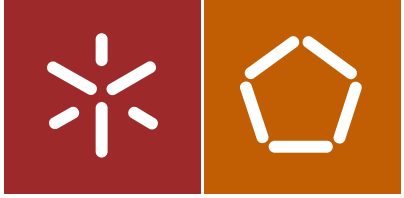
Universidade do Minho
Escola de Engenharia

Pedro Francisco Lourenço Machado

Ferramenta para Processamento de Dados da Utilização de Redes WiFi

outubro de 2014

Pedro Francisco Lourenço Machado Ferramenta para Processamento de Dados da Utilização de Redes WiFi
UMinho | 2014



Universidade do Minho
Escola de Engenharia

Pedro Francisco Lourenço Machado

Ferramenta para Processamento de Dados da Utilização de Redes WiFi

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho efetuado sob a orientação do
Professor Doutor Filipe Meneses
Professor Doutor Adriano Moreira

outubro de 2014

DECLARAÇÃO

Nome

Pedro Francisco Lourenço Machado

Endereço electrónico: a50663@alunos.uminho.pt Telefone: 925 100 721

Número do Bilhete de Identidade: 12876745

Titulo dissertação

Ferramenta para Processamento de Dados da Utilização de Redes Wifi

Orientadores:

Professor Doutor Adriano Moreira

Professor Doutor Filipe Meneses

Ano de conclusão: 2014

Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia de Comunicações

---

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Universidade do Minho, ____/____/_____

Assinatura: _____

Learning never exhausts the mind.

- Leonardo da Vinci

Agradecimentos

Com o final do meu mestrado é agora o momento de agradecer e prestar a minha homenagem a todos aqueles que direta ou indiretamente ajudaram e contribuíram para que este momento se torne realidade e não apenas um sonho.

Como não podia deixar de ser, começo por toda a minha família que me apoiou e acarinhou durante toda a minha vida. Em especial ao meu pai, à minha mãe e ao meu irmão que sempre me apoiaram e deram força nos momentos que mais necessitei.

A todos os docentes que me acompanharam na minha vida académica e com quem tive o prazer de privar conhecimento, em especial aos meus orientadores Professor Doutor Filipe Meneses e Professor Doutor Adriano Moreira e ao Karim Keramat Jahromi pelo vosso apoio, disponibilidade e sugestões durante a elaboração deste trabalho.

E por fim, a todos os meus amigos e companheiros de vida académica pelos bons momentos e alegrias que passamos juntos. Em especial aos meus amigos de longa data Abel Pinheiro, Freddy Gonçalves, Hugo Fernandes, Laurent Miranda, Luís Frutuoso e Miguel Correia e aos meus amigos na minha vida académica, em especial ao Ângelo Alves, Hugo Ferreira, Hugo Leite, Joana Silva, Luís Gonçalves, Luís Nascimento e Renato Martins.

A todos o meu muito obrigado, do fundo do coração.

Resumo

Hoje em dia com a massificação e popularidade das redes sem fios (WiFi) o numero de utilizadores a adotarem esta tecnologia é cada vez maior. Quando um utilizador se liga a uma rede WiFi, caso existam registos (*logs*) sobre a ligação do utilizador à rede, é possível determinar a localização e movimentos do utilizador no espaço, entre outros. Geralmente, quanto maior for o numero de utilizadores na rede maior é o volume de registos gerados, o que pode originar que uma rede seja capaz de gerar num curto espaço de tempo uma enorme quantidade de registos. No entanto, diferentes redes podem registar diferentes tipos de informação (*logs*) sobre os seus utilizadores.

Para se obter informação sobre um utilizador da rede é necessário uma ferramenta que analise e processe um conjunto de dados de forma a obter a informação desejada, como por exemplo, a localização e o movimento no espaço. Para tal, a ferramenta deve ser flexível o suficiente para que se adapte a diferentes tipos de dados a processar sem ser necessário redesenhar ou, em casos extremos, desenvolver uma nova ferramenta. É neste caso que surge esta dissertação, desenvolver uma aplicação suficientemente flexível e capaz de se adaptar a diferentes tipos de registos a processar com a menor quantidade de alterações e tempo despendido. Para tal foi adotado o conceito de módulos de processamento, onde cada módulo é responsável por um, ou mais, tipos de dados a processar e sempre que seja necessário desenvolver um novo mecanismo de processamento de registos apenas seja necessário desenvolver um novo módulo abstraindo-se assim de problemas e necessidades que possam surgir, por exemplo, como gerir e implementar uma, ou mais, ligações a uma base de dados, aceder e apresentar os dados a processar, entre outros. Dependendo do volume de registos a processar, o tempo de processamento pode variar imenso, podendo este ser de apenas poucos minutos a vários dias, se não mesmo semanas.

Esta dissertação descreve todo o processo para o desenvolvimento de uma aplicação que vem solucionar os problemas mencionados anteriormente. É pretendido que a aplicação forneça um conjunto de APIs que contenham todos os métodos necessários ao desenvolvimento de um módulo de processamento de dados.

Abstract

Nowadays with the widespread and popularity of wireless networks (WiFi), the number of users adopting this technology is increasing. When a user connects to a WiFi network and the user's connection activity logging is enabled it is possible to determine the user's location and movements in space, among others. Generally, the greater the number of users in the network the larger is the volume of logs generated, which can lead to a network capable of generating a massive amount of logs in a short period of time. However, different networks may be able to log different types of information about their users.

In order to get information about a user or a set of users in the the network it is required a tool to analyse and process a set of data in order to get the desirable information, e.g. the user location and movement in space. To do so, the tool must be flexible enough to adapt itself to different kinds of data to be processed without the need to redraw or, in extreme cases, to develop a whole new tool. This is where this paper arises, to develop an application capable and flexible enough to adapt itself into different kinds of records to process with the less amount of modifications and time spent. To do so, it was adopted the concept of processing modules, where each module is responsible for one, or more, types of data to process and whenever there is the need to develop a new processing mechanism there is only the need to develop a new module thus abstracting the problems and requirements that may arise, e.g. how to implement and manage one, or more, connections to a database, access and present processed data, among others. Depending on the volume of records to be processed, the amount of time required may vary quite a lot, from a couple of minutes to several days, if not weeks.

This paper describes the process of developing an application that solves the problems mentioned above. It is intended that this application provides a set of APIs containing all the necessary methods to develop a data processing module.

Índice

Lista de Figuras

Capítulo 1

Introdução

Com a constante produção e geração de informação que ocorre nos dias de hoje torna-se, por vezes, necessário desenvolver ferramentas especificas para se poder tratar e/ou trabalhar sobre um dado conjunto de dados. Isto leva a que, por vezes, seja necessário desenvolver interfaces gráficos para uma melhor interação entre o utilizador e a informação, a ser tratada. Tudo isto faz com que uma pessoa que pretenda desenvolver uma sistema para analisar um tipo de dados seja, em grande parte dos casos, obrigada a despender tempo e recursos na elaboração de um interface gráfico, gerir ligações a uma ou mais bases de dados, ler e guardar dados num ficheiro, entre outros, para a sua aplicação. Desta forma, pretende-se que o programador apenas necessite de despender o mínimo de tempo e recursos no desenvolvimento de todo um interface gráfico, gestão das ligações à(s) base(s) de dado(s) e maximize os seus recursos e tempo no seu principal objetivo, analisar e processar a informação que pretende. Para que tudo isto seja possível é disponibilizada uma *Application Programming Interface*, de ora em diante denominada por API, que abstrai e disponibiliza ao programador todo um conjunto de métodos responsáveis por aceder, ler, apresentar e guardar dados.

Um exemplo, e ao mesmo tempo um caso estudado nesta dissertação, prende-se com a análise do movimento de pessoas em ambientes com cobertura de redes *Wifi*. Hoje em dia, a popularidade das redes *Wifi* leva a que estas se encontrem disponíveis em inúmeros locais públicos e privados, desde espaços amplos a espaços mais confinados. Em áreas mais amplas, as redes *WiFi* são compostas por vários pontos de acesso que fornecem uma melhor e ampla cobertura de rede em toda a área circundante. Desta forma, sempre que um individuo recorre a uma rede *Wifi* é possível saber que a sua localização geográfica naquele instante é um local dentro da área de

cobertura do ponto de acesso a que se encontra ligado. Assim, mantendo um registo continuo dos pontos de acesso usados por um individuo é possível conhecer o seu percurso, isto é, o registo das redes *Wifi* usadas pelo individuo permite conhecer a forma como este se movimenta no espaço. Se a determinada hora estava ligado a um ponto de acesso e posteriormente a mesma pessoa encontra-se conectada a um outro ponto de acesso então pode-se concluir que o utilizador movimentou-se entre/no espaço abrangido pelos dois pontos de acesso, sendo que pode ter-se movimentado ou não pelo caminho mais curto entre os dois pontos de acesso.

1.1 Objetivos

Uma vez feito o enquadramento em que está inserida esta dissertação podem-se então os objetivos gerais da dissertação.

1. Desenvolver um interface gráfico ao qual seja possível adicionar e remover módulos de processamento.
2. Criar uma *Application Programming Interface* (*API*) a disponibilizar a cada módulo de processamento.
3. Elaborar um módulo de processamento para a deteção de um determinado tipo de eventos, *Ping Pong Events*.
4. Disponibilizar um módulo exemplo para servir como base para quem posteriormente desejar desenvolver um, ou mais, módulos.

1.2 Estrutura do Documento

Esta dissertação está dividida em sete capítulos. Neste primeiro capítulo é feita uma breve introdução ao tema a abordar assim como são identificados os objetivos propostos.

No segundo capítulo é realizada uma abordagem ao estado da arte em que esta dissertação se insere, mais concretamente às redes Wifi, ao movimento de pessoas conectadas a redes Wifi e ao processamento de dados.

No terceiro capítulo são introduzidas as tecnologias utilizadas neste trabalho, mais concretamente Java, Java Swing, MySQL, JDBC e finalmente Java Persistence API.

No quarto capítulo é feita uma introdução e descrição da aplicação, ProcessUM, desenvolvida no âmbito desta dissertação assim como as suas características e funcionalidades. Neste capítulo é também feita uma abordagem ao módulo criado, qual o seu objetivo e como se processa e interpreta informação gerada por si.

O quinto capítulo descreve como foi desenvolvida a aplicação, concretamente a sua arquitetura, bibliotecas e o funcionamento dos módulos de processamento. É também neste capítulo que é explicado como adicionar um módulo de processamento ao ProcessUM.

No sexto capítulo são apresentados os testes de performance da aplicação assim como da sua API.

Por fim, no sétimo capítulo são apresentadas as conclusões e possível trabalho a ser realizado no futuro.

Capítulo 2

Estado da Arte

2.1 Redes

A massificação das redes WiFi em conjunto com a popularidade dos *smartphones* e computadores portáteis, entre outros dispositivos com fácil acesso a redes Wifi, abriu espaço para uma nova área de investigação: o estudo do movimento de pessoas em zonas abrangidas por redes Wifi.

Neste capitulo é feita uma breve introdução a algumas tecnologias e conceitos recorrentes a redes Wifi, começando por uma descrição do que são redes Wifi e seus constituintes, *access points*. Dentro das redes Wifi é introduzido o conceito de *handover*, *ping pong* e *remote authentication dial in user service*. No final deste capitulo vem uma introdução ao estado atual do estudo do movimento de pessoas em redes Wifi.

2.2 Redes WiFi

A rede Wifi tem a sua origem no ano de 1988 [1], ano em que o Federal Communications Commission, dos Estados Unidos da América, define as bandas ISM (*Industrial, Scientific and Medical radio bands*)[2]. Já o seu *standard* foi, e continua a ser, definido pelo grupo de trabalho IEEE 802.11. Estas redes permitem a troca de dados através do meio, sem recurso a fios, entre um dispositivo eletrónico e uma rede de computadores. Rede essa que pode ser uma rede local e/ou com ligação à Internet. Deste forma, as redes Wifi recorrem a produtos certificados que pertencem à classe de dispositivos de rede local sem fios baseados no padrão IEEE 802.11, sendo que, atualmente,

é um padrão na conectividade em redes sem fios e os números falam por si. Em 2013, o número de dispositivos com conectividade Wifi cresceu 19%, atingindo assim o número de 1.9 mil milhões de unidades [3]. No segundo trimestre de 2014 o mercado de redes Wifi cresceu 9.2% [4] em comparação com o mesmo período do ano de 2013.

2.2.1 Arquitetura de Redes WiFi

A arquitetura de uma rede Wifi é constituída por *Basic Service Set (BSS)*, *Extended Service Set (ESS)*, *Distribution System (DS)* e pelos terminais. O *BBS* é constituído por um conjunto de terminais que comunicam entre si, sendo que a comunicação ocorre numa área de cobertura, *Basic Service Area*, isto é, a área de cobertura que um, ou mais, *BSS* consegue fornecer. Quando um terminal opera nesta área, pode comunicar com outros membros da *BSS*. Quanto à *ESS*, trata-se do conjunto de células *BSS* em que os seus *BSS* se encontram ligados à mesma rede. Assim, um terminal pode movimentar-se entre células da mesma *ESS* mantendo a ligação à rede. Um *(ESS)* pode ser identificado pela sua *(Service Set ID) (SSI)* que é representada por uma *string* de caracteres com um tamanho máximo de 32 bytes.

O *DS* corresponde ao *backbone* da rede Wifi e é responsável pela comunicação entre os *access points*. Um *BSS* pode funcionar em dois modos, modo *Ad-hoc* e modo de infraestrutura.

Modo Ad-Hoc

O modo *ad-hoc* é o modo mais simples de comunicação entre terminais numa rede sem fios. Este permite a comunicação entre terminais diretamente sem recorrerem a um *access point*. Neste tipo de redes, não há topologia predeterminada e tão pouco existe um controle centralizado. Deste modo, redes *ad-hoc* não requerem uma infraestrutura tal como *access points* configurados antecipadamente, uma vez que no modo ad-hoc os terminais comunicam diretamente entre si. A responsabilidade pela organização e controle da rede é distribuída entre os próprios terminais. Quando alguns pares de terminais não são capazes de comunicar diretamente entre si leva a que exista uma retransmissão de mensagens entre os terminais para que esses pacotes sejam entregues no seu destino. A figura 2.1 é uma ilustração de um rede *ad-hoc*.

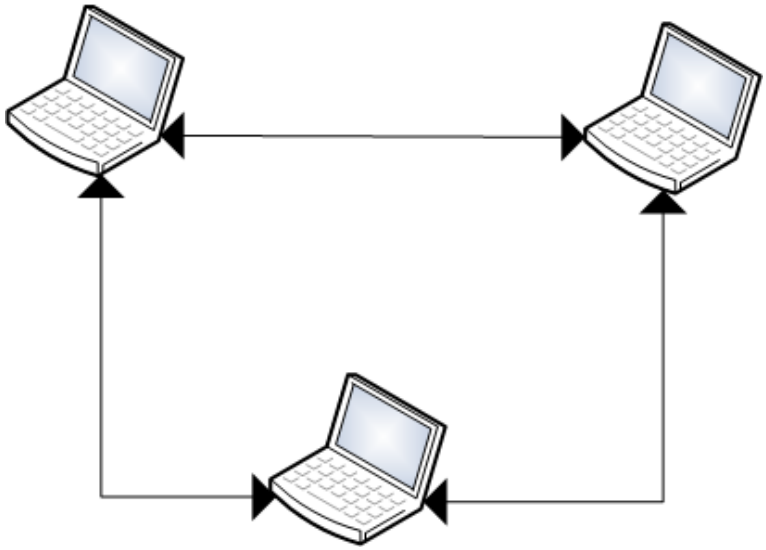


Figura 2.1: Representação de uma rede *ad-hoc*.

Modo de Infraestrutura

As redes em modo de infraestrutura distinguem-se das redes *ad-hoc* pela utilização obrigatória de um *access point* que neste tipo de redes tem a responsabilidade de interligar duas redes, sendo uma a rede fixa, por exemplo a internet, e a outra uma rede Wifi. Deste modo, um *access point* é um ponto de passagem obrigatório na comunicação entre todos os dispositivos na rede Wifi com os dispositivos na rede local. Num caso prático, e como exemplo, a rede Wifi *Eduroam* é uma rede em modo de infraestrutura pois liga os terminas conectados a um *access point* com uma rede exterior, neste caso a internet. Para tal, os dispositivos necessitam obrigatoriamente de estar numa área coberta pela rede Wifi. Assim, a área de cobertura de uma rede estruturada é definida pela área que um, ou mais, *access points* conseguem cobrir. Obrigando assim a que todos os terminais se localizem obrigatoriamente dentro da área de cobertura de um ou mais *access points*. Como os terminais comunicam diretamente com o *access point*, e não com outros terminais como no caso do modo ad-hoc, não existem restrições à distancia entre terminais. Um terminal para se ligar a uma rede Wifi primeiro tem necessariamente que se associar a um *access point* de forma a ter acesso aos serviços disponibilizados na rede. Ou seja, é semelhante aquando da ligação de um terminal a uma rede via cabo *ethernet*.

Quando um terminal se liga ao *access point* fá-lo sempre com o intuito de realizar a sua associação com o *access point*. No entanto, cabe ao *access point* permitir ou não o registo do terminal.

Uma das limitações das redes Wifi é apenas permitir que um terminal possa estar associado a um único *access point*. Do lado do *access point* não existem limites quanto ao número de terminais que pode ter associados a si. No entanto é importante referir que o numero de *ips* disponíveis para uma rede, embora seja um valor muito grande, é fixo. Não deixando de ser um aspeto a ter em conta nas limitações de redes Wifi modo de infraestrutura, sendo que este tópico já foge um pouco ao tema desta dissertação.

Extended Service Areas

Uma *Extended Service Area (ESS)* consiste numa rede Wifi através do agrupamento de várias *BBS* com uma rede *backbone* formando assim uma rede com uma elevada área de cobertura onde a cada *access point*, na mesma *ESS*, é atribuído o mesmo *Same Service Set Identifier* cuja função é de identificar a rede, do ponto de vista do utilizador.

Como se pode constatar na figura 2.2, a *ESS* é constituída pelo conjunto de quatro *BSS*, onde cada *access point* se encontra configurado com o mesmo *SSID*. Desta forma é dada a possibilidade do terminal se movimentar pela área de cobertura do *ESS* sem que este se preocupe a qual *access point* se encontra ligado.

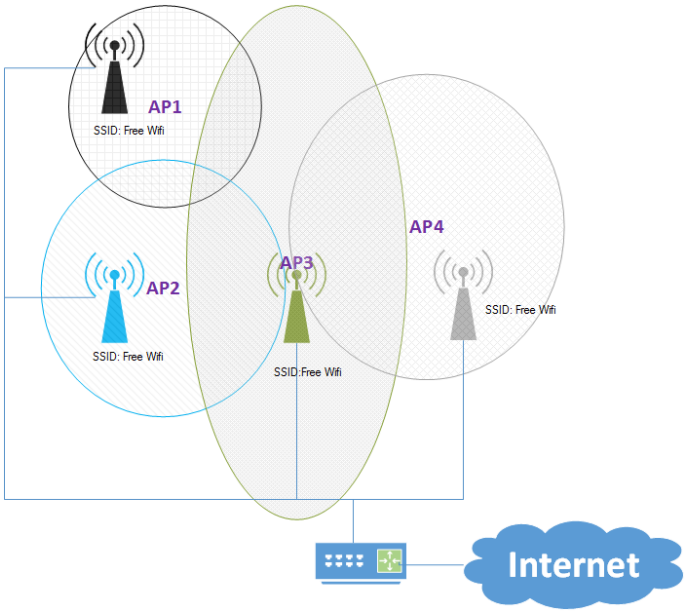


Figura 2.2: *Extended Service Areas*.

2.2.2 Access Point

Um *access point* tem como principal finalidade interligar os dispositivos móveis que estão conectados a si com os restantes elementos da rede Wifi e com a rede à qual se encontra ligado via cabo de rede, tipicamente a internet. A área que um *access point* consegue cobrir depende de vários fatores, como por exemplo, barreiras físicas, interferência de outras redes, potência do sinal emitido, entre outras. Desta forma, um dado *access point*, tipicamente, não consegue cobrir o mesmo número de metros quadrados quando este se encontra localizado numa dada posição A e posteriormente se desloca para a posição B. Por exemplo, um *access point* na localização *A* cobre 20 metros quadrados e na localização *B* cobre 40 metros quadrados.

A figura 2.3 ilustra como podem ser distribuídos os *access points* numa dada área geográfica e as suas diferentes áreas de cobertura.

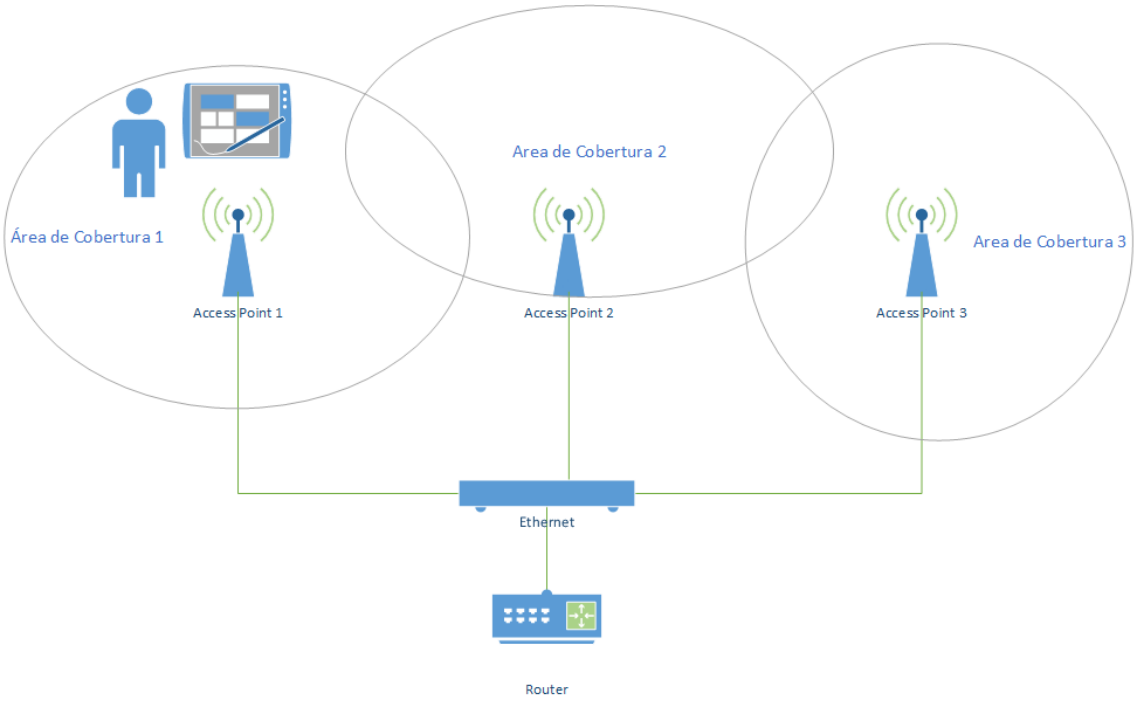


Figura 2.3: Distribuição de *access points* numa dada área geográfica.

No entanto, vários *access points* podem agir em conjunto com a finalidade de fornecer uma maior área de acesso a uma rede Wifi, como exemplificado na figura 2.3. Para tal, a área é subdividida em áreas menores sendo que cada uma delas abrangida por um, ou mais, *access point*. Desta forma é possivel fornecer acesso, sem interrupções, a uma maior área e possibilitar que um utilizador se movimente entre áreas cobertas por diferentes

access points, pertencentes à mesma rede Wifi, tal é possível recorrendo ao conceito de *handover*.

Um *access point* é, normalmente, mais complexo e consome mais energia que um terminal. No entanto um *access point*, na grande maioria dos casos, está ligado à tomada de eletricidade, não necessitando assim de baterias ou de outras fontes de energia armazenadas. O contrário acontece com a maioria dos terminais, que recorrem a baterias. Um dos aspetos importantes num terminal móvel é a autonomia. De modo a economizar a energia, quando um terminal, não se encontra a transmitir nem a receber dados, pode entrar num estado denominado por *sleep mode* por alguns períodos de tempo economizando energia e quando um *access point* se apercebe que um terminal se encontra no modo de poupança de energia, *sleep mode*, guarda as suas tramas num *buffer* possibilitando assim que o terminal possa entrar ou sair do *sleep mode* apenas para transmitir e receber as tramas contidas no buffer do *access point* [5].

2.2.3 Handover

O *handover* é o método utilizado nas redes sem fios para possibilitar a transição de um dispositivo móvel entre *access points*, na mesma rede, de forma transparente ao utilizador.

