Implantação automatizada de composições de serviços web de grande escala

Leonardo Alexandre Ferreira Leite

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
a obtenção do título
de
mestre em Ciência da Computação

Orientador: Prof. Dr. Marco Aurélio Gerosa

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro pelo projeto CHOReOS, financiado pela Comissão Europeia, e pelo projeto Baile, financiado pela HP Brasil.

São Paulo, ToDo ►MÊS◄ de 2014

Implantação automatizada de composições de serviços web de grande escala

Esta é a versão original da dissertação elaborada
pelo candidato Leonardo Alexandre Ferreira Leite,
tal como submetida à Comissão Julgadora.

Agradecimentos

Valeu galera \o/

Resumo

Fabio ►*a primeira frase está estranha pois começa falando de automatizada e depois muda para sistematizada, que são coisas diferentes.*◄

A implantação automatizada é uma necessidade no ciclo de vida de um sistema de grande escala, mas muitas organizações ainda realizam a implantação de seus sistemas de forma não-sistematizada, tornando o processo de implantação moroso, propenso a erros e não-reprodutível. Esses problemas agravam-se ao implantar um sistema distribuído, como é o caso de coreografias de serviços web, que implementam processos de negócios distribuídos entre várias organizações. A implantação de uma coreografia deve ser coordenada, pois os serviços de uma coreografia precisam conhecer a localização dos outros serviços, informação possivelmente disponível apenas em tempo de implantação. Coreografias de grande escala são mantidas de forma distribuída por várias organizações e a presença de falhas na comunicação entre seus serviços torna-se corriqueira. Considerando as vantagens da virtualização na gerência de ambientes, o crescente uso da computação em nuvem pelas organizações e os requisitos de sistemas de grande escala, investigamos o uso da computação em nuvem na implantação de coreografias de serviços web. Para isso, desenvolvemos o CHOReOS Enactment Engine, um sistema de middleware que possibilita a implantação distribuída e automatizada de coreografias de serviços web, operando como um provedor de computação em nuvem na camada de Plataforma como um Serviço. O middleware desenvolvido será avaliado pela sua escalabilidade em relação ao tempo de implantação das coreografias, operação para a qual a quantidade de serviços a ser implantada é considerada como carga do sistema, enquanto que a quantidade de máquinas virtuais acessíveis são os recursos do sistema. No atual estágio de implementação do Enactment Engine, experimentos preliminares de escalabilidade foram realizados, mostrando que um aumento de 50 vezes no número de serviços implantados provocou um aumento de cerca de apenas duas vezes no tempo de implantação quando os recursos do sistemas eram aumentados na mesma proporção que a carga aplicada, o que consideramos um resultado preliminar muito satisfatório.

Fabio ►*Esse resumo está longo e detalhado demais. Seria melhor algo entre 10 e 15 linhas concentrando-se no que é mais importante no trabalho e suas contribuições.*◄

Palavras-chave: implantação de software, coreografias, serviços web, computação em nuvem, grande escala.

Abstract

Automated deployment is mandatory in the life cycle of large-scale systems. However, some organizations still deploy their system manually, what is time-consuming, error-prone, and non-reproducible. These problems are even worse in distributed deployment, as occurs with web service choreographies, that implement distributed business process among many organizations. The deployment of a choreography must be coordinated, since their services need to retrieve the endpoints of other participant services, and these endpoints may be available only at deployment time. Large scale choreographies are maintained by multiple organizations in a distributed way, and in such scenario communication faults are commonplace. Considering the virtualization advantages in environment management, the increasing use of cloud computing by organizations, and large scale system requirements, we exploit cloud computing in web service choreography deployment. This is achieved by means of the development of the CHOReOS Enactment Engine, a middleware system that enables the automated and distributed deployment of web service choreographies, operating as a cloud computing provider in the Platform as a Service layer. Our middleware is assessed by its scalability regarding choreography deployment time, for which the amount of services to be deployed is the system load, whereas the amount of available virtual machines are the system resources. The current Enactment Engine implementation was assessed regarding its scalability, and we observed an increase about only twice when the number of services was increased by 50 times, and the resources were increased in the same proportion than the load. We consider this preliminary result very satisfactory.

Keywords: software deployment, choreography, web services, cloud computing, large scale.

Sumário

Lista de Abreviaturas **ix**

Lista de Figuras **xi**

Lista de Tabelas **xiii**

1 Introdução **1**

2 Conceitos básicos **5**

2.1 Serviços web 5
2.2 Composições de serviços web 7
2.3 O processo de implantação de sistemas 8
2.4 Computação em nuvem 11
2.5 Desafios na implantação de sistemas de grande escala 13

3 Trabalhos relacionados **17**

4 Solução proposta: o Enactment Engine **23**

4.1 Execução do Enactment Engine 24
4.2 Especificação da composição de serviços 26
4.3 Ligação entre serviços 27
4.4 Mapeamento dos serviços na infraestrutura alvo 28
4.5 Interface do Enactment Engine 28
4.6 Pontos de extensão 30
4.7 Aspectos gerais de implementação 32
4.8 Discussão: auxiliando implantações em grande escala 34

5 Avaliação **39**

5.1 Implantando coreografias com e sem o EE 39
5.2 Análise de desempenho e escalabilidade 41

6 Conclusões **45**

A Guia do Usuário do Enactment Engine **47**

Referências Bibliográficas **49**

Lista de Abreviaturas

2PC	Two Phase Commit
ADL	Architectural Description Language
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AWS	Amazon Web Services
BPEL	Business Process Execution Language
BPMN	Business Process Modeling Notation
CAP	Consistency, Availability, Partitioning
CORBA	Common Object Request Broker Architecture
EC2	Elastic Compute Cloud
GNU	GNU is not Unix
HTTP	Hyper Text Transfer Protocol
IaaS	Infrastructure as a Service
J2EE	Java Enterprise Edition
JDK	Java Development Kit
JMS	Java Message Service
JVM	Java Virtual Machine
LoC	Lines of code
MIL	Module Description Language
MIME	Multipurpose Internet Mail Extensions
MPL	Mozilla Public License
NIST	The National Institute of Standards and Technology
PaaS	Platform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
SOA	Service Oriented Architecture
TDD	Test Driven Development
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WS-CDL	Web Services Choreography Description Language
WSCI	Web Service Choreography Interface
WSDL	Web Service Description Language

Lista de Figuras

1.1 Modelos da computação em nuvem associadas ao CHOReOS Enactment Engine. . . 2

2.1 Exemplo de uma pequena coreografia de serviços em notação BPMN2. 8
2.2 Exemplo básico de pipeline de implantação. 10
2.3 Tempos de criação de instâncias EC2 observados, em segundos. 14

4.1 Ambiente de execução do CHOReOS Enactment Engine. 24
4.2 Processo de implantação implementado pelo Enactment Engine. 25
4.3 Estrutura da descrição arquitetural de uma coreografia. 26
4.4 Uma instância de Invoker é parametrizada com uma tarefa, uma quantidade de tentativas, um timeout por tentativa e um intervalo de tempo entre as tentativas. 35

5.1 Topologia das composições utilizadas em nossos experimentos. 41
5.2 Tempos médios de implantação com aumento constante na quantidade de serviços implantados, mantendo-se constante a razão serviços implantados / nós. ToDo ►*traduzir figura*◄ 43

Lista de Tabelas

3.1 Tabela comparativa com os trabalhos relacionados 21

5.1 Cenários de implantação para a análise multi-variada. 41
5.2 Resultados experimentais. 42

Capítulo 1

Introdução

O processo de implantação de um software vai do momento de aquisição do software até o momento em que o software encontra-se em execução [OMG06]. Processos totalmente automatizados são importantes na implantação de sistemas [HF11]. No entanto, muitas organizações ainda realizam a implantação de seus sistemas como descrito por Dolstra et al. [DBV05]: um processo manual, moroso, propenso a erros e não reprodutível. Ainda segundo esses autores, o problema agrava-se na implantação de sistemas distribuídos, pois o esforço de implantação cresce com a quantidade de nós do sistema.

Gerosa ►*Está começando fraco. A 1a frase precisa ser mais impactante.*◄

Serviços web possibilitam a comunicação interoperável entre máquinas pela rede [W3C04b] e podem ser compostos para implementar sofisticados processos de negócios [PTDL07]. Especialistas do setor aéreo, por exemplo, já propõe o uso de composições de serviços para automatizar os processos de negócios entre diferentes organizações que coabitam um aeroporto [CV12]. Considerando os atuais números relativos a grandes aeroportos[1] e o crescimento futuro desses números, espera-se que composições de serviços envolvam um grande número de participantes, conforme já sugerido por pesquisadores [IGH+11] **Gerosa** ►*nomes ou mais refs*◄ .

No entanto, o desenvolvimento de colaborações entre serviços trazem desafios para a formulação de mecanismos que funcionem, escalem e que sejam eficientemente implementados em um ambiente distribuído de *grande escala* [SPV12]. Nesse cenário de grande escala, o processo de implantação enfrenta diversas dificuldades, tais como falhas corriqueiras na infraestrutura, heterogeneidade tecnológica dos componentes operados, distribuição do sistema por diferentes organizações e atualização frequente dos componentes em operação. Com essas dificuldades, torna-se inviável manter a escalabilidade do processo de implantação sem a utilização de um processo de implantação totalmente automatizado.

Esses desafios podem ser tratados por soluções *ad-hoc*, nas quais um processo de implantação é automatizado tendo em vista uma composição de serviço específica. Contudo, esse caminho leva ao baixo reúso de soluções dentro de uma organização e entre as organizações participantes. Outro caminho é a utilização de soluções baseadas em um *middleware*, que resolvem os problemas comuns de implantação, fornecendo soluções potencialmente mais sofisticadas e mais bem testadas. Isso ocorre pois contribuidores interessados no problema de implantação trabalham juntos para fornecer uma infraestrutura mais robusta, enquanto usuários do middleware escrevem código menor e mais simples para automatizar o processo de implantação de composições específicas. Embora apresentem vantagens, sistemas apoiados por middleware também apresentam desvantagens, principalmente no que diz respeito às restrições impostas ao desenvolvimento da aplicação.

Nesta dissertação, estudamos o processo de implantação automatizada baseado em um middleware. Investigamos o quanto e como essa opção contribui à implantação de composições de serviço de grande escala quando confrontada com soluções *ad-hoc*. Para responder à questão colo-

[1] Heathrow [http://www.heathrowairport.com] em Londres, por exemplo, possui mais de 80 compahias aéreas, 190.000 passageiros por dia (picos de 230.000), 6.000 empregados, 1.000 pousos e decolagens por dia, e 40 serviços de refeição.

cada, nosso objetivo nesta dissertação é projetar, implementar e avaliar um middleware que forneça suporte à implantação automatizada de composições de serviços web de grande escala. Esse middleware, quando comparado a soluções *ad-hoc*, deve *facilitar* a automação da implantação de composições diversas. Nesse caso, *facilitar* significa reduzir o tempo e/ou quantidade de trabalho para a codificação e/ou aplicação da solução de implantação. Avaliamos o tempo em homens*horas e a quantidade de trabalho em linhas de código. Também espera-se que a automação do processo fornecida por esse middleware contribua para a escalabilidade do processo de implantação. Nesse caso, ser escalável significa ser capaz de implantar uma maior quantidade de serviços sem aumentar o tempo de implantação, dado que se aumente também, proporcionalmente, a quantidade de servidores disponíveis para hospedar esses serviços.

A computação em nuvem possibilita o acesso a um conjunto compartilhado de recursos computacionais que podem ser providos rapidamente [MG11]. A gerência programática de recursos virtualizados, fornecidos pela nuvem, favorece a criação de processos totalmente automatizados para a implantação de sistemas [HF11]. Além disso, sistemas distribuídos já estão migrando para ambientes de nuvem, onde são compostos e mantidos de modo descentralizado por várias organizações [SPV12]. Baseando-se nessas considerações, nosso middleware foi projetado em função dos modelos de computação em nuvem.

O middleware desenvolvido no contexto deste trabalho é o CHOReOS Enactment Engine[2] (EE), que funciona no modelo de Plataforma como um Serviço (PaaS), um dos modelos de funcionamento da computação em nuvem. O EE fornece uma API remota para disparar o processo de implantação. Essa API recebe uma especificação declarativa da composição a ser implantada. O EE interpreta a especificação recebida e realiza as tarefas de implantação em um conjunto de máquinas virtuais. Essas máquinas virtuais são criadas por provedores de infraestrutura que funcionam de acordo com o modelo de computação em nuvem denominado Infraestrutura como um Serviço (IaaS). Ao fim da implantação, as composições de serviços estão disponíveis para serem consumidas por usuários, operando no modelo de Software como um Serviço (SaaS). A arquitetura de nossa solução em função dos modelos de computação em nuvem pode ser observada na Figura 1.1. Fabio ►*essa figura não mostra a arquitetura de nossa solução, ela mostra outra coisa.*◄

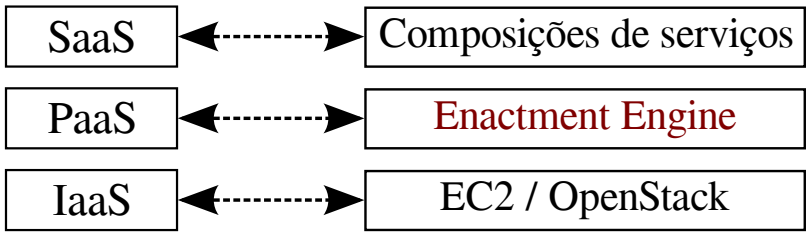

Figura 1.1: *Modelos da computação em nuvem associadas ao CHOReOS Enactment Engine.*

Esta pesquisa foi feita no contexto e com financiamento dos projetos CHOReOS[3] e Baile[4], que estudaram a aplicação de composições de serviços distribuídas, chamadas *coreografias*, em ambientes de grande escala. O projeto CHOReOS, financiado pela Comissão Europeia e composto por diversas instituições acadêmicas e industriais da Europa conjuntamente com o IME-USP, objetivou desenvolver um processo dinâmico e centrado no usuário para o desenvolvimento de coreografias em um ambiente de escala ultra grande, no qual milhares de serviços são compostos e coordenados por um middleware distribuído. O projeto Baile, uma parceria entre IME-USP e HP Brasil, estudou a solução de problemas para o desenvolvimento de coreografias, como a adoção de Desenvolvimento Guiado por Testes (TDD) no contexto de coreografias e o suporte da Computação em Nuvem à implantação de coreografias.

[2] http://ccsl.ime.usp.br/EnactmentEngine
[3] http://www.choreos.eu
[4] http://ccsl.ime.usp.br/baile

As contribuições deste trabalho são:

- A implementação de um middleware que possibilita a implantação automatizada de composições de serviços. Além de possuir aplicabilidade direta para profissionais da indústria, nosso middlware facilita a condução de avaliações empíricas ligadas à implantação de composições de serviço, tendo assim potencial para alavancar diversas outras pesquisas sobre composições de serviços.
- Uma comparação, baseada na literatura e em evidências empíricas, entre soluções de implantação automatizada implementadas de forma *ad-hoc* e implementadas com suporte por middleware.

Os esforços iniciais desta pesquisa, focando na implantação de composições de serviços em um ambiente de computação em nuvem, ainda sem considerar os desafios de grande escala, resultaram na publicação:

> Leonardo Leite, Nelson Lago, Marco Aurélio Gerosa e Fabio Kon. Um Middleware para Encenação Automatizada de Coreografias de Serviços Web em Ambientes de Computação em Nuvem. Em *31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, 2013.

Durante o desenvolvimento do software Enactment Engine, o autor desta dissertação utilizou nos testes de unidade um padrão de software que foi documentado em um artigo de sua autoria:

> Leonardo Leite. Fábrica dinâmica de dublês: testando classes que possuem dependências não injetáveis. Em *Miniconferência Latino-Americana de Linguagens de Padrões para Programação (MiniPlop Brasil)*, 2013.

Ainda no contexto deste mestrado, foi realizado um estudo sobre adaptação dinâmica de coreografias, o que resultou na publicação do artigo:

> Leonardo Leite, Gustavo Oliva, Guilherme Nogueira, Marco Aurélio Gerosa, Fabio Kon e Dejan Milojicic. A systematic literature review of service choreography adaptation. *Service Oriented Computing and Applications*, 3(7):201–218, 2013.

Esta dissertação organiza-se da seguinte forma: as fundamentações teóricas sobre composição de serviços, o processo de implantação e a computação em nuvem são apresentadas no Capítulo 2. No Capítulo 3, apresentamos os trabalhos relacionados. No Capítulo 4, apresentamos o CHOReOS Enactment Engine, discutindo como suas características arquiteturais e de implementação auxiliam na implantação de composições de grande escala. A comparação do EE com soluções *ad-hoc*, bem como sua avaliação de desempenho e escalabilidade, são apresentadas no Capítulo 5. Por fim, no Capítulo 6, apresentamos nossas conclusões.

Capítulo 2

Conceitos básicos

Neste capítulo apresentaremos conceitos que são a base para esta pesquisa. Os conceitos apresentados abordam serviços web e suas composições, implantação de sistemas e, por fim, os desafios particulares da implantação de sistemas de grande escala.

2.1 Serviços web

Serviços são entidades autônomas e independentes de plataforma, que podem ser descritas, publicadas, encontradas e compostas [PTDL07]. O conceito de serviço possui semelhanças com o conceito de *componentes*. Componentes foram idealizados para que sistemas fossem construídos com "blocos" fornecidos por terceiros [MBNR68]. Esses blocos teriam interfaces bem definidas e, com isso, seriam conectáveis entre si, sem que o desenvolvedor precise entender sobre a implementação desses blocos. Szyperski [Szy03] define componente como uma unidade de composição, possuindo uma especificação de interface contratual e declaração explícita de suas dependências.

Uma das utilidades fundamentais de componentes e serviços é proporcionar a ligação entre sistemas heterogêneos. Em busca desse objetivo, a OMG liderou esforços para a construção de uma solução para a comunicação de objetos codificados em diferentes linguagens e executados em diferentes plataformas [Szy10]. Desse esforço nasceu o Common Object Request Broker Architecture (CORBA). A especificação CORBA [OMG95] define um *objeto* como uma "entidade encapsulada e identificável que fornece um ou mais serviços que podem ser requisitados por um cliente". Ainda na especificação CORBA, uma interface é definida como uma "descrição do conjunto de possíveis operações que um cliente pode requisitar para um objeto por essa interface". Uma interface de um componente CORBA é concretamente descrita utilizando-se a Interface Description Language (IDL). Além da definição de objetos, há também a definição de *componentes* CORBA, cujas principais características são a presença de conjuntos de interface providas, interfaces requiridas, eventos emitidos e eventos recebidos [Szy10].

As características apresentadas dos objetos CORBA têm muito em comum com o conceito de serviços: interfaces bem definidas e acessíveis remotamente. Dessa forma, um serviço também pode ser considerado um componente, porém com algumas peculiaridades, como ser acessível pela Internet e expor operações relacionadas a funcionalidades do negócio [Hew09].

Como muitos dos trabalhos sobre implantação de componentes são diretamente aplicáveis na implantação de serviços, tratamos os termos "componente" e "serviço" como sinônimos, assim como faz Fowler [Fow04]. Neste trabalho utilizamos também os termos "serviço" e "serviço web" de forma equivalente, assim como feito por outros autores [WFK+06]. Damos preferência ao termo "serviço web" para evitar os significados mais gerais que a palavra "serviço" pode assumir. Exceção pode haver ao descrever trabalhos de terceiros que utilizam o termo "serviço" com algum significado mais amplo que o de "serviço web".

Um ponto de acesso (*endpoint*) de um serviço web é uma entidade referenciável para a qual se envia mensagens construídas de acordo com a especificação do serviço [W3C04a]. Um ponto de acesso é referenciado por uma URI (*Uniform Resource Identifier*). Uma URI é uma sequência

de caracteres que identifica um recurso, sendo que pode também ser chamada de URL (*Uniform Resource Locator*) quando fornece o acesso ao recurso [Gro05]. Assim como Smith e Murray [SM10], em geral também utilizamos a palavra serviço como simplificação para o ponto de acesso do serviço.

