Uma Infra-Estrutura Confiável para Arquiteturas Baseadas em Serviços Web Aplicada à Pesquisa de Biodiversidade

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Eduardo Machado Gonçalves e aprovada pela Banca Examinadora.

Campinas, 24 de novembro de 2009.

Profª. Drª. Cecília Mary Fischer Rubira

Instituto de Computação - UNICAMP
(Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Crisllene Queiroz Custódio – CRB8 / 7966

Gonçalves, Eduardo Machado
G586i Uma infra-estrutura confiável para arquiteturas baseadas em serviços Web aplicada à pesquisa de biodiversidade / Eduardo Machado Gonçalves -- Campinas, [S.P. : s.n.], 2009.

Orientador : Cecília Mary Fischer Rubira
Dissertação (Mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Confiança. 2. Arquitetura orientada a serviços. 3. Tolerância a falha (Computação). 4. Redundância (Engenharia). I. Rubira, Cecilia Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: A dependable infrastructure for service-oriented architectures applied at biodiversity research.

Palavras-chave em inglês (Keywords): 1. Dependability. 2. Service oriented architecture. 3. Computing, Fault-tolerant. 4. Redundancy (Engineering)

Área de concentração: Engenharia de Software

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dr. Cecília Mary Fischer Rubira (IC-Unicamp)
Prof. Dr. Delano Medeiros Beder (EACH-USP)
Profa. Dra. Cláudia Maria Bauzer Medeiros (IC-Unicamp)

Data da defesa: 13/11/2009

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 13 de novembro de 2009, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Delano Medeiros Beder
EACH / USP.

Profª. Drª. Claudia Maria Bauzer Medeiros
IC / UNICAMP.

Profª. Drª. Cecília Mary Fischer Rubira
IC / UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Uma Infra-Estrutura Confiável para Arquiteturas Baseadas em Serviços Web Aplicada à Pesquisa de Biodiversidade

Eduardo Machado Gonçalves

Novembro de 2009

Banca Examinadora:

- Profª. Drª. Cecília Mary Fischer Rubira
  Instituto de Computação - UNICAMP (Orientadora)
- Prof. Dr. Delano Medeiros Beder
  Escola de Artes, Ciências e Humanidades - USP
- Profª. Drª. Claudia M. Bauzer Medeiros
  Instituto de Computação - UNICAMP
- Profª. Drª. Ariadne Maria B. Rizzoni Carvalho
  Instituto de Computação - UNICAMP (Suplente Interno)
- Prof. Dr. Ivan L. M. Ricarte
  Faculdade de Engenharia Elétrica e de Computação - UNICAMP (Suplente Externo)

Resumo

A Arquitetura Orientada a Serviços (SOA) é responsável por mapear os processos de negócios relevantes aos seus serviços correspondentes que, juntos, agregam o valor final ao usuário. Esta arquitetura deve atender aos principais requisitos de dependabilidade, entre eles, alta disponibilidade e alta confiabilidade da solução baseada em serviços. O objetivo deste trabalho é desenvolver uma infra-estrutura de software, chamada de Arquitetura Mediador, que atua na comunicação entre os clientes dos serviços e os próprios serviços Web, a fim de implementar técnicas de tolerância a falhas que façam uso efetivo das redundâncias de serviços disponíveis. A Arquitetura Mediador foi projetada para ser acessível remotamente via serviços Web, de forma que o impacto na sua adoção seja minimizado. A validação da solução proposta foi feita usando aplicações baseadas em serviços Web implementadas no projeto BioCORE. Tal projeto visa apoiar biólogos nas suas atividades de pesquisa e de manutenção do acervo de informações sobre biodiversidade de espécies.

Abstract

The Service-Oriented Architecture is responsible to map the business processes relevant to its services that, together, add value to the final user. This architecture must meet the main dependability requirements, among them, high availability and high reliability, part of the service-based solution. The objective of this work is to develop a software infrastructure, called Arquitetura Mediador, that operates in the communication between the web service's clients and the web services itself, in order to implement fault tolerance techniques that make effective use of available services redundancies. The Arquitetura Mediador infrastructure was designed to be remotely accessible via web services, so that the impact on its adoption should be minimized. The validation of the proposed solution was made using web services-based applications implemented on BioCORE project. This project aims to support biologists in his/her research activities and to maintain informations about collections of species and biodiversity.

Agradecimentos

Eu gostaria de agradecer primeiramente à minha família, cujo apoio foi fundamental para a conclusão deste trabalho árduo porém recompensador.

Um agradecimento especial para todos os amigos e amigas do Instituto de Computação, seja alunos, professores e funcionários. Aos colegas do LSD, Patrick, Amanda, Leonardo, Douglas, e todos os demais, muito grato pelo suporte e muito sucesso.

Aos colegas do LIS, Jaudete, Alan, Bruno e Joana, que contribuíram bastante para que essa pesquisa se realizasse e ganhasse forma. E também aos demais alunos e professores pesquisadores do projeto BioCORE, que igualmente foram importantes para o melhor entendimento da pesquisa.

Não poderia deixar de agradecer a todos os amigos e amigas na qual convivi na DigitalAssets/Sensedia. Com certeza todos vocês tiveram parte nesta caminhada. Estarei para sempre grato pela oportunidade, pelo apoio, enfim, pelo caráter e profissionalismo que me permitiram aprender e aplicar tudo que aprendi no dia a dia de trabalho.

Aos companheiros do Serpro, muito obrigado pela apoio, que igualmente foi fundamental para a conclusão do mestrado.

Aos professores membros da banca examinadora, as suas contribuições foram de grande valia.

E para encerrar, um agradecimento todo especial para a professora Cecília Rubira, que, com sabedoria, incentivos, firmeza e principalmente sinceridade, me orientou formidavelmente em direção ao êxito deste trabalho. Os ensinamentos jamais serão esquecidos. Muito obrigado por acreditar em meu potencial.

"*Inteligência é a habilidade das espécies para viver em harmonia com o meio ambiente.*"
Paul Watson

"*O único meio de criar homens livres é educá-los. Outro modo ainda não se inventou, e com certeza nunca se inventará.*"
Olavo Bilac

Lista de Acrônimos

API Application Programming Interface

HTTP HyperText Transfer Protocol

JAX-WS Java API for XML Web Services

JAXB Java API for XML Binding

JVM Java Virtual Machine

OWL Ontology Web Language

POJO Plain Old Java Object

SOA Arquitetura Orientada a Serviços

SOAP Simple Object Access Protocol

UDDI Universal Description, Discovery and Integration

URL Universal Resource Locator

XML EXtensible Markup Language

W3C The World Wide Web Consortium

WSDL Web Service Description Language

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

E-Science (ou também *eScience*) é definido como *"desenvolvimento sistemático de métodos científicos que explorem o pensamento computacional avançado"* [1]. Estes métodos permitem que pesquisadores acessem informações e recursos computacionais científicos espalhados por grandes áreas (ou pelo mundo todo), contribuindo para a evolução do conhecimento científico. Exemplos de comunidades científicas que se beneficiam desses métodos são as Ciências Humanas e Sociais, Física e Engenharia, Biotecnologia e Ciências Naturais, dentre outros. Esta é uma área com intensa atividade de investigação, e vários projetos são empreendidos para desenvolver e aplicar padrões e ferramentas, como por exemplo, os projetos em andamento em *North-East Regional e-Science Centre* - NEReSC, situado na Universidade de Newcastle, Reino Unido [4].

Aplicações científicas são implementações de sistemas, modelos e processos, freqüentemente especificados em forma de *workflows* científicos e construídos sobre infra-estruturas que permitem o acesso à informação científica distribuída [53]. Infra-estruturas de computação em grade[1], e mais recentemente Arquitetura Orientada a Serviços (SOA)[2], são abordagens que sustentam a construção destas aplicações [47, 37]. *Krishnan et al.* [47] apresenta uma experiência de utilização de SOA voltado para sistemas biomédicos, onde um conjunto de recomendações e desafios no âmbito de SOA como ferramental, interoperabilidade de dados, segurança e tolerância a falhas foram descritas como relevantes para esta e outras aplicações científicas. *Li et al.* [49] estuda um conjunto de serviços Web oferecidos para pesquisas no domínio da Bioinformática, chamado de *Blast*, considerando os requisitos de confiabilidade e acordo em nível de serviço[3]. Como indisponibilidades na

[1]Do inglês, *grid infrastructure* ou *grid computing*.
[2]Do inglês, *Service-Oriented Architecture*.
[3]Do inglês, *Service Level Agreement*(SLA).

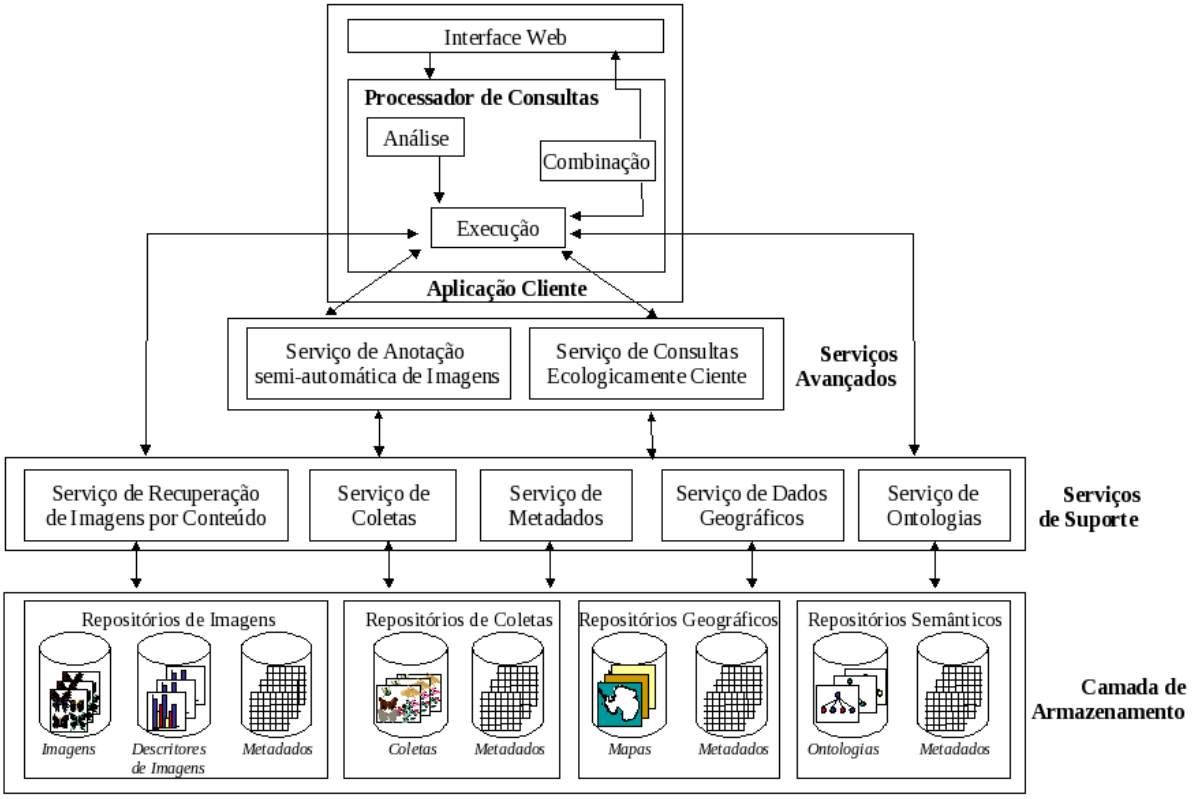


Figura 1.1: A arquitetura do sistema BioCORE.

rede e no servidor, ou também respostas incorretas ou corrompidas podem acontecer, este trabalho investiga o uso de uma ferramenta que coleta e analisa métricas de "dependability". Assim, cientistas podem fazer escolhas de qual serviço usar na execução de um experimento *in silico*, a partir de dados confiáveis disponíveis na Internet.

O projeto BioCORE (*BIOdiversity and Computing Research*)[4], um projeto em parceria entre Instituto de Computação e Instituto de Biologia da UNICAMP, envolve pesquisa de ferramentas, modelos e técnicas para apoio à pesquisa em biodiversidade. O objetivo do projeto é especificar e desenvolver ferramentas computacionais que permitam aos cientistas deste domínio, no caso biólogos, gerenciar e compartilhar seus dados, auxiliando-os na construção de modelos complexos e na análise e modelagem de ecossistemas, incluindo a descoberta de novos relacionamentos e interações entre espécies. Baseadas em serviços Web[5], tais ferramentas permitirão apoiar a cooperação entre pesquisadores em Ciências Biológicas pertencentes a grupos com distintos interesses, vocabulários e visões do mundo.

O projeto BioCORE dá continuidade a três projetos anteriores, dentre eles o projeto

[4]Website: `http://www.lis.ic.unicamp.br/projects/biocore`
[5]Do inglês, *Web services*.

WeBios (*Web Service Multimodal Tools for Biodiversity Research, Assessment and Monitoring*)[6]. O projeto WeBios envolveu o apoio a consultas exploratórias multimodais sobre fontes de dados heterogêneos em respeito ao domínio da biodiversidade. A arquitetura de software do sistema, descrita em Gomes Jr. [43], é composta por três tipos de componentes fundamentais: repositórios de dados distribuídos, interfaces de consultas e um serviço de processamento de consultas. Estes componentes são organizados em três camadas principais, conforme Figura 1.1: Serviços de Armazenamento, Serviços de Suporte e Serviços Avançados. O componente Processador de Consultas, executado no ambiente da Aplicação Cliente, é responsável pela análise, combinação e execução de uma consulta sobre os dados disponibilizados pelos demais serviços da arquitetura. O projeto BioCORE está concentrado nos aspectos de armazenamento - geração e manipulação de conteúdo digital e serviços associados - e serviços de suporte. O serviço de Consultas Ecologicamente Ciente [43] e o serviço de Ontologias [28] já foram especificados, sendo que o segundo já está implementado. Já o componente Processador de Consultas e o Serviço de Coletas estão no escopo do projeto BioCORE para suas implementações.

1.2 Motivação

A Arquitetura Orientada a Serviços (SOA) é responsável por mapear os processos de negócio relevantes para os usuários e aos seus serviços correspondentes que, juntos, agregam valor final ao usuário. Processos de negócio, ou também chamados de *workflows*, são tarefas repetitivas encadeadas em uma seqüência lógica. Em diversos casos, os passos de um processo são realizados por diferentes entidades competentes (filiais, departamentos, grupos de pesquisa, *etc.*). Também, em muitos casos, estas entidades estão fisicamente distribuídas, devido a questões como custo ou subdivisão do trabalho. Processos de negócio também estão em constante evolução. Forças internas e externas à organização produzem requisitos que fazem com que os processos sejam alterados para se adaptarem, concomitantemente as organizações buscam ser mais eficazes no custo e na qualidade dos serviços oferecidos.

Desta forma, SOA tem a preocupação em projetar processos de negócio distribuídos e que estão sujeitos a mudanças freqüentes. Sistemas distribuídos, infra-estruturas e tecnologias que visam garantir a confiabilidade (dependability) na entrega dos serviços contribuem para a distribuição dos passos do processo. Processos de gestão de serviços computacionais, como os processos baseados no guia ITIL (Information Technology Infrastructure Library) [63], já propiciam benefícios ao prover serviços que atendam às metas de qualidade estabelecidas. De forma complementar às práticas do guia ITIL, infra-estruturas ou

[6]Website: `http://www.lis.ic.unicamp.br/projects/webios`

middlewares de mensageria e barramento de serviços são exemplos de soluções alinhadas com este propósito[7]. Os padrões de composição de serviços, considerando empacotamento dos dados e especificação das interfaces (*e.g.* W3C *Web Services* [24]), contribuem para o baixo acoplamento e alta reusabilidade. A orquestração de serviços consiste em uma camada adicional em SOA para desacoplar a lógica de processo (ordem lógica na execução dos serviços, pré e pós-condições, transações, tratamento de exceções, *etc.*) da lógica de composição dos serviços. Tanto padrões de composição quanto padrões de orquestração de serviços propiciam impacto menor e velocidade maior nas mudanças necessárias ao processo de negócio.

A solução de SOA foi adotada no projeto BioCORE por apresentar os requisitos citados. A seguir é mostrada uma lista de requisitos que contém algumas necessidades dos processos de pesquisa em biodiversidade na qual os sistemas existentes e os novos sistemas devem dar apoio [43]:

Usabilidade: A arquitetura deve possibilitar a inclusão de novas interfaces de interação para atender às necessidades do usuário. A arquitetura deve simplificar o processo de publicação e ser tolerante à diversidade tecnológica.

Integração e flexibilidade na especificação dos dados: A arquitetura deve especificar mecanismos que integrem dados heterogêneos. A arquitetura deve ser capaz de lidar com freqüentes mudanças na classificação dos conceitos.

Inclusão de predicados geográficos e ecológicos: A arquitetura deve ser capaz de integrar dados geográficos e ecológicos distintos. A arquitetura deve ser capaz de analisar predicados sobre os mesmos.

Ainda na dissertação de Gomes Jr. [43], padrões de serviços Web, carro-chefe de SOA com relação à interoperabilidade na Internet, e padrões específicos de domínio para representação de dados (*e.g.* ontologias, dados geográficos e biológicos), foram propostos na especificação do WeBios. A tese de Luciano Digiampietri [29] estuda e propõe soluções computacionais para *workflows* científicos aplicados à Bioinformática. Os *workflows* científicos envolvem projetar, reusar, anotar, validar, compartilhar e documentar experimentos em Bioinformática. Apesar daquele trabalho não ter sido diretamente aplicado ao projeto BioCORE, vários processos de negócio em Bioinformática adotaram SOA, através da especificação de *workflows* via composição de serviços Web, descoberta semântica de serviços e gerenciamento da execução dos *workflows* via orquestração de serviços.

Para o sucesso no apoio à pesquisa em biodiversidade, e igualmente fundamental para qualquer negócio crítico que esteja apoiado em uma arquitetura distribuída, a solução

[7] Algumas implementações são apresentadas na seção 2.4.2.

computacional deve seguramente apresentar a confiabilidade requerida. Isto é, a arquitetura de software, juntamente com os componentes de negócio e a infra-estrutura de apoio, devem atender aos principais requisitos de *dependabilidade*[8], entre elas alta disponibilidade, confiabilidade na execução dos serviços e acesso aos dados, segurança no funcionamento e integridade da informação. O termo dependabilidade abriga estes conceitos relacionados com computação segura e confiável, com o objetivo de evitar defeitos em serviços que são mais freqüentes e mais severos que o aceitável [10]. Por exemplo, atualmente há serviços de apoio à pesquisa de biodiversidade no contexto do WeBios já implementados. Um destes serviços, chamado de *Aondê* [28], é um serviço de ontologias e tem previsto integração com outras camadas da arquitetura do projeto BioCORE, como por exemplo, o componente Processador de Consultas. A implementação dos serviços Aondê também se integra com serviços externos para execução dos algoritmos de processamento de ontologias, tais como, *WordNet* [45], ITIS[9] e *Spire*[10]. Em suma, os requisitos funcionais e de interoperabilidade, ao adotar serviços Web, foram tratados no desenvolvimento destes serviços; enquanto que outros requisitos não-funcionais como escalabilidade e desempenho foram abordados como trabalho futuro do Aondê. No entanto requisitos de dependabilidade são igualmente importantes.

1.3 Problema e Solução Proposta

Há décadas os engenheiros de software buscam meios e métodos para alcançar o nível de dependabilidade requerido em sistemas de software. As principais estratégias foram levantadas por Avizienis *et al.* [10]. Nas diversas fases desde o desenvolvimento até a utilização dos sistemas pelo usuário final, diversas ameaças à dependabilidade (e segurança) se tornam concretas, e nem todas podem ser previstas ou solucionadas de forma efetiva. Os conceitos de *falha*[11], *erro*[12] e *defeito*[13] são imprescindíveis para o entendimento e tratamento correto das ameaças à dependabilidade. Falha é um evento que leva a erros. O estado de erro no sistema que não é tratado em um tempo determinado conduzirá à manifestação de defeitos. Defeitos, desta forma, são desvios ou mudanças indesejadas no serviço especificado.

Para seguramente entregar um serviço que possa ser confiável, é preciso entender e contornar diversas falhas que são introduzidas em todas as fases de desenvolvimento, manutenção e infra-estrutura de Tecnologia da Informação e Comunicação (TIC). Com a

[8]Do inglês, *dependability*.
[9]Website: `http://www.itis.gov`
[10]Website: `http://spire.umbc.edu/ont/ethan.php`
[11]Do inglês, *fault*.
[12]Do inglês, *error*.
[13]Do inglês, *failure*.

tendência cada vez maior na adoção de arquiteturas e processos distribuídos (*e.g.* SOA e serviços Web), controlar e manter a dependabilidade destes sistemas se tornam cada vez mais desafiadoras. Os componentes de software que realizam os serviços distribuídos são inerentemente entidades autônomas, isto é, práticas levantadas por Avizienis *et al.* tais como manutenção preventiva e corretiva, testes, tratamento de erros, etc. estão fora do controle de quem usufrui o serviço provido. Em suma, é impossível garantir que serviços ou redes de comunicação individuais se tornem totalmente confiáveis, ou que todas as aplicações-clientes tenham a mesma percepção diante de uma arquitetura que trabalha de forma autônoma e no melhor esforço [18].

Desta forma, as principais soluções para aumento da dependabilidade em SOA estão apoiadas naquilo que os sistemas construídos com este paradigma têm em comum: a comunicação pela Internet e a redundância quase que natural de serviços e enlaces. O atributo de dependabilidade é alcançada através de técnicas de *tolerância a falhas*, que possibilitam ao sistema continuar funcionando, mesmo que de forma limitada, apesar da presença de falhas [9].

O objetivo deste trabalho é desenvolver uma infra-estrutura de software que preencha esta lacuna. A solução proposta é provida na forma de uma infra-estrutura *mediador*, de forma a atuar na comunicação entre as aplicações-clientes e os serviços Web, a fim de executar técnicas de tolerância a falhas que façam uso efetivo da redundância de serviços disponíveis. O nome dado à solução é *Arquitetura Mediador*, que foi projetada para ser acessível remotamente via serviços Web, de forma que o impacto na adoção da infra-estrutura seja minimizada. Desta forma, a infra-estrutura Arquitetura Mediador pode ser integrada à arquiteturas de software no papel de um conector que oferece o requisito de prover aumento da dependabilidade, atuando na requisição entre aplicações-clientes e serviços da arquitetura como um todo.

A tecnologia Java é a solução predominante na implementação da Arquitetura Mediador. O servidor de aplicação *Glassfish* [2] e a biblioteca *Java API for XML Web Services* (JAX-WS) [6] são exemplos de componentes Java que provêem suporte para a implementação da Arquitetura Mediador como uma aplicação Web. As principais funcionalidades da Arquitetura Mediador com relação às técnicas de tolerância a falhas são realizadas por um componente de terceiros chamado *WS-Mediator* [16, 18]. Desenvolvido por Y. Chen na Universidade de Newcastle, Reino Unido, esta infra-estrutura atua sobre a comunicação entre aplicações-clientes e serviços Web, provendo maior dependabilidade através da utilização de serviços redundantes para prover tolerância a falhas. O componente WS-Mediator é capaz de manipular mensagens enviadas a serviços Web de forma independente do domínio da aplicação-cliente, provendo aumento da dependabilidade na requisição destes serviços, dependendo da técnica de tolerância a falhas utilizada. A versão utilizada do WS-Mediator, nesta dissertação, oferece as funcionalidades citadas apenas

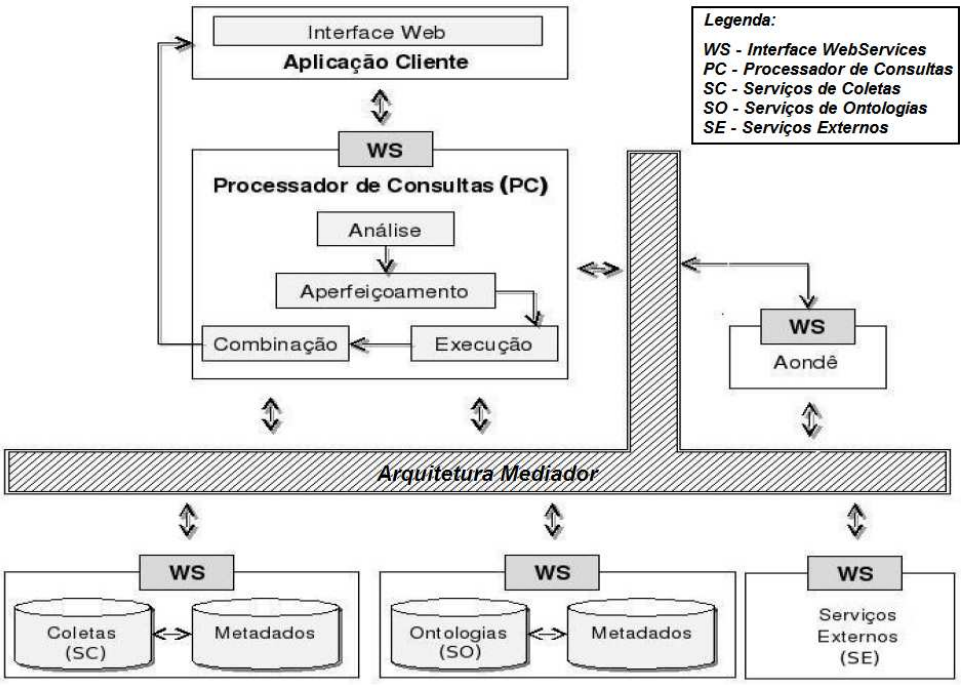


Figura 1.2: A arquitetura do sistema BioCORE integrada com a infra-estrutura Arquitetura Mediador.

como um componente Java *standalone*, isto é, o componente deve acompanhar a distribuição de cada aplicação-cliente ou componente Java que esteja executando em um ambiente independente. Desta forma, a Arquitetura Mediador contribui como um invólucro envolvendo o componente WS-Mediator, tornando suas funcionalidades acessíveis através da Web, de forma a minimizar o impacto de seu uso em aplicações-clientes.

A interface de programação[14] da Arquitetura Mediador permite que a configuração e disparo do processo de mediação sejam efetuados remotamente através de serviços Web. Adicionalmente, devido à adoção de padrões de interoperabilidade de serviços Web, as aplicações-clientes podem ser implementadas em qualquer linguagem de programação. Desta forma, basta para estas aplicações apenas depender de bibliotecas de acesso a serviços Web específica da linguagem escolhida. Operações para configurar a lista dos serviços candidatos (redundantes) da mediação; metadados de dependabilidade dos serviços individuais; políticas de operação, isto é, estratégia de tolerância a falhas a ser adotada na mediação (*e.g.* blocos de recuperação[15] e N-versões[16] [16]); disparo da mediação e obtenção do resultado da requisição do serviço Web.