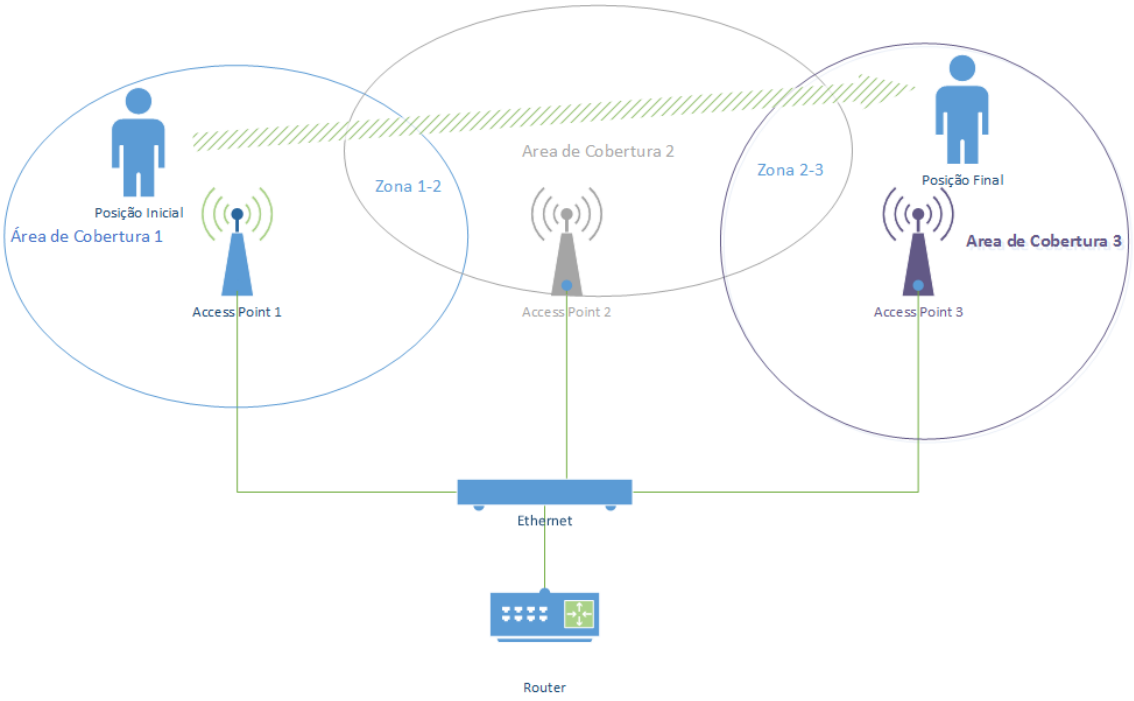


Figura 2.4: Handover *access points* numa dada área geográfica.

Como se pode constatar na figura 2.4, existe um movimento da pessoa no espaço. Assim que inicia o movimento em direção ao *access point* 3 o utilizador chega a uma região em que obtém sinal do *access point* 1 e do *access point* 2, região marcada como *Zona 1-2*. Nesta área o equipamento do utilizador vai agir conforme a sua configuração, ou se mantém ligado ao *access point* 1 ou troca a sua ligação para *access point* 2, sendo este um caso de *handover*, isto é, o equipamento desliga-se do *access point* 1 e faz uma nova ligação ao *access point* 2, tudo isto sem o utilizador se aperceber. Caso o equipamento se encontre configurado para, na região *Zona 1-2*, manter a ligação ao *access point* 1 assim que o utilizador abandona por completo a área de cobertura do *access point* 1 é feito o *handover* da ligação para o *access point* 2. O mesmo acontece para a *Zona 2-3* e quando o utilizador chega ao seu destino, área coberta pelo *access point* 3.

Desta forma, pode-se dizer que o processo de *handover* tem, por exemplo, as seguintes finalidades:

- Manter a ligação ativa aquando da movimentação do dispositivo entre diferentes áreas de cobertura transferindo a ligação entre *access points*.

- Quando um determinado *access point* se encontra sobrecarregado pode ser determinado que a melhor opção é transferir a ligação para um *access point* vizinho de forma a aliviar a carga no *access point*.

- Existência de um *access point* com maior força de sinal comparativamente com o *access point* ao qual se está conectado.

No entanto, existem dois tipo de *handover*, *hard handover* e *soft handover*.
O *hard handover* consiste em primeiro terminar a ligação corrente e só depois efetuar uma nova ligação ao novo *access point*. A interrupção na ligação é, geralmente, tão curta que o utilizador nem se apercebe. Este modo de *handover* é mais fácil de realizar e consome menos recursos, isto porque não é necessário suportar várias ligações em simultâneo.

Outro aspeto a ter em conta é o facto de que quando o dispositivo se encontra numa zona coberta por mais que um *access point*, como por exemplo na figura 2.4 as zonas *Zona 1-2* e *Zona 2-3*, o dispositivo pode estar a realizar *handovers* entre o *Access Point 1* e o *Access Point 2*, ou seja, alternando a sua ligação entre os dois *access points*. Este fenómeno chama-se *ping-pong* e é explicado mais à frente.

O *soft handover* é quando o dispositivo primeiro faz uma nova ligação e só depois termina a ligação mais antiga. Isto é, a ligação ao *access point* atual só é terminada quando existir uma nova ligação ao *access point* alvo. O

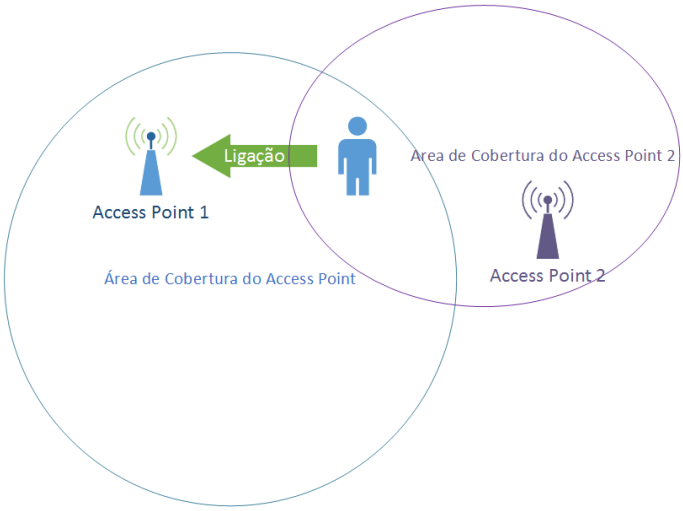


Figura 2.5: *Hard Handover*

intervalo de tempo em que existem as duas ligações ativas em simultâneo é, geralmente, muito curto ou apenas momentâneo. No entanto, *soft handover* pode envolver conexões a mais que dois *access points*.

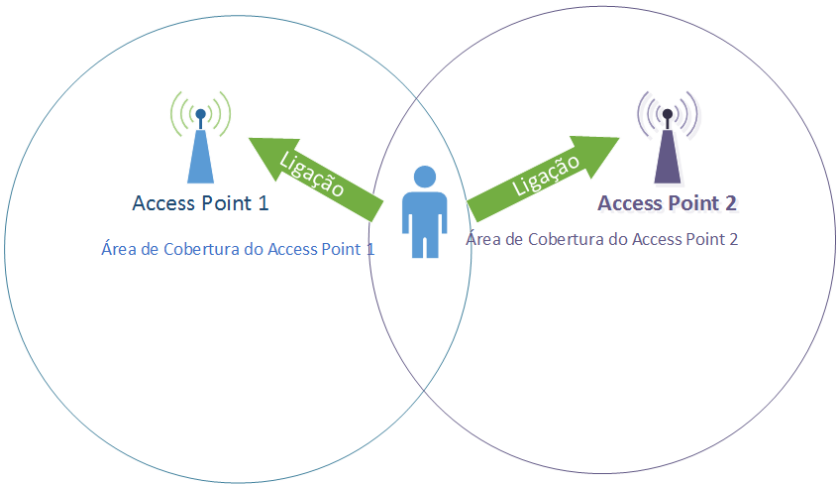


Figura 2.6: *Soft Handover*

Relativamente às vantagens e desvantagens do *hard handover* e *soft handover*, pode-se dizer que com o método de *hard handover* em qualquer momento, o dispositivo apenas tem uma ligação ativa na rede *WiFi*. O facto de não necessitar de suportar múltiplas ligações torna este método mais fácil e barato de implementar num dispositivo. Já as desvantagens do *soft handover* consistem em caso as operações de *soft handover* falhem a ligação pode ser perdida temporariamente ou terminada abruptamente. De forma a prevenir a perda da ligação,

normalmente existem mecanismos para re-estabelecer a ligação ao *access point* antigo caso a ligação ao novo *access point* falhe. No entanto re-estabelecer esta ligação pode nem sempre ser possível por diversos fatores, como por exemplo, o utilizador manteve o movimento e deixou de estar na área de cobertura do *access point*.

As vantagens do *soft handover* consistem no facto da ligação ao *access point* atual ser terminada apenas quando existe uma ligação fiável ao *access point* alvo, diminuindo desta forma as chances de existir uma perda de ligação devido à falha de *handover*. No entanto, a grande vantagem do *soft handover* é existirem, pelo menos, duas ligações ativas a, pelo menos, dois *access points* diferentes, logo, apenas existe total perda de ligação caso se perca todas as ligações durante o *soft handover*.

Como na maioria dos casos o *handover* ocorre quando existe a sobreposição de um sinal de um *access point* por um outro da mesma rede, colocando em causa a estabilidade da ligação atual, o *soft handover* trás a vantagem de se poder continuar a utilizar a ligação com a mesma, ou até mais, fiabilidade nestas circunstancias, devido à compensação do sinal de um *access point* para com o outro. No entanto, estas vantagens têm um custo. Custo esse que consiste no maior custo e complexidade para desenvolver este tipo de hardware que é capaz de processar várias ligações em paralelo.

Nas redes WiFi é utilizado o *hard handover*. A transição entre *access points* requer a cooperação dos mesmos. Pegando no exemplo da figura 2.5, aquando da ocorrência de *handover* o *access point* 2 tem que informar o *access point* 1 que o terminal está agora associado ao *access point* 2. Do lado do terminal, este desassocia-se do *access point* 1 e pede uma associação ao *access point* 2. Caso seja aceite pelo *access point* 2, o terminal reassocia-se de novo à rede.

O *standard* 802.11 não especifica detalhes quanto à comunicação entre *access points* durante a transição de *BSS*. Deste modo, mesmo que os dois *access points* se encontrem no mesmo *ESS* eles podem não estar conectados pelo mesmo *router* de *backbone*. Sendo esta uma das razões pela qual não é possível existir *soft handover* em redes WiFi.

2.2.4 Ping Pong

Quando um utilizador se encontra numa zona de interceção de sinal de dois, ou mais, *access points*, a variação da potência do sinal recebido dos respetivos *access points* faz com que o dispositivo realize operações de *handover* de forma a tentar manter a conectividade com o *access point* com melhor condição para manter uma boa conexão,

mantendo assim a fiabilidade da ligação, sendo esta condição a potência do sinal do *access point*. Por sua vez, a potência do sinal do *acces point* pode estar dependente da carga existente no *access point*, como explicado na subsecção seguinte (Network-side Load Balance), quando um *access point* se encontra com uma elevada carga este pode reduzir [6] a potência do seu sinal. No entanto, a potência do sinal e/ou a carga de um *access point* flutua frequentemente o que vai originar que o dispositivo altere constantemente o *access point* ao qual se encontra ligado. Por exemplo, se for ligado um *access point A* junto a um outro *access point B*, bastante congestionado, ao ser detetado o novo *access point A* todos os terminais que se encontram ligados ao *access point B* vão tentar associar-se com o *access point A* praticamente ao mesmo tempo. Desta forma, o *access point A* vai também ele ficar congestionado devido a este fluxo de associações provenientes do *access point B*. Em seguida o *access point B* deixa de estar congestionado e os terminais voltam a reassociar com o *access point B*. Isto pode transformar-se num ciclo vicioso [6] .

De forma a evitar a ocorrência de *ping pong* pode-se atribuir estaticamente um *access point* ou distribuir as re-associações pelo domínio do tempo. Por exemplo, um terminal procura pelo *access point* com melhores condições. Quando o encontra e caso este novo *access point* seja diferente do anterior o terminal não se associa imediatamente a este *access point* mas gera uma valor aleatório *X* [6]. O terminal apenas se associa ao melhor *access point* caso este seja identificado como o melhor por *X* vezes consecutivas[6]. Desta forma é evitado o fluxo repentino de terminais para um dado *access point*. Uma desvantagem desta abordagem consiste no facto de serem os terminais a escolher o *access point* ao qual se associam, o que leva a que não seja possível controlar a carga existente num dado conjunto de *access points*, isto é, manter a carga num *access point* estável. Isto porque, nada impede que um *access point* seja identificado como o melhor *access point* pela grande maioria dos terminais. Uma forma de contornar este problema é fazer com que a própria rede faça a sua gestão de congestionamento, mais conhecido por *network-side load balance*.

Network-side Load Balance

Na abordagem de *network-side load balance* os terminais comportam-se como agentes passivos no âmbito das associações entre terminais e *access points*. É uma entidade do lado da rede, que pode ser um *access point*, *switch* ou até mesmo um servidor dedicado, que controla a distribuição [6] da carga pelos *access points*. Existem três formas de os *access points* controlarem a sua carga.

- Ajuste da área de cobertura. Os *access points* com uma elevada carga podem reduzir a potência de transmissão do seu sinal. Existindo assim menor probabilidade de novos terminais descobrirem o *access point*.

- *Access points* podem colaborar entre si de forma a ajustarem a sua área de cobertura de modo a que *access points* com menor carga possam cobrir áreas maiores que *access points* com carga mais elevada.

- Um *access point* pode rejeitar novas associações.

- Um *access point* com elevada carga pode enviar *frames* de desassociação para um determinado numero de terminais com o objetivo que esses terminais se associem com um outro *access point* de menor carga.

Para uma visão da distribuição da carga pela rede, os *access points* podem trocar informação entre si sobre o seu nível de carga, recorrendo para tal a uma ligação física, por exemplo, via *ethernet*.

Uma outra alternativa é recorrer a um servidor dedicado que recolhe os níveis de carga dos *access points* da rede, por exemplo via protocolo *SNMP* [6]. Com esta informação, o servidor dedicado recomenda/ordena que determinado terminal se desassocie do *access point* e se associe a um outro, pré determinado, *access point*.

2.2.5 Remote Authentication Dial In User Service

Atualmente, a existência de acordos de mobilidade entre diferentes redes WiFi tornou a utilização do protocolo *Remote Authentication Dial In User Service* (*RADIUS*) fundamental para validar a associação de um terminal de uma rede *X* numa outra rede *Y* e vice-versa. Um bom exemplo prático da utilização do *RADIUS* é a rede *Eduroam*, na qual é baseado o estudo alvo desta tese de dissertação.

Assim, o *RADIUS* é um protocolo de rede que centraliza a autenticação, autorização e *accounting* no processo de gestão de dispositivos ligados a uma determinada rede. O protocolo é do tipo cliente/servidor, corre na camada de aplicação da pilha *OSI* (*Open Systems Interconnection*) utilizando o protocolo *UDP (User Datagram Protocol)*.

Os Servidores de Acesso Remoto (RAS), *Virtual Private Networks* (VPNs), *Switch* de rede com autenticação baseada nos portos e Servidores de Acesso à Rede (NAS) possuem um cliente *RADIUS* que comunica com o servidor RADIUS, servidor este que é um processo a correr em segundo plano num sistema *Unix* ou *Microsoft Windows Server*[7].

O protocolo *RADIUS* tem três objetivos.

- Autenticar utilizadores ou dispositivos antes de estes terem acesso à rede.
- Autorizar utilizadores ou dispositivos para certos serviços disponibilizados pela rede.
- Fazer a contabilização do uso dos recursos de um dado utilizador ou dispositivo.

Uma rede que recorre ao *RADIUS*, é composta por:

- Um, ou mais, clientes. Isto é, quem deseja usufruir de um recurso da rede, como por exemplo, um dispositivo que se pretende ligar a um *access point*.
- *NAS* (*Network Access Server*), consiste em quem recebe a solicitação do cliente, por exemplo um *access point*, e autentica esse pedido no servidor *RADIUS*.
- Servidor *RADIUS*, é quem valida o pedido proveniente do *NAS*. A resposta do pedido de autenticação pode ser positiva ou negativa.
  - Nas respostas positivas, os parâmetros de resposta são usados de forma a orientar o *NAS* para com o cliente. Nas redes *WiFi*, os parâmetros podem consistir no tempo máximo de ligação permitida, a chave de criptografia que deverá ser usada no canal de comunicação entre o cliente e o *NAS*, entre outros.

O *RADIUS* é amplamente utilizado para facilitar *roaming* entre diferentes prestadores de serviço, por exemplo, a rede *Eduroam* que permite um utilizador da rede da Universidade do Minho possa utilizar os seus dados de autenticação para se autenticar e ter acesso à rede da Universidade do Porto.

2.3 O Movimento de Pessoas em Redes WiFi

Nos dias que correm, as redes *WiFi* encontram-se disponíveis em inúmeros espaços públicos e privados. Nos locais mais amplos, as redes *WiFi* são compostas por vários *access points* com a responsabilidade de fornecer uma ampla cobertura de rede. Assim, sempre que um individuo recorre a uma rede *WiFi* é possível saber a sua localização geográfica naquele instante, ou seja, a sua localização é sempre um local dentro da área de cobertura do *access point* a que se encontra ligado. Assim, recorrendo aos dispositivos utilizados no dia-a-dia e que suportam redes *WiFi*, como por exemplo o computador pessoal, mantendo um registo continuado dos *access points* que um dado utilizador usou é possível conhecer e, possivelmente prever, o seu percurso.

No entanto, uma das limitações das redes sem fios é o seu alcance, o que implica que um utilizador ao sair do alcance de um *access point* o seu dispositivo vai tentar conectar-se a um outro *access point* na sua vizinhança. Desta forma, caso sejam mantidos os registos de *handover* é possível traçar um trajeto aproximado do utilizador.

Para se poder analisar o movimento das pessoas é necessário recorrer a uma base de dados que contém os registos de utilização de uma rede *WiFi*, no caso desta dissertação, a rede *WiFi* da Universidade do Minho, que contém o registo dos pontos de acesso utilizados por uma dada pessoa. Deste modo é possível conhecer a forma como as pessoas se movimentam no espaço do *campus* da Universidade do Minho. A figura 2.7 ilustra como todo este processo se desencadeia.

Como se pode ver na figura 2.7, o utilizador inicia o seu percurso pelo qual vai passando pela área de influência de diversos *access points*. Consultando o registo dos *access points* é possível saber que o utilizador se manteve sempre muito próximo dos *access points AP1, AP2, AP3, AP4, AP7, AP8* e *AP9*. O contrário acontece dos *access points AP5* e *AP6*. Tendo o mapa da infraestrutura é possível saber que o utilizador se manteve fiel ao seu corredor, isto é, não optou por seguir para o corredor do *access point AP5* e *AP6*. É com base neste conceito que é elaborada esta tese de dissertação.

No entanto, já existem alguns estudos e trabalhos sobre este tema que recorreram a dados recolhidos através de, por exemplo, redes *GSM* e redes *WiFi*. No caso das redes *GSM* foram utilizados dados referentes ao número de chamadas processadas por um *Base Transceiver Station* (*BTS*) ou a localização aproximada do utilizador sempre que este se liga a um BTS para fazer ou receber uma chamada no seu terminal móvel *GSM*. No primeiro exemplo, quantas mais chamadas processar um *BTS* mais utilizadores este tem na sua vizinhança. Este é um bom método para detetar o movimento de *massas*, como por exemplo o movimento de pessoas numa zona geográfica. Quanto

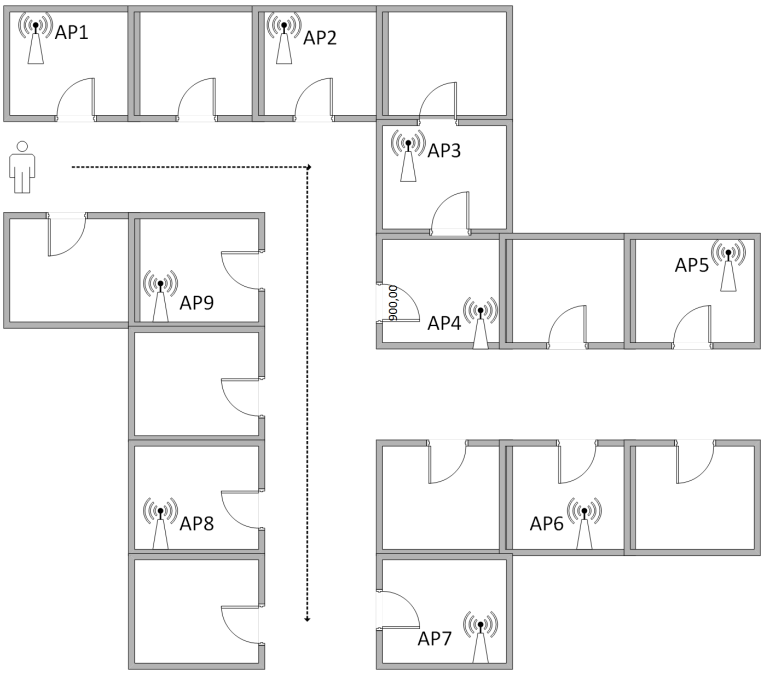


Figura 2.7: *Movimento de pessoas numa rede WiFi*

ao segundo exemplo, para um utilizador receber ou efetuar uma chamada este tem que se ligar a um determinado *BTS*, *BTS* esse que cobre uma zona geográfica, desta forma é definida a zona onde o utilizador se encontra. Nas redes *Wifi*, a técnica utilizada é em parte semelhante à do *GSM*, pois sempre que um utilizador se liga a um *access point* este encontra-se na sua área de cobertura. Tudo isto aliado a um diagrama temporal é possível determinar o movimento de um ou vários utilizadores e, consequentemente, tentar prever os seu movimentos futuros.

Estudos [8][9] concluíram que independentemente da localização do utilizador este utiliza a sua ligação *Wifi* para aceder aos mesmos conteúdos, variando os mesmos de pessoa para pessoa, e que existem dois picos de ligação às redes *Wifi* que se repetem por toda a semana. Isto deve-se quando o utilizador se liga à rede *Wifi* no seu local de trabalho e posteriormente quando faz o mesmo assim que chega a casa, desta forma os picos são, geralmente, um de manhã e outro no final do dia. Em geral, grande parte das pessoas passam a maioria do seu tempo num restrito grupo de localizações geográficas [10]. Embora a maioria das pessoas nas suas atividades diárias se encontrem confinadas a uma zona de alguns quilómetros quadrados, geralmente de 1 a 10 quilómetros, existe uma outra parte que cobre diariamente centenas de quilómetros. Estas diferenças podem dar a ideia que é mais fácil prever a localização das pessoas que se movimentam pouco do que pessoas que se movimentam bastante.

No entanto isto não é o que acontece na prática pois, apesar da aparente aleatoriedade no movimento das pes-

soas, um histórico do movimento diário das mesmas esconde inexplicavelmente um elevado grau de previsibilidade. Desta forma, pode-se dizer que indivíduos que percorrem centenas de quilómetros, diariamente, são tão previsíveis quanto indivíduos que percorrem apenas algumas dezenas de quilómetros. Em média, a previsão do movimento de um dado utilizador é, geralmente, de 93% [10], sendo que não é menor, no pior dos casos, que 80% [10]. Ao nível do movimento de massas, constatou-se que enquanto o movimento de massas numa determinada zona decresce esta aumenta numa outra zona. Assim como as variações no volume de chamadas processadas por um *BTS* são explicadas pela hora, dia e pela rotina das pessoas. Já os diferentes padrões de atividade numa cidade podem ser atribuídos à sua demografia [10], por exemplo, edifícios governamentais observam um maior número de movimentos durante o dia e em dias de semana comparavelmente com os restantes períodos do dia e semana, sendo este fator responsável por 21% da variação de atividade entre *BTSs* [10].

2.4 Processamento de Dados

O processamento de dados, de um modo geral, é o nome dado ao registo e manipulação de pedaços de informação de forma a produzir informação relevante e de fácil perceção [11]. Isto é, o processamento de dados é qualquer processo que recorra a uma determinada aplicação de software para a recolha de dados, selecione os dados relevantes, faça a sua análise e por fim transforme os dados em informação pronta a ser utilizada e/ou analisada. Este processo pode ser automatizado e envolve os seguintes passos.

