Instituto de Computação
Universidade Estadual de Campinas

Teste de Robustez de Uma Infraestrutura Confiável para Arquiteturas Baseadas em Serviços Web

Willian Yabusame Maja

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Willian Yabusame Maja e aprovada pela Banca Examinadora.

Campinas, 30 de agosto de 2011.

Eliane Martins (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 POR
Maria Fabiana Bezerra Müller – CRB8/6162 – BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA - UNICAMP

Maja, Willian Yabusame, 1986-
M288t Teste de robustez de uma infraestrutura confiável para arquiteturas baseadas em serviços Web / Willian Yabusame Maja. -- Campinas, SP : [s.n.], 2011.

Orientador: Eliane Martins.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

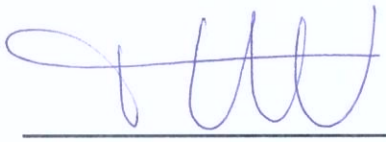
1. Software - Validação. 2. Engenharia de Software - Injeção de falhas. 3. Serviços Web. 4. Arquitetura orientada a serviços (Computação). 5. Tolerância a falha (Computação). I. Martins, Eliane, 1955-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: Robustness testing of a reliable infrastructure for web service-based architectures
Palavras-chave em inglês:
Computer software - Validation
Software engineering - Fault injection
Web services
Service-oriented architecture (Computer science)
Fault-tolerant computing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Taisy Silva Weber
Claudia Maria Bauzer Medeiros
Data da defesa: 21-06-2011
Programa de Pós-Graduação: Ciência da Computação

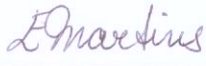
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 21 de junho de 2011, pela Banca examinadora composta pelos Professores Doutores:



Profª. Drª. Taisy Silva Weber
Instituto de Informática / UFRGS

Profª. Drª. Claudia Maria Bauzer Medeiros
IC / UNICAMP



Profª. Drª. Eliane Martins
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Teste de Robustez de Uma Infraestrutura Confiável para Arquiteturas Baseadas em Serviços Web

Willian Yabusame Maja[1]

Junho de 2011

Banca Examinadora:

- Eliane Martins (Orientadora)
- Taisy Silva Weber
  Instituto de Informática-UFRGS
- Claudia Maria Bauzer Medeiros
  Instituto de Computação-Unicamp
- Eleri Cardozo (Suplente)
  FEEC-Unicamp
- André Santanchè (Suplente)
  Instituto de Computação-Unicamp

[1]Suporte financeiro de: Bolsa do CNPq (136419/2008-9) 2008–2010

Resumo

Os sistemas baseados em serviços Web estão suscetíveis a diversos tipos de falhas, entre elas, as causadas pelo ambiente em que operam, a Internet, que está sujeita a sofrer com problemas como, atrasos de entrega de mensagem, queda de conexão, mensagens inválidas entre outros. Para que estas falhas não causem um problema maior para quem está interagindo com o serviço Web, existem soluções, como é o caso do Archmeds, que fornece uma infraestrutura confiável que melhora a confiabilidade e disponibilidade dos sistemas baseados em serviços Web. Mas, para o Archmeds ser uma solução confiável, ele também deve ser testado, pois ele também é um sistema que está sujeito a ter defeitos. Por isso, este trabalho propõe uma abordagem para teste de robustez no Archmeds e para isso, contou com o desenvolvimento de uma ferramenta de injeção de falhas chamada WSInject, que utiliza falhas de comunicação e dados de entrada inválidos nos parâmetros das chamadas aos serviços. Com isso espera-se emular as falhas do ambiente real de operação dos serviços Web e revelar os defeitos do sistema sob teste. Este trabalho também levou em conta que o Archmeds é uma composição de serviços Web, por isso também propõe uma abordagem para testar composições de serviços. Com os resultados deste estudo de caso, espera-se que esta abordagem de teste de robustez possa ser reutilizada para outros sistemas baseados em serviços Web.

Abstract

Web service-based systems are subject to different types of faults, among them, the ones caused by the environment in which they work, which is the Internet. These faults could be problems like delay of message, connection loss, invalid message request, and others. To avoid that these faults do not become a bigger problem for the clients who are interacting with the Web service, a solution can be the use of a reliable infrastructure, like Archmeds, to increases the reliability and availability of the Web-service-based systems. Although Archmeds is a solution with the aim to increase the reliability of Web services, it is also subject to faults and for this reason, it should also be tested. This work proposes an approach to test the robustness of Archmeds and to reach this goal, a fault injection tool, called WSInject, was developed. It uses communication faults and invalid inputs into services calls. In order to reveal the failures, these faults aim to emulate the real ones that affect the Web services in the real operational environment. This work also took into account that Archmeds is a Web service composition and for this reason, it was created an approach to test it. With the results of this case study, it is expected that this approach can be adapted to others applications based in Web services technology.

Agradecimentos

Finalmente, após três anos, consegui finalizar este trabalho, mas o mais importante é que nada disso seria possível sem as pessoas que me apoiaram durante todos estes anos.

Primeiramente, gostaria de agradecer aos meus pais, pois são os principais responsáveis pelo o que eu sou. Hoje, eu consigo ligar os pontos de tudo que aconteceu até agora na minha vida e vejo que tudo foi graças ao que me ensinaram e ao apoio que me deram. Tenho sorte de ter vocês como meus pais e sou feliz por isso.

Também gostaria de agradecer os meus amigos Daniel Vecchiato, Douglas Leite, Felipe Sansão, Guilherme Armigliatto, Maria Angélica Souza e Thiago Lechuga pela amizade e companheirismo. Embora o tempo tenha nos afastado, os momentos que passamos juntos serão lembrados para sempre.

Finalmente, gostaria de agradecer a todos que me ajudaram tanto diretamente quanto indiretamente para a realização deste trabalho. A vocês, muito obrigado!

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Arquitetura Orientada a Serviços[1] (SOA) em conjunto com serviços Web[2] (WS) são flexíveis, dinâmicos e fracamente acoplados, por isso um número crescente de aplicativos de *e-business*, comércio eletrônico, *e-science* e outros tipos estão sendo desenvolvidos baseados nestas tecnologias. Parte deste sucesso é devido à sua interoperabilidade, que é garantida por diversos padrões em XML (*eXtensible Markup Language*), que definem como as interfaces devem ser utilizadas e como deve ser o formato das mensagens que serão trocadas entre o cliente e o WS[7].

O BioCORE[3], um projeto desenvolvido no Instituto de Computação da Unicamp em parceria com diversos outros institutos, é um exemplo de aplicação que utiliza os WSs para conseguir prover ferramentas computacionais para o auxilio aos pesquisadores da área de biodiversidade. Como o número de aplicações que usam WS está crescendo, soluções mais confiáveis são essenciais para que os defeitos, que por ventura surgirem, não causem danos como perda de cliente, dinheiro, tempo e entre outros.

Uma solução para aumentar o grau de dependabilidade dos sistemas baseados em WSs é o uso de uma infraestrutura que isola as falhas dos sistemas antes dos defeitos se manifestarem. No BioCORE, esta infraestrutura é realizada pelo Archmeds[8][9]. Entretanto, como qualquer outro sistema computacional, o Archmeds também está suscetível a falhas e isto pode comprometer a sua efetividade em conseguir aumentar a confiabilidade e disponibilidade dos WS.

O BioCORE adotou, como infraestrutura, o Archmeds para aumentar a confiabilidade de seus WSs, porém, para que a solução seja confiável, o Archmeds deve ser testado. O teste de robustez é uma abordagem para verificar e avaliar o comportamento dos sistemas na presença de falhas e quando estão operando em ambientes sob condições de sobrecarga.

[1] Do inglês, *service oriented architecture*.
[2] Do inglês, *web service*.
[3] http://www.lis.ic.unicamp.br/projects/biocore.

O resultado deste tipo de teste expõe os defeitos de forma que eles possam ser analisados e as falhas encontradas e corrigidas.

Neste cenário, a injeção de falhas é uma técnica utilizada para teste de robustez, pois ela emula a ocorrência de falhas reais nos sistemas. Por isso, esta técnica tem o propósito de revelar problemas de robustez e consequentemente, indicar quais das falhas foram a causa do defeito.

Para que seja possível testar a robustez do Archmeds, antes é necessário ter o conhecimento de algumas de suas características. Entre elas, o fato do sistema ser composto por três WSs, pois isto indica que a abordagem de teste deve levar em conta que existe uma composição de WSs a ser testada, cuja complexidade é maior do que testar somente um WS ponta a ponta. Outra característica é o uso de uma extensão no padrão de comunicação, o que exige uma ferramenta que consiga lidar com este tipo de flexibilidade na adição de novos padrões dentro do já existente.

Estas características do Archmeds aumentam o desafio na hora de testá-lo, pois são diferentes do padrão básico proposto para WS e por isso, os trabalhos relacionados a teste de robustez em WSs, muitas das vezes, não conseguem testar serviços com as características do Archmeds.

Entre os outros desafios deste trabalho está em conseguir encontrar uma abordagem de injeção de falhas que emule falhas do ambiente real de operação do Archmeds e que também consiga expor os defeitos para que sejam analisados e corrigidos. Esta abordagem tem que ser automatizada, pois o processo de testes pode conter centenas de casos de teste e seria muito custoso e passível de erro se feito manualmente.

1.1 Objetivos e solução proposta

Com os desafios em vista, este trabalho possui o objetivo principal de testar o Archmeds em seu ambiente de operação, que é dentro da arquitetura do BioCORE. Para que isto fosse possível, foram traçados alguns objetivos para conseguir chegar a uma solução adequada para testar a robustez do Archmeds. Entre os objetivos estavam:

- Fazer uma pesquisa para encontrar uma abordagem compatível com o Archmeds.
- Criar um modelo de falhas que represente falhas do ambiente real de operação do Archmeds.
- Buscar um meio automatizado para conduzir a execução dos casos de teste.
- Criar um ambiente semelhante ao do BioCORE para que os testes fossem conduzidos em condições semelhantes ao utilizado pelos pesquisadores.

- Encontrar e classificar os defeitos encontrados nos testes.

Como o Archmeds difere de um WS básico, a solução proposta para atingir os objetivos deste trabalho foi desenvolver uma nova ferramenta de injeção de falhas para WSs. Esta ferramenta foi desenvolvida em parceria com outros alunos de mestrado e doutorado do Instituto de Computação da Unicamp e *Institut National des Telecommunications* e seus respectivos orientadores. Além disso, também houve o apoio de dois projetos, o BioCORE e o RobustWeb , este último relacionado ao desenvolvimento e validação de aplicações SOA e WSs.

A ferramenta foi batizada de WSInject e possui as características necessárias para testar o Archmeds e uma variedade de WSs. A ideia no início de seu desenvolvimento foi colher os requisitos, que possibilitem o uso desta ferramenta em uma quantidade mais abrangente de WSs, pois existem muitos padrões e arquiteturas envolvidos. Outra preocupação foi torná-la automatizada. Isto foi possível através do proveito dos padrões dos WSs para conseguir realizar a geração automática dos casos de teste de robustez, assim como a execução das mesmas.

O teste do Archmeds na arquitetura do BioCORE foi realizado de forma que emulasse o ambiente real de operação dos pesquisadores. Com isso foi possível validar a abordagem da WSInject e ao mesmo tempo, encontrar os defeitos dos sistemas sob testes.

Em suma, as principais contribuições desta dissertação são:

- Levantamento e comparação dos trabalhos relacionados e suas abordagens de testes e injeção de falhas em WS.
- Levantamento dos requisitos necessários para conseguir testar o Archmeds e outros serviços que possuem características semelhantes.
- Especificação e desenvolvimento de uma nova ferramenta de injeção de falhas em WS. Esta ferramenta foi batizada de WSInject e possui características que tornam possível testar diferentes tipos de WS.
- Uma listagem dos defeitos encontrados no Archmeds e no serviço Aondê do projeto BioCORE.
- Uma proposta de classificação para defeitos em WS, pois as classificações adotadas pelos trabalhos relacionados não se mostraram adequados para este trabalho.

1.2 Estrutura da dissertação

As próximas seções desta dissertação estão organizadas da seguinte forma. O Capítulo 2 possui a fundamentação teórica, que fornece uma visão geral dos conceitos envolvidos

neste trabalho. O Capítulo 3 discorre sobre os trabalhos relacionados que serviram como base para o desenvolvimento de uma nova ferramenta para teste de robustez em WS. O Capítulo 4 apresenta a WSInject. O Capítulo 5 descreve o estudo de caso que envolveu o teste do Archmeds na arquitetura do BioCORE. E finalmente, o Capítulo 6 contém as conclusões deste trabalho.

Capítulo 2

Fundamentação Teórica

Para entender esta dissertação, antes, é necessário ter uma visão geral dos conceitos envolvidos neste trabalho. Para isso, este capítulo começa com uma descrição breve do que é dependabilidade e como ele se encaixa no contexto de SOA e WSs, para que em seguida sejam descritos as abordagens de teste robustez existentes.

Antes de entrar em detalhes sobre a fundamentação teórica deste trabalho, primeiro, deve-se entender o que é um serviço correto, falha[1], erro[2] e defeito[3][29]. Os serviços de um sistema estão corretos quando eles operam de forma como foram especificados. Entretanto, se algum serviço desvia da especificação, então ele está com defeito. O serviço é oferecido corretamente através de uma sequência de estados, que geram um resultado correto, entretanto, um serviço defeituoso é resultado de um ou mais estados que desviam desta sequência. Este desvio é considerado um erro. Mas nem todos os erros geram defeitos, pois um estado errôneo não causa um defeito imediatamente. Já a falha é uma causa física ou de design que leva o sistema ao estado de erro[29].

2.1 Dependabilidade

Anteriormente, dependabilidade[4] era conhecido como "segurança no funcionamento", porém este termo é evitado, pois a palavra segurança é utilizada em outros contextos, o que pode confundir o real significado de dependabilidade. A primeira explicação para o significado de dependabilidade dizia que é a habilidade de fornecer um serviço no qual se pode justificadamente confiar. Entretanto, o termo "confiar" não ficou bem definido e por isso dificulta o entendimento neste contexto. Por isso, atualmente, a definição utilizada

[1]Do inglês, *fault*.
[2]Do inglês, *error*.
[3]Do inglês, *failure*.
[4]Do inglês, *dependability*.

é a habilidade do sistema de evitar defeitos que são mais frequentes ou severos que o aceitável [4][29].

O termo dependabilidade foi criado em razão da dependência depositada nos sistemas computacionais. Essa dependência existe porque os sistemas estão sendo empregados em diversas atividades essenciais da sociedade e esta, em contrapartida, começa a depender de tais sistemas, por exemplo, um piloto depende dos sistemas do avião para pilotá-lo de forma segura. Além da sociedade, um sistema A pode depender de outro sistema B, logo a dependabilidade do sistema A pode ser afetada pela dependabilidade do sistema B.

A dependabilidade de um sistema não pode ser mensurada através de valores numéricos. Há cinco atributos, que podem ser mensurados para caracterizar a dependabilidade de um sistema[4][29]:

- Confiabilidade[5]: é a capacidade de o serviço estar operacional, dentro das especificações e condições durante um período de tempo.
- Disponibilidade[6]: é a probabilidade de um serviço estar operacional durante um determinado instante de tempo.
- Inocuidade[7]: é a probabilidade do serviço não causar catástrofes ambientais, perdas de vidas humanas ou perda de valores financeiros demasiados.
- Confidencialidade[8]: é a habilidade de manter as informações disponíveis somente às pessoas que têm autorização.
- Integridade[9]: é a capacidade de manter o sistema em estado consistente.
- Mantenabilidade[10]: é a habilidade de se poder evoluir e reparar o sistema.

2.2 Arquitetura orientada a serviços

A ideia de orientação a serviços já existia em diversos contextos e com diferentes propósitos. Em suma, a ideia principal da orientação a serviços é ser uma abordagem para separar problemas distintos, isto é, decompor um problema maior em pedaços menores para facilitar a busca da solução[7].

[5]Do inglês, *reliability*.
[6]Do inglês, *availability*.
[7]Do inglês, *safety*.
[8]Do inglês, *confidentiality*.
[9]Do inglês, *integrity*.
[10]Do inglês, *maintainability*.

Como analogia, podemos pegar uma sociedade onde existem diversas empresas, onde cada uma oferece um serviço para o público. A ideia de orientação a serviços fica evidente no seguinte cenário: uma empresa que fornece um serviço de venda de pacotes de viagens, que ao invés de criar um serviço próprio de reserva de hotel, de transporte e de passagens, ela utiliza os serviços de empresas terceirizadas que já possuem estes serviços implantados. Neste caso, o problema maior, que é criar um pacote completo de viagens, pôde ser decomposto em diversos problemas menores e mais fáceis de serem resolvidas separadamente.

Apesar da ideia de dividir uma lógica de negócio em unidades menores não ser uma novidade, o termo serviço contém características que se distinguem de outras abordagens. Para exemplificar estas características, no cenário citado anteriormente, os serviços de reserva de transporte, hotel e passagens foram distribuídos para empresas terceirizadas, que não ficam necessariamente na mesma região, pois, apesar do distanciamento, os serviços destas empresas podem ser utilizados remotamente. E apesar do serviço de compra de pacote de viagem depender dos serviços de outras empresas, ele não está acoplado exclusivamente às empresas que são utilizados no momento, pois outras empresas, que fornecem serviços similares, podem substituir as que estão sendo utilizadas. Além disso, cada serviço pode evoluir isoladamente sem que interfira nos serviços das outras. Isto é possível, pois as empresas oferecem os serviços de forma padronizada, onde a lógica interna dentro da empresa (fornecedor, fluxo de trabalho etc.) pode ser modificada, mas a forma como é entregue sempre será a mesma[7].

SOA é um paradigma computacional que utiliza os serviços como elementos básicos para a criação de aplicações distribuídas. Os serviços executam funções que vão desde simples requisições atômicas até execução de processos de negócio complexos. E por ter diversos padrões, os serviços conseguem formar um meio uniformizado de acesso e isto possibilita que diversas entidades diferentes consigam compartilhar suas informações e operações. Para que isto seja possível, os serviços devem possuir os seguintes requisitos [20]:

- Tecnologia neutra, ou seja, independentemente da tecnologia utilizada, o serviço pode ser invocado através de padrões que são aceitos em qualquer ambiente computacional.
- Fracamente acoplado, ou seja, o funcionamento do serviço deve ser encapsulado de tal forma que não seja necessário ao cliente ou ao próprio serviço conhecer detalhes internos da estrutura do serviço.
- Localização transparente, ou seja, as suas definições e dados de localização devem estar armazenados em repositórios que possibilitem que diversos clientes localizem e utilizem o serviço independentemente da sua localização.

Estes requisitos são análogos às características citadas em relação aos serviços do cenário da empresa que vende pacote de viagens.

A interação no SOA é realizada através de mensagens trocadas entre os consumidores de serviços (ou cliente) e provedores de serviços. O consumidor é a entidade que vai utilizar e se beneficiar do serviço, enquanto o provedor é quem fornece o serviço. SOA possui uma lógica de design que oferece serviços aos consumidores através interfaces publicadas na rede de comunicação. Para isso, a arquitetura envolve mais uma entidade além do consumidor e provedor, o registro de serviços [7].

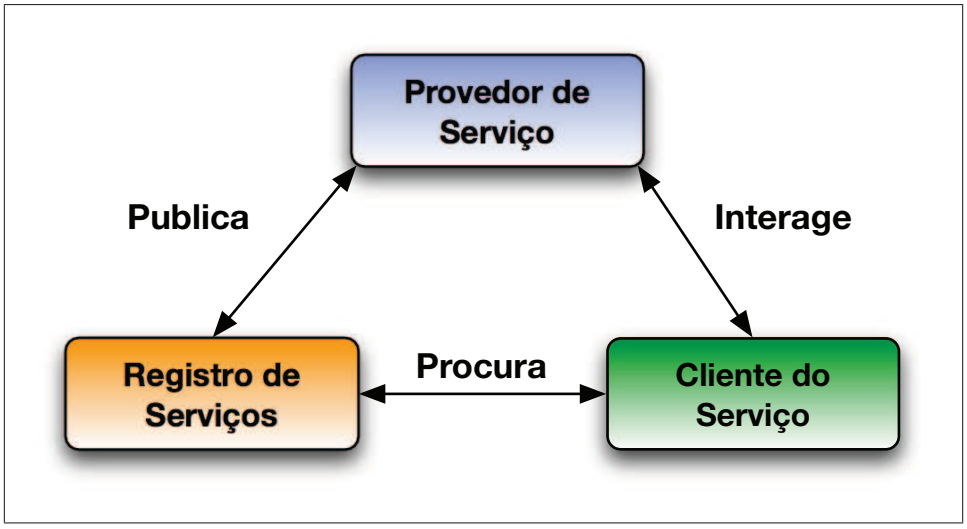


Figura 2.1: Arquitetura orientada a serviços.