[14]Do inglês, Application Programming Interface (API).
[15]Do inglês, *recovery blocks*.
[16]Do inglês, *N-version programming*.

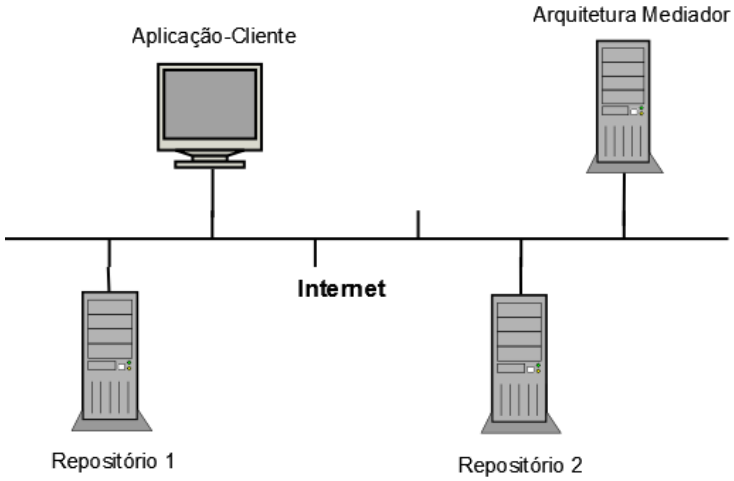


Figura 1.3: Diagrama ilustrando a distribuição dos componentes em um cenário de mediação confiável.

A partir da implementação da Arquitetura Mediador, a validação da solução é feita na forma de provas de conceito em uma arquitetura real no domínio de biodiversidade. As provas de conceito utilizam serviços implementados no contexto do projeto BioCORE. A Figura 1.2 mostra parte da arquitetura especificada do projeto BioCORE. O componente hachureado é a infra-estrutura Arquitetura Mediador implementada neste trabalho, acessível a todos os demais componentes. Os serviços do sistema WeBios são agrupados em: *Processador de consultas* (PC); *Serviço de coletas* (SC), *Serviço de ontologias* (SO) e serviços Web de consulta avançada em ontologias, ambos implementados pelo serviço de ontologias *Aondê* [28]. Os demais serviços que não estão no escopo da prova de conceito estão materializados na Figura 1.2 como Serviços externos (SEs). Aondê e SO já estão implementadas. PC e SC estão em fase de implementação.

A primeira prova de conceito envolve a mediação do acesso ao serviço de ontologias Aondê, acessado através da operação *Consulta* (*Query*). A estratégia de tolerância a falhas empregada nesta primeira prova é a técnica *blocos de recuperação*. Ela visa manter a continuidade do serviço mesmo diante da presença de falhas em um dos componentes, de forma que o componente falho é substituído por um outro componente operacional [16]. Já a segunda prova de conceito envolve novamente a mediação dos serviços Aondê, mas neste caso acessado através da operação *Busca e Ranking* (*SearchRank*). A estratégia de tolerância a falhas empregada nesta segunda prova de conceito é a técnica *N-versões*. A estratégia N-versões visa tolerar falhas de projeto e implementação, de forma que os resultados das requisições de múltiplas versões de um componente possam ser combinados ou comparados para prover um serviço mais confiável [16].

O planejamento e execução dos estudos de caso envolveram os seguintes passos: *(i)*

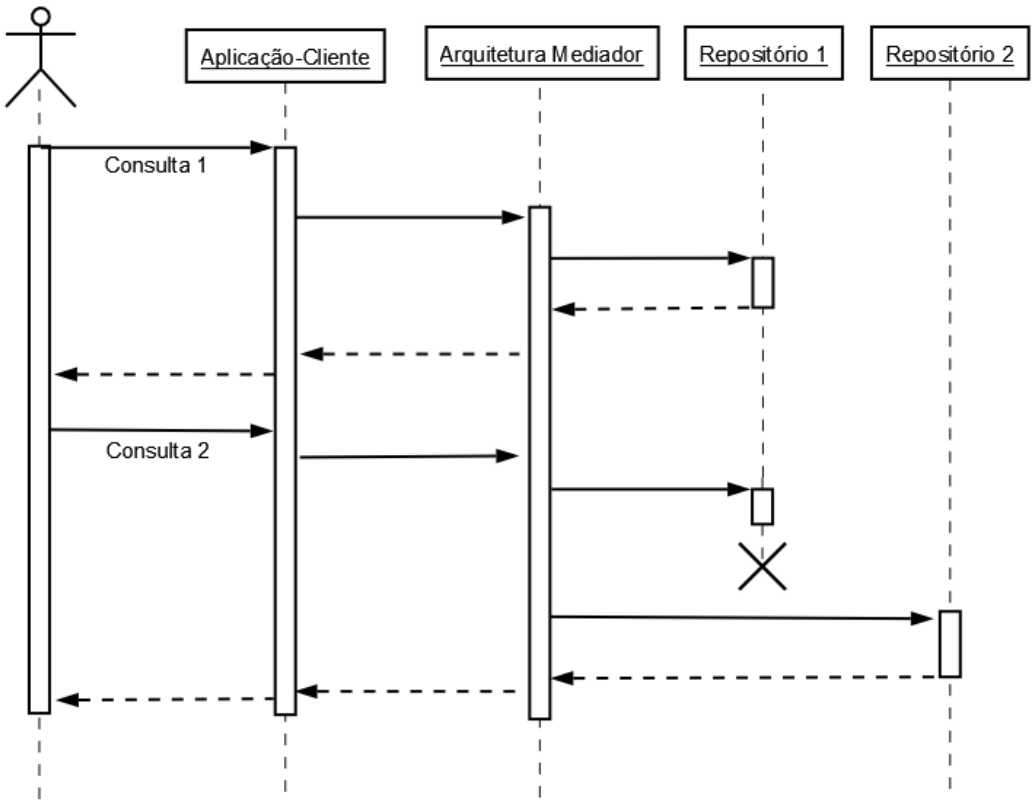


Figura 1.4: Diagrama de seqüência em UML ilustrando um cenário de mediação confiável.

definição dos repositórios populados com diferentes informações de ontologias de biodiversidade; *(ii)* escolha de uma operação disponível na interface dos serviços Aondê para requisição; *(iii)* escolha de uma estratégia de tolerância a falhas a ser executada, e *(iv)* execução da bateria de testes simulando situações de indisponibilidade dos serviços Web. Para extração dos resultados, a metodologia utilizada foi mostrar em um gráfico como as instâncias das aplicações que implementam tais serviços, a reconfiguração dinâmica das requisições ou requisições simultâneas atuaram para tornar os serviços Web mais confiáveis. O cenário descrito a seguir ilustra um cenário hipotético similar à dinâmica da primeira prova de conceito, com o objetivo de caracterizar a metodologia executada nos estudos de caso. Para a segunda prova, é também empregada a mesma metodologia. Este cenário pode ser generalizado para a mediação confiável dos demais componentes do BioCORE.

A Figura 1.3 mostra a distribuição dos componentes envolvidos na prova de conceito hipotética. Estes componentes, interligados pela Internet, são: Aplicação-Cliente, onde o usuário final executa requisições a serviços Web. Arquitetura Mediador, onde fica instalada a infra-estrutura de mediação confiável proposta neste trabalho. Repositório 1 e Repositório 2, onde ficam instalados os serviços Web e dados de interesse do usuário final, e

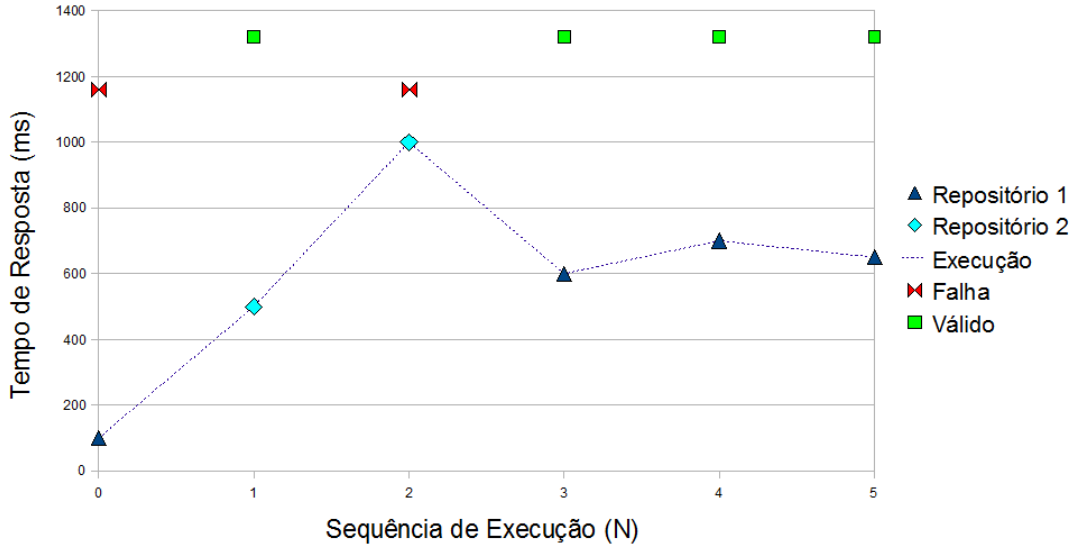


Figura 1.5: Exemplo de gráfico com o resultado de uma mediação confiável.

que atuam como réplicas neste cenário. A Figura 1.4 apresenta um diagrama de seqüência em UML com os componentes da Figura 1.3. Este diagrama ilustra como a Arquitetura Mediador, configurada para executar a estratégia bloco de recuperação, contribui para aumento da dependabilidade. A primeira requisição, chamada de *Consulta 1* e formulada pelo usuário no papel de ator do sistema, é tratada pelo objeto `Aplicação-Cliente`, que a encaminha para uma instância da `Arquitetura Mediador` para ser iniciada a mediação confiável. Esta instância da Arquitetura Mediador, por sua vez encaminha primeiramente a requisição para uma instância do `Repositório 1`, baseado no cálculo de que este serviço, dentre os serviços-candidatos, é o serviço que apresenta maior dependabilidade. A Arquitetura Mediador faz esta determinação de acordo com os metadados de dependabilidade inicialmente informados. A *Consulta 1* então é satisfeita e o resultado é retornado para o usuário. Já em um momento posterior, o usuário constrói outra consulta, chamada de *Consulta 2*, e também a submete ao objeto `Aplicação-Cliente`. Contudo, neste fluxo de execução referente à Consulta 2, o `Repositório 1` acaba por tornar-se indisponível. Através do mecanismo de detecção de erros e reconfiguração automática da Arquitetura Mediador, automaticamente a requisição ao serviço é reconfigurada para o `Repositório 2`, que neste exemplo é capaz de satisfazer a consulta, mascarando o erro.

Os resultados da interação das atividades de requisição, reconfiguração dinâmica diante de falhas, e conseqüente retorno da requisição (com sucesso ou com falha), para cada prova de conceito, são apresentados no formato gráfico similar ao da Figura 1.5. O eixo horizontal mostra as execuções efetuadas, ou bateria de testes, para uma mesma configuração da Arquitetura Mediador. Já o eixo vertical mostra o tempo de resposta

entre o envio da mensagem de uma requisição e a sua respectiva resposta. A legenda à direita mostra os serviços candidatos do cenário, a linha de execução e o resultado final da requisição do serviço (mensagem de falha ou sucesso, para cada execução, disposta na parte superior do gráfico).

Em suma, as principais contribuições desta dissertação são:

- Levantamento dos principais atributos e requisitos de dependabilidade em sistemas de software com ênfase em arquiteturas orientada a serviços Web. Levantamento das principais infra-estruturas cuja mediação confiável de serviços Web aplicadas a arquiteturas orientada a serviços Web tenha sido abordada.
- Estudo e aplicação do componente WS-Mediator[16, 18], de comprovada eficácia no que diz respeito a serviços Web no domínio de e-Science, sendo que a solução pode ser aplicada com sucesso a outros domínios.
- Implementação de uma infra-estrutura de mediação confiável distribuído. Chamada de Arquitetura Mediador, esta implementação utiliza as técnicas de tolerância a falhas oferecidas pelo componente WS-Mediator, porém acessível remotamente mediante serviços Web.
- Validação prática da infra-estrutura Arquitetura Mediador em um projeto de biodiversidade, através de provas de conceito que visam intermediar a comunicação dos componentes de consulta e armazenamento de dados de biodiversidade.

1.4 Organização da Dissertação

O restante desta dissertação está estruturada como segue. O Capítulo 2 discorre sobre a fundamentação teórica e os trabalhos correlatos. O Capítulo 3 descreve o projeto conceitual da solução de mediação confiável para serviços Web, onde são detalhadas a arquitetura de software da Arquitetura Mediador e a interface de programação. O Capítulo 4 descreve o projeto detalhado da Arquitetura Mediador, com detalhes de implementação dos serviços Web e apresentação de um cenário típico de utilização. O Capítulo 5 apresenta duas provas de conceito aplicando a infra-estrutura desenvolvida no âmbito do projeto BioCORE. Este Capítulo descreve os componentes envolvidos em cada prova de conceito, o ambiente de execução, e apresenta os resultado alcançados quantitativamente através de gráficos. Por fim, o Capítulo 6 conclui a dissertação, mostrando um resumo das contribuições e trabalhos futuros.

Capítulo 2

Fundamentação Teórica e Trabalhos Correlatos

2.1 Dependabilidade em Sistemas de Software

O termo dependabilidade[1] engloba vários conceitos relacionados com computação segura e confiável, tais como confiabilidade[2], disponibilidade [3], segurança de funcionamento[4], segurança de confidencialidade[5], integridade[6], manutenibilidade[7], etc [10, 62]. Desta forma, dependabilidade indica a qualidade do serviço fornecido por um dado sistema e a confiança depositada neste serviço fornecido [62]. A relevância do assunto está na perspectiva de que cada vez mais a sociedade "depende" dos sistemas computacionais para as atividades essenciais, que por sua vez os sistemas invariavelmente apresentam defeitos durante o fornecimento dos seus serviços. Este cenário fomenta uma outra definição de dependabilidade, que é a habilidade de evitar defeitos em serviços[8] que são mais freqüentes e mais severos que o aceitável [10].

Avizienis et al. [10] abordam dependabilidade dentro de três tópicos ou taxonomias básicas: os atributos (confiabilidade, segurança, etc.), as ameaças (falhas, erros e defeitos) e meios para obter dependabilidade (prevenção, tolerância, remoção e previsão ou avaliação de falhas). A Tabela 2.1 apresenta uma descrição dos meios para obter dependabilidade de acordo com Sommerville [59]. Desta forma, uma boa especificação e construção

[1]Do inglês, *dependability*.
[2]Do inglês, *reliability*.
[3]Do inglês, *availability*.
[4]Do inglês, *safety*.
[5]Do inglês, *security*.
[6]Do inglês, *integrity*.
[7]Do inglês, *maintainability*.
[8]Do inglês, *service failures*.

Tabela 2.1: Meios para implementar sistemas confiáveis.

Abordagem	**Descrição**
Prevenção de Falhas	O sistema é desenvolvido de modo a evitar a inserção de falhas humanas. Neste caso, o processo de desenvolvimento é organizado para detectar e corrigir falhas, antes da entrega do sistema ao usuário final.
Remoção de Falhas	São utilizadas técnicas de verificação e validação (formais ou não) para detectar e corrigir falhas. Essas técnicas também são executadas antes da entrega do sistema ao usuário final.
Tolerância a Falhas	O sistema é projetado de modo que a presença de falhas durante a sua execução não resulte em defeitos visíveis ao usuário.
Avaliação de Falhas	O sistema é simulado, a fim de identificar os estados impossíveis de serem alcançados. Essa identificação é utilizada para restringir o modelo de falhas do sistema, reduzindo os custos e aumentando a eficiência das técnicas de tolerância a falhas.

de um sistema deve considerar dependabilidade em termos dos atributos, freqüências e severidades aceitáveis de falhas de serviços para classes de falhas em um dado ambiente de uso. Por questões práticas nem todos os termos citados são requeridos para todos os sistemas.

No contexto de sistemas distribuídos, as falhas são classificadas basicamente em três tipos [40]: *(i)* falhas bizantinas, que são arbitrárias e por isso de difícil detecção; *(ii)* falhas de desempenho, que ocorrem quando a resposta do serviço solicitado acontece em um tempo inesperado, normalmente após o limite máximo estipulado; e *(iii)* falhas de omissão, quando a resposta do serviço solicitado não é recebida. As falhas de omissão são também conhecidas como falhas do tipo "falha e pára".

Os sistemas de software que possuem requisitos críticos de confiabilidade e/ou disponibilidade devem, de alguma forma, implementar atividades de *Tolerância a Falhas* [59, 9]. Essa preocupação se baseia no princípio de que não se pode garantir a completude dos modelos especificados, uma vez que mesmo utilizando técnicas de especificação formal, a intervenção humana é imprescindível. Adicionalmente, devido às técnicas de tolerância a falhas se basearem na distribuição e redundância de componentes [9], considerações a respeito de custos financeiros, impacto no desenvolvimento, entre outras considerações, devem ser criteriosamente analisadas.

Existem várias propostas para a divisão das fases de implementação de tolerância a falhas. Uma das propostas mais utilizadas é a classificação em quatro fases de aplicação

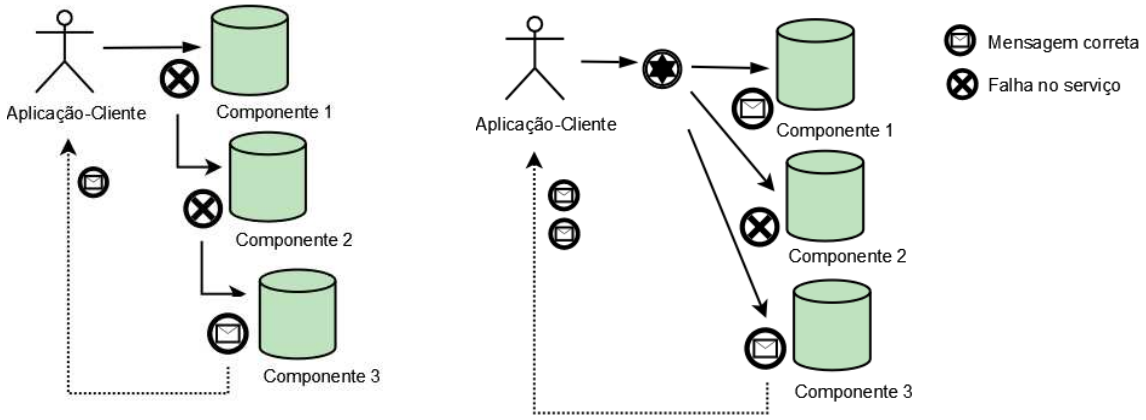
[48, 10]: *(i)* detecção do erro[9], *(ii)* confinamento e avaliação[10], *(iii)* recuperação de erros[11], e *(iv)* tratamento de falhas[12].

Uma falha primeiro se manifesta como um erro, para então ser detectada. A detecção do erro consiste na verificação da consistência do estado do sistema. De forma complementar à detecção, a fase de confinamento e avaliação tem o objetivo de conhecer e isolar as entidades inconsistentes do sistema. Após conhecer os seus componentes errôneos, é necessário recuperar o estado normal do sistema. A recuperação do erro pode ocorrer de duas maneiras: *(i)* recuperação por retrocesso[13], que consiste na restauração de um estado anteriormente válido; e *(ii)* recuperação por avanço[14], que consiste na construção de um novo estado válido a partir das informações contextuais disponíveis. Duas técnicas bastante utilizadas para a recuperação de estados errôneos são a definição de marcos de execução[15] e o tratamento de exceções. Essas técnicas são utilizadas, respectivamente, para implementar técnicas de recuperação por retrocesso e recuperação por avanço.

A última fase sugerida para as atividades de tolerância a falhas é reservada para o tratamento de falhas propriamente dito. O objetivo desta fase é identificar e eliminar as causas dos erros do sistema, de maneira que eles não voltem a acontecer. Uma técnica muito utilizada para essa tarefa é a de reconfiguração arquitetural através do uso de componentes redundantes. Com a disponibilidade de componentes críticos redundantes, é possível, por exemplo, reconfigurar a arquitetura do sistema de forma a evitar o envio de mensagens aos componentes falhos. Esse tipo de correção de erro é conhecido como *mascaramento* da falha[16].

A análise de classes de falhas será, portanto, fundamental para a implementação apropriada desta técnica de tolerância a falhas. A taxonomia que define o padrão de ativação de uma falha está definida em duas classes: *(i)* falhas sólidas[17], e *(ii)* falhas elusivas [18]. Falhas sólidas são falhas reproduzíveis (*i.e.* a partir de um mesmo padrão de estímulo ou dados de entrada). Falhas elusivas são falhas que não são reproduzíveis de forma sistemática (*i.e.* falhas bizantinas ou simplesmente difíceis de reproduzir). Desta forma, a suposição sobre a ocorrência de falhas sólidas ou elusivas determinará o uso de réplicas e múltipla computação, que pode ser seqüencial ou concorrente. No contexto do desenvolvimento de software, o conceito de réplica significa que existe uma outra instância de um

[9] Do inglês, *error detection*.
[10] Do inglês, *damage assessment*.
[11] Do inglês, *error recovery*.
[12] Do inglês, *fault treatment*.
[13] Do inglês, *backward recovery*.
[14] Do inglês, *forward recovery*.
[15] Do inglês, *checkpoints*.
[16] Do inglês, *fault masking*.
[17] Do inglês, *solid / hard faults*.
[18] Do inglês, *soft / elusive faults*.



(a) Blocos de Recuperação (b) N-versões

Figura 2.1: Duas estratégias para aplicação de redundâncias em tolerância a falhas.

mesmo componente. Para falhas elusivas, pode ser adequado uso de réplicas idênticas e que falham de forma independente. Por sua vez para falhas sólidas, pode ser empregado um componente redundante, que além de ser materializado por uma instância distinta, possui estrutura interna igualmente diferente, já que se trata de um outro componente, projetado possivelmente por uma equipe distinta de desenvolvedores (diversidade de projeto[19]).

A aplicabilidade da diversidade de projeto em tolerância a falhas na composição de serviços é bem ilustrada em um exemplo de sistema de agência de viagens. Uma agência de viagens pode, dentre outros serviços, requisitar pela Internet algum serviço de reserva de hotel na cidade de destino. Diversos hotéis operam naquela cidade, disponibilizando sistemas de reservas de quartos que são distintos na sua estrutura de hardware e software. Para que uma reserva na agência de viagens tenha probabilidade maior de sucesso, o seu sistema pode empregar os diversos sistemas de hotel como redundâncias com características de diversidade de projeto, de forma a possibilitar o mascaramento de falhas sólidas que venham a manifestar-se.

Há duas estratégias tradicionais na literatura para aplicação de redundâncias, seja idênticas ou com características de diversidade de projeto, em tolerância a falhas [16, 57, 56]: *blocos de recuperação*[20] e *N-versões*[21]. A estratégia blocos de recuperação visa manter a continuidade do serviço mesmo diante da presença de falhas em um dos componentes, de forma que o componente falho é substituído por um outro componente operacional. A Figura 2.1(a) ilustra o cenário de blocos de recuperação, de forma que a requisição

[19]Do inglês, *design diversity*.
[20]Do inglês, *recovery blocks*.
[21]Do inglês, *N-version programming*.

de uma aplicação-cliente é redirecionada de forma transparente até o terceiro componente operacional, ocasionada por falhas manifestadas pelos dois componentes acessados anteriormente.

A estratégia N-versões visa tolerar falhas de projeto e implementação, de forma que os resultados das requisições de múltiplas versões de um componente possam ser combinadas ou comparadas para prover um serviço mais confiável. A Figura 2.1(b) ilustra o cenário de N-versões, de forma que a requisição de uma aplicação cliente é submetida de forma transparente para três componentes simultaneamente. Apenas os resultados válidos retornados pelo primeiro e terceiro componentes são efetivamente retornados ao usuário, podendo essas mensagens serem combinadas ou comparadas para detecção de mensagens que apresentam desvios na especificação.

2.2 Arquitetura Orientada a Serviços (SOA)

2.2.1 Arquitetura de Software

A arquitetura de software, a partir de um alto nível de abstração, define o sistema em termos de seus componentes arquiteturais, que representam unidades abstratas do sistema; a interação entre essas entidades, que são materializadas explicitamente através dos conectores, e os atributos e funcionalidades de cada um [59]. Por conhecerem o fluxo interativo entre os componentes do sistema, é possível nos conectores, estabelecer protocolos de comunicação e coordenar a execução dos serviços que envolvam mais de um componente do sistema. Essa visão estrutural do sistema em um alto nível de abstração proporciona benefícios importantes, que são imprescindíveis para o desenvolvimento de sistemas de software complexos. Os principais benefícios são: *(i)* a organização do sistema como uma composição de componentes lógicos; *(ii)* a antecipação da definição das estruturas de controle globais; *(iii)* a definição da forma de comunicação e composição dos elementos do projeto; e *(iv)* o auxílio na definição das funcionalidade de cada componente projetado. Além disso, uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional do sistema, que quantifica determinados aspectos do seu comportamento, como confiabilidade, reusabilidade e modificabilidade [19, 59].

Uma determinada propriedade arquitetural pode ser obtida com a utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [58, 54]. Um estilo arquitetural caracteriza uma família de sistemas que são relacionados pelo compartilhamento de suas propriedades estruturais e semânticas. Esses estilos definem inclusive restrições de comunicação entre os componentes do sistema. A maneira como os componentes de um sistema ficam dispostos é conhecida como configuração. As propriedades arquiteturais são derivadas dos requisitos do sistema e

influenciam, direcionam e restringem todas as fases do seu ciclo de vida. Sendo assim, a arquitetura de software é um artefato essencial no processo de desenvolvimento de softwares modernos, sendo útil em todas as suas fases [15].

A importância da arquitetura fica ainda mais clara no contexto do desenvolvimento baseados em componentes. Isso acontece, uma vez que na composição de sistemas, os componentes precisam interagir entre si para oferecer as funcionalidades desejadas. Além disso, devido à diferença de abstração entre a arquitetura e a implementação de um sistema, um processo de desenvolvimento baseado em componentes deve apresentar uma distinção clara entre os conceitos de componente arquitetural, que é abstrato e não é necessariamente instanciável; e componente de implementação, que representa a materialização de uma especificação em alguma tecnologia específica e deve necessariamente ser instanciável. Um exemplo de processo de desenvolvimento baseado em componentes centrado na arquitetura é chamado *UML Components* [14].