Por questões de desempenho e disponibilidade, um serviço pode ter várias *instâncias*, ou *réplicas*, em execução. Cada réplica possui seu próprio ponto de acesso, mas normalmente um conjunto de réplicas é apresentado ao mundo por meio de uma única URI. Essa única URI aponta para um balanceador de carga que conhece as URIs de cada uma das réplicas e distribui as requisições entre essas réplicas.

Os padrões mais utilizados atualmente para a implementação e acesso de serviços são *SOAP* e *REST* Fabio ►*colocar refs para soap e rest*◄ . Os serviços SOAP utilizam um conjunto específico de protocolos definidos pela W3C. As mensagens trocadas pelos serviços SOAP possuem uma estrutura (envelope) encapsulada em mensagens HTTP, protocolo utilizado como um meio de transporte. Já os serviços REST utilizam o HTTP como protocolo de aplicação, utilizando assim diretamente os princípios arquiteturais que são utilizados para explicar a alta escalabilidade do protocolo HTTP e da própria World Wide Web [PZL08].

O W3C chama os serviços SOAP como "serviços web", fornecendo a seguinte definição: "serviços web possibilitam a comunicação interoperável entre máquinas pela rede, utilizando padrões abertos para a troca de mensagens e descrição da interface dos serviços [W3C04b]". Na prática, a única diferença dos serviços REST para essa definição é que em REST não se exige a descrição do serviço em linguagem legível por máquina, embora isso seja possível com a WADL [Had06]. Além disso, nessa definição da W3C também poderiam ser enquadradas outras tecnologias como CORBA [OMG95].

Serviços SOAP descrevem suas interfaces com a Web Service Descritption Language (WSDL), interagem entre si pela troca de mensagens SOAP e são publicados e descobertos em repositórios UDDI. Uma interface de um serviço web descrita em WSDL é um arquivo XML com uma estrutura padronizada, o que possibilita a outros sistemas analisarem as possíveis formas de interação com esse serviço. Mensagens SOAP também são estruturadas em XML, sendo normalmente enviadas no corpo de requisições e respostas HTTP. O envelope de uma mensagem SOAP codifica a requisição ou resposta à operação de um serviço web, descrevendo também os tipos de dados e valores envolvidos na operação.

Além dos padrões mencionados (WSDL, SOAP, UDDI), há vários outros padrões que formam o conjunto chamado de WS-*, que inclui especificações para a realização de transações entre serviços, troca de endereços de serviços, composição de processos de negócios e muitos outros. O uso desse conjunto de padrões de forma integrada relaciona-se com a criação de Arquiteturas Orientadas a Serviços (SOA). De acordo com Papazoglou [PTDL07], SOA é uma forma de projetar sistemas que forneçam serviços com interfaces publicadas que possam ser descobertas, de modo que funcionalidades da aplicação sejam reutilizáveis como serviços por outras aplicações ou serviços em um ambiente distribuído.

Serviços REST utilizam como *interface uniforme* os métodos do protocolo HTTP (GET, POST, PUT e DELETE) e comunicam-se fazendo uso do protocolo HTTP como protocolo de aplicação para a troca de *representações* de *recursos*. Recursos são entidades do domínio do negócio que são de interesse dos clientes, e são identificados por URIs. Por exemplo, a URI http://livraria.com/livros/2 identifica o recurso "livro com ID 2". As representações dos recursos não estão presas a um formato de troca de mensagens, pois em cada mensagem o formato é descrito por um tipo MIME (p.ex: xml, json, png, txt). Os tipos mais comuns de representação de dados são JSON e XML. A Listagem 2.1 mostra, como exemplo, a representação JSON do recurso /livros/2.

```
1 {
2   "nome" : "Continuous Delivery",
3   "autores" : "Jez Humble and David Farley",
4   "editora" : "Addison-Wesly",
5   "ano" : "2011"
6 }
```

Listing 2.1: *Representação JSON do recurso /livros/2.*

As operações REST são definidas em função dos conceitos da interface uniforme, recursos e representações. Assim, um exemplo de operação REST é o cadastro de um novo livro, que é implementada como uma requisição HTTP do tipo POST para a URI http://livraria.com/livros, com a representação do livro no corpo da requisição. Como resposta, o servidor retorna um *código de status*[1] que informa o resultado da operação. Informações adicionais também podem ser transmitidas pelos cabeçalhos da resposta HTTP. Em nosso exemplo, esperamos o código 201 para indicar o sucesso da criação do novo recurso, além do cabeçalho *location* contendo a URI desse recurso recém criado. Diferentemente de SOAP, em REST não existe a noção de registro de serviços, pois a identificação de recursos por URIs e o uso de hyperlinks nas próprias mensagens REST possibilitam que os serviços necessários para a aplicação sejam encontrados [PZL08].

Segundo Pautasso [PZL08], serviços REST são considerados "mais simples" do que serviços SOAP porque fazem uso de padrões já bem estabelecidos, como HTTP, XML, URI e MIME, para os quais já existe uma ampla infraestrutura implantada Fabio ►*SOAP também faz uso de padrões já bem estabelecidos. Esta frase não parece fazer sentido para mim.*◄ . Um exemplo disso, é que é possível testar alguns serviços REST apenas com um navegador comum. Parte da escalabilidade atribuída a serviços REST também vem desse uso de padrões bem estabelecidos, pois os *caches* implementados pelos servidores web tornam-se automaticamente caches para serviços REST [lT10].

Apesar das vantagens apresentadas dos serviços REST, serviços que utilizam a tecnologia WS-* ainda são mais propensos a uma série de manipulações automatizadas que se tornam mais difíceis nos serviços REST, como por exemplo a geração automatizada de clientes para uma dada linguagem de programação. Isso se deve principalmente pelo alto nível de padronização da tecnologia WS-* Fabio ►*aqui você contradiz o que disse no parágrafo anterior.*◄ e pela existência de interfaces bem definidas e processáveis por software (WSDL).

2.2 Composições de serviços web

Serviços podem ser compostos para implementar processos de negócios mais sofisticados [PTDL07]. Processos de negócio são sequências bem definidas de passos computacionais executados de uma maneira coordenada [SABS02]. Sistemas de gerenciamento de workflow são a principal tecnologia para a implementação de processos de negócios [ADM00]. Um workflow é a automação, total ou parcial, de um processo de negócio, no qual documentos, informações ou tarefas são passados de um participante (humano ou não) para outros, de acordo com um conjunto de regras de procedimento [Wor99]. Segundo Casati et al. [CCPP98], workflows são compostos de *tarefas*, unidades de trabalho a serem desempenhadas por agentes humanos ou automatizados e *conectores*, que definem a ordem em que as tarefas devem ser executadas, o que também é denominado *fluxo de controle*. Sincronizações de execuções concorrentes também são especificadas por controladores chamados "*forks*" e "*joins*". Quando uma tarefa é desempenhada por um agente automatizado, o gerenciador de workflow normalmente realiza a invocação a um serviço web, que é esse agente automatizado que participa do processo de negócio. Um exemplo de linguagem para a criação de processos de negócio a partir da composição de serviços web é a WS-BPEL [OAS07].

O modelo de composição de serviços web que possui um coordenador central que coordena o fluxo de controle da composição é denominado *orquestração* [NCS04]. O coordenador central é chamado de *orquestrador*. No caso em que processos de negócios são executados por sistemas gerenciadores de workflows, o orquestrador é o próprio sistema de workflow. Outro modelo de composição de serviços web é o de *coreografia*, no qual o conhecimento sobre o fluxo de controle é distribuído entre os participantes, ou seja, cada serviço envolvido na composição sabe quando executar suas operações e com quais outros serviços interagir, sem que seja preciso um controle centralizado [BWR09].

Exemplos de linguagens e notações de descrição de coreografias são WSCI [W3C02], WS-CDL [W3C05] e BPMN2 [OMG11]. Essas linguagens e notações descrevem sequências e restrições nas trocas de mensagens efetuadas pelos participantes da coreografia sob uma perspectiva global. Essa descrição de interações sob uma perspectiva global é vista como um contrato de negócios

[1] http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

entre duas ou mais organizações [OMG11]. Essa ideia de contrato está presente também nos trabalhos sobre o arcabouço Open Knowledge [BPGR08], no qual serviços compartilham um *modelo de interação* que deve ser conhecido por todos os participantes da interação. Apesar da perspectiva global, como ressalta a especificação do BPMN2, uma coreografia não possui um controle de execução centralizado e participantes não compartilham um espaço de dados global. Dessa forma, um participante conhece o estado de outro participante apenas pela observação de seu comportamento externo, que consiste nas trocas de mensagens efetuadas [OMG11].

Embora a especificação de uma coreografia represente um modelo global de interação entre participantes, não é necessário que a implementação de cada participante tenha conhecimento do fluxo de negócio completo da coreografia, basta que ele tenha conhecimento de sua parte nesse fluxo. Assim, cada participante da coreografia pode ter o seu comportamento modelado por uma linguagem de orquestração. Dessa forma, uma coreografia pode também ser modelada como um conjunto de orquestrações distribuídas que interagem entre si, de forma que apenas os orquestradores precisam estar cientes de condições impostas pela coreografia [Pou11].

Um diagrama BPMN2 de coreografia especifica passos na execução da coreografia, que são denominados *atividades*, e que consistem na troca de mensagens entre *participantes* [OMG11]. Uma atividade pode ocorrer entre entidades participantes (p.ex: Magalhães Viagens realiza compra de passagem aérea da Nimbus Airline) ou entre *papéis* de participantes (p.ex: uma Agência de Viagem realiza compra de passagem de uma Companhia Aérea). Dizemos que dois serviços desempenham o mesmo papel se fornecem funcionalidades equivalentes. O BPMN2 distingue um dos participantes de uma atividade como o participante iniciador, que é aquele que envia a mensagem ao outro participante. O participante iniciador é também denominado cliente ou consumidor, enquanto o outro participante é denominado provedor. O diagrama BPMN2 da Figura 2.1 ilustra os elementos explicados em um exemplo de uma pequena coreografia com apenas dois serviços.

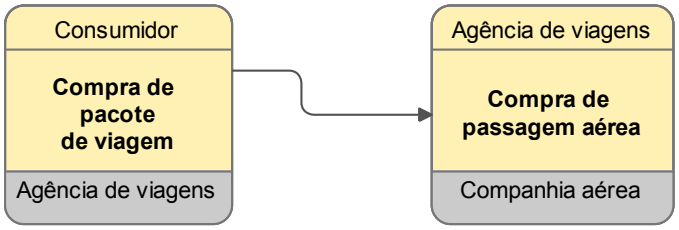

Figura 2.1: *Exemplo de uma pequena coreografia de serviços em notação BPMN2.*

Serviços podem ser projetados para participarem de uma determinada composição, mas também é possível que uma composição seja projetada para utilizar serviços já existentes. No segundo caso, é necessária a criação de serviços de coordenação (*coordenadores*) que fazem com que serviços já existentes, não-cientes da composição, comuniquem-se adequadamente [AdRdS+13].

Em artigos acadêmicos também é comum a modelagem de coreografias com notações mais formais, tais como álgebras de processos [RSF11], redes de Petri [CFN10] e autômatos [RWR06]. Essas notações possibilitam aos autores realizarem simulações e identificarem propriedades, como a verificação da consistência da evolução dinâmica de coreografias [CFN10].

Como uma orquestração é um caso particular de uma coreografia, neste trabalho utilizamos os termos "coreografia" e "composição de serviços" indistintamente. Além disso, para fins da atividade de implantação, utilizando o middleware por nós desenvolvido, não há diferença em implantar uma coreografia ou uma orquestração, uma vez que orquestradores ou eventuais coordenadores são implementados como serviços web.

2.3 O processo de implantação de sistemas

A "Especificação de implantação e configuração de aplicações distribuídas baseadas em componentes" (DEPL [OMG06]) é um padrão da OMG (Object Management Group). A implantação

é definida pelo DEPL como um *processo*, que se inicia após a aquisição de um componente, e vai até o momento em que o componente está em execução, pronto para processar chamadas. Embora o DEPL utilize o conceito de "componente", suas definições são aplicáveis e úteis ao contexto de implantação de serviços.

Os principais termos definidos no DEPL e utilizados neste trabalho são os seguintes:

Implantador: é a pessoa, ou organização, que é a "dona" do componente, e que será responsável pelo processo de implantação. Não é o software que propriamente realiza o processo de implantação.

Ambiente alvo: a máquina, ou conjunto de máquinas, onde os componentes serão implantados.

Nó: um recurso computacional onde se implanta um componente, como por exemplo uma máquina virtual; faz parte do ambiente alvo.

Pacote: artefato executável que contém o código binário do componente. É por meio do pacote que um serviço pode ser instalado e executado em um determinado sistema operacional. Existem pacotes dependentes de sistema operacional (p.ex: deb, rpm), e pacotes independentes de sistema operacional (p.ex: jar, war).

No caso de um processo de implantação automatizado, o *implantador* é o responsável por desenvolver os scripts de implantação. Em contrapartida, utilizamos o termo *desenvolvedor* para se referir ao desenvolvedor das composições de serviços web.

Ainda segundo o DEPL, o processo de implantação é composto pelas seguintes fases:

Instalação: o implantador transfere o componente adquirido para sua própria infraestrutura; a instalação está relacionada ao processo de aquisição do componente, e não se trata de mover o componente para o ambiente alvo, no qual será executado. Consideramos, portanto, que essa fase normalmente não se aplica à implantação de serviços, pois normalmente o serviço é implantado pela própria organização que o desenvolveu.

Configuração: edição de arquivos de configuração para alterar o comportamento do software; o código compilado do componente junto de sua configuração são os insumos para a produção do pacote do componente.

Planejamento: resulta em um *plano de implantação*, que mapeia como os componentes serão distribuídos pelos nós do ambiente alvo.

Preparação: procedimentos no ambiente alvo para preparar a execução do componente. Envolve configurações do sistema operacional, instalação de middlewares (p.ex. Tomcat), e a transferência do componente para a máquina onde será executado.

Inicialização: é quando finalmente o componente é iniciado e entra em execução, podendo processar chamadas de seus clientes. A inicialização também inclui a ligação entre os componentes de uma composição, para que os componentes conheçam a localização dos componentes dos quais dependem.

Todo o processo que vai desde o *commit* do código-fonte até a implantação em produção chamaremos de processo de *lançamento* de uma determinada versão do sistema. Esse processo de lançamento pode ser automatizado por um "*pipeline de implantação*" [HF11], no qual o sistema passa por uma sequência de etapas, sendo que em cada etapa um aspecto do sistema é testado. A cada etapa, mais confiança se tem sobre o sistema a ser lançado. Vencidas todas as etapas, o sistema pode ser implantado no ambiente de produção, ou em alguns casos em um ambiente de homologação. Um exemplo básico de pipeline de implantação pode ser visto na Figura 2.2. Para cada etapa do pipeline, o processo de implantação deve ser executado de forma automatizada.

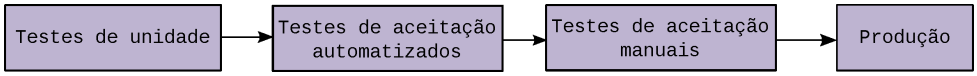

Figura 2.2: *Exemplo básico de pipeline de implantação.*

Profissionais da acadêmia e da indústria levantam a necessidade de se automatizar o processo de implantação, uma vez que o processo de implantação manual se torna moroso e propenso a erros, principalmente na implantação de sistemas distribuídos [HF11, DBV05]. Humble e Farley [HF11] afirmam que o processo de implantação manual faz com que o lançamento de uma nova versão do sistema se torne um grande evento nas organizações, em que há muita tensão e que faz as pessoas trabalharem até mais tarde. A solução para esses sintomas, segundo os autores, é a automação do processo de implantação. Em um processo de implantação automatizado tudo o que for possível é executado de forma automatizada, geralmente por meio de scripts. O objetivo de um processo de implantação automatizado é proporcionar um processo de implantação *reprodutível*, *confiável* e *fácil* [HF11].

A automação discutida nos trabalhos de Humble afeta principalmente as fases de preparação e inicialização do modelo de implantação do DEPL. A automação dessas fases normalmente é realizada com a escrita de scripts, com ou sem ferramentas específicas. Mas há também muitos trabalhos acadêmicos sobre a fase de preparação, envolvendo a escolha automática da máquina alvo de um componente baseado em seus requisitos não-funcionais. Por fim, não discutimos a automação da fase de configuração, por considerar que os pacotes fornecidos ao processo de implantação já estão configurados. Exemplo de configuração são credenciais de acesso ao banco de dados.

Um processo de implantação pode ser automatizado de várias maneiras. Pode-se utilizar linguagens de script de propósito geral (Python, shell script), ferramentas gerais voltados para o processo de implantação (p.ex: Chef[2], Capistrano[3]) ou sistemas de middleware especializados em determinados tipos de artefatos implantáveis, entre os quais se enquadram as soluções de Plataforma como um Serviço, sobre as quais discutimos na Seção 2.4. Humble e Farley recomendam a utilização de sistemas especializados, preterindo a utilização de linguagens de scripts de propósito geral.

A concretização de um processo de implantação automatizado depende bastante da integração de diferentes papéis em uma organização, principalmente da integração entre desenvolvedores e operadores, uma vez que o desenvolvimento dos scripts de implantação requer habilidades de ambos os perfis. Essa percepção levou à criação do conceito de uma cultura denominada DevOps [HM11], na qual times inter-funcionais trabalham para a concretização da implantação automatizada.

A discussão a seguir sobre as vantagens do processo de implantação automatizado são baseadas no livro "*Continuous Delivery*" [HF11].

Muitos problemas na implantação manual se dão por causa de documentação incompleta, contendo pressupostos não compartilhados por todo o time responsável por um produto ou serviço. Dessa forma é comum que a organização fique bastante dependente de uma única pessoa para realizar a tarefa de implantação. Por outro lado, um script de implantação é uma documentação completa e precisa de todos os passos do processo. Caso um script fique desatualizado, o impacto será imediato, pois não será possível implantar o sistema. Dessa forma, na prática, dificilmente tais scripts estarão desatualizados, diferentemente do que ocorre com a documentação convencional.

A facilidade de se implantar o sistema com um simples comando leva a sua utilização contínua por diferentes atores. O time de desenvolvimento estará constantemente utilizando esse script para realizar testes de integração e aceitação. Essa execução contínua do processo de implantação nos testes trará os seguintes benefícios:

- Os testes se tornam mais confiáveis por serem executados em um ambiente garantidamente similar ao ambiente de produção.

[2] http://www.getchef.com/
[3] https://github.com/capistrano/capistrano

- A quantidade de execuções de testes de integração e aceitação será maior, o que auxilia na garantia de qualidade do sistema.
- O próprio processo de implantação se torna mais confiável, pois quando o "grande dia" da implantação chega, o processo já terá sido executado várias vezes.
- Em particular, é de se esperar que defeitos no script de implantação já tenham sido detectados e corrigidos.

A utilização da implantação automatizada na execução de testes também facilita a execução concorrente de múltiplos testes em ambientes isolados. Isso, por sua vez, contribui para o aumento da bateria de testes, fazendo com que a cobertura dos testes aumente e, por fim, a própria qualidade do sistema testado também melhore.

Outro problema na implantação manual é que quando o sistema finalmente é testado no ambiente de produção, grandes mudanças arquiteturais podem ser economicamente inviáveis. A implantação automatizada favorece a prática da implantação contínua desde as versões embrionárias do sistema, ajudando a garantir que decisões arquiteturais são adequadas e evitando alterações de última hora para adequar o sistema ao ambiente de produção.

A implantação contínua e confiável do sistema é um fator determinante de apoio ao lançamento contínuo de novas versões. Isso é importante para que se consiga o *feedback* do cliente o quanto antes sobre as últimas alterações no sistema. Esse *feedback* é importante tanto do ponto de vista técnico para o aprimoramento do sistema, quanto do ponto de vista de negócio, pois é ele que ajuda o time a saber se está construindo *a coisa certa*. O encurtamento do tempo entre desenvolvimento e *feedback* do cliente é uma prática pregada pelo movimento *lean startup* [Rie11].

