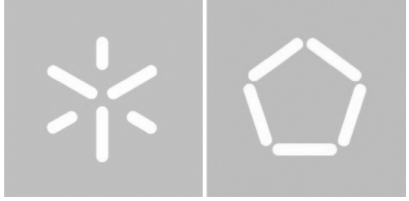

Universidade do Minho

Escola de Engenharia

Alexandre Manuel Loureiro Barbosa

Requisitos Não-Funcionais em Aplicações Orientadas a Serviços: Análise da Tecnologia Fuse ESB

Dissertação de Mestrado

Engenharia e Gestão de Sistemas de Informação

Trabalho efetuado sob a orientação do

Professor Doutor Ricardo J. Machado

Outubro de 2012

DECLARAÇÃO

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

Universidade do Minho, 30 de Outubro de 2012

Assinatura: _____

Agradecimentos

Um conjunto inúmero de pessoas merecem ser agradecidas pelo seu contributo para com este trabalho e sem as quais a realização do mesmo não seria possível. A todas elas gostaria de manifestar o meu apreço e em particular:

Ao Professor Doutor Ricardo J. Machado, não só pelo seu excepcional papel como Orientador de Dissertação mas também como Professor ao longo do meu percurso académico.

Ao meu Orientador de estágio da Bosch Car Multimédia, Marco Couto, pela sua formidável orientação, por tudo o que me ensinou e pela oportunidade que me deu em aprender e trabalhar com um vasto conjunto de interessantes tecnologias.

Ao José Carlos Martins e a todos os membros do CI da Bosch Car Multimédia pelo excelente acolhimento durante o ano que lá passei.

A toda a minha família, especialmente aos meus pais e avós, por tudo o que fizeram por mim sendo o seu apoio neste trabalho mais um pequeno exemplo.

Orientadores

Professor Doutor Ricardo J. Machado
Marco Couto

Autor

Alexandre Barbosa

Requisitos Não-Funcionais em Aplicações Orientadas a Serviços: Análise da Tecnologia Fuse ESB

Resumo

No âmbito de suporte e integração de aplicações empresariais um dos modelos arquitecturais emergentes é o *Enterprise Service Bus* (*ESB*), que desde o seu aparecimento tem ganho uma maior importância no suporte a Sistemas de Informação de elevada complexidade. As aplicações ligadas a este usam-no como um *middleware* (um intermediário) na troca de mensagens criando assim um ambiente *loose coupled*. Tal importância sobre a arquitectura *ESB* no suporte aplicacional aumenta se considerar-mos por exemplo a necessidade de uma maior interoperabilidade, disponibilidade e desempenho aplicacional. Algo que um único *ESB* não contempla, mas que a utilização de um *Cluster* de *ESB*s disponibilizando uma espécie de *ESB* distribuído por vários servidores pretende assim ajudar a resolver.

O presente trabalho que se realiza num contexto real e que assim usa um *ESB* em concreto (a tecnologia *Fuse ESB*), aborda tais necessidades. Pretendo dotar o *Fuse ESB* dos requisitos não-funcionais de Interoperabilidade Distribuída e Disponibilidade Aplicacionais. Onde primeiro requisito propõe uma melhor e transparente interacção entre aplicações de *Fuse ESB*s diferentes, sendo estes membros do mesmo *Cluster* de *Fuse ESB*s. E o segundo uma maior resistência das aplicações às falhas dos *Fuse ESB*s em que se inserem.

Para estes dois requisitos não-funcionais é apresentado como devem ser conceptualmente aplicados, tirando proveito das tecnologias e características existentes no *Fuse ESB*. E posteriormente são enunciados todos os aspectos e detalhes realizados na sua implementação.

Advisors

Professor Doutor Ricardo J. Machado
Marco Couto

Author

Alexandre Barbosa

Non-Functional Requirements in Service Oriented Applications: Analysis of the Fuse ESB Technology

Abstract

In the scope and integration of business applications one of the leading architectural models is the Enterprise Service Bus (ESB), that since it's appearance has gained a lot of importance in the support of high complexity Information Systems. The applications connected to it, use it as a middleware (a broker) in the exchange of messages therefore creating a loose coupled environment. Such significance of an ESB in it's application support grows if we consider for example the necessity of an greater application interoperability, availability and performance. Something that a single ESB does not achieve, but the creation of a Cluster of ESBs, providing a kind of distributed ESB among several servers will help resolve.

The present work, which is performed in a real environment and that so requires the use of a specific ESB (the Fuse ESB technology), tackles such necessities. Intending to provide the non-functional requirements of Application Distributed Interoperability and Availability. Where the first requirement proposes a better and transparent interaction between applications from different Fuse ESBs, in which these are members of the same Cluster of Fuse ESBs. And the second a higher application tolerance to failures of the Fuse ESBs to which they belong to.

To both of these non functional requirements it's presented how they should be conceptually applied, making use of the technologies and characteristics of the Fuse ESB. And after that the presentation of all topics and details related with their implementation.

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

DA: Disponibilidade Aplicacional

DCA: Distribuição de Carga Aplicacional

DOSGi: Distributed OSGi

ESB: Enterprise Service Bus

IAD: Interoperabilidade Aplicacional Distribuída

JAR: Java Archive

JBI: Java Business Integration

JMS: Java Message Service

JVM: Java Virtual Machine

MOM: Message-oriented Middleware

OSGi: Open Services Gateway initiative

SOA: Service-oriented architecture

SPEL: Spring Expression Language

Spring DM: Spring Dynamic Modules

Requisitos Não-Funcionais em Aplicações Orientadas a Serviços: Análise da Tecnologia Fuse ESB

Capítulo 1

Introdução

1.1 Enquadramento

Num mundo cada vez mais competitivo os Sistemas de Informação apresentam-se como fundamentais para as organizações dando suporte aos seus processos através de um vasto conjunto de soluções. A evolução destas soluções é cada vez mais munida de complexidade, que inevitavelmente torna a sua gestão e integração com outros componentes difícil de realizar.

Com este tipo de preocupações em mente, uma das abordagens que tem suscitado muito interesse na concepção, desenvolvimento e integração de sistemas é o modelo arquitectural de ***Service-oriented architecture*** (*SOA*) [Davis 2009]. *SOA* assenta num conceito em que as aplicações disponibilizam informação/valor sob a forma de serviços, proporcionado uma flexibilidade na conectividade e comunicação das aplicações [Ortiz 2007]. Deste modo, uma outra arquitectura relativamente recente e que aplica este conceito de *SOA* é o ***Enterprise Service Bus*** (*ESB*). Este é empregue na integração de unidades de negócio ligando distintas aplicações, que por sua vez podem recorrer a diferentes plataformas e ambientes [Maréchaux 2006]. Durante o seu tempo de vida a arquitectura *ESB* tem ganho bastante atenção no âmbito da integração de sistemas, sendo que actualmente existem uma panóplia de soluções, abrangendo soluções comerciais e *open source*.

Examinando ainda que superficialmente a arquitectura *ESB* é fácil constatar que resolve ou ajuda a resolver problemas de interoperabilidade aplicacional. Contudo questões como disponibilidade e distribuição de carga de aplicações ligadas a um *ESB* (isto é, a uma implementação da arquitectura ESB) ou mesmo a interoperabilidade de aplicações de diferentes *ESB*s, não são simplesmente resolvidas recorrendo a um único *ESB*. Na resolução deste tipo problemas uma das abordagens mais comuns e possivelmente a mais usual, passa pelo uso de um *Cluster*, mais especificamente um *ESB Cluster*.

Um *ESB Cluster* representa um conjunto de *ESB*s ligados entre si (de notar que cada *ESB* encontra-se numa máquina diferente) e que tem como objectivo realizar uma determinada tarefa de mediação (entre duas ou mais aplicações), dando do ponto de vista aplicacional a ilusão de se tratar de um único *ESB*. A utilização de um *Cluster* de *ESB*s, ou por outras palavras de um grupo de *ESB*s distribuídos cada um por um servidor e ligados entre si, irá melhorar o suporte às aplicações que estão contidas nele. Suporte esse que é pretendido disponibilizar sob a forma dos requisitos não-funcionais de **Interoperabilidade Aplicacional Distribuída** (IAD) e **Disponibilidade Aplicacional** (DA).

O requisito de IAD refere-se à capacidade de um *ESB* ou mais concretamente das aplicações ligadas a esse, comunicarem com outras que estão ligadas a outros *ESB*s.

Comunicação que terá de ser efectuada dentro do *Cluster* como seria de esperar, e que idealmente deve ser o mais transparente possível. Isto é, quando uma aplicação comunica com outra deve fazê-lo como tudo se trata-se de um único *ESB*, não estando ciente de que está a comunicar com outra aplicação que está noutro *ESB*, noutra máquina. Podemos também destacar que este requisito não-funcional é visto como mais relevante. Primeiro porque estabelece uma óptima plataforma para desenvolver outros requisitos não-funcionais, ou seja podem utilizá-lo para realizar os seus objectivos. E porque como este requisito requer o uso de um *Cluster*, isto pode acabar por actuar em parte como uma forma de gerar maior disponibilidade e/ou melhor distribuição de carga. Por distribuir os pontos de falha e complexidade por múltiplos *ESB*s.

Para o requisito de DA, é fácil compreender a sua importância/relevância. Sendo essencial garantir a disponibilidade das aplicações de um *ESB*, devido à sua importância face a um contexto empresarial.

1.2 Motivação

Esta Dissertação tem fundamentalmente como propósito a análise de requisitos não-funcionais no âmbito de *ESB*s, mais concretamente o de IAD e DA. A análise de tais requisitos é uma necessidade presente num contexto real, nomeadamente na *Bosch Car Multimédia*, onde o trabalho se insere. Actualmente o suporte a aplicações empresariais é dado através de vários *ESB*s (concretamente com a tecnologia *Fuse ESB*), porém não existe qualquer tipo de partilha de informação entre estes, originando possíveis problemas como a existência de pontos únicos de falha ou replicação desnecessária de serviços. Podendo eventualmente prejudicar o funcionamento das aplicações que dependem do *Fuse ESB*.

Da análise efectuada aos requisitos não-funcionais enunciados, pretende-se averiguar como é que podem ser concebidos no *Fuse ESB* ou por outras palavras o que é esperado que façam, que venham trazer ao *Fuse ESB*. Consoante esta informação é pretendido que sejam implementados os requisitos tratados, surgindo a explicação de como os implementar caso façam uso de tecnologias já existentes ou um protótipo (com a respectiva arquitectura) caso seja criada uma solução.

1.3 Objectivos

Assim face ao enquadramento e motivação desta Dissertação vejamos de seguida os principais objectivos da mesma, colocando também a seguinte questão de investigação:

"Como conceber e implementar os requisitos não-funcionais de IAD e DA na tecnologia *Fuse ESB*?".

- **Avaliar conceptualmente como elaborar os requisitos não-funcionais** - Estabelecidos os requisitos não-funcionais a desenvolver no *Fuse ESB* o primeiro passo será definir de que forma, a um nível conceptual devem estes ser implementados. Servindo esta avaliação como um esquema do comportamento que o requisito deve ter perante o *Fuse ESB* e consoante todas as características/tecnologias que o compõem. Esta informação será crucial para definir uma solução capaz de implementar o requisito.

- **Definir soluções que implementem os requisitos não-funcionais** - Com base nos requisitos não-funcionais a tratar e na forma como estes devem ser concebidos, é necessário escolher uma tecnologia ou conjunto de tecnologias capazes de os implementar no *Fuse ESB*. Sendo que estas tecnologias podem satisfazer completamente um requisito, isto é o único esforço a realizar seria basicamente a sua instalação e configuração e portanto estas tecnologias são a própria solução capaz de implementar um determinado requisito não-funcional. Ou então são tecnologias que servem como base para a construção de soluções que satisfaçam o requisito.

- **Implementar os requisitos não-funcionais** - Definidos os requisitos não-funcionais, o seu contributo para o *Fuse ESB* e as soluções que os devem implementar, o último passo será efectivamente a sua implementação. Onde deve ser descrito todo o processo de como dotar o *Fuse ESB* de tais requisitos. No caso dos mesmos serem disponibilizados por uma solução já existente deve ser explicado o processo da sua utilização/configuração, e por outro lado se for criada uma solução de raiz deve resultar num protótipo com respectiva arquitectura.

Por fim, faz-se aqui um esclarecimento sobre o conteúdo abordado em todo este trabalho e particularmente no capítulo 2 de revisão de literatura. Sendo que este mesmo conteúdo dedica-se exclusivamente à análise da arquitectura *ESB*, não abordando outro tipo de questões arquitecturais de *software* e *middleware* ou outros conceitos arquitecturais como por exemplo: *Event-Driven Architecture*, *Enterprise Application Integration,* etc. Uma decisão efectuada com base nas restrições impostas pelo contexto real, sendo a utilização da tecnologia *Fuse ESB* e consequentemente da arquitectura *ESB* uma necessidade. Assim todos os esforços foram concentrados na análise da arquitectura *ESB* e dos requisitos não-funcionais enunciados, o que por si só já são bastante complexos.

1.4 Abordagem Metodológica

Estipulados os objectivos para este trabalho, é correcto discutir a abordagem metodológica que acompanhou o projecto e que suportou o alcance dos seus objectivos. A análise, compreensão e deliberação da abordagem metodológica a usar nesta Dissertação é de grande importância no desenvolvimento de um projecto de cariz científico como este.

Do vasto conjunto de abordagens metodológicas existentes poderia ter sido escolhida qualquer uma como: *Empirical Research* utilizada na recolha e análise de dados, de provas; *Case Study Research* analisando experiências, casos passados e averiguando assim os resultados das soluções, abordagens e técnicas implementadas; ou mesmo um *Survey* analisando e recolhendo dados com base técnicas de cariz estatístico. No entanto face às circunstâncias, ao contexto em que esta Dissertação se desenvolveu, parece ser indicado afirmar que abordagens metodológicas tais como estas enunciadas não são as mais apropriadas.

Assim, a abordagem de investigação escolhida para este trabalho foi o *Design Science Research*, pela forma como esta se adapta às necessidades do mesmo. Por outras palavras, a grande orientação do *Design Science Research* à construção e aplicação de um ou mais artefactos que resolva um conjunto de problemas [Hevner et al. 2004], adequa-se às circunstâncias e objectivos.

Deste modo, através da abordagem de investigação *Design Science Research* é pretendido que se estude e perceba um determinado problema (a análise de requisitos não-funcionais) num determinado domínio (a tecnologia *Fuse ESB*, que é o *ESB* estipulado pelas necessidades do contexto real) e que através da criação e aplicação de um ou mais artefactos (as soluções capazes de implementar os requisitos não-funcionais no *Fuse ESB*), se chegue a uma resolução do problema.

1.5 Organização do Documento

À excepção do presente capítulo, a seguinte lista enuncia e detalha o conteúdo dos restantes capítulos do documento.

Capítulo 2 Secção que apresenta a revisão de literatura efectuada nesta Dissertação e onde são introduzidos e definidos os principais conceitos e problemas a tratar. Começa por abordar meticulosamente a arquitectura *ESB* observando os seus benefícios, forte ligação ao conceito de *SOA*, principais mecanismos e terminando com um olhar ao Fuse *ESB*. De seguida é analisado o principal requisito não-funcional o de IAD, atingido através de um *Cluster* de *Fuse ESB*s, bem como este poderia ser alcançado ou

concebido com uma aplicação especializada na troca de mensagens. Finalmente são abordados os requisitos não-funcionais de DA e **Distribuição de Carga Aplicacional** (DCA) olhando para a sua importância fora e dentro do âmbito de *ESB*s, os seus conceitos e algumas abordagens existentes.

Capítulo 3 Esta secção tem como principal objectivo definir conceptualmente os requisitos não-funcionais a tratar no âmbito do *Fuse ESB*, sendo estes o de IAD e DA. Para cada um é analisado o que é pretendido que façam no *Fuse ESB*, e como é que o podem fazer. No caso do de IAD é apontada a tecnologia escolhida para o implementar e os motivos da sua escolha. Inversamente para o requisito de DA é concebida uma solução de *failover* por falta de opções, e onde é explicado detalhadamente o funcionamento da mesma. Finalmente é feito um resumo do ambiente de desenvolvimento utilizado para implementar estes requisitos e as várias tecnologias que o compõem.

Capítulo 4 Aqui são analisados ao pormenor todos os detalhes técnicos da implementação dos requisitos não-funcionais de IAD e DA. Para o primeiro é mostrado como o implementar recorrendo à tecnologia escolhida e são enunciados cuidados a ter na utilizado do mesmo. Quanto ao segundo é apresentada extensivamente a arquitectura da solução de *failover* desenvolvida (que inclui as suas principais classes e o seu papel no processo), aspectos a ter atenção quanto à configuração da mesma e problemas/-limitações encontradas na solução, bem como a própria solução tenta ultrapassar-las.

Capítulo 5 Neste último capítulo é feita uma análise dos resultados e conhecimento obtido com o desenvolvimento dos requisitos não-funcionais de IAD e DA. Onde são apresentados os esforços para atingir os objectivos que foram propostos, os contributos deste trabalho e recomendações de investigação para o futuro.

Capítulo 2

Revisão de Literatura

2.1 Arquitectura *ESB*

É cada vez mais evidente e lógica, a preocupação e o esforço feito na reutilização dos recursos organizacionais, nomeadamente na área dos Sistemas de Informação onde as aplicações, os recursos organizacionais podem ser vistos como serviços e utilizados em conjunto, formando uma determinada solução de negócio. Assim a arquitectura *ESB* surge em 2002 por iniciativa de fabricantes de soluções de *middleware*, com o intuito de disponibilizar uma arquitectura para integração aplicacional em larga escala, e que suporta-se as necessidades de organizações distribuídas e bastante flexíveis [Chappell 2004].

Para melhor entender o conceito de *ESB*, e visto não existir um consenso relativamente ao seu significado, vejamos algumas definições segundo alguns autores.

2.1.1 O Conceito *ESB*

Segundo Menge [2007] um *ESB* é uma infraestrutura de integração baseada no uso de mensagens e de *open* standards, que suporta o encaminhamento, invocação e mediação de serviços com o objectivo de facilitar a interacção entre diferentes aplicações e serviços de uma forma segura e de confiança. Noutra perspectiva mais simplificada, o conceito de *ESB* poderá ser visto como uma arquitectura cujo objectivo de resolver problemas de integração [Rademakers e Dirksen 2008].

O *ESB* actua assim como um *middleware* que dá suporte à implementação de uma *SOA* numa organização [Keen et al. 2005]. Este suporte é dado através da capacidade do *ESB* em integrar e gerir os serviços, os recursos de uma organização, tornar transparentes os aspectos técnicos das interacções entre serviços e tornar transparente a implementação de um serviço ao utilizador do mesmo [Bean 2009; Keen et al. 2005].

2.1.2 Benefícios da Arquitectura *ESB*

Para perceber melhor o propósito e os benefícios que uma arquitectura *ESB* traz, vejamos um exemplo apresentado na figura 2.1, de integração de várias aplicações sem recorrer à arquitectura *ESB* ou semelhante. É importante alertar que apesar do exemplo dado nesta secção mencionar um conjunto de aplicações de uma determinada organização, um *ESB* não integra apenas aplicações. Um *ESB* integra aplicações, serviços, e outros componentes de software, que podem ser da organização em questão ou externos à organização. Pois independentemente dos componentes de software que estejam ligados ao *ESB* estes serão sempre vistos como serviços. A secção seguinte ajudará a perceber melhor este pormenor.

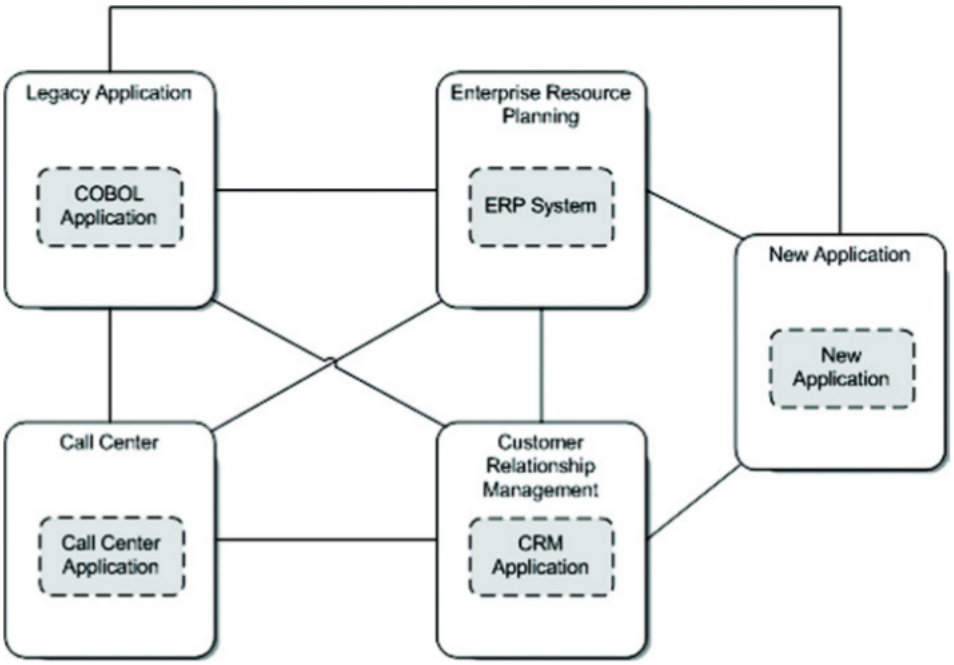

Figura 2.1: Integração de aplicações sem o uso de um *ESB*, adaptado de [Rademakers e Dirksen 2008]

Neste exemplo de integração entre aplicações, representado por uma arquitectura ponto a ponto (*point-to-point arquitecture*), o grande problema encontra-se na complexidade em ligar/integrar as várias aplicações. Cada linha representa uma ligação entre duas aplicações, e por cada linha seria necessário criar ou comprar uma solução que integre as duas aplicações.

Integrar este conjunto enorme de aplicações traria um vasto leque de problemas. Entre esses problemas está o tempo necessário (prejudicando muito provavelmente o negócio) a complexidade em criar e/ou configurar as ligações entre as duas aplicações e o custo associado a toda a integração mais o custo ligado à gestão das ligações entre as várias aplicações [Rademakers e Dirksen 2008].

Assim os benefícios da arquitectura *ESB* (ou simplesmente do *ESB*), segundo Rademakers e Dirksen [2008], podem resumir-se aos seguintes:

- Aumento da flexibilidade do ambiente composto pelas várias Tecnologias de Informação de uma organização mais os serviços externos a que esta tem acesso. Consequentemente ajudando a organização a criar novos produtos, melhorar os que já possui, etc.

- Diminuição na complexidade de gerir e integrar o leque de aplicações e serviços de

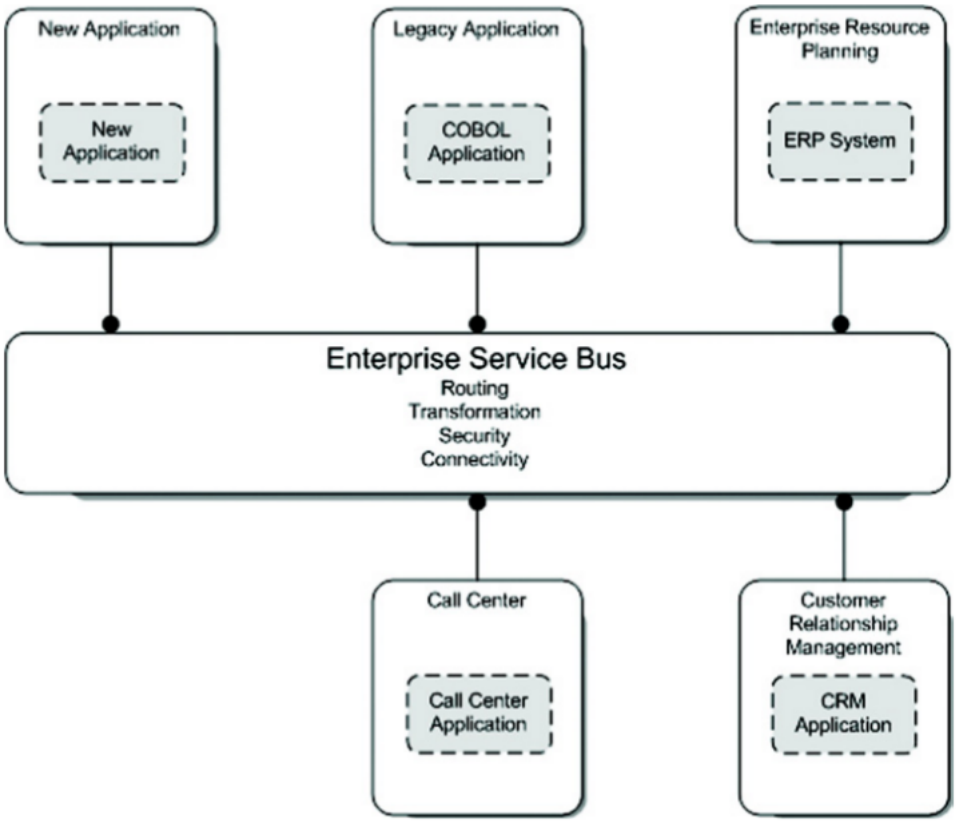

Figura 2.2: Integração de aplicações recorrendo a um *ESB*, adaptado de [Rademakers e Dirksen 2008]

uma organização.

- Redução do custo em gerir e integrar o leque de aplicações e serviços de uma organização.

A figura 2.2 mostra a integração de um conjunto de aplicações recorrendo a um *ESB*.

De notar que a complexidade de integração das várias aplicações diminui, em certos casos mais do que noutros, no entanto não desaparece. A complexidade apenas se encontra escondida pelo *ESB* [Rademakers e Dirksen 2008].

2.1.3 Incorporação de Princípios *SOA*

É evidente pelo que foi discutido até agora, e pelas definições de *ESB* apresentadas, que o *ESB* se relaciona fortemente com *SOA*, e assim não deixará de ser importante expor esse relacionamento perante dois pontos, duas questões:

- Como um *ESB* aplica uma *SOA*?
- Que valor um *ESB* acrescenta a uma *SOA*?