2.2.2 Arquitetura Orientada a Serviços (SOA)

As arquiteturas distribuídas baseadas em diversas tecnologias, tais como CORBA, DCOM, EJB e serviços Web pavimentaram os conceitos para a Arquitetura Orientada a Serviços (SOA)[22] [46], sendo que a tecnologia de serviços Web[23] baseado em XML tem se destacado pela sua qualidade em permitir interoperabilidade entre sistemas através de padrões abertos e grande adoção destes padrões por organizações e fornecedores de produtos [8].

Portanto, SOA é definido como um estilo arquitetural onde os blocos básicos são os *serviços*, utilizados na construção de aplicações ou abstração de aplicações, banco de dados e processos de negócio acessíveis pela rede (*e.g.* local, organizacional ou Internet). Os serviços são definidos em termos das mensagens trocadas entre os agentes (pessoas ou sistemas) que requisitam e fornecem o serviço, e metadados de serviços processáveis por máquinas [24].

Sob o ponto de vista de partes interessadas, como analistas de negócio e arquitetos que mantêm as arquiteturas organizacionais, o estilo arquitetural SOA é fortemente divulgado como um paradigma promissor para promover o alinhamento da tecnologia de informação com o negócio. A justificativa desta visão está nas arquiteturas organizacionais atuais, centradas em aplicações monolíticas, que tem como efeito negativo a formação de "ilhas" de automatização e dados de negócio. Este efeito resulta na falta de agilidade dos negócios em responder às mudanças e oportunidades do mercado, já que a informação e as regras de negócio ficam fragmentadas entre as aplicações que foram projetadas para não funcionarem juntas. Por outro lado, a Arquitetura Orientada a Serviços é alicerçada

[22]Do inglês, *Service-Oriented Architecture*.
[23]Do inglês, *Web services*.

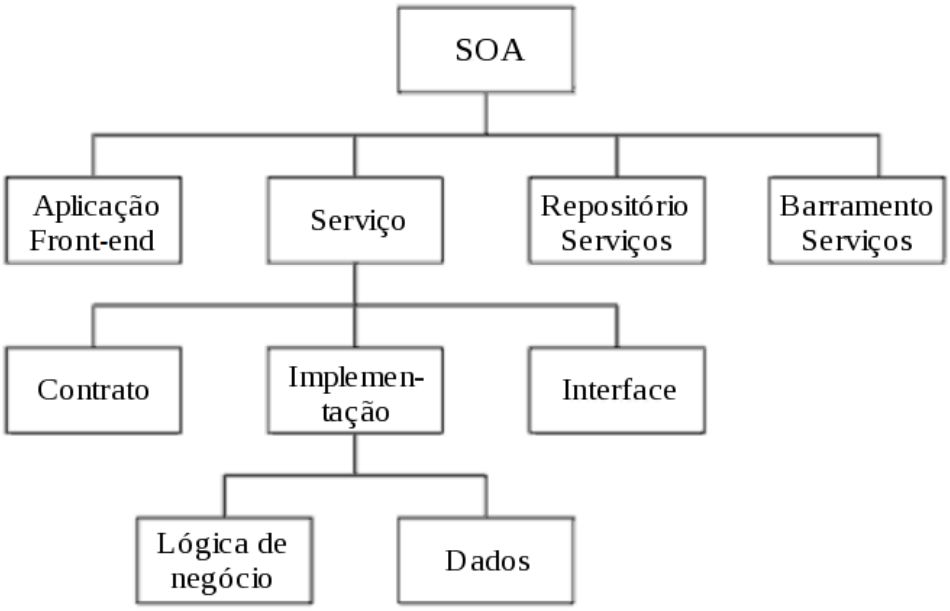


Figura 2.2: Conceitos de SOA.

por duas abstrações sobre software: *serviço de negócio* e *processo de negócio*. Serviço de negócio representa a habilidade de execução de tarefas de negócio. O processo de negócio representa a própria execução do negócio como um todo dentro do conceito de *workflows* e *Business Process Management* - BPM [5], mediante operação e orquestração de serviços [50].

Krafzig et al. [46] propõem uma definição mais concreta de SOA, apresentando um conjunto de conceitos que, juntos, formam a infra-estrutura para construção, implantação e execução de aplicações orientadas a serviços. Estes conceitos são ilustrados na Figura 2.2: serviço, aplicação *front-end*, repositório e barramento. Nesta arquitetura, os serviços constituem processos de negócio (*e.g. obtenhaReserva* ou *canceleReserva*) e encapsulam a implementação da operação (lógica de negócio e dados) via interface (grupo de operações disponíveis) e contrato (especificação do propósito, funcionalidade, restrições e meios de uso dos serviços). Já a aplicação *front-end* materializa os processos de negócios citados por *Lublinsky* [50], que nesta arquitetura são as entidades que entregam o valor de SOA para os usuários finais. Já o repositório de serviços está relacionado com a descoberta e aquisição de informações sobre serviços tanto em tempo de desenvolvimento quanto em tempo de execução (e.g. registro UDDI). Um consulta a este repositório permite que projetos localizem funcionalidades criadas em projetos anteriores e as reutilizem em forma de serviços. Por fim, o barramento de serviços está relacionado com a conexão e mediação entre todos os participantes de SOA (serviços, aplicações *front-end* e infra-

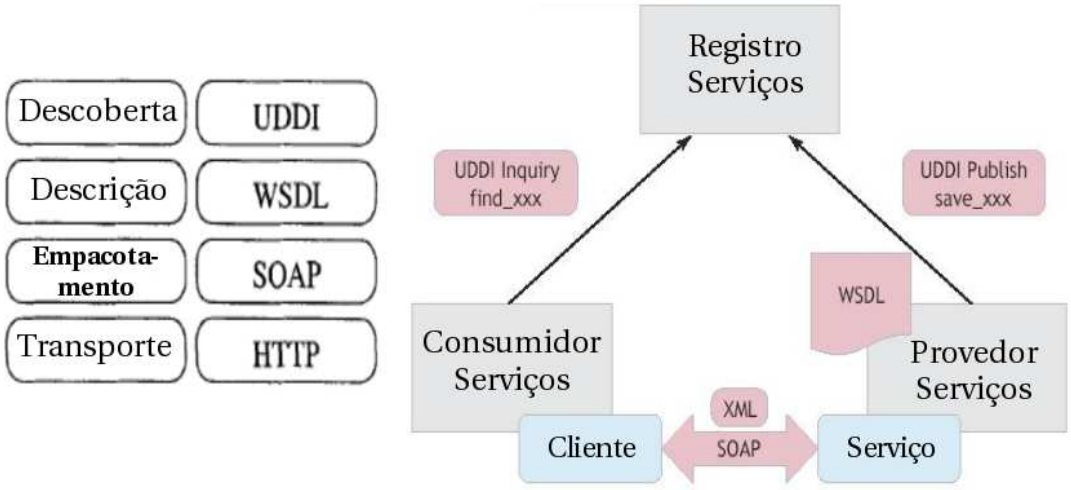


Figura 2.3: Padrões de Serviços Web e arquitetura W3C.

estrutura), fornecendo, por exemplo, transparência na localização e na heterogeneidade de comunicação entre os participantes.

A independência de implementação das tecnologias de serviços Web permitiu cenários com base em SOA onde diferentes aplicações e negócios colaboram via serviços Web distribuídos até mesmo pela Internet. Muitas destas tecnologias se incorporaram em produtos e serviços para diversos propósitos [61]. Uma das áreas mais importantes de aplicação de SOA são as integrações dentro dos cenários de *business-to-business* (B2B) e *Enterprise Application Integration* (EAI). Além disso, soluções de computação distribuída como Web semântica e serviços em grade (*Grid services*) aplicados a *e-Science* também têm adotado SOA.

2.2.3 Serviços Web como uma forma de implementar SOA

Os padrões que formam os serviços Web podem ser utilizadas como uma pilha de protocolos sobre uma camada de transporte, tal como o protocolo HTTP, que é tipicamente usada na Internet. Esta arquitetura é sugerida pela W3C (Figura 2.3), onde o principal benefício é a neutralidade perante a plataformas e fornecedores de middleware distribuído [24]. Contudo, o simples emprego destas tecnologias de serviços Web não transforma uma aplicação distribuída em uma aplicação orientada a serviços [24, 46]. Dentro da abordagem de SOA apresentada na seção 2.2, o emprego de serviços Web como uma forma de implementação é especialmente apropriado quando os serviços precisam operar pela Internet, executarem em diferentes plataformas e poderem ser implantados sem necessariamente atualizar clientes e provedores de uma só vez [24].

Serviços Web é uma proposta mediante às tecnologias tradicionais de integração de

aplicações (*e.g.* CORBA, RMI). Serviços Web são apoiados por padrões de mensagens e interação amplamente utilizados na Internet, tais como *Extensible Markup Language* (XML) e *Uniform Resource Identifier* (URI). Desta forma, o uso destas tecnologias de âmbito global visa reduzir heterogeneidade e promover interoperabilidade entre soluções de sistemas de informação distribuídos em direção ao paradigma de computação orientada a serviços (SOC[24]) [8, 41]. SOC precisamente abrange propostas de abstrações e metodologias de engenharia em níveis inter e intra-organizacionais, infra-estrutura, semântica e componentes de software para alcançar uma arquitetura baseada em serviços em direção à autonomia e heterogeneidade na integração de aplicações [41].

Além dos padrões citados acima, torna-se necessário padrões mais específicos no que diz respeito à operação de descoberta, ligação e interação entre serviços Web. Os padrões de serviços Web são publicados em sua maioria pela *World Wide Web Consortium* (W3C[25]), pela *Organization for the Advancement of Structured Information Standards* (OASIS[26]) e pela *Web Services Interoperability Organization* (WS-I[27]), através de um conjunto de recomendações e diretrizes. Além disso, esses consórcios coordenam grupos de trabalho que elaboram tecnologias que exploram o potencial dos serviços Web em integrar aplicações através da Internet. A definição de serviços Web da W3C traz o emprego de dois padrões: *"O serviço Web tem uma interface descrita em um formato processável por máquina, em específico WSDL. Sistemas interagem com serviços Web em uma maneira prescrita por sua descrição usando mensagens SOAP"* [24]. Já a OASIS propõe o UDDI como um *"método padrão para dinamicamente descobrir e invocar serviços Web"*. A seguir estão descritos os três padrões citados, formando o pilar da tecnologia de serviços Web:

WSDL (Web Service Description Language) [22]: Linguagem em XML para definir o identificador universal, formato da mensagem, tipos de dados, protocolo de transporte, formato de serialização e operações disponíveis dos serviços Web. Documentos WSDL especificam a interface pública dos serviços. Os clientes (ou mediadores representando os clientes) que requisitam um serviço devem usar estes descritores para construir a mensagem SOAP de requisição do serviço. Para isso cabe ao provedor do serviço disponibilizar o WSDL descrevendo os serviços oferecidos através de um ponto de acesso na Web, ou publicando informações sobre o serviço e o ponto de acesso mediante um registro UDDI.

SOAP (Simple Object Access Protocol) [26]: Protocolo para composição de mensagens em formado XML, chamado de envelope SOAP, para troca de informação

[24]Sigla do inglês, *service-oriented computing*.
[25]Website: `http://www.w3.org/2002/ws/`
[26]Website: `http://www.oasis-open.org`
[27]Website: `http://www.ws-i.org`

estruturada entre o agente que requisita o serviço e o agente que provê o serviço Web. Um envelope SOAP básico é formado pelo cabeçalho (opcional) e pelo corpo da mensagem (obrigatório). O cabeçalho pode conter informações específicas da aplicação ou extensões ao modelo, identificados por XML *namespace*, que podem ser usadas tanto por agentes intermediários quanto pelo destino final para manipulação e interpretação da mensagem. Já o corpo da mensagem é o lugar onde a informação bruta gerada pela aplicação é colocada (*e.g.* dados para requisição do serviço, no caso de mensagem enviada pelo cliente do serviço; ou dados de resposta da requisição, no caso de mensagem enviada pelo provedor do serviço).

UDDI (Universal Description, Discovery and Integration) [55, 11]: Esta especificação define serviços Web e modelos de dados para suporte à descrição e descoberta de outros serviços Web. Os serviços Web do UDDI, especificados também por padrões WSDL e SOAP, formam uma API de acesso universal e padronizado ao repositório. A realização desta especificação forma um repositório UDDI, isto é, forma um registro onde as informações sobre serviços Web são persistidas e acessíveis manualmente ou de forma autônoma através da Web [24]. A partir deste registro os serviços Web de negócio são publicados (*e.g.* informando o ponto de acesso via *Uniform Resource Locator* - URL) pelos provedores de serviço e consultados pelos consumidores de serviços. O modelo de dados definido nesta especificação forma uma base taxonômica para descrever os negócios, organizações e provedores dos serviços Web publicados, e também podem ser consultados pela interface pública do registro UDDI.

Os padrões que constituem a base de serviços Web permitem integração entre aplicações sobre diferentes infra-estruturas, e inclusive protegidas por *firewall*, se a comunicação for feita sobre *HyperText Transfer Protocol* (HTTP) [8]. Assim, em cenários de integração considerando composição de serviços (*e.g.* orquestração de serviços usando BPEL4WS [12]), requisitos de segurança (proteção dos dados) e confiabilidade (integridade via comunicação transacional), torna-se crucial o suporte a capacidades além das providas pelos padrões básicos citados. Em suma, o modelo de orquestração é uma hierarquia de atividades (implementadas pelos serviços) e a atribuição de um conjunto de restrições entre as atividades, tais como seqüência, fluxos de execução paralela e decisões, tratamento de eventos e *loops* [8]. Mesmo que soluções e ferramentas comerciais sejam adotadas, extensões ao SOAP são ainda necessárias para continuar existindo a interoperabilidade entre os serviços. Diversas especificações abertas foram propostas para atender aos requisitos de *composability* e confiabilidade [33], também conhecidas como WS-*. A proposta de mediação de serviços Web deste trabalho utiliza amplamente dois destes padrões, chamados WS-Policy e WS-Addressing. A seguir estão as respectivas descrições destes padrões.

WS-Policy [25]: Gramática XML para expressar políticas para sistemas baseados em serviços Web, de acordo com o modelo abstrato de políticas adotada pela especificação (*Policy Model*). Políticas para serviços Web podem expressar funcionalidades, requisitos ou características gerais. Por exemplo, políticas podem especificar seleção de modos de operação de um serviço, ou declarar características de QoS requeridos pelo usuário, entre outros cenários. Contudo, a especificação não restringe como políticas são comunicadas entre o provedor e o solicitante dos serviços. De acordo com o modelo abstrato, uma política é declarada mediante uma expressão *wsp:Policy*. Uma expressão contém coleções de assertivas, que definem assertivas alternativas (*wsp:ExactlyOne*) e conjuntivas (*wsp:All*). E por fim, uma assertiva corresponde a um requisito requerido, e são declarados de acordo com o domínio específico através de *namespaces*.

WS-Addressing [23]: Serviços Web devem ser interoperáveis mesmo operando sobre diferentes protocolos de transporte. Cada protocolo de transporte aplica uma solução própria para endereçar serviços, recursos, ou pontos de rede. Desta maneira, a especificação WS-Addressing preenche essa lacuna, definindo um mecanismo padrão para identificação de serviços Web e troca de mensagens entre múltiplos destinos (*end points*). WS-Addressing define elementos XML para identificar referências a serviços Web e garantir a identificação ponto-a-ponto em cada mensagem. A referência de destino (*endpoint reference*), denotado pela expressão *wsa: EndpointReference*, tem como principais atributos o endereço URI (*wsa:Address*) e a porta do serviço (*wsa:PortType*), de forma a permitir que um serviço em um WSDL seja uma entidade referenciável independentemente da referência concreta do protocolo de transporte. Para as mensagens SOAP, é acrescentado no cabeçalho, dentre outras informações, o campo *wsa:To*, onde o conteúdo é literalmente o endereço atribuído ao serviço de destino da mensagem.

Outros exemplos de especificações são o *WS-Security*, *WS-Coordination*, *WS-Atomic Transaction*, dentre outros.

2.3 Dependabilidade em Arquitetura Orientada a Serviços Web

A proposta deste trabalho tem foco especial em dependabilidade de sistemas cuja arquitetura é orientada a serviços Web. Devido à não-confiabilidade inerente desta arquitetura (*i.e.* autonomia da infra-estrutura de comunicação e dos serviços publicados), é importante levantar e salientar quais atributos, ameaças e meios são relevantes considerando

dependabilidade para serviços Web.

Em uma proposta de uma metodologia para construção de aplicações de serviços Web com alta dependabilidade a partir de serviços com baixa ou nenhuma dependabilidade, *Gorbenko et al.* fazem um estudo preliminar de atributos de dependabilidade e falhas em serviços Web [39]. Desempenho e responsividade são atributos importantes e facilmente obtidos usando solução de computação paralela. Porém confiabilidade e segurança são atributos mais desafiadores. Este trabalho também define uma taxonomia de falhas aplicável a serviços Web a partir de taxonomias existentes.

Tartanoglu et al. [60] exploram os requisitos para a aplicação de técnicas de tolerância a falhas para serviços Web compostos que leva em conta a natureza não confiável do ambiente operacional (Internet e serviços de terceiros). As possíveis falhas que os serviços tem que lidar em composições vão desde falhas em nível de serviços Web, notificados com mensagens de erro, como também falhas em nível de plataforma e falhas durante atualizações *on-line* de serviços e suas interfaces. Já as técnicas de recuperação de erros podem se situar entre duas grandes classes: recuperação por retrocesso (*backward recovery*, *e.g.* tentativas seguidas de chamar um serviço, cancelamento, compensação) ou recuperação por avanço (*forward recovery*, *e.g.* tratamento de exceções em nível de aplicação).

A referência mais importante dessa dissertação é a tese de *Y. Chen*, que apresenta os requisitos e implementação de um mediador de serviços Web chamado WS-Mediator [16] (estudo detalhado do WS-Mediator será feito na seção 2.4.1). Dentro do contexto de dependabilidade, esta pesquisa define quais aspectos foram relevantes para a concepção da solução computacional para mediação e quais manifestações são mais comuns em aplicações baseadas em serviços Web. Adicionalmente às proposições da seção 2.1 serem aplicáveis para se obter tolerância a falhas em SOA, os tópicos detecção de erros e tratamento de falhas foram abordados com maior profundidade.

Y. Chen oferece em sua implementação duas soluções de tolerância a falhas: blocos de recuperação (*recovery blocks*), e programação N-versões (*N-version programming*), ambas no âmbito de estratégia de recuperação de falhas por retrocesso (*backward recovery*). Blocos de recuperação visam manter a continuidade do serviço diante de presença de falhas. Se erros ocorrerem durante a transação, o sistema retrocede para um estado anterior correto, e procedimentos de tentativa ou reconfiguração para serviços redundantes é operado. Já programação N-versões é comumente aplicada em aplicações com criticidade elevada, onde falhas de projeto e de implementação devem ser toleradas. Esta técnica de compensação requer múltiplas versões dos componentes de software ou desenvolvidas de forma independente, mas com especificação funcional idênticas.

Por fim, considerando composição de serviços em SOA e padrões de *web services*, *Zalewski* [65] levanta alguns riscos críticos, já que SOA introduz novas tecnologias, levando à exposição de riscos e falhas imprevistos. É também salientado que metodologias de

análise e projeto para SOA são ainda imaturas neste aspecto. Exemplos de riscos críticos que precisam ser avaliados e tratados são as propriedades ACID (transações) em cenário onde há serviços autônomos ou *stateless*; gerenciamento de concorrência e prevenção de *deadlock*; confiabilidade da comunicação e segurança, de forma que os padrões *WS-** ainda apresentam soluções incompletas. Desta forma os serviços precisam ser especificados de acordo com os modos de falha, efeitos previstos, severidade, ações recomendadas, dentre outras informações.

2.4 Infra-estruturas confiáveis em Arquitetura Orientada a Serviços

2.4.1 WS-Mediator

No contexto das tecnologias atuais, não é possível especificar garantias relacionadas à confiabilidade dos serviços Web. Isto ocorre devido à característica distribuída da arquitetura orientada a serviços, o que pode acarretar diferentes rotas de acesso a um mesmo serviço. Desta forma a dependabilidade dos serviços fica dependente dos recursos físicos de comunicação utilizados em cada cliente. Dentro deste contexto, o componente WS-Mediator [17, 18, 16] é uma infra-estrutura distribuída e situada entre os participantes (clientes e serviços compostos). O componente WS-Mediator foi desenvolvido na Escola de Ciência da Computação da Universidade de Newcastle, Reino Unido. Este componente é composto de submediadores possivelmente distribuídos em localizações geográficas distintas, e que implementam mecanismos de monitoramento, gerando dados de dependabilidade sobre serviços Web. A Figura 2.4 [17] mostra a arquitetura interna de um submediador, que possuem as seguintes características:

- Possui um banco de dados para persistência de dados de resiliência sobre serviços Web, tais como tempo de resposta, taxa de falha, estatísticas de tipos de falhas, etc.
- Implementam interfaces Web para invocações de clientes e outros submediadores em uma arquitetura de agentes.
- Submediadores são funcionalmente idênticos e podem ser disponibilizados tanto localmente em uma LAN quanto globalmente pela Internet. Também pode executar versões leves no próprio cliente.
- O componente de reconfiguração dinâmica é o cérebro do WS-Mediator, processando as requisições e controlando a estratégia empregada pelo componente de tolerância a falhas.

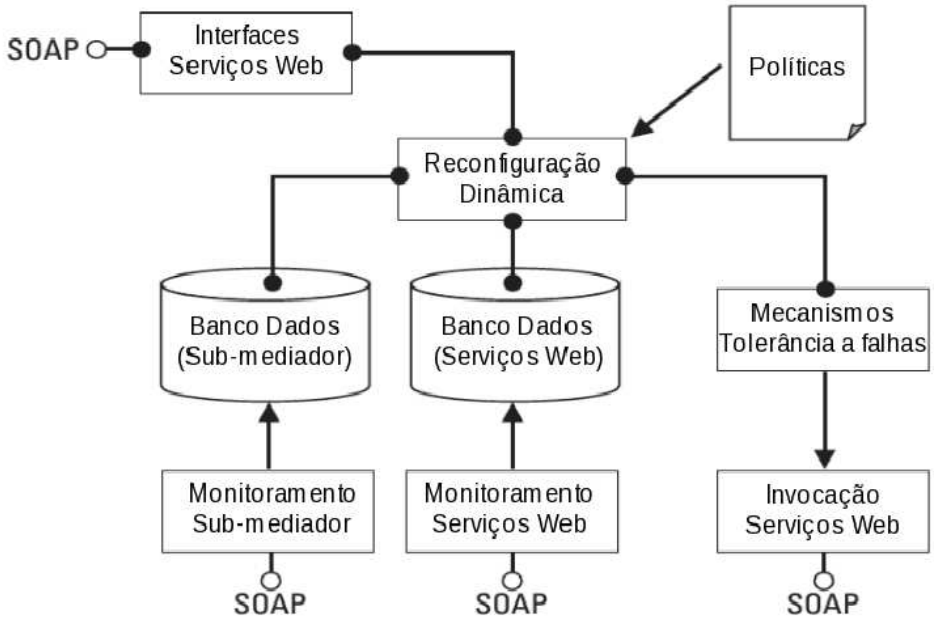


Figura 2.4: Arquitetura interna de um submediador do WS-Mediator.

- O cliente se comunica com o WS-Mediator primeiramente informando a lista de serviços Web candidatos. Já a mensagem SOAP de requisição deve conter: *(i)* todas as mensagens SOAP de requisição para cada candidato; *(ii)* uma política de execução (em formato *WS-Policy*, veja resumo desta especificação na seção 2.2.3) para cada requisição e, *(iii)* uma política de execução global, instruindo o WS-Mediator em como executar as requisições.

WS-Mediator é implementado em Java e possibilita o uso de duas técnicas de detecção de erros: blocos de recuperação e programação N-versões, além de tolerar falhas através de multi-roteamento. Particularmente a estratégia de multi-roteamento é possível graças à diversidade de localização dos submediadores. Com isso, diversos caminhos para entrega de mensagens podem ser determinadas pelos submediadores de forma a aumentar a resiliência dos serviços Web perante a complexidade e falhas inerentes da Internet. Para o desenvolvedor de aplicações, é disponibilizado o componente integrante *WS-Elite*. Trata-se de uma API Java para a aplicação-cliente fazer acesso aos submediadores, compatível com as bibliotecas de serviços Web *Axis* [34] e *Glassfish* [2]. Atualmente, apenas o componente WS-Elite está disponível publicamente para avaliação[28]. Esta versão do componente ainda incorpora as principais funcionalidades de mediação confiável, porém disponíveis para execução em *stand-alone*, isto é, localmente no ambiente onde este componente for integrado.

[28]Website: `http://research.ncl.ac.uk/ws-mediator/`

Por fim, WS-Mediator foi testado em um cenário de forma a mediar requisições a serviços Web no domínio da bioinformática chamado *Blast* [16]. Três serviços *Blast* similares mas situados em diferentes pontos do globo terrestre foram selecionados como candidatos. Foram testados dois modos de operação: usando blocos de recuperação, ordenando os serviços pelo metadado de resiliência e executando primeiro o que apresenta melhor dependabilidade; e usando N-versões, gerando requisições simultâneas e múltiplas respostas ou a resposta trivialmente correta[29] e mais rápida é retornada.

2.4.2 Outras infra-estruturas confiáveis em SOA

A seção 2.4.1 apresenta o WS-Mediator, que é a solução adotada nessa dissertação para integrar-se à solução de mediação confiável. Esta seção apresenta outras propostas de mediadores (*i.e. middlewares*) e arquiteturas confiáveis implementando técnicas de tolerância a falhas e aspectos de dependabilidade considerando serviços Web.

A proposta de uma camada intermediária entre clientes e serviços chamada *wsBus* [31] tem como motivação a adaptação dinâmica em resposta à indisponibilidade de serviços Web. O wsBus intercepta mensagens de forma transparente e aplica técnicas de tolerância a falhas tais como reconfiguração e execução paralela de serviços redundantes. Outras funções foram dadas ao wsBus para operar mediação, por exemplo capacidades como registro de serviços; *proxy* que mapeia serviços virtuais a múltiplos serviços; *gatekeeper* que aplica *WS-Security* e autenticação; suporte a multiprotolocos de transporte e transformação de mensagens via XSLT (*XML Stylesheet Transformation*). Testes foram publicados considerando tanto serviços quanto o próprio wsBus indisponível, de forma que o uso combinado de filas de mensagens, persistência de mensagens e monitores externos ao wsBus permitiu aumento da confiabilidade na comunicação.

