UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

Modularização e Extensão do InteropFrame

Felipe Oliveira Carvalho
Rodrigo Losano Fontes Calheiros

São Cristóvão – SE
2014

Felipe Oliveira Carvalho
Rodrigo Losano Fontes Calheiros

Modularização e Extensão do InteropFrame

Pré- Projeto do Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação da Universidade Federal de Sergipe.

Orientador: Prof. Dr. Tarcísio da Rocha

São Cristóvão – SE
2014

Modularização e Extensão do InteropFrame

Felipe Oliveira Carvalho
Rodrigo Losano Fontes Calheiros

Pré- Projeto do Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação da Universidade Federal de Sergipe.

Aprovado em: ___ /___ /___.

Banca Examinadora:

_____

Prof. Dr. Tarcísio da Rocha (Orientador)
Universidade Federal de Sergipe

_____

Prof. xxxxx xxxxxxx xxxxxxx
Universidade Federal de Sergipe

_____

Prof. xxxxxx xxxxxxx xxxxxx
Universidade Federal de Sergipe

São Cristóvão – SE
2014

Resumo

O desenvolvimento de Sistemas Distribuídos tem se tornado uma tarefa complexa. Uma técnica que tornou-se amplamente utilizada no desenvolvimento desses sistemas é a Engenharia de Software Baseada em Componentes (ESBC), o que resultou no surgimento de diversos modelos de componentes. Um desafio que surge do desenvolvimento baseado em componentes é o da interoperabilidade entre partes heterogêneas desenvolvidas em diferentes modelos de componentes. Em geral, o problema da interoperabilidade entre partes heterogêneas de um sistema é tratado pelo uso de *middlewares*. Sistemas de *middleware* são capazes de promover integração e abstrair do desenvolvedor detalhes dessa integração.

Dentro desse contexto foi criado o InteropFrame, um *middleware* que lida com a interoperabilidade entre sistemas distribuídos desenvolvidos nos modelos de componentes OpenCOM e Fractal, além de tratar de detalhes de comunicação remota utilizando RMI e *Web Services SOAP*. O InteropFrame é uma solução desenvolvida em Java puro e é extensível para o suporte a novos modelos de componentes e também de comunicação remota. Porém, por ser desenvolvido em Java puro, esse suporte fica dificultado, uma vez que não são estabelecidas interfaces modulares para a extensibilidade. Além dessa problemática, o InteropFrame também possui limitações na comunicação remota interna.

Este trabalho propõe a modularização do InteropFrame utilizando o modelo OSGi, de forma a reorganizar sua arquitetura em *plug-ins* para os modelos de componentes suportados e também para os modelos de comunicação remota. Visando confirmar a proposta de modularização, será adicionado o suporte ao modelo OSGi para tornar-se interoperável com os modelos já suportados pelo InteropFrame. Além da modularização, serão feitas melhorias internas na comunicação remota do InteropFrame.

Palavras-chaves: Modelos de Componentes, Interoperabilidade, Modularização, Sistemas Distribuídos.

Abstract

This is the english abstract.

Key-words: Component Models, Interoperability, Modularization, Distributed Systems.

Sumário

1 Introdução

Esta introdução tem por objetivo apresentar: a motivação para este trabalho de conclusão de curso; os objetivos gerais e os específicos deste trabalho; as principais contribuições; e a estrutura e organização deste documento.

1.1 Motivação

A necessidade de soluções em sistemas distribuídos complexos vem substituindo a visão de sistemas distribuídos homogêneos onde aplicações de domínio específico são desenvolvidas usando plataformas e *middleware* projetados especificamente para esse domínio. Soluções tecnológicas independentes têm sido interconectadas para criar estruturas ainda mais ricas, os chamados sistemas de sistemas (SoS). Um dos principais desafios dessas interconexões é a questão da interoperabilidade: a habilidade desses sistemas se conectarem, de trocarem dados e de se comunicarem (BLAIR et al., 2011), (INVERARDI; ISSARNY; SPALAZZESE, 2010), (BROMBERG et al., 2011).

Uma técnica que tem sido amplamente utilizada nos últimos anos no desenvolvimento de plataformas de sistemas distribuídos é a ESBC (Engenharia de Software Baseada em Componentes) (ROUVOY; MERLE, 2009). Como consequência do sucesso no uso de componentes de software, vários modelos de componentes diferentes emergiram. OpenCOM (COULSON et al., 2008), Fractal (BRUNETON et al., 2006), Spring (JOHNSON et al., 2007), EJB (DEMICHIEL; KEITH, 2007), OSGi (ALLIANCE,), CCM (OMG, 2002) e SCA (OASIS Open, 2011) são alguns exemplos destes modelos.

Diversas soluções têm sido criadas para lidar com a interoperabilidade entre sistemas distribuídos desenvolvidos em diferentes modelos de componentes. Uma dessas soluções é o objeto de estudo deste trabalho: O InteropFrame (NASCIMENTO, 2013).

O InteropFrame é um *middleware* extensível desenvolvido em Java que trata a questão da interoperabilidade entre sistemas desenvolvidos nos modelos de componentes OpenCOM ou Fractal através da geração automática de *proxies* para a comunicação remota. Estes *proxies* atuam como representantes locais a um sistema que repassam as chamadas de métodos para outros computadores dentro de uma rede, seja ela local ou a própria internet. No InteropFrame, as chamadas de métodos podem ser repassadas pela rede utilizando o mecanismo Java RMI (*Remote Method Invocation*) ou *Web Services* do tipo SOAP. Com isso o InteropFrame é capaz de fazer o *binding* (ligação

remota) entre partes distribuídas de modo que o utilizador não precise se preocupar com detalhes de comunicação remota nem de aspectos de interoperabilidade entre os modelos de componentes suportados. O InteropFrame é extensível do ponto de vista do desenvolvedor que deseje implementar o suporte a novos modelos de componentes e de *binding* através do desenvolvimento de *plug-ins* à parte.

Embora o InteropFrame funcione para os modelos de componentes propostos, ele possui algumas limitações. Uma dessas limitações é que o InteropFrame não é desenvolvido numa plataforma de componentes específica, sendo totalmente desenvolvido em Java. Esse fato acaba limitando o processo de extensibilidade do *middleware*. Outra limitação é a quantidade de modelos de componentes suportados – apenas Fractal e OpenCOM.

1.2 Objetivos do Trabalho

1.2.1 Objetivo Geral

Superar limitações do InteropFrame e estender o suporte a novos modelos de componentes.

1.2.2 Objetivos Específicos

- Estudo do InteropFrame;
- Fazer a portabilidade do InteropFrame para o modelo de componentes OSGi como forma de modularizá-lo em *plug-ins*. Dessa forma o suporte à extensibilidade ficará facilitado;
- Implementar o suporte ao OSGi como um modelo de componentes interoperável dentro do InteropFrame a fim de comprovar a extensibilidade do *middleware*;
- Avaliar impactos no desempenho do *middleware* devido à introdução do OSGi;
- Realizar melhorias no processo interno de comunicação remota do InteropFrame.

1.3 Contribuições deste Trabalho

As principais contribuições deste trabalho são:

- Evolução do InteropFrame.

- Disponibilização do código-fonte como forma de possibilitar estudos futuros à respeito da interoperabilidade entre sistemas distribuídos desenvolvidos em diferentes modelos de componentes.

1.4 Organização do Trabalho

Os capítulos subsequentes deste trabalho serão organizados da seguinte forma:

- Capítulo 2 - Apresentação dos conceitos básicos necessários para o entendimento do tema.
- Capítulo 3 - Descrição do InteropFrame e apresentação da solução proposta.
- Capítulo 4 - Considerações finais.

2 Conceitos Básicos

Com o objetivo de firmar o embasamento teórico do projeto e do aprimoramento do InteropFrame, serão elucidados neste capítulo os conceitos que foram utilizados no decorrer da pesquisa. Foram discutidos conceitos como Modularização, Componentes e Modelos de Componentes, Sistemas Distribuídos, Invocação Remota de Métodos, Serviços Web e Geração Automática de Código. Por fim, será feita uma breve discussão acerca dos conhecimentos elencados.

