Alessandra Adami Pinto

Avaliação de um Algoritmo para Composição Automática de Web Services

Itajubá - MG

2016

Alessandra Adami Pinto

Avaliação de um Algoritmo para Composição Automática de Web Services

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação

Universidade Federal de Itajubá – UNIFEI

Mestrado em Ciência e Tecnologia da Computação

Programa de Pós-Graduação

Orientador: Prof. Dr. Bruno Tardiole Kuehne

Coorientador: Prof. Dr. Otávio Augusto Salgado Carpinteiro

Itajubá - MG

2016

Este trabalho é dedicado aos meus pais.

Agradecimentos

Eu gostaria de agradecer aos meus pais que apoiaram na decisão de fazer o mestrado e em todos os momentos me incentivaram nas dificuldades e ao Eng. Phyllipe Lima que nunca me deixou desanimar.

Ao meu orientador, Prof. Dr. Bruno Kuehne, que muito me ajudou na conclusão dessa dissertação, tanto nos aspectos teóricos quanto práticos, agradeço a paciência e confiança em mim depositada. Gostaria de agradecer ao meu co-orientador, Prof. Dr. Otávio Carpinteiro pela revisão do texto e pelo auxílio dado no início deste trabalho.

Cabe aqui agradecer aos colegas do grupo Gpesc que sempre me ajudaram quando foi preciso e que sempre estavam dispostos a fazer um lanchinho!

"Dont't think you are. Know you are."
(Morpheus, capitão do hovercraft Nebuchadnezzar)

Resumo

A composição de *Web Services* é um tema amplamente explorado na literatura sob diferentes aspectos. Contudo, observou-se que essas pesquisas não são voltadas para o processo como um todo: da requisição criada e enviada por um cliente até o recebimento de uma resposta por este, passando antes pelas etapas de composição dos serviços e execução do fluxo de trabalho. A partir dessa brecha, o presente trabalho mostra o desenvolvimento de um sistema – implantado em rede local – capaz de realizar todas as etapas citadas anteriormente, além de medir o tempo gasto em cada uma delas. Para tal, realizou-se a integração entre as ferramentas AWSCS e EPESWS com o objetivo de fazer a avaliação de desempenho de uma composição automática de *Web Services*. Os resultados aqui exibidos conseguem revelar qual etapa é o gargalo do sistema, ou seja, aquela que leva mais tempo para sua realização.

Palavras-chaves: Composição. SOA. *Web Services*.

Abstract

The automatic composition of Web Services has been fully explored in the literature from a number of different standpoints. However, it is observed that those researches weren't executing the whole process: from the time of the request created and sent by the client to the delivery of the results, but composing services and executing the workflow before. The awareness of this gap, this present work shows the development of a system – implanted on a local network – capable of performing all these steps described before, besides measuring the times of the different activities carried out by the tool. This article examines the integration between the AWSCS and EPESWS tools with the aim of conducting a performance evaluation of an automatic composition of Web services. The results could determine which of these activities are the system's bottleneck, i.e., took up most time during the execution.

Key-words: Composition. SOA. Web Services.

Sumário

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ASC	Automatic Service Composition
ASF	Apache Software Foundation
CORBA	Common Object Request Broker Architecture
ESB	Enterprise Service Bus
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IOPE	Input, Output, Preconditions and Effects
JSP	Java Server Pages
OWL	Web Ontology Language
OWL-S	Semantic Markup for Web Services
QOS	Quality of Service
REST	Representational State Transfer
RMI	Remote Method Invocation
SAWSDL	Semantic Annotations for WSDL and XML Schema
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SWS	Semantic Web Services
UDDI	Universal Description, Discovery and Innovation
URL	Uniform Resource Locator
XML	eXtensible Markup Language
WS	Web Service
WSDL	Web Service Description Language
WSMO	Web Service Modeling Ontology

Lista de símbolos

$\in$	Pertence
$\subseteq$	Subconjunto

1 Introdução

No início da *Web* como é conhecida hoje, as poucas páginas existentes eram sempre estáticas, pois forneciam apenas texto e figuras em alguns casos. Não havia interação entre o usuário consumidor do material sendo exibido e o fornecedor desse material.

Isto é exatamente o contrário do que acontece entre *Web* e usuários em dias atuais. O que se vê atualmente é uma interação constante entre páginas e pessoas e também entre as próprias páginas. Alguns sistemas de *software* já são considerados indispensáveis no cotidiano, como sistemas de ponto eletrônico de empresas, hospitalares, bolsa de valores, controle de tráfego aéreo, entre outros.

Todavia, tamanha exigência de rápido desenvolvimento com precisão na tarefa executada e uma complexidade cada vez maior dos *softwares* elaborados levaram as empresas desenvolvedoras a uma corrida competitiva em seus negócios. Dessa forma, aqueles que conseguem resistir à pressão de seus clientes, reduzir seus custos, tempo de construção e número de recursos, saem em vantagem sobre seus concorrentes. Essas condições desfavoráveis impostas aos desenvolvedores (programadores) levou a melhorias no campo da engenharia de *software*, uma vez que "reuso" tornou-se a palavra de ordem (SYU; FANJIANG, 2013).

A reutilização de *software*, quando feita de forma planejada e controlada, pode levar a um aumento na produtividade, já que reduz o tempo gasto por desenvolvedores em determinadas tarefas e evita a reconstrução de elementos existentes. Além disso, quando um novo *software* está sendo criado, a reutilização de partes que já foram documentadas, testadas e validadas previamente leva a um aumento na qualidade deste (SYU; FANJIANG, 2013).

Dentre as técnicas de reuso está a arquitetura orientada a serviços (SOA - *Service-Oriented Architecture*). Esta arquitetura - que será explicada em mais detalhes no Capítulo 2 - tem como fundamento que as funcionalidades presentes nas aplicações sejam apresentadas como serviços. Os serviços são componentes de *software* com uma função específica, e podem ser disponibilizados para outro sistema ou outros serviços, no caso de serviços compostos.

Os serviços compostos são aqueles formados pela junção de dois ou mais serviços básicos que, quando unidos, geram um novo com uma nova funcionalidade. Estes serviços não precisam, necessariamente, pertencer à mesma empresa, desenvolvedor ou serem escritos com a mesma linguagem de programação, eles são oferecidos como produtos de negócios e é comum que a comunicação entre um sistema cliente e um consumidor seja feita com o uso de *Web Services*.

De acordo com Bartalos, Bieliková e Hluchý (2011), os *Web Services* possuem como benefício principal a interoperabilidade. Esta propriedade — assim como no caso dos serviços explicados previamente — permite que, mesmo no caso de sistemas de *software* diversos, um seja capaz de utilizar as funcionalidades do outro. Ademais, sistemas complexos podem ser construídos através da integração de aplicações diversas e independentes de plataforma não importando onde estas estão sendo executadas. Estes sistemas complexos são muitas vezes formados por uma combinação de *Web Services* e o processo de organização desses serviços em um fluxo de trabalho é chamado de composição.

A composição de *Web Services* era originalmente realizada através do trabalho braçal de especialistas de serviços (SYU et al., 2014), mas como o número de serviços sendo disponibilizados é crescente, tal tarefa tornou-se impraticável, pois, deve-se levar em conta que alguns destes podem ser descontinuados ou se tornarem indisponíveis por algum problema de rede. A partir dessa necessidade, a composição automática de serviços (ASC - *Automatic Service Composition*) foi proposta com o intuito de descobrir e compor diversos serviços de maneira a satisfazer uma consulta (JIANG et al., 2012), eliminando a intervenção humana.

1.1 Motivação

O presente trabalho utiliza ASC como parte do processo completo de composição de serviços. Este processo se inicia quando o usuário envia uma requisição ao sistema, que por sua vez encontra os serviços necessários, os executa e retorna ao cliente uma resposta. Até a elaboração deste, não foram encontradas pesquisas que realizassem todas essas etapas descritas anteriormente, da requisição do usuário à execução dos serviços escolhidos como solução. Os trabalhos utilizados como referências focam apenas no desenvolvimento e aprimoramento de algoritmos de composição e seleção de *Web Services*. Estas pesquisas desenvolvem métodos para torná-los mais rápidos, com menor uso de memória, com a criação da composição seguindo determinada exigência, ou ainda, trabalhos que implementam ambientes que são capazes de comparar tais algoritmos sob as mesmas condições. Entretanto, as requisições acontecem em ambiente simulado ou de maneira "local", ou seja, na mesma máquina. Além disso, apesar de o resultado da composição ser avaliado sob diferentes critérios de autor para autor, os serviços utilizados por esta não chegam a ser executados. Observando essa brecha, o presente trabalho teve como motivação criar um sistema, em rede local, que fosse capaz de aceitar a requisição do usuário e enviar sua resposta, passando pela composição e execução dos serviços solicitados.

1.2 Objetivo

O objetivo principal deste trabalho de mestrado é avaliar o tempo de resposta gasto pelo sistema desenvolvido desde o acesso do cliente — quando este envia uma requisição de serviços ao sistema —, elaborando a composição através de um algoritmo de ASC, até a entrega das informações por este requisitadas, após a execução dos serviços selecionados. Para tal, houve a integração de duas ferramentas que também serão aqui explicadas. Além disso, o sistema é capaz de identificar qual etapa nesse processo apresenta-se como a mais longa.

1.3 Organização do trabalho

O restante dessa dissertação está organizado como se segue: o Capítulo 2 apresenta mais detalhes sobre SOA e *Web Services*. O Capítulo 3 explica a composição de *Web Services* e cita alguns algoritmos para a composição automática. O Capítulo 4 mostra as ferramentas utilizadas no trabalho e o desenvolvimento deste. Os experimentos e resultados realizados com o sistema proposto estão presentes no Capítulo 5. Por último, tem-se as conclusões e trabalhos futuros no Capítulo 6.

2 SOA - Service-Oriented Architecture

A Internet é o meio de troca de informações que mais cresceu nas últimas décadas. Contudo, esse crescimento rápido levou a algumas questões, por exemplo: como conectar tipos de dados diferentes a uma mesma rede global, como atualizar o estado da rede em sistemas sem fio, ou ainda, como fazer com que uma informação esteja disponível para ser descoberta. Esses problemas mostram como as tecnologias de rede são heterogêneas. Fez-se necessário um novo padrão de arquitetura que permitisse a coexistência dada uma ampla variedade de aplicações (CHEN, 2010). A partir disso, foi desenvolvido a SOA, acrônimo em inglês para arquitetura orientada a serviço.

2.1 SOA

Este novo paradigma não é, de fato, uma arquitetura concreta, mas sim algo que a tenha como objetivo. Pode ser definida como uma metodologia para auxiliar no desenvolvimento de *software* ou serviços e manutenção para o processo de negócios em grandes sistemas distribuídos. A principal vantagem da SOA é o aumento de produtividade e na gerência, redução de custos e alta flexibilidade, ocasionados pela reutilização de *software* (THIES; VOSSEN, 2008). Há três pilares técnicos utilizados pela SOA: serviços, interoperabilidade por meio de barramento de serviço corporativo e baixo acoplamento (JOSUTTIS, 2007):

- Um serviço é uma unidade independente que contém uma funcionalidade básica (como armazenar ou recuperar dados de um cliente) ou complexa (como um processo de negócio para atender a um pedido do cliente) que representa uma atividade do mundo real.

- O barramento de serviço corporativo (ESB - *Enterprise Service Bus*) é a base que permite que haja interoperabilidade entre sistemas distribuídos para serviços. Ainda facilita a distribuição dos processos de negócio em múltiplos sistemas em plataformas e tecnologias distintas. Também pode ser responsável por prover conectividade, roteamento inteligente, lidar com segurança e confiabilidade, gerenciamento de serviços, entre outros.

- O baixo acoplamento é responsável por reduzir as dependências do sistema, ou seja, diminuir o grau de conexões entre partes/artefatos desse sistema. Essa característica deve ser almejada para facilitar o desenvolvimento independente dos artefatos, evitando que uma alteração ou correção em um deles resulte em uma série de altera-

ções em cadeia nos demais. Em contrapartida, a complexidade de desenvolvimento, manutenção e depuração aumenta significativamente.

Além do ponto de vista técnico, um conceito utilizado pela SOA é a centralização. Essa característica faz com que a introdução de uma nova funcionalidade não seja responsabilidade de um departamento apenas, mas que todas as decisões sejam tomadas de maneira colaborativa visando o melhor para a companhia. Trata-se de um conjunto de boas práticas que permite melhorar o relacionamento entre diferentes setores de uma empresa.

Na abordagem clássica de SOA, existem três "protagonistas" – descritos a seguir – que utilizam serviços. O primeiro deles é o provedor de serviços, que apresenta e, obviamente, provê um serviço. Em outras palavras, o provedor fornece as interfaces para um *software* responsável por gerenciar tarefas específicas.

O segundo é o serviço consumidor, que pesquisa e por vezes encontra um provedor adequado as suas necessidades. Muitas vezes, o consumidor representa uma aplicação de negócios que realiza chamadas remotas ao provedor de serviços, sendo que este último pode estar localizado na mesma rede local ou precisar ser acessado via *Internet*. Detalhes como rede, protocolos de transporte e segurança ficam sob responsabilidade de implementações específicas.

A última entidade é o serviço de registro. Ele é responsável por conhecer seus serviços disponíveis e pode estabelecer contato entre um provedor e um cliente. Como atua como um repositório, é muitas vezes comparado às páginas amarelas de uma lista telefônica, pois os serviços estão separados por categorias. É nele que os provedores de serviços publicam seus serviços.

As três entidades descritas nos parágrafos anteriores realizam três operações básicas entre si: publicação, busca e utilização. O provedor de serviços os publica no serviço de registro e o serviço consumidor busca os serviços desejados através do registro, e, finalmente, os utiliza. Conforme apresentado na Figura 1.

Sabe-se que *Web Services* são a tecnologia de conexão mais frequente em arquiteturas orientadas a serviço e que para uma SOA ser considerada bem feita, os serviços devem ser projetados adequadamente (SANTANA, 2009). A próxima seção explica algumas particularidades dos *Web Services* que são relevantes a esse trabalho.

2.2 Web Services

Pode-se entender como *Web Service* qualquer serviço que esteja disponível na *Internet*; possua uma funcionalidade bem definida; dependa ou não do estado de outros serviços; utilize um sistema de mensagens XML (*eXtensible Markup Language*) padro-

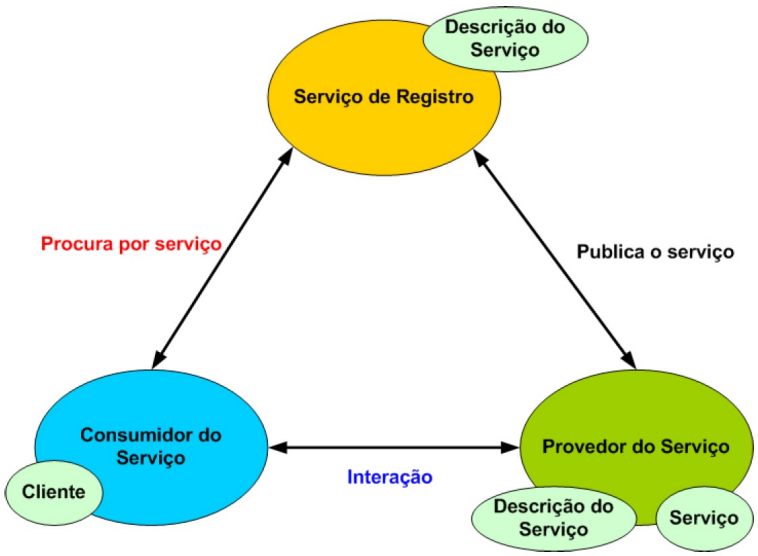


Figura 1 – Arquitetura Orientada a Serviço (KUEHNE, 2015), adaptado de (BIH, 2006)

nizado e não esteja vinculado a nenhum sistema operacional ou a alguma linguagem de programação (CERAMI, 2002).

Apesar de não ser mandatório, um *Web Service* pode ter mais duas propriedades adicionais:

- Ser autodescritivo: quando um novo serviço é publicado, uma interface pública para o serviço também deve ser divulgada. Uma interface pública pode ser descrita em uma gramática XML comum, que pode ser utilizada para identificar seus métodos públicos, argumentos e valores de retorno. Cada novo serviço deve conter, pelo menos, uma documentação legível para que outros desenvolvedores possam se integrar ao serviço.

- Ser de fácil descoberta: devem haver mecanismos simples que possibilitem que partes interessadas sejam capazes de encontrar o serviço e sua interface pública.

Os *Web Services* realizam a comunicação entre diferentes sistemas utilizando a linguagem de marcação XML e o protocolo HTTP (*Hypertext Transfer Protocol*). A linguagem XML é utilizada para descrever os dados trafegados, e já o HTTP é um protocolo de troca de mensagens usado na *Internet* (SANTANA, 2009). Como pode ser observado na Figura 2, é por meio da troca de mensagens e do envio de dados através de XML que ocorre a comunicação. Uma vez que esses dois protocolos somente definem como esta comunicação vai ocorrer, novos padrões foram desenvolvidos para que o uso de *Web Services* se tornasse mais eficiente. WSDL, SOAP e UDDI são alguns exemplos desses padrões que serão descritos nas subseções 2.2.1, 2.2.2 e 2.2.3.

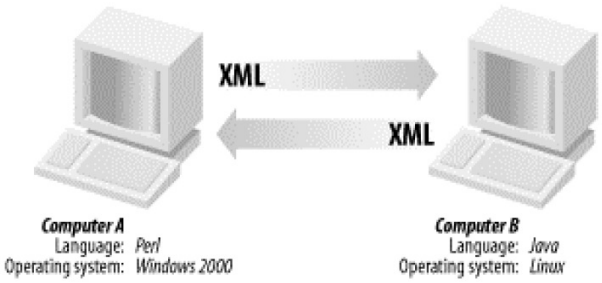


Figura 2 – Troca de mensagens através de XML, (CERAMI, 2002)

2.2.1 WSDL - Web Service Description Language

WSDL é uma linguagem baseada em XML utilizada para descrever *Web Services*, mas sem informação semântica. Dentre os elementos descritos estão as informações sobre: as interfaces que descrevem todas as funcionalidades disponíveis publicamente; os tipos de dados de todos os pedidos e respostas de mensagens; o protocolo de transporte a ser usado e o endereço para localização do serviço em questão, seu URL (*Uniform Resource Locator*).