A Figura 2.1 mostra como funciona a SOA. O primeiro passo para o cliente consumir um serviço deve ser procurá-lo em um registro de serviços que contém informações sobre a interface e localização. Logo em seguida, com estas informações, o cliente pode finalmente interagir com o provedor. No entanto, para que o serviço esteja disponível no registro, ele deve publicar as informações sobre o seu serviço no registro de serviços para que assim seja possível ele ser localizado e consumido.

Serviços podem ser simples ou compostos. No primeiro caso, o serviço é invocado diretamente e executa somente uma função atômica, por exemplo, reserva de hotel. A composição de serviços é a composição de vários serviços existentes em um novo, por exemplo, compra de pacote de viagens[7].

A Figura 2.2 mostra um exemplo de composição de serviços. Note que existem quatro serviços, sendo três simples e um composto. Os serviços simples, também chamados de serviços parceiros, oferecem serviços menos complexos, ou seja, com regras de negócios mais simples. O serviço composto, que é o serviço de venda de pacotes, possui uma lógica mais complexa, pois utiliza os serviços fornecidos pelos parceiros para fornecer um novo

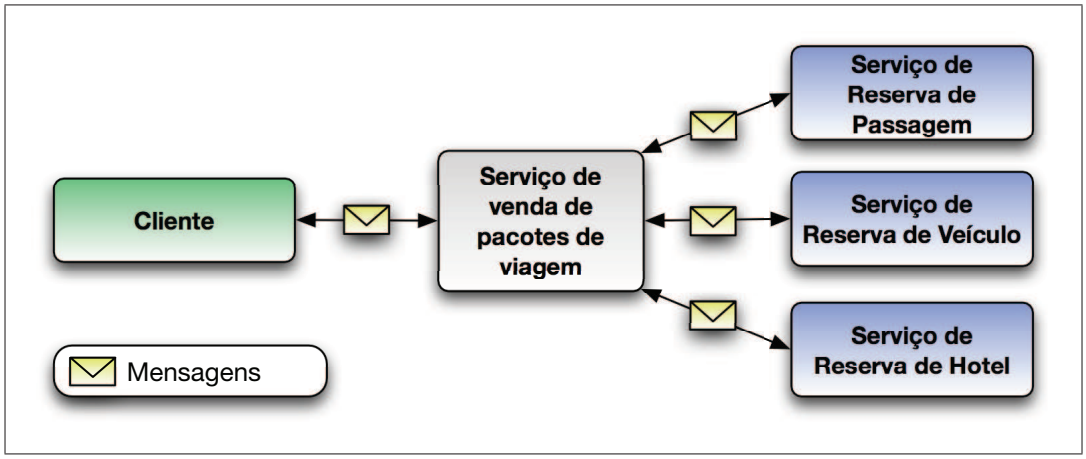


Figura 2.2: Composição de serviços.

serviço através da composição dos já existentes.

2.3 Serviços Web

Na Seção 2.2 foi explicado o conceito de SOA e serviços, nesta seção será descrita uma das formas mais populares de desenvolver um serviço, o WS, cuja característica é ser formado de diversos protocolos e padrões que operam tipicamente na internet.

Um dos fatos que permitiram o surgimento dos WSs e até mesmo do SOA foi o surgimento do *eXtensible Markup Language* (XML). Esta linguagem criou um padrão para representação dados que é facilmente estendido. Com isso, diversos novos protocolos, padrões e tecnologias se basearam no XML e tornaram possível a criação de WSs seguindo os requisitos do SOA[7].

Os WSs são vistos como uma forma de expor as funcionalidades dos sistemas de informações na internet com o uso de padrões de comunicação, que diminuem a heterogeneidade e consequentemente, diminui a complexidade de integração dos serviços[1]. Por exemplo, um dos usos que popularizou os WSs foi disponibilizar funcionalidades de sistemas legados para serem integrados em sistemas mais atuais de forma padronizada e independente da tecnologia.

Segundo a W3C, um grupo envolvido no contexto de WSs, WS é um *software* com as seguintes características [28]:

- Possui a capacidade de interação entre máquinas através da rede.
- Tem uma interface descrita em um formato processável por máquina, a *Web Service Definition Language* (WSDL).

- Tem a interação com os serviços feita através de trocas de mensagens no padrão *Simple Object Access Protocol* (SOAP) com o uso de protocolos de comunicação como o HTTP.

Esta definição da W3C, além de descrever o que é um WS, também indica quais são as tecnologias envolvidas. Atualmente, existem três padrões que estão sendo amplamente utilizados para criar WSs no padrão SOA, são eles: SOAP, WSDL e UDDI[7].

Os requisitos dos WSs exigem um formato padrão de mensagem, que sejam trocadas independentemente da localização e protocolo de transporte. Por este motivo, o SOAP foi adotado, pois além de cumprir com estes requisitos, ele é ativamente revisado e apoia extensões que incluem diversas funcionalidades avançadas.

O WSDL descreve a interface do WS através de informações abstratas e concretas. Basicamente, as informações abstratas indicam quais são as operações fornecidas pelo serviço, assim como o formato da mensagem que será utilizado como parâmetro e retorno da operação. No entanto, nenhuma informação sobre tecnologia é informada, pois para este fim, existem as informações concretas, que informam qual a *Uniform Resouce Identifies* (URI), o protocolo de transporte e entre outras tecnologias que são necessárias para que o cliente consiga estabelecer uma conexão física com o WS.

O *Universal Description, Discovery and Integration* (UDDI) trabalha como um repositório de serviços. Ele foi proposto para ser o padrão de publicação, consulta dinâmica e invocação de WSs.

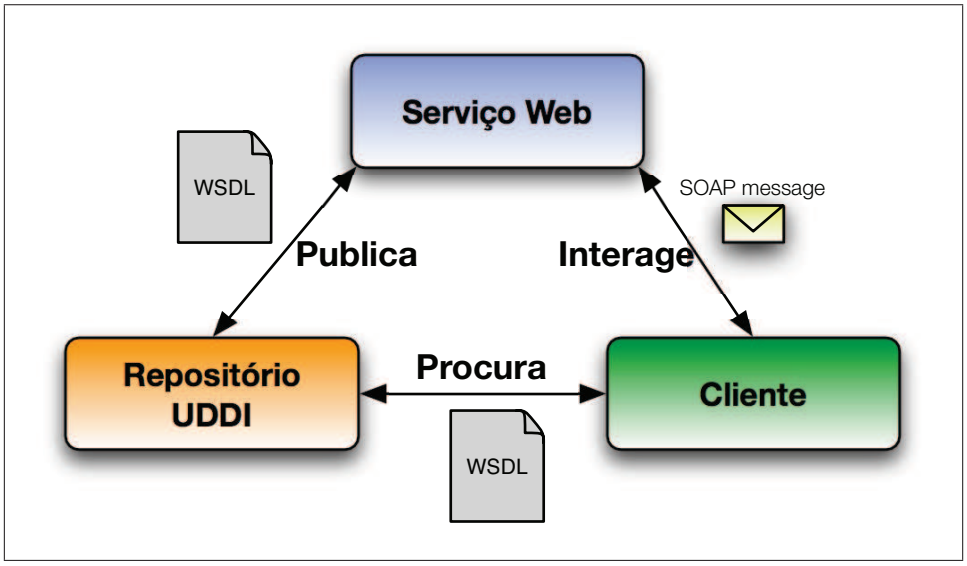


Figura 2.3: SOA com padrão de WSs.

A Figura 2.3 mostra como uma aplicação SOA ficaria estruturada usando os padrões dos WSs. Neste caso, o registro de serviços é trocado pelo UDDI e as interações entre

as entidades são feitas através de trocas de artefatos usando os padrões dos WSs. Para ocorrer o consumo de um WS, primeiro, este deve publicar a descrição WSDL de sua interface no repositório UDDI, para que, em seguida, o cliente possa buscá-la e, finalmente, consiga interagir com o WS através de trocas de mensagens SOAP.

2.4 Dependabilidade em WSs

Apesar de SOA em conjunto com WSs ser uma opção muito atrativa na construção de aplicações distribuídas, a tecnologia ainda é muito nova. Há diversos problemas que podem diminuir o nível de dependabilidade de um WS, entre eles [15]:

- Problemas na rede, que podem ser congestionamento da banda, latência alta, corrupção de mensagens, perda de pacotes entre outros.
- O WS pode não responder devido à alta carga de requisições ou por causa de pausas para manutenção.
- Os WSs podem ser de terceiros, o que também não dá garantias que ele possua alto grau de confiabilidade.
- O problema mais crítico são os dados de respostas, que podem estar errados devido às mensagens de requisição corrompidas ou com dados inválidos.

Estes problemas podem afetar atributos da dependabilidade, como disponibilidade, confiabilidade, inocuidade e integridade. Isto fica evidente em uma composição de WSs, onde o sucesso de uma operação depende do funcionamento correto de todos os serviços envolvidos, a dependabilidade da composição inteira pode ficar comprometida por causa de um WS. Por exemplo, se um serviço de venda de passagem aérea ficar com o serviço de pagamento indisponível por muito tempo, a empresa pode ficar com o serviço de venda indisponível e perder um volume grande de dinheiro.

Como a dependabilidade do serviço pode ser comprometida por causa destes problemas citados, é necessário criar mecanismos de tolerância a falhas para que estes sistemas não sofram danos com estes problemas. Por isso, é muito importante testar os WSs para validar se eles estão funcionando dentro do esperado. Além disso, os testes também devem verificar como os serviços se comportam na presença de falhas, para que assim os desenvolvedores possam criar mecanismos para tolerá-las.

2.5 Injeção de falhas

Antes de entrar com mais detalhes sobre a técnica de injeção de falhas, primeiro, é importante explicar o que é um sistema tolerante a falhas. Um sistema é dito tolerante a falhas quando, na presença destas, as funções continuam a ser executadas como foram especificadas. O principal objetivo é tornar o sistema imune a um conjunto de falhas e quando há a presença de uma delas, o sistema consiga isolá-la sem causar um defeito ou evitar que a execução continue de forma defeituosa [19].

A injeção de falhas serve para verificar como um determinado sistema alvo se comporta na presença de um conjunto de falhas que são artificialmente introduzidas para emular os problemas do ambiente real de operação do sistema. Com isso, é possível verificar se os mecanismos internos do sistema conseguem isolá-las de forma que não se manifestem como um defeito.

Dependendo da fase de desenvolvimento e, inclusive, da técnica utilizada, a injeção de falhas pode ser classificada em injeção de falhas por simulação e injeção de falhas em sistemas reais [2][10].

A injeção de falhas pode ser feita usando simulações dos modelos através de *software* ou através da emulação de *hardware*. As simulações por *software* podem testar o sistema utilizando diferentes níveis de abstração do modelo, por exemplo, o teste pode ser feito utilizando desde os modelos das portas lógicas e transistores até o nível funcional do sistema. A injeção de falhas por emulação de *hardware* utiliza modelos dos circuitos emulados através do uso de *Field Programmable Gate Array* (FPGA), que fornecem representações de *hardware* fiéis ao do sistema a ser testado [2].

A injeção de falhas em sistemas reais é feita com o uso de protótipos ou até mesmo do sistema final. Esta abordagem pode ser feita com o uso de ferramentas desenvolvidas como *hardware* ou *software*. A injeção de falhas por *hardware* utiliza ferramentas que criam perturbações ou até mesmo sinais falhos nos circuitos dos sistemas. Esta injeção pode ser feita através de conexões da ferramenta com os pinos dos circuitos ou com uso de radiações. A injeção por *software* injeta falhas no próprio *software* do sistema ou na camada entre ele e o sistema operacional. A injeção pode ocorrer tanto na fase de compilação quanto na fase de execução, sendo que na primeira, a falha é inserida no código fonte ou no *assembly*, entretanto, se feito na fase de compilação, há a desvantagem de não conseguir injetar novas falhas durante a execução do sistema alvo, pois exige que o sistema seja recompilado.

Apesar de haver várias técnicas para fazer a injeção de falhas, a escolha de qual utilizar vai depender muito do tempo e da verba disponível. A Injeção de falhas por simulação é vantajosa, pois ela dará uma análise prévia sobre a dependabilidade do sistema antes do desenvolvimento do sistema, porém ela é muito custosa e toma tempo do desenvolvimento,

já que ela exige um alto grau de detalhamento e exatidão do modelo para que o teste seja efetivo em obter um resultado fiel ao sistema real. Além disso, a injeção de falhas por simulação não isenta de fazer a injeção de falhas em sistemas reais, pois é necessário testar o sistema desenvolvido para certificar que durante o desenvolvimento e implantação não houve alguma falha. Caso ainda haja dúvida em qual abordagem utilizar, ainda há metodologias que misturam os dois tipos de injeção de falhas (simulação e sistema real), as abordagens mistas.

Neste trabalho, a técnica de injeção de falhas escolhida foi injeção de falhas em sistemas reais através de injeção por *software*. Esta decisão se deve ao fato do sistema a ser testado já ser um protótipo e, também, porque o custo de injeção por *hardware* ser muito maior do que a por *software*, pois exige a compra de ferramentas que são caras para este projeto.

2.6 Teste de robustez

Alguns sistemas, principalmente os críticos, devem possuir a habilidade de continuar executando suas funções corretamente em qualquer situação, até mesmo nas menos esperadas. Por exemplo, os sistemas de controle dos aviões são vitais para guiar o piloto durante o voo, por isso eles devem estar operando corretamente durante todo o percurso do voo. Para garantir que o sistema seja imune a certas falhas, o teste de robustez serve para verificar e avaliar se o sistema consegue tolerar ambientes estressantes e com possibilidades de falhas.

A robustez é um atributo secundário da dependabilidade que indica o grau no qual o sistema consegue operar corretamente mesmo na presença de dados de entrada inválidos ou ambientes estressantes [4][2]. Um dos objetivos do teste de robustez é conseguir ativar falhas, que resultem em defeitos, para que sejam corrigidos.

Basicamente, para executar os testes de robustez, os casos de teste têm como objetivo ativar o sistema sob teste com a execução de uma determinada operação e, em seguida, injetar falhas que podem levar o sistema a apresentar algum defeito. Neste cenário, existem três componentes que caracterizam uma abordagem de teste de robustez: a carga de trabalho, a carga de falha e a classificação dos modos de defeito encontrados[2].

A carga de trabalho é um conjunto de atividades que ativa o sistema sob teste para executar um trabalho que é normalmente executado no ambiente real de operação. A carga de falha é um conjunto de falhas, que é introduzida no sistema com o objetivo de causar um defeito. A classificação dos modos defeitos é utilizada como guia para identificar e classificar os defeitos que se manifestam devido à influência da carga de trabalho em conjunto com a carga de falhas.

Os testes de robustez podem ser adequados tanto para verificação quanto para avaliação[21]. No caso da verificação, são utilizados modelos do sistema para a extração dos

casos de teste, sendo esta uma abordagem clássica da área de testes. A avaliação é feita através do uso de carga de trabalho e carga de falhas, que são extraídas de análises que indicam quais trabalhos são normalmente executados no ambiente de operação e quais falhas são mais representativas, ou seja, tem maior chance de ocorrer.

As técnicas para fazer teste de robustez podem ser classificadas em três grupos: a primeira é focada na carga de trabalho, a segunda é focada na carga de falhas e a terceira é um misto das anteriores. Estas técnicas serão abordadas nas próximas subseções.

2.6.1 Abordagens usando carga de trabalho

Esta abordagem utiliza a carga de trabalho de forma que o sistema execute um número de tarefas além do que ele foi especificado. Assim, o sistema ficará sobrecarregado e terá todos seus recursos utilizados na sua capacidade máxima. O resultado servirá tanto para fins de verificação quanto avaliação. No caso da verificação, é analisado se o sistema reage conforme as especificações do sistema, enquanto na avaliação, são analisados os resultados de desempenho[21].

2.6.2 Abordagens usando carga de falhas

Esta abordagem serve para verificar e avaliar como o sistema se comporta em um ambiente hostil e com falhas. Ela é importante pelo fato de que qualquer sistema está sujeito a falhas e, por isso, deve-se avaliar se ele consegue atingir um nível aceitável de robustez [4].

As seções seguintes descrevem as técnicas de testes de robustez que focam na carga de falhas.

Injeção de falhas de *hardware*

Através do uso de injeção de falhas é possível simular no sistema, as consequências de uma falha de *hardware*. Isto torna possível verificar se os mecanismos de tolerância a falhas do sistema conseguem tolerar falhas nos componentes de *hardware* ou se surge algum defeito. Entre as técnicas utilizadas estão o uso de radiações, que podem alterar os bits de memória, o uso de interferências nos sinais e perturbações na fonte de alimentação[10].

As falhas de *hardware* também podem ser injetadas por *software* com o uso de *software implemented fault injection* (SWIFI). Atualmente, esta técnica é amplamente utilizada, pois ela não necessita de nenhum *hardware* extra, que normalmente custaria um valor alto. Esta técnica consegue emular falhas de *hardware* através de alterações nos valores das memórias, registradores e chamadas ao sistema [2].

Dados de entrada Inválidos

Este teste utiliza dados de entrada inválidos que geralmente são erros dos usuários ou chamadas de serviços com valores inválidos. O objetivo desta técnica é testar se o sistema consegue identificar e prevenir que estas entradas causem algum defeito que podem ser desde o congelamento do sistema até mensagens de erros inapropriados[21].

Entre as principais técnicas deste tipo de teste estão [2]:

- Dados de entrada aleatórios: onde os parâmetros e entradas são gerados aleatoriamente até encontrar, de forma exaustiva, defeitos no sistema.
- Entradas inválidas: este teste tem como objetivo usar valores inválidos nos parâmetros das chamadas dos sistemas. As ferramentas que utilizam esta técnica criam chamadas com valores excepcionais aleatórios, como, valores nulos, vazios, MAX_INT + 1 e entre outros.
- Teste para tipo específico: este teste é semelhante ao com entradas inválidas, porém ao invés de inserir valores excepcionais aleatórios, primeiro é feita uma análise da interface do sistema para verificar os tipos de cada parâmetro. Em seguida é feita uma decomposição dos tipos em classes válidas e inválidas. Os casos de teste são diversas chamadas ao sistema com os parâmetros combinados com valores válidos e inválidos. Como vantagem, este tipo de teste consegue diminuir o número de casos de teste, pois ele foca somente nos valores inválidos relacionados aos tipos de dados dos parâmetros.

Mutações no código do sistema

Esta abordagem altera o código fonte do sistema com modificações elementares. O objetivo é criar várias versões do sistema, onde cada versão gera um executável do sistema que emula uma falha.

Para fazer mutações no código, o mais comum seria injetá-las no código fonte e depois compilá-lo, entretanto há algumas abordagens que alteram o código do sistema em nível do código de máquina [21]. Isto significa que não é necessário compilar novamente o sistema para que novas falhas sejam injetadas, pois elas podem ser introduzidas no *assembly*.

2.6.3 Abordagens mistas

A abordagem mista foca tanto na carga de trabalho quanto na carga de falhas, pois, em média, 30% dos experimentos que focam somente na carga de falhas, não causam efeito algum no sistema e por isso acabam não sendo tão significativos para os testes e consomem recursos e tempo. Ao focar em ambas as cargas, os testes se tornam mais eficazes, pois

é feita uma análise de qual carga de falhas deve ser aplicada em determinada carga de trabalho. Por isso, o número de testes diminui e o tempo do experimento também [21].