Um trabalho similar ao wsBus, chamado *Reliable Web Services Bus* - RBUS [66], é também um mediador para assegurar confiabilidade em serviços Web. São avaliadas propostas para confiabilidade de serviços nos níveis de entrega (transporte e SOAP confiáveis), interação (tolerância a falhas) e comportamento (transações / compensações). Desta forma, RBUS oferece mensagens confiáveis via retransmissão e persistência; tolerância a falhas via redundância; e prioridade entre serviços. Funcionalidades adicionais são a segurança contra SOAP não autorizado via *WS-Security*, e interface para múltiplos protocolos de transporte permitindo transparência para o cliente do serviço. A avaliação do RBUS se deu em cenários onde serviços são acionados direto ou via RBUS, além de introdução de falhas como serviço, mediador ou rede indisponível. O resultado foi melhoria em dois aspectos: confiabilidade, medida em falhas por requisição, e disponibilidade,

[29] Mensagens que não representam mensagens de falhas de comunicação com o serviço remoto. Porém, podem representar mensagens na qual o serviço remoto retorna para indicar alguma falha em prover o contrato do serviço.

medida em porcentagem.

Ainda com o foco em tolerância a falhas e atributos de qualidade de serviço (*Quality of Service* - QoS), Garcia *et al.* [38] apresenta proposta de uma arquitetura que estende a arquitetura básica de serviços Web da W3C. Nesta arquitetura aumentada com tolerância a falhas, são adicionados dois componentes: um *broker* e um *monitor*. Na fase de configuração, os serviços são publicados em um repositório UDDI com interface de programação estendida para metadados de QoS. Na fase de execução, o monitor intercepta mensagens, detecta erros e outros atributos de QoS e atualiza o repositório UDDI com as informações monitoradas. Já o *broker* determina os conjuntos de réplicas de serviços de acordo com o conceito de modelos técnicos (*technical models* - tModels) do repositório UDDI, além de usar as informações de QoS e teste de disponibilidade para selecionar o serviço a ser invocado. Os atributos de QoS adotados estão divididos nas categorias de atributos dinâmicos (*e.g.* tempo de resposta, disponibilidade, etc.) e atributos estáticos, este último relacionado ao suporte às extensões SOAP WS-* para comunicação segura e transacional.

Por fim, algumas implementações de barramento de serviços, chamados de *Enterprise Service Bus* - ESB, tratam alguns requisitos de dependabilidade necessários em sistemas de missão crítica. A infra-estrutura *Apache Synapse* [35], versão 1.2, é apresentada como um ESB leve que apóia os protocolos de transporte mais comuns, transformações de mensagens XML, aderência a padrões como *WS-Addressing*, entre outras atividades de apoio à integração de serviços Web. Desenvolvido em Java, as funcionalidades do mediador Synapse podem ser estendidas a partir de novas implementações das classes de mediação e comandos, além da programação do comportamento de mediação disponível em forma de scripts. Pelo menos três cenários apóiam a confiabilidade da comunicação. O primeiro cenário corresponde à salvaguarda diante da indisponibilidade de serviços Web (*failover*). Pontos de destino são dispostos de forma que, quando o serviço primário falhar (*timeout* da conexão ou a resposta é uma mensagem de falha SOAP), automaticamente a requisição é enviada para o próximo serviço previamente configurado. Desta forma, o envio de mensagens para o serviço que era primário fica suspenso durante um tempo configurado, até que este serviço seja novamente eleito como primário. Já o segundo cenário, além de operar a salvaguarda, executa balanceamento de carga entre múltiplos pontos de destino, usando abordagem de revezamento. Por fim, o terceiro cenário opera *cache* de resposta de um serviço. Por exemplo, se a funcionalidade de *cache* estiver configurada com o limiar de tempo de 20 segundos, a primeira requisição é enviada ao serviço normalmente. E para as requisições subseqüentes dentro deste intervalo, cuja mensagem de requisição seja idêntica (calculando valor *hash*), a requisição não é executada e a mensagem em *cache* é retornada.

Já a infra-estrutura JbossESB [3], na versão 4.2, oferece solução para redundância de

serviços e balanceamento de carga, na forma de implantação dos serviços redundantes em diferentes nós executando uma instância do ESB, ou criando mapeamentos de um mesmo serviço para escutar em diferentes canais e protocolos de transporte. A escolha de qual serviço acessar se dá pela configuração de políticas. JbossESB também suporta comunicação assíncrona na forma de armazenamento das mensagens em banco de dados, e múltiplas tentativas de entrega.

2.5 Resumo

Este capítulo detalhou temas fundamentais para entendimento dos requisitos nas quais a infra-estrutura Arquitetura Mediador deve satisfazer. Dependabilidade em sistemas de software é essencial para que o projeto arquitetural de sistemas críticos esteja preparado para apoiar as atividades automatizadas e ainda apresentar características não-funcionais como confiabilidade e disponibilidade. Em arquitetura orientada a serviços, considerações sobre dependabilidade devem envolver tanto o levantamento dos requisitos específicos para este ambiente, quanto a adoção de soluções de infra-estrutura confiável que apóie todo o ciclo de requisição-resposta e comunicação considerando serviços Web. Diversas soluções para mediação ou barramento de serviços estão disponíveis na literatura e em projetos públicos na Web, alguns provendo maior ou menor suporte para aumento da dependabilidade de sistemas que os utilizam. Contudo a solução especialista, e que será aplicada neste trabalho, será o componente WS-Mediator. Esta escolha está fundamentada principalmente na maturidade da implementação disponível na Internet e da similaridade dos estudos de caso que foram desenvolvidos no campo de *e-science*.

Capítulo 3

Arquitetura Mediador: Infra-estrutura Confiável para Mediação de Serviços Web

Este capítulo apresenta uma solução de infra-estrutura confiável para a Arquitetura Orientada a Serviços Web (SOA), chamada Arquitetura Mediador. A infra-estrutura Arquitetura Mediador consiste em um mediador que atua na comunicação entre a aplicação-cliente e o provedor de serviços. Como o paradigma SOA tem se tornado um guia prático para distribuição e acesso a serviços remotos em nível da Internet, a aplicação de técnicas de tolerância a falhas na camada de comunicação permite alcançar um nível maior de dependabilidade entre os participantes da comunicação.

A principal premissa da implementação da Arquitetura Mediador é que a sua infra-estrutura possa ser aplicável em qualquer domínio de problema. Desta forma, a adoção de padrões de mercado para serviços Web contribui para este objetivo. A análise e projeto da Arquitetura Mediador apresentada neste capítulo, juntamente com os diagramas em UML, foi adaptada do processo baseado em componentes, chamado de *UML Components* [14].

A Seção 3.1 apresenta a análise conceitual da Arquitetura Mediador, com o objetivo de descrever os seus componentes de software, interfaces conceituais e interações esperadas entre os componentes. A Seção 3.2 apresenta o projeto da Arquitetura Mediador, mostrando quais componentes concretos foram selecionados para compor o sistema. O projeto da interface de programação da Arquitetura Mediador mostra quais operações são oferecidas para configurar e disparar o processo de mediação confiável.

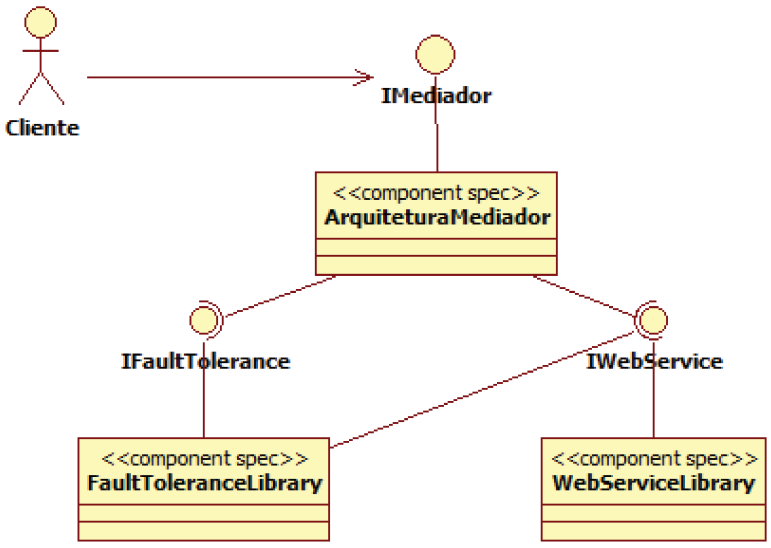


Figura 3.1: Diagrama de componentes da Arquitetura Mediador.

3.1 Descrição da Arquitetura Mediador

A Arquitetura Mediador consiste em uma aplicação Web para mediação confiável entre a aplicação-cliente e o provedor de serviços Web em uma Arquitetura Orientada a Serviços (SOA). O processo de mediação descrito neste trabalho foi concebido como um diálogo entre a aplicação-cliente e a Arquitetura Mediador. O diálogo foi implementado usando serviços Web. Desta forma, tanto a aplicação-cliente quanto a Arquitetura Mediador se beneficiam de uma biblioteca para abstração e mapeamento de operações de serviços Web para operações em diferentes linguagens de programação. O projeto da interface de programação permite a separação conceitual entre a lógica do diálogo de mediação e a implementação da mediação confiável. A mediação confiável consiste na implementação das técnicas citadas na seção 2.3 para alcançar a dependabilidade desejada em SOA. Particularmente, o objetivo é disponibilizar as estratégias de tolerância a falhas *blocos de recuperação* e *N-versões* para aplicações-clientes em geral a partir de uma infra-estrutura Web interoperável.

A Figura 3.1 apresenta o diagrama de componentes da Arquitetura Mediador em UML. Os componentes são anotados com o estereótipo *component spec*, que indica os componentes da análise ainda a serem concretizadas por implementações reais. A seguir são descritas as funcionalidades de cada componente.

Componente ArquiteturaMediador: É o componente que mantém o estado do diálogo de mediação com a aplicação-cliente. Este componente oferece a interface *IMediador*,

principal ponto de acesso aos serviços de mediação. Esta interface oferece operações para receber os parâmetros de configuração da Arquitetura Mediador, para receber a ordem de disparo da mediação e para retornar o resultado com a mensagem de resposta dos serviços Web configurados. Para realizar as funcionalidades de mediação, o componente *ArquiteturaMediador* depende da interface *IFaultTolerance*, que efetivamente executa a mediação confiável, a partir da configuração recebida via interface *IMediador*. Por fim, para que o componente *ArquiteturaMediador* consiga se comunicar de forma efetiva usando a tecnologia de serviços Web, a dependência da interface *IWebService* fornece as principais funcionalidades para publicação de serviços Web, envio e recebimento de mensagens Simple Object Access Protocol (SOAP), entre outras.

Componente FaultToleranceLibrary: Ele é responsável pela lógica de mediação confiável, e recebe as configurações, como por exemplo, estratégias de tolerância a falhas a serem executadas para cada serviço redundante informado, etc. Através da interface *IFaultTolerance*, o disparo da mediação é feito por uma operação específica, que pode retornar uma ou mais mensagens de acordo com o formato da resposta de cada serviço requisitado, ou uma mensagem de erro indicando que a requisição final ao serviço não foi concretizada. Como o componente *FaultToleranceLibrary* também deve acessar serviços Web, as funcionalidades do componente *WebServiceLibrary* também são oferecidas pela interface *IWebService*.

Componente WebServiceLibrary: Ele consiste na biblioteca para acesso à camada de comunicação pela rede Internet, por exemplo usando protocolo HyperText Transfer Protocol (HTTP), porém usando a tecnologia de serviços Web. A interface de programação desta biblioteca fornecerá meios para interpretar interfaces no formato Web Service Description Language (WSDL), enviar e receber mensagens SOAP, e mapear dados em formato EXtensible Markup Language (XML) trafegados na rede para um formato esperado pela aplicação-cliente.

A Figura 3.2 mostra um diagrama de colaboração em UML, onde os objetos são instâncias hipotéticas dos componentes apresentados na Figura 3.1. Este diagrama descreve, de forma resumida, um cenário de mediação confiável, desde a configuração inicial até o resultado final. As operações descritas na Figura 3.2 apenas representam trocas de mensagens esperadas entre os componentes em tempo de execução, e, portanto, sofrerão refinamentos para o projeto e implementação da infra-estrutura Arquitetura Mediador.

O objeto *cliente* tem associado a ele uma instância do componente *WebServiceLibrary*, chamada de *clientInstance*. Através da operação *getMediatorReference* (mensagem 1), o objeto *cliente* obtém uma referência remota para um objeto do tipo *ArquiteturaMediador*, e a partir desta referência é possível efetuar requisições de serviços Web para o

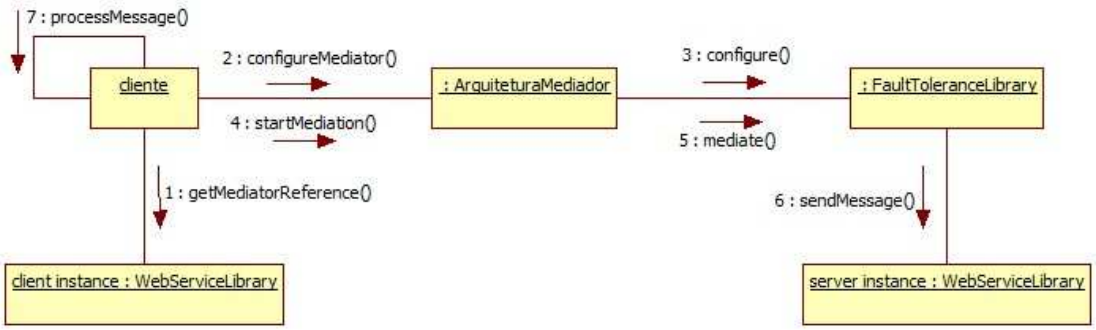


Figura 3.2: Diagrama de colaboração UML com as interações entre os componentes da Arquitetura Mediador.

mesmo objeto. O próximo passo é submeter dados de configuração, chamando a operação *configureMediator()* (mensagem 2). Por sua vez, o objeto *ArquiteturaMediador* efetiva a configuração do sistema por intermédio do objeto *FaultToleranceLibrary*, que oferece a operação *configure()* para este propósito (mensagem 3).

Em um momento posterior, o objeto *cliente* inicia o processo de mediação através da operação *startMediation()*, enviada para o objeto *ArquiteturaMediador* (mensagem 4). O objeto *ArquiteturaMediador* então dispara a operação *mediate()* (mensagem 5), que instrui o objeto *FaultToleranceLibrary* a iniciar o envio de uma ou mais mensagens para os serviços candidatos, de acordo com a política de configuração previamente escolhida. Para isso o objeto *FaultToleranceLibrary* delega as requisições recebidas para um objeto chamado *serverInstance*, do tipo *WebServiceLibrary*, requisitando a operação *sendMessage()* (mensagem 6). Na conclusão do processo, a aplicação-cliente recebe do objeto *ArquiteturaMediador*, no retorno da operação *startMediation()*, a mensagem de resposta. Para fins ilustrativos, a mensagem é finalmente processada pelo *cliente* em uma operação chamada *processMessage()* (mensagem 7).

3.2 Projeto da Arquitetura Mediador

A Seção 3.1 apresentou o modelo de análise da infra-estrutura proposta nesta dissertação. O projeto da Arquitetura Mediador define quais componentes concretos realizarão os componentes abstratos da solução proposta (Figura 3.1). A granularidade das interfaces também será modificada para atender às necessidades específicas do projeto.

A Figura 3.3 apresenta a configuração arquitetural da Arquitetura Mediador. O componente de especificação *FaultToleranceLibrary* é realizado pelo componente de terceiros chamado WS-Mediator (Seção 2.4.1). Algumas tecnologias aplicadas ao projeto do

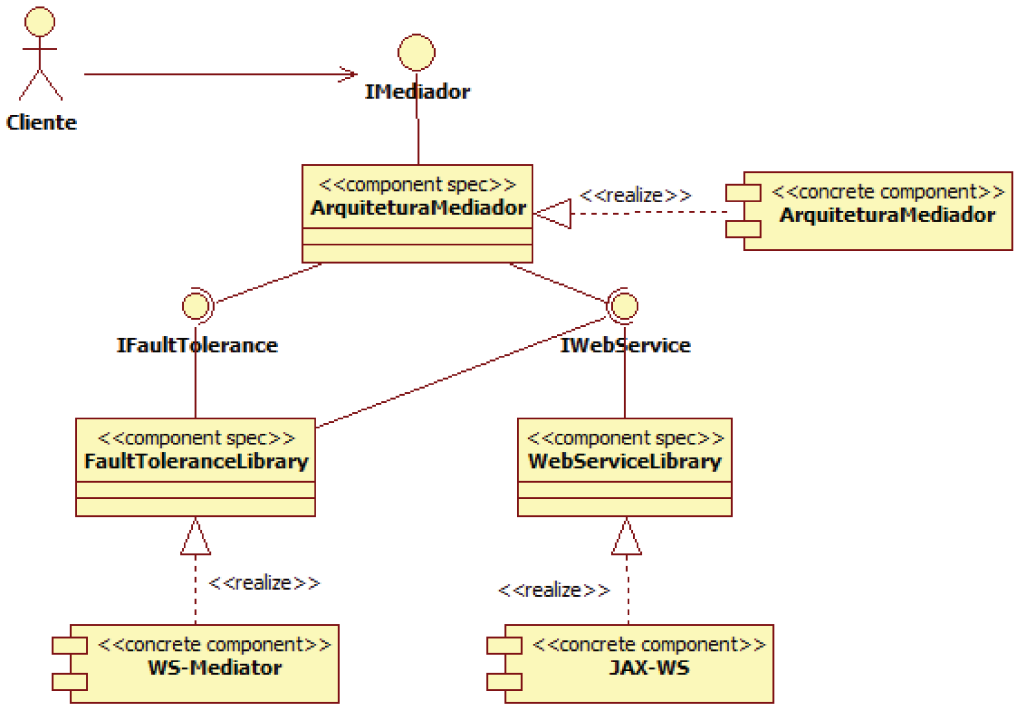


Figura 3.3: Configuração arquitetural da Arquitetura Mediador.

WS-Mediator, por questões práticas, foram também adotadas por este projeto. Para a implementação foi utilizada a plataforma de software da linguagem *Java 6 Standard Edition*. Para a realização do componente de especificação *WebServiceLibrary*, foi adotada a mesma solução escolhida pelos desenvolvedores do componente WS-Mediador. Neste caso, foi utilizada a biblioteca Java da *Sun Microsystems* chamada *Java API for XML Web Services (JAX-WS)* 2.1 [7]. O componente de especificação *ArquiteturaMediador* é realizado pelo componente concreto de mesmo nome, desenvolvido no contexto deste trabalho, utilizando as tecnologias Java citadas acima.

3.2.1 Projeto da interface de programação da Arquitetura Mediador

As interfaces *IFaultTolerance* e *IWebService* mostradas na Figura 3.3 têm o papel de encapsular a implementação das funcionalidades oferecidas pelos componentes *FaultToleranceLibrary* e *WebServiceLibrary*, respectivamente. Contudo, estas interfaces não serão desenvolvidas no âmbito deste trabalho. Elas serão materializadas por interfaces de programação já desenvolvidas pelos fornecedores dos componentes concretos em cada caso.

Os detalhes destas interfaces, por exemplo operações e formas de utilização, serão abordadas no projeto detalhado do Capítulo 4. Portanto, para o projeto conceitual desta seção, apenas será descrito o desenvolvimento da interface *IMediador*.

O projeto da Arquitetura Mediador visa oferecer na sua interface pública operações que usam protocolo SOAP com o propósito de configuração e disparo da atividade de mediação confiável. O componente concreto WS-Mediator, por intermédio da sua interface de programação, é o receptor das configurações de dependabilidade e serviços-candidatos fornecidos pela aplicação-cliente. Desta forma, tais operações na interface da Arquitetura Mediador foram implementadas com os mesmos nomes e parâmetros da interface de programação do componente WS-Mediator, por serem intuitivas no seu propósito de configuração da mediação confiável. Conseqüentemente, Arquitetura Mediador estende o modelo de uso oferecido por WS-Mediator, de forma a prover uma solução distribuída para mediação da comunicação de aplicações-clientes que utilizam serviços Web.

Estas operações foram mapeadas diretamente para operações oferecidas pela interface de programação do componente concreto WS-Mediator. Esta decisão Desta forma, Arquitetura Mediador estende o modelo de uso oferecido por WS-Mediator, de forma a prover uma solução distribuída para mediação da comunicação de aplicações-clientes que utilizam serviços Web.

Outra decisão que fundamentou o projeto da interface é o fato de que a opção pelo estado do diálogo da mediação confiável ser mantido pelo componente *ArquiteturaMediador*. A solução utilizada neste caso foi a conformidade com o projeto para serviços Web *stateful* [6, 44]. Serviços Web *stateful* são serviços cujas instâncias guardam informações referentes ao diálogo empregado entre cliente e servidor. Para isso é utilizado um protocolo que permite referenciar o objeto onde é mantido o estado do diálogo, e implementado sobre o protocolo HTTP, que é *stateless*. Por fim, é inerente a uma infra-estrutura de mediação precisar manipular mensagens SOAP (*e.g.* enviar, receber, tratar falhas em nível de comunicação) de forma não-intrusiva. A conseqüência é que a interface *IMediador* deve estar preparada para trabalhar com essas funcionalidades.

A Figura 3.4 mostra como a interface *IMediador* foi refinada para melhor acolher os requisitos detalhados nesta seção. Desta forma, o componente de especificação *ArquiteturaMediador* realiza as seguintes interfaces: *IMediatorManager*, *IMediatorServer* e *IMediatorExecution*, subtipos da interface *IMediador*. A interface *IMediatorManager* visa disponibilizar uma instância *stateful* do serviço de configuração: *IMediatorServer*. Já a interface *IMediatorExecution* é especializada no disparo da mediação confiável, orientada pelas configurações mantidas no serviço *IMediatorServer*. A Figura 3.5 ilustra o diagrama de classes em UML que mostra o componente *ArquiteturaMediador* sendo responsável por realizar as três interfaces citadas. As três interfaces são anotadas com o estereótipo *web-service interface*, indicando que elas são publicadas como serviços Web após implantadas

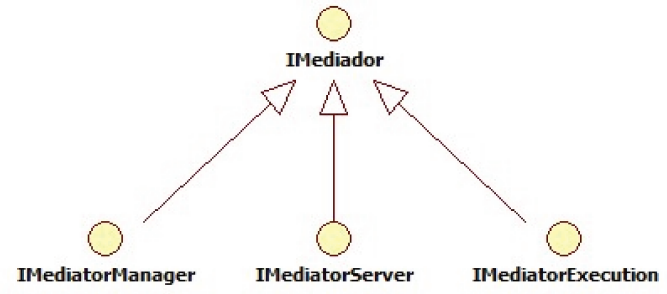


Figura 3.4: Refinamento da interface IMediador.

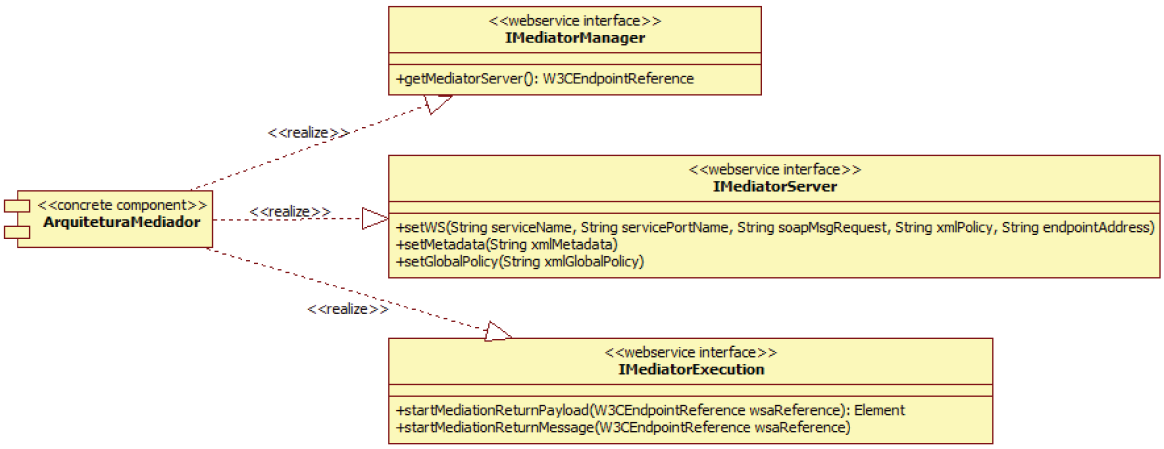


Figura 3.5: Projeto das interfaces públicas da Arquitetura Mediador.

em um servidor de aplicação compatível com tecnologia JAX-WS. As operações públicas também são apresentadas em cada interface. As Seções 3.2.2, 3.2.3 e 3.2.4 descrevem aspectos contratuais das interfaces e operações dos respectivos serviços propostos até o momento.

3.2.2 Serviço IMediatorManager

O serviço descrito pela interface *IMediatorManager* é o ponto de acesso inicial da infraestrutura Arquitetura Mediador. Através desta interface é possível obter uma instância do serviço que implementa a interface *IMediatorServer*. Portanto, o papel da interface *IMediatorManager* é o de uma fábrica de objetos.

Um objeto do tipo *IMediatorManager* é responsável por manter o estado da configuração e mediação, o qual tanto a aplicação-cliente quanto os demais componentes

do sistema precisarão referenciar. Como o requisito tratado pelo *IMediatorManager* é possibilitar referências a instâncias do serviço remoto *IMediatorServer*, a especificação recomendada para este tipo de informação, parte integrante da pilha de protocolos recomendada pela The World Wide Web Consortium (W3C), é chamada *WS-Addressing* [23]. A única operação oferecida pela interface *IMediatorManager* chama-se *getMediatorServer()*. Esta operação retorna uma referência remota de acordo com a especificação *WS-Addressing*, que é enviada como atributo XML em cada mensagem de requisição, disparada pela aplicação-cliente para que o servidor distingua o objeto do tipo *IMediatorServer* que foi associada à referência.

3.2.3 Serviço IMediatorServer

É o serviço que mantém o estado da configuração da Arquitetura Mediador. Portanto, trata-se de um serviço Web *stateful*. A principal decisão de projeto envolvida neste serviço é a possibilidade de manter o estado da configuração da Arquitetura Mediator entre mediações de diferentes serviços. Outro benefício alcançado por esta abordagem é o apoio a diversas aplicações-clientes simultâneas, onde cada aplicação-cliente tem sua própria configuração de mediação. Como citado na Seção 3.2.2, uma referência remota obtida do serviço *IMediatorManager* é necessária para que uma configuração seja acessada em um momento posterior. Se a referência remota não mais corresponder a um objeto no ambiente do servidor, ou simplesmente é informado uma referência inválida, este serviço devidamente retorna uma mensagem de erro. Neste caso, cabe à aplicação-cliente requisitar uma nova instância para o serviço *IMediatorManager*, como descrito na seção 3.2.2, e reexecutar a atividade de configuração do Mediador como feito anteriormente.

