Vinicius Nordi Esperança

Uma abordagem dirigida por modelos para distribuição tardia de aplicações

Brasil

2016, maio

Vinicius Nordi Esperança

Uma abordagem dirigida por modelos para distribuição tardia de aplicações

Exame de Dissertação apresentado ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.

Universidade Federal de São Carlos – UFSCar

Departamento de Computação

Programa de Pós-Graduação

Orientador: Daniel Lucrédio

Brasil

2016, maio

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária UFSCar
Processamento Técnico
com os dados fornecidos pelo(a) autor(a)

Esperança, Vinicius Nordi
E774a Uma abordagem dirigida por modelos para distribuição tardia de aplicações / Vinicius Nordi Esperança. -- São Carlos : UFSCar, 2016.
72 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2016.

1. Microsserviços. 2. Desenvolvimento de software dirigido por modelos. 3. Distribuição tardia. I. Título.

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de dissertação de mestrado do candidato Vinicius Nordi Esperança, realizada em 07/03/2016.

Prof. Dr. Daniel Lucrédio
(UFSCar)

Prof. Dr. Luis Carlos Trevelin
(UFSCar)

Prof. Dr. Vinicius Cardoso Garcia
(UFPE)

Certifico que a sessão de defesa foi realizada com a participação à distância do membro Prof. Dr. Vinicius Cardoso Garcia e, depois das arguições e deliberações realizadas, o participante à distância está de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa do aluno Vinicius Nordi Esperança .

Prof. Dr. Daniel Lucrédio
Presidente da Comissão Examinadora
(UFSCar)

Resumo

A necessidade de melhoria de sistemas de software para adequarem-se às novas tecnologias é tema constante de pesquisas. Conforme a computação evolui, novos desafios surgem, e novas soluções devem ser criadas. A crescente utilização de diversos tipos de aparelhos para acesso a sites e software, além da facilidade que a Internet proporciona ao acesso de informações, força pesquisadores da área a manterem grandes esforços melhorando aplicações já desenvolvidas, ou mesmo pensando em formas de facilitar o desenvolvimento de software de forma a rodar em vários dispositivos. O intuito geral desta pesquisa de mestrado foi com foco nesse problema, de distribuir sistemas de software inicialmente projetados para rodar em um único computador de maneira a eliminar muitas tarefas que o desenvolvedor repetiria exaustivamente nesse processo. A abordagem permite que programadores possam, a partir do código fonte de alguma aplicação e modelos de distribuição, realizar a divisão do seu software de maneira rápida, a fim de testar diversas distribuições sem custo de tempo. Utilizando conceitos de DSDM (Desenvolvimento de Software Dirigido a Modelos), analisadores e geradores de código, e conceitos de microsserviços, foi desenvolvido um particionador que distribui o código fonte de uma aplicação em diversos servidores, criando pequenos serviços para cada, fazendo esses serviços trocarem informações entre si, mantendo as funcionalidades da aplicação intactas. O particionador dá flexibilidade ao programador de escolher a disposição das classes entre os serviços que serão criados. Desenvolvedores podem então, facilmente, testar diversas maneiras de distribuir o código de sua aplicação, sem a necessidade de ter que desenvolver os serviços e a comunicação entre eles. Essa abordagem foi denominada distribuição tardia. Estudos experimentais foram realizados para garantir a validade do projeto. No total, cinco testes diferentes foram conduzidos, a fim de verificar se a criação dos serviços seria realizada como definida pelo programador. Em todos os casos o resultado foi satisfatório, permitindo a distribuição do código do software utilizado entre os servidores designados.

Palavras-chaves: Microsserviços, Desenvolvimento de Software Dirigido por Modelos, Distribuição Tardia.

Abstract

The need to improve software systems to adapt to the new technologies is a constant topic of research. As computing evolves, new challenges emerge and new solutions must be created. The increasing use of various types of devices for access to sites and software and the ease that the Internet provides the information access, force researchers to keep great efforts improving already developed applications, or even thinking of ways to facilitate the development of software to run on multiple devices. The general purpose of this master's research was focused on this problem, distribute software systems initially designed to run on a single computer in order to eliminate many tasks that the developer thoroughly repeat this process. The approach allows developers to use the source code of any application and distribution models, perform the division of its software quickly in order to test different distributions without time cost. Using concepts of MDD (Model-Driven Development), analyzers and code generators, and concepts of microservices, it developed a partitioner that distributes the source code of an application on multiple servers, creating small services for each, making these services exchange information with each other while maintaining the functionality of the application intact. The partitioner gives flexibility to the programmer to choose the arrangement of classes among the services that will be created. Developers can then easily test different ways to distribute the code of the application without the need of having to develop services and communication between the new services. This approach was called late distribution. Experimental studies were performed to ensure the validity of the project. In total, five different tests were conducted in order to verify the creation of services would be performed as defined by the programmer. In all cases the result was satisfactory, allowing distribution of software code used between the designated servers.

Key-words: Microservices, Model-Driven Development, Late Distribution.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AST	*Abstract Syntax Tree*
CIM	*Computation Independent Model*
CPU	*Central Processing Unit*
DSDM	*Desenvolvimento de Software Dirigido por Modelos*
DSL	*Domain-Specific Language*
EMF	*Eclipse Modeling Framework*
JDT	*Java Development Tools*
JMI	*Java Metadata Interface*
JVM	*Java Virtual Machine*
MBE	*Model-Based Enginnering*
MDA	*Model-Driven Architecture*
MDD	*Model-Driven Development*
MDE	*Model-Driven Engineering*
MOF	*Meta-Object Facility*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
PVM	*Parallel Virtual Machine*
QoS	*Quality of Service*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
PSM	*Platform-Specific Model*
REST	*Representional State Transfer*
RMI	*Remote Method Invocation*

UML	*Unified Modeling Language*
URI	*Unified Modeling Language*
WSDL	*Web Service Description Language*

Sumário

1 Introdução

1.1 Contexto

O uso de software no cotidiano das pessoas aumenta a cada dia mais, e pode-se dizer que já faz parte de, praticamente, qualquer atividade humana. A tendência de utilização da informática nos mais diversos setores produtivos é muito grande, principalmente, levando-se em consideração o surgimento de novas tecnologias. Dessa forma, o uso da informática está atingindo todos os públicos, gerando uma demanda por software excessivamente grande (Thommazo 2014). Além disso, com o advento da computação em nuvem, software cada vez mais tem sido desenvolvido para estar presente na Internet. Há também a necessidade de, em alguns casos, parte do sistema melhorar seu desempenho devido a existência de certo gargalo na aplicação, dividir melhor a memória utilizada na aplicação entre diversos computadores ou então possibilitar que dados mais críticos estejam disponíveis em servidores mais seguros.

Com o pensamento nessas situações, a arquitetura de microsserviços surge como uma possível solução. Microsserviço é uma abordagem para desenvolver aplicações como um conjunto de pequenos serviços, cada qual com seu próprio processo e comunicando-se geralmente por meio de APIs (Fowler 2014). Os microsserviços, por serem pequenos programas, são mais fáceis de manter, e em caso de gargalos de memória ou CPU, mais facilmente escaláveis. Segundo Bondi (2000), escalabilidade indica a habilidade de um sistema manipular uma porção crescente de trabalho de forma uniforme, ou estar preparado para crescer, portanto a capacidade de aumento rápido de serviços.

Ao contrário do desenvolvimento de aplicações monolíticas, onde o sistema é projetado para ser implantado em um único servidor, com microsserviços pode-se criar vários pequenos programas, cada qual sendo alocado a um servidor. Essa abordagem facilita a escalabilidade, uma vez que escalar aplicações monolíticas é um desafio porque elas geralmente oferecem vários serviços, alguns mais populares que outros (Villamizar et al. 2015). Caso serviços populares necessitem ser escalados por estar sofrendo com grande demanda, todo o conjunto de serviços também será escalado ao mesmo tempo, fazendo serviços não tão populares consumirem uma grande quantidade de recursos mesmo não precisando (Villamizar et al. 2015). Levando em consideração a criação de microsserviços, ao contrário de um sistema monolítico, é possível então que somente os serviços mais populares sejam escalados. Existe então, a necessidade de melhorar o desempenho de alguns sistemas em alguns pontos da aplicação, e a idéia de criação de pequenos programas que formam uma aplicação completa.

Esse pensamento é parecido com o realizado por Sairaman (2010). Em sua pesquisa de doutorado, Sairaman desenvolveu um *framework* cujo objetivo é, a partir do código fonte de um sistema monolítico, criar sub-programas executáveis e comunicáveis entre si. Entretanto, a abordagem de Sairaman não leva em conta a vontade do programador. Seu framework age automaticamente, priorizando algum atributo, como por exemplo a mínima latência, e muitas vezes, algumas tarefas permanecem no mesmo grupo de outras, não sendo essa a idéia do desenvolvedor. Com isso, o particionamento automático mantém o programador refém da lógica adotada no framework. Uma idéia interessante é deixar a decisão de qual conjunto de tarefas virará um grupo, sendo o desenvolvedor responsável por analisar seu próprio código, e definir qual distribuição do mesmo o satisfaz.

A decisão sobre a distribuição não tem só impacto nos casos onde as aplicações já estão prontas. É comum equipes de desenvolvimento, durante as etapas de construção inicial do software, se verem entre diferentes caminhos para distribuir o software. Diferentes topologias e esquemas de distribuição se apresentam como possibilidades, e sem a capacidade de rapidamente experimentar com elas, mudando a arquitetura para ver os resultados, a solução acaba recaindo sobre decisões baseadas em experiências anteriores e/ou intuição.

Em um processo de desenvolvimento sadio, as decisões mais importantes, arquiteturais, devem ser tomadas no início do desenvolvimento (Bass, Clements e Kazman 2012). Deve-se tomar um cuidado inicial para decidir sobre:

- Qual o esquema de distribuição da aplicação/sistema: será um modelo cliente-servidor? Modelo peer-to-peer? Ou outro modelo?
- Qual(is) o(s) provedor(es) a ser(em) utilizado(s): Amazon? GAE/Azure? Servidores próprios? e
- Qual a tecnologia de persistência utilizada: banco relacional? Banco NoSQL? Mesmo se for NoSQL, existem diferentes tecnologias.

Tomando o cuidado na tomada dessas decisões, pode-se minimizar a ocorrência de mudanças tardias muito drásticas nas aplicações, o que poderia se mostrar bastante problemático. É essa linha que segue a pesquisa em arquitetura de software, cujo principal expoente é o SEI (Nuez-Reyna e Cervantes 2009) (Islam e Rokonuzzaman 2009). A ideia é facilitar o processo de tomada de decisões de projeto arquiteturais, de forma que as decisões tomadas sejam acertadas na maioria dos casos (Bass, Clements e Kazman 2012).

O problema com essa solução é que ela vai de certa forma contra a maioria das práticas ágeis, onde a evolução sem a necessidade de um grande projeto inicial é normalmente priorizada (Fowler 2001) (Bellomo, Nord e Ozkaya 2013) (Bellomo 2013). Pode-se

argumentar que, para ser bem sucedida, a evolução depende de refatorações (*refactorings*), que são mudanças relativamente fáceis de serem executadas, e são parte central das metodologias ágeis (Janus 2012). Pode-se argumentar ainda que refatorar um sistema para corrigir imperfeições de projeto em nível de código (Fowler 1999), incorporar um novo padrão de projeto (Papotti, Prado e Souza 2012), trocar o sistema gerenciador de banco de dados (PostgreSQL 2008) (Patel e Navvluru 2008) ou a camada de apresentação (Samir, Stroulia e Kamel 2007) não é uma tarefa complexa. Agora, trocar de provedor de nuvem, modificar a distribuição, ou alterar a tecnologia de persistência NoSQL não é trivial. Assim sendo, no caso das aplicações em nuvem são necessários maiores cuidados iniciais com a arquitetura, o que tornaria o processo menos ágil, mas mais seguro dado o cenário atual (Bellomo 2013).

A pergunta que faço, em contraponto a essa argumentação, é: se tais mudanças fossem mais simples de serem realizadas, então um modelo mais evolutivo, que exige que menos decisões sejam tomadas no início do desenvolvimento, seria possível? Em particular, destaco a questão da distribuição: é possível vislumbrar um cenário onde a equipe de desenvolvimento pode começar a implementar certas funções da aplicação sem ainda ter certeza sobre todos os detalhes arquiteturais referentes à distribuição? Nesse cenário, seria viável tomar decisões importantes, como migrar parte de uma aplicação de um servidor para outro, mais tardiamente no processo?

A proposta deste trabalho, apresentada de forma resumida a seguir e detalhada no Capítulo 4, foi a de investigar essa questão sob o ponto de vista tecnológico, buscando facilitar as mudanças resultantes de decisões referentes à distribuição de aplicações.

1.2 Objetivos

O objetivo deste trabalho foi viabilizar a distribuição tardia, ou seja, facilitar o processo de distribuir uma aplicação, tornando-o flexível e rápido o suficiente para permitir que a equipe de desenvolvimento possa deixar algumas dessas questões de lado durante o início do processo de desenvolvimento. Para isso, utilizou-se conceitos do desenvolvimento de software dirigido por modelos (DSDM).

O DSDM possibilita automatizar o processo de criação de software, além de facilitar sua evolução e mudança de ambientes, por meio de transformações automáticas de modelos em código (Monteiro et al. 2012). Em DSDM, o modelo é parte central do processo de desenvolvimento, sendo responsável por capturar a essência e lógica da aplicação. Todo processo tem como objetivo diminuir os esforços em codificação por parte de programadores.

Apesar de ser um tema recente, a literatura apresenta alguns resultados promissores nessa linha. Em tese, a literatura mostra que o DSDM possibilita concentrar o esforço

de desenvolvimento em um nível conceitual independente de plataforma e posteriormente, gerar código que permita a comunicação entre a aplicação, em diversas plataformas.

Ao contrário de um processo DSDM "puro", onde os modelos são o principal artefato de desenvolvimento, podendo inclusive deixar o código-fonte em segundo plano, aqui aproveitou-se o máximo possível do código-fonte. Como a ideia foi possibilitar um desenvolvimento ágil, permite-se que o desenvolvedor crie o software usando somente uma linguagem de programação, como Java ou C#. Nesse processo inicial, pouco ou nenhum cuidado com relação à distribuição precisa ser tomado. Mais adiante, modelos serão utilizados para que o desenvolvedor analise o código já criado e decida pela distribuição. Ele pode optar, por exemplo, por particionar o código em dois ou mais módulos, sendo que cada um irá rodar em um servidor diferente. Neste momento, novos trechos de código serão gerados para efetivamente implementar a distribuição da aplicação e incorporar automaticamente detalhes das plataformas de nuvem escolhidas. Algoritmos de análise de código poderão ser utilizados para guiar essa tarefa, para garantir que a funcionalidade do código original seja mantida. Mais detalhes desta proposta são apresentados no Capítulo 4.

A principal contribuição deste trabalho é facilitar a distribuição de uma aplicação em diversos computadores. Isso permite que a equipe de desenvolvimento experimente com diferentes formas de distribuir uma aplicação sem a necessidade de muito retrabalho. Em outras palavras, a tarefa de distribuir um sistema se torna "fácil", próximo ao que se tem com as refatorações atualmente. Dessa forma, é possível adotar um modelo mais ágil/evolutivo de desenvolvimento.

1.3 Metodologia de pesquisa

Para se realizar essa pesquisa de mestrado, foram seguidos os seguintes passos:

1. Definir uma linguagem de programação, tanto para o desenvolvimento do particionador, quanto para o código fonte do software a ser distribuído;
2. Analisar a melhor abordagem para distribuição do código em diversas máquinas;
3. Analisar o modo de comunicação entre processos a ser utilizado;
4. Analisar as restrições que ocorreriam a partir do modo de separação e método de comunicação;
5. Analisar a utilização de DSDM no projeto proposto;
6. Implementar uma solução na qual, a partir do código fonte de um programa e um documento, o particionador realizará seu trabalho de separar o programa em diversos sub-programas; e

7. Buscar uma solução de software *opensource*, de ampla utilização e reconhecido no meio acadêmico e de mercado de trabalho, para serem realizados os testes.

O primeiro passo foi definir a linguagem de programação a ser utilizada. Sendo uma linguagem que contém um suporte muito bom para realizar geração de código, Java foi a escolhida.

Posteriormente, deu-se início ao estudo da melhor maneira de organizar a separação. Foi definido então, que o particionamento levaria em conta as classes do projeto, ou seja, para cada partição (sub-programa que seria gerado), um conjunto de classes do projeto monolítico seria definida, sendo que nenhuma partição conteria classes de outras partições.

A Figura 1 apresenta de maneira básica a abordagem e como o projeto fica disposto após a separação do código.

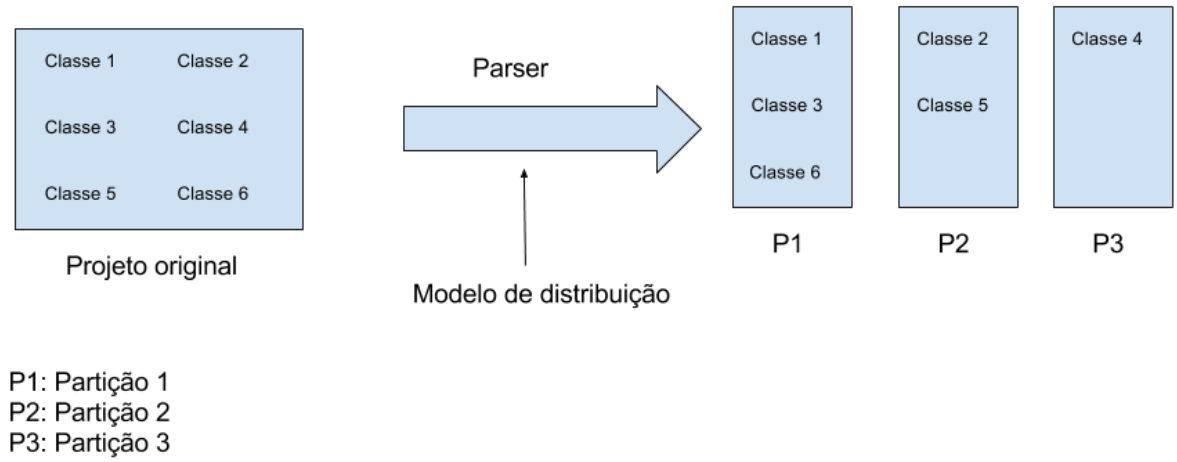


Figura 1 – Processo simplificado do particionamento

Como observado na Figura 1, após a divisão do código em sub-programas P1, P2 e P3, teremos três programas independentes, comunicando-se entre sim. A comunicação entre esses processos foi definida para utilizar Java RMI (*Remote Method Invocation*). Mais detalhes sobre RMI são abordados na Seção 2.2.