Mais especificamente, as *tags* utilizadas no arquivo WSDL são *service*, *binding*, *portType*, *message* e *types*. A *tag service* (serviço) define o endereço para a invocação do serviço especificado. Em *binding* (ligação), são especificados os protocolos e os formatos de dados das mensagens e operações definidas no *portType*. O *portType* (tipo da porta) define as operações disponíveis no serviço. Já *message* (mensagem), é uma definição abstrata da informação enviada pelo *Web Service*, seja ela de solicitação ou de resposta. E em *types* (tipos), são definidos os tipos de dados usados nas mensagens trocadas.

A partir desse arquivo de descrição, um cliente pode localizar um *Web Service* e invocar suas funcionalidades públicas. Pode-se até empregar ferramentas que automatizam esse processo, de forma a facilitar a integração de novos serviços com pouca ou mesmo nenhuma intervenção manual. WSDL é considerada a base da arquitetura de *Web Service*, pois provém uma linguagem comum para sua descrição e, também, uma plataforma para integração automática desse tipo de serviço. Fazendo uma analogia com a linguagem de programação Java, WSDL representa um contrato entre o serviço consumidor e o provedor de serviços, assim como uma interface Java representa um contrato entre o código cliente e o objeto (CERAMI, 2002).

2.2.2 SOAP - Simple Object Access Protocol

SOAP é um protocolo baseado em XML para troca de mensagens entre computadores, independente de plataforma e linguagem de programação. Dessa forma, tornou-se um elemento fundamental da infraestrutura de *Web Services*, pois permite que aplicações diversas troquem dados e serviços entre si. Por exemplo, um cliente Java SOAP

executando em Linux ou um cliente Perl executando em Solaris pode se conectar em um servidor SOAP Microsoft executando Windows 2000 (CERAMI, 2002).

O protocolo SOAP contém informações sobre como utilizar o *Web Service*, por exemplo, o formato da transferência de dados e informações requisição e resposta. A estrutura de um arquivo SOAP é composta por três elementos: "<Envelop>", "<Header>" e "<Body>", sendo que os dois últimos ficam contidos no primeiro (o elemento raiz) (SANTANA, 2009). A seguir tem-se a descrição de cada um desses elementos.

- *Envelope* (Envelope): Toda mensagem SOAP contém obrigatoriamente este elemento. É responsável por definir regras para o encapsulamento dos dados que serão transferidos entre dois pontos. O que inclui dados específicos da aplicação, como o nome do método que será invocado, seus parâmetros ou valores de retorno, e, ainda, alguma informação sobre "quem" deve processar o conteúdo do envelope e, caso haja falha, como lidar com mensagens de erro.

- *Header* (Cabeçalho): Parte opcional, mas quando for utilizado, deve ser o primeiro elemento do *Envelope*. Possui informações que auxiliam no processamento da mensagem, como o formato dos dados e o tamanho ou se a mensagem deve ser processada ou não por nós intermediários na rede (roteamento), e de segurança. São informações adicionais que ajudam a infraestrutura a lidar com as mensagens.

- *Body* (Corpo): Parte obrigatória que contém o *payload* (dados de requisição, resposta ou falha), isto é, a informação de fato que deve ser transportada a um destino final. As mensagens de *status* e erros que são retornadas pelos nós intermediários quando esses processam uma mensagem, podem ser carregadas por um elemento *Fault* que é opcional, mas pode estar contido dentro do *body*.

2.2.3 UDDI - Universal Description, Discovery and Innovation

UDDI é, como seu próprio nome anuncia, uma especificação técnica para descrever, descobrir e integrar *Web Services*. Isso faz com que seja responsável pela localização de *Web Services*, funcionando como um repositório. Além disso, contém especificações sobre como determinado *Web Service* deve ser acessado ou modificado. Ou seja, fornece aos seus usuários uma maneira unificada e sistemática de encontrar provedores através de um registro centralizado de serviços. Pode-se compará-lo, de forma grosseira, a uma lista telefônica *online* e automatizada de *Web Services* que contém "páginas brancas", "páginas amarelas" e "páginas verdes" (CURBERA et al., 2002).

O UDDI tem em suas "páginas brancas" (*White pages*) informações sobre uma companhia específica, como nome e descrição do negócio, detalhes para contato, endereço e números de telefone. Também pode incluir identificadores unívocos de negócio. Suas

"páginas amarelas" (*Yellow pages*) fornecem uma classificação para a companhia ou o serviço oferecido baseado em seus tipos. As "páginas verdes" (*Green pages*) contém informações técnicas sobre um *Web Service*. Normalmente, um ponteiro para uma especificação externa e um endereço para invocar o *Web Service*.

UDDI não descreve apenas *Web Services* baseados em SOAP, mas pode ser utilizado para descrever qualquer serviço, de uma página *web* a um endereço de *email*, passando por SOAP, CORBA (*Common Object Request Broker Architecture*) e serviços Java RMI (*Remote Method Invocation*) (CERAMI, 2002).

2.2.4 REST - Representational State Transfer

Cabe aqui citar uma alternativa mais simples ao uso dos *Web Services* SOAP. RESTful HTTP, ou simplesmente REST como é mais utilizado, é uma coleção de princípios de arquitetura de rede focada no acesso simples e *stateless* a recursos. REST utiliza quatro métodos fundamentais do HTTP: *get*, *put*, *post* e *delete*, para ler, escrever, criar e apagar recursos identificados por URLs (JOSUTTIS, 2007).

Quando comparado aos serviços tradicionais baseados em WSDL, os *Web Services* RESTful possuem como vantagens segurança e flexibilidade. Grandes empresas como Google, Facebook e Amazon expõe seus recursos via esse tipo de serviço. Entretanto, podem ser de difícil identificação na *Internet*, pois não há regras bem definidas para seu desenvolvimento. Os serviços RESTful são descritos em páginas HTML o que faz com que ferramentas de busca orientadas por WSDL tenham dificuldade em encontrá-los (ZHAO et al., 2014).

Dessa forma, caso um desenvolvedor deseje apenas fornecer acesso a seus dados e recursos, o princípio REST deve ser levado em conta, mas caso ele pretenda definir um cenário SOA de execução de processos de negócios distribuídos, REST não é a melhor opção (de acordo com Josuttis (2007) pelo menos por enquanto).

2.3 Web Services semânticos

A *Web* foi inicialmente desenvolvida para ser utilizada por seres humanos, mas este conceito foi mudando a medida em que aumentaram os esforços para torná-la mais automatizada e acessível também às máquinas. Isso ocorreu devido ao seu crescimento, já que o processamento de seus documentos tornou-se inviável se feito apenas por usuários humanos. Com isso, veio a necessidade de criarem-se informações que possuíssem conteúdo semântico que pudesse ser processado via máquina. Trata-se do cerne da *Web* Semântica.

Para entender a *Web* Semântica, é preciso saber a definição do termo "semântica". Semântica pode ser entendida como significado. Este significado permite o uso mais efetivo

dos dados, e frequentemente está ausente de suas fontes de informação, o que faz com que sejam necessários usuários ou instruções complexas de programa para supri-las.

A semântica mostra o quão mais sofisticada pode ser a exploração de recursos *Web*. A proposta da *Web* Semântica é desenvolver padrões de documentos e tecnologias para que seja possível descrever de forma estrutural e semântica o conteúdo disponibilizado na *Web*, mas sem prejudicar o acesso àqueles que foram desenvolvidos ao longo de todos esses anos.

Web Services que possuem dados adicionais que descrevem semântica são chamados de *Web Services* semânticos. Eles são o resultado da evolução da *Web* em duas direções: adição de elementos dinâmicos a *Web* e a melhoria da descrição sintática dos *Web Services*. No seu início, a *Web* oferecia apenas documentos, conteúdos estáticos. Contudo, houve uma evolução neste ambiente de modo a admitir funcionalidades via serviços, elementos dinâmicos. A semântica veio resolver o conflito da ambiguidade que pode ser causada quando há apenas descrições sintáticas disponíveis, o que causa problema quando é preciso buscar por informações relevantes ou serviços.

No contexto de composição de serviços (que é explicada no Capítulo 3), a semântica ajuda a descobrir e organizar os que são relevantes a determinada requisição. Se não houvesse a semântica, apenas as descrições sintáticas dos serviços seriam consideradas, o que não é suficiente para reconhecer a funcionalidade de um serviço e, portanto, a composição automática de serviços - que atende a objetivos dinâmicos de um usuário - não conseguiria ser realizada (BARTALOS; BIELIKOVÁ; HLUCHÝ, 2011).

Quando um usuário faz uma requisição, se há uso de semântica, ele consegue se expressar melhor, pois os dados, tanto os que precisam ser fornecidos como aqueles que são recebidos, possuem significado bem definido. Este significado é fornecido pela sua ontologia, explicada a seguir.

2.3.1 Ontologia

A ontologia é um componente crucial na composição automática de serviços, é a espinha dorsal da *Web* Semântica. O termo ontologia vem da filosofia e na ciência da computação é utilizado para modelar e representar conhecimento dentro de um domínio específico. Como as ontologias capturam os conceitos e os relacionamentos entre eles, são utilizadas por computadores para fazer o raciocínio semântico.

De acordo com pesquisas teóricas sobre composição automática, a ontologia tem três propósitos principais, ou seja, há três tipos de conhecimento e informação que são representados pelas ontologias neste campo. A primeira utilização é modelar e encapsular conhecimento em um domínio específico, que é a forma mais comum de utilização de ontologias. Outra utilização são as ontologias de serviços, estas melhoram os servi-

ços com descrição sintática em serviços semânticos com interfaces processáveis e legíveis por máquinas. Ontologias de serviços facilitam a automação de composição e de outros tipos de processamento de serviços, como sua classificação e descoberta, superando a heterogeneidade entre as interfaces dos serviços. As ontologias de serviço e domínio estão diretamente relacionadas. As ontologias de domínio definem termos (conceitos) semânticos que são especificados e indicados por um elemento da ontologia de serviço que descreve uma interface (por exemplo, a entrada de um serviço). Estas são a base dos serviços semânticos. A última utilização das ontologias é a geração de fluxo de trabalho sobre o conhecimento de domínio (SYU; FANJIANG, 2013).

Para publicar e compartilhar ontologias na *Internet* existe uma linguagem chamada OWL (*Web Ontology Language*). Ela foi desenvolvida para ser utilizada por aplicações que fossem capazes de processar informação, ao invés de simplesmente apresentá-la[1]. Para os *Web Services*, existe uma especialização dessa linguagem, a OWL-S.

2.3.1.1 OWL-S

A OWL-S[2] (*Semantic Markup for Web Services*) foi proposta como uma ontologia de serviços que permite que usuários e agentes de *software* sejam capazes de descobrir, invocar, compor e monitorar, de forma automática, recursos *Web* que ofereçam serviços específicos e apresentem propriedades peculiares.

Na condição de uma linguagem de ontologias para *Web Services*, OWL-S permite que a semântica possa ser associada às especificações destes. Essa semântica pode ser explorada a fim de auxiliar na execução de tarefas como descoberta, composição, seleção e invocação de serviços, além de dar suporte ao desenvolvimento de ferramentas e metodologias associadas aos *Web Services* semânticos.

A linguagem OWL-S utiliza ontologias para descrever atributos funcionais e não funcionais de um serviço. Dessa forma, auxilia a composição automática de *Web Services* uma vez que permite a descrição semântica das capacidades do serviço (entradas, saídas), condições relativas às entradas (precondições), bem como as saídas (efeitos) (PRAZERES; TEIXEIRA; PIMENTEL, 2009).

2.4 QoS em Web Services

Com a popularização dos *Web Services*, há cada vez mais provedores de serviço competindo entre si para fornecer a mesma operação. Sendo assim, faz-se necessário averiguar a qualidade destes através de um critério conhecido como qualidade de serviço (QoS - *Quality of Service*). QoS, então, tornou-se um ponto de distinção entre serviços

[1] http://www.w3.org/TR/owl-ref/
[2] http://www.w3.org/Submission/OWL-S/

que são equivalentes em termos de suas funcionalidades. O termo QoS, nesse contexto, é amplamente utilizado para descrever características não funcionais dos *Web Services*, como estas citadas por Kalepu, Krishnaswamy e Loke (2003):

- Disponibilidade: indica se um *Web Service* está presente ou pronto para uso. Também pode ser representada como a porcentagem de tempo que um serviço fica *up* durante um intervalo de observação. Está relacionada à confiabilidade, explicada abaixo.

- Confiabilidade: é a probabilidade de um serviço responder corretamente dentro de um período de tempo esperado. Refere-se à entrega "segura e ordenada" de mensagens sendo enviadas e recebidas por serviços consumidores e provedores.

- Preço/Custo: corresponde a quantia cobrada pelo provedor de serviços para permitir acesso ao serviço.

- Taxa de transferência (*Throughput*): é o número de requisições de um *Web Service* atendidos em um dado período de tempo. Deve-se ter uma relação de compromisso entre o *throughput* e o tempo de resposta, pois quando o último aumenta, o primeiro também aumenta, sendo que o ideal é maximizar o *throughput* e minimizar o tempo de resposta.

- Tempo de resposta: pode ser definido como a quantidade de tempo entre uma requisição e sua resposta, levando em conta o tempo de processamento e de transmissão ou, ainda, como o tempo médio necessário para completar uma requisição de serviço. É muitas vezes chamado de latência.

- Latência: tempo entre a chegada da requisição do serviço e o pedido a ser atendido.

- Desempenho: é medido com o auxílio do *throughput* e do tempo de resposta. Quando um serviço possui alto *throughput* e baixo tempo de resposta, seu desempenho é considerado satisfatório.

- Segurança: é a capacidade de garantir o não repúdio de mensagens, confidencialidade e autenticação das partes envolvidas. Também é indicado pela "rastreabilidade", habilidade para criptografar dados e resistir a ataques de negação de serviço.

- Regularidade: diz respeito ao grau de conformidade do *Web Service* perante a lei, regras e cumprimento de acordos de níveis de serviços estabelecidos (sendo este último explicado ao final desta subseção).

- Robustez/Flexibilidade: é a habilidade de um serviço de suportar entradas inválidas, incompletas ou mesmo conflitantes e ainda assim produzir o resultado correto.

- Precisão: baixa probabilidade de um serviço gerar erro.
- Reputação: é a medida da confiança de um serviço e depende das experiências dos usuários finais que os utilizaram, ou seja, é a classificação média dada a determinado serviço pelos usuários finais. Embora seja frequentemente utilizada para classificar o provedor, pode não ser um indicador correto de competência, porque é baseado apenas nas experiências dos usuários finais que não necessariamente tem um bom conhecimento sobre o assunto e talvez não saibam transformar sua experiência em números.

Entretanto, a natureza dinâmica da *Internet* faz com que um mesmo serviço possa funcionar de forma diferente para diferentes usuários, pois, dentre os erros que podem ocorrer pode-se citar os arquivos de descrição que se tornam desatualizados ou simplesmente são retirados do ar ou mesmo estão com conteúdo inválido, com erro de sintaxe, ou sem conteúdo. Além disso, há problemas de rede que podem acontecer, como erros de *gateway*, no servidor, ausência de resposta do destino e outros (ZHENG; ZHANG; LYU, 2014).

Para garantir um desempenho adequado em *Web Services* com suporte à QoS, eles são associados a um acordo de níveis de serviço (SLA - *Service Level Agreement*). Trata-se de uma relação formal entre o provedor de serviços e o cliente, que necessita, muitas vezes, de uma terceira parte para realizar a promulgação desse contrato. O SLA é composto de descrições das partes envolvidas, das definições dos serviços, parâmetros de qualidade de serviços e seus níveis de serviços e também das sanções que podem ocorrer devido ao não cumprimento dos níveis acordados. O objetivo dos SLAs em *Web Services* é facilitar os relacionamentos complexos existentes entre provedores e garantir o desempenho das aplicações (KALEPU; KRISHNASWAMY; LOKE, 2003).

2.5 Considerações Finais

Neste Capítulo, foram apresentados os conceitos do paradigma SOA, bem como as definições de *Web Services* necessários a uma melhor compreensão teórica do trabalho que será exposto no Capítulo 4. Além disso, os padrões WSDL, SOAP e UDDI foram explicados já que tornaram o uso de *Web Services* mais eficiente. Este Capítulo também discorreu sobre os *Web Services* semânticos, uma evolução dos *Web Services* simples, que permite a descoberta e organização de serviços relevantes a determinada requisição. Por último, foi exibido o conceito de QoS em *Web Services*, um dos mais importantes a ser levado em conta na hora da escolha de um serviço quando se tem mais de um que poderia ser utilizado.

3 Composição de Web Services

O conceito de *Web Services* introduziu um novo paradigma que afetou tanto a computação distribuída quanto o desenvolvimento de sistemas de *software*. Nesse contexto, a composição de serviços se destaca como um mecanismo de combinação de múltiplos serviços que, além de aumentar o potencial desses individualmente, também é capaz de construir sistemas com funcionalidades mais complexas. Isso quer dizer que se não houver um serviço que consiga fornecer determinada funcionalidade sozinho, pode ser possível criar uma composição a partir de serviços independentes e, então, prover a funcionalidade desejada.

As próximas seções deste Capítulo apresentam um pouco mais sobre a fundamentação teórica acerca do tema composição de serviços. Em 3.1, há definições básicas sobre composição de *Web Services*. A seção seguinte, 3.2, mostra mais informações sobre a composição automática e conceitos presentes nessa questão. A última seção, 3.3, faz uma revisão bibliográfica de diferentes algoritmos de composição automática de *Web Services* encontrados na literatura. O conteúdo apresentado nesse Capítulo será necessário ao leitor para uma maior compreensão da execução e do funcionamento do sistema proposto no Capítulo seguinte.

3.1 Visão Geral

De acordo com Rodriguez-Mier, Mucientes e Lama (2011), para compor *Web Services* é preciso definir os relacionamentos entre serviços. Um *Web Service* w pode ser entendido como um componente de *software* que recebe um conjunto de entradas $W_{in} = \{I_1, I_2, ...\}$ e gera um conjunto de saídas $W_{out} = \{O_1, O_2, ...\}$ quando executado. Para que as saídas de um serviço possam ser utilizadas como entradas para outro, é preciso haver uma relação semântica entre eles. Essa restrição pode ser um relacionamento hierárquico de classe/subclasse entre conceitos, por exemplo, quando um conceito C_i é uma subclasse do conceito C_j, então há uma correspondência semântica entre C_i e C_j.

