Instituto de Computação

Universidade Estadual de Campinas

Testes de Robustez em Web Services por meio de Injeção de Falhas

André Willik Valenti

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por André Willik Valenti e aprovada pela Banca Examinadora.

Campinas, 26 de setembro de 2011

Eliane Martins

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

Valenti, André Willik, 1986-
V234t Testes de robustez em web services por meio de injeção de falhas / André Willik Valenti. - Campinas, SP : [s.n.], 2011.

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

1. Tolerância a falha (Computação). 2. Software – Testes. 3. Serviços da Web. 4. Arquitetura orientada a serviços (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 web services by means of fault injection
Palavras-chave em inglês:
Fault-tolerant computing
Software - Testing
Web services
SOA (Computer science)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Ellen Francine Barbosa
André Santanchè
Data da defesa: 29-07-2011
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 29 de julho de 2011, pela Banca examinadora composta pelos Professores Doutores:

Profª. Drª. Ellen Francine Barbosa
ICMC / USP-São Carlos

Prof. Dr. André Santanchè
IC / UNICAMP

Profª. Drª. Eliane Martins
IC / UNICAMP

Instituto de Computação

Universidade Estadual de Campinas

Testes de Robustez em Web Services por meio de Injeção de Falhas

André Willik Valenti[1]

Julho de 2011

Banca examinadora:

- Profa. Dra. Eliane Martins (Orientadora)
- Profa. Dra. Ellen Francine Barbosa
  Instituto de Ciências Matemáticas e de Computação, USP-São Carlos
- Prof. Dr. André Santanchè
  Instituto de Computação, UNICAMP
- Profa. Dra. Cecília Mary Fischer Rubira (Suplente)
  Instituto de Computação, UNICAMP
- Prof. Dr. Marcelo Fantinato (Suplente)
  Escola de Artes, Ciências e Humanidades, USP-São Paulo

[1] Suporte financeiro de: Bolsa do CNPq (processo 136044/2008-5) 01/09/2008 – 31/07/2010; Projeto RobustWeb da CAPES/Cofecub (processo 623/09)

Resumo

A crescente adoção de Arquiteturas Orientadas a Serviços e de Web Services pela indústria e pela academia vem criando novos desafios para a garantia de qualidade dos sistemas. Testes de robustez possibilitam verificar o funcionamento de um sistema quando sujeito a condições adversas de operação, como entradas inesperadas ou alta carga de requisições sobre os serviços. A técnica de injeção de falhas pode ser aplicada para induzir tais situações e permitir uma avaliação do sistema nessas condições. No entanto, encontram-se poucas ferramentas adequadas para essa atividade. Nesta dissertação, apresentamos a WSInject, uma ferramenta para injeção de falhas em Web Services, além de um estudo sobre testes de robustez em sistemas baseados nessa tecnologia. A ferramenta possui as vantagens de ser flexível, configurável, extensível e minimamente intrusiva. Este trabalho é parte do projeto RobustWeb da CAPES/Cofecub, registrado sob o número 623/09.

Abstract

The increasing adoption of Service-Oriented Architectures and Web Services, both by industry and academia, has been posing new challenges for quality assurance. Robustness testing allows one to verify the behavior of a system when subject to adverse operating conditions, such as unexpected inputs or high service loads. The fault injection technique may be used to induce such scenarios in order to evaluate system behavior under these conditions. However, few tools are currently available to support this activity. In this work, we present WSInject – a fault injection tool for Web Services – and a study on Web Services robustness testing. WSInject's advantages include being flexible, customizable, extensible and minimally intrusive. This work is registered under number 623/09 as part of RobustWeb project from CAPES/Cofecub.

Agradecimentos

Agradeço à minha família, que sempre me apoiou durante o mestrado e me incentivou a estudar pelo tempo que a vida me permitisse; à minha orientadora, Eliane Martins, pela atenção, dedicação e orientação neste trabalho; ao pessoal da secretaria do IC, por seu trabalho eficiente, dedicado e atencioso; à empresa que colaborou com nosso trabalho, gentilmente cedendo seu sistema para a realização de nossos experimentos.

Agradeço aos novos amigos feitos durante o mestrado, hoje queridos velhos amigos: Aline, Anna, Lehilton, Robson, Sheila, Thiago e Thiago, todos atualmente encaminhados na vida e muitos, infelizmente, distantes.

Agradeço às pessoas que compartilham ou compartilharam residência e convivência diária comigo nesses três anos: Arthur, Lincoln, Carlos, Marsolla e Tiago. Agradeço aos meus amigos de trabalho por tão bons momentos já vividos e tantas boas ideias compartilhadas: Allan, Andrei, Caroline, Fellipe, Guilherme Santos, Oto e Thiago; Bruno, Edmagne e Guilherme Laranja.

Agradecimentos muito especiais ao Lehilton e à Mariana, pela ajuda fundamental que me deram para a tarefa de concluir o mestrado.

Agradeço a todas as pessoas que passaram pela minha vida e foram importantes para mim durante esse período do mestrado. Muitas delas continuam fazendo parte da minha vida e sendo muito importantes.

Sumário

Lista de Tabelas

Lista de Figuras

1. Introdução

As **Arquiteturas Orientadas a Serviços** (*Service Oriented Architecture* – **SOA**) vêm se consolidando nas empresas e nas universidades como uma forma eficaz de implementar sistemas que realizam processos de negócio. Esse estilo arquitetural oferece diversas vantagens sobre outros padrões de arquitetura, principalmente o baixo acoplamento.

