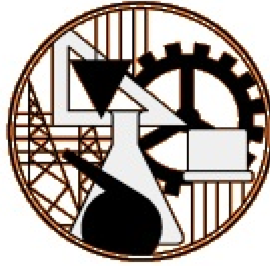


INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Mestrado em Engenharia Informática e de Computadores
(Ramo de Sistemas de Informação)

IMBus - Instant Messaging Bus

Carlos Filipe Figueira Vicente

(Licenciado em Engenharia Informática e de Computadores)

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

Orientador:

Mestre Jorge Martins

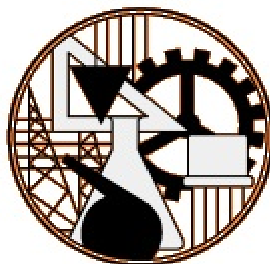
Júri:

Presidente: Mestre Vítor Almeida

Vogal: Mestre Pedro Félix

Vogal: Mestre Jorge Martins

18 de Dezembro de 2010



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Mestrado em Engenharia Informática e de Computadores
(Ramo de Sistemas de Informação)

IMBus - Instant Messaging Bus

Carlos Filipe Figueira Vicente
(Licenciado em Engenharia Informática e de Computadores)

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

Realizado por:

Carlos Filipe Figueira Vicente

Orientador:

Mestre Jorge Martins

18 de Dezembro de 2010

Resumo

Os principais sistemas de *instant messaging*, desenvolvidos pelos grandes produtores de software, têm protocolos proprietários e fechados, o que não permite que exista comunicação entre eles. Isto implica que um utilizador necessite de utilizar vários clientes de *instant messaging*, por forma a comunicar com os seus contactos em todas as redes.

Este trabalho tem como principal objectivo o desenvolvimento de um bus de *instant messaging*, que seja capaz de integrar vários serviços deste tipo. O *bus* permitirá a um utilizador comunicar com os seus contactos, também ligados ao *bus*, independentemente do seu serviço, usando apenas um cliente.

Foi realizado um estudo sobre os protocolos de *instant messaging*, tanto protocolos abertos, cujo objectivo assenta na interoperabilidade, como os serviços disponibilizados pelos grandes sistemas. Deste estudo destacou-se o protocolo aberto XMPP e o serviço de *instant messaging* da *Yahoo*, sendo estes os escolhidos para provar o objectivo do projecto.

A solução proposta tem por base um servidor que implementa parcialmente o protocolo XMPP, escolhido como formato nativo do *bus* de *instant messaging*. A interoperabilidade entre serviços de *instant messaging* é conseguida através de *Web Services* (designados por módulos de tradução), sendo cada um capaz de comunicar com outro serviço de *instant messaging*. O servidor disponibiliza também um *Web Service* (*Web Service* Central) que expõe o *bus* aos módulos de tradução. As operações do *Web Service* Central fazem a tradução para o protocolo nativo do *bus*. Desta forma existe um ponto único de processamento de funcionalidades (o servidor, processando pedidos feitos ao *Web Service* Central, por parte dos módulos), sendo todas as mensagens redireccionadas para o módulo respectivo ao utilizador destinatário.

Abstract

The most important instant messaging services, developed by the major software companies, have proprietary and closed protocols, not allowing communication between them. This fact implies that a given user must use several instant messaging clients so that he can communicate with all of his contacts, on the various networks.

This project's main goal is to develop an instant messaging bus, capable of integrating several instant messaging services. The bus allows for a user to communicate with all of his contacts connected to the bus, regardless of their service, using only one client for it.

A study on several instant messaging protocols was conducted, considering both open protocols, which are based on interoperability, and the services made available by the major systems. This study determined the use of the open protocol XMPP and the Yahoo's instant messaging service, to prove the projects main goal.

The solution proposed is based on a server that partially implements the XMPP protocol, defined as the native format to the instant messaging bus. The interoperability between instant messaging services is achieved through Web Services (called translation modules), where which one communicates with other instant messaging service. The server publishes a Web Service (Central Web Service) to expose the bus functionalities to the modules. The Central Web Service operations are translated to the native protocol used on the bus. This allows for a single point of functionality processing (the server, processing requests made to the Central Web Service by the modules), where all messages are redirected to the destination's respective module.

Palavras-chave

Integração; interoperabilidade; instant messaging; presença; serviços; Web; XMPP; protocolo aberto; Yahoo.

Keywords

Integration; interoperability; instant messaging; presence; services; Web; XMPP; open protocol; Yahoo.

Agradecimentos

Em primeiro lugar gostaria de agradecer aos meus pais, por todo o apoio que me deram para a realização do meu curso, tanto na Licenciatura como no Mestrado. É graças a eles que me foi possível passar estes 6 anos a estudar, de forma a concluir o ensino superior.

Agradeço também aos meus colegas de turma e de grupo, com quem realizei a maior parte do curso. Com o estudo e trabalho conjunto pude aprender de melhor forma, tornando-me assim um melhor profissional de Engenharia Informática e de Computadores.

Ao orientador deste projecto, o Engº Jorge Martins, agradeço a disponibilidade em me guiar nesta última etapa do Mestrado. Com a sua orientação o nível do trabalho atingiu um nível mais próximo ao melhor possível.

Por fim, mas não menos importante, quero agradecer à minha namorada, a Ana, por todo o apoio e compreensão que me deu, principalmente na fase final do trabalho.

Conteúdo

Lista de Figuras xiii

Lista de Tabelas xv

Lista de Listagens xviii

1 Introdução 1

1.1 *Instant Messaging* e *Presence* 1
1.2 Objectivos do trabalho 1
1.3 Descrição do conteúdo 3

2 XMPP 5

2.1 Princípios base 5
2.1.1 *Streams* XML persistentes 6
2.1.2 Arquitectura descentralizada 7
2.1.3 Endereços globais 7
2.1.4 Informação estruturada 8
2.1.5 Consciência de presença 8
2.1.6 Comunicação assíncrona 8
2.2 Negociação da ligação (*Stream*) 9
2.2.1 Transport Layer Security 10
2.2.2 Simple Authentication and Security Layer 10
2.2.3 Resource Binding 11
2.2.4 Estabelecimento de sessão 12
2.3 XML *stanzas* 13
2.4 Operações de *Instant Messaging* e *Presence* 13
2.4.1 Informação de presença 13
2.4.2 Troca de mensagens 16
2.4.3 Gestão do *roster* 17
2.4.4 Processamento dos *stanzas* em geral 21
2.5 Integração de serviços por *gateways* 21

3 *Yahoo! Messenger* 23
3.1 Autenticação . . . 24
3.2 Operações de *Instant Messaging* e *Presence* . . . 24
3.2.1 Criação de sessão . . . 25
3.2.2 Actualização de presença . . . 25
3.2.3 Envio de mensagens . . . 25
3.2.4 Obtenção de lista de contactos . . . 26
3.2.5 Adicionar contacto/pedido de subscrição . . . 27
3.2.6 Responder a pedido de subscrição . . . 27
3.2.7 Notificações . . . 28

4 Visão geral **29**
4.1 Problema encontrado . . . 29
4.2 Solução apresentada . . . 30
4.3 Arquitectura e sistema . . . 30
4.3.1 Descrição dos componentes . . . 31
4.3.2 Contrato do *Web Service* Central . . . 32
4.3.3 Contrato dos *Web Services* dos módulos . . . 33
4.4 Modelo de dados . . . 35
4.5 Implementação . . . 36

5 Implementação de componentes auxiliares **37**
5.1 Acesso a dados . . . 37
5.1.1 Sessão de ligação . . . 38
5.1.2 Objectos de acesso aos dados . . . 38
5.2 Objectos de sessão . . . 39
5.2.1 Elementos XMPP . . . 40
5.2.2 Escrita de elementos XMPP . . . 43
5.2.3 Leitura de elementos XMPP . . . 45
5.2.4 *Data Transfer Objects* . . . 46
5.2.5 Objectos de suporte ao *stream* . . . 46
5.3 Aplicação *Web* para registo . . . 46

6 Servidor XMPP **49**
6.1 Gestão de ligações TCP . . . 49
6.1.1 Recepção de ligações . . . 50
6.1.2 Assincronismo . . . 51
6.1.3 Terminação da ligação . . . 53
6.2 Gestão de sessões de *instant messaging* . . . 53
6.2.1 Negociação do *stream* . . . 54
6.2.2 Recepção de novo *stanza* . . . 58
6.2.3 Leituras e escritas . . . 59

6.2.4 Obtenção de informação de presença 61
6.3 Processamento de *stanzas* 62
6.3.1 Definição de processador 62
6.3.2 Processador de mensagens 62
6.3.3 Processador de presença 63
6.3.4 Processador de *info/query* 67
6.4 Ciclo de vida de um pedido 70

7 Integração com serviços existentes **73**
7.1 *IM Module Manager* 73
7.1.1 Implementação do *Web Service* Central 74
7.1.2 Transformações entre operações e objectos *stanza* 75
7.2 Caso de estudo: *Yahoo* 76
7.2.1 Gestão de sessões dos utilizadores 78
7.2.2 Comunicação com utilizador *Yahoo* 78
7.2.3 Acesso ao serviço de IM da *Yahoo* 82
7.2.4 *Web Service* gerado pelo WSDL 84

8 Conclusão **87**
8.1 Resultados obtidos 87
8.2 Comentários ao trabalho 88
8.3 Trabalho futuro 89

Bibliografia **91**

Referências **93**

A Transformações a estados de subscrição **95**

B WSDL para *Web Services* dos módulos de tradução **99**

C Hierarquia dos objectos de elementos XMPP **105**

D XML *stanzas* **107**
D.1 *Message* 107
D.2 *Presence* 108
D.3 *Info/Query* 109

Lista de Figuras

2.1 Exemplo de uma arquitectura XMPP 6
2.2 Diagrama de sequência para comunicação entre servidor e cliente XMPP . . . 9

3.1 Fluxo de autenticação através de OAuth adaptado 24

4.1 Arquitectura geral 31
4.2 Modelo de dados 35

5.1 Hierarquia de classes para transformação em XML 40
5.2 Hierarquia de fábricas para leitura de XML 41
5.3 Visualização da página de registo 47

6.1 Ilustração de uma entrada em ***connections*** 50
6.2 Ilustração dos dicionários de suporte a *Session Manager* 55
6.3 Diagrama de sequência para negociação do *stream* 56
6.4 Ciclo de vida de um *stanza* de mensagem 72

7.1 Aplicação cliente Yahoo 77
7.2 Ilustração do contentor *sessions* 78
7.3 Janela de conversação com a conta fantasma - lista de contactos e notificações de presença 80
7.4 Envio e recepção de mensagens na perspectiva de um utilizador *Yahoo* 80
7.5 *Roster* de um utilizador XMPP 81
7.6 Envio e recepção de mensagens na perspectiva de um utilizador XMPP 81
7.7 Processo de recepção e processamento de notificações 83

C.1 Hierarquia de elementos XMPP 106

Lista de Tabelas

2.1 Semântica dos atributos do elemento <stream:stream/> 10
2.2 Estados da subscrição de informação de presença 14

7.1 Transformações das operações de *Web Service* Central para objectos *stanza* . 76
7.2 Transformações de objectos *stanzas* em operações dos *Web Services* dos módulos 76

A.1 Transformações ao estado da subscrição de quem envia um *stanza 'subscribe'* 95
A.2 Transformações ao estado da subscrição de quem recebe um *stanza 'subscribe'* 95
A.3 Transformações ao estado da subscrição de quem envia um *stanza 'subscribed'* 95
A.4 Transformações ao estado da subscrição de quem recebe um *stanza 'subscribed'* 96
A.5 Transformações ao estado da subscrição de quem envia um *stanza 'unsubscribed'* 96
A.6 Transformações ao estado da subscrição de quem recebe um *stanza 'unsubscribed'* 96
A.7 Transformações ao estado da subscrição de quem envia um *stanza 'unsubscribe'* 97
A.8 Transformações ao estado da subscrição de quem recebe um *stanza 'unsubscribe'* 97

D.1 Atributos transversais a todos os tipos de *stanzas* 107
D.2 Descrição dos tipos de *stanzas* <message/> 108
D.3 Descrição dos tipos de *stanzas* <presence/> 109
D.4 Descrição dos tipos de *stanzas* <iq/> 110

Lista de Listagens

2.1 Elementos possíveis dentro de <stream:features/> 12
2.2 Exemplo de resposta a obtenção do *roster* 18
2.3 Exemplo de adição de um contacto ao *roster* 19
2.4 Exemplo de "*roster pushs*" 19
2.5 Exemplo de uma resposta a uma adição ao *roster* 19
2.6 Exemplo de uma actualização de um contacto no *roster* 20
2.7 Exemplo da remoção de um contacto do *roster* 20
3.1 Exemplo de um pedido para criação de sessão *Yahoo* 25
3.2 Exemplo de um pedido para actualizar a informação de presença 26
3.3 Exemplo do envio de mensagem para um utilizador 26
3.4 Exemplo da obtenção da lista de contactos 26
3.5 Exemplo da adição de um contacto à lista de contactos 27
3.6 Exemplo da aceitação de um pedido de subscrição 27
4.1 Exemplo da definição de uma operação *OneWay* em WSDL 34
5.1 Interface **ISession** 38
5.2 Classe base para todos os DAOs 38
5.3 Classe SimpleElement 40
5.4 Classe SimpleElementVariableChildren 42
5.5 Classe StanzaElement 43
5.6 Complemento da classe SimpleElement 44
5.7 Complemento da classe SimpleElementVariableChildren 44
5.8 Complemento da classe StanzaElement 45
5.9 Exemplo de envio de um elemento <features/> para o cliente 45
6.1 Excerto de código para recepção de uma ligação 51
6.2 *Callback* de leitura 52
6.3 *Callback* de escrita 53
6.4 Classe abstracta SaslMechanism 57
6.5 Algoritmo de autenticação 57
6.6 Método ***ThreadRun()*** 60
6.7 Método ***ContinueListening()*** 60
6.8 Classe abstracta **StanzaProcessor** 62
6.9 Classe **RequestManager** 63

6.10 Processamento de mensagens 64
6.11 Método ***Process()*** de **PresenceProcessor** 65
6.12 Método ***BroadcastPresence()*** de **PresenceProcessor** 66
6.13 Método ***DirectedPresence()*** de **PresenceProcessor** 67
6.14 Método ***Subscribe()*** de **PresenceProcessor** 67
6.15 Método ***Subscribed()*** de **PresenceProcessor** 68
6.16 Método ***Subscribed()*** de **PresenceProcessor** 68
6.17 Excerto do método ***Process()*** de **IqProcessor** 69
6.18 Delegate **ProcessInner** 69
6.19 Método ***GetRoster()*** de **RosterManager** 70
6.20 Método ***SetRoster()*** de **RosterManager** - adicionar contacto 70
6.21 Método ***SetRoster()*** de **RosterManager** - actualizar contacto 71
6.22 Método ***SetRoster()*** de **RosterManager** - remover contacto 71
7.1 Interface **IModuleManager** 74
7.2 Enumerado **PresenceState** 74
7.3 Declaração da classe **ModuleManager** 75
7.4 Classe **YahooSession** 78
7.5 Classe **Credentials** 83
7.6 Interface gerada através do ficheiro WSDL (em *Java*) 85
B.1 WSDL com contrato a implementar pelos módulos de tradução 99
D.1 Exemplo de obtenção do *roster* 110

Capítulo 1

Introdução

1.1 *Instant Messaging* e *Presence*

Instant Messaging (IM) é uma forma de comunicação entre entidades (pessoas ou não), em tempo quase-real, através da *internet*, utilizando um leque diverso de dispositivos. A sua utilização mais comum é em aplicações de conversação para computadores, onde utilizadores trocam curtas mensagens escritas.

Em IM, tal como em email, existe uma lista de contactos (*roster*), onde estão armazenados, para cada utilizador, os contactos com quem este deseja comunicar. A diferença entre esta lista de contactos e uma lista de contactos de email é o conceito de presença (*presence*). Presença entende-se como a disponibilidade de comunicação que um utilizador indica ter. É uma característica chave dos sistemas de IM, que possibilita comunicação em tempo quase-real.

Os sistemas de IM foram rapidamente adoptados tanto para utilização social como empresarial. Considerando a natureza da informação transferida, seja informação pessoal ou até mesmo segredos de negócio, estes sistemas devem ser bem protegidos, usando, por exemplo, esquemas de autenticação e cifra.

1.2 Objectivos do trabalho

O projecto consiste no desenvolvimento de uma camada de software de *instant messaging* (IM) capaz de interoperar com os serviços de IM já existentes. Esta camada deve possibilitar aos seus clientes a execução das funções básicas de qualquer sistema de IM (segundo [12]):

1. Trocar mensagens com outros utilizadores.
2. Trocar informação de presença com outros utilizadores.
3. Gerir subscrições de presença de e para outros utilizadores.
4. Gerir os elementos de uma lista de contactos.
5. Bloquear comunicações de e para utilizadores específicos.

Após um estudo mais detalhado nesta área foram encontrados dois protocolos abertos, com o objectivo de permitir interoperabilidade entre vários serviços de *instant messaging*. Esses protocolos são *eXtensible Messaging and Presence Protocol* (XMPP)[1], definido pelas especificações RFC3920[11] e RFC3921[12] (esta última define as extensões para *instant messaging*), e *SIP for Instant Messaging and Presence Leveraging Extensions* (SIMPLE), definido pelas especificações RFC3265[9], RFC3856[10], RFC3428[3] e RFC4975[2]. Ambos foram definidos pela *Internet Engineering Task Force* (IETF) e são amplamente utilizados.

O protocolo SIMPLE tem uma utilização mais direccionada para integração de *instant messaging* com comunicação multimédia. Não sendo esse o foco do trabalho e considerando que XMPP é o mais simples e extensível dos dois, a implementação do sistema será então realizada em XMPP. Esta decisão não impossibilita uma futura integração de comunicação multimédia no sistema, pois existe uma extensão para XMPP com esse intuito (*Jingle*), embora isso não seja considerado no âmbito deste projecto.

Existe uma extensão para o protocolo XMPP cujo objectivo passa por definir a utilização do mesmo para integração com serviços de *instant messaging* já existentes. Contudo, como será explicado mais adiante, esta extensão não é solução para o problema de integração que este projecto visa resolver, não sendo, por isso, utilizada. Desta forma a interoperabilidade com outros serviços, tais como *Yahoo!Messenger*, *AIM* e *Live Messenger*, será conseguida através de módulos de conversão. Estes módulos devem traduzir a informação XMPP para acesso ao serviço não XMPP e vice-versa. Desta forma será conseguida uma unificação de serviços de *instant messaging*.

A base deste projecto, e como tal o seu objectivo mínimo, passa pelo desenvolvimento de um servidor de *instant messaging* (na *framework* .NET, mais concretamente em C#) que respeite parcialmente o protocolo XMPP. A implementação do protocolo XMPP é apenas parcial, pois o objectivo do projecto não é centrado neste protocolo, mas sim na integração de serviços de *instant messaging*. Desta forma o desenvolvimento do servidor XMPP é apenas um meio para atingir o fim. Juntamente com o servidor também será implementada uma pequena aplicação Web (com a *framework* ASP.NET), para registo dos utilizadores no servidor.

De forma a atingir a finalidade do projecto, sendo assim os objectivos opcionais, fica a implementação de módulos de conversão para um ou mais serviços existentes.

O objectivo final e ideal do projecto será ter um cliente XMPP a comunicar com um cliente *Live Messenger* ou um cliente *AIM* a comunicar com um cliente *Yahoo!Messenger*.

[1]Apresentado no capítulo 2

1.3 Descrição do conteúdo

Capítulo 1 - Introdução

O capítulo actual, denominado de ”Introdução”, dá uma breve introdução sobre os conceitos mais importantes de *instant messaging* e apresenta os objectivos do trabalho. Aqui são também apresentadas breves descrições dos capítulos seguintes, de forma a apresentar a perspectiva do documento, no relato do trabalho efectuado.

Capítulo 2 - XMPP

Os capítulos 2 e 3 servem para contextualizar o leitor para os pormenores da implementação do trabalho.

Este capítulo descreve o protocolo XMPP, apresentando os seus principais conceitos e definições. A negociação da ligação (*stream* XML), tal como os elementos utilizados para o efeito e os elementos utilizados durante a vida do *stream* são aqui apresentados. São ainda descritos os procedimentos necessários para realizar as operações de *instant messaging* e *presence* e uma extensão do protocolo para utilização de *gateways*.

Capítulo 3 - Yahoo!Messenger IM SDK

O capítulo 3 apresenta uma breve descrição sobre a forma de utilização do serviço para *instant messaging* da *Yahoo*, relativamente a autenticação e principais operações de *instant messaging* e *presence*.

Capítulo 4 - Visão geral

Neste capítulo é apresentada uma visão geral sobre o problema que o trabalho visa resolver, juntamente com o desenvolvimento da solução por ele proposta. Nomeadamente a arquitectura da camada (com uma breve descrição dos componentes desenvolvidos) e o modelo de dados utilizado.

Capítulo 5 - Implementação de componentes auxiliares

Este capítulo apresenta alguns pormenores de implementação dos componentes auxiliares, nomeadamente sobre a camada de acesso a dados e objectos utilizados numa sessão XMPP. Serve apenas para ajudar a compreensão dos capítulos de implementação do servidor.

Aqui é também apresentada a aplicação *Web* desenvolvida para registo de utilizadores na camada, tanto para utilizadores XMPP como utilizadores de outros serviços.

Capítulo 6 - Servidor XMPP

O capítulo 6 é o principal capítulo do trabalho, descrevendo as principais operações e componentes do servidor, para suportar operações de *instant messaging* e *presence*, relacionando-as

com os conceitos e processos apresentados no capítulo 2. Este capítulo apresenta o resultado do objectivo mínimo.

Capítulo 7 - Integração com serviços existentes

Em forma de conclusão da implementação e para atingir o objectivo opcional do trabalho, o capítulo 7 apresenta a implementação da abordagem para integração do servidor desenvolvido com outros serviços já existentes. Como caso de estudo, é apresentada a integração com o serviço de *instant messaging* da *Yahoo*.

Capítulo 8 - Conclusões

Por fim são apresentadas as conclusões e comentários sobre os resultados atingidos no trabalho.

Capítulo 2

XMPP

Com o aumento da popularidade dos sistemas de *Instant Messaging*, cresceu também o interesse dos grandes produtores de software. O primeiro grande sistema a ser lançado a nível social foi o *AOL Instant Messaging* (AIM), sendo seguido de *MSN Messenger*, *Yahoo! Messenger* e *IBM Lotus Sametime*. Todos estes sistemas possuem protocolos proprietários e fechados, obrigando a utilizar várias aplicações clientes para comunicar com utilizadores de vários sistemas. Isto incentivou a definição de protocolos *open-source*. Os principais são o *eXtensible Messaging and Presence Protocol* (XMPP) e o *SIP for Instant Messaging and Presence Leveraging Extensions* (SIMPLE). Em seguida o protocolo XMPP será descrito em maior profundidade.

O *eXtensible Messaging and Presence Protocol* é um protocolo aberto para troca de elementos XML entre duas quaisquer entidades, em tempo quase-real. O protocolo define dois tipos de elementos XML, aqueles que são usados para a negociação da ligação (*stream* XML) e aqueles que são usados para *instant messaging*. Os elementos que são usados para *instant messaging* chamam-se "XML *stanzas*". Este protocolo foi inicialmente desenvolvido no âmbito da comunidade *open-source Jabber* em 1999. Em 2002 foi criado o XMPP *Working Group* junto da *Internet Engineering Task Force* (IETF), para a definição do *core* e outros componentes do protocolo XMPP, nomeadamente as especificações RFC 3920[11] e RFC 3921[12].

2.1 Princípios base

O protocolo XMPP é implementado através de uma arquitectura cliente-servidor descentralizada (semelhante à arquitectura de email), onde cada cliente cria uma ligação a um servidor para aceder à rede. Uma rede XMPP pode ser constituída por um ou vários servidores e ainda conter *gateways*[1] para servidores de outros protocolos. A figura 2.1 apresenta um exemplo de uma arquitectura XMPP.

[1] Um *gateway* serve para traduzir o protocolo XMPP para outro protocolo

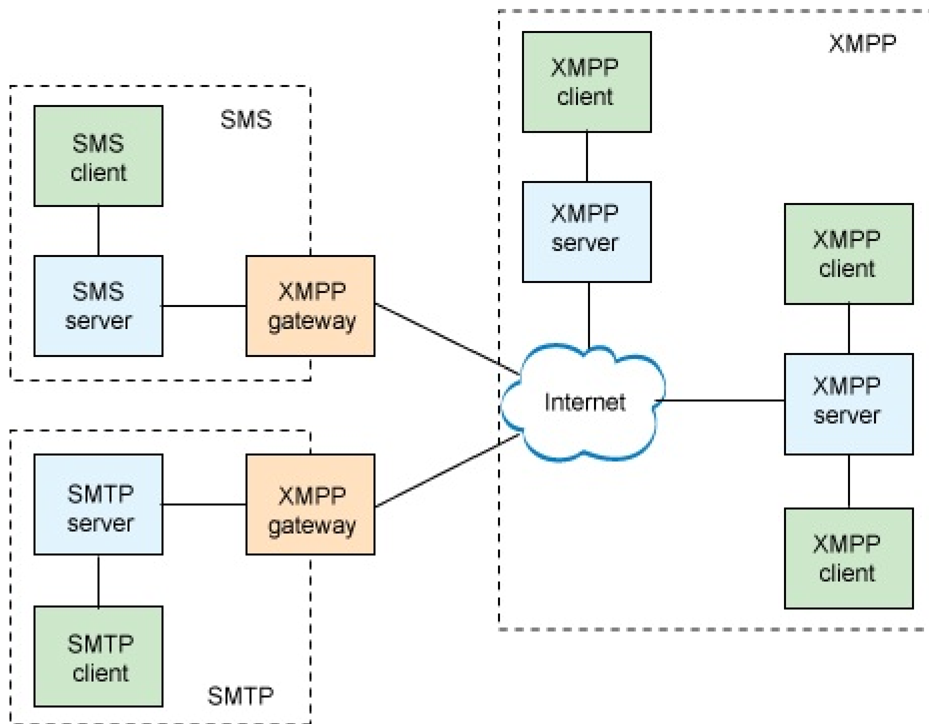


Figura 2.1: Exemplo de uma arquitectura XMPP. Fonte: *Meet the Extensible Messaging and Presence Protocol (XMPP)*[6]

A comunicação entre duas entidades é realizada de forma **assíncrona**, enviando **informação estruturada** através de ***streams* XML persistentes**[2], numa **arquitectura descentralizada** de **endereços globais**, entre clientes e servidores **conscientes da presença** de outras entidades. Estas noções são pormenorizadas de seguida.

2.1.1 *Streams* XML persistentes

O cliente estabelece um *stream* XML com o servidor para realizar operações de *instant messaging*. Esse *stream* é unidireccional, obrigando o servidor a criar um *stream* de resposta para o cliente. A negociação dos *streams* é um processo que envolve várias operações, como por exemplo negociação para cifra do canal (segurança) e autenticação. Considerando que os *streams* são usados para enviar muitas mensagens curtas, devendo estas ser entregues o mais rápido possível, é preferível mantê-los abertos (daí serem chamados *streams* persistentes). Assim quando uma entidade desejar enviar uma mensagem a outra, esta última irá recebê-la o mais rapidamente possível, realizando comunicação em tempo quase-real.

Um *stream* pode ser considerado como um documento XML em construção, no qual é possível colocar um número ilimitado de elementos, sendo representado pelo elemento raiz <stream/>. No *stream* podem ser utilizados elementos para negociação do mesmo (usando

[2] A referência a persistência indica neste caso longa vida e não permanência

Transport Layer Security (TLS)[4] e *Simple Authentication and Security Layer* (SASL)[8]) ou XML *stanzas*. Enquanto a negociação do *stream* não estiver concluída, não podem ser enviados nem recebidos quaisquer *stanzas* (excepto o *stanza* de IQ, quando usado para alguns passos da negociação do *stream*).