Uma requisição R de *Web Service* é composta de um conjunto de entradas $R_{in} = \{I_{1in}, I_{2in}, ...\}$ fornecidas pelo requisitante (usuário) e um conjunto de saídas $R_{out} = \{O_{1out}, O_{2out}, ...\}$ que é esperado pelo usuário. Dada uma requisição $R_{user} = \{R_{in}, R_{out}\}$, onde $R_{in} = \{I_{1R}, I_{2R}, ...\}$ e $R_{out} = \{O_{1R}, O_{2R}, ...\}$, e dado um *Web Service* $S = \{S_{in}, S_{out}\}$, onde $S_{in} = \{I_{1S}, I_{2S}, ...\}$ e $S_{out} = \{O_{1S}, O_{2S}, ...\}$, S só pode ser invocado se para cada entrada $I_S \in S_{in}$, existir uma entrada $I_R \in R_{in}$ sendo I_R igual ou uma subclasse de I_S ($I_R \subseteq I_S$). Além disso, R_{out} será satisfeito apenas se para cada saída $O_R \subseteq R_{out}$, existir uma saída $O_S \in S_{out}$ sendo que O_S é igual ou subclasse de O_R ($O_S \subseteq O_R$) (RODRIGUEZ-

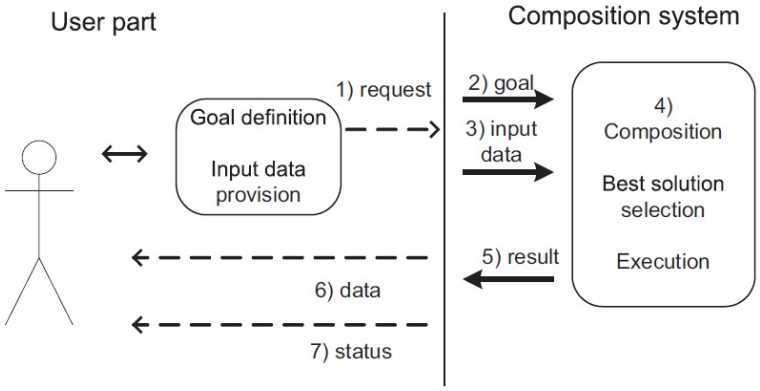


Figura 3 – Visão geral do processo de composição de serviços, retirado de (BARTALOS; BIELIKOVÁ; HLUCHÝ, 2011)

MIER; MUCIENTES; LAMA, 2011).

A Figura 3 ilustra o processo de composição de serviços. Este processo pode ser dividido em duas partes: a do usuário (*User part*) e a do sistema de composição (*Composition system*). Neste contexto, o usuário pode ser humano ou um sistema de *software*. É ele que inicia o processo fazendo uma requisição ((1) *request*) que contém uma descrição do objetivo ((2) *goal*), o usuário pode ter que fornecer ainda algum dado de entrada que seja necessário ((3) *input data*) ou o próprio sistema de composição detecta quais dados estão sendo exigidos. O próximo passo é a realização do projeto do fluxo de trabalho e a execução ((4) *Composition, Best solution selection, Execution*). Normalmente, não existe apenas uma solução possível. A solução escolhida deve levar em conta o grau de satisfação do usuário baseado na qualidade do resultado apresentado, ou seja, suas preferências. Mas também pode-se escolher uma solução que se preocupe mais com a qualidade do processo de execução afetado pelos atributos de QoS. Um sistema ótimo deve escolher uma solução subótima que tenha uma relação de compromisso entre esses dois aspectos, pois geralmente não há um resultado que os atenda simultaneamente. A solução encontrada é, finalmente, executada para produzir o resultado desejado ((5) *result*). Caso o usuário tenha requisitado dados, eles são extraídos do resultado e entregues a ele ((6) *data*). O usuário também é informado sobre o *status* do resultado da execução, por exemplo, se foi bem sucedida ((7) *status*).

Na Figura 4, tem-se um exemplo de composição de serviços. Neste caso, as informações de entrada são o ponto turístico (*A - Tourist Site*) e a data da viagem (*B - Date*), as saídas desejadas pelo usuário são informações sobre o tempo (*C - Weather*) e hotéis (*D - Hotels*) no local de destino. Como não foi encontrado no repositório de serviços algum que, sozinho, satisfizesse a requisição, foi necessário compor, conectar mais de um serviço. Serviços podem ser conectados como em uma estrutura de grafo, onde os vértices são os próprios serviços e as arestas representam a relação entre eles, como visto na parte mais baixa da figura. A Tabela 1 explica como os serviços dessa composição se relacionam.

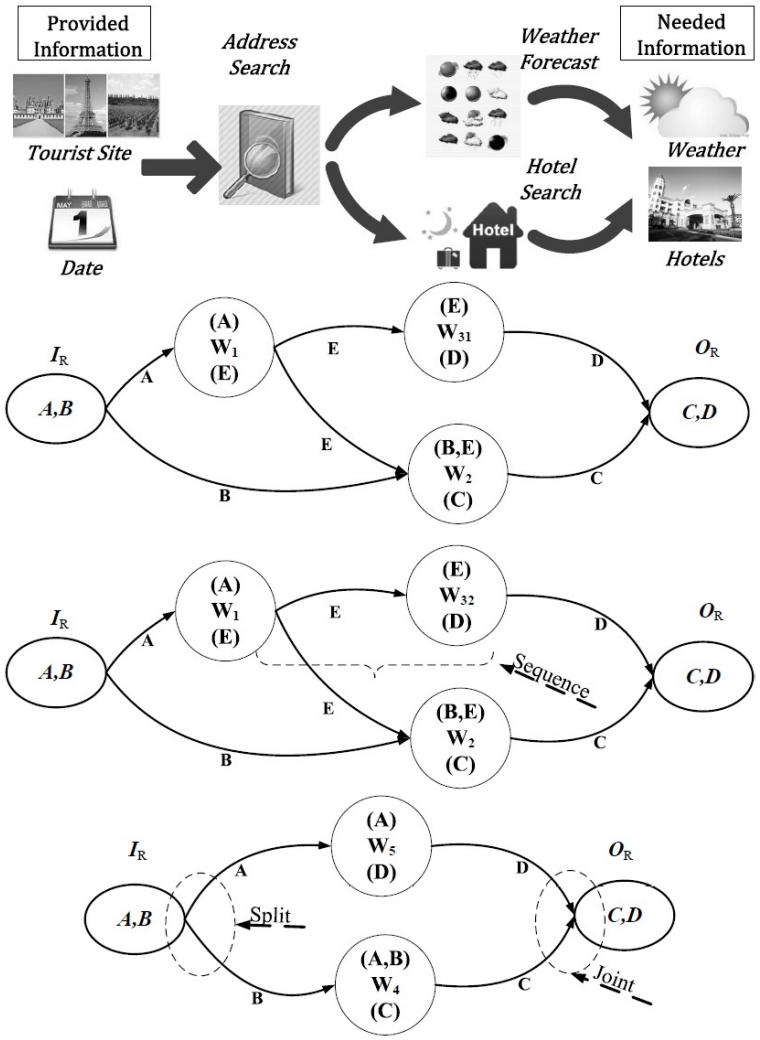


Figura 4 – Exemplo de composição de serviços, retirado de (JIANG et al., 2012)

Ainda sobre a Figura 4, pode-se observar que há caminhos sequenciais e paralelos dentro de um fluxo de execução de serviços. Os caminhos sequenciais são executados na ordem em que foram encadeados, o que ocorre sobre uma única *thread*. Já nos caminhos paralelos, como as atividades são executadas simultaneamente, há *threads* de controle individuais para cada uma. A última ilustração da Figura 4 exibe os construtores de fluxo *Split* e *Join* responsáveis por controlar esse paralelismo, sendo que o primeiro não exige uma sincronização entre a execução das atividades e no segundo, é criada uma barreira que sincroniza o término das atividades anteriores (SILVA, 2012).

A composição de serviços pode ser dividida em duas perspectivas: a primeira é a composição estática e a dinâmica, e a segunda se divide entre composição manual e automática. A primeira abordagem diz respeito ao momento em que um serviço é selecionado para integrar uma composição. Na composição estática, isso é feito durante a etapa de desenvolvimento, já na composição dinâmica, a seleção de serviços é feita ao longo da execução, sempre de acordo com os objetivos do usuário (BARTALOS; BIELIKOVÁ; HLUCHÝ, 2011).

Tabela 1 – Os serviços e a requisição

Web Service e Requisição	*Entradas*	*Saídas*	*Descrição*
Requisição (R)	Ponto turístico (A) e Data (B)	Tempo (C) e Hotéis (D)	A requisição provê o ponto turístico e a data da viagem e deseja como saída a previsão do tempo e os hotéis na cidade.
Busca de endereço (W_1)	Ponto turístico (A)	Cidade (E)	Encontra a cidade em que o ponto turístico está.
Previsão do tempo (W_2)	Cidade (E) e Data(B)	Tempo (C)	Retorna o tempo nesta cidade na data fornecida.
Busca de hotel (W_{31}, W_{32})	Cidade (E)	Hotéis (D)	Encontra hotéis na cidade dada.
Previsão do tempo (W_4)	Ponto turístico (A) e Data (B)	Tempo (C)	Retorna a previsão do tempo no ponto turístico na data.
Localização do hotel (W_5)	Ponto turístico (A)	Hotéis (D)	Encontra hotéis próximos ao ponto turístico.

A segunda abordagem indica se uma composição é feita manualmente por um especialista ou automaticamente por um sistema de composição. A composição manual é uma técnica consolidada onde os serviços são compostos pela força bruta por um desenvolvedor e pode ser decomposta em duas técnicas distintas de implementação: *top-down* e *bottom-up*. Em *bottom-up*, deve-se identificar os *Web Services* potenciais — que possuem serviços executáveis concretos — e então, conectá-los com uma lógica de processo específica. A segunda técnica, *top-down*, é o oposto, pois começa especificando o processo de negócios (fluxo de trabalho) composto de atividades não executáveis e depois, escolhe o serviço concreto mais adequado para cada atividade. As desvantagens da composição manual são evidentes, já que é um trabalho demorado e, por isso, de custo elevado (SYU et al., 2012).

Por consequência das desvantagens da composição manual, do crescente número de *Web Services* disponíveis para utilização e da complexidade que os serviços compostos podem alcançar, a composição automática de serviços tornou-se muito mais proveitosa, uma vez que oferece baixo custo, economia de tempo, redução de riscos e agilidade. A seção 3.2 explica essa técnica em mais detalhes.

3.2 Composição automática de Web Services

O processo de composição de *Web Services* completo inclui a descrição do problema e o fornecimento dos dados de entrada necessários, ambos entregues pelo usuário,

a execução da melhor combinação de serviços e a apresentação dos resultados ao usuário. Dessa forma, mesmo um sistema de composição completamente automático precisa, antes da composição começar, de uma intervenção humana mínima que é a requisição dessa composição.

A finalidade da composição automática de serviços é criar um serviço composto cuja execução resulta em um objetivo predefinido pelo usuário. Isso quer dizer que este serviço composto precisa ter um significado, que é alcançado levando-se em conta a semântica dos serviços. É este princípio que garante que o serviço composto não seja simplesmente executável, mas sim, que leve ao objetivo expressivo e desejado (BARTALOS; BIELIKOVÁ; HLUCHÝ, 2011).

A composição automática de *Web Services* possui, de acordo com (SYU et al., 2012), duas classes de pesquisa: a central e a transversal. A parte central representa problemas gerais e independentes como classificação, combinação e seleção de serviços. A parte transversal é aquela composta por conceitos que atravessam os que estão presentes na parte central como descrição de serviço e *matchmaking* de serviços. As próximas subseções descrevem esses conceitos.

3.2.1 Descrição de serviços

Como os serviços devem ser vistos como uma caixa-preta, onde não é possível alterar ou estender seu conteúdo, sua descrição é a única forma para entendê-los. A linguagem WSDL, explicada em 2.2.1, é amplamente utilizada para descrever serviços, embora seja capaz apenas de construir interfaces sintáticas para conectar e invocar serviços. Por não ter informação semântica, outras linguagens complementam a WSDL como OWL-S (vista na subseção 2.3.1.1), WSMO[1] (*Web Service Modeling Ontology*) e SAWSDL[2] (*Semantic Annotations for WSDL and XML Schema).*

Os serviços são comumente representados por tuplas. WSDL, em seu nível abstrato, descreve um serviço como uma tupla de entrada e saída (IO - *Input and Output*). Com interface semântica, a tupla poderia se tornar mais rica, contendo entrada, saída, precondições, efeitos (IOPE - *Input, Output, Pre-condition and Effect*), capacidade (C - *Capability*) e propriedades não funcionais (NF - *Non-functional*). IOPEC representa a tupla funcional, ou seja, essas características juntas expressam a funcionalidade do serviço. As propriedades NF representam atributos do serviço, como QoS. Vale ressaltar que as tuplas são utilizadas de acordo com as necessidades de cada projeto, não é necessário utilizá-las sempre juntas.

[1] http://www.w3.org/Submission/WSMO/
[2] http://www.w3.org/TR/sawsdl/

3.2.2 Matchmaking de serviços

O *matchmaking* de serviços está relacionado às similaridades ou compatibilidades funcionais entre tuplas. Dois serviços podem ser conectados entre si se a saída de um puder ser ligada a entrada do outro. Deve-se calcular se os elementos contidos na saída são exatamente iguais ou compatíveis com os elementos da entrada subsequente. Uma entrada do tipo "Comida" é compatível com uma saída contendo "Salada" ou qualquer outro subconceito de "Comida". Contudo, essa relação não é simétrica, pois uma entrada "Salada" é incompatível com uma saída "Comida".

Cada termo da tupla deve possuir descrição semântica e ter seu domínio de ontologia especificado. Atividade esta que leva tempo, uma vez que o desenvolvimento preciso de ontologias não é uma atividade trivial. Do contrário, seria necessário fazer os cálculos através de comparação ineficaz letra por letra das entradas e saídas dos serviços que estivessem sendo analisados.

3.2.3 Classificação de serviços

A classificação de serviços é um conceito importante para a composição, uma vez que auxilia nas etapas de combinação e seleção, descritas nas próximas subseções. A classificação ajuda a melhorar o desempenho e a eficiência do compositor, diminuindo o número de serviços. Já na seleção, para cada atividade abstrata compreendida em um fluxo de trabalho, a classificação agrupa os serviços que possuam a função desejada, mas com diferentes características não funcionais, a partir daí o seletor pode escolher o serviço mais apropriado.

Há dois tipos de classificação, com granularidade grossa ou fina. Para a granularidade grossa, sua unidade é o serviço, e para a granularidade fina, a operação. Em composição automática, a operação é a unidade mais adequada, o que quer dizer que a granularidade fina é melhor para ajudar nesse processo. Cada serviço normalmente oferece várias operações que têm suas funcionalidades independentes. Um serviço que seja exposto por um banco, por exemplo, pode fornecer a taxa de câmbio, operações internas que podem ser acessadas por funcionários, entre outras funções bancárias.

3.2.4 Combinação de serviços

Este conceito é o esqueleto do fluxo de trabalho que atende aos requisitos funcionais solicitados pelo requisitante. Independentemente do algoritmo de composição utilizado, existem dois mecanismos distintos de combinação. No primeiro mecanismo, o "compositor" cria um modelo de fluxo de trabalho não executável e abstrato e encaminha esse modelo à etapa de seleção de serviços. No segundo mecanismo, o próprio compositor sintetiza os componentes de serviço que estejam disponíveis e sejam conhecidos em um

serviço composto executável, o que implica em um fluxo de trabalho. Para comparação, o primeiro mecanismo se assemelha a implementação *top-down* e o segundo é similar a *bottom-up*, ambas explicadas no início deste capítulo (Capítulo 3).

3.2.5 Seleção de serviços

A seleção de serviços lida com a questão de como selecionar, de maneira eficiente, um serviço apropriado para cada atividade, de modo a construir um serviço composto que seja satisfatório. Isto pode ocorrer durante o período de desenvolvimento do serviço composto, durante sua execução, ou em ambos. Existem vários cenários possíveis. Em um desses cenários pode-se selecionar os serviços apropriados em tempo de desenvolvimento e então, "religar" (ligação dinâmica) aqueles que se tornaram inadequados ou falharam durante a execução. Ou ainda, pode-se ter uma ligação dinâmica em tempo de execução (ligação tardia), sem nenhuma seleção durante o desenvolvimento.

3.3 Algoritmos para composição automática

Na literatura, existem diversos algoritmos para a composição automática de serviços. Por ser um tópico muito vasto que possui diversos ramos e conceitos envolvidos, esta seção é dedicada a revisar alguns desses algoritmos, fornecendo uma breve descrição de seu funcionamento. Entretanto, eles não foram implementados ou testados utilizando o sistema desenvolvido nesta dissertação. As subseções seguintes dividem os algoritmos revisados em categorias: os que levam em conta QoS, os bio-inspirados, os de busca heurística e os de busca em largura. Vale ressaltar que alguns artigos que aqui serão citados se encaixam em mais de uma categoria e que em alguns deles, apesar de haver a requisição proveniente de um cliente, os serviços são compostos, mas não executados.

3.3.1 QoS

No artigo (JIANG et al., 2012), os autores levantam a questão a respeito de mudanças que podem acontecer com *Web Services* sendo utilizados em composições: eles podem se tornar indisponíveis temporária ou indefinidamente por problemas na rede ou serem retirados "do ar" por seus desenvolvedores. Visando resolver esse problema, o algoritmo proposto é capaz de substituir os serviços inoperantes por equivalentes ou mesmo reconstruir inteiramente a composição utilizando novos serviços. Este algoritmo tem como primeira etapa a construção de um grafo em que cada vértice é um *Web Service* e uma aresta ligando dois serviços só existe se a saída do primeiro for compatível com uma das entradas do segundo. O grafo possui dois vértices virtuais que não representam serviços, mas sim as entradas (*Start*) e saídas (*End*) da requisição. A segunda etapa é uma busca feita a partir do *Start* para o *End* do grafo (busca para frente) a procura de serviços

que tenham tido todas as suas entradas "casadas" com algum serviço anterior. Depois, é executada uma busca do final para o início para obter um resultado ótimo baseado na busca para frente. O grafo é atualizado sempre que algum serviço muda de alguma forma: se tornando indisponível ou tendo seu valor de QoS alterado, por exemplo. Essa atualização pode acontecer em uma parte do grafo ou acabar afetando este completamente, sendo necessária sua reconstrução. Por último, a solução final de requisição do cliente é um subgrafo dentro desse grafo que forneça o melhor valor de QoS.