A unidade de implementação em uma SOA é o **serviço**. Tipicamente, os serviços de uma arquitetura são implementados no padrão **Web Services** [29]. Com a crescente adoção de SOA e Web Services no desenvolvimento de sistemas, torna-se necessário garantir sua **robustez**, ou seja, sua capacidade de manter o correto funcionamento mesmo em condições adversas de operação [34]. A robustez é um atributo importante de um sistema de *software*, pois todo sistema poderá estar sujeito a condições adversas em algum momento de sua operação, ainda que isso não seja antecipado durante o seu desenvolvimento. Essas condições adversas podem ocorrer em duas situações: ou quando o sistema recebe uma alta carga de trabalho, ou quando recebe entradas inválidas. Web Services normalmente ficam expostos na Internet. Dessa forma, estão sujeitos a essas situações, devendo ser capazes de tolerá-las.

A robustez de um sistema deve ser avaliada por meio de **testes de robustez**, que verificam o comportamento de um sistema na presença de condições adversas. A **injeção de falhas** é uma técnica eficaz para testar a robustez de um sistema [13], [21], [30]. Ela consiste na criação intencional de situações não esperadas pelo sistema, de modo a avaliar o seu comportamento. A injeção de falhas é mais tradicionalmente empregada em testes de hardware [34] e em sistemas de tempo real [9].

A técnica é considerada promissora também no contexto de Web Services, encontrando-se algumas publicações que discutem boas técnicas e ferramentas para sua aplicação nesse contexto [30]. As publicações existentes, em geral, apresentam a injeção de falhas principalmente na forma de envio de mensagens inválidas para um serviço – ato que pode ser suficiente para causar defeitos em um serviço, às vezes de severidade alta [14].

A robustez é um atributo importante, e muitas vezes negligenciado, em sistemas de *software.* Um sistema é dito robusto quando funciona corretamente mesmo em condições

adversas de operação. É importante reforçar que todo sistema poderá estar sujeito a condições adversas em algum momento de sua operação. Por vezes, falhas pequenas podem causar grandes desastres: DeVale cita como exemplo o voo do foguete Ariane 5, em 1996, no qual uma exceção de *software* não foi tratada e culminou na explosão do foguete [7]. Essa exceção foi causada por uma simples conversão mal-sucedida de um número de 64 bits para 16 bits. O problema poderia ter sido evitado se uma atenção maior tivesse sido dada, durante o desenvolvimento do sistema, às possíveis condições adversas de operação.

A preocupação com robustez durante o desenvolvimento de sistemas deve ser constante e a ideia fundamental é bastante simples: deve-se atentar muito à **validação** dos dados recebidos como entrada. Vale o **princípio da robustez**, também conhecido como lei de Postel: "*seja conservador no que você envia, seja liberal no que você aceita*" [27]. Assim, ao implementar um sistema, o desenvolvedor deve sempre ter como objetivo produzir uma saída consistente e, paralelamente, nunca presumir que receberá do sistema anterior a ele uma entrada consistente – devendo validar essa entrada antes de continuar sua execução. Essas inconsistências, ou falhas, podem ocorrer na forma de *strings* com caracteres inesperados, *strings* muito grandes, valores fora do limite esperado, documentos mal-formados, injeção de código ou em diversas outras formas. A injeção de código, particularmente, pode ser vista tanto como um problema de robustez quanto como um problema de segurança. O conhecido ataque de injeção de SQL, sob o ponto de vista da robustez, é simplesmente uma aplicação de dados de entrada inválidos a um sistema que não valide adequadamente esses dados.

O presente trabalho propõe uma nova ferramenta de injeção de falhas, WSInject, para auxiliar na atividade de testes de robustez em Web Services. Foram analisadas e discutidas as vantagens e desvantagens de algumas técnicas e ferramentas já existentes, que serviram como base para o desenvolvimento de nossa própria ferramenta. A WSInject tem como principais objetivos: 1) permitir que o usuário tenha um alto grau de controle sobre as falhas a serem injetadas, principalmente quanto ao modelo de falhas usado; 2) permitir que a arquitetura original de um sistema seja usada nos testes com pouquíssimas modificações. Além disso, o presente trabalho também propõe um novo modelo de falhas para ser usado em testes de robustez de Web Services.

Foram realizados experimentos com diversas configurações, em dois sistemas diferentes. Foram aplicados alguns modelos de falhas encontrados na literatura e também o modelo proposto neste trabalho. Os resultados mostraram-se promissores e indicaram novos rumos para o desenvolvimento da WSInject, além de conclusões interessantes sobre a aplicação de injeção de falhas em sistemas baseados em Web Services.

A dissertação está organizada da seguinte maneira: o capítulo 2 apresenta a fundamentação teórica, explicando os conceitos de falha, erro, defeito, robustez, injeção de falhas e Web Services. O capítulo 3 apresenta alguns trabalhos relacionados, discutindo modelos de injeção de falhas e ferramentas para testes de Web Sevices. O capítulo 4 apresenta a WSInject, explica seus conceitos e sua implementação. O capítulo 5 detalha os experimentos realizados com a WSInject. Finalmente, o capítulo 6 apresenta a conclusão e os trabalhos futuros.

2. Fundamentação Teórica

Nesta seção, definimos os conceitos relacionados a robustez e os conceitos relacionados a Web Services.