Na próxima seção falamos sobre a computação em nuvem, moderna tecnologia que impacta altamente as técnicas de implantação de sistemas.

Gerosa ▶*esta seção em geral não está tão boa quanto as outras. rever o texto.*◀

2.4 Computação em nuvem

O Instituto Nacional de Padrões e Tecnologias dos Estados Unidos (NIST) define computação em nuvem como um "modelo para possibilitar acesso ubíquo, conveniente e sob demanda pela rede a um conjunto compartilhado de recursos computacionais (p.ex. redes, servidores, discos, aplicações e serviços) que possam ser rapidamente provisionados e liberados com o mínimo de esforço gerencial ou interação com o provedor do serviço" [MG11].

Zhang et al. [ZCB10] destacam as seguintes características da computação em nuvem: i) separação de responsabilidades entre o dono da infraestrutura de nuvem e o dono do serviço implantado na nuvem; ii) compartilhamento de recursos (serviços de diferentes organizações hospedados na mesma máquina, por exemplo); iii) geodistribuição e acesso aos recursos pela Internet; iv) orientação a serviço como modelo de negócio; v) provisionamento dinâmico de recursos; vi) cobrança baseada no uso de recursos, de forma análoga à conta de eletricidade.

Os serviços de computação em nuvem podem ser oferecidos a clientes internos ou externos à organização administradora da plataforma de nuvem. Uma nuvem é considerada pública quando os clientes são externos, como no caso da nuvem da Amazon; ou é considerada privada quando os clientes são internos, situação na qual a organização pode utilizar ambientes baseados em um middleware como o OpenStack [ZCB10].

À computação em nuvem são atribuídos os seguintes modelos de negócio [ZCB10], ou modelos de serviço [MG11]: Infraestrutura como um Serviço (IaaS), Plataforma como um Serviço (PaaS) e Software como um Serviço (SaaS).

O modelo de Infraestrutura como Serviço (IaaS) fornece acesso aos recursos virtualizados, como máquinas virtuais, de forma programática. Um dos principais fornecedores de IaaS na época da escrita deste texto é a Amazon, com os serviços Amazon Web Services (AWS). Dentre os vários serviços fornecidos pela plataforma, destaca-se o EC2, que possibilita a criação e gerenciamento

de máquinas virtuais na nuvem da Amazon. Na utilização de IaaS, uma das considerações chaves é "tratar hospedeiros como efêmeros e dinâmicos" [TF12]. É preciso considerar que hospedeiros podem ficar indisponíveis e que nenhuma suposição pode ser feita sobre seus endereços IPs, o que requer um modelo de configuração flexível e que a inicialização do hospedeiro leve em conta essa natureza dinâmica da nuvem. Para que as aplicações sejam escaláveis e tolerantes a falhas, a Amazon recomenda mais do que a criação de máquinas virtuais com o serviço EC2: deve-se utilizar grupos de máquinas replicadas que compartilhem um balanceador de carga [TF12]. Conforme a demanda da aplicação cresce ou diminui, máquinas podem ser dinamicamente acrescentadas ou removidas desses grupos de replicação, o que proporciona escalabilidade horizontal à aplicação. Naturalmente, essa replicação depende de um prévio preparo da aplicação para esse cenário, pois se deve levar em conta a distribuição, replicação e particionamento dos dados.

O uso de recursos virtualizados, proporcionado pelo modelo IaaS, potencializa a automação do processo de implantação [HF11]. Novos ambientes são criados dinamicamente, em poucos minutos, com a configuração de um sistema operacional recém instalado em uma máquina. Isso traz as seguintes vantagens para o processo de implantação:

- Evita-se a burocracia e custos necessários para o provisionamento de novo hardware.
- A implantação se torna facilmente reprodutível no mesmo ambiente, não é preciso reinstalar o sistema operacional ou limpar as configurações do sistema para se obter uma nova implantação do serviço.
- Se executados em diferentes máquinas virtuais, dois serviços podem dividir um mesmo servidor físico sem que a implantação e execução de um serviço afete a execução do outro serviço anteriormente implantado.

Na utilização de serviços IaaS para a implantação de serviços há duas abordagens possíveis: 1) a máquina virtual deve ser criada com base em uma imagem[4] que já contenha o serviço implantado, ou 2) deve ser criada com base em uma imagem contendo apenas um sistema operacional recém instalado, de forma que a implantação do serviço seja feita por scripts. O modelo de imagem pronta proporciona implantações mais rápidas, porém a segunda abordagem é mais flexível, pois para implantar uma nova versão do sistema evita-se a publicação de uma nova imagem, o que é um processo demorado, já que imagens são arquivos com vários gigabytes. Um compromisso entre as duas abordagens também é possível: se todos os serviços implantados são WARs, por exemplo, então a imagem base pode conter não só o sistema operacional, mas também o ambiente de execução dos serviços, o Tomcat no caso.

No modelo de Plataforma como Serviço (PaaS), os desenvolvedores da aplicação não precisam preocupar-se diretamente com a gerência dos recursos virtualizados ou com a configuração dos ambientes nos quais a aplicação será implantada, concentrando-se no desenvolvimento do código da aplicação. Um exemplo típico de PaaS é o Google App Engine[5], que oferece implantação transparente a projetos em Python, Java ou Go. O App Engine também oferece escalabilidade automática de modo mais simples que os serviços de IaaS, uma vez que a configuração prévia e as alterações na infraestrutura ocorrem de modo totalmente transparente ao desenvolvedor da aplicação. Uma desvantagem presente nos serviços PaaS são as restrições de linguagens, bibliotecas e ambientes impostas aos desenvolvedores da aplicação.

Um exemplo de SaaS é o Google Docs ou qualquer outro aplicativo online que seja diretamente utilizado pelo usuário final. Uma das aplicações desse tipo é o armazenamento de dados na nuvem, como fornecido pelo Dropbox[6]. Uma confusão comum é definir o conceito de nuvem como se fosse estritamente ligado a esse tipo de serviço de armazenamento de dados.

[4] *Imagens* são sistemas de arquivo somente-leitura contendo um sistema operacional, aplicações e dados a serem instanciados em uma ou mais máquinas virtuais.

[5] https://developers.google.com/appengine/

[6] http://dropbox.com/

Com as vantagens aqui apresentadas, é cada vez mais comum o uso dos recursos de nuvem por empresas que desenvolvem software, pois assim seus esforços concentram-se no desenvolvimento do produto, aliviando as preocupações com infraestrutura. A computação em nuvem também possibilita que organizações evitem grandes investimentos antecipados em infraestrutura, pois os recursos virtualizados são dinamicamente acrescentados conforme a carga da aplicação requeira. Pode-se então considerar o uso da nuvem uma realidade do mercado de software atual. Dessa forma, é natural esperar que a implantação de composições de serviços também se dê no ambiente de computação em nuvem, que é a abordagem deste trabalho.

2.5 Desafios na implantação de sistemas de grande escala

Na visão proposta pelo Instituto de Engenharia de Software da Universidade Carnegie Mellon, sistemas de ultra grande escala são ultra grandes em relação a todas as dimensões possíveis: linhas de código, pessoas, dados, dispositivos, etc. [Sof06]. O número estimado de linhas de código desses sistemas é de bilhões. Para efeito de comparação, o núcleo do sistema operacional GNU/Linux possui cerca de 15 milhões de linhas de código em sua versão 3.2, a mais recente no momento da escrita deste texto [Lee12]. Com isso, talvez o único sistema da atualidade que se assemelha aos sistemas de escala ultra grande previstos é a Internet.

Por outro lado, a característica mais importante de um sistema de grande escala não é seu tamanho, mas o fato de ser caracterizado como um "ecossistema sociotécnico" [Sof06], em que pessoas são parte integrante do sistema, interagindo com diferentes objetivos, de modo decentralizado e independente, porém seguindo restrições impostas. A analogia proposta é de que o desenvolvimento dos atuais sistemas de grande escala equipara-se a construção de prédios, enquanto que o desenvolvimento de sistemas de escala ultra grande equivaleriam a construção de cidades, o que é naturalmente um processo contínuo e decentralizado.

Recentemente, há ainda a consolidação da computação em nuvem, que traz um conjunto de tecnologias e práticas que se relacionam com as três características de sistemas de escala ultra grande anteriormente mencionadas. Sistemas distribuídos estão migrando para ambientes de nuvem, onde são compostos e mantidos decentralizadamente por várias organizações [SPV12]. A virtualização, um dos aspectos centrais da computação em nuvem, é de grande auxílio no provisionamento de novos ambientes [HF11], o que é importante para o processo de implantação de sistemas. A virtualização também facilita a criação de ambientes replicados, arquitetura importante para tratar falhas individuais de componentes.

A grande escala afeta os processos envolvidos no ciclo de vida dos sistemas. Estudando a literatura que aborda e discute desafios, princípios e práticas de sistemas de grande escala, identificamos os seguintes desafios que essa nova realidade traz ao processo de implantação de sistemas:

Processo: Como já foi discutido neste capítulo, a automação do processo de implantação vem se firmando como uma tendência crucial na capacidade das equipes de TI entregarem valor o mais continuamente possível, evitando as dificuldades e problemas presentes no processo manual de implantação. Tais dificuldade e problemas se tornam muito mais complicados em ambientes distribuídos e de grande escala. Por isso, nesse caso a automação dos processos se torna ainda mais fundamental. Hamilton [Ham07] lista uma série de boas práticas acumuladas por anos de experiência no desenvolvimento de serviços de grande escala. Dentre elas, Hamilton destaca a automação de todos os processos de operações dos serviços, alegando que processos automatizados são testáveis, reparáveis e, portanto, mais confiáveis Gerosa ►*processos manuais tb são testáveis e reparáveis*◄ .

Falhas de terceiros: Sistemas distribuídos de grande escala devem esperar e tratar falhas de componentes de terceiros [Ham07, HC09, Sof06]. Mesmo se a chance de falhas de cada componente é pequena, a grande quantidade de componentes e interações aumenta as chances de falhas em algum lugar do sistema [Sof06]. Mais do que ser projetado para não falhar, um componente operando em um ambiente de grande escala deve ser projetado para tratar adequadamente

situações de exceção e indisponibilidade, tanto do próprio componente, quanto de outros componentes dos quais depende.

Um exemplo de falha típica em um processo de implantação automatizado utilizando um serviço de IaaS envolve o provisionamento de máquinas virtuais. Quando um novo nó é requisitado para o provedor de infraestrutura, há uma chance de que o provisionamento falhe. Além disso, alguns nós podem levar um tempo muito maior que a média para ficarem prontos. Outras operações que podem falhar durante o processo de implantação são conexões SSH e a execução de scripts nos nós alvos.

A Figura 2.3 mostra a distribuição por nós observada empiricamente do tempo de criação de VMs quando se requisita concorrentemente a criação de 100 nós na nuvem da Amazon (esse experimento foi repetido 10 vezes). Nós contamos o tempo que vai da requisição de criação do nó até o momento em que a VM se encontra apta a receber conexões SSH, que é quando ela se torna pronta para uso na prática. Nós observamos uma taxa de falha de 0.6% quando se tenta criar concorrentemente 100 nós. É interessante notar que o tempo de criação tem uma mediana estável, mas que alguns valores altos para esse tempo são esperados quando se cria ao mesmo tempo uma grande quantidade de nós. Em nossas observações, falhas e tempos longos de provisionamento afetaram até 7% das requisições de criação de nós. **Fabio** ►*tem que explicar a figura. a figura usa uma notação que não foi explicada. faça essa figura bem maior verticalmente para ficar mais legível.*◄

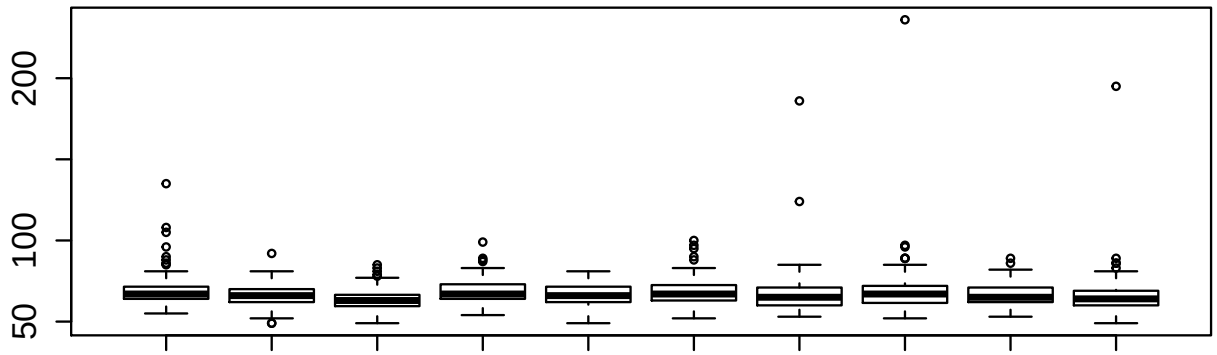

Figura 2.3: *Tempos de criação de instâncias EC2 observados, em segundos.*

Os dados utilizados para gerar a Figura 2.3 foram coletados em maio de 2013. As máquinas virtuais, do tipo *m1.small*, foram criadas na zona de disponibilidade *us-east-1b*.

Nygard [Nyg09] apresenta vários padrões de estabilidade que são de importante aplicação em sistemas de grande escala. Em sua essência, esses padrões dizem respeito a detectar falhas e evitar sua propagação, provendo um tratamento adequado a elas. Dentre as práticas recomendas pelo autor destacam-se 1) o uso de *timeouts* no cliente, que evita que um cliente fique eternamente esperando uma resposta; 2) a interrupção de tentativas do cliente quando há sintomas de indisponibilidade do provedor; 3) criação de recursos exclusivos para diferentes clientes, evitando que uma falha em um recurso compartilhado afete todos os clientes; e 4) a "falha rápida", que faz com que um provedor forneça uma resposta de erro tão logo quanto seja possível saber que a operação não terá sucesso. **Gerosa** ►*repetição de "cliente" no parágrafo*◄

Quando um sistema faz uma requisição a outro serviço, não é possível distinguir um *timeout* de uma resposta eventualmente mais lenta. Dessa forma, só é seguro, do ponto de vista funcional, o sistema cliente enviar uma nova requisição devido a *timeout* caso a operação considerada seja *idempotente*. Segundo Ramalingam [RV13], idempotência é um critério de correção pelo qual o sistema se compromete em tolerar requisições duplicadas, sendo importante para o tratamento eficiente de falhas de comunicação ou de processamento. **Fabio** ►*definição imprecisa para não dizer errada. Uma operação é idempotente quando aplicá-la várias vezes gera o mesmo resultado do que gerar uma única vez.*◄ Em interfaces REST, por exemplo, todas as operações que não sejam

POST devem ser idempotentes [All10]. A idempotência de scripts de implantação é um dos principais destaques dentre as funcionalidades do Chef[7].

Disponibilidade: Embora serviços em um sistema distribuído tenham que estar preparados para lidar com a falha de outros serviços do sistema, cada serviço deve ter sua disponibilidade aumentada tanto quanto possível. Para isso é preciso aplicar técnicas que aumentem o tempo médio entre falhas e/ou diminuam o tempo médio de reparo após uma falha.

O balanceamento de carga entre réplicas de um serviço é uma das práticas mais importantes e recomendados atualmente para aumentar a disponibilidade e escalabilidade de sistemas [TF12]. Com a replicação do serviço, a falha em uma réplica não implica na indisponibilidade do serviço. Além disso, com a utilização das tecnologias de nuvem, caso a quantidade de requisições aumente, pode-se requisitar um aumento na quantidade de réplicas, o que evita uma indisponibilidade por incapacidade de se atender a todas as requisições.

Outra prática importante para o aumento da disponibilidade é a replicação de dados [Bre01]. No entanto, a replicação síncrona de dados é inviável para sistemas de grande escala [HC09]. O Teorema CAP [Bre12] prevê que um sistema não mantém os níveis de consistência e de disponibilidade na presença de particionamentos de rede. Considerando que particionamentos de rede são intrínsecos ao ambiente da Internet, o aumento no tamanho dos sistemas inviabilizou uma consistência total com tempo de resposta satisfatório. Essa mudança representou uma quebra de paradigma na área de bancos de dados, pois agora os bancos de dados projetados para fornecer as propriedades ACID, que garantem consistência total, cedem lugar aos cada vez mais populares bancos de dados não-relacionais (NoSQL). Essa nova categoria sacrifica a consistência dos dados para obter maior disponibilidade ou escalabilidade [Cat11].

O processo de implantação deve considerar as necessidades de replicação de serviços e dados, para que ele possa configurar adequadamente as múltiplas instâncias dos serviços e das bases de dados. Deve ser possível também alterar em tempo de execução a quantidade de réplicas para a adequação à demanda observada.

Um processo utilizado para reduzir drasticamente o tempo de reparo após uma falha é o "*roll-back*" automatizado do sistema: se o ambiente de produção encontra-se em algum estado inválido, é feito uma reversão rápida e segura do sistema e do ambiente para o último estado estável [Ham07, Bre01]. Nygard [Nyg09] advoga que em caso de falha no sistema a prioridade deve ser a reversão imediata do sistema para a sua última versão estável, deixando para depois as investigações sobre as razões do problema, mesmo que a reversão custe a perda de eventuais pistas para o diagnóstico.

Escalabilidade: Quando se implanta uma grande quantidade de serviços em um ambiente distribuído, não é desejável que as implantações dos diferentes serviços sejam sequenciais. Uma vez que a implantação de diferentes serviços são tarefas independentes, implantá-los concorrentemente aumenta drasticamente a escalabilidade do processo de implantação da composição.

Uma arquitetura é perfeitamente escalável se ela continua a apresentar o mesmo desempenho por recurso, mesmo que usado em um problema de tamanho maior, conforme o número de recursos aumenta [Qui94]. No contexto de implantação, isso significa que, idealmente, o tempo de implantação deveria permanecer constante quando há um aumento proporcional no número de serviços a serem implantados e no número de nós alvos.

O número de serviços a ser implantado aumenta em duas situações: 1) quando se implanta composições maiores e 2) quando se implanta mais composições simultaneamente. A primeira situação ocorre na implantação de sistemas de grande escala. A segunda situação ocorre, por exemplo, quando se executa uma bateria de testes de aceitação de uma composição de serviços. Nesse caso um teste de aceitação pode levar um tempo considerável, já que engloba o provisionamento de um novo nó e a preparação do sistema. Em tal situação, é desejável

[7] http://docs.opscode.com/chef_why.html

que testes de aceitação sejam executados em paralelo, o que requer implantação concorrente de múltiplas instâncias da mesma composição. Quanto maior a capacidade de paralelização desse processo, mais testes poderão ser admitidos na bateria de testes.

Heterogeneidade: Componentes de sistemas de grande escala normalmente são construídos com diferentes tecnologias e hospedados em diferentes tipos de ambientes. Um dos principais caminhos para viabilizar a coexistência dessa pletora tecnológica é a Arquitetura Orientada a Serviços, incluindo as composições de serviços web.

Embora serviços web tenha surgido para resolver os problemas de heterogeneidade entre sistemas e organizações, hoje em dia há mais de um mecanismo para implementar o conceito de serviços, principalmente SOAP e REST, além de outros. Portanto, dar suporte à heterogeneidade é importante para sistemas baseados em serviços. A falta de flexibilidade para a escolha de tecnologia para o desenvolvimento de serviços e o provedor de infraestrutura ocorre em muitas soluções PaaS atualmente disponíveis.

Múltiplas organizações: Sistemas de grande escala não possuem um único dono [SPV12], sendo que seus componentes pertencem a diferentes organizações que interagem de forma coordenada. O conceito de coreografias de serviços web e notações como o BPMN surgem para formalizar a interação em tempo de execução entre serviços de organizações diferentes.

