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, setembro de 2014

Implantação automatizada de composições de serviços web de grande escala

Esta versão da dissertação/tese contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 26/05/2014. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Marco Aurélio Gerosa (orientador) - IME-USP
- Prof. Dr. Fabio Kon (coorientador) - IME-USP
- Prof. Dr. Renato Fontoura de Gusmão Cerqueira - PUC-Rio
- Prof. Dr. Raphael Yokoingawa de Camargo - UFABC

Agradecimentos

Gostaria de agradecer primeiramente a meu orientador, professor Marco Aurélio Gerosa. Pelo acompanhamento e participação contínua em minha pesquisa. Pela cobrança e exigência por um resultado de qualidade. E, principalmente, por me ajudar no desenvolvimento de um maior senso crítico por meio da compreensão do *método científico*. Acredito ser esse o aprendizado mais importante de um mestrando. Estou certo de que esse amadurecimento se aplica não somente à atuação acadêmica, mas à minha vida de forma geral.

Ao professor Fabio Kon, meu co-orientador, agradeço por todo o apoio fornecido à minha pesquisa. No contexto do projeto CHOReOS, o agradeço pelas oportunidades e pela confiança em mim depositada. Por fim, agradeço ao professor Fabio pela parte que lhe cabe em fazer do CCSL um grande local de trabalho, convivência, trocas de experiências e desenvolvimento pessoal.

Agradeço em destaque aos colegas que contribuíram diretamente para o desenvolvimento do Enactment Engine: Daniel Cukier, Thiago Colucci, Felipe Pontes, Alfonso Phocco, Paulo Moura, Carlos Eduardo Santos e Thiago Furtado. Ao Nelson Lago um agradecimento especial por toda a super-ajuda!

Agradeço ao Dr. Daniel Cordeiro pelo auxílio acadêmico recebido durante as escritas de artigos, e ao Maurício de Diana pelo grande aprendizado que tive em nossas várias discussões.

Agradeço também aos colegas do IME com quem tive a oportunidade e privilégio de conviver durante esse período, desde o time do Lab XP (Besson, Guilherme e Piva!) até colegas do projeto CHOReOS e todo o pessoal do laboratório de sistemas (felizmente são muitos!), além de outros que podiam ser encontrados na sala do café :) E há ainda os amigos oriundos do outro lado da rua (a Poli): principalmente Gui, Koga, Tássio e o pessoal do PoliGNU. Obrigado por manterem a camaradagem durante esse período!

Por fim, agradeço à Lari pelo apoio e paciência durante esses anos, e à minha família por ter possibilitado que eu chegasse até aqui.

Valeu galera \o/

Resumo

LEITE, L. A. F. **Implantação automatizada de composições de serviços web de grande escala**. 2014. 116 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2014.

A implantação de composições de serviços web de grande escala apresentam vários desafios, tais como falhas corriqueiras na infraestrutura, heterogeneidade tecnológica, distribuição do sistema por diferentes organizações e atualização frequente dos serviços em operação. Nesta dissertação, estudamos como uma implantação automatizada baseada em middleware pode auxiliar na superação de tais desafios. Para isso, desenvolvemos o CHOReOS Enactment Engine, um sistema de middleware que possibilita a implantação distribuída e automatizada de composições de serviços web em uma infraestrutura virtualizada, operando no modelo de computação em nuvem denominado Plataforma como um Serviço. O middleware desenvolvido é avaliado qualitativamente em comparação a abordagens de implantação *ad-hoc* e quantitativamente pela sua escalabilidade em relação ao tempo de implantação das composições de serviços.

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

Abstract

LEITE, L. A. F. **Automated deployment of large scale web service compositions**. 2014. 116 f. Master thesis - Institute of Mathematics and Statistics, University of Sao Paulo, Brazil, 2014.

The deployment of large-scale service compositions presents several challenges, such as infrastructure failures, technological heterogeneity, distributions across different organizations, and continuous services updating. In this master thesis, we study how the automated deployed supported by middleware can help in overcoming such challenges. For this purpose, we developed the CHOReOS Enactment Engine, a middleware system that enables the distributed and automated deployment of web service compositions in a virtualized infrastructure, operating in the cloud computing model known as Platform as a Service. The developed middleware is evaluated qualitatively by comparing it with *ad-hoc* deployment solutions, and it is also evaluated quantitatively by its scalability regarding the deployment time of service compositions.

Keywords: software deployment, service compositions, 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 Enlace entre serviços 28
4.4 Mapeamento dos serviços na infraestrutura alvo 29
4.5 Interface do Enactment Engine 29
4.6 Pontos de extensão 30
4.7 Tratamento de falhas de terceiros 33
4.8 Aspectos gerais de implementação 34
4.9 Discussão: auxiliando implantações em grande escala 37

5 Avaliação **43**

5.1 Implantando coreografias com e sem o EE 43
5.2 Análise de desempenho e escalabilidade 45
5.3 Limitações dos experimentos 48

6 Conclusões **51**

6.1 Sugestões para trabalhos futuros 52
6.2 Palavras finais 54

A Guia do Usuário do Enactment Engine **57**

Referências Bibliográficas **95**

Lista de Abreviaturas

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
JVM	Java Virtual Machine
LoC	Lines of code
MIL	Module Interconnection Language
MIME	Multipurpose Internet Mail Extensions
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. . . 3

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 Atores envolvidos no uso do Enactment Engine. 24
4.2 Ambiente de execução do CHOReOS Enactment Engine. 24
4.3 Processo de implantação implementado pelo Enactment Engine. 25
4.4 Estrutura da descrição arquitetural de uma coreografia. 27
4.5 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. 34
4.6 Biblioteca de classes para a utilização do Invoker. 35
4.7 Fluxo de threads durante a implantação de coreografias pelo EE. 37

5.1 Topologia das composições utilizadas em nossos experimentos. 45
5.2 Tempos médios de implantação com aumento constante na quantidade de serviços implantados, mantendo-se constante a razão serviços implantados / nós. 47
5.3 Comparação da execução do EE com e sem os mecanismos de tratamento de falhas de terceiros. 48
5.4 Tempo de implantação de coreografias com e sem a utilização dos mecanismos de tratamento de falhas de terceiros. 49

Lista de Tabelas

3.1 Tabela comparativa com os trabalhos relacionados. 21

5.1 Cenários de implantação para o experimento de desempenho. 45
5.2 Resultados do experimento de desempenho. 46

Capítulo 1

Introdução

Princípios que ganharam destaque com os métodos ágeis de desenvolvimento de software vêm expandindo suas fronteiras e se aderindo a outros aspectos do desenvolvimento de um produto. A importância da *iteratividade* é um desses princípios que se tornam evidentes em abordagens modernas de desenvolvimento de produtos, como o *design thinking* [Bro09], ou em abordagens de modelos de negócios, como o *lean startup* [Rie11]. Um dos benefícios em se acelerar o ciclo de iterações é o aumento de *feedback* recebido, o que pode ser usado para se redirecionar e refinar tanto o *o que fazer*, quanto o *como fazer*.

A implantação de um software é o processo que vai da aquisição à execução desse software [OMG06], sendo que a *aquisição* pode corresponder a um desenvolvimento interno na organização que irá implantar o software. Na implantação de sistemas, o princípio de iteratividade ganhou força com as ideias de entrega contínua de software [HF11], o que é uma evolução da prática de integração contínua [DMG07], já incorporada à métodos ágeis como o XP [Bec99]. Entregar software continuamente significa que normalmente cada *commit* no código-fonte, que seja aprovado por uma bateria de testes, corresponde a uma versão potencialmente implantável. E a capacidade de se implantar software continuamente é passo fundamental para que se possa entregar valor mais rapidamente ao negócio.

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, propõem 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, Pap09].

O cenário do aeroporto mencionado acima envolve um grande número de serviços participantes em uma única composição. Por outro lado, há também cenários que envolvem a implantação de uma grande quantidade de pequenas composições. Um exemplo dessa segunda situação é a execução de uma suíte de testes de aceitação automatizados de uma composição de serviços. Cada caso de teste corresponde à implantação de uma instância da composição. Nesse caso é desejável que 1) cada caso de teste seja executado em um ambiente isolado, evitando interferência entre os testes; e 2) a preparação do ambiente para cada caso de teste deve ser facilmente reprodutível, facilitando a execução de vários casos de testes simultâneos e facilitando também a execução da suíte inteira várias vezes seguidas, possibilitando as práticas de integração contínua e entrega contínua [HF11].

No entanto, o desenvolvimento de colaborações entre serviços trazem desafios para a formulação de mecanismos que funcionem, escalem e que sejam eficientemente implementados em ambientes distribuídos de *grande escala* [SPV12]. Em cenários de grande escala, o processo de implantação enfrenta diversas dificuldades, tais como falhas corriqueiras na infraestrutura, heterogeneidade tec-

[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.

nológica, distribuição do sistema por diferentes organizações e atualização frequente dos serviços em operação. Com essas dificuldades, torna-se muito difícil manter a escalabilidade do processo de implantação sem a utilização de um processo de implantação totalmente automatizado, uma vez que processos de implantação manuais tendem a ser morosos, propensos a erros e não reprodutíveis, principalmente na implantação de sistemas distribuídos [DBV05].

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 colocada, 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 execução 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 relação entre os modelos de computação em nuvem e nossa solução pode ser observada na Figura 1.1.

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

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

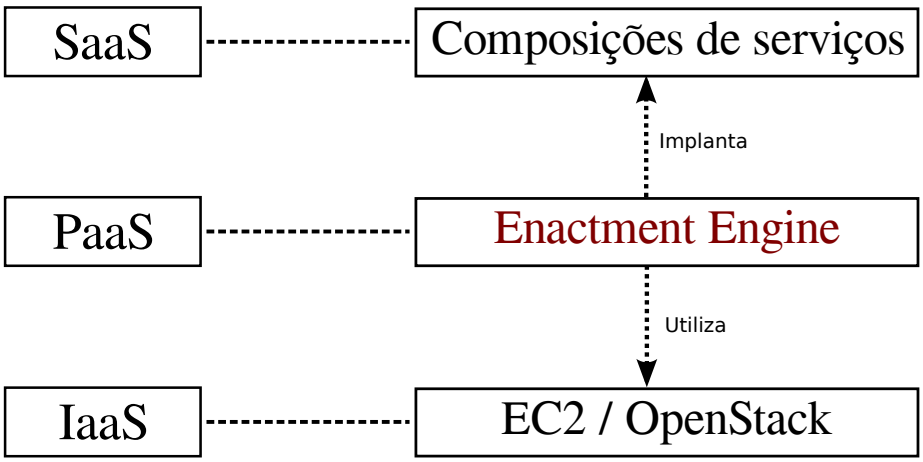

Figura 1.1: *Modelos da computação em nuvem associadas ao CHOReOS Enactment Engine.*

desenvolver um processo dinâmico e centrado no usuário para o desenvolvimento e execução 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.

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 middleware 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 seguinte 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 seguinte 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.

Finalmente, os resultados finais desta pesquisa, como se encontram nesta dissertação, foram publicados no seguinte artigo:

> Leonardo Leite, Carlos Eduardo Moreira, Daniel Cordeiro, Marco Aurélio Gerosa e Fabio Kon. Deploying large-scale service compositions on the cloud with the CHOReOS Enactment Engine. *13th IEEE International Symposium on Network Computing and Applications (NCA)*, 2014.