2.1. Falhas, Erros e Defeitos

Para se definir robustez, primeiramente devem-se definir os conceitos de falha, erro e defeito. Usaremos esses termos como correspondentes dos termos em inglês *fault*, *error* e *failure*, respectivamente. Essa terminologia em português é a mesma adotada por Weber [36]. Um **defeito** é o que ocorre quando o sistema opera de forma diferente do que foi originalmente especificado (por exemplo, exibe um resultado diferente do esperado). O **erro** é um estado incorreto do sistema que pode causar um defeito (por exemplo, um valor resultante de uma computação incorreta). Finalmente, **falha** é a causa do erro (por exemplo, uma linha de código incorreta ou um defeito físico do *hardware*).

Uma falha precisa ser **ativada** para causar um erro. Por exemplo, uma linha de código escrita incorretamente (falha) só causará um estado incorreto no sistema (erro) se for de fato executada. Se ela for executada, causando um erro, este só será percebido fora do sistema (pelo usuário) caso provoque um comportamento do sistema em desacordo com sua especificação, ou seja, um defeito.

Falhas podem ser propagadas entre componentes de um sistema (funções, classes, serviços, máquinas em uma rede etc.). A propagação de falhas ocorre quando um componente recebe dados inválidos (falhas) e não os trata adequadamente. Considerando-se dois componentes, **A** e **B**, uma saída incorreta gerada por A e transmitida para B pode ser considerada, simultaneamente, um defeito para A e uma falha para B. A Figura 1 ilustra essa possibilidade de propagação de falhas.

Figura 1. Relação entre falha, erro e defeito.

A manifestação concreta de uma falha depende do contexto em que ela ocorre. No contexto de *hardware*, uma falha pode ser um sinal elétrico indevidamente aplicado a um circuito, ou uma temperatura elevada. No contexto de *software*, normalmente uma falha apresenta-se como um valor inesperado dado como entrada a um componente. A ideia de "inesperado" depende do tipo de dado considerado. Por exemplo, um campo "ano de nascimento" em um sistema *web* não deverá aceitar números fracionários ou negativos, pois esse tipo de valor não faz sentido nesse contexto. Esses valores, nesse caso, são considerados falhas. *Strings* muito grandes também podem ser consideradas falhas, já que não são normalmente esperadas pelos componentes (exceto aqueles já projetados com essa finalidade).

2.2. Robustez

Algumas classes de sistemas, como os sistemas críticos, devem possuir a habilidade de executar suas funções corretamente em qualquer situação, mesmo sob condições adversas. Por exemplo, sistemas controladores de aviões são vitais para guiar o piloto durante o voo e devem operar corretamente em todos os momentos. Os **testes de robustez** têm o intuito de avaliar se um determinado sistema cumpre esse objetivo.

A **robustez** de um sistema é o nível em que este consegue operar corretamente, mesmo na presença de dados de entrada inválidos ou ambientes estressantes [10] *apud* [34]. Dessa forma, testes de robustez consistem em aplicar esse tipo de entrada a um sistema e verificar o seu comportamento. Um sistema é dito **robusto** quando é capaz de apresentar resultados corretos mesmo quando opera em ambientes estressantes e na presença de falhas.

A principal meta dos testes de robustez é ativar falhas de um sistema que resultem em defeitos, revelando essas falhas. A descoberta dessas falhas será o ponto de partida para sua correção. Os defeitos encontrados podem ser classificados conforme o tipo de impacto causado ao sistema. Uma escala de classificação bastante conhecida é a **CRASH**, usada por DeVale *et al.* [6], que define os seguinte **modos de defeito**: 1) ***C****atastrophic*, quando o sistema todo para ou reinicia; 2) ***R****estart*, quando a aplicação precisa ser reiniciada; 3) ***A****bort*, quando a aplicação termina de forma anormal; 4) ***S****ilent*, nenhum código de erro é

indicado embora um erro tenha ocorrido; 5) ***H**indering*, quando um código de erro incorreto é sinalizado.

Existem dois fatores distintos a serem considerados na área de testes de robustez: a **carga de trabalho** e a **carga de falhas** [34]. A carga de trabalho refere-se a entradas comuns, esperadas. A carga de falhas refere-se a entradas excepcionais, inesperadas. Como exemplo, para um campo "nome" de um sistema de cadastro de pessoas, uma carga de trabalho típica poderá conter nomes de pessoas – reais ou não –, enquanto uma carga de falhas deverá conter *strings* muito grandes, nomes com acentos e símbolos, *strings* contendo caracteres não-imprimíveis, *string* vazia etc. Em um teste, os fatores podem ser aplicados separadamente ou simultaneamente ao sistema.

A robustez de um sistema está fortemente relacionada à sua capacidade de tratar a ocorrência de grandes cargas de trabalho e a ocorrência de falhas. A propagação de falhas entre seus componentes pode causar danos ao sistema e essa situação deve ser evitada.

Uma forma simples de evitar essa situação é validar os dados de entrada do sistema. Por exemplo: um sistema que recebe como entrada um valor para "ano de nascimento" pode facilmente verificar e rejeitar valores negativos ou fracionários. Essa simples verificação pode evitar diversos problemas, conforme apontam diversos autores, incluindo DeVale *et al.* [6].

2.3. Injeção de Falhas

Injeção de falhas é uma técnica para testes de robustez que consiste em produzir falhas na interface de um sistema. O objetivo é verificar a maneira como o sistema se comporta na presença destas falhas.

A injeção pode ser de falhas de *hardware*, quando os componentes físicos de uma máquina são submetidos a condições estressantes de operação (como aplicação de radiação); ou de *software*, quando os componentes de software de um sistema são submetidos a condições estressantes de operação (como entradas inválidas ou cargas de trabalho elevadas).