O artigo apresentado em (SILVA; MA; ZHANG, 2014) contém uma comparação entre duas técnicas de composição automática de *Web Services* que utilizam o algoritmo de otimização por enxame de partículas (PSO - *Particle Swarm Otimization*), sendo uma baseada em grafo e a outra, em método guloso. O objetivo dos autores foi provar que o algoritmo PSO pode ser utilizado para composição fornecendo exatidão funcional e respeitando valores de QoS. Composições de *Web Service* são normalmente representadas por dígrafos acíclicos, entretanto, a conexão entre serviços deve levar em conta não somente suas entradas e saídas, mas também suas propriedades de qualidade, e neste artigo foram analisados a disponibilidade, a probabilidade de um serviço terminar sua execução dentro do tempo esperado, tempo de resposta e custo de execução. O primeiro passo da técnica baseada em grafo é utilizar um algoritmo para descobrir, no repositório, os serviços que possivelmente façam parte da composição, dadas as entradas e saídas da requisição. Uma vez que todos os serviços candidatos já foram selecionados, pode-se criar o grafo-mestre. Este grafo contém todas as ligações possíveis entre as entradas e saídas desses serviços e suas arestas possuem uma pontuação entre 0 e 1, sendo 1 o melhor. A partir desse grafo, pode-se extrair o fluxo de execução com o uso do algoritmo PSO. Este algoritmo caminha do final do grafo-mestre em direção ao início, escolhendo sempre a aresta que possui pontuação mais alta dentre as disponíveis. Um conceito existente em PSO é que deve existir uma função que avalie o desempenho das partículas (*fitness*) a cada iteração e nesse artigo, os autores utilizaram uma função que considerasse os valores dos atributos de QoS mencionados previamente. As iterações continuam até que se atinja determinado número de gerações ou um valor ideal de *fitness*. O ponto que tiver sido salvo por último como melhor global é a solução ótima. Feitos os testes de comparação, o algoritmo baseado em grafo mostrou-se aquele que forneceu melhores valores de QoS mesmo com um número maior de iterações. A técnica PSO é considerada também como bio-inspirada.

Em (GOLDMAN; NGOKO, 2012), os autores apresentam um algoritmo de redução de grafos que utiliza menos memória e executa menos operações quando comparado a outros já existentes na literatura. Além disso, os autores também fazem uma análise deste algoritmo para previsão de QoS, mais especificamente o tempo de resposta do serviço. Neste trabalho, a composição de serviços é estruturada como um grafo de serviço hierárquico (HSG - *Hierarchical Service Graph*) com três camadas: de operação e processos de negócio, de *Web Services* e de máquina. Para a redução, utiliza-se apenas a

primeira delas. Esta técnica substitui nós, um por vez, de um subgrafo do grafo de operação até restar apenas um nó (que corresponde a uma operação de um *Web Service*). A cada iteração deste algoritmo, o tempo de resposta do serviço (SRT - *Service Response Time*) para este novo subgrafo é calculado. A etapa da redução do grafo deve levar em conta se as operações (nós) pertencem ao mesmo *Web Service*, pois se não houver outra instância desse mesmo serviço em outra máquina, na camada de máquina do HSG, faz-se necessário que uma operação espere a execução de outra(s). Mesmo assim, segundo os autores, este algoritmo mostrou-se satisfatório quando utilizado em composições muito grandes de serviço .

3.3.2 Bio-inspirados

Os autores de (XIAO et al., 2012) utilizaram programação genética como seu método de composição automática de *Web Services*. O objetivo principal deste algoritmo é fornecer ao usuário a composição da forma mais precisa possível, mas supondo que o repositório de serviços contém todos os que sejam necessários para tal. A partir da requisição do usuário, os passos para a composição são basicamente os mesmos da execução de um algoritmo genético para outras finalidades, apenas com alterações no que diz respeito a população inicial, teste de caixa preta e alguns detalhes de implementação. Primeiro, deve-se gerar a população inicial. Trata-se de um grafo interligando os serviços atômicos do repositório de acordo com suas operações, sendo que cada serviço se torna um indivíduo nesse contexto. O segundo passo é a avaliação. Cada uma das composições candidatas (do nó raiz a uma folha do grafo/árvore), é avaliada de acordo com alguns critérios: resultado do teste de caixa-preta, comparação entre suas entradas e saídas e as exigidas pelo usuário e o número de elementos da composição. Como terceiro passo há a reprodução. Nesta etapa, dois indivíduos são escolhidos aleatoriamente como pais e se suas entradas não se sobrepuserem as entradas do usuário, então faz-se mutação nos pais para obter dois filhos. Caso contrário, faz-se o cruzamento dos pais, também, para obter dois filhos. Esse algoritmo permanece em *loop* enquanto não for atingida a condição de parada, que, neste caso, é encontrar uma composição cujo valor de avaliação atinja um valor objetivo especificado ou um número de gerações seja alcançado.

No artigo (XIA; CHEN; MENG, 2008), os autores apresentam um algoritmo baseado no método de otimização por colônia de formigas (ACO - *Ant Colony Algorithm*) para fazer a seleção, em um grafo de composições, de uma composição de *Web Services* que tenha um valor ótimo de QoS. Este artigo também poderia ter sido inserido na seção de QoS. Os algoritmos ACO tradicionais funcionam como uma imitação do processo de busca por alimento realizado pelas formigas, sendo que cada uma deposita ao longo de seu caminho uma substância chamada feromônio. Quanto mais formigas passarem por determinado caminho, mais feromônio ele vai conter e maior é a probabilidade de mais

formigas passarem por ele, ou seja, um caminho com mais feromônios é mais atrativo do que outro com menos. Dados os múltiplos atributos de QoS existentes na composição de *Web Services*, os autores utilizam uma abordagem com múltiplos feromônios, chamada de DACO (*Dynamic Ant Colony Algorithm*), para evitar a estagnação em resultados ótimos locais. Neste trabalho, no grafo de composições de serviços tem-se que cada caminho é uma composição completa, cada aresta é um serviço, cada nó é uma orquestração de serviços e há, ainda, um ponto de início e outro de fim que significam a entrada e a saída da composição, respectivamente. No DACO, as formigas devem ser colocadas no ponto inicial mencionado anteriormente. Estas irão andar pelo grafo simultaneamente de acordo com a fórmula de estratégia de feromônios desenvolvida pelos autores. A medida em que as formigas vão se deslocando no grafo, os valores de QoS de cada serviço vão sendo verificados. Se algum dos atributos não satisfizer os valores predefinidos, este serviço será descartado e a formiga volta para o serviço anterior até conseguir avançar novamente, isto é, até encontrar um próximo serviço que atinja os valores de QoS estipulados para a composição. Quando uma formiga atinge o ponto final, seu processo de seleção de serviços terminou. A cada iteração do algoritmo, o caminho em que mais formigas tiverem passado recebe um aumento em sua quantidade de feromônio. A execução é finalizada quando o número máximo de iterações definido é atingido. O caminho que contiver mais feromônios é aquele que atendeu melhor aos atributos de QoS.

3.3.3 Busca heurística

O artigo (RODRIGUEZ-MIER; MUCIENTES; LAMA, 2011) apresenta a utilização do algoritmo de busca heurística baseado no A* (A-Estrela) em grafos de composição automática de *Web Services*. Os autores relatam que repositórios com muitos serviços podem gerar grafos com um número muito alto de combinações entre tais, além de, provavelmente, existirem algumas combinações redundantes que possam ser eliminadas. Dessa forma, seria necessária a execução do algoritmo A* para encontrar a composição mínima que satisfaça a requisição. A proposta dos autores consiste de três passos. A partir da requisição do serviço, um grafo com os serviços do repositório é criado. Trata-se do primeiro passo. Este grafo contém camadas de serviços de forma que a camada i contenha todos os serviços que podem ser executados com as saídas da camada $i-1$ e assim por diante até a saída desejada pela requisição, sendo que a primeira camada corresponde as entradas da requisição. O segundo passo é eliminar serviços que não estão sendo utilizados ou combinações equivalentes de serviços de forma a reduzir a área de busca. Para tal, foram criadas pelos autores duas técnicas de otimização que são aplicadas ao grafo, resultando em um grafo reduzido. É neste grafo reduzido que o algoritmo A* é aplicado como último passo do trabalho proposto. O algoritmo de busca cruza o grafo de trás para frente, ou seja, das saídas desejadas da requisição para as suas entradas, gerando a composição ótima de serviços que utiliza o menor número desses e atende satisfatoriamente ao cliente.

A proposta dos autores do artigo (WU; KHOURY, 2012) é utilizar o método de planejamento IA (Inteligência Artificial) (*AI planning method*). Este método é um algoritmo de busca heurística para composição de *Web Services* baseado em árvore que é capaz de encontrar caminhos ótimos a partir de um início para um objetivo final. Neste artigo, a busca pelo resultado ótimo deve ainda otimizar atributos de QoS, a partir de valores de limiar fornecidos pelo usuário. Outra entrada que deve ser fornecida ao sistema pelo usuário é o seu objetivo. Providas as entradas, é chamado um método para a criação da árvore inicial. Cada nó é um *Web Service* e cada caminho do nó raiz a uma folha é um caminho de composição. Entretanto, esta árvore se torna muito grande e um processo de busca nesta seria muito complexo e levaria muito tempo. Dessa forma, faz-se necessária uma filtragem na árvore, que ocorre em duas etapas. Primeiro, os ramos que contêm composições ilegais ou que violaram os limiares de QoS fornecidos pelo usuário são podados e não precisam mais ser avaliados. Nestes dois casos, os nós são visitados em Pós-Ordem. A segunda etapa é combinar as soluções encontradas nos dois casos anteriores em uma única nova árvore que representa, então, o conjunto de composições ótimas. Nesta árvore filtrada é realizada uma busca com o algoritmo *best-first* modificado. Este algoritmo, normalmente, explora um grafo ou árvore expandindo nós promissores de acordo com uma função de avaliação específica, mas, neste caso, os autores utilizam uma função de avaliação baseada em distância. Essa função de distância calcula, para todos os caminhos, o somatório da diferença entre os atributos de QoS fornecidos pelos serviços e como entrada para cada aresta. Como saída, tem-se as soluções ótimas classificadas do menor valor de avaliação - que possui a menor distância para o objetivo - para o maior. Como visto, este algoritmo leva em consideração atributos de QoS e poderia estar presente na seção correspondente.

3.3.4 Busca em largura

O método de composição automática de serviços descrito em (SHIAA; FLADMARK; THIELL, 2008) utiliza um algoritmo de busca baseado em grafo para listar todas as possíveis composições e classificá-las de acordo com algumas métricas. Os autores desse artigo também focaram seu trabalho na melhoria das descrições semânticas dos serviços contidos no repositório, pois o processo de composição se inicia com requisições que descrevem a semântica dos serviços desejados. Além disso, a segunda entrada que deve ser fornecida pelo usuário são os critérios de validação que serão utilizados para filtrar os resultados verificando se estes atenderam, por exemplo, a determinados requisitos não funcionais. O algoritmo de busca utilizado é uma versão modificada da busca em largura que é aplicado em um grafo onde os nós são os serviços descobertos no repositório e as arestas representam uma ligação entre dois serviços. Esta ligação indica a conexão entre uma saída do primeiro serviço com uma entrada do segundo. Depois que o grafo geral de composições estiver pronto, um mecanismo de descoberta encontra os serviços correspondentes a requisição do usuário. A partir de então, o algoritmo de busca é iniciado e

conecta os nós uns aos outros de acordo com sua similaridade semântica entre as entradas e saídas dos serviços. Em seguida, estruturas de árvores acíclicas são criadas para cada nó final através da visita de todos os nós e arestas ligados a cada um deles, objetivando atingir os nós iniciais. Com as composições candidatas prontas, inicia-se o processo que as classifica de acordo com a similaridade semântica entre entradas e saídas, número de serviços na composição e propriedades não funcionais. A última etapa é a validação, onde seus resultados indicam se uma composição candidata foi capaz de satisfazer os critérios estabelecidos anteriormente.

3.4 Considerações Finais

Este Capítulo mostrou o processo de composição, diferenciando a parte do usuário e a do sistema de composição. Através do exemplo clássico de uma viagem para ilustrar a composição de serviços, o leitor pôde entender sob a óptica de um problema real a necessidade da composição quando não há nos repositórios apenas um serviço que atenda um usuário. Sobre composição automática, foram explicados conceitos centrais e transversais, além de algoritmos propostos na literatura. Estes algoritmos mostraram o quão vasto é o campo da composição automática de *Web Services*, e cabe salientar que poderiam se tornar uma alternativa ao algoritmo de Paolucci et al. (2002) utilizado nesta dissertação e detalhado na Subseção 4.1.1, certamente fazendo os ajustes necessários.

4 Metodologia

Este Capítulo descreve as ferramentas de *software* utilizadas para a elaboração do trabalho aqui descrito: os sistemas utilizados como referência (Seções 4.1 e 4.3), os algoritmos fundamentais a estes (Seção 4.1.1) e suas características (itens 4.1.1.2 e 4.1.1.1) e o *framework* para publicação de *Web Services* empregado (Seção 4.2). A última seção, 4.4, deste Capítulo será responsável por explicar como todos esses elementos apresentados foram integrados de maneira a funcionar como apenas um único sistema.

4.1 Ferramenta 1: PAACA - Plataforma para Avaliação de Abordagens de Composição Automática

Silva (2012) desenvolveu, em linguagem Java, um sistema capaz de executar diferentes algoritmos de composição de *Web Services* semânticos (explicados na subseção 4.1.1). Este sistema também submete as técnicas a um conjunto de métricas (por exemplo, desempenho, escalabilidade, granulosidade, etc.); consegue gerar *Web Services* semânticos e publicá-los em um repositório; tudo isso através de uma interface gráfica intuitiva. A ferramenta criada possui funcionalidades que possibilitam construir cenários de testes, além de exibir graficamente os fluxos de controles das composições geradas e os resultados das métricas aplicadas, para melhor entendimento do usuário. Contudo, a composição não é executada, ou seja, os *Web Services* que a compõem não são executados.

4.1.1 Algoritmo de composição de Web Services semânticos

Prazeres, Teixeira e Pimentel (2009) propuseram um algoritmo de composição de *Web Services* semânticos (SWS - *Semantic Web Services*) utilizando grafos que consideram a similaridade semântica entre as entradas e saídas dos serviços. Os autores utilizaram *subsumption reasoning* para identificar tais similaridades semânticas, já que esta técnica verifica se um conceito é mais geral do que outro e classifica os conceitos em graus de correspondência (*matching*, ver Subseção 4.1.1.2 para detalhes).

Dado um repositório de SWS com suas descrições em OWL-S, é criado um grafo direcionado. Cada serviço se torna um nó que tem um peso associado ao número de entradas e de saídas que possui, o que significa que quanto maior for esse número, maior será o custo para utilização do serviço. Já as arestas conectam dois nós apenas se houver similaridade semântica entre suas entradas e saídas, por exemplo, o nó A só se conecta com o nó B através de uma aresta se as saídas do primeiro possuírem similaridade com as entradas do segundo como sendo: *exact*, *plug in* ou *subsumes* — conceitos que serão

apresentados em seção posterior (ver Seção 4.1.1.2). Com o grafo já pronto, é possível utilizar algoritmos de caminhamento para criar as composições de serviços.

A partir de uma requisição, o grafo é atualizado com a inserção de novos nós e arestas "virtuais". É criado um nó raiz que se conecta com todos os outros nós que fornecem pelo menos uma entrada pedida na requisição. Depois, é criado um nó extra para cada saída necessária a requisição que são conectados aos serviços que provêem tais saídas. Este trabalho utiliza o algoritmo de Dijkstra[1] para encontrar o menor caminho entre o nó raiz e cada nó extra de saída. O método de composição calcula um menor caminho para cada saída utilizando uma política de descontos (ver Subseção 4.1.1.1), o que pode gerar caminhos que se interceptam, como também caminhos que não se cruzam. Contudo, é preciso gerar um único grafo juntando todos esses subgrafos de maneira que as duplicações sejam eliminadas e as intersecções sejam analisadas. Feito isso, tem-se o grafo de composições requisitado.

4.1.1.1 Política de descontos

A política de descontos apresentada em (PRAZERES; TEIXEIRA; PIMENTEL, 2009) serve para guiar o algoritmo de Dijkstra para encontrar o menor caminho no grafo direcionado. Dijkstra é executado da seguinte forma: o menor caminho é encontrado a partir de serviços que forneçam as entradas até outros serviços que contenham as saídas desejadas pela requisição, e, ainda, evita serviços que não possuam entradas ou as saídas necessárias. Primeiramente, a cada serviço é atribuído um custo inicial de três vezes a soma do número de entradas com o número de saídas que este possui. Quando há uma requisição, descontos são aplicados as entradas que se corresponderem com esta, da seguinte forma:

- Se a relação entre a entrada I_S disponível no serviço e entrada I_R solicitada na requisição for de equivalência, há um desconto de valor 3, para cada entrada que for compatível;
- Se I_R for um super-conceito de I_S, o desconto vale 2;
- Se I_R for um sub-conceito de I_S, o desconto vale 1;
- Se a similaridade falhar, não há desconto (valor 0).

O mesmo vale para qualquer saída O_S disponível no serviço e saída O_R solicitada na requisição e o desconto total de um serviço é a soma entre os descontos obtidos nas entradas e na saídas. Já o custo final de um serviço é a diferença entre seu custo inicial e seu desconto total.