Porém, existem situações onde a separação é difícil, ou mesmo impossível, de ser automatizada. Por exemplo, não é possível separar classes pertencentes a uma hierarquia não serializável.

A separação, portanto, leva em consideração algumas restrições que foram levantadas e serão apresentadas posteriormente no Capítulo 4.

Por fim, o projeto *opensource* utilizado para os testes de particionamento foi o Apache Tomcat[1]. Além de ser um projeto bem amplo e de grande utilização, é um sistema

[1] http://tomcat.apache.org/

antigo contendo uma grande quantidade de detalhes que devem ser considerados para realizar a distribuição.

1.4 Organização

Esta dissertação está organizada da seguinte forma.

A segunda parte traz uma revisão bibliográfica sobre métodos de comunicação entre processos, microsserviços e Desenvolvimento de Software Dirigido por Modelos (DSDM).

A terceira apresenta o trabalho de Sairaman, utilizando o conceito de distribuição tardia, e alguns outros projetos correlacionados ao tema proposto nesse mestrado.

A quarta parte explica os detalhes do projeto, as restrições que foram levantadas e de maneira mais detalhada, como o distribuidor foi desenvolvido.

A quinta parte aborda os resultados obtidos nos testes de separação e pós separação, utilizando métricas de memória para comparar o desempenho da aplicação Tomcat rodando com o código original, e com o código distribuído em diversas máquinas.

Por último, tem-se a conclusão, que descreve o ganho que o projeto pode proporcionar para comunidade acadêmica e trabalhos futuros relacionados ao tema.

2 Revisão de Literatura

Este capítulo descreve os principais conceitos relacionados a esta dissertação de mestrado. Primeiramente, são definidos os conceitos de desenvolvimento de software dirigido por modelos (DSDM), especificando de maneira mais detalhada JDT. Posteriormente, é explicado o funcionamento básico dos microsserviços e a idéia por traz dessa arquitetura. Por fim, são abordados alguns modos de comunicação entre processos, focando nos conceitos de RMI.

2.1 Desenvolvimento de software dirigido por modelos

Dentro da Engenharia de Software, nos últimos anos, o DSDM (Desenvolvimento de Software Dirigido por Modelos), vem sendo uma das principais áreas de pesquisa. Nesse contexto, um modelo é uma descrição ou especificação abstrata do sistema, sendo geralmente representado como uma combinação de elementos gráficos e textuais (OMG 2003).

Portanto, o DSDM faz um melhor uso de grande parte da documentação criada durante todo o processo de desenvolvimento de software. Os modelos antes utilizados simplesmente para entendimento da aplicação, agora com DSDM fazem parte diretamente da implementação do sistema. Por meio dos modelos, é possível gerar a maior parte do código fonte que formará a aplicação, simplificando o desenvolvimento.

O primeiro desafio quando inserido dentro do mundo da orientação a modelos é entender as diferenças entre o principais acrônimos utilizados nessa área. A Figura 2 apresenta o relacionamento entre os quatro principais acronimos no universo MD*[1].

MDD (*Model-Driven Development*) é um paradigma de desenvolvimento que utiliza modelos como primeiro artefato no processo de desenvolvimento. Em MDD, geralmente, a implementação é (semi)automática, ou seja, gerada a partir dos modelos.

MDA (*Model-Driven Architecture*) é uma visão particular do MDD, proposta pelo OMG (*Object Management Group*), e assim dependentes de alguns padrões criados pelo grupo. Portanto, MDA é um subconjunto do MDD, onde a modelagem e linguagens de transformações são padronizadas pelo OMG.

Por outro lado, MDE (*Model-Driven Engineering*) é um superconjunto de MDD, pois como o "E" em MDE sugere, MDE vai além do desenvolvimento puro de atividades

[1] Em uma tradução livre do inglês, a sigla MD* significa *Model-Driven* *, ou seja, qualquer coisa dirigida por modelos

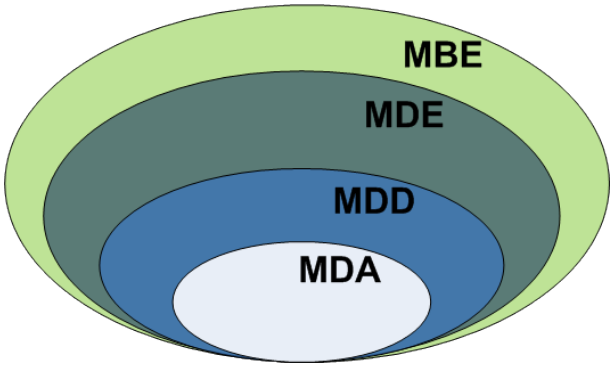


Figura 2 – Relacionamento entre os diferentes acronimos MD* (Brambilla, Cabot e Wimmer 2012)

e engloba outras tarefas baseadas em modelos de um completo processo de Engenharia de Software.

Finalmente, MBE (*Model-Based Engineering*) se refere a um processo no qual os modelos de software apresentam importantes papéis, embora eles não são necessariamente artefatos-chave do desenvolvimento.

Nesta dissertação, adota-se o acrônimo DSDM, que é a tradução brasileira do termo MDD, sendo inclusive o acrônimo escolhido pelos pesquisadores nacionais da área para o que é hoje o principal fórum dessa linha de pesquisa – o WB-DSDM (Workshop Brasileiro de Desenvolvimento de Software Dirigido por Modelos).

2.1.1 Vantagens e Desvantagens do uso de DSDM

Lucrédio (2009), em seu trabalho, fez um levantamento das principais vantagens e desvantagens no uso de DSDM para produção de software.

As principais vantagens destacadas, são:

- **Produtividade:** grande parte do tempo despendido para arquiteturação e codificação do software é utilizado para melhor modelagem e entendimento do sistema. Maior parcela do código é gerada automaticamente, sendo o foco principal de todo processo a modelagem e refinamento dos modelos de nível de abstração maior;
- **Portabilidade:** por representar uma abstração do sistema, e não o sistema em si, os modelos podem ser utilizados para gerar código para diferentes plataformas, permitindo um melhor reúso da modelagem;
- **Interoperabilidade:** partes diferentes do modelo podem ser transformadas em código para diferentes plataformas, sendo criado um sistema de intercomunicação entre as partes, a fim de desenvolver um software que executa em ambiente heterogêneo, porém mantendo sua funcionalidade global;

- **Manutenção e documentação:** o desenvolvimento comum dificulta a atualização de toda documentação, pelo fato da atividade de manutenção forçar desenvolvedores inserirem modificações direto no código. Em DSDM, o foco principal é a modelagem, assim, as alterações relativas a manutenção são realizadas diretamente nos modelos, mantendo assim uma documentação atualizada, o que facilita posteriores modificações;

- **Comunicação:** em DSDM, os diferentes profissionais possuem meios mais efetivos para comunicação, uma vez que modelos geralmente são mais abstratos que o código, não exigindo conhecimento técnico relativo à plataforma. Especialistas do domínio têm um papel mais ativo no processo, podendo utilizar diretamente os modelos para identificar mais facilmente os conceitos do negócio, enquanto especialistas em TI podem identificar os elementos técnicos;

- **Reúso:** reúso é realizado em nível de modelos, e não em nível de código, sendo os modelos adaptados ou refinados, a fim de atender novos requisitos, e posteriormente o código é gerado. Em seu trabalho, Papotti (2012) apresenta o resultado de um framework baseado em DSDM cujo objetivo é refinar um modelo anteriormente criado a fim de gerar um novo sistema, que atende melhor as tecnologias atuais;

- **Verificações e otimizações:** maior facilidade para verificações do modelo semântico e otimizações automáticas específicas do domínio podem ser executadas, reduzindo a ocorrência de erros semânticos e provendo implementações mais eficientes; e

- **Correção:** por ser uma geração automática, falhas humanas de codificação são evitadas, além da possivel identificação de erros conceituais em um nível mais alto de abstração, feita pelos geradores.

Em resumo, as vantagens destacadas acima mostram o principal foco do DSDM: evitar a realização de tarefas repetitivas de desenvolvimento, que podem ser feitas por meio da tranformação de modelos em código executável. Isso é alcançado por meio da automação dessas tranformações. O ganho de tempo, como mostra Papotti (2012) em seu trabalho, tem significado importante, pois permite que mesmo tarefas de urgência, como correção de erros, possam ser executadas sem produzir inconsistência com os modelos, mantendo-os sempre atualizados (Lucrédio 2009).

Dentre as desvantagens, podemos destacar:

- **Rigidez:** maior parte do código gerado fica fora do alcance dos desenvolvedores, dificultando modificações causando maior rigidez;

- **Complexidade:** os artefatos utilizados dentro da abordagem DSDM, como por exemplo ferramentas de modelagem e geradores de código, trazem uma maior complexidade ao processo, pois são mais difíceis de construir e manter;

- **Desempenho:** o desempenho do código pode ser diminuído, uma vez que as ferramentas geradoras de código podem adicionar código inútil ou não otimizado;

- **Curva de aprendizado:** a utilização e criação de artefatos específicos do DSDM exige um conhecimento aprofundado de construção de linguagens, ferramentas de modelagem, transformações e geradores de código. O aprendizado dessas técnicas não exige muito esforço, porém requer certo tempo para o treinamento; e

- **Alto investimento inicial:** assim como a reutilização de software, o DSDM depende de maior investimento inicial, uma vez que a construção de uma infraestrutura de reutilização orientada a modelos requer mais tempo e esforço.

2.1.2 Principais elementos do DSDM

Apesar de todos os esforços e pesquisas em relação ao DSDM, ainda assim a automatização das transformações não é uma tarefa simples. A Figura 3 mostra os principais elementos necessários para essa abordagem, e como eles são combinados.

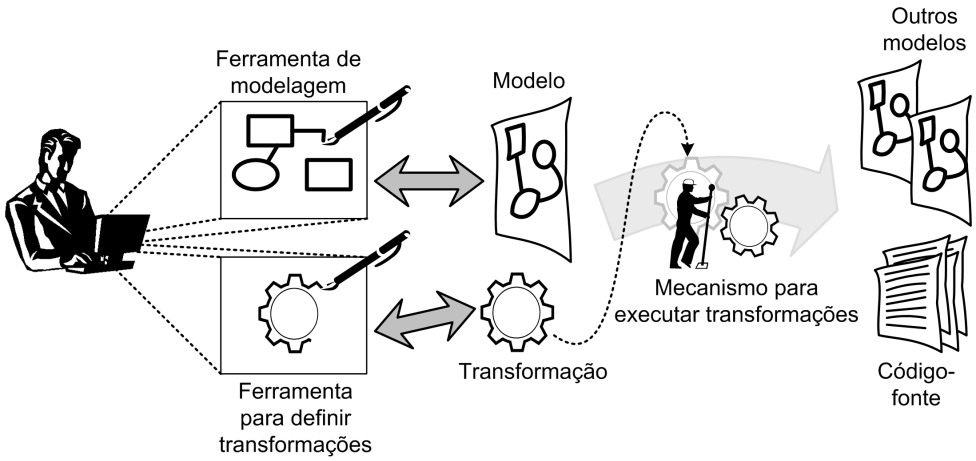


Figura 3 – Principais elementos do DSDM (Lucrédio 2009)

- **Ferramenta de modelagem:** oferecem editores gráficos para ajudar arquitetos e desenvolvedores modelarem requisitos, arquiteturas, estruturas de dados, comportamentos e outras características do sistema (Koivulahti-Ojala e Kakola 2010). Os modelos criados por essa feramenta precisam seguir regras de semântica, uma

vez que serão interpretados pelo computador. Normalmente é utilizada uma DSL[2] (*Domain-Specific Language* ou Linguagem específica de domínio);

- **Ferramenta para definir transformações:** são utilizadas para gerar outros modelos e código fonte a partir de modelos recebidos como entrada. Por meio delas são construídas as regras de mapeamento;

- **Modelos:** Os modelos servem de entrada para transformações que irão gerar outros modelos ou o código-fonte;

- **Mecanismos para executar transformação:** aplica as transformações definidas pelo engenheiro de software, mantendo informações de rastreabilidade, assim possibilitando saber a origem de cada elemento gerado, seja no modelo ou código-fonte; e

- **Outros modelos e código-fonte:** resultam do processo de transformação.

2.1.3 Eclipse JDT

Dentro do universo DSDM, de geração e manipulação de código, e utilizado nesse projeto como forma de análise de código-fonte e modelos de distribuição, existe o Eclipse JDT. O projeto Eclipse JDT[3] provê ferramentas para desenvolver aplicações Java. Esse projeto também fornece APIs para acessar e manipular código fonte Java. Ele permite acessar projetos existentes no *workspace*, criar novos projetos e modificar os mesmos. Ele também permite iniciar programas Java.

JDT possibilita acesso ao código fonte Java por duas maneiras: o *Java Model* e a *Abstract Syntax Tree* (AST).

2.1.3.1 *Java Model*

Cada projeto java é internamente representado no Eclipse como um modelo Java. O modelo Java do Eclipse é uma representação leve e tolerante a falhas do projeto Java. Não contém tantas informações quanto a AST, porém é mais rapidamente criado. Por exemplo, a visão *Outline* do Eclipse usa o *Java Model* para sua representação; dessa maneira a informação no *Outline* pode ser rapidamente atualizada.

O modelo Java é definido no plugin org.eclipse.jdt.core e é representado como uma estrutura em árvore que pode ser descrita pela Tabela 1.

[2] Uma DSL é uma linguagem projetada para ser útil para um conjunto específico de tarefas em um domínio específico (Deursen, Klint e Visser 2000)

[3] http://www.eclipse.org/jdt/

Elemento do projeto	Elemento do Java Model	Descrição
Projeto Java	IJavaProject	O projeto java contendo todos outros objetos
src folder / bin folder / ou bibliotecas externas	IPackageFragmentRoot	Pastas do projeto ou arquivos binários
Pacotes	IPackageFragment	Cada pacote está abaixo do IPackageFragmentRoot, e representam os pacotes Java
Arquivo de código Java	ICompilationUnit	Arquivo .java, que está sempre dentro de um pacote
Tipos, Campos e métodos	IType / IField / IMethod	Tipos(Classes, Interfaces, Enum), Campos e métodos

Tabela 1 – Tabela apresentando o modelo java utilizado pelo Eclipse JDT (*Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model - Tutorial*)

2.1.3.2 *Abstract Syntax Tree (AST)*

A árvore sintática abstrata (AST) é uma representação de árvore detalhada do código fonte Java. O AST define uma API para modificar, criar, ler e excluir recursos do código fonte.

O pacote principal do AST é o org.eclipse.jdt.core.dom e está localizado no plugin org.eclipse.jdt.core. Cada arquivo fonte Java é representado como uma subclasse da classe ASTNode. Cada nó específico AST fornece informação específica sobre o objeto que representa. Por exemplo, você tem MethodDeclaration (para métodos), VariableDeclarationFragment (para declaração de variáveis) e SimpleName (para qualquer String que não for uma palavra chave Java).

O AST é tipicamente criado baseado em um ICompilationUnit do *Java Model*, englobando, portanto, classes e interfaces Java.

2.1.4 Considerações sobre DSDM

O desenvolvimento dirigido por modelos eleva o nível de abstração do projeto de um sistema ao utilizar os modelos que antes serviam somente como documentação, para gerar partes (ou todo) do sistema automaticamente. Isso torna o trabalho do desenvolvedor mais simples, uma vez que a partir de uma boa arquiteturação da aplicação, ela pode ser gerada em grande parte automaticamente.

O DSDM, apesar de ser principalmente um tema de pesquisa, já encontra aplicações práticas na indústria, levando a ganhos importantes. Entre seus benefícios, destacam-se a produtividade, devido à automação, e também a portabilidade e interoperabilidade, já que modelos podem ser criados de maneira independente de plataforma. Existem algu-

mas desvantagens, principalmente devido à necessidade de maior investimento inicial, no entanto, em muitos cenários os benefícios superam os prejuízos.

O Eclipse JDT é uma ferramenta que auxilia casos de geração e manipulação de código. Isso permite que o JDT seja usado como ferramenta DSDM. A união de *Java Model* com AST pode servir como ponte de entrada para transformadores, facilitando e muito programadores que antigamente precisavam criar seus próprios *parses* e geradores de código Java, e caso bem utilizado com DSDM, formam uma poderosa mistura, trazendo grandes ganhos aos desenvolvedores.

2.2 Arquiteturas para desenvolvimento de serviços

Como citado no início desta dissertação, uma novo conceito surgiu em relação a criação de serviços, denominado microsserviços. Entretando, a criação de serviços ocorre há algum tempo. Abaixo, são listadas algumas das maneiras mais conhecidas de disponibilizar serviços.