A injeção de falhas de *software* pode ser subdividida em **injeção em tempo de compilação** e **injeção em tempo de execução** [21]. No caso de injeção em tempo de compilação, a falha é inserida no próprio código-fonte ou no código-alvo (binário ou

executável). Por exemplo, a substituição de um comando "x = x + 1" por "x = x – 1". Na injeção de falhas em tempo de execução, um item a mais estará presente na arquitetura do sistema, o **injetor de falhas**. Ele atuará causando **perturbações** no sistema (como a corrupção de pacotes de rede), ou fornecendo diretamente as entradas para o sistema a ser testado.

A injeção de falhas normalmente é feita com maior foco na aplicação da carga de falhas, embora seja possível (e até mesmo promissor [29]) empregar uma abordagem mista, com foco também na carga de trabalho.

A vantagem da injeção de falhas é facilitar a simulação de situações que normalmente levariam muito tempo para ocorrer se não houvesse qualquer intervenção. O objetivo da aplicação de falhas é causar defeitos em um sistema, apontando a presença de falhas já existentes anteriormente.

É importante diferenciar as falhas que são injetadas das falhas já existentes no sistema. As falhas já presentes, chamadas de **falhas internas**, são aquelas normalmente introduzidas por programadores durante a fase de desenvolvimento, seja por enganos durante a programação (exemplo: não-fechamento de uma conexão remota), pela não-previsão de todos os cenários aos quais o sistema seria submetido, ou mesmo pelo uso de componentes prontos já contendo falhas. As falhas injetadas no sistema, chamadas de **falhas externas**, são aquelas aplicadas propositalmente à entrada do sistema, que possivelmente causarão erros (computações incorretas), que, por sua vez, possivelmente causarão defeitos (funcionamento fora da especificação, resultados incorretos visíveis de fora do sistema). Esse cenário é ilustrado na Figura 2.

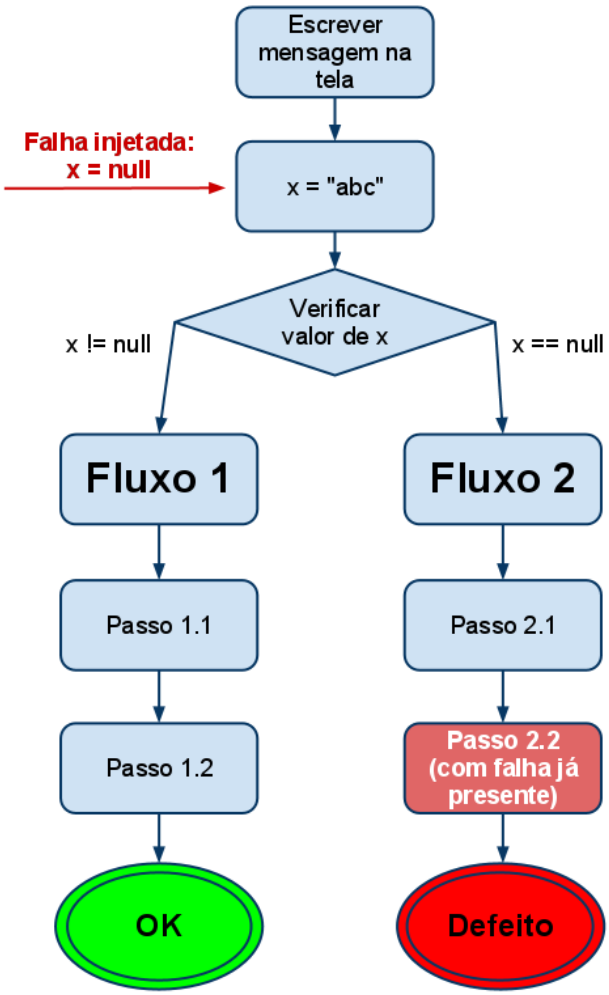

Figura 2. Falhas já presentes e falhas injetadas

A figura ilustra um cenário de execução de um programa. O fluxo 1 não contém falhas e sua execução sempre ocorre normalmente até o fim. O fluxo 2 contém uma falha no passo 2.2. Se esse passo for executado, o sistema produzirá um defeito. Note que a execução dos fluxos 1 ou 2 depende do valor de x: se x for diferente de *null*, o fluxo 1 será executado; caso contrário, o fluxo 2 será executado. Pode-se injetar uma falha de modificação do valor de x, alterando-o para o *null.* Isso fará o sistema executar o fluxo 2, causando um defeito e indicando a existência de uma falha no sistema. Note que a injeção de falhas apenas modificou o fluxo de execução do sistema, mas a falha que efetivamente causou o defeito já estava presente.

Para aplicação da técnica de injeção de falhas, deve-se definir a carga de falhas a ser aplicada, que depende do tipo de sistema a ser testado. No contexto de injeção de falhas em

código fonte, um caso típico é a modificação de um comando em uma linha de código, de forma similar ao que foi mostrado no exemplo anterior. Para o teste de um sistema *web*, exemplos de falhas são a inserção de *strings* muito grandes em campos de texto e a inserção de valores negativos em campos como "ano de nascimento". Em testes de Web Services, as falhas podem ser de vários tipos, incluindo alterações nos documentos XML enviados, inserção de atrasos na entrega de mensagens e corrupção de pacotes TCP.

2.4. Web Services

Existem diferentes definições para o conceito de Web Service. Neste trabalho, adotamos a definição do W3C (*World Wide Web Consortium*): um **Web Service** é um sistema de *software* projetado para permitir interação entre máquinas de uma rede, sendo descrito por meio da linguagem WSDL e comunicando-se por meio de mensagens no padrão SOAP [37].