As mesmas configurações requeridas pela implementação do WS-Mediator foram mapeadas para operações para este serviço Web. A seguir estão descritas as operações públicas da interface *IMediatorServer*.

setWS(String serviceName, String servicePortName, String soapMsgRequest, String xmlPolicy, String endpointAddress): Configura a Arquitetura Mediador para mediar o serviço-alvo representado pelos parâmetros `serviceName`, `servicePortName` e `endpointAddress`. Já os parâmetros `soapMessageRequest` e `xmlPolicy` constituem o corpo da mensagem de requisição e são repassados para a mensagem de requisição ao serviço-alvo.

setMetadata(String xmlMetadata): Informa, para cada serviço adicionado pelo método *setWS*, informações de dependabilidade. Essas informações são utilizadas para ordenar os serviços que serão mediados, de forma que o serviço com maior grau de dependabilidade é chamado primeiro, se a estratégia utilizada for *blocos de recuperação*. Se a estratégia for *N-versões*, informações de dependabilidade são usadas

para selecionar os melhores serviços-candidatos para serem requisitados. Em ambos os casos, também serviços com dependabilidade abaixo do limite estipulado podem não ser selecionados para o processo de mediação. Adicionalmente, ferramentas de monitoramento de serviços podem ser usados para produzirem dados de partida.

setGlobalPolicy(String xmlGlobalPolicy): Configura o modo de operação, ou estratégia de tolerância a falhas, da Arquitetura Mediador, que são mapeadas para os modos de operação atualmente implementados pelo componente WS-Mediator. Por exemplo, pode-se selecionar entre os modos *blocos de recuperação*, *N-versões quickest* ou *N-versões vote*[1]. Também pode ser configurado o *timeout* global do processo de mediação.

3.2.4 Serviço IMediatorExecution

É o serviço responsável por disparar uma mediação de requisição entre a aplicação-cliente e serviços Web previamente configurados (*i.e.* serviços-candidatos). Os serviços-candidatos são configurados por intermédio do serviço *IMediatorServer*, aplicando as políticas de mediação também informadas em *IMediatorServer*. É responsabilidade de um serviço *IMediatorExecution* entregar o resultado da mediação à aplicação-cliente. O resultado de algum dos serviços-alvo requisitados pode ser normal ou excepcional.

O projeto deste serviço levou em consideração o modelo de operação intrínseco de uma infra-estrutura como a Arquitetura Mediador: o resultado recebido pela aplicação-cliente, isto é, a mensagem de resposta após a mediação ter sido realizada com sucesso, deve ser idêntica caso o acesso aos serviços configurados fossem acessados diretamente. Esta abordagem não-intrusiva perante as mensagens de requisição e resposta exige que elas sejam tratadas em nível de mensagens SOAP. O suporte para que a interface *IMediatorExecution* opere com esta abstração é provida pelo componente *WebServiceLibrary*. A interface *IMediatorExecution* declara duas operações descritas a seguir.

startMediationReturnPayload(wsaReference): Esta operação dispara uma execução de mediação. A configuração é identificada pelo parâmetro *wsaReference*, que encapsula uma referência remota relativa a uma instância do serviço *IMediatorServer*, de acordo com a especificação *WS-Addressing*. Esta referência é obtida do serviço *IMediatorManager*, durante o processo de configuração anterior ao início da mediação. O retorno desta operação é o *conteúdo* da mensagem de resposta (*payload* do corpo da mensagem SOAP) do serviço-alvo efetivamente requisitado, encapsulada em uma

[1] N-versões *quickest* é uma implementação cujo objetivo é retornar a resposta obtida em menos tempo, dentre serviços-candidatos requisitados simultaneamente. Já as técnicas blocos de recuperação e N-versões *vote* são ilustradas na Seção 2.1.

estrutura XML de acordo com a especificação W3C DOM Element [20], declarada no retorno desta operação.

startMediationReturnMessage(wsaReference): Sob o aspecto funcional, a operação *startMediationReturnMessage* é idêntica à operação *startMediationReturnPayload*, com a diferença de que a mensagem de resposta do serviço-alvo é retornada integralmente, e não somente o *payload*. Apesar desta interface declarar uma estrutura XML para o tipo de retorno, trata-se de uma estrutura vazia, e a aplicação-cliente deve saber como processar a mensagem como um todo, incluindo o *payload* e possíveis anexos.

Ambas operações oferecem meios convenientes para solicitar o serviço de mediação. A forma como os serviços-candidatos retornam os dados de resposta e o suporte provido pelo componente *WebServiceLibrary* no ambiente de execução da aplicação-cliente são atributos que podem determinar a utilização de uma operação ou outra em específico. Por exemplo, a recuperação de arquivos anexados à mensagem de resposta, que na especificação de serviços Web da W3C é tratada pela recomendação *SOAP with Attachments* [21], é atendida pela operação *startMediationReturnMessage*. O Capítulo 4 apresenta um exemplo de solução JAX-WS para utilização de ambas operações.

3.3 Resumo

Este capítulo introduziu a análise e projeto da infra-estrutura Arquitetura Mediador, nome dado à infra-estrutura confiável para mediação de serviços Web voltada a arquiteturas orientadas a serviços. A análise da Arquitetura Mediador mostrou os componentes de software, interfaces de análise e dependências previstas. Foram especificadas as responsabilidades de cada componente, baseados em requisitos tais como manter o diálogo da mediação confiável, prover soluções de tolerância a falhas na comunicação e prover soluções para interação com serviços remotos baseados nas especificações W3C.

Em seguida foi apresentada o projeto da Arquitetura Mediador, onde alguns dos componentes especificados na análise são realizados por componentes de terceiros. Já o componente principal, que oferece os serviços Web para configuração e disparo da mediação confiável, é realizado por um componente implementado no contexto deste trabalho. Influenciada pelos projetos de interface dos componente WS-Mediator e JAX-WS, a interface pública da infra-estrutura Arquitetura Mediador foi refatorada para acomodar serviços *stateful* e mensagens em nível de envelope SOAP. As operações públicas de cada interface são abordadas, sendo que alguns aspectos de implementação são abordados no Capítulo 4, onde o projeto é detalhado na sua implementação.

Capítulo 4

Projeto Detalhado da Arquitetura Mediador

O objetivo deste capítulo é apresentar detalhes de implementação que foram considerados na construção da infra-estrutura Arquitetura Mediador, e que não foram detalhados no Capítulo 3. A Seção 4.1 apresenta o diagrama de classes em UML dos componentes implementados neste trabalho, detalhando a utilização dos componentes de terceiros WS-Mediator e Java API for XML Web Services (JAX-WS). A Seção 4.2 complementa as informações apresentadas, mostrando como uma aplicação-cliente deve utilizar a Arquitetura Mediador, em formato de diagrama de seqüência em UML. Como apoio a este exemplo foi utilizado o suporte da Application Programming Interface (API) do ambiente de programação oferecido por JAX-WS.

4.1 Implementação da Arquitetura Mediador

O projeto da Arquitetura Mediador mostrado na Seção 3.2 usa o componente de terceiros WS-Mediator como solução para mediação confiável e também usa JAX-WS 2.1 como componente Java de suporte para implementar e acessar serviços Web em XML. Outras tecnologias adotadas foram Java 6 Standard Edition e o servidor de aplicação Glassfish V2. Na Seção 3.2.1, foi definido que as interfaces dos componentes que realizam *FaultToleranceLibrary* e *WebServiceLibrary* serão materializadas por interfaces de programação já oferecidas e documentadas pelos fornecedores nas respectivas distribuições dos componentes.

A Figura 4.1 apresenta o diagrama de classes em UML da infra-estrutura Arquitetura Mediador. As classes principais são: `MediatorManagerImpl`, `MediatorServerImpl` e `MediatorExecutionImpl`, que implementam, respectivamente, os serviços Web *MediatorManager*, *MediatorServer* e *MediatorExecution*. Esses serviços são representados no

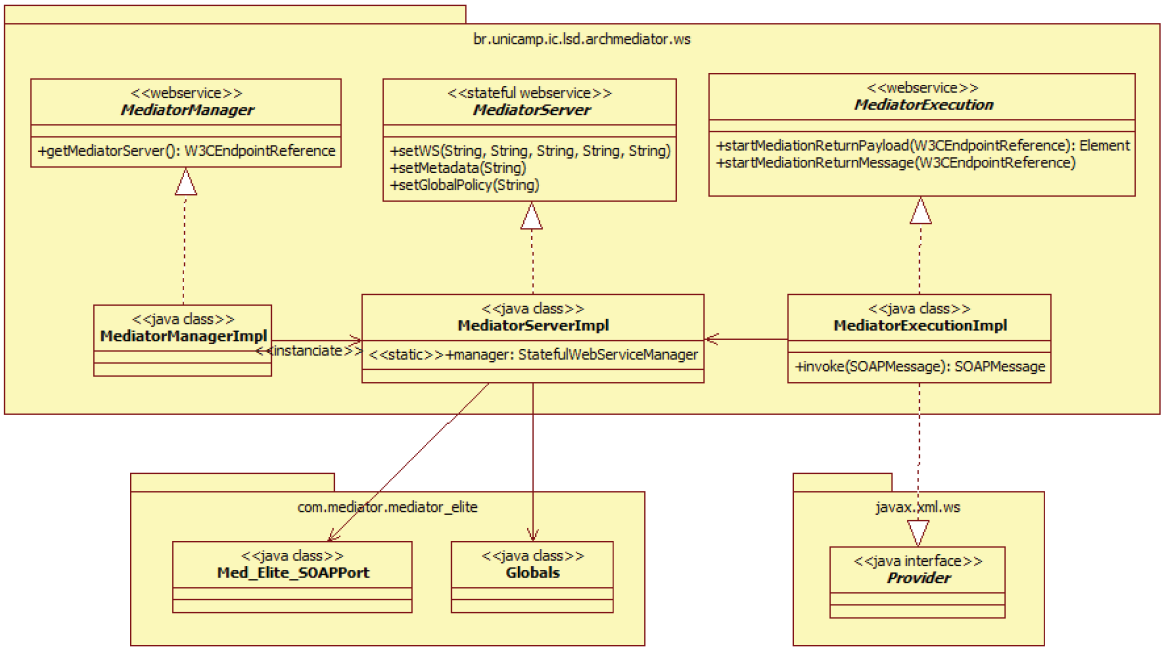


Figura 4.1: Diagrama de classes da Arquitetura Mediador.

diagrama como interfaces em UML, contudo *MediatorManager* e *MediatorServer* não são implementadas como interfaces Java, mas são implicitamente geradas pela ferramenta *wsgen*[1]. Desta forma, estas interfaces são especificadas através de metadados nas classes de implementação, via um conjunto de anotações padronizadas e disponíveis pela API do Java 6 SE.

Este projeto detalhado utilizou convenções de código na geração das interfaces, como por exemplo as operações públicas são exportadas como operações Simple Object Access Protocol (SOAP) na interface gerada. Já a interface *MediatorExecution* é especificada por um arquivo WSDL que não foi gerada automaticamente, devido à classe de implementação utilizar mecanismos para manipulação de mensagens, e, desta forma, não especificando todas as informações para esse processo.

Nas Seções 4.1.1, 4.1.2 e 4.1.3 são detalhadas as três classes de implementações de serviços, respectivamente, `MediatorManagerImpl`, `MediatorServerImpl` e `MediatorExecutionImpl`. Estas seções discorrem sobre as decisões de projeto relacionadas a estas

[1] *wsgen* é uma ferramenta disponível nas distribuições JAX-WS e *Glassfish* que, a partir de uma implementação de um serviço Web, gera artefatos portáveis (*e.g.* arquivos *.class* e arquivos no formato Web Service Description Language (WSDL)) para implantação do serviço em um servidor compatível com JAX-WS. Neste trabalho, *wsgen* foi utilizada durante processo de implantação automatizada do servidor *Glassfish*.

classes e às outras classes dependentes.

4.1.1 Classe MediatorManagerImpl

Esta classe implementa o serviço Web *IMediatorManager* concebido no projeto da infraestrutura Arquitetura Mediador (Seção 3.2.2). `MediatorManagerImpl` é responsável por instanciar a classe `MediatorServerImpl`, funcionando como uma fábrica de objetos [36].

Uma instância da classe `MediatorServerImpl` é *exportada*[2] através da operação `getMediatorServer()`, que retorna uma referência de acordo com o padrão WS-Addressing. Esta referência é encapsulada em Java pela classe `javax.xml.ws.wsaddressing.W3CEndpointReference`. Ao receber esta referência, a aplicação-cliente pode a partir dela enviar uma mensagem para o serviço referenciado, que conseqüentemente é atendida pelo objeto *stateful* associado à referência. Esta abordagem é documentada na própria especificação para serviços *stateful* da plataforma JAX-WS [6, 44].

4.1.2 Classe MediatorServerImpl

Esta classe implementa o serviço Web *IMediatorServer*, concebido no projeto da infraestrutura Arquitetura Mediador (Seção 3.2.3). `MediatorServerImpl` mantém o estado da configuração da Arquitetura Mediador, de acordo com a especificação *stateful*.

A configuração da Arquitetura Mediador informada por uma aplicação-cliente é efetivamente mantida nas instâncias das classes do WS-Mediator, representado na Figura 4.1 pelas classes `Med_Elite_SOAPPort` e `Globals`. O tempo de vida de todas as instâncias envolvidas na configuração é determinada pela operação `setTimeout()` do controlador de serviços *stateful* da plataforma JAX-WS, onde é informado em milissegundos quanto tempo as instâncias *exportadas* devem ser mantidas acessíveis pelo servidor em memória sem que algum acesso seja efetuado. Caso uma referência remota se torne inválida por este controle, seja porque o servidor foi reiniciado ou seja porque o tempo de vida do objeto expirou, basta que a aplicação-cliente reexecute a tarefa de configuração.

Sob o ponto de vista da implementação deste serviço Web, é necessário que a classe `MediatorServerImpl` dê suporte para especificação WS-Addressing, através da anotação `javax.xml.ws.soap.Addressing`, e sinalize o comportamento *stateful* através da anotação `com.sun.xml.ws.developer.Stateful`. Para completar o modelo de programação, as demais classes da aplicação que utilizam o serviço *stateful* precisam acessar a classe `com.sun.xml.ws.developer.StatefulWebServiceManager`. Uma instância de `StatefulWebServiceManager` é recebida através de um atributo de classe, no caso o atributo

[2]Exportada significa que um objeto *stateful* instanciado no contexto de uma requisição passa a ter o seu acesso gerenciado pelos objetos controladores do JAX-WS, de forma que o encaminhamento automático das mensagens ocorra a partir de uma referência remota gerada pela mesma operação de exportação.

`manager` da classe `MediatorServerImpl`, de acordo com a especificação *injection* [32, 42]. A instância de serviço criada pelo `MediatorManagerImpl` é disponibilizada para o acesso remoto pela operação `manager.export()`, que retorna uma referência única no ambiente de execução da infra-estrutura Arquitetura Mediador. Desta forma, para que uma requisição de um serviço seja processada por uma instância em particular, tanto as classes de controle de JAX-WS quanto alguma classe da aplicação (que neste caso é o `MediatorExecutionImpl`) devem chamar a operação `manager.resolve()`, passando a mesma referência retornada inicialmente.

4.1.3 Classe MediatorExecutionImpl

Esta classe implementa o serviço Web *IMediatorExecution* concebido no projeto da infra-estrutura Arquitetura Mediador (Seção 3.2.4). `MediatorExecutionImpl` é responsável por disparar a execução da mediação confiável, a partir de uma configuração mantida em um objeto do tipo MediatorServerImpl.

No projeto do serviço de execução, foi mencionado que as mensagens de requisição e resposta da mediação devem ser tratados de forma não-intrusiva, em nível de mensagens SOAP. Em Java, mensagens SOAP são encapsuladas pela classe `javax.xml.soap.SOAPMessage`. Complementarmente, a classe `MediatorExecutionImpl` usa a plataforma JAX-WS para acesso direto ao objeto `SOAPMessage`, implementando a interface `javax.xml.ws.Provider`, conforme mostrando na Figura 4.1. Para que o serviço realizado por `Provider` seja publicado com sucesso, é necessário informar a porta, *namespace* e o nome do serviço através da anotação `javax.xml.ws.WebServiceProvider`. Também a classe que realiza `Provider` precisa implementar a operação `invoke`, que responde às requisições de serviços processando o objeto `SOAPMessage` recebido como argumento da operação.

Apesar das três classes de implementação realizarem cada uma um serviço da infra-estrutura Arquitetura Mediador, a solução de encaminhamento das mensagens feita pelas classes controladoras do componente JAX-WS é diferente para um objeto `Provider`, em comparação às classes Plain Old Java Object (POJO) [42], anotadas como `WebService`, *e.g.* `MediatorManagerImpl` e `MediatorServerImpl`. Para objetos `Provider`, o fato da aplicação-cliente ter chamado uma certa operação `a()` e informado seus respectivos parâmetros de entrada, implica apenas na mensagem encaminhada para a operação `invoke()`. Todo o tratamento do conteúdo em EXtensible Markup Language (XML) deve ficar a cargo do implementador da interface `Provider` para que o serviço esperado em `a()` seja efetivado. Já no segundo caso, usando POJO, é utilizado anotações Java e convenções de código que basicamente sinalizam à plataforma JAX-WS os mapeamentos Java para XML, e vice-versa. Como resultado, estes mapeamentos são concretizados como mapeamentos da biblioteca Java API for XML Binding (JAXB). A partir deste mecanismo, a mesma

aplicação-cliente chamando a operação `a()` implica no processamento da mensagem SOAP por parte dos controladores do componente JAX-WS, que delega a requisição para o método de instância `a()` da respectiva classe Java implementadora.

Para que a aplicação-cliente consiga utilizar de forma efetiva um serviço Web atendido por um objeto `Provider`, isto é, conhecer as operações e tipos de dados que devem ser trafegados entre as partes, uma interface de serviços Web via WSDL precisa ser construída. O arquivo WSDL é associado à implementação via atributo `wsdlLocation` da anotação `WebServiceProvider`, e implantado juntamente com a classe que implementa `Provider`. Desta forma, a aplicação-cliente estará preparada para construir uma mensagem que um `Provider` espera, e também para processar a resposta do serviço.

A especificação WSDL do serviço *MediatorExecution* declara as operações `startMediationReturnPayload(wsaReference)` e `startMediationReturnMessage(wsaReference)`. Ambas operações estão descritas na interface projetada na Seção 3.2.4, também ilustrada na Figura 4.1. Sob o ponto de vista do projeto detalhado, os parâmetros `wsaReference` são instanciados como objetos do tipo `W3CEndpointReference`. Ambas operações declaram uma estrutura XML como tipo de resposta. Contudo, a operação `startMediationReturnMessage()` declara uma estrutura vazia, que na implementação é útil para instruir o processo de mapeamento da biblioteca JAXB a não gerar nenhum tipo Java de retorno (retorno `void`).

A disponibilização destas operações funcionalmente similares visa a facilidade do seu uso. Contudo, a escolha da operação correta deve ser embasada em decisões da aplicação-cliente que utiliza a infra-estrutura Arquitetura Mediador. A decisão de qual operação utilizar depende de *(i)* qual API de serviços Web a aplicação-cliente utiliza e *(ii)* quais informações serão úteis na mensagem de resposta. Como a Arquitetura Mediador não conhece a estrutura de resposta que a aplicação-cliente obtém ao final da mediação, ambas operações não especificam nenhum mapeamento em particular para o tipo do retorno. A operação `startMediationReturnPayload()` é adequada quando a aplicação-cliente utiliza *proxies*, tais como gerados pela ferramenta *wsimport*[3], pois o retorno especificado desta operação é a *tag* reservada pelo *XML Schema xs:AnyType*. Em Java, o mapeamento de *xs:AnyType* é direcionado para `java.lang.Object`. Contudo, como o serviço de execução retorna o *payload* (conteúdo contido no corpo da mensagem original), a aplicação-cliente conseguirá consumir os dados de resposta fazendo *casting* para um objeto do tipo `org.w3c.dom.Element`, caso os dados de retorno estejam estruturados como XML. `Element`

[3] *wsimport* é uma ferramenta disponível nas distribuições JAX-WS e *Glassfish* que, a partir de um WSDL local ou acessível através de uma Universal Resource Locator (URL), gera artefatos Java portáveis tanto para implantação (*e.g.* fontes ou *.class* de interfaces de serviços) quanto para acesso (*e.g.* fontes ou *.class* de mapeamentos JAXB dos tipos declarados no WSDL e classes de acesso identificados com a anotação `javax.xml.ws.WebServiceClient`). Neste trabalho, *wsimport* foi utilizado para gerar automaticamente clientes de acesso a serviços implantados.

é uma interface para navegação em documentos XML. Para os demais casos o tipo do retorno é `java.lang.String`. Já a operação `startMediationReturnMessage()` não declara retorno.

Alguns serviços Web utilizam retorno via arquivos em anexos, que na especificação de serviços Web publicada pela The World Wide Web Consortium (W3C) é tratada pelo padrão *SOAP with Attachments* [21]. A implementação de mensagens SOAP em Java provê acesso a anexos através do próprio objeto `SOAPMessage`, através de um objeto que encapsula esta informação, do tipo `javax.xml.soap.AttachmentPart`. A partir de `AttachmentPart`, o conteúdo binário do anexo pode ser obtido via `java.io.InputStream`. A operação `startMediatonReturnMessage()` visa justamente prover suporte para serviços que utilizam anexos. O serviço de execução retorna à aplicação-cliente a mensagem original retornado pelo serviço-alvo, que pode conter anexos. Para que a aplicação-cliente consiga trabalhar em nível de mensagem SOAP, e assim consumir tanto o *payload* quanto os anexos, o componente JAX-WS oferece em sua API as classes do pacote `javax.xml.ws`, com destaque para a interface `Dispatch` e para a classe `Service`. A classe `Service` encapsula informações sobre um ponto de acesso a um serviço Web, e é também a fábrica para objetos com comportamento `Dispatch`. Um objeto `Dispatch` é responsável por enviar mensagens SOAP para um serviço e retornar a mensagem SOAP de resposta.

4.2 Exemplo de Uso da Arquitetura Mediador

O projeto arquitetural e o projeto detalhado apresentado até o momento ofereceram uma visão estrutural da infra-estrutura Arquitetura Mediador. A visão dinâmica, contudo, é um artefato valioso para documentar tanto a utilização das interfaces de programação quanto disponibilizar um exemplo para a construção de aplicações que façam uso efetivo da arquitetura proposta.

Aplicações-clientes podem ser construídas usando diferentes bibliotecas, além da diversidade de plataformas e linguagens de programação. Por exemplo, duas plataformas Java para construir código de acesso a serviços Web, e que são abordadas neste trabalho, são os componentes *Apache Axis* [34] e JAX-WS [7], sendo esta última de fato uma implementação de referência da especificação de mesmo nome. Para ambos componentes, a construção de uma aplicação-cliente que usa os serviços da Arquitetura Mediador pode considerar pelo menos quatro opções:

Opção 1: A aplicação-cliente pode acessar os serviços Web da infra-estrutura Arquitetura Mediador utilizando *proxies*, gerados por ferramentas como o *wsimport*, que abstraem os tipos XML trafegados e informações de ponto de acesso remoto.

Opção 2: A aplicação-cliente pode acessar os serviços Web da infra-estrutura Arquitetura

Mediador usado a abordagem *dispatch*, através de um conjunto de classes utilitárias, tais como as classes da interface de programação JAX-WS contidas no pacote `java.xml.ws.*`. Nesta abordagem, as informações de tipos XML são em grande parte diretamente manipuladas e informações de ponto de acesso podem ficar acopladas ao código-fonte da aplicação.

Opção 3: A aplicação-cliente pode apenas consumir o *payload* das mensagens de resposta.

Opção 4: A aplicação-cliente pode consumir o *payload* e também os anexos enviados com as mensagens de resposta.

Os serviços Web *MediatorManager* e *MediatorServer* são adequados tanto para *opção 1* quanto para *opção 2*, e apenas informações são trafegadas no corpo das mensagens, como sugerida pela *opção 3*. Como visto na Seção 4.1.3 sobre o serviço *MediatorExecution*, as *opções 1* e *2* são convenientes para, respectivamente, acessar as operações `startMediationReturnPayload()` e `startMediationReturnMessage()`. Ainda sobre *MediatorExecution*, o fato da aplicação-cliente utilizar a *opção 3* ou *4* depende apenas dos serviços-alvo configurados, que também pode induzir a utilizar *opções 1* ou *2*.

A Figura 4.2 mostra um diagrama de seqüência em UML, para um cenário completo de interação entre uma aplicação-cliente implementado em Java e os serviços da infraestrutura Arquitetura Mediador. Este cenário envolve configurar, disparar e processar a resposta da mediação de serviços Web. Adicionalmente, este cenário procura ilustrar as diferentes formas de utilização dos serviços, como descrito pela *opção 1*, *opção 2*, *opção 3* e *opção 4*.