2.1 Modularização

Modularização significa a concepção de um sistema completo formado por módulos logicamente independentes (HALL et al., 2011). Ela é capaz de reduzir a complexidade do problema, dividindo-o em subproblemas mais simples, ou seja, dividindo-o em módulos (KNUTH, 1996). Um módulo define um limite lógico executável. Dessa forma os detalhes internos dele não são visíveis a outros módulos ou sistemas. Os únicos detalhes visíveis são aqueles que ele expõe explicitamente, ou seja, a API pública (HALL et al., 2011). Com isso é mais fácil de detectar problemas e resolvê-los, pois os módulos são, em princípio, independentes. Em sua característica um módulo é implementável, gerenciável, reutilizável, combinável e é uma unidade independente de software que provê interfaces a outros módulos ou sistemas (KNOERNSCHILD, 2012).

É confuso o conceito de Modularização e Orientação a Objeto. Ambas suportam a especialização, ou seja, quebram o sistema em partes pequenas dando a cada uma delas a sua devida responsabilidade. Entretanto, elas atuam de formas diferentes. Com a orientação a objeto é possível modularizar de forma lógica, referenciando a visibilidade do código. Dessa forma a orientação a objeto utiliza parte do conceito de modularização em seu contexto. Porém, a modularização abrange mais que isso. Ela pode ser utilizada tanto da forma física, onde é possível subdividir o código em vários arquivos, entretanto mantendo as mesmas dependências e comunicação entre eles, quanto da forma lógica, como na orientação a objeto (HALL et al., 2011).

A modularização ganhou a popularidade no início da década de 70, muito embora seja algo que ainda hoje não está tão presente nos requisitos não funcionais do desenvolvimento de software. Apesar disso é algo tão importante que traz grandes benefícios para a aplicação. Para isso é preciso aplicar os princípios de modularização para obter bons resultados. Princípios esses como alta coesão e baixo acoplamento. Com eles, uma das grandes vantagens que a modularização oferece é a reutilização. Isso se torna fácil quando um módulo é responsável por aquilo que realmente deve ser

e não tem um grande número de dependências. A depender do *framework* utilizado para a aplicar a modularização, a declaração das dependências pode ser feita de forma explícita como no OSGi. O que traz um ganho para a manutenção no código e para o melhor entendimento do mesmo (HALL et al., 2011).

2.1.1 Componentes e Modelos de Componentes

A engenharia de software baseada em componentes surgiu como uma abordagem para softwares de desenvolvimento de sistemas com base no reúso de componentes de software (SOMMERVILLE, 2011). Pressman (2011) menciona que componente é um bloco construtivo modular para software. Não existe consenso sobre um componente ser uma unidade independente de software que pode ser composta com outros componentes (SOMMERVILLE, 2011). Segundo Councill e Heineman (2001 apud SOMMERVILLE, 2011), componente é um elemento de software que está de acordo com um modelo de componente padrão e pode ser independentemente implantado e composto de acordo com um padrão de composição. Entretanto, Szyperski (2002 apud SOMMERVILLE, 2011) menciona que um componente de software é uma unidade de composição de interfaces contratualmente especificadas e pode ser implantado de forma independente, além de estar sujeito a ser composto por parte de terceiros.

Um componente funciona como um provedor de um ou mais serviços. Dessa forma, quando um sistema precisa de um serviço, ele chama um componente para fornecer esse serviço sem se preocupar onde esse componente está sendo executado, nem mesmo de características como linguagem de programação que o componente foi desenvolvido. Para isso, os componentes possuem dois tipos de interfaces relacionadas que refletem os serviços que o componente fornece (interface *provides*) e os serviços que o componente necessita (interface *requires*), como mostra a Figura 1 (SOMMERVILLE, 2011). De acordo com Crnkovic, Stafford e Szyperski (2011), uma interface de componente define um conjunto de propriedades funcionais de um componente.

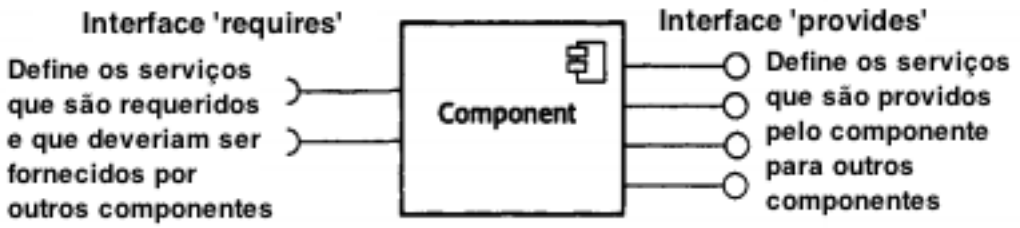

Figura 1 – Interfaces de componentes (SOMMERVILLE, 2011)

Um modelo de componente é uma definição de normas para implementação, documentação e implantação de componentes que garantem a interoperabilidade deles

(SOMMERVILLE, 2011). Existem diversos modelos de componentes, entretanto neste trabalho serão abordados os modelos OpenCOM, Fractal e OSGi.

2.1.1.1 Modelo OpenCOM

O OpenCOM é um modelo de componentes de baixo peso projetado para o desenvolvimento de *middlewares* em dispositivos de computação com poucos recursos (processamento, memória, armazenamento). Além de ser um modelo de baixo peso, o OpenCOM provê a capacidade de reconfiguração dinâmica de *middlewares* tanto no domínio estrutural quanto no comportamental (ROCHA, 2008).

O OpenCOM é fundamentado em três tecnologias (NASCIMENTO, 2013):

- Componentes - O modelo permite a especificação da estrutura de sistemas através do uso de componentes e conexões entre componentes (ROCHA, 2008). Os conceitos fundamentais no OpenCOM são interfaces, receptáculos e conexões. Uma interface representa uma unidade de provisão de serviços, enquanto que um receptáculo representa uma unidade de requerimento de serviços e é usado para tornar explícita a dependência de uma interface de um componente com outra. Uma conexão representa uma ligação entre um serviço fornecido por uma interface de um componente e um serviço requerido por um receptáculo de outro componente clarke2001;
- Reflexão computacional - O modelo OpenCOM foi projetado para suportar a reflexão computacional - que é a capacidade que um sistema tem de observar sua própria representação/estrutura e modifica-la em tempo de execução;
- *Frameworks* de componentes - Uma característica chave do OpenCOM é o uso da noção de *frameworks* de componentes. Um *framework* de componentes é definido no OpenCOM como um conjunto fortemente acoplado de componentes que coopera para resolver alguma área de interesse. O OpenCOM também fornece um protocolo de extensão bem definido para a aceitação de componentes adicionais que modificam ou estendem o comportamento do *framework* de componentes, além de restringir o modo como os componentes são organizados (COULSON et al., 2008).

Cada componente OpenCOM implementa quatro interfaces, além de interfaces personalizadas, como mostra a Figura 2 (GRACE, 2007 apud NASCIMENTO, 2013):

- ILifeCycle - fornece as operações para a inicialização e a finalização do ciclo de vida de um componente;

- IConnections (opcional) - oferece os métodos para modificar as interfaces ligadas aos receptáculos de um componente. Esta interface deve ser implementada por todos os componentes que possuem receptáculo;
- IMetaInterface - suporta a inspeção dos tipos de interfaces e receptáculos declarados pelo componente;
- IUnknown - é equivalente à interface do mesmo nome no Microsoft COM, isto é, é usada para obter a referência para a interface solicitada na instância do componente.