Esta dissertação foi organizada 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 fundamentam 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 contratual de interface 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 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* [W3C07] e *REST* [Fie00]. 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 (WS-Transaction [Mic02]), troca de endereços de serviços (WS-Addressing [W3C04a]), composição de processos de negócios (WS-BPEL [OAS07]) 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   "id" : 2,
3   "nome" : "Continuous Delivery",
4   "autores" : "Jez Humble and David Farley",
5   "editora" : "Addison-Wesly",
6   "ano" : "2011"
7 }
```

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 estado*[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].

Geralmente serviços REST são considerados mais simples e escaláveis que serviços SOAP por utilizarem diretamente o HTTP como protocolo de aplicação [PZL08]. Mais simples pela existência de um grande conjunto de ferramentas e bibliotecas que já entendem o HTTP. Mais escaláveis, dentre outros motivos, porque o *cache* de serviços REST são diretamente gerenciados pelos servidores web [lT10].

Por outro lado, 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-* e pela existência de interfaces bem definidas e processáveis por software (WSDL).

2.2 Composições de serviços web

Serviços podem ser compostos para implementar sofisticados processos de negócios [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 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

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

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.

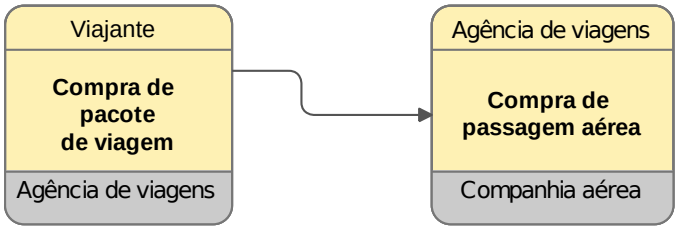

Figura 2.1: *Exemplo de uma pequena coreografia de serviços em notação BPMN2.*

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 o *enlace* entre os componentes de uma composição, para que os componentes conheçam a localização dos componentes dos quais dependem.

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 [DBV05]. Esses problemas fazem da implantação em produção um momento de grande apreensão e mais trabalho nas organizações [HF11]. A solução para esses sintomas, segundo esses 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* de ser executado [HF11].

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 candidato a lançamento. Vencidas todas as etapas, o sistema pode ser implantado no ambiente de produção, ou em alguns casos em um ambiente de homologação. Cada etapa do *pipeline* de implantação pode precisar de uma nova implantação do sistema. Um exemplo básico de pipeline de implantação pode ser visto na Figura 2.2.

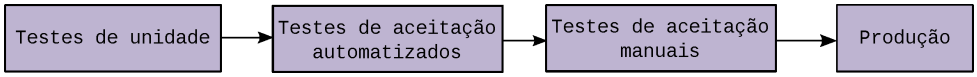

Figura 2.2: *Exemplo básico de pipeline de implantação.*

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 planejamento, 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.

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 equipes interfuncionais viabilizam a 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 se torne 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, por exemplo, 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.
- A quantidade de execuções de testes de integração e aceitação será maior, o que auxilia na garantia de qualidade do sistema.
- A implantação em produção se torna mais confiável, pois o sistema já terá sido implantado várias vezes antes de chegar à produção.

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

- Em particular, espera-se que defeitos no *script* de implantação já tenham sido detectados e corrigidos antes de ser aplicado em produção.

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.

Com a implantação manual, normalmente o sistema é executado no ambiente de produção ou homologação apenas nas fases finais do desenvolvimento. Nesse estágio, grandes mudanças arquiteturais podem ser economicamente inviáveis. Por outro lado, a implantação automatizada favorece a prática da implantação contínua desde as versões embrionárias do sistema. Isso ajuda a garantir desde o início que as decisões arquiteturais são adequadas. Também evita a necessidade de alterações emergenciais para adequar o sistema ao ambiente de produção.

A implantação contínua e confiável do sistema é determinante no 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 pode redefinir os objetivos do sistema. 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, um conjunto de modernas tecnologias com grande impacto sobre a implantação de sistemas.

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.

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

[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/

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 ultra 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.

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 mais confiáveis por evitar erros humanos na operação dos serviços.

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 no Amazon EC2. Cada um dos dez *boxplots* corresponde ao resultado observado para 100 requisições concorrentes ao EC2, cada uma requisitando a criação de uma nova VM. 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.

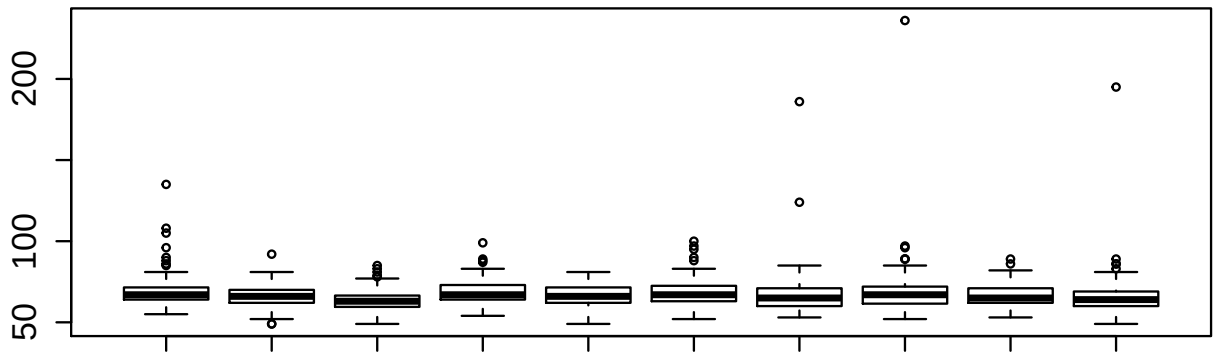

Figura 2.3: *Tempos de criação de instâncias EC2 observados, em segundos.*

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*.

Na Figura 2.3 podemos observar, pelas regiões interquartis dos *boxplots*, que o tempo de criação de VMs possui uma mediana estável. Observamos também que ao se criar ao mesmo tempo uma grande quantidade de nós, é esperado a existência de algumas tempos de criação bem mais demorados, observados nos pontos acima dos *whiskers* superiores.

Apenas o tempo das requisições que foram completadas com sucesso são observadas na Figura 2.3. No entanto, a cada tentativa de se criar simultaneamente várias VMs, nem todas as VMs requisitadas são criadas. Nos experimentos realizados para a produção da Figura 2.3, nós observamos uma taxa de falha de 0.6%. Nesses experimentos, falhas e tempos longos de provisionamento (acima dos *whiskers* superiores) afetaram 7% das requisições de criação de nós.

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*, 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.

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*. Uma operação é idempotente quando executá-la várias vezes produz o mesmo resultado que uma única execução produziria [WRPK07]. Isso implica na capacidade do sistema em tolerar requisições duplicadas, importante para o tratamento eficiente de falhas de comunicação ou de processamento [RV13]. 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 siste-

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

mas [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 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 (camada IaaS) 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, algumas das 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 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 por algum algoritmo adaptativo. Um exemplo de métrica a ser coletada é a taxa de utilização de CPU no hospedeiro do serviço. Coletar tais métricas requer a utilização de um sistema de monitoramento que deve ser implantado 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 implantação automatizada, incluindo algumas ferramentas utilizadas por profissionais da indústria.

Ao utilizar ferramentas de *gerência de configuração* como Chef[1], Capistrano[2] e Nix [DBV05], os usuários devem escrever *scripts* que realizem a configuração do ambiente (sistema operacional e middleware) e a implantação do serviço. No caso do Chef, um *script* (também chamado de *receita*) configura a máquina na qual o serviço é implantado, 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.

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 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

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

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 enlaces dinâmicos 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 no desenvolvimento de 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 enlaces 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 que o operador inicie o processo de implantação com apenas um comando, conforme defendido 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 serviços implantados e de máquinas disponíveis. 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. Embora Watson et al. avaliem o desempenho de serviços operando com o sistema proposto, não avaliam o desempenho ou escalabilidade do próprio processo de implantação.

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.

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 oferece suporte a 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 enlaces 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 enlaces entre os próprios serviços sendo implantados, cenário não considerado pelos atuais provedores de PaaS.

TOSCA (*Topology and Orchestration Specification for Cloud Applications*) é um padrão OASIS que utiliza a abordagem guiada por 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 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 gerenciador de configuração utilizado pelo EE.

Juju[4] é uma ferramenta de configuração e implantação de serviços criada pela empresa 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

[3] http://www.cloudfoundry.com/
[4] https://juju.ubuntu.com/

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 as 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 nos nos alvos 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. As características, que formam as colunas da tabela, são as seguintes:

ADL: especificação da implantação feita de forma declarativa por meio de alguma linguagem de descrição arquitetural;

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 de componentes; a principal diferenciação desse item se refere ao enlace entre os serviços implantados;

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

Heterogeneidade: a solução possibilita a implantação de serviços desenvolvidos com diferentes tecnologias e que utilizem diferentes protocolos de interoperabilidade.

Os símbolos na tabela possuem os seguintes significados:

✓ : possui a característica,

x : não possui a característica,

- : a característica não se aplica e

? : não foi possível determinar.

Trabalho	*ADL*	*Escala*	*Composições*	*Nuvem*	*Heterog.*
Chef	x	-	-	-	-
Capistrano	x	-	-	-	-
Nix [DBV05]	x	x	✓	x	-
Darwin/Regis [MDK94]	✓	x	✓	x	x
Olan [BBB+98]	✓	x	✓	x	x
[QBB+04]	✓	✓	✓	x	x
[ATK05]	✓	x	✓	x	x
[LPP04]	✓	x	✓	x	x
Dynasoar [WFK+06]	-	x	x	x	?
Open Knowledge [BPGR08]	✓	x	✓	x	x
TOSCA [WBB+13]	✓	x	✓	✓	✓
Juju	-	x	x	✓	✓
Cloud Foundry	-	?	x	✓	✓
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 Phocco, 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 guia do usuário (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. Os atores envolvidos no uso do EE são exibidos na Figura 4.1. O ambiente de execução do EE é exibido na Figura 4.2 e os componentes envolvidos são descritos a seguir.

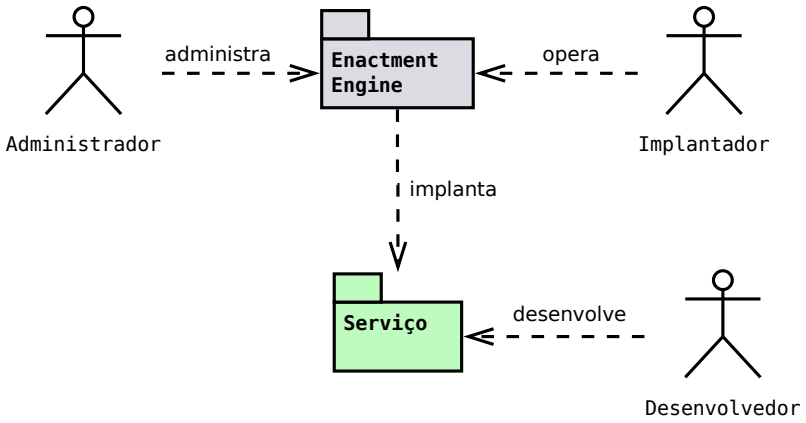

Figura 4.1: *Atores envolvidos no uso do Enactment Engine.*

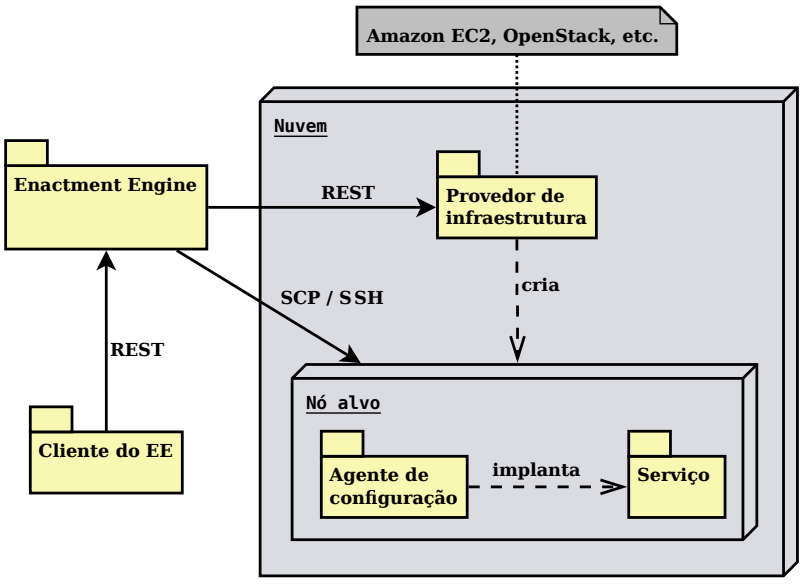

Figura 4.2: *Ambiente de execução do CHOReOS Enactment Engine.*

- 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 programa 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 4.3, e explicado logo em seguida.

A Figura 4.3 exibe o processo de implantação de composições de serviços implementado pelo Enactment Engine:

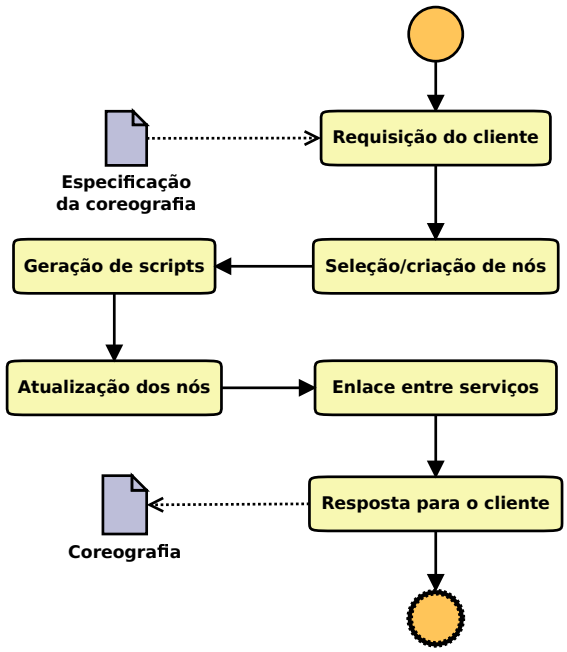

Figura 4.3: *Processo de implantação implementado pelo Enactment Engine.*

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.

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

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

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 por meio de uma *linguagem 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 4.4. 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 esquema XML da linguagem são apresentados no guia do usuário (Apêndice A).

A especificação da coreografia (classes mais claras da Figura 4.4) 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 à escrita de um *script*, no qual normalmente são descritos 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.

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

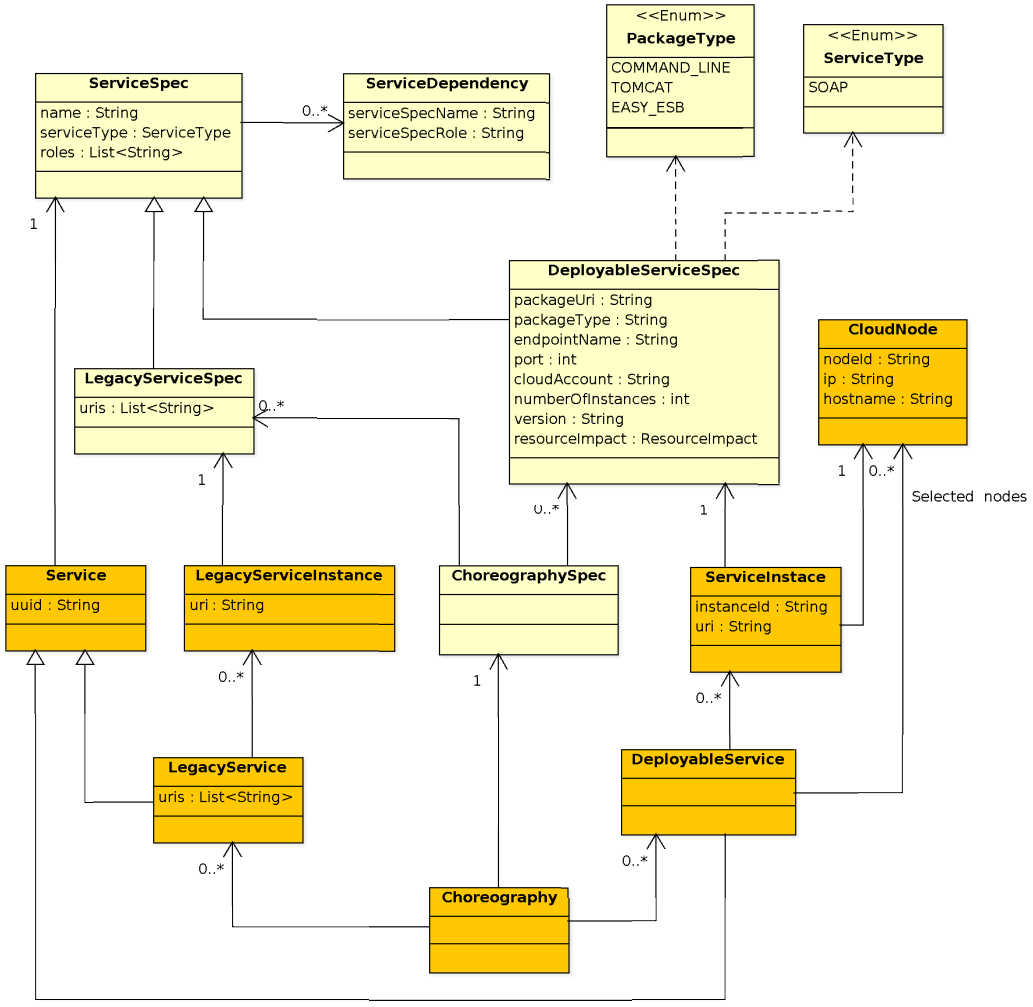

Figura 4.4: *Estrutura da descrição arquitetural de uma coreografia.*

```
1 airportBusCompanySpec =
2   new DeployableServiceSpec(AIRPORT_BUS_COMPANY_NAME,
3                   ServiceType.SOAP,
4                   PackageType.COMMAND_LINE,
5                   resourceImpact,
6                   serviceVersion,
7                   AIRPORT_BUS_COMPANY_JAR_URL,
8                   AIRPORT_BUS_COMPANY_PORT,
9                   AIRPORT_BUS_COMPANY_ENDPOINT,
10                  numberOfReplicas);
11
12 airportBusCompanySpec.setRoles(
13   Collections.singletonList(AIRPORT_BUS_COMPANY_ROLE));
14
15 airportBusCompanySpec.addDependency(
16   new ServiceDependency(AIRPORT_NAME, AIRPORT_ROLE));
```

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

4.3 Enlace 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 B é *provedor* de A. Para que uma coreografia funcione, cada serviço precisa saber o endereço de suas dependências; o processo pelo qual os serviços recebem os endereços de suas dependências é denominado *enlace*.

Segundo Dearle [Dea07], componentes podem ser ligados entre si pelo enlace em vários momentos: compilação, montagem, configuração e execução. Em nosso contexto, o enlace deve ser efetuado 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 do enlace em tempo de execução é a utilização do padrão de injeção de dependências, conforme introduzido por Fowler [Fow04]. A injeção de dependências é utilizada em contêineres como o arcabouço Spring[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 o enlace 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 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://nimbus.com/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,

[6] http://spring.io

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 as operações 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 *planejamento*, 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 de 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 na indústria. 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 estado 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 4.4). 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.

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 4.4). 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.

Na atual implementação, a chamada a essa operação é síncrona, de forma que o cliente fica aguardando a implantação da composição. Caso a implantação de alguns serviços falhem, o cliente continua aguardando normalmente até que a implantação dos outros serviços se completem ou que todas as implantações falhem.

Para que um serviço seja reimplantado utilizando uma nova versão de seu pacote, é preciso utilizar um novo valor no atributo "versão" da especificação do serviço. O novo valor de versão do serviço (atributo de DeployableServiceSpec) indica que alterações foram feitas no código ou na configuração do serviço, sendo isso motivo para sua reimplantação.

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 transações 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 encapsula 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://myhost: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. Nesta seção 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 oferece suporte para 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.

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

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.*

```
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.

Caso preciso, o *cookbook* deve ser também responsável pela implantação do contêiner de execução, (p.ex: Tomcat). Nesses casos, o contêiner já pode conter algumas bibliotecas de uso comum (p.ex: JAX-WS) para evitar que essas bibliotecas sejam carregadas em todos os pacotes implantados. Já um pacote independente de contêiner (p.ex: JAR) deve conter todas as bibliotecas das quais o serviço depende.

A complexidade de se criar uma nova receita Chef vai depender do tipo de tecnologia utilizada, mas a comunidade Chef já fornece diversas receitas prontas para a implantação dos ambientes de execução mais populares. Além disso, pode-se encontrar também na Internet exemplos de implantação de serviços utilizando as tecnologias mais populares.

Tipos de serviços: o enlace 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 se dá 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 Tratamento de 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 4.5). 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.

As invocações externas tratadas pelo Invoker no EE são as seguintes:

- Criação de nó alvo.
- Conexão SSH.
- Envio de arquivo por SCP.
- Instalação do Chef em nó alvo.

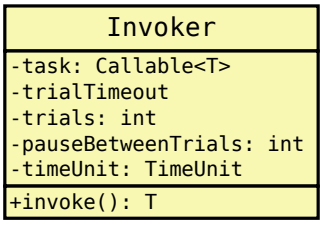

Figura 4.5: *Uma instância de* Invoker *é parametrizada com uma tarefa, uma quantidade de tentativas, um timeout por tentativa e um intervalo de tempo entre as tentativas.*

- Geração de receitas Chef em nó alvo.
- Execução de receitas em nó alvo.
- Chamada à operação `setInvocationAddress` na fase de enlace.

Mais informações sobre a flexibilidade oferecida pelo Invoker, incluindo possibilidade de extensão para adaptação dinâmica dos valores de configuração dos *invokers*, podem ser vistas na Seção 4.8.

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, sendo a reserva preenchida cada vez que a criação de um nó é requisitada. Se o tamanho da reserva é reduzido e alcança um dado limite inferior, 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.

4.8 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. Leitores não interessados nesses aspectos, podem pular diretamente para a Seção 4.9.

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 o 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.*

Invoker As classes criadas para a utilização do Invoker (Figura 4.6) foram projetadas para fornecer uma utilização flexível do conceito de *invoker*. Pode-se adotar diferentes estratégias para determinar os valores de configuração de um *invoker*, assim como diferentes estratégias de tratamento de erro implementada pelo *invoker*.

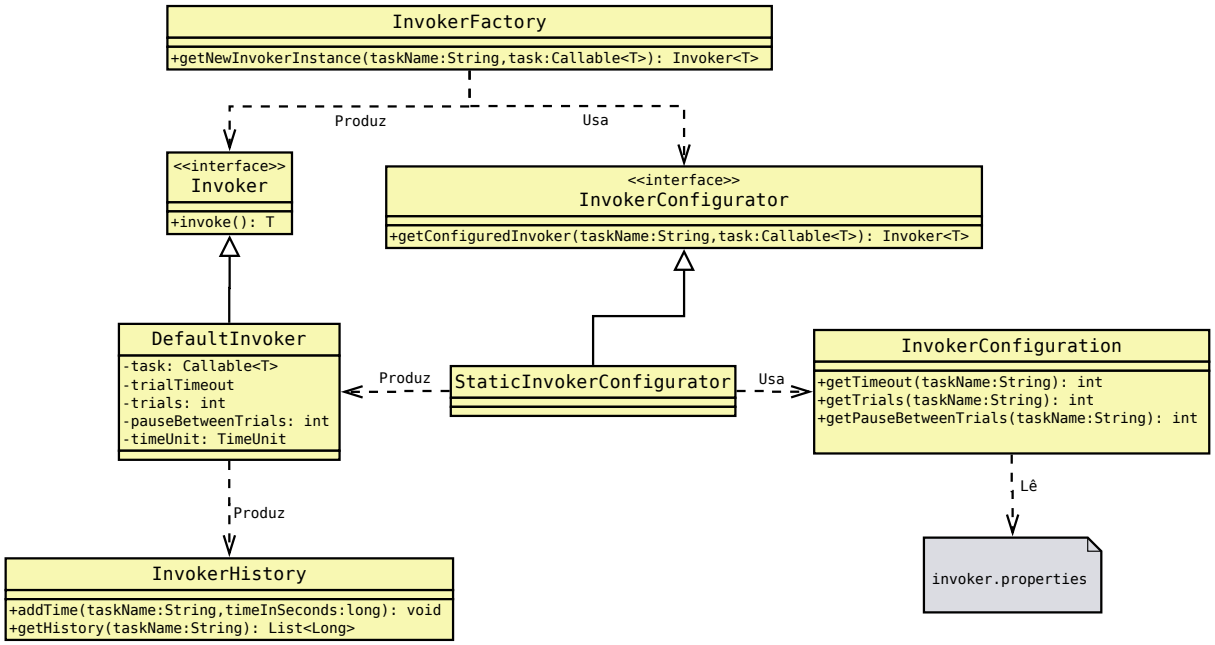

Figura 4.6: *Biblioteca de classes para a utilização do Invoker.*

A implementação utilizada de InvokerConfigurator produz *invokers* utilizando os valores configurados no arquivo `invoker.properties`. Para que os valores de configuração dos *invokers* sejam ajustados dinamicamente, pode-se implementar um novo InvokerConfigurator que utilize algum algoritmo adaptativo com base no histórico de execução dos *invokers*, que é produzido pelo próprio Invoker.

Pode-se também desejar a criação de novas implementações de Invoker. Se, por exemplo, o sistema for portado pra um ambiente mais confiável, como um aglomerado utilizando *infiniband*, podemos fazer com que a fábrica de *invokers* retorne um "*invoker* curto-circuito", que nada fará a não ser invocar a tarefa especificada. Tratamentos mais complexos que o da implementação atual também podem ser concebidos. A vantagem do desenho utilizado é que essa troca de estratégia pode ser realizada sem que seja preciso alterar o código dos clientes do *invoker*.

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) o enlace 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 no *enlace 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`.

Fluxo de threads: A Figura 4.7 ilustra a estratégia de concorrência que o EE implementa durante a implantação de coreografias. A execução é dividida em três fases: 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) o enlace entre serviços. Cada fase de execução é executada por várias *threads*. Nas fases de preparação e de enlace o EE dispara uma *thread* diferente para cada serviço a ser implantado. Já na fase de atualização dos nós, há uma *thread* para nó a ser atualizado. Durante a preparação, novas *threads* são abertas, uma para cada nó a ser criado a fim de hospedar as instâncias de um serviço. Os *scripts* de implantação de um serviço são transferidos para os nós alvos somente depois que todos os nós alvos do serviço estão criados.