- Recolha
- Análise
- Ordenação
- Seleção
- Processamento
- Difusão dos dados
- Armazenamento

No entanto é preciso não confundir o processamento de dados com a conversão de dados, isto é, quando o processo consiste em apenas converter os dados para um novo formato, não envolvendo qualquer tipo de manipulação nos dados.

De um modo mais abstrato, pode-se dizer que o processamento de dados, como exemplificado na figura 2.8, consiste numa entrada de dados que podem ser em tempo real, *entrada*, ou previamente armazenados *armazenamento*, no processamento dos dados e no resultado desse processamento que pode ser apresentado, *saída*, ou guardado, *armazenamento*.

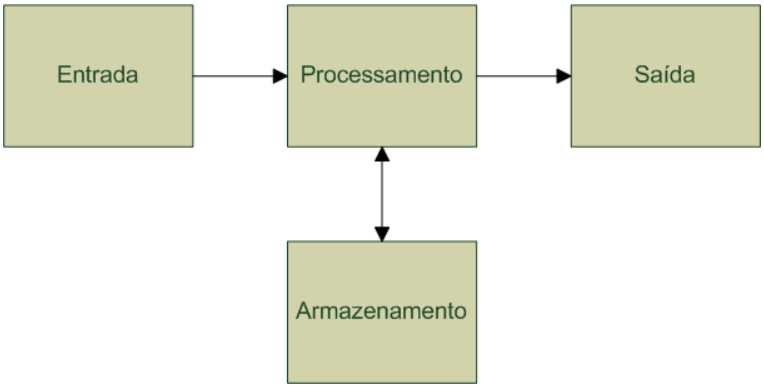


Figura 2.8: *Diagrama de blocos do processamento de dados*

Existem três modos de processamento de dados, sendo estes listados em seguida.

- Processamento em tempo real
- Processamento em sistemas distribuídos
- Processamento em lotes

Num processamento em tempo real, em analogia à figura 2.8, conforme os dados, a processar, são gerados estes são imediatamente tratados como entrada de dados sendo em seguida imediatamente processados e posteriormente apresentados ou armazenados. Este tipo de processamento leva a que possam ocorrer picos elevados de utilização da unidade de processamento assim como períodos de inativo conforme o fluxo de dados a entrar. Desta forma é necessário tem em consideração a ocorrência destes picos e que o sistema consiga dar conta dos mesmo.

Um processamento em sistemas distribuídos, como o nome indica, consiste me distribuir a carga de trabalho por diversas unidades de processamento que se encontram, de alguma forma, interligados entre si. Desta forma os dados

a processar são distribuídos pelos elementos que constituem o grupo de processamento o que torna esta abordagem bastante poderosa tanto a nível de processamento como ao nível de carga no sistema. Quando uma das unidade de processamento por algum motivo fica incapacitada de processar dados a sua carga é distribuída pelos restantes membros do sistema distribuído. A grande vantagem deste tipo de sistema é o seu elevado poder de processamento, a sua escalabilidade e a sua flexibilidade. Sendo uma das principais desvantagens os elevados custos de instalação e manutenção do sistema. No entanto se não necessitar de deter e controlar toda a infraestrutura de um sistema distribuído pode contratar uma infraestrutura distribuída de um operador de serviços distribuídos, por exemplo a *Google Cloud Computing* [1], *Amazon Elastic Compute Cloud* [2], entre outros. Um exemplo do processamento de dados em sistemas distribuídos é o projeto *Folding*[3] da Universidade de Stanford.

O processamento em lotes é o tipo de processamento adotado nesta dissertação e consiste em processar os dados em lotes, ou seja, os dados no seu todo são divididos em lotes de menor dimensão e processados lote a lote. A vantagem neste tipo de processamento é o facto da carga no sistema poder ser, geralmente, fixa e/ou previsível de tal forma que o sistema de processamento pode ser adaptado e/ou configurado para a carga prevista uma vez que a carga a aplicar no sistema de processamento pode ser previamente conhecida dando, desta forma, uma maior flexibilidade ao utilizador, para planear quando executar a tarefa. Ao contrário dos sistemas de processamento em tempo real, no processamento por lotes não existem picos de atividade ou inatividade durante a sua execução.

Um exemplo de um tipo de processamento em lotes é o pagamento do salário dos trabalhadores de uma empresa com vários departamentos. O pagamento é dividido por departamento e posteriormente por posição. Isto origina a uma partição da informação a processar que são os chamados lotes.

É importante salientar que graças à sua flexibilidade, os sistemas distribuídos podem processar dados em lotes ou em tempo real.

Um dos aspetos a ter em conta no processamento de informação é o tempo de processamento e o tipo de informação a processar. Há situações em que o tempo de processamento é um fator decisivo. Por exemplo, o processamento de uma transferência bancária é um caso bastante sensível ao tempo que demora a executar, em regra geral se for de apenas uns segundos a um minuto é tolerável, mais que isto torna todo o processo ineficiente. Assim, pode-se dizer que aquando do planeamento do processamento de dados é necessário ter em conta vários

[1] https://cloud.google.com/compute/

[2] http://aws.amazon.com/pt/ec2/

[3] http://folding.stanford.edu/

fatores, sendo estes os seguintes.

- Tempo de processamento
- Recursos necessários (*Hardware*)
- Quantidade de dados a processar

No entanto, os três fatores estão interligados entre si pois existe uma certa dependência entre eles. Quanto melhor forem os recursos a utilizar menor é o tempo de processamento para a mesma quantidade de dados a processar. Assim, se o tempo de processamento for um fator decisivo é necessário ter os recursos necessários para garantir que o tempo de execução fica entre o intervalo de tempo tolerável. No entanto, o modo como está programado o processamento dos dados pode também ter um impacto considerável no tempo de execução. Isto é, se o processamento não for otimizado para a carga de dados a processar pode originar a que o tempo de processamento e a utilização de recursos seja superior ao esperado. Da mesma forma que os recursos devem ser otimizados o processamento dos dados também o deve ser. Um exemplo de uma má otimização do processamento de informação pode ser a existência em memória de vários pedaços de informação iguais. Um caso prático é a leitura do mesmo ficheiro para memória mais que uma vez. Isto leva a uma ineficiente utilização de recursos e instruções na unidade de processamento que por sua vez podem levar a um aumento do tempo de processamento.

De um modo geral, pode-se dizer que se deve sempre seguir as três regras seguintes.

- Filtrar ao máximo os dados a processar, é bom relembrar que regra geral quantos mais dados a processar mais recursos e tempo são necessários.
- Utilizar algoritmos eficientes no processamento dos dados.
- Um boa gestão dos recursos disponíveis.
- Tempo e disponibilidade para aguardar pelos resultados a processar, uma vez que estes podem demorar várias horas ou mesmo dias.

Capítulo 3

Tecnologias

Neste capitulo são introduzidas as tecnologias e ferramentas utilizadas na elaboração da solução para o problema proposto.

3.1 Java

Java é uma linguagem de programação orientada a objetos desenvolvida na década de 90 pela empresa Sun Microsystems. A grande diferença de Java para as linguagens de programação mais convencionais consiste que em Java o código é compilado para um *bytecode* que é executado por uma máquina virtual. Algumas das principais características são as seguintes.

- Orientada a objetos.
- Portabilidade, isto é, independente da plataforma em que é executada ("*write once, run anywhere*").
- Contém um vasto conjunto de APIs.

O IDE, *Integrated Development Environment*, utilizado foi o *Netbeans* 7 e mais tarde a versão 8. O JDK, *Java Development Kit*, utilizado foi inicialmente a versão 7 e mais tarde a versão 8, não sendo utilizada nenhuma das grandes novidades da versão 8, como por exemplo *streams*.

3.2 JAVA Swing

JAVA Swing é um *widget toolkit* que faz parte do *Java Foundation Classes* [12] e é responsável por implementar um conjunto de componentes que vão constituir um interface gráfico. Esta *framework* foi criada com o objetivo de fornecer a possibilidade de se criar interfaces gráficos com a capacidade de se adaptarem ao sistema operativo no qual vai correr, isto é, manter uma aparência consistente com o próprio sistema operativo.

A *API* do *Swing* permite fazer o *rendering* de todos os componentes por sua conta em vez de delegar essa função para o sistema operativo. Assim, é independente da plataforma em que se encontra a correr uma vez que corre na máquina virtual do *Java*.

3.3 MySQL

MySQL é um sistema, de código aberto, de gestão de base de dados relacionais que recorre a SQL para interagir com os dados por si geridos. A sigla SQL representa *structured query language* [13]. A versão mais recente do MySQL é a 5.7 [14] no entanto na dissertação foi utilizada a versão 5.6.12. As principais características do MySQL são as seguintes [15].

- Suporte para diferentes sistemas operativos, entre eles Oracle Linux, Oracle Solaris, Red Hat, Debian, Microsoft, Apple.
- Suporte completo para multi thread.
- Utiliza um sistema de alocação de memória baseado em threads.
- Suporta base de dados de grandes dimensões
- Aceita conexões de clientes via sockets TCP/IP.
- Um elevado numero de APIs suportadas, desde Java a PHP e C/C++.
- Suporte para clientes em Java que recorram ao JDBC através do interface Connector/J.
- Todos os dados são guardados utilizando o conjunto de *charset* selecionado.

Nesta dissertação o MySQL foi utilizado com a finalidade de servir como armazenamento de dados a processar assim como o resultado do processamento. A razão pela escolha de MySQL em detrimento de outras deve-se ao facto da sua popularidade, elevada penetração no mercado/área e porque o servidor da universidade que contém os dados a processar ele próprio utiliza MySQL.

3.4 Java Database Connectivity - JDBC

O JDBC é uma API de conectividade entra a linguagem de programação Java e uma base de dados que utilizem SQL. Deste forma, a API JDBC permite, através da linguagem de programação Java, aplicar o conceito de *Write Once, Run Anywhere* em aplicações que requerem a utilização de dados de e/ou para uma base de dados. O JDBC pode comunicar diretamente com a base de dados alvo ou comunicar com um *driver* que posteriormente faz a conversão do pedido para a linguagem da base de dados a utilizar. A utilização de *drivers* torna o JDBC flexível e acessível de usar. Geralmente um *driver* suporta mais que um tipo de base de dados e estes são desenvolvidos por fornecedores de bases de dados.

Recorrendo ao JDBC é possível realizar as três seguintes operações [16].

- Estabelecer uma ligação com a base de dados e/ou aceder a qualquer tabela contida na base de dados.
- Enviar queries em SQL.
- Processar os resultados.

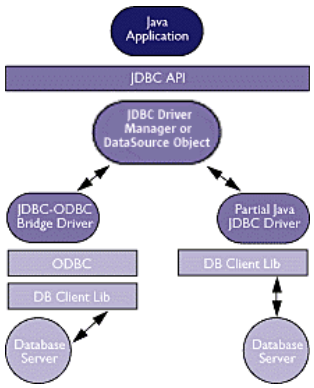


Figura 3.1: A arquitectura do JDBC [16]

Na figura 3.1 é representada a arquitetura do JDBC, onde no ramo do lado esquerdo é possível observar que o JDBC faz o pedidos diretamente na base de dados enquanto no ramo direito é utilizado um *driver* (*DB Middleware*) que faz a conversão do pedido do JDBC para a base de dados

As vantagens de recorrer ao JDBC para interagir com uma base de dados são as seguintes [16].

- Flexibilidade dado o numero de diferentes bases de dados suportadas.
- Facilidade de implementação e manutenção.
- Abstração da complexidade de aceder à base de dados.
- Não é necessária a sua instalação. Basta descarregar o ficheiro *jar* e adicionar o mesmo ao projeto.

3.5 Java Persistence API

Java Persistence API, também conhecida e de ora em diante tratada por *JPA*, é uma especificação Java para aceder, persistir e gerir dados entre objetos ou classes e uma base de dados relacional. É atualmente considerado o padrão para *Object to Relational Mapping* (*ORM*) em Java. Na prática, JPA é apenas uma especificação e não um produto, isto é, apenas especifica um conjunto de normas a implementar. Normas estas que são implementadas por vários fabricantes, por exemplo a *Red Hat* que é detentor do provedor *Hibernate*, a *Eclipse Foundation* com o *EclipseLink*, e a *Apache Software Foundation* com o *OpenJPA*. Entre os diferentes provedores que implementam JPA todas elas oferecem as mesmas funcionalidades, já que todas elas obedecem à especificação do JPA, no entanto existem algumas diferenças entre provedores ao nível de funcionalidades extras, como por exemplo a *Hibernate*, para alem do gestor de sessões padrão especificado pelo JPA, oferece um gestor de sessões especifico do *Hibernate*.

Recorrendo ao JPA, é possível persistir *Plain Old Java Objects* (*POJO*) numa determinada tabela de uma base de dados sem ser necessário que as classes implementem Java *interfaces*, já que, como foi dito anteriormente, estes já se encontram implementados no provedor escolhido pelo programador.

A grande vantagem de utilizar JPA para a interação com uma base de dados é abstrair o código da aplicação do código da base de dados. O programador não necessita de ter conhecimento da linguagem da base de dados que está a trabalhar uma vez que o JPA faz essa ponte entre o código Java e o código, por exemplo, SQL caso seja

uma base de dados SQL. Assim, é possível alterar o tipo de base de dados sem ser necessário modificar código na aplicação[1]. Isto é, alterar de uma base de dados *MySQL* para uma base de dados *PostgreSQL*, caso ambas sejam suportadas pelo provedor que implementa o JPA. No entanto, caso a base de dados não seja suportada é sempre possível alterar para um outro provedor que suporte a base de dados em questão, mais uma vez sem ser necessário alterar o código da aplicação, pode no entanto ser necessário fazer ajustes ou reconfigurar o provedor.

Um conceito importante a realçar é o conceito de classe entidade (*Entity*). Uma classe entidade é, numa perspetiva de código, uma instância de uma classe e, numa perspetiva de uma base de dados, uma linha de uma tabela. Isto é, essencialmente uma instância persistente de uma classe. Ao serem alterados valores na classe entidade funciona exatamente do mesmo modo que uma qualquer outra classe. A única diferença consiste no facto que se pode persistir essas alterações e, em regra geral, o estado atual da instância da entidade. Ao se persistir essa instância da entidade vai-se alterar os valores de uma determinada linha de uma tabela da base de dados que a entidade representa.

Na aplicação desta tese de mestrado o provedor escolhido foi a versão 2.1 do *EclipseLink*. A sua escolha deveu-se à sua forte implementação no mercado, facilidade de configuração, elevada quantidade de documentação e exemplos existentes assim como a minha familiarização com o *EclipseLink*. Este provedor suporta base de dados relacionais que sejam compatíveis com SQL e possuam um driver *JDBC*[2] compatível e bases de dados NoSQL [17]. A seguinte lista contém os tipos de bases de dados compatíveis com as duas primeiras condições, citadas anteriormente.

- Oracle
- Oracle JDBC (8, 9, 10, 11)
- MySQL
- PostgreSQL
- Derby
- DB2

[1] Caso sejam utilizados métodos inerentes ao provedor que fornece JPA tal pode não ser totalmente verdade. Assim, é altamente recomendado restringir ao máximo a utilização de métodos não especificados no JPA.

[2] Java Database Connectivity

- DB2 (mainframe)
- Microsoft SQL Server
- Sybase
- Informix
- SQL Anywhere
- HSQL
- SAP HANA
- H2
- Firebird
- Microsoft Access
- Attunity
- Cloudscape
- DBase
- PointBase
- TimesTen
- Symfoware
- MaxDB

Em seguida são apresentadas os tipos de bases de dados NoSQL compatíveis [18].

- MongoDB
- Oracle NoSQL

- XML files
- JMS
- Oracle AQ

3.5.1 Object Relational Mapping

Um ORM consiste num mapeamento em objetos Java de uma determinada base de dados em que um ou mais objetos representam um registo de uma ou mais tabelas de uma base de dados. Um bom exemplo é o de uma agenda de contactos telefónicos. Considerando que cada contacto na agenda representa uma pessoa com um ou mais números de telefone e zero ou mais endereços, a agenda pode ser modelada, segundo um conceito orientado a objetos, por um objeto "Pessoa"com os respetivos campos que correspondem a uma pessoa, o nome, uma lista de números de telefone e uma outra lista de endereços. A lista de números de telefone contém objetos do tipo "NumeroTelefone"e a lista de endereços contém objetos do tipo "Endereço". Assim, cada registo de um contacto na base de dados é representado por um objeto do tipo "Pessoa", que pode ter associado vários métodos, como por exemplo um método para retornar o numero de telefone principal, o endereço do local de trabalho, entre outros. Caso seja possível representar a lógica dos objetos numa forma atómica que seja suscetível de ser armazenada na base de dados preservando as propriedades dos objetos e suas relações e seja possível mais tarde voltarem a ser recriados, através do dados contidos na base de dados, estes objetos são considerados persistentes[19]. O JPA permite que o mapeamento de um objeto num ORM seja definido através de *annotations* (em português, anotações) ou definidos em XML. Em seguida é apresentado um bloco em pseudo-código de como aceder a um determinado registo guardado numa tabela na base de dados recorrendo à linguagem SQL.

```
String sqlQuery = "SELECT ... FROM pessoas WHERE id = 10"
DatabaseCall dbCall = new DatabaseCall(connection, sqlQuery);
Result result = dbCall.execute();
String name = result[0]["FIRST_NAME"];
```

Figura 3.2: Pseudo-código de como aceder à pessoa representada pelo "id"10 (dez) utilizando para tal linguagem SQL.

Agora é apresentado um outro bloco de pseudo-código de como realizar a mesma operação representada pela

figura 3.2 mas agora recorrendo ao JPA.

```
Person p = repository.getPerson(10); // id = 10
String name = p.getFirstName();
```

Figura 3.3: Pseudo-código de como aceder à pessoa representada pelo "id"10 (dez) recorrendo apenas aos métodos definidos pelo JPA.

Em comparação com a utilização do JDBC, o JPA é uma tecnologia de mais alto nível que por sua vez é de mais rápida implementação e fácil de manter. A sua simplicidade faz com que o programador não necessite de ter conhecimentos ao nível da linguagem SQL. Na grande maioria dos casos, se for alterada a tecnologia que suporta a base de dados não é necessário fazer alterações no código. Por sua vez, sendo de mais baixo nível, e como se trabalha diretamente com código SQL caso se pretenda alterar a tecnologia da base de dados é necessário alterar o código de forma a suportar a nova tecnologia adotada para a base de dados, no entanto JDBC é mais rápido a delegar e a executar as tarefas que o JPA, como se vai constatar no capitulo de Testes.

Capítulo 4

ProcessUM

A aplicação ProcessUM é uma ferramenta à qual podem ser adicionados módulos de processamento. Na sua API existem métodos para leitura, escrita e apresentação de dados e de interação com o utilizador. Deste modo, quem pretender adicionar um módulo de processamento apenas necessita de se preocupar em aplicar o algoritmo de processamento no seu módulo e incluir o módulo no ProcessUM, sendo que as restantes tarefas pode delegar para API do ProcessUM.

A figura 4.1 representa o interface gráfico do ProcessUM. Em seguida é caracterizado o interface gráfico da aplicação, como se pode constatar pela figura 4.1 cada retângulo representa o seguinte.

- Azul - área dos módulos de processamento de dados, *left panel*.
- Verde - área de apresentação de dados *right panel*.
- Laranja - área de apresentação de mensagens para o utilizador, *status panel*.
- Castanho - área de informação e interação com o servidor da base de dados, *toolbar*.
- Preto - Menu, área de configuração do ProcessUM e carregamento de ficheiros.

Em seguida são apresentadas as características da aplicação desenvolvida no âmbito desta dissertação.

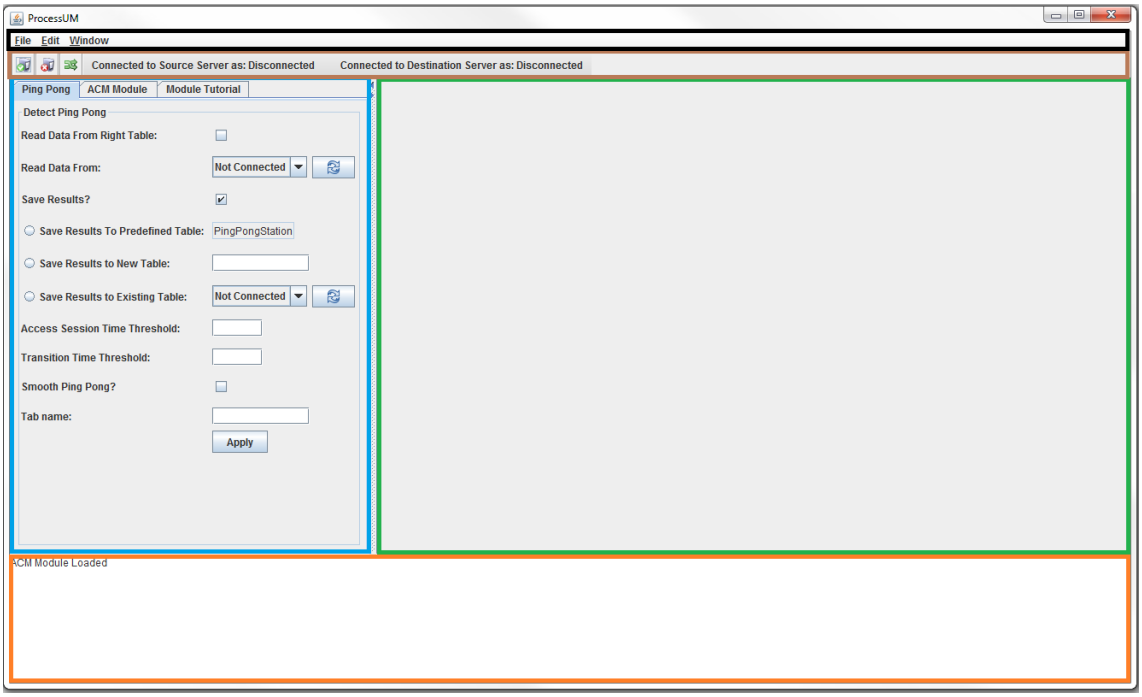


Figura 4.1: Interface gráfico da aplicação.

4.1 Ligação à Base de Dados

Na aplicação é dada a possibilidade de se realizar uma conexão à base de dados, para tal o utilizador tem de recorrer à *toolbar* delimitada a castanho na figura 4.1. No ProcessUM existe o conceito de dois servidores, um servidor que fornece os dados (*Source Server*) e um servidor que recebe os dados para os salvaguardar (*Destination Server*). Desta forma, quando é feita uma ligação à base de dados esta é sempre feita, teoricamente, a dois servidores. Teoricamente porque na prática o mesmo servidor pode ter a função de *Source Server* e *Destination Server* em simultâneo. A imagem 4.2 representa a *toolbar* e mais detalhe.

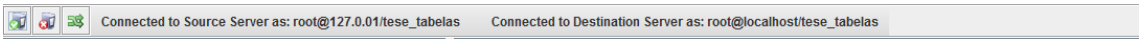


Figura 4.2: Toolbar do ProcessUM.

Como se pode observar na figura 4.2 existem três botões na *toolbar* tendo estes as seguintes funções.

- Botão à esquerda - Estabelecer uma ligação à base de dados.
- Botão ao meio - Desligar a ligação à base de dados.

- Botão à direita - Trocar os servidores, o *Source Server* passa a ser o *Destination Server* e vice-versa.

Em seguida existem duas áreas de texto que informam o utilizador sobre o estado da ligação aos servidores. Assim como o nome do utilizador e a base de dados a que estão ligados em cada servidor, como demonstrado na figura 4.2.

4.2 Leitura e Escrita de Dados

A aplicação disponibiliza, através da sua API, a possibilidade de escrever ou ler dados para processamento através de uma base de dados ou através de um ficheiro. Na leitura de dados de uma base de dados é possível, através da API, utilizar métodos que utilizam a *Java Persistence API* ou métodos que recorram à API JDBC.

A leitura de ficheiros necessita de seguir as seguintes regras.

1. Na primeira linha do ficheiro que contém os dados a importar tem que estar obrigatoriamente o nome da classe em Java que representa os dados a ler.

2. A classe que representa os dados a ler tem que existir e estar devidamente implementada.

3. Da segunda linha em diante vêm os dados a ler. Estes dados têm que estar separados por uma virgula. Geralmente denominados por *Comma-separated values*.

Na leitura de dados de um ficheiro e de uma base de dados, cada linha representa um objeto da classe definida no ponto 2 acima mencionado.

Já na escrita para um ficheiro a API do ProcessUM toma conta das regras acima enumeradas e o programador apenas necessita de passar a lista de dados a guardar sendo que a API encarrega-se do resto.