Todos os objetos ilustrados no diagrama da Figura 4.2 são instanciados no ambiente Java Virtual Machine (JVM) da aplicação-cliente. Em uma implementação deste cenário, outros objetos envolvidos foram omitidos do diagrama por simplicidade. O objeto `Cliente` representa apenas uma instância da aplicação que utiliza a Arquitetura Mediador. Os tipos `MediatorServer`, `MediatorManager`, `MediatorServer_Services` e `MediatorManager_Services` são exemplos de classes geradas pela ferramenta *wsimport*. Os dois primeiros tipos são fábricas para *proxies* dos serviços *MediatorServer* e *MediatorManager*, respectivamente. Já os dois últimos tipos são interfaces Java mapeadas para interfaces de serviço Web de mesmo nome, estando este serviços Web implantados no servidor. O estereótipo *proxy* enfatiza que são pontos de acesso a serviços remotos. Os tipos `Service`, `JAXBContext` e `Service` são as classes utilitárias oferecidos por JAX-WS para trabalhar em nível de mensagem SOAP, com a finalidade de requisitar o serviço *MediatorExecution*. A seguir está a descrição passo-a-passo das mensagens numeradas do diagrama:

1. O primeiro passo é configurar a instância de Arquitetura Mediador. Para isso é necessário obter uma referência do serviço de configuração. A mensagem 1 é a criação

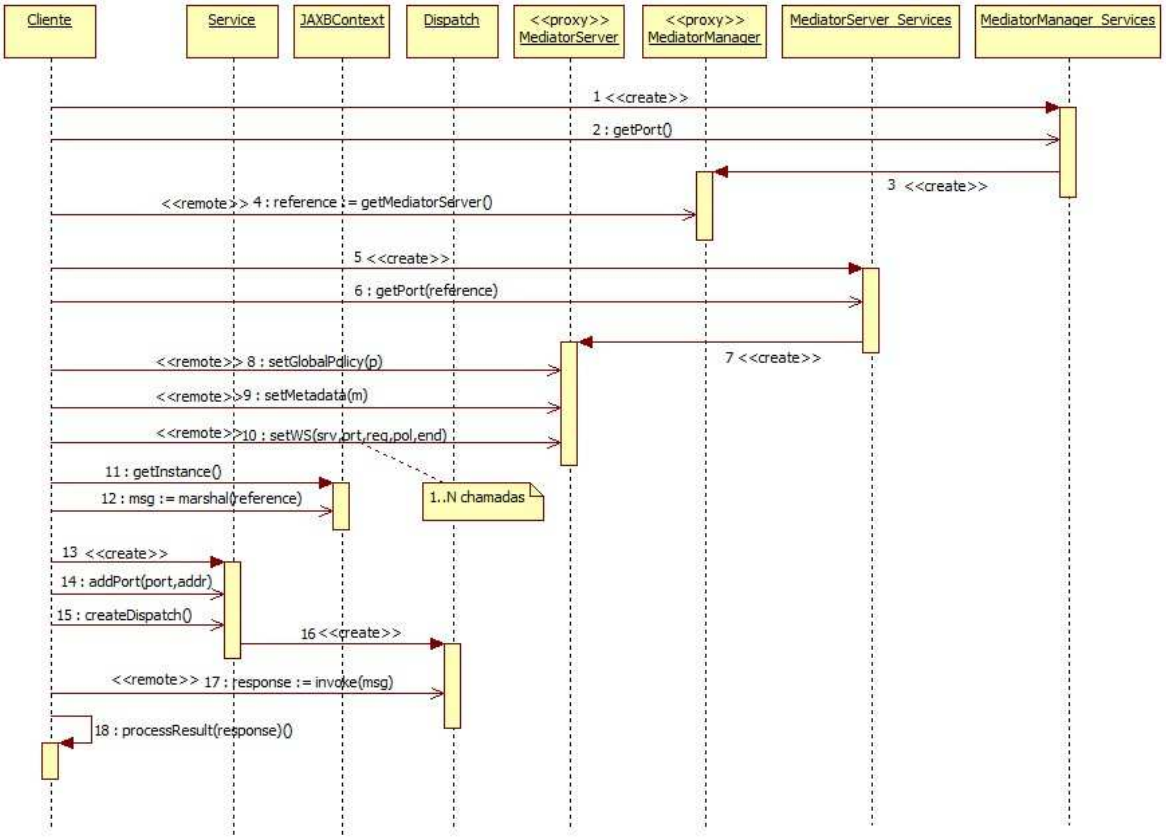


Figura 4.2: Diagrama de seqüência da comunicação remota entre uma aplicação-cliente e a Arquitetura Mediador.

da fábrica para obter `MediatorManager_Services`, representado pelo estereótipo *create*. A mensagem 2 chama método `getPort()` para criar uma instância do tipo *MediatorManager* (passo 3). O método `getMediatorServer()` é requisitado no passo 4, e o estereótipo *remote* indica que o efeito desta chamada é uma mensagem remota ao serviço Web. O retorno do tipo `W3CEndpointReference` é representado pela variável `reference`.

2. De posse da referência, o próximo passo da aplicação-cliente é obter um objeto para acessar o serviço de configuração. Os passos 5, 6 e 7 são análogos à seqüência para acessar o objeto `MediatorManager`. O resultado é a criação do objeto `MediatorServer`. As chamadas para métodos remotos se sucedem nas mensagens 8, 9 e 10 para configuração das políticas de execução, métodos `setGlobalPolicy()` e `setMetadata()`, e para os serviços-alvo da mediação, método `setWS()`. O passo 10 pode se repetir caso haja um conjunto de serviços candidatos.

3. A referência ao serviço remoto obtida no passo 1 é o principal insumo para permitir o disparo da mediação. Esta tarefa é alcançada neste cenário através da abordagem *dispatch*, que envolve manipulação das mensagens de requisição e resposta. A classe que define a mensagem de requisição foi gerada pelo *wsimport*, e esta classe já está preparada para encapsular a referência da configuração e para o processo de serialização JAXB. O objeto do tipo `javax.xml.bind.JAXBContext` é o principal objeto utilizado para a serialização. Uma instancia de `JAXBContext` é obtida no passo 11, via `getInstance()`. Em seguida, no passo 12, é construída a mensagem SOAP de requisição a partir do resultado gerado do método `marshal()`, que recebe como argumento do método a variável `reference`. Apesar de não retratado no diagrama deste passo, um objeto do tipo `org.w3c.dom.Document` é usado como um contêiner para o conteúdo serializado.

4. Criada a mensagem de requisição do serviço, resta à aplicação-cliente efetivamente disparar a Arquitetura Mediador, que consiste em acessar o serviço *MediatorExecution*. A mensagem 13 cria o objeto Service. A mensagem 14 é direcionada ao método `addPort()`, que adiciona as informações de ponto de acesso do *MediatorExecution*. Por exemplo, um ponto de acesso implantado localmente seria representado pela URL `http://localhost:8080/mediator/MediatorExecution`. Portas, no conceito de *portType* em WSDL [22], podem ser adicionadas e referenciadas ao criar um objeto do tipo `Dispatch`. A mensagem 15 chama o método `createDispatch()`, cuja instância `Dispatch` retornada no passo 16 já está preparada para requisitar o serviço de mediação. A mensagem 17 chama `invoke()`, passando o objeto `msg` resultante da serialização, que por sua vez resulta em uma chamada remota para a Arquitetura Mediador. O resultado, representado pela variável `response`, é passada para qualquer processamento implementado pela aplicação-cliente (passo 18).

4.3 Resumo

Este capítulo apresentou o projeto detalhado da infra-estrutura Arquitetura Mediador. Foi detalhado a interface de programação dos três serviços Web para configuração e acesso à infra-estrutura. Aspectos de implementação para cada uma das classes que realizam os serviços Web foram abordados de forma sucinta, apresentando as dependências e decisões de projeto que resultaram na implementação. O projeto detalhado resultante foi ilustrado em um diagrama de classes em UML, composto das interfaces Web, classes de implementação e dependências. Foi apresentado como aplicações podem ser construídas para utilizarem a infra-estrutura proposta. Com o auxílio das principais bibliotecas Java

para construção de clientes de serviços Web, é possível obter o resultado da mediação tanto acessando o conteúdo das mensagens (*payload*) como documentos de anexo. A apresentação da interação entre um cliente hipotético e as interfaces remotas no formato de diagrama de seqüência em UML foi importante para mostrar a ordem de utilização dos serviços e operações oferecidas. Para validação da solução, um estudo de caso no domínio da biodiversidade, baseado em serviços Web, será conduzido de forma similar à dinâmica de utilização mostrada neste capítulo.

Capítulo 5

Estudos de Caso: Projeto BioCORE

O objetivo da condução e extração dos resultados dos estudos de caso é demonstrar a aplicabilidade da infra-estrutura Arquitetura Mediador em um domínio de aplicação real. Os serviços Web que constituem a arquitetura do projeto BioCORE apresentam o perfil de interoperabilidade [43], onde o atributo de qualidade de dependabilidade pode ser provido pela Arquitetura Mediador. Desta forma, este trabalho complementa os demais trabalhos de implementação no projeto BioCORE, sendo que estes últimos enfocam em requisitos funcionais que permitem aos biólogos utilizarem a infra-estrutura computacional em pesquisas e compartilhamento dos resultados com a comunidade em geral. Portanto, o objetivo dos estudos de caso é demonstrar uma solução integrada à infra-estrutura presente, de forma a apresentar os requisitos não-funcionais de dependabilidade necessários ao sucesso das pesquisas apoiadas pelo BioCORE [13].

A Seção 5.1 apresenta o serviço Web de ontologias *Aondê* [28]. Os serviços Web Aondê serão utilizados nos estudos de caso deste capítulo. A Seção 5.2 apresenta o primeiro estudo de caso da Arquitetura Mediador, empregando o serviço de *Consulta* do Aondê, e a estratégia de tolerância a falhas *blocos de recuperação*. O estudo de caso 1 é organizado nos seguintes tópicos: *(i)* descrição do estudo de caso 1 e estrutura física, *(ii)* planejamento do estudo de caso 1, com as configurações e dados de domínio adotados, *(iii)* execução do estudo de caso 1, com a seqüência de execução e resultados esperados e, *(iv)* resultados obtidos do estudo de caso 1, com a análise da bateria de execuções. Por fim, a Seção 5.3 apresenta o segundo estudo de caso da Arquitetura Mediador, empregando o serviço de *Busca e Ranking* do Aondê e a estratégia de tolerância a falhas *N-versões*. O estudo de caso 2 é organizado nos seguintes tópicos: *(i)* descrição do estudo de caso 2 e estrutura física, *(ii)* planejamento do estudo de caso 2, com as configurações e dados de domínio adotados, *(iii)* execução do estudo de caso 2, com a seqüência de execução e resultados esperados e, *(iv)* resultados obtidos do estudo de caso 2, com a análise da bateria de execuções.

5.1 Serviço Web de Ontologias Aondê

O serviço Web *Aondê*, proposto por J. Daltio [28], é componente integrante do projeto BioCORE. O papel desempenhado pelo Aondê dentro da arquitetura do BioCORE é prover funcionalidades de acesso, análise e integração de ontologias no domínio de biodiversidade. Diante do requisito de integrar dados heterogêneos e distribuídos, Aondê foi proposto como um serviço Web que disponibiliza ontologias que podem estar em repositórios distribuídos. A interface de acesso do repositório foi especificada como um conjunto de serviços Web. Possíveis clientes deste serviço de ontologias são sistemas de software de apoio a especialistas, ou mesmo *websites* educacionais e museus virtuais.

Um repositório do Aondê pode conter várias ontologias, unicamente identificadas por um nome. Considerando vários repositórios, as ontologias passam a ser identificadas pelo nome e a URL do repositório. São oferecidos na interface Web do Aondê, operações para manipulação de ontologias cadastradas e consultas avançadas sobre ontologias. Para execução dos estudos de caso da Arquitetura Mediador, foram selecionados duas operações: operação de Consulta (*Query*), e a operação de *Busca e Ranking* (*SearchRank*).

5.2 Estudo de Caso 1: Mediação da Operação Aondê de Consulta

5.2.1 Descrição do Estudo de Caso 1

A operação de Consulta (*Query*), do serviço de Operações (*WSOperationsService*), é a operação usada no estudo de caso 1. A operação de Consulta permite a extração de informações armazenadas em uma ontologia. Uma consulta, em formato SPARQL, é executada para uma ontologia. O resultado da consulta contém elementos como classes, propriedades e instâncias, que casam com os termos da consulta. Aplicado ao domínio de biodiversidade, informações taxonômicas de classificação dos seres vivos são exemplos de elementos retornados pela operação Consulta.

5.2.2 Planejamento do Estudo de Caso 1

A Figura 5.1 mostra o diagrama de distribuição dos componentes envolvidos no estudo de caso 1. O serviço de Operações é, neste trabalho, requisitado por duas instâncias do Aondê. O primeiro Aondê é instanciado localmente no microcomputador chamado *localhost*, juntamente com uma instância da Arquitetura Mediador. A plataforma Web escolhida para hospedar estas instâncias é o servidor de aplicação Java *Glassfish V2* [2]. Já

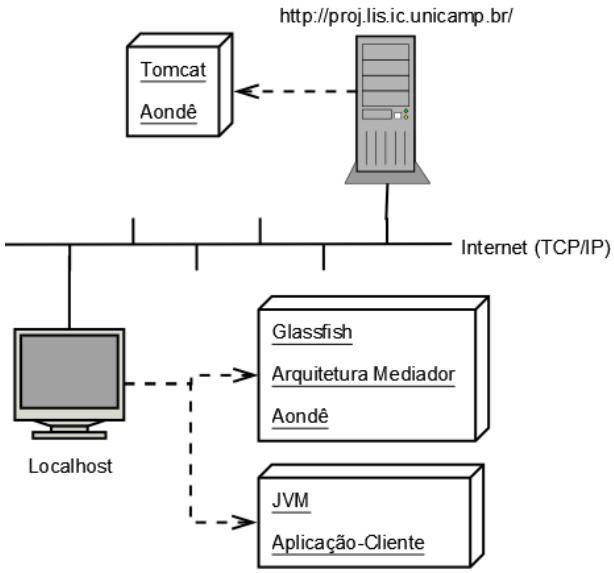


Figura 5.1: Estrutura física de distribuição dos serviços Aondê, para o estudo de caso 1.

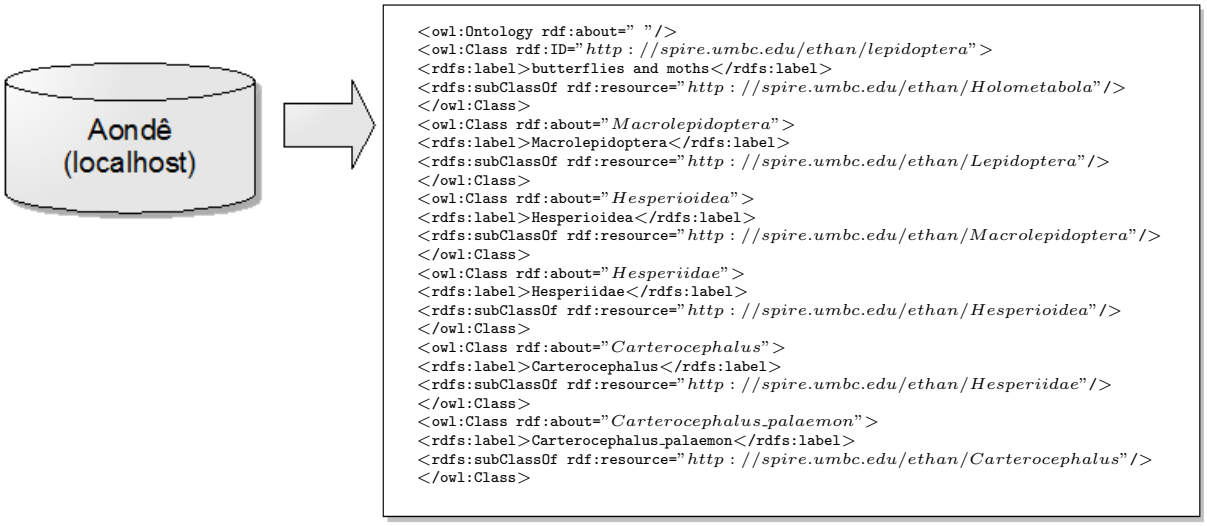


Figura 5.2: Ontologia cadastrada no repositório do Aondê em *localhost*.

o segundo repositório é instanciado no servidor de aplicação LIS-UNICAMP, no endereço `http://proj.lis.ic.unicamp.br/tomcat/aonde/services/WSSemanticRepository`.

Os repositórios utilizados no estudo de caso 1 fazem o papel de réplicas funcionalmente idênticas. Adicionalmente, os repositórios mantêm ontologias distintas. Desta forma, os repositórios simulam a situação de redundância que propicia a abordagem de tolerância a falhas por meio de blocos de recuperação (ou N-versões), empregadas na Arquitetura Mediador, se tornem efetivas. As Figuras 5.2 e 5.3 esquematizam os repositórios


```
<owl:Ontology rdf:about=" "/>
<owl:Class rdf:ID="http://spire.umbc.edu/ethan/lepidoptera">
<rdfs:label>butterflies and moths</rdfs:label>
<rdfs:subClassOf rdf:resource="http://spire.umbc.edu/ethan/Holometabola"/>
</owl:Class>
<owl:Class rdf:about="Macrolepidoptera">
<rdfs:label>Macrolepidoptera</rdfs:label>
<rdfs:subClassOf rdf:resource="http://spire.umbc.edu/ethan/Lepidoptera"/>
</owl:Class>
<owl:Class rdf:about="Hesperioidea">
<rdfs:label>Hesperioidea</rdfs:label>
<rdfs:subClassOf rdf:resource="http://spire.umbc.edu/ethan/Macrolepidoptera"/>
</owl:Class>
<owl:Class rdf:about="Hesperiidae">
<rdfs:label>Hesperiidae</rdfs:label>
<rdfs:subClassOf rdf:resource="http://spire.umbc.edu/ethan/Hesperioidea"/>
</owl:Class>
<owl:Class rdf:about="Thymelicus">
<rdfs:label>Thymelicus</rdfs:label>
<rdfs:subClassOf rdf:resource="http://spire.umbc.edu/ethan/Hesperiidae"/>
</owl:Class>
<owl:Class rdf:about="Thymelicus_sylvestris">
<rdfs:label>Thymelicus_sylvestris</rdfs:label>
<rdfs:subClassOf rdf:resource="http://spire.umbc.edu/ethan/Thymelicus"/>
</owl:Class>
```

Figura 5.3: Ontologia cadastrada no repositório do Aondê em LIS-UNICAMP.

e as ontologias cadastradas. As ontologias em formato Ontology Web Language (OWL) foram obtidas do serviço disponível na Internet chamado *Spire*[1]. Ambos repositórios mantêm ontologias taxonômicas da família das *hesperídeas*[2]. De acordo com o *website UK Butterflies*[3], as espécies de hesperídeas, um tipo de borboleta[4], também são conhecidas como "skippers". O repositório do Aondê *localhost* mantém a classe OWL para descrever a espécie *palaemon*, do gênero *Carterocephalus*[5]. Já o repositório no LIS-UNICAMP mantém a classe OWL para descrever a espécie *sylvestris*, do gênero *Thymelicus*[6].

Dado o planejamento do estudo de caso, o próximo passo é obter uma implementação de aplicação-cliente para configurar e disparar a consulta ao Aondê via Arquitetura Mediador. O projeto detalhado desta aplicação-cliente é similar ao projeto da Seção 4.2, de forma que foram adotadas as recomendações de uso para requisição e processamento de resultados provenientes da Arquitetura Mediador. A Tabela 5.1 discrimina as configurações enviadas à Arquitetura Mediador para a execução deste estudo de caso.

5.2.3 Execução do Estudo de Caso 1

Durante a execução do estudo de caso 1, foram enviados ao serviço de Consulta do Aondê consultas em formato SPARQL [27]. A Figura 5.4 mostra como essa consulta requisita

[1]Website: `http://spire.umbc.edu/ont/ethan.php`
[2]Nome científico, *Hesperiidae*.
[3]Website: `http://www.ukbutterflies.co.uk/species_family.php?name=Hesperiidae`
[4]Nome científico, *Lepidoptera*.
[5]Nome vulgar em inglês, *Chequered Skipper*.
[6]Nome vulgar em inglês, *Small Skipper*.

s

Tabela 5.1: Configurações Mantidas pela Arquitetura Mediador para o estudo de caso 1.

Serviços	URL Destino
Aondê (localhost)	`http://localhost:8080/aonde/services/WSOperationsService`
Aondê (LIS-UNICAMP)	`http://proj.lis.ic.unicamp.br/tomcat/aonde/services/WSOperationsService`

Configurações		Aondê (localhost)	Aondê (LIS-UNICAMP)
Políticas Individuais	*timeout* (ms)	60000	60000
	Tentativas após falhas	2	2
	Intervalo de tentativa (ms)	3000	3000
Metadados de Dependabilidade	Fator de dependabilidade	50	10
	Performance (ms)	100	300
	Resposta mínima (ms)	10	10
	Resposta máxima (ms)	200	300
Políticas Globais	Modo de execução	Bloco de recuperação	
	Núm. serviços candidatos	2	
	Aceitação dependabilidade	80	
	Aceitação performance (ms)	300	