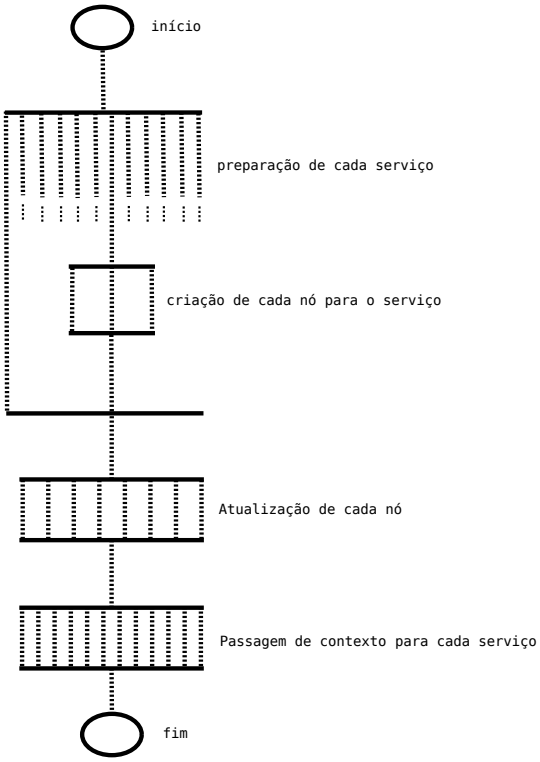

Figura 4.7: *Fluxo de threads durante a implantação de coreografias pelo EE.*

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

4.9 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: Automatizar a implantação de sistemas 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 à 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. O enlace 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: Conforme apresentado na Seção 4.7, o Enactment Engine adota duas principais estratégias para lidar com falhas de componentes de terceiros ou falhas na rede. Essas estratégias são o uso do *invoker* e da *reserva de nós ociosos*.

O uso dessas estratégias caracterizam a aplicação de interesses transversais durante a invocação de um sistema externo. Deixar que o middleware implemente interesses transversais traz uma maior robustez ao sistema, pois evita-se que o implantador esqueça de utilizar tais mecanismos em alguns pontos do sistema.

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 4.8.

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[8] 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.

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 enlace entre serviços, pois o EE

[8]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.

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 S_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 autônoma 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 enlace 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 podem ser enviadas a um serviço previamente configurado no EE (embora o EE faça a implantação do Ganglia nos nós alvos, o serviço que processará os dados coletados deve ser informado ao 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 profissionais da indústria 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

Como já mencionado na introdução, nosso objetivo nesta dissertação é projetar e implementar um middleware que forneça suporte à implantação automatizada de composições de serviços web de grande escala. Realizamos a implementação desse middleware, o Enactment Engine, com o propósito de entender como soluções de implantação baseadas em middleware contribuem para a melhoria do processo de implantação quando confrontadas com soluções *ad-hoc*. Na Seção 4.9 discutimos os benefícios trazidos por soluções de implantação baseadas em middleware com base nos desafios dos cenários de grande escala. Neste capítulo, apresentaremos avaliações a fim de verificar empiricamente 1) a vantagem do uso do EE sobre soluções *ad-hoc* e 2) a aplicabilidade do EE em contextos de grande escala.

Primeiramente apresentamos um estudo qualitativo apresentando uma comparação entre o processo de implantação utilizando o EE e um processo de implantação *ad-hoc*. Com esse estudo, evidenciamos as vantagens de soluções de implantação baseadas em middleware sobre soluções *ad-hoc*. O estudo qualitativo foi realizado com uma comparação do "esforço" do implantador ao implantar uma coreografia com e sem o EE. Medimos o "esforço" com a quantidade de código escrito pelo desenvolvedor, o tempo para a escrita desse código e o tempo de execução da implantação.

Para avaliar a aplicabilidade do EE em contextos de grande escala, avaliamos quantitativamente o desempenho, escalabilidade e quantidade de falhas do EE. Nessas avaliações, criamos experimentos automatizados responsáveis por 1) invocar o EE com diferentes parâmetros, 2) limpar o ambiente a cada execução (apagar VMs da nuvem e reiniciar o EE), 3) medir e registrar os tempos de execução e quantidade de falhas ocorridas e 4) enviar notificações por e-mail sobre o andamento do experimento a cada execução.

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 em menos de 5 minutos. 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, e 85 LoC de Ruby (para o Chef).

No restante desta 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.

Implementação da solução ad-hoc 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 o enlace entre serviços, desenvolvemos um pequeno mas não trivial programa Java, chamado `context_sender`. Ele é responsável pela invocação da operação `setInvocation Address` 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 (via 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. Conhecimento de 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 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 ferramentas que auxiliam na execução de comandos remotos, como o Capistrano[1] por exemplo, 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.

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

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 avaliação preliminar do esforço necessário para se implantar composições de serviços.

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.

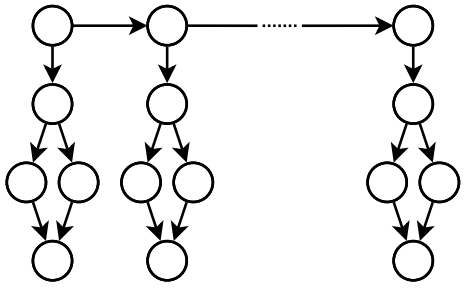

Figura 5.1: *Topologia das composições utilizadas em nossos experimentos.*

Inicialmente, conduzimos um experimento de 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.

Tabela 5.1: *Cenários de implantação para o experimento de desempenho.*

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

Nesse experimento, 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 execução. Se a

Tabela 5.2: *Resultados do experimento de desempenho.*

Cenário	*Tempo* (s)	*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%)

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 executando 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 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 relativamente bem em termos de serviços sendo implantados. Do cenário 1 para os cenários 2 e 3, o número de serviços (*Composições * Tamanho*) foi multiplicado por 10 (Tabela 5.1), e em ambas as situações o tempo de implantação cresceu aproximadamente apenas 3 vezes (Tabela 5.2). Parte da explicação para esse incremento no tempo de implantação está na ocorrência de falhas da infraestrutura. Quanto maior a quantidade de serviços implantados, maiores são as chances da ocorrência dessas falhas. E para cada falha que acontece, uma nova tentativa de execução da rotina correspondente é necessária (ver sobre o *invoker* na Seção 4.7).

Indo do cenário 1 para o cenário 4, o número de serviços por nó dobrou (Tabela 5.1). Nessa situação, os resultados da Tabela (Tabela 5.2) mostram que 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.

Nesse experimento, 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 um segundo experimento para avaliar o desempenho e escalabilidade do Enactment Engine em termos de sua capacidade de implantar grandes composições de serviços. Esse experimento foi realizado 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 Universi-

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

dade. 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 Figura 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.

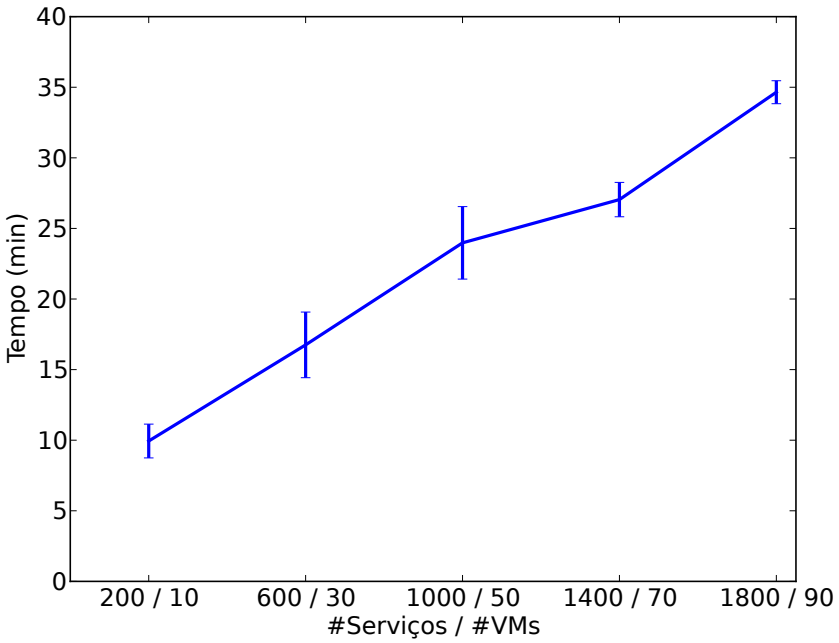

Figura 5.2: *Tempos médios de implantação com aumento constante na quantidade de serviços implantados, mantendo-se constante a razão serviços implantados / nós.*

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.

Para termos uma melhor ideia do impacto causado pela utilização dos mecanismos de tratamento de falhas do EE, realizamos mais um experimento. Nesse experimento realizamos 10 execuções em que 1 coreografia de 100 serviços é implantada em 10 nós. Esse procedimento foi realizado com duas variações: na primeira variação o *invoker* e a reserva de nós ociosos estavam ativados, enquanto que na segunda variação ambos estavam desativados.

Nesse experimento, a topologia utilizada na composição foi a mesma dos experimentos anteriores (Figura 5.1). O EE foi executado em uma máquina virtual (4 GB de RAM e 2 vCPUs de 2.3 GHz) 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 300 segundos.

Os resultados podem ser vistos na Figura 5.3. Ao utilizar os mecanismos de tratamento de falhas de terceiros fornecidos pelo EE, todos os serviços foram adequadamente implantados em todas as execuções. Por outro lado, ao não utilizar esses mecanismos, obtivemos execuções em que nem todos os serviços foram corretamente implantados.

Para as execuções realizadas, observamos que a quantidade de serviços falhos por execução foram múltiplos de 10 (10 ou 30 serviços falhos). Esse padrão sugere que as falhas observadas ocorreram devido à falha na criação de máquinas virtuais, uma vez que cada nó deveria hospedar 10 serviços. Essa consideração evidencia como a falha de criação de VMs é um dos tipos de falhas mais importantes a serem tratados, justificando o tratamento especial que demos a esse tipo de falha ao criar a reserva de nós ociosos.

Para termos uma ideia da sobrecarga causada pela utilização dos mecanismos de tratamento de falhas de terceiros, geramos o gráfico apresentado na Figura 5.4. Esse gráfico é um detalhamento do

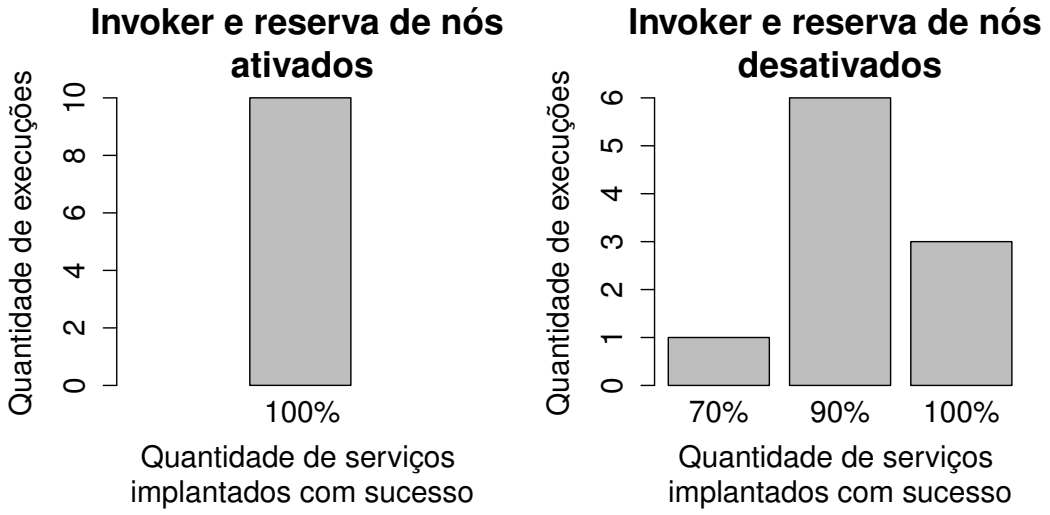

Figura 5.3: *Comparação da execução do EE com e sem os mecanismos de tratamento de falhas de terceiros.*

gráfico já apresentado na Figura 5.3, mas apresentando os tempos gastos em cada execução. Assim, cada ponto no gráfico fornece o tempo gasto que a execução levou para tentar implantar os 100 serviços da coreografia.

Pelo gráfico, observamos que o espalhamento vertical dos dois grupos de pontos (usando e não usando os mecanismos de tratamento de falhas de terceiros) ocupa aproximadamente o mesmo espaço vertical, desconsiderando apenas uma execução, que levou mais de 700 segundos. Isto é, podemos constatar visualmente que não houve uma diferença significativa entre os tempos de execução obtidos para os dois grupos de amostras.

5.3 Limitações dos experimentos

Nesta seção iremos explicitar as limitações que julgamos presentes em nossos experimentos.

No estudo qualitativo (Seção 5.1) avaliamos os benefícios de uma implantação baseada em middleware do ponto de vista do desenvolvedor. Algumas de suas limitações são:

- avaliação fortemente dependente da expertise do implantador;
- avaliação dependente das tecnologias usadas na solução *ad-hoc*;
- avaliação dependente do tamanho da coreografia implementada.

Já nas avaliações qualitativas (Seção 5.2), uma primeira limitação é não-reprodutibilidade dos resultados experimentais, uma vez que os experimentos foram executados em um ambiente de nuvem. Contudo, ao utilizar a mesma plataforma de nuvem com os mesmos tipos de VMs, esperamos que resultados similares sejam encontrados em uma eventual replicação de nossos experimentos.

Outra característica limitante foi a falta de análise em camadas mais baixas dos protocolos de redes para explicar o porquê de certos comportamentos, como por exemplos os resultados exibidos na Figura 5.2.

Por fim, mais esclarecimentos poderiam ter sido fornecidos com mais variações experimentais. Por exemplo, o experimento de escalabilidade (Figura 5.2) poderia ter sido feito também 1) sem a utilização da reserva de nós ociosos, 2) sem a utilização do *invoker* e 3) sem a utilização tanto da reserva quanto do *invoker*. Assim entenderíamos mais precisamente a contribuição desses mecanismos. Poderia haver também mais cenários para a análise de desempenho (Tabela 5.1), de forma a fornecer mais evidências sobre a participação de cada fator experimental nos resultados.

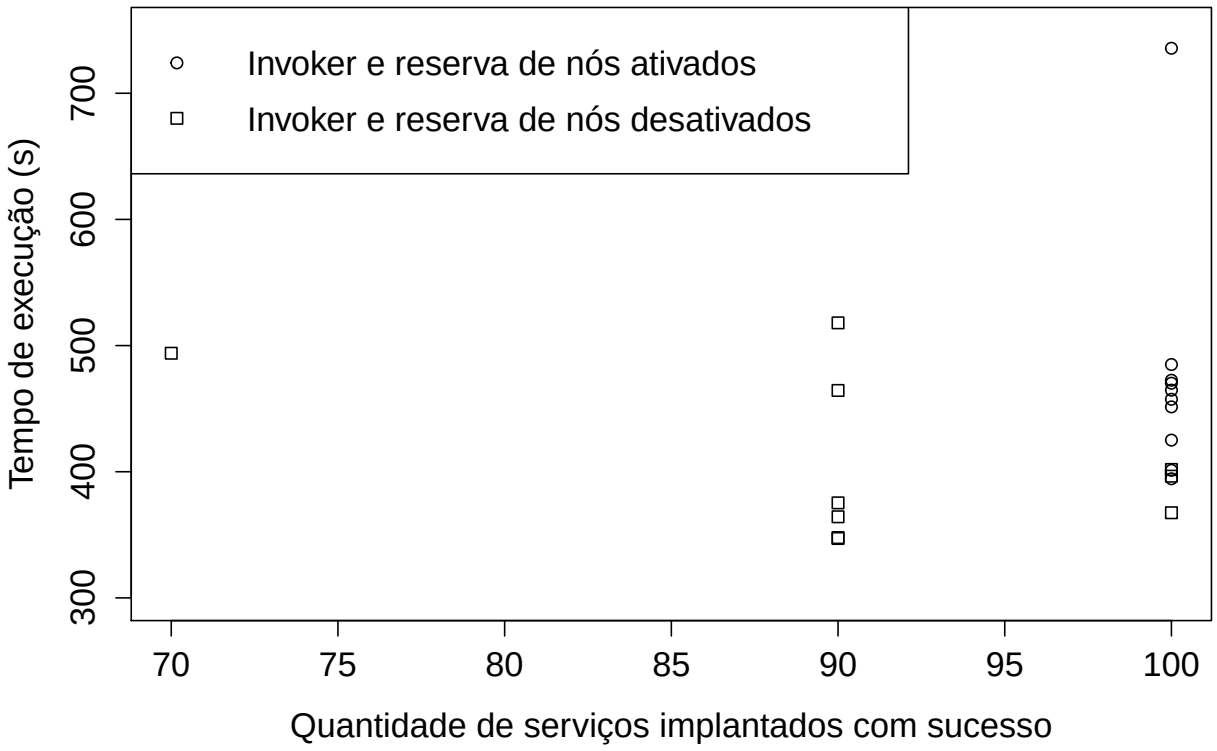

Figura 5.4: *Tempo de implantação de coreografias com e sem a utilização dos mecanismos de tratamento de falhas de terceiros.*

O grande impedimento para a execução de mais variações experimentais foi o custo dos experimentos. O custo por VM criada em nossos experimentos era de U$0,06. Executando-se apenas um experimento com a criação de 100 VMs em 30 execuções chega-se ao valor de U$180,00. Dado o alto custo dos experimentos, foi preciso considerar um compromisso entre a quantidade e a variedade dos experimentos realizados e a eventual contribuição que os resultados trariam no contexto desta pesquisa.

Capítulo 6

Conclusões

Nesta dissertação, nós introduzimos o CHOReOS Enactment Engine, um novo sistema de middleware que facilita a implantação de composições de serviços de grande escala em um ambiente de computação em nuvem.

Embora a automação do processo de implantação de composições de serviços possa ser implementada de maneiras *ad-hoc*, esse tipo de solução normalmente requer do implantador uma grande gama de conhecimentos técnicos. Por meio de experimentos, procuramos evidenciar como uma abordagem baseada em middleware, como o EE, reduz o tempo de trabalho necessário para a automação do processo de implantação, considerando tanto o tempo de desenvolvimento da solução de implantação, quanto o tempo de implantação de cada composição.

Ao revisar a literatura, identificamos desafios na implantação de sistemas de grande escala. Procuramos atender tais desafios ao aplicar no EE técnicas e soluções espalhadas na literatura, bem como refinar tais soluções com base em nossas experimentações empíricas. Listamos a seguir considerações sobre os desafios na implantação de sistemas de grande escala e sobre como o uso de uma solução de implantação baseadas em middleware contribui para a superação de tais desafios.

- Identificamos que a automação é essencial para a implantação de sistemas de grande escala. Para isso é importante que o middleware de implantação forneça uma API remota para disparar as implantações. Nesse sentido, também contribui a especificação das composições por meio de descrições declarativas de suas arquiteturas.

- Ao se tratar de software distribuído de grande escala, falhas nas interações com componentes e serviços de terceiros se tornam comum. Implementar estratégias de tratamento de falhas de terceiros na camada do middleware é um grande auxílio no desenvolvimento de uma solução robusta e escalável.