4.3 Apresentação de Dados

O ProcessUM na sua API disponibiliza um conjunto de métodos de interação com o utilizador que permitem, por exemplo, realizar as seguintes tarefas.

- Apresentar dados numa tabela.

- Esconder ou mostrar uma, ou mais, colunas da tabela com o resultado do processamento.
- Guardar o resultado do processamento num ficheiro.
- Enviar mensagens para a área de mensagens.
- Apresentar uma mensagem numa caixa de mensagens.
- Apresentar ao utilizador uma barra de carregamento, *loading*, com a finalidade de informar o utilizador que o programa se encontra ocupado com, pelo menos, uma tarefa.

A figura 4.3 ilustra o resultado do processamento de deteção da ocorrência de eventos ping pong e seu *smooth*. Nessa tabela estão apresentados todos os dados que são relevantes para a interpretação dos resultados gerados, neste caso, pelo módulo de deteção de eventos de ping pong.

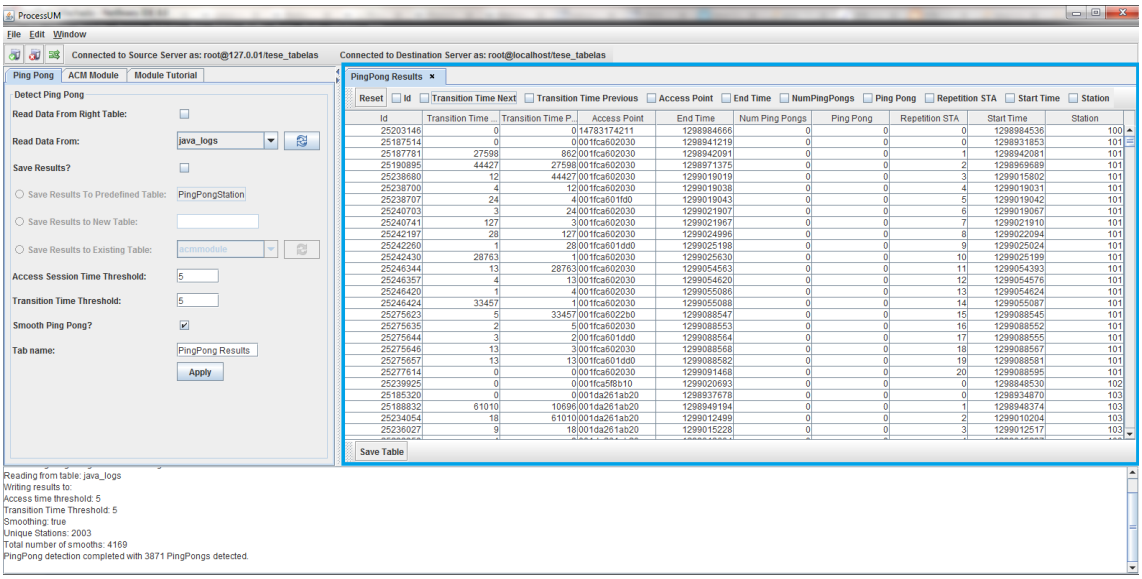


Figura 4.3: Apresentação dos resultados do para a deteção de eventos ping pong e seu *smooth*.

Caso o programador necessite de um tipo de apresentação de dados que não se enquadre em nenhum dos acima mencionados é possível criar o seu módulo de apresentação de dados e facilmente carregar o módulo na área de apresentação de dados, tal e qual faz para um módulo de processamento.

No painel de apresentação de dados também podem ser carregados módulos de processamento, isto é útil caso o utilizador necessite de ter dois módulos de processamento visíveis no ecrã simultaneamente.

Tanto o painel de módulos como o painel de apresentação de dados, como pode ser observado pela área delimitada a azul na figura 4.1, podem facilmente e sem qualquer tipo de problema conter mais que um módulo pois ambos os painéis suportam *tabs*.

4.4 Configurações e Funcionalidades

O menu de configurações pode ser acedido através do menu *Edit*. Após selecionar o menu *Edit* será apresentado um *dropdown* menu com a opção de selecionar as opções gerais ou as opções de configuração para os servidores.

Nas opções gerais é possível inserir o caminho para os ficheiros *MySQLDump.exe* e *MySQL.exe* bem como o valor de *batch inserts*, isto é, o valor total de inserções por pedido à base de dados, explicado mais em detalhe no capítulo 5. Os ficheiros *MySQLDump.exe* e *MySQL.exe* são necessários para ser possível importar e/ou exportar tabelas da/para a base dados.

Tudo isto pode ser observado na figura 4.4.

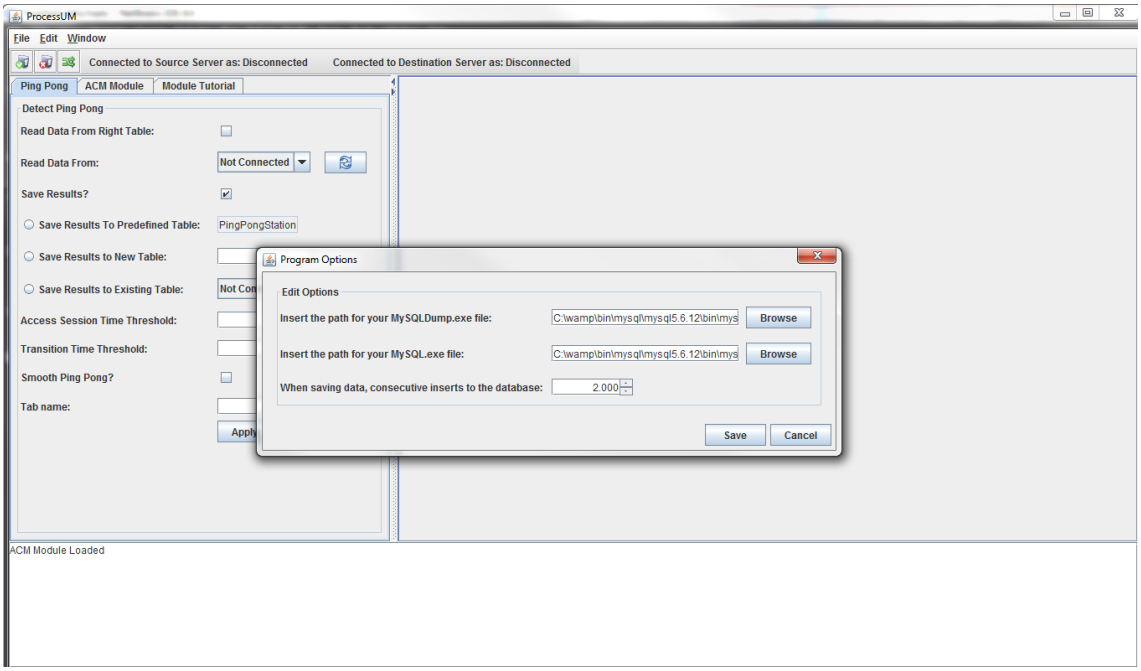


Figura 4.4: Opções gerais do ProcessUM.

As opções de configuração de servidores, figura 4.5, permite definir o endereço, porto, base de dados, nome de utilizador e a password para um dado servidor, *source* ou *destination*. Estas opções podem ser acedidas através

do menu *Edit* ou através da combinação das teclas *control + s* para o *Source Server* ou *control + d* para o *Destination Server*.

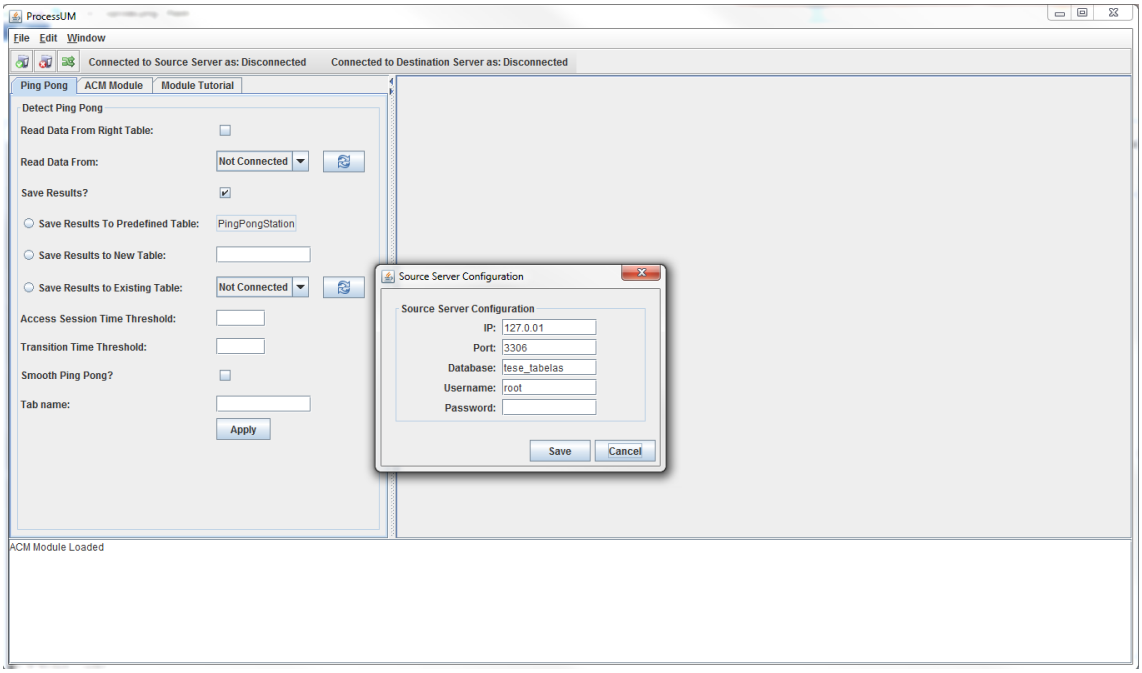


Figura 4.5: Opções de configuração de um servidor.

Um aspeto importante a referir nesta secção é o facto dos dados de configuração do ProcessUM serem guardados num objeto Java do tipo *Preferences* que por sua vez é guardado na árvore de registo do sistema operativo. Um outro aspeto importante a ressalvar é o facto da password não ser guardada em qualquer parte pela aplicação, à exceção da memória no caso da ligação estar ativa. Esta decisão foi tomada de forma a manter o mais segura possível a password. Inicialmente pensou-se em encriptar a password mas esta opção voltava a trazer o problema de onde guardar a password de encriptação de forma igualmente segura. Assim, optou-se por ser sempre pedida a password quando esta for necessária e apenas guardar o endereço, porto, a base de dados e o nome de utilizador.

Uma outra funcionalidade do ProcessUM é permitir exportar e/ou importar tabelas da/para a base de dados. Estas opções podem ser acedidas pelo menu *File* e a figura 4.6 demonstra a janela de opção para importar um tabela. Como se pode observar, é necessário pressionar o botão do servidor para o qual se pretende importar a tabela.

Ao exportar uma tabela o processo é semelhante. No entanto neste caso é pedido ao utilizador que selecione de onde e qual a tabela a exportar.

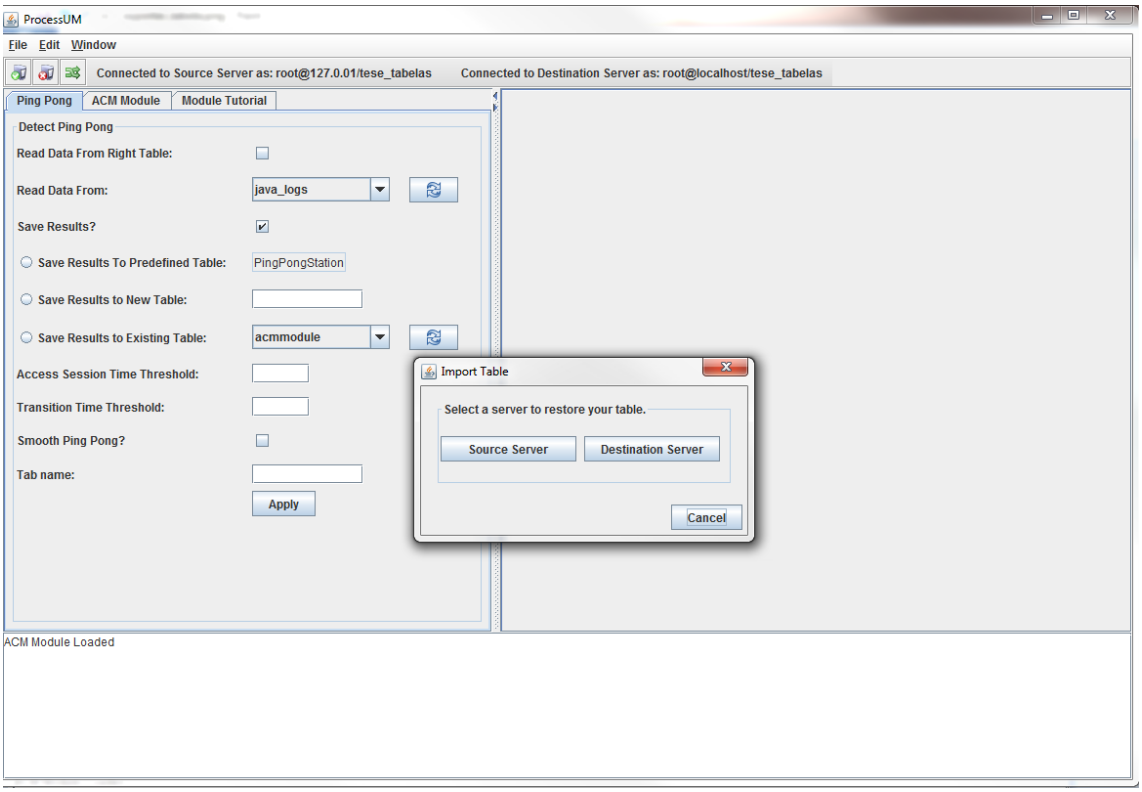


Figura 4.6: Menu para importar uma tabela.

Tanto para a importação como para a exportação de tabelas, assim que o utilizador, dentro do menu de importação ou exportação como representado pela figura 4.6 e 4.7, seleciona as opções que pretende em seguida é aberta uma janela, *File Chooser*, onde o utilizador pode navegar até à pasta fonte ou destino do ficheiro a utilizar para a sua operação. O ficheiro resultante ao exportar uma tabela é um ficheiro *.sql* já o ficheiro para importar uma tabela tem que ser um ficheiro *.sql*.

4.5 Módulo Ping Pong

O módulo Ping Pong é responsável pela deteção da ocorrência de eventos de ping pong em redes Wifi. Para tal recebe como input um conjunto de dados oriundos de uma tabela numa base de dados ou através de um conjunto de dados disponíveis no painel de apresentação de dados. Para o seu funcionamento, este módulo necessita dos seguintes dados como *input*.

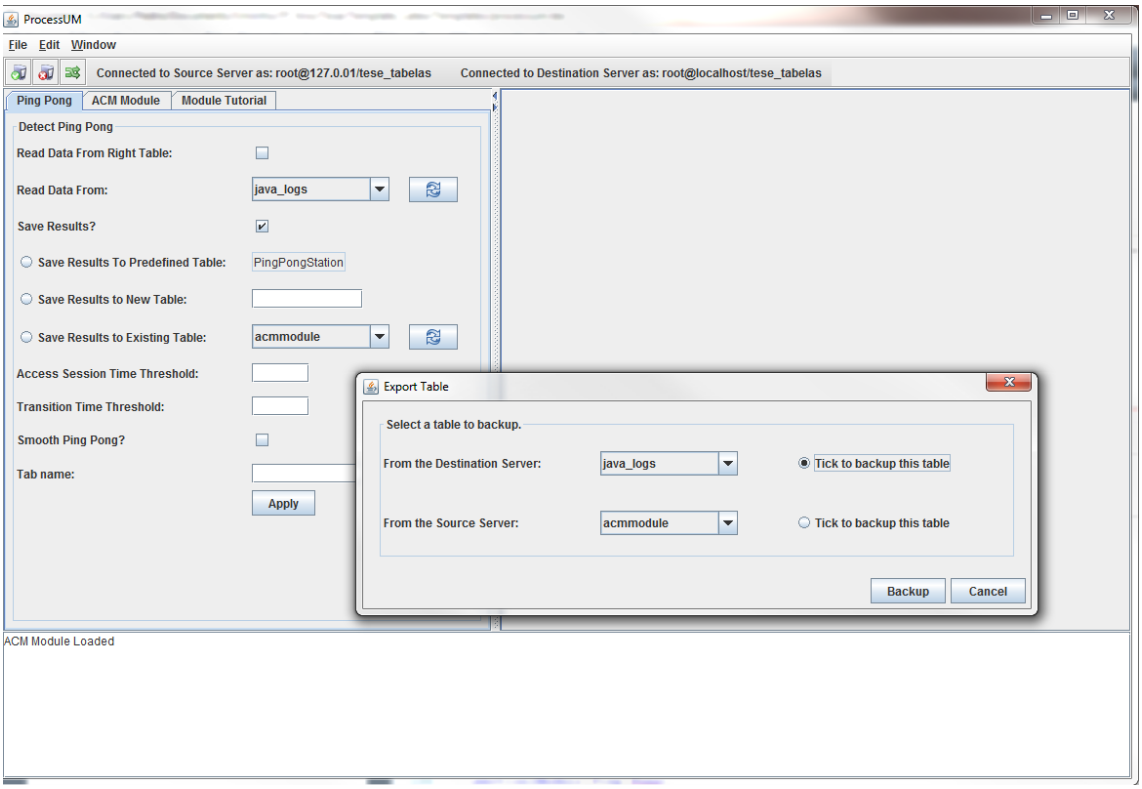
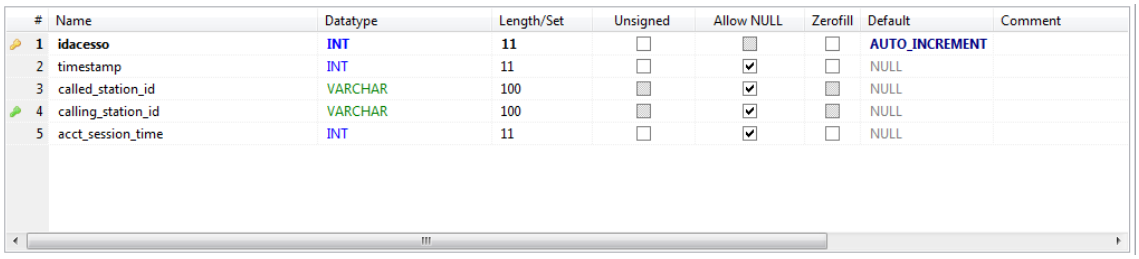


Figura 4.7: Menu para exportar uma tabela.

- Id - Identificador único.
- MAC Adress da estação.
- MAC Adress do *access point*.
- *Timestamp* de estabelecimento da ligação.
- Tempo de associação entre a estação e o *access point*.
- Tempo de transição entre uma de-associação e a seguinte associação, de uma estação e um *access point*.

Na figura 4.8 pode ser observada a estrutura da tabela na base de dados que serve como fonte, isto é, *input* de dados para o módulo de processamento ping pong. As variáveis de *input* são o tempo de associação entre a estação e o *access point* assim como o tempo de transição entre o momento em que ocorre a de-associação entre a estação e o *access point*. Antes de ser possível iniciar a deteção da ocorrência de eventos de ping pong é necessário

que o utilizador defina alguns campos. Campos esses que são a localização da fonte de dados, se pretende ou não guardar os resultados do processamento, definir as variáveis do tempo de associação e do tempo de transição entre associações e, finalmente, se deseja realizar um segundo processamento nos dados resultantes da deteção de ping pong. Este segundo processamento é chamado de *smooth*.



#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default	Comment
1	**idacesso**	**INT**	**11**	☐	☐	☐	**AUTO_INCREMENT**	
2	timestamp	INT	11	☐	☑	☐	NULL	
3	called_station_id	VARCHAR	100	☐	☑	☐	NULL	
4	calling_station_id	VARCHAR	100	☐	☑	☐	NULL	
5	acct_session_time	INT	11	☐	☑	☐	NULL	

Figura 4.8: Estrutura da tabela que contém os dados a processar pelo módulo ping pong.

O *smooth* consiste em detetar e anular os casos em que existe um elevado e anormal numero de ocorrências de eventos de ping pong num espaço de tempo muito curto uma vez que estes casos, tipicamente, não indicam a ocorrência de movimento da estação.

Ao iniciar o processamento, o módulo informa o utilizador, através da área de informações (área delimitada a castanho na figura 4.1) dos parâmetros selecionados. No final do processamento o módulo informa o utilizador que o processamento acabou e qual o resultado da sua computação.

Assim que o módulo ping pong finalizar o seu processamento este apresenta os dados ao utilizador no painel de apresentação de dados, como ilustrado na figura 4.3.

Capítulo 5

Implementação da Solução

5.1 Arquitetura do Software

Na implementação da solução, a linha de pensamento foi seguir o modelo MVC (*Model-View-Controller*), adaptando-o ao caso em estudo, uma vez que esta arquitetura é bastante utilizada e útil no desenvolvimento de interfaces gráficos. Assim, este modelo consiste em dividir a parte lógica da parte gráfica da aplicação. O modelo (*Model*) consiste nos dados da aplicação, geralmente objetos que seguem o conceito POJO (*Plain Old Java Objects*), em que os dados podem ser provenientes de uma base de dados, se for o caso. A *View* pode ser um qualquer objeto de representação de dados, isto é, que faça parte do interface gráfico, como por exemplo uma tabela. O controlador (*Controller*) é responsável por fazer a ponte/interligação entre a *view* e o modelo, como pode ser observado, de um modo mais genérico, na seguinte imagem.

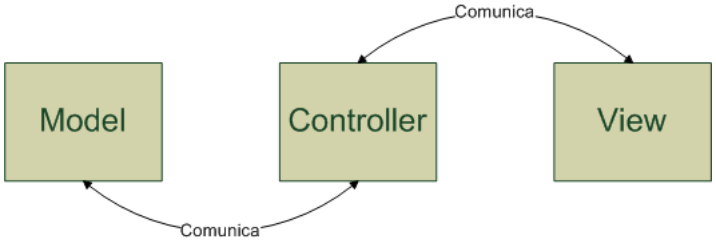


Figura 5.1: Diagrama de blocos da arquitetura MVC

Em seguida pode ser observado o fluxo de informação quando um utilizador interage com o modelo.

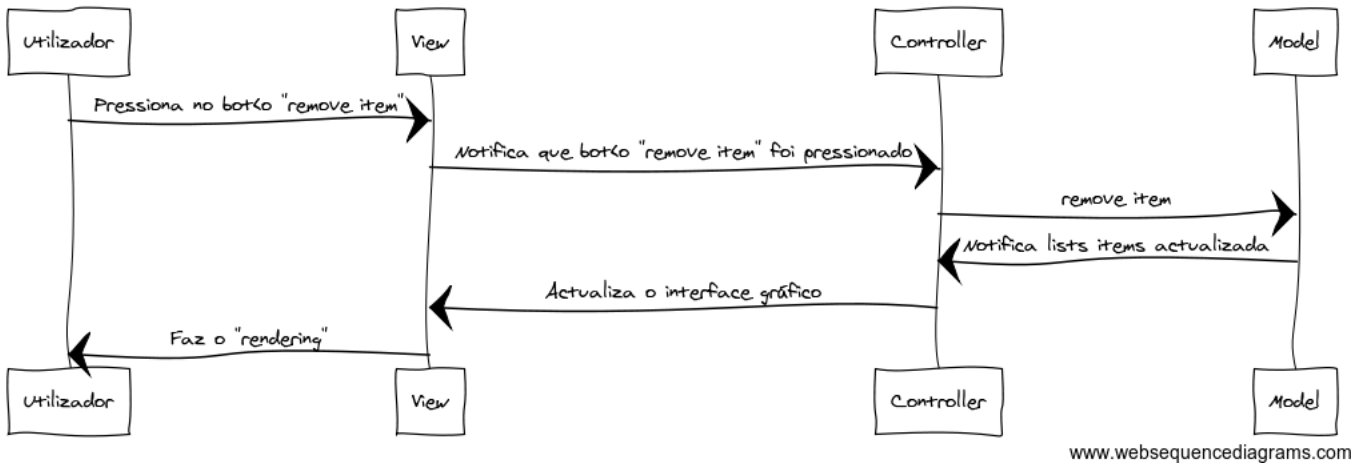


Figura 5.2: Fluxo de informação na arquitetura MVC

No entanto existe alguma controvérsia no seio da comunidade sobre onde realmente devem ser inseridos os ciclos de controlo e computação de uma aplicação, se no modelo ou no controlador. Aquando da existência de módulos, os ciclos de controlo e computação inerentes ao módulo podem estar, caso não sejam fornecidos pela API da aplicação, incluídos no próprio módulo ou no modelo do módulo, sendo esta uma decisão a tomar pelo criador do módulo. No módulo incluído nesta aplicação, os ciclos de controlo e computação encontram-se no próprio módulo, conseguindo uma melhor separação entre os dados e os ciclos de controlo e computação. Na aplicação o controlador é a classe MainFrame.