2.1.2 Arquitectura descentralizada

Uma rede XMPP é constituída por vários clientes e servidores, que comunicam entre si. Um cliente pode ligar-se a qualquer servidor da rede, daí este tipo de arquitectura ser considerada descentralizada. Assim a comunicação entre duas entidades (logicamente *peer-to-peer*), ligadas a servidores diferentes, é na realidade comunicação *client-to-server-to-server-to-client*.

Responsabilidades de um servidor

Um servidor é responsável pela gestão de ligações/sessões (*streams* XML), redireccionar de forma apropriada XML *stanzas* para os respectivos destinatários e pela gestão de contas de utilizadores (p.ex.: armazenar listas de contactos). A porta TCP recomendada pela especificação [11] para que o servidor ”escute” ligações de clientes é a 5222.

2.1.3 Endereços globais

Numa rede XMPP todas as entidades têm um endereço, que contém, para além da identificação da entidade (`nó`), a identificação do servidor (`domínio`) ao qual a entidade se encontra ligada. A este endereço chama-se *JabberID* (JID). Assim o JID de um cliente de nome ”utilizador” no servidor ”servidor.pt” será ”utilizador@servidor.pt”. Desta forma é possível identificar a que servidor um cliente se encontra ligado. Genericamente o JID pode ser apresentado na forma <nó@domínio>.

Um mesmo utilizador (conta única) poderá ter vários clientes ligados simultâneamente ao mesmo servidor, desde que cada um represente um `recurso` diferente. Desta forma um utilizador consegue comunicar com um determinado *device*, que esteja ligado através da sua conta. Este `recurso` é indicado no final do endereço, sendo escolhido pelo utilizador ou automaticamente atribuído pelo servidor. Cada um dos `recursos` é um ”ponto de presença” diferente, com as suas próprias características. Por exemplo: o utilizador com o endereço ”utilizador@servidor.pt” poderá estar ligado num computador em casa, com o endereço total ”utilizador@servidor.pt/casa” e com um computador no trabalho, com o endereço total ”utilizador@servidor.pt/trabalho”. A um endereço com parte de `recurso` chama-se *full* JID. Genericamente o *full* JID pode ser apresentado na forma <nó@domínio/recurso>.

A parte `nó` do *full* JID deve ser validada pelo perfil *Nodeprep* da especificação *Stringprep* (definido em [5]) e a parte `recurso` pelo perfil *Resourceprep* da mesma especificação.

2.1.4 Informação estruturada

Este protocolo tem como objectivo principal fornecer uma forma de *streaming* de XML. Assim toda a comunicação é realizada através de ligações TCP, chamadas *streams* XML, onde apenas são trocados blocos de informação em XML. Estes *streams* podem ser então vistos como documentos XML, que vão sendo construídos ao longo da vida da ligação. A unidade mais básica de comunicação em XMPP é o *stanza*, que é apenas um fragmento de XML enviado através de um *stream*. Existem três tipos de *stanzas*: <message/>, <presence/> e <iq/>. Estes tipos serão descritos em maior pormenor mais à frente.

Extensibilidade

A comunicação em XMPP é realizada com XML, que só por si é extensível, o que torna possível a utilização de elementos que à partida não eram previstos. Os próprios *schemas* XML dos *namespaces* ”jabber:client” e ”jabber:server” deixam espaço à extensibilidade, possibilitando que sejam usados elementos de quaisquer *namespaces* dentro dos *stanzas*. A extensibilidade é uma das características que tornam o XMPP mais apelativo, pois torna possível a utilização de novos elementos dentro dos *stanzas* já existentes.

2.1.5 Consciência de presença

Em XMPP uma entidade pode anunciar a outras entidades a sua disponibilidade para comunicação, também chamada de ”presença”. Com esta informação as outras entidades têm consciência da sua disponibilidade e podem assim iniciar comunicação.

2.1.6 Comunicação assíncrona

Em XMPP toda a comunicação é assíncrona, o que permite que um cliente envie vários pedidos ao servidor e que seja notificado à medida que esses pedidos sejam atendidos, sem ficar bloqueado à espera de uma resposta a determinado pedido. Tendo em conta a própria natureza de *instant messaging*, não faria muito sentido, pois poderia receber uma mensagem enquanto se encontrava à espera de uma resposta.

Nem todos os *stanzas* necessitam de resposta, sendo que a maioria é simplesmente enviada para o servidor, assumindo que tudo se processou como devido, de acordo com a natureza assíncrona do sistema. O cliente é notificado da ocorrência de um erro com um determinado *stanza*, quando recebe uma mensagem de erro a ele associado. Esta associação é conseguida através do atributo ***id*** do *stanza*, que o identifica.

A figura 2.2 apresenta um diagrama de sequência UML, que descreve uma possível comunicação entre um servidor e um cliente XMPP.

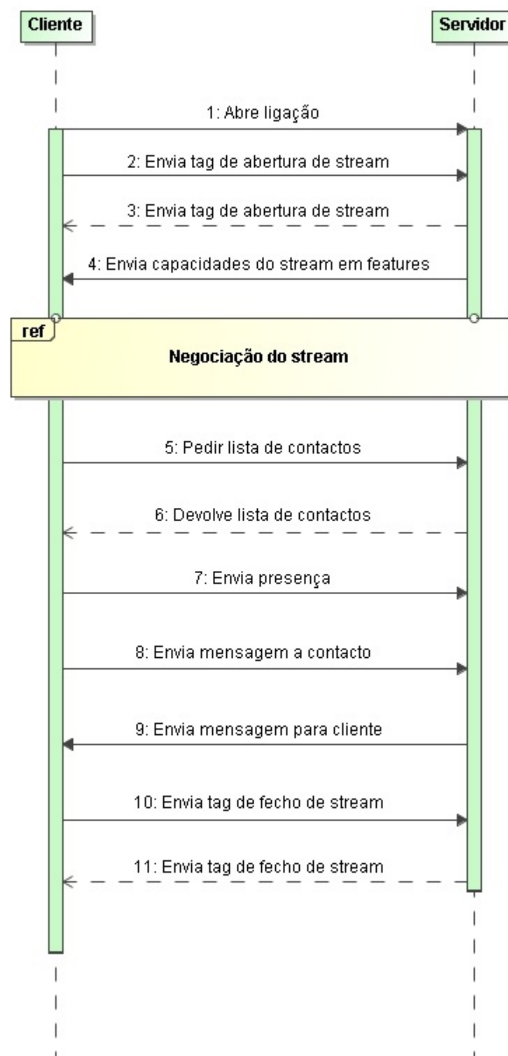


Figura 2.2: Diagrama de sequência para comunicação entre servidor e cliente XMPP

2.2 Negociação da ligação (*Stream*)

Toda a comunicação é realizada através do *stream* XML, utilizando XML *stanzas*, mas antes que possam ser trocados *stanzas* pelo *stream*, é necessário negociar o próprio *stream*.

Em XMPP é usual que a negociação do *stream* seja composto por 4 operações: cifra do canal de comunicação, autenticação do utilizador, associação de um `recurso` ao *stream* (*resource binding*) e estabelecimento de uma sessão de *instant messaging*. As operações de *resource binding* e de estabelecimento de sessão utilizam *stanzas* <iq/>, estando já parte do processo de negociação do *stream* concluído. A utilização de qualquer tipo de *stanzas* é proibido nas fases de negociação dos esquemas de cifra e autenticação do utilizador. O protocolo define os procedimentos e elementos XML a utilizar para a realização destas operações.

Tanto no início da negociação, como no final bem sucedido da negociação TLS e auten-

ticação SASL, devem ser abertos novos *streams* (*streams* lógicos dentro da mesma ligação TCP). O cliente começa por enviar a *tag* de abertura do novo elemento <stream:stream/>, com o *namespace* 'jabber:client', em seguida o servidor realiza o mesmo, utilizando o mesmo *namespace* (definido pelo standard para comunicação entre clientes e servidores). O prefixo *'stream'* está associado ao *namespace* 'http://etherx.jabber.org/streams'. É necessário ter atenção aos valores dos atributos deste elemento, cuja semântica se encontra na tabela 2.1.

	Iniciadora para receptora	Receptora para iniciadora
to	hostname do receptor	ignorado
from	ignorado	hostname do receptor
id	ignorado	chave de sessão
xml:lang	língua por omissão	língua por omissão
version	suporte a XMPP 1.0	suporte a XMPP 1.0

Tabela 2.1: Semântica dos atributos do elemento <stream:stream/>

O servidor tem que anunciar as operações de negociação para que o cliente as possa iniciar. O anúncio é realizado através do elemento <stream:features/> que indica quais as operações que o cliente pode iniciar, tendo em conta o estado actual do *stream* (enquanto não for realizada autenticação não é possível associar um `recurso` à ligação). Qualquer outro elemento XML enviado pelo cliente, não correspondente às operações que podem ser efectuadas de momento, irá resultar em falha.

Em seguida as operações atrás enunciadas são descritas em maior detalhe.

2.2.1 Transport Layer Security

O protocolo define um método para garantir a confidencialidade do canal, através de TLS (definido na especificação RFC 2246[4]), juntamente com uma extensão "STARTTLS", representada pelo elemento <starttls/>, cujo *namespace* é 'urn:ietf:params:xml:ns:xmpp-tls'. A necessidade de proceder à negociação TLS é anunciada pelo servidor, enviando o elemento <starttls/>, com <required/> como filho, indicando desta forma a obrigatoriedade da operação. Se o elemento <required/> não for especificado pelo servidor, a decisão de utilização de TLS ficará ao cargo do cliente. O *namespace* define ainda o elemento <proceed/>, que o servidor deve enviar ao cliente, informando-o para prosseguir com a negociação TLS.

Após a terminação com sucesso desta operação da negociação, o cliente deve-se autenticar perante o servidor.

2.2.2 Simple Authentication and Security Layer

Este protocolo define um perfil que adapta o protocolo SASL (definido na especificação RFC 2222[8]) para autenticar um *stream* (o cliente ligado através desse *stream*). Perfil esse que se encontra definido no *namespace* 'urn:ietf:params:xml:ns:xmpp-sasl'.

O protocolo SASL define que a autenticação é alcançada através da troca e processamento bem sucedidos de uma série de desafios e respostas. A estrutura das mensagens de desafio e

resposta, tal como o seu conteúdo, são definidos pelos vários mecanismos SASL existentes. Para autenticação baseada em pares {*username* e *password*}, estão definidos os seguintes mecanismos:

- PLAIN: este mecanismo envia em claro o *username* e *password* do utilizador, sem necessidade de qualquer desafio, e está definido no RFC 4616[17];
- DIGEST-MD5: este mecanismo implica que o cliente consiga decifrar 2 desafios até que possa ser considerado autenticado. O primeiro desafio consiste numa enumeração de configurações que o cliente pode escolher para este mecanismo. Após esta escolha, o cliente envia uma mensagem de resposta, de acordo com as configurações escolhidas. Nesta resposta está contido o *hash* MD5 (Message-Digest algoritmo 5) do *username* e respectiva *password*, de forma a provar ao servidor que o utilizador conhece esses dados. Ao validar esta resposta, o servidor envia um novo desafio que indica que a autenticação foi bem sucedida. Este mecanismo está definido na especificação RFC 2831[7].

Os mecanismos são anunciados ao cliente através do elemento <mechanisms/>, que terá um elemento <mechanism/> com o nome de cada mecanismo que o servidor disponibiliza. O cliente irá escolher um mecanismo e indicar essa escolha ao servidor através do elemento <auth/>, que pode conter uma resposta inicial, se o mecanismo assim o definir (como é o caso do PLAIN). Os desafios e respostas são enviados dentro dos elementos <challenge/> e <response/>, respectivamente. A indicação de uma autenticação bem sucedida é feita através do elemento <success/>.

Após a autenticação o cliente deve associar o *stream* a um `recurso` (*resource binding*), por forma a respeitar as regras de endereçamento definidas na especificação RFC 3920[11].

2.2.3 Resource Binding

Ao associar um `recurso` a um *stream* (*resource binding*), garante-se que esse *stream* é unicamente identificado através de um *full* JID. O servidor anuncia que permite *resource binding* através do elemento <bind/>, definido no *namespace* 'urn:ietf:params:xml:ns:xmpp-bind'. De forma a realizar esta operação, o cliente deve enviar o elemento <bind/>, que poderá estar vazio ou ter um elemento <resource/> com o `recurso` desejado. Caso esteja vazio, o servidor terá que gerar um `recurso` único para o utilizador. Caso seja fornecido o elemento <resource/>, o servidor pode rejeitar o valor fornecido, nas seguintes condições:

- Se o `recurso` não está de acordo com o perfil *Resourceprep*;
- Se não for permitido ao cliente associar mais `recursos` a um *stream* (por exemplo se atingir o número máximo de `recursos` ligados);
- Se o `recurso` escolhido já está em utilização e o servidor não permite vários *streams* para o mesmo identificador.

Na inexistência das referidas situações de erro, o `recurso` será aceite, sendo esse acontecimento comunicado ao cliente com um elemento <bind/> com um elemento <jid/> como filho, contendo o *full* JID associado ao *stream*.

A partir do momento em que o cliente escolhe realizar *resource binding*, a transferência dos elementos <bind/> é realizada dentro de elementos <iq/> (descrito em D.3).

A maioria dos servidores XMPP necessitam que o cliente crie uma sessão para realizar as tarefas associadas a *instant messaging* e *presence*. Assim após realizar *resource binding* pode prosseguir para estabelecimento de uma sessão com o servidor.

2.2.4 Estabelecimento de sessão

A especificação[11] não obriga a criação de uma sessão de *instant messaging* para realizar operações desse género. Mas este passo deve ser incluído para garantir conformidade com os clientes que exijam essa mesma condição.

Se um servidor suporta a criação de sessões, deve anunciá-lo com o elemento <session/>, definido no *namespace* 'urn:ietf:params:xml:ns:xmpp-session'. O cliente indica que quer estabelecer uma sessão enviando um elemento <session/> vazio, dentro de um elemento <iq/> (descrito em D.3). Ao qual o servidor apenas responde com um elemento <iq/> de resultado (valor '*result*' para o atributo *type*), que indica que a sessão foi bem estabelecida.

Após realizar *resource binding*, o *stream* está pronto a ser usado para *instant messaging*, assim esta última operação serve apenas para finalizar o processo. Isto dá a entender que esta operação é desnecessária, mas deve ser incluída por compatibilidade com clientes e servidores mais antigos.

Resumindo, os elementos que se podem encontrar dentro do elemento <stream:features/> são os apresentados na listagem 2.1.

```
<stream:features>

 <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'>
  <required/>
 </starttls>

 <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
  <mechanism>DIGEST-MD5</mechanism>
  <mechanism>PLAIN</mechanism>
 </mechanisms>

 <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>

 <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
</stream:features>
```

Listagem 2.1: Elementos possíveis dentro de <stream:features/>

Após uma breve descrição do processo de negociação do *stream*, percebe-se que são en-

volvidas várias operações. Assim, para que todas as comunicações sejam realizadas o mais rápido possível, em tempo quase-real, torna-se preferível manter o *stream* aberto (persistente), do que negociá-lo de cada vez. Tal como já foi referido anteriormente em 2.1.1.

2.3 XML *stanzas*

A unidade básica de comunicação em sessões XMPP é o XML *stanza* (tal como foi referido em 2.1.4). Estão definidos 3 tipos de *stanzas* nos *namespaces* 'jabber:client' e 'jabber:server': <message/>, <presence/> e <iq/>. Estes *stanzas* têm objectivos claramente distintos, mas têm uma construção bastante semelhante, tendo inclusivé os mesmos atributos. Esses atributos são: ***to***, ***from***, ***id***, ***type*** e ***xml:lang***.

Em seguida é apresentada uma breve descrição de cada um dos tipos de *stanzas*.

- <message/>: serve para enviar informação de uma entidade para outra. Existem cinco tipos de mensagens diferentes, sendo o seu tipo indicado pelo atributo ***type***. O elemento pode ter os filhos <subject/>, <body/> e <thread/>;

- <presence/>: serve para indicar, inquirir e subscrever a disponibilidade para comunicação de um determinado utilizador. Se o atributo ***type*** não for especificado, a disponibilidade do utilizador será indicada pelos elementos filhos do *stanza*, que poderão ser <show/>, <status/> e <priority/>;

- <iq/>: fornece um mecanismo de pedido-resposta, em que uma entidade realiza um pedido e mais tarde irá receber a resposta a esse pedido.

Uma descrição mais completa poderá ser encontrada no apêndice D.

2.4 Operações de *Instant Messaging* e *Presence*

As principais operações de *instant messaging* e *presence* são, tal como referido na secção 1.2, troca de mensagens, envio de informação de presença, gestão de subscrições e listas de contactos e ainda bloqueio de comunicações. A maioria destas operações são descritas de seguida com mais pormenor, tal como são apresentadas na especificação RFC3921[12].

2.4.1 Informação de presença

O *stanza* de presença tem como objectivos informar os outros utilizadores da disponibilidade de um determinado utilizador, por broadcast ou de forma direccionada, e ainda gerir os estados das subscrições de presença.

Um *stanza* de presença indica a disponibilidade de conversação quando o seu atributo ***type*** não é especificado de todo, contendo, neste caso, elementos filhos com a informação do seu estado. Os possíveis estados de presença são apresentados no apêndice D, em D.2, na

descrição dos elementos <show/> e <status/>. A indicação de indisponibilidade para conversação é indicada através de um *stanza* de presença com o valor *'unavailable'* para o atributo ***type***.

A gestão das subscrições de presença é conseguida usando *stanzas* de presença com os valores *'subscribe'*, *'subscribed'*, *'unsubscribe'* e *'unsubscribed'* para o atributo ***type***. Desta forma poderá ser alterado o estado da subscrição que um utilizador tem da informação de presença de outro.

A cada elemento do *roster* de um utilizador encontra-se associado o estado da subscrição da informação de presença do contacto, do ponto de vista do utilizador. Esse estado pode ter os valores apresentados na tabela 2.2.

none	Nenhum tem subscrição para a informação de presença do outro
from	O contacto tem subscrição para a informação de presença do utilizador, mas o utilizador não tem subscrição para a informação de presença do contacto
to	O utilizador tem subscrição para a informação de presença do contacto, mas o contacto não tem subscrição para a informação de presença do utilizador
both	Ambos têm subscrições para a informação de presença do outro

Tabela 2.2: Estados da subscrição de informação de presença

Associado ao estado da subscrição encontra-se a informação de pedido de subscrição pendente, tanto do lado do utilizador como do lado do contacto. Esta informação indica se foi realizado um pedido de subscrição da informação de presença do utilizador (*"pending in"*) e/ou se foi realizado um pedido de subscrição da informação de presença do contacto (*"pending out"*). Considerando a semântica dos estados de subscrição, juntamente com a informação de pedidos pendentes, existem as seguintes combinações: "*none*", "*none + pending in*", "*none + pending out*", "*none + pending in/out*", "*from*", "*from + pending out*", "*to*", "*to + pending in*" e "*both*".

Presença inicial

Após criar uma sessão de *instant messaging*, o cliente deve enviar a informação de presença inicial, por forma a indicar que está disponível para comunicação. Esta presença inicial consiste num *stanza* de presença, sem os atributos ***to*** e ***type***. Depois de enviar a informação de presença inicial, diz-se que o `recurso` se encontra "disponível".

Ao receber a informação de presença inicial, o servidor deve:

1. Armazenar essa informação;

2. Obter a informação de presença de todos os `recursos` dos seus contactos (com sessões criadas) cuja informação de presença é subscrita pelo utilizador (aqueles com subscrição "*to*" ou "*both*") e enviar-lhe essa informação;

3. Enviar a informação de presença a todos os seus contactos que a subscrevam (aqueles com subscrição "*from*" ou "*both*"), indicando o *full* JID do `recurso`;

4. Enviar a informação de presença a todos os seus `recursos`.

O servidor deve armazenar a informação de presença *broadcasted* pelo cliente (a presença inicial é um *broadcast*), de forma a poder enviá-la aos seus contactos que a subscrevam, sem voltar a pedi-la ao cliente.

Dos utilizadores a quem se envia e de quem se recebe informação de presença, devem ser excluídos aqueles que a quem o utilizador bloqueia envio e recepção de notificações de presença, respectivamente.

Broadcast de presença

O utilizador pode alterar a sua informação de presença a qualquer momento, bastando para isso enviar um novo *stanza* de presença. Este *stanza* não possui o atributo ***to*** e pode ou não possuir o atributo ***type***, mas caso seja especificado, apenas poderá ter o valor *'unavailable'*.

A forma de realizar um broadcast de presença é semelhante à forma de enviar a informação de presença inicial, visto que esse envio é na sua essência um broadcast de presença. A diferença encontra-se no envio de informação de indisponibilidade (*stanza* com valor *'unavailable'* no atributo ***type***). Neste caso, o *stanza* deve ser enviado também aos contactos a quem o utilizador enviou presença direccionada.

Presença direccionada

Envio de presença direccionado, é tal como o próprio nome indica, o envio de informação de presença a um determinado utilizador. Isto é conseguido, indicando o JID do utilizador no atributo ***to*** do *stanza* de presença.

O processamento de um *stanza* de presença direccionada ocorre das seguintes formas, nas diversas situações:

- Após o envio da presença inicial e antes do broadcast de indisponibilidade, se o contacto se encontra no *roster* do utilizador com subscrição "*from*" ou "*both*", o *stanza* deve ser enviado para o contacto (não sendo realizada qualquer alteração à informação de presença para *broadcast*). O contacto será incluído em novos *broadcasts* de presença.

- Após o envio da presença inicial e antes do *broadcast* de indisponibilidade, se o contacto não se encontra no *roster* do utilizador com subscrição "*from*" ou "*both*", o *stanza* deve ser enviado para o contacto (não sendo realizada qualquer alteração à informação de presença para *broadcast*). Neste caso, o contacto apenas será incluído em novo *broadcast* de presença, se esse for de indisponibilidade, caso isso ainda não tenha sido realizado de forma direccionada.

- Se o envio ocorre antes do envio da presença inicial ou depois do *broadcast* de indisponibilidade, o processamento será o mesmo que na situação anterior.

Indisponibilidade

Antes do término de uma sessão, o cliente deve enviar informação de indisponibilidade, tornando-se graciosamente indisponível. O que é conseguido com o envio de um *stanza* de presença com o valor *'unavailable'* no atributo ***type***. Contudo, o servidor não pode depender disso, pois o cliente pode terminar de forma abrupta ou até mesmo a ligação ser quebrada por outra qualquer razão. Assim, quando uma sessão não for graciosamente terminada, o servidor deve realizar *broadcast* de indisponibilidade do utilizador por conta própria. Esse *broadcast* é realizado a todos os contactos no *roster* do utilizador com subscrição "*from*" ou "*both*" (a quem o utilizador não bloqueou o envio de notificações de presença), a todos os seus `recursos` e ainda aos contactos a quem foi enviada informação de presença direccionada.

Gestão de subscrições de presença

A gestão de subscrições é realizada, tal como dito anteriormente, com recurso a *stanzas* de presença com os valores *'subscribe'*, *'subscribed'*, *'unsubscribe'* e *'unsubscribed'* para o atributo ***type***. O significado de cada um destes valores é apresentado na tabela D.3.

O envio e recepção destes *stanzas* implica transformações em alguns estados de subscrição, tanto no *roster* de quem envia como no *roster* de quem recebe. Estas transformações são apresentadas no apêndice A, tendo em conta que os estados que não sofrem alterações são omitidos.

Processamento de *stanzas* de presença

A secção 11.1 da especificação RFC3921[12] define a forma como um *stanza* de presença deve ser processado nas várias situações possíveis. Essas situações, tanto como a forma de processamento, são descritas de seguida.

- Se o endereço especificado for um *bare* JID e existir pelo menos um `recurso` disponível para esse utilizador, o *stanza* deve ser entregue a todos os `recursos` disponíveis. Excepto para os *stanzas* do tipo *'probe'*, que devem ser respondidos pelo próprio servidor, visto que este possui a informação de presença do `recurso`.

- Quando não existem quaisquer `recursos` disponíveis do utilizador especificado, os *stanzas* dos tipos *'subscribe'*, *'subscribed'*, *'unsubscribe'* e *'unsubscribed'* deve ser armazenados até que um `recurso` se torne disponível, de forma a que sejam entregues pelo menos uma vez. Contudo os *stanzas 'subscribe'* devem ser entregues ao utilizador até que este responda, positiva ou negativamente.

2.4.2 Troca de mensagens

A troca de mensagens entre entidades é a utilização mais básica de XMPP. De forma a ser possível trocar mensagens com outra entidade é necessário indicar o endereço dessa mesma entidade, através do atributo ***to*** do *stanza* de mensagem. Este endereço deve ser da forma

<utilizador@domínio/recurso> se a mensagem é enviada no contexto de um conversação, como resposta a uma mensagem previamente recebida. Se a mensagem for enviada fora de qualquer contexto de conversação, o endereço deve ser da forma <utilizador@domínio>.

Processamento de *stanzas* de mensagens

A secção 11.1 da especificação RFC3921[12] define a forma como um *stanza* de mensagem deve ser processado nas várias situações possíveis. Essas situações, tanto como a forma de processamento, são descritas de seguida.

- Se o utilizador especificado como destinatário não estiver registado no servidor, o *stanza* de mensagem deve ser retornado com o erro de <service-unavailable/>;
- Se o endereço é um *full* JID e esse `recurso` não se encontra disponível, o *stanza* será processado considerando o *bare* JID correspondente;
- Ao receber um *stanza* de mensagem endereçado a um *bare* JID, existindo pelo menos um `recurso` disponível desse utilizador, o *stanza* deve ser entregue ao `recurso` mais prioritário. Caso existam vários `recursos` com a mesma prioridade (sendo os mais prioritários), o servidor pode usar um qualquer critério (tempo de ligação, valor de <show/>, ...) para decidir a qual enviar ou então enviar a todos. O servidor nunca deve enviar um *stanza* de mensagem a um `recurso` com prioridade negativa. Isto significa que na eventualidade de apenas existirem `recursos` disponíveis com prioridade negativa, o *stanza* deve ser tratado como se não existissem quaisquer disponíveis.
- Nas situações em que não existem quaisquer `recursos` disponíveis, o servidor pode escolher guardar o *stanza*, para entregar mais tarde a um `recurso` (com prioridade igual ou superior a 0), ou enviá-lo de outra forma (como por exemplo email).

2.4.3 Gestão do *roster*

O *stanza* de IQ serve para realizar um qualquer pedido ao servidor, desde que este tenha essa capacidade. A extensibilidade do próprio *stanza* permite que este seja usado para qualquer tipo de pedido, colocando como seus elementos filhos, elementos de quaisquer *namespaces*.

A especificação do protocolo XMPP para *instant messaging* e *presence*[11] define o *namespace* 'jabber:iq:roster', que contém a definição dos elementos XML para realizar gestão de um *roster*. Esses elementos, em conjunção com o *stanza* IQ, permitem realizar operações de leitura e escrita. O elemento <query/> é o principal elemento definido por este *namespace*, que poderá ou não conter um conjunto de elementos <item/>.

O elemento <item/> corresponde a um item do *roster* do contacto e pode ter os seguintes atributos:

- ***ask***: apenas pode assumir o valor *'subscribe'* e indica que foi enviado um pedido de subscrição a esse contacto, que ainda não foi respondido (opcional);

- ***jid***: indica o endereço do contacto (obrigatório);
- ***name***: indica um *nickname* que o utilizador tenha dado ao contacto (opcional);
- ***subscription***: indica o estado da subscrição da informação de presença do contacto ou indica que o contacto deve ser removido do *roster* (opcional). Desta forma, pode ser os seguintes valores: *'none'*, *'to'*, *'from'*, *'both'* ou *'remove'*.

Além dos atributos, o elemento <item/> pode ter um conjunto de elementos <group/>, em que cada elemento representa um grupo a que o contacto está associado no *roster* do utilizador.

Obter *roster* ao realizar *login*

Após ter criado uma sessão no servidor, o cliente deve obter o seu *roster* antes de enviar a informação de presença inicial. Esta operação é considerada opcional, pois existem situações em que não é viável, como por exemplo, quando o cliente possui uma largura de banda limitada. Nestes casos, as subscrições de presença e respectivas actualizações do *roster*, não serão enviadas para o cliente.