- WSDL (*Web Service Description Language*) é uma especificação desenvolvida pela W3C[4] para descrever serviços web (Doernhoefer 2011). É um formato XML para descrever serviços de rede como um conjunto de terminais que trocam mensagens contendo informações ou orientadas a documentos ou orientadas a procedimentos. Em outras palavras, WSDL é um padrão de interface baseado em XML, usado para descrever o que um serviço web pode fazer, onde ele está localizado, e como ele pode ser chamado. Um arquivo WSDL associado com um serviço contém detalhes importantes sobre a interface do serviço para iteração com o cliente (Mehta, Lee e Shah 2006). As operações e mensagens são descritas de forma abstrata, e então envoltas em um protocolo de rede e em um formato de mensagem para definir uma camada. Camadas relacionadas são combinadas, gerando assim um serviço. WSDL é extensível para permitir a descrição de camadas e suas mensagens (W3C 2001).

- SOA (*Service Oriented Architecture*) é um estilo arquitetural que suporta orientação a serviços. É uma maneira de pensar em termos de serviços, desenvolvimento baseado em serviços e resultados desses serviços (*What is SOA?*). O uso de SOA auxilia na construção de novos aplicativos ou sistemas. O objetivo dessa arquitetura é atender os requisitos de baixo acoplamento, com base em padrões e protocolos independentes de computação distribuída (Papazoglou e Heuvel 2007). Com SOA, os recursos são "empacotados" como serviços, os quais são bem definidos, contendo módulos próprios que disponibilizam funcionalidades padrões de negócios e são independentes do estado ou contexto de outros serviços. Uma arquitetura orientada

[4] http://www.w3.org/

a serviço é projetada para permitir que desenvolvedores superem muitos desafios computacionais, incluindo integração de aplicações, gerenciamento de transações, políticas de segurança, permitindo múltiplas plataformas e protocolos a alavancar numerosos acessos a dispositivos e sistemas legados. Como exemplo de uso dos princípios SOA podemos citar o site de compras Submarino[5] e a sua relação com os serviços dos Correios[6] para efetuar cálculo do valor do frete. Para isso, o sistema faz uma chamada fornecendo alguns parâmetros como CEP de origem e destino, tipo de serviço de entrega (sedex ou comum) e peso/dimensões do produto. As informações devem ser passadas ao serviço web em formato XML, e ele enviará a resposta também em XML. Em SOA a arquitetura interna do sistema deixa explicito o uso de serviços para proporcionar mais flexibilidade e permitir que modulos específicos sejam construídos (Papazoglou e Heuvel 2007).

2.2.1 Microsserviços

Microsserviços tornaram-se um tema importante em 2014, atraindo muita atenção como uma nova maneira de pensar sobre estruturação de aplicações (Fowler 2014). Ainda não existe uma definição formal sobre o que são microsserviços, entretanto há um consenso sobre algumas características desse estilo. Como citado na Seção 1.1, microsserviço é uma abordagem para desenvolver aplicações como um conjunto de pequenos serviços, cada qual com seu próprio processo e comunicando-se geralmente por meio de APIs. Esses serviços são, segundo Savchenko, Radchenko e Taipale (2015), construídos seguindo as seguintes idéias:

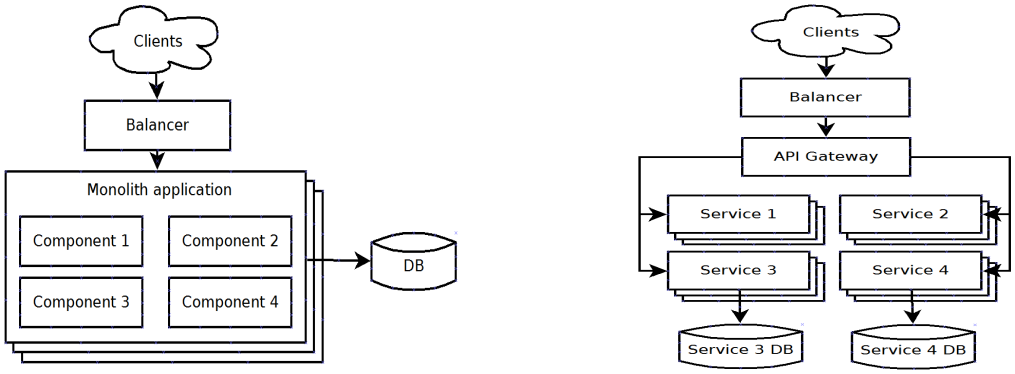
- microsserviços devem usar mecanismos simples para comunicação;
- devem ser construídos em torno de capacidades de negócios;
- microsserviços são implantados independentemente; e
- existe um mínimo de gerenciamento centralizado destes serviços, que podem ser escritos em diferentes linguagens de programação e utilizar diferentes tecnologias de armazenamento de dados.

Para explicar a idéia de microsserviços, é útil fazer uma comparação com o estilo monolítico. Uma aplicação monolítica é aquela onde todas as funcionalidades de uma aplicação estão agrupadas dentro de um único programa. Considere um sistema ERP, o qual contém os seguintes módulos: financeiro, RH, pagamento. Estando esses módulos dispostos todos dentro de um mesmo programa, essa aplicação é considerada monolítica. Agora considere a situação onde o módulo financeiro formasse sozinho uma aplicação

[5] http://www.submarino.com.br
[6] http://www.correios.com.br/

independente, o módulo RH outra e o de pagamento mais uma, e que esses três módulos conversassem entre si, utilizando APIs. Esse estilo arquitetural então é conhecido como microsserviços.

A Figura 4 mostra a diferença entre a arquitetura monolítica e de microsserviços descritas acima.



(a) Monolítico (b) Microsserviços

Figura 4 – Diferença entre arquitetura de microsserviços e monolítica (Savchenko, Radchenko e Taipale 2015)

O sistema monolítico, então, é composto em um mesmo programa, de diversos componentes, agrupando todas as funcionalidade de negócio. Em contrapartida, um sistema de microsserviços é composto por diversos pequenos programas independentes, separando cada funcionalidade de negócio, que comunicam-se de maneira a criar uma aplicação em sua totalidade.

2.2.1.1 Vantagens

Ainda que seja uma arquitetura recente, desenvolvedores já percebem alguns benefícios que a mesma traz para suas aplicações:

- código das micro aplicações é relativamente pequeno, tornando fácil o desenvolvedor entendê-lo;
- cada serviço pode ser implantado independente de outros serviços;
- os serviços podem ser escalados de forma independente uns dos outros;
- isolamento de falhas, permitindo que a aplicação não pare caso ocorra algum problema em um dos serviços; e
- diferente de sistemas monolíticos, com microsserviços é possível desenvolver cada serviço utilizando a tecnologia mais adequada para sua necessidade

2.2.1.2 Desvantagens

Por maiores que sejam as qualidades que a arquitetura de microsserviços proporciona aos desenvolvedores, existem certos problemas que os mesmos tem que lidar ao utilizar essa forma de criação de aplicações:

- complexidade adicional de desenvolvimento de sistemas distribuídos;
- complexidade operacional significativa em ambientes de produção;
- implantação de funcionalidades que abrangem vários serviços requer uma coordenação cuidadosa entre as várias equipes de desenvolvimento; e
- latência de comunicação entre os serviços pode ocorrer, uma vez que normalmente, nessa arquitetura, cada um está localizado em um servidor diferente do outro

2.3 Tecnologias de comunicação

Como já discutido no início desta dissertação, com a necessidade cada vez maior em termos programas menores, comunicando-se entre si, é necessário modos para que diferentes sistemas se comuniquem. Muitos estilos diferentes podem ser utilizados para integrar aplicações. As principais tecnologias (REST, SOAP, etc.) permitem interoperabilidade entre as aplicações (Pautasso, Zimmermann e Leymann 2008). Essas tecnologias são capazes de facilitar essa funcionalidade, conforme descrito a seguir.

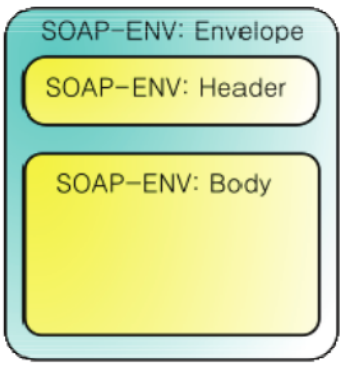


Figura 5 – Esquema do protocolo SOAP (Doernhoefer 2011)

- SOAP (*Simple Object Access Protocol*) é um protocolo elaborado para facilitar a chamada remota de funções via Internet, permitindo que dois programas se comuniquem de uma maneira tecnicamente muito semelhante à invocação de páginas Web (W3C 2007). Em um nível tecnológico, SOAP é uma linguagem XML que define uma arquitetura e formatos de mensagem, provendo um protocolo de precessamento

(Pautasso, Zimmermann e Leymann 2008). As requisições SOAP podem ser feitas em três padrões: *GET*, *POST* e SOAP. As requisições *GET* e *POST* são idênticas às requisições feitas por navegadores Internet. O SOAP é um padrão semelhante ao *POST*, mas os pedidos são feitos em XML e permitem recursos mais sofisticados como passar estruturas de dados complexas e vetores. Independente de como seja feita a requisição, as respostas são sempre em XML. O XML descreve os dados em tempo de execução e evita problemas causados por mudanças inadvertidas nas funções, já que os objetos chamados têm a possibilidade de sempre validar os argumentos das funções, tornando o protocolo robusto. Um documento SOAP define um elemento XML chamado "envelope", que contém um "header" (cabeçalho) e um "body" (corpo) como mostra a Figura 5 (Doernhoefer 2011). O "header" contém informações da infraestrutura da camada de mensagem que pode ser usado para propostas de roteamento (ex: endereçamento) e configurações de QoS[7] (ex: transações, segurança, confiança). O "body" contém a mensagem em si. SOAP é por natureza um padrão independente de plataforma. Essas características permitem uma relação de baixo acoplamento entre o requisitante e o provedor, o que é especialmente importante na Internet onde as duas partes podem residir em diferente organizações ou empresas (Papazoglou e Heuvel 2007).

- REST (*Representational State Transfer*) define um conjunto de princípios de arquitetura pelo qual pode-se criar serviços web (IBM 2008). Foi originalmente introduzido como um estilo arquitetural para construir sistemas hipermídia de larga escala (Pautasso, Zimmermann e Leymann 2008). Ele é baseado em quatro princípios: identificação de recurso por meio de URI, interface uniforme, mensagens auto-descritivas e interações *stateful* (cientes de estado) com auxílio de hiperlinks. Em identificação de recurso por meio de URI, um serviço RESTful expõe um conjunto de recursos que identificam os alvos de interação com os clientes (Pautasso, Zimmermann e Leymann 2008). Os recursos são identificados pelas URIs (Berners-Lee, Fielding e Masinter 1998), as quais disponibilizam um espaço de endereçamento global para descoberta de recursos e serviços. Com interface uniforme, recursos são manipulados usando quatro operações - *PUT*, *GET*, *POST*, *DELETE*. *PUT* cria um novo recurso, o qual pode ser excluído com *DELETE*. *GET* retorna o estado corrente de um recurso em alguma representação e POST transfere um novo estado para um recurso. Em mensagens auto-descritivas, os recursos são dissociados da sua representação, de modo que o seu conteúdo possa ser acessado em diversos formatos, como por exemplo: HTML, XML, texto simples, PDF, JPEG, etc. Por fim, interações *stateful* (cientes de estado) com auxílio de hiperlinks, toda interação com um recurso é *stateless* (sem ciência de estado), por exemplo, requisições de mensagens são independentes. Iterações *stateful* são baseadas no conceito de transferência de

[7] QoS - Abreviação de *Quality of Service* (Qualidade de Serviço)

estado explícita. Muitas técnicas existem para transferência de estados, por exemplo, reescrita de URI, *cookies*, e campos escondidos em formulários. Um pedido RESTful pode ser transmitido por qualquer número de conectores (por exemplo clientes, servidores, caches, etc.) mas não poderá ver mais nada do seu próprio pedido (conhecido com separação de capas, outra restrição do REST, que é um princípio comum com muitas outras partes da arquitetura de redes e da informação). Assim, uma aplicação pode interagir com um recurso conhecendo o identificador do recurso e a ação requerida, não necessitando conhecer se existem caches, proxys ou qualquer outra coisa entre ela e o servidor que guarda a informação. A aplicação deve compreender o formato da informação de volta (a representação), que é geralmente um documento em formato HTML ou XML, onde também pode ser uma imagem ou qualquer outro conteúdo.

Dentro desse projeto, foi utilizado RMI. O uso de RMI deu-se pela facilidade de lidar com acessos aos objetos remotos e gerenciar esses objetos. Como apresentado no capítulo 4, o projeto utiliza chamados gerenciadores, que são objetos remotos. Esses gerenciadores exportam outros objetos remotos, sendo a arquitetura do RMI mais facilmente aproveitada nesse cenário. A seção a seguir apresenta mais detalhadamente o funcionamento da arquitetura do RMI.

2.3.1 RMI

Como uma tecnologia de comunicação que engloba os conceitos listados anteriormente, pode-se citar RMI. Como WSDL/SOAP temos no RMI capacidade de descrever interfaces que abstraem operações (WSDL) permitindo a chamada remota de funções (SOAP). Além disso, caso considere SOA como um conjunto de componentes distribuídos que se comunicam a partir de um protocolo definido, pode-se dizer que RMI permite a implementação SOA.

Aplicações RMI geralmente compreendem dois programas separados, um servidor e um cliente. Um programa servidor típico cria alguns objetos remotos, tornando-os acessíveis, e esperando por clientes invocarem métodos nesses objetos. Um programa cliente típico obtém uma referência remota para um ou mais objetos remotos em um servidor e então invoca seus métodos. RMI provê o mecanismo para o qual o servidor e o cliente se comunicam e trocam informações. Tal aplicação é referida geralmente como uma aplicação de objetos distribuídos.

Uma aplicação de objetos distribuídos necessita do seguinte:

- **Localizar objetos remotos**: Aplicações podem usar vários mecanismos para obter referências para objetos remotos. Por exemplo, uma aplicação pode registrar

seus objetos remotos utilizando o registro RMI. O registro RMI armazena os objetos remotos, relacionando cada objeto a uma nomenclatura, facilitando o acesso aos mesmos. Alternativamente, uma aplicação pode passar e retornar referências à objetos remotos, como partes de outras invocações remotas;

- **Comunicação com objetos remotos**: Detalhes da comunicação entre objetos remotos são tratadas pelo RMI. Ao programador, a comunicação remota é similar a invoção de métodos no Java; e
- **Carregar definições de classes para objetos que são passados entre aplicações**: Por conta do RMI permitir que objetos transitem entre aplicações diferentes, ele disponibiliza mecanismos para carregar definições de classes assim como transmitir dados de objetos.

A ilustração a seguir retrata uma aplicação distribuída RMI que usa o registro RMI para obter a referência a um objeto remoto. O servidor chama o registro para associar um nome com um objeto remoto. O cliente procura pelo objeto remoto, por meio de seu nome, no servidor de registros, e então invoca um método nele. A ilustração também mostra que o sistema RMI usa um servidor web existente, para carregar as definições das classes, do servidor para o cliente e do cliente para o servidor, para objetos quando necessário.

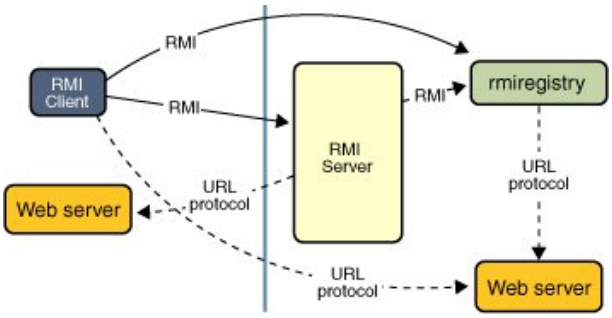


Figura 6 – Exemplo de aplicação RMI

2.3.1.1 Vantagens do carregamento de código dinâmico

Uma das características centrais e única do RMI é sua habilidade em carregar a definição de uma classe de objetos se a classe não está definida na JVM do recebedor. Todos os tipos de comportamento de um objeto, antes disponíveis somente em uma única JVM, podem ser transmitidos para outra, possivelmente remota, máquina virtual do Java. RMI passa objetos para suas classes atuais, para então o comportamento dos objetos não serem modificados quando enviados para outra JVM. Essa capacidade permite que novos tipos e comportamentos sejam introduzidos em uma JVM remota, assim estendendo dinamicamente o comportamento de uma aplicação.

2.3.1.2 Interfaces remotas, objetos e métodos

Assim como qualquer outra aplicação Java, uma aplicação distribuída construída usando Java RMI é composta de interfaces e classes. As interfaces declaram métodos. As classes implementam os métodos declarados nessas interfaces e, possivelmente, declaram métodos adicionais. Em uma aplicação distribuída, algumas implementações podem residir em alguma JVM mas não em outras. Objetos com métodos que podem ser invocados por meio de JVM são chamados objetos remotos.

Um objeto torna-se remoto implementando uma interface remota, a qual tem as seguintes características:

- Uma interface remota extende java.rmi.Remote; e
- Cada método da interface declara java.rmi.RemoteException em sua claúsula *throws*, para tratar as exceções remotas que ocorrerem.

RMI trata um objeto remoto diferentemente de um objeto não remoto quando o objeto é passado de uma JVM para outra. Ao invés de realizar uma cópia do objeto que está sendo utilizado em outra JVM, RMI passa um *stub* para um objeto remoto. O *stub* age como um representante local, ou *proxy*, para o objeto remoto, e basicamente é, para o cliente, a referência remota. O cliente invoca um método no *stub* local, o qual é responsável por executar a invocação do método no objeto remoto.