Web Service é, portanto, uma especificação para implementação de sistemas, baseada em padrões XML. Esses padrões garantem interoperabilidade entre diferentes plataformas. As plataformas podem variar tanto em sistemas operacionais (Windows, Linux etc.) quanto em linguagens de programação (Java, Python, Ruby, C# etc.). Web Services são identificados por URIs ("endereços da web", *grosso modo*).

Web Services podem ser vistos como componentes de construção de *software*, assim como os objetos na programação orientados a objetos. Web Services possuem operações, que são análogas aos métodos desses objetos. Podemos considerar Web Services como sendo uma implementação concreta do conceito abstrato de **serviço**.

Segundo Reinecke e Wolter [28], uma **arquitetura orientada a serviços** (**SOA**) consiste em um conjunto de serviços fracamente acoplados, sendo executados em diferentes locais e comunicando-se pela Internet. De acordo com Ribarov *et al.* [30], **serviços** são blocos de construção fundamentais similares a objetos e componentes, com as seguintes características: 1) combinam informação e comportamento; 2) encapsulam seu processamento interno; 3) apresentam uma interface relativamente simples. Assim, a unidade de implementação em SOA é o serviço. Na prática, essa implementação quase sempre é feita com o uso de Web Services [29].

As características de Web Services sob o ponto de vista de negócios e sob o ponto de vista técnico são descritas a seguir.

2.4.1. Ponto de Vista de Negócios

Web Services normalmente são oferecidos na Internet para possíveis clientes – pessoas ou empresas interessadas em adquirir licenças de uso. Dado que o funcionamento de uma aplicação SOA depende do funcionamento adequado dos serviços contratados, surge o conceito de **qualidade de serviço** (*Quality of Service* – **QoS**), definida por características como preço, tempo de resposta, disponibilidade e robustez. Assim, torna-se necessário estabelecer entre as duas partes um acordo formal de qualidade de serviço prestado. Esse acordo é chamado de **SLA** (*Service Level Agreement*) e é uma peça-chave do ponto de vista de negócios. Uma violação de SLA (devida, por exemplo, à ocorrência de problemas nos serviços contratados) significa quebra de contrato, podendo acarretar em multas de valor elevado para a empresa fornecedora do serviço.

2.4.2. Ponto de Vista Técnico

Web Services podem ser implementados em qualquer plataforma e linguagem de programação. Por exemplo, Web Services escritos em Java e rodando sobre Linux podem ser utilizados por um programa escrito em C e rodando sobre Windows. Os padrões baseados em XML garantem essa interoperabilidade, sendo os principais **WSDL** e **SOAP**.

WSDL (*Web Services Description Language*) é a linguagem de descrição de Web Services. É por meio dela que um serviço expõe sua funcionalidade e sua interface. Um documento WSDL contém informações como localização do serviço, parâmetros de entrada e parâmetros de saída.

SOAP (atualmente não mais uma sigla) é o protocolo de comunicação entre Web Services e seus usuários. É por meio dele que se realiza a chamada a um serviço, enviando-se os dados de entrada e recebendo-se os dados de saída.

2.4.3. Composição de Serviços

Um serviço pode ser **simples** (também chamado de **atômico**) ou **composto** (também chamado de **composição de serviços**). Enquanto os serviços simples são implementados diretamente em alguma linguagem de programação comum (como Java), um serviço

composto é implementado com base em serviços pré-existentes, normalmente por meio da linguagem **WS-BPEL** (Web Services Business Process Execution Language). WS-BPEL, também chamada simplesmente de BPEL, é um padrão específico para essa finalidade. Um serviço composto em BPEL é chamado de **processo BPEL**. Os serviços que compõem um serviço composto são chamados de **serviços parceiros**. A execução do processo BPEL, incluindo as chamadas aos serviços parceiros, é controlada por um componente do servidor chamado ***BPEL Engine*** (exemplos: Apache ODE[2], OpenESB[3]). A Figura 3 ilustra uma composição de serviços.

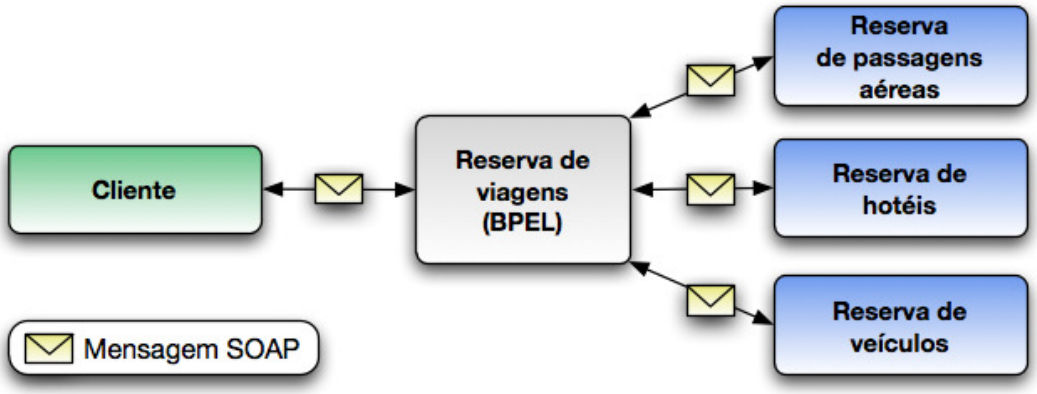

Figura 3. Exemplo de composição de serviços.