[1] http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf

Essa política de descontos mostra-se útil, segundo Prazeres, Teixeira e Pimentel (2009), pois consegue evidenciar os serviços que possuem mais entradas e saídas que sejam do interesse para determinada composição. Na ferramenta PAACA, um dos algoritmos de composição implementados utilizava a política de desconto e o outro deixava todas as arestas com o mesmo peso.

4.1.1.2 Algoritmo de Matching

O artigo (PAOLUCCI et al., 2002) apresenta o algoritmo de correspondência (*matching algorithm*) que é a base do algoritmo utilizado nesta dissertação. Neste algoritmo, o mecanismo de *matching* usa seu conhecimento semântico sobre os serviços publicados e sobre a requisição para verificar quais serviços do repositório mais correspondem com a requisição do usuário. O algoritmo de *matching* desenvolvido por seus autores possui quatro etapas descritas a seguir:

- As requisições são comparadas com os serviços contidos no repositório e se houver uma correspondência qualquer entre a requisição e o conteúdo do repositório, ela é salva e uma nota é atribuída.

- A correspondência encontrada na etapa anterior é reconhecida se, e somente se, para cada entrada da requisição existir uma entrada equivalente no repositório. O mesmo vale para as saídas. Todavia, o grau de sucesso depende do grau de correspondência encontrado.

- O grau de correspondência entre saídas e entradas está relacionado aos conceitos associados a tais saídas e entradas. Esta terceira etapa calcula esse grau de acordo com quatro níveis - *exact*, *plug in*, *subsumes* e *fail* - que serão explicados em "Níveis de correspondência". O grau *exact* é o melhor deles, seguido de *plug in*, *subsumes* e *fail*, que é o resultado mais indesejado.

- Finalmente, os resultados das correspondências são classificados. O critério de classificação principal é baseado na correspondência com maior pontuação nas saídas, pois é esperado que o usuário as prefira. A correspondência das entradas é apenas utilizada como critério de desempate entre pontuações iguais nas saídas.

Níveis de correspondência

Os níveis de correspondência entre entradas e saídas de serviços utilizados foram, como já citados, *exact*, *plug in*, *subsumes* e *fail*, como em (PAOLUCCI et al., 2002). A Figura 5, que mostra um fragmento de uma ontologia envolvendo veículos, será utilizada para explicar cada um desses níveis.

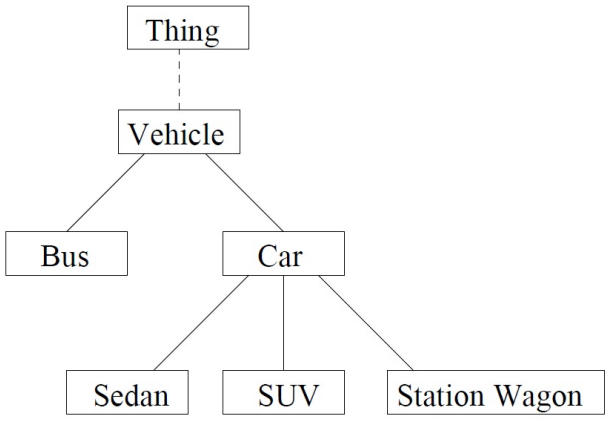


Figura 5 – Fragmento de uma ontologia de veículos, adaptado de (PAOLUCCI et al., 2002)

A relação *exact* acontece quando a saída de um serviço S_1 qualquer e a entrada de um serviço S_2 são equivalentes, ou seja, ambas apresentam o mesmo conceito semântico. Seria como se S_1 tivesse como saída *Car* e S_2 recebesse como entrada também *Car*.

Quando a saída de S_1 fornece uma especialização do conceito existente na entrada de S_2, tem-se como grau de associação *plug in*. Utilizando a ontologia de veículos, S_1 forneceria *Sedan* e S_2 receberia *Car*, o que indica que o primeiro é uma subclasse do segundo e herda todas as suas propriedades.

O grau *subsumes* indica que o provedor não atende completamente ao pedido. Por exemplo, se S_1 entregasse como saída *Car*, mas S_2 recebesse apenas *Sedan*. Como *Sedan* tem um conceito mais específico que *Car*, *Car* pode não atender ao pedido completamente.

Fail acontece quando não há relação semântica entre os serviços. Este seria o caso se S_1 tivesse como saída *SUV* e S_2 esperasse Livro, são conceitos completamente distintos.

4.2 Apache Axis2

O Apache Axis2[2] é um *framework* de código aberto baseado em Java que é utilizado no desenvolvimento de *Web Services*. Possui como uma de suas características principais o fato de ser capaz de realizar o *deploy* de serviços enquanto o sistema está executando, ou seja, não é necessário desligar o servidor para que novos serviços sejam adicionados (JAYASINGHE; AZEEZ, 2011).

Com o Axis2, o desenvolvedor pode executar facilmente as seguintes tarefas: enviar mensagens SOAP e também recebê-las e processá-las; criar um *Web Service* a partir de uma classe Java simples; criar classes de implementação para servidor e cliente usando WSDL; obter o WSDL para um serviço; entre outras. Este *framework* foi utilizado tanto na ferramenta PAACA como no trabalho dessa dissertação.

[2] http://axis.apache.org/axis2/java/core/docs/userguide.html#intro

4.3 Ferramenta 2: AWSCS - Automatic Web Service Composition System

A proposta de Kuehne (2015) foi a de criar um sistema onde fosse possível fazer a avaliação da composição de *Web Services*, baseando-se em atributos de QoS que são dependentes da execução para serem medidos. De acordo com o autor, a maioria dos trabalhos encontrados na literatura relacionados à avaliação de desempenho em composição de *Web Services* não é capaz de avaliar algoritmos para composição automática. Nessa proposta, foi criado um protótipo de um sistema para a composição automática de *Web Services*, sendo possível a publicação de algoritmos para composição automática em um mesmo ambiente de forma que uma comparação entre duas propostas diferentes pudesse ser feita de forma equitativa em termos de execução, uma vez que se tem o mesmo repositório e implementações de serviços disponíveis para ambos. Dessa forma, a melhor abordagem pode ser escolhida. Entretanto, os fluxos de execução foram criados de maneira empírica, pois não havia algoritmos de composição disponíveis. Na Figura 6, pode-se observar o ambiente de experimentos criado para a avaliação do sistema. Cabe informar que esta ferramenta também foi desenvolvida com a linguagem de programação Java.

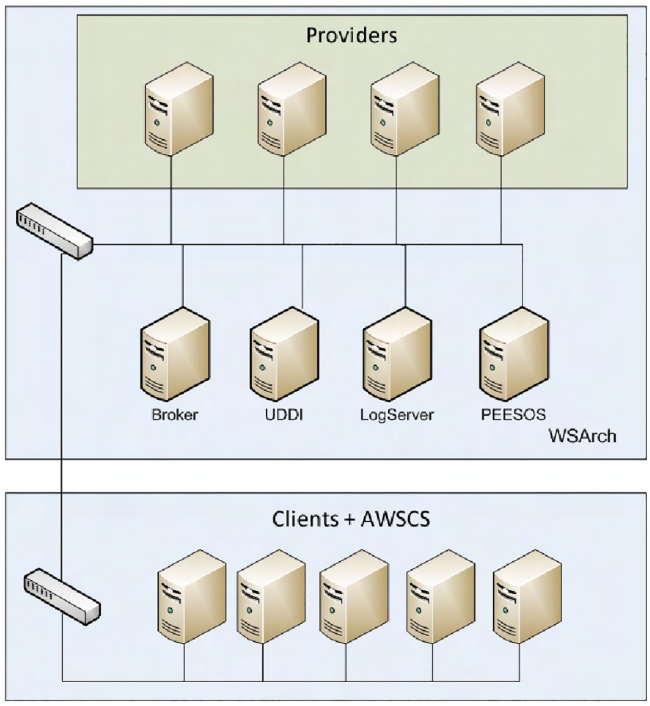


Figura 6 – Ambiente de experimentos do sistema AWSCS, retirado de (KUEHNE, 2015).

4.4 O Sistema de composição proposto: Integração entre PAACA e AWSCS

Dada a ausência de trabalhos que realizassem todo o processo de composição, este possui como propósito executar da requisição do cliente ao retorno das respostas da composição solicitada, conforme já mencionado. Para tal, dois trabalhos foram utilizados como as referências principais: PAACA, "Plataforma para Avaliação de Abordagens de Composição Automática" (SILVA, 2012) e AWSCS, "Automatic Web Service Composition System" (KUEHNE, 2015), explicados previamente nas Seções 4.1 e 4.3, nesta ordem.

Pode-se observar que estes dois trabalhos se complementam, pois no primeiro tem-se um algoritmo de composição realmente implementado e que cria composições a partir de requisições construídas no próprio sistema, e no segundo, há uma outra metodologia que consegue executar serviços dado o fluxo de sua execução. Sendo assim, esses dois sistemas foram integrados de maneira a funcionarem como um só. Para tal, algumas funcionalidades, particularidades de cada um, foram suprimidas.

- Da plataforma PAACA, são utilizadas as funcionalidades de criação de serviços atômicos, de criação de requisições e o algoritmo de composição, descrito em 4.1.1. Corresponde ao bloco *Composition Algorithm* da Figura 7.

- Do sistema AWSCS, apesar deste ser capaz de avaliar abordagens baseando-se em parâmetros de QoS, a funcionalidade utilizada pelo trabalho proposto foi a de execução de fluxos de *Web Services*. Na Figura 7, é o bloco *Web Services location and execution*.

Uma vez que os dois sistemas foram postos em funcionamento separadamente, a junção desses consistia em direcionar a saída do primeiro como entrada para o segundo. Para isso, algumas observações foram obedecidas: o sistema funciona sem a interface gráfica de (SILVA, 2012); não deve gerar automaticamente composições para todas as requisições publicadas no repositório, apenas para a que é solicitada na execução; e é publicado como um *Web Service* para que clientes em uma mesma rede local possam acessá-lo, através de uma requisição. Como essas ferramentas faziam uso de bibliotecas da linguagem Java que não possuem equivalentes em outras linguagens de programação, optou-se por manter o Java como a linguagem de implementação do sistema proposto.

Tem-se então, o sistema proposto. A Figura 7 mostra o diagrama em blocos que será utilizado para auxiliar na explicação. Trata-se de um protótipo que pode ser acessado via rede local por um cliente (*Client*, na Figura). Este envia uma requisição (*Request*) contendo o que espera receber como saída para as entradas fornecidas. Se não houver um único *Web Service* capaz de atender a requisição, o algoritmo de composição encontra

uma combinação de serviços que o faça, ou seja, o sistema a recebe e a partir do que foi "pedido", cria a composição (*Composition Algorithm*) utilizando os serviços disponíveis no repositório (*Repository*). O bloco *Repository* representa onde os serviços estão publicados.

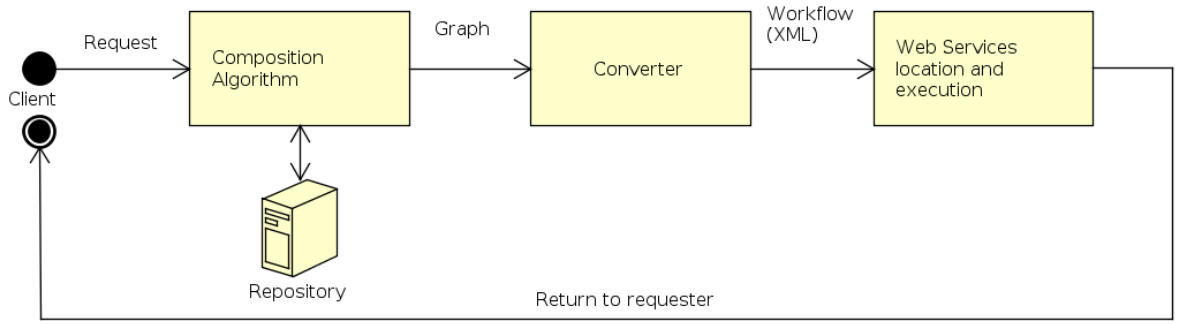


Figura 7 – Diagrama em blocos do sistema proposto.

A composição, neste momento, encontra-se como um grafo (*Graph*, ainda na Figura 7). De acordo com Silva (2012), o vértice denominado "Início" refere-se ao ponto inicial da composição, onde se encontram as representações de todas as entradas; há os vértices que representam serviços; e aqueles que representam as saídas do serviço requerido. As arestas podem interligar os vértices de três formas diferentes: ligando o vértice "Início" àqueles que representam serviços, o que significa que tal ou tais serviços possuem no mínimo uma das entradas requeridas; as que conectam dois serviços indicam que a saída de um tem relação semântica com a entrada do próximo; ou finalmente, uma aresta que liga um serviço a uma saída requerida (que é um outro vértice), indicando que este produz a saída em questão.

O grafo é então convertido, pelo bloco *Converter*, dessa estrutura de dados para uma forma interpretável por AWSCS (KUEHNE, 2015), em um arquivo XML. A conversão acontece levando em conta que nem todo vértice é um serviço do repositório, assim como nem toda aresta é uma ligação semântica entre serviços, e portanto, não precisam ser simbolizados como tais nas *tags* do XML. Essas *tags* representam cada serviço individualmente agrupando suas características como ID ("<ID>", para identificação), nome da operação realizada por ele ("<OPERATION>"), endereço ("<WSDL>", localização do serviço na rede local) e parâmetros de entrada ("<PARAMETER>", podendo haver mais de um) e saída ("<OUTPUT>", caso haja). Ao final, tem-se como saída o *Workflow*, na linguagem XML.

A ordem de execução dos serviços é determinada pela *tag* "<CONNECTION>" que possui duas *tags* aninhadas, "<ID1>" e "<ID2>". ID1 indica o vértice de origem da aresta e ID2 indica em qual vértice esta aresta incide. No exemplo a seguir, tem-se que o serviço cujo ID é 1 e deve ser executado antes dos serviços cujos ID's são 2 e 3 e que estes dois últimos podem ser executados simultaneamente.

```
0 <CONNECTION>
1  <ID1> 1 <\ID1>
2  <ID2> 2 <\ID2>
3 </CONNECTION>
4 <CONNECTION>
5  <ID1> 1 <\ID1>
6  <ID2> 3 <\ID2>
7 </CONNECTION>
```

De posse do arquivo XML completo, o sistema pode encontrar na rede local os serviços a partir de seus endereços e os executar na ordem correta, e em seguida retornar a resposta ao usuário. Todas essas atividades são realizadas pelo bloco *Web Services location and execution*. Os serviços deste repositório estão distribuídos igualmente em diferentes máquinas, assim, com essa disposição, pode-se simular também o fato de que em uma implementação real os serviços estariam localizados em diferentes servidores.

Cabe esclarecer que todas as etapas descritas nos parágrafos anteriores são transparentes ao cliente, este apenas envia a requisição e visualiza a resposta quando todo o processo já terminou.

4.4.1 Arquitetura do Sistema Proposto

O modelo arquitetural representado na Figura 8 ilustra os componentes presentes no sistema. Uma vez que esta ferramenta foi desenvolvida a partir de dois trabalhos ((KUEHNE, 2015) e (SILVA, 2012)), elementos dos diagramas de componentes de ambos e suas descrições estão contidos neste.

- *System Interface*: neste diagrama, representa a parte responsável pela interação com o usuário. É este bloco que faz a comunicação entre o cliente requisitante e o sistema e retorna o que este último concluiu.
- *Automatic Composition Module*: em (KUEHNE, 2015), este era o módulo simulado, pois ainda não havia sido encontrada uma implementação de algoritmo para composição automática. É aqui que estão publicados o *matchmaking* e o algoritmo para seleção de serviços "aproveitados" de (SILVA, 2012). Este bloco todo está encarregado de encontrar *Web Services* candidatos e criar a descrição do fluxo de execução

para atender ao pedido do cliente. A seguir tem-se a definição dos quatro blocos que compõem este módulo.

  - *Repository*: componente responsável por interagir com os serviços do repositório e abstrair para os demais componentes como um serviço é criado, publicado e lido sobre uma linguagem específica.
  - *Matchmaker*: bloco que possui funcionalidades capazes de buscar por similaridades semânticas entre o pedido do cliente (serviço requerido) e os serviços existentes no repositório.
  - *Composer*: neste componente está contida a implementação do algoritmo que compreende a técnica de composição. Os outros componentes desse bloco se relacionam com este provendo-o de serviços.
  - *Evaluator*: este bloco e o bloco anterior são o núcleo do sistema e é ele que cria as instâncias de *Composer* e seus componentes (*Repository* e *Matchmaker*).

- *Composition Execution*: este elemento recebe o fluxo de execução proveniente do *Automatic Composition Module*. O controle da execução dos *Web Services* é feito por este módulo, e assim, ele garante que serviços paralelos e sequenciais serão executados corretamente.

- *XML Schema*: contém as regras para a validação do fluxo XML criado em *Automatic Composition Module*.

- *Monitor*: apesar de não ilustrado na Figura 8, trata-se de um código presente nos componentes *Automatic Composition Module* e *Composite Execution* responsável por fazer a medição dos tempos gastos em cada parte da execução do sistema.

4.4.2 Diagrama de Atividades

Na Figura 9 é apresentado o diagrama de atividades que ajuda no entendimento do sistema proposto nesta dissertação.