2.4 Considerações finais

Como foi visto, o DSDM proporciona uma abstração na criação de software a qual facilita a vida de desenvolvedores, podendo os mesmos se dedicarem a outras tarefas ao invés da criação de código que pode ser realizada automaticamente. Junte isso com a grande demanda por aplicações online e acessíveis de qualquer local em qualquer horário, e uma arquitetura que permite um desenvolvimento de software mais tolerante a falhas, escalável e manutenível. Com a utilização de modelos e geração automática do código a partir dos mesmos, pode-se expandir para criação de aplicações compatíveis a diversas plataformas, ou seja, um sistema é modelado e automaticamente gerado para a plataforma alvo, sem a necessidade de adaptações em linhas de código por parte do desenvolvedor. Também pode ajudar na questão da distribuição, pois geradores de código podem cuidar de muitos detalhes referentes à comunicação, tornando a tarefa mais simples de ser executada.

No próximo capítulo serão apresentadas algumas pesquisas que atuam no sentido de facilitar a distribuição de aplicações.

3 Trabalhos correlacionados

Neste capítulo são apresentados alguns trabalhos correlacionados com esta proposta. Na Seção 3.1 é apresentado o trabalho de Sairaman (2010), que emprega o conceito de "distribuição tardia". Esse trabalho é apresentado e discutido em detalhes, por se aproximar bastante dos objetivos desta proposta.

3.1 Distribuição tardia

Com o ganho cada vez mais acentuado em processamento por parte do hardware, desenvolvedores vem se preocupando em produzir software que utiliza da melhor forma possível os recursos computacionais que são disponibilizados. Como citado no Capítulo 1, é cada vez mais comum que aplicações monolíticas sejam modificadas para fazer uso de múltiplos ambientes de forma a melhor atender aos requisitos e à demanda. Em outras palavras, em muitos casos deseja-se "quebrar" uma aplicação monolítica para que a mesma execute de forma distribuída.

O grande problema dentro desse universo é o fato do desenvolvimento distribuído demandar muito tempo dos programadores, que devem levar em consideração a heterogeneidade entre os sistemas que executam o programa distribuído na rede. Essas diferenças, se não bem tratadas, podem ocasionar problemas como baixa escalabilidade e o mau balanceamento de carga.

Tendo em vista esse problema, Sairaman (2010) desenvolveu em sua pesquisa de doutorado um framework que trata desse problema, para sistemas distribuídos. A idéia básica desse framework é evitar todo o trabalho de dividir em tarefas o programa, analisar qual tarefa melhor se encaixa em qual plataforma, e realizar toda a reprogramação que transforma um programa sequencial em um programa distribuído, para plataformas heterogêneas.

A Figura 7 apresenta o processo do framework, que aceita como entrada um programa sequencial, e produz na saída o mesmo programa, porém distribuído entre diversos sistemas heterogêneos que se comunicam por meio de trocas de mensagens. Os passos que definem todo o processo do framework são os seguintes:

- **Código da aplicação:** Inicialmente, um sistema é desenvolvido de maneira monolítica.
- **Geração do grafo de tarefas:** Nessa etapa, há o levantamento das tarefas que o sistema realiza e as informações sobre as mesmas. Para cada tarefa, vários detalhes

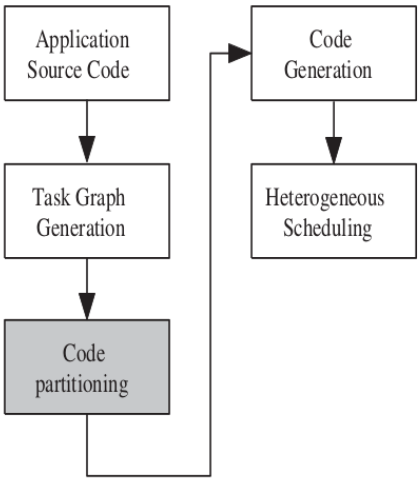


Figura 7 – Linha para mapeamento de uma aplicação em um sistema distribuído (Sairaman 2010)

como número de ciclos de CPU utilizados durante a execução da tarefa, energia gasta para realizar a tarefa, quantidade de dados necessários para executá-la, entre outras informaçõesm são coletadas. Por fim, um grafo de tarefas com as informações sobre cada é gerado. Esse grafo é construído com o auxílio de uma ferramenta de geração de grafos de tarefas (Vallerio e Jha 2003). Os nós do grafo correspondem a blocos de código da aplicação e as arestas representam dependências na execução da aplicação com os pesos correspondendo à quantidade de dados transferidos em Kilo Bytes (KB). A Figura 8 apresenta um exemplo de grafo de tarefas. Além do grafo de tarefas, o código C com anotações é gerado. A Figura 9 apresenta um exemplo do código fonte gerado nessa fase. Comentários são adicionados no código fonte original a fim de indicar quais são as tarefas e seus números.

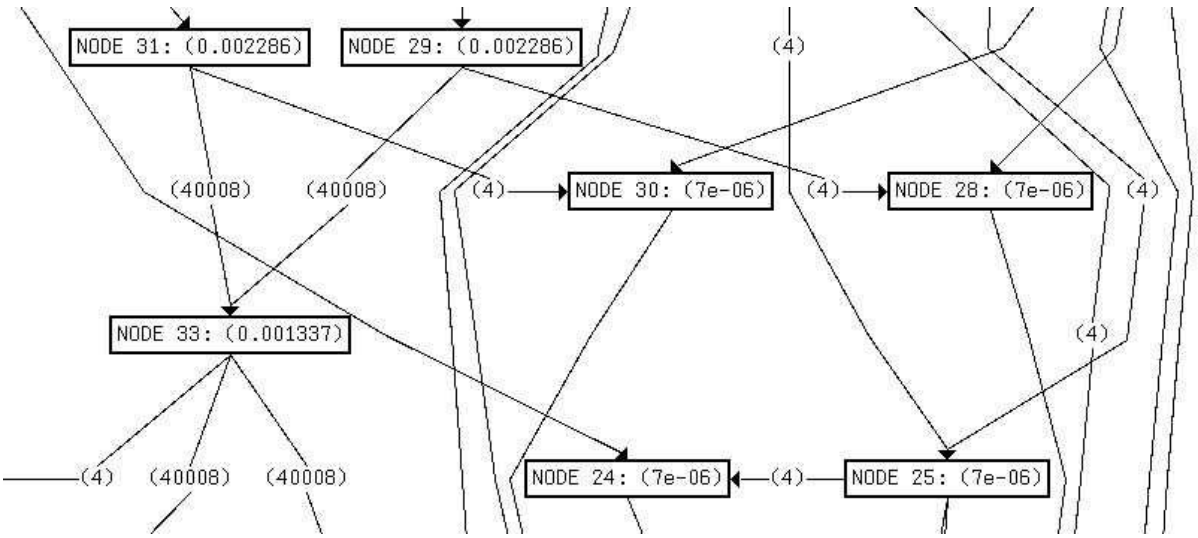


Figura 8 – Exemplo de parte de um grafo de tarefas (Sairaman 2010)

```
/*KSV main 9*/
  ave_item_diff_pre =
/*KSV main 10*/ find_diff_in_arrays (array_1, num_i
/*KSV main 11*/
  sort_array (array_1, num_in1);
/*KSV main 12*/
  sort_array (array_2, num_in2);
/*KSV main 13*/
  dual_sort (array_1, num_in1, array_2, num_in2, fu
/*KSV main 14*/
  ave_item_diff_post =
/*KSV main 15*/ find_diff_in_arrays (array_1, num_i
/*KSV main 16*/
  mean_val1 =
/*KSV main 17*/ find_mean (array_1, num_in1);
/*KSV main 18*/
  mean_val2 =
```

Figura 9 – Exemplo de código fonte anotado (Sairaman 2010)

- **Particionamento do código:** A partir do grafo de tarefas, as mesmas são combinadas em grupos, sendo cada partição mapeada em um elemento de processamento no sistema distribuído heterogêneo. Para realizar esse particionamento, são necessários uma biblioteca contendo informações específicas dos componentes das arquiteturas dos sistemas distribuídos heterogêneos, código fonte da aplicação e o grafo de tarefas extraído do código fonte. A saída desse passo é um conjunto de partições consistindo de grupos de códigos junto com informações específicas de mapeamento de cada partição, com um elemento de processamento correspondente. O algoritmo utilizado para o particionamento de código nesse passo é uma variação do algoritmo de agrupamento K-means, proposto por MacQueen em 1967. O tempo de execução desse algoritmo é muito rápido, porém não há garantias de que a solução obtida seja ótima.

- **Geração de código:** Após o agrupamento das tarefas em grupos de códigos realizado pela tarefa de particionamento, são embutidos em cada grupo primitivas de comunicação, a fim de completá-los e convertê-los então em segmentos individuais de códigos executáveis. Esses grupos são analisados individualmente para identificar e isolar as dependências de dados entre eles. Primitivas de comunicação apropriadas são incorporadas aos grupos de código a fim de permitir uma troca de dados eficiente e a execução completa do código da aplicação. A tecnologia escolhida para essa comunicação entre os sub-programas foi PVM (*Parallel Virtual Machine*), que é um pacote de software que permite que uma rede heterogênea de computadores de todos os tipos seja programada como se fosse apenas uma única "Máquina Paralela Virtual" (Geist, Kohl e Papadopoulos 1996), por conta de sua robustez e adequação para arquiteturas heterogêneas.

- **Escalonamento heterogêneo:** Por fim, esses conjuntos de códigos executáveis

são escalonados para extrair um paralelismo adicional entre eles. Esse paralelismo é alcançado devido ao modo como atua o PVM. Uma vez que a aplicação está dividida em sub-programas, uma partição pai é selecionada. As funções PVM então são inseridas na partição pai para que cada partição seja dirigida ao seus processadores apropriados. Então, para cada partição são anexadas as construções (PVM) para passagem de mensagens. Como o comando de envio de mensagens é não-bloqueante, o processo que envia a mensagem pode continuar sua execução. Isso adiciona certo paralelismo ao programa, pois permite que mesmo enquanto envia uma mensagem, prossiga o fluxo normal do programa.

Sairaman também apresenta uma análise que foi realizada em cima do algoritmo de particionamento adotado. O algoritmo de agrupamento foi testado em cinco diferentes programas, e sua função de custo foi ajustada para minimizar a sobrecarga de comunicação causada pelo agrupamento.

A Tabela 2 apresenta o número de nós (tarefas) e as arestas (relacionamentos) em cada um dos cinco programas. Para cada programa o número de parâmetros levados em consideração pelo algortimo de agrupamento foi variado de forma a estudar os efeitos na eficácia e desempenho do algoritmo.

As restrições que foram variadas para se testar o desempenho do algoritmo de agrupamento incluem: tempo de execução de cada tarefa, quantidade de dados em KB (Kilobytes) necessários a serem transferidos de uma tarefa a outra, número de ciclos de CPU necessários para tarefa completar a execução e a quantidade de energia gasta. O número padrão de parâmetros são dois. O terceiro parâmetro é o número de ciclos de CPU e o quarto o consumo de energia. Os parâmetros foram variados para cada um dos cinco programas usados. O número de grupos a serem criados pelo programa também foram alterados para cada execução. Os programas menores (menos tarefas e relacionamentos) foram testados para três até sete grupos, já os médios foram testados para cinco até nove grupos.

Programa	Quantidade de nós	Quantidade de arestas
FFT	22	26
Ferramenta estatística 1	44	97
Ferramenta estatística 2	90	163
Detecção de arestas laplaciana	105	189
Codificação aritmética	136	216

Tabela 2 – Tamanho dos programas utilizados na simulação (Sairaman 2010)

O número de parâmetros utilizados para cada execução do algoritmo de particionamento é representado na Tabela 3. Os valores para esses parâmetros foram encontrados na

fase de geração do grafo de tarefas. Os resultados são indicativos do quanto a latência em comunicação entre os grupos melhora, para o número de grupos e parâmetros utilizados.

Nro. de parâmetros	Parâmetros
i = 2	Tempo de execução + Quantidade de dados enviados
i = 3	P2 + Ciclos de CPU usados
i = 4	P3 + Energia gasta

Tabela 3 – Terminologia e quantidade de parâmetros (Sairaman 2010)

A Tabela 4 mostra o resultado da execução do algoritmo de particionamento no programa FFT, que é o mais simples em termos de número de nós e arestas (22 nós e 26 arestas, conforme apresentado anteriormente). Na tabela, são mostrados os diferentes níveis de ganho na latência de comunicação (simulada). Por exemplo, quando o algoritmo de particionamento analisa dois parâmetros (tempo de execução + quantidade de dados enviados), e produz três grupos, uma melhora de 62,8% na latência de comunicação é esperada com relação à distribuição original do programa. Isso significa que, se as 22 tarefas do programa original forem agrupadas em três grupos, haverá menos necessidade de comunicação entre as tarefas. Naturalmente, se forem gerados sete grupos, a melhora será menor, pois quanto mais grupos, maior a necessidade de comunicação, e maior a latência. Nesse exemplo, o particionamento em três grupos também terá os melhores resultados analisando-se três ou quatro parâmetros (31,8% e 36,6%, respectivamente).

Nro. de grupos	3	4	5	6	7
Nro. de Parâmetros = 2	62.8	41.7	21.2	17.0	14.2
3	31.8	30.7	23.1	16.9	16.9
4	36.6	34.9	25.4	17.1	15.6

Tabela 4 – Resultados da simulação obtidos para o programa FFT, em termos percentuais, da melhora na latência de comunicação (Sairaman 2010)

Analisando-se um programa maior, para detecção de arestas laplacianas (105 nós e 189 arestas, conforme apresentado anteriormente), observa-se que é possível particionar o programa em um número maior de grupos, de modo a obter uma melhora maior na latência. Conforme mostra a Tabela 5, analisando-se somente os dois primeiros parâmetros, um particionamento em seis grupos parece ser o melhor, com 69,9% de melhora na latência. Mas se forem levados em consideração os demais parâmetros (Ciclos de CPU e energia gasta), um particionamento em cinco grupos parece ser melhor (com 62,5% e 35,2% de melhora, respectivamente).

As simulações de particionamento permitem que o desenvolvedor experimente diversas possibilidades de agrupamento, e escolha aquela que parece ser a melhor. Com a geração de código posterior, que automaticamente produz as primitivas de comunicação

Nro. de grupos	5	6	7	8	9
Nro. de Parâmetros = 2	67.1	69.9	64.9	50.3	47.1
3	62.5	57.7	53.1	40.5	37.9
4	35.2	32.1	33.2	31.3	27.8

Tabela 5 – Resultados da simulação obtidos para o programa de detecção de arestas laplacianas, em termos percentuais, da melhora na latência de comunicação (Sairaman 2010)

de forma a implementar o particionamento escolhido, é possível rapidamente implementar e testar as diferentes configurações na prática.

Em resumo, com o framework desenvolvido por Sairaman (2010) é possível, a partir de um sistema monolítico, a geração de uma aplicação para sistemas heretogêneos distribuídos. O framework consiste de ferramentas de análise (geração do grafo de tarefas), algoritmos de particionamento de código (*k-means*) e um módulo de geração de código. A ferramenta é totalmente automatizada e não necessita de intervenção humana.

A abordagem de Sairaman efetivamente torna possível a distribuição tardia, conceito também explorado neste trabalho. No entanto, seu foco está na distribuição automática, ou seja, a decisão sobre como distribuir um sistema fica completamente a cargo dos transformadores. Há espaço para escolhas, como por exemplo com relação ao número de grupos a ser utilizado, mas apenas isso. Neste trabalho, deu-se ao desenvolvedor um papel mais ativo na decisão, por meio de modelos e ferramentas que lhe ofereçam um meio mais flexível de explorar e experimentar com o espaço de projeto. O motivo é que esse tipo de decisão, como já discutido, tem grande impacto no desenvolvimento, e nesse cenário a intervenção humana é essencial. A seção seguinte apresenta alguns projetos que tratam a questão de particionamento de código, característica muito presente na distribuição tardia.

3.2 Particionamento/Distribuição de código

A divisão de código em pequenas partes comunicantes vem, desde muito tempo, sendo de grande interesse na área da computação. Com o advento da Internet, a criação de sistemas distribuídos vem crescendo a cada dia, chegando ao ponto de explosão com a computação em nuvem, por sua facilidade de disponibilizar recursos.

Cornhill (1984) em seu trabalho, percebeu a necessidade de facilitar o desenvolvimento de sistemas, criando uma abordagem na qual software distribuído pudesse ser criado como se o sistema de destino fosse um processador único. O objetivo do projeto foi desenvolver os métodos, ferramentas e suporte de tempo de execução para criar software que consiste em um programa na linguagem Ada em execução em um sistema distribuído. O projeto foi realizado pensando num desenvolvimento em que existisse uma definição

de notação para expressar o particionamento da aplicação entre sistemas distribuídos, um compilador que identificasse as partições e gerasse o programa e um sistema Ada distribuído em tempo de execução. Com o mesmo intuito da abordagem deste mestrado, a proposta de Cornhill baseia-se na distribuição de código. Entretando, sua abordagem atua já no desenvolvimento do sistema, a partir da utilização das notações definidas, compilando o código e gerando os serviços, enquanto o projeto desenvolvido neste mestrado atua após o código do sistema alvo estar pronto.

Vahid e Gajski (1995) apresentam um conjunto de métricas para ajudar no particionamento de funcionalidades de software. Uma tarefa importante no projeto de sistemas é dividir as funcionalidades para implementação em diversos componentes, incluindo componentes de hardware e software (Vahid e Gajski 1995). As métricas apresentadas podem ser usadas por desenvolvedores, ou até mesmo algoritmos automatizados, para manter juntos objetos que deveriam pertencer a um mesmo componente. Assim como nesta pesquisa de mestrado existem algumas restrições, a abordagem de Vahid e Gajski considera métricas, que "forçam" o desenvolvedor a manter junto certos objetos. A restrições encontradas nesta pesquisa de mestrado serão apresentadas posteriormente, entretanto pode-se observar que a necessidade de definir os limites para divisão de código já são estudadas há muito tempo.