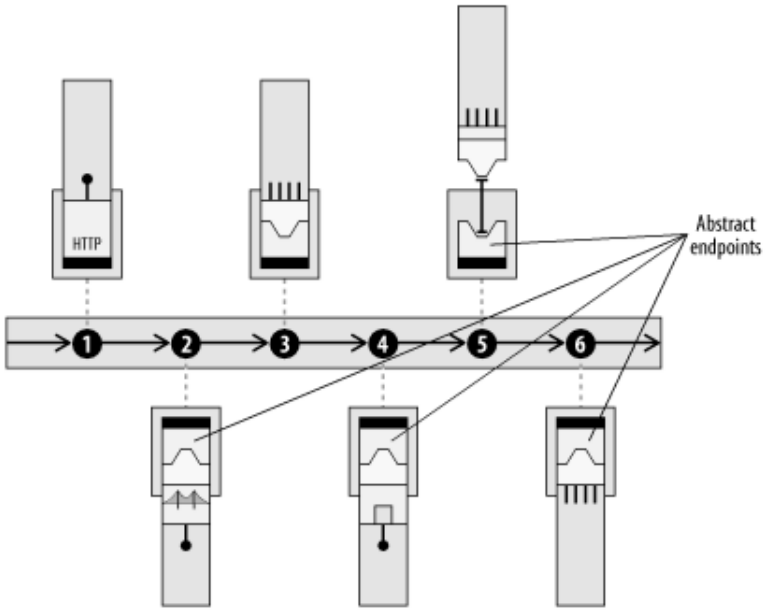

Figura 2.3: Conjunto de *abstract endpoints*, adaptado de [Chappell 2004]

Contudo antes de analisar-mos estas questões, é necessário dar um pequeno olhar à definição de *SOA*. Numa *SOA* os recursos de software nela inseridos são disponibilizados como serviços/funções [Maréchaux 2006] e possuem as seguintes características [Papazoglou e Van Den Heuvel 2007]:

- são bem definidos, ou seja aquilo que disponibilizam, o serviço prestado não é em nada ambíguo
- disponibilizam funcionalidades que estão relacionadas e que trazem valor para o negócio
- são independentes do estado ou contexto do restantes serviços

Relativamente à questão "Como um *ESB* aplica uma *SOA*?", isto é feito recorrendo ao conceito de *abstract endpoints*, ou simplesmente *endpoint*, como pode ser visto na figura 2.3. Um *endpoint* num *ESB* cria uma abstracção dos detalhes técnicos do serviço (ex: endereço, informação de ligação), permitindo aos vários serviços comunicar entre si recorrendo apenas a um nome lógico e deixando ao *ESB* a responsabilidade de fazer o mapeamento dos nomes lógicos com o respectivo endereço [Papazoglou e Van Den Heuvel 2007]. O que cada *endpoint* contem, ou representa pode ser bastante variável, desde uma simples aplicação, uma aplicação *legacy* ou mesmo um *Enterprise Resource Planning* [Chappell 2004].

Assim recorrendo a este conceito de *abstract endpoint*, é possível gerar um ambiente capaz de oferecer uma independência entre serviços, um ambiente *loose coupled*, sendo

que um qualquer serviço ligado ao *ESB* pode ser facilmente actualizado, movimentado (pressupõe-se a alteração da sua localização física) ou substituído, sem que tenha impacto nos outros serviços existentes no *ESB* [Papazoglou e Van Den Heuvel 2007].

Em relação ao segundo ponto, um *ESB* acrescenta valor a uma comum *SOA* com as operações *find*/*bind*/*invoke* [Chappell 2004]. O modelo *find*/*bind*/*invoke* passa por procurar (*find*) um serviço no *service registry* (este contêm informação sobre o local, a função e outros aspectos relativos a um conjunto de serviços) com base em determinados critérios, associar (*bind*) a ligação entre quem solicita o serviço e quem o presta e por fim invocar (*invoke*) o serviço em questão.

O problema de uma comum *SOA* é que cada solicitador de um serviço precisa de definir toda a lógica de ligação, *find*/*bind*/*invoke*, na sua aplicação. Enquanto que recorrendo a um *ESB* estas tarefas são efectuadas pelo mesmo retirando toda a carga ao solicitador, este apenas terá trabalho de configuração [Chappell 2004].

Resumindo, um *ESB* apresenta-se como uma solução ideal para implementar uma *SOA*, pois disponibiliza os mecanismos necessários para interligar o conjunto de serviços que compõem uma determinada solução sem comprometer a segurança, fiabilidade, desempenho e escalabilidade [Menge 2007].

2.1.4 Mecanismos da Arquitectura *ESB*

Sendo o *ESB* uma arquitectura de bastante complexidade, é útil clarificar as suas capacidades, funcionalidades. Para tal vejamos um conjunto de mecanismos que são mais ou menos genéricos a uma qualquer implementação da arquitectura *ESB*.

Transparência

Um *ESB* deverá de ser capaz de criar uma transparência em relação à localização de um solicitador de serviços (*service requestor*) e de um prestador de serviços (*service provider*). Por outras palavras, se um determinado prestador de serviços mudar a localização do seu servidor não deverá ter qualquer impacto no solicitador de serviços, como foi discutido anteriormente.

Segundo Rademakers e Dirksen [2008], esta transparência poderá ser implementada de três maneiras:

- Através do uso de uma simples configuração em XML
- Com uma base de dados, tornando a configuração dinâmica

- Com o *Service Registry*

Este último, o *service registry*, apesar de mais complexo irá permitir adicionar um conjunto de características interessantes, tais como o controlo da qualidade de serviço e a descoberta e gestão das capacidades de um determinado serviço. A qualidade de serviço será útil quando houver a necessidade de garantir que o *ESB* suporte interacções com níveis mínimos de qualidade para proteger a informação transmitida [Keen et al. 2005]. Assim por exemplo, um solicitador de serviços pode escolher um serviço com um nível de qualidade reduzido e ver um aumento no desempenho, na qualidade de comunicação, isto porque o *ESB* foi capaz de modificar certas características da comunicação melhorando a sua qualidade, como por exemplo o protocolo usado para as mensagens [Schmidt et al. 2005].

Para que o que foi dado como exemplo da qualidade de serviço se torne possível, o *service registry* teria que primeiro ter efectuado a descoberta das características de um determinado serviço. O *ESB service registry* é capaz de efectuar a descoberta e consequente gestão dos meta dados sobre as características de um serviço, como por exemplo as capacidades, políticas, qualidade de serviço, etc [Schmidt et al. 2005]. Com esta informação o *ESB* irá efectuar a melhor ligação possível entre um solicitador e prestador de serviços.

Conversão de protocolos de transporte

Outro dos mecanismos bastante útil num *ESB* é a conversão de protocolos de transporte (*transport protocol conversion*). Perante uma situação em que um solicitador de um serviço utiliza um protocolo de transporte diferente do prestador do serviço, o *ESB* deverá ser capaz de efectuar a conversão permitindo a comunicação entre os dois [Rademakers e Dirksen 2008].

Os componentes que permitem efectuar a conversão de protocolos num *ESB* são denominados de adaptadores de protocolos (*protocol adaptares*), se o *ESB* utilizado não suportar um determinado protocolo poderá sem comprado ou criado o adaptador [Rademakers e Dirksen 2008]. Isto torna-se possível graças à agilidade disponibilizada pelo *ESB*.

De referir que as funcionalidades descritas aqui de conversão de protocolos de transporte, transformação de mensagens e reforço de mensagens (as duas últimas são apresentadas à frente), normalmente são denominadas de funcionalidades de mediação [Menge 2007].

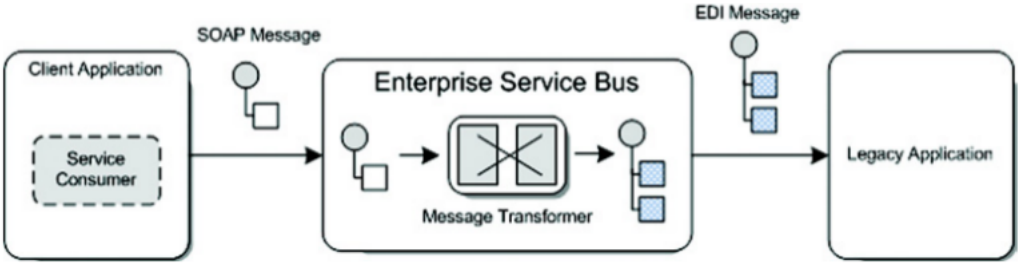

Figura 2.4: Exemplo da transformação do formato de uma mensagem num *ESB*, adaptado de [Rademakers e Dirksen 2008]

Transformação de mensagens

Caso o formato de uma mensagem trocada entre um solicitador de um serviço e o prestador de um serviço não coincida o *ESB* deverá transformar a mensagem gerada pelo solicitador no formato usado pelo prestador [Rademakers e Dirksen 2008], e vice-versa. Um exemplo desta transformação poderá ser visto na figura 2.4.

A transformação do formato de uma mensagem pode ser feita de várias maneiras como por exemplo, através de uma ferramenta adquirida no mercado, criando uma aplicação específica para o efeito ou então recorrendo a uma *XSLT* (*Extensible Stylesheet Language Transformation*) *style sheet* [Rademakers e Dirksen 2008].

Encaminhamento de mensagens

O encaminhamento (*routing*) é o mecanismo que permite ao *ESB* escolher o destino de uma determinada mensagem. Esta funcionalidade é uma das mais importantes num *ESB*, pois permitirá que uma determinada mensagem não esteja restringida a apenas um prestador de serviços [Menge 2007]. O *ESB* irá escolher o prestador que seja mais adequado perante determinadas condições.

Estas condições podem ser:

- Baseadas no conteúdo (*content-based*) da mensagem. Em que o destino da mensagem será baseado no conteúdo da mesma [Menge 2007; Rademakers e Dirksen 2008].
- Baseadas num filtro (*message filter*), que pode prevenir que a mensagem chegue a um determinado destino [Rademakers e Dirksen 2008] ou então apenas é enviada se o conteúdo da mesma cumprir um determinado critério, caso contrário a mensagem é apagada [Menge 2007].
- Baseadas numa lista de destinos (*recipient list*), em que a mesma mensagem é enviada para múltiplos destinos [Menge 2007; Rademakers e Dirksen 2008].

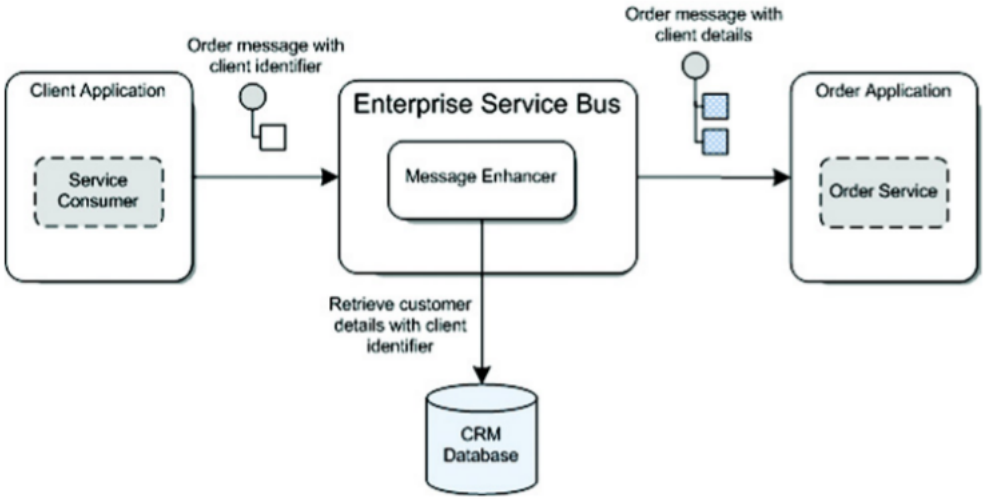

Figura 2.5: Exemplo de reforço de mensagens num *ESB*, adaptado de [Rademakers e Dirksen 2008]

Reforço de mensagens

O reforço de mensagens (*message enhancement*) e transformação de mensagens estão ligadas uma à outra, no entanto não devem ser confundidas pois a transformação de mensagens é relativa ao formato das mesmas e o reforço ao conteúdo [Rademakers e Dirksen 2008].

Pode ser solicitado a um determinado prestador de serviços informação sobre um cliente, no entanto o solicitador do serviço requer um determinado nível de detalhe de informação sobre esse cliente que o prestador não possui. O *ESB* pode então acrescentar a informação que o solicitador requer à mensagem, questionando outras base de dados sobre esse cliente.

Este exemplo pode ser melhor compreendido recorrendo à figura 2.5.

Segurança

Devido à sensibilidade da informação que um *ESB* controla, uma das grandes funcionalidades que o mesmo deverá disponibilizar é a segurança da informação.

Assim deverão ser disponibilizadas pelo *ESB*, como parte da funcionalidade de segurança da informação, as seguintes funções [Keen et al. 2005; Menge 2007; Rademakers e Dirksen 2008]:

- Identificação e autenticação
- Controlo de acessos

- Confidencialidade da informação
- Integridade da informação
- Gestão e administração da segurança
- Recuperação da informação
- Relatórios de incidentes

Monitorização e Gestão

Finalmente, um *ESB* deverá possuir um conjunto de ferramentas que ajudem a gerir e monitorizar o mesmo. Esta funcionalidade de monitorização e gestão tornar-se-à indispensável à medida que o *ESB* aumenta em tamanho e complexidade.

As ferramentas de monitorização e gestão e as suas características vão depender de *ESB* para *ESB*, no entanto no geral deverão ter funcionalidades como: monitorizar o estado de um determinado *web service* ou serviço, ou seja se está disponível ou não, gerir o conjuntos de *web services* e serviços ligados a um *ESB*, entre outras [Rademakers e Dirksen 2008].

2.1.5 A Tecnologia *Fuse ESB*

Após toda esta análise feita a vários aspectos de um qualquer *ESB*, é indicado acabar esta secção efectuando uma breve descrição do *ESB* a ser utilizado neste projecto, o *Fuse ESB*, e dos seus componentes. De tal forma, esta análise ainda que pequena, proporcionará uma melhor compreensão da complexa ferramenta que é o *Fuse ESB* e dos conteúdos discutidos mais adiante.

A tecnologia *Fuse ESB* refere-se a um *ESB open source* baseado no *Apache ServiceMix* e que as seguintes são as suas principais características [FuseSource 2011]:

- Uso de um *kernel* leve capaz de ser executado na maioria das plataformas e que utiliza a *framework* ***Open Services Gateway initiative*** (*OSGi*). A *framework* *OSGi* permite tratar as aplicações como componentes, facilitando assim a gestão das suas dependências e o seu ciclo de vida [OSGi Alliance nd].
- Suporte de ***Java Business Integration*** (*JBI*).
- Suporte de ligações a serviços de outras infraestruturas com vários tipos de protocolos e formatos.

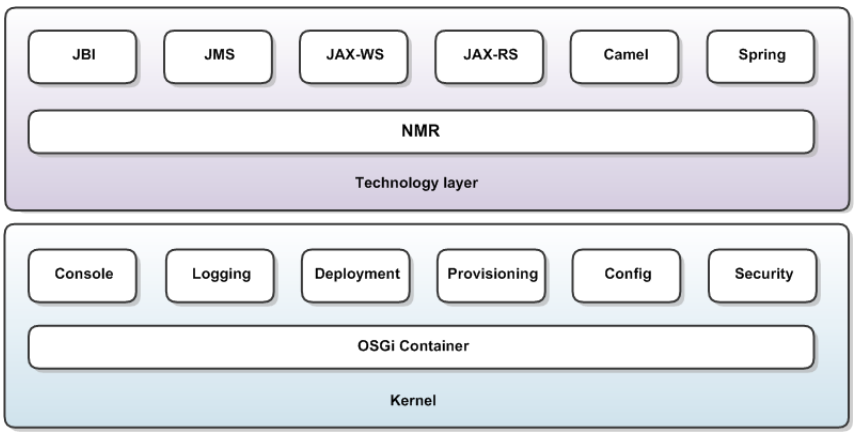

Figura 2.6: Arquitectura do Fuse *ESB*, adaptado de [FuseSource 2011]

- Uso de vários standards limitando as dependências do *ESB* a determinadas tecnologias.

Na figura 2.6 é exposta a arquitectura do *Fuse ESB* dando uma visão geral das funcionalidades e tecnologias utilizadas. Das tecnologias exibidas destaca-se como a mais relevante no âmbito deste projecto a *framework OSGi*.

A *framework OSGi* definida pela *OSGi Alliance*, e concretizada no *Fuse ESB* como um *container OSGi*, declara-se com o objectivo de corrigir o suporte para modularidade da plataforma *Java*, introduzindo um paradigma de *SOA* numa perspectiva local [Hall et al. 2011, p.4]. A *framework OSGi* do *Fuse ESB* revela-se como sendo de extrema importância para o presente trabalho. Isto porque é nela que assentam todas as aplicações do *Fuse ESB* disponibilizando uma excelente infraestrutura, a qual traz vários benefícios ao nível da modularidade e comunicação aplicacionais. Sendo importante esclarecer que esta comunicação ou interacção aplicacionais referem-se apenas a um ambiente local, ou seja na mesma ***Java Virtual Machine*** (*JVM*).

Note-se também que apesar da *framework OSGi* ser essencial para com o *Fuse ESB*, a mesma não está agarrada ou não funciona exclusivamente no *Fuse ESB*. Isto é, a *framework OSGi* é isolada do *Fuse ESB* e apenas é usada por este na implementação do conceito de *ESB*. Tanto que apenas em versões mais recentes do *Fuse ESB* é que a *framework OSGi* começou a ser usada.

2.2 O Requisito Não-Funcional de Interoperabilidade Aplicacional Distribuída (IAD)

Anteriormente, foi discutida com algum detalhe a complexa arquitectura que é o *ESB* e que é alvo de trabalho desta Dissertação. Nesta secção é discutido o requisito não-funcional de IAD (ou também designado de *clustering*), que como pode ser compreendido por tudo o que foi dito até ao momento, pretendente dotar as aplicações contidas num *ESB* de uma maior capacidade de comunicação com outras de outros *ESB*s. E acabando também por actuar como uma plataforma que ajudará a concretizar os requisitos enunciados futuramente de DA e DCA.

Neste projecto de Dissertação que faz uso do *Fuse ESB* para dar suporte a aplicações, o uso desta abordagem passa mais concretamente pela criação de um *Cluster* de *Fuse ESB*s. Isto é, um conjunto de máquinas interligadas entre si e onde a sua ligação é da responsabilidade da rede de *Fuse ESB*s. Desta forma, o *Cluster* aos olhos do utilizador, acaba por se tornar num único *ESB* que é distribuído por um conjunto de servidores.

O resto desta secção estrutura-se do seguinte modo. Primeiro é apresentada uma breve definição do conceito de *Cluster* e um conjunto de informação adicional, como por exemplo o tipo de *Clusters* existentes. Posteriormente, é analisada com algum detalhe uma das formas possíveis de trabalhar o requisito não-funcional de IAD no *Fuse ESB*. Neste caso a abordagem apresentada, inclui o uso da tecnologia *Apache ActiveMQ*, que faz parte da distribuição do *Fuse ESB*. Contudo é importante alertar que apesar da análise desta tecnologia quanto ao requisito de IAD, a mesma não é utilizada no resto deste documento como uma forma de implementar tal requisito. Mesmo assim a utilidade da sua discussão centra-se em dois pontos. Primeiro mostra uma possível forma ou área para trabalhar tal requisito. Baseada numa ferramenta que faz parte do próprio *Fuse ESB*, muito conceituada e utilizada no âmbito de troca de mensagem entre aplicações. Segundo permite a análise e esclarecimento de conceitos importantes e transversais ao domínio aplicacional distribuído, conceitos estes que se apresentam úteis para realização deste trabalho.

2.2.1 *Clusters* Aplicacionais

Um *Cluster* é um grupo de computadores *loose coupled* trabalhando de um modo cooperativo, de tal forma que podem ser vistos em muitos aspectos como uma única máquina [Sharma et al. 2009]. Tradicionalmente a ligação e gestão das várias máquinas do *Cluster* é feita via software e o caso de um *Fuse ESB Cluster* não é excepção, como é discutido mais à frente.

Geralmente o uso de um *Cluster* tem como objectivo abordar questões de disponibilidade, distribuição de carga e capacidade computacional [Sadashiv e Kumar 2011]. Assim, Sharma et al. [2009] categoriza um *Cluster* de três formas possíveis:

High-availability Clusters criados com o propósito de melhorar a disponibilidade dos serviços contidos no *Cluster*. A técnica mais utilizada nestas situações é a redundância de servidores, garantindo assim que se um servidor falhar existirá outro que o substitua [Chappell e Berry 2007].

Load balancing Clusters estes recebem a carga e distribuem-na para outros servidores, preparados para o efeito.

High-performance Clusters são concebidos especificamente para alcançar níveis mais elevados de desempenho, partindo a carga computacional pelos vários nós.

A utilização de *Clusters* é portanto a solução tipicamente usada para que as infraestruturas de uma determinada organização disponham de um maior desempenho e disponibilidade, não só para situações inesperadas em que existe uma falha do sistema, mas também por exemplo para situações de manutenção.

2.2.2 Interoperabilidade Aplicacional Distribuída com o *ActiveMQ*

Percebendo que o uso de um determinado *ESB* por parte de um conjunto de aplicações é feito fundamentalmente com o propósito de permitir a comunicação entre as mesmas, sendo deste modo o *ESB* a ponte que gere a troca de mensagens entre as aplicações de uma forma *loose coupled*, é natural concordar que o *ESB* possui um papel fulcral neste tipo de contextos. Esta importância é estendida ainda mais se considerar-mos dois pontos. Primeiro as duas características que têm vindo a ser discutidas, no sentido que é pretendido que o *ESB* esteja disponível permitindo que as mensagens sejam trocadas e que o facto de ser o *ESB* a efectuar a troca de mensagens melhore o desempenho de quem as envia e recebe. Segundo, que para atingir tais características temos que possuir um ambiente distribuído, um *Cluster* de *ESB*s. Permitindo a comunicação de aplicações e sendo a irrelevante a sua localização dentro do *Cluster*.

Concretamente no Fuse *ESB* uma ferramenta capaz de efectuar a troca de mensagens entre as várias aplicações (de diferentes *ESB*s ou do mesmo) é o Apache *ActiveMQ*, que segue a especificação ***Java Message Service*** (*JMS*). O *ActiveMQ* é um *message broker* empregue na comunicação remota entre sistemas que utilizem a especificação *JMS*, disponibilizando assim um conjunto de características (ex: disponibilidade, desempenho,

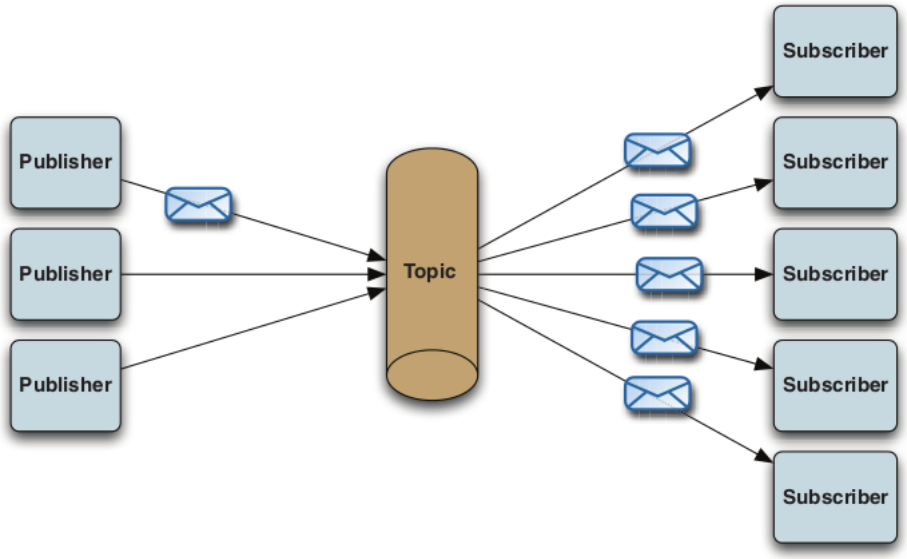

Figura 2.7: Domínio *Publish/Subscribe*, adaptado de [Snyder et al. 2011]

escalabilidade, etc) para a comunicação empresarial [Snyder et al. 2011]. O *ActiveMQ* entra assim na categoria de software ***Message-oriented Middleware*** (*MOM*), em que age como um mediador de mensagens entre quem as envia e quem as recebe disponibilizando um ambiente *loose coupled*, ou seja uma ambiente em que quem envia as mensagens não ter qualquer conhecimento acerca de quem as recebe, e vice-versa. De notar que este tipo de ambiente permite também que as mensagens sejam trocadas de forma assíncrona, visto não haver um ligação directa entre quem cria e recebe.

Vejamos então com algum detalhe as técnicas utilizadas pelo *ActiveMQ* na criação de um *Cluster*, começando por perceber levemente dois domínios na troca de mensagens identificados pela especificação *JMS*.

Domínios *JMS*

Os dois domínios identificados pela especificação *JMS* são o domínio de *Publish/Subscribe* e *Point-to-Point* [Oracle 2002].

No primeiro domínio são definidos tópicos, para os quais são publicadas mensagens que são replicadas para todos aqueles que sobrescreveram ao tópico em questão, como pode ser percebido pela figura 2.7. Por defeito se um determinado subscritor não estiver disponível no momento em que a mensagem foi enviada não a irá receber, no entanto a subscrição poderá ser durável infinitamente sendo enviada posteriormente quando o subscritor estiver disponível novamente.