Quando o usuário envia sua requisição ao sistema, é o módulo *System Interface* que recebe essa solicitação e não havendo nenhum erro de sintaxe, essa requisição é enviada ao módulo seguinte, *Evaluator*. É esse módulo que vai analisar a requisição e criar os objetos necessários ao funcionamento do sistema. No módulo seguinte, *Repository*, é feita a busca no repositório para encontrar os serviços requeridos. Quando todos são encontrados, o grafo da composição é criado, no módulo *Composition*. Em seguida, na partição *XML Operations*, o arquivo XML responsável por representar o grafo é criado e a correta ordenação dos serviços é feita pelo próximo módulo, *Composition Execution*. De volta ao módulo anterior, o fluxo de trabalho contido no XML é verificado através de um *parse* e,

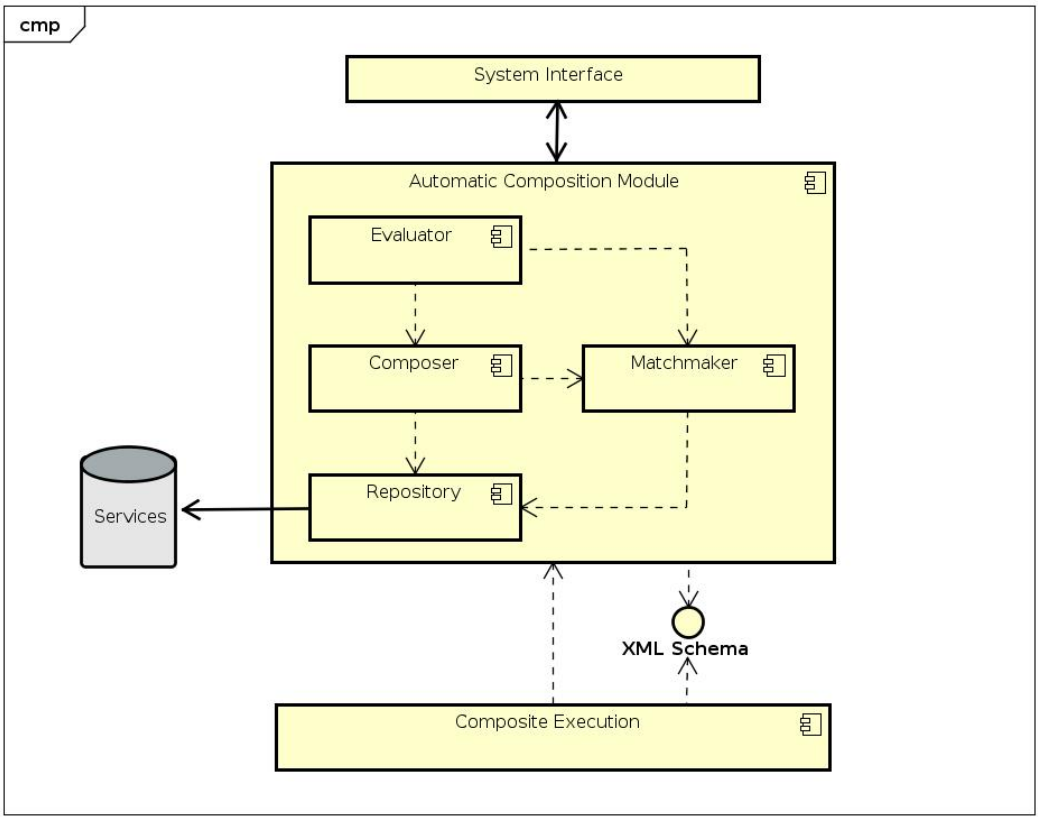


Figura 8 – Modelo arquitetural do sistema proposto.

finalmente, os serviços são executados. Ao final da execução, o módulo *System Interface* retorna a resposta da requisição ao cliente.

No diagrama é possível observar que há quatro pontos de finalização. O primeiro acontece quando há erro de sintaxe na requisição (partição *System Interface*). A segunda finalização inesperada acontece quando, após a busca no repositório, não são encontrados os serviços necessários para atender a requisição (iniciado na partição *Repository* da Figura). Para evitar esse encerramento abrupto, foram criadas requisições onde já se sabia que todos os serviços necessários estariam contidos no repositório. Apesar de esse não ser o comportamento de um usuário comum, visava-se aqui analisar o processo de composição como um todo mesmo que não cobrisse todas as situações possíveis. O terceiro ponto de finalização ocorre quando, no momento da execução do fluxo de trabalho, há a perda de conexão com algum serviço, impedindo seu acesso (iniciado na partição *Composition Execution*). O último ponto de finalização acontece quando há sucesso em toda a execução do sistema e o usuário recebe a resposta de sua requisição (partição *Client*).

4.5 Considerações Finais

O presente Capítulo explicou sobre as ferramentas PAACA e AWSCS ambas de suma importância para o desenvolvimento dessa dissertação. Foram apresentados detalhes

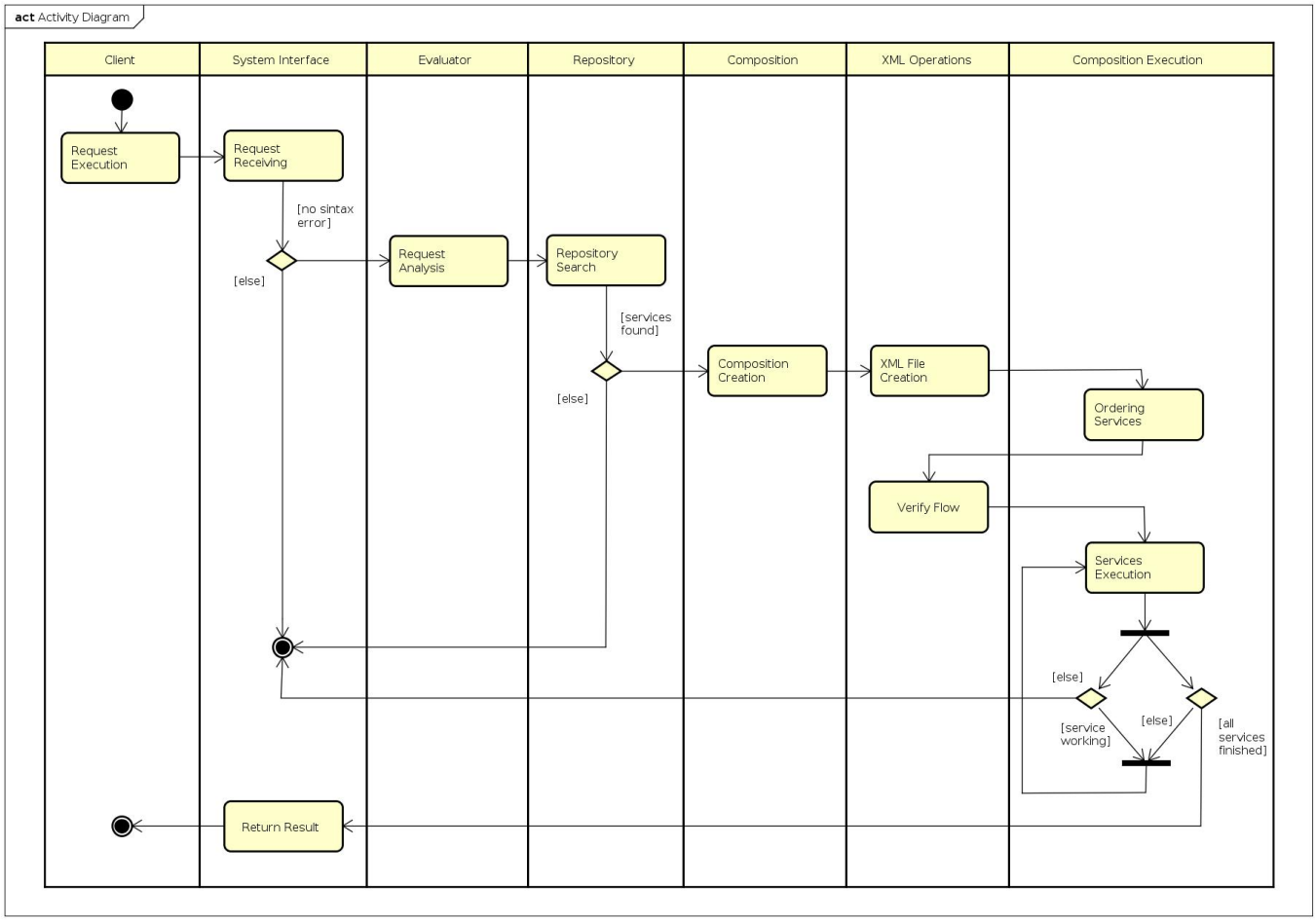


Figura 9 – Diagrama de atividades do sistema proposto.

do algoritmo de *matching* utilizados, assim como a política de descontos empregada por este para guiar o algoritmo de Dijkstra e como funcionam os níveis de correspondência. Todas as características funcionais utilizadas pelo sistema aqui desenvolvido foram explanadas e na Seção 4.4 foi mostrado como a integração entre as duas ferramentas ocorreu e, para tal, utilizou-se um diagrama em blocos. Por fim, foram exibidos a arquitetura e o diagrama de sequência para facilitar o entendimento e formalizar a apresentação do código como um todo.

5 Experimentos e Resultados

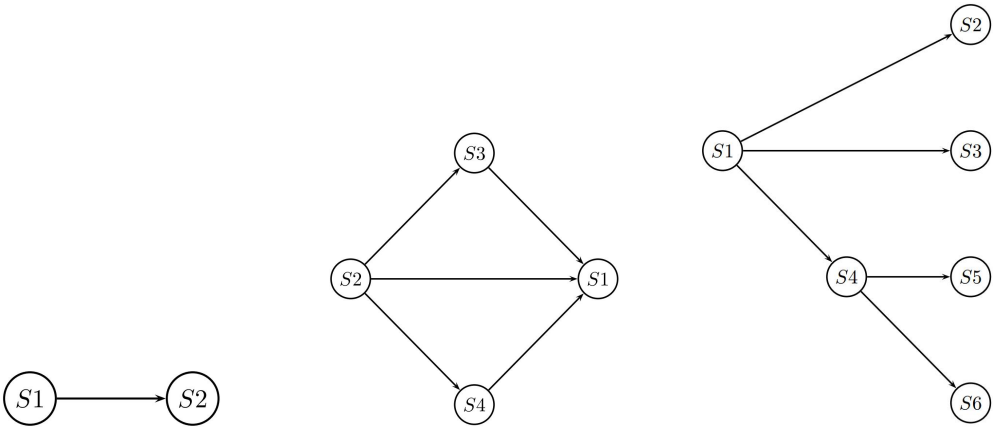
5.1 Experimentos

Para testar o funcionamento do sistema pronto, foram criados previamente duzentos serviços atômicos com a ferramenta "Plataforma para Avaliação de Abordagens de Composição Automática" (PAACA). Esses serviços são criados na linguagem OWL-S e publicados no *framework* Axis2. A partir dos nomes desses serviços, foram criadas classes em Java com um método que representava a operação realizada por esse *Web Service*. Para criar tais classes, utilizou-se um *script* atuando como um *parser* que extraía o conteúdo do arquivo OWL-S que representava o serviço e utilizava esses dados para a geração da classe em Java. Dessa forma, ganhou-se agilidade, uma vez que eram muitos os serviços a serem implementados nesta linguagem e fazer essa extração de conteúdo manualmente e individualmente implicaria em muitas horas de trabalho. Todavia, tratam-se apenas de serviços "ocos", sem funcionalidade definida. Esperava-se assim que apenas o algoritmo de composição fosse avaliado, já que o conteúdo dos serviços seria inexistente. Essas implementações de serviços foram publicadas nas máquinas servidoras.

Além disso, com o auxílio de (SILVA, 2012), três requisições também foram criadas. Essas requisições necessitavam de dois, quatro e seis dos serviços publicados, conforme é possível observar nos grafos ilustrativos na Figura 10.

As requisições serão referidas como $R1$ (Figura 10a), $R2$ (Figura 10b) e $R3$ (Figura 10c) neste texto. Cabe salientar que os serviços em cada uma das requisições eram distintos, mas optou-se aqui por representá-los com nomes similares. Em $R1$, tem-se que o serviço $R1.S1$ deve ser executado antes de $R1.S2$, ou seja, a saída do primeiro alimenta a entrada do segundo, que, por sua vez, fornece a saída da requisição. Para a requisição $R2$, observa-se que $R2.S2$ fornece as entradas de $R2.S3$, $R2.S1$ e $R2.S4$ e que $R2.S1$ é o serviço responsável pela saída final. Já em $R3$, o fluxo de trabalho se inicia com $R3.S1$ e as saídas requeridas são produzidas por $R3.S2$, $R3.S3$, $R3.S5$ e $R3.S6$, sendo que estes dois últimos podem ser executados apenas após $R3.S4$ ter sido finalizado.

O sistema foi implantado em um *cluster* com dez máquinas, como ilustrado na Figura 11. Os duzentos serviços criados foram divididos igualmente em quatro máquinas que atuam como provedores de serviços através do *framework* Axis2. Uma máquina fica responsável por publicar o sistema como um *Web Service*. As outras cinco máquinas restantes representam os clientes que podem enviar requisições ao sistema. Os computadores desse conjunto possuem a configuração descrita na Tabela 2 e estão conectados entre si por um *Switch Gigabit Ethernet*.



(a) R1 (b) R2 (c) R3

Figura 10 – Grafos que representam as requisições dos clientes.

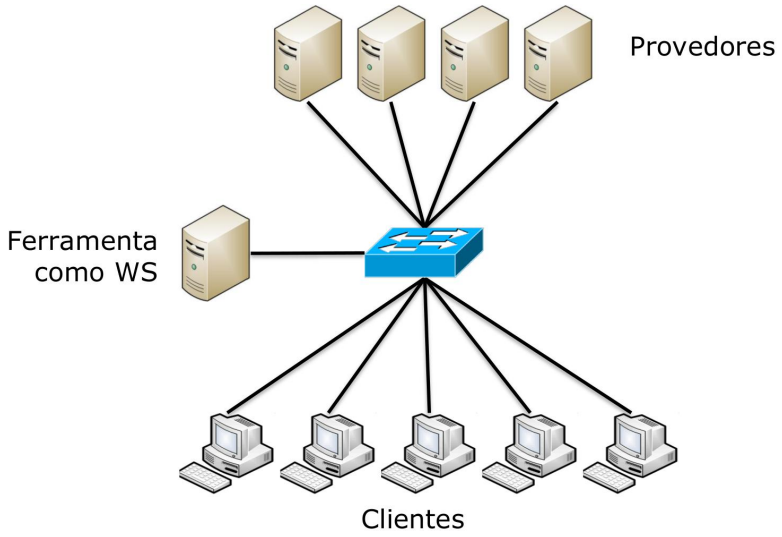


Figura 11 – Ligação ilustrativa das máquinas do *cluster*.

Tabela 2 – Configuração do Cluster

	Cluster
Processor	CPU Intel(R) Core(TM)2 Quad CPU Q8200 @2.33GHz
Memory	6GB
Operating System	Ubuntu Server 14.04.2
Switch	Gigabit Ethernet

5.1.1 Planejamento de Experimentos

Os experimentos foram planejados com 2 fatores e cada fator com três níveis, conforme apresentado na Tabela 3. O fator **Requisições** foi variado em três níveis conforme apresentado na Figura 10. O fator **Workload** representa a quantidade de clientes fazendo requisições simultaneamente no sistema, variando entre 50, 25 e 10. No nível com 50 requisições cada máquina cliente dispara 10 requisições, no nível com 25 requisições cada

máquina cliente dispara 5 requisições e no nível com 10 requisições cada máquina cliente dispara 2 requisições.

Foram realizadas duas baterias de testes em uma sequência de 10 repetições em cada configuração possível, num total de 90 execuções cada. Para calcular o intervalo de confiança, foi escolhido um nível de confiança de 95% em todos os casos.

Tabela 3 – Planejamento de Experimento para as duas baterias de testes

Requisições (Feitas pelos clientes)	Workload (Workload total)	Workload (de cada um dos 5 clientes)
R1	50	10
	25	5
	10	2
R2	50	10
	25	5
	10	2
R3	50	10
	25	5
	10	2

A diferença entre as duas baterias de teste está em como foram feitas as medições de tempo. Na primeira, foi medido o tempo total de execução da ferramenta, ou seja, quanto tempo foi gasto da requisição ao retorno da resposta. A segunda bateria foi realizada para verificar qual parte do sistema estava sendo mais onerosa. Dessa forma, além do tempo total de execução, foram medidos o tempo gasto para a composição, a criação do arquivo XML e a execução dos serviços.

Os resultados dessas medições das duas baterias são apresentados na seção seguinte (5.2).

5.2 Resultados

Conforme já dito na Seção 5.1, cada teste das duas baterias foi repetido dez vezes, contudo, preferiu-se aqui mostrar apenas a média dos valores computados para melhor síntese dos dados. As próximas subseções apresentam os resultados.

5.2.1 1ª Bateria

Para o gráfico exibido na Figura 12, tem-se o tempo médio, em segundos, de execução de cada requisição de um cliente quando 50, 25 ou 10 deles enviam requisições simultaneamente, eixos vertical e horizontal, respectivamente. O tempo de execução foi medido na ferramenta: iniciando quando o cliente enviava uma requisição e parando a

contagem do tempo quando acontecia a execução do último serviço utilizado na composição. Além disso, também são apresentados os limites superior e inferior calculados de acordo com o intervalo de confiança.

Em todos os casos, ao se considerar os intervalos de confiança representados no gráfico, observa-se que, para as três variações de cargas de clientes, há a sobreposição de valores, o que significa que estes acabaram ficando muito próximos uns dos outros.

O número de serviços presentes na composição não afetou diretamente a média dos tempos de execução da ferramenta, pois todos os serviços foram implementados como classes Java vazias. Ao observar os resultados para 10 clientes simultâneos, por exemplo, apesar de $R1$ conter dois serviços e $R3$ seis, o primeiro teve uma média final maior que o segundo, com mais serviços. Isso também foi verdade para 25 clientes simultâneos e pode ter acontecido devido a maneira como os serviços estavam dispostos no grafo, como visto na Figura 10, uma vez que a ordenação dos vértices (serviços) permite a execução paralela de alguns deles, por exemplo, em $R3$, os serviços $S2$, $S3$, $S5$ e $S6$ podem ser executados ao mesmo tempo desde que não haja dependências anteriores.

Nesta bateria de testes, pôde-se notar que quando houve o aumento de 10 para 25 clientes — 2,5 vezes maior — o número médio para o sistema fornecer uma resposta ao usuário subiu de 64s para 152s, valor 2,4 vezes maior. E quando havia 50 requisições simultâneas, — o dobro de requisições do teste anterior — o tempo de resposta era de 294s, ou seja, 1,9 vezes maior. O aumento do tempo entre as cargas de clientes testadas foi quase linear.

A partir dos resultados dessa primeira bateria de testes, viu-se que o sistema funcionava corretamente. Entretanto, era necessário coletar mais informações acerca das diferentes etapas realizadas pelo sistema e isso foi feito na segunda bateria de testes.