Sistemas de objetos distribuídos trazem muitas vantagens à computação (Hunt e Scott 1998). Contudo o processo de distribuição é muito custoso. Programadores muitas vezes necessitam manualmente dividir as aplicações em sub-programas, sendo que as técnicas utilizadas geralmente são *ad hoc*. Com vistas nisso, Hunt e Scott (1998) apresentaram um sistema de particionamento distribuído automático conhecido como Coign (Hunt e Scott 1999). Coign analisa uma aplicação, escolhe sua distribuição e produz a distribuição desejada totalmente sem acesso ao código do aplicativo, pois ele trabalha em cima de binários. O pensamento de Hunt e Scott vai de encontro com o deste mestrado, e até mesmo do projeto de Sairaman. Contudo, enquanto na abordagem de Sairaman e deste mestrado é utilizado o código fonte para a distribuição, Hunt e Scott atuam diretamente no arquivo binário do software desenvolvido, e não tem acesso ao código fonte. Além disso, fica a cargo do Coign a escolha da forma de distribuição, enquanto neste projeto, a distribuição fica a cargo do desenvolvedor.

Por fim, sendo a computação em nuvem um assunto atual e do futuro para soluções empresariais, muitos sistemas legados necessitam de adaptação para encaixar com os requisitos da computação em nuvem ou serem reescritos desde o princípio (Zhou, Yang e Hugill 2010). Contudo, software empresarial pode ser complexo. Zhou, Yang e Hugill propuseram uma abordagem para reengenharia de sistemas de software de empresas para computação em nuvem, criando uma ontologia para esses sistemas e então particionando essa ontologia, decompondo o software legado em serviços. São três passos realizados na

abordagem: criar ontologias para o código fonte, dados e aplicação, integrar as ontologias encontradas e implantar a ontologia encontrada. Por fim, analisando os conceitos e relações na ontologia, o software empresarial será entendido e decomposto como diferentes serviços. Assim como neste mestrado, a idéia central é criar diversos serviços a partir do código fonte. Zhou, Yang e Hugill realizam essa tarefa tentando encontrar serviços que se encaixem no ambiente de computação em nuvem, especificamente, tentando entender e detectar componentes fracamente acoplados para torná-los serviços. Em contrapartida, a abordagem deste mestrado é mais ampla, permitindo o desenvolvedor realizar suas escolhas a partir de suas preferências, e facilmente gerar a comunicação entre os serviços que desejar criar.

3.3 Considerações finais

A literatura mostra evidências da necessidade da distribuição de código e software, por diversos motivos: evolução das tecnologias, possíveis gargalos encontrados, motivações arquiteturais, como criação de sistemas distribuídos. No geral, as abordagens não utilizam uma geração automática de código. Em alguns casos, a atuação é antes do desenvolvimento do software, em outros, são formas de analisar um software já pronto, e o próprio desenvolvedor, manualmente, realizar as alterações, seguindo certos passos.

Por outro lado, existe a pesquisa em distribuição tardia, como o trabalho de Sairaman (Seção 3.1), que dá margem a um processo mais automatizado. Entretando, esse fato pode levar a decisões não ótimas em alguns cenários, e nesse sentido também acarreta em certa rigidez.

A proposta deste mestrado foi automatizar a distribuição tardia por meio do DSDM, de forma flexível, dando ao desenvolvedor papel ativo nas decisões, e possibilitando um modelo ágil de desenvolvimento. Sob a ótica da pesquisa, pode-se dizer que não foi desenvolvida nenhuma técnica ou ferramenta completamente nova, mas sob uma ótica de inovação, pretende-se oferecer contribuições importantes em termos da combinação das técnicas bem-sucedidas já existentes. O próximo capítulo apresenta a abordagem em detalhes.

4 Uma abordagem dirigida por modelos para distribuição tardia de aplicações

Como abordado no Capítulo 1, a crescente necessidade de atualização de sistemas traz consequências para os programadores. Sistemas legados, e até novos sistemas, tendem a serem divididos em pequenos software para facilitar a manutenção, escalabilidade e o desenvolvimento. Com esse pensamento em mente, foi desenvolvida uma abordagem que facilita a distribuição de software, em pequenos serviços, a partir do código fonte do mesmo e alguns modelos de distribuição. A Figura 10 ilustra a abordagem desenvolvida.

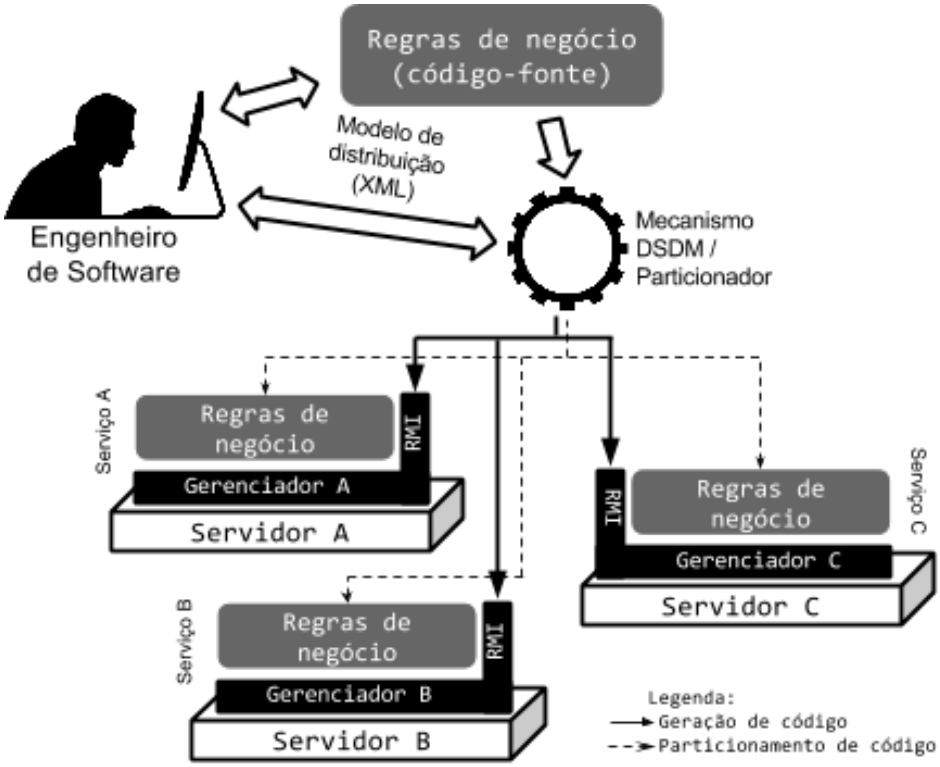


Figura 10 – Abordagem para distribuição tardia de aplicações

Como apresentado na Figura 10, o Engenheiro de Software tem dois papeis iniciais principais que são: codificação do sistema e modelagem da distribuição. A codificação consiste no processo tradicional de desenvolvimento, baseado em código-fonte, e não precisa levar em consideração aspectos de distribuição. A modelagem da distribuição consiste em uma especificação formal de como aquele código-fonte deve ser distribuído entre múltiplos servidores.

A partir desses dois artefatos, um mecanismo DSDM é responsável por realizar duas tarefas:

- Distribuir o código segundo a especificação fornecida pelo Engenheiro de Software (Linhas tracejadas na Figura 10). Isto envolve analisar o código em busca de dependências para que a funcionalidade seja mantida; e

- Gerar código para implantação de cada pedaço distribuído de software em sua respectiva plataforma e para a sua comunicação (Linhas contínuas na Figura 10). Envolve a geração de código específico de cada serviço, e também código de comunicação utilizando RMI. Também inclui a criação de um gerenciador para cada servidor, responsável pelo ciclo de vida dos objetos disponibilizados remotamente por cada aplicação, permitindo que outras aplicações acessem esses gerenciadores, e requisitem a criação de um novo objeto remoto para sua utilização.

A princípio, essas tarefas podem ser realizadas de maneira automática. No entanto, devido algumas restrições de implementação, intervenção humana é exigida em alguns momentos, como explicado mais adiante.

Como o código-fonte em si é desenvolvido de maneira monolítica, facilita-se a aplicação de metologias de desenvolvimento ágeis, já que o aspecto da distribuição não precisa ser completamente definido desde o início. Esta é uma primeira contribuição deste trabalho.

Com a utilização do mecanismo DSDM, a tarefa de distribuir uma aplicação poderá ser executada rapidamente. Ainda que o código final precise ser ajustado ou complementado, como é comum no DSDM, isso permite que o Engenheiro de Software possa experimentar com diversas opções de distribuição. Ao contrário da distribuição usual, que pode levar meses para ocorrer em sistemas de médio a grande porte, com a abordagem desenvolvida o trabalho de separação pode ser realizado de alguns dias até algumas poucas horas. Ele pode, por exemplo, gerar três ou quatro versões de distribuição diferentes, testá-las exaustivamente, e decidir pela melhor com maior confiança e sem depender da intuição, em curto espaço de tempo. Esta é uma segunda contribuição deste trabalho.

4.1 Modelo de distribuição

O modelo de distribuição adotado é baseado na separação de classes. Quando existem gargalos em sistemas, em geral estão relacionados ou à criação de certos objetos, ou à execução de certas tarefas específicas a certas classes. Então a divisão de código se baseia em escolher quais classes serão mantidas em quais servidores.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <clouds>
3   <cloud name="Servico1" url="200.124.244.22">
4     <class>JarFactory</class>
```

```
5       <class>Library</class>
6       <class>Constants</class>
7     </cloud>
8     <cloud name="Servico2" url="220.222.350.456">
9       <class>MethodInfo</class>
10      <class>Authenticator</class>
11      <class>Config</class>
12    </cloud>
13  </clouds>
```

Listing 4.1 – Exemplo de modelo de distribuição

O XML acima mostra um exemplo de utilização do modelo de distribuição. Ele contém, então, a separação das classes JarFactory, Library, Constants, MethodInfo, Authenticator e Config em dois servidores. O primeiro servidor estará localizado no endereço 200.124.244.22, com o nome do serviço gerado sendo Servico1, contendo as classes JarFactory, Library e Constants. O Servico2 por sua vez, estará localizado no servidor com endereço 220.222.350.456 e as classes que farão parte de sua aplicação são MethodInfo, Authenticator e Config.

Como apresentado, para ser possível separar as classes, um modelo XML foi adotado. Esse modelo consiste das seguintes *tags*:

- **clouds**: *tag* que engloba o conjunto todo do XML. Dentro dessa existem as *tags* *cloud*;
- **cloud**: indica a criação de um dos serviços, dentro dela estão dispostas as classes que compõe o serviço. Deve-se colocar os atributos *url* (endereço que o serviços estará localizado e acessível - pode ser IP) e *name* (nome do novo serviço - nomes de serviços não podem ser iguais); e
- **class**: nome da classe. Não pode existir nomes de classes repetidos. As classes devem estar contidas dentro do projeto a ser distribuído. Isso facilita ao JDT trabalhar com as classes.

Outro documento utilizado para o modelo de distribuição tem relação com a vontade ou não do desenvolvedor de exportar para outros serviços algumas classes. O engenheiro de software pode desejar que algumas classes não sejam expostas como objetos de serviços, ou seja, como foi utilizado RMI para comunicar entre os programas, então classes que estiverem nesse documento não serão consideradas remotas. Isso implica que todas as classes relacionadas a essas, ou seja, que as utilizam como parâmetros ou retorno de funções, variáveis globais da classe, ou como uma classe herdada, não serão remotas também.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <class>JarFactory</class>
3 <class>MethodInfo</class>
```

Listing 4.2 – Exemplo de modelo de definição de classes não remotas

O exemplo acima mostra o formato do documento que discrimina que as classes JarFactory e MethodInfo não serão remotas. O formato desse documento é como o exemplo apresenta. Um conjunto de *tags class*, indicando o nome das classes que não deve ser remotas.

Como explicado em 2.2, classes remotas são aquelas que implementam alguma interface que extende a interface java.rmi.Remote. Além disso, para um objeto de uma classe poder transitar de um servidor a outro, a classe deve implementar a interface java.io.Serializable. Essa interface permite que objetos sejam transformados em bytes e vice-versa. Então, caso o programador, antes de realizar o particionamento de código optar por uma classe não ser remota, ele estará optando por não permitir que a mesma transite entre JVMs. Caso exista um objeto remoto, que utiliza como parâmetro um objeto não serializável, acione a função e passe esse objeto não serializável como argumento, ocorrerá um problema de serialização, pelo objeto não permitir essa opção, por isso a necessidade de todos os objetos relacionados a um objeto não remoto serem proibidos de serem remotos. Na Seção 4.3 são apresentados exemplos de quando objetos não são remotos.

Por fim, com esses dois documentos, o de separação de serviços e o de classes não remotas, e mais o código fonte, o particionador está apto a começar o seu trabalho automático de divisão de código.

4.2 Mecanismo DSDM / Particionador

O particionador é o responsável por analisar o código fonte, utilizando as bibliotecas JDT, e em conjunto com os modelos de distribuição, realizar a quebra de código nos serviços designados pelo XML, criando toda estrutura de comunicação entre os mesmos. São seguidos os seguintes passos no processo de particionamento:

1. Carregamento do XML contendo a separação dos serviços e classes, análise se todas as classes do projeto estão definidas no mesmo XML e carregamento do XML contendo quais classes não são remotas;

2. Análise de quais classes do projeto podem ser remotas, utilizando tanto as definidas no XML quanto outros parâmetros que podem indicar se uma classe é serializável ou não. Esses outros parâmetros refletem algumas limitações da implementação atual

do particionador, e são discutidos em detalhes mais à frente. Tem-se ao final uma lista de classes remotas e um de não remotas;

3. Criação dos gerenciadores de classes para cada serviço e da classe responsável por iniciar o processo em cada servidor;

4. Para todas as classes, cria-se os conectores com qualquer outra classe de outros serviços. Essa conexão e acesso é feita também por meio dos gerenciadores;

5. Toda classe remota terá uma interface específica, onde estarão declaradas as funções que são acessíveis remotamente. Para isso, cada função da classe é analisada, gerando uma declaração da mesma na interface relativa à classe. Durante a análise dos métodos, sempre que houver um construtor declarado, há a criação de uma função no gerenciador responsável do servidor, que é responsável por retornar a criação do objeto da classe utilizando o construtor. Caso a função seja uma função estática, uma entrada no gerenciador também é criada, para permitir acesso ao recurso, pois como o acesso a objetos remotos é feito por meio de interface, e não há possibilidade de declarar funções estáticas nas mesmas, fica a cargo do gerenciador realizar essa chamada;

6. Finalizada essa análise, inicia-se a análise que localiza instanciações de objetos e chamadas estáticas de funções. Para criação de objetos, a análise identifica sua criação, verifica se é um objeto que pode ser remoto, identifica a qual serviço o objeto pertence, e altera a criação por uma chamada da função do gerenciador responsável por criar o objeto remotamente. Para as funções estáticas o processo é parecido. A análise identifica chamadas estáticas, verifica se a chamada é de um objeto que pode ser remoto, e em caso positivo, substitui pela chamada da função do gerenciador que é responsável por essa chamada estática no outro serviço;

7. O passo seguinte corresponde a substituir referências a classes remotas por sua interface respectiva. Isso ocorre pelo fato de que no RMI, quando um objeto é exportado, o mesmo só pode ser referenciado por meio da interface relativa ao mesmo, por isso a necessidade de substituir referências de classes por suas interfaces remotas; e

8. Por fim, com todas as chamadas já transformadas em chamadas remotas, para criações de objetos remotos e acesso a funções estáticas criadas, classes remotas separadas, gerenciadores completos, cria-se um projeto para cada serviço, levando em consideração quais classes pertencem a cada serviço.

Observa-se o gerenciador sendo parte central da nova arquitetura. Os gerenciadores são objetos remotos, que carregam todas as informações relativas as classes que podem ser remotas, e suas funções estáticas. Cada classe que inicialmente criava um objeto que

agora é remoto, tem uma referência para cada gerenciador que necessita, podendo assim, remotamente, realizar pedidos para que esse gerenciador crie objetos e os exporte para utilização na classe necessitada.

Vejamos a seguir um exemplo de como as classes A e B ficam após a distribuição. Para esse exemplo, foi desejado que a classe A ficasse em um serviço e a classe B em outro. Neste exemplo, os métodos de A são chamados por B, e nenhum método de B é chamado, por isso a transformação de B em uma classe remota não será ilustrada.

```
1  public class A{
2    public A(){
3      // corpo do construtor
4    }
5
6    public static void funcaoEstatica(){
7      // corpo funcao estatica
8    }
9
10   public void funcao(){
11     // corpo da funcao
12   }
13 }
```

```
1  public class B{
2    public static void main(String [] args){
3      A a = new A();
4      a.funcao();
5      A.funcaoEstatica();
6    }
7  }
```

Verificamos acima que a classe B, em seu método *main()*, cria um objeto da classe A, e utiliza tanto uma chamada de função desse objeto, além de realizar uma chamada de função estática da mesma classe A. Deseja-se então, separar as classes, tornando a classe A uma classe remota. Após a geração de código do particionamento, as seguintes classes e interfaces passam a existir: Gerenciador, GerenciadorInterface, ARemoteInterface.

```
1  public interface GerenciadorInterface
2    extends Remote{
3
4    public ARemoteInterface getA() throws RemoteException;
5    public void AfuncaoEstatica() throws RemoteException;
6  }
```

A interface GerenciadorInterface é portanto uma interface remota que será implementada pela classe Gerenciador. Um objeto da classe Gerenciador será um objeto remoto, ficando exposto na JVM relativa ao serviço o que compõe, permitindo que outros serviços o acessem, criando requisições de novos objetos das outras classes do projeto. Essa nova interface tem a escrita dos métodos criados getA (exporta um novo objeto remoto da classe A) e AfuncaoEstatica (faz uma chamada remota para a função estática funcaoEstatica da classe A).

```
1  public class Gerenciador
2    extends UnicastRemoteObject
3    implements GerenciadorInterface{
4
5    public ARemoteInterface getA() throws RemoteException{
6      return (ARemoteInterface)
7        UnicastRemoteObject.
8        exportObject(new A(), 1099);
9    }
10
11   public void AfuncaoEstatica() throws RemoteException{
12     A.funcaoEstatica();
13   }
14 }
```

A classe Gerenciador, além de realizar a interface GerenciadorInterface, também estende a classe UnicastRemoteObject, uma classe específica do RMI que permite que esse objeto seja remoto. Observe o corpo das funções: a primeira (getA) cria um objeto A e depois o exporta, deixando-o acessível na porta 1099, tornando-o remoto para quem solicitou ao gerenciador (no caso do exemplo, para a variável de nome “a”, na classe B), a segunda (AfuncaoEstatica) realiza uma chamada da função funcaoEstatica, estaticamente.

```
1  public interface ARemoteInterface
2    extends Remote, Serializable{
3    public void funcao() throws RemoteException;
4  }
```

A interface ARemoteInterface faz-se necessária, pois assim objetos remotos da classe A poderão realizar chamadas remotas para suas funções.

```
1  public class B{
2    GerenciadorInterface gerenciador;
3
4    public static void main(String [] args){
5      ARemoteInterface a = gerenciador.getA();
```

```
6       a.funcao();
7       gerenciador.AfuncaoEstatica();
8     }
9
10    /* funcao que recupera gerenciador */
11    public void startManagers() ...
12  }
```

Comparemos agora a primeira versão da classe B com sua nova versão. Na versão antiga, a criação do objeto A era feita por meio da utilização do *new*. Agora, por ser uma classe remota, a criação é realizada pelo gerenciador de objetos, e então exportada para utilização em B. O mesmo ocorre com a chamada estática, sendo ela substituída pela chamada do gerenciador.