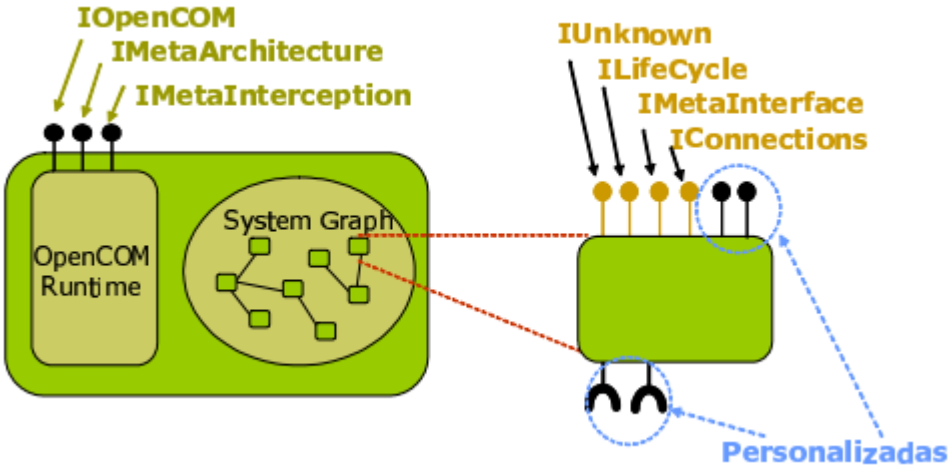

Figura 2 – Interfaces dos metamodelos do OpenCOM (ROCHA, 2008)

O OpenCOM implanta um substrato padrão em *runtime* que está disponível em todo o espaço de endereçamento. Insto é implementado através de um componente *singleton* chamado “OpenCOM” que exporta uma interface chamada IOpenCOM. O papel do *runtime* do OpenCOM é o de gerir um repositório de componentes disponíveis e, assim, permitir a criação e exclusão de componentes. Além disso, a interface IOpenCOM serve como um ponto centralizado para a submissão de todas as solicitações de conexão ou desconexão entre receptáculos e interfaces no seu espaço de endereçamento. Para facilitar a reconfiguração, o *runtime* registra cada criação e exclusão de cada componente ou conexão em um espaço de meta-estrutura chamado de *system graph* (grafo do sistema). Isto permite que o OpenCOM suporte consultas que, através de um identificador de conexão, fornece detalhes sobre o receptáculo e as interfaces participantes da conexão, juntamente com detalhes de seus componentes que os implementam (CLARKE et al., 2001).

2.1.1.2 Modelo Fractal

O Fractal é definido em (BRUNETON et al., 2006) como um modelo de componentes geral e extensível, projetado para implementar, implantar e gerenciar sistemas de software complexos, incluindo, em particular, sistemas operacionais e *middlewares*. As principais motivações do modelo são: (i) Composição de componentes, onde um componente pode conter outros componentes, permitindo uma visão uniforme das aplicações em vários níveis de abstração; (ii) Compartilhamento de componentes entre estruturas compostas de componentes, como forma de compartilhar recursos enquanto se mantém o encapsulamento de um componente; (iii) Capacidades reflexivas, para monitorar e controlar um sistema em execução; (iv) Capacidades de reconfiguração, como forma de implantar e configurar dinamicamente um sistema.

De acordo com Coupaye e Stefani (2007), o modelo de componentes Fractal suporta várias linguagens de programação, como por exemplo, Java e C, e de forma experimental .NET, SmallTalk, Python e C++.

No contexto do Fractal os componentes são entidades de *runtime* que estão em conformidade com o modelo, estão encapsuladas, possuem identificações únicas e suportam uma ou mais interfaces. As interfaces são os pontos únicos de interação entre os componentes e expressam a dependência desses em termos de interfaces requeridas e providas. Os *bindings* são os canais de comunicação entre as interfaces dos componentes (COUPAYE; STEFANI, 2007) (BRUNETON et al., 2006).

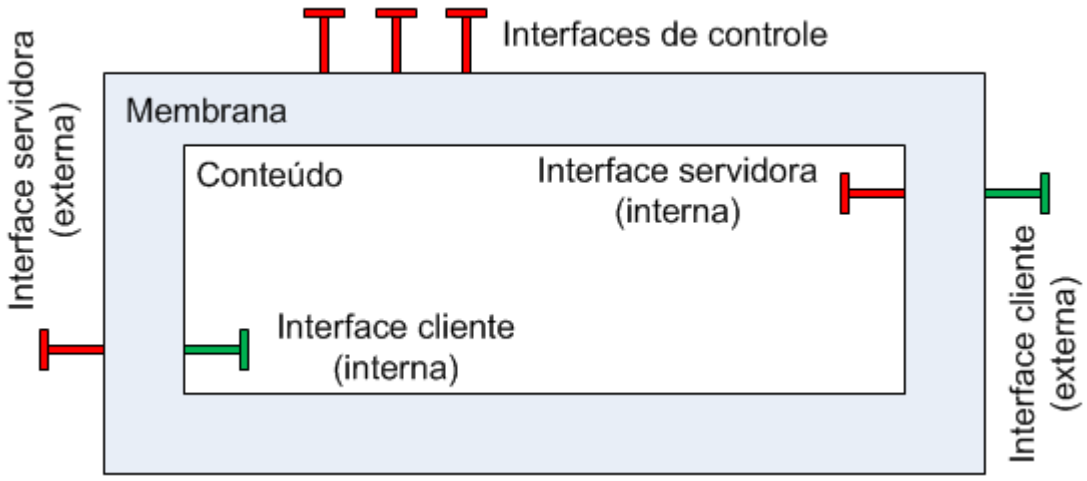

Figura 3 – Estrutura de um componente Fractal (BRUNETON et al., 2006 apud NASCIMENTO, 2013).

Um componente Fractal é a composição de uma membrana e um conteúdo, como pode ser observado na Figura 3. A membrana tem o papel de fornecer interfaces para um controle reflexivo sobre o conteúdo. O conteúdo consiste num conjunto finito de outros subcomponentes. A membrana de um componente pode ter interfaces internas,

acessíveis somente pelos subcomponentes internos, e externas, acessíveis de fora do componente. Além disso, uma membrana possui diversas interfaces de controle, que atuam como interceptadores entre as operações de chamada que entram e saem do componente, e adicionam comportamentos aos manipuladores de tais operações (**??**).

2.1.1.3 Modelo OSGi

Segundo Alliance (), o OSGi é um conjunto de especificações que definem um sistema de componentes dinâmico para o Java. Com essas especificações é possível criar um sistema composto dinamicamente por diversos componentes reusáveis. O OSGi permite que os componentes escondam suas implementações de outros componentes enquanto se comunicam através de serviços. Os serviços são objetos compartilhados de maneira especifica entre componentes.

A arquitetura do OSGi é composta por camadas como mostra a Figura 4. Elas são brevemente descritas a seguir (ALLIANCE,):

- *Bundles* - Os *bundles* são os componentes OSGi implementados pelos desenvolvedores.
- *Services* - A camada de Serviços conecta os *bundles* de maneira dinâmica. Os serviços são publicados pelos *bundles*, e são passíveis posteriormente de busca e conexão.
- *Life-Cycle* - Parte da API do OSGi que permite a instalar, desinstalar, executar, parar, e atualizar *bundles*.
- *Modules* - Camada que define como um *bundle* pode importar e exportar código.
- *Security* - Camada que manipula os aspectos de segurança do OSGi.
- *Execution Environment* - É o ambiente de execução. Define quais métodos e classes estarão disponíveis na plataforma específica.

O OSGi é apenas uma especificação de um *framework* para o desenvolvimento de aplicações modulares. Existem diversas implementações dessa tecnologia, sendo as mais conhecidas: Equinox, Felix e Knopflerfish. A primeira é uma implementação da especificação OSGi desenvolvida pela Eclipse Foundation. Ela é utilizada em diversas aplicações, inclusive na IDE Eclipse. Já a implementação Felix é desenvolvida e mantida pela Apache Software Foundation. A implementação Knopflerfish é desenvolvida e mantida pela Makewave.