Para que esta abordagem seja efetiva, os testes têm que ter como apoio o uso dos modelos do sistema, entretanto, caso isto não seja possível, é necessário que seja feita uma observação minuciosa da execução do sistema no ambiente normal de trabalho.

Entre as técnicas que utilizam esta abordagem estão [21]:

- Baseado em stress: esta técnica aplica uma carga de trabalho com o objetivo de sobrecarregar os componentes de *hardware* do sistema (o processador, a memória e as entradas e saídas). A injeção de falhas é feita no componente que estiver com nível de atividade mais alto ou acima da média. Isto melhora a taxa de ativação das falhas injetadas, porém não garante que todas serão ativadas.

- Baseado em caminhos: esta técnica utiliza uma pré-injeção de falhas em conjunto com a carga de trabalho. Assim, é possível analisar, no sistema, os caminhos que são ativados durante a execução. Posteriormente, esta análise será utilizada para filtrar somente as falhas que possuem maiores chances de serem ativadas.

- Colapso de falhas: usa-se esta abordagem quando os resultados de alguns experimentos já são conhecidos de antemão. Dessa forma, o número de experimentos desnecessários pode ser reduzido com a simples eliminação destes experimentos. Um exemplo do uso desta técnica é a alteração de um bit da memória antes de haver uma escrita nela. Neste caso, esta falha causa um resultado trivial, pois o valor desse bit não importa ao sistema enquanto não houver uma escrita nela.

2.7 Considerações finais

Neste capítulo foi dada uma visão geral sobre os conceitos básicos envolvidos nesta dissertação. Com este conhecimento será possível entender, nos capítulos seguintes, os trabalhos relacionados a esta pesquisa, a solução proposta para conseguir atingir os objetivos e finalmente, os resultados obtidos.

Capítulo 3

Trabalhos Relacionados

Este capítulo possui informações sobre os trabalhos relacionados à área de teste de robustez. A estrutura deste capítulo começa com uma descrição das metodologias mais conhecidas que, apesar de serem utilizadas para testar sistemas operacionais e não serem compatíveis com WSs, serviram como base para o desenvolvimento das metodologias mais recentes que se aplicam em WSs.

Em seguida será feita uma descrição dos requisitos necessários para testar a robustez dos WSs. Isto será importante para verificar se as metodologias e ferramentas existentes conseguem cumprir estes requisitos, pois caso contrário, não serão compatíveis para realizar o estudo de caso. Por fim, será feita uma análise das abordagens de teste de robustez existentes para WSs e será explicado o porquê de ter sido criada uma nova ferramenta.

3.1 Metodologias para teste de robustez

Dentre as metodologias para testes de robustez mais conhecidas estão a Ballista[12] e a MAFALDA[3]. Ambas foram desenvolvidas com intuito de testar sistemas operacionais, porém vale ressaltar que, apesar de terem uma aplicação alvo bem definida, estas metodologias possuem abordagens que são capazes de serem adaptadas para outros domínios de aplicações. Por isso, é importante conhecer as abordagens antecessoras para entender a aplicação para a qual ela foi criada e como elas evoluíram para conseguir testar novos domínios de aplicação.

Como foi abordado no Capítulo 2, o que caracteriza uma abordagem de teste de robustez são: a carga de trabalho, a carga de falha e a classificação dos modos de defeito. Nesta seção serão descritas as abordagens tanto da Ballista quanto da MAFALDA em relação a estas características de cada abordagem. Nesta seção, também serão descritos alguns estudos de caso em que elas foram utilizadas para validar a metodologia.

3.1.1 Ballista

A Ballista é uma metodologia simples, porém muito conhecida na área de teste de robustez. A abordagem consiste em criar chamadas de funções usando combinações de parâmetros com valores válidos e inválidos. Estes valores são atribuídos através do uso de valores armazenados em um repositório, cujo conteúdo é resultante da análise dos tipos de dados dos parâmetros das funções.

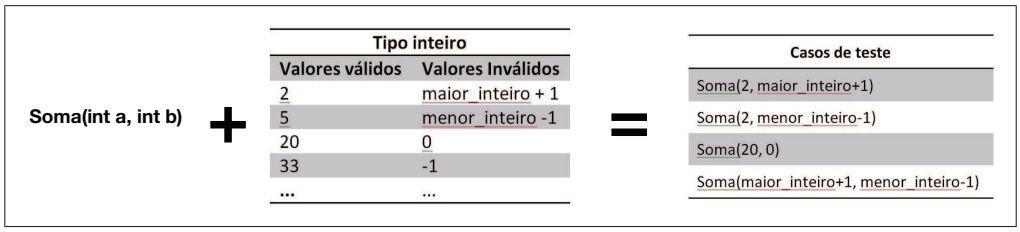


Figura 3.1: Abordagem da Ballista.

A Figura 3.1 exemplifica de uma forma simples como são formados os casos de teste. Neste caso, a função a ser testada é uma simples soma com parâmetros "a"e "b" do tipo inteiro. Com a análise dos tipos de dados da função soma, a Ballista gera um conjunto de chamadas a esta função com a combinação de valores válidos e inválidos obtidos do repositório de dados. Estes valores inválidos são gerados a partir de uma análise, que verifica quais são os valores referentes ao tipo de dado que causam mais defeitos. Por isso, o valor zero, apesar de não ser um valor inválido, dependendo do contexto, pode causar um defeito, por exemplo, uma divisão por zero. No fim, o produto gerado é um conjunto de casos de teste compostos de chamadas de funções.

Para se obter um resultado completo dos testes de robustez com a Ballista, o conjunto de casos de teste deve possuir todas as combinações distintas dos valores válidos e inválidos dos parâmetros das funções. Assim, os testes terão a cobertura total das falhas que a Ballista oferece. Mas, a vantagem da abordagem é que ela diminui significativamente o número de casos de teste, pois antes existe um filtro para selecionar somente as falhas relativas aos tipos de dados dos parâmetros.

Ao executar um caso de teste, o sistema alvo poderá apresentar um comportamento robusto, onde o serviço é oferecido corretamente ou senão, apresentar um defeito de robustez. Os modos de defeito da Ballista foram classificados usando a escala CRASH. Vale ressaltar que esta escala foi utilizada nos estudos de caso da ferramenta, cujos sistemas alvos eram *Application Programming Interface* (API) de sistemas operacionais. Por este motivo, estes modos de defeito são referentes ao domínio dos sistemas operacionais.

A escala CRASH, que classifica os modos de defeito, é um acrônimo para Catastrófico[1],

[1]Do inglês, *catastrophic*.

Reinicio[2], Aborto[3], Silencio[4] e Obstrução[5] respectivamente. Cada um representa um modo de defeito, sendo eles:

- Catastrófico: É um defeito quando ocorre uma pane geral no sistema operacional. O efeito observado pode ser corrupção nos dados, congelamento ou reinicio do sistema operacional.
- Reinicio: Ocorre quando a chamada a função nunca retorna o resultado, ou seja, a tarefa está "suspensa" e ela tem que ser reiniciada.
- Aborto: É observado quando o módulo que está sendo testado termina a tarefa de forma abrupta devido a algum problema interno.
- Silencio: É um defeito quando o sistema não retorna um aviso de erro, quando ele obrigatoriamente deveria ter retornado. Por exemplo, ocorre quando uma das falhas injetadas consegue fazer o sistema escrever em um arquivo de somente leitura, entretanto o sistema não retorna um aviso de erro ao usuário.
- Obstrução: Ocorre quando o sistema retorna um erro diferente do ocorrido. Isto é considerado um defeito, pois, além de estar fora da especificação do sistema, ele dificulta os mecanismos de tolerância a falhas de se recuperarem deste erro.

Os estudos de casos da Ballista foram as APIs dos sistemas POSIX[12] e Windows[22], onde foram testados 15 sistemas operacionais POSIX. As suítes de testes englobavam 233 chamadas de sistemas e dentre os resultados mais interessantes estão que seis funções apresentaram defeitos catastróficos e houve uma taxa de 6% a 19% de defeitos de silêncio.

Já os testes com APIs do Windows foram feitas para, além de procurar defeitos, comparar a robustez de diversas versões do Windows (95, 98, 98 SE, CE, 2000 e NT). Entre os resultados, foi observado que as versões para servidores (2000 e NT) apresentaram um comportamento mais robusto que as versões para usuários, entretanto diversos defeitos se repetiam em diferentes versões.

Em resumo, a Ballista é composta por uma carga de trabalho, que são as chamadas às funções; uma carga de falhas, que é composta por valores inválidos que serão atribuídas nas chamadas ao sistema alvo; e a classificação dos modos de defeito que utiliza a escala CRASH.

[2]Do inglês, *restart*.
[3]Do inglês, *abort*.
[4]Do inglês, *silent*.
[5]Do inglês, *hindering*.

3.1.2 MAFALDA

A *microkernel Assesment by Fault Injection and Design Aid* (MAFALDA) é uma metodologia para teste de robustez em sistemas operacionais baseados na tecnologia de *microkernel*. Estes sistemas operacionais possuem a característica de dividir em serviços cada módulo do sistema operacional (por exemplo, controladores de memória, comunicação, sincronização etc.), isto torna possível que sistema operacional seja composto por módulos desenvolvidos por diferentes vendedores de *software*. Com isto, o cliente possui também a possibilidade de escolher quais módulos serão integrados no sistema operacional.

Um dos fatos que levou a criação da MAFALDA foi essa heterogeneidade de serviços, onde cada módulo pode ser desenvolvido por diferentes vendedores de *software*. Em consequência disto, pode haver problemas de integração, compatibilidade ou até mesmo o comprometimento do sistema operacional por causa dos defeitos apresentados pelos serviços.

Diferente da Ballista, MAFALDA utiliza uma abordagem para testes de robustez em que a carga de falhas possui falhas que, além de emular parâmetros inválidos, também emula falhas de *hardware*. Os parâmetros inválidos são emulados através de corrupções pseudoaleatórias nos bytes que compõe as chamadas das funções, enquanto a falha física é emulada através de corrupções nos bytes de um determinado endereço de memória de um componente de *software*. Esta corrupção é feita nos segmentos de códigos e dados. Para ambos os tipos de falhas, elas são injetadas através de um ou mais *bit-flip*, que consiste em inverter o valor de um bit dentro de uma sequência de bytes.

Como o estudo de caso da MAFALDA foram sistemas operacionais que utilizam *microkernel* como tecnologia de design, a classificação dos modos de defeito é o seguinte:

- Erro detectado: O sistema operacional detecta o erro e o reporta para a camada de aplicação.
- Falência do Kernel: O *microkernel* congela, ou seja, por algum motivo o *microkernel* entra em laço infinito, *dead lock* ou espera um evento fictício.
- Erro propagado: O erro não é detectado pelos mecanismos de detecção de erros do *microkernel* e por isso o erro se propaga para a camada de aplicação.

Como resultado interessante deste estudo, além dos defeitos encontrados, foi observado que a injeção de falhas nos parâmetros das chamadas de funções conseguiu encontrar mais defeitos que a injeção de falhas de *hardware*.

3.2 Requisitos para testar robustez de WSs

As abordagens para testes de robustez apresentadas nas seções anteriores foram desenvolvidas principalmente para testes de sistemas operacionais e por isso, a carga de trabalho, a carga de falhas e os modos de defeito não se adaptam completamente para o ambiente de WSs. No entanto, muitos dos conceitos foram reaproveitados para a criação das novas metodologias para testes de robustez em WSs. Antes de entrar em detalhes sobre cada metodologia, primeiro serão descritos alguns dos requisitos necessários para se testar um WS. Estes requisitos foram baseados nas características da tecnologia e padrões dos WSs[24].

A primeira característica a se levar em conta é que WSs, geralmente, trabalham em servidores geograficamente distribuídos e por isso, nem sempre há acesso aos seus códigos fontes. Logo, a abordagem de teste deveria levar em conta que o serviço por muitas vezes será uma caixa preta, ou seja, somente as interfaces públicas poderão ser acedidas.

Deve-se levar em conta que os WSs possuem o ponto forte de ter a interface e a comunicação padronizadas por diversos protocolos. Por este motivo, a abordagem pode e deve aproveitar as descrições da comunicação e interface para criar uma abordagem generalizada, que analisa e cria automaticamente os casos de teste através da análise dos documentos (por exemplo, WSDL) fornecidos pelo serviço.

Existem diversos padrões para os WSs e uma característica desses padrões é que eles são extensíveis para incrementar e adicionar novas funcionalidades ao serviço, por exemplo, o WS-Security, que adiciona uma camada de segurança na troca de mensagem SOAP entre o serviço e o consumidor. Por isso a abordagem teria que levar em conta que os WSs têm a opção de adotar ou não as extensões. Consequentemente, a abordagem tem que ser compatível com essa flexibilidade de ter diferentes padrões sendo aplicados na comunicação. Inclusive, a abordagem deve ser apta a aceitar novas extensões que estão a surgir.

A abordagem deve apoiar o teste de serviços simples e compostos. No caso de teste de serviços simples, a complexidade da arquitetura é menor já que o teste será feito somente com um WS. No caso de serviços compostos, há uma complexidade maior e com isso um desafio de conseguir testar toda a composição em diferentes aspectos. Para explicitar este desafio, imagine que o cliente só enxerga a interface que fornece o serviço, mas por trás há todo um fluxo de trabalho e comunicação entre os serviços parceiros. Por isso, todo esse fluxo deve ser testado para garantir a robustez dos WSs compostos.

Entre outros desafios de se testar composições de WSs, existem as composições que são orquestradas e as que são coreografadas. No primeiro caso, todo fluxo de negócio é centralizado, ou seja, existe um processo central que coordena todos os serviços parceiros a fim de atingir o objetivo da composição. Já a coreografia não possui essa execução

centralizada, pois todos os WSs participantes conhecem os seus fluxos de serviços e sabem como devem interagir nas composições. Então, ao invés de depender do processo central para decidir quais os passos a ser seguido, o serviço executa a sua tarefa no seu devido tempo sem a necessidade de um coordenador.

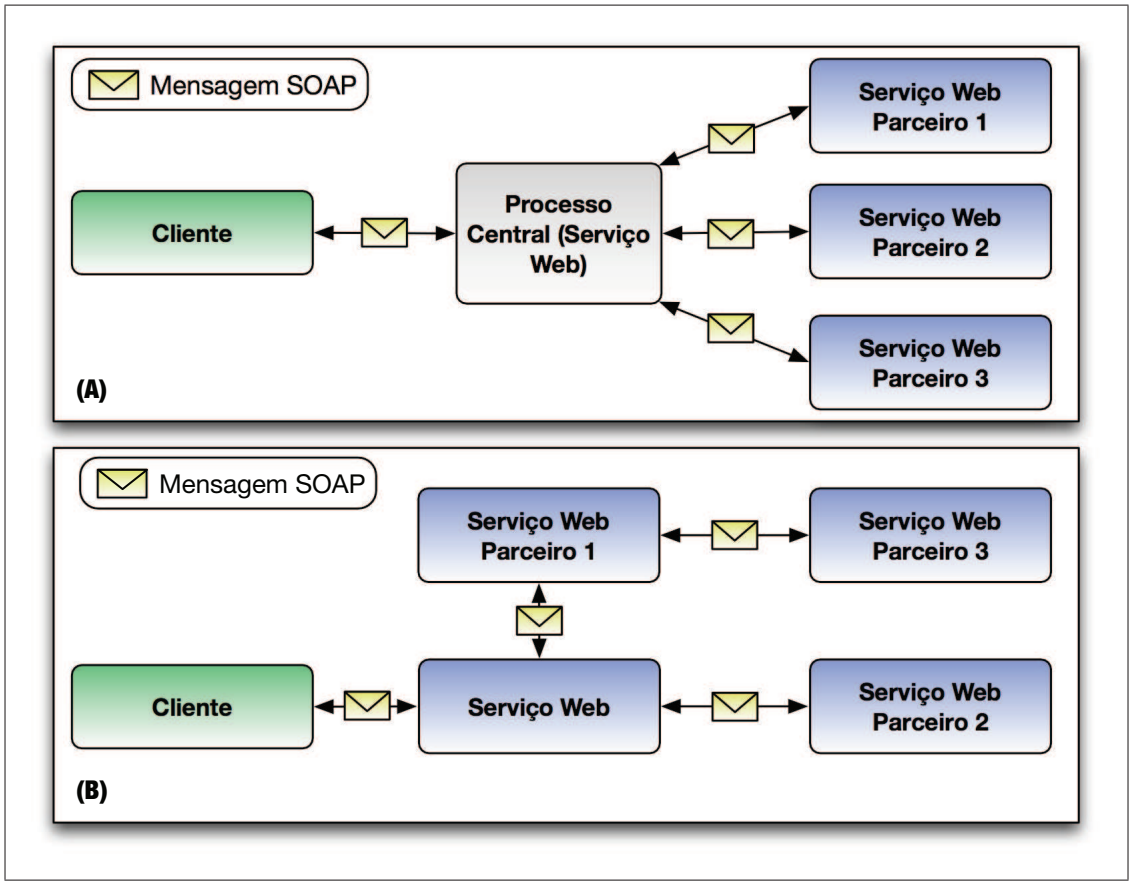


Figura 3.2: (A) Orquestração e (B) Coreografia.

A Figura 3.2 mostra um exemplo de orquestração e outro de coreografia. Note que a orquestração exige que todas as mensagens da composição sejam trocadas por uma figura central, neste caso chamado de processo central, então a estratégia de teste poderia focar nesta entidade que consome e coordena toda comunicação entre os WSs. No caso da coreografia, há uma complexidade um pouco maior já que a comunicação é mais distribuída, como é o caso do WS Parceiro 1 que faz comunicação isolada e direta com o WS Parceiro 3. Por isso a estratégia do teste de robustez tem que considerar que nem toda comunicação passa por uma entidade central, o que acarreta um aumento na complexidade dos testes.

Finalmente, o modelo de falhas deve emular os problemas de dependabilidade citados na Seção 2.4, pois são problemas que ocorrem no ambiente real de operação dos WSs.

Estes problemas podem ser tanto defeitos na comunicação, como atraso, duplicação nos dados, etc. quanto dados corrompidos ou inválidos.

3.3 Metodologias para teste de robustez em WSs

Esta seção possui a descrição de algumas metodologias e ferramentas para teste de robustez em WSs. Estes trabalhos correlatos foram encontrados em pesquisas bibliográficas e suas análises foram feitas de forma que fossem verificados quais requisitos são cobertos por cada um. No final será feita uma comparação destes trabalhos correlatos a fim de saber quais deles seriam mais adequados para o estudo de caso deste trabalho.

3.3.1 WebSob

A ferramenta WebSob[18] foi desenvolvida na Universidade Estadual da Carolina do Norte, EUA, com o objetivo de realizar testes de robustez em WSs. A metodologia desta ferramenta se baseia na automação de geração de código fonte do cliente, geração de testes, execução de testes e análises das respostas.

A geração de código fonte do cliente do WS é realizada através da análise do documento WSDL fornecido pelo serviço. Através deste documento, o WebSob analisa as informações de protocolos, formato da mensagem SOAP, operações oferecidas entre outros dados para gerar o cliente do WS. Este cliente servirá para executar os casos de teste e fazer a chamada ao WS.

A geração de testes também utiliza o WSDL para analisar todas as operações públicas do WS e seus respectivos parâmetros. Estas informações são utilizadas para gerar os casos de teste com o uso de uma abordagem semelhante à Ballista. Então, cada caso de teste é uma chamada a uma operação fornecida pelo serviço, cujos parâmetros são as combinações de valores válidos e inválidos para cada tipo de dado da chamada.

Com o cliente e os casos de teste gerados automaticamente, o WebSob utiliza o cliente para executar todas as chamadas contidas nos casos de teste, a fim de realizar os testes de robustez.

Ao fim da execução é feita uma análise manual das respostas, porém com algumas heurísticas, que selecionam as respostas que indicam problemas de robustez. Estas heurísticas têm o objetivo de diminuir o trabalho manual de análise. Além disso, para facilitar ainda mais a análise de resultados, também existe um filtro que é feito através de um monitor da ferramenta. Este monitor analisa o conteúdo das respostas e procura no conteúdo os códigos de erros referentes ao protocolo HTTP, que são:

- 404 (arquivo não encontrado): indica que a requisição do cliente chegou ao servidor,

porém este não encontrou o que foi requisitado ou foi configurado para não completar essa requisição.