4.3 Restrições para particionamento

Durante o desenvolvimento desse projeto, algumas restrições para a distribuição de código foram definidas, e outras encontradas. Isso ocorreu para facilitar a implementação do particionador, uma vez que, caso fossemos abranger todos os detalhes, o projeto ficaria muito extenso e não caberia dentro do período do mestrado. Lembrando também, por ser um projeto sem nenhuma base de comparação, tudo teve que ser descoberto ao longo do mestrado, então muitos dos detalhes presentes somente foram encontrados no momento em que o particionador começou a ser testado mais amplamente. Mas em tese essas restrições podem ser superadas com mais tempo de desenvolvimento.

Segue, abaixo, uma lista de restrições utilizadas no projeto.

1. Não devem existir classes com mesmo nome, e cada classe representa um único arquivo, portanto, cada arquivo pode conter somente uma classe nele. Isso facilita a atuação do JDT na hora de trabalhar com as classes;

2. Classes que estendem alguma classe que não pertence ao projeto automaticamente são consideradas não remotas. Isso ocorre pelo fato de, com a herança, é permitido que a instanciação da classe que estende uma classe não pertencente ao projeto seja feita a uma variável com o tipo da classe que está fora do projeto. Existe também a possibilidade nesse caso, de a classe pai não ser uma classe serializável, implicando que, mesmo a classe implementando a interface Serializable, não poder ser serializada, pois a classe pai não tem esse poder. Vale lembrar que esse processo é recursivo, ou seja, caso uma classe estenda outra que é do projeto, chamemos de pai A, e a pai A estenda uma classe que não pertence ao projeto, então a hierarquia toda é impossibilitada de ser remota;

3. Classes que implementam alguma interface que não pertence ao projeto podem tornar-se não remotas. Caso a interface em questão não estenda em sua hierarquia de interfaces, em momento algum, a interface Serializable, então as classes que implementam essa interface são consideradas não remotas. A mesma idéia dá-se caso alguma outra classe estenda a primeira em questão, ou caso alguma interface extenda a interface em questão; e

4. Como citado acima, existem classes e interfaces que não são serializaveis e estão fora do escopo do projeto. Quando ocorre sua utilização sendo tipo de alguma variável de classe, retorno de função ou parâmetro de função, essas classes são consideradas não remotas. Isso ocorre pois as JVMs, quando vão serializar e desserializar esses objetos que transitam remotamente, não conseguem, pois a interface que permite isso não está presente. Então, um objeto que contém uma variável instanciada com um objeto não serializável causará uma quebra no processo de transformar bytes em objeto. O mesmo ocorre quando é realizada uma chamada remota para alguma função. Caso essa função tente retornar um objeto não serializavel, as JVMs não consegue realizar a tradução do mesmo em bytes, quebrando assim o processo de exportação de objetos.

Para ilustrar as restrições 2 e 3, os seguintes exemplos são apresentados:

```
1 public class ABC extends Permission{
2     ...
3 }
4
5 public class XYZ {
6     public static void main(String [] args){
7         Permission a = new ABC()
8     }
9 }
```

A primeira das restrições encontradas na restrição 2 é apresentada acima. A classe ABC estende a classe Permission, que não pertence ao projeto. Na classe XYZ, existe a criação de um objeto ABC utilizando como tipo da variável “a” a classe Permission. Entretando, caso ABC pudesse ser remota, o retorno para variável “a” seria um *stub*, pois sua criação seria realizada por um gerenciador, e um objeto remoto seria exportado, sendo que a classe Permission não dá suporte aos *stubs* criados para classe ABC.

```
1 public class A extends ClasseNaoSerializavel{
2     ...
3 }
4
```

```
5  public class B extends A{
6      ...
7  }
8
9  public class C extends B{
10     ...
11 }
```

No exemplo acima, temos a classe A que estende ClasseNaoSerializavel. Considere que a classe ClasseNaoSerializavel não é serializavel e nem pertence ao projeto a ser distribuído. Então, como explicado na restrição 2, a classe A não pode ser remota, pois ClasseNaoSerializabel também não tem essa possibilidade. Em consequência, a classe B, por estender a classe A, não será remota, assim como a classe C que estende a classe B.

A seguir, temos um exemplo que ilustra a situação encontrada na restrição 3:

```
1  public interface Qualquer extends Comparable<Object>{
2      ...
3  }
4
5  public class A implements Qualquer{
6      ...
7  }
8
9  public class B extends A{
10     ...
11 }
```

O código acima demonstra a restrição 3. A interface Qualquer estende de uma interface base do Java denominada Comparable. Essa interface Comparable não é serializável, tornando todas as interfaces que a estendem não serializáveis. Uma vez que Qualquer não é serializável, então qualquer classe que a implementa não poderá ser remota. Como no caso anterior, B estende da classe A, então B não pode ser remoto também.

4.4 Considerações finais

Como apresentado durante este capítulo, a abordagem utiliza ideias de DSDM para facilitar a distribuição de código em diversos servidores. Esse processo é realizado semi-automaticamente, pois o desenvolvedor, ao final da distribuição de código, ainda terá que realizar alguns ajustes que não foram cobertos pelo particionador. Em geral, esses ajustes são pequenos e podem ser contemplados pelo particionador em uma evolução do mesmo.

Além disso, foram citadas as restrições encontradas durante todo o projeto. Elas implicaram diretamente no desenvolvimento do projeto, e não foram triviais de se encontrarem. Alguns casos ainda podem ser tratados, porém são casos muito extremos e que necessitariam de um tempo muito maior de desenvolvimento. Um pesquisador, tomando como base os estudos realizados neste mestrado, poderá ampliar a capacidade do particionador mais facilmente.

A distribuição de código entre diversos servidores abre caminho para outros pensamentos. Mesmo que a abordagem deste mestrado não tenha criado serviços pensando na arquitetura de microsserviços, o caminho para realizar isso segue os mesmos princípios. Como explicado na Seção 2.2.1, microsserviços exigem um mínimo de gerenciamento centralizado, ou seja, deve existir um núcleo da aplicação, com a qual os outros microsserviços comunicam-se. Além disso, microsserviços utilizam APIs para troca de informações, e devem ser construídos em torno de capacidades de negócio. O projeto deste mestrado não explicita essa criação, entretanto, em questão de construção de cada serviço em torno de capacidades de negócio, é possível, caso o desenvolvedor saiba os limites de cada serviço, e escolha uma combinação de classes que permita a distribuição levando em consideração as capacidades de negócio. O gerenciamento centralizado pode ser realizado da mesma maneira, distribuindo de modo que exista um serviço central, com o qual todos os outros serviços se comunicam. A comunicação, realizada utilizando RMI, pode ser substituída por chamadas de API, assim como prega a arquitetura de microsserviços, sendo outra barreira transponível. Portanto, pode-se observar que conseguir uma distribuição cujo objetivo é a criação de microsserviços não é algo tão distante. Seguir alguns passos na escolha de distribuição, e modificar o modo de comunicação, a grosso modo, permite o desenvolvimento de um projeto voltado para criação de microsserviços, assunto tão importante atualmente.

Por fim, para testar o funcionamento da abordagem desenvolvida e se a mesma atendia a expectativa inicial, a de quebrar o código de um projeto de pequenas partes comunicantes, foram realizados alguns experimentos. Os mesmos estão detalhadamente explicados no próximo capítulo.

5 Avaliação

Após todo o processo de desenvolvimento do particionador, alguns testes foram planejados a fim de verificar a possibilidade de diversas formas de divisão e a facilidade de realizá-la.

Para realizar os experimentos, o programa a ser dividido foi o Apache Tomcat. A escolha deve-se a esse software ser reconhecido como um software importante dentro da computação, não havendo assim questionamentos sobre uma possível facilitação de um sistema implementado pelo próprio pesquisador dessa pesquisa.

Conforme já mencionado ao longo desta dissertação, a abordagem possibilita que o desenvolvedor consiga experimentar com diferentes configurações de distribuição, sempre com o objetivo de melhorar algum aspecto do sistema, como desempenho, segurança ou uso da memória. No caso do Tomcat, testes iniciais mostraram que o desempenho quando a execução é distribuída é significativamente pior do que quando a execução acontece em uma única máquina. Isso porque existe a questão da latência inerente à comunicação entre as partes do sistema. Para serem observados ganhos reais de desempenho, seria necessário, além da distribuição, também a replicação, balanceamento de carga, entre outras técnicas de sistemas distribuídos. Como a abordagem atual não prevê a replicação, foi analisado o aspecto de uso de memória pelo Tomcat.

Em uma única máquina, o Tomcat consome uma certa quantidade de memória. Pretende-se experimentar com outras configurações que possivelmente diminuam a quantidade de memória individual em cada máquina. Assim, por exemplo, ao invés de ser necessária uma única máquina com 2 Gigabytes de memória previstas para o Tomcat, poderiam ser usadas quatro máquinas com 500 Megabytes cada uma, ou outra configuração diferente.

Ressalta-se que este é apenas um exemplo, cujo objetivo é demonstrar a viabilidade da abordagem, portanto não são esperados ganhos reais com o Tomcat, mesmo porque este é um sistema já consagrado e otimizado ao longo dos anos por seus desenvolvedores.

Os experimentos foram guiados utilizando o JMeter[1], uma ferramenta gráfica de testes que permite ao programador criar "cenários" de teste, inserindo quais URLs acessar, passando os parâmetros ou mesmo um *form* de dados. A ferramenta utilizada para medida de memória é uma ferramenta paga chamada Yourkit[2], contudo, ela permite um período de testes de 15 dias, sendo suficiente esse tempo para análises.

O passos seguidos para os testes foram os seguintes:

[1] http://jmeter.apache.org/
[2] https://www.yourkit.com/

1. Escolha da quantidade de serviços a serem criados, suas respectivas localizações, e separação das classes entre esses serviços;

2. Distribuição do código, gerando os serviços;

3. Implantação dos serviços nos servidores adequados e inicialização dos processos dos serviços;

4. Inicialização da leitura de memória relativa ao serviço em cada computador; e

5. Realização dos testes automatizados e coleta de dados de memória.

Os testes automatizados foram 3 casos principais: executar ações disponíveis no Tomcat (CT1 e CT2), deploy de uma aplicação no servidor Tomcat (CT3), executar ações disponíveis no sistema que foi realizado o deploy (CT4 e CT5).

O caso de teste 1 (CT1) é referente a navegação no Tomcat. Algumas URLs do software gerenciador do Tomcat foram acessadas, além de realizar ações de login. Para esse teste, foram realizados 30 requisições, sendo 3 vezes de 10 requisições simultâneas.

O caso de teste 2 (CT2) executa as mesmas ações que CT1, entretanto, ao invés de 30 requisições com 3 sequências de 10 requisições simultâneas, foram executadas 300 requisições, com 3 sequências de 100 requisições simultâneas.

O caso de teste 3 (CT3) representa o *deploy* de um sistema no Apache Tomcat. Esse software é um sistema e-commerce chamado Hipergate[3].

Após realizar o *deploy* do Hipergate, algumas de suas funcionalidades foram testadas. O caso de teste 4 (CT4) é referente a navegação no Hipergate. Algumas URLs do e-commerce foram acessadas e ações de login realizadas. Para esse teste, foram realizados 30 requisições, sendo 3 vezes de 10 requisições simultâneas.

O caso de teste 5 (CT5) executa as mesmas ações que CT4, entretanto, ao invés de 30 requisições com 3 sequências de 10 requisições simultâneas, foram executadas 300 requisições, com 3 sequências de 100 requisições simultâneas.

Os testes foram realizados nas mesmas condições para todas as quantidades de serviços. O que diferencia dos testes CT1 e CT2, como mostra a tabela 6, é o número de *threads* utilizadas simultâneamente. O mesmo ocorre com os casos CT4 e CT5, que realizam os mesmos testes, entretanto mudando a quantidade de *threads* simultâneas. Pode ser observado também, que o teste CT3 foi realizado somente com uma *thread* e uma vez para cada combinação de máquinas. Isso porque esse teste é o deploy de uma aplicação no Tomcat, não sendo possível executá-lo mais do que 1 vez.

[3] http://www.hipergate.org/

1 máquina indica os testes somente para o software Tomcat, sem nenhuma separação. O serviço nessa máquina contém 1508 classes. 2 máquinas leva em consideração a criação de dois serviços. Essa separação levou em conta somente classes não remotas e classes remotas. O primeiro serviço manteve somente as classes não remotas (1302 classes), e o segundo as classes remotas (206 classes). 4 máquinas são quatro serviços criados. A separação realizada nesse caso levou em conta um estudo prévio das classes que mais gastavam memória, tentando simular uma situação real de utilização do particionador. Nesse caso, o primeiro serviço continha somente as classes não remotas (1302 classes), o segundo serviço as classes que mais gastavam memória que podiam ser remotas (40 classes), o terceiro algumas classes de constantes (37 classes), e o quarto serviço as outras classes menos utilizadas pelo software (129 classes). 10 máquinas gerou dez serviços, entretanto esse caso a separação foi totalmente arbitrária, sendo realizada somente para provar o ponto de que é possível criar quantos serviços forem desejados.

A tabela 6 apresenta a forma resumida como foram realizados os testes. O formato de apresentação dos dados é *threads* simultâneas / qtd. vezes realizada.

Teste/Qtd. Serviços	1 máquina	2 máquinas	4 máquinas	10 máquinas
CT1	10/3	10/3	10/3	10/3
CT2	100/3	100/3	100/3	100/3
CT3	1/1	1/1	1/1	1/1
CT4	10/3	10/3	10/3	10/3
CT5	100/3	100/3	100/3	100/3

Tabela 6 – Resumo dos experimentos realizados: *threads* simultâneas / qtd. vezes realizada

A utilização de 10 ou 100 *threads* simultâneas serve para simular o acesso ao sistema. 10 *threads* simula o acesso de 10 usuários diferentes ao mesmo tempo, enquanto 100 *threads* simula 100 acessos simultãneos.

Para realizar os testes, foram utilizados computadores contendo 8GB de memória RAM e processadores Intel(R) Core(TM) i5 CPU 660 @ 3.33GHz de 64 bits, todos pertencendo a mesma rede. O sistema operacional presente em todas as máquinas foi o Ubuntu 14.04, e a versão Java utilizada para executar os serviços foi a Java 1.7.0.79 OpenJDK 64-Bit Server VM (build 24.79-b02, mixed mode).

As seções a seguir apresentam os resultados de cada caso de teste.

5.1 Caso de teste 1 (CT1)

O Gráfico 11 apresenta os dados agrupados de todas as separações entre serviços, portanto o teste CT1 realizado sem separações, 2 serviços criados, 4 serviços criados e 10 serviços criados.