O domínio *Point-to-Point* (figura 2.8) define um destino para as mensagens, inti-

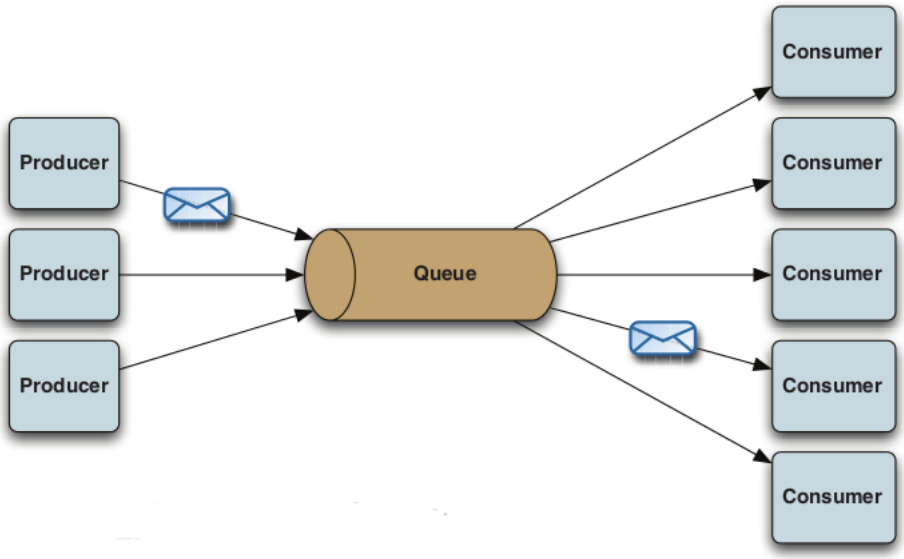

Figura 2.8: Domínio *Point-to-Point*, adaptado de [Snyder et al. 2011]

tulado de *queue*, para onde os *message producers* enviam mensagens que iram ser recebidas por um dos *message consumers* que está registado na *queue*. É importante focar que apenas um dos *message consumers* poderá consumir a mensagem, isto traduz-se numa melhor distribuição da carga entre os vários *message consumers*, sendo que se um morrer ou apresentar níveis de desempenho menos desejados os outros iram compensar [Apache Software Foundation nda].

Cluster de *Brokers*

Recorrendo ao *ActiveMQ* é possível a criação de um *Cluster* de *Brokers*, ou seja a ligação de várias instâncias do *ActiveMQ* (cada um num *Fuse ESB* diferente), disponibilizando às aplicações que interagem com os *message brokers* uma maior disponibilidade nos serviços de troca de mensagens. Este *Cluster* como é de esperar possibilita o uso de qualquer um dos domínios discutidos anteriormente.

O *Cluster* é então formado pela ligação dos vários *Brokers*, ligação essa que pode ser pré-definida recorrendo a endereços estáticos ou feita dinamicamente, em que estes dão a conhecer os seus serviços e ficam a conhecer os dos outros *Brokers* [Snyder et al. 2011, p.89]. Nesta rede de *Brokers* (intitulada de *Network of Brokers*) é utilizado o mecanismo de *store and foward*, que se traduz na passagem de uma determinada mensagem por vários *Brokers* até chegar ao seu destino, como pode ser percebido pela figura 2.9. Contudo a mensagem pertence unicamente a um *Broker*, que foi aquele que a recebeu inicialmente e que a guardou (se estiver a ser utilizada a persistência das mensagens, para eventualidade de

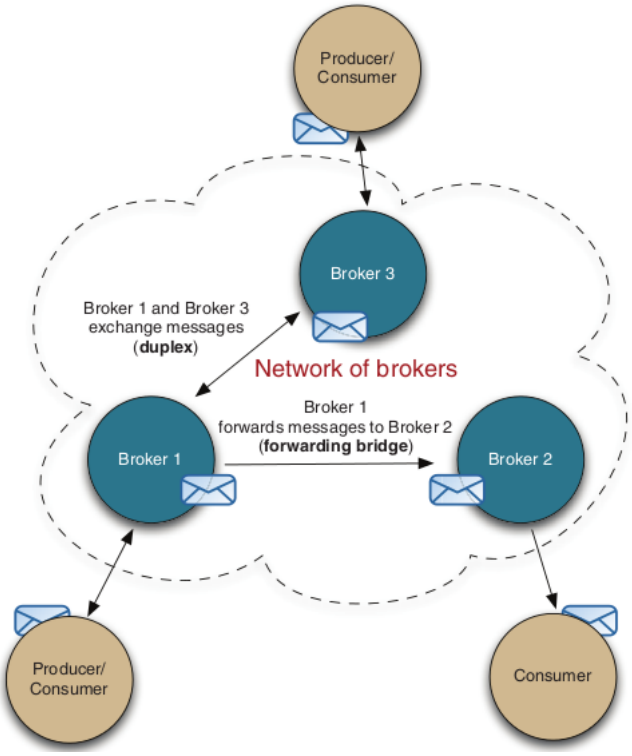

Figura 2.9: Exemplo de um *Cluster* de *Brokers* (*Network of Brokers*), adaptado de [Snyder et al. 2011]

existir uma falha no sistema) antes de a enviar para outro dos *Brokers* do *Cluster* [Apache Software Foundation ndb].

Do lado do cliente, seja este um *message consumer*, *message producer* ou ambos, não existe a preocupação das ligações feitas entre os vários *Brokers*, este apenas efectua a sua ligação a um dos *Brokers* para entregar e/ou receber mensagens. Contudo, para evitar uma situação em que o cliente não consegue efectuar a troca de mensagens com o *Broker* ao qual se ligou por causa de uma eventual falha, existe a possibilidade do cliente recorrer ao protocolo *failover* possibilitando que este se ligue a outro *Broker* do *Cluster*. O protocolo *failover*, à semelhança do que acontece com os *Brokers* do próprio *Cluster*, permite que os clientes se liguem a outro *Broker* de uma lista pré-definida ou efectuem uma descoberta dinamicamente [Apache Software Foundation nda].

Apesar desta possibilidade de um cliente se ligar a outro *Broker* do *Cluster* no caso de existir uma falha num deles, a informação persistida e que pertence ao *Broker* que falhou, estará indisponível até que este seja reiniciado [Apache Software Foundation nda]. Assim de forma a precaver este tipo de ocorrências o *ActiveMQ* utiliza o conceito de *Master/Slave*. Neste tipo de configuração (figura 2.10) existem dois *Brokers*, o *Master* e o

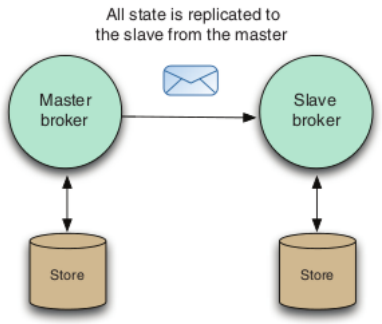

Figura 2.10: Exemplo de uma configuração *Master/Slave*, adaptado de [Snyder et al. 2011]

Slave, e enquanto o primeiro está activo e a responder aos clientes e a outros *Brokers* do *Cluster*, vai replicando todas as mensagens recebidas para o *Slave*. No momento em que o *Slave* detecta que o *Master* falha, este assume a posição de *Master* [Snyder et al. 2011, p.258]. Esta combinação de *Master/Slave* (pode existir mais do que um *Slave* noutras configurações, mas que apresentam um mecanismo diferente na replicação das mensagens) forma um *Broker* lógico que pode ser perfeitamente ligado ao *Cluster* de *Brokers*, e que vem consequentemente aumentar a disponibilidade do sistema [Apache Software Foundation ndb].

2.3 O Requisito Não-Funcional de Disponibilidade Aplicacional (DA)

Dando seguimento ao propósito desta Dissertação nesta secção é analisado o requisito não-funcional de DA, e também o DCA. Mesmo sabendo que apenas o primeiro é analisado ao longo do resto do trabalho, mas tendo em consideração a forte relação dos dois vê-se como benéfico que o requisito DCA seja também focado nesta secção.

De seguida é feita uma breve compreensão da importância destes requisitos não só de um ponto de vista em que as aplicações existentes valem-se de um *ESB*, como de uma situação em que não existe um *ESB*. Posteriormente são analisados efectivamente os conceitos de DCA e DA, bem como outros conceitos relacionados. Finalmente é apresentada uma secção que expõe sucintamente uma das muitas soluções existentes na resolução das necessidades enunciadas, que recorre a um *ESB* distribuído.

2.3.1 Importância

Para uma qualquer aplicação a sua disponibilidade e o seu desempenho são sem dúvida relevantes num contexto de suporte ao negócio. Estas poderão por vezes, numa fase

inicial, ser menosprezadas e tidas apenas em consideração posteriormente, à medida que as aplicações e os sistemas em que estas se inserem vão ganhando complexidade.

Marcus e Stern [2003] apontam uma série de consequências, de custos para um negócio relativos à falta de disponibilidade. Este tipo de custos poderão ser directos, sendo a falta de produtividade o exemplo mais comum, e que dependendo do tipo de negócio e do tipo de actividade em causa, poderá revelar-se mais ou menos custoso.

Por outro lado, os custos para o negócio podem ser custos indirectos [Marcus e Stern 2003], que englobam por exemplo: a insatisfação do cliente, que vê o serviço ou produto que pretende usufruir, atrasado ou indisponível; uma má reputação para com os clientes; desmotivação por parte dos colaboradores que vêm o negócio para o qual contribuem a falhar; etc.

Se a disponibilidade e o desempenho de uma comum aplicação tornam-se tão vitais para um bom funcionamento do negócio, a importância das mesmas características aumenta ainda mais com a inclusão de um *ESB*. À medida que o *ESB* se vai enraizando no negócio, e no suporte que disponibiliza ao mesmo, é inevitável que este se torne crítico. Sendo o *ESB* essencialmente um canal de comunicação utilizado pelas aplicações existentes no negócio, recorrer a um único *ESB*, a um *ESB* com uma arquitectura central, acaba por se tornar num ponto de falha único (*single point of failure*) [Yin et al. 2009] . Por outras palavras, se é o *ESB* que tem a função de realizar a comunicação entre as várias aplicações, a falha ou desempenho inadequado do mesmo irá acabar por prejudicar as aplicações que se ligam e dependem dele , por impossibilitar a sua comunicação [IBM 2005].

Por esta razão é correto afirmar, que é vital que o *ESB* trabalhe quando for necessitado e que seja capaz de aguentar com a carga que lhe é atribuída, apresentando níveis de disponibilidade e desempenho tão ou mais elevados que as aplicações que liga [IBM 2005].

2.3.2 Definição

Vista assim a importância da DCA e DA, surge então o momento de apresentar uma possível definição para estes conceitos.

Disponibilidade Aplicacional

Apesar de muita da discussão efectuada até ao momento e de um dos objectivos desta Dissertação ser de facto a DA, o conceito de disponibilidade (também denominado de *high availability*) em si não se resume única e exclusivamente ao software, estando também

presente ao nível do hardware. Deste modo, o conceito de disponibilidade é relevante a vários níveis, com vários propósitos.

Vejamos assim, o conceito perante duas perspectivas. Uma que define disponibilidade não fazendo qualquer tipo de pressuposto, para além dos requisitos do sistema, e outra que define disponibilidade, com o pressuposto que esta disponibilidade é criada recorrendo ao uso de um *middleware*, mais especificamente o *ESB*. Note-se que estas não são as únicas perspectivas existentes, mas são umas que se apresentam como relevantes para este trabalho.

Marcus e Stern [2003] definem disponibilidade como o nível de capacidade que um determinado sistema tem em estar operacional. Nível esse que está implícito na forma como o mesmo foi concebido, por outras palavras na sua arquitectura, e que pretende atingir ou ultrapassar os requisitos do negócio, para o qual o sistema foi implementado. Esta perspectiva como mencionado, toma apenas em consideração os requisitos do sistema na definição de disponibilidade do mesmo, não sendo influenciado por aspectos como por exemplo tecnologia [Marcus e Stern 2003].

Noutra perspectiva, em que se pretende específicamente DA num sistema onde existe um *ESB*, a disponibilidade deste sistema é concebida com o objectivo de evitar a perda de serviços, reduzindo ou gerindo as falhas e minimizando o tempo em que o *ESB* está inoperante [Astrova et al. 2010; Kruessmann et al. 2009].

Outro conceito que é relevante identificar neste momento e que está relacionado com DA e IAD, apesar do último ser apenas discutido numa secção futura, é o conceito de *failover*. Supondo uma situação em que existe um *Cluster*, de *ESB*s ou não, o *failover* é simplesmente a migração dos serviços de um servidor para outro [Marcus e Stern 2003]. A migração dos serviços durante este processo de *failover* deverá apresentar no mínimo as seguintes características [Marcus e Stern 2003]:

- Transparente, o processo de *failover* deve ser o menos intrusivo possível
- Rápido
- Com o mínimo de intervenção manual possível
- Após o processo de *failover* a informação deve estar sincronizada

Distribuição de Carga Aplicacional

Relativamente à DCA, esta é a capacidade de um determinado sistema em distribuir pedidos por vários recursos/aplicações [Koschel et al. 2011], com o objectivo criar um

equilíbrio na carga de trabalho dos mesmos recursos [IBM 2005].

Similarmente ao que acontece com o conceito de disponibilidade, mecanismos relativos ao de distribuição de carga (também denominado de *load balancing*) podem ser implementados em vários níveis, como os seguintes [Wang et al. 2007]:

- Nível de rede
- Nível de Sistema Operativo
- Nível Aplicacional, recorrendo ao *middleware*

Sendo o último aquele que se enquadra no âmbito da arquitectura *ESB*, e que representa o requisito de DCA.

2.3.3 Abordagens Existentes

Para colmatar o problema de disponibilidade e distribuição de carga existem várias abordagens, vários trabalhos efectuados, e em que de facto o tipo de solução mais usual passa pela utilização de *Clusters*. Concretamente no tópico dos *ESB Clusters*, podem ser identificados trabalhos, como por exemplo o de Yin et al. [2009] e Ning et al. [2008], que apresentam a sua solução na resolução deste tipo de problemas.

O último apresenta-se bastante relevante no âmbito desta Dissertação por tratar o mesmo conjunto de necessidades e de uma forma similar. Este expõe o seu *ESB* distribuído e baseado em *JBI*, que tem o propósito de ultrapassar as limitações impostas por um ambiente centralizado, como a falta de desempenho, ocorrência de um ponto de falha único e perda de mensagens no caso de ocorrer uma falha do sistema.

Apesar da semelhança entre o trabalho a ser realizado por esta Dissertação e o trabalho realizado por Ning et al. [2008]existem essencialmente duas diferenças entre os dois.

Primeiro, o anterior descarta o uso do *ESB ServiceMix* (que é aquele no qual o *Fuse ESB* se baseia) por na altura não satisfazer um conjunto de questões, o que actualmente poderá já não ser verdade com aparecimento de novas versões. E segundo, no trabalho de Ning et al. [2008] é utilizada exclusivamente a tecnologia *JBI* no âmbito de integração aplicacional, enquanto que nesta Dissertação será utilizada a *framework OSGi*.

2.4 Conclusões

Terminando este capítulo da revisão de literatura façamos um breve resumo da informação que esta contempla. Dividindo-se essencialmente na análise pormenorizada da

arquitectura *ESB* e de requisitos não-funcionais, nomeadamente DA e IAD.

Para a arquitectura *ESB* foi focado o seu conceito, os benefícios resultantes do uso de tal arquitectura de um ponto de vista aplicacional, a sua forte ligação ao conceito de SOA e uma descrição completa dos mecanismos que definem tal arquitectura. Terminado com uma apresentação do *ESB* utilizado em concreto nesta Dissertação, o *Fuse ESB*, bem como o seu principal componente que é a *framework OSGi*.

Efectuou-se também a análise do requisito não-funcional de IAD que se encontra fortemente ligado ao conceito de *Cluster*, visto necessitar deste para ser concretizado. Além disto foi discutido como poderia ser alcançado ou concretizado este requisito de IAD com recurso à tecnologia *ActiveMQ* (uma tecnologia utilizada na troca de mensagens entre aplicações e que faz parte do *Fuse ESB*). Contudo é importante esclarecer que esta tecnologia não é utilizada para atingir este requisito como será percebido com os futuros capítulos, mas fica aqui a sua análise para que sejam percebidos conceitos transversais ao domínio aplicacional distribuído.

E finalmente foram discutidos os requisitos não-funcionais de DCA e DA. Mesmo sabendo que só o último é tratado no resto do trabalho, mas que face à forte relação existente entre os dois foi considerado benéfica a sua análise. Para ambos os requisitos foi discutida a sua importância, definição e abordagens existentes.

Capítulo 3

Conceptualização dos Requisitos Não-Funcionais de IAD e DA na Tecnologia *Fuse ESB*

3.1 Introdução

A finalidade do presente capítulo é tratar os requisitos não-funcionais de IAD e DA que são alvo da preocupação desta Dissertação, mas puramente a um nível conceptual deixando detalhes técnicos da sua implementação para um capítulo seguinte. A análise conceptual destes requisitos vem explicar para cada um, o seu objectivo ou aquilo que pretendem trazer para o *Fuse ESB* e como é que o conseguem fazer. Fazendo-se valer das tecnologias/componentes existentes e particularmente da *framework OSGi*.

Para cada um dos requisitos é também apresentada a solução a utilizar para depois realizar a sua implementação. Sendo que para o requisito de IAD é indicada a tecnologia que o satisfaz completamente e os motivos da sua escolha. Inversamente para o DA é exposta a solução concebida para o implementar esclarecendo os conceitos e objectivos para a mesma.

Finalmente é feito um resumo do ambiente de desenvolvimento usado e das tecnologias que o compõem, e qual é o seu papel para com os esforços aqui realizados.

3.2 Interoperabilidade Aplicacional Distribuída

Nesta secção será abordado o propósito do requisito não-funcional de IAD para o com o trabalho em causa, a explicação da relevância deste no âmbito de suporte aplicacional e a sua utilidade quando relacionado com outros requisitos não-funcionais. Também é analisada a forma como conceptualmente o requisito de IAD pode ser concebido no *Fuse ESB*, com recurso às estruturas/capacidades do mesmo e finalmente como este foi alcançado/desenvolvido em concreto nesta Dissertação.

3.2.1 Definição e Propósito no Suporte Aplicacional

À luz deste trabalho, desta Dissertação, o requisito não-funcional de IAD é entendido como a capacidade de expor serviços/funcionalidades entre diferentes *Fuse ESB*s e consequentemente entre diferentes máquinas, preferencialmente de uma forma simples e transparente. Serviços estes que não se referem a funcionalidades que fazem parte do conceito de *ESB*, como por exemplo a transformação de mensagens discutida no capítulo 2, mas sim aqueles criados por quem usa o *Fuse ESB* e que necessitam deste para serem executados.

É então apenas com a união de várias instâncias do *Fuse ESB*, através da definição de um *Cluster* que é possível combater problemas tais como baixos níveis de desempenho, dificuldade de gestão de funcionalidades e indisponibilidade destas funcionalidades. Tudo

simplesmente através da distribuição de funcionalidades e consequentemente complexidade por todos os membros, por todos os membros do *Cluster*. Funcionalidades que serão acedidas sobre a forma de um serviço, independentemente do *Fuse ESB* que o disponibiliza, tornando inútil e desnecessária a replicação de serviços visto que estarão acessíveis a todo *Cluster*.

É então perceptível através da discussão efectuada em torno do requisito não-funcional de IAD que o seu propósito/relevância para esta Dissertação destaca-se em dois pontos. O primeiro e sem dúvida o mais relevante, é que seja possível num determinado *Cluster* de *Fuse ESB*s que os seus membros trabalhem de uma forma colaborativa expondo e consumindo serviços. Que corresponde à definição de IAD apresentada.

Sendo o requisito alcançado criando um ambiente colaborativo das *frameworks OSGi* de cada um dos *Fuse ESB*s, e em que as funcionalidades são expostas e consumidas via serviços *OSGi*, preferencialmente da forma mais transparente possível. A este ambiente distribuído de *frameworks OSGi* é dado o nome de ***Distributed OSGi*** (*DOSGi*), e cujos detalhes serão melhor compreendidos mais à frente.

O segundo ponto, é que este requisito de IAD será também útil na aplicação de outros requisitos não-funcionais, como por exemplo o de DA e DCA que são aqueles mais habituais ou mais preocupantes. Contudo é verdade que não será absolutamente necessário o IAD para atingir estes requisitos. Mas considerando o uso do *Fuse ESB* e modularidade da *framework OSGi* que faz parte do mesmo (ou seja, a capacidade de disponibilizar funcionalidades como módulos/componentes que podem ser facilmente instalados em qualquer ambiente *OSGi*), sem dúvida que o IAD se revela útil para os atingir.

A título de exemplo o IAD será bastante útil para com o requisito não-funcional de DA, analisado numa secção futura, por transpor a preocupação de tornar o *Fuse ESB* inteiro disponível para apenas determinadas funcionalidades. Isto é, se o relevante é garantir a disponibilidade de funcionalidades, podem ser movimentados os componentes que as disponibilizam entre *Fuse ESB*s do *Cluster*, o que se tornará muito mais conveniente e rápido. O IAD facilita esta movimentação de componentes por a tornar transparente, ou seja estes componentes podem desempenhar as suas funções em qualquer um dos *Fuse ESB*s por conseguirem interagir com outros componentes através de serviços que são expostos no *Cluster*.

3.2.2 Concepção na Tecnologia *Fuse ESB*

Efectuada uma compreensão detalhada do requisito não-funcional de IAD e da sua relevância no suporte aplicacional, vejamos com algum pormenor a forma pela qual é aplicado/concebido o mesmo requisito no *Fuse ESB*. E onde uma das suas mais vali-

osas características, mencionada várias vezes, a *framework OSGi* desempenha um papel fundamental.

A *framework OSGi* pode ser vista como uma estrutura adicional à plataforma *Java* e comprometida em ajudar a construir aplicações com um maior nível de modularidade [Hall et al. 2011, p.4]. Ou dito de outra forma, pretende a ajudar na construção de aplicações que são divididas em partes lógicas e em que cada uma trabalha um conjunto problemas, um conjunto de temáticas.

A qual divide-se nas seguintes três camadas [Hall et al. 2011]:

- *Module Layer* que define o modo como são agregadas e partilhadas as aplicações. Nesta é definido o conceito de módulo da *framework OSGi*, intitulado de *bundle* e que se trata de um ficheiro ***Java Archive*** (*JAR*) com um conjunto de meta-dados adicionais.

- *Lifecycle Layer*, controla o acesso de cada um dos módulos à *framework* e efectua a gestão do ciclo de vida dos mesmos.

- *Service Layer* que disponibiliza uma forma e estrutura para a interacção entre *bundles*.

Sendo o objectivo aqui a colaboração de *Fuse ESB*s num *Cluster* através da exposição e consumo de serviços, é relevante olhar para o conceito de *bundle* e para a sua interacção com outros *bundles*. Interacção essa que pode ser realizada de uma forma indirecta (usando a *module layer*), isto é onde existem dependências de um determinado *bundle* para com outros *bundles*, da mesma forma como um comum componente depende de um série de livrarias. Ou por outro lado uma interacção mais directa (usando a *service layer*) e a mais relevante aqui, em que um *bundle* expõe/publica serviços acessíveis a todos os outros que se encontram na *framework OSGi*, tal como a figura 3.1 exemplifica. De tal forma que o *bundle* que consome um determinado serviço não conhece, nem precisa, os detalhes da implementação ou que *bundle* o disponibiliza. Isto porque o acesso a um serviço será feito indirectamente através do *OSGi service registry*, que irá disponibilizar uma possível implementação desse mesmo serviço. Podendo o serviço consumido variar à medida que vão sendo introduzidas novas implementações, novos *bundles* a expor o mesmo serviço.

Esta forma de comunicação directa entre *bundles* pode parecer que se refere exactamente ao requisito não-funcional de IAD que tem vindo a ser discutido. Contudo apesar de se aproximar bastante, não o é porque a *framework OSGi* e a sua modularidade aqui focada referem-se a um ambiente local. Por outras palavras, tudo isto acontece na mesma *JVM* e portanto realiza-se numa única máquina. Revelando-se necessário adicionar a capacidade de interacção entre *bundles* de diferentes *frameworks OSGi* e logicamente diferentes

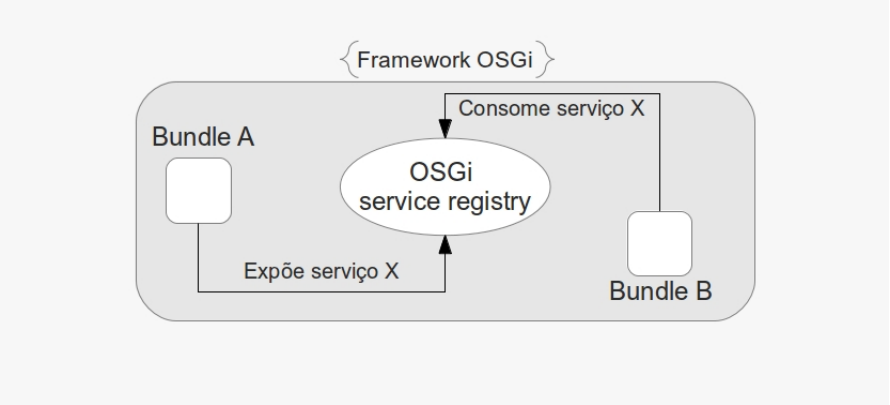

Figura 3.1: Exemplo da exposição e consumo de serviços na *framework OSGi*

*Fuse ESB*s, recorrendo à exposição e consumo de serviços como demonstrado com o pequeno exemplo da figura 3.2.