- 405 (método não permitido): indica que o cliente não pode aceder a operação no WS referente à URL indicada pelo cliente.
- 500 (exceção interna do servidor): Este erro ocorre quando o servidor não consegue processar a operação executada pelo WS.
- Suspenso: O WS fica suspenso indefinidamente ou o servidor demora mais de 30 segundos para responder e por isso, é considerado que nenhuma resposta foi enviada pelo serviço.

Nos testes feitos com esta ferramenta, 35 WSs foram testados com milhares de requisições. Um ponto importante é que a análise de resultados se mostrou um desafio, pois é muito difícil definir se a resposta é uma mensagem de erro esperada ou um defeito de fato. Por isso o autor define como uma tarefa importante, a criação de uma abordagem mais sofisticada para análise de resultados de WSs.

3.3.2 Wsrbench

A ferramenta wsrbench[13] foi desenvolvida no CISUC da Universidade de Coimbra, Portugal. É uma ferramenta *online* que realiza testes de robustez em WSs. Ela é baseada em um trabalho anterior[25] desenvolvido na mesma instituição.

O trabalho anterior, que gerou a wsrbench, foi um trabalho que utiliza uma abordagem de injeção de falhas que posiciona a ferramenta de injeção de falhas entre o cliente e o WS, cuja função é atuar como um mediador. A Figura 3.3 mostra como funciona a configuração da comunicação do cliente e o WS.

Esta abordagem foi adaptada para realizar mutações nas mensagens trocadas entre o serviço e o cliente. Estas mutações são feitas nos parâmetros das requisições SOAP para simular a abordagem da Ballista, onde o parâmetro é modificado por um valor inválido e o resultado é uma mensagem SOAP corrompida, que é enviada ao WS. Essas mutações são feitas com base nos tipos dos parâmetros obtidos pelo WSDL do serviço.

Como este trabalho inicial se baseou no modelo de falhas da Ballista, a classificação dos modos de defeito também foi baseada na escala CRASH, porém neste contexto foi nomeado de wsCRASH, pois foi readaptada para WSs. Os modos de defeito são:

- Catastrófico: o servidor de aplicação congela ou reinicia.
- Reinicio: a execução do WS fica suspensa indefinidamente.

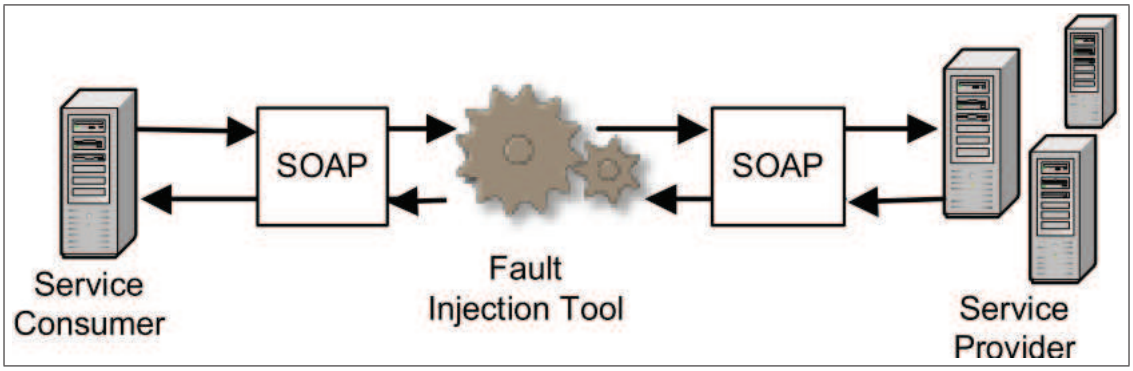


Figura 3.3: Configuração do ambiente de injeção de falhas[25].

- Aborto: término abrupto da execução do WS.
- Silencio: o tempo de conexão expira e nenhum erro é retornado, ou a operação não pôde ser realizada.
- Obstrução: o erro retornado pelo WS é incorreto ou a resposta é atrasada.

Após este trabalho, surgiu a wsrbench, cuja arquitetura e classificação dos modos de defeito foram modificadas para atender uma nova abordagem, que junta o cliente e a ferramenta de injeção de falhas em um mesmo componente. Este componente foi desenvolvido como uma ferramenta *online* disponível na Web para criar e realizar testes de robustez em WSs.

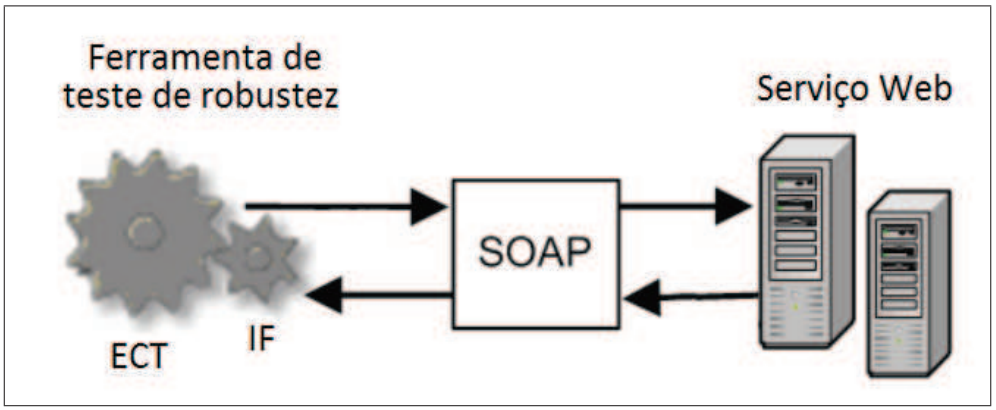


Figura 3.4: Nova abordagem da wsrbench[26].

A Figura 3.4 ilustra esta nova abordagem onde a ferramenta de teste de robustez é composta pelo executor da carga de trabalho (ECT) e o injetor de falhas (IF) no mesmo componente. A ferramenta se encarrega de consumir e injetar as falhas no WS.

A abordagem também consiste em utilizar o modelo de falhas da Ballista, onde o WSDL é analisado para colher informações referente às chamadas aos WSs, tais como operações, parâmetros e tipos de dados. Com estas informações, a wsrbench gera automaticamente a carga de trabalho e de falhas, sendo esta última composta de chamadas pertencentes à carga de trabalho com parâmetros contendo valores inválidos.

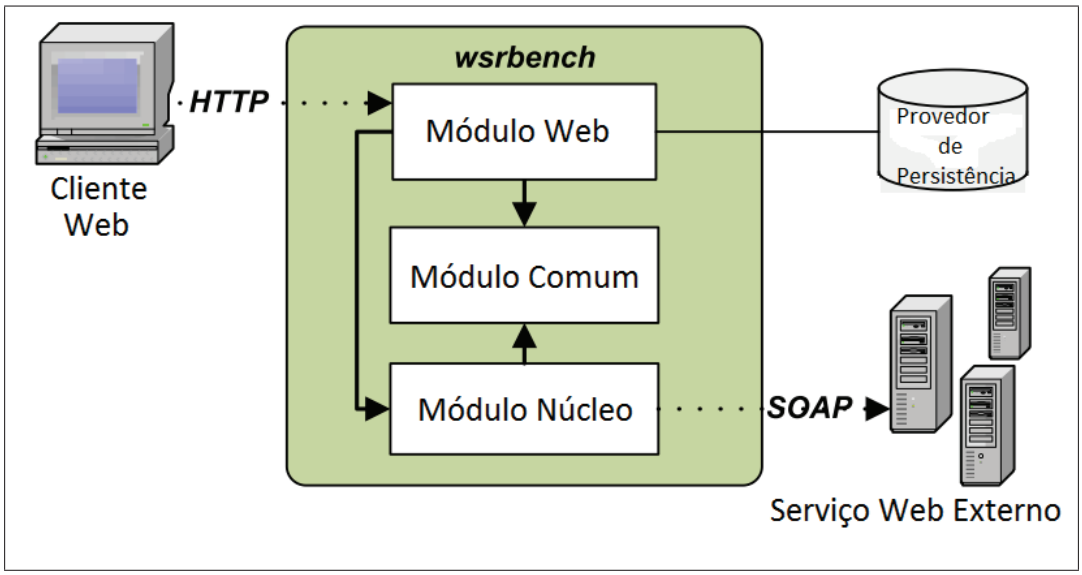


Figura 3.5: Arquitetura da wsrbench[13].

A Figura 3.5 mostra os módulos que fazem parte da arquitetura da wsrbench. Nesta arquitetura a ferramenta de injeção de falhas e o cliente do WS estão no mesmo componente e por isso, tanto a execução da carga de trabalho quanto à injeção de falhas são feitas pela wsrbench. Nesta figura, o Cliente Web é o usuário da ferramenta, que acessa e configura a wsrbench para realizar os testes de robustez. Os módulos que compõe wsrbench são:

- Módulo Comum: possui as classes que são reutilizadas pelos outros módulos. O objetivo deste módulo é manter um repositório de classes independentes para serem reutilizadas sem que haja dependência entre os desenvolvedores.
- Módulo Núcleo: possui as principais funcionalidades da ferramenta tais como geração de carga de trabalho, geração de falhas, análise de respostas entre outras funcionalidades.
- Módulo Web: provê funcionalidades avançadas que vão desde persistência de dados até criação e gerenciamento de *threads*, que é utilizado para simular um ambiente com diversos usuários acessando o WS.

Em relação à classificação dos modos de defeito, a escala wsCRASH foi reavaliada e simplificada para wsAS, onde os modos de defeito encontrados são os de aborto e silêncio. Esta simplificação ocorreu devido ao fato da wsbench atuar somente como um cliente e por consequência, ter um ponto de vista limitado dos defeitos. Logo, do ponto de vista do cliente, não há como classificar os defeitos catastróficos, reinícios e obstruções, pois o cliente não tem acesso ao servidor de aplicação no qual o serviço está rodando.

O estudo de caso da wsrbench contou com testes de robustez de 100 WSs públicos. Neste estudo foram encontrados 61 defeitos de aborto e nenhum de silêncio. Do total de WSs, 35% apresentaram defeitos de robustez.

3.3.3 WS-FIT

A WS-FIT[16][17] é uma ferramenta de injeção de falhas para análise de dependabilidade de WSs, que foi desenvolvida na Universidade de Durham, Reino Unido. Esta ferramenta se baseou na injeção de falhas em nível de rede e utiliza um método semelhante ao empregado na perturbação nos pacotes de rede.

A perturbação nos pacotes de rede se baseia em corrupções aleatórias nos bytes das mensagens que trafegam na rede[17]. A WS-FIT otimizou esta abordagem, pois consegue ser mais significativa nos locais onde são injetadas as falhas, que, neste caso, as falhas são injetadas nos parâmetros das mensagens SOAP. Além disto, a ferramenta consegue emular falhas de comunicação entre o WS e o consumidor do serviço.

Para tornar possível a injeção destes tipos de falhas, a WS-FIT utiliza uma abordagem que instrumenta a pilha de protocolos dos WSs. Com isto, é possível interceptar as mensagens de entrada e saída que passam pelos canais de comunicação. Assim, as mensagens são intermediadas pela WS-FIT e são passíveis de terem seus conteúdos corrompidos ou que sejam alvos de algum defeito no canal de comunicação.

A Figura 3.6 mostra um exemplo de como a WS-FIT funciona em um cenário onde existem três WSs: o termostato, a resistência e o controlador. Esta composição de serviços tem o objetivo de controlar a temperatura de um local através de trocas de mensagens entre os WSs. Para controlar a temperatura, o termostato mede a temperatura do local e, dependendo do valor medido, o controlador calcula a quantidade de energia que a resistência irá consumir para manter a temperatura desejada.

Como o exemplo é uma composição, então existe uma complexidade a mais na estratégia de como testar cada WS. Neste caso, o objetivo foi limitado a testar o serviço da resistência e por isso, a instrumentação foi feita somente no termostato. Isto possibilita injetar falhas nas mensagens que indicam a temperatura do local e consequentemente verificar como a resistência ira se comportar na presença de um termostato, que envia e recebe mensagens com falhas.

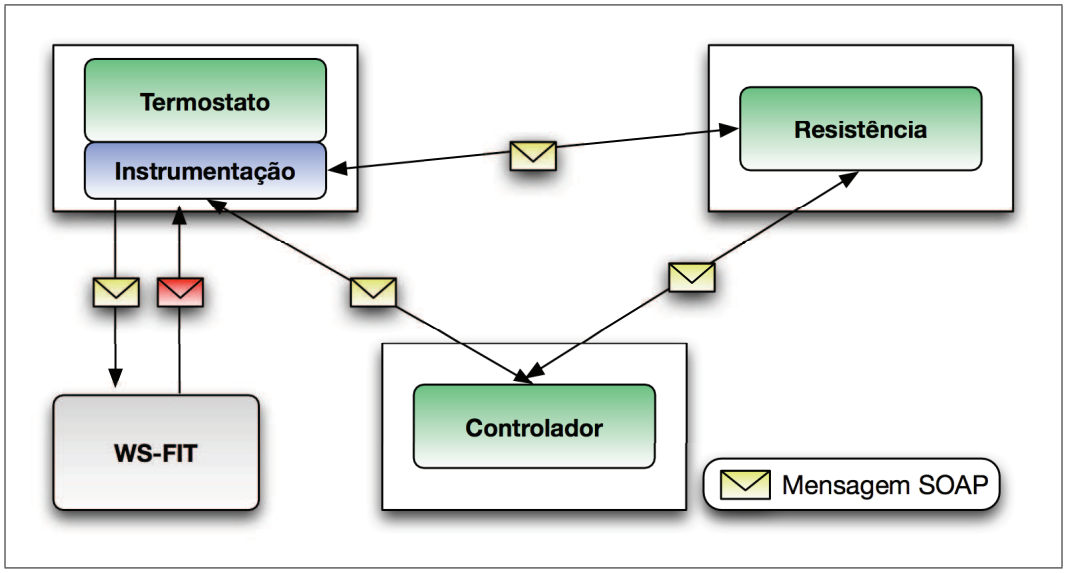


Figura 3.6: Exemplo de como a WS-FIT funciona.

Na Figura 3.6 a instrumentação faz com que as mensagens sejam interceptadas pela WS-FIT antes que elas sejam enviadas ou recebidas pelo termostato. Então, caso a mensagem seja alvo de alguma injeção de falhas, a mensagem é corrompida ao passar pela WS-FIT e em seguida ela é destinada ao serviço alvo.

O modelo de falhas da WS-FIT foi criado a partir de uma análise que estudou e selecionou as falhas mais recorrentes no ambiente de WSs. Nos trabalhos publicados foram listadas falhas de comunicação e mutação nos parâmetros das chamadas aos WSs. O modelo de falhas de comunicação contém falhas referentes à duplicação, omissão, corrupção e reordenação das mensagens.

Uma das características da WS-FIT é utilizar scripts para ativar a injeção de falhas, pois quando a ferramenta recebe a mensagem do WS, ela precisa ter meios de identificar a mensagem alvo e selecionar quais falhas devem ser injetadas. Para realizar esta tarefa, a ferramenta utiliza scripts que fornecem informações necessárias para reconhecer tais mensagens e, caso elas sejam reconhecidas, ativar "gatilhos" que iniciam o processo de injeção de falhas com as respectivas falhas descritas no script.

Os modos de defeito foram classificados observando os seguintes efeitos: suspensão da instância do WS, suspensão do servidor de aplicação, corrupção dos dados, duplicação de mensagens, omissão de mensagens e atraso de mensagens. Alguns destes defeitos são semelhantes às falhas injetadas pelo WS-FIT, isto se deve ao fato de que em uma composição de serviços, um dos serviços pode apresentar um defeito, e este propagar para os outros serviços. Este tipo de evento pode ser considerado uma falha para o outro serviço participante da composição. Por este motivo, a carga de falhas da WS-

FIT também considera que alguns defeitos dos WSs podem ser uma falha dentro de uma composição.

3.3.4 Comparação das ferramentas estudadas

Esta seção tem como objetivo comparar as ferramentas disponíveis para testes de robustez em WSs. Esta comparação é feita através da análise de quais requisitos, citados na Seção 3.2, são oferecidos pelas ferramentas.

Os requisitos discutidos na Seção 3.2 foram:

- Trabalhar em rede: os serviços devem utilizar protocolos de comunicação em rede, pois os servidores são remotos e os testes são feitos através da internet ou intranet.
- Ser compatível com a abordagem de caixa preta: muitas vezes os serviços não disponibilizam seus códigos fontes, logo o alvo do teste de robustez é uma caixa preta.
- Fazer a análise do WSDL: o WSDL contém informações importantes referentes a protocolos, operações, parâmetros e tipos de dados e por isso deve ser utilizado no processo de geração automático dos testes de robustez.
- Realizar testes de composições: a ferramenta deve ser compatível com testes em ambientes onde os serviços são compostos tanto por orquestração quanto coreografia.
- Ser compatível com extensões: os padrões que estendem as funcionalidades dos WSs estão em constante evolução, então é importante que a ferramenta consiga trabalhar com novos padrões, ou a evolução de um já existente sem que a ferramenta tenha que ser atualizada.
- Injetar falhas de comunicação e dados de entrada inválidos: para emular as falhas que os WSs estão suscetíveis, a carga de falhas da ferramenta deve ser compatível com ambos os tipos de falhas.

A Tabela 3.1 relaciona as ferramentas com os requisitos listados anteriormente. Na análise foi constatado que todas as ferramentas estudadas trabalham em rede, já que é um requisito mandatório.

Quanto a compatibilidade com a abordagem de caixa preta, tanto o WebSob quanto a wsrbench são ferramentas que conseguem atingir este objetivo. Isto se deve ao fato de ambos englobarem o cliente e o injetor de falhas dentro de um mesmo componente. Isto faz com que a ferramenta consiga gerar a mensagem SOAP e injetar as falhas como se

Tabela 3.1: Comparação das ferramentas.

Requisitos	**WebSob**	**wsrbench**	**WS-FIT**
Trabalhar em rede	Sim	Sim	Sim
Ser compatível com a abordagem de caixa preta	Sim	Sim	Não
Fazer a análise do WSDL	Sim	Sim	Sim
Realizar testes de composições	Não	Não	Sim
Ser compatível com extensões	Não	Não	Não informado
Injetar falhas de comunicação e dados de entrada inválidas	Não	Não	Sim

fosse um cliente normal do WS. A WS-FIT necessita de instrumentação no código fonte para conseguir injetar as falhas, por isso ela não consegue cumprir este requisito.

Em relação ao uso do WSDL para gerar os casos de teste, todos conseguem realizar, o que mostra que este é um requisito prioritário.

Como comprovado pelo cenário de teste da WS-FIT, somente ela realiza testes de composições de serviços. O WebSob e a wsrbench, no estágio atual, não realizam testes em composições, pois a abordagem limita que a configuração de teste seja feita somente ponto a ponto, ou seja, como as ferramentas se comportam como o cliente, a conexão da ferramenta é feita diretamente com somente um WS. Como consequência disto, eles são impossibilitados de que outros serviços sejam monitorados e testados ao mesmo tempo, pois necessitariam de uma abordagem para conectar com todos os serviços durante os testes.

Em relação à compatibilidade de novos padrões que estendem os WSs, o WebSob e wsrbench não conseguem atingir este requisito. Como ambas as ferramentas geram a mensagem SOAP, então ambas precisam de atualizações para realizar a criação de mensagens SOAP nos novos padrões. A WS-FIT, por interceptar a mensagem SOAP gerada pelo consumidor do serviço, não teria problemas de compatibilidade das novas extensões, já que o consumidor do serviço fica a cargo de gerar uma requisição SOAP válida. Porém, como não foi possível avaliar um protótipo da WS-FIT e não há informações publicadas sobre este requisito, na comparação foi mantido o status de "Não informado" para ela.

Nos estudos de casos da WebSob e wsrbench não foram mencionadas falhas de comunicação, somente a de dados de entrada inválidos. Já a WS-FIT possui injeção de falhas de comunicação, o que pode ser considerado um ponto forte no caso de se querer testar um ambiente cuja comunicação não possui um grau alto de confiabilidade.

