publicados eventos numa máquina ou encaminhados para uma máquina, o sistema de eventos encaminha-os através da rede de ligações físicas, onde existem subscrições.

2.7.3.4 Estrutura do sistema

A arquitectura dos servidor de eventos é composta por: **Event Router**, **Publishing Applications**, **Subscribing Applications** e **Shared Memory**. A Figura 7 mostra a mostra as ligações entre estes componentes.

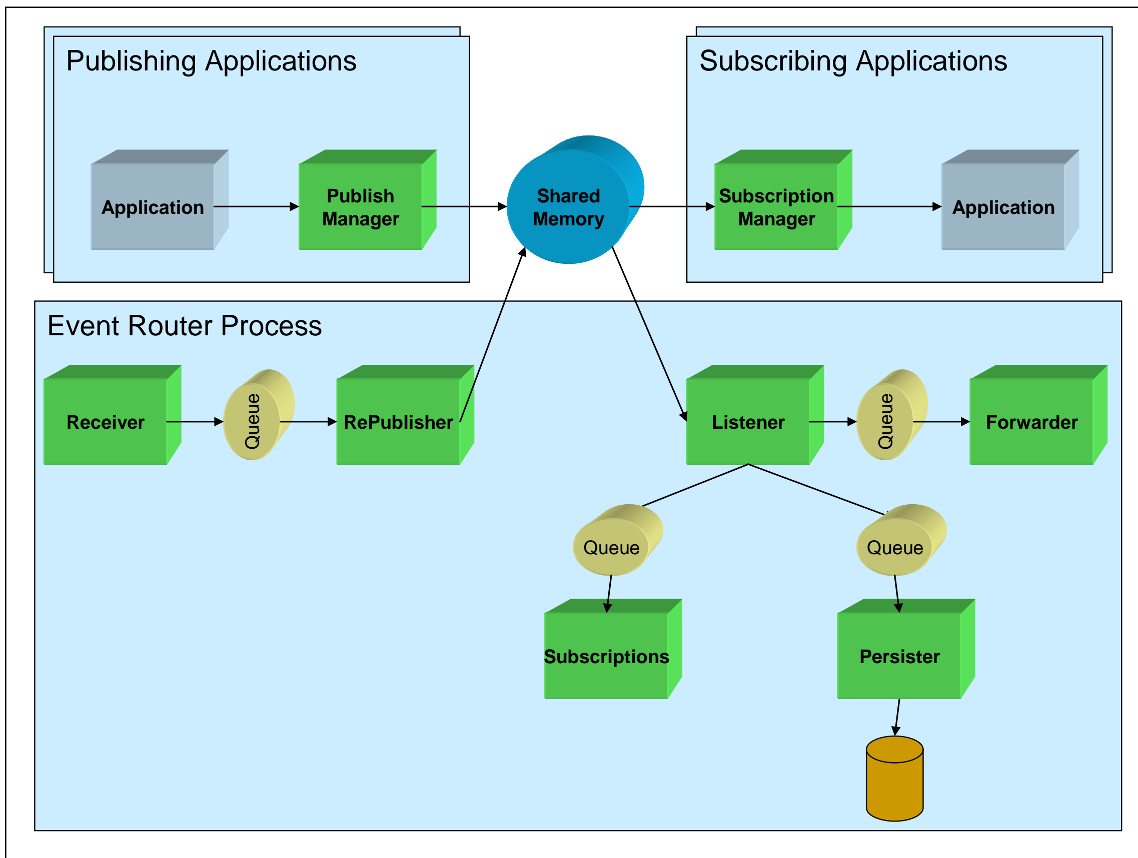


Figura 7 – Arquitectura de sistema ,imagem retirada de [55]

O **Receiver** e **Fowarder** são as interfaces de comunicação para interagir com as máquinas pai e filho. Quando um evento chega ao **Receiver**, este envia o evento para o **RePublisher**, que publica o evento no sistema local. Publicar o evento, neste sentido, significa colocar o evento no *buffer* de memória partilhada. Os subscritores interessados copiam o evento da memória partilhada, tal como é o caso do **Listener**. Este último reencaminha o evento para outros componentes, caso seja apropriado fazê-lo. Caso um evento seja de subscrição, o **Listener** envia-o para **Subscriptions**, para que sejam actualizadas as tabelas de encaminhamento. Se o tipo de evento for para ser persistido, o **Listener** encaminha o evento para o **Persister**. Quando as tabelas de encaminhamento indicam que o evento precisa de ser encaminhado, o **Listener** encaminha o evento para o **Forwarder**, que por sua vez envia o evento para outra máquina.

As aplicações de publicação limitam-se a colocar eventos no *buffer* de memória partilhada. Quando um publicador e subscritor estão na mesma máquina, o evento é colocado na memória partilhada, pelo publicador, e recolhido desta, pelo subscritor.

Nesta situação nunca existem custos acrescidos do evento passar pelo servidor de eventos. Uma vez que podem existir N subscritores à escuta de eventos colocados na memória partilhada, é possível que todos os N subscritores recolham o evento ao mesmo tempo e de forma eficiente.

3 Solução

O projecto de mestrado a que este documento se refere implementa um *middleware* orientado a eventos, com o objectivo de homogeneizar a comunicação entre diferentes clientes, abstraindo-os dos detalhes de comunicação entre estes. Esta solução chama-se Message Integration Bus (MIB) e oferece garantias de entrega de mensagens e processamento transaccional.

3.1 Arquitectura da solução

O MIB é um *middleware* que implementa diversos protocolos, e utiliza um sistema *publish/subscribe*, no seu funcionamento central. O sistema *publish/subscribe* é implementado num servidor de eventos, sendo que este assume uma arquitectura semi-distribuída, com o objectivo de dar maiores garantias de desempenho e escalabilidade à solução.

A arquitectura lógica do MIB consiste em seis entidades, sendo elas:

- ***Cliente***
- ***Subscrições***
- ***Eventos***
- ***Blackbox***
- ***Receptor***
- ***Emissor***

O MIB trabalha com subscrições, que registam o interesse em determinada informação, e eventos, que consistem na informação que o sistema transporta entre clientes.

Os clientes, em determinada altura no tempo, podem ser publicadores de informação ou subscritores, sendo que na primeira situação publicam eventos e na segunda submetem subscrições e esperam por eventos.

O grande problema no desenho da arquitectura desta solução é o facto de o MIB ser um *middleware* com o objectivo de interacção entre diferentes protocolos. Por ser objectivo que o conjunto de protocolos de comunicação disponíveis seja dinâmico, foram criadas as entidades ***Receptor*** e ***Emissor***, sendo que a primeira recebe subscrições e eventos e a segunda emite eventos. Estas entidades implementam protocolos de comunicação e

operam sobre um sistema de *pluggins*, criado para facilitar a extensibilidade dos protocolos implementados pelo MIB.

A entidade central do MIB é o ***Blackbox*** e implementa um sistema *publish/subscribe* baseado no conteúdo, sendo que os ***Receptores*** e ***Emissores*** traduzem o seu modo de operação para o sistema *publish/subscribe*.

A Figura 8 mostra a arquitectura lógica da solução.

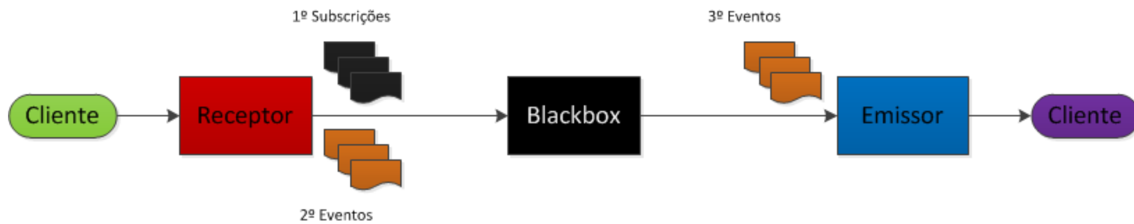


Figura 8 - Arquitectura Lógica MIB

A arquitectura física da solução é apresentada na Figura 9. Tal como a figura mostra, as várias entidades ***Receptor*** e ***Emissor***, assim como a entidade ***Blackbox***, devem ser executadas em diferentes máquinas, por forma a garantir uma melhor escalabilidade.

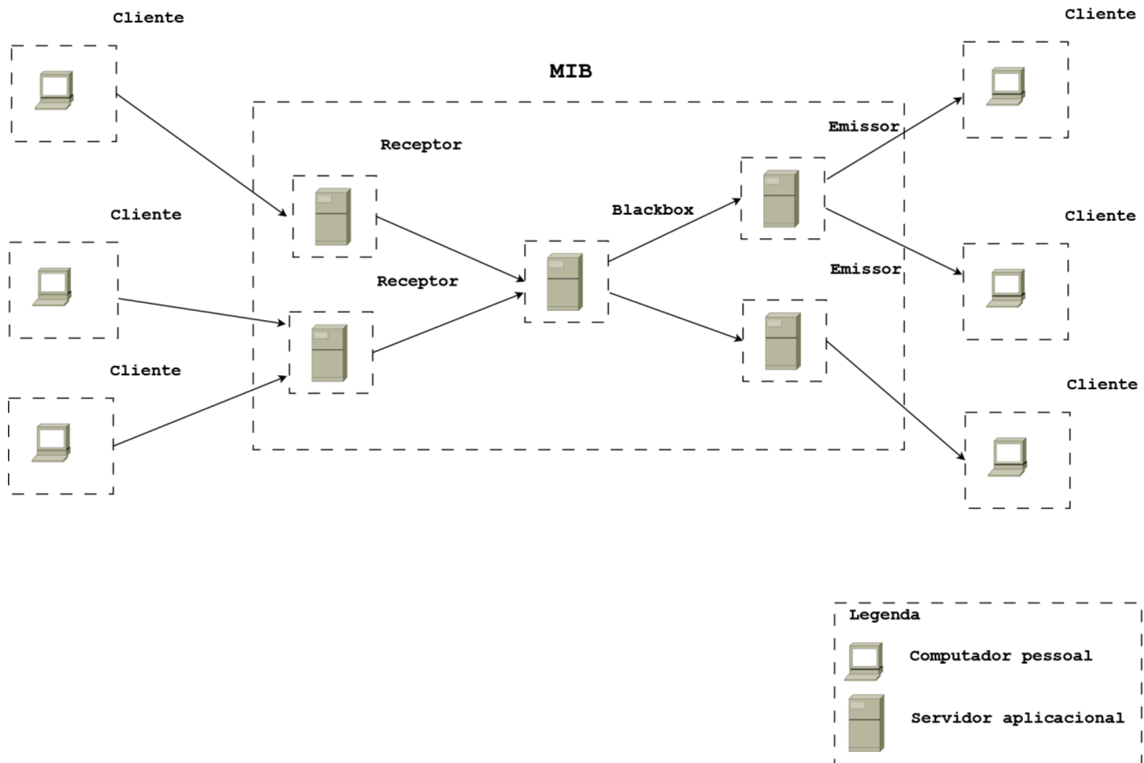


Figura 9 - Arquitectura física MIB

Tal como já foi referido o MIB é um servidor de eventos semi-distribuído, sendo que podem existir várias instâncias do MIB, ligadas entre si, e cada uma delas ligada a vários clientes. A Figura 10 mostra um exemplo destas ligações.

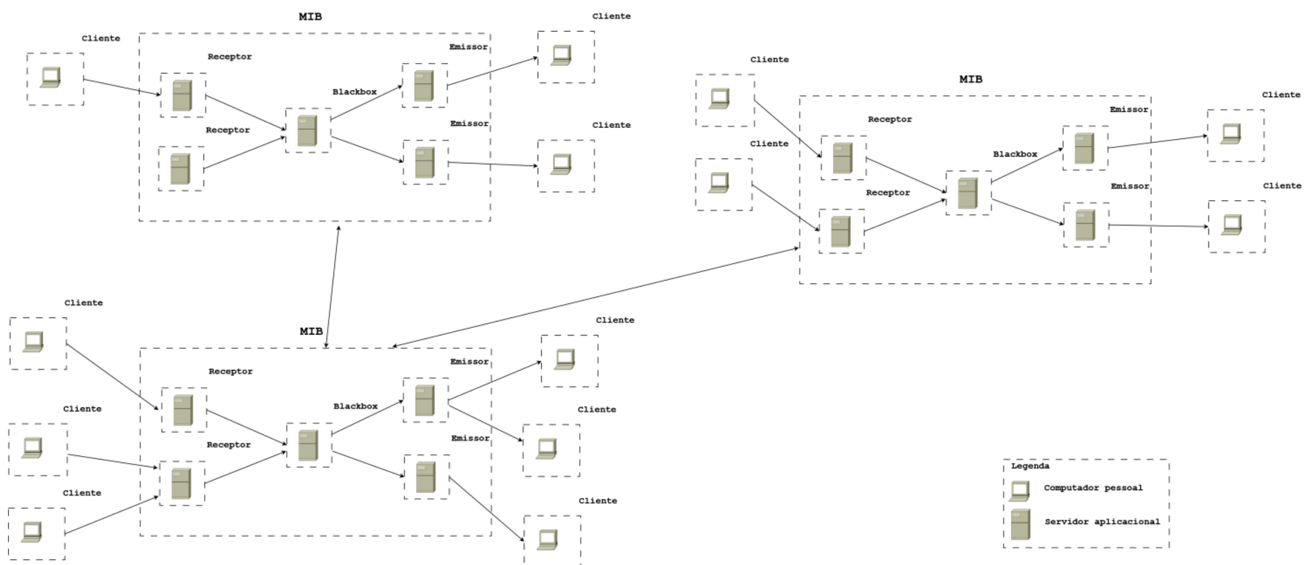


Figura 10 - Arquitectura física de MIB semi-distribuído

As ligações existentes entre cada uma das instâncias de servidores de eventos são conhecidas pelas próprias. Uma vez que um conjunto de ligações entre instâncias de servidores de eventos pode causar um *loop* entre estas, ou pode consistir em caminhos alternativos para uma mesma instância de servidor de eventos, estas ligações podem ser caracterizadas como caminhos possíveis por onde um evento ou subscrição pode ser enviado, ou como redundantes e a não serem utilizadas.