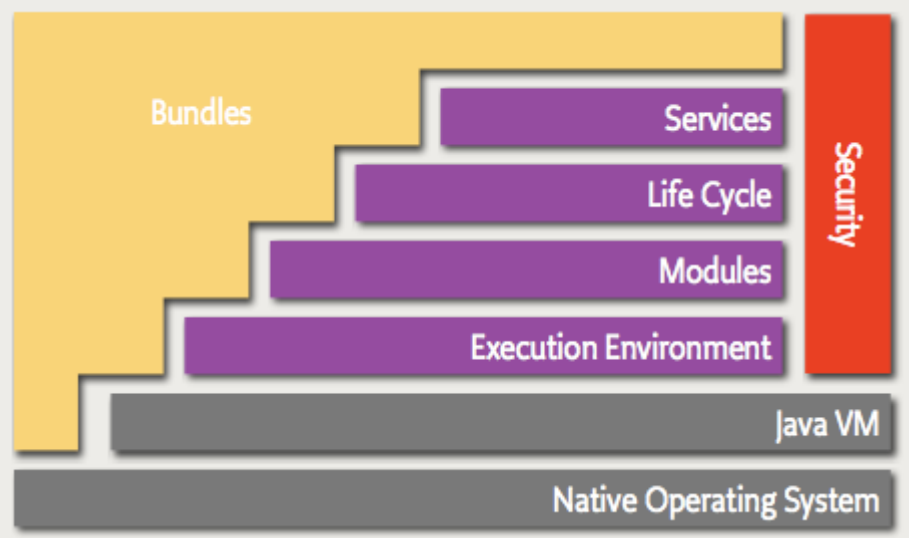

Figura 4 – Camadas da arquitetura OSGi (ALLIANCE,).

2.2 Sistemas Distribuídos

Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens (COULOURIS; DOLLIMORE; KINDBERG, 2007). Tanenbaum e Steen (2007) definem um sistema distribuído como uma coleção de computadores independentes que aparece para o usuário como um único sistema. Sommerville (2011) menciona que os sistemas distribuídos são mais complexos que os sistemas centralizados, o que os torna mais difíceis de projetas, implementar e testar. Apesar dessa complexidade, praticamente todos os grandes sistemas computacionais são distribuídos.

(COULOURIS; DOLLIMORE; KINDBERG, 2007 apud SOMMERVILLE, 2011) identifica vantagens da utilização de uma abordagem distribuída no desenvolvimento de sistemas:

- Compartilhamento de recursos: Um sistema distribuído permite o compartilhamento de recursos de hardware e software.
- Abertura: Os sistemas distribuídos são projetados para protocolos-padrão que permitem que os equipamentos e software de diferentes fornecedores sejam combinados.
- Concorrência: Em um sistema distribuído, vários processos podem operar simultaneamente em computadores separados na rede.
- Escalabilidade: Em princípio, os recursos de um sistema distribuído podem ser aumentados pela adição de novos recursos a depender da necessidade do

sistema.

- Tolerância a defeitos: Um sistema distribuído pode ser tolerante a algumas falhas de hardware e software dispondo de vários computadores e replicando as informações importantes para o sistema.

2.3 Invocação Remota de Métodos (RMI)

A RMI é uma extensão da invocação a método local que permite a um objeto que está em um processo invocar os métodos de um objeto que está em outro processo (COULOURIS; DOLLIMORE; KINDBERG, 2007). Harold (2004 apud NASCIMENTO, 2013) menciona que a diferença entre objetos remotos e objetos locais é que os objetos remotos estão localizados em máquinas virtuais diferentes, assim como na Figura 5. Devido à possibilidade de falhas independentes dos objetos invocadores e invocados, as RMIs têm semânticas diferentes das invocações a métodos locais, onde a transparência total não é necessariamente desejável (COULOURIS; DOLLIMORE; KINDBERG, 2007).

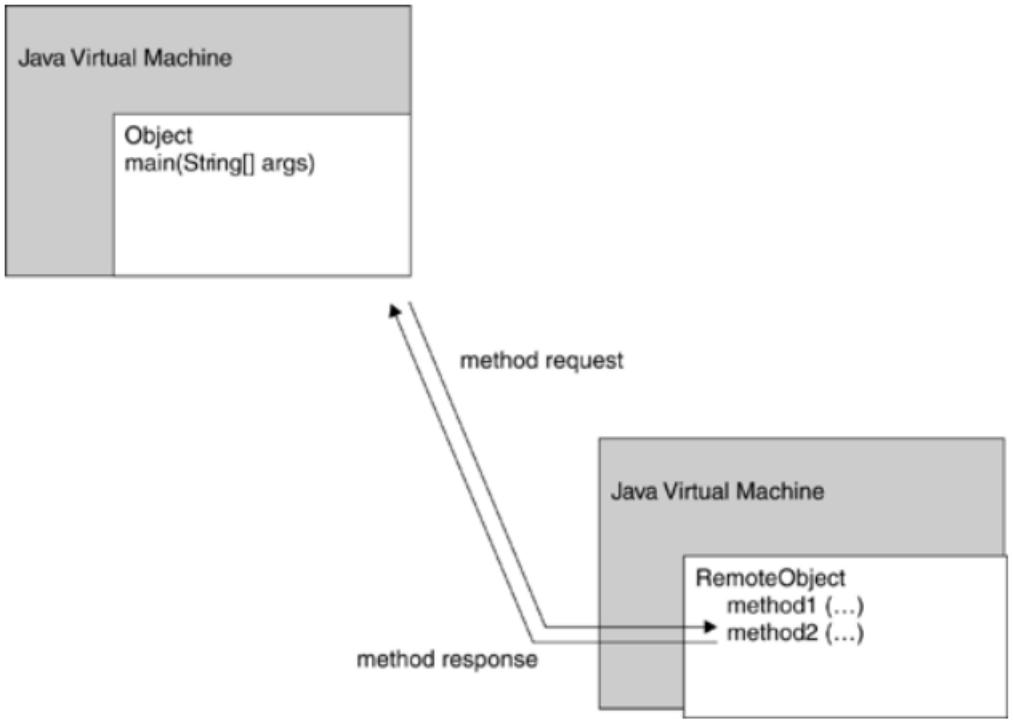

Figura 5 – Invocação Remota de métodos (REILLY; REILLY, 2002).

De acordo com Reilly e Reilly (2002), cada serviço RMI é definido por uma interface que descreve os métodos dos objetos que podem ser chamados remotamente. Segundo Nascimento (2013) e Reilly e Reilly (2002), essa interface deve ser compartilhada por todos os desenvolvedores uma vez que eles são incentivados a definir os

métodos que podem ser chamados remotamente antes mesmo da implementação. Nascimento (2013) menciona que várias implementações da interface podem ser criadas, e os desenvolvedores não precisam estar cientes de que a implementação está sendo usada e nem aonde está localizada.

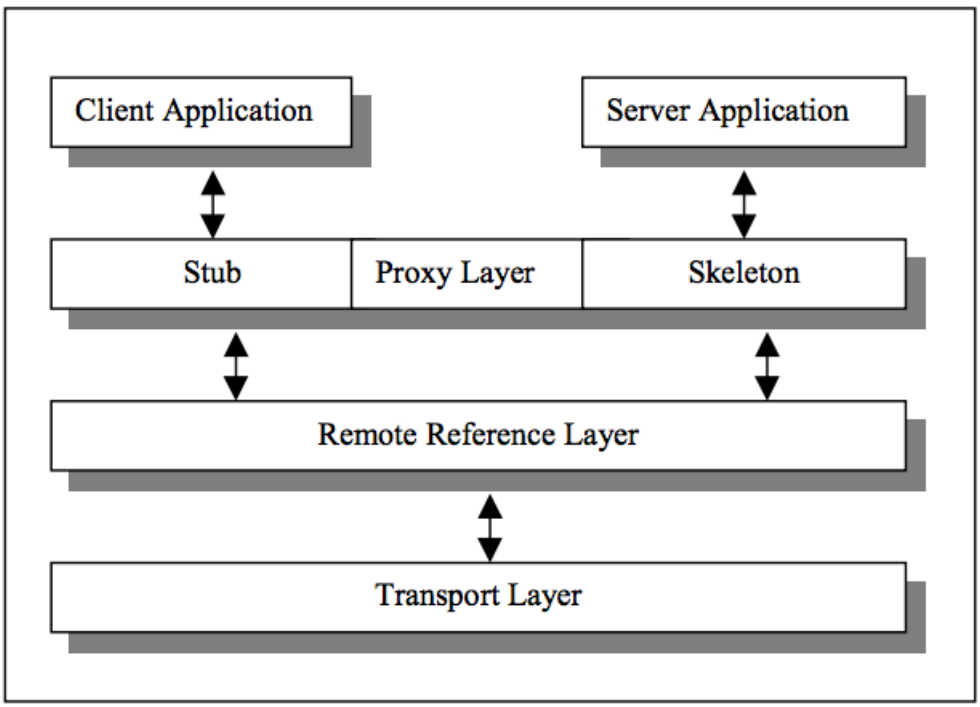

Figura 6 – Arquitetura em Camadas da RMI (RUIXIAN, 2000).