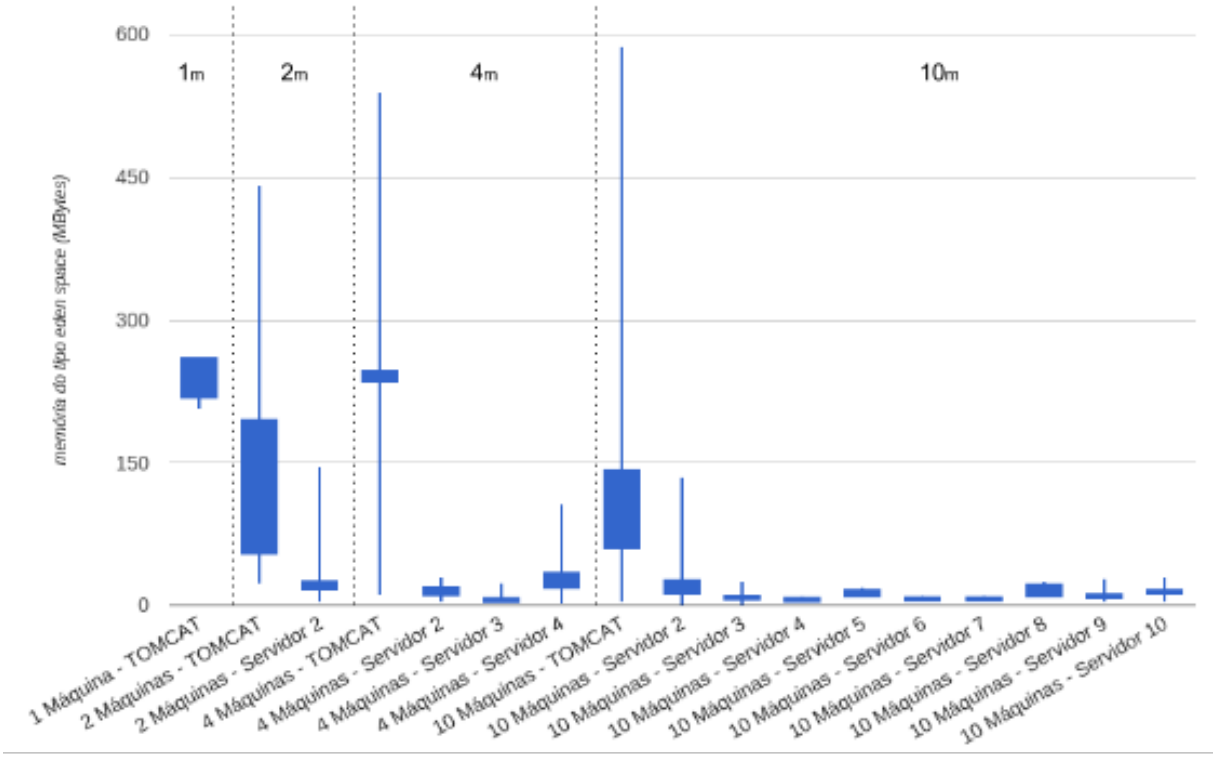


Figura 11 – Gráfico de velas japonesas para caso de teste 1

Na legenda da Figura 11 é possível observar 1 Máquina - TOMCAT, 2 Máquinas - TOMCAT, 4 Máquinas - TOMCAT e 10 Máquinas - TOMCAT. Essas são as memórias das JVMs principais, aquelas que representam os servidores que iniciaram os processos do Tomcat. As outras memórias são relativas às JVMs dos processos que iniciaram os serviços.

O gráfico de velas japonesas apresenta os dados da seguinte maneira: cada vela agrupa leituras ao longo do tempo de execução do caso de teste. O ponto mais alto apresenta o valor máximo de um conjunto, o ponto mais baixo o menor valor, e a parte mais grossa a variação de valores que mais ocorre no conjunto, entre primeiro e terceiro quartis.

O teste CT1 então nos apresenta fato interessantes. Quando comparada a memória máxima utilizada com somente o Tomcat não particionado, com as memórias tanto do serviço Tomcat, quanto com o serviço Servidor 2, da separação em duas máquinas, é possível observar que houve uma real separação da memória entre os servidores. Mesmo que grande parte da memória tenha se mantido em um dos serviços, verifica-se que uma parte da memória foi migrada para o segundo serviço.

Observemos as memórias dos serviços quando estão separados em quatro máquinas. O serviço Tomcat ainda mantém grande parte do uso de memória do conjunto todo, mas como quando separado em dois serviços, esse caso a separação também mostra uma divisão

da memória entre os quatro serviços. O mesmo ocorren quando criados dez serviços.

Isso demonstra que a separação está de fato ocorrendo, e que a sobrecarga de memória em cada máquina diminui à medida que o número de máquinas aumenta, considerando-se 10 usuários simultâneos.

5.2 Caso de teste 2 (CT2)

Analisaremos agora o CT2. Como citado anteriormente, o CT2 realiza os mesmos passos do CT1, entretanto o número de *threads* agora para cada conjunto de requisições, utilizadas concorrentemente, são 100 *threads*.

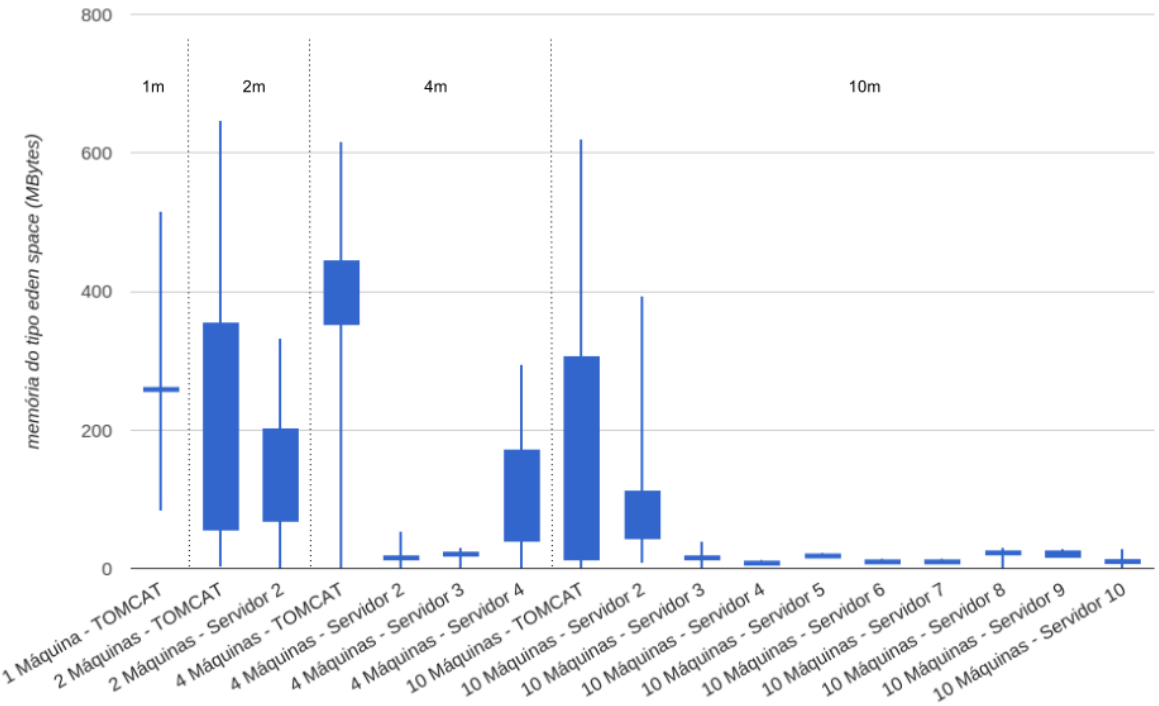


Figura 12 – Gráfico de velas japonesas para caso de teste 2

A Figura 12 apresenta um gráfico um pouco diferente do Gráfico 11. Nesse caso, a separação não originou ganhos para a aplicação, a não ser quando foram criados dez serviços. Nos outros casos, a quantidade de memória utilizada foi muito maior com a separação do que sem ela. Esse fato ocorre pois o número de objetos remotos criados é muito maior, sendo esses objetos remotos mais "pesados" em questão de memória que os objetos "originais". Isso demonstra que o particionamento escolhido para 2 e 4 máquinas não seria adequado caso fosse desejada redução da sobrecarga de memória em situações com muito acesso simultâneo (100 usuários).

5.3 Caso de teste 3 (CT3)

O caso de teste 3 apresenta o *deploy* do Hipergate.

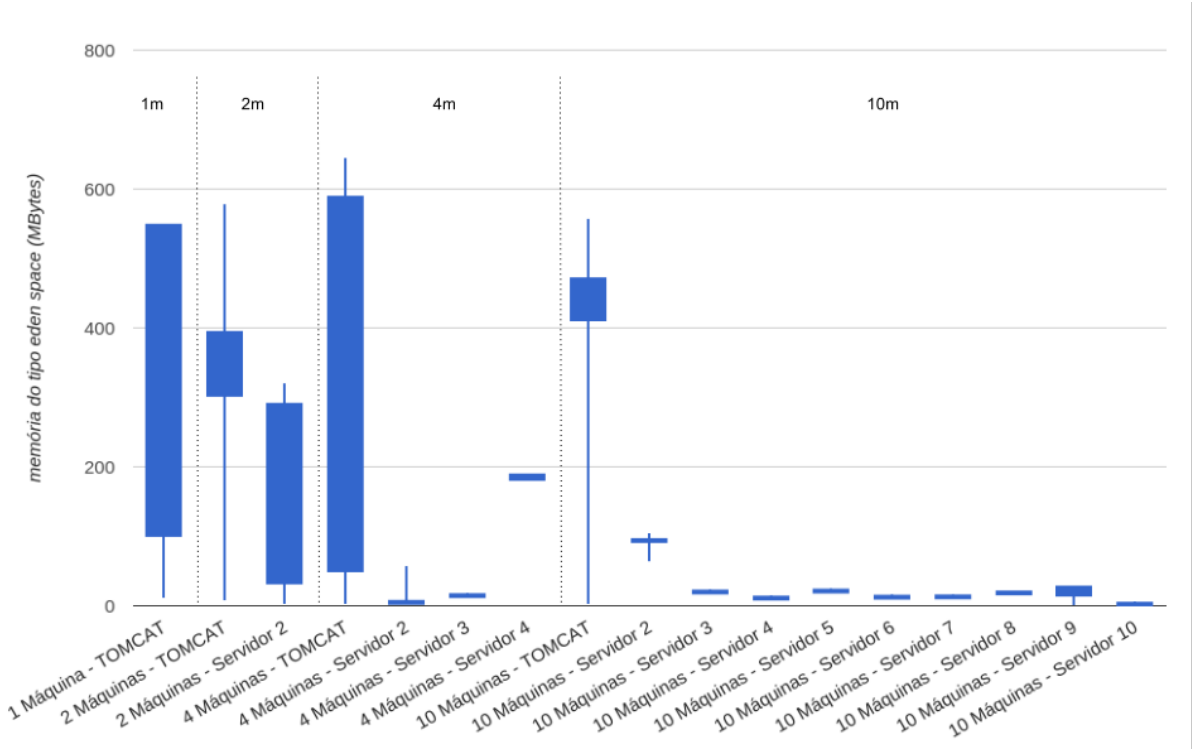


Figura 13 – Gráfico de velas japonesas para caso de teste 3

Verificando as memórias distribuídas do Gráfico 13, os casos com dois serviços e dez serviços conseguiram separar a memória de maneira eficiente, enquanto quatro serviços piorou o desempenho do sistema. Isso significa que para a tarefa de *deploy*, o particionamento para 4 máquinas não foi eficiente, sendo a escolha de distribuição nesse caso ruim.

5.4 Caso de teste 4 (CT4)

No caso de teste 4 foi realizada a navegação no Hipergate, utilizando 10 *threads* simultâneas em 3 sequências distintas.

Como observado na Figura 14, há ganho em todos os casos, assim como ocorreu em CT1. Isso significa que para a tarefa de navegação, a distribuição em todos os casos foi eficiente, diminuindo o consumo de memória em certas máquinas, e repassando o trabalho para outras máquinas.

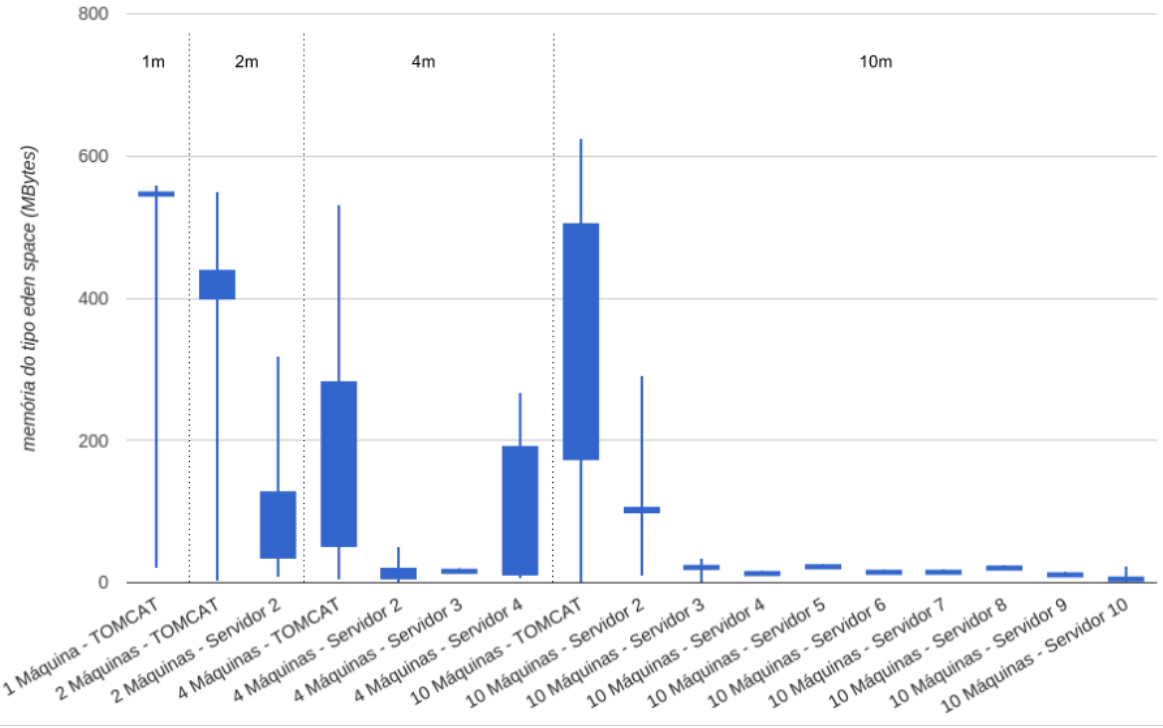


Figura 14 – Gráfico de velas japonesas para caso de teste 4

5.5 Caso de teste 5 (CT5)

Por fim, o último caso de teste, é o mesmo realizado em CT4, entretanto, ao invés de dez *threads* simultâneas, foram utilizadas 100.

Analisando o Gráfico 15, seu comportamento se parece com o ocorrido em CT2. A não ser quando foram criados 10 serviços, não houve ganho em questões de memória, e sim um aumento da quantidade de memória geral. Isso significa que a distribuição escolhida para 2 e 4 máquinas não seria adequada caso fosse desejada redução da sobrecarga de memória em situações com muito acesso simultâneo (100 usuários).

5.6 Fatores de influência nos resultados

Alguns fatores externos podem ter influenciado nos resultados obtidos. Os testes foram todos realizados em laboratório, utilizando software automatizado. Por melhor que seja a simulação, dificilmente ela reflete um ambiente real, onde os usuários acessam o sistema de maneira aleatória, e não padronizada como foi testado. Com o JMeter foi possível realizar uma sequência de acessos automaticamente, com múltiplas *threads* em paralelo, diferente do que aconteceria em um ambiente real, onde usuários do sistema acessariam partes diferentes do mesmo.

Como explicado, os testes com 2 e 10 máquinas foram realizados sem um estudo

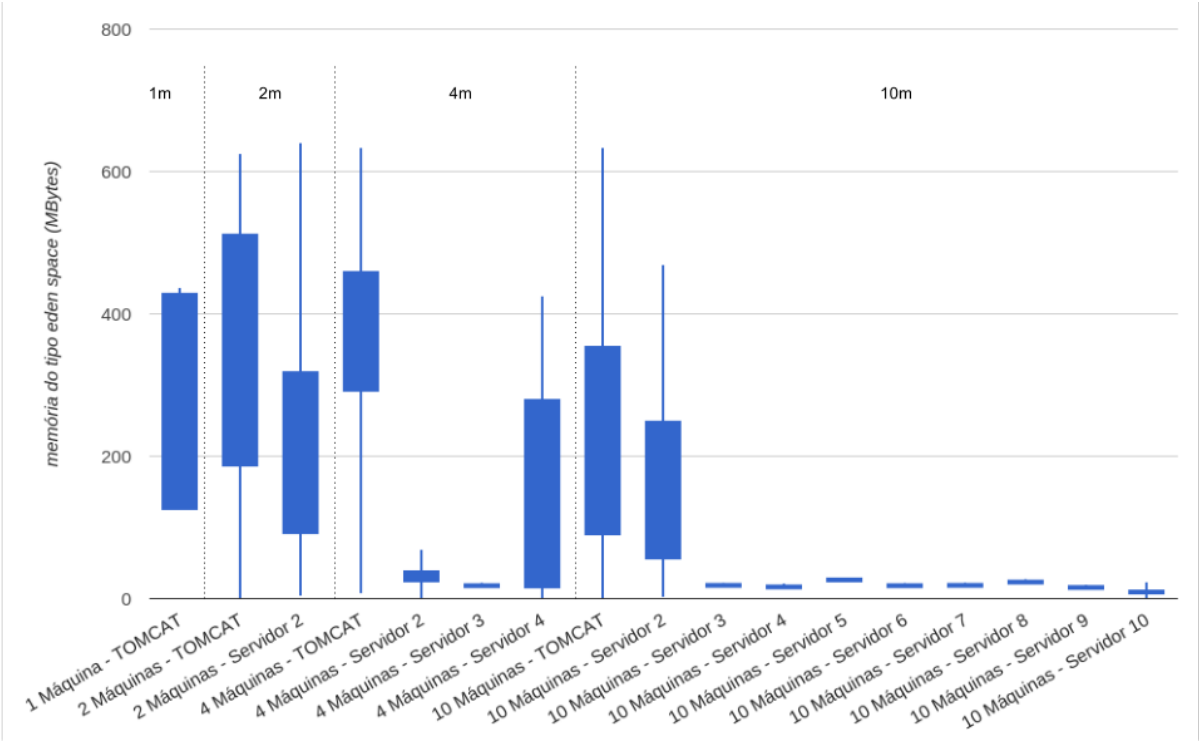


Figura 15 – Gráfico de velas japonesas para caso de teste 5

profundo do comportamento do Tomcat. Para distribuição em 2 máquinas foram somente separadas as classes que não podem ser remotas em um servidor, e as que podem em outro. Em contrapartida, a distribuição em 10 servidores foi realizada de maneira aleatória, para provar o ponto de que é possível distribuir em um número desejado de máquinas, não havendo limitação nesse sentido. Para esse caso, então, não houve também um estudo mais detalhado dos comportamentos das classes do Tomcat.