A obtenção do *roster* é realizada com um *stanza* IQ com *type 'get'* e um elemento filho <query/> vazio. A resposta a este pedido é realizada com um *stanza* IQ com *type 'result'* e um elemento filho <query/> com todos os elementos <item/> que representam os contactos do utilizador. Um exemplo do pedido de obtenção é apresentado na listagem D.1 e um exemplo de resposta é apresentado na listagem 2.2.

```
<iq to='utilizador@servidor.pt/recurso' type='result' id='r_1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='utilizador2@servidor.pt'
          subscription='both'>
      <group>Amigos</group>
      <group>Colegas</group>
    </item>
    <item jid='utilizador3@servidor.pt'
          nickname='Utilizador␣3'
          subscription='from'>
      <group>Amigos</group>
    </item>
  </query>
</iq>
```

Listagem 2.2: Exemplo de resposta a obtenção do *roster*

Na listagem 2.5 é apresentado um *roster* com dois contactos. Um deles, o "utilizador2@servidor.pt", encontra-se associado a dois grupos, "Amigos" e "Colegas", e tanto o contacto como o utilizador possuem uma subscrição da informação da presença um do outro. O contacto "utilizador3@servidor.pt" pertence apenas a um grupo, "Amigos", e possui uma subscrição da informação de presença do utilizador, mas o contrário não se verifica. Este contacto tem ainda o *nickname* "Utilizador 3".

Adicionar um contacto

A adição de um contacto é também realizada através de um *stanza* IQ, desta feita com ***type*** *'set'* e com um elemento filho <query/>, que contém a descrição do contacto a adicionar num elemento <item/>. A listagem 2.3 apresenta um exemplo de adição de um contacto.

```
<iq from='utilizador@servidor.pt/recurso' type='set' id='r_2'>
  <query xmlns='jabber:iq:roster'>
    <item jid='utilizador4@servidor.pt'/>
    </item>
  </query>
</iq>
```

Listagem 2.3: Exemplo de adição de um contacto ao *roster*

O servidor cria um novo item no *roster* do utilizador de forma persistente, realiza um "*roster push*" a todos os `recursos` disponíveis que pediram o *roster* (incluindo o que iniciou a operação) e por fim envia um *stanza* IQ de resultado ao `recurso` que iniciou a operação. O "*roster push*" é semelhante ao *stanza* utilizado para adição do contacto, mas o elemento <item/> deve agora ter também o atributo ***subscription*** com o estado actual da subscrição, que após a adição do contacto é *'none'*. A listagem 2.4 apresenta os "*roster pushs*" que seriam realizados após a adição do contacto na listagem 2.3. Na listagem 2.5 está apresentado o *stanza* IQ de resultado a enviar ao `recurso` que iniciou a operação.

```
<iq to='utilizador@servidor.pt/recurso' type='set' id='abc'>
  <query xmlns='jabber:iq:roster'>
    <item jid='utilizador4@servidor.pt'
          subscription='none'/>
    </item>
  </query>
</iq>

<iq to='utilizador@servidor.pt/recurso2' type='set' id='def'>
  <query xmlns='jabber:iq:roster'>
    <item jid='utilizador4@servidor.pt'
          subscription='none'/>
    </item>
  </query>
</iq>
```

Listagem 2.4: Exemplo de "*roster pushs*"

Cada `recurso` deve responder ao servidor com um *stanza* IQ de resultado para o "*roster push*".

```
<iq to='utilizador@servidor.pt/recurso' type='result' id='r_2'/>
```

Listagem 2.5: Exemplo de uma resposta a uma adição ao *roster*

Actualizar um contacto

A forma de actualizar um contacto é, para o cliente, igual à adição de contacto, bastando-lhe enviar um *stanza* IQ *'set'* com a descrição completa do contacto num elemento <item/>. O servidor tem também um processamento semelhante à adição de contacto, mas neste caso, em vez de criar um novo item, actualiza a informação de um item existente em armazenamento persistente. A listagem 2.6 apresenta um exemplo de uma actualização de um contacto.

```
<iq from='utilizador@servidor.pt/recurso' type='set' id='r_3'>
  <query xmlns='jabber:iq:roster'>
    <item jid='utilizador4@servidor.pt'
          subscription='none'>
      <group>Amigos</group>
    </item>
  </query>
</iq>
```

Listagem 2.6: Exemplo de uma actualização de um contacto no *roster*

Remover um contacto

A remoção de um contacto é também realizada com um *stanza* IQ *'set'*, mas o elemento <item/> que especifica o contacto deve apenas conter os atributos ***jid*** e ***subscription***. O atributo ***jid*** contém o endereço do contacto a remover e o atributo ***subscription*** contém o valor *'remove'*. O servidor deve então remover o contacto do *roster* em armazenamento persistente, realizar um "*roster push*" a todos os `recursos` disponíveis que requisitaram o *roster* e responder ao `recurso` que iniciou a operação com um *stanza* IQ de resultado. A listagem 2.7 apresenta um exemplo da remoção de um contacto.

```
<iq from='utilizador@servidor.pt/recurso' type='set' id='r_4'>
  <query xmlns='jabber:iq:roster'>
    <item jid='utilizador4@servidor.pt' subscription='remove'/>
  </query>
</iq>
```

Listagem 2.7: Exemplo da remoção de um contacto do *roster*

Processamento de *stanzas* de *info/query*

A secção 11.1 da especificação RFC3921[12] define a forma como um *stanza* de *info/query* deve ser processado nas várias situações possíveis. Essas situações, tanto como a forma de processamento, são descritas de seguida.

- Se o utilizador especificado como destinatário não estiver registado no servidor ou se o endereço é um *full* JID e esse `recurso` não se encontra disponível, o *stanza* de *info/query* deve ser retornado com o erro de <service-unavailable/>;

- Ao receber um *stanza* de *info/query* endereçado a um *bare* JID, quer existam ou não `recursos` disponíveis associados a esse utilizador, o servidor deve responder ao cliente que o enviou com um *stanza* IQ de resultado ou de erro, conforme o resultado do processamento do mesmo. Não deve reencaminhar esse *stanza* para nenhum `recurso` e caso não possua a capacidade de o processar, deve retornar o erro <service-unavailable/>.

2.4.4 Processamento dos *stanzas* em geral

Cada tipo de *stanza* tem um processamento específico para as situações descritas em cada um dos tipos, mas existem uma situação em que o processamento a realizar é independente do tipo do *stanza*. Quando o JID especificado como destino for um *full* JID e esse `recurso` estiver disponível, o servidor deve-lhe entregar o *stanza*.

2.5 Integração de serviços por *gateways*

Uma das principais características do protocolo XMPP é a definição de *gateways*, que servem para realizar comunicação com servidores de outros protocolos, traduzindo o protocolo XMPP para o protocolo alvo. A figura 2.1 apresenta *gateways* para servidores de SMS (*Short Message Service*) e SMTP (*Simple Mail Transfer Protocol*).

Desta forma, um *gateway* pode também ser usado para comunicar com outros sistemas de *instant messaging* já existentes, tal como *AIM*, *Live Messenger* e *Yahoo! Messenger*. A extensão XEP-0100[13] define a interacção entre um cliente XMPP e um *gateway* que se faz passar por esse mesmo cliente perante um serviço não XMPP. O *gateway* é então visto como um cliente para o serviço não XMPP. A interacção entre o *gateway* e esse serviço não está definida por esta extensão, pois isso irá depender do serviço em si.

É um requisito desta extensão que um utilizador XMPP esteja registado no serviço com que deseja comunicar. Ao registar-se no *gateway*, um utilizador XMPP deverá fornecer as suas credenciais nesse serviço, para que o *gateway* se faça passar pelo utilizador. Isto implica que o utilizador se registe em cada um dos serviços não XMPP com que deseja comunicar. Desta forma um utilizador XMPP consegue comunicar com utilizadores de outros serviços de *instant messaging*, por via de uma conta *proxy*, gerida pelo *gateway*.

Capítulo 3

Yahoo! Messenger

A *Yahoo* disponibiliza um serviço para realização de operações de *instant messaging* e *presence*. Este serviço está documentado em [16].

O serviço possibilita a criação de uma aplicação cliente para operações de *instant messaging* na rede *Yahoo*, através de uma conta *Yahoo*. Isto implica autenticação do cliente perante o serviço. Esta autenticação é realizada através de uma variante do protocolo *OAuth*[14].

Após a autenticação é possível realizar várias operações de *instant messaging* e *presence* em nome do utilizador autenticado. As principais operações estão enumeradas de seguida.

- Criação de uma sessão de *instant messaging* e *presence*;
- Actualização da informação de presença;
- Envio de mensagens;
- Obtenção de lista de contactos;
- Adicionar contacto/pedido de subscrição;
- Responder a pedido de subscrição;
- Obtenção de notificações.

O acesso ao serviço é realizado através do protocolo HTTP[1], cuja forma de utilização corresponde a um simples pedido-resposta. A ligação ao servidor apenas se encontra aberta até ao momento em que a resposta é recebida, não existindo qualquer relação entre um cliente e um endereço. Desta forma é necessário que o próprio cliente realize um pedido, autenticado, para obter as notificações que lhe são devidas. Estas notificações são, por exemplo, mensagens recebidas e pedidos/respostas de subscrição de presença.

[1] *Hypertext Transfer Protocol*

3.1 Autenticação

A autenticação é realizada através de uma adaptação do protocolo *OAuth*[14], tal como já foi referido. Esta adaptação consiste em enviar um primeiro pedido com o par {*username*, *password*}, em vez de redireccionar o utilizador para uma página web onde pode inserir essa mesma informação. A figura 3.1 apresenta um diagrama de sequência que mostra o fluxo para autenticação.

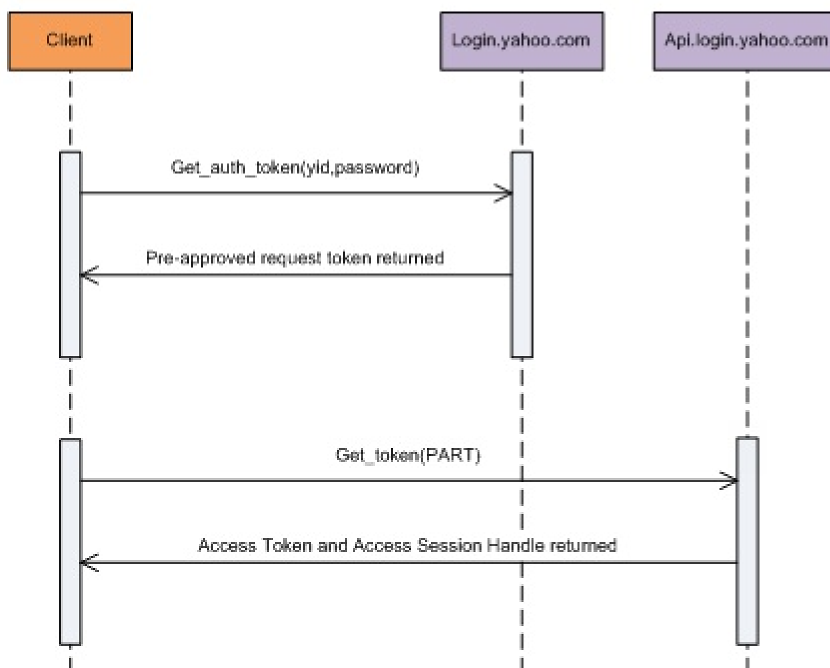


Figura 3.1: Fluxo de autenticação através de OAuth adaptado. Fonte: *Sequence Diagram for Authentication Flow*[15]

A resposta ao envio do *username* e *password* é um *Pre Approved Request Token* (PART), que deve ser posteriormente enviado ao servidor de credenciais *OAuth*, de forma a obter as credenciais para realizar operações de *instant messaging* e *presence* autenticadas.

Fora o primeiro pedido, todos os pedidos ao serviço, inclusive a requisição de credenciais, devem enviar a informação de autenticação no *header* HTTP *Authorization*, conforme definido em [14].

As credenciais obtidas têm um tempo de vida limitado e devem ser obtidas novas, quando esta validade terminar.

3.2 Operações de *Instant Messaging* e *Presence*

As operações de *instant messaging* e *presence* apenas podem ser realizadas no contexto de uma sessão. Desta forma, imediatamente após uma autenticação bem sucedida, antes que possa ser realizada qualquer outra operação, deve ser criada uma sessão.

3.2.1 Criação de sessão

A criação de uma sessão, que pode ser visto como realizar *login*, é executada pela API[2] de gestão de sessão. Esta API está acessível no endereço `http://developer.messenger.yahooapis.com/v1/session`. Para criar a sessão é realizado um pedido HTTP POST a este endereço, indicando, opcionalmente, a presença inicial no corpo do pedido (apenas é suportado o formato JSON[3]). A listagem 3.1 apresenta um exemplo de um pedido para criação de sessão.

```
POST /v1/session
Host: developer.messenger.yahooapis.com
Authorization: < credenciais OAuth >
Content-Type: application/json;charset=utf-8
Content-Length: 26
{
   "presenceState" : 0,
   "presenceMessage" : "My login status message"
}
```

Listagem 3.1: Exemplo de um pedido para criação de sessão *Yahoo*

A resposta à criação de uma sessão inclui o identificador da sessão (que será utilizado em todos as operações executadas subsequentemente), o endereço do servidor ao qual deve dirigir os pedidos e o endereço do servidor ao qual deve dirigir os pedidos de obtenção de notificações. Estes servidores serão daqui em diante referidos como <servidor> e <servidor-notificações>, respectivamente.

3.2.2 Actualização de presença

Ao longo do tempo de vida da sessão é possível alterar a informação de presença de utilizador, tal como acontece aquando da criação da sessão. Esta alteração é realizada através da API de gestão de presença, disponível no endereço `<servidor>/v1/presence`. A informação de presença é caracterizada por um estado (*presenceState*) e por uma mensagem opcional (*presenceMessage*). A operação em si é realizada através de um pedido HTTP POST, com a informação de presença no corpo do pedido, em formado *JSON*. A listagem 3.2 apresenta um exemplo deste pedido.

3.2.3 Envio de mensagens

O envio de mensagens é a principal operação de *instant messaging*, sendo esse o seu principal objectivo, comunicação. Isto é realizado através da API de gestão de mensagens. O envio de uma mensagem a um determinado utilizador é realizada através do seguinte endereço `<servidor>/v1/message/{rede}/{identificadorUtilizador}`. Em {rede} deve estar o valor "*yahoo*", sendo o único possível de momento. Em {identificadorUtilizador} deve

[2] *Application Programming Interface*
[3] JavaScript Object Notation

```
PUT /v1/presence?sid=sessionid
Host: <servidor>
Authorization: < credenciais OAuth >
Content-Type: application/json;charset=utf-8
Content-Length: 25
{
   "presenceState" : 2,
   "presenceMessage" : "My␣custom␣status␣message"
}
```

Listagem 3.2: Exemplo de um pedido para actualizar a informação de presença

estar o identificador do utilizador que irá receber a mensagem, que corresponde ao endereço *Yahoo*, sem a parte *"@yahoo.com"*. A listagem 3.3 apresenta um exemplo de envio de mensagem para o utilizador <utilizador1@yahoo.com>.

```
POST /v1/message/yahoo/utilizador1?sid=sessionid
Host: <servidor>
Authorization: < credenciais OAuth >
Content-Type: application/json;charset=utf-8
Content-Length: 25
{
    "message" : "Hey␣there"
}
```

Listagem 3.3: Exemplo do envio de mensagem para um utilizador

3.2.4 Obtenção de lista de contactos

Como qualquer servidor de *instant messaging*, o serviço da *Yahoo* armazena a lista de contactos dos seus utilizadores. A lista de contactos pode ser obtida de duas formas. A primeira forma é durante a criação da sessão, bastando para isso adicionar o atributo "fieldsBuddyList" com o valor "+groups" ao URI do pedido. Na listagem 3.1 o pedido ficaria da seguinte forma "/v1/session?fieldsBuddyList=%2Bgroups". O valor "%2B" corresponde à codificação em URL do carácter "+".

A segunda forma de obter a lista de contactos passa pela utilização da API de gestão de lista de contactos, disponível no endereço `<servidor>/v1/contacts`. Esta operação é realizada através de um pedido HTTP GET, simplesmente com o identificador de sessão, tal como é apresentado na listagem 3.4.

```
GET /v1/contacts?sid=msgrsessionid
Host: <servidor>
Authorization: < credenciais OAuth >
```

Listagem 3.4: Exemplo da obtenção da lista de contactos

3.2.5 Adicionar contacto/pedido de subscrição

A adição de um novo contacto é realizada através da API para gestão de grupos de contactos, com o endereço
`<servidor>/v1/group/{nomeGrupo}/contact/{rede}/{identificadorUtilizador}`.
Em {rede} deve estar o valor "*yahoo*", sendo o único possível de momento. Em {identificadorUtilizador} deve estar o identificador do utilizador que será adicionado à lista de contactos do utilizador que se encontra ligado através da API. Esta adição irá desencadear automaticamente um pedido de subscrição da informação de presença do utilizador adicionado, que deverá ser aceite ou negada, por parte desse utilizador. A listagem 3.5 apresenta um exemplo de envio de mensagem para o utilizador <utilizador2@yahoo.com>.

```
PUT /v1/group/Amigos/contact/yahoo/utilizador2?sid=msgrsessionid
Host: <servidor>
Authorization: < credenciais OAuth >
Content-Type: application/json;charset=utf-8
Content-Length: 2

{}
```

Listagem 3.5: Exemplo da adição de um contacto à lista de contactos

3.2.6 Responder a pedido de subscrição

A resposta a um pedido é realizada através da API de gestão de autorizações de lista de contactos. Esta API está disponível no endereço `<servidor>/v1/buddyrequest/{rede}/{identificadorUtilizador}`. Tal como nos casos anteriores, o único valor possível para {rede} é "*yahoo*". Em {identificadorUtilizador} deve estar o identificador do utilizador que realizou o pedido de subscrição a ser aceite ou negado. A diferença entre aceitação ou negação do pedido de subscrição encontra-se no pedido HTTP realizado, sendo o endereço utilizado o mesmo. A aceitação será realizada através de um pedido HTTP POST e a negação, um pedido HTTP DELETE. A listagem 3.6 apresenta um exemplo da aceitação de um pedido de subscrição por parte do utilizador <utilizador3@yahoo.com>.

```
POST /v1/buddyrequest/yahoo/utilizador3?sid=msgrsessionid
Host: <servidor>
Authorization: < credenciais OAuth >
Content-Type: application/json;charset=utf-8
Content-Length: 2

{}
```

Listagem 3.6: Exemplo da aceitação de um pedido de subscrição

3.2.7 Notificações

Um serviço de *instant messaging* e *presence* necessita de uma forma de receber mensagens e alterações de presença, entre outras notificações. Desta forma existe a API para obtenção de notificações, podendo ser utilizada de duas formas distintas, por *polling*[4] periódico e por *Comet-Style push*.

O endereço para obter notificações por *polling* periódico é `<servidor>/v1/notifications` e apenas deve ser acedido de 5 em 5 minutos, nunca menos do que isso.

A obtenção de notificações por *Comet-Style push* consiste em realizar um pedido de longa vida ao servidor de notificações obtido aquando da criação da sessão. Este pedido ficará bloqueado no servidor, durante um determinado tempo, até que existam notificações a retornar ao cliente.

O acesso, em ambas as formas, será realizado através de um pedido HTTP GET e a resposta está no formato *JSON*, cujo conteúdo pode ser variado e encontra-se documentado em [16].

Os tipos de notificações considerados mais significativos para *instant messaging* e *presence* são as seguintes:

- ***buddyInfo***: obtida quando algum utilizador da lista de contactos realiza *login* no serviço;
- ***buddyStatus***: obtida quando algum utilizador da lista de contactos altera a sua informação de presença;
- ***logOff***: obtida quando algum utilizador da lista de contactos realiza *logout* no serviço;
- ***message***: obtida quando algum utilizador da lista de contactos envia uma mensagem ao utilizador que está ligado pela API;
- ***buddyAuthorize***: obtida quando algum utilizador do serviço envia um pedido de subscrição ou responde a um pedido de subscrição efectuado pelo utilizador que está ligado pela API;
- ***disconnect***: obtida quando a sessão no servidor expirou (apenas acontece quando se usa a forma de notificações *Comet-Style push*).

[4] Verificar se existe alguma notificação

Capítulo 4

Visão geral

Após um estudo na área chega-se à conclusão que um servidor de *instant messaging*, que use o protocolo XMPP, deve (além das funcionalidades apresentadas em 1.2):

- Aceitar e gerir ligações TCP;
- Realizar *parse* e geração de XML;
- Armazenar informação de e para utilizadores;
- Processar elementos XMPP.

A primeira fase do trabalho passa pelo desenvolvimento de um servidor de *instant messaging*, que suporte as funcionalidades atrás descritas e implemente parcialmente as especificações RFC3920[11] e RFC3921[12].

Na segunda fase será implementado um módulo de conversão para um serviço já existente, como forma de atingir o objectivo opcional do trabalho, que passa pela integração de vários serviços de *instant messaging*. Desta forma existe interoperabilidade entre dois serviços, um em XMPP, desenvolvido no âmbito deste trabalho, e um dos grandes serviços proprietários, criando assim um *bus* de *instant messaging*.

Devido à limitação de tempo para a realização do trabalho, apenas será considerado um serviço já existente, o que bastará para atingir e provar o conceito do mesmo.

A investigação realizada sobre os serviços de *instant messaging* já existentes, nomeadamente *Live Messenger*, AIM e *Yahoo!Messenger*, ajudou a identificar aquele que melhor se enquadre neste trabalho. Assim o escolhido é o serviço para IM da *Yahoo*[16], por ser o mais directo e simples de utilizar.

4.1 Problema encontrado

Em 2.5 é apresentada uma extensão ao protocolo XMPP que tem por objectivo definir a forma de utilização do protocolo para acesso a um *gateway* para outro serviço de *instant messaging*.

Com esta extensão apenas é possível que um utilizador XMPP comunique com um utilizador de outro serviço, caso possua conta nesse mesmo serviço. Um utilizador desse outro serviço não consegue comunicar com utilizadores XMPP que não possuam conta nesse serviço.

4.2 Solução apresentada

Este trabalho apresenta uma solução que torna possível a comunicação entre clientes de serviços heterogéneos, sem que nenhum possua uma conta no outro serviço.

A solução proposta por este trabalho consiste num servidor de *instant messaging*, para suportar um conjunto de módulos de tradução para outros serviços, onde cada um comunica com um serviço de *instant messaging* já existente.

O servidor de *instant messaging* implementa parcialmente o protocolo XMPP. Além de disponibilizar um porto TCP para comunicação XMPP, disponibiliza também um *Web Service* (a que se chamará *Web Service* Central) para receber informação dos módulos de tradução.

Os módulos de tradução devem também disponibilizar um *Web Service*, para receber informação do servidor. Cada módulo deve ainda gerir uma conta "*fantasma*" no serviço com que comunica. Esta conta irá representar o servidor de *instant messaging* perante os clientes desse serviço. Toda a comunicação entre um cliente de um serviço já existente e o servidor irá ser realizada através da respectiva conta fantasma. Desta forma é criado um bus de *instant messaging*, capaz de interligar diversos serviços de *instant messaging* que utilizam protocolos heterogéneos, sem que o utilizador possua uma conta em cada um desses serviços.

4.3 Arquitectura e sistema

O desenvolvimento do trabalho começa pela idealização de uma possível arquitectura para o *bus* de *instant messaging*. Tendo em conta as ideias e solução atrás enunciadas, foi idealizada a arquitectura apresentada na figura 4.1.

Os principais componentes do servidor são apresentados a azul-escuro, sendo os verdes e azul-claro componentes auxiliares. O componente laranja é apresentado como um componente externo ao servidor em si, mas ainda parte do sistema.

O servidor foi concebido para ser utilizado sozinho, ou seja, ao contrário do que está definido na especificação XMPP[11], não suporta integração numa rede de servidores. O servidor foi desenvolvido de forma a obter o máximo conhecimento sobre o funcionamento de um servidor de *instant messasing*, para realizar integração com outros serviços. Logo a implementação à risca do protocolo não se encontra nos objectivos principais do trabalho.

Em seguida é apresentada uma breve descrição para cada um dos componentes apresentados na figura 4.1.

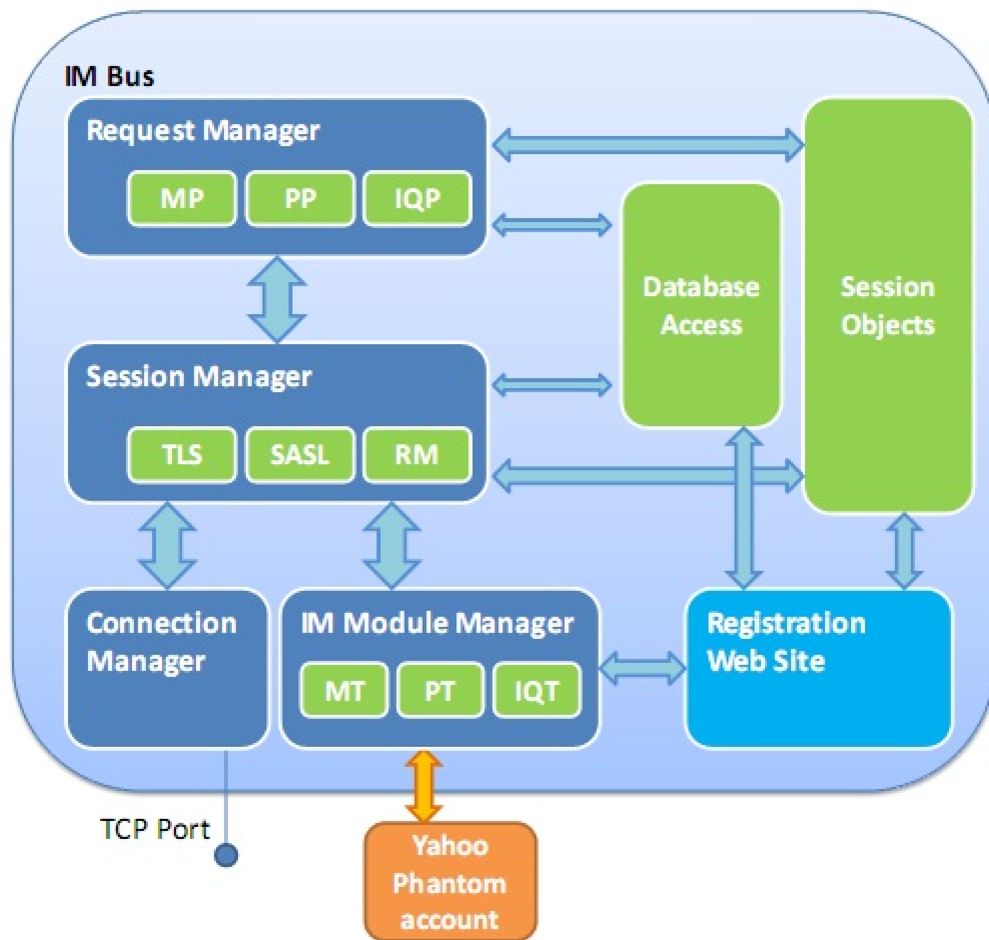


Figura 4.1: Arquitectura geral

4.3.1 Descrição dos componentes

- **Connection Manager**: ponto de entrada no servidor para um cliente XMPP. É responsável pela gestão das ligações com todos os clientes, realizando toda a comunicação com estes;

- **Session Manager**: responsável pela gestão das sessões dos clientes, principalmente pela negociação das sessões XMPP. Todos os *stanzas* enviados pelo cliente passam por aqui, sendo este componente o responsável por transformar a informação XML em objectos (com auxílio a tipos criados para esse efeito). Por este cliente passam também todos os pedidos recebidos pelo *IM Module Manager*, de forma a ligá-los a uma sessão com um serviço externo;

  - **TLS Component** (TLS): componente responsável pela negociação TLS;

  - **SASL Component** (SASL): componente responsável pela negociação SASL;

- **Resource Manager** (RM): componente responsável por realizar *resource binding*.