5.2.2 2ª Bateria

Para medir os tempos de diferentes atividades realizadas pela ferramenta aqui apresentada e verificar qual delas toma mais tempo durante a execução, foram instalados módulos de medição em partes fundamentais do sistema. Dessa forma, puderam ser medidos o tempo total de execução da requisição ao envio da resposta ao cliente; o tempo gasto para compor os serviços requeridos; o tempo de criação do arquivo XML representando os serviços da composição; e o tempo de execução dos serviços pela ferramenta. Sendo que neste último, também foram medidos os tempos de cada serviço individualmente.

Assim como na 1ª Bateria, o tempo de execução foi medido na ferramenta. No gráfico apresentado na Figura 13, tem-se os valores de tempo médio de execução para as variações do fator tipos de requisições (R1, R2 e R3) e também das cargas de trabalho (50, 25 e 10 clientes). Sendo o tempo exibido no eixo vertical e a quantidade de requisi-

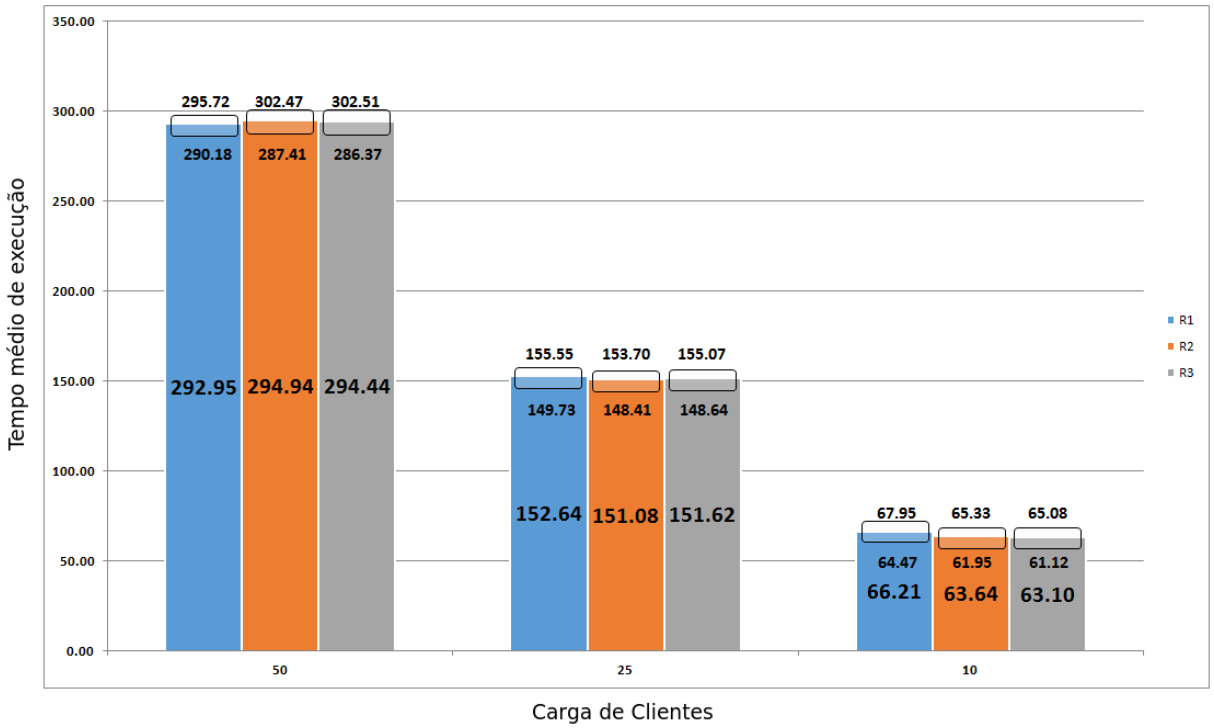


Figura 12 – Resultados da primeira bateria de testes para as requisições $R1$, $R2$ e $R3$, valores em segundos.

ções simultâneas, no eixo horizontal. Neste gráfico também é possível observar os limites superior e inferior do intervalo de confiança.

Os resultados apresentados na Figura 13 mostram uma proximidade muito grande entre os valores no mesmo nível do fator *workload*, mostrando que a variação de tempo de resposta só começa ser possível observar no nível com 50 requisições simultâneas, mas ainda com intervalo de confiança entre (R1 e R3) e (R2 e R3) ainda se sobrepondo tornando assim difícil afirmar que existe uma diferença significativa entre os tempos de resposta.

Nos testes realizados, mesmo com a carga máxima de clientes imposta ao sistema (50 requisições), este ainda consegue produzir respostas com um tempo médio de 292s para cada usuário, 151s para 25 e 64s para 10 (considerando a média aritmética para todos os testes do tempo total medido).

Desses valores, nota-se que há quase um aumento linear entre as diferentes cargas de clientes e o tempo de execução. Quando a quantidade de clientes passa de 10 para 25 - valor 2,5 vezes maior - o tempo médio para cada requisição aumenta 2,3 vezes. Já de 25 para 50 - o dobro de requisições - o tempo médio aumenta em 1,9 vezes.

Na Tabela 4, é apresentada percentualmente qual etapa da composição foi o maior gargalo em relação ao tempo total da execução. É possível observar na coluna "*Composition*" que em todas as configurações, o processo de composição de serviços é responsável pela maior parte da execução, com 99% ou 98%do tempo, aproximadamente.

É importante frisar que os *Web Services* executados não continham uma implementação de fato fazendo com que os provedores pudessem retornar uma resposta imediatamente.

Uma possível técnica para melhorar o desempenho em relação ao tempo de geração das composições de *Web Services* é a utilização de técnicas como *cache* de composição, fazendo assim com que o sistema não tenha que processar por inteiro uma requisição que já tenha sido processada. No entanto, essa técnica poderia gerar a execução de *Web Services* desatualizados, por isso é importante um estudo de como se implantar *caches* que tenham um tempo de expiração adequado.

Tabela 4 – Percentual correspondente as atividades medidas na segunda bateria.

		Activities (%)			
Requests	Client Loads	Composition	XML	Services	Overhead
R1	50	99.754	0.124	0.012	0.127
	25	99.622	0.192	0.009	0.172
	10	99.180	0.262	0.012	0.534
R2	50	99.547	0.286	0.018	0.167
	25	99.531	0.322	0.009	0.135
	10	99.305	0.462	0.006	0.214
R3	50	99.459	0.442	0.021	0.142
	25	99.154	0.578	0.011	0.267
	10	98.468	1.083	0.013	0.412

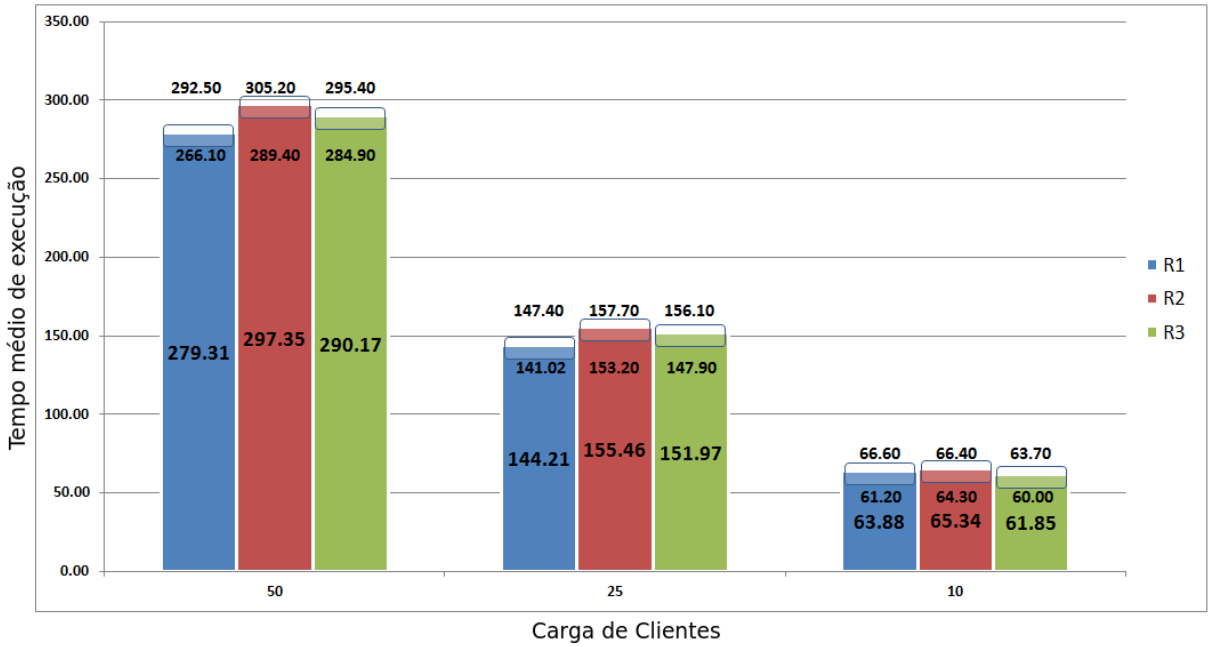


Figura 13 – Resultados da segunda bateria de testes para as requisições $R1$, $R2$ e $R3$, valores em segundos.

O custo da criação do arquivo XML é proporcional ao número de serviços utilizados pela composição. A medida que o número de requisições aumenta, o custo para criação do XML se torna proporcionalmente menor, já que o custo para a criação da composição tende a aumentar.

A coluna "*Services*" mostra a percentagem de tempo gasta na execução dos serviços. Contudo, pode-se verificar que nessa etapa gasta-se um tempo irrisório quando este é comparado com as outras atividades. O tempo médio, em segundos, gastos na execução de cada serviço nas três requisições pode ser encontrado nas Tabelas 5, 6 e 7 (ver Anexos) para as requisições $R1$, $R2$ e $R3$, respectivamente. Os intervalos de confiança mostrados nas tabelas foram calculados para uma confiança de 95%.

A última coluna, "*Overhead*", mostra um tempo que foi calculado e não medido. Trata-se da diferença entre o tempo total utilizado pela ferramenta e o somatório dos tempos medidos e corresponde às trocas de mensagens entre as máquinas envolvidas. Para as três requisições, observa-se que o *overhead* é proporcionalmente maior quando há menos clientes simultâneos.

A média[1] dos tempos mensurados na execução de cada serviço das três requisições são apresentados nas Figuras 14, 15 e 16. Em cada barra também pode-se notar a representação dos intervalos de confiança (ligeiramente fora de escala para facilitar a visualização) para uma confiança de 95%. Nestes gráficos, pode-se observar que um mesmo serviço – qualquer que seja a requisição – leva mais tempo para ser executado quando há um aumento na carga de trabalho.

Na Figura 14, vê-se que, tanto para $R1.S1$ quanto para $R1.S2$, quando o número de clientes vai de 10 para 25, o tempo de execução de cada serviço aumenta em 1,7 vezes e de 25 para 50 clientes, o tempo quase triplica.

Quando observada a Figura 15, nota-se que quando o número de clientes vai de 10 para 25 o tempo de execução de cada serviço triplica para $R2.S2$, $R2.S3$ e $R2.S4$, e para a execução de $R2.S1$, há um aumento de mais de quatro vezes. De 25 para 50 clientes, o tempo de execução aumenta igualmente para $R2.S2$, $R2.S3$ e $R2.S4$ em quatro vezes e em $R2.S1$, o aumento é de quase três vezes.

Pela Figura 16, observa-se que ao aumentar o número de 10 para 25, todos os serviços tem o tempo de execução multiplicado por dois praticamente. Quando o número de clientes parte de 25 para 50, $R3.S3$ e $R3.S4$ têm seu tempo de execução triplicado, já os outros serviços, levam o quádruplo do tempo.

Verifica-se que todos os diferentes serviços presentes numa mesma composição apresentam aproximadamente a mesma duração média quando estão sendo requisitados

[1] Os tempos médios de execução de cada serviço podem ser encontrados nos Anexos deste trabalho. Assim como os valores dos intervalos de confiança calculados para uma confiança de 95%.

pelo mesmo número de clientes, sejam eles 50, 25 ou 10. Já quando os intervalos de confiança são levados em conta, percebe-se uma diferença grande entre os tempos com 50 requisições simultâneas. Uma hipótese para este comportamento pode ser o fato de o sistema estar sob uma carga limite, beirando seu colapso e o impossibilitando de fornecer respostas em tempo hábil, o que fez com que houvesse serviço que respondeu em poucos milissegundos em uma repetição e com um tempo duas vezes maior em outra, por exemplo. Dessa forma, o intervalo de confiança ficou extenso quando o fator **Workload** encontrava-se em 50 para as três variações do fator **Requisições**.

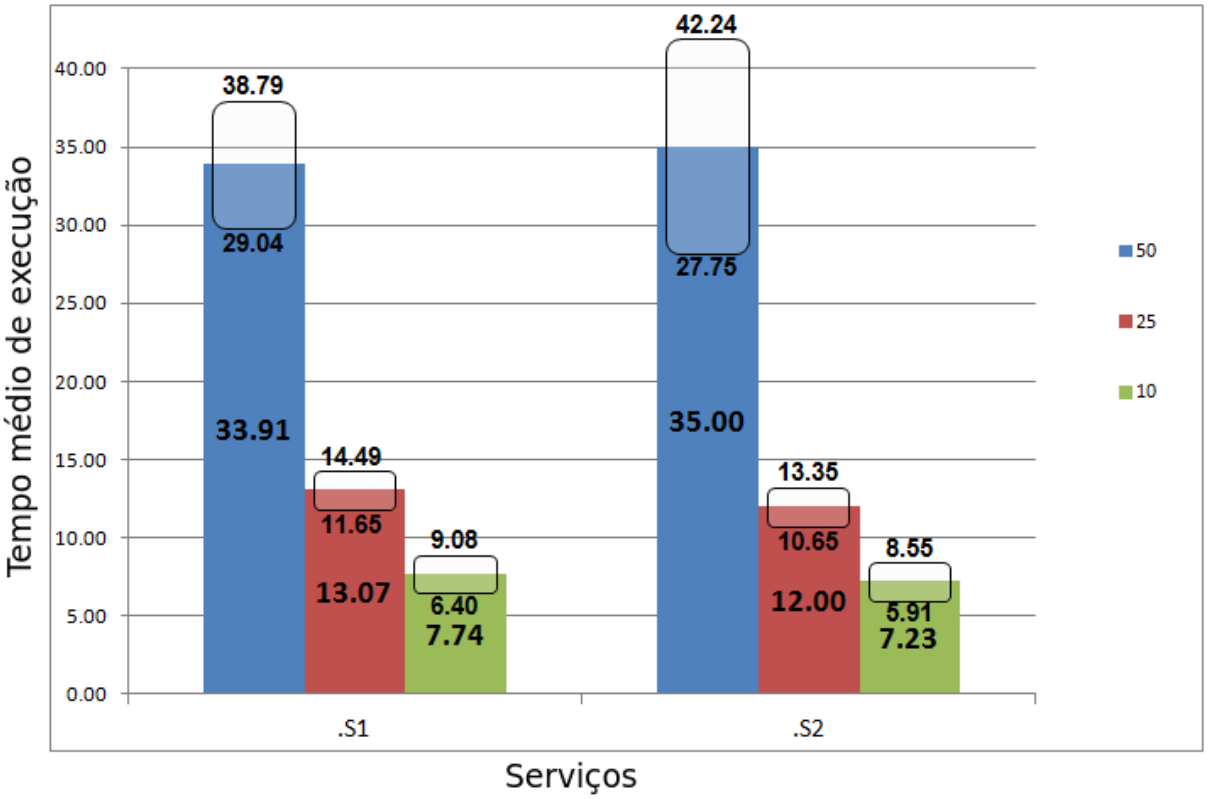


Figura 14 – Médias do tempo de execução de cada serviço utilizado para atender a requisição $R1$ nas três cargas de trabalho, valores em milissegundos.

5.3 Considerações Finais

O Capítulo corrente apresentou como foram planejados os experimentos: a criação dos serviços e requisições pela ferramenta PAACA; a ligação das máquinas do *cluster*, sua configuração e qual a função de cada uma; a elaboração dos diferentes casos de testes realizados e suas variações. Os resultados das execuções dos testes também foram apresentados e analisados em duas baterias. Sendo que a primeira delas media o tempo total do processo e a segunda media o tempo de etapas de execução, além do tempo total. Na segunda bateria também foram mostradas as médias de tempo para a execução de cada serviço nas combinações de requisições e cargas de clientes.

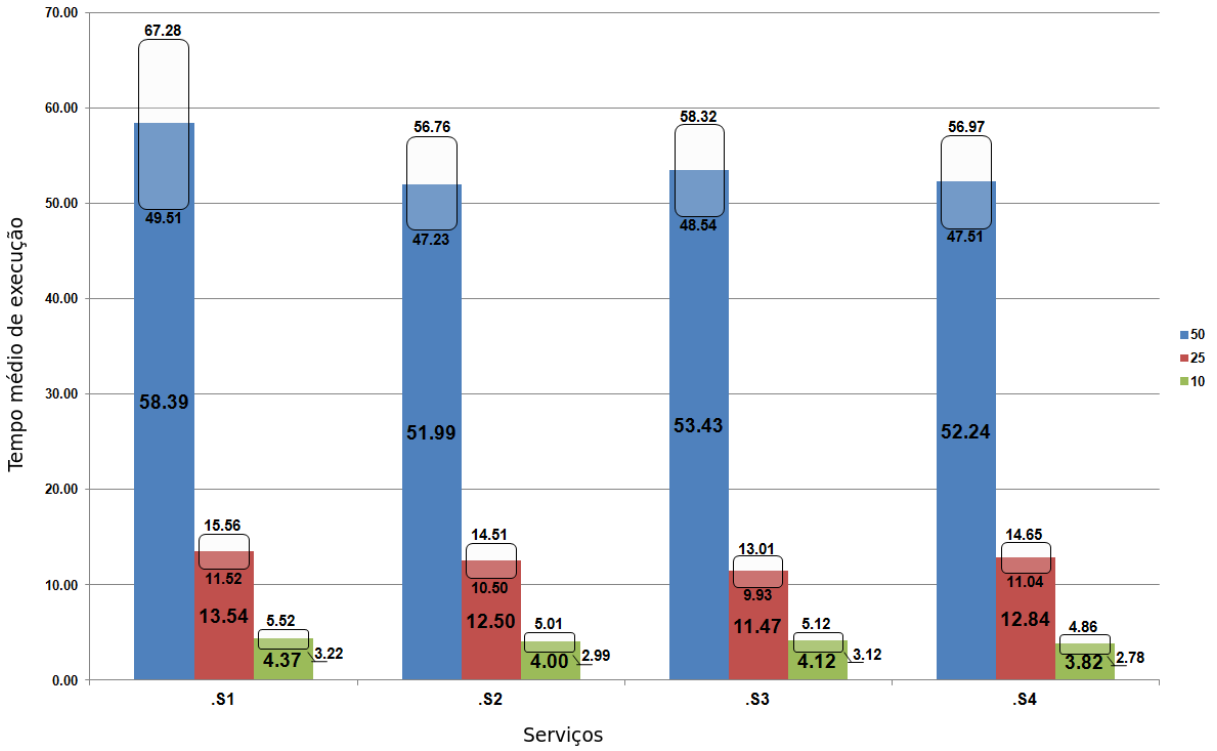


Figura 15 – Médias do tempo de execução de cada serviço utilizado para atender a requisição $R2$ nas três cargas de trabalho, valores em milissegundos.