Segundo Ruixian (2000), a arquitetura do RMI baseia-se em quatro camadas, assim como a Figura 6: Camada de Aplicação, Camada de *Proxy*, Camada de Referência Remota e Camada de Transporte.

A camada de mais alto nível, a de aplicação, é onde encontram-se as implementações das aplicações tanto do lado cliente quanto do lado servidor. A segunda delas, a camada de *proxy*, é responsável pelas chamadas aos objetos remotos. Nela é feito o empacotamento do parâmetros e o retorno do objeto. Para isso o lado cliente e o lado servidor assumem papéis diferentes. O primeiro é representado por *Stub*, já o segundo é representado por *Skeleton*. Em seguida vem a terceira camada, a de referência remota. Nela é feita a abstração entre a Camada de *Proxy* e a Camada de Transporte. Por fim vem a camada de transporte que define uma conexão entre as máquinas cliente e servidor (RUIXIAN, 2000).

2.4 Serviços Web

Um serviço web (*web service*) fornece uma interface de serviço que permite aos clientes interagirem com servidores de uma maneira mais geral do que acontece

com os navegadores web (COULOURIS; DOLLIMORE; KINDBERG, 2007). De acordo com Deitel e Deitel (2010), um serviço web é um componente de software armazenado em um computador que pode ser acessado por um aplicativo(ou outro componente de software) em outro computador por uma rede. Um serviço web possui uma interface descrita em um formato processável por máquina, especificamente a WSDL (*Web Services Definition Language*). Outros sistemas interagem com um *Web Service* utilizando mensagens de acordo com um padrão, tipicamente utilizando HTTP com uma serialização de XML, além de outros padrões relacionados a Web.

Os clientes acessam as operações na interface de um serviço web por meio de requisições e respostas formatadas em XML (*Extensible Markup Language*) e, normalmente, transmitidas por HTTP (*HyperText Transfer Protocol*) (W3C, 2004). A XML é uma representação textual que, embora mais volumosa do que as representações alternativas, foi adotada por sua legibilidade e pela consequente facilidade de depuração (COULOURIS; DOLLIMORE; KINDBERG, 2007) De acordo com Deitel e Deitel (2010), o serviço web pode ser apoiado em duas arquiteturas. A primeira é baseada no *Simple Object Access Protocol* (SOAP) e a segunda é baseada no *Representational State Transfer* (REST).

Segundo Sommerville (2011), SOAP é um padrão de trocas de mensagem que oferece suporte à comunicação entre serviços. O SOAP é um protocolo independente de plataforma que utiliza a XML para fazer chamadas de procedimento remoto, geralmente sobre o HTTP (DEITEL; DEITEL, 2010). De acordo com Coulouris, Dollimore e Kindberg (2007), originalmente o protocolo SOAP era baseado apenas em HTTP, mas a versão atual é projetada para usar uma variedade de protocolos de transporte, incluindo o SMTP, TCP ou UDP.

Segundo Deitel e Deitel (2010), o REST refere-se a um estilo arquitetônico de implementar serviços Web. REST é uma estratégia com um estilo de operações muito restrito, no qual os clientes usam URLs e as operações HTTP, GET, PUT, DELETE e POST para manipular recursos representados em XML (FIELDING, 2000). Segundo Deitel e Deitel (2010), o REST também não está limitado a retornar dados no formato XML. Ele pode utilizar vários formatos, como XML, JSON, HTML, texto sem formatação e arquivos de mídia.

2.5 Geração Automática de Código

Um gerador de código é um sistema desenvolvido para criar automaticamente código fonte de alto nível em linguagens de programação como .NET, C++, C#, Java e outros (ADAMATII, 2006). A geração automática de código ajuda a aumentar a eficácia da produção de software complexo, reduzindo o custo e tempo associado com o esforço

de codificação (KORNECKI; JOHRI, 2006). Segundo Adamatii (2006), é possível criar, a partir de um banco de dados, objetos de acesso à base de dados, telas para consulta, pesquisa e edição de dados e toda a base para um sistema, restando à equipe de desenvolvimento, implementar regras de negócio e especialização das funcionalidades. Com isso, um gerador de código automático pode trazer vantagens como qualidade no código, consistência, produtividade e abstração.

Algumas plataformas utilizam mecanismos ou ferramentas para automação de geração de código para melhorar a produtividade e eficiência nos processos de desenvolvimento de software, como por exemplo, O MDA (*Model-driven Architecture*) (OMG, 2012) - que é uma abordagem para desenvolvimento de sistemas dirigido a modelos e o Acceleo (ACCELEO, 2012) - que é um *plug-in* do Eclipse que baseado em MDA permite a geração automática de código a partir de modelos, como por exemplo, um modelo UML ou um metamodelo definido pelo usuário.

2.6 Discussão

A proposta de extensão e modularização do InteropFrame exige conhecimentos diversos que foram elencados nesse capítulo. Mais especificamente foi observado o conceito de modularização, como forma de reconstruir o InteropFrame de forma modular utilizando a distribuição Equinox do OSGi. Além disso, será feita a extensão do *framework* para o suporte ao modelo de componentes OSGi de forma interoperável dentro da ferramenta.

Além da proposta de modularização, o módulo Configurador Distribuído, que será apresentado no capítulo 3, será refeito a partir de soluções existentes no OSGi, que utilizam conceitos de comunicação remota.

3 Solução Proposta

Neste capítulo será apresentado inicialmente o InteropFrame em sua implementação atual. Em seguida será apresentada a solução proposta para lidar com as limitações do InteropFrame, além de uma proposta de extensão para o suporte ao modelo de componentes OSGi dentro da ferramenta.

3.1 InteropFrame

Segundo Nascimento (2013), o papel do *framework* InteropFrame é o de prover uma solução de *binding* (interconexão) transparente entre componentes distribuídos de modelos diferentes. Ele possibilita que os componentes envolvidos na construção de aplicações distribuídas interajam através dos mecanismos de interoperabilidade providos pelo *framework*.

Com o InteropFrame é possível tornar interoperáveis sistemas desenvolvidos nos modelos de componentes OpenCOM e Fractal. Esta interoperabilidade pode ocorrer tanto em sistemas locais, como também com partes distribuídas pela rede. O *binding* remoto entre os componentes é suportado através dos mecanismos Java RMI e *Web Services SOAP*.

A Figura 7 apresenta a arquitetura do InteropFrame. A seguir são explicados os módulos dessa arquitetura (NASCIMENTO, 2013):

- Configurador Distribuído (CD) - módulo responsável pelo gerenciamento do serviço de interoperabilidade entre os componentes distribuídos. Este módulo coordena e controla as operações dos demais módulos do *framework* distribuído;
- *Plug-ins* de Modelos de Componentes (PMC) - cada *plug-in* permite que o InteropFrame suporte um modelo de componentes específico. Um *Plug-in* de modelo de componente é composto pelos seguintes submódulos:
- Gerador de *Proxies* (GP) - responsável pela geração automática dos *proxies* que possibilitam a interoperabilidade entre os componentes distribuídos de modelos diferentes. Este módulo baseia-se na geração de código a partir de *templates* pré-definidos para cada modelo de componentes específico;
- Montador de *Proxies* (MP) - responsável pela execução sob demanda dos *proxies* criados pelo GP, bem como a disponibilização do serviço de interoperabilidade entre os componentes distribuídos;

- Repositório de *Proxies* (RP) - repositório para armazenamento dos *proxies* gerados pelo GP.