Em uma composição inter-organizacional a coordenação do processo de implantação se torna um desafio. Normalmente não se admite que um coordenador em uma organização possa tomar decisões sobre a implantação de serviços de outra organização, pois esse processo envolve custos, acesso à infraestrutura e acesso ao pacote do serviço. Dessa forma, não é possível o uso de um orquestrador para coordenar o processo de implantação. As organizações devem agir de forma colaborativa para que o processo de implantação da composição tenha sucesso. No entanto, isso não é tão simples, pois no caso de implantação simultânea, é preciso haver algum protocolo de comunicação para que uma organização receba por notificação os endereços de serviços recém implantados por outra organização, quando esses serviços são dependências de seus próprios serviços sendo também implantados.

Adaptabilidade: No futuro, sistemas deverão operar em um mundo altamente dinâmico, sendo preciso lidar com alterações imprevistas, como condições ambientais, incluindo desastres naturais, adequação legal, etc. [DNGM+08]. É de se esperar que em sistemas de grande escala a capacidade de agir autonomicamente seja vital para manter um funcionamento adequado, uma vez que a intervenção manual se torna mais custosa.

Quando requisitos funcionais ou não-funcionais são violados, as possíveis ações a serem tomadas são: 1) substituição de versão de serviços; 2) aumento na quantidade de réplicas de um serviço; e 3) migração da instância de um serviço para outro hospedeiro. Uma vez que todas essas ações tem relação com o processo de implantação, pode-se dizer que sistemas auto-adaptativos e autonômicos precisam estar cientes e ter pleno controle das atividades do processo de implantação.

Para tomar as decisões de adaptação, um sistema auto-adaptativo precisa monitorar a si próprio para coletar métricas a serem utilizadas em algum algoritmo adaptativo. Exemplo de métrica a ser coletada é a taxa de utilização de CPU no hospedeiro do serviço. Coletar tais métricas requer a utilização de uma infraestrutura de monitoramento que deve ser implantada na infraestrutura alvo. Portanto, o processo de implantação de sistemas auto-adaptativos também deve considerar a implantação de sistemas auxiliares que realizam esse monitoramento.

Capítulo 3

Trabalhos relacionados

Neste capítulo apresentamos os trabalhos relacionados à nossa pesquisa e algumas ferramentas referentes à implantação automatizada de serviços.

Um dos aspectos fundamentais de um processo de implantação automatizado é a linguagem de configuração utilizada para definir o processo de implantação. Essa linguagem pode ser procedimental [DBV05] ou declarativa [MK96, BBB+98]. Outro aspecto relevante é o escalonamento de recursos computacionais para os serviços a serem implantados, o que pode ser feito, por exemplo, com o auxílio de sistemas de computação em grade [WFK+06].

Antes de instalar um serviço é preciso configurar adequadamente o sistema operacional e a plataforma na qual o serviço será implantado. Para utilizar ferramentas como Chef[1], Capistrano[2] e Nix [DBV05], os usuários devem escrever scripts que realizem a configuração do ambiente e a implantação do serviço. No caso do Chef, um script configura a máquina na qual o serviço é instalado, enquanto que o Capistrano possibilita a coordenação da implantação de serviços em diferentes nós. Com as expressões do Nix, é possível também unificar a especificação da implantação com o *build* da aplicação em um único script, possibilitando a edição parametrizada de arquivos de configuração da aplicação em função do local de implantação.

ToDo ►*taktuk*◄

A abordagem procedimental, com scripts, fornece uma grande flexibilidade para especificar a implantação de sistemas, mas normalmente requer especialização de seus usuários, pois todos os detalhes do processo devem ser especificados. Wettinger et al. [WASL13] afirmam que ferramentas como Chef são usadas para a criação de planos de implantação específicos para cada aplicação, promovendo pouca reusabilidade. Esses scripts de implantação também deveriam ser desenvolvidos com o mesmo rigor do código da aplicação, inclusive com o uso de testes automatizados [HF11]. O descumprimento dessa recomendação torna o processo de implantação pouco robusto e até mesmo não confiável. Uma alternativa que evita essa sobrecarga no processo de desenvolvimento é o uso de sistemas especializados na implantação de determinados tipos de aplicações e que recebam, como entrada, uma simples especificação declarativa do sistema a ser implantado.

Um exemplo de abordagem declarativa é o uso de Linguagens de Descrição Arquitetural (ADLs), como a Darwin [MK96]. ADLs são uma evolução do conceito de Linguagens de Interconexão de Módulo (MILs) [DK76], que descrevem a interconexão entre módulos de um sistema. A motivação dos autores da MIL era contribuir com novas formas de se produzir software de grande porte, diferenciando essa atividade da programação de pequenos algoritmos. De forma similar, a linguagem Darwin concentra-se nos aspectos estruturais de sistemas distribuídos, descrevendo a conexão entre os módulos do sistema, mas sem descrever implementações ou sequências de interações entre os módulos. Em nosso trabalho, também descrevemos o sistema a ser implantado por meio de sua descrição estrutural, uma vez que é esse o aspecto necessário para que se possa automatizar o processo de implantação. Magee e Kramer demonstraram a utilidade prática da linguagem Darwin ao utilizá-la de forma integrada a componentes CORBA [MTK97], padrão de interoperabilidade de sistemas

[1] http://www.opscode.com/chef
[2] https://github.com/capistrano

distribuídos dominante no mercado à época. Darwin possui também um ambiente de execução, Regis [MDK94], que realiza a implantação dos sistemas descritos em Darwin. Regis possui duas políticas de distribuição de programas por estações de trabalho. A primeira política é o mapeamento definido pelo usuário de forma estática, abordagem não apropriada para ambientes de computação em nuvem. A segunda opção de política é a alocação automática em função da carga na CPU das estações de trabalho, não havendo flexibilidade para a consideração de outros recursos, como espaço em disco ou memória, por exemplo. Uma similaridade entre Regis e o Enactment Engine desenvolvido em nossa pesquisa é o uso do middleware para o envio de mensagens contendo referências remotas dos componentes implantados para que eles possam estabelecer ligações dinâmicas entre si.

Olan [BBB+98] é um ambiente para a descrição, configuração e implantação de aplicações distribuídas em ambientes heterogêneos, e que também utiliza uma ADL própria. Baseando-se na entrada descrita na ADL, Olan gera scripts de Configuração de Máquina, que definem a execução do processo de implantação dos componentes no ambiente distribuído e o ajuste dos canais de comunicação entre esses componentes. A abordagem de gerar um script de configuração a partir de uma especificação declarativa é também implementada pelo Enactment Engine. A ADL de Olan também possibilita a especificação de restrições sobre a localização da implantação do componente, porém sem flexibilidade para a adoção de estratégias dinâmicas de alocação de nós.

Apesar de os trabalhos sobre Darwin e Olan já falarem sobre software de "grande porte", o que se entendia por grande porte já se alterou significativamente desde a época em que esses trabalhos foram feitos. Uma evidência dessa diferente percepção de escala são os exemplos de aplicações fornecidos no artigo sobre Olan, em que se fala sobre componentes muito granulares, como pedaços de interfaces gráficas, e que não consideram possíveis falhas de comunicação que são comuns na Internet. Além disso, os próprios autores do artigo sobre Olan admitem que não se preocuparam com questões de desempenho. Hoje, há novos desafios e requisitos que precisam ser considerados na construção software de grande escala, inclusive no processo de implantação, conforme visto na Seção 2.5.

O trabalho de Akkerman et al. [ATK05] concentra-se na implantação distribuída de componentes da plataforma J2EE, oferecendo ligações entre os componentes e suas dependências, especificados por uma ADL, e replicação dos componentes para fins de escalabilidade. No entanto, a solução apresentada para o gerenciamento do processo de implantação baseia-se numa aplicação de interface gráfica, o que dificulta a automação completa do processo. Outros trabalhos, como o de Lan et al. [LHM+05], também tratam o processo de implantação como realizado manualmente por um operador humano, enquanto que nosso objetivo é de que o operador inicie o processo de implantação com apenas um comando, conforme advogado por Humble e Farley [HF11].

O estudo de Quéma et al. [QBB+04] é o único encontrado a realizar avaliações empíricas sobre desempenho e escalabilidade do processo de implantação de componentes, além de oferecer tolerância a falhas no processo de implantação. Os autores apresentam uma solução na qual agentes executam de forma distribuída o processo de implantação, comunicando-se de forma assíncrona e hierárquica conforme a estrutura da composição de componentes sendo implantada, que é descrita por uma ADL. Os agentes também possuem propriedades transacionais que garantem a tolerância a falhas do processo de implantação, mas isso não é avaliado no texto. Os autores avaliam o desempenho e escalabilidade do processo de implantação variando a quantidade de componentes, a topologia da composição de componentes e a quantidade de máquinas. O resultado é um crescimento linear no tempo de implantação quando se aumenta na mesma proporção o número de máquinas disponíveis (o ideal seria que o tempo fosse constante). Os autores explicam que há uma sobrecarga na manutenção das sessões de comunicação entre os agentes, o que impede que o número de agentes seja muito grande.

A principal limitação do trabalho de Quéma et al. é a restrição de que a composição de componentes deve se organizar em uma estrutura hierárquica. Essa estrutura hierárquica, no entanto, é apenas um caso particular das possibilidades na topologia de uma coreografia de serviços, sendo que nossa solução, o CHOReOS Enactment Engine, não impõe essa restrição. Além disso, o ambiente

utilizado para a implantação no trabalho de Quéma et al. é um aglomerado, enquanto que nosso estudo é realizado em ambientes de nuvem.

Os trabalhos anteriores apresentam abordagens simples para o problema da distribuição dos componentes implantados pelas máquinas disponíveis. Já o trabalho de Watson et al., apresenta uma abordagem mais completa para esse problema com o uso de grades computacionais [WFK+06]. O foco dessa solução está em escolher dinamicamente o provedor de infraestrutura e a máquina em que um serviço web deve ser implantado considerando os requisitos não-funcionais do serviço web. Isso é realizado não somente para a primeira implantação do serviço web, mas também para as replicações que ocorrem quando as instâncias existentes não conseguem mais atender aos requisitos não-funcionais. Uma desvantagem dessa abordagem é a carga adicional gerada pela análise dos requisitos não-funcionais a cada troca de mensagens efetuada pelos serviços implantados.

Outro trabalho sobre implantação de componentes em um ambiente de grade é o de Lacour et al. [LPP04], no qual a escolha do nó de implantação é feita dinamicamente de acordo com alguns requisitos do componente. Uma desvantagem desse trabalho é o desenvolvimento específico para componentes CORBA, além de não haver preocupação com falhas no sistema distribuído.

Embora os trabalhos de Watson et al. e Lacour et al. avancem na problemática da distribuição dos serviços, nenhum dos trabalhos analisados considera as potencialidades e desafios dos ambientes de computação em nuvem [TF12], que oferecem serviços de infraestrutura para a gerência de recursos virtualizados. Portanto, em nossa pesquisa, procuramos dar um passo além ao explorar como o ambiente de computação em nuvem pode trazer benefícios ao processo de implantação, bem como ao considerar as restrições que esses ambientes impõem, como a falta de previsibilidade dos endereços das máquinas em tempo de configuração do serviço e as falhas da própria plataforma de nuvem.

ToDo ►*kadeploy, grid 5000*◄

Uma tendência recente para se atingir os objetivos de uma implantação simples, rápida, automatizada e escalável é a utilização de serviços de computação em nuvem que oferecem Plataforma como um Serviço (PaaS), que se encarregam não só da implantação da aplicação, como também do processo de criação e configuração do ambiente. O Cloud Foundry[3] é um PaaS de código aberto, podendo ser instalado na infraestrutura de uma organização para a oferta de serviços a clientes internos ou externos. O Cloud Foundry suporta uma grande diversidade de linguagens, arcabouços e bancos de dados a serem utilizados pela aplicação. Operadores do Cloud Foundry podem configurá-lo para utilizar diferentes provedores de Infraestrutura como um Serviço (IaaS), desacoplando as escolhas de IaaS e PaaS, o que é também adotado no Enactment Engine.

O Cloud Foundry tem como objetivo facilitar a implantação de aplicações web, e não a implantação de composições de serviços. Durante a implantação de uma aplicação pelo Cloud Foundry, o operador pode realizar ligações entre a aplicação e serviços tipicamente utilizados por aplicações web, como bancos de dados, que serão criados e configurados pela própria plataforma. Essa escolha deve ser feita dentro de um conjunto fechado de serviços oferecidos (MySQL, MongoDB, etc.). No entanto, ao implantar-se composições de serviços é preciso estabelecer também ligações entre os próprios serviços sendo implantados, cenário não considerado pelos atuais provedores de PaaS.

ToDo ►*outras ferramentas de cloud: vagrant, Amazon Simple Workflow Service, Force.com Visual Process Manager*◄

TOSCA (Topology and Orchestration Specification for Cloud Applications) é um padrão OASIS que utiliza a abordagem de orientação a modelos para o gerenciamento de recursos e serviços na nuvem [WBB+13]. Ao utilizar o TOSCA, seu usuário define um "modelo de serviço" (*service template*) para especificar, em alto nível, como os serviços são implantados e conectados a outros serviços. Contudo, artefatos de implementação ainda são necessários para implementar as operação definidas nos modelos. A ênfase dada nos trabalhos sobre o TOSCA é na portabilidade para que a implantação de um serviço possa utilizar diferentes componentes de middleware [WASL13] ou diferentes gerenciadores de configuração [WBB+13]. Essa abordagem torna o TOSCA um sistema altamente flexível e portável, mas obriga o desenvolvedor a definir os artefatos de implementação e a descrever como eles se relacionam às operações definidas no modelo. Com o CHOReOS Enactment

[3] http://www.cloudfoundry.com/

Engine, o ambiente de execução e a gerência de configuração são abstraídos de forma que os usuários não precisam se preocupar com os componentes de middleware utilizados para executar os serviços, e nem sequer precisam saber que o Chef é o gerenciados de configuração utilizado pelo EE.

Juju[4] é uma ferramenta de configuração e implantação de serviços criada pela Canonical. Os conceitos utilizados no Juju se assemelham muito ao TOSCA. "*Charms*" encapsulam as configurações da aplicação, definem como serviços são implantados, como serviços são conectados uns aos outros e como eles escalam. A cada operação definida para um serviço na *charm* também deve ser associado um artefato que implemente a operação, normalmente um *shell script*. Uma das limitações apontadas para o Juju é o fato de a ferramenta e suas *charms* serem altamente acopladas ao sistema operacional Ubuntu. Embora a versão atual do nosso CHOReOS Enactment Engine também utilize o Ubuntu como sistema operacional dos nós alvos, a utilização do Chef como gerenciador de configuração facilita a eventual utilização de outros sistemas operacionais, uma vez que as receitas Chef abstraem algumas peculiaridades do sistema operacional utilizado.

Um arcabouço voltado especificamente para a implantação e encenação de coreografias é o Open Knowledge [BPGR08, SDK+07]. Nesse arcabouço, o projetista da coreografia define o fluxo global de troca de mensagens entre os serviços em uma notação formal (*Lightweight Coordination Calculus*). A partir dessa descrição, o arcabouço gera *coordenadores* para cada participante, decentralizando a lógica de coordenação. Assim, o desenvolvedor do serviço implementa apenas a lógica de negócio, uma vez que a lógica de coordenação está desacoplada da implementação do serviço. O arcabouço Open Knowledge possui uma ênfase no problema da descoberta dinâmica de pares que satisfaçam os requisitos da interação projetada. Uma desvantagem, porém, é o forte acoplamento necessário na implementação dos serviços participantes ao arcabouço para que a lógica de coordenação possa ser fornecida ao serviço. Uma consequência desse forte acoplamento é que os serviços que utilizam o Open Knowledge devem necessariamente ser escritos em Java. Outra limitação, do ponto de vista da automação do processo de implantação, é que a infraestrutura do Open Knowledge deve já estar disponível antes da implantação dos serviços, pois a implantação é realizada nessa infraestrutura.

A Tabela 3.1 realiza uma comparação entre os estudos e ferramentas apresentados nesta seção em relação a características presentes em nossa solução, o Enactment Engine. Os símbolos na tabela possuem os seguintes significados: "✓" para "possui a característica", espaço vazio para "não possui a característica", "+/-" para "não possui a característica de forma totalmente satisfatória" e "-" para quando a característica não se aplica ou não foi possível determinar Gerosa ►*melhor separar o caso "não foi possível determinar", usando um "?" para isso*◄ . As características, que formam as colunas da tabela, são listadas a seguir:

Automatizado: processo de implantação automatizado;

ADL: especificação da implantação feita de forma simples e declarativa;

Escala: implantação escalável e capaz de lidar com os problemas típicos de sistemas de grande escala, principalmente com a falha de componentes de terceiros;

Composições: solução voltada para a implantação de composições de serviços, ou componentes; a principal implicação desse item é a realização da ligação entre os serviços implantados;

Nuvem: consideração das potencialidades e desafios trazidos por ambientes de computação em nuvem.

No próximo capítulo prosseguimos mostrando como o Enactment Engine implementa as características descritas. Também mostraremos os resultados de uma avaliação preliminar da escalabilidade do processo de implantação automatizado pelo Enactment Engine.

[4] https://juju.ubuntu.com/

Trabalho	*Automatizado*	*ADL*	*Escala*	*Composições*	*Nuvem*
[MDK94, MK96]	-	✓		✓	
[BBB+98]	-	✓		✓	
[QBB+04]	✓	✓	+/-	✓	
[ATK05]	+/-	✓	-	✓	-
[LPP04]	✓	-		✓	
[DBV05]	✓		-	-	
[WFK+06]	✓	-	+/-	✓	
Chef	✓		-	-	-
Capistrano	✓		-	-	-
Cloud Foundry	✓	-	✓		✓
Enactment Engine	✓	✓	✓	✓	✓

Tabela 3.1: *Tabela comparativa com os trabalhos relacionados*

Capítulo 4

Solução proposta: o Enactment Engine

O CHOReOS Enactment Engine (EE) é um middleware implementado no contexto deste trabalho. Uma vez instanciado, ele fornece serviços que automatizam a implantação de composições de serviços[1] em ambientes de computação em nuvem, funcionando no modelo denominado Plataforma como um Serviço (PaaS). O EE possui funcionalidades e características que foram projetadas para auxiliar o implantador de composições de grande escala.

Para utilizar o EE, o implantador, usuário do EE, descreve a composição a ser implantada na Linguagem de Descrição Arquitetural do EE, uma especificação de alto nível que diz *o que* deve ser implantado, e não o *como*. Finalmente, o usuário deve fornecer essa descrição ao EE por meio de sua API remota.

As funcionalidades fornecidas pelo Enactment Engine ao usuário são as seguintes:

- API para automatizar a implantação de composições de serviços em ambientes de computação em nuvem.
- Criação automatiza de infraestrutura virtualizada (nós na nuvem).
- Implantação escalável de coreografias de grande escala.
- Suporte a implantação multi-nuvem.
- Utilização de serviços de terceiros na composição a ser implantada.
- Implantação automatizada de infraestrutura de monitoramento dos recursos utilizados.
- Remoção automática de recursos da nuvem não utilizados.
- API para escalamento vertical e horizontal.

Para a implementação do arcabouço Enactment Engine contribuíram os alunos de pós-graduação Daniel Cuckier, Carlos Eduardo do Santos, Felipe Pontes, Alfonso Diaz, Nelson Lago, Paulo Moura, Thiago Furtado e demais colegas dos projetos Baile e CHOReOS. O Enactment Engine é software livre sob a Licença Pública Mozilla 2[2] e está disponível em http://ccsl.ime.usp.br/enactmentengine.

Neste capítulo, apresentamos a arquitetura e aspectos de implementação do Enactment Engine. Destacamos ao final do capítulo como as decisões arquiteturais e de implementação auxiliam o implantador a superar os desafios presentes na implantação de composições de grande escala. Alguns aspectos aqui discutidos são tratados em alto nível, priorizando o que é importante para o entendimento das contribuições acadêmicas deste trabalho. Detalhes técnicos sobre o middleware, principalmente do ponto de vista do usuário, são encontrados no CHOReOS Enactment Engine User Guide (Apêndice A).