Figura 5.3: Separação entre o Modelo e a *View*

As principais ideias no conceito do MVC são a reutilização de código e a separação de conceitos. Isto é, se for necessário alterar algo no interface gráfico apenas é necessário alterar na *view*. O mesmo acontece no caso de ser necessário alterar algo na parte dos ciclos de controlo e computação em que apenas é necessário alterar no modelo.

5.1.1 Diagrama de Classes

A aplicação está dividida em três *packages*, model, view e API, como ilustrado na seguinte imagem.

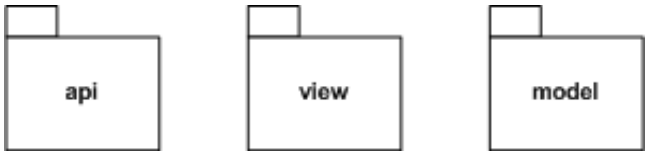


Figura 5.4: Os três principais *packages* que formam a aplicação.

Dentro de cada *package* existe um conjunto de classes. No caso do *package* model existe o seguinte diagrama de classes, simplificado de forma a poder ser de fácil leitura. No entanto em anexo podem ser consultados todos os diagramas no formato completo, assim como um diagrama de classes da aplicação no seu global.

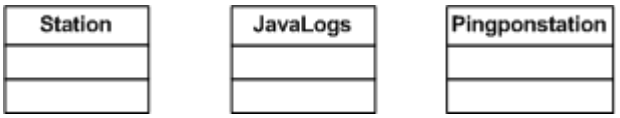


Figura 5.5: Diagrama de classes do *package* model.

Como se pode constatar na figura 5.5, não existe qualquer ligação/comunicação entre as diferentes classes. Isto porque, cada uma é independente de si e representa um tipo de informação específica para cada um dos módulos. Neste caso a classe Station representa o modelo de um terminal móvel (um computar, smartphone, entre outros), a classe JavaLogs representa o modelo ORM de uma tabela de uma determinada base de dados (recorrendo ao JPA) e o mesmo acontece com a classe Pingpongstation.

No *package* api tem-se o seguinte diagrama de classes, também ele simplificado.

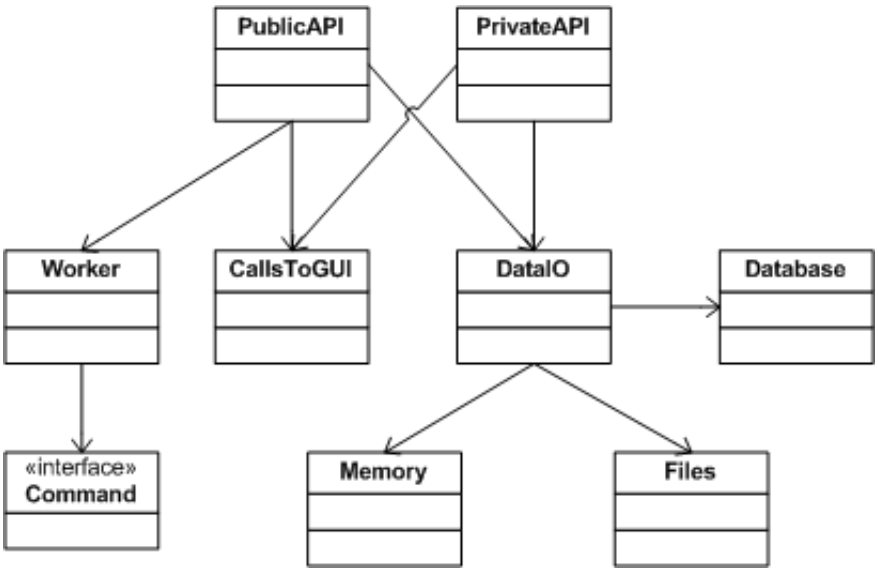


Figura 5.6: Diagrama de classes do *package* api.

Como se pode observar na figura 5.6 existem duas APIs, sendo uma pública e a outra privada. A API privada é a API interna do ProcessUM que contêm métodos internos para o bom funcionamento do ProcessUM e que por sua vez não são necessários para o desenvolvimento de módulos de processamento. Já a API pública é a API ao qual os módulos devem ter acesso e contém os métodos necessários para o desenvolvimento de módulos, como por exemplo executar instruções numa base de dados.

E por fim, no *package* view tem-se o seguinte diagrama de classes, também ele simplificado.

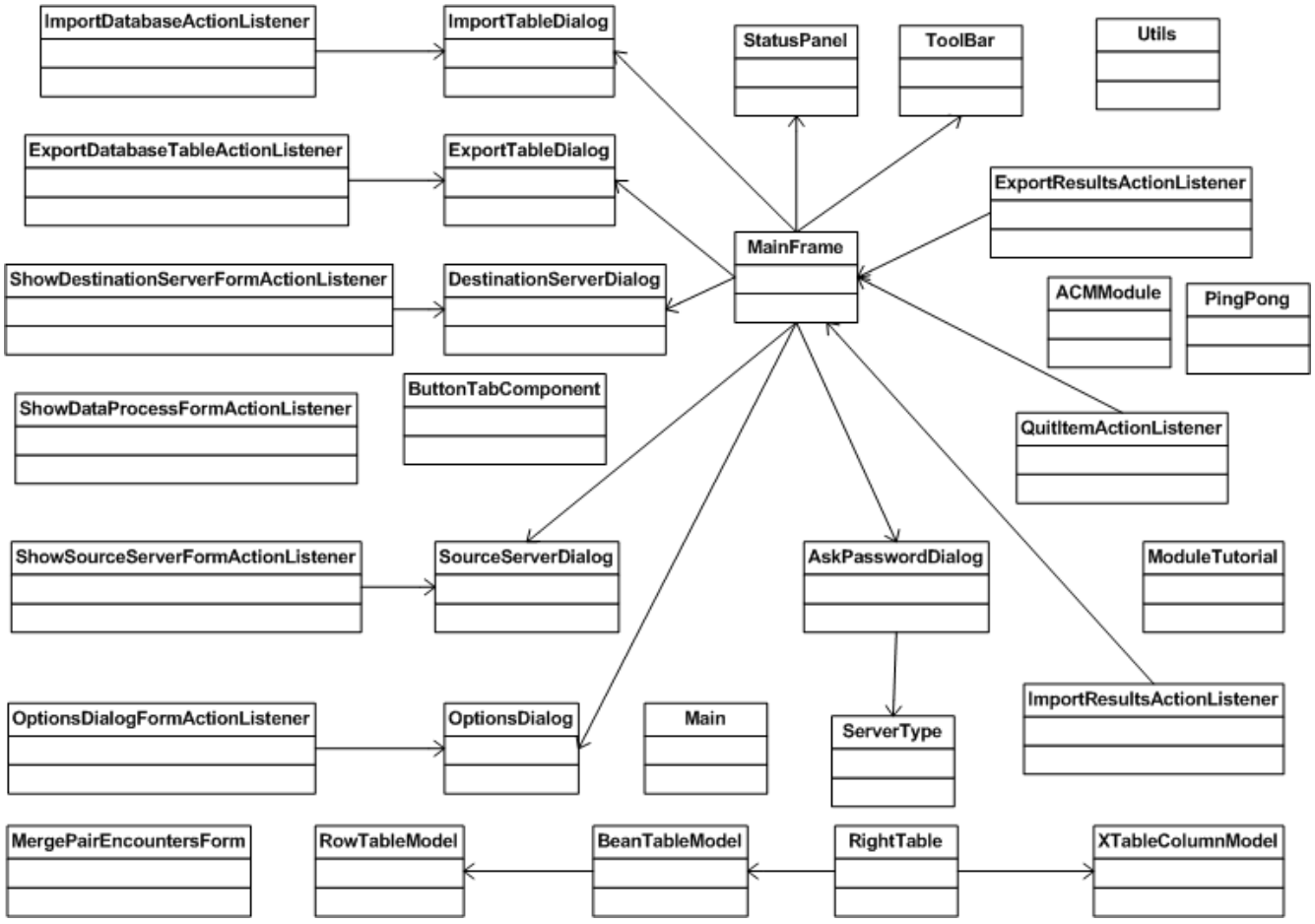


Figura 5.7: Diagrama de classes do *package* view.

Como pode ser observado pela figura 5.7, a classe MainFrame, como dito anteriormente, é o controlador no modelo MVC. Isto é, recebe pedidos e encaminha eventos no interface gráfico. No entanto é preciso notar que algumas classes, como por exemplo a classe RightTable, não aparentam, segundo a figura 5.7, ter qualquer ligação com o controlador. No entanto elas existem e tal deve-se ao facto de essas ligações serem feitas através de um *Event Bus*, que é explicado no que consiste na secção seguinte.

5.1.2 Biblioteca RetroEventBus

O RetroEventBus é uma biblioteca que foi, também ela, desenvolvida durante a dissertação de modo a facilitar a comunicação entre os diferentes componentes da aplicação. O RetroEventBus recorre à biblioteca *Guava*, da Google, que implementa um EventBus bastante customizável.

O Event Bus fornece comunicação entre diferentes objetos segundo a premissa do "evento-subscritor"sem ser necessário que ambos os objetos se conheçam ou se registem entre si. O envio de eventos pelo bus pode ser feito pelo método *post(tipoEventoObjecto)* e é recebido pelo objeto que esteja registado no bus, com o método *register(this)*, e tenha subscrito a receção de eventos do mesmo tipo que *tipoEventoObjecto*. O exemplo da figura 5.8 exemplifica o que acabou de ser explicado.

```
public class Emissor {
    private final String mensagem;
    private final RetroBus bus;

    public Emissor() {
        mensagem = "Ola, mensagem enviada pelo Emissor";
        bus = BusService.getRetroBus();

        sendMessage();
    }

    private void sendMessage() {
        bus.post(mensagem);
    }
}

public class Receptor {
    private final RetroBus bus;

    public Receptor() {
        bus = BusService.getRetroBus();
        register();
    }
    private register() {bus.register(this);}

    @Subscribe
    public void onEvent(String msg) {
        System.out.println("Mensagem recebida do emissor: " + msg);
    }
}
```

Figura 5.8: Exemplo de subscrição e envio de eventos entre diferentes objectos, utilizando o RetroEventBus

Se o código da figura 5.8 for executado o output gerado será *Mensagem recebida do emissor: Ola, mensagem enviada pelo Emissor*. Isto porque como se pode observar na figura 5.8 na classe Emissor é guardada uma

referencia para uma instância da classe RetroBus. Aquando da construção do objeto Emissor é chamado o método privado *sendMessage* que é responsável por chamar o método *post* do objecto RetroBus. A classe Receptor recebe uma instância da mesma classe RetroBus e regista-se para receber eventos do tipo *String*. Um método é de receção de eventos se, e só se, tiver a anotação *Subscribe*, como na figura 5.8. Um método que subscreve eventos vai receber eventos apenas e só do mesmo tipo que está declarado no seu parâmetro, no caso da figura 5.8 é do tipo *String* mas pode ser de qualquer tipo definido. Se quiser receber todos os eventos gerado pode subscrever a eventos do tipo *Object*, como por exemplo *@Subscribe public void onEvent(Object obj)* . Em suma, para a subscrição de eventos um objeto tem que:

- Registar-se no bus, através do método *register()*
- Subscrever a um determinado tipo de eventos.
- O método que subscreve eventos tem que retornar *void*

Já para o envio de eventos é necessário apenas chamar o método *post(evento)* do bus.

Observando a figura 5.8 repara-se que é obtida uma instância do bus de forma estática através da classe BusService. Isto porque, um bus apenas recebe eventos que forem enviados na sua instância, isto é, existindo duas instâncias, busA e busB, do mesmo tipo de bus e se for colocado um evento no busA apenas os subscritores do busA o vão receber. É aqui que entram as funcionalidades RetroEventBus em relação ao EventBus. Funcionalidade essas que são as seguintes.

- Existência de três tipos de EventBus. O RetroBus, RetroSwingBus e o RetroAsyncBus.
- Cada bus é um objecto singleton.

A implementação do padrão *singleton*, força que apenas exista uma e uma só instância de uma dada classe no sistema. Ou seja, sempre que se receba uma instância para um dado bus essa é sempre a mesma. Quanto à existência de três tipos de bus esta está relacionada com as diferentes possibilidades em que se possam aplicar um bus e também com a possibilidade dada pelo EventBus, que permite definir um *Executor* na construção de um EventBus. Assim são passados três diferentes *Executor*s para cada um dos três tipos de EventBus disponibilizados pelo RetroEventBus, o RetroBus, RetroSwingBus e o RetroAsyncBus.

Tomando o exemplo da figura 5.8, no RetroBus o método *sendMessage* e o método *onEvent* são ambos executados na mesma thread, ou seja, a thread que chama o método *sendMessage* é a mesma que vai tratar o método *onEvent*. Já no RetroSwingBus o método *onEvent* é sempre executado na *Event Dispatch Thread* (EDT), que é nada mais nada menos que a thread responsável por atualizar e/ou interagir com componentes da interface gráfica, independentemente da thread que chamou o método *sendMessage*. E por fim, o RetroAsyncBus, como o nome indica, é o bus indicado para eventos assíncronos. Isto é, o método *onEvent* é sempre tratado numa nova thread independentemente da thread que chamou o método *post*.

Com estes três tipos de RetroEventBus é possivel ter um sistema multi-thread de gestão de eventos sem que o utilizador da biblioteca necessite de andar a criar e verificar thread. No entanto é da responsabilidade do utilizador da biblioteca implementar os seus *event handlers* de forma correta para evitar situações típicas de má utilização de ambientes de multi-threading, como por exemplo *deadlocks*.

5.1.3 SwingWorker

Em Java Swing a thread principal tem o nome de *Event Dispatch Thread* (EDT) e apenas esta thread deve realizar operações no interface gráfico. Quando se pretende realizar uma tarefa que pode demorar um certo período de tempo esta não deve ser realizada na EDT. Caso contrário a EDT fica ocupada na realização da tarefa que por sua vez vai originar a que o interface gráfico deixe de responder ao utilizador. Tal resulta numa má experiência por parte do utilizador. Uma forma de solucionar este problema é correr a tarefa numa nova thread, libertando assim a EDT para manter o interface gráfico disponível ao utilizador, proporcionando assim uma melhor e mais fluída utilização do software.

Para criar uma nova thread, para correr a tarefa pretendida, pode-se recorrer à biblioteca RetroEventBus, mais concretamente através de um RetroAsyncBus, ou recorrendo a um SwingWorker. O SwingWorker é uma classe abstrata que permite, com relativa facilidade, criar de uma nova thread e ir atualizando o interface gráfico à medida que for necessário.

No classe SwingWorker existe um método que é obrigatório implementar, *doInBackground()*. O que for executado dentro do método *doInBackground()* é executado numa thread à parte. No entanto existem mais dois métodos, com bastante utilidade, que podem ser implementados, sendo estes o método *process(List<V> chunks)* e *done()*. Ambos os métodos são executados na EDT, logo ambos os métodos podem, de forma segura, atualizar

ou interagir com o interface gráfico. O primeiro método, *process(List<V> chunks)*, tem como finalidade receber, na EDT, mensagens enviadas, através do método *publish(mensagem)* da thread que está a ser executada à parte (*doInBackground()*) e usar essas mensagens para, por exemplo, ir atualizando o utilizador sobre o estado da tarefa (uma barra de progresso, por exemplo). O segundo método, *done()* é executado assim que o método *doInBackground()* retornar, podendo neste caso, por exemplo, informar o utilizador do resultado final da execução da tarefa realizada.

A figura 5.9 exemplifica como criar e executar uma tarefa recorrendo a um SwingWorker.

```
public void createSwingWorkerAndDoSomeTask() {
    SwingWorker<Integer, String> worker = new SwingWorker<Integer, String>() {
        @Override
        protected Integer doInBackground() throws Exception {
            publish("Executing task 1");
            // run task 1
            publish("Task 1 executed.\nExecuting task 2");
            // run task 2
            publish("Task 2 executed.");
            return 0;
        }

        @Override
        protected void process(List<String> chunks) {
            // process the last received message from doInBackground()
            System.out.println(chunks.get(chunks.size() - 1));
        }

        @Override
        protected void done() {
            // task done!
            System.out.println("Task done.");
        }
    };
    worker.execute();
}
```

Figura 5.9: Exemplo da implementação e execução de um SwingWorker.

Como se pode observar na figura 5.9, existem dois parâmetros a definir, *SwingWorker<Integer, String> worker = new SwingWorker<Integer, String>() {}*, que correspondem ao tipo de dados a retornar do método

doInBackground(), que no caso da figura 5.9 é do tipo *Integer*, e o tipo de dados contidos na lista a receber pelo método *process(List<V> chunks)*, que no caso da figura 5.9 é do tipo *String*.

Caso pretenda recorrer à API do ProcessUM para a criação e execução de um SwingWorker tal é possível recorrendo ao interface Command. Deste modo, o programador apenas necessita de implementar os métodos do interface Command e passar esse objeto para a API, encarregando-se esta de tratar de tudo o resto. É apenas necessário ressalvar que existe uma ligeira diferença entre implementar manualmente a classe abstrata SwingWorker e implementar o interface Command. Essa diferença está ao nível das mensagens a enviar durante o processamento da tarefa no método *doInBackground()*. Apenas é possível enviar uma mensagem antes de iniciar e uma outra depois acabar o processamento da tarefa no método *doInBackground()*.

A figura 5.10 demonstra a implementação do interface Command. Após a sua implementação pode ser passado para a API da seguinte forma: *api.startWorker(new Example(), "Mensagem de Inicio", "Mensagem de Fim", new ObjectParameterToDoInBackground());*

```
public class Example implements Command {

    PublicAPI api = PublicAPI.getInstance();

    @Override
    public Command doInBackground(Object data) {
        // Run the task in a new thread.

        return null;
    }

    @Override
    public void process(List<String> chunks) {
        // Show the messages where you desire
        final String message = chunks.get(chunks.size() - 1);
        api.displayMessageBox(message, "Message");
    }

    @Override
    public void done() {
        // Task finished!
    }
}
```

Figura 5.10: Exemplo da implementação do interface Command.

5.2 API

A API pública contém um variado conjunto de métodos que englobam ligações e pedidos a bases de dados, escrita e leitura de ficheiros, execução de tarefas e interação com o utilizador.

5.2.1 Ligação à Base de Dados

Na API pública não estão disponíveis métodos para ligar ou desligar diretamente a uma base de dados. Esta decisão foi tomada de forma a evitar que quem tem acesso à API possa criar ligações indesejadas sem conhecimento do utilizador e também para abstrair o programador de criar e gerir ligações a uma base de dados. Assim, o utilizador da aplicação está sempre em total controlo sobre a qual servidor está conectado. Desta forma, os métodos de ligação à base de dados encontram-se apenas na API privada e interna do ProcessUM. No entanto na API pública existem métodos para aceder ao objeto *java.sql.Connection*, para o caso de SQL, ou ao objeto *javax.persistence.EntityManager*, no caso do JPA, que fornece acesso à ligação à base de dados.

É também importante referir que existem duas formas de ligar à base de dados, com a flag *rewriteBatchedStatements* como verdadeira, *rewriteBatchedStatements=true*, ou como falsa, *rewriteBatchedStatements=false*. Se for como verdadeira torna possível tomar partido do agrupamento de *queries* em uma só *query*. Eliminando assim grande parte do *overhead* na comunicação entre o ProcessUM e a base de dados que por sua vez leva a uma melhor performance, como se pode constatar no capítulo de testes.

Esta flag ganha bastante relevo na execução de *queries* do tipo *INSERT*, *DELETE* e *UPDATE* pois no caso de, por exemplo, se pretender realizar a inserção de cem mil registos na base de dados, *INSERT INTO tabela (nome) VALUES ("Pedro")*, se a flag *rewriteBatchedStatements* estiver como falsa serão enviados aproximadamente cem mil pedidos do tipo *INSERT* à base de dados. Caso a flag *rewriteBatchedStatements* estiver como verdadeira os pedidos serão agrupados em lotes e enviados assim que uma das seguintes condições for alcançada primeiro.

- O tamanho do pacote de rede chegar ao tamanho máximo definido pelo servidor de base de dados.
- O numero de *queries* chegar ao valor definido para o tamanho do lote, *batch size*

Assim, o pacote a enviar contém a *query* agrupada da seguinte forma: *INSERT INTO tabela (nome) VALUES ("Pedro"), ("Pedro"), ("Pedro"), ("Pedro"), ("Pedro"), ("Pedro")....* Assim reduz-se drasticamente

o numero de pacotes a enviar que por sua vez reduz drasticamente o *overhead* que por sua vez reduz o tempo de conclusão da operação.

A figura 5.11 representa o conteúdo do pacote enviado à base de dados e pode-se constatar que a ligação utilizada tem a flag *rewriteBatchedStatements* como verdadeira pois a *query* foi agrupada, como explicado anteriormente.

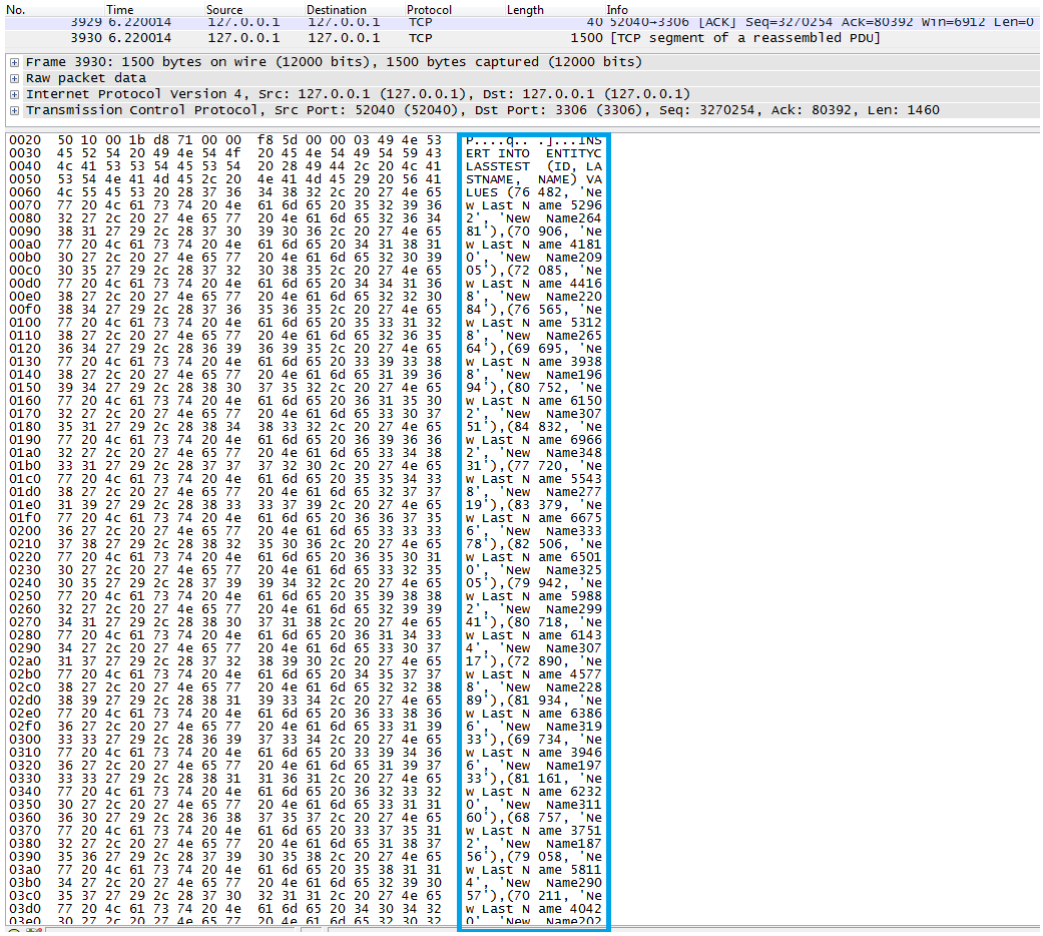


Figura 5.11: Conteudo do pacote para uma *query* do tipo *INSERT* com uma ligação à base de dados com a flag *rewriteBatchedStatements* como verdadeira.