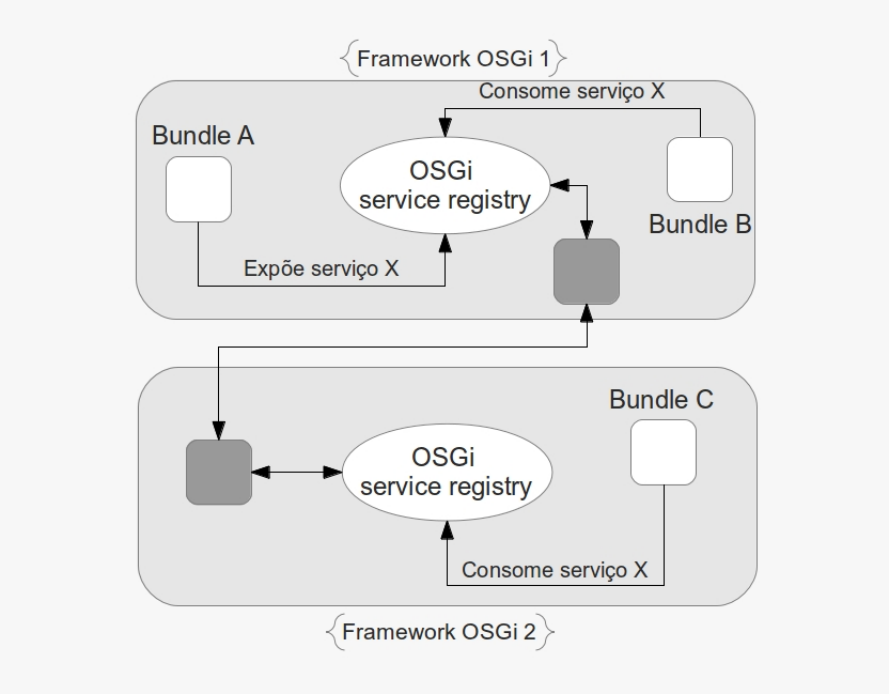

Figura 3.2: Exemplo da exposição e consumo de serviços *OSGi*, entre diferentes *frameworks*

Este exemplo idealiza o conceito subjacente à exposição e consumo de serviços *OSGi* entre diferentes *frameworks* até agora discutido. Onde a exposição de um serviço numa *framework* provocará a exposição desse mesmo serviço noutra. Em concreto a exposição do serviço X através do *OSGi service registry* da *framework* 1, leva a que seja também exposto na *framework* 2. Tornando o serviço X acessível tanto ao *bundle* B como ao C e em que cada *bundle* acede ao serviço através do *service registry* da sua *framework*, de

uma forma transparente como se trata-se de um comum serviço *OSGi*. E que eventualmente quando for acedido poderá tratar-se de um serviço local como é no caso do *bundle* B ou de um serviço remoto para o *bundle* C. No caso deste último, ao aceder ao serviço X estará a aceder à implementação disponibilizada pelo *bundle* A, noutra *framework OSGi*, sem ter a noção de que o está a fazer.

Finalmente um último detalhe essencial a focar, é que a concretização de toda esta lógica aqui descrita sobre a exposição e consumo de serviços remotos recai sobre um outro *bundle*, ou conjunto de *bundles*. Que é notificado da exposição de novos serviços pelo *OSGi service registry* e que comunica esses eventos a outros *bundles* iguais a este e instalados noutras *frameworks OSGi*, noutros *Fuse ESB*s. Os quais por sua vez vão registar estes serviços remotos nas suas *frameworks* e responder a pedidos efectuados, tratando de reencaminhar o pedido para uma *framework* em que de facto existe uma implementação do serviço.

É portanto este *bundle* especial que actua como a solução capaz de implementar e disponibilizar o requisito não-funcional de IAD.

3.2.3 Solução na Tecnologia *Fuse ESB*

Com o objectivo de trazer o requisito não-funcional de IAD tal como descrito e definido em 3.2.1 e da forma em 3.2.2 para o *Fuse ESB*, foram pesquisadas soluções existentes que satisfizessem estes pontos e que pudessem ser usadas no mesmo. Apenas duas foram encontradas, o *CXF DOSGi* e o *Karaf Cellar*, sendo esta última a solução seleccionada.

As razões da escolha do *Karaf Cellar* (ou simplesmente *Cellar*, que é um sub-projecto do *Karaf*), como solução a usar para implementar o requisito não-funcional de IAD resumem-se a :

- Disponibilizar totalmente o requisito de IAD. Permitindo expor e consumir serviços entre vários *Fuse ESB*s.
- O requisito é definido de uma forma muito transparente. A exposição de serviços requerer apenas o acrescento de um conjunto de meta-dados, algo que é comum e *framework OSGi* permite por defeito. E o consumo de serviços é realizado sem qualquer alteração a um normal serviço *OSGi*, sendo mesmo impossível controlar se o serviço consumido será local ou remoto caso existam várias implementações.
- O *Cellar* é uma ferramenta baseada numa das tecnologias base do *Fuse ESB*, o *Karaf*. Trabalhando com conceitos e recursos conhecidos por parte de quem usa o *Fuse ESB*.

- Face à outra alternativa o *Cellar* possui um melhor desempenho, visto que se baseia em protocolos binários para efectuar a sua comunicação, ao contrário dos *Web Services* usados pelo *CXF DOSGi*.

Em suma o *Cellar* permite criar uma ambiente colaborativo de *Fuse ESB*s, no domínio aplicacional de uma maneira simples e transparente. Note-se ainda que detalhes sobre a sua configuração, utilização e alguns cuidados a ter, serão apresentados no capítulo 4.

3.3 Disponibilidade Aplicacional

3.3.1 Concepção na Tecnologia *Fuse ESB*

Cientes da enorme complexidade do *Fuse ESB* e das suas inúmeras funcionalidades, serviços e soluções que pode albergar é mais que evidente que a sua disponibilidade e a dos seus componentes poderá estar em risco. Assim nesta secção será discutido mais um requisito não-funcional, o de DA a dotar ao *Fuse ESB* e que se relacionará activamente com o outro já discutido.

Relembrando o que foi explanado na secção 2.3.2, o conceito de DA refere-se à capacidade de manter um determinado sistema e os seus elementos operacionais. Ligado fortemente a este conceito está também o de *failover*, que será a transição dos componentes de um sistema para outro disponível, na eventualidade de uma falha. Tal processo quando executado estará efectivamente a contribuir para manter os níveis de disponibilidade de um sistema, minimizando o quanto possível danos provocados por falhas inesperadas e o tempo que o sistema fica indisponível.

Passando a um cenário mais tangível e concretamente aquele que engloba o *Fuse ESB*, a questão que se coloca é como garantir a disponibilidade dos componentes e respectivos serviços contidos nele. A resposta encontra-se precisamente nas capacidades do próprio ambiente até agora discutido, através:

- Da modularidade da *framework OSGi*.
- Do conceito de *features* do *Karaf* que se refere a uma colecção única de *OSGi bundles* e que facilita a definição, instalação e remoção de uma determinada funcionalidade/-solução.
- Do requisito não-funcional de IAD.

Assim com base nestas capacidades, a melhor maneira de garantir a DA é efectuando o processo de *failover* das aplicações/componentes de um *Fuse ESB* para outro dentro

de um *Cluster*, quando necessário. Sendo mais relevante garantir a disponibilidade de componentes únicos a um determinado membro desse mesmo *Cluster*, e excluindo aqueles que são comuns. Note-se também que é mais benéfico tratar estes componentes como *features* em vez de *bundles*, que são o elemento base da *framework OSGi*, simplesmente pela comodidade que estas trazem sem modificar ou prejudicar o funcionamento normal dos *bundles* que as compõem.

E finalmente é fulcral destacar que graças à implementação do requisito não-funcional de IAD o processo de *failover* que provocará a movimentação de componentes entre *Fuse ESB*s, em nada afecta o bom funcionamento desses mesmo componentes. Isto porque caso necessitem de serviços disponibilizados por outros componentes, conseguirão acedê-los por estarem expostos no *Cluster*.

3.3.2 Solução de *Failover* na Tecnologia *Fuse ESB*

Definido o conceito de como conseguir níveis acrescidos de DA através de um processo de *failover*, o próximo passo será apontar uma solução que utilize tal conceito. Infelizmente não foram encontradas quais queres soluções de *failover* compatíveis como o *Fuse ESB* e que trabalhassem este tipo de conceitos.

Para colmatar esta lacuna foi desenvolvida uma solução de *failover* capaz de efectuar o que foi descrito na secção anterior, compatível com o *Fuse ESB* e que se baseia no *Cellar* aproveitando as funcionalidades disponibilizadas por esta ferramenta. Criando então um ambiente onde não só é trabalhado o requisito não-funcional de DA mas também será compatível com o de IAD.

O objectivo desta solução é bastante simples e passa pela capacidade de num *Cluster* de *Fuse ESB*s, a falha de um dos membros seja compensada sem que as funções e serviços que o mesmo disponibiliza desapareçam. Este objectivo demonstra assim a necessidade de um ambiente onde cada *Fuse ESB* possui o seu conjunto de serviços, de funções, sem que haja replicação dos mesmos por outros. Mas que na eventual falha de um dos membros do *Cluster* os seus serviços sofrem um processo de *failover*, onde são transpostos para outro *Fuse ESB* da forma mais transparente e rápida possível.

De seguida são descritos vários aspectos do funcionamento da solução desenvolvida, permitindo perceber a forma como a solução idealiza todo o processo de *failover*. Desde como são definidos os serviços de cada membro do *Cluster*, ao momento em que um dos membros falha e os seus serviços são disponibilizados por outro membro. Detalhes sobre a sua configuração, utilização, limitações e alguns cuidados a ter, serão apresentados no capítulo 4.

Grupos *Cellar*

Utilizando o *Cellar* esta solução de *failover* tira proveito de dois importantes conceitos, o de grupo e *features*. Uma *feature* como já indicado é uma utilidade do *Karaf* e que corresponde a um conjunto de *OSGi bundles*. E por outro lado um grupo é uma entidade que possui um conjunto de recursos distribuídos e os quais partilha a todos os seus membros ou *Nodes*[1], seguindo a terminologia do *Cellar*. Aqui um *Node* representa cada um dos *Fuse ESB*s que fazem parte do *Cluster* e onde se encontra instalado o *Cellar*.

Um destes recursos distribuídos e aquele relevante para esta solução é uma lista de *features*, que pode ser sincronizada com cada *Node* que pertença ao grupo e instalando localmente as *features* desta lista. A figura 3.3 demonstra exactamente isso em que um *Node* ao sincronizar as suas *features* com as do grupo, vê os *bundles* que compõem estas a serem instalados na sua *framework OSGi*.

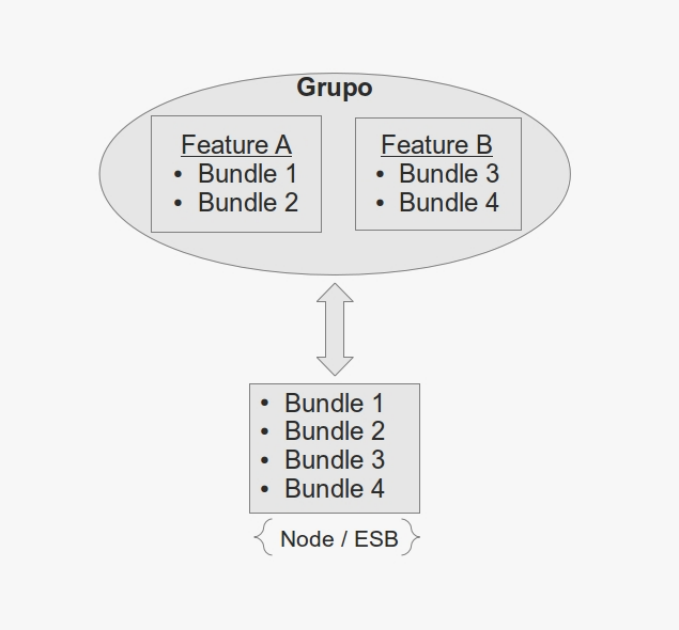

Figura 3.3: Exemplo da instalação de *features* de um grupo para um *Node*

De tal modo nesta solução foram definidos dois tipos de grupos, Grupo *Commons* e Grupo de Serviços, que são utilizados no processo de *failover* de forma a tirar proveito das funcionalidades disponibilizadas pelo *Cellar*. Especialmente a sua capacidade de definir um conjunto de *features* para cada grupo, o que facilita a movimentação das mesmas entre *Nodes*.

[1] O termo *Node* é utilizado pelo *Cellar* para se referir a uma determinada máquina, ou mais correctamente a uma *framework OSGi* no qual este se encontra. Sendo que ao longo deste documento, o uso do termo *Node* é sinónimo de *Fuse ESB*. E sempre que utilizado será num contexto relacionado com *Cellar*.

Grupo *Commons*

O **Grupo *Commons*** é um grupo único, isto é só poderá existir um deste tipo, servindo dois objectivos. Primeiro será uma forma simples de indicar que um determinado *Node* participa no processo de *failover* desta solução, querendo isto dizer que qualquer *Node* registado nesta solução será automaticamente forçado a pertencer a este grupo. Segundo, e ainda mais importante, é que o Grupo *Commons* permite como qualquer grupo do *Cellar*, que sejam definidas *features* específicas a este e que podem ser passadas para cada *Node* do grupo. Seguindo o raciocínio todos os *Nodes* que participem nesta solução de *failover*, vão partilhar este conjunto de *features*.

Esta lógica do Grupo *Commons* pode contudo parecer estranha se considerámos que um dos objectivos desta solução é que não existe-se uma replicação desnecessária de serviços entre *ESB*s, mais concretamente entre *Nodes*. Mas esta decisão foi tomada propositadamente para que numa situação em que todos os *Nodes* do *Cluster* necessitem da mesma *feature* (como poderia ser o caso de um determinado conjunto de livrarias) esta possa ser facilmente aplicada. Assim a não replicação de serviços é focada para as *features* disponibilizadas pelos Grupos de Serviços em vez das do Grupo *Commons*, como iremos ver de seguida.

Mas para que fique ainda mais clara a razão pela qual o Grupo *Commons* permite replicar *features* por todos os *Nodes* da solução imaginemos o seguinte cenário. Um determinado serviço (por exemplo o serviço A) implementado por um determinado *Node* e exposto no *Cluster*, com ajuda do requisito não-funcional de IAD, só poderá ser consumido pelos restantes *Nodes* se estes conhecerem o *interface* do serviço A. E se este *interface* estiver associado a uma determinada *feature*, esta deverá ser instalada em qualquer *Node* que pretenda usar o serviço A. Neste cenário o uso do Grupo *Commons* seria então uma forma perfeita de disponibilizar o *interface* do serviço A para que eventualmente qualquer *Node* do *Cluster* possa usar o serviço em si.

Grupo de Serviços

O **Grupo de Serviços** ao contrário do Grupo *Commons* não é único, o que significa que podem existir mais do que um, e idealmente cada Grupo de Serviços deve estar associado a um único *Node* que será responsável pelo mesmo. Assim na eventual falha de um determinado *Node* os Grupos de Serviços em que este é responsável sofrem o processo de *failover* para outro *Node*.

Note-se também que cada um dos Grupos de Serviços possui um conjunto de

features, que apesar de poder ter algumas *features* em comum não deverá. Isto porque no caso de *features* que necessitem de estar em vários *Nodes* simultaneamente poderá ser usado o Grupo *Commons*, e porque no caso de *features* que exponham serviços é de relembrar que estes estarão disponíveis a qualquer membro do *Cluster*, graças ao requisito não-funcional de IAD. Contudo se mesmo assim for necessário que tal aconteça não deverá existir qualquer problema, porque na eventualidade de um *Node* pertencer a grupos com *features* iguais estas não podem ser instaladas duas vezes no *Node*. E porque na remoção de um dos grupos as *features* comuns não serão removidas se ainda forem necessárias a outro grupo.

Então tirando proveito do conceito de grupos do *Cellar*, esta solução utiliza os mesmos para conseguir facilmente mover *features* entre *Nodes* quando assim for necessário. Neste caso os grupos que desempenham este papel são os Grupos de Serviços, configurados e definidos por esta solução para o efeito.

Outra característica definida nesta solução e comum a ambos os grupos, *Commons* e de Serviços, são os repositórios de *features*. O *Cellar* permite definir um conjunto de repositórios que são usados para obter as *features* pretendidas, sendo que cada grupo possui os seus repositórios. Mas no caso desta solução os repositórios definidos serão comuns a qualquer grupo por esta controlado.

Configuração dos Grupos *Commons* e de Serviços

Esclarecido anteriormente, para cada grupo criado no *Cellar* são definidos um conjunto de recursos distribuídos, como uma lista de *features*, e que serão partilhados com os *Nodes* que fazem parte de um determinado grupo.

Por defeito existe uma sincronização constante entre os recursos locais de um *Node* e os recursos distribuídos do grupo. Ou seja a alteração de um recurso distribuído, digamos a instalação de uma *feature* num grupo, levará a que a mesma seja também instalada localmente (de forma automática) em todos os *Nodes* que são membros do grupo. E por outro lado, a instalação de uma *feature* num *Node* suscita a sua instalação nos grupos a que este pertence.

A figura 3.4 exemplifica este comportamento, no que diz respeito à sincronização entre os recursos locais de um *Node* e os recursos do seu grupo. Neste caso a *feature* 1 instalada no *Node* A, tem repercussões tanto nas *features* do Grupo A como nas *features* locais do *Node* B que também é membro do Grupo A. Existindo portanto uma sincronização entre o *Node* A e o Grupo A, e posteriormente entre o Grupo A e o *Node* B.

Seguindo o raciocínio, este comportamento de sincronização de recursos definido por defeito no *Cellar* poderá acabar por gerar uma sincronização exagerada dos recursos

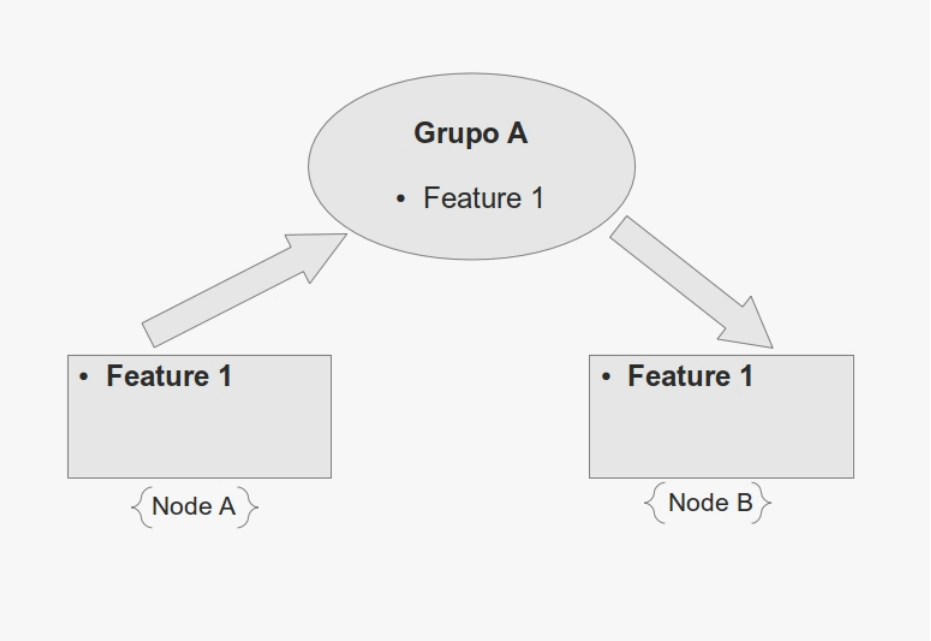

Figura 3.4: Exemplo da sincronização de *features* do *Cellar*

de todos os *Nodes* e grupos de um *Cluster*. A figura 3.5 demonstra tal cenário, em que uma alteração às *features* do *Node* A leva a uma sincronização com grupos a que este nem pertence. Fruto de *Nodes* que pertençam a múltiplos grupos, como é o caso do *Node* B.

Será então bastante fácil de compreender que este comportamento não será compatível com a finalidade assente aos dois tipos de grupos, o Grupo *Commons* e de Serviços. Felizmente o *Cellar* permite configurar tal comportamento para cada um dos grupos, controlando vários aspectos da sincronização efectuada entre os recursos de um grupo e dos seus *Nodes*. Neste caso o comportamento pretendido será semelhante para o Grupo *Commons* e para o Grupo de Serviços, restringindo o conjunto de *features* com as quais o grupo trabalha. Isto é, um grupo deverá trabalhar apenas com um conjunto de *features* definidas previamente para este, ignorando e bloqueando qualquer acção relacionada com uma que esteja fora do conjunto.

Detalhes técnicos de todo este processo de configuração do Grupo *Commons* e do Grupo de Serviços, será demonstrado na secção 4.3.2.

Node Master e *Backup*

Com a definição e configuração do Grupo *Commons* e dos Grupos de Serviços o próximo passo será a atribuição dos mesmos a *Nodes*. Como referido todos os *Nodes* são adicionados de forma automática ao Grupo *Commons*, mas por outro lado os Grupos de Serviços necessitam de ser associados a um e só um *Node* garantindo que não existem

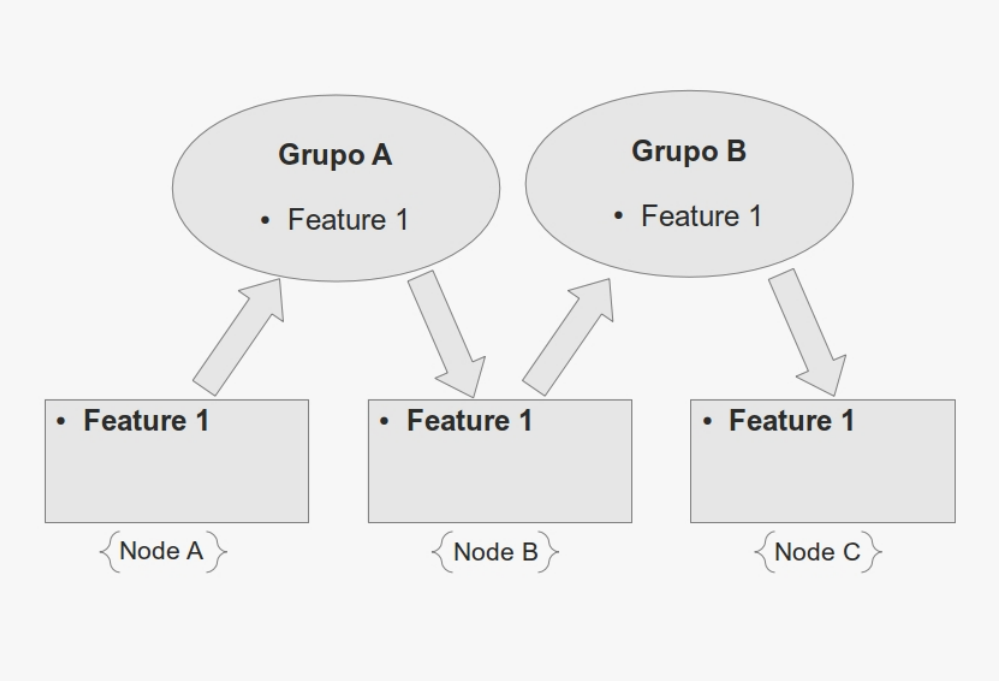

Figura 3.5: Exemplo do problema da sincronização de *features* do *Cellar*

replicações desnecessárias de serviços. E com isto em mente foram definidos os conceitos de *Node Master* e *Node Backup*.

Um ***Node Master*** ou somente ***Master***, será o *Node* responsável por um determinado Grupo de Serviços. Querendo isto dizer que numa situação ideal o único membro de um Grupo de Serviços é o seu *Master*, cabendo a este a expor os serviços do grupo (sendo os serviços implementados e exportados pelo conjunto de *features* do grupo). Note-se ainda que um determinado *Node* poderá ser perfeitamente o *Master* de vários Grupos de Serviços simultaneamente.

Mas como sabemos poderá existir uma eventual falha destes *Masters* deixando os serviços dos seus Grupos de Serviços de ser expostos, o que leva à necessidade de existir um processo de *failover* dos serviços para outro *Node*. A forma mais simples e correcta de o fazer é juntar outro *Node* ao Grupo de Serviços, sendo este denominado de *Node Backup*.

O propósito de um ***Node Backup*** ou somente ***Backup***, é substituir um *Master* para um ou mais dos seus Grupos de Serviços até que este volte a ficar operacional novamente. O que significa que na falha de um *Master*, os Grupos de Serviços pelos quais é responsável serão distribuídos por outros *Nodes* do *Cluster* sendo esta distribuição dinâmica e decidida em tempo de execução. Dizemos assim que um *Node* é definido como Backup para um determinado Grupo de Serviços, e não Backup para um *Master*.

Escolha de *Nodes Backup*

Percebido o conceito de *Node Master* e *Node Backup* e os papeis que cada um tem no processo de *failover*, será necessário esclarecer a forma como são definidos os *Backups*. A qual roda à volta de dois eventos, entrada e saída de *Nodes* do *Cluster*.

Na saída de um *Node* do *Cluster*, seja esta saída propositada ou uma falha do *Node*, este evento é captado por os restantes *Nodes*, os quais tentam definir-se como *Backup* para os Grupos de Serviços em que o *Node* que saiu era *Master*. Assumindo que existem vários *Nodes* a tentar competir por se tornarem um *Backup*, aquele que será definido é o que conseguir modificar primeiro o recurso distribuído que gere a configuração dos *Backups*. No caso de um *Node* não conseguir se tornar *Backup* porque tornou-se primeiro, este vai tentar novamente mas para outro dos Grupos de Serviços do *Node* que saiu e que ainda não tenham um *Backup* atribuído.

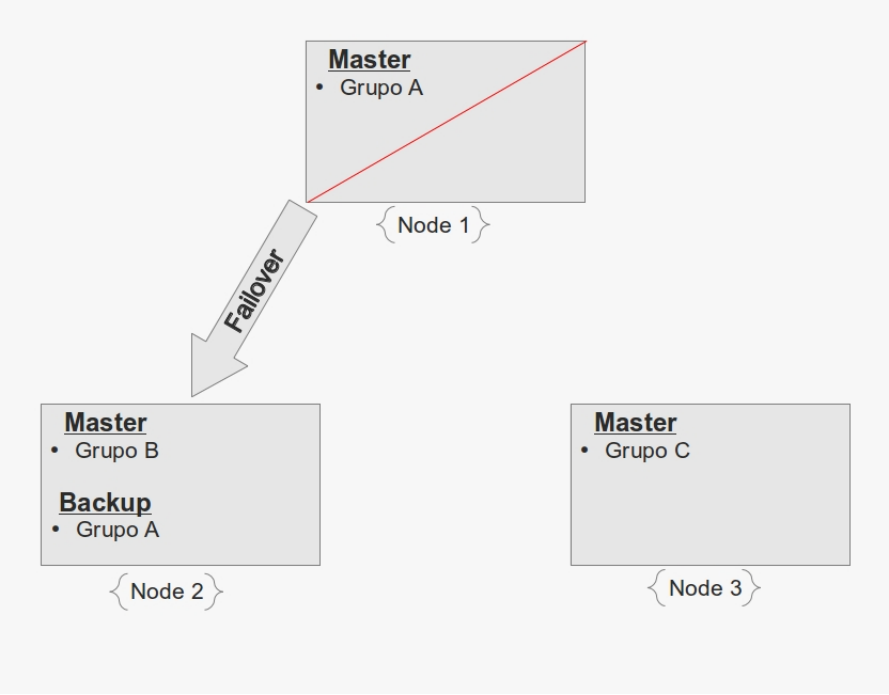

Figura 3.6: Cenário de saída de um *Node* do *Cluster*

A figura 3.6 mostra um simples cenário da saída de um *Node* do *Cluster*, o *Node* 1, sendo este *Master* do Grupo de Serviços A. A quando da sua falha o *Node* 2 e 3 competem pelo único Grupo de Serviços, o Grupo A, ficando o *Node* 2 como *Backup* do grupo.