A composição possibilita, entre outras coisas, que uma empresa utilize serviços de outras empresas para criar um serviço maior. Podemos tomar como exemplo a composição TravelReservationService, uma aplicação de demonstração disponível na ferramenta NetBeans[4]. Há os serviços de reserva de veículo, reserva de hotel e reserva de passagem aérea. Compondo-os, podemos criar um novo serviço que agregue os três, de modo a reservar esses três itens em uma só operação. Além de unir os três serviços em um só, a composição permite que sejam adicionadas novas funcionalidades; por exemplo, realizar as três reservas em uma única transação. Dessa forma, ela somente será efetivada se as três reservas forem feitas com sucesso. Do contrário, nenhuma reserva será feita.

Um serviço composto é exposto ao seu cliente da mesma forma que um serviço comum: possui interface definida em WSDL e se comunica por meio de mensagens SOAP.

[2] http://ode.apache.org
[3] http://openesb-dev.org/
[4] http://netbeans.org/

Portanto, um cliente não diferencia um serviço atômico de um serviço composto. Em geral, o cliente nem mesmo terá acesso individual aos serviços parceiros da composição, tendo a ilusão de estar usando um serviço simples.

2.5. Testes de Web Services

Ribarov *et al.* afirmam que o teste de serviços possui similaridades com o teste de componentes, já que serviços podem ser considerados "componentes executados em máquinas remotas" [30]. Um detalhe importante dos Web Services é que eles não possuem interface direta com o usuário final. Segundo SeCSE Team, isso os torna mais difíceis de tratar manualmente, mas bons candidatos para uso de testes automatizados [31].

A área de testes no contexto de SOA e de Web Services é bastante ampla. Canfora e Di Penta [3] propõem o conceito de **perspectivas de teste**. Uma perspectiva de teste diz respeito à parte envolvida (*stakeholder*) no teste do serviço. As principais perspectivas estão descritas a seguir.

Desenvolvedor de serviços: refere-se apenas à parte que desenvolve o serviço, não sendo necessariamente a mesma que o disponibiliza para uso. O principal objetivo de seus testes é garantir que o serviço esteja o máximo possível livre de falhas. Essa parte tem acesso ao código-fonte e pode executar testes de caixa branca, além de não ter custos referentes às chamadas aos serviços. Por outro lado, não tem a possibilidade de testar os serviços no ambiente da aplicação em que serão utilizados, o que interfere na representatividade dos resultados de testes não-funcionais.

Provedor de serviços: disponibiliza serviços para os usuários. O objetivo principal dos testes dessa parte é garantir que o serviço seja capaz de cumprir o SLA prometido. Essa parte também não sofrerá cobrança na chamada dos serviços. Porém, em geral, não tem acesso ao código-fonte. Também não é capaz de testar os serviços no ambiente da aplicação.

Integrador de serviços: constrói a aplicação, por meio da integração de serviços. O principal objetivo de seus testes é assegurar que os serviços utilizados atendam a seus requisitos funcionais e não-funcionais. Essa parte não possui controle sobre a implementação dos serviços. A implementação pode, inclusive, ser modificada pelo desenvolvedor enquanto a aplicação já está em produção. Em geral, o integrador arca com

custos associados às chamadas aos serviços, o que impõe limitações aos testes. Além disso, a chamada repetitiva de um mesmo serviço pode causar uma situação de negação de serviço (*denial-of-service*). Testes de robustez também incorrem nesse risco, já que aplicam cargas estressantes de entrada aos serviços (ver seção 2.1. Falhas, Erros e Defeitos).

Usuário: aquele que usa o sistema construído com base em serviços. Em geral, não possui qualquer conhecimento sobre os testes dos serviços.

2.6. Injeção de Falhas em Web Services

Conforme análise de Looker *et al.* [21] e demonstrações realizadas por Jensen *et al.* [12], a técnica de injeção de falhas é bastante eficaz no contexto de Web Services e é capaz de simular a presença de falhas tanto no serviço como no cliente. A principal vantagem dessa técnica é que ela possibilita conduzir o sistema a um estado tal que levaria muito tempo para ser alcançado naturalmente [21]. Isso é bastante relevante no contexto de SOA e Web Services, pois, em geral, os serviços são disponibilizados na Internet, estando sujeitos a diversos tipos de entradas inválidas vindas dos usuários. Assim, a injeção de falhas permite avaliar diversas situações pelas quais os serviços poderão passar ao serem publicados.

Essas falhas poderão vir principalmente das seguintes fontes:

- Usuários mal-intencionados
- Sistemas clientes apresentando defeitos

A situação de usuários mal-intencionados é bem conhecida e é tratada principalmente pela área de segurança. Ainda assim, em alguns casos, essa situação pode ser enquadrada na área de robustez. Exemplo disso ocorre com os chamados ataques de injeção de SQL. Injeção de SQL consiste em fornecer como entrada uma *string* contendo código SQL que será, indevidamente, executado pelo sistema. Essa *string* contém código SQL válido, mas é inválida em relação à informação esperada naquele momento (nome, endereço ou telefone, por exemplo). Dessa forma, pode ser considerada uma mensagem com falhas.

A situação de sistemas clientes apresentando defeitos ocorre quando existe um sistema de software chamando as operações de um Web Service e esse sistema apresenta defeitos. Essa situação é bastante apropriada para a aplicação de testes de robustez, pois alguns defeitos somente são encontrados na integração de sistemas [30]. Idealmente, pouco ou nada deve ser presumido com relação à validade da entrada de um sistema: todas as verificações de consistência possíveis devem ser feitas [27], [30]. A ocorrência de defeitos no sistema cliente pode levar à produção de falhas em sua saída, atingindo, assim, o Web Service.