- Não só o tratamento de falhas de terceiros, mas o tratamento de concorrência pelo middleware é importante também para a obtenção de um resultado robusto e escalável, sem que o implantador tenha que entender a fundo essas questões complicadas.

- Soluções *ad-hoc* levam vantagem na questão de suportar soluções heterogêneas. Contudo, soluções baseadas em middleware podem oferecer suporte pronto para as tecnologias mais utilizadas do mercado. Além disso, uma arquitetura extensível auxilia no desenvolvimento de suporte a novas tecnologias, de forma que o resultado de cada nova extensão é compartilhável por todos os usuários do middleware.

- Composições de serviços podem ser compostos por serviços de diferentes organizações. Um middleware de implantação deve considerar a colaboração dos serviços por ele implantado com serviços já existentes.

- A evolução auto-adaptativa de composições de serviços é assunto bastante estudado. A implantação automatizada, robusta e escalável é necessária para o desenvolvimento de composições

de serviços auto-adaptativas. Assim, mesmo que um middleare de implantação não forneça mecanismos de evolução auto-adaptativos para suas implantações, ele é peça importante que pode facilitar o desenvolvimento de novos sistemas adaptativos.

Para evidenciar os benefícios das técnicas aplicadas, realizamos avaliações empíricas do desempenho e escalabilidade da implantação de composições de serviços de grande escala operada pelo EE. Os resultados experimentais evidenciam a aplicabilidade da arquitetura proposta e que desempenho e escalabilidade satisfatórios podem ser obtidos.

Acreditamos que as conclusões já listadas e mais algumas lições aprendidas no desenvolvimento do EE e desta dissertação possam auxiliar no desenvolvimento de outras soluções baseadas em middleware no contexto de computação de grande escala. Listamos agora algumas dessas lições aprendidas:

- Arquiteturas escaláveis não devem depender de um ponto central de processamento. Em versões anteriores do EE, a utilização do Chef Server impunha grandes restrições à escalabilidade do processo de implantação. Em um estágio mais anterior de nossa pesquisa acreditávamos que o Chef Server não seria um problema, pois para cada requisição seu trabalho era extremamente simples e rápido. Mas vários detalhes mostraram o contrário, como 1) a grande quantidade de requisições feitas ao Chef Server, vindas do EE e dos nós alvos; 2) a quantidade de memória RAM gasta no EE para se invocar o Chef Server, uma vez que pra cada requisição criávamos uma nova instância de um programa Ruby (o *knife*); e 3) o sistema de filas utilizado pelo Chef Server para receber suas requisições prejudicava o tempo de resposta dessas requisições.

- Embora defendida pela literatura [Ham07] e considerada um valor em nosso grupo, a *simplicidade* é difícil de por em prática. Desenvolvimento iterativo que a cada iteração procure fornecer *a coisa mais simples que funcione* é importante também no desenvolvimento de sistemas de grande escala. No caso do EE, o projeto inicial previa a divisão do EE em três módulos que se comunicariam por serviços. Com o passar do tempo vimos que esse desenho impunha diversas complicações, e em um primeiro momento passamos de três para dois módulos, e em um estágio mais final do desenvolvimento reduzimos os dois módulos a apenas um. Uma das dificuldades da divisão anterior era a da sincronia entre estruturas de dados replicadas em diferentes módulos. No fim, essa simplificação arquitetural não impediu a aplicação de um bom projeto de classes, e nem impede, no futuro, a possibilidade de se replicar as instâncias do EE para que se suporte uma carga maior de requisições.

6.1 Sugestões para trabalhos futuros

Listamos agora alguns possíveis trabalhos futuros envolvendo o Enactment Engine.

Análise multivariável de fatores que influenciam a escalabilidade. Outro experimento para melhor entender o desempenho e escalabilidade do EE seria aplicar uma análise multivariável para determinar o quanto que o tempo de implantação é influenciado por fatores como a quantidade de composições sendo implantada, a quantidade de serviços em cada composição e a quantidade de nós disponíveis. Nesse sentido, começamos a realizar esse experimento utilizando a análise fatorial 2^k com replicação [Jai91], mas dificuldades com a distribuição dos dados e o alto custo para se obter novas amostras dificultaram a conclusão desse experimento.

Identificação e eliminação de gargalos internos do EE. Casado com a proposta de trabalho anterior, estaria a melhoria no fluxo de execução de *threads* do EE (Seção 4.8), evitando gargalos na escalabilidade do processo de implantação. Uma estratégia seria a eliminação das barreiras existentes (visíveis na Figura 4.7) para que falhas em uma determinada *thread* não atrasem a implantação de serviços não relacionados com essa *thread*. Exemplo: hoje o enlace

dos serviços só é efetuado após a implantação de todos os serviços. Mas se a implantação de apenas um serviço falhar, isso será motivo para atrasar o enlace entre todos os outros serviços. A atualização de um nó também poderia se dar assim que todos os serviços destinados ao nó já estejam preparados. Ou seja, o funcionamento parcial de uma coreografia poderia ser antecipado com um retrabalho nos fluxos de execução de *threads*.

Utilizar mais chamadas assíncronas. Na linha de eliminar gargalos e diminuir a fragilidade do código, outros pontos do código do EE poderiam ser mais assíncronos. Por exemplo, em vez de uma *thread* esperar outra criar uma VM, a *thread* que cria uma VM poderia acionar um *callback* para continuar o fluxo de execução. Além disso, o cliente da API REST de criação de nós fica bloqueado por minutos esperando uma resposta HTTP. Nesse caso deveria se usar os mecanismos REST para criação demorada de recursos (código de estado 202). O mesmo princípio vale para a própria operação de implantação dos serviços, que atualmente possui uma implementação síncrona.

Experimentos com desenvolvedores. Na Seção 5.1 realizamos uma avaliação qualitativa para ajudar a expandir nosso entendimento sobre o valor que o EE agrega ao processo de implantação. Dada as limitações de nosso experimento, seria interessante expandi-lo com a participação de diversos desenvolvedores de software e administradores de sistemas assumindo o papel de implantadores de uma composição de serviços. Nesse caso, a ideia seria utilizar uma abordagem mais rigorosa, dentro das possibilidades de experimentos de engenharia de software. Comparações com outros arcabouços de implantação também poderiam ser realizadas.

Desenvolvimento incremental com middleware. Considerando a utilização de prática ágeis, que pregam pela simplicidade ("a coisa mais simples que funcione"), seria razoável considerar que desenvolvedores de uma coreografia não utilizassem um middleware como o Enactment Engine em um primeiro momento. Seria interessante estudar como as características de um middleware como EE poderiam auxiliar os desenvolvedores no momento da transição, quando decidem adotar o uso do middleware.

Algoritmos adaptativos para tratamento de falhas. Acreditamos que os algoritmos do EE que tratam falhas de terceiros podem ser melhorados. Tanto a reserva de nós ociosos quanto o *invoker* são adequados para utilizarem algoritmos adaptativos que aprendem com o histórico de execuções. Assim, a reserva de nós ociosos poderia alterar seu tamanho dinamicamente, evitando desperdícios de VMs extras. Da mesma forma, o *invoker* poderia utilizar valores de *timeout* mais adequados, evitando longas esperas desnecessárias ou desistindo de tarefas que logo estariam prontas. Um desafio interessante para a adaptação dinâmica do *invoker* é considerar a alteração dinâmica de suas três propriedades: *timeout*, quantidade de tentativas e tempo de pausa entre as tentativas.

Outras estratégias para a reserva de nós ociosos. Outros algoritmos adaptativos para reger o aumento e diminuição do *reservoir* também podem ser experimentados e avaliados. Até mesmo a estratégia geral do *reservoir* poderia ser revista: em vez de apenas conter nós a serem usados em caso de falha de provisionamento de um nó, o *reservoir* poderia já conter uma quantidade de nós o suficiente para ser utilizado por todos os serviços implantados, de forma que o processo de implantação de cada serviço já encontrasse um nó pronto para uso.

Federação de instâncias do EE. Uma instância do EE realiza a implantação de serviços pertencentes a uma dada organização. Se algum dos serviços implantados depende de um serviço pertencente a outra organização, o implantador pode modelar esse serviço de terceiros como um *serviço legado* na modelagem da composição. O problema é que mudanças nos serviços de terceiros, como mudança de URI, podem causar impactos na composição implantada. Além disso, se serviços interdependentes de diferentes organizações são implantados em paralelo, torna-se difícil utilizar o recurso de *serviço legado*, uma vez que o implantador de uma organização ainda não dispõe das URIs dos serviços sendo implantados pela outra organização.

Para tornar a implantação de composições inter-organizacionais mais dinâmica, um caminho promissor é a federação de instâncias do EE. Assim, uma instância pertencente a uma organização A pode avisar a uma outra instância pertencente a uma organização B sobre mudanças nas coreografias de A que possam impactar as coreografias pertencentes a B.

Utilização de um balanceador de carga. Na atual implementação do EE, quando um serviço é replicado em várias instâncias, seus clientes recebem as URIs de todas as instâncias disponíveis do serviço. Assim, cabe a cada cliente distribuir a carga pelas diferentes réplicas disponíveis. No entanto, melhor seria que o EE implantasse um balanceador de carga que distribuísse as requisições entre as diferentes instâncias do serviço. Assim, os clientes receberiam apenas uma URI, que seria a URI do balanceador de carga.

Utilização de um barramento de serviços. Caso um serviço dependa de outro serviço com um protocolo de comunicação diferente, o EE assume que é de responsabilidade do serviço cliente conhecer o protocolo do serviço provedor. Contudo, para facilitar a implementação dos serviços, a conversão entre diferentes protocolos de comunicação poderia ser tratada por um barramento de serviços. Assim, uma possibilidade seria de que o EE implantasse automaticamente instâncias de um barramento de serviços para interligar serviços de protocolos diferentes. No entanto, para essa tarefa é necessário a utilização de um barramento de serviços que considere a natureza dinâmica de ambientes de computação em nuvem, onde serviços são replicados e passam a possuir múltiplas URIs.

Aprimorar a atualização de coreografias. Embora tenhamos implementado a operação de atualização de uma coreografia, ao longo deste trabalho houve um foco muito grande na primeira implantação de uma coreografia. Como exemplo, todos os experimentos consideraram apenas o cenário de primeira implantação. Seria então importante investir no aprimoramento do processo de atualização, principalmente utilizando novos e diferentes casos de teste que talvez exponham cenários ainda não considerados.

Atualização dinâmica de composições de serviços. Na atual implementação do EE, a atualização de coreografias pode ocasionar falhas em transações correntes, o que ocorre mesmo com serviços sem estado. Vários trabalhos [MK90, VEBD07, MBG+11] estudam o processo de atualização dinâmica, pelo qual as transações correntes são preservadas durante a atualização de um serviço. Acreditamos que seria muito interessante possibilitar ao administrador do EE a definição de políticas e algoritmos de atualização para tratar da adequada finalização das transações correntes. Dessa forma, o EE poderia se tornar uma plataforma de apoio à comparação empírica entre diferentes estratégias presentes na literatura.

Outras pendências menores estão registradas na página de *issues* do Enactment Engine[1]. Colaborações técnicas e de pesquisa sobre as possibilidades levantadas são bem vindas.

6.2 Palavras finais

Composições de serviços se mostram promissoras em possibilitar a integração de sistemas de grande escala. Tomando um exemplo governamental em que fosse necessário uma interação entre governo federal e todos os municípios do Brasil, já teríamos um cenário de potencial utilização de composições de serviços de grande escala, em que os serviços participantes pertencem a organizações diferentes.

Em sistemas de grande escala, as incertezas iniciais são maiores do que em pequenos projetos. Portanto, nesse contexto a aplicação de abordagens iterativas se torna mais promissora do que grandes planos inicias altamente detalhados. No entanto, implantar sistemas distribuídos de grande escala é tarefa difícil, sendo a automação do processo de implantação condição fundamental para a

[1] https://github.com/choreos/enactment_engine/issues?state=open

capacidade de entrega contínua. Na intenção de repetir continuamente o processo de implantação, a computação em nuvem surge como poderosa aliada. A automação da criação e remoção de nós virtualizados facilita que o processo de implantação seja executado continuamente.

No futuro, esperamos uma maior integração entre as organizações de forma automatizada e em grande escala. Para que isso funcione, a capacidade de implantação contínua dessas composições será importante para que esses sistemas possam evoluir adequadamente. Esperamos que as ideias apresentadas nesta dissertação possam ser de auxílio a implementadores e implantadores de tais integrações de grande escala. Acreditamos também que o Enactment Engine desenvolvido no contexto desta dissertação possa apoiar outras pesquisas sobre composições de serviços, pesquisas essas que também podem auxiliar na viabilização do cenário futuro de alta integração entre as organizações.

Apêndice A

Guia do Usuário do Enactment Engine

O Guia do Usuário do Enactment Engine contido neste apêndice é um documento independente desta dissertação, voltado aos interessados em utilizar o EE. Ele está aqui presente para fornecer aos leitores da dissertação um melhor entendimento sobre o Enactment Engine. A versão aqui contida é a mais recente em março de 2014, mas uma possível versão mais atualizada do guia pode ser encontrada em http://ccsl.ime.usp.br/enactmentengine.

O guia possui os seguintes capítulos:

1. *Installation Guide*
2. *Enactment Engine API*
3. *How to package services to be deployed by the Enactment Engine*
4. *Extending Enactment Engine*
5. *Elasticity and QoS management*

Chapter 1

Installation Guide

1.1 Introduction

The CHOReOS Enactment Engine (EE) provides a Platform as a Service (PaaS) that automates the distributed deployment of service choreographies in cloud environments. This chapter is targeted mainly to EE *administrators*, providing instructions about how to install, configure, and run the Enactment Engine.

We will describe now each one of the components running on the Enactment Engine execution environment. They are depicted in Fig. 1.1.

Infrastructure provider creates and destroys virtual machines (also called *nodes*) in a cloud computing environment. Currently, only Amazon EC2 and OpenStack are supported as infrastructure providers, but the Enactment Engine can be extended to support other virtualization technologies.

Chef-solo is installed by the Enactment Engine in each cloud node to manage "recipes" execution. *Chef recipes* are scripts written in a Ruby-like Domain Specific Language that implement the process of configuring operational system, installing required middleware, and finally deploying the services.

EE cliente is a script, written by deployers, that specifies the choreography deployment and invokes the EE to trigger the deployment process. *Deployer* is the human operator responsible by the deployment process.

Enactment Engine deploys choreography services according to the specification sent by the client.

1.2 Requirements

Before you run Enactment Engine, you will need:

- Git;
- Java 6 or later (we are using OpenJDK);
- Maven 3 (`http://maven.apache.org/download.html`);
- access to Infrastructure Provider services, as detailed in Section 1.3.

1.3 Cloud Provider

A CloudProvider is an Enactment Engine interface that specifies methods to CRUD virtual machines. It is expected that a Cloud Provider implementations will act just as a client of some Infrastructure Provider. In this section we describe the currently available CloudProvider implementations and how to use them. New CloudProviders may be implemented to support other virtualization tools. One example would be creating a VirtualBoxCloudProvider to create VMs using VirtualBox.

Whatever the cloud provider you choose, ensure that the required TCP ports of the created VMs are unblocked. Required ports: 22 to SSH, 8080, 8009, and 8005 to Tomcat, the ports used by your JAR services, and the port 8180 to EasyESB.

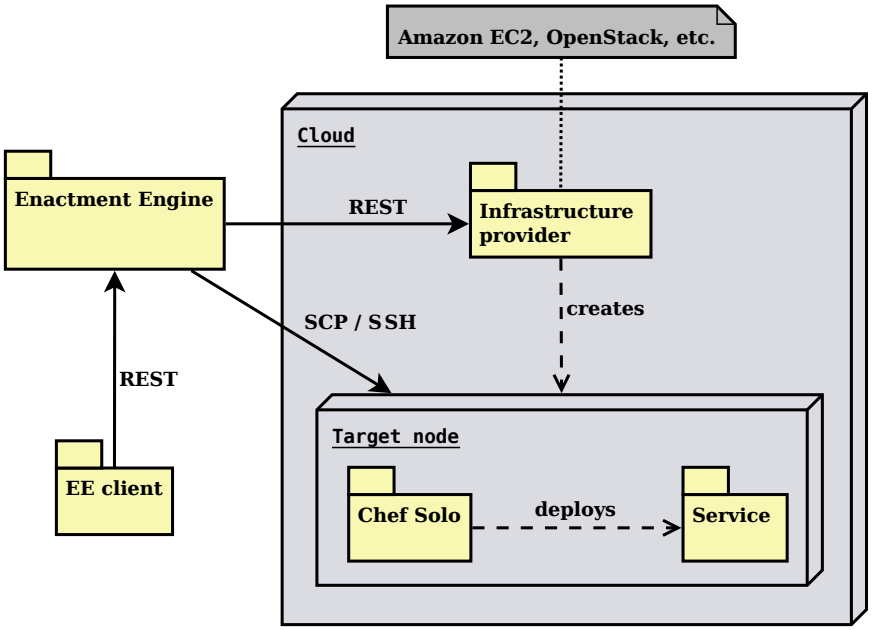

Figure 1.1: Enactment Engine execution environment.

1.3.1 Amazon EC2

Amazon EC2 service is the simplest choice to dynamically retrieve VMs as you need. You need just to create an account at `http://aws.amazon.com` and configure a pair of keys to access the VMs through SSH. The trade-off is that you must pay to Amazon!

Some hints:

- Request credits to education purposes: `http://aws.amazon.com/grants/`. The first of us earned $500, but the others $100. Maybe it depends on your research project description.
- Initially there is a limit of 20 VMs that you can run simultaneously.
- Request to increase Amazon EC2 instance limit: `http://aws.amazon.com/contact-us/ec2-request/`. At USP we got a 50 VMs limit.
- If you are going to use the EC2 API directly, pay attention to the "one-second rule": `http://www.a2sdeveloper.com/page-working-with-the-one-second-rule.html`. Nonetheless, Enactment Engine already implements the enforcement of this rule.
- As you will be charged per hour, don't forget to shutdown/stop unused VMs.
- You can use the Amazon EC2 web console to unblock TCP ports necessary to the choreography execution.
- You can also use the `ec2` command line tools to manage your VMs.
- The Enactment Engine creates VMs in the "US East" Amazon datacenter.
- The SSH keypairs are datacenter-dependent. Therefore if you create a keypair in the EU datacenter, it won't be valid for your VMs.

1.3.2 OpenStack

OpenStack is an open source private cloud platform that provides services to retrieve VMs as you need, in the same way that Amazon EC2. However, you must install OpenStack in your own infrastructure,

which means you must own at least a little cluster (or a very powerful machine) to host the created VMs. Moreover, the OpenStack installation and configuration is not a simple task.

Some hints:

- OpenStack does not provide public IPs, therefore some VPN configuration is necessary to log into the provisioned nodes.
- You can also use the `nova` command line tool to manage your VMs.
- You need to host an Ubuntu Server 12.04 image within your OpenStack infrastructure. You will use the ID of this image to configure Enactment Engine.

1.3.3 Fixed cloud provider

If you are learning how to use the Enactment Engine and want just to experiment it, the FixedCloudProvider may be the most suitable option to you. It is also useful if you want to use Enactment Engine with your own already existing cluster machines. With it you are responsible to manually creating and setting virtual machines and telling to Enactment Engine which machines must be used by it. This avoids the overhead of dealing with a cloud environment.