Esta competição dos *Nodes* por se tornarem um *Backup* acaba por actuar como uma espécie de distribuição de carga, porque primeiro um *Node* menos atarefado em teoria será mais rápido a definir-se como um *Backup* do que um *Node* mais atarefado. E segundo

porque no caso do *Node* que saiu ser um *Master* para múltiplos Grupos de Serviços, a probabilidade de ficarem todos associados a apenas um *Node* é muito reduzida, devendo em princípio ser distribuídos mais ou menos de forma igual entre os *Nodes* existentes.

Note-se ainda que esta solução está desenhada para na saída de um *Node* do *Cluster* além de definir *Backups* para os Grupos de Serviços em que o *Node* que saiu era *Master*, também substitui os *Backups* do mesmo. Isto é, numa situação em que o próprio *Node* que saiu já era *Backup* para determinados Grupos de Serviços, esses mesmos grupos devem também sofrer um processo de *failover* novamente para outro *Node*, tendo assim definido um novo *Backup* como a figura 3.7 exemplifica.

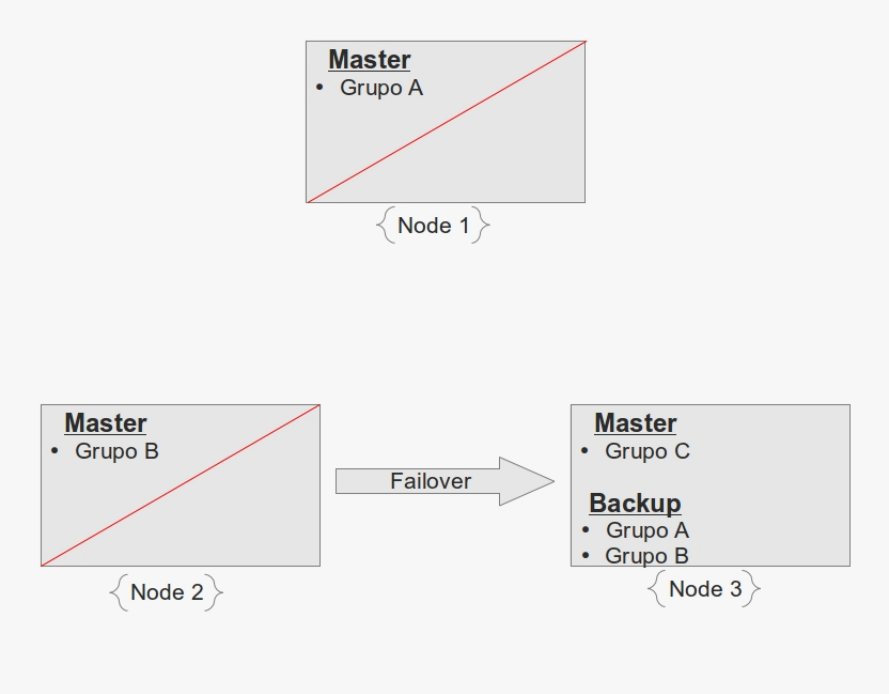

Figura 3.7: Cenário em que o *Node* que sai do *Cluster* tem *Backups*

Novamente, no caso de haver múltiplos *Nodes* a competir por se tornar *Backup* esta substituição segue a mesma lógica anterior, ou seja o *Node* a substituir o *Backup* do *Node* que saiu é o que conseguir modificar primeiro o recurso distribuído que gere as configurações dos *Backups*.

A figura 3.7 retoma o cenário da figura 3.6, mas aqui o *Node* 2 que originalmente se tinha tornado *Backup* para os Grupos de Serviços do *Node* 1, sai agora do *Cluster*. Com o *Node* 1 e 2 fora do *Cluster* e sendo o *Node* 3 o único disponível, este torna-se *Backup* para o Grupo de Serviços A e B, substituindo o *Node* 2 na função de *Backup* do Grupo de Serviços A. Note-se então que se o *Node* 2 volta-se a entrar no *Cluster* não estaria relacionado de algum modo com o Grupo de Serviços A, porque este foi substituído da sua função de

Backup para o grupo em questão.

No outro evento, entrada de um *Node* no *Cluster* será feito o inverso da saída de um *Node*, ou seja serão removidos os *Backups* criados para os Grupos de Serviços em que o *Node* que entrou é *Master*. Contudo apenas o *Node* que é *Backup* para um determinado Grupo de Serviços é que pode tomar a iniciativa de o remover, quando receber o evento de entrada no *Cluster* do *Node* que é *Master* para o grupo.

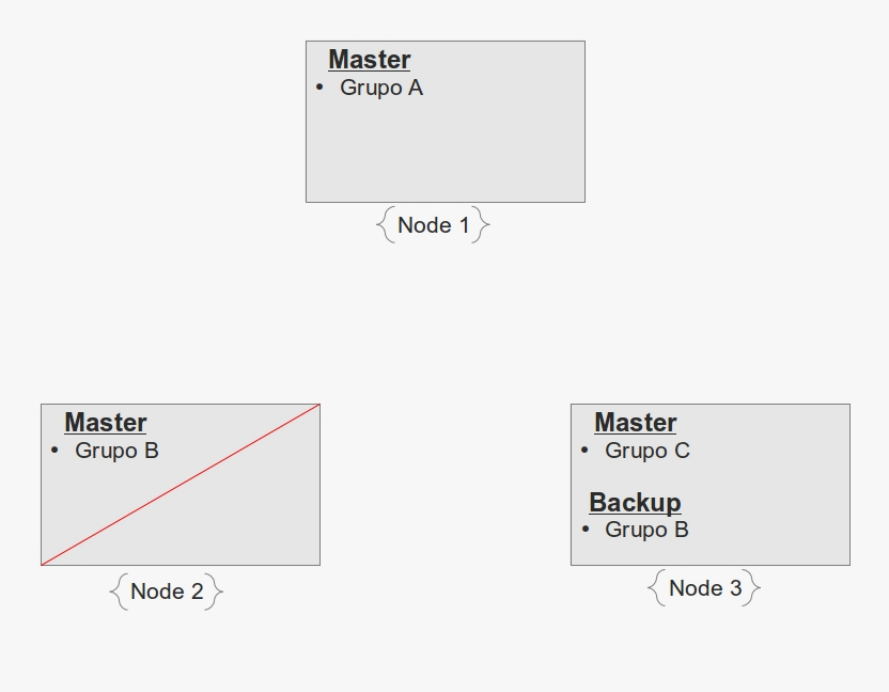

Figura 3.8: Cenário em que um *Node* entra no *Cluster*

Na figura 3.8 ocorre a entrada do *Node* 1 no *Cluster*, e como tal o *Node* 3 perde o *Backup* de todos os Grupos de Serviços em que o *Node* 1 era *Master*. Como o *Node* 3 era o detentor do *Backup* para o Grupo de Serviços A, só este pode removê-lo e modificar o recurso distribuído que gere a configuração dos *Backups*.

3.4 Ambiente de Desenvolvimento

Nesta última secção do capítulo façamos uma breve análise do ambiente de desenvolvimento deste projecto e que permitirá concretizar os requisitos não-funcionais debatidos conceptualmente. Efectuando assim um resumo de todas as tecnologias utilizadas identificando o seu objectivo e propósito, algumas já foram mencionadas e outras são em secções futuras. As seguintes são as principais tecnologias utilizadas, e o Apêndice B específica as suas versões.

OSGi A *framework OSGi* que é base de trabalho de todas as tecnologias do *Fuse ESB*, incluindo o *Cellar* e a solução de *failover* desenvolvida. E graças à sua modularidade e forma como permite a comunicação entre esses módulos, aplicações será imprescindível para elaborar os requisitos de DA e IAD, respectivamente.

Karaf Uma ferramenta base utilizada pelo *Fuse ESB*, actuando como uma shell, um invólucro da *OSGi framework* e disponibilizando uma série de utilidades de interacção com a mesma.

Cellar Uma solução de *clustering* criada no âmbito do *Karaf* e que permite realizar a gestão de um *Cluster* de instâncias do *Karaf*. Esta é ferramenta principal no desenvolvimento dos dois requisitos não-funcionais do presente trabalho.

Hazelcast Solução de *clustering* e distribuição de informação, e que é utilizada como base pelo *Cellar*. Neste trabalho é utilizada também directamente para permitir a distribuição de informação entre todos os *Nodes* do *Cluster*.

***Spring Dynamic Modules* (*DM*)** Uma extensão da *framework Spring*[2] permitindo trazer as suas vantagens/características para um ambiente *OSGi* e facilitando a interacção com o mesmo. E apesar de não ser fundamental para alcançar os objectivos deste projecto é certamente vantajosa.

Finalmente é importante alertar para a decisão que foi tomada em utilizar apenas o *Karaf* em vez do *Fuse ESB,* no desenvolvimento propriamente dito dos dois requisitos enunciados. Decisão que foi proporcionada essencialmente por dois motivos.

O primeiro motivo e o mais forte é que a versão do *Cellar* utilizada nesta Dissertação, que é até ao momento a versão estável mais recente, depende de uma versão do *Karaf* que ainda não se encontra disponível nas versões existentes do *Fuse ESB*. E como tal existe aqui um problema de dependências, sendo de momento apenas possível correr o *Cellar* e consequentemente a solução de *failover* aqui desenvolvida numa instância do *Karaf*. No caso do requisito não-funcional de IAD é possível utilizá-lo com uma versão do *Cellar* mais antiga compatível com a do *Fuse ESB* definida no Apêndice B, mas considerando que para o requisito de disponibilidade o mesmo já não é verdade foi decidido trabalhar os dois no mesmo ambiente.

O segundo motivo é que limitando o desenvolvimento dos dois requisitos apenas ao *Karaf* em vez do *Fuse ESB*, será possível limitar o número de variáveis que os possam

[2] *Spring* é uma *framework* utilizada no desenvolvimento de aplicações *Java* e conhecida por trabalhar conceitos como: injecção de dependências, inversão de controlo e programação orientada a aspectos.

influenciar. Criando assim um ambiente mais controlável, mas sabendo perfeitamente que assim que exista uma versão do *Fuse ESB* compatível com a do *Cellar* utilizada neste trabalho será possível implementar os dois requisitos não-funcionais num completo *ESB*.

3.5 Conclusões

Resumindo, neste capítulo realizou-se uma análise conceptual dos requisitos não-funcionais de IAD e DA percebendo como é que estes podem-se tornar parte do *Fuse ESB*. E onde a tecnologia *Cellar* foi determinante para a sua implementação, pois no caso do primeiro requisito esta implementa-o completamente e para o segundo disponibiliza um conjunto de funcionalidades e conceitos, bem como o próprio requisito de IAD.

Assim o que o requisito não-funcional de IAD propõe fazer, é através da definição ou elaboração de *Cluster* de *Fuse ESB*s permitir que as aplicações de cada um deles consigam comunicar de uma forma simples e transparente. Como estivessem apenas contidas num único *Fuse ESB*, sendo esta a ilusão criada do ponto de vista das aplicações e não havendo qualquer razão para que estas estejam cientes de que estão a comunicar com outras de um *Cluster*. Isto é alcançado com uma espécie de extensão do funcionamento natural da *framework OSGi*. Por defeito a comunicação entre aplicações contidas nela ocorre localmente, porque a própria *framework OSGi* opera exclusivamente numa única JVM. Sendo o intuito do requisito de IAD permitir que um serviço *OSGi* (que é a forma pela qual as aplicações comunicam) seja exposto não só localmente, mas também em todas as *frameworks OSGi* dos *Fuse ESBs* do *Cluster*. Podendo este serviço ser invocado remotamente, mas como se trata-se de um serviço local. A solução escolhida para criar este ambiente colaborativo de *frameworks OSGi* e que melhor se enquadra no *Fuse ESB* foi o *Cellar*, como mencionado. E que estando presente em todos os *Fuse ESB*s do *Cluster*, vai-se responsabilizar em permitir esta comunicação transparente e criando a ilusão de um ambiente único.

Por outro lado o requisito não-funcional de DA pretende garantir a disponibilidade destas aplicações contidas nas *frameworks OSGi* dos *Fuse ESBs* do *Cluster*, na eventualidade de ocorrer falha de um destes. Isto é feito movimentando as aplicações de um *Fuse ESB* que falhou para outro disponível e o inverso quando o original recuperar, um processo ao qual chama-mos de *failover*. Não sendo o funcionamento das aplicações que sofreram este processo afectado por conseguirem perfeitamente comunicar com outras graças ao requisito não-funcional previamente discutido. E assim foi apresentada extensivamente a solução de *failover* definida, que se baseia e aproveita os conceitos do *Cellar* e do requisito de IAD para conseguir implementar o de DA.

É então com esta análise conceptual dos requisitos não-funcionais de IAD e DA e também com a definição do ambiente de desenvolvimento que vai ser possível partir para a sua implementação. Algo detalhado no próximo capítulo.

Capítulo 4

Implementação dos Requisitos Não-Funcionais de IAD e DA na Tecnologia *Fuse ESB*

4.1 Introdução

Este capítulo apresenta todos os detalhes de cariz puramente técnicos relativos à implementação dos requisitos não-funcionais de IAD e DA, e consoante o que foi descrito no capítulo 3.

Para o requisito de IAD a apresentação da sua implementação é bastante simples, devido ao facto de ser completamente suportado pelo *Cellar*. E portanto resume-se à discussão de como deve ser efectuada a exposição e consumo de um serviço *OSGi* distribuído num *Cluster* de *Fuse ESB*s. Bem como alguns cuidados, aspectos a ter em mente na utilização deste tipo de serviços.

No que toca ao requisito de DA e visto que para este foi definida uma solução de *failover* para o implementar, esta será analisada detalhadamente. Inicialmente é descrita a sua arquitectura, os seus componentes e o papel destes no processo de *failover*, relacionando-se obviamente com os conceitos da solução previamente esclarecidos na secção 3.3. Posteriormente são analisados os detalhes por de trás da configuração da solução, onde primeiro é explicado como é que ela automaticamente configura determinados recursos do *Cellar* e depois como é que se define que funcionalidades/aplicações estão associadas a um determinado *Fuse ESB*. Finalmente são enunciados alguns problemas encontrados no desenvolvimento da solução e como foram controlados.

4.2 Interoperabilidade Aplicacional Distribuída

Discutido e apresentado conceptualmente o requisito não-funcional de IAD no capítulo 3, entramos agora em detalhes sobre a sua aplicação e uso no *Fuse ESB*. Relembrando que o objectivo, que o propósito do mesmo requisito era expor e consumir serviços entre diferentes *Fuse ESB*s de um *Cluster*, ou para ser mais concreto serviços *OSGi* de diferentes f*rameworks OSGi*. Esta secção tem exactamente este fim, que é demonstrar como expor e consumir este tipo de serviços num *Cluster* de *Fuse ESB*s, recorrendo ao *Cellar*. Que como será percebido em muito pouco varia relativamente a um normal serviço *OSGi*. Alem disto também são apresentados alguns detalhes sobre cuidados a ter na utilização deste mesmo requisito não-funcional.

Adicionalmente a aplicação de tudo descrito nesta secção sobre o requisito de IAD e na futura sobre o requisito de disponibilidade requer uma configuração apropriada do *Cellar*. O Apêndice A detalha tal procedimento.

4.2.1 Exposição de Serviços

Na exposição de um serviço *OSGi* e independentemente da sua natureza, isto é se estamos a falar de um serviço distribuído ou não, é possível adicionar um conjunto de propriedades ao serviço. Propriedades essas que são usadas como meta-dados sobre o serviço, e que podem ser definidas por quem expõe o serviço detalhando vários tipos de aspectos. E é então através do uso de uma destas propriedades que se define um serviço a ser exposto no *Cluster*. Pois cada vez que um serviço *OSGi* é registado no *Service Registry* com a propriedade `service.exported.interfaces` e esta com o valor que será o conjunto de *interfaces* pretendidos expor como serviços remotos ou `*` para indicar que serão todos os *interfaces*, este passará a ser acessível aos outros membros do *Cluster*. A figura 4.1 demonstra tal exposição de um serviço exemplo, feita de forma declarativa através da tecnologia *Spring DM*.

```
<osgi:service ref="cityService" interface="example.service.CityService">
    <osgi:service-properties>
        <entry key="service.exported.interfaces" value="*" />
    </osgi:service-properties>
</osgi:service>
```

Figura 4.1: Exemplo da exposição de um serviço para o *Cluster*, via *Spring DM*

O que sucede nos bastidores é que assim que o *Cellar* recebe um evento da *framework OSGi* anunciando a exposição de um serviço (que neste momento é apenas uma exposição local, ou seja numa só *framework*) com a propriedade discutida, este vai tratar de o expor remotamente a todos os outros *Nodes* do *Cluster*. A maneira como o concretiza sem entrar em grandes detalhes, é criando um registo do serviço em todas as outras *frameworks OSGi*. Porém como é de esperar este registo não aponta a um implementação real do serviço, porque tal não existe localmente, mas sim a um *proxy* que reencaminha pedidos para o *Node* onde a implementação do serviço de facto existe. Criando a ilusão que o serviço está exposto localmente quando não está.

É importante ainda esclarecer um outro pequeno pormenor para não causar confusão. Um serviço que foi exposto para o *Cluster*, obviamente também estará acessível a *bundles* locais. Ou seja a *bundles* que estão na mesma *framework OSGi* que o *bundle* que expõe o serviço. Tal facto é mais uma prova da transparência e simplicidade com a qual o *Cellar* implementa o requisito não-funcional de IAD. Tanto na exposição de serviços como no consumo de serviços, algo explanado de seguida.

4.2.2 Consumo de Serviços

Quanto ao consumo de serviços remotos, pode se adivinhar por tudo aquilo que foi afirmado que o processo é ainda mais trivial. Pois ao contrário do que acontece com a exposição de serviços, que requer o uso de uma simples propriedade indicando que o serviço será exposto no *Cluster*, o consumo de um serviço não requer qualquer alteração em relação à forma normal de um consumir um serviço *OSGi*.

```
<osgi:reference id="cityService"
    interface="example.service.CityService" />
```

Figura 4.2: Exemplo do consumo de um serviço *OSGi*, via *Spring DM*

A figura 4.2 apresenta o consumo de um serviço exemplo e que é o mesmo serviço apresentado na figura 4.1. Este processo é de tal maneira transparente que é impossível distinguir apenas pela invocação do serviço se este é remoto ou local, o que não é de admirar se considerarmos o que foi dito anteriormente. Isto é, do ponto de vista de quem consome o serviço e na prática encontra-se na *framework OSGi* e perfeitamente acessível. Contudo como é sabido não corresponde ao serviço real, mas sim a um apontador para o mesmo.

E é este facto que suscita a necessidade da haver um certo cuidado na utilização de serviços remotos, para que a própria exposição e consumo possam ser efectuadas de uma forma correcta. Algo que será abordado de seguida.

4.2.3 Cuidados na Utilização

Apresentado como utilizar o requisito não-funcional de IAD é fácil constatar a simplicidade que o *Cellar* traz ao mesmo. Contudo será relevante enunciar alguns aspectos relativos à utilização deste requisito, e que apesar de parecerem bastante óbvios não devem ser esquecidos.

Construção de *Bundles*

Um ponto no qual deve haver um algum cuidado em relação ao requisito de IAD é a construção de aplicações, e neste caso em particular a construção de *bundles*. O que isto quer dizer é que se os serviços *OSGi* são o canal de comunicação/interacção de *bundles* e o objectivo do requisito é permitir esta interacção num ambiente altamente dinâmico, é necessário ter cuidado com tal dinamismo. Aqui por dinamismo entende-se o facto de determinados serviços entrarem e saírem da *framework OSGi* ao longo do tempo, podendo haver alturas que por exemplo um *bundle* necessita de um determinado serviço que não se

encontra disponível.

Este dinamismo é algo bastante natural e que faz parte do conceito por de trás da *framework OSGi*, tanto que num ambiente local existem mecanismos da própria *framework* ou de tecnologias complementares como é caso do *Spring DM* que lidam com este dinamismo. A título de exemplo, com o uso do *Spring DM* na invocação de um serviço que não se encontra disponível, o pedido é bloqueado até que volte a ficar disponível.

Assim que passamos para um ambiente distribuído, para um *Cluster* onde existe o requisito não-funcional de IAD e lidamos com serviços remotos começam a surgir problemas. Isto no caso do uso do *Cellar* e na ausência de um serviço remoto, este vai aparecer como disponível devido à forma como o *Cellar* expõe serviços remotos. Mas como não estão disponíveis se forem acedidos vão provavelmente ocorrer excepções . Tudo porque na prática o serviço aparenta estar disponível mas é apenas um apontador como explicado anteriormente na secção 4.2.1. Esta situação e os problema que podem provocar serão melhor compreendidos com aquilo que é discutido na secção 4.3.4.

Resumindo, a ideia importante em salientar é o cuidado a ter na criação de *bundles* que lidem com serviços remotos. Ponderando que os serviços podem não estar disponíveis e na opinião do autor deste documento os *bundles* devem antecipar e estar preparados para tais situações.

Objectos Serializáveis

Um dos aspectos a ter em conta no que toca ao uso do requisito de IAD é que um determinado serviço *OSGi* exposto para o *Cluster*, não é mais do que um método *Java*, que deve trabalhar com objectos de classes que sejam serializáveis. Algo obviamente necessário quando lidamos com um *Cluster*, no qual existem várias *JVM*s.

No caso do *Cellar* estas classes devem preferencialmente cumprir as seguintes regras:

- Implementar o *interface DataSerializable* do *Hazelcast*. O qual o *Cellar* usa para realizar a transmissão de objectos entre membros do *Cluster*.
- Definir o campo `serialVersionUID`.
- Fazer *override* ao método `equals()` e `hashCode` da classe *Object* do *Java*.

Estas regras não referem algo que não seja comum à programação *Java* em ambientes distribuídos, mas que é necessário ter em mente quando for utilizado o requisito não de IAD.

Uso Correcto de um Serviço Remoto

Um último aspecto a ter cuidado no uso deste tipo de serviços que poderá parecer trivial mas que não deve deixar se ser mencionado, é que estes devem ser usados correctamente com recurso aos parâmetros e retorno do método, que é o serviço. Porque enquanto que num ambiente *OSGi* tradicional, por se situar numa única *JVM*, é possível passar um objecto a um determinado serviço e depois aceder a esse mesmo objecto através da sua referência sem explicitamente utilizar o retorno do serviço, o mesmo já não é verdade para um serviço exposto no *Cluster*.

Assim é importante alertar que interacções com serviços *OSGi*, no âmbito do requisito não-funcional de IAD, devem ser exclusivamente feitas através dos parâmetros e retorno do método remoto. Pois se estamos a trabalhar num *Cluster* de *Fuse ESB*s, estamos também a lidar com *JVM*s diferentes e com conjuntos de objectos diferentes, sendo portanto esta a única e a forma correcta de trabalhar objectos num ambiente distribuído.

4.3 Disponibilidade Aplicacional

Nesta secção será explanada a implementação neste trabalho do requisito não-funcional de DA tal como discutido, mais concretamente da solução de *failover*. A presente secção pretende detalhar todos os aspectos considerados e efectuados no desenvolvimento da solução, e que serão fundamentais para perceber como aplica todos os conceitos enunciados na secção 3.3.

Inicialmente é apresentada a arquitectura da solução concebida e onde o foco incide nos componentes da mesma, ou seja as principais classes que a compõem. Para cada uma são analisados os seus objectivos e os detalhes relevantes de como os atinge. Note-se também que a solução deve ser aplicada em todos os *Fuse ESBs* ou *Nodes* do *Cluster*, sendo que esta está preparada e pensada para trabalhar com esse facto. Isto é, o funcionamento de várias instâncias da solução e de várias instâncias dos mesmos componentes (mesmas instâncias em diferentes *Nodes*), foi previsto na concepção da solução e cria assim um ambiente distribuído onde cada instância da solução actuará como uma espécie de representante do *Node* no que toca ao requisito de DA. Como tal as acções de cada um dos componentes da solução será sempre em do seu *Node* do *Cluster* e nunca afectando directamente outros.

Posteriormente são analisados pormenores de como é efectuada a configuração da solução. Na perspectiva de como é efectuada por parte da solução a configuração dos grupos do *Cellar*, pois esta é feita de forma automática com base em informação dos grupos. E como é que são configurados, de forma declarativa, os detalhes dos grupos da solução como

é caso das suas *features* e que *Nodes* são responsáveis por eles.

Finalmente será dedicada uma secção a enunciar os problemas encontrados no desenvolvimento da solução, de uma forma simplista sem querer entrar em grandes detalhes de cariz técnico, e de como foram concebidas correcções para esses problemas. Correcções essas que são aplicadas pelos próprios componentes da solução.

4.3.1 Arquitectura da Solução de *Failover*

Tendo em conta que esta solução de *failover* foi desenvolvida para ser usada na *framework OSGi* fará todo sentido tirar proveito das suas capacidades/funcionalidades, mais concretamente da sua forte preocupação com a modularidade. Consequentemente esta solução foi divida em dois *OSGi bundles*, permitindo dividir tarefas e preocupações em duas grandes partes.

O primeiro *bundle*, **failover.core**, foi desenvolvido com dois grandes propósitos. Ser dentro da solução uma forma simples, correcta e única de todos os restantes componentes interagirem (de forma indirecta) com as tecnologias que a solução utiliza. Como exemplo disso temos a instalação e remoção de *features* nos grupos do *Cellar*, remoção de *features* do *Karaf*, entre outros.