No contexto de Web Services, consideramos dois tipos de falhas: as **falhas de interface** e as **falhas de comunicação** [2], [9], [28]. Falhas de interface são aquelas que afetam o conteúdo das mensagens, como a corrupção de dados. Falhas de comunicação são as que afetam a entrega das mensagens, como a queda de conexão.

2.7. Considerações finais

Este capítulo apresentou os conceitos usados como base teórica para a realização deste trabalho. A robustez é a característica de um sistema comportar-se adequadamente mesmo em situações inesperadas. Tais situações são caracterizadas pela ocorrência de falhas, erros e defeitos. A injeção de falhas é uma técnica que consiste em aplicar falhas intencionalmente e pode ser usada para avaliar a robustez de um sistema.

Web Services são uma maneira padronizada de se criar serviços de *software* independentes de linguagem de programação, e permite a implementação de sistemas desacoplados e facilmente integráveis. A injeção de falhas nesse contexto permite avaliar o comportamento dos serviços em situações inesperadas e é útil para avaliar se um Web Service é capaz de fornecer o SLA prometido.

A partir dessa fundamentação teórica, foram estudados certos modelos de falhas e ferramentas já existentes. Além disso, foi proposta uma nova ferramenta, a partir da qual foram realizados experimentos. Essas etapas estão detalhadas nos próximos capítulos.

3. Trabalhos Relacionados e Modelos de Falhas

Diversas publicações indicam a possibilidade de ocorrência de sérios problemas quando um sistema não prevê certos tipos de entrada. O trabalho de Jensen *et al.* [12] demonstra a facilidade, em alguns casos, com que se consegue causar defeitos em Web Services com a aplicação de entradas inesperadas.

Neste capítulo, discutimos alguns trabalhos relacionados contendo métodos, modelos e ferramentas de injeção de falhas. Analisamos suas principais vantagens e desvantagens.

3.1. Métodos e modelos de falhas

Um modelo de falhas define a maneira como as falhas serão injetadas no sistema em testes. O modelo pode ser mais abstrato ou mais concreto. Um modelo mais abstrato é aplicável a um número maior de cenários e domínios de experimentos, enquanto um modelo mais concreto é mais específico para determinados cenários ou domínios.

Uma técnica de teste de robustez bem conhecida é a aplicação de valores limites e valores fora do domínio de entrada esperado pelo componente ou sistema [34]. Por exemplo, se um parâmetro espera um valor inteiro positivo, alguns dos valores interessantes para serem incluídos na carga de falhas são zero, números negativos, o menor valor inteiro possível e o maior valor inteiro possível. O método mais conhecido que aplica essa ideia em seu modelo de falhas é o Ballista. Outro método bastante conhecido é o Fuzzing. Este possui como principal característica o fato de gerar entradas aleatórias, o que o torna bastante abstrato e abrangente.

Estes dois modelos possuem natureza bastante similar e baseiam-se em mensagens SOAP geradas corretamente. Com o intuito de testar a ocorrência de mensagens SOAP que possuam defeitos estruturais e aumentar a abrangência de nossos testes, propomos um terceiro modelo, baseado na alteração da estrutura XML do documento SOAP. As próximas três seções descrevem os três modelos em mais detalhes, e a seção seguinte analisa brevemente um modelo similar.

3.1.1. Fuzzing

A técnica *Fuzzing* ou *Fuzz Testing* [25] consiste em gerar casos de teste contendo dados totalmente aleatórios. Não são usados quaisquer modelos ou descrições do sistema em testes. De forma simplificada, considera-se que um caso de teste causa defeito quando a aplicação bloqueia, ou seja, deixa de responder, e que não causa defeito quando a aplicação não trava, ou seja, emite uma resposta – usamos a tradução "travar" para "*hang*". É uma técnica muito simples de ser aplicada e mostrou-se bastante eficaz em testes de interfaces de sistemas operacionais como UNIX e Windows.

Como exemplos de aplicação, os autores citam os testes de interfaces gráficas e de linha de comando em sistemas operacionais. Para interfaces em linha de comando, um caso de teste é apenas uma sequência aleatória de caracteres ASCII. Para interfaces gráficas, os casos de teste são eventos aleatórios de mouse ou teclado.

A principal vantagem desta técnica é que ela é fortemente independente do tipo de sistema em testes, não sendo necessária praticamente nenhuma especificação do sistema. Além disso, ela pode ser facilmente implementada e automatizada, e os resultados podem ser comparados entre aplicações diferentes.

Como desvantagem, podemos citar a necessidade de se gerar um número muito grande de casos de teste para se obter um bom resultado, já que as entradas são aleatórias e não há qualquer indicativo de que um caso de teste irá ou não causar defeitos. Além disso, não é dada ênfase na capacidade de repetição dos testes: nem sempre os experimentos poderão ser facilmente repetidos. Quando um caso de teste causa um defeito, ainda que ele seja registrado para poder ser executado novamente no futuro, o defeito pode ter ocorrido devido à execução prévia de outros casos de teste, e nem sempre é viável executar toda a sequência de casos de teste novamente.

3.1.2. Ballista

Ballista [6] é mais um método de testar robustez por meio de injeção de falhas. Inspirado, em parte, no Fuzz Testing, ele também se baseia no princípio de que meros dados de entrada inesperados são capazes de causar defeitos nos sistemas. Ao contrário do Fuzz Testing, no entanto, o Ballista usa um modelo de falhas mais definido, com aplicação

de valores limites e valores críticos para cada tipo de dado. A Tabela A ilustra alguns desses valores, considerando-se a linguagem C.

Tabela A. Ballista: exemplos de falhas por tipo de dado [6].