When creating a virtual machine to be used by the Enactment Engine, be sure:

- to use the Ubuntu 12.04 as operating system;
- it is possible to SSH into the node without typing a password: `http://www.integrade.org.br/ssh-without-password`[1];
- use sudo in the machine without typing a password: type `$sudo visudo` and add the line `<user> ALL = NOPASSWD: ALL` at the end (change `<user>` by the actual user);
- to synchronize the machine clock: `#ntpdate ccsl.ime.usp.br`;

To verify if your VM was properly set, you may run the org.ow2.choreos.deployment.nodes.cloudprovider.FixedConnectionTest test.

The Enactment Engine *will not* take care of bootstrapping (installing Chef) on your machines, since this process is taken only when creating new machines. You *must* bootstrap your machines by running the org.ow2.choreos.deployment.nodes.cm.BootstrapFixedMachines class. When you run the BootstrapFixedMachines class, Enactment Engine will bootstrap the configured fixed machines. We will talk about configuration soon.

Depending on how you create your VMs, some network configuration may be needed. In case of using VirtualBox, you can refer to `http://ccsl.ime.usp.br/foswiki/bin/view/Choreos/VMs`.

1.4 Checkout and Compilation

To checkout the code: `git clone https://github.com/choreos/enactment_engine.git`.

After installing Maven 3, open the terminal at the `enactment_engine` folder, and run the `build.sh` script. It can take several minutes. Internet access is necessary during compilation.

1.5 Configuration

Open the folder `EnactmentEngine/src/main/resources`, and create a `ee.properties` file by copying the `ee.properties.template` file. The new properties file must be created in the same folder. Open the just created properties file and edit it following instructions on the template file. The Listing 1.2 shows an example. Do the same to the `clouds.properties.template` file; in the `clouds.properties` file you will define configuration to access your infrastructure provider.

Listing 1.1: ee.properties example.

```
1 EE_PORT=9102
2
3 # Value must be a <cloud account name> in clouds.properties
```

[1] Obs: do not use a key with password.

```
4  DEFAULT_CLOUD_ACCOUNT=MY_AWS_ACCOUNT
5
6  # Values in node_selector.properties
7  NODE_SELECTOR=LIMITED_ROUND_ROBIN
8
9  # Maximum number of VMs that can be created; set if using NODE_SELECTOR=LIMITED_ROUND_ROBIN
10 VM_LIMIT=10
11
12 # Creates a reservoir of extra VMs to make the deployment faster and more scalable.
13 # The trade-off is the cost of some more VMs.
14 # If the pool size reaches the threshold, the pool size is increased by one.
15 # To not increase your pool size, set threshold as negative or greater than the initial pool size.
16 RESERVOIR=true
17 RESERVOIR_INITIAL_SIZE=5
18 RESERVOIR_THRESHOLD=-1
```

Listing 1.2: cloud.properties example.

```
1  MY_CLUSTER.CLOUD_PROVIDER=FIXED
2  MY_CLUSTER.FIXED_VM_IPS=192.168.56.101, 192.168.56.102
3  MY_CLUSTER.FIXED_VM_PRIVATE_SSH_KEYS=/home/leonardo/.ssh/nopass, /home/leonardo/.ssh/nopass
4  MY_CLUSTER.FIXED_VM_USERS=choreos, choreos
5
6  MY_AWS_ACCOUNT.CLOUD_PROVIDER=AWS
7  MY_AWS_ACCOUNT.AMAZON_ACCESS_KEY_ID=SECRET
8  MY_AWS_ACCOUNT.AMAZON_SECRET_KEY=SECRET
9  MY_AWS_ACCOUNT.AMAZON_KEY_PAIR=leoflaws
10 MY_AWS_ACCOUNT.AMAZON_PRIVATE_SSH_KEY=/home/leonardo/.ssh/leoflaws.pem
11 MY_AWS_ACCOUNT.AMAZON_IMAGE_ID=us-east-1/ami-1ccc8875
12
13 MY_OPENSTACK_ACCOUNT.CLOUD_PROVIDER=OPEN_STACK
14 MY_OPENSTACK_ACCOUNT.OPENSTACK_KEY_PAIR=leofl
15 MY_OPENSTACK_ACCOUNT.OPENSTACK_PRIVATE_SSH_KEY=/home/leonardo/.ssh/nopass
16 MY_OPENSTACK_ACCOUNT.OPENSTACK_TENANT=CHOReOS_Sandbox
17 MY_OPENSTACK_ACCOUNT.OPENSTACK_USER=leofl
18 MY_OPENSTACK_ACCOUNT.OPENSTACK_PASSWORD=SECRET
19 MY_OPENSTACK_ACCOUNT.OPENSTACK_IP=http://172.15.237.10:5000/v2.0
20 MY_OPENSTACK_ACCOUNT.OPENSTACK_IMAGE_ID=RegionOne/1654b5b6-49b7-4039-b7b7-0e42e85480f4
```

The options to NODE_SELECTOR are:

ALWAYS_CREATE: a new VM is created to each deployed service instance.

ROUND_ROBIN: NodeSelector makes a round robin using the available VMs, without creating any new VM; usually it makes sense to use it only when using the fixed cloud provider.

LIMITED_ROUND_ROBIN: initially the NodeSelector behaves like the ALWAYS_CREATE, until a limit of created VMs is reached (VM_LIMIT). After this limit, the selector behaves like the ROUND_ROBIN. When using this selector, it is necessary to declare the integer VM_LIMIT property in the configuration file.

The AMAZON_IMAGE_ID enables you to specify a customized image to be used by Enactment Engine. This feature is intended to use an image of a node already bootstrapped. In this way, the bootstrap process becomes much faster. The same may be applied to the OPENSTACK_IMAGE_ID. But in both cases, the image must still provide an Ubuntu Server 12.04 system.

At the `clouds.properties`, each *cloud account* is configured by a group of properties grouped by a common prefix. Let's call this prefix as "cloud account name". These cloud account names will be compared with the `owner` attribute in the service specifications, so a service can be specified to be deployed under a specific cloud account. If there is no match, the DEFAULT_CLOUD_ACCOUNT value declared on `ee.properties` will be used as cloud account name.

Attention: inline comments are not allowed in properties files. Therefore, the following would not work: `VM\_LIMIT=3 \# how many instances we can afford to pay`.

1.6 Execution

After compiling the project, to run the Enactment Engine you have just to run the main method on the class org.ow2.choreos.ee.rest.EnactmentEngineServer.

This task can be easier accomplished if you import the Enactment Engine projects in the Eclipse IDE. After importing the project, open the menu `Window>>Preferences>>Java>>Build Path>>Classpath` variables, and set the `M2_REPO` variable pointing to your Maven repository folder, usually the `.m2/repository` folder within your home folder. Obs: we have used the Eclipse Indigo version.

Another way is using maven:

```
EnactmentEngine$ mvn exec:java
```

If you successfully start the EE, you must see the following message on the console:

```
Enactment Engine has started [http://localhost:9102/enactmentengine/]
```

To verify if it is everything OK, run the org.ow2.choreos.chors.SimpleChorEnactmentTest. This test will deploy a simple choreography composed of two services and try to invoke it.

Chapter 2

Enactment Engine API

2.1 Introduction

This chapter provides detailed information about the Enactment Engine REST API and its target mainly to choreography deployers[1]. Understanding the API enables you to write your own code to enact a choreography.

This chapter is organized as follows. Section 2.2 presents the data model that defines XML representations exchanged by API messages. Section 2.3 describes all the operations provided by the REST API, detailing parameters and return structures. Section 2.4 presents our client implementation that can be used within any Java program.

2.2 Data model

As in any API, Enactment Engine operations receive and return complex data structures representing real world concepts. Figure 2.1 presents these concepts in the UML notation. Although the REST API handles XML representations, we use here the UML notation, since it makes easier to the reader to understand the concepts.

We proceed with a brief explanation about each class:

ChoreographySpec: represents what the middleware needs to know to enact a choreography;

ServiceSpec: a super class for common data of `DeployableServiceSpec` and `LegacyServiceSpec`;

LegacyServiceSpec: represents an already existing service to used by the choreography;

DeployableServiceSpec: represents what the middleware needs to know to deploy a service (with one or more instances for load balancing);

ServiceDependency: represents dependencies among services (if `service A` invokes `service B`, we say `service A` depends on `service B`);

Choreography: provides information about a choreography instance;

Service: a super class for common data of `DeployableService` and `LegacyService`;

LegacyService: provides information about a legacy service;

DeployableService: provides information about a deployed service;

ServiceInstance: provides information about a specific instance, also called replica, of a deployed service (URI, node data etc.).

LegacyServiceInstance: provides information about a specific instance of a legacy service.

Node: information about the node, including IP address, where a service instance is running.

[1] Deployer is the human operator responsible by the deployment process.

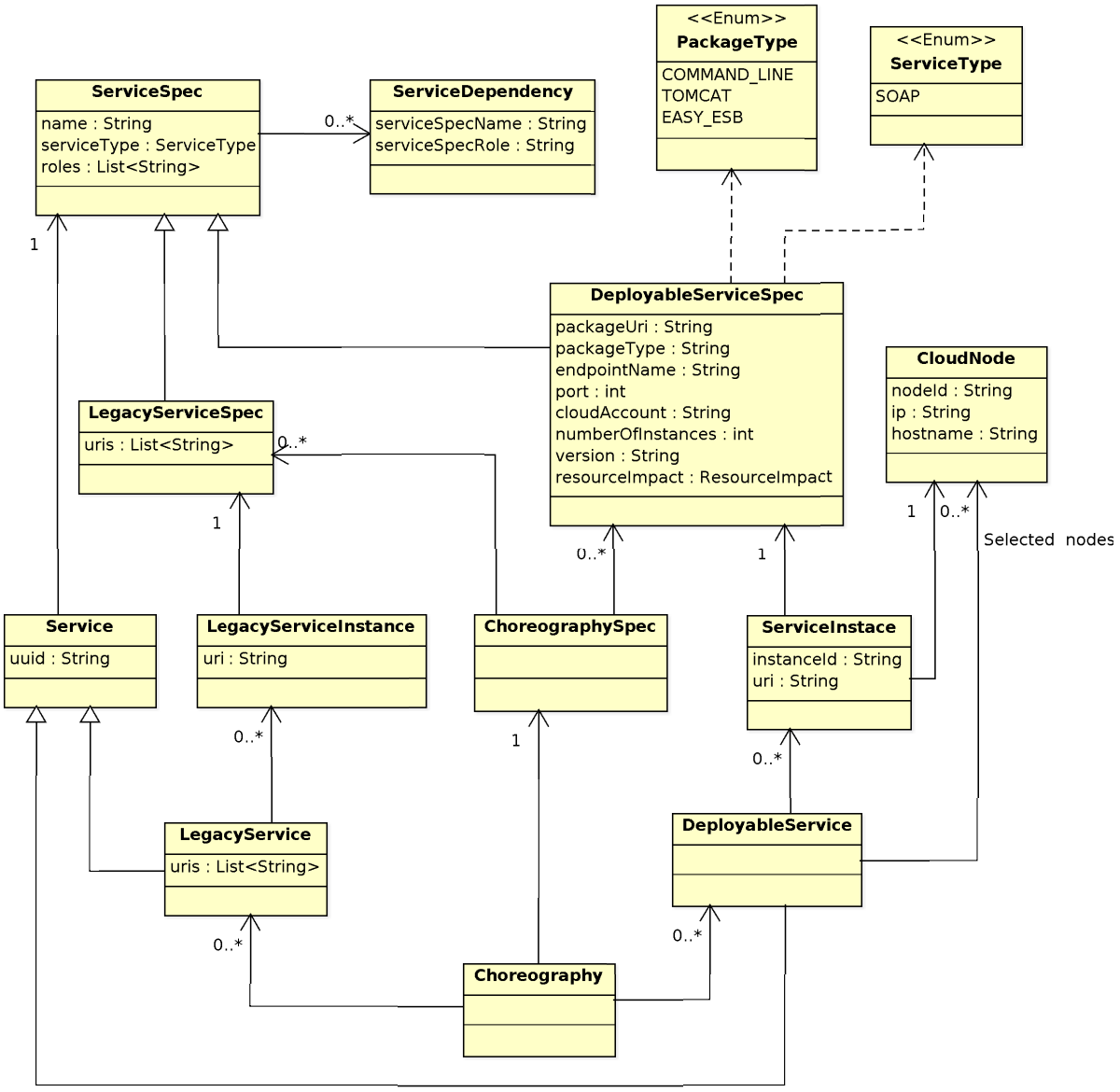

Figure 2.1: Enactment Engine REST API data model.

To request a choreography deployment, it is important to understand very well the `ServiceSpec`, `DeployableServiceSpec`, and the `LegacyServiceSpec` classes. Therefore, the description of them follows:

1. `ServiceSpec`

   name: a unique character sequence within the choreography specification;

   type: whether the service is a SOAP service or a REST service. More types can be added as necessary.

   roles: list of roles implemented by the service;

   dependencies: list of `ServiceDependency` entries; each entry describes the name of the dependency (matching the `name` attribute), and the the role provided by the dependency;

2. `DeployableServiceSpec`

   packageUri: the location of the binary file to be deployed;

   packageType: the type of the deployable package, according to the `PackageType` enumeration[2].

   endpointName: the endpoint suffix after deployment. For example, if the service will be deployed as `http://<some_ip>/choreos/service`, the endpoint name is `choreos/service`. Note that multiple replicas of a single service will all use the same endpoint;

   port: the TCP port used by the service. Note that multiple replicas of a single service will all use the same port. Mandatory if type is COMMAND_LINE;

   owner: must match a *cloud account* name configured on EE. It will define under which cloud infrastructure the service will be deployed.

   numberOfInstances: How many instances of the service should be deployed (onto different virtual machines) in order to allow the load to be distributed;

   version: the service version, which is used by the Enactment Engine to define which services must be redeployed in a choreography update (not used currently);

   resourceImpact: General information regarding the expected type of machine needed to run the service (see *Resource impact specification*);

3. `LegacyServiceSpec`

   URIs: The URIs for the various replicas of the service.

More about dependencies

In a service composition, some services depends on other services. A service that depends on other services is a *consumer* service, and the service that provides functionality to the dependent service is the *provider*. In simple service compositions, such dependency relations are hardcoded on consumer services. But decoupling the consumer service implementation from the actual provider endpoint is a good practice, which enables dynamic adaptation. Moreover, dependency hardcoding is not possible on cloud environments, since we do not know service addresses before deployment. Therefore, in the CHOReOS environment each consumer service is declared as depending on *roles* rather than other service implementations. The consumer service must receive the actual provider endpoint of a service fulfilling the required role through the `setInvocationAddress` operation, which every consumer service must implement.

The Enactment Engine will use `ServiceDependency` data to know which calls it must perform to the `setInvocationAddress` operation of participant services. Thus, the Enactment Engine will be able to tell, for example, to `ServiceA` that it must use `ServiceB` as `Role1`, where `ServiceB` is the list of endpoint URIs corresponding to the multiple instances of `ServiceB`. In this way, the CHOReOS middleware provides a *dependency injection*[3] mechanism to wire up service dependencies.

Obs: to SOAP services, the URI passed to the `setInvocationAddress` operation does not contain the '`?wsdl`' suffix.

[2]When the package type i s COMMAND_LINE the service will be executed by the "java -jar" command.

[3]Dependency Injection pattern, by Martin Fowler: `http://martinfowler.com/articles/injection.html`

Resource impact specification

The `DeployableServiceSpec` class has also an attribute to specify non-functional requirements. This attribute is called "resource impact", and it can be used by the `NodeSelector` to choose the node in which the service should be deployed. `NodeSelector` will try to choose a node that enables the service to fulfil such requirements.

This attribute is not described in this document because its structure is not fully defined yet. But it is expected to define, among others, required values of CPU, memory, and disk usage.

XML representation

Each class is mapped to and from an XML representation according to the default behaviour of the JAXB API[4]. To properly build and read these XML representations, you can rely on the schema definition (Section 2.5). We provide here an example of `ChorSpec` (Listing 2.1) and `Choreography` (Listing 2.2) XML representations to a little choreography with just two services (airline and travel-agency services).