- **Request Manager**: responsável por processar os pedidos que chegam ao servidor, conforme as regras descritas na especificação RFC3921[12] para o processamento de *stanzas*;
  - **Message Processor** (MP): processa os *stanzas* de mensagem;
  - **Presence Processor** (PP): processa os *stanzas* de presença;
  - **Info/Query Processor** (IQP): processa os *stanzas* de *info/query*;
- **Database Access**: trata de todo o acesso aos dados;
- **Transport Layer**: trata de toda a transformação entre objectos e elementos XML e contém ainda os tipos de transporte de dados. Este componente define os tipos a usar no contexto de um pedido;
- **Registration Web Site**: pequena aplicação *Web* para registar os utilizadores no *bus* de *instant messaging*;
- **IM Module Manager**: componente responsável pela comunicação com os módulos de tradução dos serviços já existentes;
  - **Message Translator** (MT): realiza conversões entre os tipos de objectos *stanzas* de mensagem e os tipos utilizados nos *Web Services*;
  - **Presence Translator** (PT): realiza conversões entre os tipos de objectos *stanzas* de presença e os tipos utilizados nos *Web Services*;
  - **Info/Query Translator** (IQT): realiza conversões entre os tipos de objectos *stanzas* de *info/query* e os tipos utilizados nos *Web Services*;
- **Yahoo Phantom Account**: componente responsável por se fazer passar pelo *bus* de *instant messaging*, enviando toda a informação em nome dos utilizadores. Realiza comunicação com o serviço de IM da *Yahoo!*[16].

4.3.2 Contrato do *Web Service* Central

O componente responsável por gerir a comunicação com os módulos de tradução é o *IM Module Manager*, o que implica que seja este componente a publicar o *Web Service* Central. Este *Web Service* deve disponibilizar as operações principais de *instant messaging*, de forma a que os módulos as possam utilizar para enviar a informação pertinente para o *bus*. O contrato do *Web Service* Central deve então disponibilizar as seguintes operações:

- Obtenção da lista de contactos de um utilizador (***GetContactList***);
- Enviar informação de presença de um utilizador a outro ou simplesmente realizar broadcast dessa informação (***SendPresence***);

- Adicionar um contacto à lista de contactos de um utilizador e subsequentemente realizar um pedido de subscrição da sua informação de presença (***AddContact***);

- Remover um contacto da lista de contactos de um utilizador e subsequentemente cancelar todas as subscrições de presença entre eles (***RemoveContact***);

- Enviar uma resposta a um pedido de subscrição de presença (***AnswerSubscriptionRequest***);

- Enviar uma mensagem de um utilizador a outro (***SendMessage***).

Os sistemas de *instant messaging* têm um carácter assíncrono, no que respeita ao processamento dos pedidos executados. Isto pode-se verificar na forma de utilização do protocolo XMPP, em que um cliente realiza um pedido e não fica bloqueado à espera que o mesmo seja processado, tendo em conta que na maior parte dos casos não existe uma resposta a esse pedido. Caso ocorra algum erro com o processamento do mesmo, o cliente será posteriormente notificado dessa ocorrência. De forma a manter o carácter assíncrono na chamada ao *Web Service* Central, estas operações devem ser marcadas como sendo *One Way*, ou seja, não têm mensagem de retorno. Desta forma um cliente do *Web Service* irá invocar uma operação e prosseguir imediatamente, não ficando bloqueado durante a execução do mesmo.

4.3.3 Contrato dos *Web Services* dos módulos

A implementação dos módulos de tradução depende maioritariamente do serviço em questão, mas este tem obrigatoriamente que disponibilizar um *Web Service* com um contrato específico. Este contrato permite que o *IM Module Manager* invoque as mesmas operações sobre os diferentes módulos da mesma forma, unificando a forma de comunicação entre eles. Desta forma o sistema é modular, podendo adicionar novos módulos sem ser necessário grandes adaptações (basta adicionar em dois locais a informação referente ao novo módulo). A definição do contrato de um módulo é independente da implementação dos módulos, pois cada um deles deve respeitar esse mesmo contrato, o que leva à utilização da técnica *Contract-First Web Services*. Esta técnica implica que o contrato, no caso de *Web Services*, o WSDL[1], seja definido *à priori* e o código seja gerado através dessa mesma definição.

Segundo [1] um ficheiro WSDL deve conter:

- Uma secção para os tipos de mensagens a trocar na invocação e resposta das operações (*types*);

- Uma secção com a definição abstracta das partes de uma mensagem, associada a um tipo existente (*message*);

- Uma secção que define um conjunto abstracto de operações (contrato), indicando quais são as mensagens de entrada e saída (*portType*);

[1] *Web Services Description Language*

- Uma secção que define o protocolo e o formato dos dados para as operações de um contrato (*binding*);
- Uma secção que define um endereço para um *binding* em específico (*port*);
- Uma secção que agrega um conjunto de *ports* (*service*).

Os *Web Services* disponibilizados pelos módulos de tradução devem manter a natureza assíncrona, que caracteriza o servidor e o *Web Service* Central. Desta forma também as operações dos *Web Services* dos módulos devem ser *One Way*. Em WSDL uma operação fica marcada como *OneWay*, quando na secção *portType*, não é indicada uma mensagem de saída (*output*). Desta forma, a definição da operação fica da forma apresentada na listagem 4.1.

```
<operation name="SendMessage">
   <input message="tns:SendMessage_InputMessage" />
</operation>
```

Listagem 4.1: Exemplo da definição de uma operação *OneWay* em WSDL (excerto do ficheiro WSDL para os *Web Services* dos módulos de tradução)

O ficheiro WSDL que define o contrato que deve ser implementado pelos módulos de tradução está apresentado no apêndice B. Este contrato deve definir as seguintes operações de *instant messaging*:

- Realizar o pedido de subscrição de um utilizador à informação de presença de outro (***AddContact***);
- Enviar a informação de presença de um utilizador a outro (***SendPresence***);
- Enviar uma mensagem de um utilizador para outro (***SendMessage***);
- Enviar a respectiva lista de contactos a um utilizador (***SendContactList***);
- Obter a informação de presença de um utilizador do serviço com que o módulo comunica (***GetPresence***).

Operações de sistema

Além das operações utilizadas para *instant messaging* estão definidas outras três, que serão chamadas operações de ”*sistema*”. Estas operações existem apenas para serem usadas pelo sistema e não para efeitos de *instant messaging*.

Tal como mencionado anteriormente, a solução por este trabalho proposta para a integração de serviços de *instant messaging*, consiste em ter contas ”*fantasmas*” que se irão fazer passar pelo *bus* de *instant messaging* perante os clientes dos serviços respectivos. A necessidade das operações de sistema surge desta solução.

O registo de um utilizador no *bus* consiste então, considerando esta solução, em adicionar o utilizador à lista de contactos da conta fantasma do respectivo sistema. Ao adicionar

o utilizador, será realizado automaticamente um pedido de subscrição da sua informação de presença. Para que isto seja possível foi introduzida a operação ***RegisterUser*** no contrato dos *Web Services* dos módulos. Quando o utilizador aceitar o pedido realizado, será automaticamente realizado um pedido de subscrição da presença da conta fantasma, por parte do utilizador em questão, devendo este ser imediatamente aceite. Apenas após esta fase de aceitação de subscrições de presença é que o utilizador se encontra realmente registado no *bus* de *instant messaging*.

Os utilizadores dos serviços existentes, após registo, apenas podem interagir com a conta fantasma se esta estiver disponível. Assim quando o *bus* inicia a sua execução, deve garantir que todas as contas se encontram disponíveis para comunicar com os utilizadores dos respectivos serviços. Para cumprir este requisito foram introduzidas as operações ***LogIn*** e ***LogOut***.

4.4 Modelo de dados

As necessidades de persistência de informação de um servidor básico de *instant messaging* são ao nível da autenticação dos utilizadores, itens dos *rosters* dos utilizadores (tanto a nível de ligações entre utilizadores, como ligação com grupos de contactos) e *stanzas* que devem ser posteriormente entregues aos clientes. Na figura 4.2 é apresentado o esquema de dados, através do modelo entidade-associação.

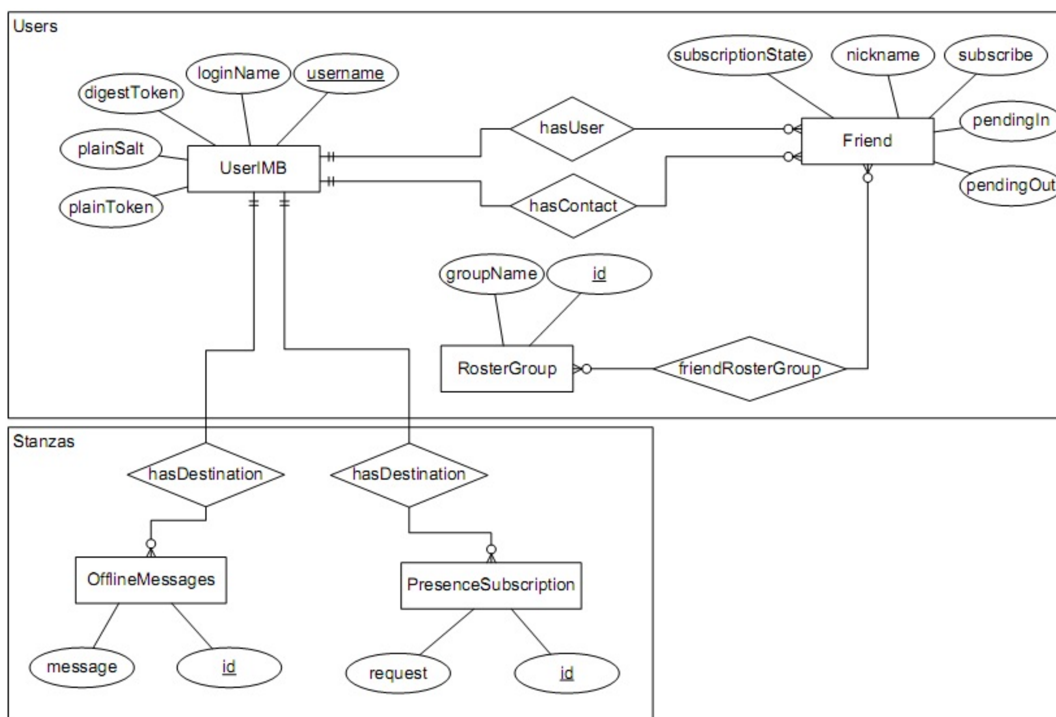


Figura 4.2: Modelo de dados

4.5 Implementação

Após a definição de uma arquitectura para o *bus* de *instant messaging*, deve começar a implementação. O capítulos seguintes irão apresentar os pontos considerados mais importantes da implementação.

Capítulo 5

Implementação de componentes auxiliares

Neste capítulo serão apresentados pormenores de implementação de alguns componentes auxiliares, apresentados na figura 4.1. Os componente referidos realizam acesso a dados (*Database access*), leitura/escrita de elementos XMPP (*Session Objects*) e registo de utilizadores (*Registration Web Site*).

5.1 Acesso a dados

O modelo de dados é simples, o que faz com que o acesso aos dados seja também simples, não apresentando grandes requisitos funcionais. Desta forma a tecnologia escolhida para construir a camada de acesso a dados foi ADO.NET.

De forma a simplificar a gestão das ligações à base de dados foram criados objectos de sessão, nos quais essa função é delegada. Esta delegação permite aos componentes que desejam realizar acesso a dados, fazê-lo de forma mais directa e simplificada, pois não é necessário preocuparem-se com gerir a sessão de ligação ao repositório de dados.

Embora o acesso aos dados fique sempre comprometido com a própria tecnologia usada, neste caso ADO.NET, é possível abstrair da tecnologia utilizada para armazenamento dos dados. Este é o outro objectivo dos objectos sessão. Foi criada uma interface que utiliza os tipos ADO.NET para acesso genérico, que deve ser implementada pelas classes responsáveis por criar ligações à base de dados de uma determinada tecnologia.

O acesso aos dados é realizado através de *Database Access Objects* (DAOs), que implementam as operações CRUD[1]. Estes objectos são construídos com o objecto sessão que devem utilizar para aceder aos dados, conhecendo apenas os métodos da interface criada. Desta forma consegue-se que qualquer alteração na tecnologia de armazenamento dos dados tenha o mínimo impacto possível na camada de acesso a dados. O ponto de alteração está nos

[1] Create, Read, Update and Delete

objectos de sessão.

5.1.1 Sessão de ligação

De forma a atingir os objectivos dos objectos de sessão foi criada a interface **ISession**, tal como referido anteriormente, que é apresentada na listagem 5.1.

```
public interface ISession
{
   bool IsConnected();
   void Connect();
   void Disconnect();
   IDbCommand CreateCommand();
   IDbCommand CreateCommand(string statement);
   IDbDataParameter CreateParameter(IDbCommand command, string name,
      DbType type, ParameterDirection dir, object value);
   IDbDataParameter CreateParameter(IDbCommand command, string name,
      DbType type, ParameterDirection dir, object value,
      bool isNullable);
   DbTransaction CreateTransaction(IsolationLevel level);
   void Commit();
   void Rollback();
}
```

Listagem 5.1: Interface **ISession**

Desta forma o comprometimento com a tecnologia usada para armazenamento dos dados, encontra-se apenas nas classes que derivem da interface **ISession**. A implementação necessária para acesso à base de dados em SQL Server, é realizada na classe **SqlSession**, que implementa a interface **ISession**.

5.1.2 Objectos de acesso aos dados

Como todos estes objectos irão ser construídos com a sessão (**ISession**) através da qual irão realizar o acesso aos dados, foi criada uma classe para servir como base a todos as classes DAO. A classe DAO_Common é apresentada na listagem 5.2.

```
public abstract class DAO_Common
{
   protected ISession session;

   protected DAO_Common(ISession s)
   {
      session = s;
   }
}
```

Listagem 5.2: Classe base para todos os DAOs

Os tipos desenvolvidos para aceder aos dados são: **DAO_User**, **DAO_ContactList**, **DAO_OfflineMessage** e **DAO_SubscriptionRequest**. Estes tipos disponibilizam métodos para realizar as operações CRUD sobre os objectos da base de dados.

Procedimentos armazenados

Nunca é realizado acesso directo às tabelas da base de dados, são sempre utilizados procedimentos armazenados. No seguimento da criação do utilizador para acesso aos dados, este apenas tem permissões para executar procedimentos armazenados, em qualquer um dos *schemas*.

5.2 Objectos de sessão

Neste componente estão definidos os objectos utilizados no contexto de uma sessão de ligação entre um cliente e o servidor.

De forma a representar os vários elementos XML do protocolo (tanto elementos de negociação do *stream*, como *stanzas*) em objectos, foi criada uma hierarquia de classes para unificar todas as características semelhantes. As classes desta hierarquia são **SimpleElement**, **SimpleElementVariableChildren** e **StanzaElement**. Conforme as características de cada elemento XMPP, a classe que o representa deriva de uma destas. Tendo em conta que apenas as classes que representam os *stanzas* derivam de **StanzaElement**. A figura C.1 no apêndice C apresenta uma visão geral sobre as classes definidas, dividas pelos respectivos *namespaces*, conforme a sua funcionalidade. O elemento <stream/> é tratado de forma diferente dos restantes, pois é um elemento que se mantém aberto durante toda a vida da ligação. O que implica que também a classe que o representa, a classe **XmppStream**, seja tratada de forma diferente das restantes. É possível verificar pela imagem C.1 que esta classe não deriva de nenhuma das três classes base. Isto acontece pois a sua utilização é diferente, tendo em conta que será escrita a sua *tag* de abertura no início da ligação e a sua *tag* de fecho apenas será escrita no final da ligação.

O *stream* XML, tal como já foi dito anteriormente, pode ser visto como um documento XML em construção. O que indica que alguns elementos XML utilizados pelo servidor estariam mal formados, embora isso não ocorra na maioria das vezes. Isto acontece pois a escrita de elementos XML ocorre sempre em fragmentos e nunca no contexto do documento XML em construção. Assim nem sempre seria possível utilizar a classe **XmlDocument** da *framework* .NET. Isto levou à criação de um conjunto de classes, cujo objectivo é passar a informação nelas contida para XML. Por coerência, toda a transformação de objectos para XML é realizada através dessas classes, mesmo que fosse possível a utilização de **XmlDocument**. A figura 5.1 apresenta o conjunto destas classes e respectivas relações.

A transformação de XML para objectos é realizado através de um conjunto de fábricas de objectos. De forma a unificar a utilização destas fábricas, foi criada uma classe base, **XmppReader**. A leitura de elementos XML apenas é realizada sobre elementos enviados pelo cliente, que ao contrário do servidor, nunca acontece uma situação em que o elemento possa estar mal formado, por não se encontrar no contexto do elemento global. Isto permite que seja utilizada a classe **XmlDocument** da *framework* .NET para leitura dos elementos XML. Na figura 5.2 é apresentada a hierarquia de fábricas.

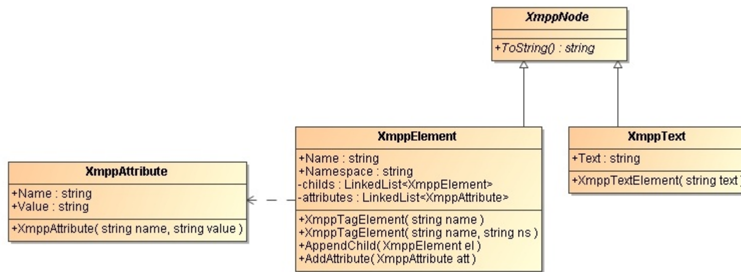


Figura 5.1: Hierarquia de classes para transformação em XML

É possível verificar pela figura 5.2, que tal como na transformação para XML, o elemento <stream/> também é tratado de forma diferente na transformação de XML. Existe também uma fábrica para realizar o *parse* do elemento <stream/>, **StreamFactory**, mas esta não deriva de **XmppReader**, pois **XmppStream** também não deriva de **SimpleElement**. Embora acabe por seguir um padrão semelhante.

Após aceder aos dados é necessário transferi-los para a entidade que os requisitou, utilizando para isso *Data Transfer Objects* (DTO). Neste componente estão definidos as classes usadas para conter a informação obtida da base de dados.

Foram criadas as classes **Connection** e **Session** que servem para tornar um determinado aspecto da sessão mais simples. Aos objectos destas classes chamam-se objectos de suporte ao *stream*.

Nas seguintes subsecções são detalhados os conceitos aqui apresentados.

5.2.1 Elementos XMPP

As principais classes da hierarquia, **SimpleElement**, **SimpleElementVariableChildren** e **StanzaElement** são parcialmente apresentadas nas listagens 5.3, 5.4 e 5.5, respectivamente.

```
public abstract class SimpleElement
{
   public String Name { get; set; }
   public String Namespace { get; set; }

   public SimpleElement() { }
   public SimpleElement(string name)
   {
      Name = name;
   }
   public SimpleElement(string name, string ns)
   {
      Name = name;
      Namespace = ns;
   }
   //...
}
```

Listagem 5.3: Classe SimpleElement

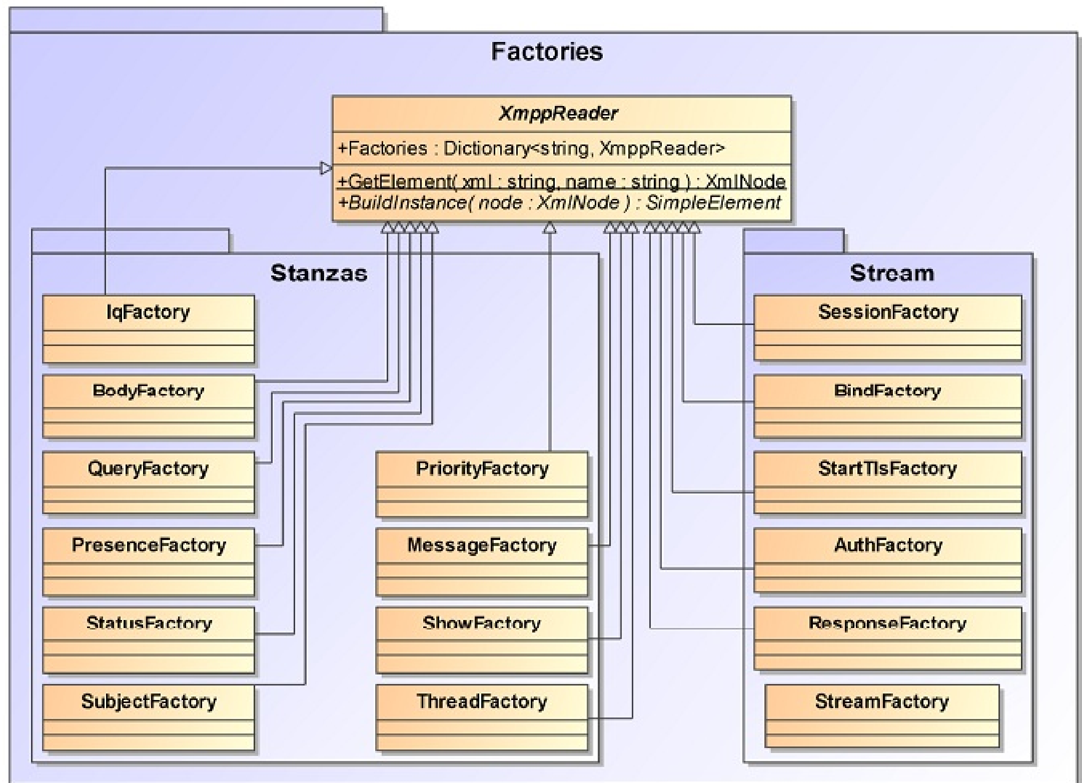


Figura 5.2: Hierarquia de fábricas para leitura de XML

Elementos para negociação do *stream*

A principal classe, que representa elementos XML, é aquela que representa o *stream* em si, a classe **XmppStream**. Devido à sua natureza, esta classe não deriva de nenhumas das mencionadas atrás, pois a sua construção ocorre durante toda a vida do *stream*.

A classe **SimpleElement** serve de base para classes que representam elementos XMPP que possuem elementos filhos fixos, enquanto que **SimpleElementVariableChildren** serve de base para classes que representam elementos XMPP que podem conter um número e tipo de elementos variado. Em seguida estão enumerados os tipos criados, que derivam de **SimpleElement** ou **SimpleElementVariableChildren**, para representar os restantes elementos XML, utilizados na negociação do *stream*.

- **Features**: elemento para anunciar as capacidades do servidor, no momento da negociação;

- **StartTls**: elemento que anuncia a capacidade de fornecer segurança a nível do canal, com TLS, e para indicar que quer realizar negociação TLS;

- **Proceed**: elemento que indica para prosseguir com a negociação TLS;

```
public abstract class SimpleElementVariableChildren : SimpleElement
{
   public LinkedList<SimpleElement> Children { get; private set; }

   public SimpleElementVariableChildren(string name) : base(name)
   {
      Children = new LinkedList<SimpleElement>();
   }

   public SimpleElementVariableChildren(string name, string ns)
      : base(name, ns)
   {
      Children = new LinkedList<SimpleElement>();
   }
   //...
}
```

Listagem 5.4: Classe SimpleElementVariableChildren

- **Mechanisms**: elemento que anuncia os vários mecanismos de autenticação SASL suportados pelo servidor;
- **Auth**: elemento com o mecanismo SASL escolhido e possivelmente uma resposta inicial, caso o mecanismo assim o definir;
- **Challenge**: elemento que contém o desafio de um mecanismo SASL;
- **Response**: elemento que contém uma resposta a um desafio de um mecanismo SASL;
- **Success**: elemento que indica que a autenticação SASL foi bem sucedida;
- **Bind**: elemento que anuncia a possibilidade de *resource binding* e que serve para iniciar esse mesmo processo;
- **Resource**: elemento que indica qual o `recurso` escolhido pelo cliente para realizar *resource binding*;
- **JID**: elemento que indica qual o *full* JID após *resource binding*;
- **Session**: elemento que anuncia a possibilidade de estabelecer uma sessão de *instant messaging* e serve para dar começo a esse mesmo processo;
- **Failure**: elemento que indica que ocorreu uma falha, algures no processo de negociação do *stream*.

Stanzas

As classes que representam os *stanzas*, tal como o nome indica derivam de **StanzaElement**, que possui já todos os atributos que um *stanza* pode ter. As classes definidas são **Message**, **Presence** e **Iq**, que tal como o nome indica, cada uma representa um dos *stanzas* apresentados no apêndice D.

```
public abstract class StanzaElement : SimpleElementVariableChildren
{
   //Static members

   public string From { get; set; }
   public string To { get; set; }
   public string Id { get; set; }
   public string Type { get; set; }
   public string XmlLang { get; set; }

   public StanzaElement(string name, string from, string to,
      string id, string type, string lang) : base(name)
   {
      From = from;
      To = to;
      Id = id;
      Type = type;
      XmlLang = lang;
   }
   //...
}
```

Listagem 5.5: Classe StanzaElement

A utilização destes elementos só por si não trás grande funcionalidade, desta forma a especificação[12] define já um conjunto de elementos que se pode encontrar dentro dos *stanzas*. Esses elementos são também referidos no apêndice D. Para cada um destes elementos existe também uma classe que o representa, que pode derivar de **SimpleElement** ou **SimpleElementVariableChildren**.

5.2.2 Escrita de elementos XMPP

A principal classe da hierarquia utilizada para transformar objectos em XML é **XmppNode** (tal como apresentado na figura 5.1). Esta classe representa um elemento XMPP de forma genérica, sendo estendida por duas outras classes. Uma delas representa um elemento que pode conter atributos e elementos filhos (instâncias de **XmppNode**), a outra representa um elemento textual (que apenas deve ser usado como filho de outros). Estas classes são, respectivamente, **XmppElement** e **XmppText**. Os atributos são representados pela classe **XmppAttribute**.

Estas classes são unicamente usadas na hierarquia de **SimpleElement**, em que cada elemento constrói a sua própria representação num **XmppNode** (**XmppElement** ou **XmppText**). Assim **SimpleElement** contém 2 novos métodos, **GetSelf()** e **ToXml()**. O método **GetSelf()** retorna uma instância de **XmppNode** e é virtual, de forma a ser implementado pelas classes que dela derivam (caso seja necessário). O método **ToXml()** retorna a representação do elemento em XML (através do elemento retornado por **GetSelf()**). A classe **SimpleElementVariableChildren** introduz ainda um novo método virtual, **GetAttributes()**, pois ela própria implementa o método **GetSelf()**, de forma a que as suas classes filhas não tenham que o fazer. Desta forma as suas classes filhas implementam **GetAttributes()**, para terem algum controlo sobre o elemento criado. Um complemento às listagens 5.3, 5.4 e 5.5 é

apresentado pelas listagens 5.6, 5.7 e 5.8.

```
public abstract class SimpleElement
{
   //...
   public string ToXml()
   {
      return GetSelf().ToString();
   }

   internal virtual XmppNode GetSelf() { return null; }
}
```

Listagem 5.6: Complemento da classe SimpleElement

```
public abstract class SimpleElementVariableChildren : SimpleElement
{
   //...
   internal override XmppNode GetSelf()
   {
      XmppElement element;
      if (Namespace == null)
         element = new XmppElement(Name);
      else
         element = new XmppElement(Name, Namespace);

      LinkedList<XmppAttribute> attributes = GetAttributes();
      if (attributes != null)
      {
         foreach (XmppAttribute att in attributes)
            element.AddAttribute(att);
      }

      foreach (SimpleElement se in Children)
      {
         element.AppendChild(se.GetSelf());
      }

      return element;
   }

   internal virtual LinkedList<XmppAttribute> GetAttributes()
      { return null; }
}
```

Listagem 5.7: Complemento da classe SimpleElementVariableChildren

Com todo este suporte, para enviar um determinado elemento XMPP para o cliente, basta criar uma instância do elemento a enviar, invocar o seu método **ToXml()** e escrever o resultado no *stream*. A listagem 5.9 apresenta um exemplo de utilização. Este exemplo consiste no envio do elemento <features/> com indicação da obrigatoriedade da negociação TLS e os mecanismos de autenticação disponíveis.

```
public abstract class StanzaElement : SimpleElementVariableChildren
{
   //...
   internal override LinkedList<XmppAttribute> GetAttributes()
   {
      LinkedList<XmppAttribute> attributes =
         new LinkedList<XmppAttribute>();
      attributes.AddLast(
         new XmppAttribute(TYPE_ATTRIBUTE, Type));

      if (From != null)
         attributes.AddLast(new XmppAttribute(FROM_ATTRIBUTE, From));
      if (To != null)
         attributes.AddLast(new XmppAttribute(TO_ATTRIBUTE, To));
      if (Id != null)
         attributes.AddLast(new XmppAttribute(ID_ATTRIBUTE, Id));
      if (XmlLang != null)
         attributes.AddLast(new XmppAttribute(XML_LANG_ATTRIBUTE,
            XmlLang));
      return attributes;
   }
}
```

Listagem 5.8: Complemento da classe StanzaElement

```
Connection conn = new Connection(client.GetStream());
//...
Features features = new Features();
StartTls stls = new StartTls();
Mechanisms ms = new Mechanisms(SaslComponent.GetAvailableMechanisms());
features.Children.AddLast(stls);
features.Children.AddLast(ms);

xml = features.ToXml();
conn.Write(xml);
```

Listagem 5.9: Exemplo de envio de um elemento <features/> para o cliente

5.2.3 Leitura de elementos XMPP

Às classes derivadas de **XmppReader** basta-lhes implementar o método **BuildInstance**. Apenas foram criadas classes para realizar o *parse* dos elementos que um cliente XMPP pode enviar. Estas classes estão divididas em duas categorias, as fábricas que realizam *parse* a elementos de negociação do *stream* e as fábricas que realizam *parse* aos *stanzas* e seus elementos associados. Todas estas classes são apresentadas na figura 5.2. Na primeira categoria estão as classes: **AuthFactory**, **BindFactory**, **ResponseFactory**, **SessionFactory** e **StartTlsFactory**. Na segunda categoria estão as classes: **BodyFactory**, **IqFactory**, **MessageFactory**, **PresenceFactory**, **PriorityFactory**, **QueryFactory**, **ShowFactory**, **StatusFactory**, **SubjectFactory** e **ThreadFactory**. Pelos próprios nomes é possível verificar quais os elementos a que realizam o *parse*.