3.4 Considerações finais sobre os trabalhos relacionados

A Tabela 3.1 mostra que as ferramentas estudadas não conseguem cobrir todos os requisitos coletados neste trabalho. Além disto, somente a wsrbench estava publicamente disponível para ser utilizada, pois as referências dos outros trabalhos estavam desatualizadas e não foi possível encontrar as ferramentas. Por causa destes motivos, foi necessário desenvolver uma ferramenta nova que cobrisse todos os requisitos necessários para realizar testes de robustez em WSs. Esta ferramenta foi batizada de WSInject[24][5].

Capítulo 4

WSInject

A WSInject é uma ferramenta que foi gerada a partir da necessidade de cobrir todos os requisitos da Seção 3.2 para realizar os testes de robustez em WSs. Com base nos trabalhos relacionados, algumas características e funções foram adotadas ou aprimoradas de forma que fosse possível cumprir estes requisitos.

A WSInject foi um produto gerado a partir da colaboração entre três alunos de pós-graduação, sendo dois de mestrado e um de doutorado, com seus respectivos orientadores. Os autores da ferramenta podem ser encontrados nas referências [24] e [5].

Este capítulo possui uma breve explicação desta ferramenta.

4.1 Abordagem

Nos estudos relacionados foram observadas duas abordagens de injeção de falhas. A primeira foi a baseada em mediador e a outra baseada em cliente. A diferença entre as abordagens é que na injeção de falhas baseada em mediador, a ferramenta de injeção de falhas fica posicionada como intermediária na comunicação entre o consumidor e o WS (por exemplo, WS-FIT e o primeiro estágio da wsbench), enquanto na baseada em cliente, o cliente engloba o consumidor e a ferramenta de injeção de falhas no mesmo componente (por exemplo, wsrbench e WebSob).

A Figura 4.1 ilustra a diferença entre a abordagem baseada em mediador (a) e a baseada em cliente (b). No caso da abordagem baseada em mediador, o cliente envia a mensagem SOAP para a ferramenta de injeção de falhas e esta injeta as respectivas falhas na mensagem, para que em seguida, a mensagem SOAP com falhas seja entregue ao WS destino. No caso da abordagem baseada em cliente, o consumidor já envia a mensagem SOAP contendo as falhas direto ao WS.

Como foi visto nas seções anteriores, a injeção de falhas baseada em cliente não é uma boa opção para realizar injeção de falhas em composições de serviços, por este motivo

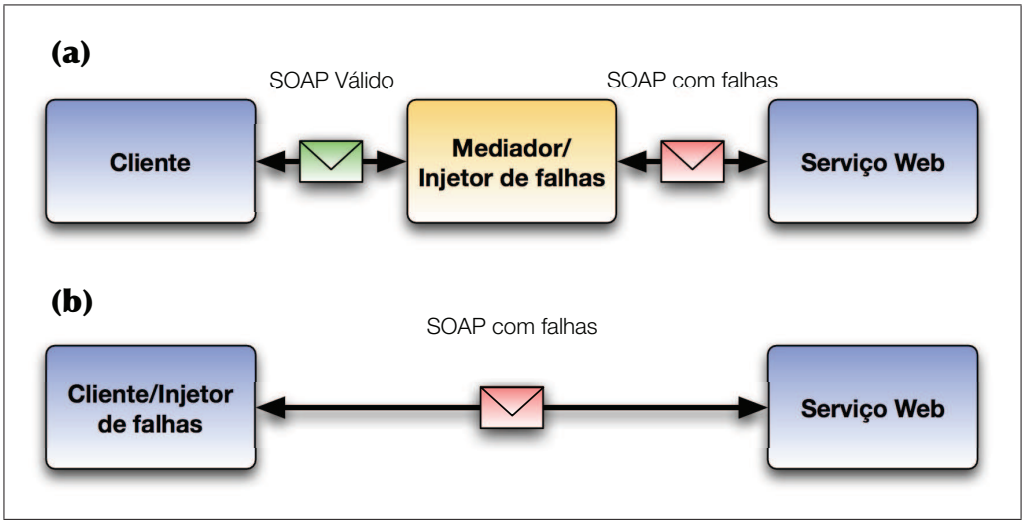


Figura 4.1: (a) Baseada em *proxy* (b) Baseada em cliente.

ela foi descartada como opção de arquitetura da WSInject. Entretanto, a baseada em mediador da WS-FIT exige instrumentação de código, o que gera um impedimento na adoção da abordagem da WS-FIT.

A solução encontrada foi criar um injetor de falhas que fosse acoplado em um *proxy* de rede. O uso de um *proxy* de rede não necessita de instrumentação no código, pois ele consegue mediar a comunicação do consumidor com o WS, somente com uma mudança de configuração nas propriedades do consumidor. Os clientes teriam somente que configurar o endereço e a porta do *proxy* para que a comunicação entre ele e o WS seja intermediada. Como esta configuração não exige a alteração no código fonte, ambos o cliente e o serviço se mantêm como uma caixa preta.

Com esta abordagem a WSInject consegue mediar a comunicação entre o consumidor e o WS inclusive em ambientes de composição. Além disso, ela também permite que sejam feitas injeções com falhas de comunicação e com dados de entrada inválidos. As falhas de dados de entrada inválidos são injetadas nas mensagens através de mutações nos valores dos parâmetros. Já as falhas de comunicação são emuladas através do *proxy* de rede que, por mediar todo canal de comunicação, consegue emular as falhas no nível de aplicação. Com isto, os problemas citados na Seção 2.4 podem ser emulados com o uso da ferramenta.

Para configurar a ferramenta de forma que ela consiga identificar as mensagens alvos da injeção de falha, a ferramenta utiliza a leitura de scripts. Estes scripts fornecem informações em alto nível, que indicam quais são as mensagens alvos e quais falhas devem ser injetadas. A geração destes scripts pode ser feito tanto manualmente, quanto automaticamente através da análise do WSDL do WS.

Tabela 4.1: Comparação das ferramentas correlatas com a WSInject.

Requisitos	**WebSob**	**wsrbench**	**WS-FIT**	**WSInject**
Trabalhar em rede	Sim	Sim	Sim	Sim
Ser compatível com a abordagem de caixa preta	Sim	Sim	Não	Sim
Fazer a análise do WSDL	Sim	Sim	Sim	Sim
Realizar testes de composições	Não	Não	Sim	Sim
Ser compatível com extensões	Não	Não	Não informado	Sim
Injetar falhas de comunicação e dados de entrada inválidas	Não	Não	Sim	Sim

Como é mostrado na Figura 4.1, o cliente é independente da ferramenta de injeção de falhas, isto é, a geração da mensagem SOAP é desacoplada do injetor de falhas. Caso o WS exija uma mensagem SOAP que utilize extensões com novas funcionalidades, a ferramenta não precisa ser atualizada para ser compatível com estas extensões, pois isto fica delegado ao cliente. A ferramenta fica a cargo somente da análise e injeção de falhas. Dessa forma o cliente pode ser trocado facilmente, pois ele não é fortemente acoplado a ferramenta.

As características da WSInject citadas nesta seção, juntas, conseguem cumprir todos os requisitos listados na Tabela 3.1. Se comparada com as outras ferramentas, o resultado fica como apresentado na Tabela 4.1.

4.2 Arquitetura e desenvolvimento

A arquitetura da WSInject foi baseada em um estudo, que publicou um padrão arquitetural das ferramentas de injeção de falhas [14]. Este estudo levou em conta que a arquitetura deve, além de injetar falhas, monitorar o sistema sob teste; ativar o sistema; controlar o processo inteiro; informar o usuário sobre os resultados dos testes; e aceitar configurações do usuário para saber quais falhas injetar e onde injetar.

O estudo identificou que uma ferramenta de injeção de falhas deveria ter os seguintes subsistemas:

- Ativador: é componente que ativa o sistema sob teste para executar uma carga de trabalho, que representa o trabalho do ambiente real de operação do sistema alvo do teste.

- Injetor de falhas: é componente que injeta as falhas.

- Monitor: é componente que monitora o sistema para avaliar como o sistema esta se comportando.
- Controlador: é componente que controla todos os outros subsistemas para que eles trabalhem coordenados.
- Interface com usuário: é componente que faz a interface com o usuário.

Baseado nestes subsistemas, a WSInject foi desenvolvida com objetivo de possuir esta arquitetura básica, onde os componentes são desacoplados e por isso, mais fáceis de serem desenvolvidas por múltiplos desenvolvedores. Apesar da arquitetura proposta por [14] ser genérica para diversos injetores de falhas, para a WSInject foi necessário estendê-la para conseguir injetar falhas em WS. O resultado é a arquitetura da Figura 4.2.

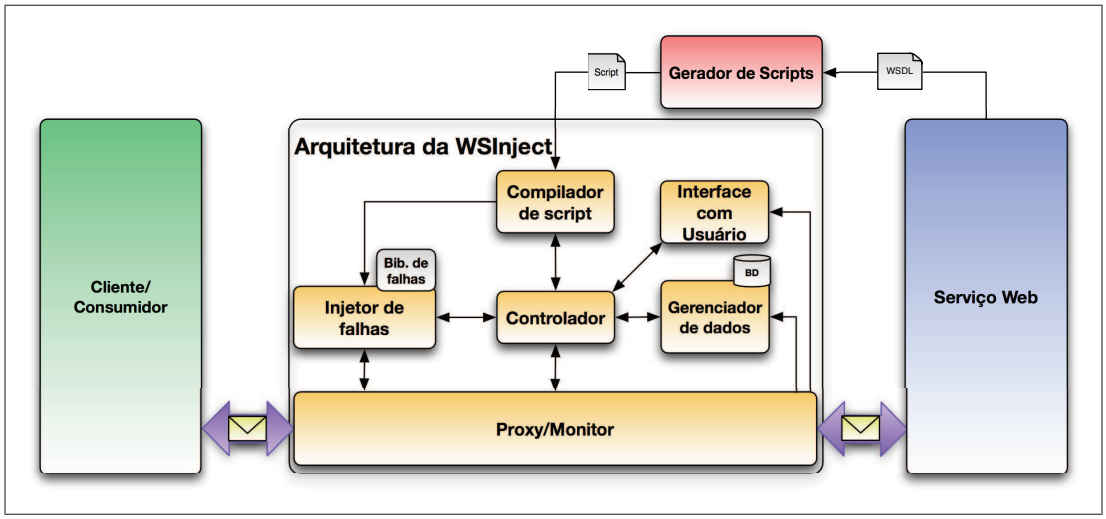


Figura 4.2: Arquitetura da WSInject.

A arquitetura da WSInject foi adaptado para conter os seguintes componentes:

- *Proxy*/Monitor: este componente tem o objetivo de interceptar as mensagens SOAP que trafegam entre o consumidor e o WS. É através da interceptação das mensagens que é possível aplicar a abordagem da WSInject, de corromper as mensagens e injetar falhas de comunicação. Além disso, por ter controle sobre a comunicação, este componente também monitora todos os eventos que ocorrem nas mensagens que trafegam.
- Gerenciador de dados: componente que gerencia os dados coletados pelo *Proxy*/Monitor.

- Compilador de script: este componente serve para transformar um script, escrito em linguagem de alto nível, em uma estrutura de dados que configura a WSInject. Com isso é possível para WSInject identificar as mensagens e injetar as suas respectivas falhas.

- Injetor de falhas: este componente recebe a configuração obtida através da compilação do script do usuário e realiza de fato a identificação e injeção de falhas das mensagens SOAP obtidas através do *Proxy*/Monitor.

- Interface com o usuário: este componente serve para configurar e mostrar as informações obtidas pelo *Proxy*/Monitor. A Figura 4.4 mostra a interface gráfica.

- Controlador: coordenador de todos os componentes, ele é responsável pela instanciação e controle de sincronização de todos componentes da arquitetura.

- Gerador de scripts: a carga de falhas é criada automaticamente pelo Gerador de scripts. Este componente analisa o documento WSDL do WS e através dele cria um conjunto de scripts contendo os casos de teste.

Para mostrar como estes componentes se sincronizam para inicializar a WSInject, a Figura 4.3 é um diagrama de sequência que mostra as trocas de mensagens entre os componentes da arquitetura. O primeiro passo é a compilação do script, cujas informações são utilizadas para configurar o injetor de falhas. Este, por sua vez, é incorporado no *Proxy*/Monitor para que, ao passar uma mensagem SOAP, o injetor consiga interceptá-la e injetar as falhas.

A Figura 4.4 mostra a interface gráfica da WSInject que, para explicar a função de cada componente visual, a figura foi destacada em grupos onde cada grupo engloba um conjunto de funcionalidades, que são:

1. O menu principal da ferramenta, cujas opções vão desde inicializar o *proxy* a escolher o script de configuração da WSInject. As opções *Log* e *Database* são funcionalidades que, respectivamente, abre um arquivo de eventos da ferramenta e recupera os dados dos testes executados anteriormente.

2. Uma tabela onde ficam armazenadas as conexões que foram interceptadas pela ferramenta. Nele o usuário pode verificar o horário da conexão, o endereço do WS e se houve algum erro.

3. Botões que gerenciam quais conexões serão exibidas na tabela do item 2.

4. Campos de textos que mostram as mensagens das requisições e respostas. O painel da esquerda mostra a mensagem original, enquanto o da direita mostra a mensagem com as falhas.

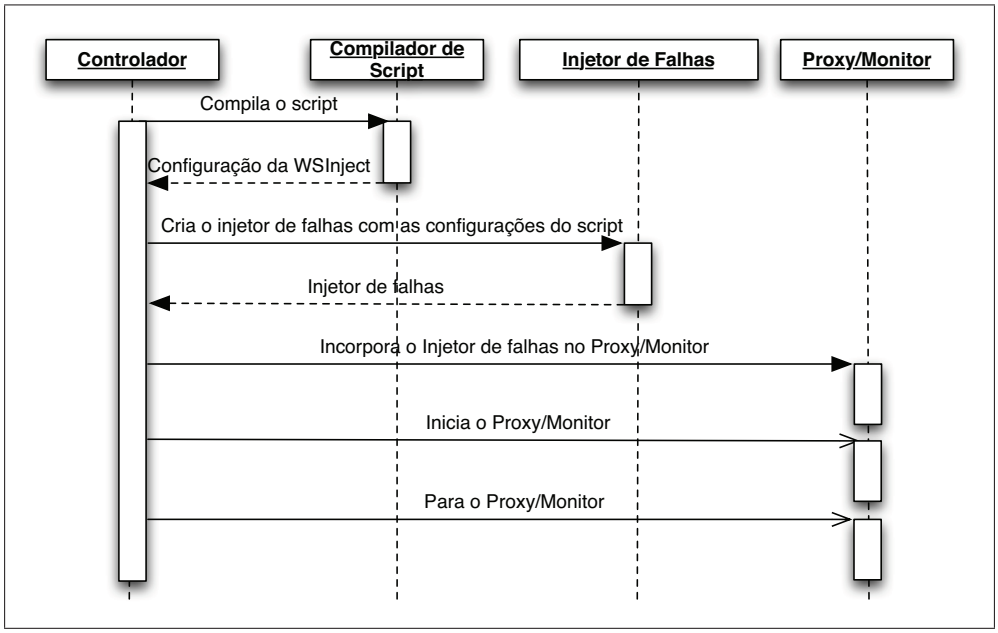


Figura 4.3: Diagrama de sequência da inicialização da WSInject.

5. É um painel de informação que indica o estado da ferramenta, se o *Proxy*/Monitor foi inicializado e se o script foi carregado.

O componente ativador (cliente), que ativa o sistema sob teste com a carga de trabalho, neste caso, foi decidido ser um componente à parte da ferramenta e é um componente que fica a cargo do usuário escolher. O cliente pode ser desde um gerador automático de carga de trabalho, quanto o cliente real da aplicação. A primeira opção fica para os casos em que se deseja automatizar todo o processo de teste, enquanto a última fica para os casos em que o usuário quer manter a representatividade do sistema, ou seja, manter o ambiente de trabalho o mais próximo do real possível.

Os scripts são arquivos de texto simples contendo uma ou mais expressões de injeção de falhas. Cada expressão é um conjunto de comandos que especificam qual a mensagem a ser injetada e com quais falhas. Para isso, cada expressão é um conjunto de condições e ações, sendo as condições um conjunto de diretivas para identificar a mensagem através de, por exemplo, o endereço do destino da mensagem. Já as ações são as falhas que serão injetadas caso a mensagem seja encontrada.

Para exemplificar uma expressão de falhas, em alto nível, uma expressão seria um conjunto de condições e ações que, caso uma mensagem com destino *www.serviçoweb.com* seja encontrada, deve-se injetar uma falha de atraso seguido de corrupção dos dados. Na linguagem de script, este exemplo ficaria da seguinte forma: *uri("www.serviçoweb.com") && isRequest(): Delay(), xPathCorrupt("//arg0", null)*;

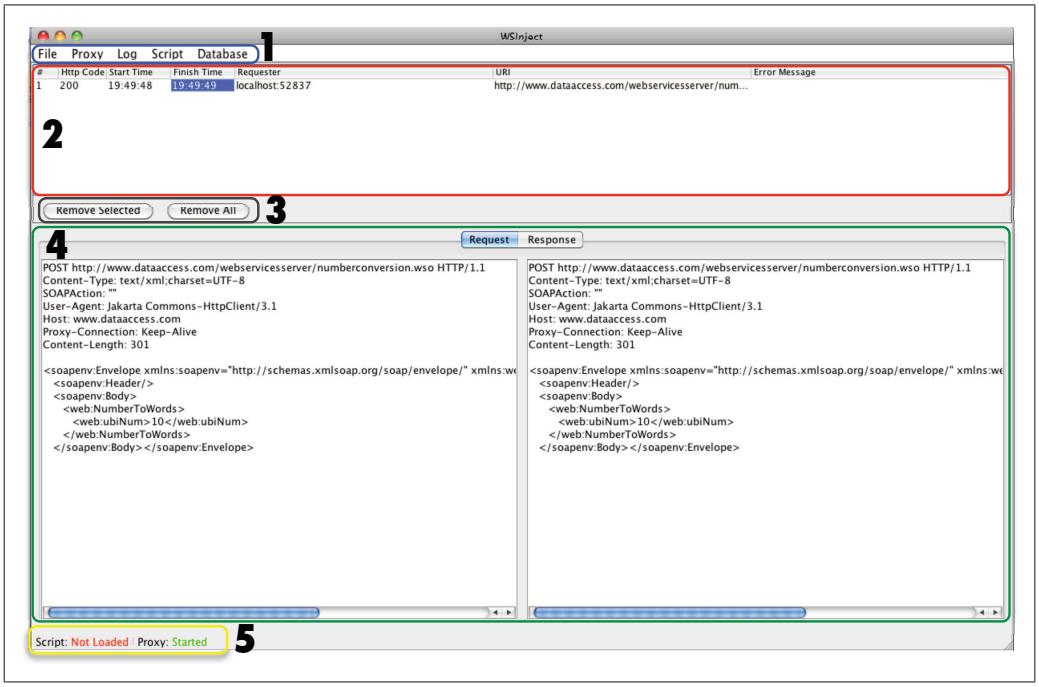


Figura 4.4: Interface gráfica da WSInject.

Para ser possível criar estes scripts automaticamente, a WSInject conta com um componente que analisa os dados da WSDL e extrai quais as operações o serviço Web fornece. Com estes dados, o componente Gerador de scripts consegue extrair as condições necessárias para identificar a mensagem (por exemplo, a operação e o nome do serviço) e também os tipos de dado de cada parâmetro, tornando assim possível alterá-los com valores inválidos.

Vale ressaltar que a linguagem de programação escolhida para desenvolver a WSInject foi o Java. Uma das características resultantes desta decisão foi a portabilidade da ferramenta, que pode ser executada em qualquer sistema operacional desde que este tenha suporte à máquina virtual Java.

4.3 Considerações finais

A WSInject foi um produto resultado de um trabalho em equipe. Por isso é necessário listar as contribuições deste trabalho para o desenvolvimento desta ferramenta, que foram:

- Análise de requisitos: listagem dos requisitos da ferramenta, que foi discutido e criado em conjunto com todos os autores da ferramenta.
- Arquitetura e abordagem da ferramenta: a criação do modelo conceitual com todos

os componentes necessários para cumprir os requisitos da ferramenta foi discutido e criado por todos os autores da ferramenta.