3.2 Aspectos de implementação

A entidade ***Blackbox*,** que executa o funcionamento central do sistema MIB, implementa um sistema *publish/subscribe* baseado no conteúdo, sendo que este tipo de sistemas *publish/subscribe* permite que sejam efectuadas subscrições a eventos, baseadas em descritores de conteúdos de um evento. Este tipo de sistema

publish/subscribe introduz problemas de desempenho por tornar a correspondência entre eventos e subscrições um processo mais dispendioso em termos temporais. A implementação MIB procurou resolver esse problema através da estrutura apresentada na secção 3.2.6 Algoritmo de correspondência de eventos.

Esta implementação também dá garantias de serviço de persistência e transacção, sendo que a primeira permite resistir a certos tipos de falhas e a segunda permite dar a garantia de que uma operação, ou tem sucesso na sua totalidade, ou então nenhuma acção desta é realizada. Dado que estas garantias degradam o desempenho, são fornecidas de modo opcional.

O problema da compatibilidade com diferentes protocolos foi abordado pela implementação de um mecanismo de carregamento de *pluggins*, que é explicado em detalhe na secção 3.2.8 Loader.

A interacção entre clientes a comunicar utilizando diferentes protocolos de comunicação introduz o problema dos eventos num determinado protocolo poderem ter uma semântica e formato diferente de outros protocolos. A solução adoptada para este problema consiste em existirem diferentes representações de eventos, conforme a compatibilidade entre protocolos. A entidade ***Transformador*** foi criada e é responsável por converter as diferentes representações de eventos. Esta entidade é explicada em detalhe na secção 3.2.7

Transformadores.

Os ***Clientes*** comunicam com o sistema MIB utilizando adaptadores específicos a protocolos, estes adaptadores comunicam com os ***Receptores*** e ***Emissores***, específicos ao protocolo utilizado na comunicação.

Neste projecto partiu-se do pressuposto que o MIB é utilizado num ambiente controlado, por exemplo uma rede local com controlo de acessos próprio, e portanto os aspectos de segurança relacionados com a autenticação e autorização são delegados no controlo de acessos, sendo que todos os clientes com acesso ao MIB são válidos e apenas efectuam as operações para as quais são autorizados. Decidiu-se tomar este aspecto em consideração por o foco do trabalho não serem mecanismos de autenticação

e autorização e portanto não ser o objectivo de estudo, outro factor que levou a esta tomada de decisão foi o facto do tempo adicional necessário a despender na sua implementação. Sugere-se uma implementação destes sistemas como um possível trabalho futuro.

3.2.1 Comunicação com Blackbox

A entidade ***Blackbox*** implementa um sistema *publish/subscribe* que recebe e trata de eventos e subscrições. Esta entidade, além de receber eventos e subscrições de clientes locais, também recebe encaminhamentos de subscrições e eventos remotos, de outras instâncias de ***Blackbox***.

Foi criada a classe ***Operation***, que encapsula a propriedade comum a todas as mensagens trocadas com uma instância de ***Blackbox***. A propriedade referida é ***propagationToken***, que diz respeito ao processamento transaccional que será explicado em detalhe na secção 3.2.4 Transacções.

3.2.1.1 Eventos e subscrições

A comunicação entre a entidade ***Blackbox*** e ***Receptores*** e ***Emissores*** é feita através de eventos e subscrições. Existem características comuns às mensagens que transportam eventos e subscrições, portanto foi criada a classe ***PubSubOperation*** para concentrar estes aspectos comuns entre estes dois tipos de mensagens. A Figura 11 mostra o diagrama UML(Unified Modeling Language) de classes da hierarquia dos tipos ***Operation***, ***PubSubOperation***, ***Event*** e ***Subscription***.

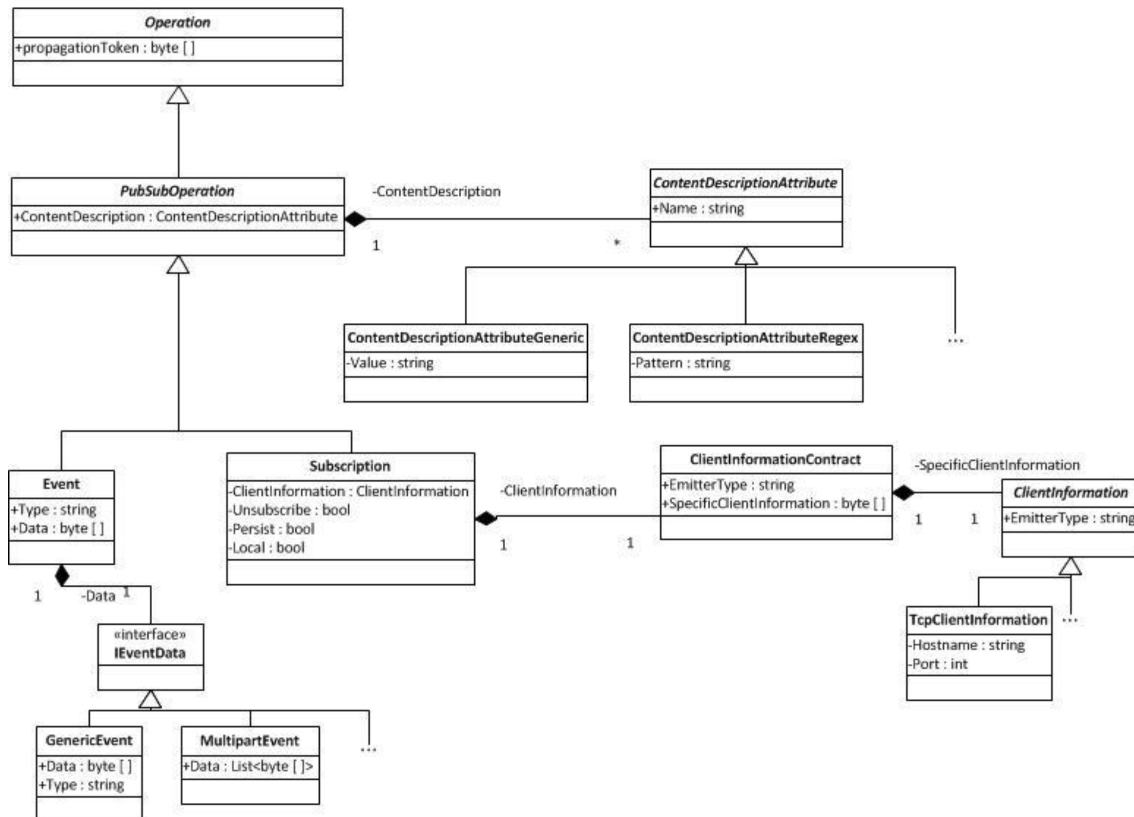


Figura 11 - Diagrama UML de classes de Operation, PubSubContract, Event e Subscription

Tal como a Figura 11 mostra, os eventos e subscrições são caracterizados por um conjunto de descritores de conteúdo, sendo que no caso de eventos, estes descrevem os eventos, e no caso de subscrições, estes descrevem as características procuradas nos eventos a subscrever.

3.2.1.1.1 Descritores de conteúdo

Os descritores de conteúdo são representados por derivados da classe ***ContentDescriptionAttribute***. Esta classe pode ser estendida para implementar um novo tipo de descritor de conteúdo. Os tipos de descritores de conteúdo disponibilizados são ***ContentDescriptionAttributeGeneric***, que representa um descritor textual simples, e ***ContentDescriptionAttributeRegex***, que representa um descritor de expressões regulares.

3.2.1.1.2 Subscrições

As subscrições contêm várias informações, concretamente:

- Se representa uma subscrição ou uma anulação de subscrição, ***Unsubscribe***

- Se requer a persistência dos eventos correspondidos, ***Persist***
- Se é uma subscrição local ou representa apenas a informação de encaminhamento de eventos, ***Local***
- Informação do cliente que criou a subscrição, sendo que esta informação é concreta a cada protocolo e é encapsulada numa concretização de ***ClientInformation***. Esta é por sua vez seriada e encapsulada numa instância de ***ClientInformationContract***, concretamente no campo ***SpecificClientInformation***. Toda esta estrutura foi criada para que a entidade ***Blackbox*** possa ser agnóstica quanto aos diferentes protocolos implementados e que apenas os ***Receptores*** e ***Emissores*** de cada protocolo, trabalhem com o tipo concreto que representa a informação do cliente.

3.2.1.1.3 Eventos

Os eventos são caracterizados pelo protocolo a que são específicos, campo ***Type***, e pelo resultado da seriação de uma concretização de ***IEventData***, que representa um evento no formato do protocolo do evento, que é armazenado no campo ***Data***.

3.2.1.2 Encaminhamento de subscrições e eventos

A comunicação entre as várias instâncias de ***Blackbox*** é feita com o objectivo de encaminhar subscrições e eventos entre estas. Dado que existe informação comum entre o encaminhamento de subscrições e eventos, foi criada a classe ***Interbroker***, que contém o campo ***FromLinkName***, que consiste na informação comum entre estes tipos de encaminhamento. A Figura 12 mostra o diagrama das classes ***Operation***, ***Interbroker***, ***Routing*** e ***Forwarding***.

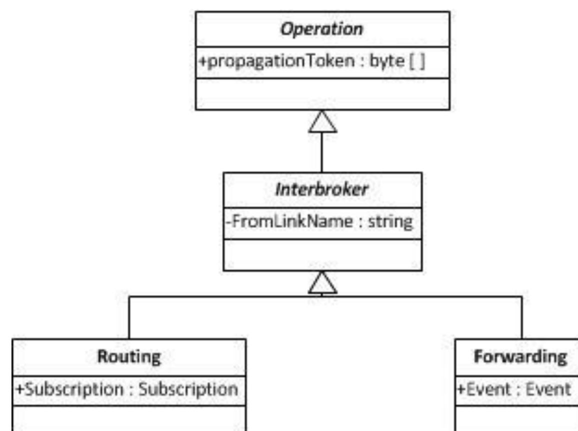


Figura 12 - Diagrama UML de classes das classes Operation, Interbroker, Routing e Forwarding

A classe ***Routing*** representa o encaminhamento de uma subscrição, sendo que contém a subscrição encaminhada. A classe ***Forwarding*** representa o encaminhamento de um evento, contendo o evento encaminhado.

3.2.2 Arquitectura semi-distribuída

O MIB assume uma arquitectura semi-distribuída, em que a lógica do servidor é distribuída por diversas instâncias MIB, ao invés de assumir uma arquitectura centralizada, em que a lógica do servidor é concentrada numa única instância MIB, ou por outro lado, o caso de uma arquitectura distribuída, em que a lógica do servidor é distribuída pelos vários clientes, sendo que estes passam a ser ao mesmo tempo clientes e partes do servidor. A arquitectura semi-distribuída foi escolhida em detrimento da distribuída por não ser objectivo que os clientes do MIB assumam os custos operacionais do *middleware* de comunicação. A arquitectura centralizada não foi escolhida por concentrar a lógica do *middleware* de comunicação numa única instância do MIB, ou seja, numa única máquina, o que impossibilita a utilização de várias máquinas, tornando-se, desta forma, o *middleware* menos escalável.

A lógica do servidor é implementada pela entidade ***Blackbox***, esta implementa um servidor de eventos com base no paradigma *publish/subscribe* orientado ao conteúdo. A arquitectura semi-distribuída define que a lógica do servidor deve ser distribuída por diversas instâncias ***Blackbox***, sendo que os clientes comunicam com uma destas instâncias, com o objectivo de comunicar com outros clientes. Os vários clientes comunicam com uma das instâncias ***Blackbox*** e estas realizam as operações necessárias, de forma opaca para o cliente, para que a comunicação do cliente atinja os clientes destino, independentemente destes estarem ligados à mesma instância ***Blackbox*** ou a uma instância remota.

3.2.2.1 Esquema de comunicação

As instâncias ***Blackbox*** formam uma rede de servidores de eventos, sendo que cada instância tem conhecimento das instâncias que estão ligadas a si e portanto que pode contactar. Quando uma nova instância ***Blackbox*** é adicionada, apenas as instâncias que vão poder comunicar com esta devem ser reconfiguradas por forma a terem conhecimento da nova instância.

A Figura 13 mostra um exemplo de uma rede de instâncias ***Blackbox***.

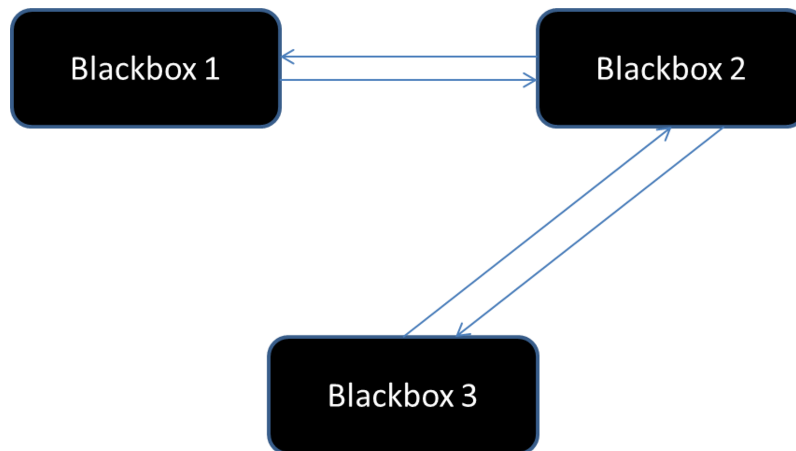


Figura 13 - Exemplo de uma rede de instâncias Blackbox

As subscrições são conhecidas por todas as instâncias ***Blackbox***, para que na recepção de um evento, seja possível saber se este satisfaz alguma subscrição do sistema e desta forma ser encaminhado para os clientes subscritores. A recepção de uma subscrição despoleta a propagação desta para todas as instâncias ***Blackbox*** alcançáveis. Caso a subscrição seja enviada por um cliente local, é registada como uma subscrição local e reencaminhada para as instâncias alcançáveis. Caso a subscrição seja um reencaminhamento, deve ser registada como remota e registada, na máscara da subscrição, a informação da instância que encaminhou a subscrição.