Listing 2.1: ChorSpec XML representation example.

```
<choreographySpec>
    <deployableServiceSpecs>
        <name>airline</name>
        <roles>airline</roles>
        <serviceType>
            <type>SOAP</type>
        </serviceType>
        <endpointName>airline</endpointName>
        <numberOfInstances>1</numberOfInstances>
        <packageType>
            <type>COMMAND_LINE</type>
        </packageType>
        <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.jar</
            packageUri>
        <port>1234</port>
        <resourceImpact/>
        <version>0.1</version>
    </deployableServiceSpecs>
    <deployableServiceSpecs>
        <dependencies>
            <serviceSpecName>airline</serviceSpecName>
            <serviceSpecRole>airline</serviceSpecRole>
        </dependencies>
        <name>travelagency</name>
        <roles>travelagency</roles>
        <serviceType>
            <type>SOAP</type>
        </serviceType>
        <endpointName>travelagency</endpointName>
        <numberOfInstances>1</numberOfInstances>
        <packageType>
            <type>COMMAND_LINE</type>
        </packageType>
        <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency-3.0.0.jar
            </packageUri>
        <port>1235</port>
        <resourceImpact/>
        <version>0.1</version>
    </deployableServiceSpecs>
</choreographySpec>
```

Listing 2.2: Choreography XML representation example.

```
<choreography>
    <choreographySpec>
        <deployableServiceSpecs>
            <name>airline</name>
            <roles>airline</roles>
            <serviceType>
                <type>SOAP</type>
            </serviceType>
```

[4] Java Architecture for XML Binding (JAXB): allows Java developers to map Java classes to XML representations.

```
 9              <endpointName>airline</endpointName>
10              <numberOfInstances>1</numberOfInstances>
11              <packageType>
12                  <type>COMMAND_LINE</type>
13              </packageType>
14              <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.jar<
                    /packageUri>
15              <port>1234</port>
16              <resourceImpact/>
17              <version>0.1</version>
18          </deployableServiceSpecs>
19          <deployableServiceSpecs>
20              <dependencies>
21                  <serviceSpecName>airline</serviceSpecName>
22                  <serviceSpecRole>airline</serviceSpecRole>
23              </dependencies>
24              <name>travelagency</name>
25              <roles>travelagency</roles>
26              <serviceType>
27                  <type>SOAP</type>
28              </serviceType>
29              <endpointName>travelagency</endpointName>
30              <numberOfInstances>1</numberOfInstances>
31              <packageType>
32                  <type>COMMAND_LINE</type>
33              </packageType>
34              <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency
                    -3.0.0.jar</packageUri>
35              <port>1235</port>
36              <resourceImpact/>
37              <version>0.1</version>
38          </deployableServiceSpecs>
39      </choreographySpec>
40      <deployableServices>
41          <spec xsi:type="deployableServiceSpec" xmlns:xsi="http://www.w3.org/2001/
                XMLSchema-instance">
42              <dependencies>
43                  <serviceSpecName>airline</serviceSpecName>
44                  <serviceSpecRole>airline</serviceSpecRole>
45              </dependencies>
46              <name>travelagency</name>
47              <roles>travelagency</roles>
48              <serviceType>
49                  <type>SOAP</type>
50              </serviceType>
51              <endpointName>travelagency</endpointName>
52              <numberOfInstances>1</numberOfInstances>
53              <packageType>
54                  <type>COMMAND_LINE</type>
55              </packageType>
56              <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency
                    -3.0.0.jar</packageUri>
57              <port>1235</port>
58              <resourceImpact/>
59              <version>0.1</version>
60          </spec>
61          <UUID>9fb5c93a-b4d1-4a56-9b4b-120e89681e31</UUID>
62          <selectedNodes>
63              <hostname>choreos-node</hostname>
64              <id>2</id>
65              <ip>192.168.56.102</ip>
66          </selectedNodes>
67          <serviceInstances>
68              <instanceId>travelagency1</instanceId>
69              <nativeUri>http://192.168.56.102:1235/travelagency/</nativeUri>
70              <node>
71                  <hostname>choreos-node</hostname>
72                  <id>2</id>
73                  <ip>192.168.56.102</ip>
74              </node>
75              <serviceSpec>
76                  <dependencies>
77                      <serviceSpecName>airline</serviceSpecName>
```

```
78                      <serviceSpecRole>airline</serviceSpecRole>
79                  </dependencies>
80                  <name>travelagency</name>
81                  <roles>travelagency</roles>
82                  <serviceType>
83                      <type>SOAP</type>
84                  </serviceType>
85                  <endpointName>travelagency</endpointName>
86                  <numberOfInstances>1</numberOfInstances>
87                  <packageType>
88                      <type>COMMAND_LINE</type>
89                  </packageType>
90                  <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/TravelAgency
                        -3.0.0.jar</packageUri>
91                  <port>1235</port>
92                  <resourceImpact/>
93                  <version>0.1</version>
94              </serviceSpec>
95          </serviceInstances>
96      </deployableServices>
97      <deployableServices>
98          <spec xsi:type="deployableServiceSpec" xmlns:xsi="http://www.w3.org/2001/
                XMLSchema-instance">
99              <name>airline</name>
100             <roles>airline</roles>
101             <serviceType>
102                 <type>SOAP</type>
103             </serviceType>
104             <endpointName>airline</endpointName>
105             <numberOfInstances>1</numberOfInstances>
106             <packageType>
107                 <type>COMMAND_LINE</type>
108             </packageType>
109             <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.jar<
                    /packageUri>
110             <port>1234</port>
111             <resourceImpact/>
112             <version>0.1</version>
113         </spec>
114         <UUID>68d8e82b-f6e6-4314-9415-d2a18f61edcf</UUID>
115         <selectedNodes>
116             <hostname>choreos-node</hostname>
117             <id>1</id>
118             <ip>192.168.56.101</ip>
119         </selectedNodes>
120         <serviceInstances>
121             <instanceId>airline0</instanceId>
122             <nativeUri>http://192.168.56.101:1234/airline/</nativeUri>
123             <node>
124                 <hostname>choreos-node</hostname>
125                 <id>1</id>
126                 <ip>192.168.56.101</ip>
127             </node>
128             <serviceSpec>
129                 <name>airline</name>
130                 <roles>airline</roles>
131                 <serviceType>
132                     <type>SOAP</type>
133                 </serviceType>
134                 <endpointName>airline</endpointName>
135                 <numberOfInstances>1</numberOfInstances>
136                 <packageType>
137                     <type>COMMAND_LINE</type>
138                 </packageType>
139                 <packageUri>http://valinhos.ime.usp.br:54080/enact_test/v3/Airline-3.0.0.
                        jar</packageUri>
140                 <port>1234</port>
141                 <resourceImpact/>
142                 <version>0.1</version>
143             </serviceSpec>
144         </serviceInstances>
145     </deployableServices>
146     <id>1</id>
```

```
147 </choreography>
```

2.3 REST API

The Enactment Engine clients access its features through the REST API that is described in this section.

Create a choreography

HTTP Method	*URI*	*Request body*	*Responses*

POST	/chors	`ChorSpec` XML representation (see Listing 2.1)	`201 CREATED`
			`location = "/chors/{id}"`
			
			`400 BAD REQUEST`
			
			`500 ERROR`

Creates a specification of the choreography on the Enactment Engine. It does not deploy the choreography.

Obs: `application/xml` is the value to the `Content Type` header when XML representations are written in the request or response body.

Retrieve choreography information

HTTP Method	*URI*	*Request body*	*Responses*

GET	/chors/{id}	-	`200 OK`
			`location = "/chors/{id}"`
			
			`Body:`
			
			`Choreography` XML representation (see Listing 2.2)
			
			`400 BAD REQUEST`
			
			`404 NOT FOUND`
			
			`500 ERROR`

If this operation is invoked after the creation and before the deployment of a choreography, the body response will be a `Choreography` representation without any deployed service.

Deploy a choreography

HTTP Method	*URI*	*Request body*	*Responses*

POST	/chors/{id}/deployment	-	`200 OK`
			`location = "/chors/{id}"`
			`Body:`
			
			`Choreography` XML representation (see Listing 2.2)
			
			`400 BAD REQUEST`
			
			`404 NOT FOUND`
			
			`500 ERROR`

With this invocation, services will be finally deployed. The response arrives only after the deployment of all services, if no deployment fails. It is possible to parse the output to find out failed deployments, which will be the services without associated nodes.

Update a choreography (only partially implemented at this time)

HTTP Method	*URI*	*Request body*	*Responses*

PUT	/chors/{id}	`ChorSpec` XML representation (see Listing 2.1)	`200 OK`
			`location = "/chors/{id}"`
			`Body:`
			
			`Choreography` XML representation (see Listing 2.2)
			
			`400 BAD REQUEST`
			
			`404 NOT FOUND`
			
			`500 ERROR`

This operation has the same behavior of the create choreography operation. To apply the changes it is necessary to invoke the deployment operation again. When the new deployment is invoked, the Enactment Engine will detect the changes that have been inserted in the choreography and deploys new services, remove unneeded services and redeploy services where some aspect (such as version number, number of instances etc) has changed. Currently, the only detected changes are increased or decreased number of instances and increased or decreased memory consumption. Services on the old and new versions of the choreography are correlated by means of the `name` attribute of ServiceSpec.

2.4 Java client

In the `EnactmentEngineAPI` project there is the `EEClient` class, which implements the `EnactmentEngine` interface (Listing 2.3) and handles the REST communication with the Enactment Engine server. This means you can invoke the Enactment Engine by using a simple Java object, without worrying with XML details.

Listing 2.3: Enactment Engine Java interface.

```
1  package org.ow2.choreos.chors;
2
3  import org.ow2.choreos.chors.datamodel.Choreography;
4  import org.ow2.choreos.chors.datamodel.ChoreographySpec;
5
6  public interface EnactmentEngine {
7
8      public String createChoreography(ChoreographySpec chor);
9
10     public Choreography getChoreography(String chorId)
11         throws ChoreographyNotFoundException;
12
13     public Choreography deployChoreography(String chorId)
14         throws DeploymentException, ChoreographyNotFoundException;
15
16     public void updateChoreography(String chorId, ChoreographySpec spec)
17         throws EnactmentException, ChoreographyNotFoundException;
18
19 }
```

To use the Enactment Engine Java client in your code, it's enough to import the API project into your project. One way of doing this is using Maven: install the API project into your local maven repo (`EnactmentEngineAPI$mvn install`), add it as a dependency of your project by editing your pom.xml (Listing 2.4), and finally compile your project.

Listing 2.4: Adding EnactmentEngineAPI as a dependency of your project.

```
<dependency>
  <groupId>org.ow2.choreos</groupId>
  <artifactId>EnactmentEngineAPI</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The Listing 2.5 shows an example of how to use the Java API to create a choreography specification. This example is equivalent to the XML in Listing 2.1.

Listing 2.5: Example of choreography specification using the Java API.

```
public class ChorSpecExample {

    public static final String AIRLINE = "airline";
    public static final String TRAVEL_AGENCY = "travelagency";
    public static final String AIRLINE_JAR =
            "http://valinhos.ime.usp.br:54080/airline.jar";
    public static final String TRAVEL_AGENCY_JAR =
            "http://valinhos.ime.usp.br:54080/travel.jar";
    public static final int AIRLINE_PORT = 1234;
    public static final int TRAVEL_AGENCY_PORT = 1235;

    private ChoreographySpec chorSpec;
    private DeployableServiceSpec airlineSpec;
    private DeployableServiceSpec travelSpec;

    public ChoreographySpec getChorSpec() {
        createAirlineSpec();
        createTravelAgencySpec();
        chorSpec = new ChoreographySpec(this.airlineSpec, this.travelSpec);
        return chorSpec;
    }

    private void createAirlineSpec() {
        airlineSpec = new DeployableServiceSpec();
        airlineSpec.setName(AIRLINE);
        airlineSpec.setServiceType(ServiceType.SOAP);
        airlineSpec.setPackageType(PackageType.COMMAND_LINE);
        airlineSpec.setPackageUri(AIRLINE_JAR);
        airlineSpec.setPort(AIRLINE_PORT);
        airlineSpec.setEndpointName(AIRLINE);
        airlineSpec.setRoles(Collections.singletonList(AIRLINE));
    }

    private void createTravelAgencySpec() {
        travelSpec = new DeployableServiceSpec();
        travelSpec.setName(TRAVEL_AGENCY);
        travelSpec.setServiceType(ServiceType.SOAP);
        travelSpec.setPackageType(PackageType.COMMAND_LINE);
        travelSpec.setPackageUri(TRAVEL_AGENCY_JAR);
        travelSpec.setPort(TRAVEL_AGENCY_PORT);
        travelSpec.setEndpointName(TRAVEL_AGENCY);
        travelSpec.setRoles(Collections.singletonList(TRAVEL_AGENCY));
        ServiceDependency dependency = new ServiceDependency();
        dependency.setServiceSpecName(AIRLINE);
        dependency.setServiceSpecRole(AIRLINE);
        travelSpec.addDependency(dependency);
    }
```

Finally, Listing 2.6 is an example of how to use the Java API to invoke the EE.

Listing 2.6: Deploying a choreography using the Java API.

```
public class Deployment {

    public static void main(String[] args) throws DeploymentException,
        ChoreographyNotFoundException {

        final String EE_URI = "http://localhost:9102/enactmentengine";
        EnactmentEngine ee = new EnactmentEngineClient(EE_URI);
        ChorSpecExample example = new ChorSpecExample();
        ChoreographySpec chorSpec = example.getChorSpec();

```

```
10          String chorId = ee.createChoreography(chorSpec);
11          Choreography chor = ee.deployChoreography(chorId);
12
13          System.out.println(chor); // checking output
14      }
15  }
```

2.5 Choreography XML Schema Definition (XSD file)

```
1   ChorSpec XSD:
2   <?xml version="1.0" encoding="UTF-8"?><xs:schema xmlns:xs="http://www.w3.org/2001/
        XMLSchema" version="1.0">
3   <xs:element name="choreographySpec" type="choreographySpec"/>
4   <xs:element name="deployableServiceSpec" type="deployableServiceSpec"/>
5   <xs:element name="legacyServiceSpec" type="legacyServiceSpec"/>
6   <xs:element name="resourceImpact" type="resourceImpact"/>
7   <xs:complexType name="choreographySpec">
8           <xs:sequence>
9               <xs:element maxOccurs="unbounded" minOccurs="0" name="deployableServiceSpecs"
                     nillable="true" type="deployableServiceSpec"/>
10              <xs:element maxOccurs="unbounded" minOccurs="0" name="legacyServiceSpecs"
                    nillable="true" type="legacyServiceSpec"/>
11          </xs:sequence>
12      </xs:complexType>
13  <xs:complexType name="deployableServiceSpec">
14          <xs:complexContent>
15              <xs:extension base="serviceSpec">
16                  <xs:sequence>
17                      <xs:element minOccurs="0" name="cloudAccount" type="xs:string"/>
18                      <xs:element minOccurs="0" name="desiredQoS" type="desiredQoS"/>
19                      <xs:element minOccurs="0" name="endpointName" type="xs:string"/>
20                      <xs:element name="numberOfInstances" type="xs:int"/>
21                      <xs:element minOccurs="0" name="packageType" type="packageType"/>
22                      <xs:element minOccurs="0" name="packageUri" type="xs:string"/>
23                      <xs:element name="port" type="xs:int"/>
24                      <xs:element minOccurs="0" ref="resourceImpact"/>
25                      <xs:element minOccurs="0" name="version" type="xs:string"/>
26                  </xs:sequence>
27              </xs:extension>
28          </xs:complexContent>
29      </xs:complexType>
30  <xs:complexType abstract="true" name="serviceSpec">
31          <xs:sequence>
32              <xs:element maxOccurs="unbounded" minOccurs="0" name="dependencies" nillable=
                    "true" type="serviceDependency"/>
33              <xs:element minOccurs="0" name="name" type="xs:string"/>
34              <xs:element maxOccurs="unbounded" minOccurs="0" name="roles" nillable="true"
                    type="xs:string"/>
35              <xs:element minOccurs="0" name="serviceType" type="serviceType"/>
36          </xs:sequence>
37      </xs:complexType>
38  <xs:complexType name="desiredQoS">
39          <xs:sequence>
40              <xs:element minOccurs="0" name="responseTimeMetric" type="responseTimeMetric"
                    />
41          </xs:sequence>
42      </xs:complexType>
43  <xs:complexType name="responseTimeMetric">
44          <xs:sequence>
45              <xs:element name="acceptablePercentage" type="xs:float"/>
46              <xs:element name="maxDesiredResponseTime" type="xs:float"/>
47          </xs:sequence>
48      </xs:complexType>
49  <xs:complexType name="packageType">
50          <xs:sequence>
51              <xs:element minOccurs="0" name="type" type="xs:string"/>
52          </xs:sequence>
53      </xs:complexType>
54  <xs:complexType name="resourceImpact">
55          <xs:sequence>
56              <xs:element minOccurs="0" name="memory" type="memoryType"/>
```

```
57              <xs:element minOccurs="0" name="cpu" type="xs:string"/>
58              <xs:element minOccurs="0" name="storage" type="xs:string"/>
59              <xs:element minOccurs="0" name="network" type="xs:string"/>
60          </xs:sequence>
61      </xs:complexType>
62  <xs:complexType name="serviceDependency">
63          <xs:sequence>
64              <xs:element minOccurs="0" name="serviceSpecName" type="xs:string"/>
65              <xs:element minOccurs="0" name="serviceSpecRole" type="xs:string"/>
66          </xs:sequence>
67      </xs:complexType>
68  <xs:complexType name="serviceType">
69          <xs:sequence>
70              <xs:element minOccurs="0" name="type" type="xs:string"/>
71          </xs:sequence>
72      </xs:complexType>
73  <xs:complexType name="legacyServiceSpec">
74          <xs:complexContent>
75              <xs:extension base="serviceSpec">
76                  <xs:sequence>
77                      <xs:element maxOccurs="unbounded" minOccurs="0" name="nativeURIs"
                            nillable="true" type="xs:string"/>
78                  </xs:sequence>
79              </xs:extension>
80          </xs:complexContent>
81      </xs:complexType>
82  <xs:simpleType name="memoryType">
83          <xs:restriction base="xs:string">
84              <xs:enumeration value="SMALL"/>
85              <xs:enumeration value="MEDIUM"/>
86              <xs:enumeration value="LARGE"/>
87          </xs:restriction>
88      </xs:simpleType>
89  </xs:schema>
90
91  Choreography XSD:
92  <?xml version="1.0" encoding="UTF-8"?><xs:schema xmlns:xs="http://www.w3.org/2001/
        XMLSchema" version="1.0">
93  <xs:element name="choreography" type="choreography"/>
94  <xs:element name="choreographySpec" type="choreographySpec"/>
95  <xs:element name="cloudNode" type="cloudNode"/>
96  <xs:element name="deployableService" type="deployableService"/>
97  <xs:element name="deployableServiceSpec" type="deployableServiceSpec"/>
98  <xs:element name="legacyServiceSpec" type="legacyServiceSpec"/>
99  <xs:element name="resourceImpact" type="resourceImpact"/>
100 <xs:complexType name="choreography">
101         <xs:sequence>
102             <xs:element minOccurs="0" ref="choreographySpec"/>
103             <xs:element maxOccurs="unbounded" minOccurs="0" name="deployableServices"
                    nillable="true" type="deployableService"/>
104             <xs:element minOccurs="0" name="id" type="xs:string"/>
105             <xs:element maxOccurs="unbounded" minOccurs="0" name="legacyServices"
                    nillable="true" type="legacyService"/>
106         </xs:sequence>
107     </xs:complexType>
108 <xs:complexType name="choreographySpec">
109         <xs:sequence>
110             <xs:element maxOccurs="unbounded" minOccurs="0" name="deployableServiceSpecs"
                    nillable="true" type="deployableServiceSpec"/>
111             <xs:element maxOccurs="unbounded" minOccurs="0" name="legacyServiceSpecs"
                    nillable="true" type="legacyServiceSpec"/>
112         </xs:sequence>
113     </xs:complexType>
114 <xs:complexType name="deployableServiceSpec">
115         <xs:complexContent>
116             <xs:extension base="serviceSpec">
117                 <xs:sequence>
118                     <xs:element minOccurs="0" name="cloudAccount" type="xs:string"/>
119                     <xs:element minOccurs="0" name="desiredQoS" type="desiredQoS"/>
120                     <xs:element minOccurs="0" name="endpointName" type="xs:string"/>
121                     <xs:element name="numberOfInstances" type="xs:int"/>
122                     <xs:element minOccurs="0" name="packageType" type="packageType"/>
123                     <xs:element minOccurs="0" name="packageUri" type="xs:string"/>
```