- *Plug-ins* de Geradores de *Bindings* (GB) - cada *plug-in* gerador de *binding* é responsável pela geração automática do código-fonte de um tipo diferente de *binding* entre componentes remotos. O módulo Gerador de *Proxies* faz uso de um tipo específico de Gerador de *Binding* para enxertar nos *proxies* o código que promove a interconexão remota.

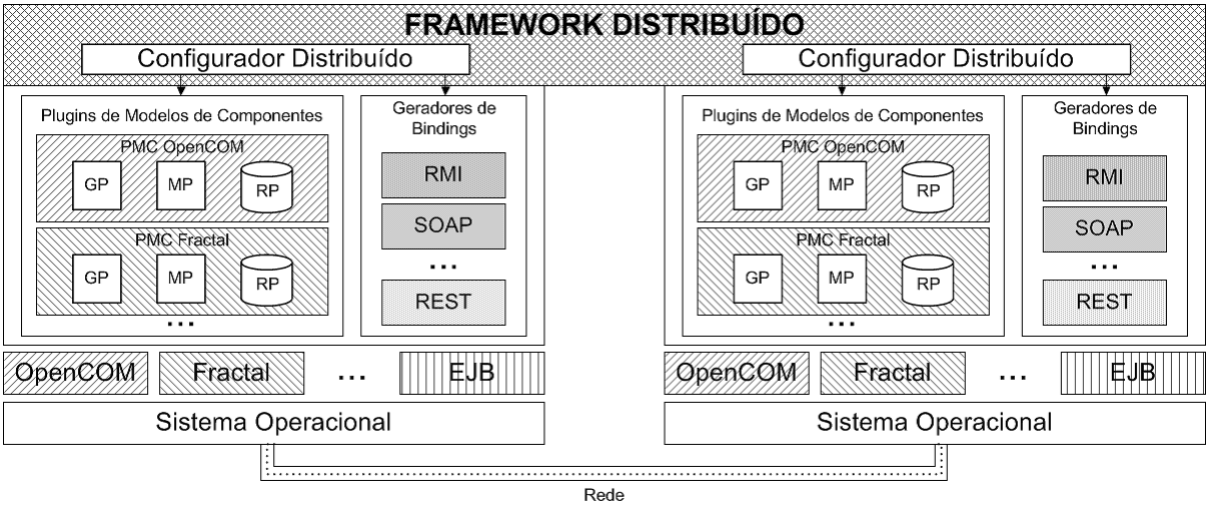

Figura 7 – Arquitetura do InteropFrame (NASCIMENTO, 2013).

A Figura 8 mostra o funcionamento do InteropFrame. Neste cenário o usuário deseja interconectar os componentes “A” e “B”, desenvolvidos respectivamente em OpenCOM e Fractal. O componente “A” é tratado aqui como componente cliente pois requisita os serviços do componente servidor “B” de modo remoto.

O detalhamento deste processo de funcionamento para o exemplo da Figura 8 é descrito a seguir (NASCIMENTO, 2013):

- Do lado do componente cliente

- (1C) O Configurador Distribuído (CD) verifica se o *proxy* do lado cliente, necessário para promover a interoperabilidade, já se encontra no Repositório de *Proxies* (RP), caso contrário ele solicita a geração do mesmo no passo 2C. Caso o componente *proxy* já exista, o próximo passo será o 5C, onde esse componente será utilizado pelo Montador de *Proxies* (MP);

- (2C) O CD solicita ao submódulo Gerador de *Proxies* (GP) do modelo de componentes OpenCOM para gerar automaticamente o código do componente “X” que representa o *proxy* do lado cliente;

- (3C) O GP solicita ao submódulo Gerador de *Bindings* (GB) do RMI para gerar automaticamente a parte do código do componente “X” responsável pela comunicação remota;
- (4C) O GP armazena no RP o componente “X” gerado;
- (5C) O CD solicita ao MP que proceda com a inicialização do *proxy* do lado cliente;
- (6C) O MP obtém e inicializa o componente “X” do lado cliente no ambiente de execução OpenCOM conectando o receptáculo do componente “A” à interface provida do *proxy* “X”. O *proxy* “X” do lado cliente representa o componente “B” no lado cliente e tem seus serviços requisitados pelo componente “A”.

- Do lado do componente servidor

- (1S) O Configurador Distribuído (CD) verifica se o *proxy* do lado servidor já se encontra no Repositório de *Proxies* (RP), caso contrário ele solicita a geração do mesmo no passo 2S. Caso o componente *proxy* já exista, o próximo passo será o 5S, onde esse componente será utilizado pelo Montador de *Proxies* (MP);
- (2S) O CD solicita ao submódulo GP do modelo de componentes Fractal para gerar automaticamente o código do componente “Y” que representa o *proxy* (também chamado de *skeleton*) do lado servidor;
- (3S) O GP solicita ao submódulo GB do RMI para gerar automaticamente a parte do código do componente “Y” responsável pela comunicação remota;
- (4S) O GP armazena no RP o componente “Y” gerado;
- (5S) O CD solicita ao MP que proceda com a inicialização do *proxy* do lado servidor;
- (6S) O MP obtém e inicializa o componente “Y” do lado servidor no ambiente de execução Fractal conectando o receptáculo do *proxy* “Y” à interface provida do componente “B”. O *proxy* “Y” do lado servidor representa o componente “A” no lado servidor que requisita os serviços do componente “B”.

Com o *binding* executado, o componente “A” agora pode utilizar os serviços do componente “B” de forma transparente. Quando uma requisição é feita no componente “A” ela é repassada via RMI do componente *proxy* “X” para o componente *skeleton* “Y” e este por sua vez repassa para o componente “B”. A resposta dessa requisição é feita pelo caminho inverso, de “B” para “Y”, de “Y” para “X” e de “X” para “A”. Na prática, “A” e “X” são componentes do modelo OpenCOM, assim como “B” e “Y” são do modelo

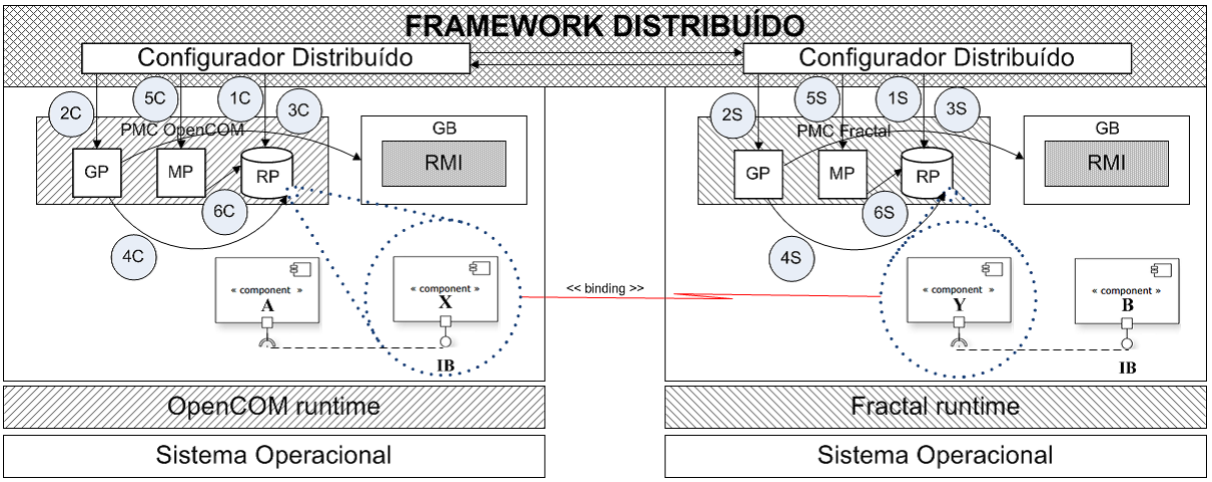

Figura 8 – Funcionamento do InteropFrame (NASCIMENTO, 2013).