Quando uma instância ***Blackbox*** recebe um evento deve consultar quais são as subscrições, de clientes remotos, que correspondem ao evento recebido. A partir das máscaras, das subscrições correspondidas, que indicam quais as instâncias ***Blackbox*** que devem receber o evento correspondido, é construída uma máscara que indica as ligações a serem utilizadas para encaminhar o evento. Desta forma cada instância ***Blackbox*** apenas necessita de ter conhecimento de quais as ligações a serem utilizadas para encaminhar os eventos, sem que seja necessário ter conhecimento de todos os clientes do MIB.

A comunicação entre instâncias ***Blackbox*** é levada a cabo por instâncias das entidades ***Emissor*** e ***Receptor***, respectivamente ***BlackboxEmitter*** e ***BlackboxReceiver***, sendo que

a primeira é responsável por emitir informação para outra instância ***Blackbox***, e a segunda por receber informação de outras instâncias ***Blackbox***.

O esquema de comunicação entre instâncias ***Blackbox*** é expresso na Figura 14.

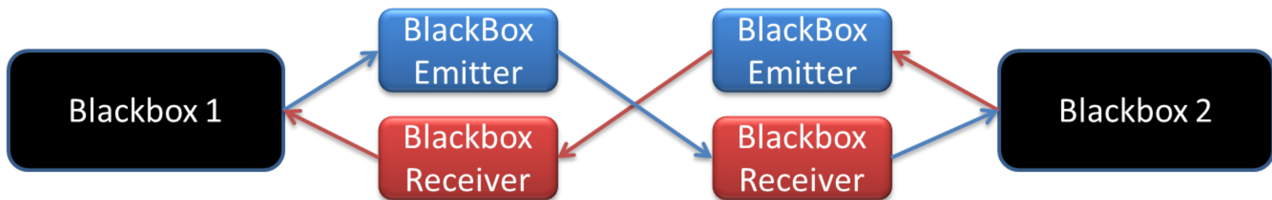


Figura 14 - Esquema de comunicação entre instâncias Blackbox

Tal como a figura mostra, quando uma instância ***Blackbox*** deseja emitir informação para outra instância ***Blackbox***, deve recorrer a ***BlackboxEmitter***. Quando uma instância ***Blackbox*** recebe informação de outra instância ***Blackbox***, esta informação é recebida e interpretada pela implementação da entidade ***Receptor*** ***BlackboxReceiver***. A comunicação entre ***BlackboxEmitter*** e ***BlackboxReceiver*** é realizada utilizando o protocolo **TCP**. A comunicação entre instâncias ***Blackbox*** é realizada por implementações das entidades ***Emissor*** e ***Receptor***, portanto podem ser criadas novas implementações destas entidades, para que se realize a comunicações entre instâncias ***Blackbox*** perante outros moldes, basta para isto criar um *pluggin* que implemente a entidade ***Emissor*** ou ***Receptor***, e definir o tipo destas comunicações, no caso do ***Emissor***. A definição do tipo de um ***Emissor*** é feita no campo ***Type***, da interface ***ILaunchableEmitter***, sendo este do tipo *string*, e o valor do tipo de comunicações entre instâncias ***Blackbox*** é a constante *INTERBROKER*.

As instâncias ***Blackbox*** trocam mensagens entre si com dois objectivos, sendo estes, propagar subscrições por toda a rede de instâncias ***Blackbox*** e encaminhar eventos para os clientes que os subscrevem. Estas mensagens são codificadas em instâncias das classes ***Routing*** e ***Forwarding***, respectivamente.

3.2.2.2 Persistência

A comunicação entre as instâncias ***Blackbox*** não pode ter perdas de informação, pois não é aceitável que perante uma falha se perca uma subscrição ou um evento, dado que esta perda pode fazer com que um cliente deixe de receber um ou mais eventos e desta forma o funcionamento do MIB seja comprometido. Esta garantia é sempre utilizada para que o correcto funcionamento do MIB seja sempre assegurado.

A persistência no envio de informação entre instâncias ***Blackbox*** permite que numa falha de uma instância, a informação que lhe é destinada seja armazenada, sendo que desta forma se evita a perda de informação.

3.2.2.3 Transacções

O processamento transaccional entre instâncias ***Blackbox*** permite a confirmação da recepção e armazenamento da informação, por parte da instância receptora da comunicação, desta forma garante-se que a entidade emissora tem uma confirmação e pode apagar a informação enviada, para que caso exista uma falha após a recepção, no processo de persistência, não seja perdida a informação propagada. Esta garantia é obrigatória por forma a garantir o correcto funcionamento do MIB. O processamento transaccional permite garantir que, ou todas as acções nas comunicações são realizadas, ou nenhuma acção é realizada.

3.2.2.4 Reenvio de informação

As instâncias ***Blackbox*** estão sujeitas a falhas, como qualquer sistema informático, estas falhas não podem provocar perdas de informação transmitida entre instâncias ***Blackbox***, pois estas perdas poderiam representar falhas no correcto funcionamento do MIB. A forma encontrada para evitar estas falhas consiste na utilização das garantias de processamento transaccional e persistência. Desta forma a informação a ser propagada fica armazenada no emissor até que o receptor o notifique da correcta recepção e efectivação dos dados recebidos.

A ocorrência de uma falha leva à utilização dos dois mecanismos de garantias, ou seja, a mensagem trocada entre instâncias é persistida no emissor, e o receptor, ou não se encontra disponível, ou de facto recebe a mensagem mas não tem condições para assegurar a sua efectivação com sucesso e portanto o mecanismo de garantia de

processamento transaccional também falha. Após ocorrer uma falha, a informação que devia ter sido propagada é armazenada no emissor. Esta informação é utilizada quando é detectado o correcto funcionamento da instância destino da comunicação falhada.

A notificação do correcto funcionamento de uma instância ***Blackbox*** dá-se no momento em que essa instância comunica, ou quando uma outra instância comunica com ela e a comunicação ocorre com sucesso. Após ser detectado o correcto funcionamento de uma instância ***Blackbox***, a instância que detecta este facto tenta reenviar todas as mensagens que estão armazenadas e que têm como destino a instância que passou a funcionar correctamente.

3.2.3 Pipeline

O ***Pipeline*** consiste num conjunto, ordenado, de acções a realizar, com o objectivo de tratar a recepção de um evento, uma subscrição, ou o encaminhamento de um evento ou subscrição, recebido de outra instância ***Blackbox***, na entidade ***Blackbox***.

Este conjunto de acções é dividido em cinco subconjuntos, sendo eles:

- Geral
- Evento
- Subscrição
- Encaminhamento de Subscrição
- Encaminhamento de Evento

O subconjunto de acções Geral é aplicado a qualquer tipo de operação recebida e é constituído pelas seguintes acções:

- Autenticação
- Autorização

Tal como foi explicado no capítulo Arquitectura da solução, o MIB não contempla os aspectos de segurança relacionados com autenticação nem autorização, sendo que desta forma estas acções não são implementadas. No futuro, quando se contemplarem os aspectos de segurança acima nomeados, estes serão os pontos de alteração para introduzir os mecanismos de segurança no ***Pipeline*** do MIB.

O subconjunto de acções Evento é aplicado ao tratamento da recepção de eventos e é constituído pelas seguintes acções:

- Correspondência: faz a correspondência entre o evento e as subscrições armazenadas no sistema.
- Persistência: persiste todos os eventos que sejam correspondidos a subscrições, e que assim o exijam, desde que estas sejam locais ao servidor de eventos corrente.
- Reenvio: consiste no envio do evento através de uma ligação que diga respeito a um servidor de eventos que também tenha subscrições para o evento a ser tratado.
- Envio: consiste no envio do evento para os clientes locais.

O subconjunto de acções Subscrição é aplicado ao tratamento da recepção de subscrições e é constituído pelas seguintes acções:

- Subscrição, que consiste no processo de armazenamento da subscrição, eliminação desta, ou ainda o resumo da actividade de uma subscrição já existente no sistema.
- Encaminhamento, que consiste no envio da subscrição para outras instâncias do servidor de eventos, por forma a garantir que estes tomam conhecimento da existência da subscrição, na corrente instância.

O subconjunto de acções Encaminhamento de Evento é aplicado ao tratamento da recepção de um encaminhamento, por parte de outra instância ***Blackbox***, de um evento, este subconjunto é constituído pelas seguintes acções:

- Persistência: salvar a mensagem de encaminhamento de evento.
- Correspondência: faz a correspondência entre o evento e as subscrições armazenadas no sistema.
- Registar interesse: salva a correspondência entre o evento encaminhado e as subscrições locais.
- Encaminhamento: reencaminhamento do evento para as instâncias ***Blackbox*** que o devem receber.
- Envio: consiste no envio do evento para os clientes locais.
- Reenvio: consiste no envio de informação armazenada com destino à instância de ***Blackbox*,** emissora do encaminhamento a ser tratado.

O subconjunto de acções Encaminhamento de Subscrição é aplicado ao tratamento da recepção de um encaminhamento, por parte de outra instância **Blackbox**, de uma subscrição, este subconjunto é constituído pelas seguintes acções:

- Persistência: salvar a mensagem de encaminhamento de subscrição.
- Subscrição: consiste no processo de armazenamento da subscrição, eliminação desta, ou ainda o resumo da actividade de uma subscrição já existente no sistema.
- Encaminhamento: reencaminhamento da subscrição para todas as instâncias ***Blackbox*** que a devem receber.
- Reenvio: consiste no envio de informação armazenada com destino a instância de ***Blackbox*** emissora do encaminhamento a ser tratado.

3.2.4 Transacções

As transacções devem poder ser partilhadas entre os clientes e o ***Blackbox***, assim como entre diferentes instâncias de ***Blackbox***, por forma a possibilitar o processamento transaccional entre os clientes e o ***Blackbox***, e entre várias instâncias de ***Blackbox***. A partilha de transacções entre diferentes computadores é conseguida através de transacções distribuídas. Sendo a transacção iniciada num computador, é necessário que outro possa obter uma referência para essa transacção e utilizá-la para se alistar na transacção distribuída. A passagem da referência da transacção poderia ser feita utilizando o suporte fornecido pelo *namespace* **System.Transactions** do .Net 2.0, mas uma vez que o MIB é um *middleware* e que é desejável que suporte a comunicação com clientes que utilizem diferentes plataformas, inclusive plataformas não .NET, torna-se imperativa a utilização de técnicas não especificas para .NET.

A forma mais genérica encontrada para transmitir referências de transacções foi a utilização do **propagationToken**, inserido no campo ***propagationToken*** de ***OperationContract***. Este é obtido através do método ***GetTransmitterPropagationToken***, da classe **TransactionInterop**, e consiste num *array* de bytes, resultado da seriação de uma transacção **DTC**, que permite transmitir uma referência para uma transacção distribuída. Desta forma uma transacção pode ser seriada, utilizando este método, num computador e obtida noutro, através do método ***GetTransactionFromTransmitterPropagationToken***, da mesma classe, ou equivalentes noutra linguagem de programação.

3.2.4.1 Transacções entre clientes e Blackbox

As transacções entre clientes e ***Blackbox*** assumem um carácter opcional, pelo facto destas tornarem a comunicação mais dispendiosa, a nível de processamento e de tempo de execução. Um cliente pode não enviar uma transacção num evento ou numa subscrição, sendo que desta forma não será integrado numa transacção do ***Blackbox*** e da mesma forma, se um cliente não se registar na transacção do ***Blackbox***, não será integrado nesta.

As transacções são criadas no cliente e transportadas para o Blackbox, falando no sentido de emissores de eventos, e do ***Blackbox*** para o cliente, falando no sentido de receptores de eventos. Os emissores enviam as transacções e estas são *committed* no ***Blackbox***, no momento em que o evento ou a subscrição são persistidos na base de dados do MIB. As transacções no sentido dos receptores de eventos são enviadas para o cliente, pelo ***Blackbox***, caso o cliente subscreva a transacção, esta tem de ser *committed* para que o evento seja considerado recebido e assim eliminado da base de dados.

3.2.4.2 Transacções entre instâncias Blackbox

As transacções entre instâncias ***Blackbox*** assumem carácter obrigatório, uma vez que são trocados eventos e subscrições, entre as diferentes instâncias ***Blackbox***, que não se podem perder em caso de falha de um dos intervenientes na comunicação. O processamento transaccional ocorre entre o envio do encaminhamento, quer seja de evento ou subscrição, e a persistência deste no ***Blackbox*** destinatário.

3.2.5 Persistência

A persistência é um aspecto fundamental, uma vez que permite disponibilizar uma garantia de serviço bastante útil, sendo que em caso de falha da parte do cliente ou do servidor, caso o evento ou subscrição seja persistido, a actividade pode ser resumida sem perdas de dados.

Quando um cliente falha e resume a sua actividade, subscrevendo a informação que tinha subscrevido antes de ter falhado, o servidor de eventos detecta este facto e caso existam eventos por transmitir ao cliente, estes são enviados.

A persistência é uma funcionalidade fundamental mas é disponibilizada de forma opcional, sendo esta escolha feita na subscrição. Esta funcionalidade é opcional por a

sua utilização tornar a comunicação mais demorada e mais dispendiosa em termos de memória.

A persistência também é utilizada na comunicação entre instâncias ***Blackbox***, por forma a possibilitar a entrega de encaminhamentos de eventos ou subscrições, sem perdas de informação.

3.2.5.1 MIB

A persistência dos dados da entidade central do MIB, ***Blackbox***, é feita de um modo simplista, ou seja, os dados armazenados consistem apenas no *hash* do objecto e na seriação do respectivo. Este modo de persistência simplista foi escolhido, por não existir a necessidade de aceder aos detalhes da informação armazenada para consulta, uma vez que esta é persistida apenas com o objectivo de dar garantias de serviço, ao possibilitar a recuperação de informação em casos de falha do servidor de eventos ou dos clientes.