[1] Como explicado na Seção 2.2, utilizamos os termos "composição de serviço" e "coreografia" indistintamente.
[2] http://www.mozilla.org/MPL/2.0/

4.1 Execução do Enactment Engine

O Enactment Engine é um sistema de middleware de código aberto que primeiramente deve ser instalado e configurado por um *administrador*. Uma vez em execução, a instância do EE fornece serviços que podem ser consumidos por algum sistema cliente, desenvolvido e operado pela figura do *implantador*. O administrador e o implantador podem pertencer à mesma organização, mas é possível que o administrador forneça o EE como um serviço (SaaS) a terceiros, cobrando por sua utilização. Para esses terceiros a vantagem seria evitar o trabalho de instalação e configuração do EE. O ambiente de execução do EE é exibido na Figura 4.1 e os componentes envolvidos são descritos a seguir.

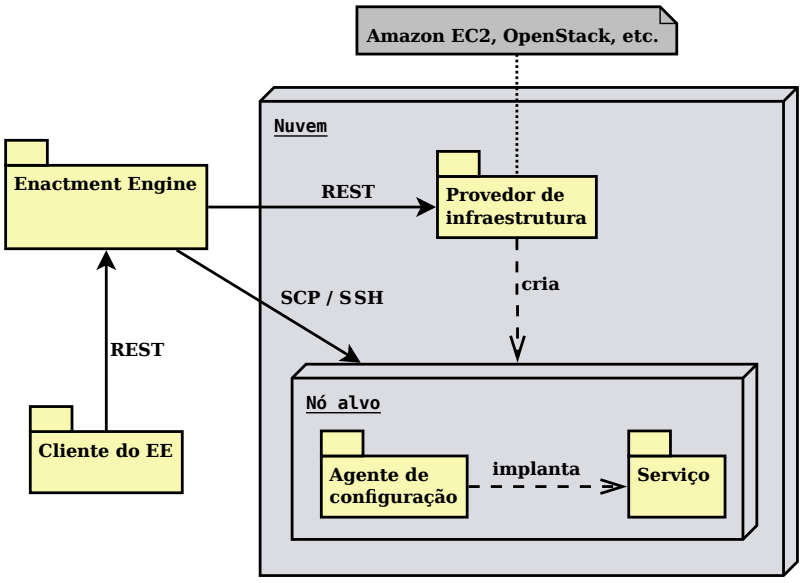

Figura 4.1: *Ambiente de execução do CHOReOS Enactment Engine.*

- O *provedor de infraestrutura* é um serviço capaz de criar e destruir máquinas virtuais (também chamadas de *nós*), normalmente em um ambiente de computação em nuvem. Atualmente o Enactment Engine oferece suporte para o Amazon EC2 e o OpenStack.
- O *agente de configuração* é executado nos nós alvos e dispara os scripts que implementam as fases de preparação e inicialização da implantação dos serviços[3]. O Enactment Engine utiliza o Chef Solo[4] como seu agente de configuração.
- O *cliente do Enactment Engine* é um programa ou script desenvolvido pelo implantador, no qual a especificação da composição de serviços é definida. Esse script deve enviar a especificação da composição para o Enactment Engine por meio das operações REST fornecidas pelo Enactment Engine. Uma opção para implementar essas chamadas é utilizar a biblioteca Java por nós fornecida, que abstrai os detalhes das chamadas REST.
- O *Enactment Engine* implanta os serviços de uma composição com base na especificação enviada pelo cliente. O processo implementado pelo Enactment Engine para efetuar a implantação é descrito na Figura 4.2, e explicado logo em seguida.

A Figura 4.2 exibe o processo de implantação de composições de serviços implementado pelo Enactment Engine:

[3] Sobre a nomenclatura das fases de implantação, ver a Seção 2.3.
[4] http://docs.opscode.com/chef_solo.html

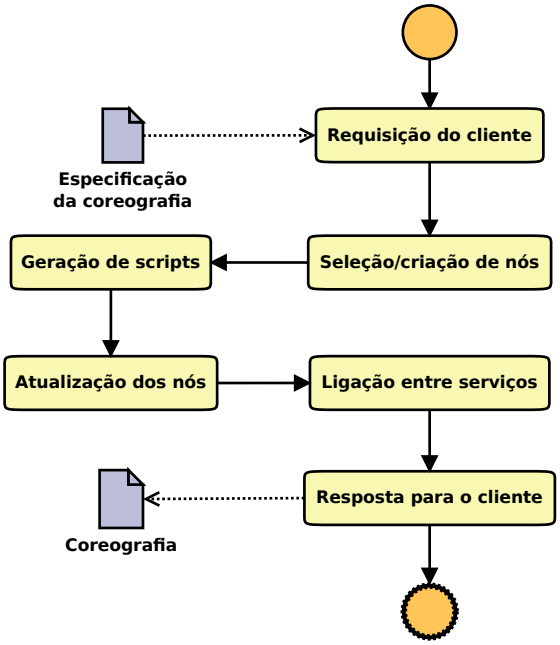

Figura 4.2: *Processo de implantação implementado pelo Enactment Engine.*

1. *Requisição do cliente:* o EE recebe a especificação da composição a ser implantada. O formato dessa especificação é descrito na Seção 4.2.

2. *Seleção/criação de nós*: para cada serviço especificado, o EE seleciona um ou mais nós onde o serviço será implantado (um serviço pode ter várias réplicas implantadas). Se preciso, o EE requisitará ao provedor de infraestrutura a criação de novos nós. Esse processo de seleção/criação de nós pode levar em conta os requisitos não-funcionais dos serviços a serem implantados. A política de seleção de nós é definida pelo administrador do EE, sendo que novas políticas podem ser criadas.

3. *Geração de scripts*: para cada serviço da composição, o EE gera dinamicamente os scripts de preparação do ambiente e inicialização do serviço. O EE acessa então o nó alvo selecionado para o serviço, e configura o agente de configuração desse nó para executar o script gerado.

4. *Atualização dos nós*: para cada nó alvo que receberá serviços da composição, o EE dispara a execução do agente de configuração, que por sua vez executa os scripts de preparação e inicialização dos serviços atribuídos ao nó. Dessa forma, os serviços entram em estado de execução na infraestrutura alvo.

5. *Ligação entre serviços*: após os serviços terem sido iniciados, para cada relação de dependência na coreografia (ex: serviço TravelAgency depende do serviço Airline), o EE fornece o endereço da dependência (ex: http://airline.com/ws) ao serviço dependente. Mais informações sobre o processo de ligação são fornecidas na Seção 4.3.

6. *Resposta para o cliente*: o EE responde ao seu cliente, informando em que nó cada serviço foi implantado e as URIs de acesso a cada serviço da composição. O formato da resposta é descrito na Seção 4.2.

Há também alguns outros passos opcionais que não descrevemos por estarem fora do escopo deste trabalho. Um exemplo é a implantação da infraestrutura de monitoramento dos nós alvos. O agente de monitoramento (Ganglia[5]) é implantado nos nós alvos pelo EE e coleta valores de uso de CPU, memória e disco dos nós.

[5] http://ganglia.sourceforge.net

4.2 Especificação da composição de serviços

O Enactment Engine recebe de seus clientes a especificação da composição na forma de uma descrição arquitetural com as informações necessárias e suficientes para que se possa realizar a implantação da composição. O EE também devolve ao seu cliente informações sobre o resultado da implantação, em especial as localizações de acesso aos serviços. As descrições da composição e de sua especificação são feitas com uma *linguagens de descrição arquitetural* (ADL), assim como a dos trabalhos vistos no Capítulo 3. A ADL do EE define a estrutura de classes apresentada na Figura 4.3. Em nossa implementação, representações de instâncias desse modelo são trocadas entre o EE e seu cliente em formato XML. A descrição detalhada de cada atributo e o *schema* XML da linguagem são apresentados no CHOReOS Enactment Engine User Guide.

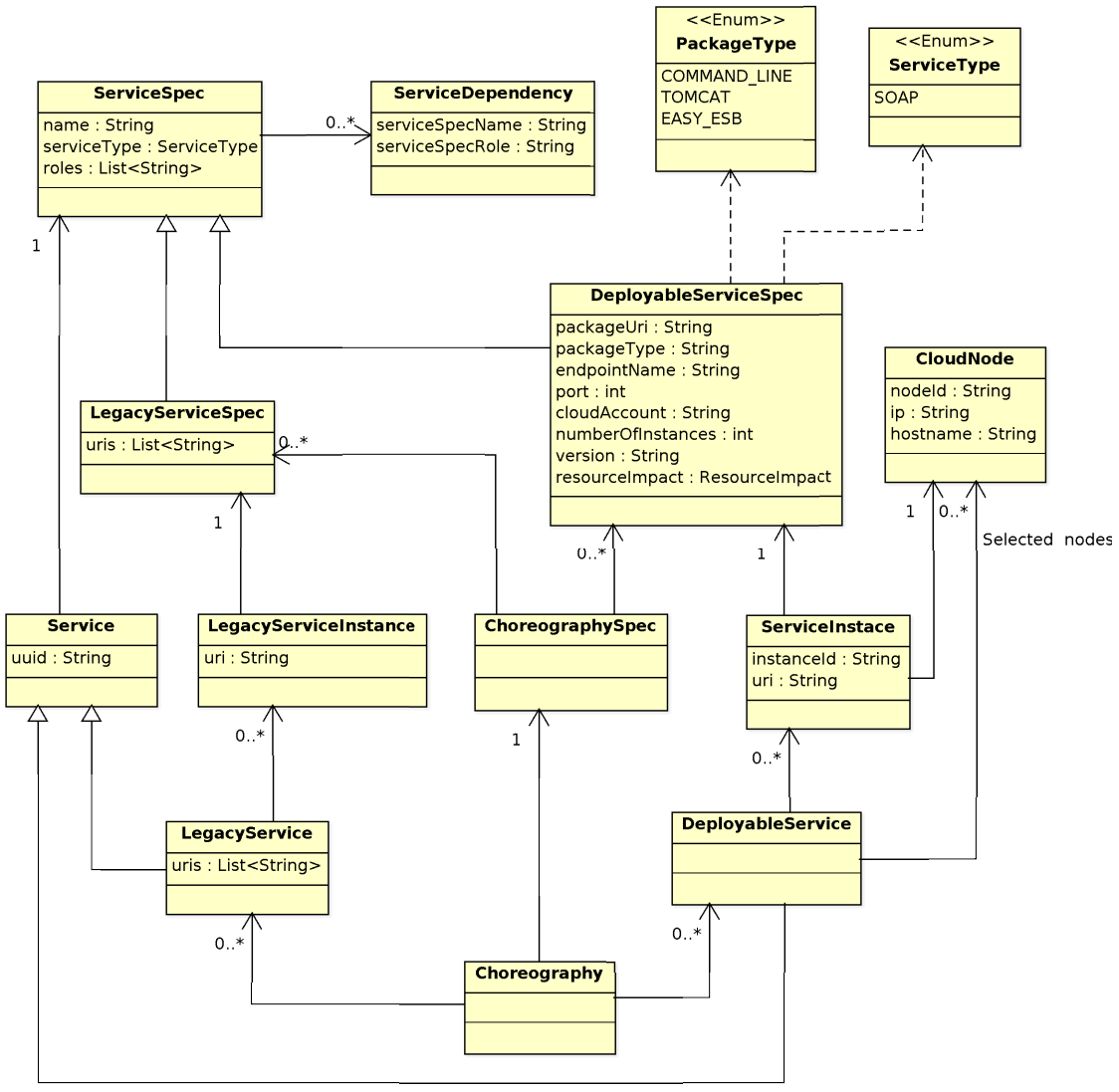

Figura 4.3: *Estrutura da descrição arquitetural de uma coreografia.*

A especificação da coreografia fornece todas as informações para a implantação da composição, possibilitando que o implantador descreva em alto-nível apenas *o que* deve ser implantado, e não os detalhes de implementação de *como* deve ser implantado. Assim, a escrita de uma especificação declarativa se contrapõe a escrita de um script, no qual normalmente se descreve os passos de *como* o sistema deve ser implantado.

Na ADL do EE, para cada serviço, especifica-se de onde o pacote do serviço pode ser baixado,

qual o tipo do pacote (WAR, JAR, etc.), quantas réplicas devem ser implantadas, etc. Pode-se especificar também a existência de serviços de terceiros que já estão disponíveis na Internet e que devem ser consumidos por serviços da composição.

O implantador pode escrever a especificação da coreografia diretamente em XML ou utilizando objetos Java (POJOs). A Listagem 4.1 apresenta um trecho da especificação escrita em Java, no qual um dos serviços participantes é definido, incluindo sua dependência de outro serviço participante.

```
1 airportBusCompanySpec =
2   new DeployableServiceSpec(AIRPORT_BUS_COMPANY, ServiceType.SOAP, PackageType.
      COMMAND_LINE, resourceImpact, serviceVersion, AIRPORT_BUS_COMPANY_JAR_URL,
      AIRPORT_BUS_COMPANY_PORT, AIRPORT_BUS_COMPANY, numberOfReplicas);
3 airportBusCompanySpec.setRoles(
4   Collections.singletonList(AIRPORT_BUS_COMPANY));
5 airportBusCompanySpec.addDependency(
6   new ServiceDependency(AIRPORT, AIRPORT_ENDPOINT));
```

Listing 4.1: *Trecho da especificação de uma coreografia.*

4.3 Ligação entre serviços

Em uma composição de serviços, alguns serviços se comunicam com outros serviços para implementar o fluxo de negócio. Quando um serviço A invoca um serviço B, dizemos que o serviço A depende do serviço B. Dizemos também que A é *dependente* de B, enquanto que B é *dependência* de A, ou ainda que A é *consumidor* de B, enquanto que B é *provedor* de A. Para que uma coreografia funcione, cada serviço precisa saber o endereço de suas dependências, e o processo pelo qual os serviços recebem os endereços de suas dependências é denominado *ligação*.

Segundo Dearle [Dea07], componentes podem ser ligados entre si em vários momentos: compilação, montagem, configuração e execução. Em nosso contexto, a ligação deve ser efetuada em tempo de execução, pois é somente nesse momento que os endereços completos dos serviços implantados estão disponíveis. Uma das possibilidades apontadas por Dearle para efetivação da ligação em tempo de execução é a utilização do padrão de injeção de dependência, conforme introduzido por Fowler [Fow04]. A injeção de dependências é utilizada em contêineres como o Springer[6], no qual o middleware passa ao componente referências de suas dependências. No entanto, Dearle ainda alega que há uma falta de arcabouços para a aplicação da injeção de dependência de forma distribuída.

A solução adotada no Enactment Engine para possibilitar a ligação entre serviços envolve a utilização do middleware para a passagem de endereços dos serviços implantados aos seus consumidores. Essa solução consiste na aplicação do padrão de injeção de dependência de forma distribuída, e é similar ao que foi feito nos trabalhos sobre a linguagem Darwin [MK96, MDK94]. Note que nessa solução, a "inteligência" em determinar quais serviços satisfazem as necessidades de outros serviços está na camada que produz a entrada do EE. As dependências entre os serviços são definidas na especificação da coreografia, pela lista de objetos ServiceDependency pertencentes a um ServiceSpec. Cada serviço na coreografia que possua dependências deve implementar uma operação denominada `setInvocationAddress`. Essa operação, por nós padronizada, recebe como argumentos as seguintes informações sobre a dependência:

Papel: é um nome associado a uma interface, ou seja, define as operações fornecidas por um serviço. A associação entre o nome e a interface deve ser previamente acordada pelas organizações participantes da coreografia e a implementação do serviço deve estar ciente dos nomes e interfaces de suas dependências.

Nome: é um nome que identifica univocamente o serviço no contexto de uma coreografia. Serve para que o serviço dependente possa diferenciar serviços com o mesmo papel. Exemplo: se um serviço de pesquisa de preços utiliza serviços do papel *supermercado*, ele utilizará o

[6] http://www.springer.org

nome do serviço para diferenciar os serviços de supermercados diferentes. Com essa semântica, o EE pode atualizar os endereços de um supermercado com uma nova chamada ao `setInvocationAddress`, sem que o serviço dependente considere que se trata de um novo supermercado.

Endereços: são as URIs das réplicas pelas quais pode-se acessar a dependência.

Assim, em uma coreografia em que, por exemplo, um serviço de agência de viagem dependa do serviço de uma companhia aérea, o EE executa a seguinte invocação ao serviço da agência de viagens : `setInvocationAddress('Companhia Aérea', 'Nimbus Airline', ['http://192.168.56.107:8080/nimbus/ws/' ])`.

A descrição fornecida até aqui é abstrata e independente de tecnologia. A definição exata da assinatura da operação deve ser definida de acordo com a tecnologia utilizada. A versão atual do EE já define essa assinatura para serviços SOAP. Para detalhes, ver o guia do usuário (Apêndice A).

Apesar dos benefícios da solução adotada no EE, Dearle [Dea07] também alerta sobre a desvantagem em forçar componentes a aderirem convenções de codificação impostas pelo middleware, o que poderia restringir o serviço a uma determinada linguagem de programação ou a algum middleware específico. Reconhecemos que esse problema existe em nossa solução, mas acreditamos que o desenho adotado ameniza os problemas levantados, pois tudo o que o serviço é obrigado a fazer é implementar a operação `setInvocationAddress` e conhecer os papéis de suas dependências, o que implica em conhecer a interface sintática de cada papel. Dessa forma, nossa solução não restringe o serviço a nenhuma linguagem e não impede a utilização do serviço em outro middleware.

4.4 Mapeamento dos serviços na infraestrutura alvo

Em algum momento do processo de implantação é preciso definir em que nó cada instância de serviço será hospedado. Chamamos de *mapeamento*, ou seleção de nós, essa fase do processo de implantação. Na forma mais simples de seleção de nó, o IP do nó alvo é definido estaticamente no script de implantação do serviço. O trabalho de Magee e Kramer [MTK97] apresenta a seleção de nós em função da utilização de CPUs nos nós existentes, não havendo possibilidade de utilização de outros critérios, como memória, disco, custo etc. Nos sistemas apresentados por Dolstra et al. [DBV05] e Balter et al. [BBB+98] é preciso que a distribuição dos serviços seja especificada com o uso dos IPs das máquinas nas quais os serviços devem ser implantados, o que não é possível em um ambiente de nuvem. Por fim, o *broker* apresentado por Watson et al. é o componente que mais se assemelha ao nosso NodeSelector, pois os autores deixam claro que várias implementações diferentes são possíveis, considerando-se diferentes tipo de requisitos e diferentes fontes de monitoramento. Como a escolha é feita em tempo de execução do serviço, seria também possível uma seleção que independa de IPs estabelecidos em tempo de projeto. No entanto, os autores não explicam como os usuários de seu sistema, os provedores de infraestrutura, deveriam proceder para criar seus próprios *brokers* personalizados.

Para avançar em relação às limitações dos trabalhos anteriormente citados, a seleção de nós no Enactment Engine considera os requisitos de dinamicidade do ambiente de nuvem, que nos impede de conhecer os IPs das máquinas em tempo de desenvolvimento ou configuração do script de implantação. O EE utiliza um seletor de nós automatizado que escolhe em tempo implantação os nós alvos para um dado serviço. A escolha de uma política ótima para o seletor é assunto de diversas pesquisas. Portanto, adotamos aqui uma abordagem extensível, com o fornecimento inicial de políticas como “sempre cria um novo nó” ou “cria nós até um limite, e depois faz rodízio entre eles”.

4.5 Interface do Enactment Engine

Os clientes do Enactment Engine utilizam suas funcionalidades por meio de uma API REST, que é descrita nesta seção. Por se tratar de uma API REST, o cliente pode ser implementado em

qualquer linguagem e ambiente que possua alguma biblioteca HTTP. Também disponibilizamos um cliente na forma de uma biblioteca na linguagem Java, tornando o uso do EE ainda mais simples para os usuários da linguagem Java, atualmente uma das mais utilizadas do mercado. Seguimos agora com uma descrição de alto nível de cada uma das operações disponíveis na API REST do EE. Detalhes da API, como os códigos de status HTTP retornados, são fornecidos no guia do usuário (Apêndice A).