- Desenvolvimento dos componentes: Proxy/Monitor, Gerenciador de dados, Compilador de script, Interface com o usuário, Controlador e Gerador de scripts são contribuições deste trabalho.

A WSInject, como qualquer outra ferramenta de injeção de falhas, causa uma intrusividade no ambiente de testes, por isso é importante salientar que, como todas as mensagens são interceptadas pela ferramenta, deve-se levar em conta que este processo pode causar um atraso nas mensagens. Ainda não foi feito uma análise completa sobre o impacto desta interferência, porém, para os resultados obtidos neste trabalho, o atraso não foi significativo o suficiente para invalidá-los.

As publicações [24] e [5] possuem mais detalhes sobre a ferramenta.

Capítulo 5

Estudo de caso

Este capítulo descreve o estudo de caso feito para testar a robustez do Archmeds na arquitetura do projeto BioCORE. Este estudo teve dois objetivos, o primeiro é avaliar o grau de robustez da arquitetura de ferramentas do BioCORE com o Archmeds e validar a abordagem da WSInject.

5.1 Projeto BioCORE

O BioCORE[1] é um projeto desenvolvido no Laboratório de Sistemas de Informação (LIS) no Instituto de Computação da Universidade Estadual de Campinas (Unicamp) e conta com a parceria de pesquisadores de outros institutos como IME-USP, UNIFACS e do Instituto de Biologia da Unicamp e USP. Este projeto tem o objetivo de desenvolver ferramentas computacionais com a finalidade de auxiliar os pesquisadores da área de biodiversidade a gerenciar e compartilhar dados, assim como ajudar a criar modelos complexos de ecossistemas com suas relações e interações entre espécies.

A Figura 5.1 representa a arquitetura proposta para o projeto BioCORE. Ele é composto por WSs, que são agrupados em quatro camadas, sendo elas: armazenamento, serviços de suporte, serviços avançados e aplicação cliente. Para explicar como funcionam estas ferramentas, um breve fluxo de trabalho será utilizado como exemplo.

Resumidamente, um caminho simples de uso do cliente seria aceder à interface Web para fazer a consulta desejada. Esta interface fará com que o processador de consultas converta estas consultas em requisições, que são enviadas aos WSs avançados e suportes, para que em seguida, os dados dos repositórios sejam recuperados, processados e enviados de volta para o cliente[11].

Um exemplo de WS do BioCORE é o Aondê, com operações para armazenamento, ge-

[1]Informações sobre o projeto em http://www.lis.ic.unicamp.br/projects/biocore.

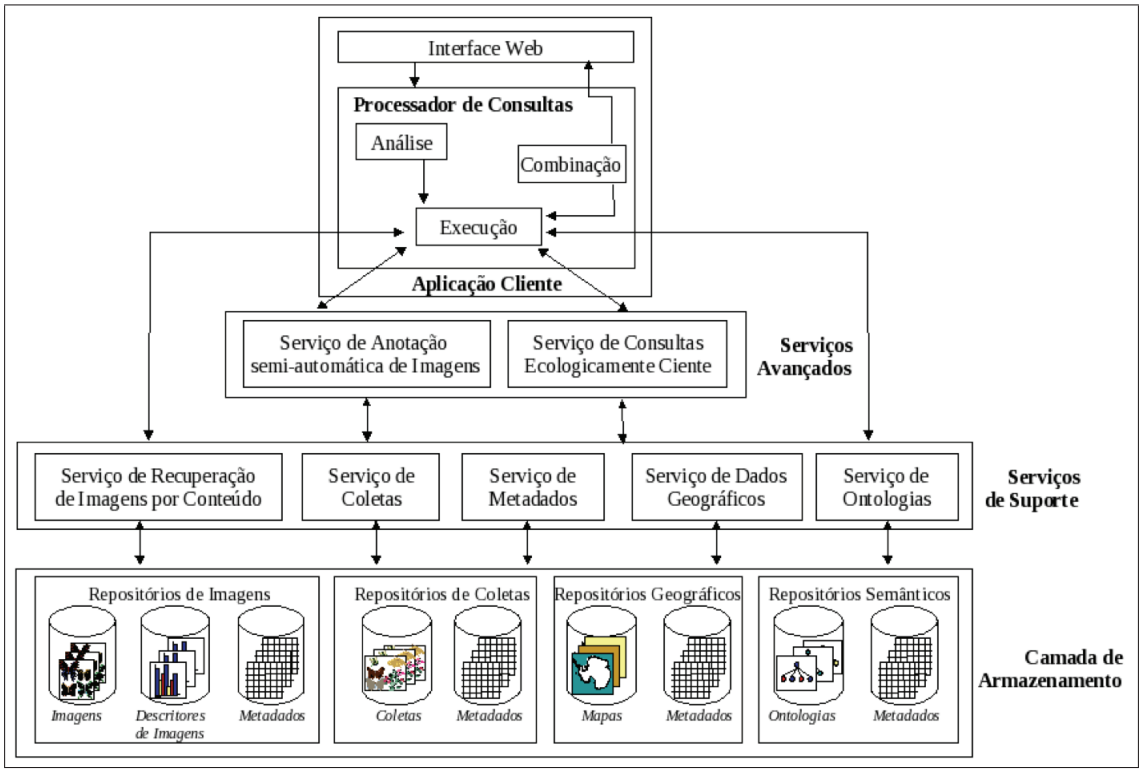


Figura 5.1: Arquitetura do BioCORE [8] apud [11].

renciamento, busca, *ranking*, análise e integração de ontologias. Estas ontologias são modelos de dados que representam um conjunto de conceitos de um domínio e seus relacionamentos[6].

Um dos indicadores de sucesso do projeto BioCORE é conseguir fazer com que este conjunto de ferramentas fique disponível e confiável para os clientes a todo momento. Para isso, foi feita uma pesquisa com objetivo de criar uma nova camada na arquitetura, que aplica algoritmos de tolerância a falha e consequentemente, aumenta a disponibilidade e confiabilidade dos WSs. O produto dessa pesquisa foi o Archmeds, uma infraestrutura confiável para arquiteturas baseadas em WSs, que será descrito na próxima seção.

5.2 Archmeds

O Archmeds [9][8] é uma camada que adiciona tolerância a falhas na arquitetura do BioCORE. O que ele faz é mediar a comunicação entre o consumidor e o WS e ao encontrar alguma falha, trabalha para que esta falha não comprometa a disponibilidade do serviço requerido.

A Figura 5.2 mostra a arquitetura do BioCORE com a camada do Archmeds, que faz

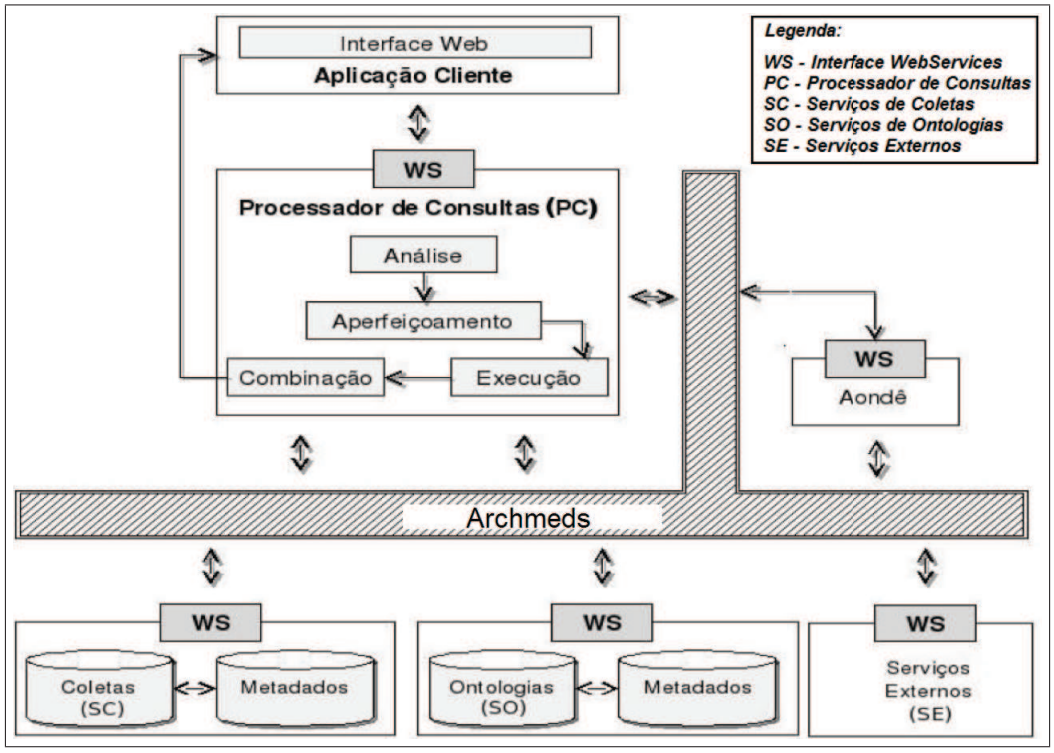


Figura 5.2: Arquitetura do BioCORE com Archmeds [8].

a mediação da comunicação entre os WSs e os consumidores. Nesta arquitetura, antes de um serviço ser consumido, a requisição passa pelo Archmeds. Em consequência disso, a requisição pode ser pré-processada a fim de aumentar a chance de ela chegar ao WS e ser respondida.

Para exemplificar essa mediação confiável, a Figura 5.3 mostra um diagrama de sequência com a mediação do Archmeds na interação com os WSs. Neste diagrama de sequência existem duas requisições, a primeira mostra uma requisição onde não há falhas e a segunda, uma requisição onde existe um defeito na comunicação entre o Archmeds e o WS 1. No segundo caso, o Archmeds identifica que o WS 1 apresentou um defeito e por isso, ele envia a mesma requisição para um WS redundante, neste caso, o WS 2. Isto faz com que o cliente receba uma resposta mesmo que o WS principal esteja fora de serviço.

Atualmente, o Archmeds possui duas estratégias para tolerância a falhas, o N-versões e os blocos de recuperação. Ambos os mecanismos são baseados em redundância dos WSs, entretanto, cada um trata a tolerância a falhas de forma diferente[8].

A estratégia dos blocos de recuperação consegue aumentar a disponibilidade dos WSs através da continuidade do serviço mesmo na presença de falhas em um deles. Esta estratégia é aplicada através da substituição de componentes defeituosos por outros que estão funcionando corretamente. A Figura 5.4 (a) ilustra como os blocos de recuperação

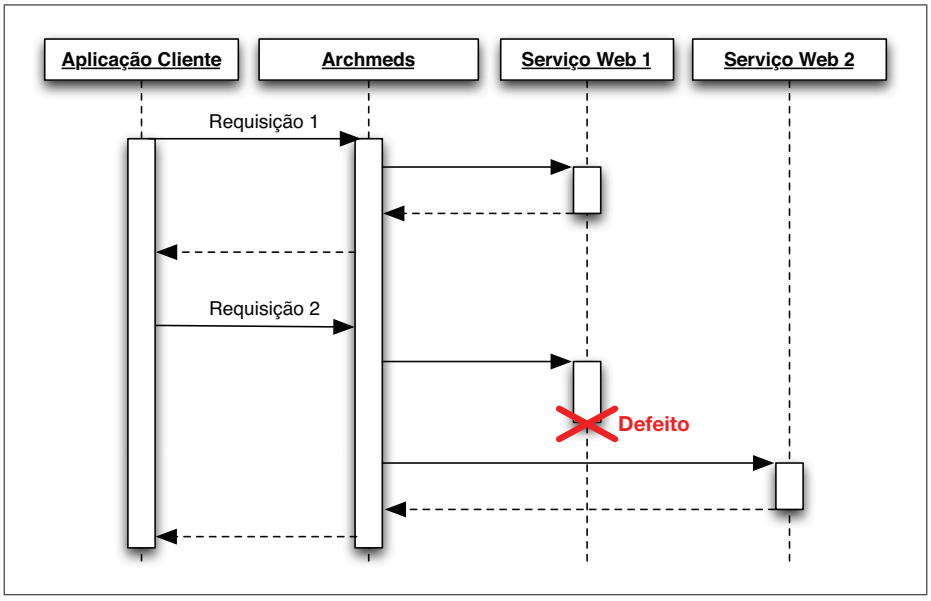


Figura 5.3: Diagrama de sequência de uma mediação confiável.

funcionam. Primeiro, ele envia a requisição para o WS 1, caso este falhe, a requisição é enviada para o WS 2 e assim por diante até que um dos WSs responda com uma mensagem correta. Vale ressaltar que, no Archmeds, esse mecanismo é transparente para o cliente, pois este envia a requisição ao Archmeds e recebe uma resposta sem que tenha que atuar na execução do mecanismo de tolerância a falhas entre Archmeds e o WS.

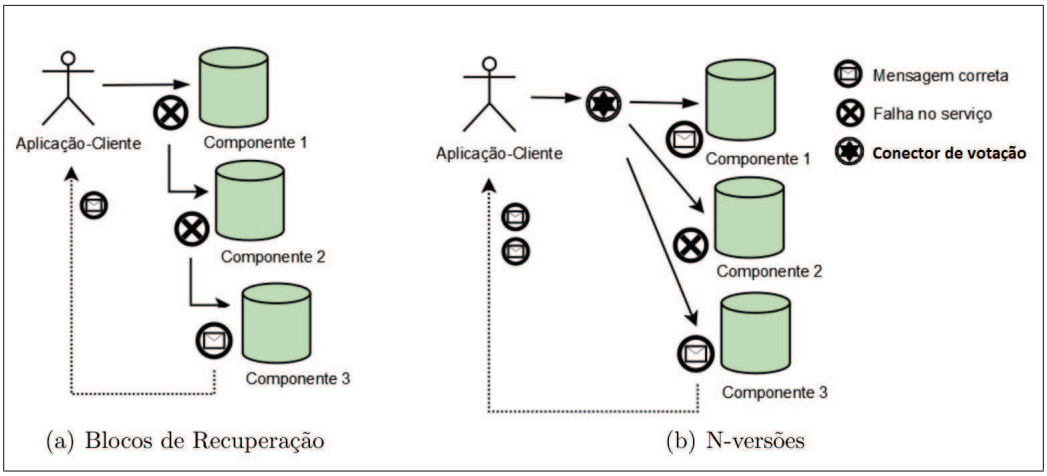


Figura 5.4: Estratégias de tolerância a falhas [8].

Na estratégia do N-versões, a redundância é feita através do uso de dois ou mais WSs em versões diferentes. Neste caso, a requisição é enviada a todos os WSs e os resultados são comparados para detectar desvios da especificação dos serviços. Esta análise de

resultados é feita através de um conector de votação que escolhe uma resposta seguindo alguns critérios pré-estabelecidos. Com isso espera-se, além de aumentar a disponibilidade, aumentar a confiabilidade do WS. A Figura 5.4 (b) ilustra um exemplo do uso do N-versões. Neste caso a mensagem é enviada aos três WSs, entretanto somente os WSs 1 e 3 retornam uma resposta.

O Archmeds utiliza uma arquitetura baseada em WSs para fornecer a mediação confiável. Como ocorre com qualquer outro WS, o cliente precisa consumir as operações do Archmeds através de trocas de mensagens SOAP. Entretanto, esta abordagem foi criada com o uso de extensões que ampliam as funcionalidades de um WS comum. Normalmente, os WSs não alteram seu estado quando ocorre uma comunicação entre cliente e servidor, ou seja, internamente a mesma instância do WS atende todas as requisições vindas dos clientes. Em contrapartida, existe também a possibilidade de salvar o estado do WS para cada cliente. Isto é possível através da criação de uma instância do serviço para cada cliente, onde ele pode ser configurado e seus estados alterados para atender individualmente cada requisição. Isto torna possível que o cliente consiga enviar uma configuração ao WS e este processar as requisições seguintes do cliente conforme o que foi configurado.

Para o WS possuir esta funcionalidade de salvar o estado é necessário que ele utilize o WS-Addressing[27], especificado pela World Wide Web Consortium (W3C), que possui a finalidade de estender as mensagens SOAP com a inclusão de um conjunto de campos no cabeçalho. Este conjunto identifica a instância do WS que irá processar a requisição do cliente, inclusive, um dos campos é uma identificação única que é atribuída ao cliente para ele identificar a sua instância do WS.

Para ser possível usar o WS-Addressing, o cliente deve ser compatível com esta extensão, senão ele não consegue consumir o serviço.

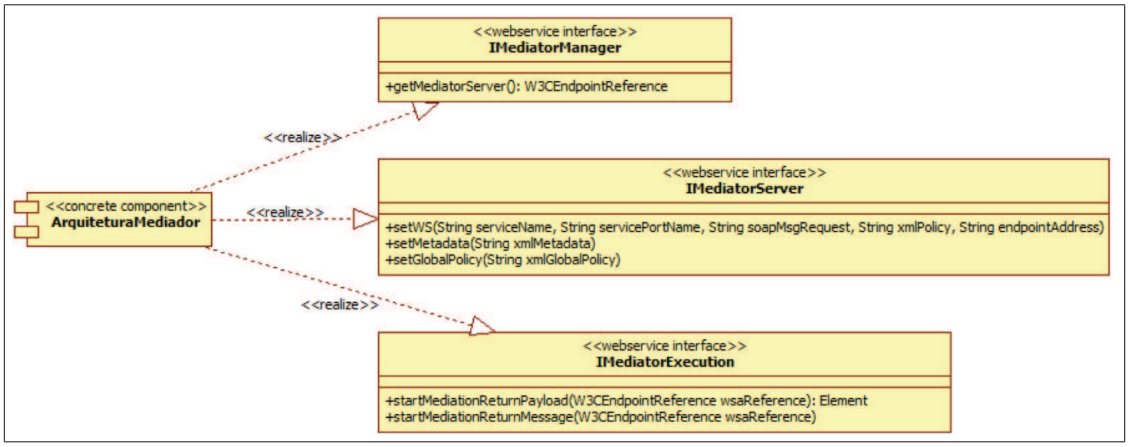


Figura 5.5: Interfaces públicas que compõe o Archmeds.

Para realizar a mediação confiável, a arquitetura do Archmeds é composta por três

WSs: o IMediatorManager, o IMediatorServer e o IMediatorExecutor. A Figura 5.5 mostra as interfaces de cada serviço que compõe o Archmeds.

O IMediatorManager é o ponto inicial para utilizar o Archmeds. O papel deste serviço é ser basicamente uma fábrica de objetos. Nele, o cliente consome uma única operação provida, o getMediatorServer(), que cria uma instância do Archmeds e retorna o número de identificação do serviço para o cliente.

O IMediatorServer é o próximo serviço a ser utilizado. Ele é gerado pelo IMediatorManager e por isso, o cliente utiliza a identificação que ele recebeu no passo anterior para aceder a sua instância do IMediatorManager. Neste serviço é feita a configuração do Archmeds para que ele receba as informações necessárias para realizar a mediação confiável. Para conseguir receber estas informações, o serviço fornece as seguintes operações: o SetWS(), que é utilizado para informar quais são os WSs redundantes; a operação setMetadata(), que informa o grau de dependabilidade dos WSs; e a operação setGlobalPolicy(), que informa qual mecanismo de tolerância a falhas será utilizado.

Finalmente, após o Archmeds estar configurado, o último serviço a ser consumido é o IMediatorExecution. O cliente envia a requisição ao serviço com o número de sua instância para que em seguida, seja executada a mediação confiável conforme a configuração do IMediatorManager. Existem duas operações neste serviço, o startMediationReturnPayload() e o startMediationReturnMessage(), onde é retornado ou a carga útil da mensagem SOAP, ou a mensagem SOAP respectivamente.

Para exemplificar esta sequência de trocas de mensagens, a Figura 5.6 é um diagrama de sequência que ilustra como o Archmeds funciona no ambiente do BioCORE. Neste diagrama estão presentes dois serviços do Aondê e o Cliente. É importante observar a interação entre os componentes, pois eles se comportam como uma composição de WSs, que trocam mensagem entre si para fornecer os serviços do Aondê de forma confiável. Isto causa um impacto na forma como será conduzido o estudo de caso, como será visto nas próximas seções.