No início da operação de um ***Blackbox***, todas as subscrições persistidas na base de dados são carregadas para memória, sendo que desta forma é possível o seu acesso sem aceder à base de dados. A alteração a nível das subscrições é reflectida na base de dados, para que os dados estejam sempre consistentes. A Figura 15 mostra o modelo da base de dados de suporte à persistência do MIB.

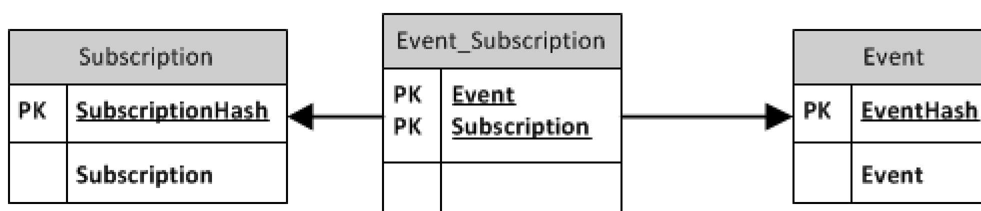


Figura 15 - Modelo de dados MIB

3.2.5.2 Comunicação entre Blackbox

A persistência dos dados para a comunicação entre instâncias ***Blackbox*** é efectuada com o objectivo de possibilitar o reenvio de encaminhamentos de subscrições ou eventos, cujo envio prévio tenha falhado.

A informação persistida consiste nas instâncias ***Blackbox*** ligadas à instância corrente, encaminhamentos de eventos e subscrições, assim como a necessidade de entrega de um encaminhamento a uma dada instância de ***Blackbox***.

A Figura 16 mostra o modelo da base de dados de suporte à persistência da comunicação entre instâncias ***Blackbox***.

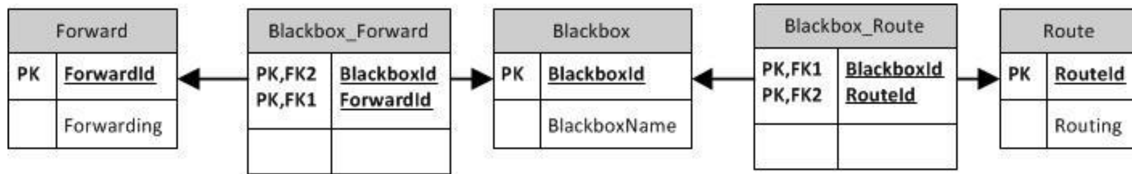


Figura 16 - Modelo de dados de comunicação entre Blackbox

3.2.5.3 Pluggins

Os ***Emissores*** e ***Receptores*** são considerados *pluggins*, portanto a sua informação é isolada da do ***Blackbox***, pois desta forma consegue-se o isolamento da informação do servidor de eventos, por forma a evitar a corrupção desta por *pluggins* defeituosos. A informação de persistência dos *pluggins* deve ser armazenada em bases de dados próprias, hospedadas no mesmo **SGBD**(Sistema de Gestão de Base de Dados) que o que hospeda a base de dados da entidade ***Blackbox***.

3.2.6 Algoritmo de correspondência de eventos

A correspondência de eventos baseada no conteúdo introduz problemas de desempenho, uma vez que a correspondência se faz ao nível de vários atributos e estes são dinâmicos, ou seja, não existe um conjunto de atributos possível ou sequer domínio de valores possível para estes.

A solução apresentada para uma correspondência de eventos baseada no conteúdo destes, com bom desempenho, baseia-se na ordenação e organização de subscrições numa estrutura de dados **PST**. Nesta estrutura cada subscrição corresponde a um caminho desde o nó raiz até a um nó folha. Cada nó da estrutura contém um atributo e um valor. O processo de correspondência começa no nó raiz e consiste no percorrer de todos os caminhos que o evento satisfaça, até aos nós folha. Esta estrutura escala bem horizontalmente, uma vez que explora os aspectos em comum entre as várias subscrições, ao todas estas corresponderem a caminhos iniciais comuns desde o nó raiz.

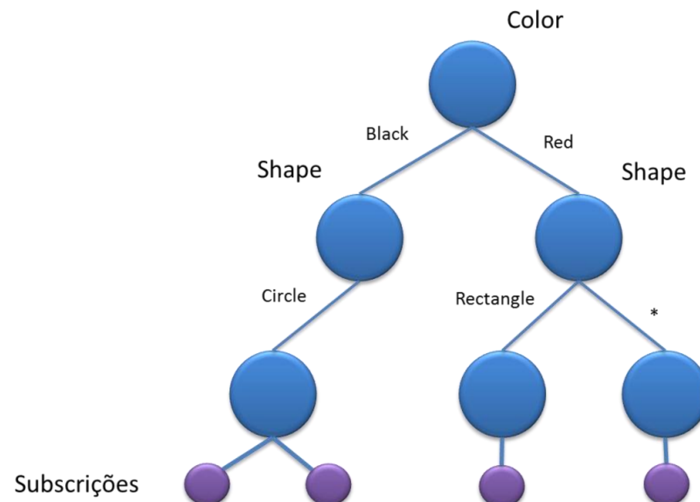


Figura 17 – Exemplo de uma PST

A Figura 17 mostra um exemplo de uma **PST** em que estão armazenadas quatro subscrições, sendo que duas estão igualmente anotadas com os descritores de contexto Color, com o valor Black, e Shape, com o valor Circle. A terceira subscrição está anotada com os descritores de contexto Color, com o valor Red, e Shape, com o valor Rectangle. A quarta subscrição está anotada com o descritor de conteúdo Color, com o valor Red.

A correspondência começa na raiz, tomando em consideração um atributo a1. Em cada um dos nós não folha, encontra-se o valor vj, do evento, correspondente a um atributo aj, em consideração no nó, e percorre-se a ligação do nó que tenha associado o valor igual a vj, caso exista. Caso nenhuma das ligações do nó não folha, corrente, tenha o valor vj, então é percorrida a ligação que assume o valor *, significando *don't care* para o atributo aj. Inicia-se o processo de correspondência em cada um dos nós atingidos pelas ligações percorridas. Quando alguma das pesquisas atinge um nó folha, as subscrições ligadas a este são acrescentadas à lista de subscrições com correspondência à pesquisa efectuada.

3.2.6.1 Implementação

A implementação deste algoritmo de correspondência é realizada nas classes ***SubscriptionTree***, ***NodeConditionBase*** e seus derivados, e ***NodeLeaf***. A classe ***SubscriptionTree*** corresponde à estrutura que contém as subscrições e é constituída por instâncias de ***NodeConditionBase*** e ***NodeLeaf***. A primeira corresponde a nós não folha

e a segunda a nós folha, que contêm subscrições correspondentes ao caminho desde a raiz.

A Figura 18 mostra o diagrama UML de classes da ***SubscriptionTree***.

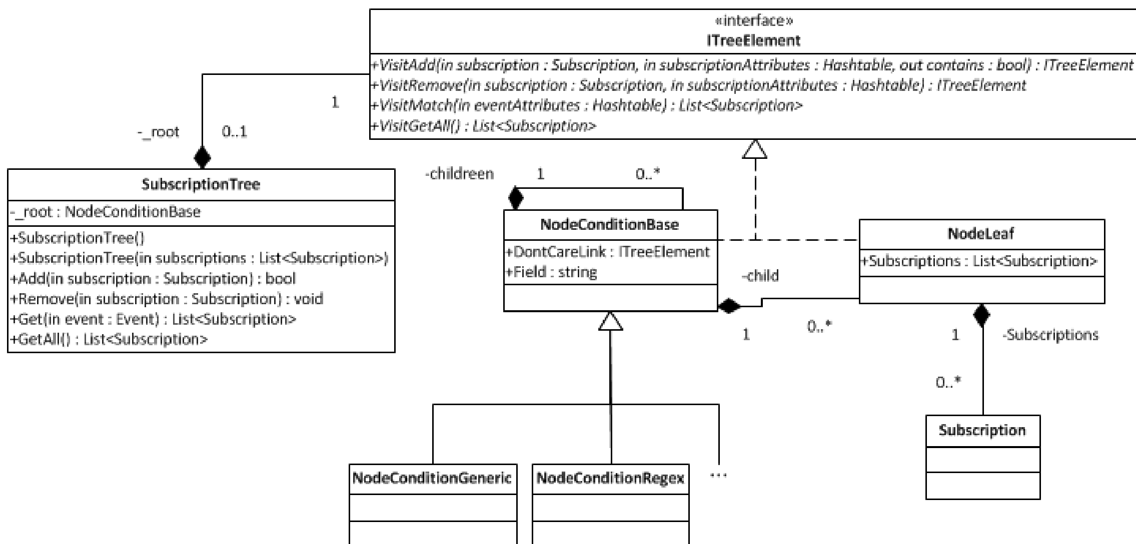


Figura 18 - Diagrama UML de classes de SubscriptionTree

A classe ***SubscriptionTree*** consiste numa árvore de subscrições, sendo que esta árvore é composta por nós não folha, ***NodeConditionBase***, que criam um caminho até nós folha, ***NodeLeaf***. Os nós não folha são constituídos por um campo ***Field***, que refere o nome do descritor de conteúdo do evento a que este nó corresponde, e um campo ***DontCare***, que representa o caminho das subscrições que não referem o descritor de conteúdo do evento a que o nó corresponde. Os nós folha contêm um conjunto de subscrições que subscreve os eventos com os descritores de conteúdo correspondentes a cada um dos representados pelos nós condicionais, desde a raiz até ao nó folha onde estão presentes.

Como a Figura 18 ilustra, é possível estender os tipos de ***NodeCondition***, sendo que estes representam os nós não folha onde se realiza a correspondência entre um determinado campo do evento e das subscrições contidas na ***SubscriptionTree***. Desta forma torna-se fácil adicionar um novo tipo de condição. O tipo de nó não folha mais simples é o ***NodeConditionGeneric***, que se limita a comprar por igualdade o valor de cada ligação do nó não folha, com o valor correspondente ao campo do evento. A implementação do protocolo **RestMS** exige a criação de um tipo de nó não folha que

faça a comparação por expressões regulares, por forma a cumprir este requisito foi criada a classe ***NodeConditionRegex***.

3.2.6.1.1 Sincronização

A sincronização da estrutura ***SubscriptionTree*** é baseada em **ReadWriterLock**s [56], sendo que cada nó contém um **ReadWriterLock**. Estes objectos permitem obter exclusão mútua de acesso a cada nó, com duas semânticas, leitura ou escrita, sendo que em modo leitura podem existir vários acessos de leitura e em escrita apenas esse acesso é permitido.

Uma vez que cada nó depende do contexto fixado pelos nós anteriores, o percurso de acesso à estrutura ***SubscriptionTree*** pressupõe sempre a aquisição da exclusão mútua de cada nó percorrido, em modo de leitura. Quando um acesso pressupõe a escrita no nó corrente adquire-se a exclusão mútua, desse nó, em modo de escrita. Desta forma impossibilita-se a alteração do contexto de um nó, uma vez que se obtém a exclusão de todos os nós percorridos até ele e permite-se a alteração de outros ramos da estrutura sem que para isso seja necessário bloquear todos os acessos à mesma. Esta forma de sincronização não pressupõe a criação de cópias da estrutura para garantir a sua coerência, poupando-se, desta forma, memória.

3.2.6.1.2 Anotação para comunicação com outras instâncias Blackbox

A comunicação entre diferentes instâncias ***Blackbox*** pressupõe o conhecimento da origem das subscrições, este conhecimento é armazenado na estrutura ***SubscriptionTree***, uma vez que é nesta estrutura que se guardam as subscrições e se faz a correspondência entre eventos e subscrições.

As subscrições contêm informação relativamente às instâncias ***Blackbox*** de onde provêm. Esta informação é armazenada na forma de um *array* de booleanos, em que cada posição diz respeito a uma das instâncias ***Blackbox***, ligada à instância corrente.

A informação de que a subscrição foi feita localmente, é armazenada sob a forma de um booleano e provoca a emissão dos eventos correspondidos a esta, para um cliente local.

A informação de que a subscrição foi feita por um cliente remoto, e foi reenviada por uma instância ***Blackbox***, é armazenada no *array* de booleanos. No processo de

correspondência de um evento às subscrições armazenadas, a posição, da estrutura referente às ligações da instância ***Blackbox***, referente à instância que reenviou a subscrição, torna-se verdadeira, aquando a correspondência de um evento a uma subscrição reenviada. Como resultado do processo de correspondência é retornada a informação das instâncias ***Blackbox*** que devem receber o evento.

3.2.7 Transformadores

A entidade ***Transformador*** transforma eventos de um determinado protocolo em eventos de outro protocolo. Desta forma isola-se o conhecimento dos diferentes protocolos, da entidade ***Blackbox***, delegando esta especificidade para uma entidade que é implementada como um *pluggin*. É assim possível adicionar suporte para novos protocolos através da implementação de novos ***Receptores*** e ***Emissores***. A comunicação entre os novos protocolos é conseguida ao implementar-se um ***Transformador*** para cada combinação de protocolos.

A Figura 19 mostra o esquema de funcionamento da entidade ***Transformador***.