5.2.4 *Data Transfer Objects*

As classes criadas para transferir a informação obtida da base de dados para quem a requisitou foram **User** e **Contact**. Estas classes contêm, respectivamente, a informação de um utilizador (para autenticação) e a informação respectiva a um item de um *roster*.

É possível verificar que não existem DTOs para obter a informação de mensagens *offline* ou *stanzas* para subscrição de presença. A informação obtida é directamente retornada em objectos das classes que representam os respectivos *stanzas*. Tendo em conta que essa informação é obtida para ser entregue aos clientes, é imediatamente obtida no formato ideal para o realizar.

5.2.5 Objectos de suporte ao *stream*

Para auxiliar o funcionamento do servidor, tal como já foi indicado anteriormente, foram criados duas classes, **Connection** e **Session**, que são descritas em seguida.

- **Connection**: tem como objectivo auxiliar na escrita e leitura do *stream*. Funciona como um *wrapper* sobre o *stream* .NET;
- **Session**: representa uma sessão, com toda a informação que lhe é inerente, como por exemplo *username* e `recurso` associado. É construída ao longo da negociação do *stream*.

5.3 Aplicação *Web* para registo

Antes que um cliente se possa ligar ao *bus* de *instant messaging*, o utilizador deve registar uma conta.

Existe uma extensão para o protocolo XMPP que define uma forma de registo de um utilizador XMPP, antes que este se autentique, a XEP-0077, denominada de "*In-Band Registration*".

A extensão XEP-0077 não será implementada por duas razões. Em primeiro lugar a implementação completa do protocolo XMPP não é o objectivo principal do trabalho, em segundo lugar, este tipo de registo não permite que utilizadores não XMPP efectuem o seu registo.

Com o objectivo de criar uma forma única de registo, foi criada uma pequena aplicação *Web*, que permite que todos os utilizadores dos serviços suportados se registem no *bus* de *instant messaging*. A figura 5.3 apresenta uma visualização da página *Web* desenvolvida.

Figura 5.3: Visualização da página de registo

Capítulo 6

Servidor XMPP

Este capítulo apresenta pormenores de implementação das principais funcionalidades do servidor XMPP, juntamente com a descrição dos componentes por elas responsáveis, apresentados na figura 4.1. Estas funcionalidades são apresentadas de seguida, indicando qual o componente que a implementa.

- Gestão de ligações TCP: *Connection Manager*;
- Gestão de sessões de *instant messaging*: *Session Manager*;
- Processamento de *stanzas*: *Request Manager*.

As classes que implementam os componentes *Connection Manager* e *Session Manager*, implementam o padrão *Singleton*, de forma a apenas existir uma única instância da classe. O que acontece pois estas classes possuem o estado da informação vital para o funcionamento do servidor. A classe que implementa o componente *Request Manager* não implementa este padrão, pois tem como único objectivo processar os *stanzas* que lhe são entregues, não armazenando qualquer tipo de informação. Desta forma é definida com uma classe estática, tendo apenas métodos estáticos.

A implementação do servidor será realizada na plataforma .NET, como já foi referido anteriormente.

6.1 Gestão de ligações TCP

A gestão das ligações TCP consiste em aceitar novas ligações e gerir as já existentes. A gestão das ligações em si consiste no armazenamento das mesmas para realizar todos os acessos de leitura e escrita. Esta funcionalidade é implementada pelo componente *Connection Manager*, que é um dos principais componentes do servidor.

A negociação da ligação é um processo sequencial, bem definido e independente da gestão da mesma, além de ser controlado por outro componente (*Session Manager*). Devido à sua natureza, este processo ocorre de forma síncrona entre o cliente e o servidor. Por outro

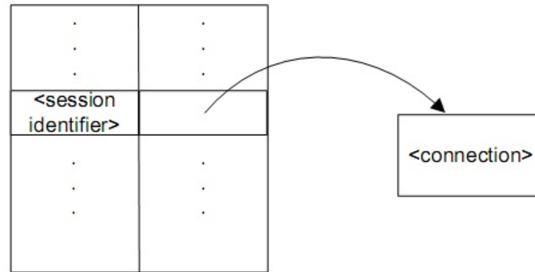


Figura 6.1: Ilustração de uma entrada em ***connections***

lado, a gestão das ligações ocorre sempre de forma assíncrona, pois trata-se de um processo completamente aleatório, não existindo qualquer sequência de acções definida. O cliente pode enviar um *stanza* ao servidor a qualquer altura, o que torna inviável que o servidor fique bloqueado à espera de informação. As escritas podem ser muitas vezes executadas durante o processamento de um *stanza*, podendo implicar escritas para vários *streams*. De forma a acelerar o conjunto das escritas, cada uma delas será efectuada de forma assíncrona. Assim todas as leituras e escritas são iniciadas de forma assíncrona e completadas por *IO completion threads* do *ThreadPool*.

A gestão das ligações passa também pelo armazenamento das mesmas, tal como foi dito anteriormente, por forma a poder usá-las para receber e enviar informação a qualquer altura. Mas manter as ligações não basta, é necessário associar uma ligação a uma sessão de *instant messaging*, por forma a saber qual o `recurso` de que utilizador realizou o pedido. Assim este componente armazena um dicionário (chamado ***connections***), em que cada entrada tem como chave o identificador da sessão e como valor o objecto ligação (instância de **Connection**, referida em 5.2.5). A figura 6.1 ilustra uma entrada nesse dicionário.

O acesso a este dicionário é apenas realizado na posse do *lock* associado ao próprio dicionário.

A classe responsável por implementar este componente é a classe **ConnectionManager**.

6.1.1 Recepção de ligações

As novas ligações são recebidas pelo porto TCP 5222 (tal como referido em 2.1.2, é a porta definida pela especificação para comunicação entre servidor e cliente). O componente inicia uma espera assíncrona sobre o porto, cuja função de *callback* será executada por uma *IO completion thread*. Este *callback* deve ter um tempo de execução curto, de forma a libertar a *IO completion thread* o mais rápido possível. Logo a sua execução consiste em apenas iniciar um nova espera assíncrona e iniciar a negociação do *stream* XML numa *worker thread* do **ThreadPool**.

Se a negociação for bem sucedida, será devolvido um par <identificador de sessão, instância de **Connection**>, que será adicionado a ***connections***. De seguida será iniciada a primeira leitura assíncrona sobre a ligação. A listagem 6.1 apresenta um excerto de código usado na recepção de uma ligação.

```
TcpClient client = (TcpClient)obj;

SessionManager manager = SessionManager.GetInstance();
KeyValuePair<string, Connection>? sp = manager.Negotiation(client);

if (sp.HasValue)
{
    //Key: sessionIdentifier
    //Value: connection
    lock (connections)
    {
        connections.Add(sp.Value.Key, sp.Value.Value);
    }
    sp.Value.Value.BeginRead(this.ReadCallback, sp.Value);
}
else
{
    try
    {
        client.Close();
    }
    catch (IOException) { }
}
```

Listagem 6.1: Excerto de código para recepção de uma ligação

6.1.2 Assincronismo

Um servidor de *instant messaging* deve estar constantemente disponível para receber *stanzas* de clientes, que podem surgir a qualquer momento. Tendo em conta que um qualquer sistema de *instant messaging* deve realizar comunicação em tempo quase-real, a utilização de uma solução assíncrona ganha força, contra uma solução de *pooling* periódico, por exemplo. A utilização de uma *thread* dedicada a cada ligação é inviável, pois com uma utilização massiva do servidor tornava-se incomportável a manutenção das mesmas. Com tudo isto a melhor solução é realizar as operações de leitura e escrita de forma assíncrona.

Recepção de *stanzas*

A leitura, tal como já foi referido, é efectuada de forma assíncrona, o que implica que a notificação da recepção de um novo *stanza* seja realizada através de um *callback*. Este *callback* irá obter a informação que o cliente enviou e transmiti-la de imediato ao *Session Manager*, para que o seu processamento seja iniciado.

É de notar que neste momento não é iniciada uma nova leitura assíncrona, pois não é expectável que um mesmo cliente realize vários pedidos em paralelo. Desta forma a nova leitura assíncrona apenas é iniciada após o processamento do pedido corrente.

A listagem 6.2 apresenta o *callback* para terminar a operação de leitura.

Enviar stanzas

O envio de *stanzas* é também realizado de forma assíncrona, embora não pelos mesmos motivos que a recepção. Enquanto que a recepção de um *stanza* pode ocorrer a qualquer

```
private void ReadCallback(IAsyncResult iar)
{
   KeyValuePair<string, Connection> pair =
      (KeyValuePair<string, Connection>)iar.AsyncState;

   string xml = null;
   try
   {
      xml = pair.Value.EndRead(iar);
   }
   catch (IOException)
   {
      //Terminar a ligação
      //...
   }

   SessionManager.GetInstance().NewRequest(xml, pair.Key);
}
```

Listagem 6.2: *Callback* de leitura

momento no tempo, o mesmo não acontece com o envio, que ocorre em momentos bem definidos (após o processamento dos mesmos). A principal motivação para realizar as operações de escrita de forma assíncrona apresenta-se em situações em que é necessário enviar *stanzas* a vários clientes, como resultado do processamento de um *stanza*. Uma situação destas ocorre, por exemplo, quando um cliente realiza broadcast da sua presença. Em casos destes é vantajoso que a escrita seja efectuada de forma assíncrona, não sendo necessário terminar a escrita para uma ligação para começar a escrever para outra.

De acordo com o processamento de *stanzas*, não é possível serem realizadas várias operações de leitura, em simultâneo, na mesma ligação, mas é possível ocorrerem várias escritas simultâneas. Poderá acontecer uma situação em que dois utilizadores enviem uma mensagem a um outro utilizador, num momento de tempo próximo o suficiente para que quando a segunda escrita seja iniciada, a primeira ainda se encontre em curso. Isto é algo que logicamente não faz sentido, pois a informação escrita seria intercalada e completamente imperceptível. Desta forma foi adicionado algum suporte à classe **Connection** para gerir esta necessidade. Sempre que se desejar iniciar uma escrita sobre a ligação, deve-se verificar se não existe já outra em curso. Se não existir, a operação será iniciada, caso contrário a informação a escrever será armazenada na instância de **Connection**, numa fila, para ser escrita mais tarde. A necessidade de uma fila para pedidos de escrita pendentes tem a mesma origem que a necessidade de escritas assíncronas. Nos casos em que é necessário realizar escritas em vários *streams* (como por exemplo, *broadcast* de informação de presença), se existir uma escrita em curso num desses *streams*, a *thread* não pode ficar bloqueada à espera que essa escrita termine. Desta forma a informação é armazenada na fila de espera. Na execução do *callback* de escrita é verificado se existe informação na fila e em caso positivo, será iniciada uma nova escrita assíncrona com essa informação. O acesso a esta fila é realizado em exclusão mútua.

Tal como na leitura, a escrita apenas termina quando for executado o respectivo *callback*. Após o término de uma operação de escrita é possível iniciar outra, assim é neste *callback*

que é verificada a existência de informação pendente para escrita. Se existir, é imediatamente iniciada uma operação de escrita com essa mesma informação.

A listagem 6.3 apresenta o *callback* para terminar a operação de escrita.

```
private void WriteCallback(IAsyncResult iar)
{
   KeyValuePair<string, Connection> pair =
      (KeyValuePair<string, Connection>)iar.AsyncState;

   try
   {
      lock (pair.Value)
      {
         pair.Value.EndWrite(iar);
         Connection conn = pair.Value;
         string message = conn.GetWriteJob();
         if (message != null)
         {
            if (message.Equals(Connection.CLOSE_STREAM_INDICATION))
            {
               conn.Close();
               return;
            }
            conn.BeginWrite(message, this.WriteCallback,
               new KeyValuePair<string, Connection>(pair.Key, conn));
         }
      }
   }
   catch (IOException)
   {
      //Terminar a ligação
      //...
   }
}
```

Listagem 6.3: *Callback* de escrita

6.1.3 Terminação da ligação

A terminação da ligação consiste, do ponto de vista do *Connection Manager*, em remover a respectiva entrada de ***connections***, enviar o fecho do elemento <stream:stream/> e fechar a ligação.

O *Connection Manager* irá fechar a ligação na ocorrência de excepções na comunicação com o cliente.

6.2 Gestão de sessões de *instant messaging*

Esta gestão é da responsabilidade do componente *Session Manager*, que deve negociar novos *streams* XML, por forma a criar uma sessão de *instant messaging*, e manter as sessões já existentes.

O processo de negociação do *stream* deve seguir as operações apresentadas em 2.2, enunciadas de seguida.

- Negociação da segurança do canal, através de TLS;
- Autenticação do utilizador, através de SASL;
- Associar um `recurso` ao *stream* criado;
- Estabelecer uma sessão para realizar operações de *instant messaging* e *presence*.

O *Connection Manager*, ao receber um *stanza* no *callback* de leitura, transmite-o de imediato para o *Session Manager*, tal como se pode verificar na listagem 6.2, invocando o método ***NewRequest()***. O *callback* é executado no contexto de uma *IO completion thread* do *ThreadPool*, cujo objectivo é terminar a operação de IO, o mais rápido possível, de forma a estar disponível para atender outros pedidos. O método ***NewRequest()*** recebe o stanza e o identificador da sessão pela qual o *stanza* foi enviado. O *Session Manager* deve realizar o *parse* do *stanza*, de forma a obter a sua representação em objecto, e obter o objecto sessão associado ao identificador recebido. O objecto que representa o *stanza* será então processado pelo *Request Manager*, no contexto da sessão pela qual foi enviado. A conjunção de todas estas operações faz com que o processamento total associado à recepção de um *stanza* se torne demasiado demorado para ser executado por uma *IO completion thread*. Desta forma, todo este processamento deve ser realizado no contexto de um nova *thread*, neste caso uma *worker thread* do *ThreadPool*.

Este componente tem três dicionários para tornar mais fácil obter o objecto sessão a partir de um *bare* JID ou através de um *full* JID. Esses dicionários são:

- ***users***: tem como chave um *username* e como valor uma lista de *full* JIDs, que indicam os endereços dos `recursos` que se encontram ligados de momento;
- ***resources***: tem como chave um *full* JID e como valor o identificador da respectiva sessão;
- ***sessions***: tem como chave o identificador da sessão e como valor o próprio objecto sessão (instância de **Session**).

O acesso a estes contentores é sempre realizado em exclusão mútua, na posse do *lock* associado a um objecto criado para o efeito (***sessionsLock***). A figura 6.2 ilustra estes dicionários e as suas relações.

A classe **SessionManager** é responsável pela implementação deste componente.

6.2.1 Negociação do *stream*

O processo de negociação é implementado pelo método ***Negotiation()*** de **SessionManager**, que segue a sequência apresentada na figura 6.3.

Este método começa por iniciar uma instância de **Connection** (conn) e iniciar uma instância de **Session** (session). Esta última será construída ao longo de todo o processo.

No início do processo de negociação decorrerá uma troca de *tags* de abertura do elemento <stream/> entre cliente e servidor. De seguida é iniciada a negociação, que como se pode ver

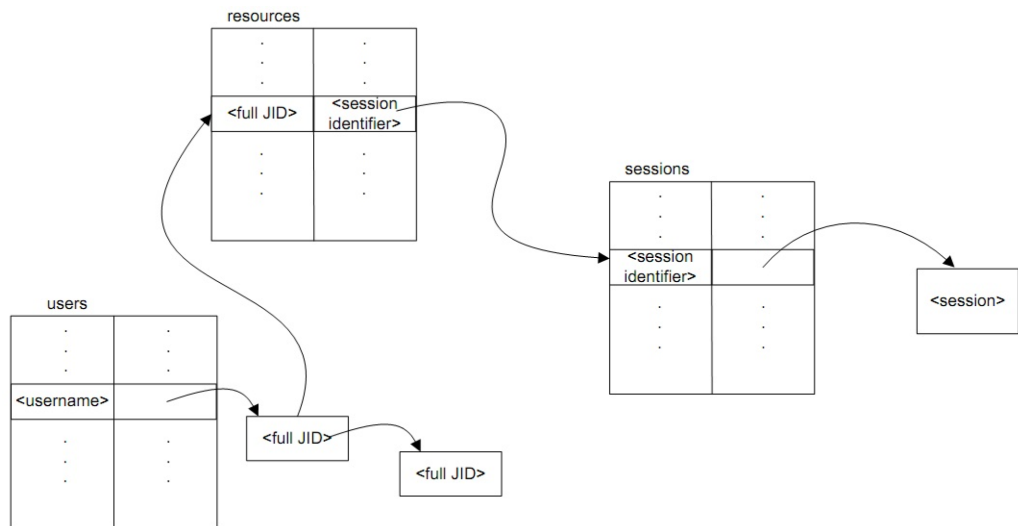


Figura 6.2: Ilustração dos dicionários de suporte a *Session Manager*

na figura 6.3, começa com o envio do elemento <features/>, anunciando a obrigatoriedade da negociação TLS e os mecanismos de autenticação disponíveis. Este elemento é construído através de um objecto da classe **Features**, que possui como filhos, uma instância de **StartTls** e outra de **Mechanisms** (este código está apresentado na listagem 5.9, como exemplo de escrita de elementos XMPP).

TLS

A negociação TLS, referida em 2.2.1, é levada a cabo pela classe auxiliar **TlsComponent**. Este componente utiliza a classe **SslStream**, disponível na *framework* .NET, para construir o *stream* cifrado. A classe **SslStream** implementa, juntamente com outros, o protocolo TLS, que é o protocolo definido para cifra do *stream* pela especificação RFC3920[11].

O certificado utilizado pelo servidor é obtido gratuitamente (através de `http://www.startssl.com/`), tendo como *Certification Authority* a *StartCom Certification Authority*.

SASL

A autenticação SASL, tal como referida em 2.2.2, é implementada pela classe **SaslComponent**. Esta classe tem 2 métodos: ***GetMechanism()*** e ***Authenticate()***. O método ***GetMechanism()*** tem como objectivo a negociação do mecanismo que deve ser usado pra autenticação, devolvendo um objecto que o implemente. O método ***Authenticate()*** realiza a autenticação, usando um determinado mecanismo (obtido na negociação realizada por ***GetMechanism()***).

O processo de autenticação foi definido de forma a que funcione com qualquer mecanismo, o que levou à definição de uma classe genérica, que define as funcionalidades que cada mecanismo deve implementar. A classe chama-se **SaslMechanism** e é apresentada na listagem 6.4.

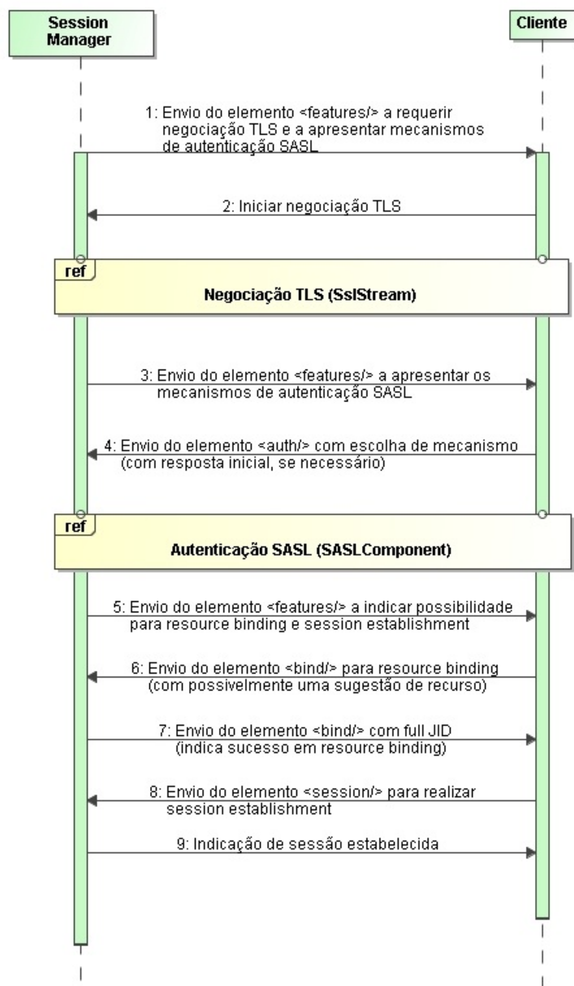


Figura 6.3: Diagrama de sequência para negociação do *stream*

Neste momento, o único mecanismo implementado é o mecanismo PLAIN, referido em 2.2.2.

Tendo uma definição unificada dos mecanismos de autenticação, o processo que deve ser seguido para realizar autenticação está apresentado na listagem 6.5.

A existência de uma resposta inicial enviada pelo cliente está dependente do mecanismo a usar. Se o mecanismo não definir uma resposta inicial, ser-lhe-á fornecido o valor *null*, tornando o processamento inicial numa simples geração de um primeiro desafio.

O nome de utilizador, para ser aceite na autenticação, deve estar de acordo com o perfil *Nodeprep* (implementado parcialmente pelo método ***NodePrep()*** da classe **Stringprep**) da especificação *Stringprep* [5].

```
public abstract class SaslMechanism
{
        public enum Result
        {
                SUCCESS,
                FAILED,
                CONTINUE
        }

        public abstract Result ProcessResponse(string response,
                out string nextChallenge, out string username);
}
```
Listagem 6.4: Classe abstracta SaslMechanism

```
{
<mechanism, initialResponse> = sasl.GetMechanism(conn);

authenticated = sasl.Authenticate(conn, mechanism, initialResponse):
   response = initialResponse;
   do
   {
      <res, cha> = mechanism.ProcessResponse(response);

      if(res == SaslMechanism.Result.SUCCESS)
         send_success();
      if(res == SaslMechanism.Result.FAILED)
         send_failure();

      send_challenge(cha);

      response = receive_response();
   }
   while(res == SaslMechanism.Result.CONTINUE);

return authenticated;
}
```
Listagem 6.5: Algoritmo de autenticação

Resource binding

A classe que implementa *resource binding*, referido em 2.2.3, é a classe **ResourceManager**. Esta classe tem como método principal, o método ***NegotiateResource()***, que trata de negociar com o cliente qual o `recurso` a associar ao *stream*. A associação de um `recurso` ao *stream* é realizada através de troca de *stanzas* <iq/> (os elementos indicados nos passos 6 e 7 da figura 6.3 são enviados dentro de *stanzas* <iq/>). Desta forma é utilizada a fábrica **IqFactory** para construir os objectos **Iq**, segundo a informação recebida, e são criados novos objectos **Iq** para responder ao cliente.

Existe a possibilidade de o cliente não fornecer um `recurso`, mas ainda assim desejar realizar *resource binding*. Nesse caso o próprio servidor irá gerar um `recurso` único para o utilizador em questão.

Se o cliente fornecer um `recurso` para associar ao *stream*, o processo poderá falhar se alguma das seguintes condições se verificar:

- O utilizador já atingiu o número máximo de `recursos` ligados ao servidor;
- O `recurso` já se encontra em uso;
- O valor do `recurso` não se encontra em conformidade com o perfil *Resourceprep* (implementado parcialmente pelo método ***ResourcePrep()*** da classe **Stringprep**), segundo a especificação *Stringprep* [5].

Caso nenhuma das condições atrás apresentadas seja verificada, então o `recurso` é aceite e o processo encontra-se finalizado.

Estabelecimento de sessão

Por fim, deve ser estabelecida uma sessão, para que o cliente utilize as funcionalidades de *instant messaging* fornecidas pelo servidor, tal como referido em 2.2.4. Esta operação é implementada pelo método ***EstablishSession()*** de **SessionManager**, usando *stanzas* <iq/>, tal como em *resource binding* (o elemento indicado no passo 8 da figura 6.3 é enviado dentro de um *stanza* <iq/>).

É utilizada a fábrica **IqFactory** para obter a instância **Iq** que representa a informação recebida. Se essa informação for referente ao estabelecimento de uma sessão (contém o elemento <session/>) é utilizada outra instância de **Iq** de resultado, para indicar o sucesso da operação.

6.2.2 Recepção de novo *stanza*

O *Session Manager* pode receber um *stanza* de duas formas, vindo do *Connection Manager* ou vindo do *IM Module Manager*. Estas formas de recepção são realizadas através de redefinições do método ***NewRequest()***.

O *IM Module Manager* entrega um *stanza* já no objecto que o representa, daí não será necessário realizar o *parse*. Tendo em conta que o *Session Manager* não possui sessões de utilizadores de outros serviços, será criada uma sessão externa temporária para a execução do *stanza* por parte do *Request Manager*. A *thread* utilizada para realizar esta operação não é uma *IO Completion thread*, logo será esta a *thread* utilizada para realizar todo o processamento do *stanza*.

O processamento que um *stanza* necessita, vindo do *Connection Manager*, pela razões apresentadas atrás, será realizado no contexto de uma *worker thread* do **ThreadPool**.

As próximas duas subsecções ("Utilização do **ThreadPool**" e "*Parse* do *stanza*") apenas são relativas a *stanzas* recebidos por parte do *Connection Manager*.

Utilização do ThreadPool

O método ***NewRequest()*** (utilizado pelo *Connection Manager*) tem assim o objectivo de requisitar a execução de um trabalho ao **ThreadPool**, colocando-o na sua fila, de forma a que

esse trabalho seja executado por uma *worker thread* disponível. O trabalho consiste na execução do método ***ThreadRun()***, que recebe o *stanza* e o identificador da sessão pela qual o *stanza* foi enviado. Desta forma o *stanza* será processado no contexto de uma nova *thread*, permitindo à *IO completion thread* atender mais pedidos de IO.

Parse do *stanza*

O *parse* dos *stanzas* é realizado da mesma forma que o *parse* dos restantes elementos, por via de uma fábrica. Existem três fábricas para *stanzas*, uma por cada tipo, tal como referido em 5.2.3. A escolha da fábrica a utilizar depende do nome do elemento.

Processamento - *Session Manager*

Por fim, após obter o objecto que representa o *stanza* e o objecto sessão correspondente ao identificador recebido, o *stanza* pode ser processado pelo *Request Manager*. Caso a recepção tenha sido efectuada pelo *IM Module Manager*, os passos atrás apresentados não terão sido necessários e este passo será imediatamente executado.

O processamento é iniciado invocando o método estático ***Process()*** de **RequestManager**. Este método é genérico, podendo assim receber todos os tipos de *stanzas*, sendo ele responsável por reencaminhá-lo para o processador adequado.

A listagem 6.6 apresenta as operações que o método ***ThreadRun()*** realiza, as operações atrás enunciadas, em pseudo-código.