E também que seja este *bundle* o responsável em configurar todas essas tecnologias utilizadas por esta solução, alcançando o ambiente correcto e necessário para executar todo o processo de *failover*. Como se trata da criação dos grupos definidos por esta solução, configuração das *features* de cada grupo, etc.

O segundo *bundle*, **failover.node**, através do *bundle* failover.core irá preocupar-se em estabelecer o papel de cada *Node* do *Cluster* envolvidos no processo de *failover*. Seja no momento em que um *Node* é associado ao Grupo *Commons* e aos Grupos de Serviços em que é *Master*, quando o mesmo é definido como Backup para um determinado Grupo de Serviços ou quando deixa de ser *Backup*.

Face a esta divisão da solução em dois *bundles* apresentam-se de seguida os diagramas de classes para cada um dos mesmos. As classes destes diagramas não contêm atributos ou operações para uma mais fácil visualização, sendo que os diagramas completos encontram-se nos Apêndice C.

Como pode ser averiguado na figura 4.3 e 4.4, ambos os *bundles* foram concebidos através do forte uso de interfaces na interacção de classes dentro e fora do mesmo *bundle*. Isto é feito para evitar que na eventualidade de ser introduzida uma nova implementação de um interface, seja necessário efectuar grandes alterações, e também para que os *bundles* da solução consigam expor as suas funcionalidades de forma simples via serviços *OSGi*. Como

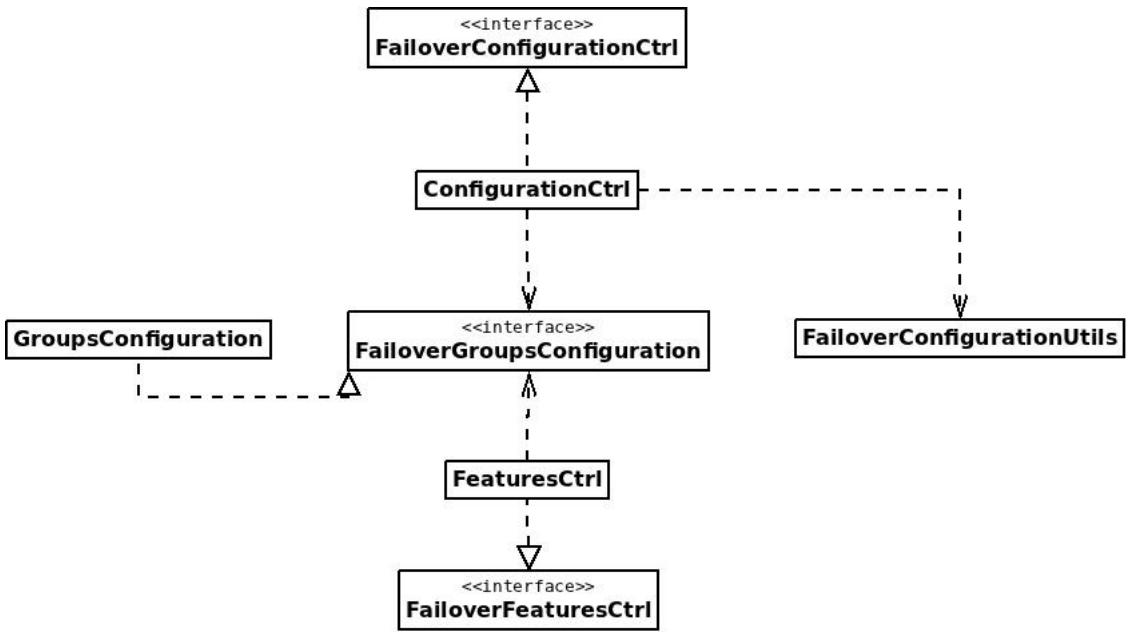

Figura 4.3: Diagrama de classes simplificado do *OSGi bundle* failover.core

é o caso do *bundle* failover.core, que expõe por exemplo o serviço "FailoverFeaturesCtrl", que por sua vez é consumido pelo *bundle* failover.node.

Note-se ainda que para todas as classes apresentadas em ambos os diagramas, da figura 4.3 e 4.4 , apenas poderá existir uma instância de cada ou seja, estas implementam o padrão *singleton*. A implementação deste padrão advém do uso da tecnologia *Spring DM* que permite definir as classes que gere como *singleton*, e visto que não existe necessidade em qualquer situação de ter mais do que uma instância destas classes, a implementação deste padrão vê-se como bastante vantajosa.

De seguida serão discutidas todas as classes da figura 4.3 e 4.4 de uma perspectiva mais técnica, percebendo com detalhe o papel que cada uma desempenha em todo o processo de *failover*.

GroupsConfiguration

A classe **GroupsConfiguration** é responsável por definir e deter a informação de cada um dos grupos envolvidos no processo de *failover*. Informação essa que se refere à definição do Grupo *Commons*, dos Grupos de Serviços, das *features* e repositórios destes. A figura 4.5 apresenta um exemplo da configuração desta classe efectuada via *Spring*, com o Grupo *Commons*, dois Grupos de Serviços e respectivas *features*. Os detalhes de como efectuar a configuração dos grupos serão explicados mais à frente noutra secção.

Esta classe actua assim como uma fonte de onde as restantes vão beber informação

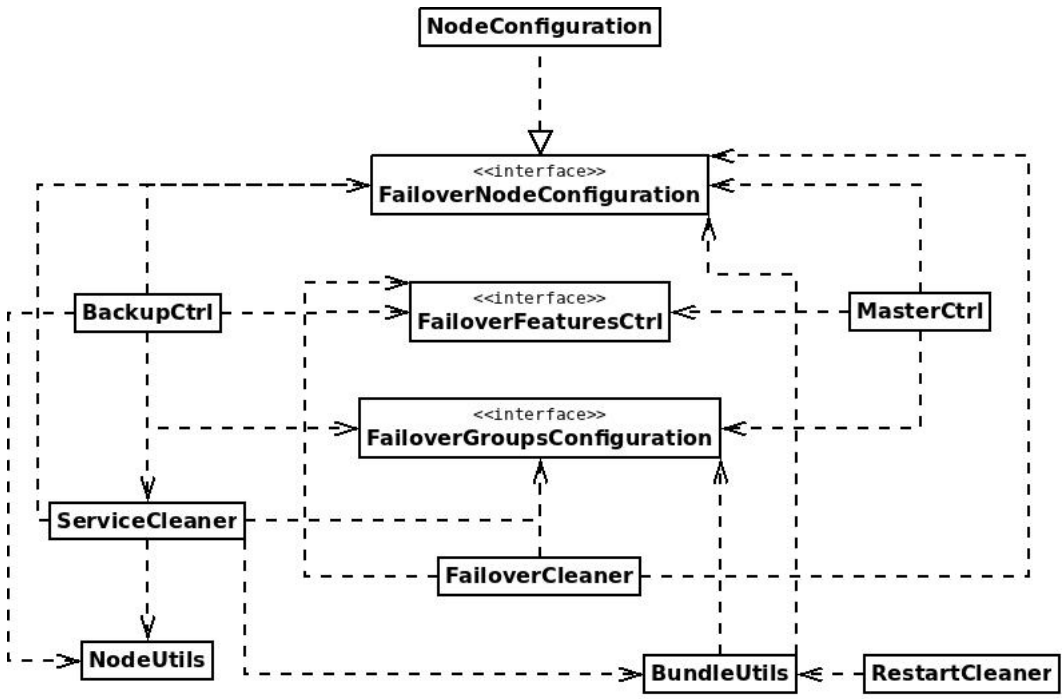

Figura 4.4: Diagrama de classes simplificado do *OSGi bundle* failover.node

sobre os grupos configurados para a solução. Seja esta informação a referência aos próprios grupos, *features* ou repositórios dos mesmos.

Finalmente é importante salientar que esta classe implementa o *interface* **FailoverGroupsConfiguration**, o qual foi baseado nas características desta classe e que é usado pelas restantes através de um serviço *OSGi*.

ConfigurationCtrl

A classe **ConfigurationCtrl,** que implementa o *interface* **FailoverConfigurationCtrl** e na qual este se baseia, foi concebida para se preocupar com a configuração dos grupos do *Cellar* usados por esta solução. Configuração esta que não se refere apenas às *features* dos grupos, mas também aos seus repositórios.

Os grupos configurados por esta classe são todos aqueles definidos na classe GroupsConfiguration, e como esclarecido na secção 3.3.2 a configuração destes grupos é baseada nas *features* atribuídas a cada um deles, ignorando e bloqueando quaisquer outras. Informação também disponibilizada por GroupsConfiguration. Esta configuração partilhada por todos os *Nodes* do *Cluster* é efectuada via um dos serviços da *framework OSGi*, o *OSGi* Configuration Admin.

As acções desta classe resumem-se então à configuração dos grupos quando a

```
<bean id="groupsConfiguration" class="bosch.failover.core.impl.GroupsConfiguration">
    <property name="commonsGroup" value="#{groupManager.createGroup('commons')}" />
    <property name="commonsFeatures">
        <set>
            <value>eventadmin</value>
        </set>
    </property>
    <property name="serviceGroupsConfiguration">
        <map>
            <entry key="#{groupManager.createGroup('grupoA')}">
                <set>
                    <value>example.city.server</value>
                </set>
            </entry>
            <entry key="#{groupManager.createGroup('grupoB')}">
                <set>
                    <value>example.city.city</value>
                </set>
            </entry>
        </map>
    </property>
    <property name="featuresRepositories">
        <set>
            <value>mvn:org.apache.karaf.cellar/apache-karaf-cellar/2.2.4/xml/features</value>
            <value>mvn:org.apache.karaf.assemblies.features/enterprise/2.2.7/xml/features</value>
            <value>mvn:org.apache.karaf.assemblies.features/standard/2.2.7/xml/features</value>
        </set>
    </property>
</bean>
```

Figura 4.5: Excerto da configuração do *bundle* failover.core

mesma é inicializada. Contudo existe um pormenor importante destacar e o qual está relacionado com um problema encontrado e melhor enunciado na secção 4.3.4. Este pormenor refere-se à elaboração de uma correcção desse problema e que consiste em verificar se as configurações foram realizadas correctamente. Se na verificação for detectada alguma incoerência, será efectuada novamente a configuração para tentar corrigir o problema.

Esta verificação e correcção ocorrerá várias vezes até que a configuração seja realizada com sucesso ou até ser atingido um determinado limite. Ultrapassado este limite, e se a configuração de qualquer um dos grupos continuar incorrecta é lançada uma excepção que leva o *bundle* failover.core a passar a um estado de falha. O que indica que esta solução de *failover* não funcionará correctamente se a classe ConfigurationCtrl não conseguir completar as suas configurações.

Finalmente, não será demais referir que a classe ConfigurationCtrl depende da classe FailoverConfigurationUtils para efectuar as configurações dos grupos. A última serve apenas para dar suporte à configuração de grupos do *Cellar* via *OSGi* Configuration Admin, disponibilizando constantes e funções.

FeaturesCtrl

A última classe a discutir do *bundle* failover.core é a **FeaturesCtrl**. Esta é responsável por todas as operações desta solução que estão relacionadas com *features*. O que inclui:

- instalação de *features* para o Grupo *Commons* e dos Grupos de Serviços
- instalação das *features* dos grupos localmente, ou seja no *Karaf* e *fram*ework *OSGi* de cada *Node* do *Cluster*
- remoção de *features* locais
- recolha das *features* instaladas localmente

Esta classe implementando o *interface* **FailoverFeaturesCtrl**, disponibiliza assim a todas as outras as classes desta solução uma forma simples e correcta de realizar instalação e remoção de *features* localmente, algo esclarecido ainda melhor de seguida.

Como toda a lógica desta solução de *failover* assenta no uso de grupos do *Cellar* e da possibilidade de definir *features* específicas para cada, a primeira preocupação desta classe é instalar correctamente as *features* de cada um dos grupos. Algo que efectua assim que é inicializada. Instaladas as *features* correctamente, outras classes podem usar esta (preferencialmente via o seu *interface* FailoverFeaturesCtrl) para instalar *features* desses grupos localmente ou remover um determinado conjunto de *features*. A instalação das *features* dos grupos geridos por esta solução num determinado *Node* é feita com recurso ao método pullGroup(), da classe FeaturesCtrl. A invocação deste método vai sincronizar toda a informação do *Node* local com a do grupo especificado. Ou por outras palavras a informação do Grupo, que engloba as suas *features* e configurações, será copiada das fontes de informação distribuídas geridas pelo *Cellar* para o *Node* local. Levando então a que as *features* do grupo sejam instaladas localmente, se já não estiverem.

Note-se também que este método só funcionará se o *Node* onde é invocado pertencer ao grupo especificado e se alguma das *features* do grupo não estiver de encontra as configurações do mesmo, esta não será instalada. Isto porque se o *Cellar* tentar instalar uma *feature* num grupo ou instalar uma *feature* de um grupo num *Node*, este primeiro vai verificar se a *feature* é permitida. Se esta não for permitida, algo que é definido nas configurações do grupo, obviamente não será instalada localmente.

Pode parecer esquisito que um grupo tenha instalada uma *feature*, que nas suas configurações esta é bloqueada. Contudo isto advém do facto que nesta solução de *failover*

não é possível através da classe FeaturesCtrl, ou qualquer outra, remover *features* do Grupo *Commons* ou dos Grupos de Serviços. Esta decisão foi tomada para evitar que alterações nas *features* de um grupo (isto é à classe GroupsConfiguration), tivessem efeitos negativos em *Nodes* que ainda necessitem dessas *features*. Pois se fosse invocada a remoção de uma *feature* de um grupo e nesse momento existissem *Nodes* que eram membros desse grupo, isto levaria a que a mesma fosse removida não só do grupo como do conjunto de *features* locais de cada um dos *Nodes*, podendo levar a problemas de dependências. Para evitar estes possíveis problemas, alterações às *features* dos grupos controlados por esta solução não vão levar a que sejam removidas do grupo, mas sim bloqueadas. O que implica que se um *Node* estiver associado ao grupo não vai ver a *feature* removida localmente, algo que se for pretendido terá que ser feito manualmente. E para *Nodes* que entrem no grupo futuramente não vão ver a *feature* instalada porque está bloqueada.

Contudo e apesar da existência desta situação, é importante focar que esta solução não foi concebida para que as configurações que definem as *features* dos grupos fossem alteradas com frequência. Mesmo assim é possível recorrer ainda à classe FeaturesCtrl para remover um conjunto de *features* do *Node* local com o método removeLocalFeatures(). Que será utilizado principalmente quando um *Node* deixa de ser Backup, algo abordado mais à frente.

NodeConfiguration

Definida no *bundle* failover.node, a classe **NodeConfiguration** está encarregue em definir, deter e disponibilizar a informação sobre que grupos, dos Grupos de Serviços geridos nesta solução, estão atribuídos a cada um dos *Nodes* do *Cluster*. Seja esta informação relativa aos *Nodes Master* ou *Nodes Backup*.

Esta classe implementa o *interface* **FailoverNodeConfiguration** o qual é usado pelas outras classes do *bundle* failover.node, e que também pode ser usado por outras classes de outros *bundles* visto que é exposta como um serviço *OSGi*.

A definição do *Node* que é *Master* para cada um dos Grupos de Serviços é estática e feita através da configuração *Spring* da classe NodeConfiguration, como ilustrado na figura 4.6. Esta configuração é definida através de um mapa em que a chave é o nome do Grupo de Serviços e o valor o identificador do *Node*, sendo possível que um determinado *Node* seja *Master* para múltiplos Grupos de Serviços.

Pelo contrário a definição dos *Nodes Backup* é feita dinamicamente e durante o período de execução da solução, à medida que *Nodes* vão saindo e entrando no *Cluster*. E também como no caso dos *Nodes Master* os *Nodes Backups* são definidos através de um

```
<bean id="nodeConfiguration" class="bosch.failover.node.impl.NodeConfiguration">
    <property name="masterNodesConfiguration">
        <props>
            <prop key="grupoA">alex-desktop.home:5701</prop>
            <prop key="grupoB">alex-laptop.home:5701</prop>
        </props>
    </property>

    ...
```

Figura 4.6: Excerto da configuração do *bundle* failover.node

mapa, contudo este é distribuído, obtido via *Hazelcast* e acessível a todas as instâncias da classe NodeConfiguration. Possibilitando a qualquer *Node* do *Cluster* saber que *Nodes* são *Backup* para que Grupos de Serviços. O mapa segue a mesma lógica anterior, em que a chave do mapa é o nome do Grupo de Serviços e o valor o identificador do *Node*, podendo o mesmo *Node* ser *Backup* para vários grupos.

Finalmente é necessário apontar que esta classe disponibiliza mais duas utilidades. Primeiro dispõe a referência ao *Node* local, ficando a mesma facilmente acessível a outros que a necessitem. A disponibilização desta referência foi feita apenas como uma comodidade para com as restantes classes do *bundle* failover.node, mas que também está disponível a outros *bundles* via o *interface* FailoverNodeConfiguration que é exposto como um serviço *OSGi*.

Segundo, visto que esta classe lida com um mapa distribuído e para garantir que não existem problemas de concorrência no acesso a este mapa por parte dos vários *Nodes* do *Cluster*, em todas as operações desta solução que incidem sobre este mapa os *Nodes* adquirem primeiro um *lock* do recurso. Seja este *lock* a apenas uma chave e valor do mapa ou ao mapa todo.

No processo de obter o *lock* dos recursos distribuídos é também necessário apontar o tempo máximo de espera até conseguir obter o *lock* de um determinado recurso. Havendo sempre a possibilidade de o tempo de espera ser esgotado e o *Node* não conseguir aceder ao recurso distribuído.

Esta classe define uma variável (`lockTimeout`) que representa em segundos o tempo de espera pelo *lock* de um recurso. Esta variável é usada na mesma classe em que é definida, NodeConfiguration, e por outras do *bundle* failover.node como um tempo de espera comum. E que tal como no caso da referência ao *Node* local mencionada anteriormente, esta variável também é exposta para fora do *bundle* failover.node.

FailoverCleaner

A classe **FailoverCleaner** foi concebida para efectuar uma limpeza a todos os *Nodes* do *Cluster*. Esta limpeza será realizada antes de serem atribuídos quaisquer grupos e *features* ao *Node* local, e consiste em remover aquilo que não está correcto. Ou seja os grupos e *features* que não devem estar associados com o *Node* local, de acordo com as configurações definidas em GroupsConfiguration e NodeConfiguration.

Apesar de simples esta classe é relativamente importante para o funcionamento adequado desta solução de *failover* porque se da última vez que a solução esteve operacional, o *Node* sofreu uma falha ou as configurações (dos grupos e/ou respectivas *features*) foram alteradas entretanto, pode deixar o *Node* a pertencer a grupos e/ou possuir *features* que não devia. Cabendo à classe FailoverCleaner garantir que isto não ocorra.

No inicio da classe, o *Node* local é removido de todos os grupos a que pertence que não são os Grupos de Serviços em que o mesmo é definido como *Master* ou *Backup* e que não são o Grupo *Commons*. Nota que é possível no início da solução um *Node* ser ainda *Backup*, se não houve outro *Node* que o substituí-se, daí a necessidade de verificar-se o *Node* local ainda está definido como *Backup* para algum Grupo de Serviços, antes de o remover.

De seguida, serão removidas todas as *features* do *Node* local que não pertencem aos grupos ao qual este está associado, mas apenas aquelas que são conhecidas pela solução. Isto é, *features* que pertencem a algum dos grupos definidos na classe GroupsConfiguration. A razão pela qual a classe FailoverCleaner só remove *features* conhecidas pela solução é porque a lista de *features* do *Node* local inclui *features* que nunca devem ser removidas, de forma a preservar a integridade do sistema, como é o caso do próprio *Cellar*. E como no sistema podem ser introduzidas *features* que em nada estão relacionadas com a solução, é preferível remover apenas *features* conhecidas.

Contudo é necessário estar ciente de uma limitação com esta decisão. Se por exemplo o Grupo *Commons* possuir a *feature* A, e as configurações são alteradas removendo esta *feature* do Grupo *Commons*, esta não será designada como uma *feature* conhecida da próxima vez que a classe FailoverCleaner iniciar. Acabando por ficar instalada no *Node* local, e para que seja removida terá que ser feito de forma manual.

MasterCtrl

A classe **MasterCtrl** responsabiliza-se pelas acções de inicialização e destruição de *Nodes Master* para um ou mais determinados Grupos de Serviços. Porém cada instância

desta classe só irá se interessar com o *Node* na qual se encontra, definindo o mesmo como *Master* dos Grupos de Serviços especificados na classe NodeConfiguration (usando o *interface* FailoverNodeConfiguration). Quando a instância do MasterCtrl é inicializada é também verificado se o *Node* local pertence ao Grupo *Commons* e aos Grupos de Serviços em que o mesmo é definido como *Master*. Se ainda não pertencer a algum desses grupos será registado e de seguida serão instaladas as *features* dos grupos localmente, recorrendo ao método `pullGroup()` do FailoverFeaturesCtrl tal como foi discutido anteriormente.

Antes do fecho da classe MasterCtrl o *Node* local será removido dos Grupos de Serviços a que pertencia. Relembrando que sair dos Grupos de Serviços não levará a que as suas *features* sejam removidas, dos grupos ou localmente, nem por outro lado será invocada qualquer operação explicita que o faça. A razão para tal é bastante simples, porque se fosse invocada a remoção de *features* de um grupo poderia causar problemas, como foi discutido em 4.3.1. E remover as *features* localmente não traz quaisquer benefícios, pois se estas se manterem da próxima vez que a solução se inicia haveria todo o trabalho de as instalar novamente.

Um último pormenor a discutir sobre a classe MasterCtrl, é que a mesma depende da classe FailoverCleaner e para ser mais preciso necessita que esta seja iniciada primeiro. Para garantir que o *Node* não possui *features* ou pertence a grupos indesejados. Contudo esta dependência não é directa, mas sim uma dependência forçada através de *Spring* em que é definido explicitamente que a MasterCtrl só será inicializada após a FailoverCleaner.

BackupCtrl

A próxima classe do *bundle* failover.node, **BackupCtrl**, é a mais importante de toda a solução de *failover* por ser a responsável em executar todas as actividades que permitem a definição de *Backups*. Actividades que incluem escuta de eventos que anunciam a saída e entrada de *Nodes* no *Cluster*, definição de *Nodes Backup* para Grupos de Serviços no qual o *Node* removido era *Master* ou *Backup* e remoção de *Backups* na entrada de um *Node*. Todas estas actividades e toda a lógica da definição de *Backups* é aplicada por esta classe e pelos seus métodos `memberRemoved()` e `memberAdded()`, apresentados na figura 4.7 e 4.8 respectivamente. Estes métodos são invocados exactamente na saída e entrada de um *Node* do *Cluster* e seguem à risca o conceito que foi detalhado anteriormente na secção 3.3.2.

Mas recapitulando tal conceito, na saída de um *Node* cada um dos restantes compete por se tornar *Backup* dos Grupos de Serviços em que o *Node* que saiu era *Master* ou mesmo *Backup* (substituindo o *Backup* já existente). Esta competição deve resultar numa

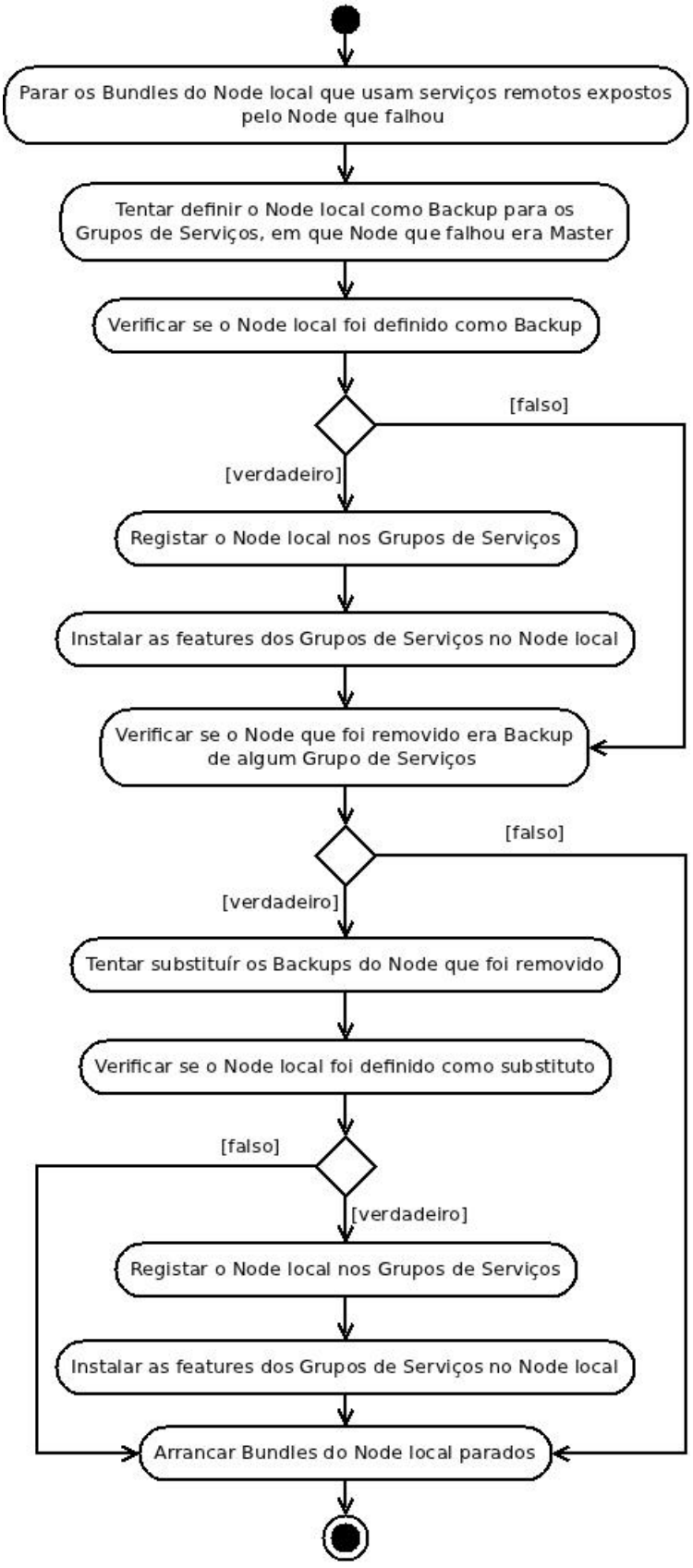

Figura 4.7: Diagrama de actividades da saída de um *Node* do *Cluster*

distribuição mais ou menos equilibrada dos Grupos de Serviços que necessitam de *Backup* pelos vários *Nodes* do *Cluster*. Isto porque assim que um *Node* detecte que não conseguiu se tornar *Backup* para um determinado grupo, vai tentar para o próximo. Enquanto que aquele que conseguiu fica ocupado a juntar-se ao grupo e a instalar as suas *features*. Que são as tarefas que qualquer *Node* vai executar mal consiga se tornar *Backup* de um Grupo de Serviços.