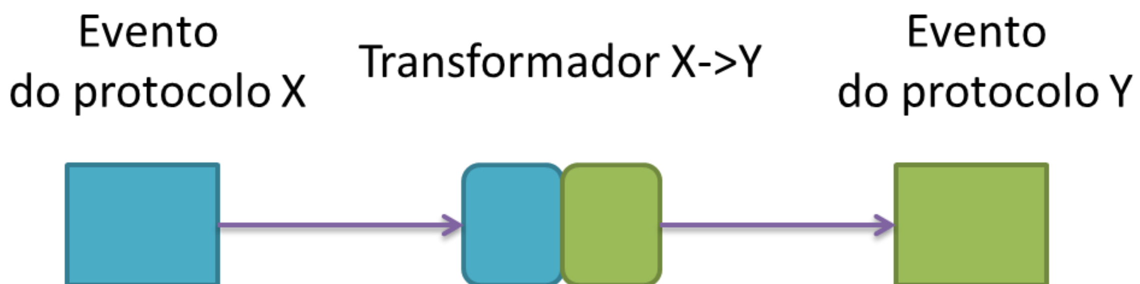


Figura 19 - Transformação de um evento do protocolo X num do protocolo Y

Esta solução pressupõe que cada ***Transformador*** tem um protocolo de entrada e um protocolo de saída, desta forma são necessários n^2 ***Transformadores***, sendo n o número de protocolos implementados. Este requisito pode ser um problema pelo número de ***Transformadores*** necessários, mas foi a única forma encontrada para possibilitar a expansibilidade de protocolos, não retirando especificidade aos tipos de evento de cada protocolo, ao implementar-se, por exemplo, um formato canónico de evento. De notar que apenas são necessários ***Transformadores*** para permitir a troca de eventos entre diferentes protocolos, sendo que sua a inexistência não põe em causa o

normal funcionamento do sistema, apenas impossibilita a troca de eventos entre protocolos para os quais não existem ***Transformadores***.

3.2.8 Loader

O sistema MIB pode ter a sua funcionalidade estendida através de *pluggins*, sendo que estes dão suporte a novos protocolos. Uma vez que se pretende o máximo automatismo na instalação destes *pluggins*, concebeu-se que estes não necessitam de se registar no sistema, numa base de dados ou num ficheiro de configuração, simplesmente necessitam de implementar a interface correcta e colocar os ficheiros que necessitam na pasta *pluggins*, da directoria de trabalho da entidade ***Blackbox***.

O carregamento de *pluggins* é efectuado através do carregamento dos tipos presentes nos **assemblies**, contidos na pasta *pluggins*. Após se inspeccionar o tipo de *pluggin* é realizado o processo adequado.

3.2.8.1 Receptores

Os receptores devem implementar a interface ***IReceiver***. As instâncias desta entidade devem esperar a recepção de mensagens, estando a maior parte do tempo bloqueados, e por forma a manter o isolamento processual entre a entidade ***Blackbox*** e qualquer *pluggin* de terceiros, estes são lançados num novo processo.

3.2.8.2 Transformadores

Os transformadores devem implementar a interface ***ITransformer***. Este tipo de *pluggin* consiste apenas num tipo, ou num conjunto de tipos, utilizado para transformar eventos de diferentes tipos de protocolos, desta forma basta que estes sejam carregados para o **appdomain** corrente e seja retornada uma referência para uma instância deste.

3.2.8.3 Emissores

Os *pluggins* emissores devem implementar a interface ***ILaunchableEmitter***, que define tudo o que é necessário para o lançamento de um emissor. Este é o tipo de *pluggin* mais complexo, uma vez que, além de ter as mesmas necessidades de funcionamento e isolamento que os ***Receptores***, também necessita de ser contactado pela entidade ***Blackbox***, portanto é necessário que exista um *proxy* para cada um dos emissores

criados. A criação de um *proxy* pressupõe o conhecimento do endereço do destinatário, tipo de *binding* de comunicação e contracto de comunicação. O *binding* e endereço do destinatário são propriedades estáticas expostas pela interface ***ILaunchableEmitter*** e o contracto de comunicação é ***IEmitter***, uma vez que se trata de um emissor.

3.2.9 Adaptadores para clientes

O MIB é um *middleware* de comunicação, que tem como principal objectivo abstrair os clientes da camada de comunicação, permitindo que qualquer cliente utilize um adaptador e que, através deste comunique com outros clientes, utilizando o protocolo que lhe for mais conveniente. As versões dos adaptadores, específicas aos protocolos, a que o MIB actualmente disponibiliza suporte, são **TCP** e **RestMS**, sendo que a adição de novos tipos de adaptadores, apenas exige a compatibilidade com o protocolo implementado por um ***Emissor*** e/ou ***Receptor*** em funcionamento.

Esta entidade interage com a entidade ***Emissor***, para receber eventos subscritos, e com a entidade ***Receptor***, para enviar subscrições a eventos ou eventos.

Os adaptadores são a interface para os clientes, disponibilizando as seguintes operações:

- Recepção assíncrona
- Recepção síncrona
- Emissão assíncrona
- Emissão síncrona
- Operações específicas

A recepção pode ser feita de forma síncrona ou assíncrona, sendo que desta forma é permitida a flexibilidade quanto ao sincronismo das operações, o mesmo acontece com a operação de emissão. As operações específicas são particulares a protocolos, como por exemplo no caso do protocolo **RestMS**, em que existem entidades no protocolo que têm de ser criadas e manipuladas, como por exemplo o ***Pipe*** ou ***Feed***.

A garantia de persistência dos eventos enviados, para no caso de uma falha na comunicação, não se perder o evento, é disponibilizada de forma opcional.

A garantia de processamento transaccional, que garante o sucesso numa operação, assim como a atomicidade entre várias operações, é disponibilizada de forma opcional

3.2.9.1 Interface com MIB

A interface que é disponibilizada para comunicar com a entidade ***Receptor*** ou ***Emissor***, dos protocolos que não definem um formato de mensagens a trocar entre cliente e servidor, consiste nas classes ***EventContract*** e ***SubscriptionContract***, sendo que a primeira representa um evento e a segunda uma subscrição.

A Figura 20 mostra a hierarquia das classes ***OperationContract***, ***EventContract*** e ***SubscriptionContract***, que constituem as representações de eventos e subscrições que podem ser enviadas num canal de comunicação, quando o protocolo não define um formato para o efeito.

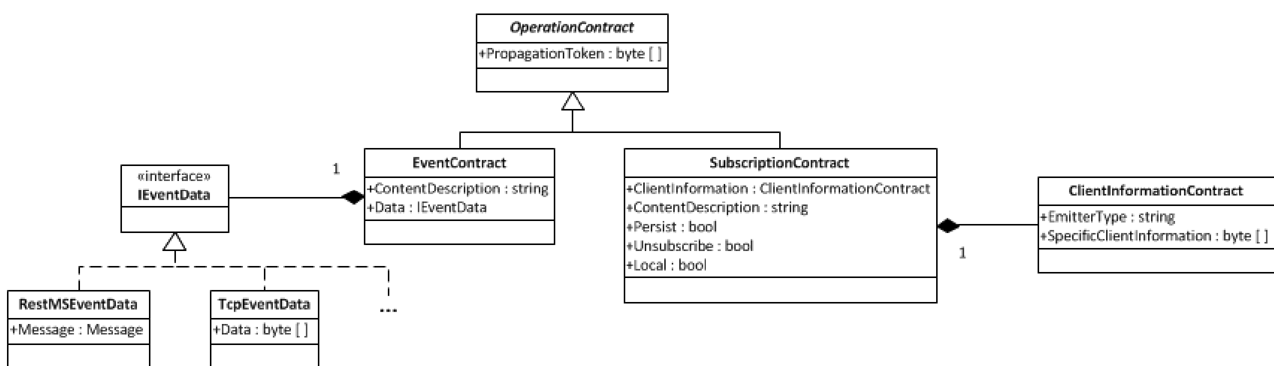


Figura 20 - Diagrama UML de classes da interface a utilizar com entidade Emissor e Receptor, quando o protocolo não fornece formato para as mensagens

A classe ***ClientInformation*** é a base da hierarquia de classes que representa a informação de clientes nos diversos protocolos de comunicação. Esta classe contém o campo ***EmitterType***, que permite identificar o tipo de entidade ***Emissor*** a utilizar para enviar o evento correspondido. Desta forma a entidade ***Blackbox*** pode ser agnóstica quanto aos tipos utilizados pelos protocolos implementados, limitando-se a fazer correspondências dos nomes dos tipos de protocolo, para seleccionar as entidades ***Transformador*** e ***Emissor*** afectas aos protocolos a utilizar.

A interface ***IEventData*** representa a base da hierarquia de tipos que representam os dados dos eventos a serem transportados. Esta hierarquia permite a injecção de dependências na classe ***EventContract***, possibilitando que as várias instâncias das entidades ***Emissor*** e ***Receptor*** tenham conhecimento apenas dos tipos de dados transportados nos eventos do protocolo de que são específicos.

3.2.9.2 Sincronismo

A garantia de sincronismo, no acto de recepção ou emissão de dados, é dada pelo adaptador para cliente. Este é responsável por abstrair o cliente dos aspectos necessários para garantir a comunicação síncrona ou assíncrona, sendo que o cliente apenas indica se deseja fazer a emissão ou recepção de informação de forma síncrona ou assíncrona.

3.2.9.2.1 Emissão

A emissão de eventos ou subscrições provém da interacção entre o adaptador e a entidade ***Receptor***, que segue o mesmo padrão para todos os protocolos implementados. É implementado um único modelo assíncrono. Este define a existência de uma fila de pedidos de envio de eventos ou subscrições. Um cliente que deseja fazer um envio assíncrono coloca um pedido de envio na fila de pedidos. O envio de informação de forma síncrona provoca não só a colocação do pedido de envio na fila de pedidos, como o bloqueio do fio de execução do utilizador do adaptador para clientes. Este fio de execução é desbloqueado aquando da satisfação, com sucesso, do pedido de envio. Existe um fio de execução responsável por recolher os pedidos da fila de envio e satisfaze-los.

A Figura 21 mostra o encadeamento de acções que decorrem na emissão de um evento/subscrição de forma síncrona.

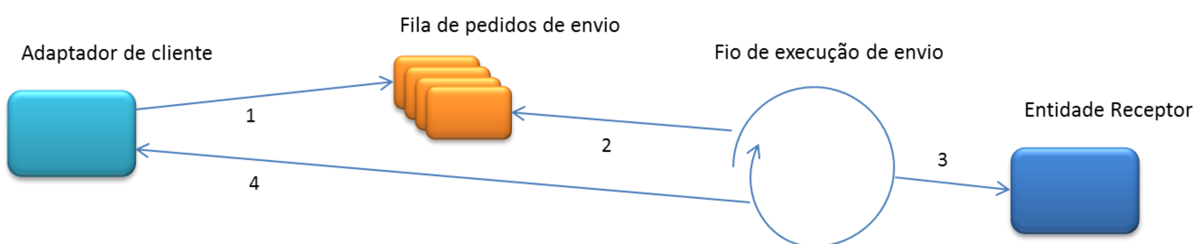


Figura 21 - Emissão de um evento/subscrição de forma síncrona

A Figura 22 mostra o encadeamento de acções que decorrem na emissão de um evento/subscrição de forma assíncrona.

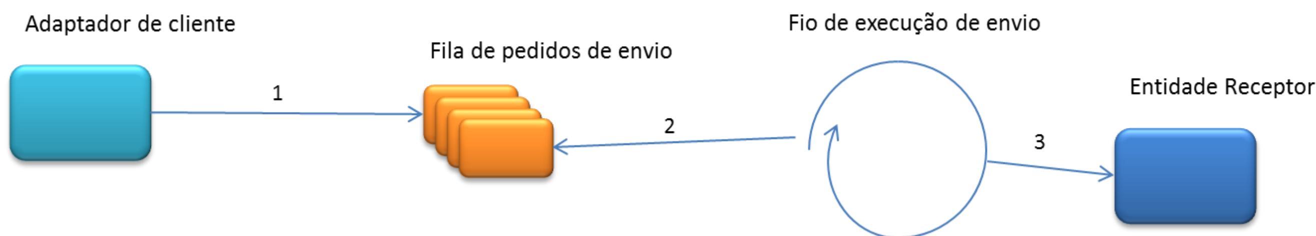


Figura 22 - Emissão de um evento/subscrição de forma assíncrona

3.2.9.2.2 Recepção

A recepção de eventos resulta da interacção entre o adaptador para clientes e a entidade ***Emissor***, esta interacção pode ser realizada segundo duas filosofias:

- Pull
- Push

A primeira, refere-se à interacção em que o receptor, neste caso o adaptador para clientes, faz pedidos à entidade ***Emissor*** para receber os eventos subscritos. A segunda, refere-se à interacção em que o receptor, neste caso o adaptador para clientes, é contactado pela entidade ***Emissor*** com o objectivo de lhe serem entregues eventos subscritos.

O tipo de interacção entre o adaptador para clientes e a entidade ***Emissor*** depende do protocolo utilizado para comunicação, sendo que os adaptadores para clientes devem seguir a implementação da filosofia de interacção aconselhada. A filosofia de interacção Push é implementada no adaptador do protocolo **TCP**, sendo esta a implementação aconselhada. Da mesma forma a filosofia de interacção Pull é implementada no adaptador do protocolo **RestMS**, sendo esta a implementação aconselhada para esta filosofia de interacção.

3.2.9.2.2.1 Pull

A filosofia de interacção *Pull* pressupõe que o cliente, neste caso o adaptador para clientes, contacta a entidade ***Emissor*** para recolher eventos subscritos.

A implementação aconselhada para esta filosofia de interacção é feita no adaptador do protocolo **RestMS**. Uma vez que a entidade ***Emissor*** é definida, pelo protocolo, como um servidor **HTTP**, não pode comunicar com os clientes/adaptadores para clientes, sendo que a comunicação tem de ser sempre iniciada pelos clientes.

Quando um cliente requisita a recepção de um evento, este pedido é colocado numa fila de pedidos de recepção, existente no adaptador para clientes. Quando o pedido é satisfeito, no caso de se tratar de um pedido de recepção síncrono, o fio de execução do utilizador do adaptador de cliente estava bloqueado, e por o pedido ser satisfeito, é retornado o evento recolhido e desbloqueado este fio de execução. No caso de um pedido de recepção assíncrono, aquando a satisfação do pedido de recepção, é executado o *callback* definido na criação do pedido, recebendo como argumento o evento recolhido. Existe um fio de execução responsável por verificar a fila de pedidos de recepção e recolher os eventos da entidade ***Emissor***, com o propósito de satisfazer os pedidos de recepção.

A Figura 23 mostra o encadeamento de acções que decorrem na recepção do tipo *Pull*, de forma síncrona. A recepção utilizando esta filosofia de interacção de forma assíncrona, difere da mesma recepção de forma síncrona, por no passo 4, no esquema síncrono, ocorrer o desbloqueio de um fio de execução e na forma assíncrona, ocorrer a chamada de um *callback*.