Fractal. Os componentes “X” e “Y” se comunicam através de RMI, garantindo assim a interoperabilidade entre os componentes “A” e “B”.

3.2 Modularização do InteropFrame

O InteropFrame foi desenvolvido em Java “puro”, de forma a permitir a extensibilidade para novos modelos de componentes e de *bindings*. Cada *plug-in* de modelo de componentes ou de *binding* fornece o suporte a um modelo de componentes ou de *binding* específico. Com o desenvolvimento de novos *plug-ins* a ferramenta passa a suportar novos modelos.

Segundo Hall et al. (2011) o Java provê alguns aspectos de modularização através da orientação a objetos, porém não foi proposto para suportar modularização de alta granularidade. hall2011 ainda cita algumas limitações do Java no quesito modularização:

- Baixo nível de controle de visibilidade de código: Os modificadores de acesso do Java (*public, protected e private*) tratam em baixo nível o encapsulamento da orientação a objetos e não no nível de particionamento lógico do sistema. Em Java, um *package* (pacote) é tipicamente utilizado para particionar código. Para este código ser visível por um outro *package*, ele deve ser declarado como *public*. Algumas vezes, a estrutura lógica da aplicação faz chamadas a códigos de *packages* diferentes, significando que qualquer dependência entre os pacotes deve ser exposta como *public*. Dessa maneira, os detalhes de implementação tornam-se públicos, dificultando a evolução do sistema devido a possível criação de dependências da API não pública.

- Conceito de *Classpath* propenso a erros: Aplicações são compostas de várias

versões de bibliotecas e componentes. O *Classpath* do Java não lida com versões de código, retornando assim o primeiro que encontra. O modo de construção do *Classpath* não permite especificar versões de um mesmo código. Em Java apenas se vai colocando as bibliotecas (comumente arquivos JAR) até que a JVM (Java Virtual Machine) pare de acusar erros sobre classes faltantes.

- Implantação limitada e suporte a gerenciamento: Não há maneira fácil em Java de se implantar um conjunto particular de dependências de código versionadas e executar a aplicação. Também é dificultada a evolução da aplicação e seus componentes após a implantação. O Java não possui um suporte direto à criação de *plug-ins* dinâmicos, o que é conseguido apenas através do uso de *Class Loaders* - mecanismos de baixo nível e propensos a erros.

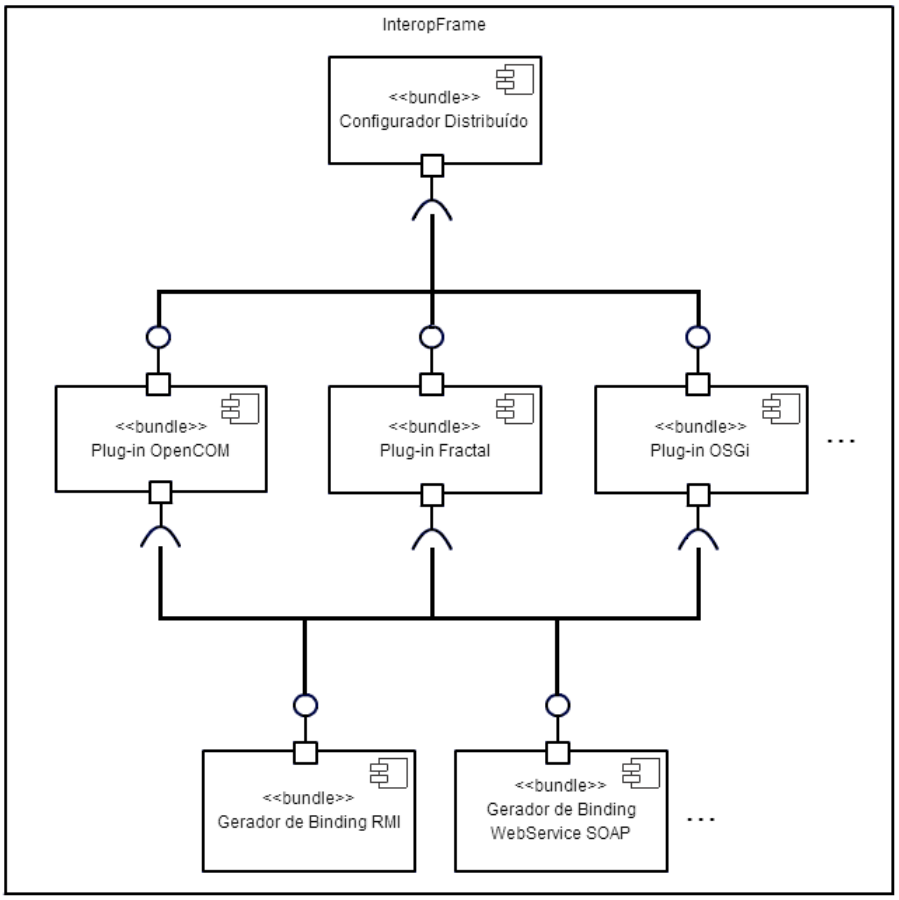

Figura 9 – Bundles do InteropFrame em OSGi

Tendo em vista as limitações do Java, necessita-se de uma maneira mais eficiente para a modularização do InteropFrame. A proposta deste trabalho consiste na adoção do OSGi como plataforma de modularização. A extensibilidade do InteropFrame passará a ter um suporte facilitado, uma vez que a aplicação será desenvolvida em *bundles* independentes como mostra a Figura 9. Cada *Plug-in* de Modelo de Componente

e seus respectivos submódulos seriam portados para *bundles* OSGi individuais. Da mesma forma, cada Gerador de *Binding* também se tornaria um *bundle* independente, bem como o núcleo do InteropFrame e o Configurador Distribuído.

3.3 Solução de comunicação do Configurador Distribuído

O Configurador Distribuído, módulo responsável pelo gerenciamento entre as partes distribuídas do InteropFrame, é desenvolvido utilizando a tecnologia Java RMI. O Configurador Distribuído atua na comunicação remota entre os lados servidor e cliente do InteropFrame. O lado servidor é responsável por propagar pela rede uma interface provida de um componente de um dado sistema. O lado cliente faz a utilização desse serviço fornecido pelo lado servidor.

Para garantir uma comunicação distribuída de forma modular, este trabalho propõe a adoção da comunicação remota baseada no ECF (*Eclipse Communication Framework*) para a implementação do Configurador Distribuído. O ECF consiste num conjunto *frameworks* para a construção de servidores distribuídos e aplicações. Provê implementação modular do padrão de serviços remotos do OSGi, juntamente ao suporte para *Web Services REST* (FOUNDATION,).

3.4 Extensão para o modelo de componentes OSGi

Além da modularização utilizando o OSGi, também é proposto neste trabalho a extensão para o suporte ao modelo de componentes OSGi dentro do InteropFrame.

Após a portabilidade do InteropFrame para a plataforma OSGi, será criado um novo *plug-in* para que o *framework* passe a suportar o OSGi como um modelo de componentes interoperável com os já existentes (OpenCOM e Fractal). Essa proposta tem como objetivo avaliar o processo de desenvolvimento de um novo *plug-in* de modelo de componentes.

4 Considerações Finais

O InteropFrame é uma solução que auxilia na resolução de problemas relacionados à interoperabilidade entre componentes distribuídos de modelos heterogêneos. Porém, essa solução está apenas limitada aos modelos OpenCOM e Fractal, além de promover a comunicação remota apenas utilizando Java RMI ou *Web Service SOAP*. Outra limitação é que o InteropFrame foi desenvolvido em Java "puro", o que dificulta a sua extensibilidade. Outro problema do InteropFrame é a sua comunicação interna (entre os lados cliente e servidor), que é feita através de Java RMI.