Por fim, o Hipergate utiliza banco de dados, e esse tipo de interação entre o e-commerce e o banco de dados não foi testado. Somente acessos básicos a URLs e o login, que não realizavam nenhuma operação de troca de informações com o banco de dados foram feitos. Portanto, a aplicação Hipergate não foi testada em sua totalidade.

5.7 Considerações finais

Considerando-se o objetivo desta pesquisa, que foi verificar a validade da distribuição automática, envolvendo a criação dos serviços e da comunicação entre eles, considera-se que o trabalho foi bem sucedido. O Apache Tomcat, mesmo sendo um sistema complexo e não projetado para ser executado de forma distribuída, continuou funcionando corretamente após ser distribuído de diferentes maneiras. Demonstrou-se assim ser possível, a partir do código fonte de um software qualquer, separá-lo em diversos pequenos serviços, realizar a comunicação entre esses serviços, sem demandar muito esforço do desenvolvedor,

que ficou somente responsável por escolher o modo desejado de disposição das classes, e ajustar alguns detalhes que o particionador ainda não consegue abordar.

Em palavreado mais coloquial, ficou demonstrado que o desenvolvedor pode "brincar" com várias configurações rapidamente para melhorar algum aspecto da execução do sistema.

Nos estudos de caso deste trabalho, foi escolhido o aspecto de uso da memória, que trouxe ideias interessantes do potencial que esse tipo de ação pode realizar em um sistema. Houve casos nos quais o ganho de memória foi real (CT1 e CT4), um caso onde houve ganho com certas separações e outras não (CT3).

Vale ainda ressaltar que qualquer outro aspecto poderia ser testado, como segurança, privacidade, custo ou desempenho. Apesar de não ter sido o alvo dos estudos de caso desta dissertação, este último merece ser pesquisado de forma mais aprofundada, uma vez que este aspecto é dos mais importantes no cenário atual, conforme discutido no início desta dissertação. No próximo capítulo são apresentadas algumas ideias para trabalhos futuros neste sentido.

6 Considerações finais

A necessidade de melhoria de sistemas de software para adequarem-se às novas tecnologias é tema constante de pesquisas. Como apresentado no Capítulo 3, conforme a computação evolui, novos desafios surgem, e novas soluções devem ser criadas. Um dos grandes problemas na atualidade é a questão de distribuição. A crescente utilização de diversos tipos de aparelhos para acesso a sites e software, além da facilidade que a Internet proporciona ao acesso de informações, força pesquisadores da área a manterem grandes esforços melhorando aplicações já desenvolvidas, ou mesmo pensando em formas de facilitar o desenvolvimento de software de forma a rodar em vários dispositivos.

O intuito geral dessa pesquisa de mestrado foi de distribuir sistemas de software inicialmente projetados para rodar em um único computador de maneira a eliminar muitas tarefas que o desenvolvedor repetiria exaustivamente nesse processo. A idéia foi permitir que programadores possam, a partir do código fonte de alguma aplicação e os modelos de distribuição, realizar a divisão do seu software de maneira rápida, a fim de testar diversas distribuições sem custo de tempo.

6.1 Contribuições alcançadas

Esta dissertação de mestrado apresentou um particionador semi-automático de código, onde o desenvolvedor escolhe a forma que deseja criar seus serviços, a partir de código fonte e documentos DSDM, e são gerados aplicativos que comunicam-se entre si, porém mantendo todas as funcionalidades do software inicial, sendo essa a contribuição principal. Isso auxilia desenvolvedores, permitindo que inicialmente concentrem seus esforços no projeto de software, não preocupando-se com a distribuição dos mesmos. Após o software pronto, os desenvolvedores utilizam o particionador para testar modos de distribuição diferentes, não precisando se preocupar com a implementação dessa distribuição.

Uma outra contribuição que surgiu ao longo da pesquisa é referente à utilização de RMI na programação de sistemas distribuídos. Quando inserido nesse universo, muitos desenvolvedores não tem noção de que podem existir casos específicos onde um objeto não pode ser transportado de uma JVM para outra. A pesquisa traz alguns casos que, caso um desenvolvedor de sistemas deseje utilizar RMI, existem restrições para programação, facilitando certas escolhas no desenvolvimento do software distribuído.

Por fim, uma contribuição importante desse projeto de mestrado é expor as pesquisas relacionadas à área. Muitas dessas são sobre particionamento de código, contudo pensando na arquitetura distribuída, ou facilitadoras de desenvolvimento de sistemas dis-

tribuídos. O projeto desenvolvido abre espaços para novas linhas de pesquisa que podem ser exploradas, diferentes das vistas tradicionalmente na área de distribuição de código. Essa área, como mencionada durante todo o projeto, denomina-se distribuição tardia. Portanto, a última contribuição dessa pesquisa é a iniciativa em uma nova linha de pesquisa pouco explorada até então.

6.2 Trabalhos futuros

Como mencionado na Seção 6.1, uma nova linha de pesquisa pouquíssimo explorada pode atrair a atenção de pesquisadores da área de computação e principalmente particionamento de código e sistemas distribuídos. Essa área denominamos de distribuição tardia.

O projeto desenvolvido nesse mestrado traz a utilização de linguagens e tecnologias específicas. Java e RMI foram utilizados, verificados seu funcionamento, facilidades e restrições. Novas pesquisas podem abordar outras linguagens, identificando ser possível ou não desenvolver um particionador para as mesmas. Mantendo o pensamento em Java, pode-se explorar outras tecnologias de comunicação, como REST, para verificar a eficácia com esse modelo de troca de informações entre serviços e permitindo que os microsserviços sejam expostos de forma mais padronizada e acessível a outros sistemas.

Seguindo em outra área, pode-se imaginar o mesmo problema abordado nesse projeto para banco de dados. Pensando no modelo relacional como exemplo, podem existir tabelas que são muito mais acessadas, tanto para leitura quanto para gravação de dados, enquanto outras não são tão recorrentemente usadas. Havendo uma maneira de distribuir conjuntos de tabelas em servidores diferentes gerando toda comunicação necessária entre elas, pode-se tentar melhorar o desempenho do sistema. Então, caso existam algumas tabelas que sejam acessadas mais que outras, distribuir essas tabelas em outros servidores e aplicar algum processo de replicação, pode-se tentar alcançar um melhor desempenho, evitando gargalos de acesso às tabelas. Pode-se pensar na questão de segurança de dados. Existem sistemas contendo tabelas com dados críticos. Essas tabelas podem ser separadas da aplicação, e colocadas em servidores mais seguros.

Por fim, é necessário realizar mais estudos de caso focados no desempenho, que não foi devidamente explorado nesta pesquisa. Conforme discutido no capítulo anterior, nas análises feitas houve perda de desempenho na maior parte das distribuições, principalmente devido à latência e necessidade de gerenciamento dos objetos remotos. Não foi feito nenhum tipo de esforço de escalabilidade com os nós de processamento, uma vez que a abordagem limitou-se a viabilizar a distribuição da aplicação em microsserviços, sem considerar sua replicação. Em outras palavras, a aplicação foi particionada de forma que cada partição fosse executada em um único nó. Em um cenário real, o próximo passo seria

adicionar um distribuidor de carga e promover a replicação daqueles nós que hospedam os serviços mais acessados, melhorando assim o desempenho global, a exemplo do que foi apresentado na Figura 4 da Seção 2.2.1.

Além disso, deve-se considerar o fato de que a aplicação utilizada nos testes pode não ser a candidata ideal para uma possível melhora de desempenho. O Apache Tomcat é uma aplicação tipicamente monolítica, otimizada com o passar dos anos. Em futuros estudos de caso, sugere-se a utilização de outro tipo de aplicação como objeto da distribuição.

Bibliografia

Bass, L., P. Clements e R. Kazman (2012). *Software Architecture in Practice*. SEI Series in Software Engineering. Pearson Education. ISBN: 9780132942782. URL: http://books.google.com.br/books?id=-II73rBDXCYC (ver p. 16).

Bellomo, Stephany (2013). *Agile and Architecture Practices for Rapid Delivery*. Acessado em: fev, 2014. URL: http://blog.sei.cmu.edu/post.cfm/agile-architecture-for-rapid-delivery-272 (ver pp. 16, 17).

Bellomo, Stephany, Robert L. Nord e Ipek Ozkaya (2013). "A Study of Enabling Factors for Rapid Fielding: Combined Practices to Balance Speed and Stability". Em: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, pp. 982–991. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486923 (ver p. 16).

Berners-Lee, T., R. Fielding e L. Masinter (1998). *Uniform Resource Identifiers (URI): Generic Syntax*. RFC Editor, United States (ver p. 31).

Bondi, André B. (2000). "Characteristics of Scalability and Their Impact on Performance". Em: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. Ottawa, Ontario, Canada: ACM, pp. 195–203. ISBN: 1-58113-195-X. DOI: 10.1145/350391.350432. URL: http://doi.acm.org/10.1145/350391.350432 (ver p. 15).

Brambilla, Marco, Jordi Cabot e Manuel Wimmer (2012). *Model-Driven Software Engineering in Practice*. Anglais. Morgan & Claypool, p. 182. ISBN: 9781608458820. URL: http://hal.inria.fr/hal-00755006 (ver p. 22).

Cornhill, Dennis (1984). "Partitioning Ada* programs for execution on distributed systems". Em: *Data Engineering, 1984 IEEE First International Conference on*, pp. 364–370. DOI: 10.1109/ICDE.1984.7271294 (ver p. 40).

Deursen, Arie van, Paul Klint e Joost Visser (2000). "Domain-specific Languages: An Annotated Bibliography". Em: *SIGPLAN Not.* 35.6, pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035. URL: http://doi.acm.org/10.1145/352029.352035 (ver p. 25).

Doernhoefer, Mark (2011). "Surfing the Net for Software Engineering Notes". Em: *SIGSOFT Softw. Eng. Notes* 36.2, pp. 10–18. ISSN: 0163-5948. DOI: 10.1145/1943371.1943375. URL: http://doi.acm.org/10.1145/1943371.1943375 (ver pp. 27, 30, 31).

Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley. ISBN: 0-201-48567-2 (ver p. 17).

Fowler, Martin (2001). “Extreme Programming Examined”. Em: ed. por Giancarlo Succi e Michele Marchesi. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Cap. Is Design Dead?, pp. 3–17. ISBN: 0-201-71040-4. URL: http://dl.acm.org/citation.cfm?id=377517.377518 (ver p. 16).

— (2014). *Microservices - a definition of this new architectural term*. Acessado em: jan, 2015. URL: http://martinfowler.com/articles/microservices.html (ver pp. 15, 28).

Geist, G. A., J. A. Kohl e P. M. Papadopoulos (1996). “PVM and MPI: a Comparison of Features”. Em: *Calculateurs Paralleles* 8, pp. 137–150 (ver p. 37).

Hunt, Galen C. e Michael L. Scott (1999). “The Coign Automatic Distributed Partitioning System”. Em: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI ’99. New Orleans, Louisiana, USA: USENIX Association, pp. 187–200. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296826 (ver p. 41).

Hunt, G.C. e M.L. Scott (1998). “A guided tour of the Coign automatic distributed partitioning system”. Em: *Enterprise Distributed Object Computing Workshop, 1998. EDOC ’98. Proceedings. Second International*, pp. 252–262. DOI: 10.1109/EDOC.1998.723260 (ver p. 41).

IBM, Alex Rodriguez (2008). *RESTful Web services: The basics*. Acessado em: jan, 2014. URL: http://www.ibm.com/developerworks/webservices/library/ws-restful/ (ver p. 31).

Islam, S.M.S. e M. Rokonuzzaman (2009). “Adaptation of ATAMSM to software architectural design practices for organically growing small software companies”. Em: *Computers and Information Technology, 2009. ICCIT ’09. 12th International Conference on*, pp. 488–493. DOI: 10.1109/ICCIT.2009.5407288 (ver p. 16).

Janus, André (2012). “Towards a Common Agile Software Development Model (ASDM)”. Em: *SIGSOFT Softw. Eng. Notes* 37.4, pp. 1–8. ISSN: 0163-5948. DOI: 10.1145/2237796.2237803. URL: http://doi.acm.org/10.1145/2237796.2237803 (ver p. 17).

JDT, Eclipse. *Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model - Tutorial*. Acessado em: jan, 2015. URL: http://www.vogella.com/tutorials/EclipseJDT/article.html (ver p. 26).

Koivulahti-Ojala, M. e T. Kakola (2010). “Framework for Evaluating the Version Management Capabilities of a Class of UML Modeling Tools from the Viewpoint of Multi-Site, Multi-Partner Product Line Organizations”. Em: *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pp. 1–10. DOI: 10.1109/HICSS.2010.213 (ver p. 24).

Lucrédio, Daniel (2009). “Uma Abordagem Orientada a Modelos para Reutilização de Software”. 277 páginas. Tese de doutorado. USP - São Carlos (ver pp. 22–24).

Mehta, Mayur R, Sam Lee e Jaymeen R Shah (2006). "Service-Oriented Architecture: Concepts and Implementation". Em: *Proceedings of the Information Systems Education Conference (ISECON)*. Vol. 23. 2335, p. 1 (ver p. 27).

Monteiro, Rui et al. (2012). "Model-Driven Development for Requirements Engineering: The Case of Goal-Oriented Approaches". Em: *Eighth International Conference on the Quality of Information and Communications Technology*, pp. 75–84 (ver p. 17).

Nuez-Reyna, I. e H. Cervantes (2009). "Using Adapted Software Architecture Development Methods in a SOA Context". Em: *Computer Science (ENC), 2009 Mexican International Conference on*, pp. 240–251. DOI: 10.1109/ENC.2009.12 (ver p. 16).

OMG (2003). *MDA Guide Version 1.0.1*. Acessado em: nov, 2013. URL: http://www.omg.org/mda (ver p. 21).

OpenGroup. *What is SOA?* Acessado em jan, 2014. URL: http://www.opengroup.org/soa/source-book/soa/soa.htm (ver p. 27).

Papazoglou, Mike P. e Willem-Jan Heuvel (2007). "Service Oriented Architectures: Approaches, Technologies and Research Issues". Em: *The VLDB Journal* 16.3, pp. 389–415. ISSN: 1066-8888. DOI: 10.1007/s00778-007-0044-3. URL: http://dx.doi.org/10.1007/s00778-007-0044-3 (ver pp. 27, 28, 31).

Papotti, Paulo Eduardo, Antonio Francisco do Prado e Wanderley Lopes de Souza (2012). "An Approach to Support Legacy Systems Reengineering to MDD Using Metaprogramming". Em: *XXXVIII Conferencia Latinoamericana En Informatica*, pp. 1–10 (ver pp. 17, 23).

Patel, Hardik e Sailaja Navvluru (2008). *IBM Migration Toolkit support for migrating data from MySQL to DB2 and informix Dynamic Server*. Acessado em: fev, 2014. URL: http://www.ibm.com/developerworks/data/library/techarticle/dm-0807patel/ (ver p. 17).

Pautasso, Cesare, Olaf Zimmermann e Frank Leymann (2008). "Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision". Em: *Proceedings of the 17th International Conference on World Wide Web*. WWW '08. Beijing, China: ACM, pp. 805–814. ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367606. URL: http://doi.acm.org/10.1145/1367497.1367606 (ver pp. 30, 31).

PostgreSQL (2008). *Converting from other Databases to PostgreSQL*. Acessado em: fev, 2014. URL: http://wiki.postgresql.org/wiki/Converting_from_other_Databases_to_PostgreSQL (ver p. 17).

Sairaman, Viswanath (2010). "A Generalized Framework for Automatic Code Partitioning and Generation in Distributed Systems". 111 páginas. Tese de doutorado. University of South Florida (ver pp. 16, 35–40).

Samir, H., E. Stroulia e A. Kamel (2007). "Swing2Script: Migration of Java-Swing Applications to Ajax Web Applications". Em: *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pp. 179–188. DOI: 10.1109/WCRE.2007.48 (ver p. 17).

Savchenko, D.I., G.I. Radchenko e O. Taipale (2015). "Microservices validation: Mjolnirr platform case study". Em: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, pp. 235–240. DOI: 10.1109/MIPRO.2015.7160271 (ver pp. 28, 29).

Thommazo, André Di (2014). "Um conjunto de abordagens para a geração da matriz de rastreabilidade de requisitos com suporte de técnicas de inteligência computacional". 150 páginas. Tese de doutorado. UFSCar - São Carlos (ver p. 15).

Vahid, F. e D.D. Gajski (1995). "Closeness metrics for system-level functional partitioning". Em: *Design Automation Conference, 1995, with EURO-VHDL, Proceedings EURO-DAC '95., European*, pp. 328–333. DOI: 10.1109/EURDAC.1995.527425 (ver p. 41).

Vallerio, K.S. e N.K. Jha (2003). "Task graph extraction for embedded system synthesis". Em: *VLSI Design, 2003. Proceedings. 16th International Conference on*, pp. 480–486. DOI: 10.1109/ICVD.2003.1183180 (ver p. 36).

Villamizar, M. et al. (2015). "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". Em: *Computing Colombian Conference (10CCC), 2015 10th*, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476 (ver p. 15).

W3C (2001). *Web Services Description Language (WSDL) 1.1*. Acessado em: jan, 2014. URL: http://www.w3.org/TR/wsdl (ver p. 27).

— (2007). *SOAP Version 1.2 Part 1: Messaging Framework*. Acessado em: jan, 2014. URL: http://www.w3.org/TR/soap12-part1/ (ver p. 30).

Zhou, Hong, H. Yang e A. Hugill (2010). "An Ontology-Based Approach to Reengineering Enterprise Software for Cloud Computing". Em: *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pp. 383–388. DOI: 10.1109/COMPSAC.2010.46 (ver p. 41).