6.2.3 Leituras e escritas

No *callback* de leitura, em *Connection Manager*, foi mencionado que não era iniciada uma nova leitura assíncrona de imediato, pois não era expectável que o cliente realizasse dois pedidos em simultâneo. Esta nova leitura apenas será iniciada após o processamento do *stanza* recebido, ficando isso a cargo do *Request Manager*, sendo este o componente responsável pelo processamento do *stanza*.

Os componentes são executados de forma sequencial, começando pelo *Connection Manager*. Cada um dos componentes fornece um nível de indirecção para auxiliar o seguinte. Desta forma o *Request Manager* não comunica directamente com o *Connection Manager*, sendo necessário realizar essa comunicação através do *Session Manager*. Com o objectivo de satisfazer esta necessidade, foram criados métodos em *Session Manager* para requisitar ao *Connection Manager* a realização das operações de leitura e escrita. Estes métodos não servem apenas para invocar métodos do *Connection Manager*, introduzem também algum processamento que é da responsabilidade do próprio *Session Manager*.

```
{
   sid = state.Key;
   stanza = state.Value;

   if(IsStreamClosingTag(stanza))
   {
      TerminateSession(sid);
      ConnectionManager.TerminateConnection(sid);
      return;
   }
   name = GetStanzaName(stanza);
   factory = GetStanzaFactory(name);

   if(name == null || factory == null)
   {
      //Envia erro para o cliente
      ConnectionManager.ContinueListening(sid);
      return;
   }

   object = factory.BuildInstance(stanza);
   if (stanza == null)
   {
      //Envia erro para o cliente
      ConnectionManager.ContinueListening(sid);
      return;
   }

   lock (sessionsLock)
   {
      if(sessions.ContainsKey(sid))
      sess = sessions[sid];
   }

   try {
      if (sess != null) RequestManager.Process(sess, stanza);
   } catch (Exception e) { Console.WriteLine(e.Message); }
}
```

Listagem 6.6: Método ***ThreadRun()***

Continuar a leitura

O método para leitura chama-se ***ContinueListening()*** e tem como objectivo indicar ao *Connection Manager* para iniciar uma nova leitura assíncrona sobre a sessão que este recebeu. Este método é apresentado na listagem 6.7.

```
public void ContinueListening(Session origin)
{
   if(origin.Server.Contains(SERVER_NAME))
      ConnectionManager.GetInstance().ContinueListening(
         origin.Identifier);
}
```

Listagem 6.7: Método ***ContinueListening()***

Como se pode verificar na listagem 6.7, este método apenas surtirá efeito, se a sessão sobre a qual for invocado seja uma sessão interna. O módulo de tradução irá tratar da comunicação com respectivo utilizador.

Enviar *stanza*

O envio de *stanzas* pode ser feito de várias formas, conforme o próprio processamento do *stanza* que origina esse envio. Desta forma foram criados os seguintes métodos para envio de *stanzas*:

- ***SendStanzaToFullJID()***: este método tem como objectivo enviar o *stanza* a um determinado `recurso`, indicando o endereço do mesmo;
- ***SendStanzaToBareJID()***: este método tem como objectivo enviar o *stanza* a todos os `recursos` de um determinado utilizador;
- ***SendStanzaToSession()***: este método tem como objectivo enviar o *stanza* a um determinado `recurso`, indicando o objecto sessão a ele associado.

É de notar que todos estes métodos recebem o objecto que representa o *stanza* a enviar, sendo da sua responsabilidade obter o elemento XML correspondente. Isso é realizado utilizando o método ***ToXml()*** do objecto, pois tal como foi referido em 5.2.2, todos os objectos têm a capacidade de construir a sua representação em XML.

Estes métodos são invocados conforme o processamento do *stanza*, daí todos devem estar preparados para enviar a informação ao *IM Module Manager*, se o destinatário do *stanza* for um utilizador externo. A separação dos utilizadores é realizada através da parte servidor do endereço destinatário (ou campo ***Server***, no caso em que é utilizado um objecto **Session**). Se o utilizador destinatário for externo, o *stanza* será enviado ao *IM Module Manager* para que este o redireccione correctamente.

6.2.4 Obtenção de informação de presença

A informação de presença é armazenada pelo servidor, para que quando for necessário, seja imediatamente enviada ao seu destinatário. Isto acontece quando um utilizador se torna disponível e deve receber a informação de presença de todos os contactos de quem possui uma subscrição de presença.

Esta informação é armazenada no objecto de sessão correspondente ao `recurso` que a enviou. Tendo em conta que, tal como já foi referido anteriormente, não existem objectos sessão para os utilizadores externos, o servidor não possui a sua informação de presença.

Este problema é simplesmente resolvido com um conceito do protocolo XMPP. Os *stanzas* de presença com ***type='probe'*** são utilizados em redes de servidores XMPP, para obter a informação de presença de um utilizador. Quando um servidor necessita da informação de presença de um determinado utilizador, envia um destes *stanzas* ao servidor a que este se encontra ligado.

Quando for necessário obter a informação de presença de um utilizador externo, será construído um stanza de presença com ***type=*'*probe*'**, que será enviado ao *IM Module Manager*. Por sua vez, este irá enviar o *stanza* ao módulo correcto que irá utilizar a operação ***SendPresence*** do *Web Service* Central para enviar a informação de presença requisitada.

6.3 Processamento de *stanzas*

O componente responsável pelo processamento "pesado", ou seja, todo o processamento específico de cada tipo de *stanza*, é o componente *Request Manager*. Este componente é implementado pela classe **RequestManager**.

Para o processamento de cada tipo de *stanza* foram desenvolvidos tipos específicos. De forma a unificar as definições desses tipos, foi definido um tipo abstracto que lhes servirá como base, **StanzaProcessor**. Os tipos criados são **MessageProcessor**, **PresenceProcessor** e **IqProcessor**, que processam, respectivamente, *stanzas* de mensagens, de presença e de IQ.

6.3.1 Definição de processador

Aos tipos utilizados para processar os *stanzas* chama-se "processadores". Como forma de definir a assinatura dos métodos para processar os *stanzas* foi criado um tipo abstracto, **StanzaProcessor**, apresentado na listagem 6.8.

```
abstract class StanzaProcessor
{
   public abstract void Process(Session sess, StanzaElement stanza);
}
```

Listagem 6.8: Classe abstracta **StanzaProcessor**

Os tipos processadores derivam então desta classe, o que permite que a invocação de todos os processadores seja realizada da mesma forma, independentemente do tipo de *stanza* a processar. Desta forma a definição do método ***Process()*** da classe **RequestManager** torna-se simples, sendo apenas necessário invocar o processador específico, conforme o tipo do stanza. De forma a ter uma melhor percepção sobre esta operação, a classe **RequestManager** é apresentada na listagem 6.9.

6.3.2 Processador de mensagens

O processador de mensagens, implementado pela classe **MessageProcessor**, tem um processamento simples, baseado nas regras de processamento, especificadas em [12] e mencionadas em 2.4.2.

As mensagens do tipo *'groupchat'* (tipos de mensagens enunciados em D.1) não são processados pelo servidor, pois estas não se enquadram no âmbito do trabalho. Inclusive o seu processamento são se encontra definido na extensão do protocolo XMPP para *instant messaging* e *presence*[12].

```
abstract class RequestManager
{
   private static Dictionary<Type, StanzaProcessor> processors;

   static RequestManager()
   {
      processors = new Dictionary<Type, StanzaProcessor>()
      {
         {typeof(Message), new MessageProcessor()},
         {typeof(Presence), new PresenceProcessor()},
         {typeof(Iq), new IqProcessor()}
      };
   }

   public static void Process(Session sess, StanzaElement stanza)
   {
      StanzaProcessor p = processors[stanza.GetType()];
      p.Process(sess, stanza);
   }
}
```

Listagem 6.9: Classe **RequestManager**

A listagem 6.10 apresenta o processamento realizado pelo **MessageProcessor**, no seu método ***Process()***, em pseuco-código.

Na listagem 6.10, as variáveis ***message*** e ***session*** são referentes ao objecto que representa o *stanza* de mensagem e o objecto sessão sobre a qual a mensagem foi enviada, respectivamente. No ponto (1) apenas será retornado um objecto sessão se o endereço em ***message.To*** for um *full* JID. O ponto (2) está a salvaguardar a entrega da mensagem. A única forma de chegar a este ponto é não existir qualquer `recurso` ligado, que esteja associado ao utilizador referido. Desta forma, a mensagem é armazenada para mais tarde ser entregue ao utilizador, quando um `recurso` seu se ligar ao servidor.

6.3.3 Processador de presença

O processador de presença, implementado pela classe **PresenceProcessor**, tem um processamento mais complexo que o processador de mensagens. Neste caso é necessário ter em conta os vários tipos possíveis (enunciados em D.2) para um *stanza* de presença e as suas diferentes funcionalidades.

Um *stanza* de presença sem tipo ou com tipo *'unavailable'* serve para enviar informação de presença. Se possuir atributo ***to***, essa informação será apenas enviada a um utilizador (presença direccionada), caso contrário será enviada a todos os utilizadores que se encontrem na sua lista de contactos com um estado de subscrição a *'from'* ou *'both'*. Os restantes tipos de *stanzas* de presença têm a sua própria funcionalidade, associada à gestão das subscrições de presença.

O processamento dos *stanzas* (descritos em maior pormenor de seguida) obedece às regras definidas pela especificação[12], enunciadas em 2.4.1.

A listagem 6.11 apresenta a separação de funcionalidades realizada no método ***Process()***

```
{
   if(message.Type == GROUPCHAT)
      throw Exception;

   if(!UserExists(message.To))
   {
      SessionManager.ContinueListening(session);
      SessionManager.SendStanzaToSession(serviceUnavailable, session);
      return;
   }

   s = SessionManager.GetUserSession(message.To);        (1)
   if(s == null)
      s = GetHighestPrioritySession(message.To);

   SessionManager.ContinueListening(session);

   message.From = session.FullJID;
   if(s != null)
      SessionManager.SendStanzaToSession(message, session);
   else
      StoreMessage(message, session);    (2)
}
```
Listagem 6.10: Processamento de mensagens

de **PresenceProcessor**.

Broadcast de presença

O broadcast de presença é realizado pelo método ***BroadcastPresence()***. A funcionalidade deste método está apresentada na listagem 6.12, em forma de pseudo-código.

Na listagem 6.11 o ponto (1) é referente a casos em que o cliente terminou a sessão, sem antes ter enviado um *stanza* de indisponibilidade. O próprio servidor deve garantir que esse *stanza* é enviado, através deste método. Nestes casos, não deve ser iniciada uma nova leitura assíncrona, pois a ligação com o cliente será fechada, se ainda não o foi.

Presença direccionada

O método ***DirectedPresence()*** trata de processar os *stanzas* de presença direccionada. A listagem 6.13 apresenta, em pseudo-código, as operações realizadas por este método.

O ponto (1) na listagem 6.13 verifica se o contacto tem ou não uma subscrição com valor *'from'* ou *'both'*. No caso negativo, o seu endereço deve ser armazenado, para que quando for realizado um broadcast de indisponibilidade, esse utilizador seja também notificado desse mesmo acontecimento (se isso ainda não se realizou de forma direccionada).

Subscribe

Um pedido de subscrição da informação de presença de outro utilizador é processado pelo método ***Subscribe()***, que é apresentado na listagem 6.14, em pseudo-código.

```
public override void Process(Session sess, StanzaElement stanza)
{
   //...

   switch (presence.Type)
   {
      case null:
      case Presence.UNAVAILABLE:
         if (presence.To == null)
            BroadcastPresence(sess, presence);
         else
            DirectedPresence(sess, presence);
         break;
      case Presence.SUBSCRIBE:
         Subscribe(sess, presence);
         break;
      case Presence.SUBSCRIBED:
         Subscribed(sess, presence);
         break;
      case Presence.UNSUBSCRIBE:
         Unsubscribe(sess, presence);
         break;
      case Presence.UNSUBSCRIBED:
         Unsubscribed(sess, presence);
         break;

      default:
         break;
   }

   if (!sess.Terminating)                              (1)
      SessionManager.GetInstance().ContinueListening(sess);
}
```

Listagem 6.11: Método ***Process()*** de **PresenceProcessor**

A listagem 6.14 começa com dois blocos, (1) e (2). Nestes blocos é verificado se as respectivas entradas existem nos *rosters* do utilizador e do contacto, agindo em conformidade, segundo as regras enunciadas em 2.4.1. No ponto (3) são enviados *roster pushs* a todos os `recursos` do utilizador que tenham pedido o *roster*. Este *roster push* deve conter o atributo ***ask*** com o valor *'subscribe'*.

Subscribed

Quando o contacto aceita um pedido de subscrição, anuncia-o com o envio de um *stanza 'subscribed'*, que será processado pelo método ***Subscribed()***, de acordo as regras descritas em 2.4.1. Este método está apresentado, através de pseudo-código, na listagem 6.15.

Nos pontos (1) e (2) da listagem 6.15 os estados das respectivas subscrições são alterados, conforme é referido em 2.4.1. No ponto (3) são enviados *stanzas* com a informação de presença de todos os `recursos` do utilizador actual, para o utilizador que enviou o *stanza 'subscribe'* original.

```
{
   presence.From = session.FullJID;
   StorePresenceForBroadcast(presence);

   if(presence.Type == null)
   {
      if(!session.Active)
      {
         presences = GetPresencesFromFriends(session);
         session.Active = true;
         foreach(p in presences)
            SessionManager.SendStanzaToSession(p, session);
      }
   }
   else
   {
      //type = 'unavailable'
      session.Active = false;
      SendPresenceToDirected(presence, session);
   }

   SendPresenceToFriends(presence, session);
   SendPresenceToOwnResources(presence, session);
   SendOfflineMessages(session);
}
```

Listagem 6.12: Método ***BroadcastPresence()*** de **PresenceProcessor**

Unsubscribe

Quando o utilizador deseja cancelar a subscrição da presença de outrem, envia um *stanza 'unsubscribe'*, que será processado pelo método ***Unsubscribe()***. Este método segue um padrão muito semelhante ao apresentado em pseudo-código na listagem 6.15. As diferenças encontram-se nas alterações realizadas aos estados de subscrição, nos pontos (1) e (2), tal como é referido em 2.4.1. No ponto (3) são enviados *stanzas* de indisponibilidade (referente à presença do contacto) para o remetente do *stanza* em processamento.

Unsubscribed

Ao rejeitar um pedido de subscrição é enviado um *stanza 'unsubscribed'*, que é processado pelo método ***Unsubscribed()***. Este método é apresentado, em pseudo-código, na listagem 6.16.

Os blocos (1) e (2), na listagem 6.15, mais uma vez, apresentam as respectivas alterações aos estados das subscrições, tal como enunciados em 2.4.1. O ponto (3) indica que apenas será realizado o *roster push* aos `recursos` do utilizador que realizou o pedido em primeiro lugar, tendo em conta que não foi realizado qualquer *roster push* para os `recursos` do contacto (utilizador actual), quando este recebeu o *stanza 'subscribe'*.

```
{
   if(!ContactHasSubscription(presence.To))                          (1)
   {
      if(presence.Type == null)
         session.DirectedPresenceJIDs.Add(presence.To)
      else
         session.DirectedPresenceJIDs.Remove(presence.To)
   }

   presence.From = session.FullJID;
   SessionManager.SendStanzaToBareJID(presence, presence.To);
}
```

Listagem 6.13: Método ***DirectedPresence()*** de **PresenceProcessor**

```
{
   //(1)
   if(ContactExistsInRoster(session.Username, presence.To))
      UpdateContact(session.Username, presence.To);
   else
      InsertContact(session.Username, presence.To);

   //(2)
   if(ContactExistsInRoster(presence.To, session.Username))
      UpdateContact(presence.To, session.Username);
   else
      InsertContact(presence.To, session.Username);

   RosterPush(session.Username, presence.To);           (3)

   SendStanzaToContact(presence, presence.To);
}
```

Listagem 6.14: Método ***Subscribe()*** de **PresenceProcessor**

6.3.4 Processador de *info/query*

O processador de *stanzas info/query* é implementado pela classe **IqProcessor**, cujo método ***Process()*** tem uma lógica um pouco diferente dos restantes, devido à natureza do próprio *stanza*.

O *stanza* de IQ tem por objectivo realizar pedidos ao servidor, que este pode ter ou não capacidade de processar. Essa capacidade é verificada através do nome do seu elemento filho e respectivo *namespace*.

A listagem 6.17 apresenta o excerto do método ***Process()*** de **IqProcessor**, em que se verifica se o elemento filho é conhecido.

Tal como foi definida uma classe abstracta para unificar a assinatura dos métodos de processamento dos diferentes *stanzas*, foi também definido um *delegate* para unificar a assinatura para processamento dos diversos filhos do *stanza* IQ. Esse *delegate* chama-se **ProcessInner** e está apresentado na listagem 6.18.

Na implementação deste servidor, foi apenas considerado um elemento filho para o *stanza* IQ, usado na gestão do *roster*, o elemento <query/>, do *namespace* 'jabber:iq:roster'.

```
{
   UpdateContact(session.Username, presence.To);                 (1)
   UpdateContact(presence.To, session.Username);                 (2)

   RosterPush(presence.To, session.Username);
   RosterPush(session.Username, presence.To);
   SendStanzaToContact(presence, presence.To);
   SendPresences(session, presence.To);                 (3)
}
```

Listagem 6.15: Método ***Subscribed()*** de **PresenceProcessor**

```
{
   UpdateContact(session.Username, presence.To);                 (1)
   UpdateContact(presence.To, session.Username);                 (2)

   RosterPush(presence.To, session.Username);           (3)
}
```

Listagem 6.16: Método ***Subscribed()*** de **PresenceProcessor**

Processador de *queries*

A classe definida para processar o elemento <query/> chama-se **QueryProcessor** e contém um único método, que respeita o *delegate* **ProcessInner**, chamado ***ProcessQuery()***. Este método apenas verifica se o *namespace* do elemento <query/> é aquele que consegue processar, ou seja, 'jabber:iq:roster'. Se realmente for esse o *namespace*, então sabe-se que é o elemento <query/> para realizar gestão do *roster*. A gestão do *roster* não é directamente tratada por esta classe, mas sim por uma outra desenvolvida para o efeito, chamada **RosterManager**. Desta forma, o processamento do *stanza* é nela delegado, através do método ***ProcessQueryRoster()***, que também respeita o *delegate* **ProcessInner**.

Gestor de *roster*

A classe **RosterManager** tem um método principal, chamado ***ProcessQueryRoster()***, mencionado atrás. Este método verifica o tipo do *stanza* IQ e inicia a operação respectiva. Um *stanza* IQ, como mencionado em D.3, pode ser um dos tipos *'get'*, *'set'*, *'result'* ou *'error'* (que não tem qualquer significado para gestão de *roster*). As funcionalidades aqui indicadas estão descritas em pormenor, a nível do protocolo, em 2.4.3.

Um *stanza 'get'* serve para obter o *roster* do utilizador, operação normalmente realizada após a negociação da ligação. Esta operação é implementada pelo método ***GetRoster()***, que é apresentado, em pseudo-código, na listagem 6.19.

Um *stanza 'set'* pode servir para adicionar, actualizar ou remover um contacto do *roster*. Todas estas operação são implementadas pelo método ***SetRoster()***. Estas operações são, respectivamente, apresentadas nas listagens 6.20, 6.21 e 6.22, em pseudo-código.

No ponto (1) das listagens 6.20, 6.21 e 6.22 são feitas as respectivas alterações ao *roster* do utilizador.

```
public override void Process(Session sess, StanzaElement stanza)
{
   Iq iq = stanza as Iq;

   if (iq == null)
      throw new Exception("Wrong␣stanza␣type");

   SessionManager instance = SessionManager.GetInstance();

   //...
   SimpleElement se = iq.Children.First.Value;

   if (innerProcessors.ContainsKey(se.GetType()))
   {
      ProcessInner pi = innerProcessors[se.GetType()];
      pi(sess, iq);
   }
   else
   {
      Error error = new Error(Error.ErrorType.Cancel);
      error.Children.AddLast(new ServiceUnavailable());
      iq.Children.AddLast(error);
      iq.Type = Iq.ERROR;

      SessionManager.GetInstance().ContinueListening(sess);
      SessionManager.GetInstance().SendStanzaToSession(iq, sess);
   }
}
```

Listagem 6.17: Excerto do método ***Process()*** de **IqProcessor**

```
private delegate void ProcessInner(Session sess, Iq stanza);
```

Listagem 6.18: Delegate **ProcessInner**

No ponto (2) da listagem 6.22 são realizadas as alterações necessários ao *roster* do contacto que é removido, que consiste em remover a subscrição da presença do utilizador actual. Após remover a subscrição de presença, é necessário indicar que o utilizador ficou indisponível, enviando *stanzas* de indisponibilidade ao contacto, o que é realizado no ponto (3).

Notas:

- Todas as listagens aqui apresentadas, em pseudo-código, mostram apenas alguns pormenores de implementação que são considerados relevantes, devido à sua longa extensão. Para uma completa compreensão sobre o método em si, deve ser verificado o código.

- Estes processadores foram desenvolvidos com a premissa de que o servidor foi desenvolvido para ser único e não para integrar uma rede, tal como indica a especificação XMPP[11].

```
{
   contacts = GetContacts(session);
   queryRoster = new Query('jabber:iq:roster');
   foreach(c in contacts)
   {
      queryRoster.Add(new Item(c));
   }
   iq = new Iq();
   iq.Add(queryRoster);

   SessionManager.ContinueListening(session);
   SessionManager.SendStanzaToSession(iq, session);
}
```

Listagem 6.19: Método ***GetRoster()*** de **RosterManager**

```
{
   itemRoster = queryRoster.Children.First;

   AddContact(session, ir);                    (1)

   RosterPush(session.Username, ir.JID);
   SessionManager.ContinueListening(session);
   SessionManager.SendStanzaToSession(iqResult, session);
}
```

Listagem 6.20: Método ***SetRoster()*** de **RosterManager** - adicionar contacto

6.4 Ciclo de vida de um pedido

De forma a concluir a apresentação do servidor XMPP, é apresentada a figura 6.4, que ilustra o ciclo de vida de um *stanza* de mensagem.

Na figura 6.4 são apresentados processos (com as setas circulares e letras) e transições (com círculos e números).

```
{
   itemRoster = queryRoster.Children.First;

   UpdateContact(session, ir);                  (1)

   RosterPush(session.Username, ir.JID);
   SessionManager.ContinueListening(session);
   SessionManager.SendStanzaToSession(iqResult, session);
}
```

Listagem 6.21: Método ***SetRoster()*** de **RosterManager** - actualizar contacto

```
{
   itemRoster = queryRoster.Children.First;

   RemoveContact(session, ir);                  (1)

   RosterPush(session.Username, ir.JID);

   UpdateContact(ir.JID, session.Username);             (2)
   RosterPush(ir.JID, session.Username);
   SendPresences(session, ir.JID);                      (3)

   SessionManager.ContinueListening(session);
   SessionManager.SendStanzaToSession(iqResult, session);
}
```

Listagem 6.22: Método ***SetRoster()*** de **RosterManager** - remover contacto

Em seguida são apresentados os processos que ocorrem no ciclo de vida do *stanza*.

A Recepção do *stanza* enviado em (1), de forma assíncrona, associando-o a um identificador de sessão;

B Obter o objecto que representa o *stanza* recebido e a sessão com o identificador recebido, correspondentes à informação enviada em (2);

C Processar o objecto *stanza* de mensagem, identificando o recurso para qual redireccionar o stanza;

D Transforma o objecto recebido em (4) numa *string* XML e obtém o identificador da sessão do recurso destinatário;

E Escrever a informação recebida em (5) de forma assíncrona para o *stream* XML correcto.

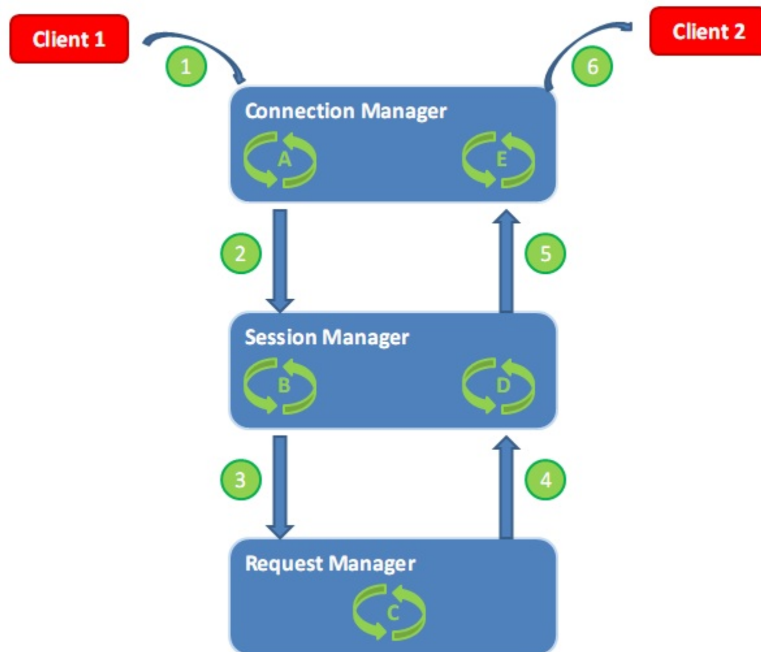


Figura 6.4: Ciclo de vida de um *stanza* de mensagem

Capítulo 7

Integração com serviços existentes

A conclusão da implementação do servidor de *instant messaging* atinge o objectivo mínimo do projecto, fornecendo uma base de trabalho para a integração com outros serviços.

A figura 4.1 ilustra a arquitectura do sistema, onde é possível verificar a existência de um componente chamado *IM Module Manager*. Este componente tem como objectivo fornecer um ponto de comunicação com os módulos de tradução dos serviços não XMPP. O componente ganha forma através da implementação da classe **ModuleManager**. É esta a classe que implementa o *Web Service* Central, na plataforma *Windows Communication Foundation* (WCF).

O caso de estudo utilizado para comprovar que os objectivos opcionais são realmente alcançados, permitindo a integração com pelo menos um serviço já existente, foi o serviço de *instant messaging* da *Yahoo* (apresentado em 3).

7.1 *IM Module Manager*

Na arquitectura apresentada (figura 4.1) o componente *IM Module Manager* encontra-se ao mesmo nível que o *Connection Manager*, pois tem funções semelhantes, embora sejam invocados de formas diferentes. A disposição dos blocos é encarada desta forma pois ambos os componentes são considerados um ponto de comunicação do servidor, comunicando directamente com o componente seguinte no processamento de um pedido (*Session Manager*).

Este componente é implementado pela classe **ModuleManager**, que tem duas facetas. Por um lado implementa o *Web Service* Central, sendo invocado pelos módulos de tradução, e por outro é invocado pelo *Session Manager* para enviar a informação a esses mesmos módulos.

Na vertente de *Web Service* Central, esta classe recebe os pedidos realizados pelos módulos e transforma-os em objectos que representem *stanzas* correspondentes. Estes objectos são imediatamente delegados ao *Session Manager*, que irá iniciar o seu processamento.

Ao ser invocado pelo *Session Manager*, o componente deve realizar o processo inverso. O que consiste em transformar a informação presente no objecto que representa o *stanza*, na

informação a enviar ao módulo de tradução a que essa informação é destinada, através do *Web Service* respectivo.