```
124                     <xs:element name="port" type="xs:int"/>
125                     <xs:element minOccurs="0" ref="resourceImpact"/>
126                     <xs:element minOccurs="0" name="version" type="xs:string"/>
127                 </xs:sequence>
128             </xs:extension>
129         </xs:complexContent>
130     </xs:complexType>
131 <xs:complexType abstract="true" name="serviceSpec">
132         <xs:sequence>
133             <xs:element maxOccurs="unbounded" minOccurs="0" name="dependencies" nillable=
                    "true" type="serviceDependency"/>
134             <xs:element minOccurs="0" name="name" type="xs:string"/>
135             <xs:element maxOccurs="unbounded" minOccurs="0" name="roles" nillable="true"
                    type="xs:string"/>
136             <xs:element minOccurs="0" name="serviceType" type="serviceType"/>
137         </xs:sequence>
138     </xs:complexType>
139 <xs:complexType name="desiredQoS">
140         <xs:sequence>
141             <xs:element minOccurs="0" name="responseTimeMetric" type="responseTimeMetric"
                    />
142         </xs:sequence>
143     </xs:complexType>
144 <xs:complexType name="responseTimeMetric">
145         <xs:sequence>
146             <xs:element name="acceptablePercentage" type="xs:float"/>
147             <xs:element name="maxDesiredResponseTime" type="xs:float"/>
148         </xs:sequence>
149     </xs:complexType>
150 <xs:complexType name="packageType">
151         <xs:sequence>
152             <xs:element minOccurs="0" name="type" type="xs:string"/>
153         </xs:sequence>
154     </xs:complexType>
155 <xs:complexType name="resourceImpact">
156         <xs:sequence>
157             <xs:element minOccurs="0" name="memory" type="memoryType"/>
158             <xs:element minOccurs="0" name="cpu" type="xs:string"/>
159             <xs:element minOccurs="0" name="storage" type="xs:string"/>
160             <xs:element minOccurs="0" name="network" type="xs:string"/>
161         </xs:sequence>
162     </xs:complexType>
163 <xs:complexType name="serviceDependency">
164         <xs:sequence>
165             <xs:element minOccurs="0" name="serviceSpecName" type="xs:string"/>
166             <xs:element minOccurs="0" name="serviceSpecRole" type="xs:string"/>
167         </xs:sequence>
168     </xs:complexType>
169 <xs:complexType name="serviceType">
170         <xs:sequence>
171             <xs:element minOccurs="0" name="type" type="xs:string"/>
172         </xs:sequence>
173     </xs:complexType>
174 <xs:complexType name="legacyServiceSpec">
175         <xs:complexContent>
176             <xs:extension base="serviceSpec">
177                 <xs:sequence>
178                     <xs:element maxOccurs="unbounded" minOccurs="0" name="nativeURIs"
                            nillable="true" type="xs:string"/>
179                 </xs:sequence>
180             </xs:extension>
181         </xs:complexContent>
182     </xs:complexType>
183 <xs:complexType name="deployableService">
184         <xs:complexContent>
185             <xs:extension base="service">
186                 <xs:sequence>
187                     <xs:element maxOccurs="unbounded" minOccurs="0" name="selectedNodes"
                            nillable="true" type="cloudNode"/>
188                     <xs:element maxOccurs="unbounded" minOccurs="0" name="
                            serviceInstances" nillable="true" type="serviceInstance"/>
189                 </xs:sequence>
190             </xs:extension>
```

```
191         </xs:complexContent>
192     </xs:complexType>
193 <xs:complexType abstract="true" name="service">
194         <xs:sequence>
195             <xs:element minOccurs="0" name="spec" type="serviceSpec"/>
196             <xs:element minOccurs="0" name="UUID" type="xs:string"/>
197         </xs:sequence>
198     </xs:complexType>
199 <xs:complexType name="cloudNode">
200         <xs:sequence>
201             <xs:element minOccurs="0" name="cpus" type="xs:int"/>
202             <xs:element minOccurs="0" name="hostname" type="xs:string"/>
203             <xs:element minOccurs="0" name="id" type="xs:string"/>
204             <xs:element minOccurs="0" name="image" type="xs:string"/>
205             <xs:element minOccurs="0" name="ip" type="xs:string"/>
206             <xs:element minOccurs="0" name="privateKeyFile" type="xs:string"/>
207             <xs:element minOccurs="0" name="ram" type="xs:int"/>
208             <xs:element minOccurs="0" name="so" type="xs:string"/>
209             <xs:element minOccurs="0" name="state" type="xs:int"/>
210             <xs:element minOccurs="0" name="storage" type="xs:int"/>
211             <xs:element minOccurs="0" name="user" type="xs:string"/>
212             <xs:element minOccurs="0" name="zone" type="xs:string"/>
213         </xs:sequence>
214     </xs:complexType>
215 <xs:complexType name="serviceInstance">
216         <xs:sequence>
217             <xs:element minOccurs="0" name="instanceId" type="xs:string"/>
218             <xs:element minOccurs="0" name="nativeUri" type="xs:string"/>
219             <xs:element minOccurs="0" name="node" type="cloudNode"/>
220             <xs:element minOccurs="0" name="serviceSpec" type="deployableServiceSpec"/>
221         </xs:sequence>
222     </xs:complexType>
223 <xs:complexType name="legacyService">
224         <xs:complexContent>
225             <xs:extension base="service">
226                 <xs:sequence>
227                     <xs:element maxOccurs="unbounded" minOccurs="0" name="
                            legacyServiceInstances" nillable="true" type="
                            legacyServiceInstance"/>
228                 </xs:sequence>
229             </xs:extension>
230         </xs:complexContent>
231     </xs:complexType>
232 <xs:complexType name="legacyServiceInstance">
233         <xs:sequence>
234             <xs:element minOccurs="0" name="spec" type="legacyServiceSpec"/>
235             <xs:element minOccurs="0" name="uri" type="xs:string"/>
236         </xs:sequence>
237     </xs:complexType>
238 <xs:simpleType name="memoryType">
239         <xs:restriction base="xs:string">
240             <xs:enumeration value="SMALL"/>
241             <xs:enumeration value="MEDIUM"/>
242             <xs:enumeration value="LARGE"/>
243         </xs:restriction>
244     </xs:simpleType>
245 </xs:schema>
```

Chapter 3

How to package services to be deployed by the Enactment Engine

3.1 Introduction

There are two main attributes in the DeployableServiceSpec class that define constraints on how a service must be coded and packaged. The `packageType` attribute defines which kind of deployable package is expected by the Enactment Engine and, therefore, the process of deploying and running the specified service. Examples of package types are COMMAND_LINE and TOMCAT. The `serviceType` attribute defines the process of invoking the specified service, which is used by the Enactment Engine to properly invoke the `setInvocationAddress` operation during deployment. Examples of service types are SOAP and REST types, although only SOAP services are currently supported by the Enactment Engine.

This chapter is targeted to service *developers* that intend to develop EE compatible services. Descriptions and hints encompass coding and packaging phases. This guide considers only the service and package types currently supported by EE.

3.2 COMMAND_LINE package type

Services whose package type are specified as COMMAND_LINE must be provided as JAR packages. The JAR must contain all the dependencies and resources within it. When using this package type, it is mandatory to fill the attributes `port` and `endpointName` on DeployableServiceSpec.

It must be possible to run the JAR by typing the command `java -jar <file_name>`, where `<file_name>` must be replaced with the name of the JAR file. Every JAR file contains a file called `MANIFEST.MF` within the `META-INF` folder, which is in the root of JAR file. A runnable JAR contains the following entry in its `MANIFEST` file: "`Main-Class: <class>`", where `<class>` must be replaces by the full qualified name of the class containing the main method, as for example `org.ow2.choreos.AirlineStarter`.

The Listing 3.1 provides an example of main class within a JAR file. The Airline interface is the business interface, and the AirlineService is the implementing class that uses the JAX-WS framework to expose SOAP services. In this example, the used TCP port and the endpoint name are defined in the SERVICE_ADDRESS assignment (line 7). The used port is the 1234, and the endpoint name is "airline". *Attention!* The use of "0.0.0.0" instead of "localhost" is necessary to make the service accessible from outside the machine where it is running.

Listing 3.1: Example of a class with the main method within a JAR file.

```
1  package org.ow2.choreos;
2
3  import javax.xml.ws.Endpoint;
4
5  public class AirlineStarter {
6
7    public static final String SERVICE_ADDRESS = ``http://0.0.0.0:1234/airline'';
8    private static Endpoint endpoint;
9
10   public static void start() {
11     Airline service = new AirlineService();
12     endpoint = Endpoint.create(service);
```

```
13      endpoint.publish(SERVICE_ADDRESS);
14    }
15
16    public static void main(String[] args) {
17      start();
18    }
19  }
```

Runnable JARs can be easily generated by the export menu in Eclipse or by Maven. To generate a runnable JAR using Maven 3, add the excerpt in the Listing 3.2 in your `pom.xml` file, properly replacing the content of the `mainClass` element. In this way, when generating the JAR file (`mvn package`), Maven will be in charge of properly generating the `MANIFEST` file.

Listing 3.2: Excerpt of pom file to generate a runnable JAR using Maven 3.

```
1  <build>
2    <finalName>airline-service</finalName>
3    <!-- <finalName>travel-agency-service</finalName> -->
4    <plugins>
5      <plugin>
6        <groupId>org.apache.maven.plugins</groupId>
7        <artifactId>maven-shade-plugin</artifactId>
8        <version>2.0</version>
9        <executions>
10         <execution>
11           <phase>package</phase>
12           <goals>
13             <goal>shade</goal>
14           </goals>
15           <configuration>
16             <shadedArtifactId>airline-service</shadedArtifactId>
17             <shadeSourcesContent>true</shadeSourcesContent>
18             <transformers>
19                 <transformer
20 implementation=''org.apache.maven.plugins.shade.resource.ManifestResourceTransformer''>
21                 <mainClass>org.ow2.choreos.AirlineStarter</mainClass>
22               </transformer>
23             </transformers>
24           </configuration>
25         </execution>
26       </executions>
27     </plugin>
28   </plugins>
29 </build>
```

An important issue about using JAR packages is that the Enactment Engine is not able to prevent TCP port conflicts. Therefore, try to avoid the use of the same port in multiple services. Also, be sure that the required ports are not blocked by the cloud infrastructure. However, a better way to prevent such port issues is not to use JAR packages, but use WAR packages instead.

3.3 TOMCAT package type

Services whose package type are specified as TOMCAT must be provided as WAR packages. The Enactment Engine will be in charge of deploying and starting a Tomcat instance, or select an already running instance, and then deploying the WAR package onto the selected instance. If the service port is not specified in the service specification, the middleware assumes the port as 8080, the Tomcat default port. If the endpoint name is not specified, it is assumed as the WAR file name, without the "war" extension, as it is the default behavior in Tomcat.

Dependencies (JAR libraries) must be packaged within the WAR file. However, we provide a set of libraries in our Tomcat installation that are usually used in Java projects, specially projects using the JAX-WS framework. If some of these JARs are used by your service, they are not required to be packaged within your WAR file, since they are already on the Tomcat class path. This strategy helps in decreasing the size of WAR files and, therefore, decreasing the deployment time. The provided libraries are the following:

- `activation-1.1.jar`
- `ecj-3.7.1.jar`

- gmbal-api-only-3.1.0-b001.jar
- ha-api-3.1.8.jar
- istack-commons-runtime-2.2.1.jar
- javax.annotation-3.1.1-b06.jar
- jaxb-api-2.2.3.jar
- jaxb-impl-2.2.4-1.jar
- jaxws-api-2.2.5.jar
- jaxws-rt-2.2.5.jar
- jsr181-api-1.0-MR1.jar
- management-api-3.0.0-b012.jar
- mimepull-1.6.jar
- policy-2.2.2.jar
- resolver-20050927.jar
- saaj-api-1.3.3.jar
- saaj-impl-1.3.10.jar
- stax-api-1.0-2.jar
- stax-api-1.0.1.jar
- stax-ex-1.4.jar
- stax2-api-3.1.1.jar
- streambuffer-1.2.jar
- tomcat-api.jar
- tomcat-jdbc.jar
- tomcat-util.jar
- tomcat_libs.tar.gz txw2-20090102.jar
- woodstox-core-asl-4.1.1.jar
- wstx-asl-3.2.3.jar

One way to be sure that you are using the required versions is making your project depending on JAX-WS by adding the fragment of Listing 3.3 in your Maven's pom:

Listing 3.3: Making your project depending on JAX-WS using Maven.

```
1  <dependency>
2      <groupId>com.sun.xml.ws</groupId>
3      <artifactId>jaxws-rt</artifactId>
4      <version>2.1.4</version>
5  </dependency>
```

If you write your web service using JAX-WS, your WAR file must also package a `sun-jaxws.xml` file, as Listing 3.4. As any other WAR file, it must also contain a `web.xml` file. If your service was built with JAX-WS, your `web.xml` file must be similar to the one presented in the Listing 3.5. Be aware that besides the usual definition of `servlet` and `servelet-mapping` elements, it is also necessary to declare the `listener` element exactly as in the example (lines 8, 9, and 10)[1].

[1]The instructions about the `sun-jaxws.xml` and `web.xml` files were retrieved from `http://www.mkyong.com/webservices/jax-ws/deploy-jax-ws-web-services-on-tomcat/`

Listing 3.4: Example of `sun-jaxws.xml` file.

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<endpoints
  xmlns=''http://java.sun.com/xml/ns/jax-ws/ri/runtime''
  version=''2.0''>
  <endpoint
      name=''Airline''
      implementation=''org.ow2.choreos.AirlineService''
      url-pattern=''/airline''/>
</endpoints>
```

Listing 3.5: Example of `web.xml` file.

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<!DOCTYPE web-app PUBLIC ''-//Sun Microsystems,
Inc.//DTD Web Application 2.3//EN''
''http://java.sun.com/dtd/web-app_2_3.dtd''>

<web-app>
    <listener>
        <listener-class>
                com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>airline</servlet-name>
        <servlet-class>
                com.sun.xml.ws.transport.http.servlet.WSServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>airline</servlet-name>
        <url-pattern>/airline</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>120</session-timeout>
    </session-config>
</web-app>
```

3.4 EASY_ESB package type

The Enactment Engine is also responsible for the coordination delegates deployment, that are executed by the EasyESB service bus. To enable this functionality, we have created the EASY_ESB package type, that is a tar.gz package containing a `config.xml` file with instructions for the bus. In the package, some resources needed for the deployment can be added. This process enables not only the deployment of coordination delegates, but actually any interaction with EasyESB.

The `config.xml` file must be built according to the schema in the Listing 3.6. `Configuration` is the root element. `Service` is an element containing a set of `Actions` made on a particular EasyESB node. These actions can be:

Deploy: deploys an artifact in EasyESB (BPEL, CD, etc.). It must contain the `MainResource` element and can have additional `Resource` elements.

Bind: binds a running web service onto an EasyESB; this action receives as parameter the web service URL and the web service WSDL location.

Proxify: binds a running web service onto an EasyESB node and re-export it using the same parameters used in the `Bind` action.

Expose: exposes an EasyESB internal service as a web service. Parameters are `ServiceNamespace` and `ServiceName`, that correspond to the `QName` of the service defined in the WSDL (usually it is the WSDL target namespace plus the name attribute of the service element).

Listing 3.6: `config.xml` schema.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema targetNamespace="http://ebmwebsourcing.com/cli/schema"
3   elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
4   xmlns:tns="http://ebmwebsourcing.com/cli/schema">
5
6    <element name="Configuration" type="tns:ConfigurationType" />
7
8    <complexType name="ConfigurationType">
9      <sequence>
10       <element maxOccurs="unbounded" name="Service"
11         type="tns:ServiceConfigurationType" />
12     </sequence>
13   </complexType>
14
15   <complexType name="ServiceConfigurationType">
16     <sequence>
17       <element maxOccurs="unbounded" name="Action" type="tns:ActionType" />
18     </sequence>
19     <attribute name="url" type="string" />
20   </complexType>
21
22   <complexType name="ActionType">
23     <choice>
24       <element name="Deploy" type="tns:DeployType" />
25       <element name="Bind" type="tns:BindType" />
26       <element name="Expose" type="tns:ExposeType" />
27       <element name="Proxify" type="tns:ProxifyType" />
28       <element name="AddNeighbourNode" type="tns:AddNeighbourNodeType" />
29     </choice>
30   </complexType>
31
32   <complexType name="AddNeighbourNodeType">
33     <sequence>
34       <element name="NeighbourAdminAddress" type="string" />
35     </sequence>
36   </complexType>
37
38   <complexType name="DeployType">
39     <sequence>
40       <element name="MainResource" type="string" />
41       <element maxOccurs="unbounded" name="Resource" type="string" />
42     </sequence>
43   </complexType>
44
45   <complexType name="BindType">
46     <sequence>
47       <element name="Url" type="string" />
48       <element name="Wsdl" type="string" />
49     </sequence>
50   </complexType>
51
52   <complexType name="ExposeType">
53     <sequence>
54       <element name="ServiceNamespace" type="string" />
55       <element name="ServiceName" type="string" />
56       <element name="EndpointName" type="string" />
57     </sequence>
58   </complexType>
59
60   <complexType name="ProxifyType">
61     <sequence>
62       <element name="Url" type="string" />
63       <element name="Wsdl" type="string" />
64     </sequence>
65   </complexType>
66
67 </schema>
```

The Listing 3.7 shows an example of `config.xml` file that makes the deployment of a coordination delegate in a scenario where the correspondent business service is already running and available[2]. The

[2]Indeed, the main scenario envisioned by CHOReOS is that there are already a lot of the services running "on the wild",

`weatherforecastservice.lts` and `cdweatherforecastservice.wsdl` files, referenced in lines 8 and 9, are provided within the tar.gz package. The use of the "../../" in these lines is mandatory. The url `http://192.168.56.101:8192/weatherforecastservice` provided in line 15, as well as the correspondent WSDL indicated in line 15, are references to a service already running and accessible. The `ServiceNamespace` (line 20) is the `targetNamespace` defined in the service WSDL. The value of the `ServiceName` element (line 21) must correspond to the value of the `name` attribute of the `service` element in the service WSDL. The value of the `EndpointName` element (line 22) must correspond to the `name` of the `portType` element in the coordination delegate WSDL. The lts file pointed by the `config.xml` is provided in the Listing 3.8, and the value of its `endpoint` attribute must correspond to the `name` of the `portType` element in the already-running service WSDL.

Listing 3.7: Example of `config.xml` that deploys a coordination delegate.

```
1  <?xml version=''1.0'' enc\begin{lstlisting}oding=''UTF-8''?>
2  <Configuration xmlns=''http://ebmwebsourcing.com/cli/schema''
3                 xmlns:xsi=''http://www.w3.org/2001/XMLSchema-instance''
4                 xsi:schemaLocation=''http://ebmwebsourcing.com/cli/schema conf-schema.xsd''>
5    <Service url="http://localhost:8180/services/adminExternalEndpoint">
6      <Action>
7        <Deploy>
8          <MainResource>../../weatherforecastservice.lts</MainResource>
9          <Resource>../../cdweatherforecastservice.wsdl</Resource>
10       </Deploy>
11     </Action>
12     <Action>
13       <Bind>
14         <Url>http://192.168.56.101:8192/weatherforecastservice</Url>
15         <Wsdl>http://192.168.56.101:8192/weatherforecastservice?wsdl</Wsdl>
16       </Bind>
17     </Action>
18     <Action>
19       <Expose>
20         <ServiceNamespace>http://services.choreos.org/</ServiceNamespace>
21         <ServiceName>WeatherForecastServiceService</ServiceName>
22         <EndpointName>CDWeatherForecastServicePort</EndpointName>
23       </Expose>
24     </Action>
25   </Service>
26 </Configuration>
```

Listing 3.8: LTS file of a simple coordination delegate that acts as a proxy.

```
1  endpoint=WeatherForecastServicePort
```

3.5 Choreographing services that are already running

A service choreography can be composed of services that are running before the choreography deployment. Although a service like this do not need to be deployed by the middleware, it must be declared on the choreography specification with the LegacyServiceSpec class. The `URIs` attribute must contain the list of URIs where the multiple replicas of the service are accessible, which will be used by the middleware to invoke the `setInvocationAddress` operation of other services.