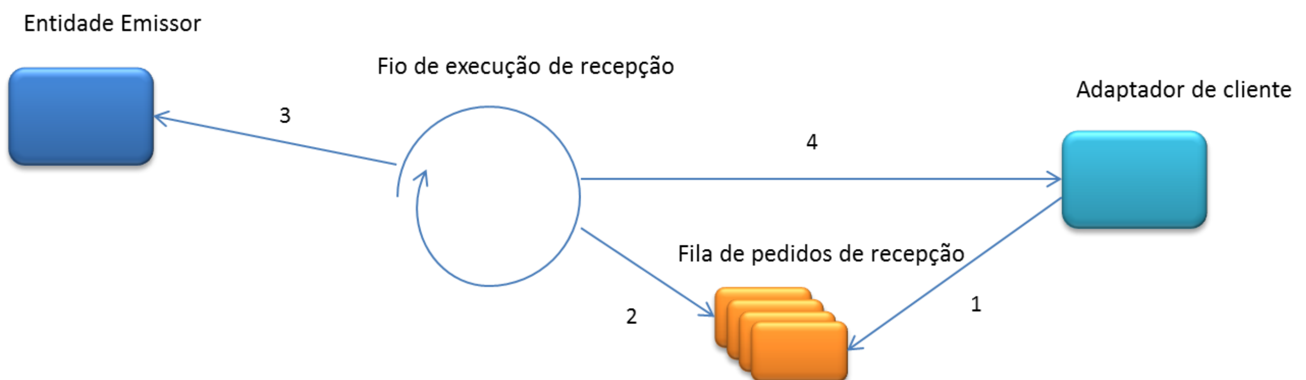


Figura 23 - Recepção de eventos, utilizando interacção do tipo Pull de forma síncrona

3.2.9.2.2.2 Push

A filosofia de interacção *Push* pressupõe que o servidor, neste caso entidade Emissor, contacta o adaptador para cliente para entregar eventos subscritos.

A implementação aconselhada para esta filosofia de interacção é feita no adaptador para cliente do protocolo **TCP**. No caso deste protocolo, define-se que a entidade ***Emissor*** é um servidor ***TCP*** e portanto tem a iniciativa de contactar os seus clientes/adaptadores para cliente do protocolo **TCP**, para entregar eventos subscritos. Esta filosofia de

interacção não pode transparecer para o utilizador do adaptador de cliente **TCP**, sendo que este deve enviar eventos/subscrições ou receber eventos quando desejar, sem que esteja à escuta das comunicações da entidade Emissor. A solução adaptada para este problema passa por implementar uma fila de eventos recebidos, sendo que é criada uma instância de **TcpListener** [57] e é agendada a recepção assíncrona de uma comunicação. Aquando a recepção de uma comunicação, por parte da entidade ***Emissor***, é reagendada outra recepção assíncrona e coloca-se o evento recebido na fila para esse efeito.

A recepção dos eventos, por parte dos clientes do adaptador para cliente, pode ser realizada de forma síncrona ou assíncrona, sendo que a primeira é realizada utilizando a segunda. O mecanismo de recepção assíncrona define que aquando um pedido de recepção, é colocado um pedido de recepção numa fila criada para este propósito e é sondada a fila de eventos recebidos, com o propósito de inferir se existem condições para satisfazer o pedido de recepção assíncrona, caso seja possível satisfaze-lo, é executado o *callback* definido no pedido de recepção, passando como argumento o evento recebido. Caso se trate de um pedido de recepção síncrono, após colocar o pedido de recepção na fila para este efeito, o fio de execução do utilizador do adaptador de cliente é bloqueado, aquando a satisfação do pedido, é retornado o evento recebido e o fio de execução do utilizador é desbloqueado. Existe um fio de execução responsável por recolher eventos da fila de eventos recebidos e satisfazer pedidos existentes na fila para esse efeito.

A Figura 24 mostra o encadeamento de acções que forma a recepção de eventos, utilizando a filosofia de interacção *Push* de forma síncrona. A diferença entre a forma síncrona e assíncrona consiste no facto de na primeira, o passo 5 ser o desbloqueio de um fio de execução, e na segunda, ser a chamada de um *callback*.

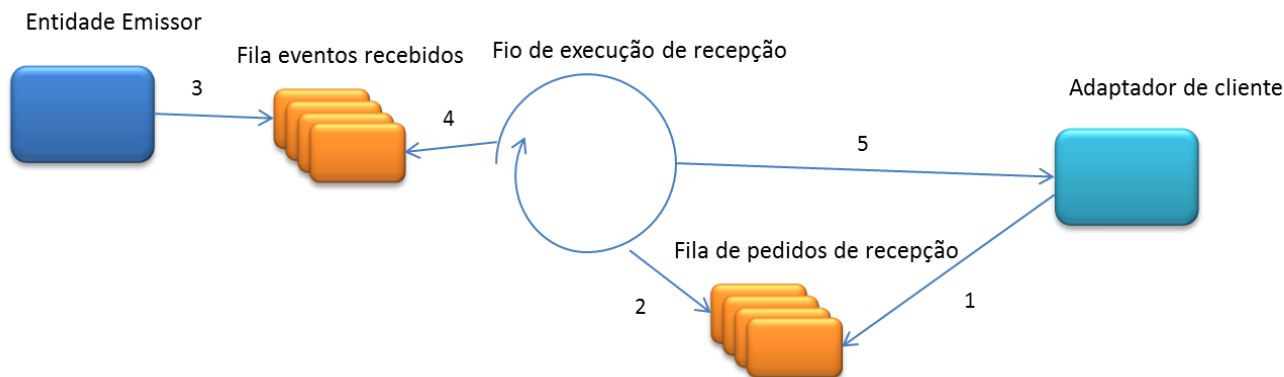


Figura 24 - Recepção de eventos, utilizando interacção Push, de forma síncrona

3.2.9.3 Persistência

A garantia de persistência dos eventos enviados, para no caso de uma falha na comunicação, não se perder o evento, ou mesmo para permitir o desacoplamento temporal, entre o cliente que envia o evento e o cliente que subscreve esse mesmo evento, é opcional e é definida na subscrição de eventos. Esta garantia pode ser uma opção a disponibilizar ao utilizador, como no caso do adaptador do protocolo **TCP**, ou um aspecto a ser definido pelo protocolo, sem interferência do cliente, como no caso do protocolo **RestMS**.

A utilização da garantia de persistência é activada no acto da subscrição, na propriedade ***Persist***, da classe ***Subscription***. Esta classe representa uma subscrição.

Esta garantia permite que os eventos correspondidos a uma subscrição sejam armazenados até que o subscritor receba, com sucesso, os eventos. Caso ocorra uma falha no cliente subscritor do evento, aquando a reactivação da subscrição, após resolução da falha, os eventos persistidos são reenviados para o cliente. Desta forma além de se conseguir a recuperação de falhas, sem perda de eventos, também se consegue o desacoplamento temporal, entre o acto de emissão de evento e recepção do mesmo, pois no caso da subscrição ser realizada com a garantia de persistência activa, todos os eventos correspondidos, em caso de insucesso na entrega ao subscritor, são armazenados. Aquando o reactivar da subscrição, todos os eventos armazenados são entregues. O acto de reactivar uma subscrição consiste no reenvio de uma subscrição já existente no sistema.

3.2.9.4 Transacções

O processamento transaccional é uma garantia que permite que um cliente tenha a confirmação que um evento/subscrição é entregue ao MIB, ou que o MIB tenha a confirmação que um cliente recebe com sucesso um evento, sendo que esta confirmação é disponibilizada na forma de uma transacção. Esta garantia é opcional, pois acarreta custos no processamento.

Os adaptadores para cliente devem implementar a garantia de processamento transaccional da forma aconselhada, que é a implementada nos adaptadores para cliente **TCP** e **RestMS**. A implementação aconselhada consiste na integração das transacções da *framework*, disponibilizada pela linguagem do adaptador para cliente, em conjunção com transacções distribuídas.

No caso da linguagem .NET é aconselhada a utilização da classe **Transaction** [58], sendo que esta é a forma disponibilizada para o cliente registar o interesse na utilização de garantia de processamento transaccional. Quando é invocada uma operação, do adaptador para cliente, no contexto de uma transacção, o adaptador para cliente deve transformar essa transacção numa transacção distribuída e colocar o seu símbolo de propagação no campo **PropagationToken**, das classes ***Subscription*** e ***Event***, que representam subscrições ou eventos a serem entregues ao MIB. Desta forma na recepção da subscrição ou evento, o MIB detecta a existência da necessidade de utilização de processamento transaccional e torna-se um participante da transacção distribuída criada pelo cliente, sendo que caso a subscrição ou evento sejam recebidos e registados com sucesso, no MIB, esta é terminada com sucesso. A forma aconselhada para disponibilizar o processamento transaccional no acto de recepção de um evento, consiste na disponibilização de uma instância da classe **Transaction**, que representa a transacção distribuída obtida a partir do símbolo de propagação emitido pelo MIB, caso o cliente deseje utilizar processamento transaccional no acto da recepção do evento, tem de se tornar um participante na transacção. Caso termine as operações, que consistem a recepção do evento, com sucesso deve simbolizar esse facto com a conclusão da transacção com sucesso.

3.3 Protocolos

A solução descrita implementa um *middleware*, uma vez que esta é uma ferramenta que se pretende que homogeneíze a comunicação entre diferentes clientes, é fundamental que implemente o maior número de protocolos de comunicação, devendo manter-se actualizado quanto aos protocolos *standard*.

3.3.1 TCP

O protocolo TCP foi criado com o intuito de fornecer uma comunicação sem protocolo de nível de aplicação, tendo como objectivo realizar comunicações com requisitos temporais elevados, utilizando desta forma o paradigma de comunicação *publish/subscribe*. Esta implementação comunica os eventos e subscrições seriando-os em *bytes*, uma vez que o formato destes é o mesmo que o utilizado na entidade ***Blackbox***, desta forma não é necessário qualquer mapeamento de formatos.

A implementação deste protocolo não necessitou da utilização de uma base de dados.

3.3.1.1 Cliente

O adaptador para cliente **TCP** permite que os clientes comuniquem com outros, utilizando o protocolo **TCP** como protocolo de comunicação. Este adaptador, por o seu protocolo não ter operações específicas, limita-se a implementar as operações de emissão e recepção, quer de forma síncrona, como assíncrona.

3.3.1.1.1 Interface com o cliente

Os utilizadores deste adaptador para cliente não têm disponível uma interface idêntica à exposta pela própria entidade ***Emissor*** do protocolo **TCP**, em que se comunica utilizando derivados da classe ***OperationContract***, quer seja ***SubscriptionContract***, que representa uma subscrição, quer seja ***EventContract***, que representa um evento. Esta não é a interface exposta, pois estas classes têm um formato mais próprio para ser utilizado na comunicação, do que para ser preenchido por um programa utilizador, como por exemplo o campo ***ClientInformation***, da classe ***SubscriptionContract***, que representa a informação de um cliente, utilizando a hierarquia explicada na secção Eventos e subscrições. Ou por exemplo, o caso da propriedade ***ContentDescription***, da classe ***EventContract*** e ***SubscriptionContract***, que representa os descritores de conteúdo utilizados no paradigma *publish/subscribe* orientado ao conteúdo, sendo que

esta propriedade assume valores do tipo **String**, preferindo-se a interface de um conjunto de pares chave-valor para definir os descritores de conteúdo do evento ou subscrição.

A Figura 25 mostra a hierarquia de classes utilizada para interface com os utilizadores do adaptador para cliente **TCP**.

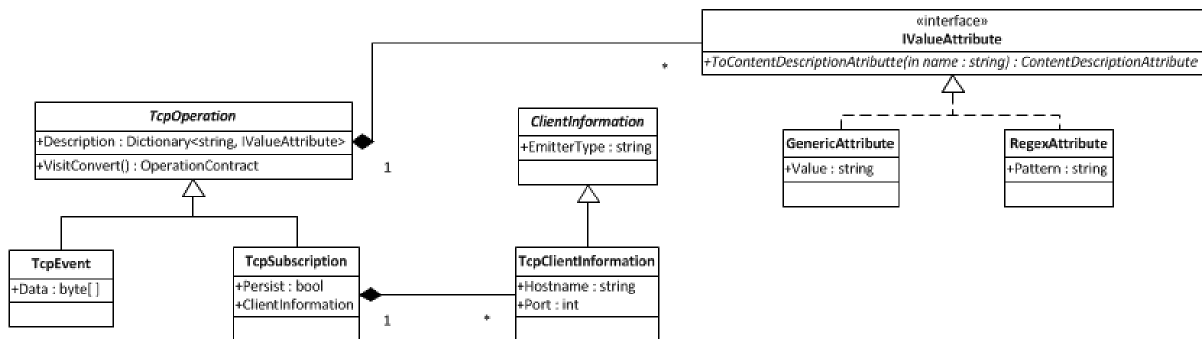


Figura 25 - Diagrama UML de classes da hierarquia de classes para interface de adaptador para cliente TCP

A representação dos descritores de conteúdo, quer dos eventos, quer das subscrições, é um par nome-valor, sendo que o valor é uma instância de um tipo que implementa a interface ***IValueAttribute***. Esta injecção de dependências permite a utilização de diferentes tipos de descritores de conteúdo, até mesmo a adição de novos tipos de descritores, após a criação do adaptador, sem que seja necessária qualquer alteração no código existente. Este problema foi abordado e resolvido de forma semelhante no subcapítulo Descritores de conteúdo.

A classe ***TcpClientInformation*** representa a informação de um cliente subscritor, que deseja receber os eventos subscritos utilizando o protocolo **TCP**.

3.3.1.1.2 Persistência

A garantia de persistência é opção do utilizador, sendo que para este fazer uso desta garantia deve assinalar a propriedade ***Persist***, da classe ***TcpSubscription***, que representa uma subscrição para um utilizador do adaptador para cliente.