Criar coreografia: registra a especificação de uma coreografia no EE. Essa especificação é a descrição arquitetural da coreografia, estruturada de acordo com a classe ChorSpec (Figura 4.3). Essa operação não realiza a implantação da coreografia.

Obter coreografia: obtém informações sobre uma coreografia registrada no EE. Essas informações referem-se à especificação da coreografia e ao estado da implantação de seus serviços, como os nós em que os serviços foram implantados, no caso de a implantação já ter sido realizada.

Implantar coreografia: realiza a implantação de uma coreografia já registrada no EE. Ao fim do processo, detalhes do resultado da implantação são retornados de forma estruturada de acordo com a classe Choreography (Figura 4.3). A implementação dessa operação deve possuir duas importantes propriedades: 1) a falha na implantação de parte da coreografia não deve interromper a implantação do resto da coreografia; 2) a operação deve ser *idempotente*, ou seja, uma nova requisição para a implantação da mesma coreografia não deve reimplantar os serviços já implantados, mas somente aqueles cujas implantações falharam na última execução. Para que serviços sejam atualizados, é preciso utilizar um novo valor no atributo "versão" da especificação do serviço.

Atualizar coreografia: registra uma nova versão de uma coreografia no EE. Os serviços atualizados na nova versão da coreografia devem possuir um novo número de versão em suas especificações. Essa operação, assim como a criação da coreografia, não implanta a nova coreografia. Para isso, é preciso invocar novamente a operação de implantação.

A atualização de serviços não é o foco de nosso trabalho. Dessa forma, em nosso trabalho a atualização dos serviços será feita da forma mais simples possível: apenas substituindo o serviço existente por sua nova versão. Contudo, tal procedimento pode provocar falhas na comunicação entre os serviços de uma coreografia. Vários trabalhos [MK90, VEBD07, MBG+11] estudam o processo de atualização dinâmica, pelo qual as conversações correntes são preservadas durante a atualização de um serviço. Embora não esteja no escopo de nosso trabalho, esperamos que a arquitetura do EE possa ser evoluída para que a operação de atualização de coreografia utilize procedimentos seguros de atualização dinâmica, dentre os quais destacamos a proposta de Xiaoxing et al [MBG+11].

Na Listagem 4.2 fornecemos um exemplo de um programa Java invocando o EE para implantar uma coreografia. Nesse exemplo, a classe MyChorSpec está escondendo a especificação da coreografia.

```
1 public class Deployment {
2
3     public static void main(String[] args) throws DeploymentException,
          ChoreographyNotFoundException {
4
5         final String EE_URI = "http://localhost:9102/enactmentengine";
6         EnactmentEngine ee = new EnactmentEngineClient(EE_URI);
7         ChoreographySpec chorSpec = MyChorSpec.getChorSpec();
8
9         String chorId = ee.createChoreography(chorSpec);
10        Choreography chor = ee.deployChoreography(chorId);
11
12        System.out.println(chor); // vamos ver o que aconteceu...
13    }
14 }
```

Listing 4.2: *Programa Java que invoca o Enactment Engine para implantar uma coreografia.*

4.6 Pontos de extensão

Para lidar com as particularidades do ambiente de cada organização, o Enactment Engine fornece alguns pontos de extensão. Esses pontos de extensão são classes que desenvolvedores devem escrever na linguagem Java e que, de acordo com as configurações do sistema, poderão ser executadas pelo arcabouço. Neste capítulo descreveremos os pontos de extensão de nosso middleware, mostrando as interface associadas a cada um deles. Para mais detalhes sobre todos os passos necessários para implementar uma extensão, verificar o guia do usuário (Apêndice A).

Provedor de infraestrutura: implementando a interface CloudProvider (Listagem 4.3) é possível acrescentar ao EE o suporte a novos provedores de infraestrutura. Atualmente o EE suporta o serviço EC2 do AWS e o OpenStack como provedores de infraestrutura. Cada um deles possui sua própria implementação de CloudProvider.

```
1 public interface CloudProvider {
2
3     public String getCloudProviderName();
4
5     public CloudNode createNode(NodeSpec nodeSpec) throws
          NodeNotCreatedException;
6
7     public CloudNode getNode(String nodeId) throws NodeNotFoundException;
8
9     public List<CloudNode> getNodes();
10
11    public void destroyNode(String id) throws NodeNotDestroyed,
          NodeNotFoundException;
12
13    public CloudNode createOrUseExistingNode(NodeSpec nodeSpec) throws
          NodeNotCreatedException;
14
15    public void setCloudConfiguration(CloudConfiguration cloudConfiguration
          );
16
17 }
```

Listing 4.3: *Interface CloudProvider.*

Os métodos da interface CloudProvider referem-se basicamente às operações de CRUD de máquinas virtuais em uma infraestrutura de nuvem. Além disso, a implementação pode acessar configurações específicas através do objeto `cloudConfiguration`. Tais configurações podem incluir credenciais de acesso de uma conta de nuvem (quem paga pelos nós), tipo das instâncias de VMs a serem criadas (afeta preço), chave de acesso aos nós criados, etc. A Listagem 4.4 apresenta um exemplo de configurações fornecidas à implementação AmazonCloudProvider. Essas informações são definida pelo administrador em um arquivo de configuração do EE.

```
1 LEO_AWS_ACCOUNT.CLOUD_PROVIDER=AWS
2 LEO_AWS_ACCOUNT.AMAZON_ACCESS_KEY_ID=secret!
3 LEO_AWS_ACCOUNT.AMAZON_SECRET_KEY=secret_too!
4 LEO_AWS_ACCOUNT.AMAZON_KEY_PAIR=leofl
5 LEO_AWS_ACCOUNT.AMAZON_PRIVATE_SSH_KEY=/home/leonardo/.ssh/leoflaws.pem
6 LEO_AWS_ACCOUNT.AMAZON_IMAGE_ID=us-east-1/ami-3337675a
7 LEO_AWS_ACCOUNT.AMAZON_INSTANCE_TYPE=m1.medium
```

Listing 4.4: *Configuração do AmazonCloudProvider.*

Para facilitar o desenvolvimento de novas implementações, nós fornecemos uma implementação base, a classe JCloudsCloudProvider. Ela utiliza a biblioteca JClouds[7], que já é apta a acessar uma ampla gama de provedores de infraestrutura disponíveis no mercado. Essa implementação base foi utilizada para a implementação das classes AmazonCloudProvider e OpenStackKeyStoneCloudProvider, que contaram, respectivamente, com 79 e 96 linhas de código-fonte.

Política de seleção de nós: a implementação da interface NodeSelector (Listagem 4.5) define uma nova política de alocação de serviços em nós da nuvem, que pode levar em conta os requisitos não-funcionais do serviço e propriedades dos nós à disposição. Algumas políticas já fornecidas são "sempre cria um novo nó" e "cria novos nós até um certo limite, depois faz rodízio entre eles".

```
1 public interface NodeSelector extends Selector<CloudNode,
      DeployableServiceSpec> {
2 }
3
4 public interface Selector<T, R> {
5     public List<T> select(R requirements, int objectsQuantity) throws
          NotSelectedException;
6 }
```

Listing 4.5: *Interface NodeSelector acompanhada de sua classe pai Selector.*

As implementações de NodeSelector devem criar novos nós ou devolver nós já cadastrados no EE. Os requisitos não-funcionais podem ser acessados pelo objeto `deployableServiceSpec` fornecido pelo middleware à implementação do NodeSelector. A implementação deve tomar especial cuidado com concorrência, já que o EE mantêm apenas uma instância por tipo de NodeSelector. Essa característica é importante para que políticas como rodízio de nós funcionem adequadamente.

Tipos de pacotes de serviços: um serviço pode ser distribuído por diferentes tipos de pacotes, como em um JAR ou em um WAR, por exemplo. Como existem muitas outras opções, é preciso que esse seja um ponto de flexibilidade. Para cada novo tipo de pacote, escreve-se um modelo de um *cookbook* Chef que implemente a preparação e a inicialização do serviço. Um *cookbook* possui vários arquivos, mas os principais são os arquivos da *receita*, que é o script de instalação em si, e o arquivo que define *atributos* a serem usados nas receitas. A Listagem 4.6 mostra a receita do *cookbook* modelo para implantação de WARs, enquanto que a Listagem 4.7 mostra o arquivo de atributos do mesmo *cookbook*.

```
1 include_recipe "apt"
2 include_recipe "tomcat::choreos"
3
4 remote_file "war_file" do
5         source "#{node['CHOReOSData']['serviceData']['$NAME']['PackageURL
              ']}"
6         path "#{node['tomcat']['webapp_dir']}/$NAME.war"
7         mode "0755"
8         action :create_if_missing
9 end
10
11 file "#{node['tomcat']['webapp_dir']}/$NAME.war" do
12         action :nothing
13 end
```

Listing 4.6: *Receita modelo para a implantação de WARs.*

[7] http://jclouds.incubator.apache.org/

```
1 default['CHOReOSData']['serviceData']['$NAME']['PackageURL'] = "
    $PACKAGE_URL"
```

Listing 4.7: *Arquivo modelo de atributos para a implantação de WARs.*

Os arquivos listados acima são modelos não executáveis, uma vez que apenas em tempo de implantação os símbolos *$NAME* e *$PACKAGE_URL* serão substituídos por valores adequados. Essa substituição é feita pelo próprio EE. Ou seja, criar um novo modelo de *cookbok* para o EE significa simplesmente criar um novo *cookbok* Chef utilizando adequadamente os símbolos *$NAME* e *$PACKAGE_URL*. O símbolo *$PACKAGE_URL* será substituído pela URL do pacote do serviço, enquanto que o *$NAME* será substituído por uma identificação única dentro do EE.

Tipos de serviços: a ligação entre serviços de uma composição depende da passagem de endereços que é feita do Enactment Engine para os serviços. Para isso, o EE precisa invocar a operação `setInvocationAddres` dos serviços. A implementação de tal invocação dar-se-á de forma diferente de acordo com o tipo de tecnologia de serviço empregada (SOAP ou REST, por exemplo). A implementação da interface ContextSender define como a operação `setInovcationAddress` é invocada. Atualmente o EE possui uma implementação de ContextSender, utilizada para serviços SOAP. Nota-se que para cada nova implementação, é preciso definir uma convenção para a assinatura sintática da operação `setInvocationAddres`.

```
1 public interface ContextSender {
2     public void sendContext(String serviceEndpoint,
3                             String partnerRole,
4                             String partnerName,
5                             List<String> partnerEndpoints) throws
                                  ContextNotSentException;
6 }
```

Listing 4.8: *Interface ContextSender.*

4.7 Aspectos gerais de implementação

Nesta seção descrevemos alguns detalhes sobre a implementação do Enactment Engine que podem ser especialmente úteis a eventuais desenvolvedores de nosso middleware.

Linguagem: O EE é desenvolvido na linguagem Java 6. Durante o desenvolvimento utilizamos como ambiente de execução a JVM OpenJDK 7. O EE é compilado com o Maven 3.

Chef-solo: O Chef é o sistema voltado à implantação de sistemas sobre a qual construímos o Enactment Engine. De certa forma, o EE é uma camada de abstração que facilita o uso do Chef. A versão utilizada do Chef-Solo é a 11.8.0. Em versões anteriores do EE, utilizamos o Chef Server, mas acabamos por abandoná-lo, devido ao gargalo na escalabilidade que ele gerava, além do pouco benefício funcional que ele agregava. As receitas Chef são escritas em uma Linguagem Específica de Domínio (DSL) que permite a livre utilização da linguagem Ruby, mas que possui construtos específicos para as tarefas de implantação, visando proporcionar principalmente mecanismos de idempotência. Um exemplo pode ser observado na Listagem 4.9, no qual se especifica o download de um arquivo que será baixado somente caso ele ainda não exista no sistema alvo.

```
1 remote_file "#{node['easyesb']['downloaded_file']}" do
2   source "#{node['easyesb']['url']}"
3   action :create_if_missing
4 end
```

Listing 4.9: *Trecho de receita Chef que ilustra uso de idempotência.*

Apache CXF: Uma das principais bibliotecas utilizadas pelo EE é o Apache CXF, que traz uma série de utilidades para o desenvolvimento de serviços em Java, dentre elas a implementação do padrão JAX-RS, voltado ao desenvolvimento de serviços REST.

Configuração por imagem: Na gerência de configuração de ambientes, há duas abordagens, já discutidas na Seção 2.4, sobre como configurar um ambiente: 1) utilização de imagem de disco já contendo serviço a ser implantado e 2) utilização de scripts para instalação do serviço. Enquanto a primeira abordagem prima pelo desempenho, a segunda opção oferece maior flexibilidade e facilidade de evolução. A abordagem padrão no Enactment Engine é se utilizar a configuração por scripts (gerados pelo EE). Mas o EE fornece a opção de que o administrador configure qual imagem será utilizada para criar os nós alvos. Isso possibilita que o administrador configure uma imagem que já contenha o middleware sobre o qual os serviços serão executados. Assim, se o administrador sabe que o EE será utilizado para implantar WARs, ele pode configurar uma imagem que já contenha o Tomcat instalado. Essa abordagem reduz o tempo de implantação.

Testes: Os testes de unidade do Enactment Engine são executados com o comando `mvn test`. Embora o EE contenha vários testes de unidade e isso seja fundamental, há uma limitação considerável desses testes, já que executar comandos que provoquem efeitos colaterais no sistema operacional não é adequado em testes de unidade. Tais "efeitos colaterais" são sempre provocados durante a execução das receitas Chef.

Por isso o EE possui também vários testes de *integração* automatizados, no qual máquinas virtuais são utilizados para a execução de testes nos quais o EE possa interagir com um sistema operacional. Esses testes incluem a implantação completa de coreografias. Embora esses testes sejam importantes para validar o correto funcionamento do sistema, eles são muito custosos, tanto em termos financeiros quanto de tempo, uma vez que máquinas virtuais são criadas durante esses testes.

De nossa experiência neste trabalho, acreditamos que o desenvolvimento de tecnologias de máquinas virtuais voltadas para o ambiente de teste de aceitação, de forma que as máquinas sejam criadas mais rapidamente, seja uma contribuição relevante para a prática de desenvolvimento de software.

Idempotência: a implementação da operação implantação de forma idempotente considera que falhas podem acontecer no processo de implantação em cada uma dessas três etapas: 1) preparação do nó, que consiste na seleção do nó para uma instância, incluindo a transferência dos scripts de implantação para o nó selecionado; 2) a atualização do nó, que é quando os scripts são executados; e 3) a ligação entre serviços.

Caso a falha ocorra na *preparação do nó*, o problema poderá ser corrigido na próxima execução da implantação, pois o EE sempre tenta criar $n_{spec} - n_{instancias}$ instâncias do serviço, onde n_{spec} é a quantidade de instâncias que um serviço deve ter e $n_{instancias}$ é a quantidade de instâncias que um serviço possui no momento.

Para tratar falhas ocorridas na *atualização do nó*, a cada execução da implantação o processo de atualização é executado em todos os nós novamente. Nesse passo, o EE está se aproveitando da idempotência dos scripts de implantação gerados, que no caso são receitas Chef. As receitas utilizam recursos específicos da linguagem do Chef para implementar a idempotência.

Por fim, falhas na *ligação entre serviços* são recuperadas pois todas as invocações à operação `setInvocationAddress` são refeitas. Nesse passo a idempotência é garantida pela assinatura idempotente da própria operação `setInvocationAddress`.

Software livre: por fim, todas as bibliotecas utilizadas pelo Enactment Engine são software livre.

4.8 Discussão: auxiliando implantações em grande escala

Nesta seção discutimos como as características arquiteturais e de implementação do Enactment Engine impactam na implantação de composições de serviço de grande escala. Explicamos como o EE contribui para a resolução de cada um dos desafios apresentados na Seção 2.5. Durante a discussão, destacamos como uma solução de middleware traz vantagens sobre abordagens *ad-hoc* de implantação em nosso contexto. Essa discussão, apoiada pelo efetivo funcionamento do EE demonstrado por sua avaliação (Capítulo 5), fornece também subsídios para a implementação de novos sistemas de implantação de grande escala, mesmo que não voltados a composições de serviços, e até mesmo para soluções *ad-hoc*.

Processo: Tornar a implantação de sistemas "*Internet-scale*" processos totalmente automatizados é necessário para que a implantação se torne testável, flexível e confiável [Ham07], conforme discutido na Seção 2.3. O EE possibilita a automação do processo de implantação graças a sua interface remota (REST), que recebe a especificação da composição a ser implantada e devolve o resultado do processo. Embora uma interface gráfica para a implantação de composições seja viável, tal opção não favorece a implantação automatizada, e por isso não foi priorizada em nosso trabalho.

O uso de uma especificação declarativa, como já utilizado em outros trabalhos [BBB+98, MK96], também facilita o desenvolvimento do script de implantação para cada nova composição a ser implantada. Isso ocorre porque com uso de uma linguagem declarativa o implantador descreve em alto nível apenas *o que* deve ser implantado, e não os detalhes de *como* deve ser implantado. O uso de linguagens declarativas requer algum tipo de middleware que interprete a descrição declarativa, executando as ações adequadas. Portanto, soluções *ad-hoc* dificilmente usariam linguagens declarativas, sendo em geral orientadas ao uso de scripts.

O EE segue a tendência atual na implantação de sistemas de grande escala, que é o uso de recursos elásticos possibilitados pela computação em nuvem. Recursos virtualizados fornecidos pela nuvem potencializam a automação do processo de implantação [HF11]. Diferentemente dos cenários estudados em trabalhos anteriores sobre implantação de sistemas baseados em componentes [BBB+98, MK96], em uma infraestrutura de nuvem, os nós alvos são mais dinâmicos. Não é possível conhecer os endereços IPs dos nós alvos quando se está escrevendo a especificação da composição a ser implantada. A ligação entre serviços é feita em tempo de execução, o que o EE faz via `setInvocationAddress`, e a política de alocação de nós deve ser flexível, i.e., um serviço não deve ser alocado a um IP estático antes do tempo de implantação. O EE possibilita que políticas de alocação de nós escolham, em tempo de implantação, em que nós um serviço deve ser implantado, considerando inclusive o casamento de requisitos não-funcionais do serviço com características dos nós disponíveis.

Falhas de terceiros: Seguindo recomendações gerais feitas por Nygard [Nyg09], adotamos no Enactment Engine uma abordagem simples para tratar falhas externas. A lógica de invocação a sistemas externos foi encapsulada em uma classe, chamada Invoker (Figura 4.4). Toda vez que se deve acessar um sistema externo, o EE utiliza um *invoker*. O Invoker recebe os seguintes parâmetros: uma *tarefa*, que é uma rotina que se comunicará com algum sistema externo, a quantidade de *tentativas* para executar a tarefa, o *timeout* de cada tentativa e um *intervalo* entre as tentativas.

Uma instância do Invoker deve ser configurada de acordo com sua tarefa (por exemplo, nós descobrimos que três tentativas não é o suficiente para transferência de arquivos por SCP). Em vez de ter esses valores fixados no código-fonte, eles são explicitamente ajustados em arquivos de configuração. Desta forma, pode-se ajustar esses valores de acordo com as características do ambiente alvo. Portanto, essa abordagem é também uma estratégia para colaborar com a heterogeneidade de plataformas e tecnologias. A classe Invoker pode ser decorada[8] para ajustar

[8] Ver padrão de projeto *decorator* [GHJV95].

dinamicamente os valores de timeouts de cada tarefa. Para implementar essa adaptação, o desenvolvedor tem acesso ao histórico dos tempos de execução do *invoker*.