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX ont:<http://dummy-ontologies.com/dummy.owl#>
SELECT ?subclass ?superclass
WHERE {?subclass rdfs:subClassOf ?superclass .?superclass rdfs:label ?label .
FILTER (?label = "Hesperiidae")}
```

Repositório Aondê *localhost*
Carterocephalus palaemon

Repositório Aondê LIS-UNICAMP
Thymelicus sylvestris

Figura 5.4: Consulta submetida aos serviços Web Aondê e as espécies retornadas são *Carterocephalus palaemon* e *Thymelicus sylvestris*, cujas ilustrações estão dispostas abaixo da consulta.

todas as espécies que são subclasses de *hesperídeas*. Dadas as ontologias cadastradas em cada repositório utilizado neste estudo de caso, a Figura 5.4 ilustra qual espécie é retornada em cada caso[7].

Pela Tabela 5.1, item *modo de execução*, a estratégia de tolerância a falhas selecionada é a de blocos de recuperação. A Figura 5.5 é um diagrama de seqüência em UML que reflete a estratégia adotada, materializada na seqüência das mensagens trocadas entre

[7]Ilustrações de hesperídeas retiradas do *website UK Butterflies*.

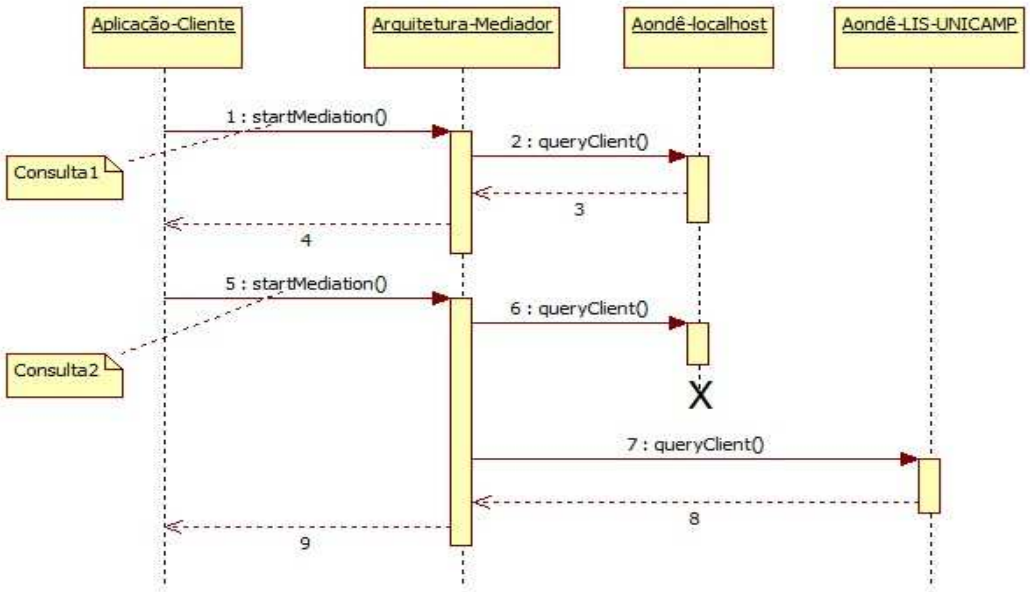


Figura 5.5: Diagrama de seqüência entre Arquitetura Mediador e instâncias do Aondê em UML, para o estudo de caso 1.

a aplicação-cliente, Arquitetura Mediador e instâncias do Aondê. Este cenário mostra apenas duas iterações de requisição da operação Consulta, onde uma requisição é enviada para o repositório cujo metadado *fator de dependabilidade* indica maior confiabilidade: Aondê *localhost*. Neste caso, a requisição chamado de *Consulta1*, é atendida. Em seguida, para uma requisição chamada *Consulta2* efetuada em um momento posterior, há uma mudança de curso após a queda do serviço Web Aondê *localhost*. Esta queda é propositalmente efetuada para fins de validação da reconfiguração dinâmica da Arquitetura Mediador. Desta maneira, é selecionado, de forma transparente para a aplicação-cliente, o próximo serviço com maior dependabilidade no bloco de recuperação: serviço Web em LIS-UNICAMP. Neste caso, o cliente recebe uma resposta que atende à consulta mesmo na presença de uma falha.

Mantendo os dados dos repositórios distintos entre si ao longo das iterações para uma mesma consulta, o cliente acabará por receber resultados diferentes. Por exemplo, se a aplicação-cliente aplicada no cenário da Figura 5.5 tivesse como requisito exibir as ilustrações das espécies retornadas, a *Consulta1* exibiria a ilustração de *Carterocephalus palaemon* conforme Figura 5.4. Igualmente, a *Consulta2* terminaria exibindo a ilustração de *Thymelicus sylvestris*. Portanto, é importante salientar que este comportamento é exclusivamente devido à escolha dos repositórios funcionalmente idênticos, mas que podem ser mantidos independentes.

Esta proposta está relacionada ao cenário esperado no contexto do projeto BioCORE, onde instâncias do repositório mantidos por diferentes grupos de pesquisa, institutos, universidades, etc., podem atender a uma consulta de forma diferente. Estes repositórios seriam os componentes fundamentais em uma arquitetura orientada a serviços voltada à pesquisa em biodiversidade.

Por fim, na aplicação-cliente construída para a execução do estudo de caso 1 da infraestrutura Arquitetura Mediador, mediando requisições para o serviço Web de ontologias Aondê, foi implementado o seguinte algoritmo: após a configuração inicial, uma mesma requisição ao serviço de Consulta é enviada de 2 em 2 minutos, em uma bateria de testes que dura 40 minutos, totalizando 20 requisições enviadas.

5.2.4 Discussão dos Resultados Obtidos

A metodologia utilizada para apresentar a aplicabilidade da Arquitetura Mediador é ilustrar como as informações de dependabilidade e reconfiguração dinâmica atuam para tornar o serviço provido mais confiável. A Figura 5.6 mostra um gráfico que ilustra a dinâmica da mediação confiável sob a perspectiva do resultado recebido pela aplicação-cliente. Neste gráfico, o eixo horizontal mostra as execuções efetuadas para uma mesma configuração da Arquitetura Mediador em ordem cronológica. Já o eixo vertical mostra o tempo de resposta entre o envio da mensagem de uma requisição e a respectiva resposta. A legenda à direita mostra os serviços candidatos do cenário, a linha de execução e o resultado final da requisição do serviço, isto é, mensagem de falha ou sucesso, para cada execução, disposta na parte superior do gráfico.

Entre a execução 0 até 6, o serviço hospedado no servidor *localhost* foi escolhido para atender à requisição, de forma que os resultados foram todos válidos, isto é, uma mensagem com as espécies que atendam à consulta foram retornadas. Para a execução 7, foi desabilitada o serviço Aondê *localhost*[8], de forma que a respectiva requisição foi enviada ao serviço Aondê-LIS. As requisições subseqüentes até execução 13 foram resultados válidos, contendo as espécies que estavam cadastradas neste repositório. Uma observação sobre a latência apresentada no gráfico nas execuções 7 a 13 é que estão incluídas as sobrecargas introduzidas pelas configurações *tentativas após falhas* (2 tentativas) e *intervalo de tentativa* (3 segundos), conforme Tabela 5.1. Nas execuções 14, 15 e 16, o serviço *localhost* ainda se manteve desabilitado, e adicionalmente a conexão com a Internet foi interrompida. O resultado, portanto, foi uma mensagem de falha gerada pela Arquitetura Mediador informando a indisponibilidade de comunicação com todos os serviços. Por fim, as execuções 17 até 19 refletiram o retorno do serviço Aondê *localhost*.

Conforme mostrado no gráfico da Figura 5.6, em uma amostra de 20 requisições,

[8]Desabilitada manualmente via console de administração do servidor Glassfish.

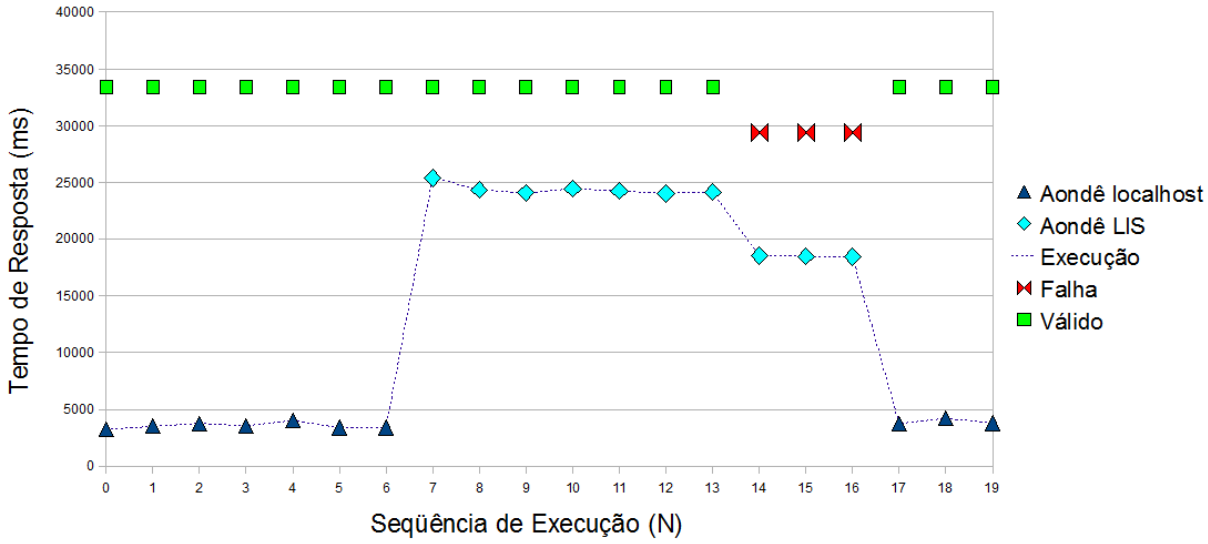


Figura 5.6: Bateria de execuções para o estudo de caso 1.

apenas 3 não foram atendidas, resultando em uma disponibilidade de *85%*. Ainda considerando o conjunto das 20 requisições, para 10 delas algum serviço estava indisponível. Portanto, considerando todos os componentes da arquitetura da prova de conceito, em *50%* do tempo o sistema operou com falhas em pelo menos um dos componentes. O tempo de resposta médio durante todo o estudo de caso, considerando sucessos e falhas, foi de *13,1* segundos. Supondo um período de operação longo, este tempo médio poderia ser melhorado se o componente WS-Mediator, que é responsável pela implementação do algoritmo de mediação confiável da Arquitetura Mediador, atualizasse os metadados dos serviços configurados. O reflexo desta atualização é a determinação dinâmica de qual serviço seria requisitado em cada interação.

5.3 Estudo de Caso 2: Mediação da Operação Aondê de Busca e Ranking

5.3.1 Descrição do Estudo de Caso 2

A operação de *Busca e Ranking* (*SearchRank*), do serviço de Operações (*WSOperationsService*), é a operação usada no estudo de caso 2. A operação de *Busca e Ranking* retorna, a partir de um conjunto de repositórios, ontologias que possuam classes ou instâncias cujos nomes casam (exata ou parcialmente) com os termos buscados. Considerando ontologias que descrevem informações de biodiversidade, exemplos de termos buscados podem ser táxons situados em qualquer nível de um sistema de classificação (*i.e.* reinos, gêneros e

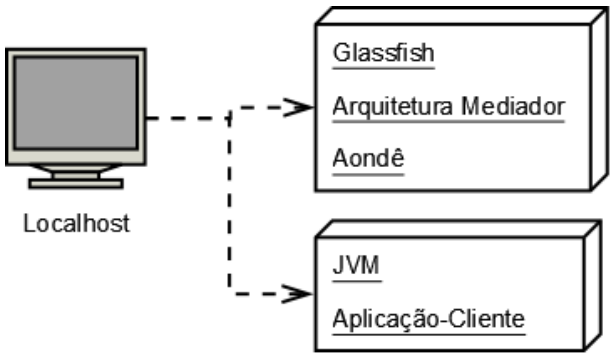


Figura 5.7: Estrutura física de distribuição dos serviços Aondê, para o estudo de caso 2.

espécie), de forma a designar univocamente um tipo ou um grupo de seres vivos. Exemplos de táxons são: *Animalia* (reino), *Arthropoda* (filo) e *Lepidoptera* (ordem). Adicionalmente, a operação de *Busca e Ranking* emprega métricas de *ranking*, com argumentos como casamento exato ou parcial, densidade, centralidade e similaridade semântica. O retorno desta operação é no formato [Nome da ontologia; URL do repositório; ranking]. As ontologias retornadas estão ordenadas pela métrica de *ranking*, isto é, a ordem da ontologia que apresenta maior similaridade com o táxon informado.

5.3.2 Planejamento do Estudo de Caso 2

A Figura 5.7 mostra o diagrama de distribuição dos componentes envolvidos no estudo de caso 2. O serviço de *Busca e Ranking* é requisitado por duas instâncias do serviço Web Aondê, ambas instanciadas localmente no microcomputador utilizado (*localhost*), juntamente com uma instância da Arquitetura Mediador. A plataforma Web escolhida para hospedar estas instâncias é o servidor de aplicação Java *Glassfish V2* [2]. Cada instância do Aondê mantém o próprio repositório de ontologias, e a mensagem de requisição do serviço instrui para que o serviço execute e busque as ontologias no próprio repositório. Por exemplo, o serviço Web, cuja instância é chamada neste estudo de caso de *Aondê-localhost-1*, buscará as ontologias no próprio repositório da instância. O mesmo caso acontece para o serviço Web Aondê cuja instância é chamada *Aondê-localhost-2*.

A Figura 5.8 esquematiza os repositórios e as suas ontologias cadastradas. Esta configuração visa a aproximação do cenário onde diferentes repositórios de ontologias são mantidos por grupos de pesquisa do Instituto de Biologia da UNICAMP. Neste caso, a hipótese é que cada repositório armazena coleções do Museu de Zoologia IB-UNICAMP[9].

[9]Exemplos de coleções foram extraídas de informações do protótipo do *website* Museu de Zoologia, elaborado no contexto do projeto BioCORE. Este protótipo pode ser acessado pelo *website* `http://proj.lis.ic.unicamp.br/biocore/collections/`

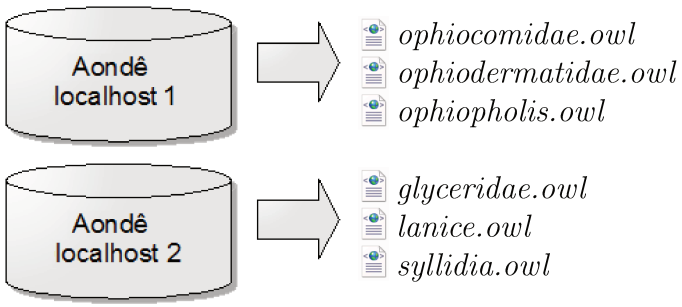


Figura 5.8: Ontologias cadastradas nos repositórios do Aondê instanciados na máquina *localhost*.

Tabela 5.2: Configurações Mantidas pela Arquitetura Mediador para o estudo de caso 2.

Serviços	URL Destino
Aondê localhost 1	`http://localhost:8080/aonde1/services/WSOperationsService`
Aondê localhost 2	`http://localhost:8080/aonde2/services/WSOperationsService`

Configurações		Aondê 1	Aondê 2
Políticas Individuais	*timeout* (ms)	60000	
	Tentativas após falhas	2	
	Intervalo de tentativa (ms)	3000	
Metadados de Dependabilidade	Fator de dependabilidade	50	
	Performance (ms)	100	
	Resposta mínima (ms)	10	
	Resposta máxima (ms)	200	
Políticas Globais	Modo de execução	N-versão	
	Processamento resultados	Votação	
	Aceitação dependabilidade	80	
	Aceitação performance (ms)	300	

O repositório da instância do Aondê chamada *Aondê-localhost-1* armazena ontologias da classe *Ophiuroidea*, filo *Echinodermata*. Já o repositório da instância chamada *Aondê-localhost-2* armazena ontologias da classe *Polychaeta*, filo *Annelida*. As ontologias em formato OWL foram obtidas do serviço disponível na Internet chamado *Spire*[10]. Os nomes das ontologias apresentadas na Figura 5.8 representam famílias, que por sua vez contêm exemplos de espécies (*i.e.* classes OWL), atualmente catalogadas pelo Spire.

Dado o planejamento do estudo de caso, o próximo passo é obter uma implementação de aplicação-cliente para configurar e disparar a consulta ao Aondê via Arquitetura Mediador. O projeto detalhado desta aplicação-cliente é similar ao projeto da Seção 4.2, de forma que foram adotadas as recomendações de uso para requisição e processamento de resultados provenientes da Arquitetura Mediador.

[10]Website: `http://spire.umbc.edu/ont/ethan.php`

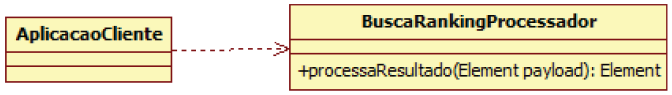


Figura 5.9: Diagrama de classe em UML que mostra a dependência entre a aplicação-cliente e um conector de votação.

A Tabela 5.2 discrimina as configurações enviadas à Arquitetura Mediador para a execução deste estudo de caso. Pelo item *modo de execução*, a estratégia de tolerância a falhas selecionada é a N-versões, e pelo item *processamento resultados*, os resultados devem ser avaliados por votação. Com base neste parâmetros, a Arquitetura Mediador não retorna somente uma resposta, mas N respostas, sendo N o número de serviços-candidatos. Diante desta configuração, a aplicação-cliente precisa de um conector que implemente uma heurística de tratamento das N respostas obtidas, de forma que uma única resposta que represente o resultado final da mediação seja repassada à aplicação-cliente.

A Figura 5.9 é um diagrama de classes em UML que documenta esta decisão. O conector é implementado pela classe `BuscaRankingProcessador`, que oferece o método `processaResultado()`. Este método recebe um argumento do tipo `org.w3c.dom.Element`, que é o documento encapsulador das N mensagens, e retorna também um `Element`, que é o resultado final da mediação na qual a aplicação-cliente espera de todo o processo. A implementação de um conector de votação pode variar de acordo com o domínio de aplicação, e a respectiva implementação não faz parte do escopo da Arquitetura Mediador, que é independente de domínio. Para a operação de *Busca e Ranking*, uma solução de heurística proposta para processamento dos resultados válidos é combiná-los em uma única busca estendida. Antes desta combinação, os resultados individuais são novamente ordenados pelo *ranking*, de forma a manter o contrato da operação de *Busca e Ranking*.

5.3.3 Execução do Estudo de Caso 2

Durante a execução do estudo de caso, são enviados ao serviço de Consulta do Aondê requisições à operação *Busca e Ranking*. A Figura 5.10 apresenta a chamada desta operação com os argumentos configurados de forma a requisitar a busca em cada repositório de ontologias. Neste caso, o termo buscado é o termo vulgar "animal", com o objetivo de buscar por espécies do reino *Animalia*. A operação *Busca e Ranking*, por sua vez, efetuará uma busca com um termo aproximado. Também é especificado, como segundo parâmetro na chamada da operação, as métricas de similaridades. A execução desta operação nos repositórios populados da Figura 5.8 retorna as ontologias listadas na Figura 5.10. Essas

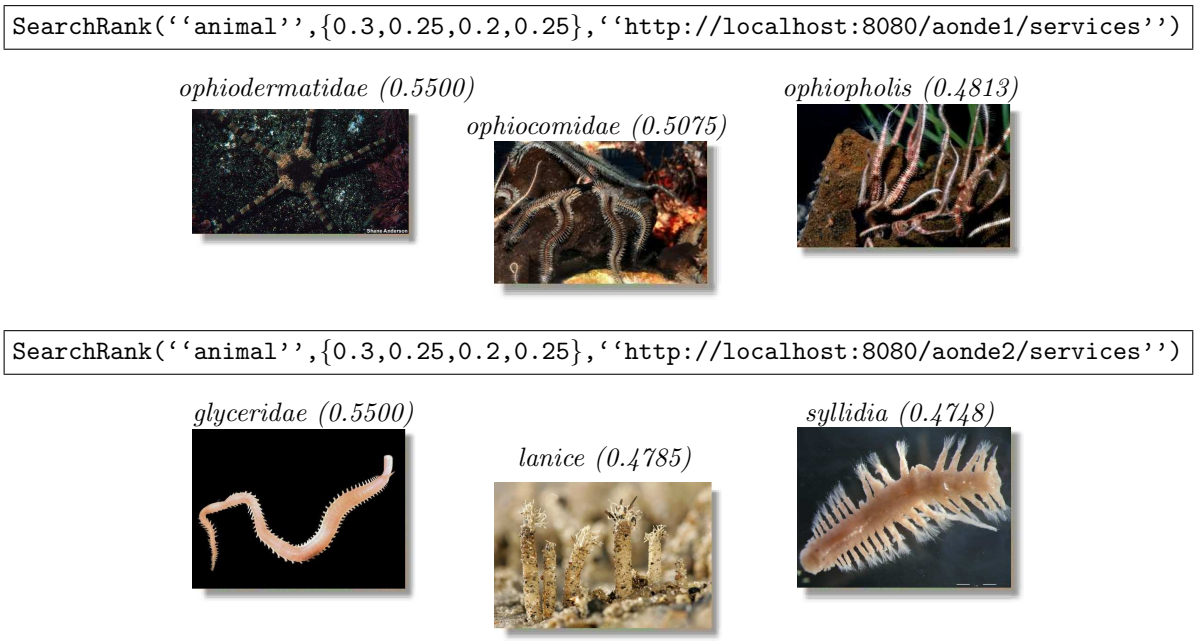


Figura 5.10: Busca com ranking submetida aos serviços Web Aondê e exemplares de espécies descritas pelas ontologias, cujos nomes das respectivas famílias são retornadas pela operação.

ontologias estão associadas aos respectivos *rankings* calculados e também ilustrações de espécies representativas da família retornada[11].

A Figura 5.11 é um diagrama de seqüência em UML que reflete a estratégia N-versões adotada, materializada na seqüência das mensagens trocadas entre a aplicação-cliente, Arquitetura Mediador, e as instâncias do Aondê. Este cenário também reflete a configuração de votação para o processamento dos resultados. Desta forma, a Arquitetura Mediador dispara requisições simultâneas para os serviços Web configurados (*i.e.* objetos `Aondê-localhost-1` e `Aondê-localhost-2`) e espera uma resposta, válida ou de erro, de todas as instâncias. A notação em UML para mensagens concorrentes é melhor comunicada agrupando-as em um fragmento do tipo **par** [30]. Isto significa que, apesar da notação seqüencial do diagrama, qualquer das mensagens 2 ou 4, associadas às respectivas respostas 3 e 5, podem ser executadas em ordem não determinada. O objeto `Aplicação-Cliente` no diagrama já representa ambos conector de votação e cliente final, conforme Figura 5.9.

Em uma requisição chamada *Consulta1*, ambos serviços Web Aondê respondem à requisição, e os resultados são entregues encapsuladas em uma única mensagem de resposta. Em seguida, para uma requisição chamada *Consulta2* efetuada em um momento posterior,

[11]Ilustrações de espécies de *Echinodermata Ophiuroidea* e *Annelida Polychaeta* foram retiradas do acervo online *EOL Encyclopedia of Life*. Website: `http://www.eol.org/`

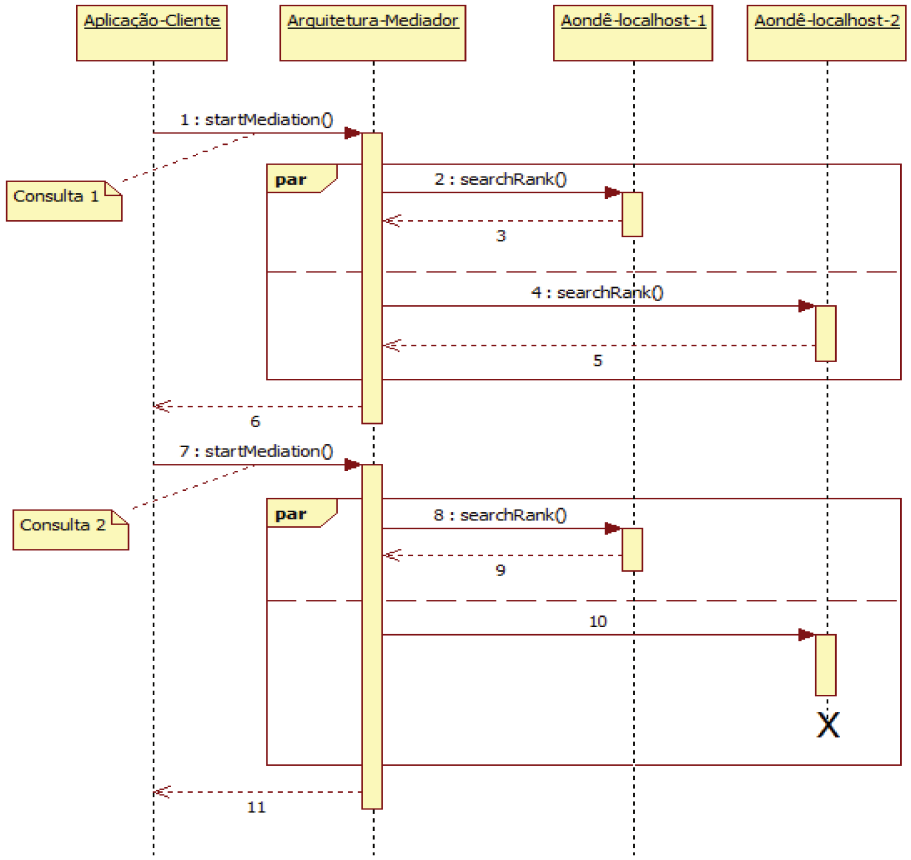


Figura 5.11: Diagrama de seqüência entre Arquitetura Mediador e instâncias do Aondê em UML para o estudo de caso 2.

é efetuada uma queda proposital do serviço Web chamado `Aondê-localhost-2`. Desta maneira, a Arquitetura Mediador seleciona apenas a única mensagem válida e a retorna como um resultado geral da mediação. Neste caso, a aplicação-cliente recebe uma resposta que atende à consulta mesmo na presença de uma falha.

Apesar de não haver um serviço Web acessível remotamente no cenário de execução, espera-se do estudo de caso 2, a apresentação de resultados que mostrem como o modo N-versões, mais a estratégia de votação, podem ser utilizados tanto para prover maior dependabilidade à arquitetura, quanto para aperfeiçoar os resultados disponibilizados pelos serviços. Portanto, considerando o cenário da Figura 5.11, o usuário que efetua a *Consulta1* receberá como resposta tanto as ontologias da classe *Echinodermata Ophiuroidea* quanto da classe *Annelida Polychaeta*. No entanto, um outro usuário que efetua a *Consulta2* receberá apenas as ontologias da classe *Echinodermata Ophiuroidea*.

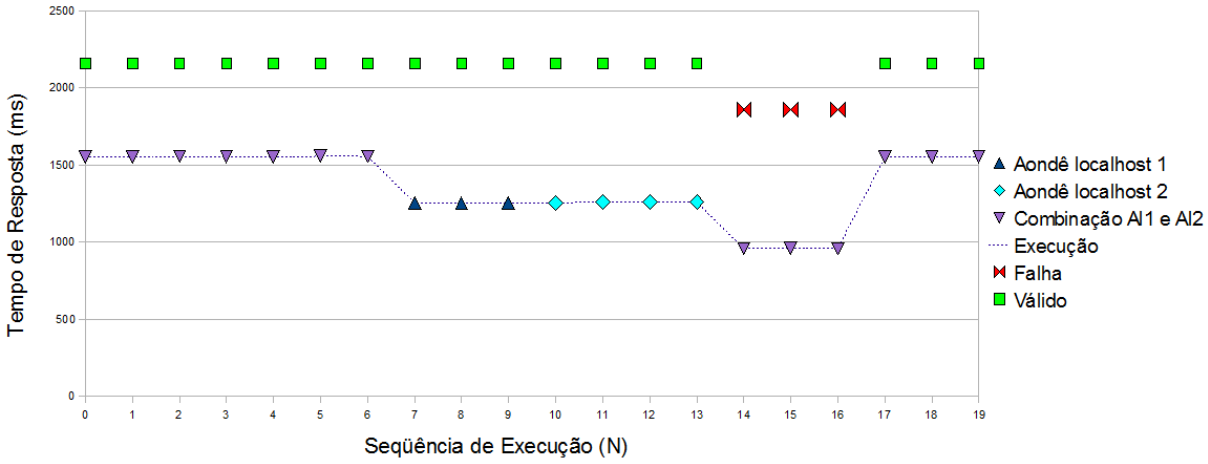


Figura 5.12: Bateria de execuções para o estudo de caso 2.

5.3.4 Discussão dos Resultados Obtidos

Na aplicação-cliente construída para o estudo de caso 2 da infra-estrutura Arquitetura Mediador, mediando requisições para o serviço Web de ontologias Aondê, foi implementado o seguinte algoritmo: após a configuração inicial, uma mesma requisição ao serviço de *Busca e Ranking* é enviada de 1 em 1 minuto, em uma bateria de testes que dura 20 minutos, totalizando 20 requisições enviadas.

A metodologia utilizada para apresentar a aplicabilidade da Arquitetura Mediador é ilustrar como as informações de dependabilidade e a utilização de múltiplas instâncias de um serviço atuam simultaneamente para tornar o serviço provido mais confiável. A Figura 5.12 mostra um gráfico que ilustra a dinâmica da mediação confiável sob a perspectiva do resultado recebido pela aplicação-cliente. Neste gráfico, o eixo horizontal mostra as execuções efetuadas para uma mesma configuração da Arquitetura Mediador em ordem cronológica. Já o eixo vertical mostra o tempo de resposta entre o envio da mensagem de uma requisição e a respectiva resposta. A legenda à direita mostra os serviços candidatos do cenário, a linha de execução e o resultado final da requisição do serviço, isto é, mensagem de falha ou sucesso, para cada execução, disposta na parte superior do gráfico.

Entre a execução 0 até 6, ambos os resultados são combinados em um único resultado final, demonstrado no gráfico pela legenda "Combinação Al1 e Al2" sobre a linha de execução. Para a execução 7 até 9, foi desabilitada o serviço *Aondê localhost 2*[12], de forma que apenas o resultado do *Aondê localhost 1* foi obtido, com as respectivas ontologias cadastradas e que casam com a busca. Um decréscimo no tempo de resposta observado para estes resultados é devido ao tratamento trivial do conector de votação, que simplesmente

[12]Desabilitada manualmente via console de administração do servidor Glassfish.

retorna a única mensagem disponível. Situação similar ocorre para as execuções 10 até 13, onde o serviço disponível é o *Aondê localhost 2*. Nas execuções 14, 15 e 16, o serviço *Aondê localhost 1* ainda se manteve desabilitado, e adicionalmente o serviço *Aondê localhost 2* foi também desabilitado. Neste caso não foi possível combinar os resultados dos serviços-candidatos, e uma mensagem geral de falha foi gerada pela Arquitetura Mediador informando a indisponibilidade de comunicação com todos os serviços. Para estas mensagens, um decréscimo no tempo de resposta em comparação com execuções anteriores é principalmente devido a serviços desabilitados em nível de *container* Web sequer terem sido requisitados. O servidor Glassfish foi, portanto, o responsável por gerar uma mensagem de erro. Por fim, as execuções 17 até 19 refletiram o retorno dos serviços *Aondê localhost 1* e *Aondê localhost 2*.

Conforme mostrado no gráfico da Figura 5.12, em uma amostra de 20 requisições, apenas 3 não foram atendidas, resultando em uma disponibilidade de *85%*. Ainda considerando o conjuto das 20 requisições, para 10 delas algum serviço estava indisponível. Portanto, considerando todos os componentes da arquitetura da prova de conceito, em *50%* do tempo o sistema operou com falhas em pelo menos um dos componentes. O tempo de resposta médio durante todo o estudo de caso, considerando sucessos e falhas, foi de *1,5* segundos.

Quantitativamente, os resultados de dependabilidade do estudo de caso 2 foram idênticos aos obtidos no estudo de caso 1, contudo a estratégia N-versões é mais sensível a discrepâncias dos metadados de disponibilidade, em comparação com a estratégia bloco de recuperação. Por exemplo, o tempo de resposta total para a estratégia N-versões corresponderá ao serviço Web mais lento. Em contrapartida, a estratégia N-versões facilita o aumento da confiabilidade dos dados recebidos dos serviços Web, supondo a presença e a corretude da implementação do conector de votação. De modo geral, a atualização dos metadados de dependabilidade ao longo do tempo de operação da Arquitetura Mediador, operando na estratégia N-versões, servirá para descartar serviços-candidatos que deixarem de atender aos limites mínimos de dependabilidade configurados. Esta funcionalidade de atualização de metadados durante execução ainda não está implementada na versão do WS-Mediator utilizada.

5.4 Resumo

Este capítulo apresentou dois estudos de caso da infra-estrutura Arquitetura Mediador em uma implementação real. O projeto BioCORE define uma arquitetura orientada a serviços Web que oferece serviços interoperáveis, de forma a tornar a sua implementação ideal para os estudos de caso. As operações de *Consulta* e de *Busca e Ranking* do serviço Web de ontologias Aondê foram utilizadas em cenários distintos de mediação confiável,

sempre utilizando dados reais de ontologias sobre taxonomias de seres vivos. No primeiro estudo de caso, foi planejado a utilização do modo de operação *blocos de recuperação*. O objetivo era permitir que instâncias locais e remotas dos repositórios de ontologias do Aondê pudessem ser requisitadas mesmo diante de falhas induzidas. No segundo estudo de caso, foi experimentado o modo de operação *N-versões*. O objetivo era permitir que instâncias locais pudessem colaborativamente construir um resultado dos serviços estendido, e ao mesmo tempo oferecer nível maior de disponibilidade diante de falhas induzidas. A eficácia da Arquitetura Mediador em aumentar a dependabilidade dos serviços Web do projeto BioCORE foi, portanto, validada em baterias de testes e gráficos que ilustraram o comportamento dinâmico dos componentes envolvidos em cada estudo de caso.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Visão Geral

A Arquitetura Orientada a Serviços (SOA) é responsável por mapear os processos de negócios relevantes aos seus serviços correspondentes que, juntos, agregam o valor final ao usuário. A tecnologia de serviços Web, baseado em XML, tem se destacado pela sua qualidade em permitir interoperabilidade entre sistemas através de padrões abertos e grande adoção destes padrões por organizações e fornecedores de produtos. Além dos requisitos mencionados, SOA deve atender aos principais requisitos de dependabilidade. Dependabilidade engloba vários conceitos relacionados com computação segura e confiável, entre elas disponibilidade, confiabilidade, dentre outros atributos. As principais soluções para aumento da dependabilidade considerando serviços distribuídos atuam na camada de comunicação e usufruem da redundância natural de serviços e enlaces na Internet.

Para atender ao requisito de aumento da dependabilidade em arquiteturas orientada a serviços Web, esta dissertação apresentou uma solução de infra-estrutura de software que implementa mediação confiável. Chamada de Arquitetura Mediador, esta solução pode ser integrada a arquiteturas de software no papel de um conector que atua na comunicação entre os cliente e os serviços Web, a fim de executar técnicas de tolerância a falhas que façam uso das redundâncias de serviços disponíveis. A Arquitetura Mediador foi implementada usando tecnologia Java e foi projetada para ser acessível via serviços Web, de forma que o impacto na adoção da infra-estrutura seja minimizada. A Arquitetura Mediador reutilizou um componente para implementação e disponibilização de serviços Web (JAX-WS), e um componente que implementa técnicas de tolerância a falhas em SOA (WS-Mediator). Por ter as suas funcionalidades disponibilizadas como serviços Web, aplicações-clientes podem ser implementadas em qualquer linguagem de programação. Desta forma, basta para estas aplicações apenas depender de bibliotecas de acesso a serviços Web específicas da linguagem escolhida.