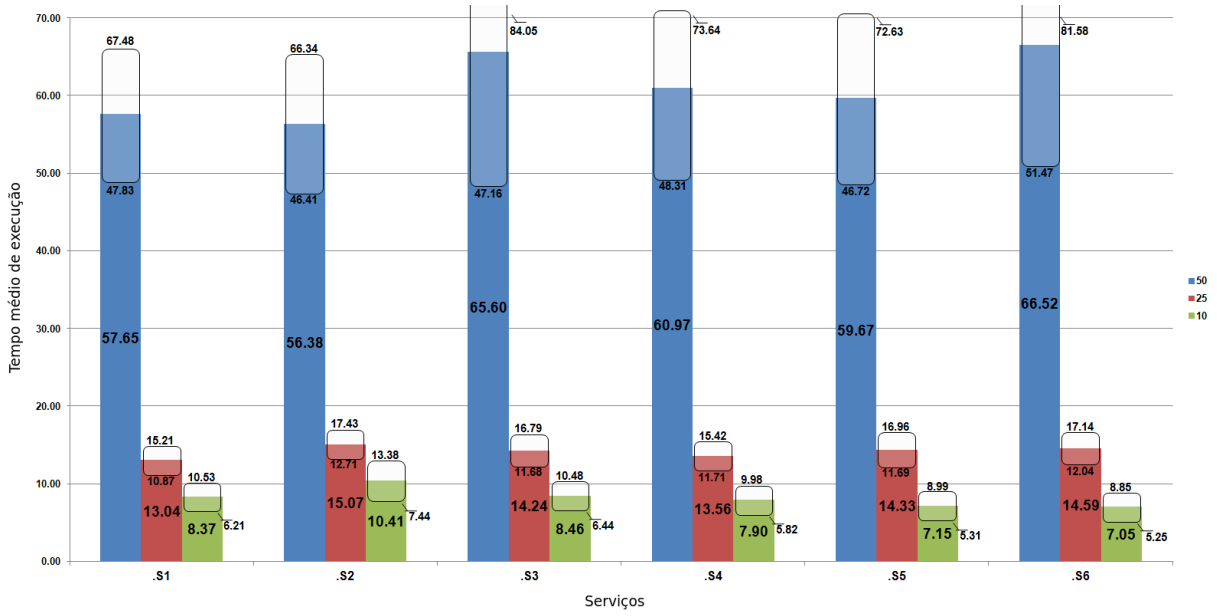


Figura 16 – Médias do tempo de execução de cada serviço utilizado para atender a requisição $R3$ nas três cargas de trabalho, valores em milissegundos.

6 Conclusão

Conforme foi apresentado ao longo dessa dissertação, a área de composição de serviços, e mais especificamente, a área de composição automática de *Web Services*, está em franco desenvolvimento e por possuir várias frentes de pesquisa propicia diferentes ramos de estudo, seja com a criação de novos algoritmos para composição, ou formas de acesso aos repositórios de serviços, ou ainda novas técnicas de descoberta destes.

Entretanto, observou-se nas pesquisas a falta de um sistema completo. Tanto nos trabalhos aqui apresentados como nos que foram pesquisados, mas não entraram na versão final desta, notou-se que estes realizavam sempre parte de todo o processo de composição automática de *Web Services*. Alguns executavam da requisição até a composição, outros a partir do fluxo de execução de serviços a execução destes. Ou ainda, focavam em desenvolver novos algoritmos de composição automática.

O presente trabalho conseguiu, de fato, realizar o que se propôs, pois foi apresentada a avaliação de desempenho de um sistema capaz de fazer uma composição de *Web Services* automática. Isso inclui a requisição enviada por um cliente e a entrega de uma resposta ao seu pedido, o que ainda não havia sido mostrado na literatura.

Para que esse objetivo fosse atingido, houve a junção de duas ferramentas: AWSCS e PAACA. A partir dessa integração, a realização de experimentos que pudessem atestar o funcionamento desse novo sistema elaborado foi possível. Assim sendo, criou-se um roteiro de testes a fim de que se pudesse medir os tempos de resposta total e em diferentes atividades ao longo de todo o processo ao variar a carga de clientes imposta ao sistema.

Pelos resultados apresentados, pôde-se ver claramente que a composição ocupa quase todo o tempo de execução do sistema, visto que é a etapa mais complexa, pois é nela em que o algoritmo de composição é executado e o grafo que representa essa composição é construído. Além disso, o tempo para criação do arquivo XML mostrou-se maior quando mais serviços precisam ser adicionados a ele, já que há mais vértices e arestas para serem analisados e, em seguida, convertidos ou não em *tags* dentro desse arquivo.

O pouco tempo gasto na execução dos serviços se deve à maneira como estes foram criados, representando serviços, mas sem uma funcionalidade implementada e ao contrário do que foi observado no tempo total de execução, o tempo de execução individual dos serviços não aumentou na mesma proporção com o aumento da carga de trabalho em todos os casos. Dos resultados apresentados nos gráficos de medição dos tempos individuais dos serviços, percebe-se que os valores dos intervalos de confiança mantêm-se aproximadamente constantes quando há 10 ou 25 requisições simultâneas ao serviço, qualquer que seja este. Todavia, para 50 requisições, além do tempo de resposta ser muito maior, houve

uma variação considerável dos tempos ao longo das repetições, o que ocasionou em um intervalo de confiança disperso. Mas ainda assim, o tempo de execução dos serviços foi ínfimo, mesmo no pior dos casos.

6.1 Dificuldades ao longo do projeto

Dentre os desafios encontrados para a realização do trabalho, o primeiro deles foi o de trabalhar com códigos vindos de outros pesquisadores. Apesar de a utilização de códigos de terceiros ser prática comum em desenvolvimento de *software*, os dois projetos utilizados como principais referências não tinham como objetivo final serem integrados e funcionarem como apenas um. Também havia o uso de bibliotecas desconhecidas pela autora o que retardou a conclusão de algumas etapas da elaboração do projeto.

Uma outra dificuldade encontrada foi a de usar a coleção OWLS-TC4[1] e suas definições semânticas de *Web Services* e a partir delas criar as implementações dos serviços. Houve uma incompatibilidade com as bibliotecas já empregadas no código legado, e como não havia tempo para solucioná-la, optou-se por utilizar as ontologias criadas na ferramenta PAACA.

Como foi desenvolvido aqui um protótipo que deveria funcionar em rede local, fez-se uso do *cluster* pertencente ao grupo de pesquisas (GPESC — Grupo de Pesquisa em Engenharia de Sistemas e de Computação) da universidade para a realização dos testes. A preparação do ambiente incluiu a instalação de bancos de dados para a coleta de *logs* com os tempos de execução total e de cada atividade. Além disso, foi necessária a distribuição dos arquivos adequados às máquinas de acordo com a sua finalidade. Foi preciso também sincronizar a execução das requisições dos cinco clientes para que os testes pudessem ser realizados de maneira coerente.

6.2 Trabalhos Futuros

Como trabalhos futuros, ficam as propostas de melhorar a interação com o sistema, criar uma *cache* de composições, elaborar implementações reais aos serviços e alteração do algoritmo de composição.

- A primeira proposta facilitaria a utilização por outros pesquisadores do sistema desenvolvido, mas ainda seriam necessárias alterações mais aprofundadas para que esta ferramenta fosse utilizada por um usuário comum;
- Para o segundo caso, não será mais necessário criar uma nova composição quando uma requisição, igual a alguma que já tenha sido recebida pelo sistema, é enviada

[1] http://projects.semwebcentral.org/projects/owls-tc/

a este novamente. Neste caso, deve-se atentar para o tamanho em disco que esta *cache* teria e com qual frequência ela deveria ser atualizada, já que algum serviço pode ser descontinuado ou pode haver problemas na rede que impossibilitem seu acesso, fazendo com que seja preciso gerar a composição outra vez;

- A terceira proposta pode ajudar a medir de maneira mais satisfatória o tempo gasto pela ferramenta na execução de serviços legítimos e que realmente realizem operações ao invés de serem simplesmente "cascas". Uma das soluções seria pesquisar a incompatibilidade com a coleção OWLS-TC4 citada na seção anterior para encontrar a origem do problema, e dependendo do que for obtido, utilizar uma outra biblioteca com funções equivalentes;

- A última proposta é a menos trivial, pois modifica o cerne da composição, mas mesmo assim, é provável que forneça bons resultados (menor tempo de execução) se implementada adequadamente. Para isso, seriam necessários novos estudos acerca tanto dos algoritmos de composição mostrados nas referências como de outros novos. E para saber qual deles seria melhor, suas características deverão ser analisadas, bem como sua equivalência com o que já está implementado.

Referências

BARTALOS, P.; BIELIKOVÁ, M.; HLUCHÝ, L. Automatic dynamic web service composition: A survey and problem formalization. *Comput. and Inf*, p. 793–827, 2011. Citado 6 vezes nas páginas 13, 2, 11, 16, 17 e 19.

BIH, J. Service oriented architecture (soa) a new paradigm to implement dynamic e-business solutions. *Ubiquiti: Information everywhere*, v. 2006, p. 4, agosto 2006. Disponível em: <http://ubiquity.acm.org/article.cfm?id=1159403>. Acesso em: 19 ago. 2015. Citado 2 vezes nas páginas 13 e 7.

CERAMI, E. *Web Services Essentials*. [S.l.]: O'Reilly, 2002. Citado 5 vezes nas páginas 13, 7, 8, 9 e 10.

CHEN, D.-K. Systematic review of applying service oriented architecture in networking. *Sixth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, p. 167–170, outubro 2010. Citado na página 5.

CURBERA, F. et al. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, v. 6, p. 86–93, abril 2002. Citado na página 9.

GOLDMAN, A.; NGOKO, Y. On graph reduction for qos prediction of very large web service compositions. *2012 IEEE Ninth International Conference on Services Computing (SCC)*, junho 2012. Citado na página 22.

JAYASINGHE, D.; AZEEZ, A. *Apache Axis2 Web Services*. Birmingham, Reino Unido: Packt Publishing Ltd., 2011. Citado na página 30.

JIANG, W. et al. Continuous query for qos-aware automatic service composition. *2012 IEEE 19th International Conference on Web Services (ICWS)*, junho 2012. Citado 4 vezes nas páginas 13, 2, 17 e 21.

JOSUTTIS, N. M. *SOA in Practice: The Art of Distributed System Design*. Sebastopol, EUA: O'Reilly, 2007. Citado 2 vezes nas páginas 5 e 10.

KALEPU, S.; KRISHNASWAMY, S.; LOKE, S. W. Verity: a qos metric for selecting web services and providers. *Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops*, p. 131–139, dezembro 2003. Citado 2 vezes nas páginas 13 e 14.

KUEHNE, B. T. *Avaliação de desempenho para seleção de abordagens visando à composição automática de web services em arquiteturas orientadas a serviços e com QoS*. Tese (Doutorado em Ciências de Computação e Matemática Computacional) — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, São Carlos, 2015. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-30062015-150058/>. Acesso em: 22 ago. 2015. Citado 6 vezes nas páginas 13, 7, 31, 32, 33 e 34.

PAOLUCCI, M. et al. Semantic matching of web services capabilities. *First International Semantic Web Conference Sardinia, Italy*, junho 2002. Citado 4 vezes nas páginas 13, 26, 29 e 30.

PRAZERES, C. V. S.; TEIXEIRA, C. A. C.; PIMENTEL, M. d. G. C. Semantic web services discovery and composition: Paths along workflows. *Seventh IEEE European Conference on Web Services (ECOWS)*, novembro 2009. Citado 4 vezes nas páginas 12, 27, 28 e 29.

RODRIGUEZ-MIER, P.; MUCIENTES, M.; LAMA, M. Automatic web service composition with a heuristic-based search algorithm. *2011 IEEE International Conference on Web Services (ICWS)*, julho 2011. Citado 3 vezes nas páginas 15, 16 e 24.

SANTANA, F. d. C. Fundamentos de soa e web services. *Grupo de Estudos de interação do LST - Escola Politécnica da Universidade de São Paulo*, São Paulo, abril 2009. Disponível em: <http://lts-i.pcs.usp.br/xgov/pub/anexos_xgov-/@0036%20SANTANA%20Fundamentos%20de%20SOA%20e%20Webservices>. Acesso em: 19 ago. 2015. Citado 3 vezes nas páginas 6, 7 e 9.

SHIAA, M. M.; FLADMARK, J. O.; THIELL, B. An incremental graph-based approach to automatic service composition. *IEEE International Conference on Services Computing, 2008. (SCC)*, v. 1, julho 2008. Citado na página 25.

SILVA, A. S. d.; MA, H.; ZHANG, M. A graph-based particle swarm optimisation approach to qos-aware web service composition and selection. *2014 IEEE Congress on Evolutionary Computation (CEC)*, julho 2014. Citado na página 22.

SILVA, E. d. C. *Avaliação de abordagens automáticas para composição de serviços web semânticos*. 99 p. Monografia (Mestrado em Sistemas de Computação) — Universidade Salvador – UNIFACS, Salvador, 2012. Citado 6 vezes nas páginas 17, 27, 32, 33, 34 e 39.

SYU, Y.; FANJIANG, Y.-Y. A survey to service composition methods using aspects classification. *2013 IEEE 7th International Symposium on Service Oriented System Engineering (SOSE)*, março 2013. Citado 2 vezes nas páginas 1 e 12.

SYU, Y. et al. A survey on automated service composition methods and related techniques. *2012 IEEE Ninth International Conference on Services Computing (SCC)*, junho 2012. Citado 2 vezes nas páginas 18 e 19.

SYU, Y. et al. A review of the automatic web service composition surveys. *2014 IEEE International Conference on Semantic Computing (ICSC)*, junho 2014. Citado na página 2.

THIES, G.; VOSSEN, G. Web-oriented architectures: On the impact of web 2.0 on service-oriented architectures. *IEEE Asia-Pacific Services Computing Conference*, p. 1075–1082, dezembro 2008. Citado na página 5.

WU, C.-S.; KHOURY, I. Tree-based search algorithm for web service composition in saas. *2012 Ninth International Conference on Information Technology: New Generations (ITNG)*, abril 2012. Citado na página 25.

XIA, Y.-m.; CHEN, J.-l.; MENG, X.-w. On the dynamic ant colony algorithm optimization based on multipheromones. *Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, maio 2008. Citado na página 23.

XIAO, L. et al. Automated web service composition using genetic programming. *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (COMPSACW)*, julho 2012. Citado na página 23.

ZHAO, Y. et al. Towards effectively identifying restful web services. *2014 IEEE International Conference on Web Services (ICWS)*, p. 518–525, junho 2014. Citado na página 10.

ZHENG, Z.; ZHANG, Y.; LYU, M. R. Investigating qos of real-world web services. *IEEE Transactions on Service Computing*, v. 7, p. 32–39, 2014. Citado na página 14.

Anexos

Tabela 5 – Tempo médio de execução dos serviços presentes na requisição R1, valores em milissegundos.

Average time for each service in request R1					
Request	Client Loads	Services	Confidence interval (ms)		
			Inferior limit	Mean	Upper limit
R1	50	*.S1*	29.04	33.91	38.79
		.S2	27.75	35.00	42.24
	25	*.S1*	11.65	13.07	14.49
		.S2	10.65	12.00	13.35
	10	*.S1*	6.40	7.74	9.08
		.S2	5.91	7.23	8.55

Tabela 6 – Tempo médio de execução dos serviços presentes na requisição R2, valores em milissegundos.

Average time for each service in request R2					
Request	Client Loads	Services	Confidence interval (ms)		
			Inferior limit	Mean	Upper limit
R2	50	*.S1*	49.51	58.39	67.28
		.S2	47.23	51.99	56.76
		.S3	48.54	53.43	58.32
		.S4	47.51	52.24	56.97
	25	*.S1*	11.52	13.54	15.56
		.S2	10.50	12.50	14.51
		.S3	9.93	11.47	13.01
		.S4	11.04	12.84	14.65
	10	*.S1*	3.22	4.37	5.52
		.S2	2.99	4.00	5.01
		.S3	3.12	4.12	5.12
		.S4	2.78	3.82	4.86

Tabela 7 – Tempo médio de execução dos serviços presentes na requisição R3, valores em milissegundos.

Average time for each service in request R3					
Request	Client Loads	Services	Confidence interval (ms)		
			Inferior limit	Mean	Upper limit
R3	50	*.S1*	47.83	57.65	67.48
		.S2	46.41	56.38	66.34
		.S3	47.16	65.60	84.05
		.S4	48.31	60.97	73.64
		.S5	46.72	59.67	72.63
		.S6	51.47	66.52	81.58
	25	*.S1*	10.87	13.04	15.21
		.S2	12.71	15.07	17.43
		.S3	11.68	14.24	16.79
		.S4	11.71	13.56	15.42
		.S5	11.69	14.33	16.96
		.S6	12.04	14.59	17.14
	10	*.S1*	6.21	8.37	10.53
		.S2	7.44	10.41	13.38
		.S3	6.44	8.46	10.48
		.S4	5.82	7.90	9.98
		.S5	5.31	7.15	8.99
		.S6	5.25	7.05	8.85