É necessário ainda acrescentar que a primeira e última actividade da figura 4.7 referem-se à invocação da classe ServiceCleaner (ainda por abordar), e são actividades necessárias para garantir uma melhor qualidade no processo de *failover*. Estas actividades realizam-se antes e após o processo de *failover* propriamente dito, consistindo primeiro em parar todos os *bundles* que façam uso de serviços que eram expostos no *Node* que saiu e depois voltar a arrancá-los. Os motivos da existência destas actividades e o benefício que realmente trazem será abordado com maior detalhe mais à frente.

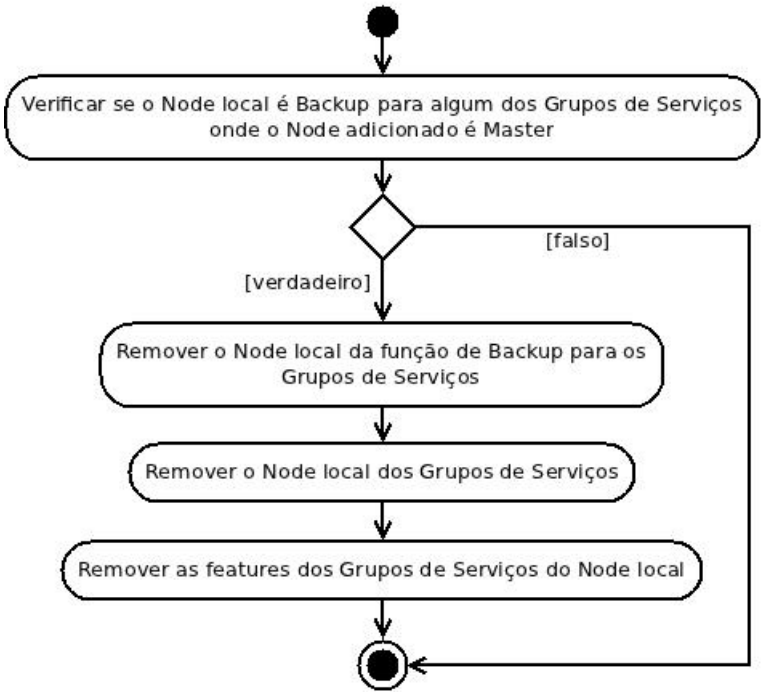

Figura 4.8: Diagrama de actividades da entrada de um *Node* do *Cluster*

Na entrada de um *Node*, cada um dos outros que já se encontravam no *Cluster* e que receberam um evento neste sentido, vão averiguar se estão a desempenhar o papel de *Backup* para algum dos Grupos de Serviços onde o *Node* que entrou está definido como *Master*. Se este for o caso vão retirar-se desta função (alterando o recurso distribuído que define a configuração dos *Backups*), sair dos grupos em questão e remover as suas *features*, mas apenas localmente e garantindo que saiu primeiro dos grupos. Repare-se também que

na entrada de um *Node* só serão abordados os Grupos de Serviços em que este é *Master* ignorando os seus *Backups*. Isto porque muito simplesmente este se era *Backup* quando inicialmente falhou agora que recuperou já não o é, pois foi substituído dessa função por outro qualquer *Node* do *Cluster* no momento da sua falha.

Outro aspecto necessário focar sobre a classe BackupCtrl são as suas operações realizadas durante o seu arranque, retratado pela figura 4.9. A mesma regista-se como *listener* à escuta de eventos lançados pelo *Hazelcast*, na entrada e saída de membros do *Cluster*. E também verifica se o *Node* local está definido como *Backup* para algum Grupo de Serviços. Se tal for o caso e o *Node Master* desse Grupo de Serviços permanecer fora do *Cluster*, o *Node* local passará a desempenhar o seu papel de Backup. Se o *Node Master* estiver activo, o *Node* local será removido de *Backup*.

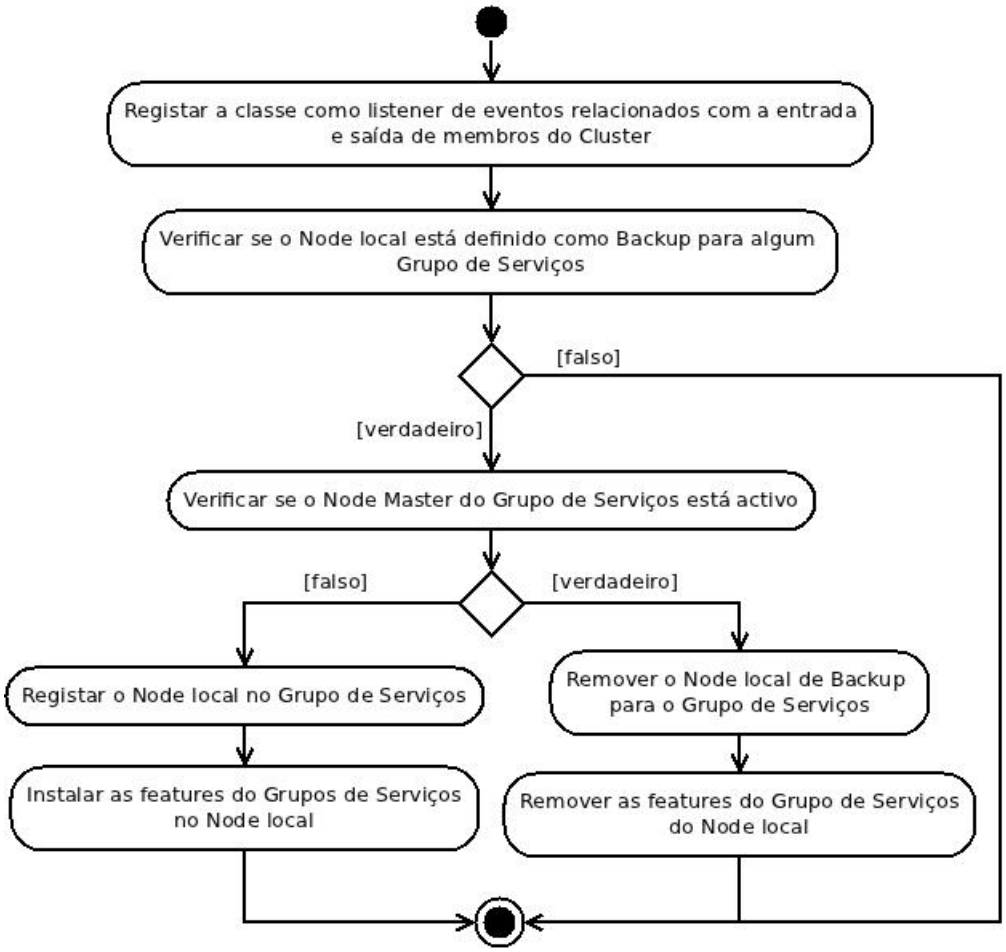

Figura 4.9: Diagrama de actividades do arranque da classe BackupCtrl

Por outro lado, antes da destruição desta classe a mesma deixará de escutar eventos relacionados com membros do *Cluster* e remove o *Node* local dos grupos em que é *Backup*.

Mas atenção que esta última acção não leva a que o *Node* local deixe de estar definido como *Backup* para determinados Grupos de Serviços. Por outras palavras, esta acção não modifica em nada o mapa distribuído que controla a informação dos *Backups* realizados.

Finalmente existem dois detalhes a salientar sobre a classe BackupCtrl. Tal como acontece com a classe MasterCtrl, a BackupCtrl também possui uma dependência efectuada através do *Spring*, neste caso em relação à própria MasterCtrl. O que garante que a classe BackupCtrl só é iniciada após o inicio da MasterCtrl, e consequentemente após o inicio da FailoverCleaner. E é também que em todas as operações efectuadas por esta classe que usem o mapa distribuído que contem a informação sobre os *Backups* efectuados, é efectuado primeiro um *lock* aos recursos pretendidos. Tentando evitar problemas de concorrência.

RestartCleaner

Outra das classes do *bundle* failover.node é a **RestartCleaner** e que foi concebida para tentar garantir um bom funcionamento das *features*, e dos *bundles* que compõem estas. Isto porque no início de um determinado *Node* principalmente quando este recupera de uma falha inesperada *bundles* que tivessem usado previamente serviços remotos vão arrancar e utilizar referências de serviços que já não existem. Um problema que está relacionado com a forma como o *Cellar* expõe e permite o acesso a serviços remotos, algo explicado em 4.2.1.

E assim como o próprio nome da classe indica, esta simplesmente reinicia todos os *bundles* controlados pela solução e apenas uma vez no início da mesma . Querendo isto dizer, *bundles* que compõem *features* dos Grupos de Serviços em que o *Node* local é *Master* e *Backup*. Mas havendo o cuidado de executar esta acção precisamente após a instalação de todas as *features* dos Grupos de Serviços, ou por outras palavras depois do MasterCtrl e BackupCtrl terem efectuado as suas tarefas iniciais.

Para conseguir realizar esta tarefa a classe RestarCleaner faz uso de uma outra classe de suporte **BundleUtils**, concebida simplesmente para fazer o mapeamento das *features* e dos *bundles* relevantes ao *Node* local e para interagir com a *framework OSGi* no toca à paragem e arranque de *bundles*. Sendo ainda relevante mencionar que tanto a classe BundleUtils como a classe RestarCleaner trabalham apenas com *features* e *bundles* dos Grupos de Serviços excluindo o Grupo *Commons*. Sendo a razão muito simples e passando por o facto do Grupo *Commons* não estar pensado, concebido para albergar *features* que usem ou disponibilizem serviços remotos, pois esse é o propósito dos Grupos de Serviços como é esclarecido na secção 3.3.2.

ServiceCleaner

Finalmente a última classe do *bundle* failover.node é a **ServiceCleaner**. Que surge com o mesmo objectivo do que a classe RestartCleaner, ou seja garantir o quanto possível o funcionamento adequado das *features* e respectivos *bundles* controlados por esta solução de *failover*, mas noutra ocasião. Neste caso durante o próprio processo de *failover*, melhorando a qualidade do mesmo.

Para tal a mesma define um mapa distribuído e acessível a todos os *Nodes* do *Cluster* que contêm a informação sobre que serviços remotos é que cada um dos *Nodes* expõe. Mesmo sabendo que o próprio *Cellar* possui um recurso similar e com o mesmo tipo de informação, mas que infelizmente não satisfaz o propósito desta classe porque simplesmente não transmite a informação necessária durante o processo de *failover*. É portanto mais pertinente criar um, algo feito tornando a própria classe num *ServiceListener OSGi*. Querendo isto dizer que esta vai receber eventos cada vez que é registado um serviço *OSGi* que possui a propriedade, `service.exported.interfaces`, discutida na secção 4.2.1. Sendo óbvio que serão apenas mapeados serviços remotos expostos por *features* de Grupos de Serviços.

Assim através da informação deste mapa distribuído e no qual cada um dos *Nodes* do *Cluster* insere informação sobre os serviços remotos que exporta é possível saber no momento da saída de um *Node* seja esta intencional ou não, os serviços que vão passar a não estar disponíveis até que o processo de *failover* em si termine. Com esta informação a classe ServiceCleaner, nos restantes *Nodes* do *Cluster*, vai procurar cada um dos *bundles* que usava tais serviços remotos e pará-los até que uma nova implementação do serviço esteja disponível. Evitando uma série de erros se estes serviços fossem acedidos, porque como foi explicado anteriormente tecnicamente o serviço existe localmente mas foi disponibilizado pelo *Cellar* servindo como apontador para a real implementação e falhará por já não existir tal implementação.

E só depois do processo em si de *failover* estar completo é que os *bundles* parados serão novamente arrancados, obrigando assim que novas referências a serviços remotos sejam obtidas e resultando num bom e correcto funcionamento dos mesmos. Notando finalmente que os *bundles* serão arrancados apenas quando o processo de *failover* terminar em todos os *Nodes* do *Cluster,* ou por outras palavras quando todos os serviços que deixaram de estar disponíveis sejam disponibilizados novamente. Algo que é verificado recorrendo ao mapa distribuído enunciado, pois cada um dos *Nodes* irá consultar o mesmo garantindo que todos os serviços estão a ser expostos por *Nodes* vivos (a verificação de tal facto é feita com recurso à classe **NodeUtils**, que disponibiliza uma série de utilidades neste sentido).

4.3.2 Configuração do Grupos da Solução de *Failover*

Relembrando aquilo que foi discutido na secção 3.3.2, para que seja possível dar ao Grupo *Commons* e ao Grupo de Serviços o comportamento pretendido é necessário restringir o conjunto de *features* com as quais cada grupo trabalha. Ignorando e bloqueando qualquer acção relacionada com uma *feature* que não esteja definida para um determinado grupo, algo essencial para não permitir que as *features* de diferentes grupos se misturem quando um determinado *Node* pertence a múltiplos grupos.

Por defeito o *Cellar* assume um conjunto de valores para as configurações de cada grupo e para o Grupo *Commons* e de Serviços estas configuração serão usadas, excepto nos seguintes casos:

- features.repositories.sync
- features.sync
- features.blacklist.inbound
- features.blacklist.outbound
- features.whitelist.inbound
- features.whitelist.outbound

Estas são propriedades das configurações de cada um dos grupos do *Cellar*, e no caso do Grupo *Commons* e de Serviços estas propriedades devem ter uns valores especiais para reflectir as *features* de cada. Tais propriedades serão alteradas recorrendo ao *OSGi* Configuration Admin Service.

No caso das propriedades `features.repositories.sync` e `features.sync` estas definem se o *Cellar* irá efectuar a sincronização automática dos recursos de cada *Node* com os dos Grupos a que pertence. Neste caso a sincronização refere-se à informação dos repositórios de *features* e das *features* em si, sendo que a sincronização da primeira será permitida com o valor `true` e a segunda será bloqueada com o valor `false`. A ideia é que os repositórios de *features* poderão ser livremente e automaticamente partilhados entre grupos, pois como foi referido estes são por defeito nesta solução comuns tanto ao Grupo *Commons* como a qualquer Grupo de Serviços. E pelo contrário, as *features* não podem ser misturadas entre grupos sendo portanto a sua sincronização bloqueada. Note-se também que o bloqueio desta sincronização de *features* não torna completamente impossível modificar as *features* de um grupo, quando um *Node* instala alguma. Para garantir isso é necessário recorrer às propriedades discutidas de seguida.

As quatro restantes propriedades servem para definir a forma como *Cellar* controla o acesso às *features* de um grupo, por parte de qualquer *Node* que faça parte do mesmo. Por outras palavras, estas propriedades definem se uma *feature* é permitida ou bloqueada (*whitelist*/*blacklist*) num determinado grupo, e se essa permissão ou não é relativa à escrita ou leitura (*outbound*/*inbound*) do conjunto de *features* do grupo.

Cada um dos grupos definidos nesta solução, vai ter a propriedade `features.whitelist.inbound` e `features.whitelist.outbound` definida com o valor que é o conjunto de *features* configuradas para o grupo. Ou sejas as *features* definidas nesta solução para cada grupo serão o valor a colocar nestas propriedades. O que indica que um determinado *Node* pode obter as *features* do grupo sendo estas instaladas localmente no *Node*, e que também pode instalar as *features* no grupo se já não estiverem.

As outras duas propriedades, `features.blacklist.inbound` e `features.blacklist.outbound`, seguem a mesma lógica de escrita e leitura das *features* de um grupo mas neste caso referem-se à não permissão. O valor que qualquer grupo vai ter nestas propriedades é nulo, ou seja não tem um valor associado, resultando num bloqueio de qualquer *feature* que não esteja definida na *whitelist*.

```
grupoA.features.repositories.sync = true
grupoA.features.sync = false
grupoA.features.whitelist.inbound = example.city.server
grupoA.features.whitelist.outbound = example.city.server
grupoA.features.blacklist.inbound =
grupoA.features.blacklist.outbound =
```

Figura 4.10: Exemplo das configurações de um grupo

A figura 4.10 mostra um exemplo das configurações descritas para um determinado grupo, e neste caso em particular o grupo (intitulado de "grupoA") possui apenas uma *feature* (`example.city.server`) que é permitida através das propriedades `features.whitelist.inbound` e `features.whitelist.outbound`. Note-se ainda que o "grupoA" é um Grupo de Serviços, mas se fosse no Grupo *Commons* a lógica inerente às configurações seria exactamente a mesma, variando apenas nas *features* do próprio grupo.

4.3.3 Configuração da Solução de *Failover*

Por fim será necessário apresentar alguns pormenores sobre a instalação e configuração da solução de *failover* propriamente dita, e que pressupõe que foi instalado o *Cellar* conforme é descrito no Apêndice A. A instalação da solução numa instância do *Karaf* é feita através da instalação dos dois *bundles* mencionados, failover.core e failover.node, e a qual

deve ser realizada em todos os *Nodes* do *Cluster*. A instalação dos mesmos poderá ser feita através dos comandos (do *Karaf*) `osgi:install mvn:bosch/failover.core` e `osgi :install mvn:bosch/failover.node`, assumindo que as instâncias do *Karaf* têm acesso a um repositório *Maven*[1] que possui instalados estes *bundles*. E para que tudo funcione correctamente o *bundle* failover.core deve ser arrancado primeiro do que o failover.node, seguindo a ordem inversa quando estes forem parados.

Quanto à configuração da solução, esta é feita modificando alguns valores do ficheiro de configuração *Spring* (`/META-INF/spring/spring-context.xml`), contido em cada um dos *bundles*. No bundle failover.core as propriedades a configurar são detalhadas de seguida, e a figura 4.5 dá um exemplo de tal configuração.

commonsGroup Esta propriedade representa o Grupo *Commons* e que por defeito apresenta o seguinte valor, `#{groupManager.createGroup('commons')}`. Este valor é uma expressão de ***Spring Expression Language*** (*SPEL*)[2] utilizada para invocar um método que recebe o nome a dar ao grupo e retorna o mesmo. O nome por defeito é `commons` e apesar poder ser alterado não deverá existir qualquer necessidade de o fazer.

commonsFeatures Define as *features* para o Grupo *Commons*, exemplo: `eventadmin`.

serviceGroupsConfiguration Define o mapa que contêm cada um dos Grupos de Serviços e as suas *features*. Cada uma das chaves do mapa é o Grupo de Serviços, obtido como o Grupo *Commons* com uma expressão *SPEL*, como por exemplo com o valor `#{ groupManager.createGroup('grupoA')}`. E para cada um dos Grupos de Serviços, as suas *features* tal como `example.city.server`.

featuresRepositories Indica os repositórios de *features* para todos os grupos, como é o caso do exemplo `mvn:org.apache.karaf.cellar/apache-karaf-cellar/2.2.4/ xml/features`.

No *bundle* failover.node as configurações a realizar dedicam-se exclusivamente a definir um *Node Master* para cada um dos Grupos de Serviços definidos na configuração anterior, como a figura 4.6 o demonstra. Em que a chave será o nome do Grupo de Serviços e o seu valor o *hostname* e porto do *Node*.

[1] O *Apache Maven* é uma ferramenta de gestão de projectos de *software*, permitindo a definição de repositórios para esses mesmo projectos

[2] *SPEL* corresponde a uma forma mais dinâmica de fazer a injecção de dependências na *framework Spring*

4.3.4 Problemas e Limitações

Finalmente nesta ultima secção relacionada com a solução de *failover* elaborada serão apresentados um conjunto de problemas e limitações identificados no desenvolvimento e teste desta solução. Problemas estes que não surgem da solução propriamente dita, mas sim das tecnologias usadas para a desenvolver e cujas as causas são difíceis de apurar. Independentemente disto, a própria solução de *failover* toma acções para minimizar ou anular estes problemas e que serão também explicadas.

Informação Incorrecta sobre os Membros dos Grupos do *Cellar*

O seguinte problema refere-se a uma incoerência da informação disponibilizada pelo *Cellar* sobre os membros dos seus grupos. Isto é, em determinados momentos se questionado que *Nodes* pertencem aos grupos existentes (algo que pode ser feito através do comando `cluster:group-list` ou programaticamente através dos serviços *OSGi* disponibilizados) o *Cellar* indicará *Nodes* que não se encontram activos, quando estes estão de facto inactivos. Mas curiosamente se o *Cellar* for questionado apenas sobre os *Nodes* que estão no *Cluster* independentemente do grupo, este não irá identificar os *Nodes* que estão efectivamente inactivos.

Esta incoerência poderia ser explicada se fosse aceite que o *Cellar* mostrá-se *Nodes* nos grupos aos quais pertenciam quando estavam activos. Mas esta incoerência nem sempre acontece, sendo que esta manifesta-se quando existe a saída de um *Node* do *Cluster*. E pelos testes efectuados foi constatado que se a saída do *Node* for inesperada, em que por exemplo o *Node* é encerrado de um modo forçado em vez da forma correcta, esta incoerência irá manifestar-se. Por outro lado se a saída for efectuada de uma forma correcta, invocando o encerramento do *Node*, esta incoerência poderá ou não suceder. Mas ainda mais curioso, é que se esta a incoerência não se manifestar no encerramento de um *Node*, o *Node* A por exemplo, e futuramente existe também o fecho do *Node* B, esta incoerência irá manifestar-se novamente e em que o *Node* A irá ser identificado nos grupos aos quais pertencia mas que na realidade continua inactivo.

Felizmente esta incoerência em nada afecta a solução de *failover* aqui desenvolvida. Isto porque a mesma não utiliza esta informação do *Cellar* sobre a que grupos pertencem os *Nodes* do *Cluster* e porque no caso de saber quando ocorre a entrada e saída de *Nodes* no *Cluster* esta solução recorre aos eventos lançados pelo *Hazelcast*, os quais disponibilizam uma informação correcta.

E por fim é possível afirmar, pelos testes realizados, que esta incoerência em nada

tem haver com a solução de *failover* e será muito provavelmente uma falha do *Cellar*. Porque tal problema foi replicado com *Nodes* e grupos, sem usar ou instalar a solução.

Impossibilidade de Configurar Correctamente um Grupo do *Cellar*

No arranque desta solução, uma das suas primeiras tarefas é a configuração dos grupos com os quais lida. Contudo nos testes efectuados surgiram ocasiões em que determinadas configurações não ficaram completamente correctas, nomeadamente no que toca ao bloqueio ou não de *features* para um determinado grupo. Resultando em situações onde as *features* dos grupos desta solução se misturavam ou eram removidas prematuramente.

As causas deste problema não foram descobertas, mas foi conferido que não surgem da solução em si através do registo de *log* no qual é possível verificar que a configuração é efectuada com os valores correctos. Contudo para resolver o dito problema foi implementada uma verificação e correcção das configurações caso o problema aconteça, aliás como foi explicado na secção 4.3.1.

Duplo *Backup*

A grande e principal funcionalidade que esta solução de *failover* disponibiliza é a capacidade de na eventual falha de um determinado *Node* definir *Backups* para os seus Grupos de Serviços. E neste processo foi encontrado um problema difícil de replicar e o qual pode ser descrito como a definição de dois *Backups* para o mesmo Grupo de Serviços, em vez de apenas um.

Este problema foi detectado num cenário em que perante três *Nodes* e com a falha de um deles, os dois restantes acabaram por actuar como *Backup*. Mas atenção porque na realidade apenas um dos *Nodes* é que realmente foi definido como *Backup* (chame-mos-lhe de *Node* A), enquanto que o outro *Node* (o *Node* B) juntou-se ao Grupo de Serviços do *Node* que falhou e acabou por adquirir as *features* também. Significando isto que apenas um deles, o *Node* A, é que correctamente pode ser intitulado de *Backup*, apesar de o *Node* B estar na prática a desempenhar as funções daquilo que é considerado como Backup nesta solução.

Após uma análise mais cuidada do *log* dos dois *Nodes*, constatou-se que o *Node* A é correctamente o primeiro a conseguir-se definir-se como *Backup* alterando o mapa distribuído que controla tal informação e que o *Node* B apercebe-se que o A foi definido como *Backup*, acabando por não executar qualquer acção relacionada com o processo de definição de um *Backup*. O que indica que este problema não está relacionado com o mecanismo de escolha de *Backups* desta solução.

Infelizmente as razões do problema em si são desconhecidas, mas pode ser acrescentado que a razão pela qual as *features* foram instaladas no *Node* B deve-se provavelmente ao facto de o A se ter tornado *Backup*. Isto é, o problema aqui é realmente que o *Node* B se junta ao Grupo de Serviços do *Node* que falhou justamente quando o *Node* A é definido com Backup, mas que o facto de B possuir as *features* do Grupo de Serviços é porque quando elas são instaladas em A também são instaladas em B, porque este também pertence ao Grupo de Serviços em questão.

Ainda outra consequência deste problema é que quando o *Node* A deixar de ser um *Backup* o mesmo não irá acontecer ao B porque este sabe que não está definido como um *Backup* e não fará nada para sair do Grupo de Serviços e/ou remover as *features* localmente.

Assim para anular as consequências deste problema foi implementada uma pequena correcção, nos métodos `backupMasterServiceGroup()` e `replaceBackupServiceGroup()` da classe BackupCtrl. Que consiste na verificação por parte de um qualquer *Node* que tentou definir-se como *Backup* mas não conseguiu, se encontra-se incorrectamente no Grupo de Serviços em questão. Se tal for o caso este *Node* sairá do grupo e remove localmente as suas *features*, não removendo aquelas que possam ainda ser precisas por outro grupo.