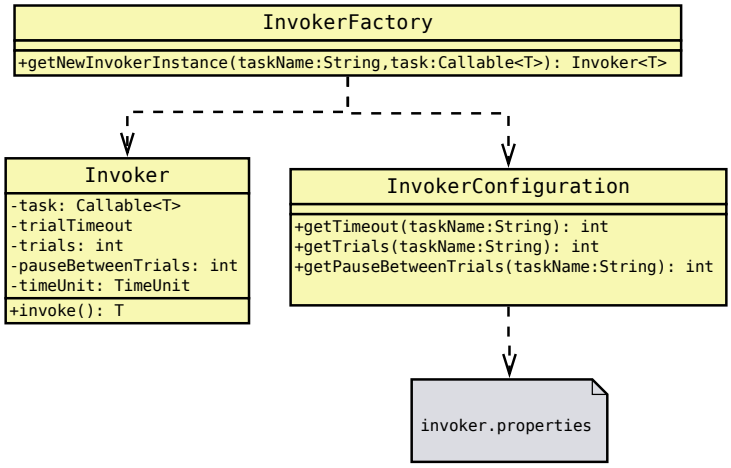

Figura 4.4: *Uma instância de* Invoker *é parametrizada com uma tarefa, uma quantidade de tentativas, um timeout por tentativa e um intervalo de tempo entre as tentativas.*

O EE utiliza os *invokers* nos momentos adequados e necessários. Isso fornece uma robustez mais facilmente proporcionada por sistemas de middleware, nos quais interesses transversais podem ser implementados sem que o implantador tenha que se preocupar com isso.

O EE adota uma estratégia particular par lidar com falhas durante a criação de novas VMs. Quando uma requisição chega, o EE tenta criar um novo nó. Se a criação falha ou demora muito, um nó já criado é recuperado de uma reserva de nós ociosos. Essa estratégia evita que se tenha que esperar novamente pelo tempo de se criar um novo nó. A capacidade inicial da reserva é definida por configuração e ela é preenchida cada vez que a criação de um nó é requisitada. Se o tamanho da reserva é reduzido e alcança um dado limite, a capacidade é aumentada, de forma a tentar evitar uma situação futura de se encontrar uma reserva vazia em um momento de necessidade.

A abordagem da reserva impõe um custo extra de se manter algumas VMs a mais em execução em um estado ocioso. Contudo, esse problema é tratado pelo EE por um algoritmo de gerenciamento distribuído em cada nó: se o nó está em um estado ocioso por $N - 1$ minutos, onde N é um limite de tempo que implica custo adicional, o nó envia ao EE um pedido para sua própria finalização. Assim, depois de um tempo de inatividade no EE, a reserva se torna vazia em algum momento, sendo preenchida novamente somente quando chegam novas requisições de criação de nós.

Considerando que o tempo de criação de um nó é bem maior que o tempo de inicialização (*boot*) desse nó, uma melhoria a ser feita na reserva de nós ociosos é manter o nó ocioso desligado até o momento de uso, uma vez que provedores de infraestrutura, como a Amazon, costumam não cobrar por máquinas desligadas. Essa abordagem pode diminuir um pouco o desempenho da reserva de nós, mas irá economizar mais recursos.

Outra prática importante relacionada a tolerância a falhas é a *degradação suave* [Bre01, Ham07]. Em nosso contexto, degradação suave significa que se um serviço não foi implantado apropriadamente, não é aceitável que o processo de implantação de toda a composição seja interrompido. Com o EE, se algum serviço não é implantado, o processo de implantação continua, e a resposta do EE fornece informações sobre os problemas ocorridos, possibilitando ações de recuperação.

Contudo, é importante destacar que a responsabilidade pela degradação suave deve ser compartilhada com a implementação dos serviços, uma vez que cada serviço deve saber como se

comportar na ausência de uma ou mais de suas dependências. De outra forma, cada serviço se tornaria um *ponto de falha único* na composição, o que é altamente indesejável.

Por fim, a operação de implantação fornecida pelo EE foi implementada de forma idempotente. Isso garante que caso a resposta à requisição de implantação de coreografia não chegue ao cliente, o cliente possa repetir a requisição sem alterar o resultado do processo de implantação. Caso a operação de implantação seja chamada pela segunda vez, o EE não implantará instâncias adicionais de um serviço caso ele já esteja corretamente implantado, mas implantará somente as instâncias necessárias para a correta finalização da implantação da coreografia. Mais detalhes sobre a implementação da garantia de idempotência, ver a Seção 4.7.

Disponibilidade: A especificação de um serviço na ADL do Enactment Engine possibilita a definição da quantidade de réplicas de um serviço a ser implantado pelo EE. Essa quantidade inicial de réplicas pode ser alterada pelo implantador em tempo de execução com a atualização da especificação da coreografia. A definição da quantidade adequada de réplicas, definida pelo implantador, possibilita não só uma melhora de desempenho, mas também um aumento na disponibilidade do serviço, já que uma falha em uma réplica específica não afeta as outras réplicas disponíveis.

Por questões de simplificação, em nosso trabalho omitimos a relação que serviços possuem com bancos de dados. Dessa forma, é importante que versões futuras do EE contemplem a automação da implantação de bancos de dados a serem utilizados pelos serviços implantados. Nesse estágio, deverá ser considerado também a replicação do banco de dados, e que os dados são utilizados simultaneamente por várias réplicas do serviço.

Escalabilidade: Para implementar o EE, em matéria de programação concorrente, não foi preciso saber muito mais que coordenar a abertura e encerramento de múltiplas *threads*, além de sincronizar o acesso a recursos compartilhados por diferentes *threads*. Assim, depreendemos que o nível de conhecimento de programação concorrente para implementar um processo de implantação escalável seja básico. Contudo, programação concorrente por si própria é reconhecida como difícil e propensa a erros. Muitas vezes, linguagens de scripts não oferecem um bom suporte à programação concorrente. O tratamento adequado de falhas de terceiros também é um requisito importante para a obtenção de um sistema escalável. Portanto, implementar concorrência e tratamento a falhas na camada de middlware é um passo significativo para facilitar a implementação efetiva de um processo de implantação escalável.

Uma lição aprendida na prática para se atingir a escalabilidade foi evitar componentes que se tornem gargalos no sistema. Em versões anteriores do EE, o Chef Server era um ponto central constantemente requisitado por processos em outros nós. A mudança da arquitetura do EE da utilização do Chef Server para o Chef Solo[9] foi essencial para se obter desempenhos razoáveis com uma grande quantidade de serviços implantados..

No Capítulo 5 apresentamos a avaliação em detalhes da escalabilidade fornecida pelo Enactment Engine.

Heterogeneidade: Na Seção 4.6 apresentamos os pontos de extensão do Enactment Engine, que possibilitam mais facilmente adaptá-lo para diversos provedores de infraestrutura e tecnologias de desenvolvimento e empacotamento de serviços. Essa flexibilidade ajuda a superar as atuais limitações de soluções de Plataformas como um Serviço que restrigem as opções tecnológicas disponíveis aos desenvolvedores de aplicações. Essas restrições normalmente se aplicam justamente sobre provedor de infraestruturas e a linguagem de programação da aplicação. Exemplos: para utilizar o PaaS da Amazon (Elastic Beanstalk) é preciso utilizar o IaaS da Amazon, enquanto que para utilizar o PaaS do Google (App Engine) é preciso escolher entre as linguagens Java, Pyhton, PHP ou Go.

[9]Na versão Chef Solo, o EE passa os scripts de implantação diretamente ao nó onde o script será executado. Já com o Chef Server, esses scripts eram primeiro armazenados no Chef Server, para depois serem acessados pelos nós envolvidos na implantação.

Oferecer suporte a variações de um padrão é um desafio para sistemas de middleware. Adequar um middleware para a particularidade de uma aplicação pode não ser fácil. No suporte a diferentes tecnologias, as abordagens *ad-hoc* encontram realmente um espaço de importância. No entanto, uma vez que a adequação para uma nova tecnologia seja feita no middleware, o esforço para o desenvolvimento de futuras aplicações utilizando a mesma tecnologia se torna menor.

Múltiplas organizações: O Enactment Engine possui dois principais mecanismos para implantar composições cujos serviços pertencem a diferentes organizações. O primeiro mecanismo é a definição da "conta de nuvem" a ser usada na implantação de um serviço. Essa definição é feita na especificação do serviço e deve bater com configurações previamente feitas pelo administrador no EE. Uma "conta de nuvem" não indica apenas a nuvem alvo (Amazon, por exemplo), mas também quem vai pagar pela infraestrutura (qual conta da Amazon será utilizada, por exemplo). Uma vez que os serviços de cada organização sejam configurados para serem implantados nas contas de nuvem adequadas, o EE irá implantar adequadamente uma composição multi-organizacional. No entanto essa abordagem ainda apresenta limitações sérias no quesito de segurança, pois a configuração da conta de nuvem deve ser fornecida ao administrador do EE, que seria uma das organizações ou um terceiro. Esse e outros problemas surgem do fato que diferentes organizações teriam que compartilhar uma mesma instância do EE.

O segundo mecanismo é a utilização da entidade *serviço legado* na especificação da composição. O serviço legado é um serviço já existente na Internet, e que portanto não será implantado pelo EE. A utilidade de utilizar esse mecanismo está na fase de ligação entre serviços, pois o EE irá fornecer aos serviços implantados os endereços dos serviços legados declarados como suas dependências. A maior limitação dessa abordagem é a dificuldade em se lidar com a alteração de URIs dos serviços legados. Quando isso ocorre, uma nova especificação da composição deve ser feita e enviada ao EE, mas o problema é saber *quando* isso deve ser feito.

Considerando as limitações dos mecanismos até aqui implementados, divisamos como importante trabalho futuro uma arquitetura de federação entre instâncias do EE. Caso um serviço S_A, implantado com o EE pela organização O_A, dependa de um serviço legado B, também implantado com o EE, mas para a organização O_B, a instância do EE em O_B poderia manter a instância do EE em O_A informada sobre o estado de S_B. Para que essa funcionalidade seja implementada é preciso projetar um protocolo de comunicação entre instâncias do EE.

Nesse estágio proposto (federação dos sistemas implantadores de cada organização) uma abordagem orientada a middleware se torna importante por questão de padronização. Abordagens *ad-hoc* deveriam ser desenvolvidas de forma coordenada entre as diferentes organizações, o que seria mais custoso do que a adoção de uma plataforma comum.

Adaptabilidade: O Enactment Engine por si só não garante que uma composição será autonômica ou auto-adaptativa. Contudo, ele fornece suporte para o desenvolvimento de tais sistemas.

Sistemas auto-adaptativos e autonômicos precisam estar cientes e ter pleno controle das atividades de implantação. Para equipar tais sistemas, o EE fornece informação e controle das seguintes funcionalidades:

- atualização das composições;
- migração de serviços;
- replicação de serviços;
- implantação de infraestrutura de monitoramento.

Atualizações de composições de serviços podem ser necessárias quando as regras de negócio ou os requisitos não-funcionais mudam. O EE possibilita, por uma API REST, a adição, remoção e reconfiguração dos serviços e seus recursos computacionais associados.

A migração de um serviço para um nó com mais recursos computacionais é uma funcionalidade oferecida pelo EE chamada de *escalamento vertical* [Pri08]. No entanto, para a construção de sistemas escaláveis se recomenda a replicação de serviços associada ao balanceamento de carga [TF12], o que é conhecido como *escalamento horizontal* [Pri08].

O EE possibilita a replicação de serviço por meio da implantação de múltiplas instâncias do serviço e da notificação aos serviços consumidores sobre a existência dessas réplicas durante a fase de ligação de serviços. A quantidade inicial de réplicas é definida por atributo na especificação do serviço fornecida ao EE, e, depois, pode ser redefinida dinamicamente. Um trabalho futuro é configurar automaticamente um balanceamento de carga entre réplicas de um serviço, de forma que o consumidor de um serviço não tenha necessidade de saber sobre suas diversas réplicas.

Por fim, o EE fornece opcionalmente a implantação de uma infraestrutura de monitoramento na infraestrutura alvo. Utilizamos o Ganglia, que coleta métricas do sistema operacional, como consumo de CPU, por exemplo. As métricas coletadas são enviadas a um serviço previamente configurado no EE. Esse serviço de monitoramento pode então disparar ações de adaptação com base nos dados recebidos. Uma ação de adaptação envolve a geração de uma nova especificação de composição e a atualização da composição em execução de acordo com a nova especificação.

O CHOReOS Enactment Engine é uma ferramenta útil a praticantes e pesquisadores para a implantação de composições de serviços, especialmente no contexto de grande escala. Mas as funcionalidades relacionadas à adaptação são de especial interesse aos pesquisadores que trabalham com a auto-adaptação de composições de serviços. O EE facilita a implementação de sistemas adaptativos por possibilitar que pesquisadores se foquem mais nos problemas de adaptação em alto nível, abstraindo detalhes altamente específicos do gerenciamento de implantação. Diferentes pesquisadores desse campo de pesquisa podem se beneficiar ao utilizarem uma plataforma comum, potencializando a troca de experiência sobre o processo de implantação.

Capítulo 5

Avaliação

Neste capítulo apresentamos experimentos realizados com o Enactment Engine com objetivo de avaliar seu uso, contrastando com soluções *ad-hoc* de implantação automatizada. Também avaliamos seu desempenho e escalabilidade para verificar a viabilidade de seu uso no contexto de composições de serviços de grande escala.

5.1 Implantando coreografias com e sem o EE

Nesta seção, nós avaliamos como o Enactment Engine melhora o processo de implantação pelo fato de ser uma solução baseada em middleware. Para essa avaliação, desenvolvemos uma solução *ad-hoc* para a implantação de uma coreografia particular. A "coreografia do aeroporto" é um exemplo fornecido por especialistas no domínio aeroportuário [CV12] e que contém 15 serviços. Também implantamos a mesma coreografia utilizando o EE. Ambas as soluções estão disponíveis em https://github.com/choreos/airport_enactment.

Para implantar a coreografia do aeroporto com o EE, escrevemos a especificação da coreografia e o programa cliente para invocar o EE, disparando assim a implantação. A especificação da coreografia foi escrita com objetos Java em 40 minutos, contendo 162 linhas de código (LoC), uma média de 11 linhas por serviço. O autor dessa especificação foi um aluno de doutorado, já acostumado ao conceito de coreografias de serviços web, e que contribuiu com o código do EE. O programa cliente, a classe `AirportEnact`, utiliza a API Java do EE, tem apenas 22 linhas de código, e foi escrito pelo autor desta dissertação. Depois que essas classes foram escritas, a implantação da coreografia em três nós, utilizando o EE, levou apenas 4 minutos.

Para desenvolver a solução *ad-hoc* foi necessário aproximadamente 9 horas de desenvolvimento de um programador (o autor desta dissertação), e mais 60 minutos para o mesmo programador executar a implantação, distribuindo os 15 serviços por três nós alvos. Essa solução precisou da escrita de 100 LoC de shell scripts, 220 LoC de Java, and 85 LoC de Ruby (para o Chef).

No restante dessa seção, descreveremos o processo de criação e execução da solução *ad-hoc*. Destacaremos as dificuldades no processo que o implantador encontra sem o uso do EE.

A implantação de cada serviço é executada por uma receita Chef contida em um *cookbook*. Nós escrevemos um *cookbook* modelo, para implantar pacotes JARs, e o usamos para gerar *cookbooks* para os 15 serviços participantes. O processo de criar os 15 *cookbooks* foi parcialmente automatizado pelo script `generate` que escrevemos. As URLs dos pacotes dos serviços tiveram que ser manualmente inseridas nos *cookbooks* depois da execução da script `generate`.

Para implementar a ligação entre serviços, desenvolvemos um pequeno mas não trivial programa Java, chamado called `context_sender`. Ele é responsável por invocar a operação `setInvocationAddress` de um dado serviço. Implementamos o `context_sender` como um programa Java para aproveitar a API SOAP fornecida pelo ambiente Java SE. Também desenvolvemos o script `bind_services`, responsável por executar o programa `context_sender` para cada dependência presente na coreografia. Uma vez que os IPs dos serviços são conhecidos apenas após a implantação, o script `bind_services` é na verdade um modelo com lacunas que devem ser manualmente preenchidas

com os IPs dos serviços implantados.

A execução da solução *ad-hoc* possui vários passos, inclusive alguns manuais. Para cada nó alvo, o implantador deve se conectar ao nó (SSH), instalar o git, baixar os *cookbooks*, executar o script `install_chef` para instalar o Chef, editar alguns arquivos de configuração para definir quais serviços serão implantados no nó, e executar o Chef-Solo. Após implantar os serviços, o implantador deve editar o script `bind_services` com os IPs dos serviços implantados, e finalmente executar o script `bind_services`. Alguns dos problemas dessa solução *ad-hoc* são:

- Três diferentes tecnologias são utilizadas: shell script, Java e Chef. Expertise em linha de comando também foi necessária em alguns passos, como utilizar o editor *vim* ou o comando `ps` para verificar o estado dos processos dos serviços implantados. Isso sugere que se requer uma ampla gama de habilidades técnicas do desenvolvedor de soluções de implantação. Algumas dessas habilidades, como utilizar o Chef, são notoriamente não-fáceis de se aprender. O código Java utilizado para realizar a invocação de serviços SOAP pode também ser considerado como não-trivial para um programador não acostumado com o padrão SOAP.

- Replicação de código nos *cookbooks* gerados. Se algo muda no modelo, é preciso regenerar todos os *cookbooks* e realizar a edição manual também. Contudo, as edição manuais mencionadas poderiam ser evitadas com um script mais complexo. Replicação de código poderia também ter sido evitada com a criação de um "LWRP" (light weight resource provider) do Chef, mas isso seria uma tarefa para usuários avançados do Chef.

- Para cada nó alvo, o implantador deve realizar alguns passos manuais que são demorados. Alguns deles (executar `install_chef`, por exemplo) poderiam ser evitados com a utilização de uma ferramenta como Capistrano[1], mas isso demandaria mais uma tecnologia a ser aprendida. Outros passos manuais, como a edição de arquivos de configuração, são bastante propensos a erros. Esquecer-se de vírgulas ou digitar errado o nome de serviços são erros bem prováveis de acontecerem.

- Há muita pouca paralelização no processo. Com os scripts construídos, o implantador poderia melhorar um pouco o paralelismo utilizando ferramentas como o Byobu[2] para digitar o mesmo comando em várias máquinas. Mas isso demandaria mais uma habilidade a ser aprendida pelo implantador, além de ser uma forma muito limitada de escalar o processo.

Nesse exemplo, usamos uma composição de apenas 15 serviços. Composições de grande escala aumentariam muito mais a complexidade da solução *ad-hoc*. Para se obter uma solução completa com a abordagem *ad-hoc*, um esforço extra de desenvolvimento seria necessário para implementar funcionalidades já presentes no EE, como o tratamento de falhas de terceiros, atualização de coreografias, seleção dinâmica de nós, implantação concorrente, etc. Além disso, para desenvolver a solução *ad-hoc*, utilizamos códigos que já estavam disponíveis no EE, tais como os modelos dos *cookbooks* e o `context_sender`. Implantadores teriam que começar tudo do zero.

Nós reconhecemos que essa avaliação por comparação com uma solução *ad-hoc* tem suas limitações, uma vez que os resultados dependem fortemente das habilidades técnicas do implantador. Conduzir um experimento rigoroso de engenharia de software com vários desenvolvedores assumindo o papel de implantador traria uma evidência melhor. Contudo, acreditamos que a avaliação descrita aqui já é o suficiente para expandir nosso entendimento sobre o valor agregado por uma solução com suporte de middleware, como o Enactment Engine, uma vez que temos agora uma boa ilustração do esforço necessário para se implantar composições de serviços.

[1] http://www.capistranorb.com/
[2] http://byobu.co/

Tabela 5.1: *Cenários de implantação para a análise multi-variada.*

Cenário	*Composições*	*Tamanho*	*Nós*	*Serviços/Nós*
1	10	10	9	11 or 12
2	10	100	90	11 or 12
3	100	10	90	11 or 12
4	10	10	5	20

5.2 Análise de desempenho e escalabilidade

Conduzimos experimentos para avaliar o desempenho e escalabilidade do Enactment Engine em função de sua capacidade de implantar um número significativo de composições em uma plataforma de nuvem utilizada no mercado.

Nossos experimentos utilizam uma carga sintética modelada conforme ilustrado na Figure 5.1. A direção das flechas vão do serviço consumidor para o provedor. Embora respostas não sejam representadas por questão de simplicidade, elas são sempre enviadas sincronamente. Essa topologia foi escolhida porque 1) é um exemplo representativo de processos de negócios (potencialmente compostos por ramificações – chamadas para outros sistemas – e junções correspondentes) e 2) segue um padrão repetitivo que pode ser usado para aumentar suavemente o tamanho da composição, para que possamos analisar o desempenho do EE conforme a carga aumenta.