SQL

Quanto ao SQL, a API fornece um variado conjunto de métodos. Este métodos permitem a execução de *queries* para:

- Aceder à ligação com a base de dados.

- Ter acesso a uma lista com o nome de todas as tabelas na base de dados.
- Executar *queries* do tipo *SELECT DISTINCT*.
- Executar *queries* do tipo *INSERT*, *UPDATE* e *DELETE* com o modo *rewriteBatchedStatements* ativo ou inativo.
- Criar uma tabela.
- Remover uma tabela.
- Limpar uma tabela.
- Criar uma *View*
- Ter acesso ao nome da base de dados.
- Aceder ao valor do lote para operações em lotes, que o utilizador definiu nas configurações do ProcessUM.

JPA

Na utilização do JPA, a API disponibiliza métodos para poder realizar as seguintes operações.

- Inserir um objeto numa tabela.
- Inserir uma lista de objetos numa tabela.
- Remover um registo de uma tabela, dada uma chave primária.
- Apagar um conjunto de registos da tabela.
- Eliminar um registo numa tabela.
- Remover um conjunto de registos de uma tabela.
- Aceder a um registo numa tabela.
- Saber o nome de uma tabela à qual corresponde uma *Entity Class*.

- Aceder à ligação com a base de dados, *EntityManager*.
- Executar uma *query* SQL do tipo *INSERT*, *UPDATE* e *DELETE*
- Receber uma lista com todos os registos de uma determinada tabela.

5.2.2 Memória, Leitura e Escrita de Dados

A API também oferece um conjunto de métodos que interagem com a memória RAM do computador, mais concretamente guardar uma lista de objetos em memória. Ter acesso a uma cópia (*unmodifiableList)* [20] da lista de objetos em memória que não pode ser alterada, isto é, apenas tem permissões de leitura. Assim como limpar a lista em memória e, finalmente, adicionar e remover um objeto à lista.

A grande vantagem de utilizar uma lista com apenas direitos de leitura é que esta pode ser apresentada ao utilizador e este não a pode alterar, pois só tem direitos de leitura.

Já quanto à escrita e leitura de dados, a API disponibiliza métodos de leitura e escrita de ficheiros *Comma-separated values* (*.csv)*.

5.2.3 Interação com o Utilizador

Quanto à interação com o utilizador, API fornece as seguintes funcionalidades, sendo que uma tab representa um componente.

- Apresentação de uma caixa com uma mensagem de erro, sucesso ou de vários tipos de interação com o utilizador.
- Criar uma tabela e adicionar a dita ao painel de apresentação de dados.
- Criar uma painel ao qual podem ser adicionadas componentes.
- Aceder à tab e/ou nome da tab ativa, isto é, a tab que o utilizador está a visualizar naquele preciso momento.
- Ter acesso a todos os componentes que se encontrem no painel de apresentação de dados.
- Aceder ao componente que se encontra numa dada posição.

- Enviar mensagens para a área de mensagens, área a laranja na figura 4.1.
- Mostrar ou esconder uma barra de progresso.
- Inserir uma mensagem na barra de progresso, por exemplo *"Loading..."*.

Um aspeto a ressalvar na tabela de apresentação de dados, mencionada acima no segundo ponto, é o facto de ser genérica. Isto é, aceita uma lista de um dado tipo de objetos e recorrendo à *Java Reflection*[1] acede aos dados através dos seus métodos *get*, constrói a tabela e apresenta os dados. O único requisito é que exista a classe do objeto na lista a apresentar e esta contém os métodos *get* definidos para os valores a apresentar. Por exemplo, a classe *Station* pretende apresentar o seu *mac address* e para tal basta que tenha definido o método *getMacAddress()*.

5.3 Módulos

Neste capitulo é discutido o módulo implementado e como criar e adicionar módulos ao ProcessUM.

Módulo Ping Pong

Este módulo tem como finalidade detetar a ocorrência de ping pongs num conjunto de dados de utilizadores de uma determinada rede Wifi. A figura 5.12 ilustra o módulo, delimitado a azul, carregado na aplicação.

Este módulo, como pode ser observado na figura 5.12 é constituído por uma série de componentes, desde *JTextField* a *JButton*. Os dados a analisar, isto é, os dados de input encontram-se numa base de dados MySQL a correr num determinado servidor. Para tal o módulo necessita de aceder a essa base dados. Para tal basta chamar o método *getConnectionSourceServer()* e/ou *getConnectionDestinationServer()* que se encontra na API pública, isto é, a API que é fornecida a cada módulo. Após receber a conexão à base de dados remota o módulo encontra-se em condições de interagir com o mesmo através de uma série de métodos disponíveis na API. Alguns desses métodos são os seguintes.

- getDatabaseTables
- getValueFromTable

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html

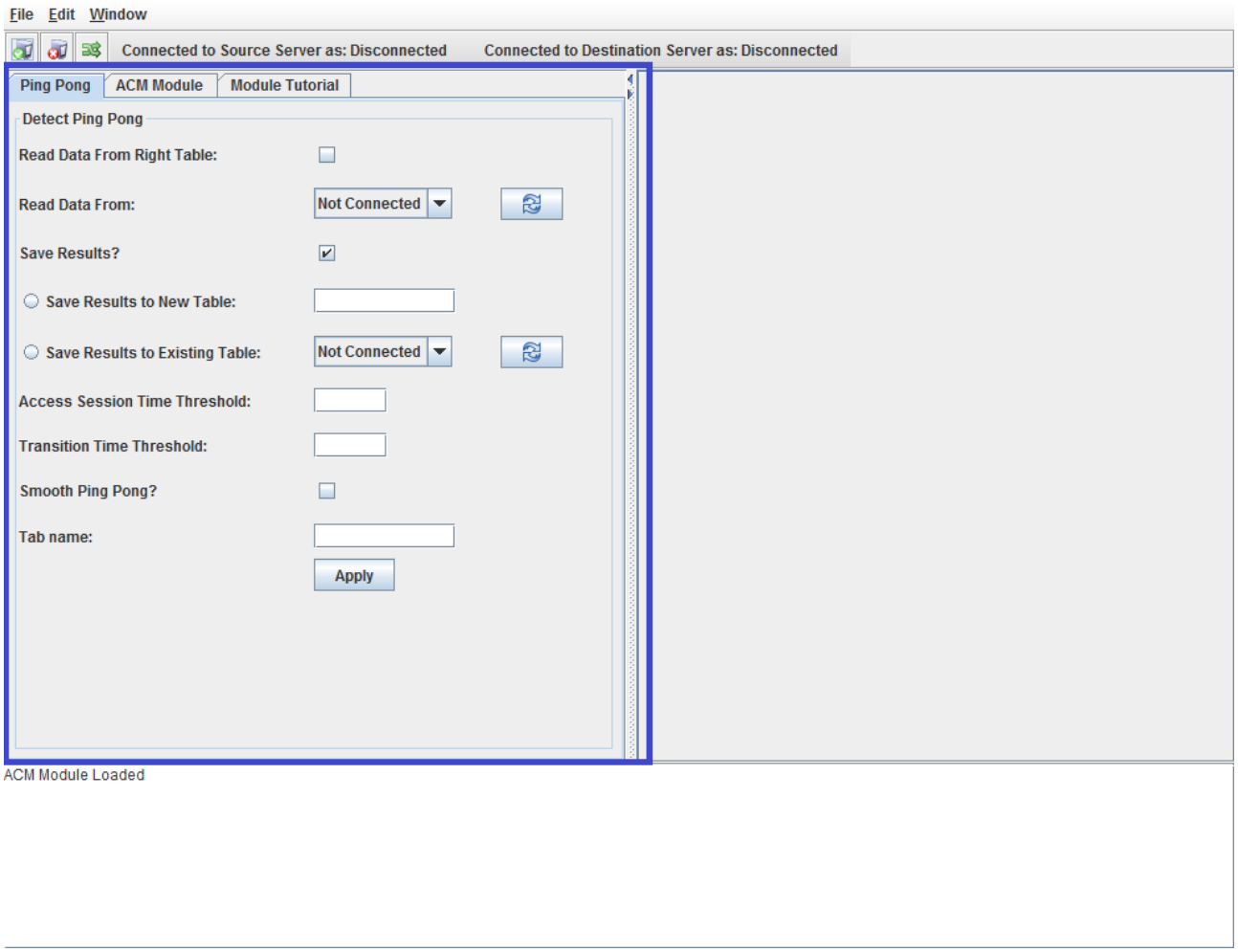


Figura 5.12: Módulo Ping Pong na aplicação

- insertObjectToTable
- updateTableRowFromObject
- deleteTableRowFromObject
- getSourceDatabase
- createDatabaseTableFromClass

Assim que que utilizador pressiona o botão *Apply* é iniciado o processo de deteção de eventos ping pong. Para

tal é criado um SwingWorker de forma a manter todo o interface gráfico responsivo. Todo o processo de deteção de eventos de ping pong, e posterior smooth se for caso disso, é feito numa nova thread à parte e posteriormente atualizado o interface gráfico através da thread principal do Java Swing, a *Event Dispatch Thread*.

Ping Pong Detection e Smooth

Tanto na deteção como no smooth da ocorrência de eventos de ping pong é utilizado um algoritmo [21] desenvolvido pelo aluno de doutoramento, pela Universidade do Minho, Karim Keramat Jahromi. Antes de aplicar o algoritmo é apenas necessário agrupar estação a estação pelo seu *mac address* e em seguida aplicar o algoritmo ilustrado pela figura 5.13 a cada estação.

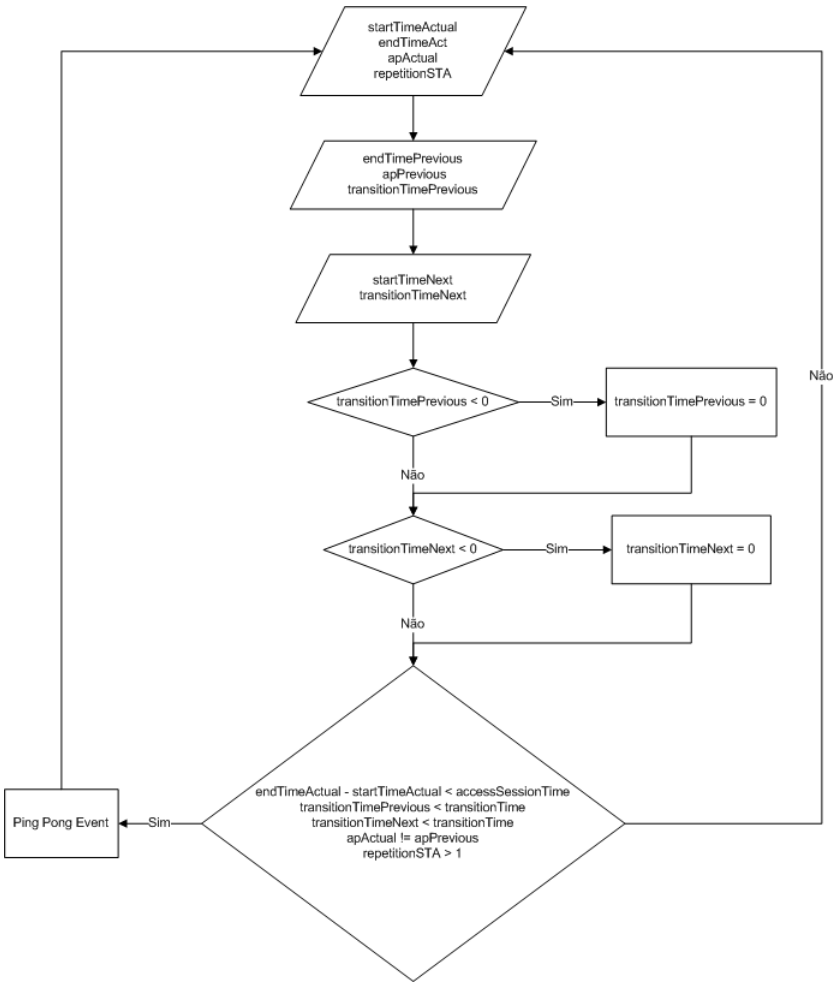


Figura 5.13: Algoritmo de deteção de eventos ping pong

Para a deteção de um evento de ping pong são necessários dados referentes a três ligações, a um *access point*. A ligação atual, que está a ser alvo de estudo quanto à ocorrência de um ping pong, a ligação imediatamente anterior à atual e a ligação imediatamente a seguir à ligação alvo de estudo. Os dados necessários para a ligação anterior são a momento, no tempo, em que se deu a desconexão entre a estação e o *access point* (endTimePrevious), o *mac address* do *acess point* (apPrevious) e o tempo de transição entre a de-associação do *access point* anterior para o *access point* atual (transitionTimePrevious). Para a ligação alvo de estudo é necessário o momento no tempo em que a estação se associou e de-associou ao *access point* (startTimeActual e endTimeAct, respetivamente) o *mac address* do *acess point* (apActual) e se é a primeira ocorrência daquela estação ou não (repetitionSTA). Já os dados necessários da ligação imediatamente a seguir é o momento no tempo em que foi feita a associação entre a estação e o *access point* e o tempo de transição entre a de-associação entre a estação e o *access point* da ligação alvo de estudo e a associação entre a estação e o *access point* da ligação imediatamente a seguir.

É no entanto importante referir que para cada conjunto de estações, o primeiro e ultimo registo são ignorados pois não existem registos anteriores ou posteriores para serem retirados os dados necessários.

Por razões de privacidade, aquando da deteção de eventos de ping pong para cada *mac address* da estação é atribuído um valor inteiro. Isto é, supondo que existe o *mac address* "00-B0-D0-86-BB-F7"este é substituído, por exemplo, pelo valor "100". Assim, sempre que era esperado aparecer o *mac address* "00-B0-D0-86-BB-F7"vai aparecer o valor de 100 para o *mac address*.

Para realizar o processo de smooth foi também utilizado um algoritmo [21] desenvolvido por Karim Keramat Jahromi.

O algoritmo de smooth de eventos de ping pong tem como objetivo substituir os *access points* que estão envolvidos em eventos de ping pong por um *access point* considerado dominante que é um dos *access points* de fronteira de uma série de eventos ping pong. Para tal é feita uma procura por eventos de ping pong consecutivos, para cada conjunto de eventos de ping pong consecutivos o *access point* dominante é aquele que não tem um evento de ping pong associado e que tem o maior tempo de sessão entre si e a estação, como ilustrado na figura 5.14.

Caso ambos os *access points* de fronteira tenham o mesmo valor para o tempo de sessão então a escolha recai sobre o *access point* ao qual a estação se associou mais vezes ou por períodos mais longos durante o intervalo de ping pong. Por fim, o algoritmo dita que todos registos aos quais não foram detetados eventos de ping pong são mantidos intactos.

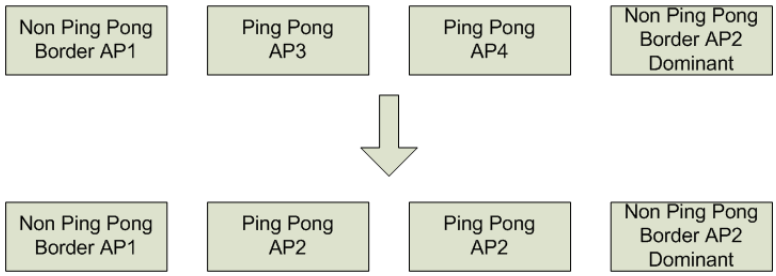


Figura 5.14: Ping pong smoothing. Onde AP1 e AP2 são os *access points* de fronteira, sendo AP2 considerado o *access point* dominante.

5.3.1 Adicionar Novo Módulo

Como mencionado várias vezes nesta dissertação, é possível a uma determinada pessoa criar um módulo e adicionar o mesmo à aplicação. Em seguida é explicado como criar um novo módulo e posteriormente como o adicionar à aplicação.

Criar Módulos

Para adicionar um novo módulo primeiro é necessário criar uma nova classe. Classe essa tem que estender a classe *JPanel* e o seu construtor tem que receber um objeto do tipo PublicAPI, como exemplificado na seguinte porção de código.

```
public class ModuleTutorial extends JPanel {
    // The class constructor
    public ModuleTutorial ( PublicAPI api ) {}
}
```

Para adicionar botões e/ou outros elementos à parte gráfica do módulo é necessário definir esses mesmos elementos e posteriormente definir a sua localização dentro do módulo. Para definir os botões e/ou outros elementos, primeiro é necessário definir e inicializar os mesmos.

```
public class ModuleTutorial extends JPanel {
    // Label
```

```
    private JLabel label ;
   // Botão
    private JButton button ;
   // Campo de texto
    private JTextField textField ;
   // Uma combo box
    private JComboBox comboBox;
   // Botão rádio
    private JRadioButton radioButton ;
   // ButtonGroup , util para agrupar botões rádio
    private ButtonGroup buttonGroup ;
   // Check box :
    private JCheckBox checkBox ;

   // The class constructor
    public ModuleTutorial ( PublicAPI api ) {
        button = new JButton (" Texto no butao ") ;
        label = new JLabel (" Texto na label ") ;
         textField = new JTextField (10) ; // Comprimento de, mais ou menos, 10 caracteres .
       comboBox = new JComboBox() ;
        radioButton = new JRadioButton (" Texto se necessario ") ;
        buttonGroup = new ButtonGroup () ;
        checkBox = new JCheckBox (" Texto se necessario ") ;
    }
}
```

Após esta etapa apenas fica a faltar colocar os elementos no interface gráfico. Para tal (ainda dentro do construtor do módulo) pode-se fazer, como exemplificado a seguir.

```
    public ModuleTutorial ( PublicAPI pubAPI ) {
```

```
        // Se pretender manter o padrão de design do módulo Ping Pong, pode-se adicionar um titulo à
            Border
        // E utilizar uma linha 5 pixeis dentro das bordas do módulo.
            Border innerBorder = BorderFactory.createTitledBorder("Module Tutorial");
            Border outterBorder = BorderFactory.createEmptyBorder(5, 5, 5, 5);
            // Agora apenas falta aplicar as bordas, chamando o metodo setBorder
            setBorder(BorderFactory.createCompoundBorder(outterBorder, innerBorder));

            // Chamar o metodo para colocar os elementos visíveis no interface gráfico
            layoutComponents();
    }

    // Neste exemplo é utilizado um GridBagLayout, que normalmente é suficiente na
    // maioria dos casos e funciona como que um eixo de coordenadas X Y, onde
    // o ponto 0,0 corresponde ao canto superior do esquerdo. Funcionando
    // de cima para baixo, à medida que se vai incrementando o valor de Y, e da
    // esquerda para a direita, à medida que se vai incrementando o valor de X
    // Para consultar uma lista de todos os layouts disponiveis pode consultar o
    // seguinte link: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

    private void layoutComponents() {
        // Fazer setLayout e passar um GridBagLayout
        setLayout(new GridBagLayout());
        // GridBagConstraints é o objecto ao qual é dado as coordenadas para cada elemento
        GridBagConstraints gc = new GridBagConstraints();
        // Caso não seja necessário preenchimento das células
        gc.fill = GridBagConstraints.NONE;
        // Caso pretenda especificar um alinhamento da célula.
        gc.anchor = GridBagConstraints.WEST; // Neste caso fica alinhada à esquerda
        // Especificar como distribuir o espaço horizontal
```

```
        gc.weighty = 2;
        // Especificar como distribuir o espaço vertical
        gc.weightx = 8;
        // E finalmente, posicionar os elementos
        // Começando pela posição 0,0 -> Top Esquerda
        gc.gridx = 0;
        gc.gridy = 0;
        // Addicionar o elemento
        add(label, gc);
        // Avançando uma posição (para a direita) na horizontal
        gc.gridx = 1;
        gc.gridy = 0; // mantem a posição segundo Y
        // Adicionar o elemento
        add(comboBox, gc);
        gc.gridx = 0;
        gc.gridy = 1;
        add(button, gc);
        gc.gridx = 1;
        gc.gridy = 1;
        add(textField, gc);
        // E por aí em diante ...
    }
```

Assim que o módulo estiver concluído fica apenas a faltar dar a conhecer à aplicação para que o carregue assim que arrancar. Para tal apenas é necessário ir à classe Main, que se encontra no *package* view, e registar o módulo na aplicação. É possível registar um módulo no painel do lado esquerdo, geralmente reservado para módulos, ou no painel do lado direito, geralmente reservado para a apresentação de dados e/ou resultados. No entanto fica ao critério do programador onde prefere carregar o seu módulo. Caso pretenda registar o módulo no painel do lado esquerdo basta adicionar o seguinte código, dentro do método *main*.

```
MainFrame.ModuleRegistry.INSTANCE.registerModuleForLeftPanel(new ModuleInterface() {
```

```
    @Override
    public String getTitle() {
        return "Titulo do Módulo"; // Aqui define o titulo do módulo a carregar
    }
    @Override
    public Component createModule(PublicAPI api) {
        return new NovoModulo(api); // Aqui cria uma nova instância do módulo a carregar
    }
});
```

Para o painel do lado direito basta adicionar o seguinte código, dentro do método *main*.

```
MainFrame.ModuleRegistry.INSTANCE.registerModuleForRightPanel(new ModuleInterface() {
    @Override
    public String getTitle() {
        return "Titulo do Módulo"; // Aqui define o titulo do módulo a carregar
    }
    @Override
    public Component createModule(PublicAPI api) {
        return new NovoModulo(api); // Aqui cria uma nova instância do módulo a carregar
    }
});
```

Caso seja necessário criar uma classe modelo, esta deve ir para o *package* model, por uma questão de boa organização, usabilidade e respeito para com os padrões definidos inicialmente. Como nota final, apenas realçar que não é obrigatório a codificação manual de um módulo. É perfeitamente possível utilizar um editor gráfico para o layout do módulo. A única parte em que é necessário codificar manualmente é no registo do módulo na classe Main.

JPA Entity Classes

As classes que representem uma entidade (*entity*), devem também elas ser colocadas no *package* model. Um classe entidade deve implementar *Serializable* e fazer *override* dos métodos *hashCode*, *equals* e, opcionalmente, do

método *Override*.

Para criar uma classe entidade podem ser utilizados os *wizards* dos IDEs (pelo menos é possível com Netbeans e Eclipse) ou definir manualmente. O método mais simples e rápido é utilizar o *wizard* do IDE. Pelo IDE existem dois modos de criar uma entidade, entidade vazia, *Entity Class*, (apenas com o esqueleto base para desenvolver uma entidade) ou uma entidade que represente uma tabela de uma base de dados, *Entity Classes From Databases*. Caso a tabela já exista, a melhor solução é, geralmente, utilizar a opção *Entity Classes From Databases*, pois deste modo a entidade já vem totalmente configurada.

No Netbeans pode ser feito do seguinte modo.

1. Clicar com o botão direito do rato no local onde se pretende adicionar a classe entidade.
2. Selecionar: New -> Entity Classes From Databases.
3. Caso não se encontre visível, selecionar: Other. Na nova janela que abre selecionar novamente Other e depois selecionar Entity Classes From Databases ou Entity Class, conforme o caso.
4. Se for selecionada Entity Class, a classe está criada e pronta a ser finalizada manualmente.
5. Caso contrário prossiga.
6. Ao selecionar Entity Classes From Databases, na nova janela configure uma conexão à base de dados pretendida.
7. Escolher na caixa do lado esquerdo a tabelas a persistir e pressionar em *Add* seguido de *Next*.
8. Retirar o visto da opção: Gerar anotações JAXB já que tal não é geralmente necessário, a menos que se esteja a trabalhar com serviços REST Jersey ou JAX-WS.
9. Pressionar *next*.
10. Selecionar java.util.list na *collection type* e pressionar *finish*.
11. A classe entidade foi criada.

Apesar de ser automática a geração das classes de entidade, convém saber o que significa cada uma das anotações geradas. Assim,

- @Entity - identifica que a classe é uma entity class.
- @Table(name = "nomeDaTable") - Esta anotação identifica o nome da tabela que a entidade representa
- @NamedQueries(@NamedQuery(name = "nome", query = "...query...") - São queries generalistas já pré carregadas que podem ser chamadas pelo seu nome, neste caso 'nome'.
- @Id - identifica como sendo a variável que corresponde à ID na tabela.
- @GeneratedValue(strategy = GenerationType.IDENTITY) - Indica que o provedor deve atribuir as chaves primárias para a entidade recorrendo uma coluna de ids na base de dados. [2]
- @Column(name = "id") - identifica a que coluna corresponde a variável em questão, neste caso representa a coluna id.