Serviços Remotos Inutilizáveis

Finalmente, o seguinte problema afecta o processo de *failover* realizado por esta solução e já foi descrito de forma indirecta noutras secções. Este advém da incapacidade do *Cellar* em corrigir serviços remotos que ficam inutilizáveis quando os *Nodes* que os implementam ficam inactivos.

Este problema gera uma série de erros quando um *bundle* tenta aceder a um desses serviços remotos que efectivamente já não existe. E a exposição de um novo, fruto do *Backup* dos Grupos de Serviços que o disponibilizam também não resolve o problema pois os *bundles* vão continuar a tentar ao serviços.

Para resolver o problema que torna de certa maneira inútil o processo de *failover*, porque prejudica a comunicação entre *bundles*, foram introduzidas duas acções antes e depois do processo. Estas acções consistem em para os *bundles* que têm referências a serviços inutilizáveis, e depois de efectuado o processo de *failover* e da exposição dos mesmo serviços são arrancados novamente. Esta correcção é implementada pela classe ServiceCleaner e resolve completamente o problema.

4.4 Conclusões

Concluindo, foram discutidos neste capítulo todos aspectos técnicos das implementações dos requisitos não-funcionais de IAD e DA no *Fuse ESB*. Algo feito fortemente com recurso à tecnologia *Cellar* e seguindo o que foi definido no capítulo 3.

Visto que o *Cellar* é a solução escolhida para implementar completamente o requisito de IAD, o foco que foi dado ao mesmo incide sobre como utilizar correctamente esta solução para permitir a interacção de aplicações de diferentes *Fuse ESB*s. Mais específicamente como definir e consumir um serviço *OSGi* distribuído e exposto num *Cluster* de *Fuse ESB*s. Algo que é efectuado de uma forma simples, sendo que a exposição de um serviço *OSGi* para o *Cluster* necessita da introdução de um conjunto de meta-dados que o indiquem como um serviço remoto e o consumo dum serviço deste tipo não difere do consumo de um serviço *OSGi* normal. Foram também destacados alguns pequenos cuidados a ter na utilização deste tipo de serviços *OSGi*, que podendo parecer triviais não devem ser esquecidos quando se está perante um ambiente distribuído, como é um *Cluster* de *Fuse ESB*s.

No que diz respeito ao requisito de DA foi dedicada uma grande secção com a descrição da solução de *failover* desenvolvida e que implementa tal requisito. Para esta solução é detalhada a sua arquitectura e respectivos componentes que implementam o processo de *failover* propriamente dito. Desde o momento inicial onde são definidas que *features* (conjunto de *OSGi bundles*, e que representam as aplicações cuja disponibilidade se pretende garantir) pertencem a cada um dos *Nodes* (*Fuse ESB*s) do *Cluster*, algo realizado com o conceito de grupo do *Cellar* e mais especificamente através do conceito de Grupo *Commons* e Grupo de Serviços definidos nesta solução de *failover*. Até ao momento em que ocorre a falha de um dos *Nodes* e as *features* pelas as quais este era responsável são distribuídas por outros *Nodes*, acontecendo o inverso quando o *Node* voltar a ficar activo. Para todos os aspectos deste processo (que foi explicado detalhadamente na secção 3.3) é apontado o papel de cada um dos componentes da solução e como o realiza tecnicamente.

Além disto foi explicada a configuração desta solução, não só aquela que quem a pretende usar tem de efectuar. Indicando que *features* compõem cada um dos grupos e que *Nodes* são responsáveis por cada um desses. Bem como aquela que a própria solução efectua automaticamente para com o *Cellar* e baseada nas *features* definidas.

Finalmente são apontados problemas encontrados ao longo do desenvolvimento da solução. Problemas que não estão relacionados com a solução em si, mas com incoerências do *Cellar* e que acabam por condicionar o funcionamento da solução. Para cada um deles

são explicadas as suas consequências e como os componentes da solução os conseguiram resolver.

Capítulo 5

Conclusões

5.1 Considerações Finais

Focado inúmeras vezes ao longo deste documento, um ESB apresenta-se como uma ferramenta de elevada complexidade e cujo suporte a aplicações empresariais é indiscutível. Actuando então como um mediador de aplicações, sejam estas aplicações que estão directamente ligadas a este, contidas nele, ou então externas mas que interagem com ele. Permitindo com certeza uma maior interoperabilidade aplicacional.

Conhecendo então o papel preponderante de um ESB e a importância que existe em permitir/conceder uma maior e melhor interoperabilidade aplicacional é que se mostrou pertinente que esta Dissertação analisa-se um conjunto de aspectos, requisitos não-funcionais a disponibilizar ao *Fuse ESB*. Do universo de requisitos não-funcionais a analisar considerou-se como relevantes os seguintes: IAD e DA.

Querendo isto dizer que estes requisitos não-funcionais no âmbito de *ESB*s pretendem: permitir uma maior interoperabilidade aplicacional, entre aplicações de diferentes *Fuse ESB*s dentro do mesmo *Cluster* (IAD) e dotar as aplicações de maiores níveis de disponibilidade, tornando as mesmas mais resistentes a cenários de falha de um *Fuse ESB* ao qual pertencem (DA).

Assim com o foco bem definido na análise dos requisitos não-funcionais de IAD e DA no prisma do *Fuse ESB*, o primeiro passo efectuado foi uma análise conceptual destes dois requisitos. Análise essa que define conceptualmente o propósito, a função do requisito no *Fuse ESB*.

No que diz respeito ao requisito não-funcional de IAD, este pretende a definição de um *Cluster* de *Fuse ESB*s e no qual as aplicações de cada um conseguem consumir e disponibilizar serviços para esse mesmo *Cluster*. Exactamente da mesma maneira que o fazem localmente num único *Fuse ESB* através da *framework OSGi* que faz parte deste. Sendo o intuito permitir que serviços de diferentes *frameworks OSGi* sejam consumidos em qualquer uma. Por outras palavras podemos dizer que o *Cluster* de *Fuse ESB*s é feito ao nível da *framework OSGi* e através dos serviços nela publicados.

Quanto ao requisito não-funcional de DA e assumindo como essencial que este seja aplicado paralelamente ao de IAD, o último é utilizado como base. Ou seja, o requisito de DA faz uso da modularidade da *framework OSGi* para movimentar componentes entre *Fuse ESB*s quando for necessário (em caso de falha de algum destes). Algo que é possível de realizar sem comprometer o funcionamento das aplicações graças à premissa que estas conseguem comunicar a partir de qualquer *Fuse ESB*, pois os serviços são acessíveis em qualquer um do *Cluster*.

Posteriormente e com as funções dos requisitos não-funcionais definidas foi necessário definir soluções que os apliquem. Para o requisito de IAD foi encontrada uma solução, o *Cellar*, capaz de implementar completamente tal requisito no *Fuse ESB* de uma forma muito simples e transparente. Relativamente ao requisito de DA não foi encontrada uma solução que o implementa-se da maneira pretendida e/ou compatível com o *Fuse ESB*. Portanto foi definida, concebida de raiz uma solução de *failover* capaz de satisfazer tal requisito, utilizando o *Cellar* e as suas funcionalidades. Tornando ainda mais forte a relação entre os dois requisitos.

Finalmente, o último passo realizado foi efectivamente a implementação dos requisitos não-funcionais estabelecidos. Para o de IAD foi explicado como aplicá-lo através do *Cellar*, bem como alguns cuidados a ter quanto à utilização em si. Paralelamente para o requisito de DA foram explicados extensivamente os detalhes e pressupostos da solução desenvolvida, nomeadamente da sua arquitectura e respectivos componentes. Terminando como uma identificação de problemas encontrados e como a própria solução os combate.

Assim esta Dissertação analisa e trabalha dois requisitos não-funcionais importantes no âmbito do *Fuse ESB*, concebendo e implementando-os. Disponibilizando um novo olhar sobre o *Fuse ESB* no que diz respeito aos requisitos não-funcionais que o mesmo deve ter, algo pouco trabalhado do ponto de vista de investigação. E se por um lado o trabalho do requisito de IAD já tinha sido de certa maneira abordado, aliás como o facto de existir uma solução como o *Cellar* o demonstra, o requisito de DA ainda não. Pelo menos que seja do conhecimento do autor deste trabalho. Neste tópico este trabalho apresenta portanto uma solução de *failover* única, para o âmbito do *Fuse ESB* e da *framework OSGi*. Interagindo com o requisito de IAD e desenvolvendo o de DA.

5.2 Trabalho Futuro

Concluindo o que foi efectuado ao longo deste trabalho, vejamos agora possíveis tópicos de trabalho futuro que seguem a mesma linha de investigação, isto é os requisitos não-funcionais no âmbito do *Fuse ESB*. Como tal são sugeridos os seguintes temas: uma extensão do requisito não-funcional de IAD e a implementação do requisito de DCA.

A primeira sugestão tem como objectivo levar o requisito de IAD mais além permitindo uma interacção facilitada dentro de um *Cluster* com sistemas externos. Pois como é possível constatar por tudo o que foi discutido ao longo do trabalho sobre o requisito, é que este permite que exista uma comunicação transparente de aplicações de diferentes *Fuse ESB*s mas que estejam contidos nele, ou seja na sua *framework OSGi* (através de serviços

OSGi). Mas quando é pretendida uma interacção com um sistema externo (exemplo um dispositivo móvel) é necessário definir explicitamente um *bundle* de *gateway* para poder disponibilizar um serviço nesse sentido. A ideia seria portanto elaborar uma solução capaz de agilizar tal processo, facilitando a interacção com sistemas externos, sendo que o requisito de IAD não serviria apenas para expor serviços de aplicações de um determinado *Fuse ESB* para o *Cluster*, mas também serviços externos a que este *Fuse ESB* tem acesso para o *Cluster*.

E a segunda sugestão seria tratar o requisito não-funcional de DCA no *Fuse ESB*. Efectuando um análise conceptual e posteriormente a implementação tal como aconteceu aqui com o requisito de IAD e DA, apesar de poder ser relativamente mais complexo. Note-se também uma curiosidade relativa ao desenvolvimento da solução de *failover*, pois os conceitos subjacentes a esta foram pensados de maneira a poder futuramente utilizá-los na implementação do requisito de DCA. Sendo a ideia introduzir quando necessário mais *Nodes* num determinado grupo, tendo simultaneamente várias instâncias de um serviço a ser expostas. Assim quando um *Node* tivesse uma carga excessiva seriam apontados mais *Nodes* para todos os seus grupos, ou então apenas para um grupo em especifico. Sendo o grande desafio saber quando uma máquina ou grupo de *features* tem níveis acrescidos de carga. E quando fossem introduzidos mais *Nodes* em determinados grupos fazer uma nova atribuição dos serviços existentes pelos consumidores dos mesmos, garantindo de facto que existe uma distribuição de carga.

Referências

Apache Software Foundation (n.d.a). Apache ActiveMQ ™ – clustering. http://activemq.apache.org/clustering.html acedido em 11 de Fevereiro de 2012

Apache Software Foundation (n.d.b). Apache ActiveMQ ™ – how do distributed queues work. http://activemq.apache.org/how-do-distributed-queues-work.html acedido em 12 de Fevereiro de 2012

Astrova, I., A. Koschel e T. Kruessmann (2010). Comparison of enterprise service buses based on their support of high availability. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2495–2496.

Bean, J. (2009). *SOA and Web Services Interface Design: Principles, Techniques, and Standards* (1 ed.). Morgan Kaufmann.

Chappell, D. (2004). *Enterprise Service Bus: Theory in Practice*. O'Reilly Media.

Chappell, D. e D. Berry (2007). SOA-ready for primetime: The next-generation, grid-enabled service-oriented architecture. *SOA Magazine*.

Davis, J. (2009). *Open Source Soa* (1st ed.). Manning Publications Co.

FuseSource (2011). Fuse ESB - product introduction (Version 4.4.1)

Hall, R. S., K. Pauls, S. McCulloch e D. Savage (2011). *OSGi in action*. Manning Publications.

Hevner, A. R., S. T. March, J. Park e S. Ram (2004). Design science in information systems research. *MIS Q. 28*(1), 75–105.

IBM (2005). Clustering and high availability in an enterprise service bus. White paper.

Keen, M., O. Adinolfi, S. Hemmings, A. Humphreys, H. Kanthi e A. Nottingham (2005). *Patterns: SOA with an Enterprise Service Bus*. IBM Redbooks.

Koschel, A., T. Schneider, J. Westhuis, J. Westerkamp, I. Astrova e R. Roelofsen (2011). Providing load balancing and fault tolerance in the OSGi service platform. In *Proceedings of the 13th WSEAS international conference on Mathematical and computational methods in science and engineering*, pp. 426–430.

Kruessmann, T., A. Koschel, M. Murphy, A. Trenaman e I. Astrova (2009). High availability: Evaluating open source enterprise service buses. In *Information Technology Interfaces, 2009. ITI'09. Proceedings of the ITI 2009 31st International Conference on*, pp. 615–620.

Marcus, E. e H. Stern (2003). *Blueprints for High Availability* (2 ed.). Wiley.

Maréchaux, J. (2006). Combining service-oriented architecture and event-driven architecture using an enterprise service bus. *IBM Developer Works*.

Menge, F. (2007). Enterprise service bus. In *Free and open source software conference*.

Ning, F., Z. Xingshe, W. Kaibo e Z. Tao (2008). Distributed enterprise service bus based on JBI. In *Grid and Pervasive Computing Workshops, 2008. GPC Workshops' 08. The 3rd International Conference on*, pp. 292–297.

Oracle (2002). Java(TM) message service specification final release 1.1

Ortiz, S. (2007). Getting on board the enterprise service bus. *Computer 40*(4), 15–17.

OSGi Alliance (n.d.). OSGi alliance | about / the OSGi architecture. http://www.osgi.org/About/WhatIsOSGi acedido em 5 de Fevereiro de 2012

Papazoglou, M. e W. Van Den Heuvel (2007). Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal 16*(3), 389–415.

Rademakers, T. e J. Dirksen (2008). *Open-Source ESBs in Action: Example Implementations in Mule and ServiceMix* (1 ed.). Manning Publications.

Sadashiv, N. e S. Kumar (2011). Cluster, grid and cloud computing: A detailed comparison. In *Computer Science & Education (ICCSE), 2011 6th International Conference on*, pp. 477–482.

Schmidt, M., B. Hutchison, P. Lambros e R. Phippen (2005). The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal 44*(4), 781–797.

Sharma, P., B. Kumar e P. Gupta (2009). An introduction to cluster computing using mobile nodes. In *Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on*, pp. 670–673.

Snyder, B., D. Bosanac e R. Davies (2011). *ActiveMQ in Action* (Pap/Psc ed.). Manning Publications.

Wang, J., Y. Ren, D. Zheng e Q. Wu (2007). Agent based load balancing middleware for service-oriented applications. *Computational Science–ICCS 2007*, 974–977.

Yin, J., H. Chen, S. Deng, Z. Wu e C. Pu (2009). A dependable ESB framework for service integration. *Internet Computing, IEEE 13*(2), 26–34.

Apêndice A

Configuração do *Cellar*

Antes de poder utilizar e aplicar os dois requisitos não-funcionais, de IAD e DA, discutidos neste trabalho é imprescindível que seja efectuada uma instalação e configuração correcta do *Cellar*. De seguida será apresentado como efectuar isso mesmo, sendo o ponto de partida para tal uma instância do *Fuse ESB* ou então simplesmente do *Karaf*. As configuração é exactamente para os dois casos, porque na realidade o *Karaf* é um dos componentes do *Fuse ESB*.

Com o ponto de partida mencionado, a configuração do *Cellar* deverá ser efectuada com o seguintes passos:

1. No ficheiro `/etc/config.properties`, encontrado na directoria do *Karaf*, deve ser alterada a propriedade `karaf.framework` para o valor `equinox`. Esta propriedade define a implementação *OSGi* a usar, que pode ser o *Equinox* ou o *Apache Felix*, e esta solução foi desenvolvida com base no *Equinox*.

2. Iniciado o *Karaf*, é necessário instalar o *Cellar* através dos seguintes comandos `feature :add-url mvn:org.apache.karaf.cellar/apache-karaf-cellar/2.2.4/xml/features` e `feature:install cellar`. O primeiro comando adiciona um URL que contem a *feature* do *Cellar* e o segundo executa a sua instalação.

3. É necessário também que sejam instaladas as *features* do *Spring* e *Spring DM* se já não estiverem, com recurso ao comando `feature:install spring` e `feature:install spring-dm`.

4. No caso de serem executadas duas ou mais instâncias do *Karaf* simultaneamente na mesma máquina, é necessário que todas elas tenham portos diferentes nas propriedades `rmiRegistryPort` e `rmiServerPort` do ficheiro `/etc/org.apache.karaf.management.cfg`.

5. No ficheiro `/etc/hazelcast.xml`, os seguintes elementos têm que ser alterados, resultando numa configuração semelhante à apresentada na figura A.1.

   (a) `<port auto-increment="false">5701</port>`, o valor é modificado para `false` e o porto deve ser modificado se existirem mais do que uma instância do *Karaf* na mesma máquina, caso contrário pode permanecer com o valor por defeito.

   (b) `<multicast enabled="false">`, a funcionalidade de *multicast* que vem activada por defeito no *Hazelcast* deve ser desactivada.

   (c) `<tcp-ip enabled="true">`, e em contraste à funcionalidade de *multicast* esta deve ser activada. Ainda dentro desta *tag* XML devem ser adicionados os *host-*

```
<network>
    <port auto-increment="false">5701</port>
    <join>
        <multicast enabled="false">
            <multicast-group>224.2.2.3</multicast-group>
            <multicast-port>54327</multicast-port>
        </multicast>
        <tcp-ip enabled="true">
            <hostname>alex-laptop.home:5701</hostname>
            <hostname>alex-desktop.home:5701</hostname>
        </tcp-ip>
 ...
```

Figura A.1: Excerto da configuração do *Hazelcast*

names e portos de todos os membros que irão participar no *Cluster*, e incluindo o do membro local como por exemplo: `<hostname>alex-desktop.home:5701</hostname>`. Sendo esta lista igual na configuração de todas as instâncias do *Karaf*.

Apêndice B

Versões das Tecnologias Utilizadas

Tabela B.1: Versões das tecnologias utilizadas pelo ambiente de desenvolvimento

Nome	Versão
Fuse ESB	4.4.1
Karaf	2.2.7
OSGi (Equinox)	3.6.2.R36x_v20110210
Cellar	2.2.4
Hazelcast	1.9.4.8
Spring DM	1.2.1

Atenção que a versão do *Karaf* identificado na tabela B.1 é separado do *Fuse ESB*. Algo que é explicado na secção 3.4.

Apêndice C

Diagramas de Classes da Solução de *Failover*

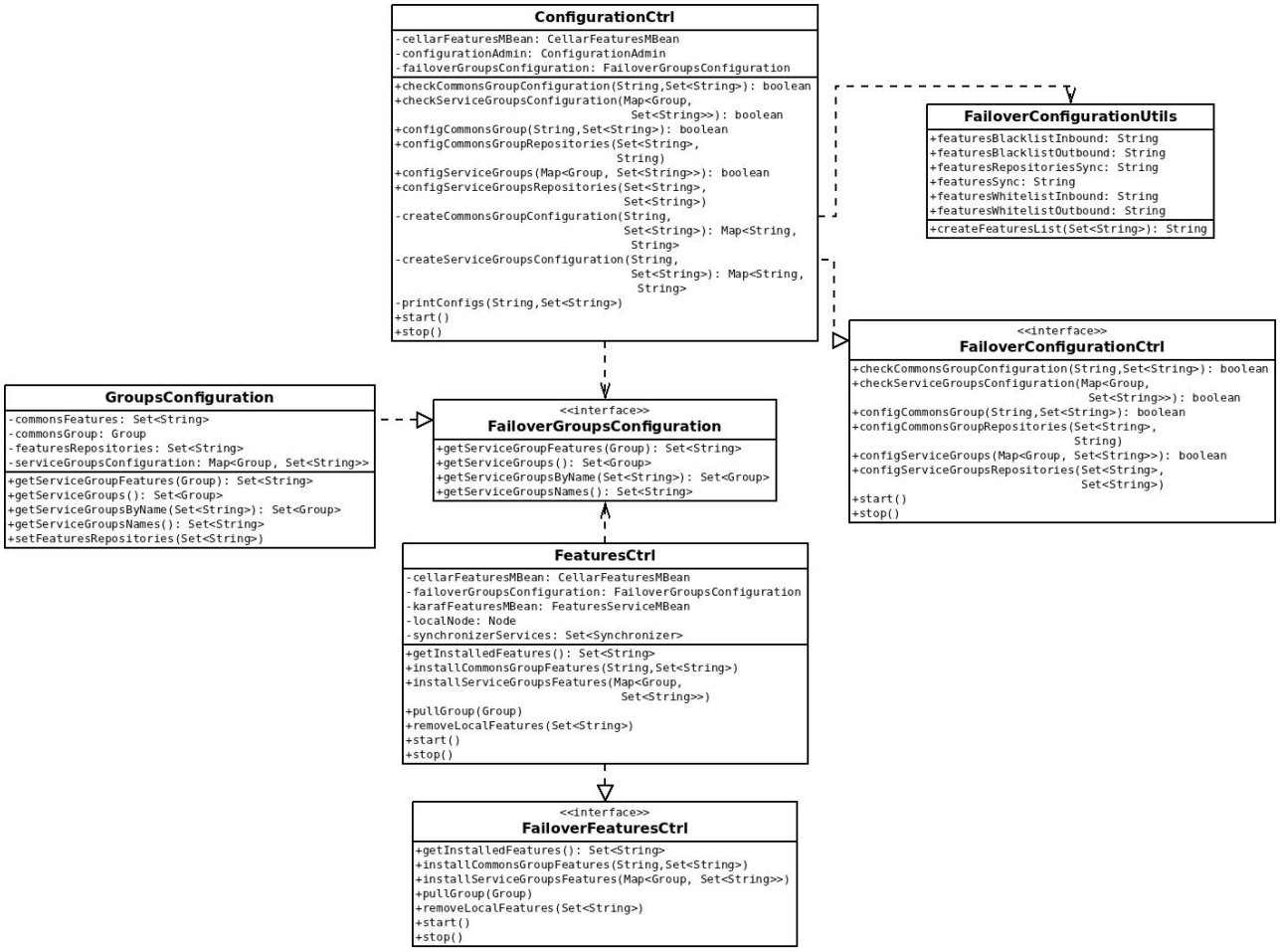

Figura C.1: Diagrama de classes do *OSGi bundle* failover.core

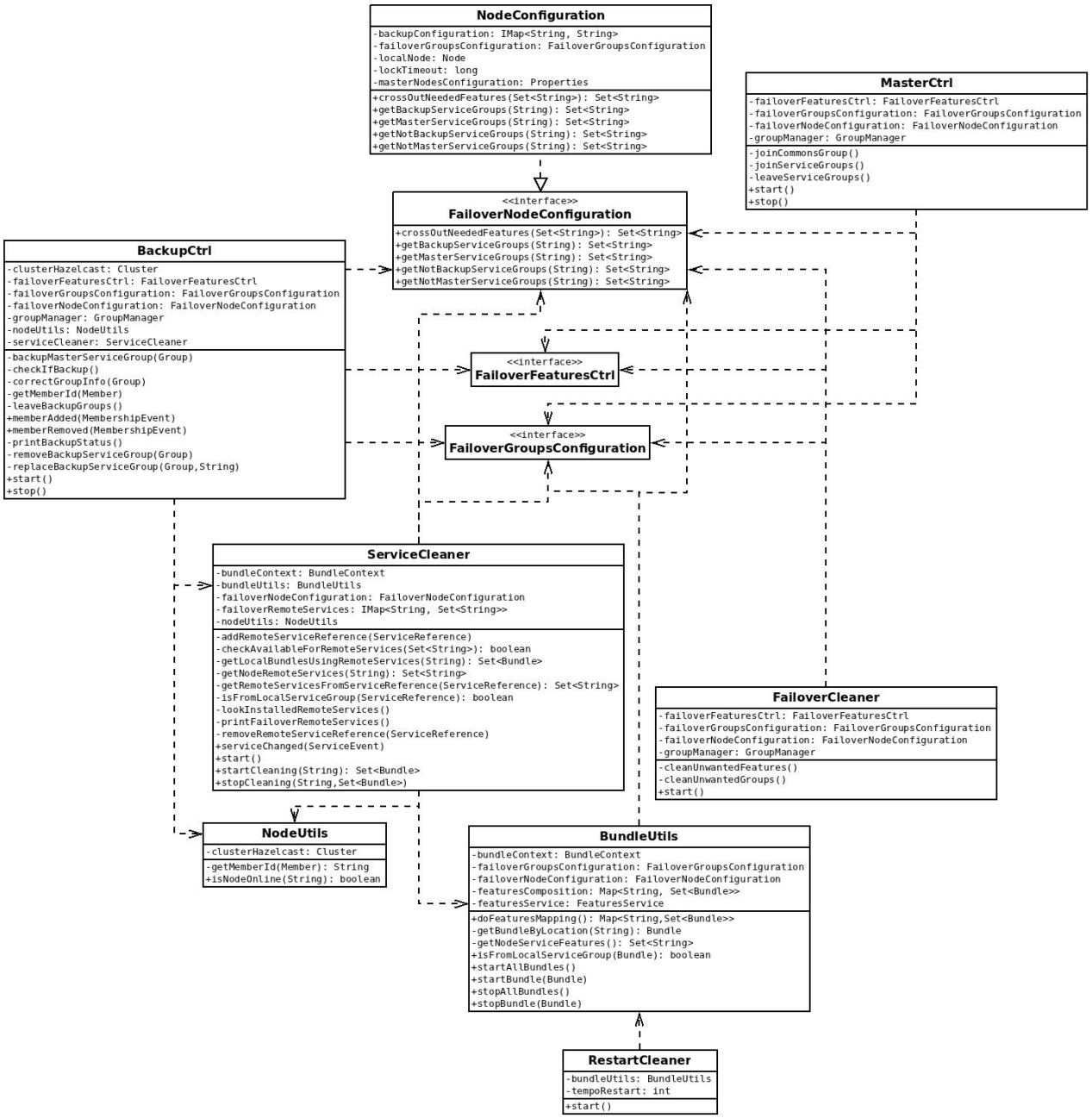

Figura C.2: Diagrama de classes do *OSGi bundle* failover.node