5.3 Arquitetura do ambiente de testes

Para fazer este estudo de caso, a plataforma de testes foi composta de três computadores, que foram distribuídos na rede de forma a emular um ambiente real de operação do Aondê com o Archmeds, a Figura 5.7 ilustra esta configuração. O Computador 1 foi usado para emular a máquina do cliente, enquanto os outros dois emulam servidores que proveem os serviços do Archmeds e Aondê. Nesta arquitetura de testes foi utilizada a rede local do laboratório de sistemas distribuídos (LSD) do Instituto de Computação da UNICAMP.

Como foi explicado na Seção 4.1, a WSInject possui uma abordagem que deixa o cliente fracamente acoplado da injeção de falhas. Isto permitiu que, ao invés de usar o cliente

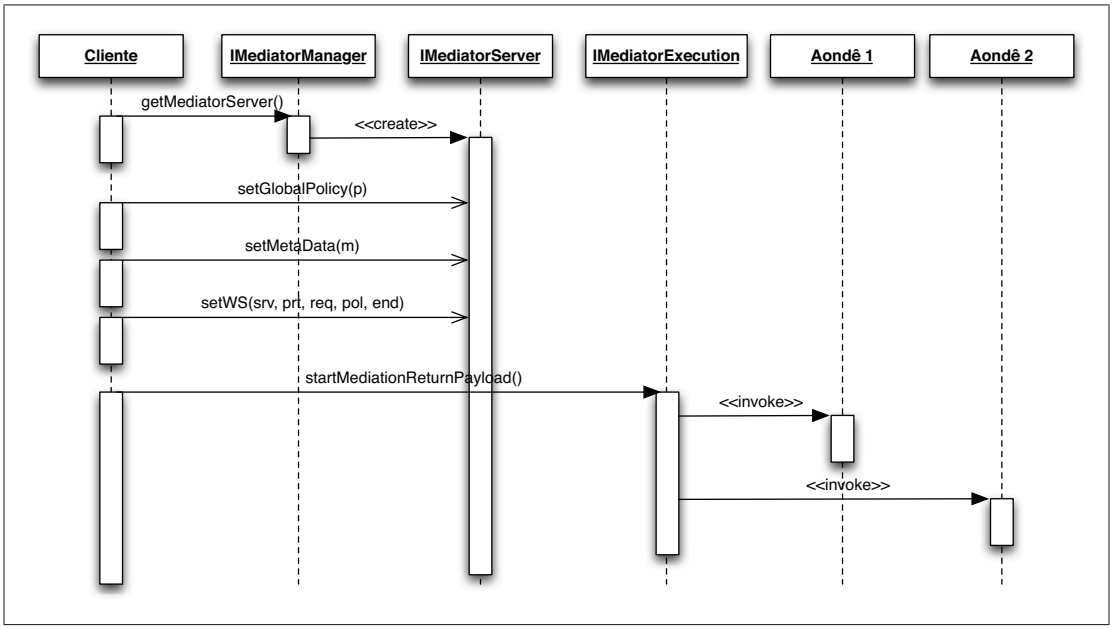


Figura 5.6: Diagrama de sequência do funcionamento do Archmeds.

real da aplicação, fosse utilizado o soapUI[2] como gerador de carga de trabalho e executor. O soapUI é uma ferramenta de testes de WSs, que possui diversas funcionalidades que facilitam tanto a geração de carga de trabalho, quanto a análise de resultados, pois a ferramenta conta com geração automática da mensagem SOAP através do WSDL e possui assertivas que indicam se o teste falhou ou não.

Os Computadores 2 e 3 possuem uma instância do Aondê cada. Isto permite emular dois serviços redundantes do Aondê espalhados em computadores diferentes pela rede. O servidor de aplicação utilizado para prover o Aondê foi o Tomcat/5.5.29 da Apache Software Foundation.

O Archmeds ficou instalado no Computador 2 utilizando o servidor de aplicações Glassfish/2.1.1 da Sun Microsystem.

A WSInject foi instalada na mesma máquina do cliente, porém toda a comunicação dos WSs e consumidores foram configuradas para enviar e receber as mensagens SOAP através do WSInject. Por isso, toda troca de mensagem SOAP, que ocorre na rede, passa antes pela WSInject para que em seguida seja enviada ao destinatário.

A Figura 5.8 mostra, de uma forma lógica, a arquitetura de testes. Todas as mensagens são trocadas com a mediação da WSInject, isto foi feito propositalmente para que todas elas pudessem ser alvo de uma injeção de falhas. Com isso, é possível que os serviços da composição sejam testados com falhas tanto no envio quanto no recebimento de mensagens

[2]http://www.soapui.org/.

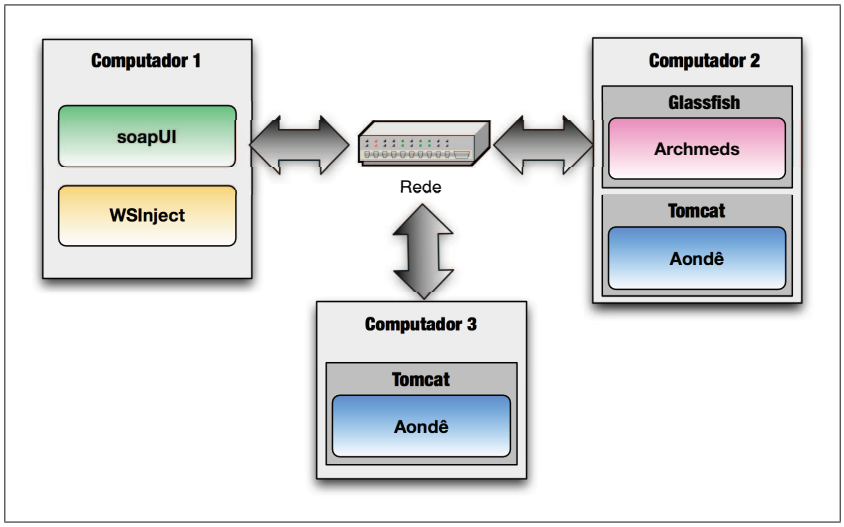


Figura 5.7: Configuração da arquitetura de testes.

SOAP.

5.4 Campanhas de teste

Como foi explicado na Seção 5.3, a carga de trabalho é gerada e executada pelo soapUI. A criação da carga de trabalho é feita através da geração de requisições SOAP, que são criadas automaticamente pela ferramenta. A única parte manual é preencher os parâmetros com valores válidos que são utilizados para emular um ambiente real de operação dos sistemas. E como resultado, a carga de trabalho é composta de requisições que executam todos os passos descritos na Figura 5.6, da instanciação do mediador até a configuração e execução da mediação confiável.

A fim de testar os serviços, foram feitas três campanhas de injeção de falhas. A Campanha 1 foi criada com o propósito de testar os serviços do Archmeds utilizando falhas de dados de entrada inválidos. A Campanha 2 foi criada com o objetivo de injetar falhas de comunicação na interação entre o Archmeds e os serviços Aondê. E finalmente, a Campanha 3 com objetivo de injetar falhas no Aondê com e sem o Archmeds. Este último tem o objetivo de mensurar qual o ganho quantitativo na confiabilidade do serviço do Aondê com o uso dos mecanismos de tolerância a falhas do Archmeds.

Todas as campanhas foram executadas pelo menos três vezes. Isto foi feito para verificar se os resultados das campanhas possuíam alguma diferença em uma das execuções. Isto foi importante para validar se o defeito poderia ser repetido e se o ambiente de testes não causava alguma interferência.

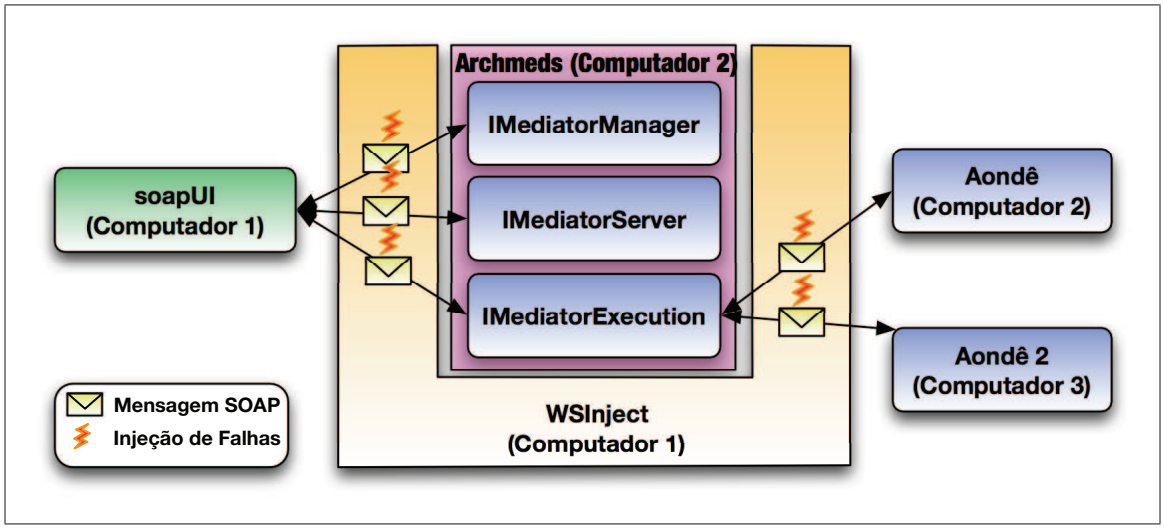


Figura 5.8: Visão lógica da arquitetura de testes.

Tabela 5.1: Campanha 1.

WS	Operação	Parâmetros	Casos de Teste
IMediatorExecution	StartMediationReturnPayload	3	13
IMediatorServer	SetWS	5	6
	SetMetaData	1	6
	SetGlobalPolicy	1	30

5.4.1 Campanha 1

Na Campanha 1 foi utilizada uma abordagem semelhante ao da wsrbench e seu intuito era de injetar falhas nos valores dos parâmetros das operações do Archmeds. Para isso, cada caso de teste foi gerado automaticamente com a análise do WSDL do Archmeds e, em seguida, utilizado para a criação dos scripts da WSInject. Nesta campanha não foram utilizadas falhas de comunicação, pois este tipo de falha é mais efetivo para testar o cliente consumidor do que o provedor do serviço. Isto se deve ao fato de que, se algo acontecer na comunicação, a consequência só poderá ser vista no cliente, pois o serviço estará remoto e muitas vezes o cliente não tem acesso para saber o seu estado. Além disto, no caso de haver uma falha de comunicação, o WS não saberá se uma mensagem SOAP do cliente chegou com atraso ou se alguma outra mensagem foi omitida.

Os WSs que fazem parte do Archmeds são o IMediatorManager, o IMediatorServer e o IMediatorExecution, onde cada serviço possui uma ou mais operações. O IMediatorManager possui somente a operação getMediatorServer(), que provê a criação de uma instância do IMediatorServer. No entanto, por esta operação não possuir parâmetros de entrada, as mensagens destinadas a esta operação não foram injetadas com falhas.

Tabela 5.2: Exemplos de casos de teste para injeção de falhas em tipo de dado literal.

Exemplos de casos de teste
uri("MediatorServerServices") && isRequest(): xPathCorrupt("//arg0", null);
uri("MediatorServerServices") && isRequest(): xPathCorrupt("//arg0/text()", "423jh45j4kj");
uri("MediatorServerServices") && isRequest(): xPathCorrupt("//arg0/text()",);
uri("MediatorServerServices") && isRequest(): xPathCorrupt("//arg0/text()", "\n\r\t\b\f");

Tabela 5.3: Campanha 2.

WS	**Casos de teste**
Aondê (Computador 2)	3
Aondê (Computador 3)	3
Aondê (Computador 2 e 3)	3

Em relação aos outros serviços, a Tabela 5.1 relaciona as operações fornecidas pelos WS com os parâmetros de entrada e total de casos de teste criados. As falhas consistiam basicamente em fazer mutações nos parâmetros das chamadas ao WS com valores inválidos. Neste caso, os parâmetros eram na sua maioria do tipo literal. Então os parâmetros foram alterados por valores como nulo, vazio, caracteres aleatórios e caracteres inválidos para emular falhas na mensagem SOAP.

A Tabela 5.2 possui quatro exemplos de casos de teste para injetar valores inválidos em um parâmetro do tipo literal. Estas expressões são descritas dentro dos scripts, que são usados para configurar a WSInject. Nestas expressões, o termo uri("MediatorServerServices") serve para indicar qual o destino da mensagem SOAP; o isRequest() indica que a mensagem é uma requisição e não uma resposta; e o xPathCorrupt("//arg0", null) indica que a falha será a injeção do valor nulo no argumento arg0 da chamada.

5.4.2 Campanha 2

A Campanha 2 usou falhas de comunicação baseadas no modelo de falhas da WS-FIT (isto é, perda de conexão, omissão de mensagens e atrasos). O objetivo foi testar o comportamento do Archmeds na situação onde a comunicação entre o Archmeds e o Aondê possui falhas.

A Tabela 5.3 possui a relação da quantidade de casos de teste para cada instância do WS Aondê nos computadores do ambiente de teste. Assim, o Archmeds é testado para analisar o seu comportamento quando cada computador possui uma falha de comunicação, ou quando ambos possuem falhas.

Tabela 5.4: Campanha 3.

WS	Operações	Casos de teste
WSOperationsService	6	184
WSSemanticService	17	213

5.4.3 Campanha 3

Na Campanha 3, foram injetadas falhas no Aondê, porém com e sem o Archmeds. Os testes sem o Archmeds foram analisados para determinar quantos casos de teste causam defeitos no Aondê. Nos testes com Archmeds, as falhas eram injetadas nas mensagens que tramitavam entre o Archmeds e o Aondê, porém somente em um dos serviços redundantes. Os resultados destes testes foram analisados para avaliar se o Archmeds conseguiria funcionar corretamente quando o Aondê apresentasse algum defeito.

A Tabela 5.4 relaciona a quantidade de operações com os casos de teste por WSs do Aondê. Neste caso, os parâmetros das requisições destinados ao Aondê foram alterados para valores inválidos conforme o tipo de dado, semelhante à Campanha 1. No total foram 387 casos de teste, porém executados duas vezes cada, pois a primeira parte foi executada com a mediação confiável e a segunda sem.

5.5 Resultados

Para analisar os resultados, o soapUI foi utilizado para fazer uma pré-análise automática das mensagens SOAP. Esta tarefa foi feita com o auxilio de assertivas que indicavam se a mensagem era válida ou se era uma mensagem de erro. A Figura 5.9 ilustra a tela do usuário, onde a área destacada (1) mostra a mensagem SOAP de requisição e a resposta, enquanto a área destacada (2) mostra quais assertivas passaram. Esta interface gráfica é uma forma interativa e manual de configurar o soapUI, porém para a execução automatizada dos casos de teste, ela não foi utilizada, pois a execução foi feita com arquivos de execução em lote.

Os defeitos foram classificados através de uma análise manual e usando as categorias do wsAS, citado na Seção 3.3.2. A escolha do wsAS ao invés do proposto pela WS-FIT se deve ao fato da WS-FIT ser específica demais quanto aos defeitos e não conseguir classificar alguns dos defeitos observados, pois não era compatível com os modos de defeito listados.

A Tabela 5.5 sumariza os resultados da Campanha 1 em relação aos serviços IMediatorServer e IMediatorExecution. O IMediatorManager não possui casos de teste, pois o serviço não possui parâmetro na requisição para serem geradas requisições inválidas. Por este motivo, a abordagem adotada nesta campanha não cobre este tipo de requisição.

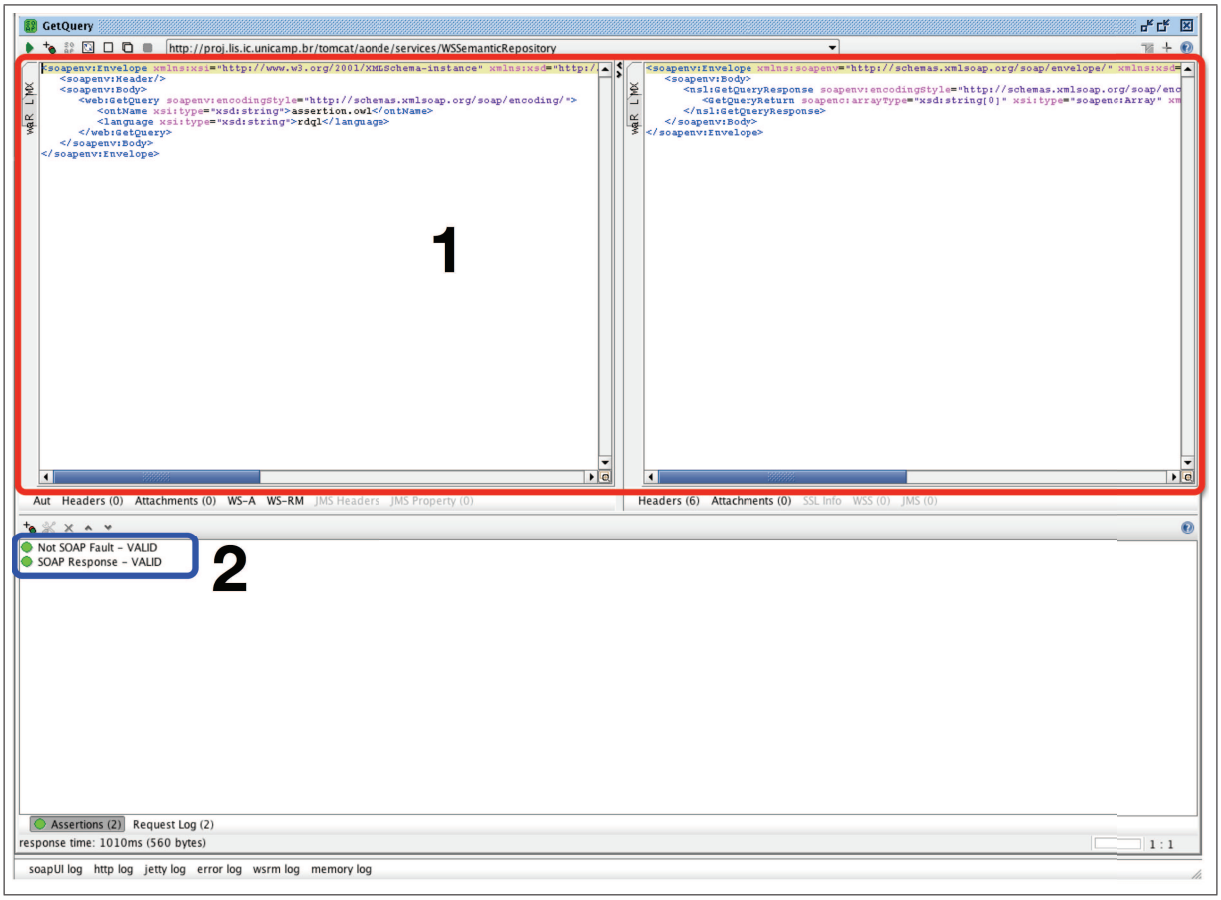


Figura 5.9: Tela do soapUI. (1) Mensagem de requisição e resposta. (2) Assertivas.

Os modos de defeito observados no IMediatorServer e IMediatorExecution foram classificados como defeito de silêncio. Estes defeitos ocorreram porque o Archmeds não informou ao cliente que os parâmetros de configuração eram inválidos e a execução ocorreu de forma errônea. As requisições inválidas foram processadas pelo serviço, porém levando-o para um estado incorreto. Este estado faz com que na hora do IMediatorExecution executar a mediação confiável, ela seja feita de forma incorreta devido ao estado errôneo do Archmeds. Com este resultado, é possível inferir que, em WSs que utilizam o WS-Addressing para criar serviços, o defeito pode manifestar em estados posteriores ao do que foi injetada a falha.

Tabela 5.5: Resultados da Campanha 1.

WS	**Casos de teste**	**Silêncio**
IMediatorServer	42	23
IMediatorExecution	13	5

Em relação às falhas de comunicação injetadas durante a Campanha 2, nenhum defeito foi observado, isto mostra que o Archmeds é robusto em relação as falhas de comunicação injetadas.