3.6 SOAP service type

By convention, services of this type must provide an operation named "`setInvocationAddress`"; this operation is used to inform the service about the remote service endpoints that implement the various *roles* it depends on. The `setInvocationAddress` arguments are the following:

dependency role: defines the operations provided by the dependency. A service may depend on multiple services with different roles, so this argument is necessary to the service know how to use the received dependency. It is a requirement that the service must to know the available operations of each role from which it depends. The role of each service must be also defined in the choreography specification, that is the Enactment Engine input.

and choreographies are made just to compose these already-running services, situation in which only coordels are actually deployed.

dependency name: the name of the dependency that implements the role above. It works as a label that the service may use to distinguish different available services playing the same role. These different services are actually different implementations, possibly belonging to different organizations.

dependency endpoints: the list of alternative URIs to access the dependency. It has several URIs because a service may has multiple instances to improve its scalability. It is expected, but not required, from the dependent service to implement some load balancing between the different URIs. However, the dependent service may simply pick up anyone of the received endpoints. URIs passed to the `setInvocationAddress` operation do not contain the '`?wsdl`' suffix.

The expected interface of the `setInvocationAddress` operation is formally expressed by the WSDL elements presented in the Listing 3.9.

Listing 3.9: Parts of the service WSDL that define the `setInvocationAddress` operation.

```
<xs:schema version=''1.0'' targetNamespace=''http://services.choreos.org/''>
  ...
  <xs:complexType name=''setInvocationAddress''>
    <xs:sequence>
      <xs:element name=''arg0'' type=''xs:string'' minOccurs=''0''/>
      <xs:element name=''arg1'' type=''xs:string'' minOccurs=''0''/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="arg2" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name=''setInvocationAddressResponse''>
    <xs:sequence/>
  </xs:complexType>
  ...
</xs:schema>

<message name=''setInvocationAddress''>
  <part name=''parameters'' element=''tns:setInvocationAddress''/>
</message>
<message name=''setInvocationAddressResponse''>
  <part name=''parameters'' element=''tns:setInvocationAddressResponse''/>
</message>

<portType ... >
...
  <operation name=''setInvocationAddress''>
    <input message=''tns:setInvocationAddress''/>
    <output message=''tns:setInvocationAddressResponse''/>
  </operation>
</portType>
```

If you are using the JAX-WS framework, you can easily create a compatible `setInvocationAddress` operation by using the code provided in the Listing 3.10.

Listing 3.10: Implementing `setInvocationAddress` with JAX-WS.

```
1  @WebService
2  public class SomeWebServiceClass {
3
4      ...
5
6      @WebMethod
7      public void setInvocationAddress(String role, String name, List<String> endpoints) {
8                  ...
9      }
10 }
```

3.7 COORDEL service type

This service type must be used when declaring service specifications to CHOReOS Coordination Delegates (CDs). Although a coordination delegate proxifies all the operations of a SOAP service, the proxified `setInvocationAddress` operation will be not invoked. If a coordination delegate cdA is declared to depend on a coordination delegate cdB, the Enactment Engine will link the EasyESB nodes hosting the coordination delegates by invoking the `addNeighbour` operation of the EasyESB hosting cdA passing as neighbor the EasyESB node hosting cdB. Such operation enables coordination delegates to communicate directly among them by using the primitives (UPDATE_STATE(), WAIT(), NOTIFY()) provided by the CD-component running on EasyESB nodes. Every CD depend on some business service (SOAP service). But this dependency must be not declared on the service specification. It is a implicit dependency that must be declared on the config.xml file of the correspondent CD. Finally, when some SOAP service is declared to depend on some CD, the SOAP service will receive through `setInvocationAddress` the CD endpoint exposed by the bus. Such CD endpoint proxifies the service related to the CD.

3.8 Coding guidelines

Here are just some few important reminders:

- Do not forget to unblock the TCP ports used by your services. Often, this may be accomplished by the usage of management tools of your cloud environment.
- WAR packages are preferable to JAR packages. WAR packages avoid port conflict issues, and it is easier to manage the life-cycle of services distributed in WAR packages thanks to the management utilities of Tomcat. Life-cycle management matters, for example, when debugging if a service is actually running or not. Handling the life-cycle of JAR packaged services requires directly handling Unix processes.
- Never use absolute paths to retrieve resources, since the service will not run in the same machine where it was compiled. A good way to access resources is using the `getResourceAsStream` method of the current class loader[3].
- When starting a JAR packaged service, do not use the "localhost" address to create the endpoint, since the service will be not remotely accessible. Instead, use the address "0.0.0.0" that will make your service listen in every possible address in the machine, including the localhost. This practice makes the service accessible from other machines.
- Do not use `System.out.println`, use a log tool instead. Since the services are deployed in an automated way, it might be impossible to retrieve the console output, which will make debugging harder. Using a logger, as Log4j for example, makes the service record its messages in a file, which helps developers and operators in debugging.
- Do not forget to validate the WSDL files of your web services, specially if there is some manual edition applied on them.

[3] `http://docs.oracle.com/javase/6/docs/api/java/lang/ClassLoader.html#getResourceAsStream(java.lang.String)`

- Ensure that the service port address in the WSDL file (see Listing 3.11), when seen from remote locations, do not use "localhost", "0.0.0.0", or other unsuitable addresses, as lan private IPs for example. Remember the client that sees the WSDL needs an accessible endpoint.

Listing 3.11: Good example of service port address on a WSDL file.

```
1 <service name="AirlineServiceService">
2   <port name="AirlineServicePort" binding="tns:AirlineServicePortBinding">
3     <soap:address location="http://200.221.3.47:1234/airline"/>
4   </port>
5 </service>
```

- If service A needs to invoke service B, there is no problem if service A is compiled with classes used to build service B. Actually, usually it is very useful to A having access to B interfaces. Nonetheless, this class dependency must be only static. There is no point in service A trying to access objects states of service B, or access resources, as configuration files, bundled in service B package.

- The `packageUri` attribute defines the URL from where the Enactment Engine retrieves the package to be deployed. Therefore, all the services packages need to be already Internet accessible at deployment time. This can be accomplished, for example, by hosting the packages on a web server.

- Packages cannot be downloaded from https URLs. This restricts using some services, such as Dropbox, to host the packages.

Chapter 4

Extending Enactment Engine

4.1 Introduction

Current PaaS solutions available on market are well known for their low flexibility. Some of them work only on a specific cloud environment, others only with a few development frameworks. Enactment Engine tries to overcome this issue by providing an extensible architecture. Although the out-of-box version of Enactment Engine is quite limited, with some programming is possible to extend it to provide support to new *i)* cloud providers, *ii)* package types, *iii)* service types, and *iv)* node selection policies. By "extending" we mean no current Enactment Engine code need to be changed, and that each new extension can be implemented by the means of a well-defined process, which are now described in this section.

4.2 Supporting new cloud providers

In Enactment Engine, cloud providers are just a source of virtual machines provisioning. Any technology able to create new virtual machines may be used as "cloud provider".

To implement a new cloud provider, it is necessary to implement the CloudProvider interface (Listing 4.1). Current implementations are AWSCloudProvider (that uses EC2 service), OpenStackKeystoneCloudProvider, and FixedCloudProvider (that always points to the same user-defined VMs). An example of new cloud provider implementation could be the VirtualBoxCloudProvider, that would use VirtualBox on the developer machine to create new VMs (this is an example more suited to test environments).

Listing 4.1: CloudProvider interface.

```
1  public interface CloudProvider {
2
3      public String getCloudProviderName();
4
5      public CloudNode createNode(NodeSpec nodeSpec)
6          throws NodeNotCreatedException;
7
8      public CloudNode getNode(String nodeId)
9          throws NodeNotFoundException;
10
11     public List<CloudNode> getNodes();
12
13     public void destroyNode(String id)
14         throws NodeNotDestroyed, NodeNotFoundException;
15
16     public CloudNode createOrUseExistingNode(NodeSpec nodeSpec)
17         throws NodeNotCreatedException;
18
19     public void setCloudConfiguration(CloudConfiguration cloudConfiguration);
20
21 }
```

Implementations should use the `cloudConfiguration` object to retrieve configuration properties supplied by EE administrators. Such properties usually encompass user credentials to access the infrastructure provider service, and options such as VM types or images. The `cloudConfiguration` object is injected into the cloud provider instance by the EE.

Important note: in the current implementation, Enactment Engine is tailored to work with nodes running *Ubuntu 12.04*. Therefore, CloudProvider implementors should provide Ubuntu 12.04 nodes.

The next step is to edit the `extensible/cloud_providers.properties` file, located on EnactmentEngine resources folder. You must add a line in the format `NAME=full.qualified.class.name`, where the key is just an alias that you can freely define (since it does not conflict with other existing keys on the same file), and the value is the full qualified name of the CloudProvider implementing class. It is also necessary to recompile the EnactmentEngine project in such way it can access the implementing class. One suggestion is by adding your class in your local maven repository and edit the EnactmentEngine project's pom to make EnactmentEngine dependent on your project holding the new cloud provider

Finally, to use your new cloud provider, it is necessary to configure the `clouds.properties`, adding a cloud account whose `CLOUD_PROVIDER` property values the `NAME` defined in the `cloud_providers.properties` file.

4.3 Supporting new package types

Services may be delivered in different package types, such as JARs, WARs, etc. Each package type has its own specific deployment procedures, as well its specific process to start the service. When using different technologies, such as Python, to write new services, you will need to define a new package type, as well the deployment procedure associated with it. Such procedure is specified in Chef recipe.

So, the first step is to create a new Chef cookbook similar to the "jar" and "war" recipes already provided by Enactment Engine. These cookbooks are actually templates that EE will use to create specific cookbooks to each service to be deployed. You can use the the constants $PACKAGE_URL and $NAME within your cookbook recipe and attributes files. These constants will be injected by Enactment Engine to each specific recipe. You can have an idea about how to use them by looking to the WAR cookbook implementation, in Listing 4.2 and Listing 4.3. After writing the new recipe, you must associate this recipe to the new package type by editing the `extending/cookbooks.properties` file.

Listing 4.2: Recipe template for WAR deployment.

```
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 4.3: Attributes template for WAR deployment.

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

It is up to EE to "guess" the service URI too. A service URI follows the format `http://IP:PORT/CONTEXT`. And the CONTEXT formation rule is package type dependent. Therefore, when extending package type, it is necessary to create a new URIContextRetriever implementation and to link this implementation to its package type in the URIContextRetrieverFactory class. To make this relationship, it is enough to add a single line in the factory, by adding a new entry in the `classMap` variable. Both URIContextRetriever and URIContextRetrieverFactory classes are in the org.ow2.choreos.services.datamodel.uri package, on the EnactmentEngineAPI project.

Hint: if your package type is based on some kind of container to run the services, such as Tomcat, it may be a good idea to prepare a new image with this container already installed and running. So, you can configure EE to create VMs with an already running instance of your chosen container (e.g. JBoss). This strategy helps in achieving a faster deployment.

4.4 Supporting new service types

Although web services came to tackle the interoperability issue, today we have a couple of technologies implementing the concept of services. The main standards in this context are SOAP and REST, but other technologies could be used to implement services, such as JMS.

In the Enactment Engine context, the service type affects only how the `setInvocationAddress` is invoked. Therefore, to support a new service type, you have only to write a new ContextSender (Listing 4.4) implementation.

Listing 4.4: ContextSender interface.

```
1 public interface ContextSender {
2
3     public void sendContext(String serviceEndpoint,
4                             String partnerRole,
5                             String partnerName,
6                       List<String> partnerEndpoints) throws ContextNotSentException;
7 }
```

The final step is to edit the `extensible/context_sender.properties` file, located on EnactmentEngine resources folder. You must add a line in the format `SERVICE_TYPE=full.qualified.class.name`, where the key is the name of the new service type, and the value is the full qualified name of the ContextSender implementing class. It is also necessary to recompile the EnactmentEngine project in such way it can access the implementing class. Now you can create new service specs using the just-created service type! But be sure services implementation are prepared to receive the `setInvocationAddress` invocation.

4.5 Supporting new node selection policies

A node selection policy defines the *mapping* of services to cloud nodes. Since cloud nodes are dynamically created, node selection policies must be flexible, and not rely on hard-coded IPs. A node selection policy may define nodes to be used based on services non-functional properties. To create a new node selector, you must create a new NodeSelector (Listing 4.5) implementation. Pay attention that such implementation must be thread-safe, since multiple threads will invoke concurrently the method `select`.

Listing 4.5: NodeSelector interface.

```
1 public interface NodeSelector extends Selector<CloudNode, DeployableServiceSpec> {
2
3 }
4
5 public interface Selector<T, R> {
6
7     public List<T> select(R requirements, int objectsQuantity) throws NotSelectedException;
8
9 }
```

After writing the new node selector, you must associate this selector to a label by editing the `extensible/node_selector.properties` file, at EnactmentEngine resources folder. To use the new selector, finally, you must attribute the defined label to the NODE_SELECTOR property on the `ee.properties` file.

Chapter 5

Elasticity and QoS management

As mentioned in Chapter 2, the Enactment Engine is able to modify a choreography that is currently running. For instance, one may decide to switch from using a service offered by one provider to a compatible service by a different provider, or to increase/decrease the number of deployed replicas of a given service in order to adapt to fluctuations in usage load. To do this, the user simply uses the API again to submit an updated version of the choreography specification to the Enactment Engine and requests the redeployment of the choreography. The Enactment Engine, in turn, detects the modifications made to the choreography and performs the requested modifications, by deploying new versions of services, removing service replicas etc.

This capability, together with the flexibility offered by the CHOReOS monitoring subsystem, presents the user with the framework necessary to adjust the run-time environment of the choreography according to QoS parameters and constraints, such as response time or cost. In order to accomplish this, the user needs to create a separate daemon that acts both as a monitoring client and as a client for the Enactment Engine. As a client for the monitoring system, this daemon uploads rules to the Glimpse Monitoring CEP and awaits for notifications from it whenever such rules are triggered; as a client for the Enactment Engine, it requests modifications to running choreographies by sumitting updated Choreography Deployers when notifications are received from the monitoring subsystem.

An example of such a daemon is available in the Enactment Engine source code repository, in the "reconfiguration" directory. We show below some code snippets from this example daemon and explain its general mechanism.

During startup, the daemon loads predefined rules from a static file and submits them to the Glimpse monitor:

```
public static void main(String[] args) {
    String rules = Manager.ReadTextFromFile(
        this.getClass().getClassLoader().getResource("rules/SLAViolations.xml").getFile());
    new EnactmentEngineGlimpseConsumer([... properties ...], rules);
}
```

Whenever a rule is triggered, the daemon is notified and runs the code from a class whose name is contained in the notification event:

```
public void messageReceived(Message arg0) {
    ObjectMessage responseFromMonitoring = (ObjectMessage) arg0;
    response = (ComplexEventResponse) responseFromMonitoring.getObject();
    event = new HandlingEvent(response.getResponseValue(), response.getRuleName());
    Class<ComplexEventHandler> theClass;
    theClass = (Class<ComplexEventHandler>) Class.forName(
        "org.ow2.choreos.chors.reconfiguration.events." + event.getEventData());
    handler = theClass.newInstance();
    handler.handleEvent(event);
}
```

The submitted example rules define that whenever more than 5% of requests during the last 2 minutes have a response time above 120 miliseconds, new replicas of the service should be created (the class AddReplica should be run):

```
    when
        $ev : ResponseTimeEvent() over window:time(2m);

        Number( $eventSum : doubleValue ) from accumulate(
            $event : ResponseTimeEvent($ev.service == service, $ev.chor == chor, $ev.ip == ip)
                over window:time(2m), count($event)
        );

        Number( intValue > $eventSum*0.05 ) from accumulate(
            $sEvent : ResponseTimeEvent(
                value > 120, $ev.service == service, $ev.chor == chor, $ev.ip == ip)
                over window:time(2m), count($sEvent)
        );

    then
        ResponseDispatcher.NotifyMeValue("AddReplica",
            "eeConsumer", (String) $ev.ip, (String) $ev.service);
    end
```

Finally, the AddReplica class then interacts with the Enactment Engine to update the number of replicas of the service:

```
public void handleEvent(HandlingEvent event) {
    List<DeployableService> services = registryHelper.getServicesHostedOn(event.getNode());
    List<DeployableServiceSpec> serviceSpecs = registryHelper.getServiceSpecsForServices(services);

    Choreography c = registryHelper.getChor(event.getNode());
    ChoreographySpec cSpec = c.getChoreographySpec();

    for (DeployableServiceSpec spec : serviceSpecs) {
        for (DeployableServiceSpec s : cSpec.getDeployableServiceSpecs()) {
            if (s.getName().equals(spec.getName())) {
                s.setNumberOfInstances(s.getNumberOfInstances() + 1);
                break;
            }
        }
    }

    registryHelper.getChorClient().updateChoreography([... id ...], cSpec);
    registryHelper.getChorClient().enactChoreography([... id ...]);
}
```

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. 14

[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. 18, 21, 29, 37, 38

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999. 1

[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. 7, 20, 21

[Bre01] Eric A. Brewer. Lessons from giant-scale services. *Internet Computing, IEEE*, 5(4):46 –55, 2001. 15, 38

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

[Bro09] Tim Brown. *Change by Design: How Design Thinking Transforms Organizations and Inspires Innovation*. HarperBusiness, 2009. 1

[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, 43

[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. 2, 9, 17, 21, 29

[Dea07] Alan Dearle. Software deployment, past, present and future. Em *2007 Future of Software Engineering (FOSE '07)*, páginas 269–284. IEEE, 2007. 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

[DMG07] Paul M. Duvall, Steve Matyas e Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007. 1

[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

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Tese de Doutorado, University of California, 2000. http://www.ics.uci.edu/ fielding/pubs/dissertation/abstract.htm. 6

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern, Janeiro 2004. http://martinfowler.com/articles/injection.html. 5, 28

[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, 37, 38, 52

[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, 17, 18, 38

[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

[Jai91] Raj Jain. Capítulo 18: 2^{kr} factorial designs with replications. Em *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design Measurements Simulation And Modeling*. Wiley, 1991. 52

[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. Capítulo 9: Creating scalable web services with REST. Em *Developing Web 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. 30, 54

[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. 17, 21, 28

[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

[Mic02] Microsoft and BEA Systems and IBM. Web Services Transaction (WS-Transaction), 2002. http://msdn.microsoft.com/en-us/library/ms951259.aspx. 6

[MK90] Jeff Magee e Jeff Kramer. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990. 30, 54

[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, 28, 37, 38

[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, 29

[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, 33

[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. 6, 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

[Pap09] Dimitri Papadimitriou. Future internet. The cross-ETP vision document. Relatório técnico, Janeiro 2009. 1

[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. 40

[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. 1, 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, 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. Capítulo 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. 11, 12, 15, 19, 40

[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. 30, 54

[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, 6

[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

[W3C07] W3C. SOAP version 1.2, Abril 2007. http://www.w3.org/TR/soap12-part1/. 6

[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, 21

[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, 18, 21

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

[WRPK07] D Yu Weider, Rachana B Radhakrishna, Sumana Pingali e Vijaya Kolluri. Modeling the measurements of qos requirements in web service systems. *Simulation*, 83(1):75–91, 2007. 14

[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