3.3.1.1.3 Transacções

A garantia de processamento transaccional é opção do utilizador, sendo que a sua utilização é feita de acordo com a implementação aconselhada, que é explicada em 3.2.4 Transacções.

3.3.2 RestMS

O **RestMS** é um serviço de mensagens **RESTful** que providencia comunicação assíncrona através duma interface **REST**, utilizando **HTTP/HTTPS**.

Este protocolo tem a particularidade de implementar uma filosofia de comunicação *pull*, do ponto de vista do receptor de eventos, ou seja, não é o ***Emissor*** que envia os eventos para os clientes, mas os clientes que os recolhem.

3.3.2.1 Arquitectura

O protocolo **RestMS** define o seguinte conjunto de entidades:

- Domain
- Profile
- Feed
- Pipe
- Join
- Mensagens
- Conteúdos

De seguida é feita uma pequena explicação das entidades apresentadas na figura, assim como das entidades mensagem e conteúdos, que por uma questão de simplificação não aparecem na figura, mas fazem parte do funcionamento do **RestMS**.

Domains

Os *domains* são *namespaces* para *pipes* e *feeds*. Um servidor pode ter vários *domains* e cada um deles pode ter credenciais de segurança diferentes. Um *domain* contém os *profiles* que nele podem ser utilizados.

Informação relevante:

- Nome
- Título
- *Feeds*
- *Pipes*
- *Profiles*

Profiles

A entidade *profile* contém um conjunto de semânticas que o servidor implementa, estas semânticas podem definir tipos de *feeds*, *pipes*, códigos de resposta e utilizações dos objectos definidos.

Informação relevante:

- Nome
- Título
- Referência

Feeds

As *feeds* são o ponto de entrada de mensagens no sistema. Vários clientes podem escrever mensagens numa *feed* e a ordem das *feeds* deve ser mantida, relativamente a cada cliente. As *feeds* podem ser criadas dinamicamente por clientes.

Informação relevante:

- Nome
- Tipo
- Título
- Licença
- Conteúdos para mensagens

Pipes

Os *pipes* consistem na forma de entrega de mensagens a clientes. Sobre os *pipes* apenas se podem realizar leituras e são ordenados para um leitor. São criados de forma dinâmica e são privados para os clientes que os criam. O servidor pode limpar *pipes* que não sejam utilizados ou que estejam a ficar sobrecarregados de mensagens.

Informação relevante:

- Tipo
- Título
- *Joins*
- Mensagens

Joins

Os *joins* definem o encaminhamento que deve ser feito das mensagens recebidas nas *feeds* para os *pipes* subscritos por clientes. São criados de forma dinâmica e devem sempre ser privados. A criação de *joins* deve ser feita sempre para o **URI** do *pipe* respectivo. Se a *feed* ou o *pipe* a que o *join* diz respeito forem destruídos, o *join* também o será.

Informação relevante:

- Tipo
- Endereço que filtra mensagens
- *Feed* que referência
- *Headers* que filtram mensagens

3.3.2.1.1 Mensagens e conteúdos

As mensagens e conteúdos estão ligados, sendo que uma mensagem é um envelope que pode conter conteúdos, nela embebidos, ou endereços para conteúdos sobre a forma de recursos. Estes recursos são especificados com um tipo **MIME** e são tratados como *blobs*.

Informação relevante de mensagens:

- Endereço
- Identificador da mensagem
- Endereço de resposta
- Conteúdos
- *Headers*

Informação relevante de conteúdos:

- Tipo
- Codificação
- Valor

3.3.2.2 Implementação

A informação que um cliente **RestMS** produz consiste numa mensagem, esta informação tem de ser mapeada para um evento que irá ser encaminhado para os clientes que o subscrevam.

Uma mensagem **RestMS** contém a seguinte informação que se considera relevante para o seu encaminhamento:

- Campo *address* da mensagem
- *Feed* para onde a mensagem foi enviada
- Os diversos conteúdos da mensagem, quer os embebidos nesta, quer os referenciados

Um evento **RestMS** consiste num conjunto de descritores de conteúdo, sendo que o campo *address* da mensagem e a *feed* para onde a mensagem foi enviada são codificados como descritores de conteúdo. Os diversos conteúdos da mensagem são embebidos como diversos conteúdos do evento, sendo que os conteúdos referenciados são obtidos da base de dados para passarem a ser conteúdos embebidos.

A criação de um *join* representa a criação de uma subscrição que contém como informação do cliente destino, o *pipe* a que este diz respeito, e como descritores de conteúdo, o endereço, os *headers* do *join* e a *feed* a que o *join* diz respeito.

3.3.2.2.1 Persistência

O funcionamento do **RestMS** requer a utilização de uma base de dados para armazenar informação para o ***Receptor*** e ***Emissor*** deste protocolo. O ***Receptor*** necessita de armazenar as *feeds* existentes no sistema, assim como os conteúdos submetidos antes de ser enviada uma mensagem que referencia estes conteúdos. O ***Emissor*** necessita de armazenar os *pipes* que existem, assim como as mensagens que estes contêm. É armazenada a informação dos *joins* existentes para que estes possam ser visualizados aquando a apresentação de um *pipe*.

A Figura 26 mostra o modelo relacional da base de dados criada para o **RestMS.**

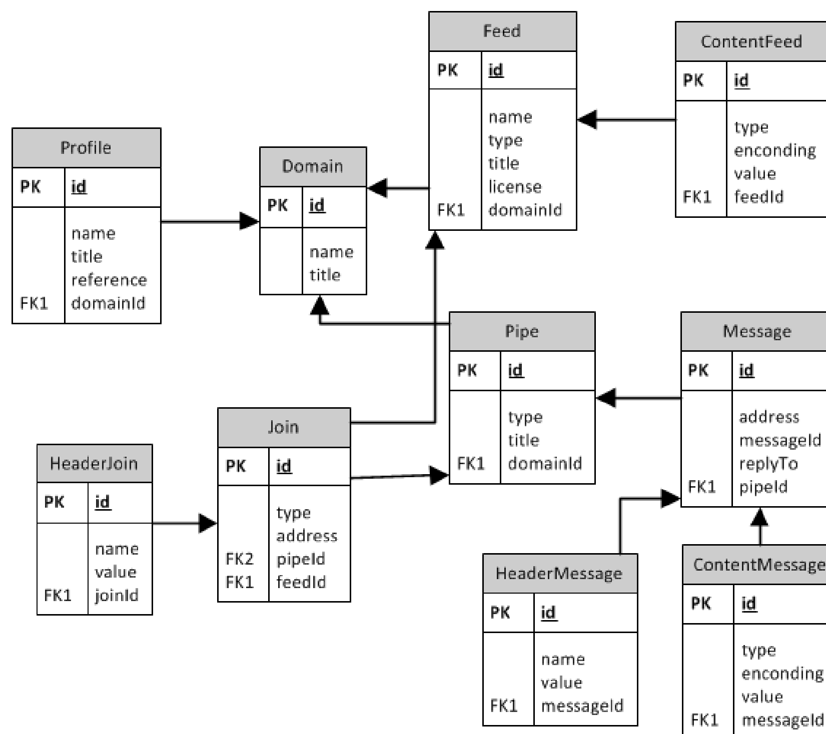


Figura 26 - Modelo relacional da base de dados RestMS

3.3.2.2.2 Cliente

O adaptador para cliente **RestMS** é a peça de *software* que permite que um cliente comunique com outros, utilizando o protocolo **RestMS**. Este adaptador para cliente define, além das operações que todos os adaptadores devem definir, recepção e emissão, síncrona ou assíncrona, as operações específicas ao protocolo **RestMS**, sendo elas:

- CreateJoin: Cria uma instância da entidade *Join*
- CreateFeed: Cria uma instância da entidade *Feed*
- CreatePipe: Cria uma instância da entidade *Pipe*
- DeleteMessage: Apaga uma mensagem

O formato das mensagens a circular no canal de comunicação é o definido pelo protocolo **RestMS**, sendo que as mensagens circulam em formato **XML**(Extensive Markup Language), sob a forma de elementos definidos pelo protocolo.

A garantia de persistência é opcional e definida pelo próprio protocolo, sem dar a hipótese de decisão ao utilizador.

A garantia de processamento transaccional é opcional e é o utilizador que decide a sua utilização.

3.3.2.2.2.1 Interface

O adaptador para cliente **RestMS** define um conjunto de classes que representam as entidades definidas no protocolo, utilizando a representação que se acredita ser a mais adequada para interface com um utilizador. Estas entidades são:

- Feed: representa um ponto de recepção de mensagens, por parte do MIB
- Content: representa um conteúdo a circular no MIB
- Header: representa um descritor de conteúdo, quer de uma subscrição, quer de um evento
- Message: representa um evento no contexto do protocolo **RestMS**, sendo constituída por descritores de conteúdo e conteúdos
- Pipe: representa um ponto de recepção de mensagens, por parte de um cliente
- Join: representa a directiva de encaminhamento que define das mensagens que chegam a determinada Feed, quais as que devem ser encaminhadas para um determinado Pipe.

A Figura 27 mostra o conjunto de classes que representam as entidades do protocolo **RestMS**, na forma mais adequada para interface com o utilizador.

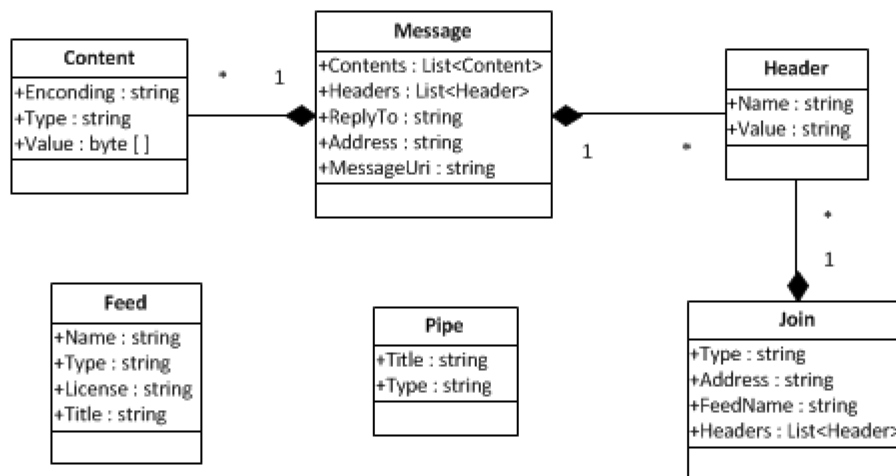


Figura 27 - Diagrama UML de classes de conjunto de classes interface com cliente de adaptador RestMS

O adaptador para cliente **RestMS** faz a conversão entre as representações para interface com o utilizador e as representações definidas pelo protocolo **RestMS**, das entidades acima descritas.

3.3.2.2.2.2 Persistência

O protocolo **RestMS** define que as mensagens, ou seja eventos, devem ser mantidas no *Pipe*, até que o subscritor receba a informação com sucesso, portanto a entidade ***Receptor*** **RestMS**, ao receber a criação de um *Join*, cria uma subscrição com a persistência activa. Uma vez que a filosofia de interacção do protocolo **RestMS** é *Pull*, o subscritor nunca pode receber a informação no momento da sua chegada ao emissor, pois o cliente é quem deve recolher os eventos da entidade ***Emissor***, portanto é necessário que seja utilizada a garantia de persistência.

3.3.2.2.2.3 Transacções

A garantia de processamento transaccional é opção do utilizador, sendo que a sua utilização é feita de acordo com a implementação aconselhada, que é explicada em Transacções.

(Página propositadamente deixada em branco)

4 Conclusão

O trabalho de pesquisa permitiu aprofundar os conhecimentos sobre integração de sistemas, quer a nível dos paradigmas de comunicação, quer a nível dos protocolos mais utilizados. Uma das conclusões que foram retiradas deste trabalho de pesquisa foi o estado actual da indústria de *software* e as necessidades existentes na integração de aplicações de forma a criar aplicações com valor acrescentado.

O resultado do trabalho de pesquisa, relativamente aos paradigmas de comunicação, permitiu concluir que a evolução neste campo chegou a um ponto onde se considera que o paradigma mais viável e com melhores mais-valias para as comunicações locais, que se trata do problema foco a resolver por este trabalho, é o *publish/subscribe*.

Os resultados obtidos com o estudo efectuado sobre sistemas *publish/subscribe* permitiu concluir que as implementações deste tipo de sistemas são baseadas em tópicos, sendo que desta forma a implementação tem desempenhos em termos de tempo aceitáveis, mas reduz a riqueza da semântica de subscrição de eventos, existindo desta forma "canais" de eventos e cada evento apenas pode ser inserido no canal ou repetido pelos vários canais. A abordagem seguida pela solução apresentada consiste num sistema *publish/subscribe* baseado no conteúdo, este tipo de sistema acarreta maiores problemas de desempenho, mas por contra partida permite uma semântica de subscrição muito mais rica, uma vez que cada subscrição selecciona eventos baseados nos seus conteúdos, mais concretamente nos descritores de conteúdo destes.

A implementação do sistema *publish/subscribe* levou a uma pesquisa sobre a forma de realizar a implementação deste tipo de sistema evitando os elevados custos de desempenho. Após se realizar o estudo inicial sobre este tipo de sistema, chegou-se à conclusão que a melhor forma de evitar os elevados custos de desempenho seria distribuir o sistema, ao invés deste ser centralizado. Foi estudada a possibilidade de implementar este sistema utilizando uma **DHT**(Distributed Hash Table), sendo que existiria um esquema de mapeamento entre os eventos e clientes, e cada cliente seria uma parte do sistema *publish/subscribe*. Esta abordagem não pôde ser utilizada, pois o mapeamento pressuporia um conjunto de descritores de conteúdo fixos, assim como um

domínio de valores estático, estando-se, desta forma, a empobrecer a semântica de subscrição de eventos.