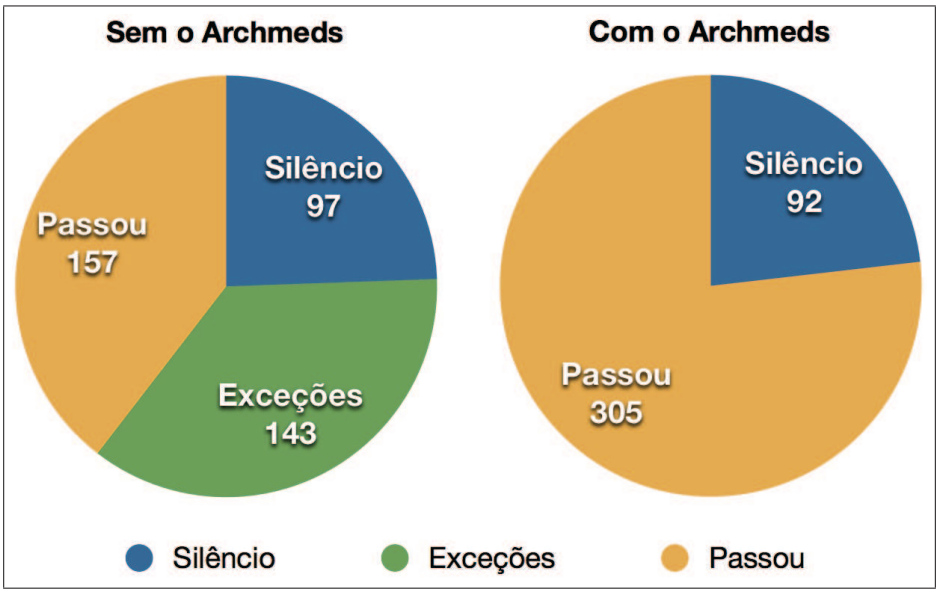


Figura 5.10: Resultados da Campanha 3.

A Figura 5.10 mostra os resultados da Campanha 3 quando injetadas falhas na interface do Aondê. Sem o Archmeds, houve 97 defeitos de silêncio observados. Os defeitos foram analisados através da análise do conteúdo da resposta do Aondê, que possuíam dados de resposta inválidos. Também foram observadas mensagens com formato errado em algumas respostas, pois algumas das respostas continham informações sobre exceções relacionadas às requisições inválidas, porém estas informações não seguiam o formato especificado pelo protocolo SOAP. Já o resultado com o uso do Archmeds encontrou somente 92 defeitos de silêncio, cinco a menos do que sem o Archmeds. Porém, vale ressaltar que, sem o Archmeds, o Aondê levantou 143 exceções, entretanto, com a mediação confiável aplicado pelo Archmeds, todas estas exceções foram tratadas e corrigidas.

Com estes resultados foi possível verificar que o Archmeds não analisa o conteúdo da mensagem SOAP e por esta razão, ele não consegue identificar mensagens contendo resultados inválidos e nem formato inválido. Isto fica evidenciado pelos defeitos de resultados inválidos e em formato errôneo em que o Archmeds deixou ser retornado ao cliente.

Os resultados se mostraram consistentes nas três execuções das campanhas, pois todos os casos de testes apresentaram resultados semelhantes em todas as execuções.

5.6 Discussão sobre os resultados

Os resultados dos testes de robustez mostraram que as falhas de interface foram mais efetivas em encontrar defeitos no Archmeds. Em contrapartida, as falhas de comunicação também foram importantes para testar os mecanismos de tolerância a falhas do Archmeds, que, neste caso, se mostraram robustos o suficiente para tratar as falhas de comunicação injetadas.

Os resultados deixaram evidente que as classes do wsAS não foram suficientes para representar os modos de defeito encontrados, pois, diferentes defeitos (isto é, espera indefinida, entrada inválida aceita, resultado incorreto e formato errôneo) foram classificados em uma classe muito genérica, a de silêncio. Isto se torna um problema, pois o sintoma não fica evidente para o desenvolvedor, que for corrigir a falhas, e por isso fica mais difícil a correção.

Devido a esta necessidade de ter uma classificação mais detalhada dos modos de defeito, foi proposto utilizar uma nova classificação. Esta classificação deveria ser genérica o suficiente para poder ser utilizada em diversos outros testes de robustez, mas por outro lado, específica o suficiente para retratar o sintoma do defeito. Neste trabalho não foi considerado criar uma nova classificação para modos de defeito em WSs, pois, um estudo como este seria muito custoso, já que necessita de um experimento com uma quantidade enorme de dados, o que não seria viável.

Como alternativa, foi decidido usar uma classificação existente. Entre as pesquisadas estavam a ODC[3] e IEEE *std.* 1044[23]. A ODC é um padrão proposto pela IBM que classifica a atividade (ou gatilho), que gerou o defeito, ou seja, a falha que levou ao defeito. Como ele não classifica o sintoma, mas a causa do defeito, então não serve para os propósitos deste trabalho.

O IEEE *std.* 1044[23] é um padrão para classificação de anomalias. O significado de anomalia é qualquer condição que foge do esperado. Então, uma anomalia não envolve somente defeitos, mas também condições que podem ser melhoradas. Este padrão tem o objetivo de prover um processo de classificação de anomalias que é dividida em quatro passos:

- Reconhecimento[4]: ocorre quando a anomalia é encontrada, identificada e classificada.
- Investigação[5]: A anomalia é investigada para identificar problemas relacionados e propor soluções.

[3]http://www.research.ibm.com/softeng/ODC/ODC.HTM.
[4]Do inglês, *recognition*.
[5]Do inglês, *investigation*.

Tabela 5.6: Sintomas de anomalias segundo a IEEE *std.* 1044 (tradução nossa).

Sintomas gerais	
Catástrofe do sistema operacional	Defeito total no produto
Espera indefinida	Erros de mensagem do sistema operacional
Outros	
Problemas nos dados de entrada	
Entrada correta não aceita	Entrada com descrição incorreta ou em falta
Entrada incorreta aceita	Parâmetros incompletos ou em falta
Problemas nos dados de saída	
Formato errôneo	Incompleto ou em falta
Resultado incorreto	Erro gramatical
Cosmético	

- Ação[6]: É estabelecido um plano de ação baseados nos resultados da investigação.
- Parecer[7]: Dá o parecer final da anomalia após a execução do plano de ação.

Com o propósito de classificar os modos de defeito dos WSs, somente a fase de reconhecimento foi considerada, pois nela existe uma classificação de modos de defeito que incluem todos os defeitos encontrados neste trabalho e nos trabalhos correlatos (isto é, WS-Fit e wsAS). Além disto, tem a vantagem de ser um padrão adotado na indústria.

A Tabela 5.6 possui todos os sintomas utilizados para classificar as anomalias de *software* segundo a IEEE *std.* 1044. Apesar de esta classificação ser anterior ao surgimento dos WSs, é importante levar em conta que ela foi criada com base em anomalias de diferentes domínios de *software* e por isso, pode, inclusive, classificar modos de defeitos de WSs, pois é genérica o suficiente para isso. Além disso, na pesquisa feita neste trabalho, não foi encontrada nenhuma outra classificação para WSs, com exceção das já citadas.

Com a IEEE *std.* 1044 deve-se notar que nem todos os sintomas podem ser considerados defeitos de fato, por exemplo, cosmético é mais uma necessidade de melhoria do que um defeito em si. No entanto, não foi feito uma seleção de quais, realmente, são defeitos de robustez de WSs, pois para isto, teria que ser feito um estudo para analisar um banco de dados de defeitos para chegar a uma conclusão de quais defeitos deveriam ser filtrados, o que seria muito custoso.

Usando a classificação do IEEE *std.* 1044, os defeitos encontrados na Campanha I foram redistribuídos da forma como está sumarizado na Tabela 5.7. Os resultados das falhas de interface da Campanha 3 foram reclassificados como ilustrado na Figura 5.11.

[6]Do inglês, *action.*
[7]Do inglês, *diposition.*

Tabela 5.7: Resultados da Campanha 1 reclassificados.

WS	Casos de teste	Resultado Incorreto	Espera Indefinida
IMediatorServer	42	17	6
IMediatorExecution	13	5	0

Em relação aos objetivos deste trabalho, os resultados apontaram 120 defeitos na arquitetura do BioCORE com o uso do Archmeds. Isto representa 26,03% do total de casos de teste. E como conclusão, estes resultados mostram uma necessidade de validar os dados de entrada por parte dos WSs, pois assim estes tipos de defeitos poderiam ser evitados.

O Archmeds conseguiu melhorar a dependabilidade do serviço do Aondê, pois ele evitou cinco defeitos de espera indefinida, nove falhas de comunicação e 143 exceções. Isto indica que aproximadamente 34,05% dos casos de teste obtiveram uma resposta válida mesmo onde não era esperado, ou seja, um ganho de 34,05

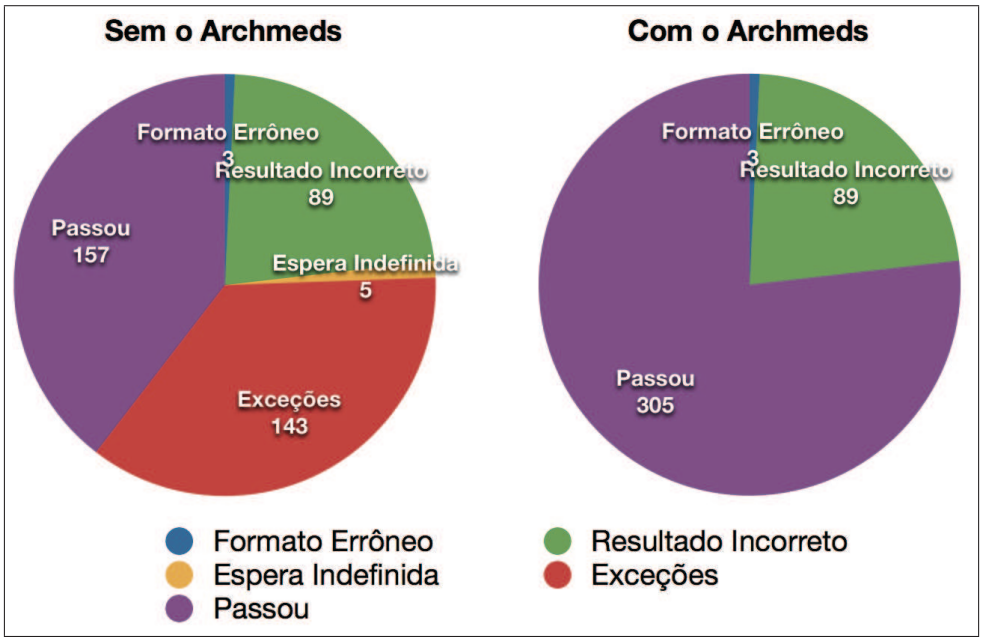


Figura 5.11: Resultados da Campanha 3 reclassificados.

Em relação ao WSInject, o processo de execução de testes de robustez foram todos automatizados, desde a geração dos casos de teste até a execução. Um dos pontos negativos foi a análise de resultados manual, pois só contou com a assistência das assertivas do soapUI para analisá-las .

Os resultados obtidos com esta ferramenta mostraram que os requisitos da Seção 3.2 puderam ser cumpridos pela WSInject, pois o estudo de caso mostrou que:

- Foi compatível com a extensão WS-Addressing, sem que a ferramenta tivesse que ter a compatibilidade específica desta extensão.
- Foi possível testar a composição de serviços do Archmeds.
- Foi automatizado o processo de criação de casos de teste através da análise do WSDL.
- Trabalhou em rede.
- Não foi necessário ter o código fonte do cliente e nem dos serviços.
- Foram injetadas falhas tanto de comunicação quanto de dados de entrada inválidos.

Capítulo 6

Conclusões

Esta dissertação apresenta uma solução para testar a robustez do Archmeds, uma infraestrutura confiável para aumento do grau de confiabilidade de sistemas baseados em WSs, implantado na arquitetura do BioCORE. Esta solução, batizada de WSInject, foi utilizada no estudo de caso que através de 461 casos de teste, apontou defeitos em 120 deles.

A WSInject foi desenvolvida com requisitos que envolvem tanto os requisitos para testar o Archmeds quanto os WSs cobertos pelos trabalhos relacionados. Isto demonstrou um avanço na abordagem para injeção de falhas em WSs, pois a ferramenta consegue ser compatível com um número maior de WSs.

A abordagem de teste de robustez foi automatizada através do uso do padrão WSDL, que forneceu informações referentes ao WS e tornou possível automatizar o processo de geração dos casos de teste com injeções de falhas. Quanto ao modelo de falhas adotado, foi utilizada a proposta da Ballista e wsrbench, que consiste em injetar parâmetros inválidos nas chamadas às operações. Além disso, o modelo também foi estendido para que fosse possível injetar falhas de comunicação, pois arquiteturas baseadas em WSs também são suscetíveis a esse tipo de falhas.

Para analisar os modos de defeito encontrados no estudo de caso, foi utilizada a classificação dos modos de defeito dos trabalhos relacionados, porém foi apontada uma necessidade de utilizar uma classificação mais condizente com os sintomas do defeito, pois isto facilita na análise para encontrar uma solução para o problema. Para isto, foi proposto utilizar o padrão IEEE *std.* 1044, pois, além de ser um padrão utilizado pela indústria, é genérico o suficiente para conseguir cobrir os modos de defeito encontrados e específico o bastante para caracterizar os sintomas dos defeitos dos WSs.

As campanhas de injeção também mostraram que o Archmeds melhora a confiabilidade da arquitetura BioCORE. Como pôde ser visto na Campanha 3, com o uso do Archmeds, houve uma a maior quantidade casos de teste que passaram, 305, ou seja 148 a mais do

que sem o uso da infraestrutura. Isto mostra que ao adotar o Archmeds, o BioCORE estaria ganhando com este aumento de confiabilidade e disponibilidade dos WSs. Outra evidência de sua efetividade foi a Campanha 2, em que ele conseguiu isolar todas as falhas de comunicação.

6.1 Contribuições

A WSInject rendeu um artigo [5] e um relatório técnico [24]. Além disso, existem outros trabalhos de mestrado e doutorado que contribuíram com a WSInject e utilizam-na para realizarem os seus estudos de caso, como são os casos dos coautores do artigo [5].

O levantamento e a comparação das ferramentas correlatas são fontes úteis para os interessados em testar a robustez dos WSs. Com estes resultados é possível visualizar as ferramentas com as características desejadas para realizar os testes de robustez nos WSs em questão.

Caso algum pesquisador esteja interessado em continuar ou até mesmo criar uma nova ferramenta de teste de robustez para WSs, os requisitos listados neste trabalho poderão servir como base para serem estendidos com novos requisitos.

Além disso, os resultados deste trabalho poderão ser consultados pelos pesquisadores do BioCORE com a finalidade de corrigir os defeitos encontrados. A classificação da IEEE *std.* 1044 possui uma descrição dos modos de defeitos que ajudará a encontrar as falhas que serão corrigidas.

Finalmente, também existem pesquisas que utilizam a WSInject para testar a segurança de WSs. Isto seria possível, pois a ferramenta trabalharia como um agente malicioso que altera as mensagens SOAP e realiza ataques aos WSs.

6.2 Trabalhos futuros

Um dos pontos negativos da abordagem adotada neste estudo de caso foi o processo manual de classificação dos modos de defeito. Este processo acaba tomando muito tempo e, dependendo da quantidade de recursos disponíveis, fica inviável classificar um número grande de casos de teste. Como um trabalho futuro importante, seria criar uma abordagem automática para análise e classificação dos modos de defeito.

Outro trabalho seria filtrar o padrão proposto pela IEEE *std.* 1044 para listar sintomas relacionados somente aos modos de defeito dos WS. Para isso teria que ser feito um estudo com um repositório de defeitos de WSs para extrair esta lista.

Finalmente, poderia haver uma pesquisa para encontrar novas aplicações para a WSInject e incrementá-la com novas funcionalidades, pois a abordagem utilizada pela ferra-

menta pode ser utilizada em outros domínios de teste. A aplicação proposta para a ferramenta foi o teste de robustez, porém, existem outras pesquisas que poderiam se beneficiar desta ferramenta, como é o caso de teste de segurança de WSs.

Referências Bibliográficas

[1] Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.

[2] AMBER. Amber: Assesing, measuring, and benchmarking resilience. State of the Art D2.2, University of Coimbra, 2009.

[3] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of cots microkernel-based systems. *IEEE Trans. Computers*, 51(2):138–163, 2002.

[4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[5] Fayçal Bessayah, Ana R. Cavalli, Willian Maja, Eliane Martins, and Andre Willik Valenti. A fault injection tool for testing web services composition. In *TAIC PART*, pages 137–146, 2010.

[6] Jaudete Daltio. Aondê: Um serviço web de ontologias para interoperabilidade em sistemas de biodiversidade. Dissertação, Universidade Estadual de Campinas - Instituto de Computação, 2007.

[7] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[8] Eduardo Machado Gonçalves. Uma infra-estrutura confiável para arquiteturas baseadas em serviços web aplicada à pequisa de biodiversidade. Master's thesis, Instituto de Computação, Unicamp, Campinas, SP, 2009.

[9] Eduardo Machado Gonçalves and Cecília Mary Fischer Rubira. Archmeds: An infrastructure for dependable service-oriented architectures. In *ECBS*, pages 371–378, 2010.

[10] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[11] Luiz Celso Gomes Junior. Uma arquitetura para consultas a repositorios de biodiversidade na web. Dissertação, Universidade Estadual de Campinas - Instituto de Computação, 05 2007.

[12] Philip Koopman and John DeVale. Comparing the robustness of posix operating systems. In *FTCS*, pages 30–37, 1999.

[13] Nuno Laranjeiro, Salvador Canelas, and Marco Vieira. wsrbench: An on-line tool for robustness benchmarking. In *IEEE SCC (2)*, pages 187–194, 2008.

[14] Nelson Leme, Eliane Martins, and Cecília Mary Fischer Rubira. A software fault injection pattern system. In *Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing*, 2001.

[15] Peter Li, Yuhui Chen, and Alexander Romanovsky. Measuring the dependability of web services for use in e-science experiments. In *ISAS*, pages 193–205, 2006.

[16] Nik Looker, Malcolm Munro, and Jie Xu. Ws-fit: A tool for dependability analysis of web services. In *COMPSAC Workshops*, pages 120–123, 2004.

[17] Nik Looker and Jie Xu. Assessing the dependability of soap rpc-based web services by fault injection. In *WORDS Fall*, pages 163–170, 2003.

[18] Evan Martin, Suranjana Basu, and Tao Xie. Websob: A tool for robustness testing of web services. In *ICSE Companion*, pages 65–66, 2007.

[19] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25, 1990.

[20] Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, pages 3–12, 2003.

[21] ReSIST. Resist: Resilience for survivability in ist. State of art, LAAS-CNRS, 2006.

[22] Charles P. Shelton, Philip Koopman, and Kobey Devale. Robustness testing of the microsoft win32 api. In *DSN*, pages 261–, 2000.

[23] IEEE Computer Society. Ieee standard classification for software anomalies. Technical Report 0-7381-0406-X, IEEE Computer Society, 1994.

[24] A. W. Valenti, W. Y. Maja, E. Martins, F. Bessayah, and A. Cavalli. WSInject: A Fault Injection Tool for Web Services Technical Report 1.0. Technical Report IC-10-22, Institute of Computing, University of Campinas, July 2010.

[25] Marco Vieira, Nuno Laranjeiro, and Henrique Madeira. Assessing robustness of web-services infrastructures. In *DSN*, pages 131–136, 2007.

[26] Marco Vieira, Nuno Laranjeiro, and Henrique Madeira. Benchmarking the robustness of web services. In *PRDC*, pages 322–329, 2007.

[27] World Wide Web Consortium (W3C). Web services addressing (ws-addressing). `http://www.w3.org/Submission/ws-addressing/`, 2004.

[28] World Wide Web Consortium (W3C). Web services glossary. `http://www.w3.org/TR/ws-gloss/`, Outubro 2010.

[29] Taisy Silva Weber. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. `http://www.inf.ufrgs.br/~taisy/disciplinas/textos/Dependabilidade.pdf`, Outubro 2010.