7.1.1 Implementação do *Web Service* Central

A implementação do *Web Service* Central começa pela definição da interface que este irá implementar e publicar, e pelos tipos usados pela implementação. Ao contrário dos *Web Services* dos módulos, não existe necessidade de utilizar a técnica *Contract-First* para definição do *Web Service* Central. Isto pois o *Web Service* Central é uma parte integrante do servidor, com uma definição completa. Os *Web Services* dos módulos serão implementados *à posteriori*, daí ser necessário definir em primeira mão, qual o contrato que devem respeitar. As listagens 7.1 e 7.2 apresentam, respectivamente, a interface **IModuleManager**, que define o contrato do *Web Service*, e o enumerado **PresenceState**, que declara os possíveis estados de presença.

```
[ServiceContract(SessionMode=SessionMode.NotAllowed)]
public interface IModuleManager
{
   [OperationContract(IsOneWay = true)]
   void GetContactList(string username);

   [OperationContract(IsOneWay = true)]
   void SendPresence(string sender, string target, PresenceState show,
      string status, bool newSession);

   [OperationContract(IsOneWay = true)]
   void AddContact(string target, string sender);

   [OperationContract(IsOneWay = true)]
   void RemoveContact(string target, string sender);

   [OperationContract(IsOneWay = true)]
   void AnswerSubscriptionRequest(string target, string sender,
      bool answer);

   [OperationContract(IsOneWay = true)]
   void SendMessage(string target, string sender, string message);
}
```

Listagem 7.1: Interface **IModuleManager**

```
public enum PresenceState
{
   Online,
   Away,
   Busy,
   ExtendedAway,
   Unavailable
}
```

Listagem 7.2: Enumerado **PresenceState**

Estas operações devem ser todas *One Way*, tal como foi anteriormente mencionado, que

em WCF é realizado assinalando o campo ***IsOneWay*** a *true*, através do atributo **OperationContract**.

A classe **ModuleManager** é a responsável pela implementação do *Web Service* Central, o que significa que implementa a interface **IModuleManager**. A declaração desta classe é apresentada na listagem 7.3.

```
[ServiceBehavior(
    InstanceContextMode=InstanceContextMode.Single,
    ConcurrencyMode=ConcurrencyMode.Multiple
)]
public class ModuleManager : IModuleManager, IDisposable
{...}
```

Listagem 7.3: Declaração da classe **ModuleManager**

Na listagem 7.3 é possível verificar que a instância do serviço é uma instância *singleton*[1], através do campo ***InstanceContextMode*** do atributo **ServiceBehavior**. Esta opção tem duas razões por base. A primeira razão vem de encontro à natureza dos sistemas de *instant messaging*, cuja comunicação deve acontecer em tempo quase-real. Desta forma a instância encontra-se sempre disponível para atender pedidos, não sendo necessário construir uma instância a cada pedido, o que torna o processo mais rápido. A segunda razão surge da solução proposta para integração de vários serviços, com a utilização de uma conta fantasma. Estas contas devem-se encontrar disponíveis durante o tempo de vida da instância do serviço. Assim, após a criação da instância, será invocado o método ***Init()***, que invoca sobre todos os *Web Services* dos módulos a operação ***LogIn***. De forma análoga, no momento de terminação do servidor, será invocado o método ***Dispose()*** (implementação da interface **IDisposable**), que invoca sobre todos os *Web Services* dos módulos a operação ***LogOut***.

7.1.2 Transformações entre operações e objectos *stanza*

As transformações referidas são delegadas em classes desenvolvidas para o efeito, cujos nomes são **PresenceGenerator**, **MessageGenerator** e **IqGenerator**, que representam, respectivamente, os componentes *Presence Translator*, *Message Translator* e *Info/Query Translator*, apresentados na figura 4.1. A tabela 7.1 apresenta as transformações sobre as operações do *Web Service* Central para objectos *stanza*, e qual a classe responsável por essas transformações.

As transformações de objectos *stanzas* em chamadas a operações dos *Web Services* dos módulos estão apresentadas na tabela 7.2.

É possível verificar pela tabela 7.2 que nem todas as transformações têm uma classe responsável. Isto acontece pois algumas transformações não precisam de "tradução", originando uma invocação directa de uma determinada operação. Da mesma forma nota-se também que nem todos os *stanzas* têm uma operação que lhes corresponde. Os *stanzas* de <presence/> com os valores *'unsubscribed'*, *'subscribed'* e *'unsubscribe'* para o atributo

[1]Padrão de desenho *Singleton*

Operação	Objecto	Responsável
GetContactList	Instância de **Iq** com **QueryRoster** sem filhos	**IqGenerator**
SendPresence	Instância de **Presence** com respectivos filhos	**PresenceGenerator**
AddContact	Instância de **Iq** com **QueryRoster** com o respectivo **Item**/Instância de **Presence** com ***type***=*'subscribe'*	**IqGenerator/ PresenceGenerator**
RemoveContact	Instância de **Iq** com **QueryRoster** com respectivo **Item** com ***subscription***=*'remove'*	**IqGenerator**
AnswerSubscriptionRequest	Instância de **Presence** com ***type*** respectivo à resposta	**PresenceGenerator**
SendMessage	Instância de **Message** com respectivos filhos	**MessageGenerator**

Tabela 7.1: Transformações das operações de *Web Service* Central para objectos *stanza*

Objecto	Operação	Responsável
Objecto **Message**	***SendMessage***	————
Objecto **Presence** sem ***type***, com **Status** e **Show** (elementos filhos)	***SendPresence***	**PresenceGenerator**
Objecto **Presence** com ***type***=*'subscribe'*	***AddContact***	**PresenceGenerator**
Objecto **Presence** com ***type***=*'unavailable'*	***SendPresence***	————
Objecto **Presence** com ***type***=*'unsubscribed'*	————	————
Objecto **Presence** com ***type***=*'subscribed'*	————	————
Objecto **Presence** com ***type***=*'unsubscribe'*	————	————
Objecto **Presence** com ***type***=*'probe'*	***GetPresence***	————
Objecto **Iq** com **QueryRoster** com os elementos **Item**, respectivos à lista de contactos	***SendContactList***	**IqGenerator**

Tabela 7.2: Transformações de objectos *stanzas* em operações dos *Web Services* dos módulos

type, servem apenas para informar o cliente do sucedido, que por sua vez não gera qualquer informação para o cliente. Com este factor em consideração foi optado por não enviar essa informação aos módulos, pois estes *stanzas* apenas têm significado para clientes XMPP.

7.2 Caso de estudo: *Yahoo*

Devido à limitação de tempo, tal como já foi referido, apenas foi desenvolvido um módulo de tradução. Este módulo realiza comunicação com o serviço de *instant messaging* da *Yahoo* (apresentado em 3).

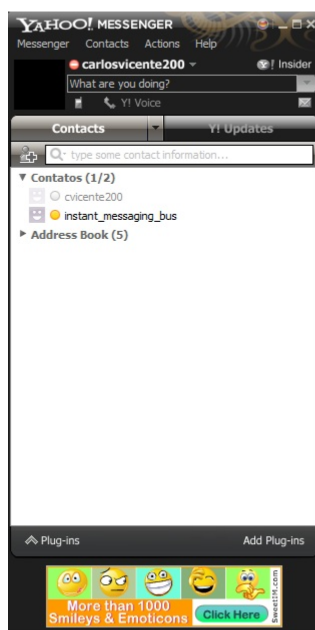
Com o objectivo de comprovar a interoperabilidade entre linguagens de programação, fornecida pela utilização de *Web Services* para realizar comunicação, o desenvolvimento deste módulo é realizado na plataforma *Java*. Basta que a linguagem utilizada suporte a versão 1.1 de WSDL.

Numa rede XMPP podem existir vários servidores, onde cada um destes gere as sessões

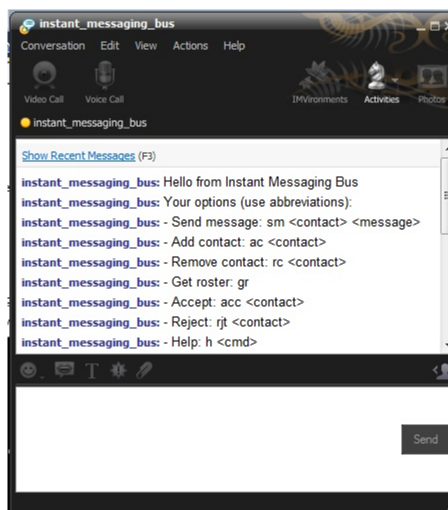
com os clientes a eles ligados, incluindo manter a mais recente informação de presença. Da mesma forma também os módulos devem gerir as sessões com os utilizadores dos respectivos sistemas.

De forma a facilitar o acesso ao serviço de *instant messaging* da *Yahoo*, foram criados alguns tipos auxiliares, tornando transparente o acesso através de pedidos HTTP. Estes tipos permitem realizar as várias operações de *instant messaging*, de forma autenticada.

O protocolo de *instant messaging* e *presence* da *Yahoo* é um protocolo proprietário e fechado, logo não é possível realizar tradução do protocolo XMPP para o protocolo da *Yahoo*. Assim a comunicação com um utilizador *Yahoo* será realizada através do serviço de IM, por via de mensagens, utilizando a conta fantasma *Yahoo* como intermediária. Para que esta comunicação seja possível é necessário que o utilizador *Yahoo* se encontre na lista de contactos da conta fantasma, ou seja, a conta fantasma deve possuir uma subscrição da informação de presença do utilizador (o utilizador pode-se registar através da aplicação *Web* para este efeito). Com este tipo de comunicação, não é necessário realizar qualquer tipo de intervenção ao nível da aplicação cliente *Yahoo*. Assim será utilizada uma janela de conversação da aplicação cliente para realizar interacção entre o módulo de tradução e o utilizador *Yahoo*. Essa janela de conversação é respectiva à conta fantasma *Yahoo*. A figura 7.1 apresenta a lista de contactos na aplicação cliente Yahoo (em 7.1a) e a janela de conversação com a conta fantasma (em 7.1b).



(a) Lista de contactos



(b) Janela de conversação com a conta fantasma

Figura 7.1: Aplicação cliente Yahoo

Este módulo deve então disponibilizar um *Web Service* que respeite o contrato definido no ficheiro WSDL, por forma a receber a informação do *bus* de *instant messaging*.

7.2.1 Gestão de sessões dos utilizadores

Este módulo deve gerir as sessões com os utilizadores *Yahoo*, armazenando as mesmas no dicionário ***sessions***, que possui como chave o identificador do utilizador e como valor uma instância de **YahooSession**.

O tipo **YahooSession**, apresentado na listagem 7.4, serve apenas para armazenar a informação de presença mais recente dos utilizadores ligados no momento.

```
public class YahooSession
{
    public String username;
    public PresenceState state;
    public String status;
}
```

Listagem 7.4: Classe **YahooSession**

A figura 7.2 ilustra o contentor *sessions*.

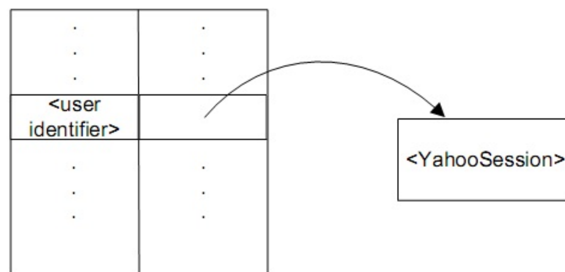


Figura 7.2: Ilustração do contentor *sessions*

7.2.2 Comunicação com utilizador *Yahoo*

O módulo de tradução comunica com o utilizador *Yahoo* através da troca de mensagens com a conta fantasma, tal como já foi referido. Para isto será utilizada a janela de conversação da aplicação cliente *Yahoo*, com a conta fantasma, como apresentado na figura 7.1.

Não sendo realizada qualquer intervenção, na realização deste trabalho, ao nível da aplicação cliente *Yahoo*, não é possível utilizar a interface gráfica disponibilizada por essa aplicação para realizar comunicação com o módulo de tradução, como para comunicar com outros utilizadores *Yahoo*. Isto levou ao desenvolvimento de uma sintaxe de comunicação, utilizando comandos. Estes comandos são realizados, enviando mensagens à conta fantasma, através da respectiva janela de conversação. Os comandos serão então processados pelo módulo de tradução e convertidos em operações a realizar no *bus* de *instant messaging* (através do *Web Service* Central).

Com o objectivo de facilitar a gestão e processamento de comandos, foram criadas as classes **CommandList** e **Command**. A classe **CommandList** representa uma lista de comandos, que podem ser enviados ao módulo de tradição (via mensagens enviadas à conta

fantasma). Cada instância é constituída por um contentor com instâncias da classe abstracta **Command**, que por sua vez representa um comando. Esta classe define um método abstracto chamado ***process***, que deve ser redefinido por cada classe que dela derivar para implementar uma nova operação.

No âmbito deste módulo foram disponibilizadas os seguintes comandos:

- Enviar uma mensagem a um contacto;
- Adicionar um utilizador à lista de contactos;
- Remover um utilizador da lista de contactos;
- Obter a lista de contactos;
- Aceitar um pedido de subscrição da informação presença;
- Rejeitar um pedido de subscrição da informação presença;
- Pedir a descrição detalhada de um comando específico.

A figura 7.1b apresenta a janela de conversação de um utilizador *Yahoo* com a conta fantasma *Yahoo*, apresentando a lista de comandos.

Exceptuando o último comando, cada um deles, sendo correctamente utilizado, irá desencadear a invocação de uma operação, com a respectiva semântica, no *Web Service* Central.

Ao analisar as operações do *Web Service* Central, verifica-se que não existe comando para alteração da informação de presença. Isto acontece pois o utilizador pode alterar a sua informação de presença na aplicação cliente *Yahoo* e a conta fantasma será notificada desse mesmo acontecimento. O que faz com que a operação ***SendPresence*** do *Web Service* Central apenas seja invocada por via de notificações (referido mais à frente, em 7.2.3).

As figuras 7.3 e 7.4 apresentam a janela de conversão com a conta fantasma *Yahoo*. Na figura 7.3 estão realçadas as mensagens que indicam a lista de contactos do utilizador e notificações da alteração de presença dos seus contactos no *bus* de *instant messaging*. Na figura 7.4 são apresentadas as mensagens utilizadas para realizar comunicação com clientes ligados ao *bus*.

Não é possível utilizar directamente o protocolo *Yahoo* para *instant messaging* e *presence*, mas é possível criar *stanzas* XML de forma a que os clientes XMPP visualizem a informação relativa a utilizadores *Yahoo*. De forma a contrastar com as figuras 7.3 e 7.4, são apresentadas as figuras 7.5 e 7.6. A figura 7.5 apresenta uma aplicação cliente XMPP, mostrando a lista de contactos, onde o contacto *Yahoo* é visto como um contacto XMPP. A figura 7.6 apresenta a janela de conversação com um utilizador *Yahoo*, provando que os *stanzas* são criados de forma a apresentar a informação enviada do módulo *Yahoo*, como se fossem enviados por um cliente XMPP.

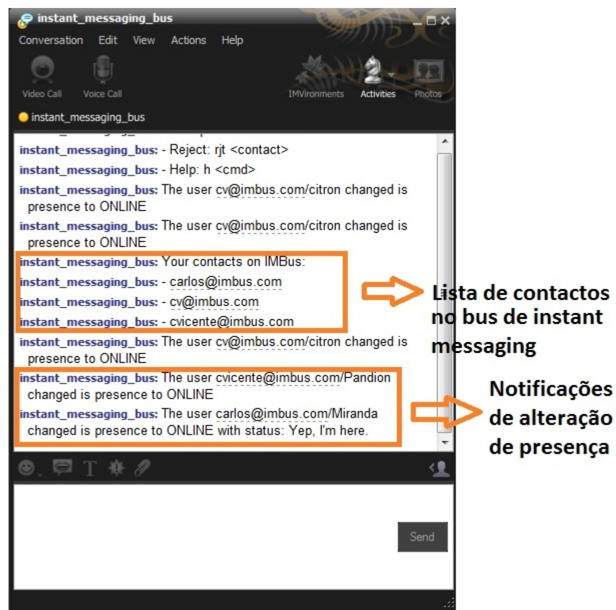


Figura 7.3: Janela de conversação com a conta fantasma - lista de contactos e notificações de presença

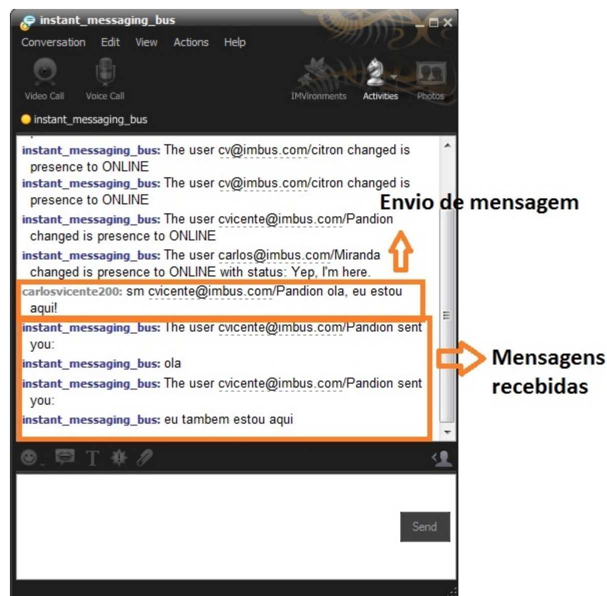


Figura 7.4: Envio e recepção de mensagens na perspectiva de um utilizador *Yahoo*

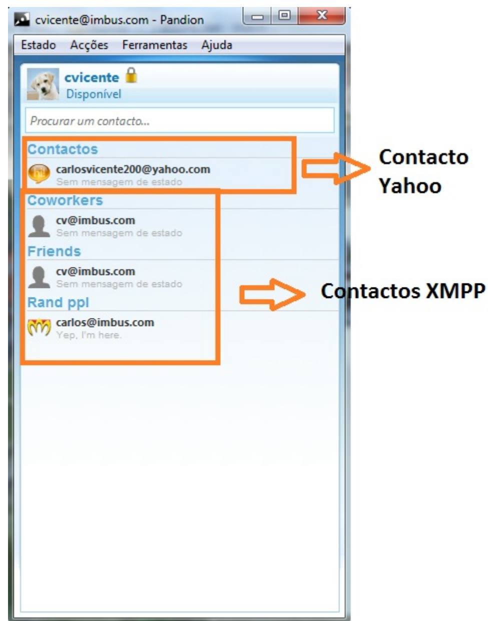


Figura 7.5: *Roster* de um utilizador XMPP

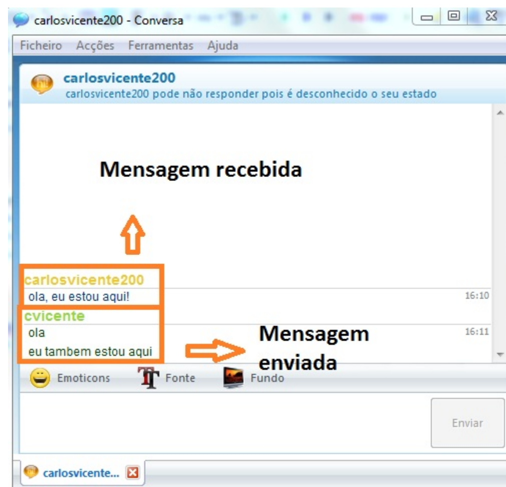


Figura 7.6: Envio e recepção de mensagens na perspectiva de um utilizador XMPP

7.2.3 Acesso ao serviço de IM da *Yahoo*

O acesso ao serviço de *instant messaging* da *Yahoo* é realizado através de alguns tipos auxiliares, tal como referido anteriormente. Entre estes, encontra-se o tipo **YahooMessengerIM**, que disponibiliza método de instância para realizar pedidos HTTP ao serviço de *instant messaging* da *Yahoo*. Esta classe disponibiliza métodos para realizar as funcionalidade apresentadas em 3, apresentados de seguida.

- ***Authenticate***: realiza autenticação *OAuth* de forma a obter os *tokens* necessários para realizar acessos autenticados à API de IM;
- ***RefreshSession***: renovar os *tokens* de acesso autenticado à API de IM;
- ***InitSession***: cria uma nova sessão de *instant messaging*, enviando informação de presença inicial (estado de presença 0);
- ***Logout***: termina a sessão de *instant messaging*;
- ***GetContactList***: obtém a lista de contactos;
- ***AddContact***: adiciona um utilizador à lista de contactos;
- ***AcceptContact***: aceita o pedido de subscrição da sua informação de presença;
- ***GetNotifications***: obtém as notificações disponíveis, através da forma *Comet-Style push* (referida em 3.2.7), no formato JSON;
- ***SendMessage***: envia uma mensagem a um utilizador.

À excepção do método ***Authenticate***, todos os outros são realizados de forma autenticada (credenciais de autenticação no *header* HTTP *Authorization*). Os métodos que executam operações de *instant messaging* são realizados em nome do utilizador autenticado.

A utilização típica desta classe passa por criar uma instância e começar por invocar os métodos ***Authenticate*** e ***InitSession***. Após autenticação e iniciação de sessão, é possível realizar outras operações de *instant messaging*, de forma autenticada e no âmbito de uma sessão. O que implica que esta instância armazene as credenciais de autenticação. O armazenamento das credenciais é realizado através de uma instância do tipo **Credentials**, que é apresentado na listagem 7.5. Por fim deve ser invocado o método ***Logout***.

Recepção de notificações

A obtenção de notificações é realizada da forma *Comet-Style push*, tal como mencionado anteriormente, o que implica que após a recepção de um conjunto de notificações, seja imediatamente realizado um novo pedido. Desta forma foi criado um receptor de notificações, cujo objectivo é obter novas notificações, invocando o método ***GetNotifications*** de **YahooMessengerIM**. Este receptor é implementado pela classe **NotificationsReceiver**, que implementa a

```
public class Credentials
{
   public String Token;
   public String Token_Secret;
   public String Session_Handle;
   public long Expires_In;
   public long Authorization_Expires_In;

   public static Credentials GetCredentials(String response) {...}

   public static String GetAuthorizationHeader(String part, String ck,
      String cs, Credentials cre) {...}
}
```

Listagem 7.5: Classe **Credentials**

interface **Runnable**. O receptor é então atribuído a uma *thread*, de forma a que esta execute o seu método ***run*** (definido em **Runnable**).

O método ***run*** de **NotificationsReceiver** apenas obtém as notificações, não as processa. O processamento das notificações fica a cargo da classe **NotificationProcessor**. Assim que um conjunto de notificações é obtido, segundo as regras na documentação[16], deve ser iniciado imediatamente um novo pedido para obter novas notificações. Isto implica que o processamento das notificações seja realizado numa nova *thread*, daí é utilizada uma instância de **ExecutorService**, que representa um *pool* de *threads* com cache. Desta forma, ao receber notificações, é criada uma instância de **NotificationProcessor**, que por sua vez é entregue ao **ExecutorService** para ser posteriormente processada. A figura 7.7 ilustra este processo.

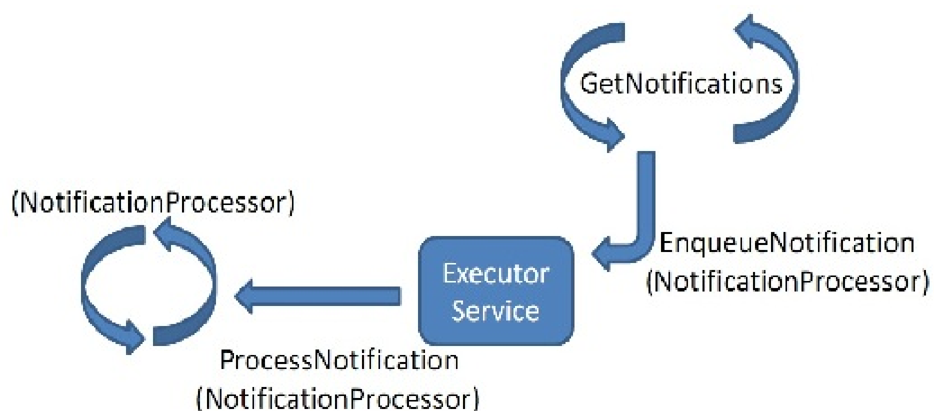


Figura 7.7: Processo de recepção e processamento de notificações

Em seguida são apresentados os tipos de notificações (apresentados em 3.2.7) e respectivos processamentos executados pela instância de **NotificationProcessor**.

- ***buddyInfo***:

  1 Cria uma sessão no módulo (armazena informação de presença);

  2 Envia o menu de opções a realizar sobre o *bus* de *instant messaging*;

3 Requisita a lista de contactos do utilizador no *bus* de *instant messaging*;

4 Envia a sua informação de presença para o *bus* de *instant messaging*.

- ***buddyStatus***:

  1 Actualiza a informação de presença na sessão com o utilizador;

  2 Envia a informação de presença para o *bus* de *instant messaging*.

- ***logOff***:

  1 Envia a informação de indisponibilidade do utilizador para o *bus* de *instant messaging*;

  2 Apaga a sessão com o módulo.

- ***message***:

  1 Verifica se o comando utilizado existe;

  2 Se existir, invoca-o;

  3 Se não existir, volta a enviar o menu de opções a realizar sobre o *bus* de *instant messaging*.

- ***buddyAuthorize***:

  1 Verifica se é um pedido de subscrição ou uma resposta a um pedido de subscrição;

  2 Se for um pedido de subscrição, irá aceitar essa subscrição.

Expiração da sessão de autenticação

A resposta que o servidor devolve ao criar uma sessão contém, além dos *tokens* para autenticação, um tempo de validade. Desta forma, é necessário agendar uma renovação da sessão (*tokens* de autenticação). Esta operação é realizada através do tipo **ExpirationManager**, que implementa a interface **TimerTask**, de forma a que seja agendada uma tarefa quando a validade da sessão terminar.

7.2.4 *Web Service* gerado pelo WSDL

A geração do código (com a ferramenta *wsimport* de *Java*) que suporta um *Web Service* com este contrato, origina a interface apresentada na listagem 7.6 e um esqueleto de uma classe que a implementa, a classe **YahooIMB**.

As operações apresentadas na listagem 7.6 são invocadas pelo *IM Module Manager* e devem realizar as respectivas operações. A forma de execução do próprio módulo influência a forma como essas operações são propagadas para o utilizador destino (quando isso se justifica). O módulo apresenta-se perante o utilizador como uma conta fantasma, logo o cliente irá realizar toda a comunicação com o módulo através da janela de conversação da conta fantasma que o representa, tal como foi referido atrás.

```java
@WebService(name="IModule", targetNamespace="http://imbus.com/module/")
public interface IModule {

    @WebMethod
    @Oneway
    public void logIn();

    @WebMethod
    @Oneway
    public void logOut();

    @WebMethod
    @Oneway
    public void registerUser(String username);

    @WebMethod
    @Oneway
    public void addContact(String target, String sender);

    @WebMethod
    @Oneway
    public void sendPresence(String target, String sender,
       String status, PresenceState show);

    @WebMethod
    @Oneway
    public void sendMessage(String target,String sender,String message);

    @WebMethod
    @Oneway
    public void sendContactList(String username, ArrayOfUser users);

    @WebMethod
    @Oneway
    public void getPresence(String target, String sender);

}
```

Listagem 7.6: Interface gerada através do ficheiro WSDL (em *Java*)

A implementação fornecida pela classe **YahooIMB** para as operações de *instant messaging* e *presence* consiste em enviar mensagens para o utilizador, através da janela de conversação com a conta fantasma *Yahoo*, com o conteúdo das respectivas operações.

Capítulo 8

Conclusão

Tendo atingido os principais objectivos, com o término do tempo de realização do trabalho, este é dado como concluído, mas existindo ainda aspectos a melhorar.

8.1 Resultados obtidos

A definição do protocolo XMPP tem como base a interoperabilidade com outros serviços de *instant messaging*, estando já definida no *core* do protocolo a utilização de *gateways* para outros servidores. Estes *gateways* realizam tradução para o protocolo com que comunicam, mas sempre de um ponto de vista de um utilizador XMPP, não sendo possível utilizá-lo por outra forma que não XMPP.

A extensão XEP-0100[13] demonstra esta limitação, tendo em conta que define a utilização do protocolo na interacção com um *gateway* para comunicação com outros protocolos de *instant messaging*, nomeadamente, aqueles que são fechados. A extensão define uma forma de um utilizador XMPP interagir com utilizadores de outro serviço, por meio de uma conta *proxy* nesse serviço, cujas credenciais devem ser fornecidas aquando do registo do utilizador no *gateway*. No entanto esta solução não apresenta qualquer possibilidade para que um utilizador de outro serviço comunique com utilizadores XMPP, sem que estes utilizem esse *gateway*.