No entanto nos testes realizados obteve-se uma melhor performance caso os campos @GeneratedValue fossem alterados de @GeneratedValue(strategy = GenerationType.IDENTITY) para @GeneratedValue(generator="ORD_SEQ") e acrescentando, em seguida, a seguinte anotação @TableGenerator(name="ORD_SEQ", allocationSize=500) em que o valor do *allocationSize* deve ser pelo menos do mesmo tamanho que a *batch*, ou lote, a ser utilizado. Esta alteração faz com que seja criada, na base de dados, uma tabela auxiliar que tem como função gerar as IDs necessárias. Neste caso gera 500 IDs por lote, assim que são geradas 500 IDs o JPA seleciona-as e atribui uma ID a cada objeto a persistir *on the fly.*

Tanto no caso da anotação @Column(name = "id") como na anotação @Table(name = "nomeDaTable"), caso estas não existam o nome da coluna que uma determinada variável representa é a coluna com o mesmo nome da variável. O mesmo acontece para o nome da tabela, sendo que neste caso a tabela assume o nome da classe.

Caso tenha sido criada uma classe entidade através da seleção de uma tabela válida da base de dados, durante o *wizard*, não é necessário realizar o passo seguinte. No entanto é boa prática verificar se a classe entidade se encontra declarada no ficheiro *persistence.xml*.

No caso de ter criado a classe entidade manualmente ou através da criação de uma *Entity Class*, apenas falta configurar o JPA, isto é, configurar o ficheiro *persistence.xml* localizado na pasta src\main\resources\META-INF ou na pasta *Other Sources - src\main\resources - META-INF* casos seja utilizado um IDE.

[2] https://docs.oracle.com/javaee/7/api/javax/persistence/GenerationType.html#IDENTITY

Em seguida é apresentado o conteúdo do ficheiro *persistence.xml*. É preciso ter em atenção que caso seja pretendido utilizar o método de operações na base de dados em lotes é necessário adicionar *?rewriteBatchedStatements=true* no *url* da ligação. Caso não esteja presente *?rewriteBatchedStatements=true* no *url* da ligação não é um problema grave, apenas deixa de ser possível enviar pedidos em lotes uma vez que que a o provedor do JPA deteta automaticamente o valor da flag *rewriteBatchedStatements* e faz a gestão automaticamente sem necessidade de intervenção do utilizador ou do programador.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="TesePersistentUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <!-- List of Entity classes -->
    <class>model.JavaLogs</class>
    <class>model.Station</class>
    <class>model.Pingpongstation</class>
    <properties>
      <!-- Weaving -->
      <property name="eclipselink.weaving" value="static"/>
      <!-- SQL dialect / Database type -->
      <!-- <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/> -->
      <property name="eclipselink.target-database" value="MySQL"/>
      <!-- Create/update tables automatically using mapping metadata -->
      <property name="javax.persistence.jdbc.url"
          value="jdbc:mysql://localhost:3306/tese_tabelas?rewriteBatchedStatements=true"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <!-- Tell the JPA provider to, by default, create the table if it does not exist. -->
```

```
        < property  name="javax.  persistence  .schema-  generation  . database .  action "  value =" create "/>
         <!-  No  logging  ( For  development  change  the  value  to  "FINE")  ->
        < property  name="  eclipselink  .  logging  .  level  "  value ="OFF"/>
         <!-  Enable  batch  writing  ->
        < property  name="  eclipselink  . jdbc . batch -  writing  "  value ="JDBC"/>
         <!-  Batch  size  ->
        < property  name="  eclipselink  . jdbc . batch -  writing  . size "  value ="500"/>
      </  properties  >
    </  persistence  - unit >
</  persistence  >
```

Já existe uma configuração válida, no entanto é apenas necessário adicionar a classe entidade na lista de entidades. Para tal basta adicionar *<class>package.class</class>* após a declaração do provedor de JPA.

Em anexo a esta dissertação encontra um código mais completo para a criação de um módulo.

Capítulo 6

Resultados e Testes

6.1 Performance da Base de Dados

Na realização dos testes foram sempre utilizados 50 000 registos aleatórios representados por 50 000 objetos em *Java* agrupados num *ArraList*, isto é, cada registo dos 50 000 é representado por um objeto dos 50 000 objetos *Java*.

Os testes foram executados num computador portátil HP Pavilion dv6-6060ep. Alguns dos aspetos a ter em consideração neste computador são os seguintes

- Processador: Intel Core i7-2630QM Quad Core (Sandy Bridge) com 6 MB de cache.
- Memória RAM: 6 GB, DDR3
- Sistema operativo: Windows 7 Home Premium 64-bit
- Versão do servidor MySQL: 5.6.12 - MySQL Community Server
- Tabelas da Base de Dados em innodb, versão 5.6.12
- Variável, no ficheiro my.ini do MySQL, que define o tamanho máximo do pacote: max_allowed_packet = 1M
- Java IDE: Netbeans 8.0
- JPA ProvideR: EclipseLink (JPA 2.1)

- Jar do Driver JDBC: mysql-connector-java-5.1.24.jar
- O cliente e o servidor da base de dados a correr na mesma máquina.
- Foram utilizados vinte ensaios para cada teste, sendo posteriormente feita a média dos resultados obtidos para cada ensaio

Nos caso não otimizados foi utilizada a configuração por defeito tanto do driver JDBC como do JPA. Quanto à otimização foi, no caso do JDBC foram feitas as seguintes otimizações.

- Adicionada a flag *rewriteBatchedStatements=true*.
- Utilizados os métodos da API de operações em lotes, *batch*, com valores variáveis para o tamanho do lote.

No caso do JPA foram feitas as seguintes otimizações.

- Adicionada a flag *rewriteBatchedStatements=true*
- Selecionado o modo de *Weaving*, como estático.
- Selecionada a opção de *batcj writing*.
- Foi dado um valor para o tamanho da *batch*.
- Utilizados os métodos da API de operações em lotes, *batch*, com valores variáveis para o tamanho do lote.

Na figura 6.1 pode ser observado o teste realizado para testar a performance na persistência de objetos recorrendo ao Java JPA. Como se pode constatar a otimização teve um bom resultado. Chegando a ter mais de 5 vezes melhor performance que sem qualquer otimização. A diferença entre sem otimização e com o tamanho do lote de um valor é pequena, o que faz sentido pois ter o tamanho do lote como uma unidade é o mesmo que não utilizar lotes, uma vez que ao serem inseridos 50 000 registos, com um lote de tamanho 1, vão ser feitos na mesma 50 000 pedidos de inserção. Com um lote do tamanho de 100 e com um lote de tamanho 18 000 a diferença é praticamente nula. Isto pode ser, em grande parte dos casos, devido a um dos seguintes motivos.

- O tamanho do pacote a enviar excede o tamanho máximo definido no servidor de base de dados. Sendo que neste caso o pacote tem que ser repartido levando a que na prática um lote de 18 000 não seja transmitido apenas num pacote mas sim em vários, baixando assim a performance.

- O processamento necessário no cliente para gerar um lote de 18 000 não compensa com o que ganha ao reduzir o *overhead do pacote*.

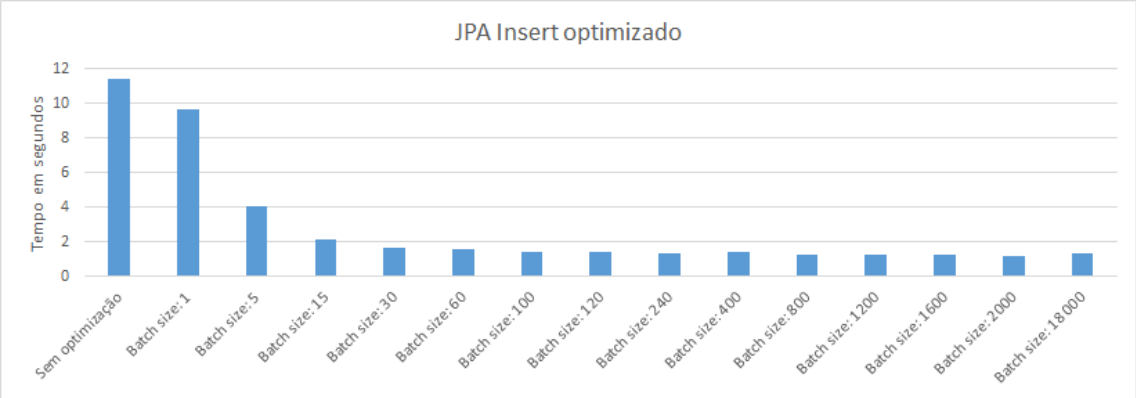


Figura 6.1: Resultado da inserção de 50 000 linhas na base de dados recorrendo ao JPA

Na figura 6.2, foi feito uma seleção de todos os dados, 50 000, contidos numa tabela. Neste caso não há grande diferença ao nível de otimizado vs não otimizado. Um outro fator a ter em conta é que o JPA perde por uma diferença significativa contra o JDBC, em ambos os casos (otimizado e não otimizado). A possível razão pode estar no facto de o JPA construir 1 objeto para cada registo selecionado na base de dados, neste caso resulta na criação de 50 000 objetos.

A figura 6.3 representa uma inserção de 50 000 registo numa base e dados recorrendo ao SQL, através do driver JDBC. Como se pode observar, a utilização de lotes faz uma enorme diferença. Basta observar a diferença entre um lote de tamanho 1 contra um lote de tamanho 5. A performance é aproximadamente duas vezes melhor entre um lote de tamanho 1 para um lote de tamanho 5. Ainda maior é a diferença para um lote de tamanho 100. O tempo começa a estabilizar por volta do lote de valor 100, o que leva a crer que esse seja um valor próximo do valor ótimo para a maquina em questão. O mesmo teste mas sem otimização pode ser observado na figura 6.4.

Como se pode observar na figura 6.4, existe uma diferença entre uma batch de valor 1 e as restantes. Isto deve-se ao facto de quando a flag *rewriteBatchedStatements* está desativada o driver JDBC, internamente, faz uma

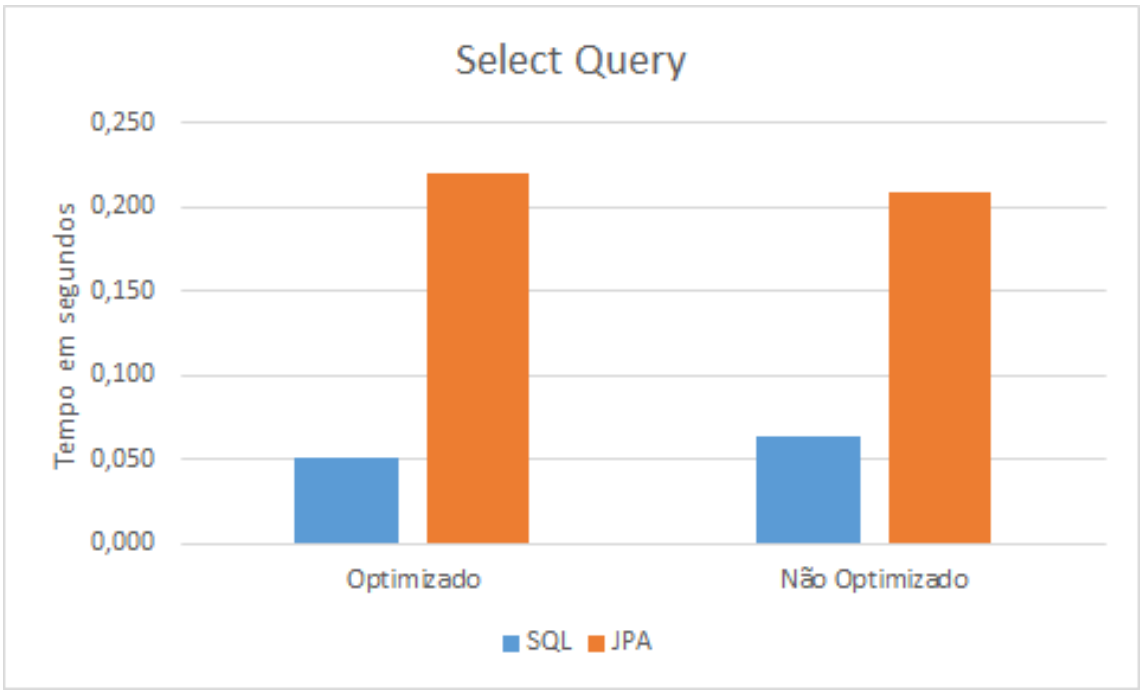


Figura 6.2: Resultado da seleção de todos os dados na tabela.

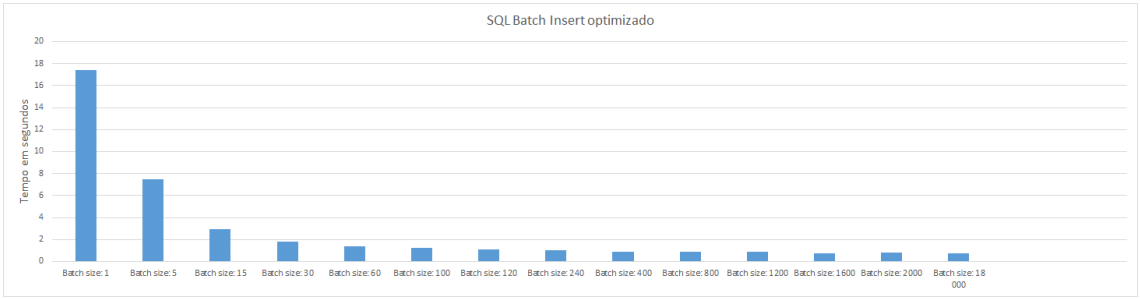


Figura 6.3: Inserção de 50 000 linhas na base de dados recorrendo utilizando SQL otimizado

emulação do envio de pedidos em lotes. No entanto continua a não ser eficiente, em comparação com os resultados da figura 6.3. A partir da batch de valor 5 o tempo de inserção estabiliza, aproximadamente, nos 7 segundos.

Na figura 6.5 a diferença entra a versão otimizada e a não otimizada é baixa, pelo que não aparenta haver grande diferença neste tipo de *queries*.

A figura 6.6 representa o resultado de quatro diferentes tipos de testes. É importante salientar neste caso um aspeto importante que levou à realização destes quatro testes. Quando se recorre aos métodos na API que utilizam o JPA é necessário, tal como nos métodos que recorrem ao JDBC, passar por parâmetro a ligação à base de dados, neste

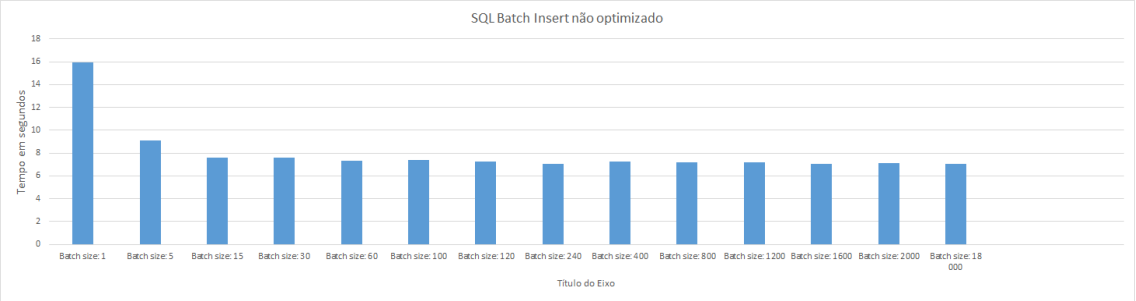


Figura 6.4: Inserção de 50 000 linhas na base de dados recorrendo utilizando SQL não otimizado

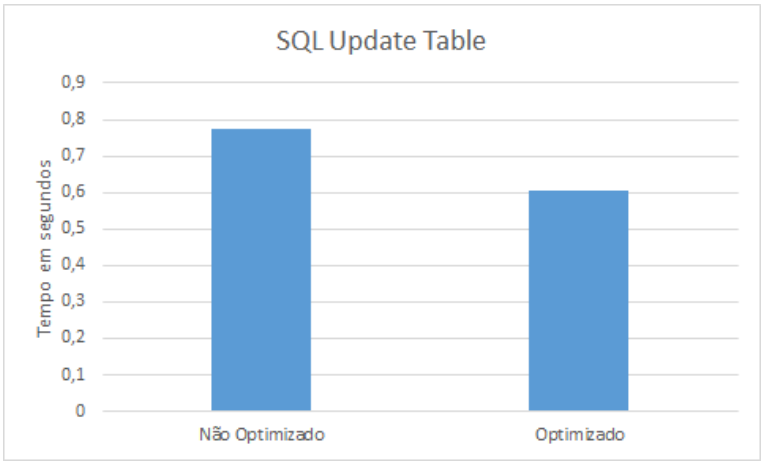


Figura 6.5: Atualização de um campo em todas as 50 000 linhas na base de dados recorrendo ao SQL

caso um objeto do tipo *EntityManager*. Sempre que é feita uma operação na base de dados estas só tomam efeito assim que for chamado o método *close()* no objeto *EntityManager*. Quando um *EntityManager* se encontra "aberto"este tem um contexto associado a si e todas as operações realizadas nesse contexto são aplicadas na base de dados assim que o *EntityManager* for fechado. Para passar objetos entre contextos, isto é, entre diferentes objetos *EntityManager* (diferentes porque têm contexto diferente) é necessário recorrer ao método *merge()* no *EntityManager*, sendo assim feita uma cópia do objeto do contexto "antigo"para o "novo".

Estes quatros testes representam a influencia de recorrer à cópia de um objeto de um contexto para um outro na performance da operação.

Começando pela observação da utilização de lotes ou não chega-se novamente à conclusão que melhora a performance a sua utilização. Sendo que neste caso a diferença não é muito considerável. Quanto à cópia de objetos entre contextos, como já era de esperar, esta tem um impacto significativo na performance. Sendo assim aconselhável

evitar a sua utilização.

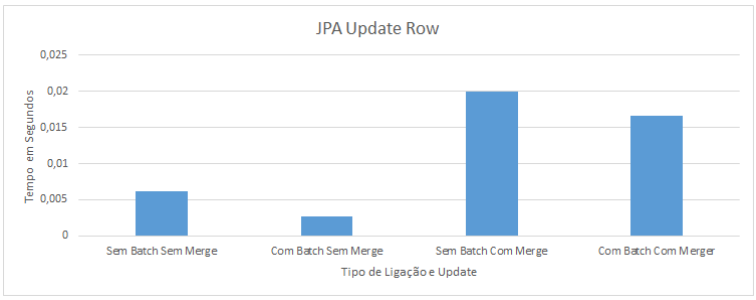


Figura 6.6: Atualização de um campo em todas as 50 000 linhas na base de dados recorrendo ao JPA

Quanto à figura 6.7, o teste consiste em eliminar um registo de uma tabela, por ID (em ambos os casos, JPA e SQL através do JDBC) e por um dado objeto. Neste teste o JPA, por objeto, ganha por uma boa diferença ao SQL. Sendo que aqui o fato de usar um *Object Relational Mapping* trás vantagens. No entanto a remoção por ID no JPA perde por uma diferença grande para o SQL/JDBC. Esta diferença pode ser explicada pelo facto de no caso do JPA remover por ID primeiro faz um search pela ID, em seguida constrói o objeto que representa a linha da ID e em seguida chama o método de remoção desse mesmo objeto.

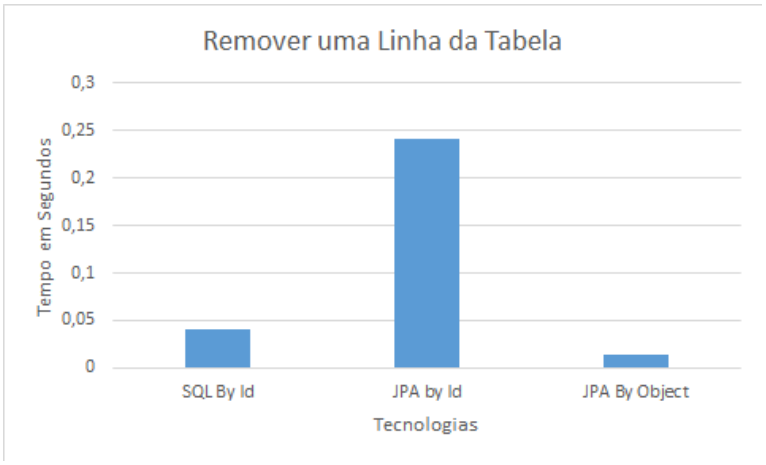


Figura 6.7: Remoção de um registo de uma tabela.

A figura 6.8 representa o teste de um Batch Delete, recorrendo ao JPA, quando o modo *rewriteBatchedStatements* não está ativo. O valor, o JPA quando configurado para utilizar operações por lote mas o *rewriteBatchedStatements* se encontra como *false* tenta, como o JDBC, emular esta característica daí a ligeira melhoria nos resultados obtidos para lotes maiores que um.

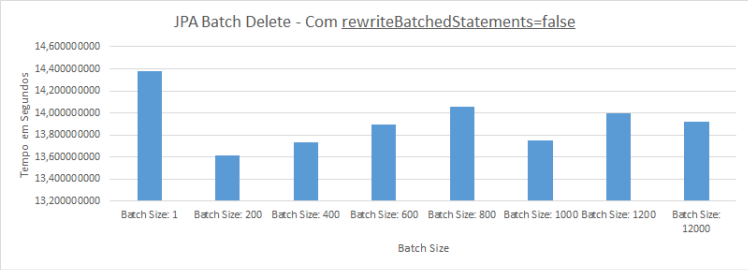


Figura 6.8: JPA Batch Delete sem otimização.

Por fim, a figura 6.9 representa o tempo para estabelecer uma ligação à base de dados, recorrendo ao JDBC e ou JPA. Mais uma vez o JDBC/SQL é mais rápido que o JPA. Isto deve-se ao facto de quando é estabelecida uma ligação à base de dados ser feito o mapeamento das tabelas, como explicado na secção *Object-relational mapping* desta dissertação.

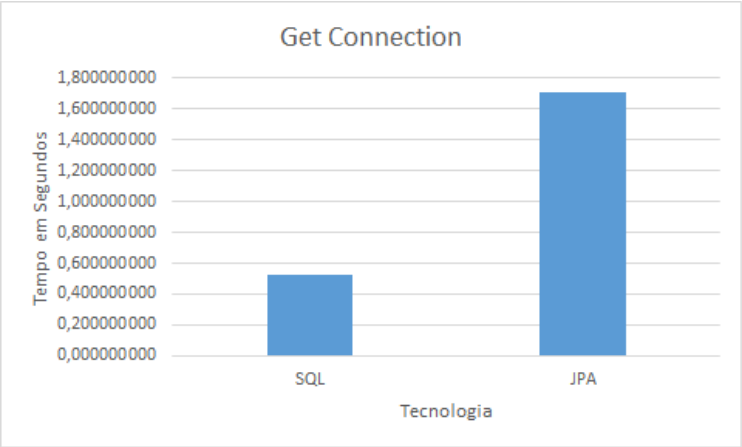


Figura 6.9: Realizar uma ligação à base de dados

6.2 Resultados Ping Pong

Através da figura 6.10 pode-se constatar que o tempo médio para a deteção de eventos de ping pong é de aproximadamente 16,6 segundos. Já a deteção e smooth de eventos de ping pong é de aproximadamente 17,5 segundos. Assim, de um modo geral o tempo que o smooth acrescenta à deteção de eventos de ping pong é, mais ou menos, de um segundo o que é aceitável.

Já na figura 6.11 é possível observar que o tempo total da deteção de eventos de ping pong e guardar os dados numa tabela na base de dados é de aproximadamente 27 segundos. Se a isto acrescentar também o processo de smooth o tempo total ascende a mais ou menos 28,5 segundos, o que mais uma vez volta a estar num valor aceitável e dentro do que foi observado na figura 6.10.

Como nota, fica o valor do tempo de inserção dos dados numa tabela que é de 9 segundos. O que mais uma vez vem de encontro ao valores observados pela figura 6.10 e 6.11.

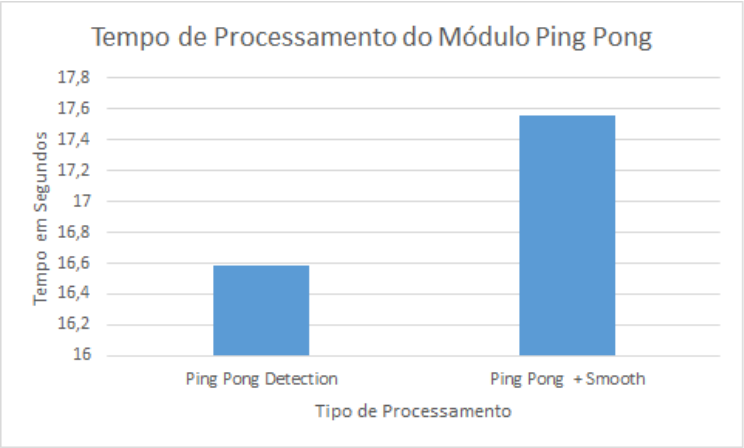


Figura 6.10: Tempo de processamento do módulo de ping pong, com e sem smooth.

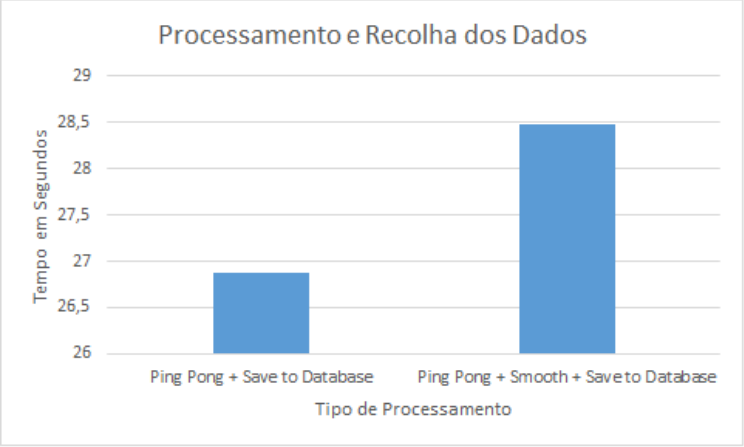


Figura 6.11: Módulo de ping pong, com e sem smooth. Tempo de processamento e recolha dos dados para uma base de dados.

Capítulo 7

Conclusões e Trabalho Futuro

Com o acentuado crescimento da quantidade de informação produzia a cada instante faz com que exista, cada vez mais, uma necessidade urgente em encontrar ferramentas que sejam capazes de processar essa informação de acordo com cada situação. Foi de forma a dar um maior poder de escolha, flexibilidade e customização que surgiu esta dissertação de desenvolver uma ferramenta flexível e customizável para o processamento dos mais variados tipos de dados.

Esta dissertação foi desenvolvida em proximidade com partes da tese de doutoramento do aluno, da Universidade do Minho, Karim Keramat Jahromi. Mais concretamente o desenvolvimento do módulo de deteção de eventos de ping pong o smooth. Uma vez que este módulo será útil para processamento dos dados a utilizar na sua pesquisa.

Um dos requisitos nesta dissertação era fornecer a aplicação de meios para que fosse possível trabalhar de forma direta, ou indiretamente, com uma base de dados. Este requisito foi implementado com sucesso assim como adicionada a possibilidade de leitura e escrita de ficheiros, a possibilidade de criar uma tabela de apresentação de dados completamente genérica, isto é, que aceita qualquer tipo de dados (objetos) para apresentar, dar a possibilidade a programadores desenvolverem os seus módulos e adicionar os mesmos à aplicação fornecendo para tal um conjunto de métodos numa API que abstraia o programador da necessidade de implementar métodos e conceitos não estritamente relacionados com a forma como o seu módulo vai processar a informação. Foi também desenvolvida uma biblioteca de auxilio que permite a fácil troca de informação entre objetos sem que exista um relacionamento direto entre ambos, possibilitando assim que diferentes componentes da aplicação funcionem e relacionem independentemente uns dos outros. Dando assim a vantagem de tornar mais acessível a manutenção do código e a troca de

blocos da aplicação, entre si ou por um novo bloco, com o mínimo esforço possível.

Desta forma é possível dizer que os objetivos desta dissertação foram alcançados com sucesso. Os testes realizados demonstram que existe, na grande parte das vezes, a possibilidade de o programador escolher entre recorrer a métodos que utilizem o driver JDBC ou Java JPA, podendo desta forma tirar o melhor partido que cada um tem ao seu dispor. Nomeadamente a facilidade e simplicidade de interação dada pelo JPA, ou a robustez e velocidade de execução fornecida pelo driver JDBC.

Como trabalho futuro a desenvolver nesta aplicação seria o carregamento de módulos durante a sua execução, ou seja, a possibilidade de um programador criar o seu módulo num ficheiro jar e em seguida adicionar o jar na aplicação enquanto esta se encontra a correr sem haver a necessidade de fechar, compilar e voltar a iniciar a aplicação, possivelmente seguindo a especificação *Open Service Gateway initiative*.[1]

É também importante ressalvar que ainda é possível desenvolver mais a integração com o JPA, as bases estão lançadas e a facilidade com que é possível trocar de provedor na aplicação sem necessidade de alterar o código fonte da aplicação torna este passo bastantes poderoso, uma vez que as oportunidades que os provedores dão nos seus mundos são enormes. Não estando assim apenas limitado aos métodos especificados pelo JPA. No entanto este passo deve ser tomado com as devidas precauções e com o devido conhecimento das vantagens e desvantagens que cada provedor pode trazer. Uma vez que depois para se fazer o caminho inverso pode ser um passo bastante penoso.

Também como trabalho futuro deste projeto é aumentar e fortalecer o conceito de componentes independentes entre si, para tal a utilização da biblioteca RetroEventBus será uma valiosa ajuda. Uma vez que esta biblioteca foi desenvolvida já na parte final da dissertação não houve tempo de refatorizar o código de alguns componentes de forma a que se tornem ainda mais modulares e independentes entre si.

[1] www.osgi.org

Bibliografia

[1] W. Lemstra, V. Hayes, and J. Groenewegen, *The innovation journey of Wi-Fi: the road to global success*. The Edinburgh Building, Cambridge CB2 8RU, UK: Cambridge University Press, 2010. ISBN 978-0-521-19971-1.

[2] C. Michael Marcus, J. Burtle, B. Franca, A. Lahjouji, and N. McNeil, "Unlicensed Devices and Experimental Licenses Working Group," Novembro 2002.

[3] D. Watkins, "Embedded WLAN (Wi-Fi) CE devices: Global market forecast." http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=9404. Acedido a: 18-12-2014.

[4] I. D. Corporation, "Worldwide WLAN market shows continued growth in second quarter of 2014, according to IDC." http://www.idc.com/getdoc.jsp?containerId=prUS25077714. Acedido a: 18-12-2014.

[5] I. Grupo de Redes de Comunicação, "Wireless LANs." http://pwp.net.ipl.pt/deetc.isel/pribeiro/site/disciplinas/TAR/acetatos/01%20-%20WLANv10.pdf. Acedido a: 24-03-2014.

[6] L.-H. Yen, T.-T. Yeh, and K.-H. Chi, "Load Balancing in IEEE 802.11 Networks," vol. 13, pp. 56–64, Janeiro - Fevereiro 2009.

[7] C. Systems, "How does RADIUS work?." http://www.cisco.com/en/US/tech/tk59/technologies_tech_note09186a00800945cc.shtml. Acedido a: 18-03-2014.

[8] A. Sevtsuk, S. Huang, F. Calabrese, and C. Ratti, *Mapping the MIT campus in real time using WiFi*. Hershey, PA: IGI Global, 2009.

[9] A. Sevtsuk and C. Ratti, "iSPOTS - How Wireless Technology is Changing Life on the MIT Campus," 2009.

[10] C. Song, Z. Qu, N. Blumm, and A.-L. Barabási, "Limits of Predictability in Human Mobility," *Science*, vol. 327, pp. 1018–1021, Fevereiro 2010.

[11] E. Oz, *Management Information Systems, Sixth Edition*. Boston, MA, United States: Course Technology Press, 2008. ISBN 1423901789, 9781423901785.

[12] Oracle, "Swing (Java™ Foundation Classes)." http://docs.oracle.com/javase/7/docs/technotes/guides/swing/. Acedido a: 10-09-2014.

[13] S. Suehring, *Mysql Bible*. New York, NY, USA: John Wiley & Sons, Inc., 1st ed., 2002. ISBN 0764549324.

[14] MySQL, "MySQL 5.7 Release Notes." http://dev.mysql.com/doc/relnotes/mysql/5.7/en/index.html. Acedido a: 22-10-2014.

[15] MySQL, "The Main Features of MySQL." http://dev.mysql.com/doc/refman/5.6/en/features.html. Acedido a: 22-10-2014.

[16] Oracle, "JDBC Overview." http://www.oracle.com/technetwork/java/overview-141217.html. Acedido a: 19-10-2014.

[17] T. E. Foundation, "ECLIPSELINK / FAQ / JPA." http://wiki.eclipse.org/EclipseLink/FAQ/JPA#What_databases_are_supported.3F. Acedido a: 08-08-2014.

[18] T. E. Foundation, "ECLIPSELINK / FAQ / NOSQL." http://wiki.eclipse.org/EclipseLink/FAQ/NoSQL. Acedido a: 08-08-2014.

[19] Hibernate, "What is Object/Relational Mapping?." http://hibernate.org/orm/what-is-an-orm/. Acedido a: 08-08-2014.

[20] Oracle, "Class Collections." http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableList(java.util.List). Acedido a: 29-10-2014.

[21] K. Keramat Jahromi, F. Meneses, and A. Moreira, "Impact of Ping-Pong Events on Connectivity Properties of Node Encounters," *Wireless and Mobile Networking Conference (WMNC), 2014 7th IFIP*, 2014.

Apêndices

Apêndice A

Criar um Módulo

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package view.modules;

import api.PublicAPI;
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
```

```
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JTextField;
import javax.swing.border.Border;

/**
 *
 * @author Pedro
 */

// The module must extends JPanel class
public class ModuleTutorial extends JPanel {
    // Declare you buttons, text labels, text fields and so on...

    // A label:
     private JLabel label;
    // A button:
     private JButton button;
    // A text field:
     private JTextField textField;
    // A combo box:
     private JComboBox comboBox;
    // A radio button:
     private JRadioButton radioButton;
    // A radio button, in most cases, should have a Button Group. A ButtonGroup:
     private ButtonGroup buttonGroup;
    // A check box:
     private JCheckBox checkBox;
     private PublicAPI api;
```

```
// Declare your constructor :
public ModuleTutorial ( PublicAPI pubAPI ) // it must be declared as final in order to be used
    inside the anonymous classes .
{
    // If we want to set a border and a tittle border for the panel we can do so like this :
    // Set a border tittle :
    Border innerBorder = BorderFactory . createTitledBorder ("Module Tutorial ");
    // Create an empty border (no tittle , just a line ) around the panel with 5 pixels distance
    // from the border being created and the border itself .
    Border outterBorder = BorderFactory . createEmptyBorder (5, 5, 5, 5);
    // Now it 's time to set the border
    setBorder ( BorderFactory . createCompoundBorder ( outterBorder , innerBorder ));
    api = pubAPI ;
    // If we want to set a minimum size for the panel we do so like this :
    // Create a dimension object :
    Dimension dim = new Dimension (400, 100);
    // set the preferred size , in this case it will be 400x100
    setPreferredSize (dim);
    // set the minimum size , in this case it will be 400x100.
    // the panel will never be smaller than 400x100 but it can be bigger .
    setMinimumSize (dim);

    // Initialize your variables
    button = new JButton ("Some text if you would like so");
    label = new JLabel (" Label 's text ");
    textField = new JTextField (10); // it accepts different paramenters , in this case it will be
        10 units long .
    comboBox = new JComboBox(); // A ComboBox requires a model , array of objects or a vector with
        the items to list
```

```
        radioButton = new JRadioButton("Some text if required"); // Also an icon, or other parameters
            are accepted
        buttonGroup = new ButtonGroup(); // no parameters are accepted here
        checkBox = new JCheckBox("A text if required."); // An icon, action and other parameters are
            accepted.

        // Action Listeners
        // If there is the need to add action listeners, usually to JButtons,
        // we do it using the addActionListener method. This method receives
        // an ActionListener interface as a parameter. We create an anonymous class
        // and code the actionPerfomed method. When a user clicks in the button
        // the actionPerformed method is the method that will be executed.
        // in this case we are printing some text to the StatusPanel using the
        // provided API.
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                api.printToStatusPanel("\nModule tutorial, someone clicked me!\n");
            }
        });
        // The addActionListener method is available to all the components listed above
        // and more!

        // Call the layout components:
        layoutComponents();

    }

    // Now it's time to place the components where we want them to be
    // As in this example it is used the GridBagLayout, usually the most commun
```

```
// for most cases, it works like an axis with X and Y coordenates.
// The 0,0 point is on the very top left corner of the component. So it works,
// from top -> down and from the left -> right.
 private void layoutComponents() {
     // Let's set the layout type, there are more then one layout avaibalbe
     // in Java, http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
      setLayout(new GridBagLayout());
       GridBagConstraints gc = new GridBagConstraints(); // class that specifies where goes what we
           want

     // We dont need any fill in our cells, so we set it to none.
     gc.fill = GridBagConstraints.NONE;
     // Specify to which side the component will stick to
     gc.anchor = GridBagConstraints.WEST; // stick to the left hand side
     // how much space it takes relatively to other cells, on the X axis.
     gc.weighty = 2;
     // how much space it takes relatively to other cells, on the Y axis.
     gc.weightx = 8;
     // Now it's time to start placing the components.
     // So, on the top left corner we place a label
     gc.gridx = 0;
     gc.gridy = 0;
     add(label, gc);
     // On the right, next to the label we have the combo box
     gc.gridx = 1; // we move one unit to the right
     gc.gridy = 0; // we keep on the same line
     add(comboBox, gc);
     // Under the label we put a button
     gc.gridx = 0; //we go back to the left side
     gc.gridy = 1; // we move one line down
```

```
        add( button , gc);
        // Next to the button we place the text field
        gc. gridx = 1; // we move one unit / cell to the right
        gc. gridy = 1; // we keep on the same line
        add( textField , gc);
        // And so on ... until we have placed all the buttons we need.
    }

}
```