A implementação do sistema *publish/subscribe* baseada no conteúdo dos eventos escolhida foi com base numa arquitectura semi-distribuída, em que exista a entidade de servidor, mas esta é distribuída por várias instâncias da mesma. Esta implementação acarreta problemas relativamente ao encaminhamento dos eventos e manutenção das subscrições existentes no sistema. O estudo sobre este assunto levou à conclusão que a implementação deveria recorrer ao algoritmo de distribuição de eventos **Link Matching**, uma vez que este evita a sobrecarga dos vários nós do sistema com a distribuição desnecessária de diversos eventos.

A correspondência entre as subscrições e os eventos é outro ponto onde os sistemas *publish/subscribe* baseados no conteúdo dos eventos têm contra partidas, uma vez que desde o armazenamento das subscrições, até às comparações entre os descritores de conteúdo destas e dos eventos, os custos de desempenho são elevados. A forma encontrada para reduzir estes custos consiste na utilização de uma estrutura **PST** baseada nos descritores de conteúdo das subscrições. Esta estrutura além de providenciar um armazenamento para as subscrições também permite um melhor desempenho do processo de correspondência entre subscrições e eventos. Após várias implementações teste, conclui-se que a melhor implementação para as comparações consiste na definição de vários tipos de descritor de conteúdo, sendo que cada tipo tem associado um comparador e um tipo de nó na **PST**, desta forma a semântica para descrever os eventos e subscrições é enriquecida.

O sistema de *pluggins* revelou-se uma mais-valia, pois cumpre na perfeição a sua função, que consiste na fácil adaptação a novos protocolos. Sendo que o MIB se trata de um *middleware*, tem como objectivo facilitar a comunicação com clientes, para cumprir este objectivo tem a obrigação de acompanhar a adição de novos protocolos e desta forma conseguir também abranger um maior número de clientes.

A utilização de adaptadores para cliente facilita o trabalho dos clientes do MIB, pois a interface dos adaptadores é pensada para ser específica para o propósito que serve. O adaptador para cliente também realiza a implementação do protocolo de comunicação,

tornando-a opaca para o cliente final do MIB. Todo o trabalho necessário para conseguir a utilização de transacções, permitir a utilização dos mecanismos de persistência, assim como conseguir o desacoplamento temporal, de sincronismo e espacial, é repetitivo e poupado ao cliente final do MIB, que se limita a utilizar o adaptador para cliente, específico para o protocolo que pretende utilizar na comunicação.

Durante a execução do trabalho foi feito um ajuste nos objectivos do mesmo. Inicialmente foi planeada a disponibilização do suporte para o protocolo **AMQP**, este objectivo foi removido do plano, por se considerar a sua utilidade para fins académicos pouco relevante, uma vez que o suporte para outros protocolos permitiu demonstrar as potencialidades do MIB, assim como as boas práticas para cada situação de interacção das entidades. Também não foi realizada a implementação do adaptador para cliente **Java**, este objectivo não foi realizado por se considerar que a implementação do sistema no todo demonstra a facilidade de interacção com outras linguagens de programação, mesmo que o trabalho não faça uma demonstração deste caso.

Os restantes objectivos definidos no início do trabalho foram cumpridos, sendo que foram também realizadas tarefas que não foram previstas inicialmente, nomeadamente:

- Estudo da viabilidade da utilização de **DHT** para implementar o algoritmo *multicast*
- Implementação de mecanismos de garantia de entrega de mensagens e processamento transaccional entre clientes e entre instâncias ***Blackbox***
- Implementação de sistema de *pluggins*
- Implementação de sistema de compatibilidade entre mensagens de diferentes protocolos
- Implementação de protocolo **RestMS**

O estudo da viabilidade da utilização de mecanismos **DHT** para implementar o algoritmo de comunicação *multicast* considerou-se relevante, por existirem diversos artigos que apontam para essa como a possível solução para implementar um sistema *publish/subscribe* baseado no conteúdo com desempenho interessante. Esta alternativa não foi escolhida por empobrecer a semântica da linguagem de subscrição de eventos.

A implementação dos mecanismos das garantias de entrega de mensagens e processamento transaccional possibilita tornar as comunicações dentro do sistema MIB, assim como com clientes do mesmo, fidedigna. Estas garantias têm as suas vantagens e são disponibilizadas para comunicações que assim o exijam. Os casos em que estas

garantias não são necessárias, não as utilizam, sendo que desta forma se reduzem os custos de comunicação.

A implementação do sistema de *pluggins* foi efectuada por se determinar que um *middleware* deixa de ser interessante a partir do momento que deixa de implementar a maior parte dos protocolos *standards* do momento. Não se pretende que a adição de novos protocolos cause demasiadas alterações em peças de *software* já feitas, considerou-se imperativa a implementação de um mecanismo de extensibilidade para as entidades específicas a protocolos.

A implementação do sistema de compatibilidade entre mensagens de diferentes protocolos considerou-se um aspecto fundamental de um *middleware*, uma vez que as soluções deste tipo devem facilitar a comunicação entre diversos clientes, pretendendo-se que os clientes possam utilizar os protocolos que desejarem. É obrigatória a implementação de um mecanismo que possibilite a interacção entre clientes a comunicarem utilizando diferentes protocolos.

A implementação do protocolo **RestMS** foi realizada por se considerar interessante ilustrar a implementação de um protocolo com uma filosofia de interacção entre entidade ***Emissor*** e ***Receptor*** e adaptador para cliente, sendo esta *PULL*.

5 Trabalho futuro

A implementação corrente do MIB contém toda a parte central de funcionamento, nomeadamente a definição do sistema *publish/subscribe* e as definições aconselhadas para resolver os problemas de comunicação entre os clientes e o MIB. O sistema de *pluggins* prevê a extensibilidade do MIB, relativamente à adição de suporte para novos protocolos.

O trabalho que se considera poder enriquecer o que já foi feito consiste no seguinte:

- Implementação do protocolo **AMQP**, que actualmente é considerado um *standard* dos *middlewares*.
- Implementação de um mecanismo que detecte a existência de ligações redundantes entre instâncias ***Blackbox***, este trabalho deveria substituir a configuração manual de cada instância ***Blackbox***, relativamente às ligações a utilizar para comunicar com outras instâncias. O mecanismo a implementar deveria de alguma forma definir, autonomamente, a rede de instâncias ***Blackbox***, de forma a evitar ligações circulares, procurando definir as ligações que permitam interligar instâncias ***Blackbox*** da forma mais eficiente.
- Implementação de um mecanismo de autenticação e autorização de clientes, por forma a autenticar os clientes do MIB e definir autorizações para cada um destes. Este mecanismo deve permitir que o MIB seja utilizado num ambiente não controlado, sendo que qualquer utilizador possa tentar ligar-se ao MIB, mesmo que de forma maliciosa, sendo-lhe recusada a ligação.

(Página propositadamente deixada em branco)

6 Referências bibliográficas

1. **Bakken, David E.** Middleware. *Washington State University.*

2. *Generic Constraints for Content-Based Publish/Subscribe.* **Mühl, Gero.**

3. *RestMS.* [Online] http://www.restms.org/.

4. *TCP, Transmission Control Protocol.* [Online] http://www.networksorcery.com/enp/protocol/tcp.htm.

5. *A case for message oriented middleware.* **Gurudth Banavar, Tushar Chandra, Robert Strom and Daniel Sturman.**

6. *Remote Procedure Calls (RPC).* [Online] http://www.cs.cf.ac.uk/Dave/C/node33.html.

7. *The Many Faces of Publish/Subscribe.* **Patrick TH. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrec.**

8. ORACLE. *Introduction to CORBA.* [Online] http://java.sun.com/developer/onlineTraining/corba/corba.html.

9. futures. *alice manual.* [Online] http://www.ps.uni-saarland.de/alice/manual/futures.html.

10. Observer Pattern. *OODesign.* [Online] http://www.oodesign.com/observer-pattern.html.

11. *Tuple Space.* [Online] http://c2.com/cgi/wiki?TupleSpace.

12. *The Nature of JavaSpaces.* [Online] http://www.dancres.org/cottage/javaspaces.html.

13. *Internet Indirection Infrastructure.* **Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, Sonesh Surana.**

14. Tech target. *Talarian.* [Online] http://media.techtarget.com/searchWebServices/downloads/Talarian.pdf.

15. *USENET - A General Access UNIX Network.* **Stephen Daniel, James Ellis, Tom Truscott.**

16. w3schools. *Introduction to SQL.* [Online] http://www.w3schools.com/sql/sql_intro.asp.

17. *Filter Constraint Language.* [Online] http://documentation.progress.com/output/Iona/orbix2000/2.0/notification_java/html/Filter4.html.

18. W3C. *XML Path Language (XPath).* [Online] http://www.w3.org/TR/xpath/.

19. SUN. *Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms.* [Online] http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/.

20. *Content-based publish/subscribe with structural reflection.* **P. Eugster, R. Guerraoui.**

21. *An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems.* **Gurudth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao.** ICDCS '99 Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999.

22. *Design, analysis, and implementation of a parallel tree search algorithm.* **Akl SG, Barnard DT, Doran RJ.**

23. Webopedia. *All About Broadband/ICS Routers.* [Online] http://www.webopedia.com/DidYouKnow/Hardware_Software/2005/router.asp.

24. TIB/Rendezvous . *PCMAG.* [Online] http://www.pcmag.com/encyclopedia_term/0,2542,t=TIBRendezvous&i=52888,00.asp.

25. *Gryphon: An Information Flow Based Approach to Message Brokering.* **Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, Michael Ward.**

26. *Extending the Siena Publish/Subscribe System.* **Heimbigner, Dennis.**

27. **Stoica, Ion.** *UC Berkeley.* [Online] http://www.cs.virginia.edu/~cs757/slidespdf/757-09-overlay.pdf.

28. *Content-Based Publish-Subscribe over Structured Overlay Networks.* **Roberto Baldoni, Carlo Marchetti, Antonino Virgilito, Roman Vitenberg.**

29. **Ratnasamy, Sylvia.** A Scalable Content-Addressable Network, PH.D. Thesis. 2002.

30. *Chord: A scalable peer-to-peer lookup service for internet applications.* **Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan.**

31. *Pastry - A substrate for peer-to-peer applications.* [Online] http://research.microsoft.com/en-us/um/people/antr/Pastry/.

32. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing.* **Ben Y. Zhao, John Kubiatowicz, Anthony D. Joseph.**

33. *Gryphon: An Information Flow Based Approach to Message Brokering.* **Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, Michael Ward.**

34. *Hermes: A Distributed Event-Based Middleware Architecture.* **Peter R. Pietzuch, Jean M. Bacon.**

35. *The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS.* **Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta.**

36. *Publish/Subscribe on the Web at Extreme Speed.* **Françoise Fabret, François LLirbat, João Pereira, Dennis Shasha.**

37. *Scribe: a large-scale and decentralized application-level multicast infrastructure.* **M. Castro, P. Druschel, A.-M Kermarrec, A.I.T. Rowstron.**

38. *Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination.* **Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, John D. Kubiatowicz.**

39. *Finding SHA-1 Characteristics: General Results and Applications.* **Christophe De Cannière, Christian Rechberger.**

40. RFC 791 - Internet Protocol. [Online] http://www.faqs.org/rfcs/rfc791.html.

41. **Tyagi, Sameer.** RESTful Web Services.

42. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. [Online] http://www.ietf.org/rfc/rfc2045.txt?number=2045.

43. **Patrick Eugster, Rachid Guerraoui, Joe Sventek.** [Online] http://lpdwww.epfl.ch/peugster/pres/DACsECOOP.pdf.

44. **Fischli, Dr. Stephan.** [Online] http://www.sws.bfh.ch/~fischli/kurse/eduswiss/jms/JMS.pdf.

45. **Raj, Gopalan Suresh.** *The COM+ Model.* [Online] http://gsraj.tripod.com/com/complus.html.

46. *IBM's MQSeries Transaction Messaging Software.* [Online] http://www.fiendish.demon.co.uk/MQSeries/.

47. *Introduction to Oracle Advanced Queuing.* [Online] http://download.oracle.com/docs/cd/B10500_01/appdev.920/a96587/qintro.htm.

48. *Generic support for distributed applications.* **J. Bacon, K. Moody, J. Bates, Chaoying Ma, A. McNeil, O. Seidel, M. Spiteri.**

49. **Chebrolu, Kameswari.** [Online] http://home.iitk.ac.in/~chebrolu/scourse/slides/sockets-tutorial.pdf.

50. Introducing BizTalk Server 2010. *msdn.* [Online]

51. Web Services Architecture. *W3C.* [Online] http://www.w3.org/TR/ws-arch/.

52. SharePoint. [Online] http://sharepoint.microsoft.com/en-us/Pages/default.aspx.

53. Windows Communication Foundation. [Online] http://msdn.microsoft.com/en-us/netframework/aa663324.

54. A Universally Unique IDentifier (UUID) URN Namespace. [Online] http://www.ietf.org/rfc/rfc4122.txt.

55. **Hamilton, Keith.** Distributed Publish/Subscribe (Pub/Sub) Event System. *CodePlex.* [Online] http://pubsub.codeplex.com/releases/view/49398#DownloadId=136855.

56. ReadWriterLock Class. *msdn.* [Online] http://msdn.microsoft.com/en-us/library/system.threading.readerwriterlock.aspx.

57. TcpListener Class. *msdn.* [Online] http://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener.aspx.

58. Transaction Class. *msdn.* [Online] http://msdn.microsoft.com/en-us/library/system.transactions.transaction.aspx.

59. *Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems?* **Pietzuch, Peter R.**

60. *The Evolution of Publish/subscribe Comunication Systems.* **Roberto Baldoni, Mariangela Contenti, Antonio Virgillito.**

61. *Web Solutions Platform Event System: A Distributed Publish/Subscribe Event System.* **Hamilton, Keith.**

62. Introducing BizTalk Server 2010. *MSDN.* [Online] http://msdn.microsoft.com/en-us/library/aa547058(v=BTS.70).aspx.

63. *Apache Qpid.* [Online] http://qpid.apache.org/.

64. *Future Pattern.* [Online] http://help.kapowtech.com/8.1/index.jsp?topic=/doc/java/FuturePattern.html.

65. dofactory. *Observer.* [Online] http://www.dofactory.com/Patterns/PatternObserver.aspx.