A solução proposta por este trabalho vem colmatar esta limitação. A utilização dos módulos de conversão é semelhante à solução com *gateways*, diferenciando-se pela forma de acesso ao serviço externo. O facto da solução apresentada pela extensão se restringir a ter uma conta por cada utilizador XMPP, vem incapacitar a solução. Isto acontece porque a comunicação apenas é possível se o utilizador XMPP possuir uma conta noutro serviço. A implementação dos módulos de tradução deste trabalho é baseada no conceito de conta "*fantasma*", ou seja, uma conta que represente todo o *bus* de *instant messaging* e não apenas um utilizador. Desta forma um utilizador de outro serviço pode interagir com os utilizadores XMPP do *bus*, sem que estes possuam uma conta nesse serviço.

Esta solução apresenta também as suas limitações. No caso da extensão ao protocolo

XMPP, a comunicação fica enquadrada no próprio protocolo, sendo a informação apresentada através do cliente do serviço alvo. A solução aqui proposta não fica enquadrada no protocolo alvo, sendo necessário utilizar comandos, enviados através de mensagens para aceder às funcionalidades do *bus* de *instant messaging*.

Esta limitação talvez pudesse ser contornada, com o desenvolvimentos de *plug-ins* para os clientes dos serviços. Esta vertente apenas não foi explorada pois ora o suporte para os *plug-ins* tinha sido descontinuado pelo serviço, ora existia um processo demorado na aprovação e disponibilização do mesmo, que tornava completamente impossível realizar desenvolvimento e testes. Talvez esta possibilidade possa ser explorada mais tarde.

8.2 Comentários ao trabalho

Tal como foi mencionado várias vezes ao longo deste relatório, o objectivo do trabalho passava pela integração de vários serviços de instant messaging e não pela implementação completa e mais correcta do protocolo XMPP. O que levou a que alguns aspectos de implementação do protocolo tenham sido deixados de parte em prol do principal objectivo do trabalho.

O servidor XMPP desenvolvido apenas se encontra preparado para funcionar sozinho, enquanto que a especificação[11] do protocolo, define que um servidor XMPP deve ser preparado para funcionar numa rede de servidores.

Existe também uma funcionalidade definida na extensão para *instant messaging* e *presence*[12] do protocolo que foi negligenciada. Essa funcionalidade não se prende apenas ao protocolo, mas também aos sistemas de *instant messaging* em geral. Não foi implementada pois não foi considerada como essencial para provar o conceito do trabalho. A funcionalidade mencionada é a gestão de listas de bloqueio de utilizadores, referida em 1.2.

O protocolo define vários elementos de erro, para fornecer a máxima informação possível sobre uma determinada falha que tenha ocorrido. Em muitas situações de falha, a implementação simplesmente fecha a ligação, sendo essa a forma incorrecta de terminar um *stream* XML. A forma correcta consiste em enviar um *stanza* com a informação de erro descrita para que esta seja evitada futuramente, sendo que apenas após este envio o *stream* XML pode ser terminado.

A implementação da parte da gestão de módulos tem dois aspectos que não estarão o melhor possível.

A informação sobre a localização dos *Web Services* dos módulos é estática, encontrando-se no ficheiro de configuração. Se a referência para a informação não se encontrar no ficheiro de configuração aquando da iniciação da execução do servidor, o módulo não poderá ser utilizado. A forma mais correcta de estabelecer esta ligação seria de forma dinâmica, o que permitiria que fossem adicionados novos módulos de tradução com o servidor em execução. Isso poderia ser alcançado com uma nova operação no *Web Service* Central para registo de módulos.

A utilização dos *Web Services* dos módulos fica muito dependente do componente *IM Module Manager*, sendo este a realizar a comunicação com os *Web Services*. Na arquitectura

proposta para o servidor, seria possível definir uma interface que seria implementada pelo *Connection Manager* e pelos *proxies* para os *Web Services*, que seria utilizada pelo *Session Manager*. Esta interface iria permitir ao *Session Manager* realizar uma separação mais fluida da informação.

8.3 Trabalho futuro

Em qualquer trabalho consegue-se sempre identificar alguns aspectos que podiam ser melhorados e/ou novas funcionalidades que possam ser implementadas. Este trabalho não foge à regra.

Além das melhorias apresentadas na secção anterior, podem ser consideradas como trabalho futuro as seguintes tarefas:

- Implementação completa do protocolo XMPP;
- Estudo extensivo das extensões ao protocolo XMPP e APIs disponibilizadas para acesso aos serviços existentes, de forma a melhorar os contratos dos *Web Services* para suportar um maior leque de funcionalidades relacionadas com *instant messaging*;
- Implementação de novos módulos de tradução, como por exemplo, para o serviço *Live Messenger*.

Bibliografia

Peter Saint-Andre, Kevin Smith and Remko Tronçon. *XMPP: The Definitive Guide*. O'Reilly, first edition, 2009

Iain Shigeoka. *Instant Messaging in Java - The Jabber Protocols*. Manning, first edition, 2002

John W. Rittinghouse and James F. Ransome. *Instant Messaging Security*. Elsevier Digital Pres, first edition, 2005

Peter Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core (internet draft)*.
`http://xmpp.org/internet-drafts/draft-ietf-xmpp-3920bis-04.html`

Peter Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence (internet draft)*.
`http://xmpp.org/internet-drafts/draft-ietf-xmpp-3921bis-03.html`

Referências

[1] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl.

[2] B. Campbell et al. The message session relay protocol (msrp). http://www.ietf.org/rfc/rfc4975.txt.

[3] J. Rosenberg et al. Session initiation protocol (sip) extension for instant messaging. http://www.ietf.org/rfc/rfc3428.txt.

[4] Tim Dierks et al. The tls protocol version 1.0. http://tools.ietf.org/rfc/rfc2246.txt.

[5] Paul Hoffman and Marc Blanchet. Preparation of internationalized strings ('stringprep'). http://tools.ietf.org/rfc/rfc3454.txt.

[6] M. Tim Jones. Meet the extensible messaging and presence protocol (xmpp). http://www.ibm.com/developerworks/xml/library/x-xmppintro/index.html?ca=drs-.

[7] Paul Leach and Chris Newman. Using digest authentication as a sasl mechanism. http://tools.ietf.org/rfc/rfc2831.txt.

[8] John G. Myers. Simple authentication and security layer (sasl). http://tools.ietf.org/rfc/rfc2222.txt.

[9] A. B. Roach. Session initiation protocol (sip)-specific event notification. http://www.ietf.org/rfc/rfc3265.txt.

[10] J. Rosenberg. A presence event package for the session initiation protocol (sip). http://www.ietf.org/rfc/rfc3856.txt.

[11] Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. http://www.ietf.org/rfc/rfc3920.txt.

[12] Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Instant messaging and presence. http://www.ietf.org/rfc/rfc3921.txt.

[13] Peter Saint-Andre and Dave Smith. Xep-0100: Gateway interaction. http://xmpp.org/extensions/xep-0100.html.

[14] OAuth Core Workgroup. Oauth core 1.0 revision a. http://tools.ietf.org/html/rfc5849.

[15] Yahoo! Sequence diagram for authentication flow. http://developer.yahoo.com/messenger/guide/ch05s02.html.

[16] Yahoo! Yahoo! messenger im sdk. http://developer.yahoo.com/messenger/.

[17] Kurt D. Zeilenga. The plain simple authentication and security layer (sasl) mechanism. http://tools.ietf.org/rfc/rfc4616.txt.

Apêndice A

Transformações a estados de subscrição

O envio de um *stanza* de *'subscribe'* origina as transformações de estado, no *roster* de quem envia, apresentadas na tabela A.1 e a recepção deste mesmo *stanza* origina as transformações, no *roster* de quem recebe, apresentadas na tabela A.2. Em qualquer um destes casos, se ainda não existir o respectivo item nos *rosters* em questão, este deve ser criado (com as consequências associadas, ou seja, "*roster push*" para todos os `recursos` disponíveis que pediram o *roster*).

Estado antigo	Estado novo
"none"	"none + pending out"

Tabela A.1: Transformações ao estado da subscrição de quem envia um *stanza 'subscribe'*

Estado antigo	Estado novo
"none"	"none + pending in"

Tabela A.2: Transformações ao estado da subscrição de quem recebe um *stanza 'subscribe'*

Nas tabelas A.3 e A.4 estão apresentadas as transformações do estado da subscrição para quando é enviado e recebido um *stanza 'subscribed'*, respectivamente.

Estado antigo	Estado novo
"none + pending in"	"from"
"none + pending in/out"	"from + pending out"
"to + pending in"	"both"

Tabela A.3: Transformações ao estado da subscrição de quem envia um *stanza 'subscribed'*

Estado antigo	Estado novo
”none + pending out”	”to”
”none + pending in/out”	”to + pending in”
”from + pending out”	”both”

Tabela A.4: Transformações ao estado da subscrição de quem recebe um *stanza ’subscribed’*

Ao enviar ou receber um *stanza ’unsubscribed’*, devem ser realizadas as transformações apresentadas nas tabelas A.5 e A.6, respectivamente.

Estado antigo	Estado novo
”none + pending in”	”none”
”none + pending in/out”	”none + pending out”
”to + pending in”	”to”
”from”	”none”
”from + pending out”	”none + pending out”
”both”	”to”

Tabela A.5: Transformações ao estado da subscrição de quem envia um *stanza ’unsubscribed’*

Estado antigo	Estado novo
”none + pending out”	”none”
”none + pending in/out”	”none + pending in”
”to”	”none”
”to + pending in”	”none + pending in”
”from + pending out”	”from”
”both”	”from”

Tabela A.6: Transformações ao estado da subscrição de quem recebe um *stanza ’unsubscribed’*

Um *stanza ’unsbuscribe’* origina as transformações no estado de subscrição de quem envia e de quem recebe, respectivamente, nas tabelas A.7 e A.8.

Estado antigo	Estado novo
”none + pending out”	”none”
”none + pending in/out”	”none + pending in”
”to”	”none”
”to + pending in”	”none + pending in”
”from + pending out”	”from”
”both”	”from”

Tabela A.7: Transformações ao estado da subscrição de quem envia um *stanza 'unsubscribe'*

Estado antigo	Estado novo
”none + pending in”	”none”
”none + pending in/out”	”none + pending out”
”to + pending in”	”to”
”from”	”none”
”from + pending out”	”none + pending out”
”both”	”to”

Tabela A.8: Transformações ao estado da subscrição de quem recebe um *stanza 'unsubscribe'*

Apêndice B

WSDL para *Web Services* dos módulos de tradução

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   xmlns:tns="http://imbus.com/module/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   name="IMBusModule" targetNamespace="http://imbus.com/module/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
   <types>
      <xs:schema
         xmlns:tns="http://imbus.com/module/"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         elementFormDefault="qualified"
         targetNamespace="http://imbus.com/module/">
         <complexType name="ArrayOfUser">
            <sequence>
               <element minOccurs="0" maxOccurs="unbounded"
                  name="User" type="tns:User" />
            </sequence>
         </complexType>
         <element name="ArrayOfUser" nillable="true"
            type="tns:ArrayOfUser" />
         <complexType name="User">
            <sequence>
               <element minOccurs="0" name="Name" nillable="true"
                  type="string" />
            </sequence>
```

```
</complexType>
<element name="User" nillable="true" type="tns:User" />
<simpleType name="PresenceState">
   <restriction base="string">
      <enumeration value="Online" />
      <enumeration value="Away" />
      <enumeration value="Busy" />
      <enumeration value="ExtendedAway" />
      <enumeration value="Unavailable" />
   </restriction>
</simpleType>
<element name="PresenceState" type="tns:PresenceState" />
<element name="LogIn">
   <complexType><sequence /></complexType>
</element>
<element name="LogOut">
   <complexType><sequence /></complexType>
</element>
<element name="AddContact">
   <complexType>
      <sequence>
         <element minOccurs="0" name="target"
            nillable="true" type="string" />
         <element minOccurs="0" name="sender"
            nillable="true" type="string" />
      </sequence>
   </complexType>
</element>
<element name="SendPresence">
   <complexType>
      <sequence>
         <element minOccurs="0" name="target"
            nillable="true" type="string" />
         <element minOccurs="0" name="sender"
            nillable="true" type="string" />
         <element minOccurs="0" name="status"
            nillable="true" type="string" />
         <element minOccurs="0" name="show"
            type="tns:PresenceState" />
      </sequence>
   </complexType>
</element>
<element name="SendMessage">
   <complexType>
      <sequence>
         <element minOccurs="0" name="target"
            nillable="true" type="string" />
         <element minOccurs="0" name="sender"
            nillable="true" type="string" />
         <element minOccurs="0" name="message"
            nillable="true" type="string" />
      </sequence>
```

```
          </complexType>
       </element>
       <element name="RegisterUser">
          <complexType>
             <sequence>
                <element minOccurs="0" name="username"
                   nillable="true" type="string" />
             </sequence>
          </complexType>
       </element>
       <element name="SendContactList">
          <complexType>
             <sequence>
                <element minOccurs="0" name="username"
                   nillable="true" type="string" />
                <element minOccurs="0" name="users"
                   nillable="true" type="tns:ArrayOfUser" />
             </sequence>
          </complexType>
       </element>
       <element name="GetPresence">
          <complexType>
             <sequence>
                <element minOccurs="0" name="target"
                   nillable="true" type="string" />
                <element minOccurs="0" name="sender"
                   nillable="true" type="string" />
             </sequence>
          </complexType>
       </element>
    </xs:schema>
</types>
<message name="LogIn_InputMessage">
   <part name="InputMessagePart" element="tns:LogIn" />
</message>
<message name="LogOut_InputMessage">
   <part name="InputMessagePart" element="tns:LogOut" />
</message>
<message name="AddContact_InputMessage">
   <part name="InputMessagePart" element="tns:AddContact" />
</message>
<message name="SendPresence_InputMessage">
   <part name="InputMessagePart" element="tns:SendPresence" />
</message>
<message name="SendMessage_InputMessage">
   <part name="InputMessagePart" element="tns:SendMessage" />
</message>
<message name="RegisterUser_InputMessage">
   <part name="InputMessagePart" element="tns:RegisterUser" />
</message>
<message name="SendContactList_InputMessage">
   <part name="InputMessagePart" element="tns:SendContactList" />
```

```
</message>
<message name="GetPresence_InputMessage">
   <part name="InputMessagePart" element="tns:GetPresence" />
</message>
<portType name="IModule">
   <operation name="LogIn">
      <input message="tns:LogIn_InputMessage" />
   </operation>
   <operation name="LogOut">
      <input message="tns:LogOut_InputMessage" />
   </operation>
   <operation name="AddContact">
      <input message="tns:AddContact_InputMessage" />
   </operation>
   <operation name="SendPresence">
      <input message="tns:SendPresence_InputMessage" />
   </operation>
   <operation name="SendMessage">
      <input message="tns:SendMessage_InputMessage" />
   </operation>
   <operation name="RegisterUser">
      <input message="tns:RegisterUser_InputMessage" />
   </operation>
   <operation name="SendContactList">
      <input message="tns:SendContactList_InputMessage" />
   </operation>
   <operation name="GetPresence">
      <input message="tns:GetPresence_InputMessage" />
   </operation>
</portType>
<binding name="IMBBinding" type="tns:IModule">
   <soap:binding
      transport="http://schemas.xmlsoap.org/soap/http" />
   <operation name="LogIn">
      <soap:operation
         soapAction="http://imbus.com/module/IMBModule/LogIn" />
         <input> <soap:body use="literal" /></input>
   </operation>
   <operation name="LogOut">
      <soap:operation
         soapAction="http://imbus.com/module/IMBModule/LogOut" />
         <input><soap:body use="literal" /></input>
   </operation>
   <operation name="AddContact">
      <soap:operation
         soapAction="http://imbus.com/module/IMBModule/AddContact" />
         <input><soap:body use="literal" /></input>
   </operation>
   <operation name="SendPresence">
      <soap:operation
        soapAction="http://imbus.com/module/IMBModule/SendPresence"/>
         <input><soap:body use="literal" /></input>
```

```
      </operation>
      <operation name="SendMessage">
         <soap:operation
           soapAction="http://imbus.com/module/IMBModule/SendMessage"/>
            <input><soap:body use="literal" /></input>
      </operation>
      <operation name="RegisterUser">
         <soap:operation
            soapAction="http://imbus.com/module/IMBModule/RegisterUser"/>
            <input><soap:body use="literal" /></input>
      </operation>
      <operation name="SendContactList">
         <soap:operation
         soapAction="http://imbus.com/module/IMBModule/SendContactList"/>
            <input><soap:body use="literal" /></input>
      </operation>
      <operation name="GetPresence">
         <soap:operation
            soapAction="http://imbus.com/module/IMBModule/GetPresence"/>
            <input><soap:body use="literal" /></input>
      </operation>
   </binding>
   <service name="IMBModule">
      <port name="IMBPort" binding="tns:IMBBinding">
         <soap:address location="" />
      </port>
   </service>
</definitions>
```

Listagem B.1: WSDL com contrato a implementar pelos módulos de tradução

Apêndice C

Hierarquia dos objectos de elementos XMPP

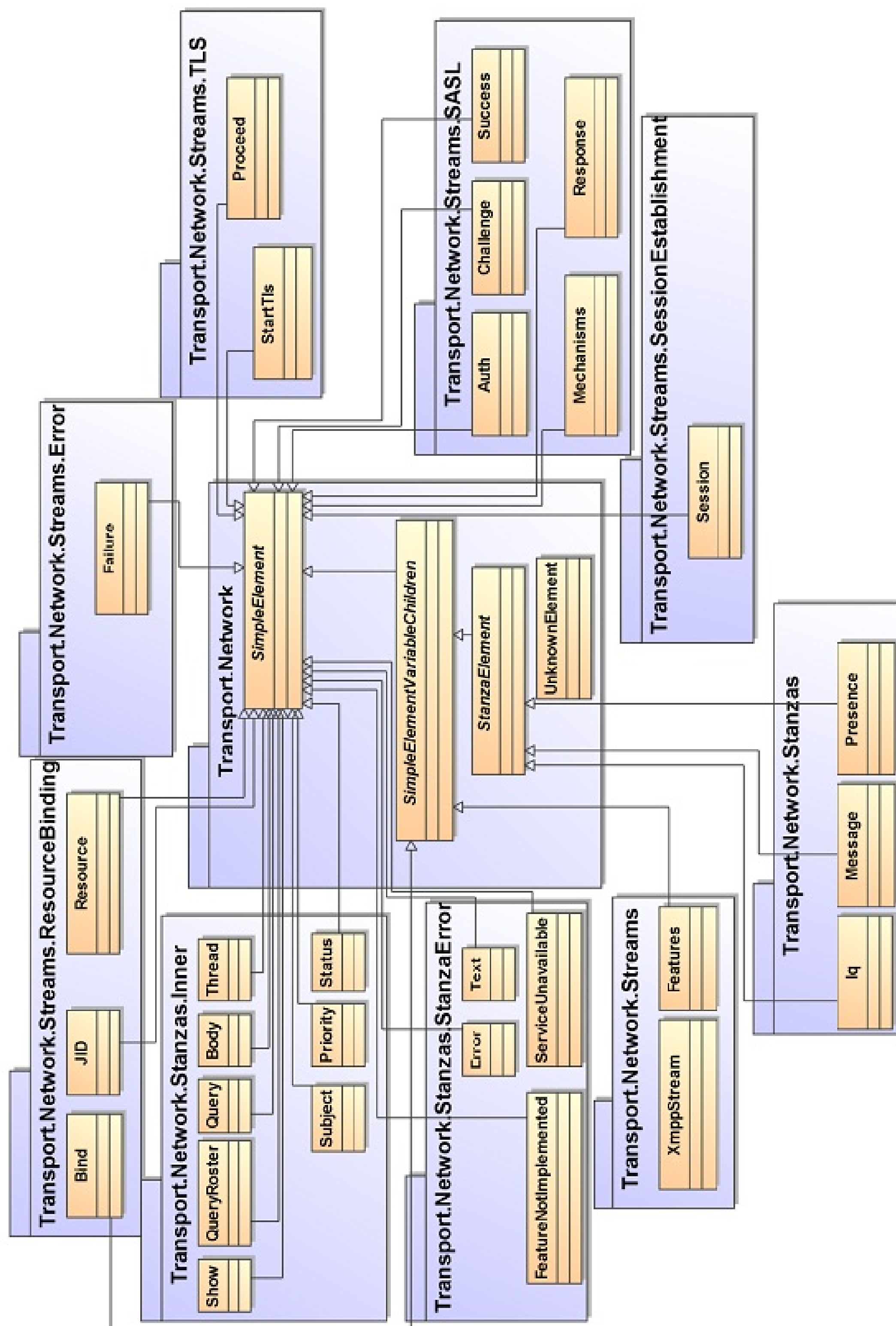


Figura C.1: Hierarquia de elementos XMPP

Apêndice D

XML *stanzas*

A unidade básica de comunicação em sessões XMPP é o XML *stanza* (tal como foi referido em 2.1.4). Estão definidos 3 tipos de *stanzas* nos *namespaces* 'jabber:client' e 'jabber:server': <message/>, <presence/> e <iq/>. Estes *stanzas* têm objectivos claramente distintos, mas têm uma construção bastante semelhante, tendo inclusivé os mesmos atributos, apresentados na tabela D.1.

to	Indica o destinatário do *stanza*
from	Indica o remetente do *stanza*
id	É o identificador do *stanza*, serve para seguir os *stanzas* (normalmente usado no *stanza* <iq/>)
type	Indica o tipo do *stanza*, que depende do contexto do próprio *stanza*
xml:lang	Indica a língua em que se encontra a informação escrita contida no *stanza* (se não existir é assumida a língua por omissão do *stream*)

Tabela D.1: Atributos transversais a todos os tipos de *stanzas*

Em seguida serão descritos em maior pormenor a semântica dos tipos de *stanza*.

D.1 *Message*

O *stanza* <message/>, tal como o nome sugere, serve para enviar informação de uma entidade para outra. Tipicamente não existe confirmação da recepção de mensagens, sendo o seu envio realizado por um mecanismo de "fire-and-forget". O cliente apenas receberá uma resposta em relação a uma mensagem, se ocorrer um erro no processamento da mesma.

Existem cinco tipos de mensagens diferentes, sendo o seu tipo indicado pelo atributo ***type***, cujos valores possíveis são apresentados na tabela D.2.

Este *stanza* tem ainda, tal como qualquer mensagem, um remetente e um destinatário, que são indicados pelos atributos ***from*** e ***to*** respectivamente. Podendo ainda ter um atributo ***id*** por motivos de monitorização. Os atributos ***from*** e ***to*** têm os endereços JID das respectivas

normal	Uma mensagem única que pode ou não ter resposta
chat	Mensagens trocadas numa ”sessão de conversação” em tempo real entre duas entidades
groupchat	Mensagens trocadas numa ”sessão de conversação” entre várias entidades (mais do que duas)
headline	Mensagem usada para enviar alertas e notificações, não sendo expectável qualquer tipo de resposta
error	Mensagem de erro, enviada pela entidade que detectar um problema com uma mensagem previamente enviada

Tabela D.2: Descrição dos tipos de *stanzas* <message/>

entidades, sendo o atributo ***from*** introduzido pelo servidor do remetente, por forma a evitar *spoofing*[1] de endereços. O atributo ***to*** pode conter tanto um *bare* JID como um *full* JID.

Os *namespaces* ’jabber:client’ e ’jabber:server’ definem que o elemento <message/> pode ter os filhos <subject/>, <body/> e <thread/> (apresentados de seguida).

- <subject/>: contém informação, legível por humanos, que especifica o tópico da mensagem. Podem existir várias instâncias de <subject/> dentro da mesma mensagem, desde que tenham o atributo ***xml:lang*** diferente;
- <body/>: tem os conteúdos da mensagem em si, em formato legível por humanos. Tal como <subject/>, podem existir várias instâncias, com ***xml:lang*** diferentes;
- <thread/>: contém informação não legível por humanos, para identificar uma *thread* de conversação. Este valor é gerado pelo servidor.

D.2 *Presence*

O *stanza* <presence/> serve para indicar, inquirir e subscrever a disponibilidade para comunicação de um determinado utilizador, e tal como o *stanza* <message/> não tem confirmação de entrega.

A informação de presença pode ser enviada de duas formas, direccionada ou por *broadcast*. Estas duas formas são diferenciadas pela inclusão ou não, respectivamente, do atributo ***to*** no *stanza* de presença. Ao enviar informação de presença de forma direccionada, esta apenas será entregue a um utilizador. No caso de broadcast, esta informação será entregue aos utilizadores que se encontrem no *roster* do utilizador em questão e possuam uma subscrição dessa mesma informação.

Se o atributo ***type*** não for especificado, a disponibilidade do utilizador será indicada pelos elementos filhos do *stanza* (descritos mais à frente). Caso for especificado, pode conter um dos valores que estão enumerados e descritos na tabela D.3.

[1]Falsificação de endereços, onde uma entidade se faz passar por outra.

probe	Serve para pedir a informação de presença de um determinado utilizador, ao servidor a que este se encontra ligado
unavailable	Indica que a entidade deixou de estar disponível para comunicação
error	Mensagem de erro, enviada quando é detectado um problema com um *stanza* previamente enviado
subscribe	O remetente realiza um pedido para subscrever a informação de presença do recipiente
subscribed	O remetente aceitou o pedido de subscrição do recipiente
unsubscribe	O remetente cancela a subscrição que tem da informação de presença do destinatário ou cancela um pedido de subscrição que tinha realizado
unsubscribed	O remetente cancela a subscrição que o destinatário possuía da sua informação de presença ou rejeita um pedido de subscrição

Tabela D.3: Descrição dos tipos de *stanzas* <presence/>

Os *namespaces* 'jabber:client' e 'jabber:server' definem que o elemento <presence/> pode ter os filhos <show/>, <status/> e <priority/> (estes elementos são descritos de seguida).

- <show/>: contém informação, não legível por humanos, que especifica uma disponibilidade para comunicação. Não pode existir mais do que uma instância, num elemento <presence/> e quando existir apenas pode ter um dos seguintes valores (caso não exista, assume-se que a entidade está *online* e disponível):
  - *away*: a entidade está temporariamente indisponível;
  - *chat*: a entidade está activamente interessada em conversar;
  - *dnd*: a entidade está ocupada (*dnd* = "*Do Not Distub*")
  - *xa*: a entidade encontra-se indisponível por um período longo.
- <status/>: fornece informação sobre a disponibilidade da entidade, em formato legível por humanos. É normalmente usada em conjunção com o elemento <show/> para dar informação mais detalhada sobre o estado de disponibilidade. Podem existir várias instâncias dentro do mesmo elemento <presence/>, desde que o atributo ***xml:lang*** tenha um valor diferente;
- <priority/>: indica o nível de prioridade do `recurso`. Não pode existir mais do que uma instância.

D.3 *Info/Query*

O *stanza* <iq/> fornece um mecanismo de pedido-resposta, em que uma entidade realiza um pedido e mais tarde irá receber a resposta a esse pedido. Neste caso para poder ligar uma resposta a um determinado pedido, é obrigatário a utilização do atributo ***id***.

get	pedido por informações
set	fornece informação, para ser afectada ou substituída
result	resultado a um *stanza* de *'get'* ou *'set'*
error	indica um aconteceu um erro com um *stanza* de *'get'* ou *'set'*

Tabela D.4: Descrição dos tipos de *stanzas* <iq/>

Os possíveis tipos (valores de ***type***) de stanzas <iq/> estão descritos na tabela D.4. Também este *stanza* está definido nos *namespaces* 'jabber:client' e 'jabber:server'.

Qualquer um dos *stanzas* pode ter outros elementos filhos, que estejam definidos noutros *namespaces* que não 'jabber:client' e 'jabber:server'. É aqui que se encontra a extensibilidade do XMPP, qualquer stanza pode conter qualquer elemento, desde que tenha o *namespace* respectivo explícito. Esta capacidade torna-se mais evidente no caso do *stanza* <iq/>.

Um exemplo desta extensibilidade é a gestão do *roster*, sendo realizada através do *stanza* <iq/>, com elementos filhos definidos no *namespace* 'jabber:iq:roster'. A obtenção do *roster* de um utilizador está exemplificada na listagem D.1.

```
<iq from='utilizador@servidor.pt/recurso' type='get' id='r_1'>
  <query xmlns='jabber:iq:roster'/>
</iq>
```

Listagem D.1: Exemplo de obtenção do *roster*

Independentemente do tipo de *stanza*, o servidor deve seguir as regras de processamento de *stanzas* enunciadas na secção 11.1 da especificação RFC3921[12].