Apêndice B

Dados complementares

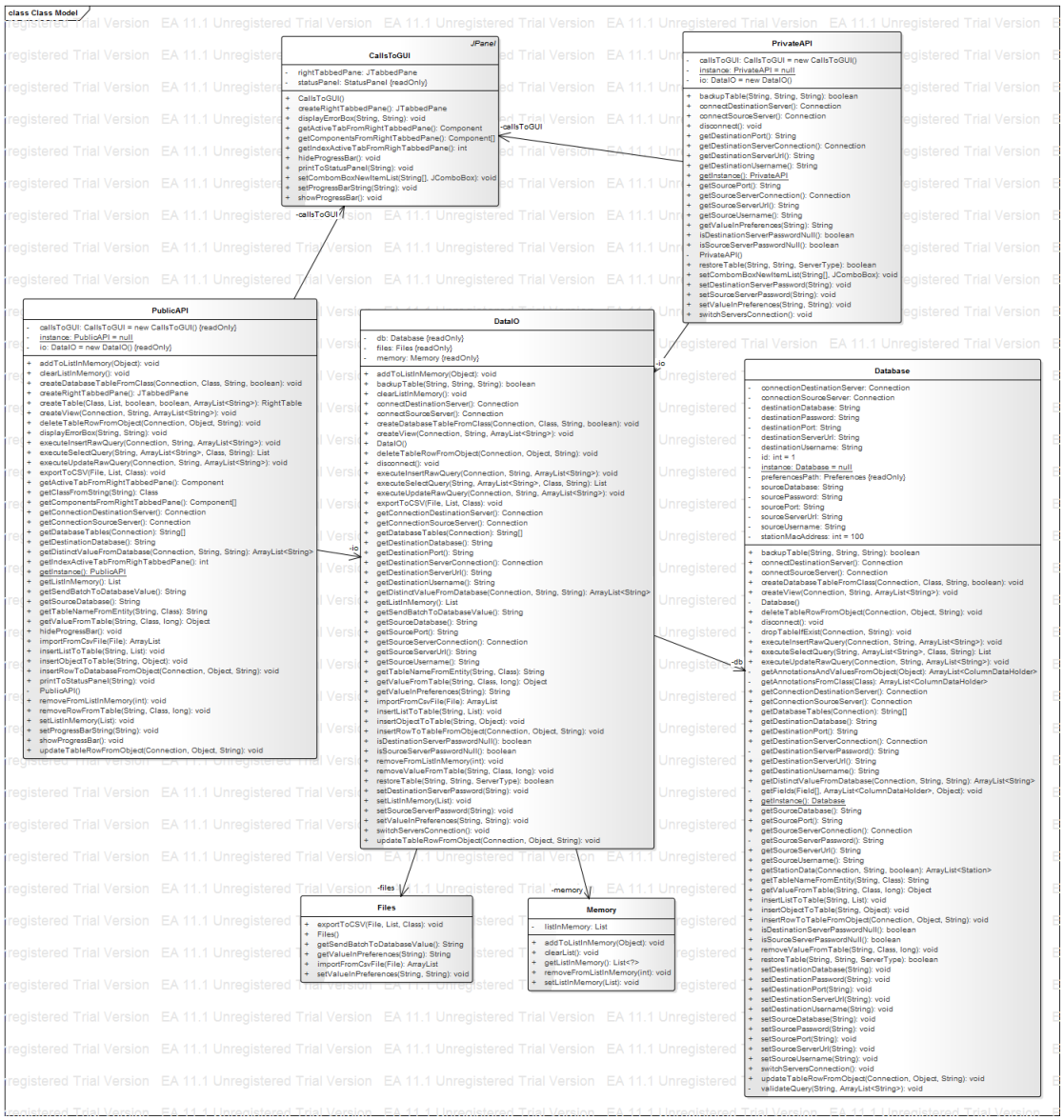


Figura B.1: Diagrama de classes da API

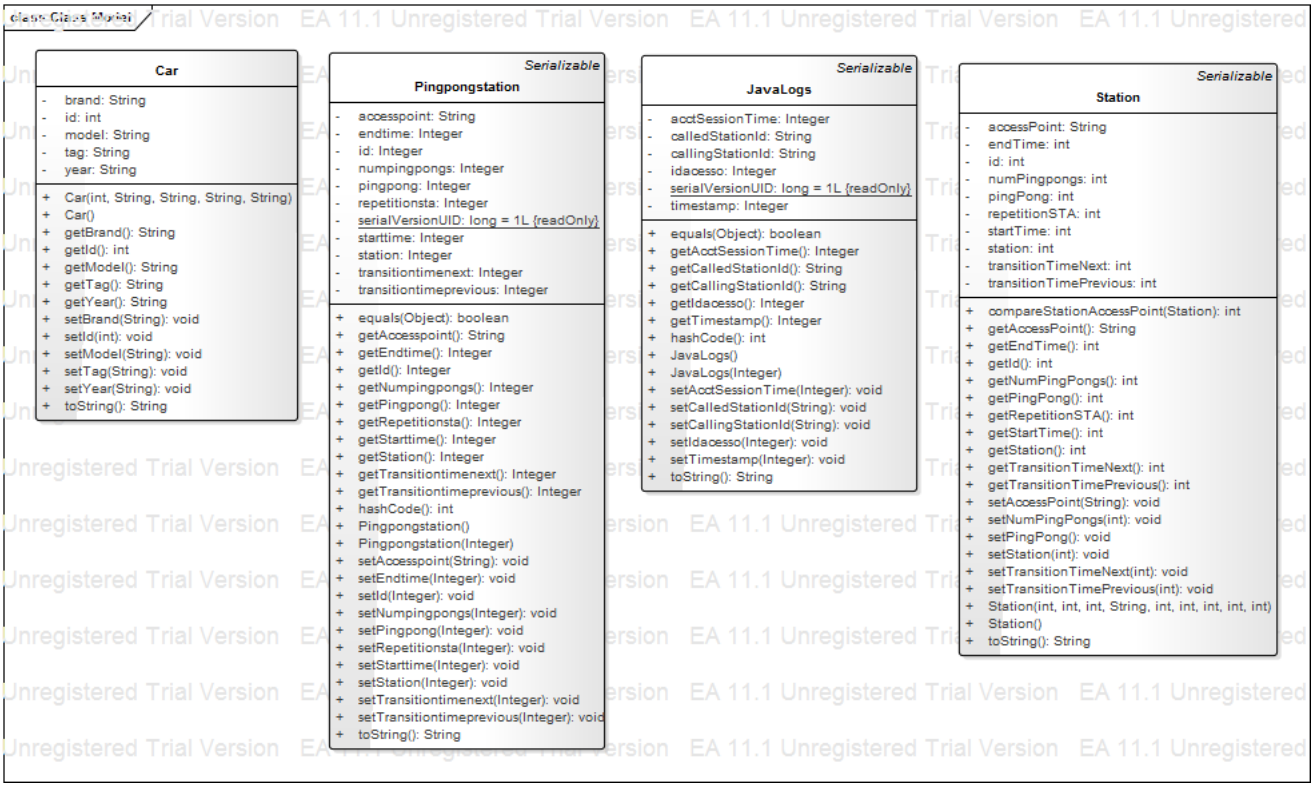


Figura B.2: Diagrama de classes do modelo

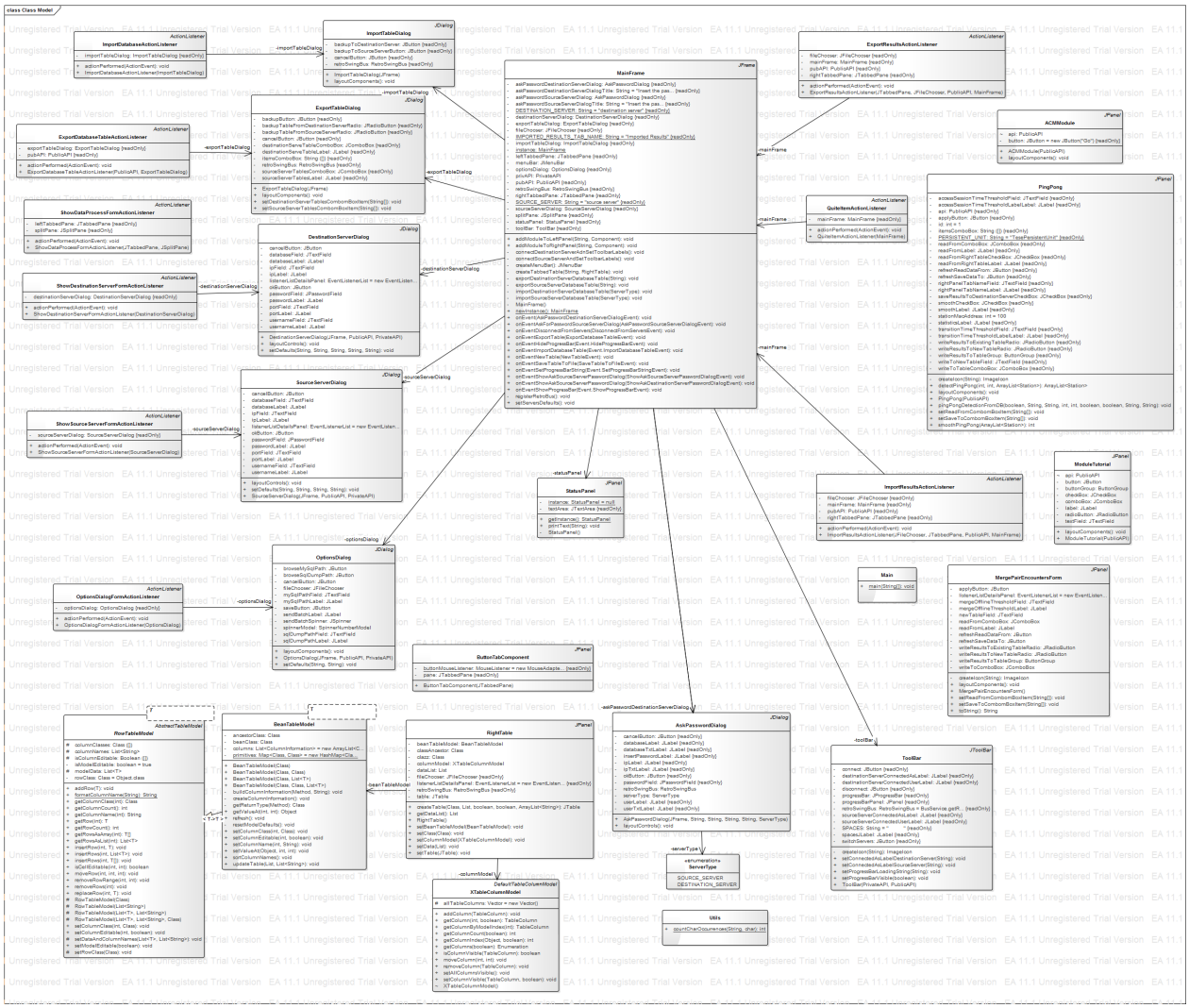


Figura B.3: Diagrama de classes da view