Este trabalho propõe uma possível solução para essa limitação através da utilização do OSGi como forma de modularizar o InteropFrame, além da extensão para o OSGi como um modelo de componentes interoperável dentro da ferramenta. Também propõe a resolução dos problemas relacionados à comunicação interna através dos mecanismos de comunicação providos pelo *Eclipse Communication Framework*. Dessa forma, o InteropFrame torna-se mais extensível, coeso e desacoplado.

Referências

ACCELEO. *Acceleo's website*. 2012. Disponível em: <http://www.acceleo.org/pages/home/en>. Acesso em: 16 jun. 2012.

ADAMATII, P. M. FUMIGANT: Gerador de código Java a partir de Base de Dados. Faculdade Cenecista Nossa Senhora dos Anjos, Gravataí. 2006.

ALLIANCE, O. *The OSGi Architecture*. Disponível em: <http://www.osgi.org/Technology/WhatIsOSGi>. Acesso em: 22 maio 2014.

BLAIR, G. et al. Interoperability in Complex Distributed Systems. In: BERNARDO, M.; ISSARNY, V. (Ed.). *Formal Methods for Eternal Networked Software Systems*. Springer Berlin / Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6659). p. 1–26. ISBN 978-3-642-21454-7. 10.1007/978-3-642-21455-4_1. Disponível em: <http://dx.doi.org/10.1007/978-3-642-21455-4_1>.

BROMBERG, Y. et al. Bridging the interoperability gap: overcoming combined application and middleware heterogeneity. *12th IFIP/ACM/USENIX International Middleware Conference*, Springer, p. 390–409, 2011.

BRUNETON, E. et al. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 36, n. 11-12, p. 1257–1284, set. 2006. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.v36:11/12>.

CLARKE, M. et al. An efficient component model for the construction of adaptive middleware. In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. London, UK, UK: Springer-Verlag, 2001. (Middleware '01), p. 160–178. ISBN 3-540-42800-3. Disponível em: <http://dl.acm.org/citation.cfm?id=646591.697779>.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Sistemas Distribuídos: Conceitos e Projeto*. 4 ed.. ed. Porto Alegre, RS, Brasil: Bookman, 2007. ISBN 9788560031498. Disponível em: <http://books.google.com.br/books?id=KSZ1rIRWmUoC>.

COULSON, G. et al. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 26, n. 1, p. 1:1–1:42, mar. 2008. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/1328671.1328672>.

COUNCILL, B.; HEINEMAN, G. T. Component-based software engineering. In: HEINEMAN, G. T.; COUNCILL, W. T. (Ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. cap. Definition of a Software Component and Its Elements, p. 5–19. ISBN 0-201-70485-4. Disponível em: <http://dl.acm.org/citation.cfm?id=379381.379438>.

COUPAYE, T.; STEFANI, J.-B. Fractal component-based software engineering. In: SÃ$\frac{1}{4}$DHOLT, M.; CONSEL, C. (Ed.). *Object-Oriented Technology. ECOOP 2006 Workshop Reader*. Springer Berlin Heidelberg, 2007, (Lecture Notes in

Computer Science, v. 4379). p. 117–129. ISBN 978-3-540-71772-0. Disponível em: <http://dx.doi.org/10.1007/978-3-540-71774-4_13>.

CRNKOVIC, I.; STAFFORD, J.; SZYPERSKI, C. Software components beyond programming: From routines to services. *Software, IEEE*, IEEE, v. 28, n. 3, p. 22–26, 2011.

DEITEL, H.; DEITEL, P. *Java: como programar*. 8 ed.. ed. SÃ£o Paulo, SP, Brasil: PEARSON BRASIL, 2010. ISBN 9788576055631. Disponível em: <http://books.google.com.br/books?id=xWMVRAAACAAJ>.

DEMICHIEL, L.; KEITH, M. *JSR 220: Enterprise JavaBeans*. 3.0. ed. Santa Clara, 2007.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado), 2000. AAI9980887.

FOUNDATION, T. E. *Eclipse Communication Framework Project Home*. Disponível em: <http://www.eclipse.org/ecf/>. Acesso em: 05 junho 2014.

GRACE, P. The OpemCOMJ Handbook. Technical report, Distributed Multimedia Research Group. Lancaster University, Lancaster. 2007.

HALL, R. et al. *Osgi in Action: Creating Modular Applications in Java*. 1st. ed. Greenwich, CT, USA: Manning Publications Co., 2011. ISBN 1933988916, 9781933988917.

HAROLD, E. R. *Java Network Programming*. 3. ed. Beijing: O'Reilly, 2004. ISBN 978-0-596-00721-8.

INVERARDI, P.; ISSARNY, V.; SPALAZZESE, R. A theory of mediators for eternal connectors. *Leveraging Applications of Formal Methods, Verification, and Validation*, Springer, p. 236–250, 2010.

JOHNSON, R. et al. *The spring framework - reference documentation*. 2.0.6. ed. [S.l.], 2007.

KNOERNSCHILD, K. *Java Application Architecture: Modularity Patterns with Examples Using OSGi*. Prentice Hall, 2012. (Agile Software Development Series). ISBN 9780321247131. Disponível em: <http://books.google.com.br/books?id=iOtwFoU1Dt4C>.

KNUTH, D. *Selected Papers on Computer Science*. University of Chicago Press, 1996. (CSLI lecture notes: Center for the Study of Language and Information). ISBN 9781881526919. Disponível em: <http://books.google.com.br/books?id=hqBpQgAACAAJ>.

KORNECKI, A. J.; JOHRI, S. Automatic Code Generation: Model-Code Semantic Consistency. In: *Software Engineering Research and Practice'06*. [S.l.: s.n.], 2006. p. 191–197.

NASCIMENTO, S. C. do.

Um Framework Extensível para Interoperabilidade Dinâmica entre Componentes Distribuídos — Universidade Federal de Sergipe, Aracaju, SE, Brasil, 2013.

OASIS Open. *Service component architecture (SCA) assembly model specification*. 1.1. ed. [S.l.], 2011.

OMG. *OMG CORBA component model (CCM) specification*. 3.0. ed. Needham, 2002.

OMG. *Model Driven Architecture*. 2012. Disponível em: <http://www.omg.org/mda/>. Acesso em: 16 jun. 2012.

PRESSMAN, R. S. *Engenharia de Software: Uma Abordagem Profissional*. 7 ed.. ed. Porto Alegre, RS, Brasil: AMGH, 2011. ISBN 9788563308337. Disponível em: <http://books.google.com.br/books?id=eRIOuQAACAAJ>.

REILLY, D.; REILLY, M. *Java network programming and distributed computing*. ADDISON WESLEY Publishing Company Incorporated, 2002. ISBN 9780201710373. Disponível em: <http://books.google.com.br/books?id=xKJQAAAAMAAJ>.

ROCHA, T. da. *Serviços de transação abertos para ambientes dinâmicos*. Tese (Doutorado) — Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, Brasil, 2008.

ROUVOY, R.; MERLE, P. Leveraging component-based software engineering with Fraclet. *Annals of Telecommunications*, Springer Paris, v. 64, p. 65–79, 2009. ISSN 0003-4347. 10.1007/s12243-008-0072-z. Disponível em: <http://dx.doi.org/10.1007/s12243-008-0072-z>.

RUIXIAN, B. Distributed computing via rmi and corba. 2000. Disponível em: <http://europepmc.org/abstract/CIT/497529>.

SOMMERVILLE, I. *Engenharia de Software*. 9 ed.. ed. São Paulo, SP, Brasil: Pearson Brasil, 2011. ISBN 9788579361081. Disponível em: <http://books.google.com.br/books?id=H4u5ygAACAAJ>.

SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201745720.

TANENBAUM, A.; STEEN, M. van. *Sistemas distribuídos: princípios e paradgmas*. 2 ed.. ed. Pearson Prentice Hall, 2007. ISBN 9788576051428. Disponível em: <http://books.google.com.br/books?id=r2SGPgAACAAJ>.

W3C, C. *Web Services Glossary*. 2004. Disponível em: <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/>. Acesso em: 19 maio 2014.