As arquiteturas de software conceitual e detalhada da Arquitetura Mediador foram apresentadas, onde foram documentados os componentes, as interfaces e outras decisões de projeto que permearam a implementação. A Arquitetura Mediador oferece em sua interface Web operações para configuração e disparo da mediação confiável. A principal configuração disponível é a seleção da estratégia de tolerância a falhas a ser empregada: blocos de recuperação e N-versões. A estratégia blocos de recuperação visa manter a continuidade do serviço mesmo diante da presença de falhas em um dos componentes, de forma que o componente falho é substituído por um outro componente operacional. Já a estratégia N-versões visa tolerar falhas de projeto e implementação, de forma que os resultados das requisições de múltiplas versões de um componente possam ser combinados ou comparados para prover um serviço mais confiável.

A validação da implementação da Arquitetura Mediador foi feita baseada em estudos de caso utilizando serviços Web implementados no projeto BioCORE. O projeto BioCORE visa apoiar os biólogos nas atividades de pesquisa e manutenção do acervo de informações ecologicamente cientes e biodiversidade. A principal implementação utilizada é o serviço Web de ontologias Aondê, que provê funcionalidades de acesso, análise e integração de ontologias no domínio de biodiversidade. Estas funcionalidades são acessíveis via interfaces em serviços Web, de forma que instâncias distribuídas dos repositórios subjacentes possam ser acessados. Dois estudos de caso foram conduzidos neste ambiente.

O planejamento e execução dos estudos de caso envolveram os seguintes passos: *(i)* definição dos repositórios populados com diferentes informações de ontologias de biodiversidade; *(ii)* escolha de uma operação disponível na interface do Aondê para requisição; *(iii)* escolha de uma estratégia de tolerância a falhas a ser executada, e *(iv)* execução da bateria de testes com introdução de falhas como indisponibilidade dos serviços dos repositórios. Para extração dos resultados, a metodologia utilizada foi mostrar em um gráfico como as instâncias do serviço Web Aondê, a reconfiguração dinâmica das requisições ou requisições simultâneas atuaram para tornarem os serviços Web mais confiáveis. Em ambos cenários, a disponibilidade total dos serviços ao longo de uma bateria de testes alcançou 85%, onde 50% das requisições foram atendidas com pelo menos um repositório em estado de indisponibilidade. Um benefício adicional foi alcançado no estudo de caso utilizando N-versões ao obter e combinar resultados oriundos de múltiplas instâncias do Aondê em um resultado mais extenso, isto é, com um número maior de informações de biodiversidade em uma única requisição.

Um impacto previsto na adoção da infra-estrutura como Arquitetura Mediador é o aumento da latência na requisição do serviço-alvo, devido ao nó adicional na qual as mensagens tem que passar. Portanto, uma relação de compromisso tem que ser estudada ao adotar não só a Arquitetura Mediador mas qualquer solução baseada em *middleware*. Se uma solução para requisição confiável de serviços Web ficar implantado em cada ambiente

de execução, considerando toda a heterogeneidade envolvida destes ambientes, dificuldades de evolução, manutenção e acoplamentos indesejados podem criar latências ainda mais difíceis de contornar. Em ambientes onde há a distribuição e autonomia dos serviços, manter este paradigma e oferecer essa funcionalidade de mediação confiável como serviços Web autônomos e interface de acesso padronizado é a abordagem com menor impacto na arquitetura.

A Arquitetura Mediador também pode ser componente ativo de *workflows* científicos baseados em serviços Web. Muitos *workflows* científicos têm como requisitos a descoberta, composição dinâmica e invocação de um conjunto de passos realizados por serviços Web, sempre levando em consideração a interoperabilidade das interfaces e dados. Nenhum dos estudos de caso desta dissertação explorou essa aplicabilidade. Contudo, espera-se que a Arquitetura Mediador provenha os requisitos de workflows científicos citados. Esta aplicabilidade também tem o potencial de produzir novos requisitos que possam ser endereçados pela Arquitetura Mediador, já que este seria um componente onipresente entre diversos serviços orquestrados por estes workflows.

A principal dificuldade no projeto e implementação do componente Arquitetura Mediador foi expor as funcionalidades de mediação confiável oferecidas pelo componente de terceiros WS-Mediator, usando tecnologia de serviços Web. A separação da lógica de diálogo com as aplicações e a implementação da mediação confiável foi a principal diretriz para o projeto arquitetural. O próprio componente WS-Mediator apresentou necessidades de evolução durante a sua reutilização, que foram implementadas e é uma das contribuições deste trabalho. A opção por serviços *stateful* por um lado requereu investigação maior da interface de programação e provas de conceito mais sofisticados, mas por outro lado trouxe o benefício de usabilidade maior dos serviços e facilidade de implantação. Ainda considerando serviços Web, o fato da operação de *Consulta* do serviço Web Aondê retornar o resultado como um anexo levou a esforços substanciais de implementação relacionados à interface Web, inclusive requerendo a codificação de interfaces utilizando especificação *XML Schema*. O componente JAX-WS apoiou a contento a implementação dos requisitos citados.

Por fim, a definição dos cenários de distribuição e informações envolvidas nos estudos de caso visaram principalmente a simplicidade no entendimento, ao mesmo tempo mantendo a aplicabilidade em um domínio real de biodiversidade, materializado, por exemplo, em dados reais populados no repositório do Aondê. O desafio neste caso foi propor cenários que aliam a interoperabilidade de todos os serviço Web envolvidos e as estratégias de tolerância a falhas. Desta forma foi possível integrar a Arquitetura Mediador nas implementações desenvolvidas no projeto BioCORE. O desafio ainda se mantém em apoiar as novas implementações previstas.

6.2 Contribuições

As principais contribuições deste trabalho são:

- Levantamento dos principais atributos e requisitos de dependabilidade em sistemas de software com ênfase em arquiteturas orientadas a serviços Web. Levantamento das principais infra-estruturas cuja mediação confiável de serviços Web aplicadas a arquiteturas orientada a serviços Web tenha sido abordada.

- Estudo e aplicação do componente WS-Mediator, de comprovada eficácia no que diz respeito a serviços Web no domínio de e-Science, contudo a solução pode ser aplicada com sucesso a outros domínios. Esta aplicação detectou alguns defeitos de implementação do WS-Mediator. O pesquisador Y. Chen, autor do WS-Mediator, prontamente atendeu às sugestões de correção. Atualmente, a versão do WS-Mediator disponível na Web, já oferece as correções que originaram deste trabalho.

- Implementação de uma infra-estrutura de mediação confiável distribuída. Chamada de Arquitetura Mediador, esta implementação utiliza as técnicas de tolerância a falhas oferecidas pelo componente WS-Mediator, porém acessível remotamente mediante serviços Web. Com a Arquitetura Mediador, arquiteturas existentes e novas podem usufruir de mediação confiável sem grande impacto, pois apenas será necessário acessar serviços Web interoperáveis entre plataformas e linguagens.

- Validação prática da infra-estrutura Arquitetura Mediador em um projeto de biodiversidade, através de provas de conceito que visam intermediar a comunicação dos componentes de consulta e armazenamento de dados de biodiversidade e ecologicamente cientes. Desta maneira, esta infra-estrutura estará apta a ser parte integrante do projeto, auxiliando no objetivo principal que é prover soluções computacionais confiáveis para os cientistas no domínio do projeto BioCORE. Também estará apta para outras pesquisas em dependabilidade e SOA.

6.3 Trabalhos Futuros

Um trabalho futuro de grande valor é a aplicação da Arquitetura Mediador nos demais componentes e serviços desenvolvidos no projeto BioCORE. A seção 6.3.1 apresenta o repositório de coletas com expansão de consultas, componente integrante da arquitetura do BioCORE, que foi desenvolvido mas não foi experimentado sob os requisitos de mediação confiável. Desta forma é indicada, como trabalho futuro, uma prova de conceito

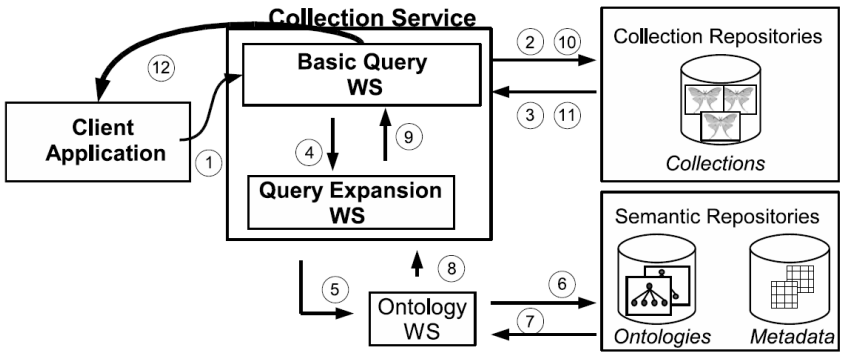


Figura 6.1: Arquitetura do repositório de coletas com serviço de expansão de consultas.

envolvendo a implementação do repositório de coletas e a infra-estrutura Arquitetura Mediador. Em seguida, a seção 6.3.2 apresenta uma lista de sugestões de extensões para a Arquitetura Mediador.

6.3.1 Repositório de Coletas com Expansão de Consultas

O *repositório de coletas*, proposto por J. Malaverri [51], consiste em um banco de dados para registros de coletas obtidos em observações de campo. O modelo de dados proposto para o repositório também combina informações coletadas com informações já catalogadas em museus. As funcionalidades básicas para acesso ao repositório de coletas são expostas como serviços Web. A linguagem para exprimir consultas adotada pelo repositório é a linguagem SQL, de forma que os serviços Web abstraem o acesso aos bancos de dados que mantêm os dados ecológicos. As consultas convencionais que são submetidas diretamente aos bancos de dados de coletas são consideradas como consultas *sem expansão*.

Também como pesquisa integrante do projeto BioCORE, os serviços do repositório de coletas usufruem do *serviço de expansão de consultas*, proposto por B. Vilar [52]. Este serviço faz uso de ontologias obtidas do serviço de ontologias Aondê [28] para expandir semanticamente uma consulta. Uma expressão de consulta expandida é uma reformulação da consulta original que incorpora termos e conceitos que não estão no repositório de coletas, mas são informações que fazem parte da visão conceitual dos biólogos.

A Figura 6.1 apresenta a arquitetura da solução para gerenciamento de informações de coletas, com expansão de consultas [52]. Os componentes de nomes com sufixo "WS" são implementações de serviços Web. Os fluxos enumerados representam seqüências de mensagens trocadas entre os componentes tanto para consultas sem expansão quanto para consultas com expansão semântica.

Como trabalho futuro, ambas consultas podem usufruir da mediação confiável usando Arquitetura Mediador. A aplicação-cliente (*client application*) pode utilizar redundâncias

de serviços de coletas (*collection service*) para aumento da dependabilidade, em configurações e estratégias de tolerância a falhas similares aos aplicados nos estudos de caso feitos com o repositório de ontologias nas seções 5.2 e 5.3. Especificamente para uma consulta com expansão, o serviço de coleta faz o papel de aplicação-cliente do repositório de ontologias (*semantic repositories*). Para satisfazer uma consulta expandida por ontologias do domínio da biodiversidade, diversas instâncias do serviço Web Aondê podem ser envolvidos simultaneamente. Ontologias parciais podem estar distribuídas nestes diversos repositórios. Logo, se uma instância da Arquitetura Mediador em modo N-versões for aplicada na comunicação entre os componentes envolvidos na consulta expandida, um benefício previsto é o aumento da probabilidade de que a consulta seja reformulada com sucesso. A razão é que mais termos semânticos oriundos de diferentes ontologias seriam incorporados ao processo de expansão. Naturalmente, para este último cenário, outro benefício esperado é tolerar falhas em serviços Web individuais, de forma que a consulta expandida seja realizada mesmo utilizando um conjunto de termos reduzidos provenientes apenas dos serviços que satisfizerem a requisição.

6.3.2 Extensões

Abaixo são discorridas algumas possíveis direções futuras de extensão e implementação da infra-estrutura Arquitetura Mediador:

- A proposta do WS-Mediator prevê o oferecimento de outras soluções acerca de mediação confiável, como monitoramento de serviços Web e emprego de multi-rotas para usufruir de redundâncias de enlaces na Internet. Estas propostas podem ser incorporadas à Arquitetura Mediador. Particularmente para monitoramento de serviços, os benefícios já foram previstos na discussão dos estudos de caso (seções 5.2.4 e 5.3.4).

- Uma instância da Arquitetura Mediador pode servir a vários componentes e aplicações-clientes simultaneamente, portanto tornando-se um ponto crítico nestes sistemas. Em cenários onde o número de requisições se tornar alto, faz-se necessário a implantação da Arquitetura Mediador em um ambiente com alta disponibilidade e escalabilidade, por exemplo utilizando servidores em *cluster* e mecanismos de tolerância a falhas em nível de servidor de aplicação, sistema operacional e hardware.

- A configuração das políticas que regem a execução da Arquitetura Mediador (*e.g.* serviços candidatos, estratégia de tolerância a falhas, metadados de dependabilidade) são mantidas com tempo de vida equivalente a um diálogo de mediação, através de serviços Web *stateful*. Em cenários onde o tempo de vida das confi-

gurações precisa ser estendido, faz-se necessário utilizar persistência, por exemplo utilizando banco de dados.

- A configuração das políticas que regem a execução da Arquitetura Mediador atualmente ficam disponíveis na interface pública, sendo a aplicação-cliente responsável por informar estes parâmetros. Contudo, práticas de governança SOA [64] podem definir atribuições e responsabilidades, de forma a definir quais configurações estarão disponíveis para o parque de aplicações. Faz parte também da governança SOA definir quais serviços poderão efetivamente usufruir da mediação confiável. Desta forma, a Arquitetura Mediador pode oferecer subsídios para apoiar governança SOA, por exemplo, oferecendo uma interface restrita por autenticação ou console de configuração para os administradores da infra-estrutura.

- A seção 5.3.2 ilustrou na prática um componente conector que é executado no ambiente do cliente responsável por manipular mensagens retornadas pela heurística de votação, quando a estratégia de tolerância a falhas executada é N-versões. Estas mensagens são repostas consideradas "trivialmente corretas", pois não são mensagens de falha na comunicação, que são descartas pela implementação da Arquitetura Mediador. Se o domínio em questão dos serviços-candidatos tiver um conjunto de heurísticas em comum para detectar falhas ou combinar mensagens, estas implementações podem, portanto, serem executadas de forma eficiente no ambiente do servidor que hospeda a instância da Arquitetura Mediador. A Arquitetura-Mediador, portanto, pode dinamicamente selecionar e executar uma heurística baseado em uma política de execução. Por parte das aplicações-clientes, além da reutilização de software proporcionada, o emprego final da Arquitetura Mediador seria ainda minimizada pois esta implementação adicional (*i.e.* conector) não necessitaria mais estar replicada em cada cliente.

Referências Bibliográficas

[1] E-Science. Research Concils UK, 2008. http://www.rcuk.ac.uk/escience/default.htm.

[2] Glassfish - Open Source Application Server, 2008. https://glassfish.dev.java.net/.

[3] JBossESB Stateless Service Clustering, 2008. http://wiki.jboss.org/wiki/JBossESBStatelessServiceClustering.

[4] NEReSC Projects, 2008. http://www.neresc.ac.uk/projects/index.php.

[5] The State of Workflow by Tom Baeyens, 2008. http://www.jboss.com/products/jbpm/stateofworkflow.

[6] Java API for XML Web Services (JAX-WS) - Stateful Web Service with JAX-WS RI, 2009. https://jax-ws.dev.java.net/nonav/jax-ws-21-ea2/docs/statefulWebservice.html.

[7] JAX-WS Reference implementation, 2009. https://jax-ws.dev.java.net/.

[8] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2003.

[9] T. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.

[10] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), 2004.

[11] T. Bellwood. Understanding UDDI: Tracking the Evolving Specification. IBM DeveloperWorks, 2002. http://www.ibm.com/developerworks/library/ws-featuddi/.

[12] BPMI.org. BPML — BPEL4WS - A Convergence Path toward a Standard BPM Pack. Position Paper, 15 de agosto de 2002, 2008. `http://www.bpmi.org/downloads/BPML-BPEL4WS.pdf`.

[13] A. P. L. Carvalho, A. Brayner, A. Loureiro, A. Furtado, A. Staa, C. J. P. Lucena, C. S. Souza, C. B. Medeiros, C. Lucchesi, E. S. Silva, F. R. Wagner, I. Simon, J. Wainer, J. C. Maldonado, J. P. M. Oliveira, L. Ribeiro, L. Velho, M. A. Goncalves, M. C. C. Baranauskas, M. Mattoso, N. Ziviani, P. O. A. Navaux, R. S. Torres, V. A. F. Almeida, W. Meira Jr, and Y. Kohayakawa. Grandes Desafios da Pesquisa em Computação no Brasil. Technical report, SBC, 2006. `http://www.sbc.org.br` (English and Portuguese).

[14] J. Cheesman and J. Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] F. Chen, Q. Wang, H. Mei, and Fuqing Yang. An architecture-based approach for component-oriented development. In *26th Annual International Computer Software and Applications Conference*, 2002.

[16] Y. Chen. *WS-Mediator for Improving Dependability of Service Composition*. PhD thesis, Newcastle University, Newcastle upon Tyne, United Kingdom, 2008.

[17] Y. Chen and A. Romanovsky. Improving Service Availability without Improving Availability of Individual Services. Technical report, University of Newcastle upon Tyne, 2007. `http://www.cs.ncl.ac.uk/research/pubs/trs/papers/1025.pdf`.

[18] Y. Chen and A. Romanovsky. Improving the Dependability of Web Services Integration. *IEEE IT Professional*, 10(3), May-June 2008.

[19] P. Clements and R. Kazman. *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc.,, Boston, MA, USA, 2003.

[20] World Wide Web Consortium. Document Object Model (DOM) Level 2 Core Specification v1.0. W3C Recommendation, 2000. `http://www.w3.org/TR/DOM-Level-2-Core/`.

[21] World Wide Web Consortium. SOAP Messages with Attachments. W3C Note, 2000. `http://www.w3.org/TR/SOAP-attachments`.

[22] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. W3C Note, 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[23] World Wide Web Consortium. Web Services Addressing (WS-Addressing). W3C Member Submission, 2004. http://www.w3.org/Submission/ws-addressing/.

[24] World Wide Web Consortium. Web Services Architecture. W3C Working Group Note, 2004. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#whatis`.

[25] World Wide Web Consortium. Web Services Policy 1.2 - Framework (WS-Policy). W3C Member Submission, 2006. `http://www.w3.org/Submission/WS-Policy/`.

[26] World Wide Web Consortium. SOAP Version 1.2 Part 0: Primer (Second Edition). W3C Recommendation, 2007. `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/`.

[27] World Wide Web Consortium. SPARQL Query Language for RDF. W3C Recommendation, 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[28] J. Daltio. Aondê: Um serviço web de ontologias para interoperabilidade em sistemas de biodiversidade. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, 2007.

[29] L. A. Digiampietri. *Gerenciamento de workflows científicos em bioinformática*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, 2007.

[30] H.E. Eriksson, M. Penke, B. Lyons, and D. Fado. *UML 2 Toolkit*. Wiley Publishing, Inc., 2004.

[31] A. Erradi and P. Maheshwari. A Broker-based Approach for Improving Web Services Reliability. In *Proc. IEEE Int. Conf. on Web Services (ICWS)*, 2005.

[32] M. Fayad and D.C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

[33] D.F. Ferguson, T. Storey, B. Lovering, and J. Shewchuk. Secure, Reliable, Transacted Web Services. IBM and Microsoft, 2003. `http://www.ibm.com/developerworks/webservices/library/ws-securtrans`.

[34] The Apache Software Foundation. Apache Axis2 - Java, 2009. `http://ws.apache.org/axis2/`.

[35] The Apache Software Foundation. Apache Synapse Enterprise Service Bus (ESB), 2009. `http://synapse.apache.org/`.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[37] D. Gannon, B. Plale, and D.A. Reed. Service Architectures for e-Science Grid Gateways: Opportunities and Challenges. In R. Meersman et al., editor, *OTM, Part II*, volume 4804 of *LNCS*, pages 1179–1185. Springer Berlin / Heidelberg, 2007.

[38] D.Z.G. Garcia and M.B.F. de Toledo. A Fault Tolerant Web Service Architecture. In *5th IEEE Latin American Web Congress*, 2007.

[39] A. Gorbenko, V. Kharchenko, P. Popov, A. Romanovsky, and A. Boyarchuk. Development of Dependable Web Services out of Undependable Web Components. Technical report, School of Computing Science, University of Newcastle upon Tyne, United Kingdom, 2004. `http://www.cs.ncl.ac.uk/research/pubs/trs/papers/863.pdf`.

[40] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[41] M.N. Huhns and P.S. Munindar. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, Jan-Feb 2005.

[42] R. Johnson. J2EE development frameworks. *IEEE Computer*, 38(1):107–110, 2005.

[43] L. C. Gomes Jr. Uma Arquitetura para Consultas a Repositórios de Biodiversidade na Web (An architecture to query biodiversity data on the Web). Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, 2007.

[44] K. Kawaguchi. Stateful web service with JAX-WS RI 2.1 EA2, 2006. `http://weblogs.java.net/blog/kohsuke/archive/2006/10/stateful_web_se.html`.

[45] H. Kong, M. Hwang, and P. Kim. A New Methodology for Merging the Heterogeneous Domain Ontologies Based on the WordNet. In *Proc. of the International Conference on Next Generation Web Services Practices (NWESP'05)*, page 235, Washington, DC, USA, 2005.

[46] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service Oriented Architecture Best Practices*. Prentice-Hall, Englewood Cliffs, 2005.

[47] S. Krishnan and K. Bhatia. SOAs for Scientific Applications: Experiences and Challenges. *Future Generation Computer Systems*, 25(4):466–473, April 2009.

[48] J.C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *IEEE Computer*, 27(7):39–51, 1990.

[49] P. Li, Y. Chen, and A. Romanovsky. Measuring the Dependability of Web Services for Use in E-Science Experiments. In D. Penkler et al., editor, *Service Availability*, volume 4328 of *LNCS*, pages 193–205. Springer Berlin / Heidelberg, 2006.

[50] B. Lublinsky. Defining SOA as an architectural style: Align your business model with technology. IBM DeveloperWorks, 2007. `http://www.ibm.com/developerworks/architecture/library/ar-soastyle/`.

[51] J.E.G. Malaverri. Um serviço de gerenciamento de coletas para sistemas de informação de biodiversidade. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, 2009.

[52] J.G. Malaverri, B. Vilar, and C.B. Medeiros. A Tool Based on Web Services to Query Biodiversity Information. In *5th International Conference on Web Information Systems and Technologies (Webist)*, 2009.

[53] C. B. Medeiros, J. Perez-Alcazar, L. Digiampietri, G. Z. Pastorello, A. Santanche, R. Torres, and E. Madeira. WOODSS and the Web: Annotating and Reusing Scientific Workflows. *ACM SIGMOD Record*, 34(3):18–23, 2005.

[54] R.T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, 1997.

[55] OASIS. UDDI Version 3.0.2 Specification. UDDI Spec Technical Committee Draft, 2004. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[56] B. Randell, A. Romanovsky, C.M.F. Rubira, R.J. Stroud, Z. Wu, and J. Xu. From Recovery Blocks to Concurrent Atomic Actions. In H. Kopetz, J.C. Laprie, R. Brian, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 87–101. Springer-Verlag New York, Inc., 1995.

[57] B. Randell and J. Xu. Chapter 1: The Evolution of the Recovery Block Concept. In *Software Fault Tolerance*, pages 1–22. John Wiley & Sons Ltd, 1994.

[58] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[59] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[60] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Dependability in the Web Services Architecture. In R. de Lemos et al., editor, *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 90–109. Springer Berlin / Heidelberg, 2003.

[61] V. Tosic, A. van Moorsel, and R. Wong. Introduction to the Proceedings of the EDOC 2006 Workshop Middleware for Web Services (MWS). In *10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06)*, 2006.

[62] T.S. Weber. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. Curso de Especialização em Redes e Sistemas Distribuídos, Instituto de Informática, UFRGS, 2002. `http://www.inf.ufrgs.br/~taisy/disciplinas/textos/Dependabilidade.pdf`.

[63] The Official ITIL Website. What is ITIL?, 2009. `http://www.itil-officialsite.com/AboutITIL/WhatisITIL.asp`.

[64] P.J. Windley. SOA Governance: Rules of the Game. *InfoWorld*, 4:29–35, 2006.

[65] A. Zalewski. A FMECA framework for Service Oriented Systems based on Web Services. In *2nd Int. Conf. on Dependability of Computer Systems (IEEE DepCoS-RELCOMEX'07)*, 2007.

[66] Y. Zhu, D. Ma, H. Sun, S. Zhang, and J. Li. A QoS-Aware Middleware for Ensuring Web Services Reliability. In *Proc. 25th Int. Multi-Conf. Parallel and Distrib. Comp. and Networks (IASTED)*, Innsbruck, Austria, 2007.