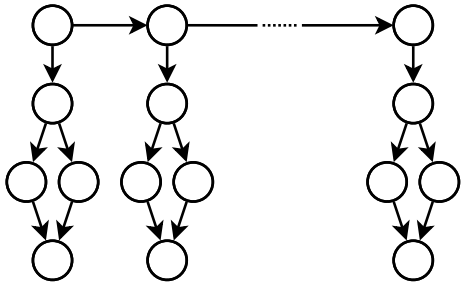

Figura 5.1: *Topologia das composições utilizadas em nossos experimentos.*

Inicialmente, conduzimos uma análise multi-variada do desempenho do EE implantando composições nos seguintes cenários: 1) um pequeno conjunto de pequenas composições; 2) um pequeno conjunto de composições maiores; 3) um conjunto maior de pequenas composições; 4) uma razão maior de serviços por nó. A Tabela 5.1 quantifica cada cenário.

Em nossos experimentos, a política de alocação de nós foi "cria nós até um limite, e depois faz rodízio entre eles", na qual a quantidade de nós utilizados é configurada antes de cada experimento. Se a quantidade de nós não é divisível pelo número de nós, alguns nós hospedarão um serviço a mais que outros nós. O tamanho da reserva de nós ociosos era 5, e o *timeout* de criação de nós era 300 segundos. Utilizamos a Amazon EC2 como provedor de infraestrutura, com instâncias do tipo *small*, cada uma com 1.7 GiB of RAM, uma vCPU com processamento equivalente a 1.0–1.2 GHz, e rodando Ubuntu GNU/Linux 12.04. O EE foi executado em uma máquina com 8 GB de RAM, com processador Intel Core i7 CPU de 2.7 GHz e GNU/Linux kernel 3.6.7. A versão do EE utilizada nos experimentos e os dados coletados estão disponíveis on-line[3] para garantir a reprodutibilidades dos resultados.

Cada cenário foi executado 30 vezes e a Tabela 5.2 apresenta, para cada cenário, o tempo necessário para implantar todas as composições mais o tempo para invocá-las, o que é feito para certificar que elas foram corretamente implantadas. Os valores são médias com intervalo de confiança de 95%. A tabela também mostra quantas composições e serviços foram corretamente implantados.

Os resultados mostram que o EE escala bem em termos de serviços sendo implantados. Embora o número de serviços foi multiplicado por 10, o tempo de implantação cresceu aproximadamente

[3] http://ccsl.ime.usp.br/enactmentengine

Tabela 5.2: *Resultados experimentais.*

Cenário	*Tempo*	*Composições com sucesso*	*Serviços com sucesso*
1	467.9 ± 34.8	10.0 ± 0	100.0 ± 0 (100%)
2	1477.1 ± 130.0	9.3 ± 0.3	999.3 ± 0.4 (99.9%)
3	1455.2 ± 159.1	98.9 ± 0.8	998.5 ± 1.3 (99.9%)
4	585.2 ± 38.1	10.0 ± 0.1	100.0 ± 0.1 (100%)

apenas 3 vezes nos cenários 2 e 3. Esse incremento no tempo de implantação é causado principalmente pelo fato de que quanto maior a quantidade de serviços, maiores são as chances da ocorrência de falhas, o que dispara a re-execução de algumas rotinas.

Os resultados também mostram que o quando o número de serviços por nó dobrou (cenário 4), o tempo de implantação cresceu aproximadamente 25%. Parte dessa sobrecarga foi causada pelo incremente no número de receitas Chef que devem ser executadas (sequencialmente) nos nós.

Durante os experimentos, observamos que, graças aos mecanismos de tolerância a falhas do EE, a quantidade de falhas na implantação foi baixa: todos os serviços foram corretamente implantados em mais de 75% das execuções. Por "falha" consideramos que um serviço não foi corretamente implantado. O cenário 1 não apresentou falhas, enquanto que no cenário 4 houve apenas uma falha. No cenário 2, a pior situação, foram 3 falhas dentre 1000 serviços. No cenário 3, houve uma execução com 20 falhas, mas esse foi um evento excepcional, uma vez que a segunda pior situação contou com apenas 3 falhas.

Finalmente, observamos que 80% das execuções não utilizaram a reserva de nós ociosos. Quando a reserva foi usada, houve um acesso máximo por execução de 6 nós, mas na maioria das vezes o acesso foi de apenas 1 nó. Também observamos que o tempo de implantação não foi significativamente afetado quando falhas no ambiente de nuvem ocorriam, uma vez que novos nós eram imediatamente recuperados da reserva.

Também conduzimos experimentos para avaliar o desempenho e escalabilidade do Enactment Engine em termos de sua capacidade de implantar grandes composições de serviços. Esses experimentos foram realizados em 5 cenários, nos quais se variou o tamanho das composições sendo implantadas e a quantidade de nós disponíveis no ambiente de nuvem, enquanto manteve-se a razão de 20 serviços implantados por nó. Cada cenário foi executado 10 vezes.

A topologia utilizada na composição foi a mesma de antes (Figura 5.1). O EE foi executado em uma máquina virtual (8 GiB de RAM e 4 vCPUs) hospedada na infraestrutura de nossa Universidade. Os nós alvos criados pelo EE eram instâncias *small* da Amazon EC2 e o *timeout* de criação dos nós era de 250 segundos. Os tempos médios de implantação, com intervalo de confiança de 95%, são mostrados na Tabela 5.2.

Sobre as falhas de implantação, as piores execuções de cada cenário tiveram 1, 1, 2, 2, e 4 serviços não implantados corretamente dentre, 200, 600, 1000, 1400 e 1800 serviços, respectivamente.

Esses resultados mostram uma boa escalabilidade em termos de serviços implantados. Aumentando-se 9 em vezes o número de serviços implantados, o tempo de implantação aumentou 3,5 vezes. Em números absolutos, cada incremento em 400 serviços implantados foi responsável pelo incremento de 180 a 460 segundos no tempo de implantação.

Gerosa ►*E aquela questão de olhar o real overhead vs o crescimento esperado pela variação estatística do tempo individual de implantação?*◄

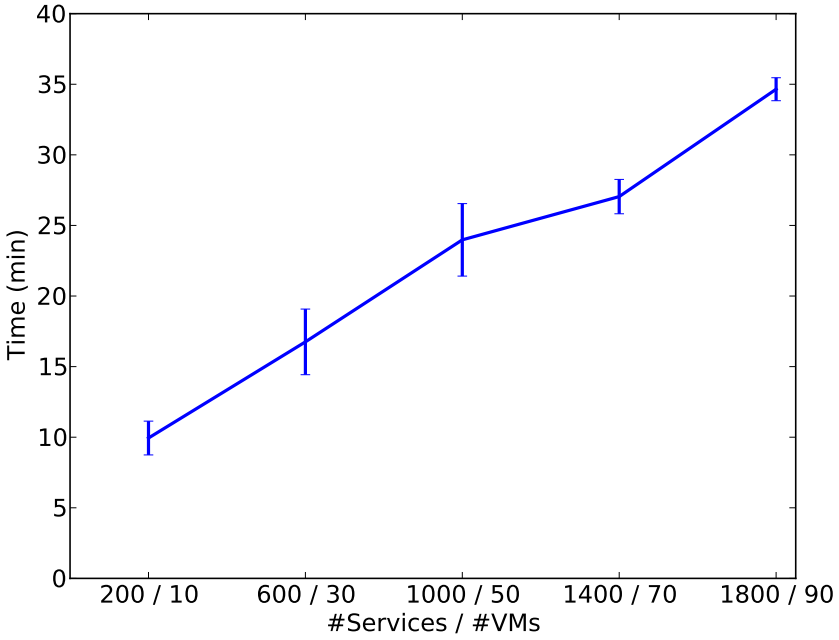

Figura 5.2: *Tempos médios de implantação com aumento constante na quantidade de serviços implantados, mantendo-se constante a razão serviços implantados / nós.* ToDo ►traduzir figura◄

Capítulo 6

Conclusões

Fim :)

Apêndice A

Guia do Usuário do Enactment Engine

Referências Bibliográficas

[ADM00] Alessandra Agostini e Giorgio De Michelis. Improving flexibility of workflow management systems. Em *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, páginas 289–342. Springer Berlin / Heidelberg, 2000. 7

[AdRdS+13] Marco Autilli, Davide di Ruscio, Amleto di Selle, Paola Inverardi e Massimo Tivoli. A model-based synthesis process for choreography realizability enforcement. Em *16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013. 8

[All10] Subu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly Media, Inc., 2010. 15

[ATK05] Anatoly Akkerman, Alexander Totok e Vijay Karamcheti. Infrastructure for automatic dynamic deployment of J2EE applications in distributed environments. Em *Component Deployment*, volume 3798 of *Lecture Notes in Computer Science*, páginas 17–32. Springer Berlin Heidelberg, 2005. 18, 21

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill e J.-Y. Vion-Dury. Architecturing and configuring distributed application with Olan. Em *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, páginas 241–256. Springer-Verlag, 1998. 17, 18, 21, 28, 34

[BPGR08] Paolo Besana, Vivek Patkar, David Glasspool e Dave Robertson. Distributed workflows: The openknowledge experience. Em Robert Meersman, Zahir Tari e Pilar Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*, volume 5333 of *Lecture Notes in Computer Science*, páginas 965–975. Springer Berlin Heidelberg, 2008. 8, 20

[Bre01] Eric A. Brewer. Lessons from giant-scale services. *Internet Computing, IEEE*, 5(4):46 –55, 2001. 15, 35

[Bre12] Eric A. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012. 15

[BWR09] Adam Barker, Christopher D. Walton e David Robertson. Choreographing Web Services. *IEEE Transactions on Services Computing*, 2(2):152–166, 2009. 7

[Cat11] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, 2011. 15

[CCPP98] F. Casati, S. Ceri, B. Pernici e G. Pozzi. Workflow evolution. *Data & Knowledge Engineering*, 24(3):211–238, 1998. 7

[CFN10] Franco Cicirelli, Angelo Furfaro e Libero Nigro. A service-based architecture for dynamically reconfigurable workflows. *Journal of Systems and Software*, 83(7):1148–1164, 2010. 8

[CV12] Pierre Chatel e Hugues Vincent. Deliverable D6.2. Passenger-friendly airport services & choreographies design. Disponível on-line em: http://choreos.eu/bin/Download/Deliverables, 2012. 1, 39

[DBV05] Eelco Dolstra, Martin Bravenboer e Eelco Visser. Service configuration management. Em *Proceedings of the 12th international workshop on Software configuration management (SCM '05)*, páginas 83–98. ACM, 2005. 1, 10, 17, 21, 28

[Dea07] Alan Dearle. Software deployment, past, present and future, 2007. 27, 28

[DK76] F. DeRemer e H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, 1976. 17

[DNGM+08] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou e K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3):313–341, 2008. 16

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern, Janeiro 2004. http://martinfowler.com/articles/injection.html. 5, 27

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Chap. 4 Structural patterns. Em *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 34

[Gro05] Network Working Group. Uniform Resource Identifier (URI): Generic syntax, Janeiro 2005. http://tools.ietf.org/html/rfc3986. 6

[Had06] Marc Hadley. Web application description language (wadl), Abril 2006. http://labs.oracle.com/techrep/2006/abstract-153.html. 6

[Ham07] James Hamilton. On designing and deploying internet-scale services. Em *Proceedings of the 21st Large Installation System Administration Conference (LISA '07)*, páginas 231–242. USENIX, 2007. 13, 15, 34, 35

[HC09] Pat Helland e Dave Campbell. Building on quicksand. arXiv.org, 2009. http://arxiv.org/abs/0909.1788, acessado em fevereiro de 2013. 13, 15

[Hew09] Eben Hewitt. Introduction to SOA. Em *Java SOA Cookbook*. O'Reilly, 2009. 5

[HF11] Jez Humble e David Farley. *Continuous Delivery*. Addison-Wesly, 2011. 1, 2, 9, 10, 12, 13, 17, 18, 34

[HM11] Jez Humble e Joanne Molesky. Why enterprises must adopt devops to enable continuous delivery. *Cutter IR Journal, The Journal of Information Technology Management*, 24(8):6–12, 2011. 10

[IGH+11] Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Panos Vassiliadist, Marco Autili, MarcoAurélio Gerosa e AmiraBen Hamida. Service-oriented middleware for the future internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2:23–45, 2011. 1

[Lee12] Thorsten Leemhuis. What's new in linux 3.2, Janeiro 2012. http://h-online.com/-1400680, acessado em fevereiro de 2013. 13

[LHM+05] Ling Lan, Gang Huang, Liya Ma, Meng Wang, Hong Mei, Long Zhang e Ying Chen. Architecture based deployment of large-scale component based systems: The tool and principles. Em *Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, páginas 123–138. Springer Berlin Heidelberg, 2005. 18

[LPP04] Sébastien Lacour, Christian Pérez e Thierry Priol. Deploying CORBA components on a computational grid: General principles and early experiments using the globus toolkit. Em *Component Deployment*, volume 3083 of *Lecture Notes in Computer Science*, páginas 35–49. Springer Berlin Heidelberg, 2004. 19, 21

[lT10] Kent Ka lok Tong. Creating scalable web services with rest. Em *Developing We Services with Apache CXF and Axis2*. TipTec Development, 2010. 7

[MBG+11] Xioxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna e Jian Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. Em *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE 2011)*, páginas 245–255, 2011. 29

[MBNR68] M. Douglas McIlroy, J. M. Buxton, Peter Naur e Brian Randell. Mass-produced software components. Em *Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering*, páginas 88–98, 1968. 5

[MDK94] Jeff Magee, Naranker Dulay e Jeff Kramer. A constructive development environment for parallel and distributed programs. Em *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, páginas 4–14, 1994. 18, 21, 27

[MG11] Peter Mell e Timothy Grance. The NIST definition of cloud computing (draft), 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Acessado em 2 de dezembro de 2012. 2, 11

[MK90] Jeff Magee e Jeff Kramer. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990. 29

[MK96] Jeff Magee e Jeff Kramer. Dynamic structure in software architectures. Em *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT '96)*, páginas 3–14. ACM, 1996. 17, 21, 27, 34

[MTK97] Jeff Magee, Andrew Tseng e Jeff Kramer. Composing distributed objects in CORBA. Em *Proceedings of the Third International Symposium on Autonomous Decentralized Systems, 1997. ISADS 97.*, páginas 257–263, 1997. 17, 28

[NCS04] Mangala Gowri Nanda, Satish Chandra e Vivek Sarkar. Decentralizing execution of composite web services. Em *Proceedings of the 19th annual ACM SIGPLAN conference on object oriented programming, systems, languages, and applications (OOPSLA '04)*, páginas 170–187. ACM, 2004. 7

[Nyg09] Michael T. Nygard. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2009. 14, 15, 34

[OAS07] OASIS. Web services business process execution language, version 2.0, Abril 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 7

[OMG95] OMG. The common object request broker architecture and specification, 1995. Revision 2.0. 5, 6

[OMG06] OMG. Deployment and configuration of component-based distributed applications (DEPL), Abril 2006. http://www.omg.org/spec/DEPL/. 1, 8

[OMG11] OMG. Business process model and notation (BPMN), version 2.0, Janeiro 2011. http://www.omg.org/spec/BPMN/2.0. 7, 8

[Pou11] Michael Poulin. Collaboration patterns in the SOA ecosystem. Em *Proceedings of the 3rd Workshop on Behavioural Modelling*, páginas 12–16. ACM, 2011. 8

[Pri08] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008. 38

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar e Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007. 1, 5, 6, 7

[PZL08] Cesare Pautasso, Olaf Zimmermann e Frank Leymann. 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)*, páginas 805–814. ACM, 2008. 6, 7

[QBB+04] Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet e Serge Lacourte. Asynchronous, hierarchical, and scalable deployment of component-based applications. Em *Component Deployment*, volume 3083 of *Lecture Notes in Computer Science*, páginas 50–64. Springer Berlin Heidelberg, 2004. 18, 21

[Qui94] Michael Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 2nd edition edição, 1994. 15

[Rie11] Eric Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011. 11

[RSF11] N. Roohi, G. Salaün e V. France. Realizability and dynamic reconfiguration of Chor specifications. *Informatica: An International Journal of Computing and Informatics*, 35(1):39–49, 2011. 8

[RV13] G Ramalingam e Kapil Vaswani. Fault tolerance via idempotence. Em *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '13)*, páginas 249–262. ACM, 2013. 14

[RWR06] S. Rinderle, A. Wombacher e M. Reichert. Evolution of process choreographies in DYCHOR. Em *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, páginas 273–290. Springer, 2006. 8

[SABS02] Heiko Schuldt, Gustavo Alonso, Catriel Beeri e Hans-Jörg Schek. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, 2002. 7

[SDK+07] Ronny Siebes, Dave Dupplaw, Spyros Kotoulas, Adrian Perreau De Pinninck, Frank Van Harmelen e David Robertson. The openknowledge system: an interaction-centered approach to knowledge sharing. Em *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, páginas 381–390. Springer, 2007. 20

[SM10] Virginia Smith e Bryan Murray. Automated service evolution. dynamic version coordination between client and server. Em *SERVICE COMPUTATION 2010 : The Second International Conferences on Advanced Service Computing*, páginas 21–26. IARIA, 2010. 6

[Sof06] Software Engineering Institute of Carnegie Mellon University. *Ultra-Large-Scale Systems, The Software Challenge of the Future*. 2006. 13

[SPV12] Maarten Steen, Guillaume Pierre e Spyros Voulgaris. Challenges in very large distributed systems. *Journal of Internet Services and Applications*, 3(1):59–66, 2012. 1, 2, 13, 16

[Szy03] Clemens Szyperski. Component technology: what, where, and how? Em *Proceedings of the 25th International Conference on Software Engineering*, páginas 684–693, 2003. 5

[Szy10] Clemens Szyperski. Chapter 13. The OMG way: CORBA, CCM, OMA, and MDA. Em *Componente Software. Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition edição, 2010. 5

[TF12] Matt Tavis e Philip Fitzsimons. Web Application Hosting in the AWS Cloud: Best Practices. Relatório técnico, Amazon, Setembro 2012. 12, 15, 19, 38

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers e Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007. 29

[W3C02] W3C. Web service choreography interface (WSCI), version 1.0, Agosto 2002. http://www.w3.org/TR/2002/NOTE-wsci-20020808. 7

[W3C04a] W3C. Web services addressing (ws-addressing), Agosto 2004. http://www.w3.org/Submission/ws-addressing/. 5

[W3C04b] W3C. Web services architecture, Fevereiro 2004. http://www.w3.org/TR/ws-arch/. 1, 6

[W3C05] W3C. Web services choreography description language (WS-CDL), version 1.0, Novembro 2005. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109. 7

[WASL13] Johannes Wettinger, Vasilios Andrikopoulos, Steve Strauch e Frank Leymann. Enabling dynamic deployment of cloud applications using a modular and extensible PaaS environment. Em *IEEE Sixth International Conference on Cloud Computing*, páginas 478–485. IEEE, 2013. 17, 19

[WBB+13] Johannes Wettinger, Michael Behrendt, Tobias Binz, Uwe Breitenbücher, Gerd Breiter, Frank Leymann, Simon Moser, Isabell Schwertle e Thomas Spatzier. Integrating configuration management with model-driven cloud management based on tosca. Em *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*, páginas 437–446. SciTePress, 2013. 19

[WFK+06] Paul Watson, Chris Fowler, Charles Kubicek, Arijit Mukherjee, John Colquhoun, Mark Hewitt e Savas Parastatidis. Dynamically deploying web services on a grid using Dynasoar. Em *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, página 8. IEEE, 2006. 5, 17, 19, 21

[Wor99] Workflow Management Coalition. Workflow management coalition terminology & glossary, Fevereiro 1999. 7

[ZCB10] Qi Zhang, Lu Cheng e Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. 11