Inteiro	Ponto flutuante	Data	String	Ponteiro
MAX_INT	0	12/1/1899	BigString	NULL
MIN_INT	1	1/1/1900	StringLen1	Deleted
0	-1	2/29/1984	AllASCII	1K
1	2	4/31/1998	NonPrintable	PageSize
-1	PI	13/1/1997		MaxSize
2	PI / 2	12/0/1994		Size1
4	PI * 2	8/31/1992		Invalid
8	E	8/32/1993		
16	DBL_MAX	12/31/1999		
32	DBL_MIN	1/1/2000		
64	DBL_EPSILON	12/31/2046		
1K	-DBL_EPSILON	1/1/2047		
64K		1/1/8000		

O método Ballista pode ser usado para testar módulos de um sistema (como funções) e consiste em combinar valores válidos e inválidos para cada tipo de dado. Os autores citam o exemplo de uma função com a seguinte assinatura:

```
inttrap(double a, double b, int N);
```

Essa função possui três parâmetros, dois do tipo *double* e um do tipo *int*. Cada caso de teste gerado para testar essa função possuirá um valor para *a*, um para *b* e um para *N*, e estes serão escolhidos a partir de um conjunto de dados pré-definido contendo valores válidos e inválidos. Entre os valores inválidos estão os mencionados na tabela acima.

3.1.3. Modelo de falhas XML

XML – eXtensible Markup Language – é uma linguagem para escrita de documentos estruturados [38]. Ela é usada como base para os padrões WSDL e SOAP.

Propomos, neste trabalho, um novo modelo de falhas, específico para uso com Web Services. Este modelo é baseado na estrutura da XML e no trabalho de Jensen *et al.* [12].

Um documento XML pode ser visto como uma árvore de elementos. Um elemento é composto de uma *tag* de abertura e uma de fechamento (ou somente uma, quando o elemento for vazio), além de atributos e conteúdo (ambos opcionais). Exemplo:

```
<tag>
     <outraTag umAtributo="valor1"
               outroAtributo="valor2">
          <maisUmaTag> conteúdo </maisUmaTag>
          <elementoVazio />
     </outraTag>
</tag>
```

Figura 4. Exemplo de documento XML.

As falhas deste modelo alteram ou corrompem a estrutura do documento, com o intuito de prejudicar seu processamento. Em geral, as alterações têm o objetivo de gerar uma estrutura muito grande, difícil de ser processada corretamente; e a corrupção tem o objetivo de transformar um documento válido em inválido, tentando induzir o validador de XML do servidor a propagar essas falhas. As falhas do modelo são as seguintes:

- Elementos injetados (ex: alteração de <a> para <a>)
- Elementos muito profundos (ex: <a> <a> <a> <a> <a> <a> <a> <a> <a> ...)
- Elementos repetidos muitas vezes (ex: <a>   ...)
- Troca de caracteres válidos por inválidos (ex: <a> <#a> <#b>)

Em situações reais, falhas similares podem ocorrer principalmente nas seguintes situações:

1. Introduzidas acidentalmente por programadores
2. Introduzidas propositalmente por usuários mal-intencionados
3. Introduzidas acidentalmente por sistemas de software apresentando defeitos

Considerando-se que muitas vezes um documento XML é escrito por programadores em editores de texto simples, que não oferecem nenhuma validação, é fácil

perceber que esses documentos inválidos podem ocorrer em sistemas em produção. Por esse motivo, tais sistemas devem estar preparados para receber e tratar esse tipo de entrada, sob pena de apresentarem comportamentos inesperados caso não o façam. Entre outros exemplos, Jensen *et al.* [12] relatam casos em que a injeção de elementos pode ser usada para alterar indevidamente seu valor, representando uma possível brecha de segurança.

3.2. Perturbação de mensagens XML

Xu *et al.* [40] propõem um modelo de geração de casos de teste para Web Services por meio da perturbação de XML Schemas. Um XML Schema é um documento que define a estrutura válida para um determinado tipo de documento XML, de forma similar a uma gramática em relação a uma linguagem formal. Os autores definem uma série de operadores de perturbação que alteram a estrutura do XML Schema, e esse Schema alterado é usado como base para gerar os casos de teste.

Além deste trabalho, Offutt e Xu [26] propõem um modelo similar de perturbação, que opera diretamente em documentos XML. A aplicação deste modelo produz as mensagens SOAP alteradas, não necessitando do XML Schema. As perturbações consistem principalmente em inserir, alterar, remover ou trocar a ordem de nós da árvore XML. Também são aplicados valores limites – menor número inteiro possível, maior número inteiro possível etc. –, de maneira similar ao modelo Ballista. Almeida e Vergílio [1] e Silveira e Melo [32] estendem os modelos propostos por Xu e Offutt, propondo novos operadores de perturbação de mensagens.

Apesar de todos estes trabalhos aplicarem injeção de falhas em mensagens SOAP, eles possuem um escopo diferente: seu objetivo é encontrar defeitos funcionais, enquanto o nosso objetivo é encontrar defeitos de robustez.

3.3. Ferramentas

Comentaremos nesta seção algumas ferramentas de testes e/ou injeção de falhas encontradas em nossa revisão bibliográfica. Algumas foram projetadas especificamente para Web Services, outras são voltadas para outros fins.

Os autores do trabalho Ballista, além do método para injeção de falhas, também implementaram uma ferramenta para aplicar a técnica proposta. Infelizmente, ela não está mais disponível para uso, não podendo, portanto, ser avaliada.

3.3.1 Jaca

A ferramenta Jaca foi desenvolvida na UNICAMP por Martins, Rubira, Leme *et al.* [11], [22], [23]. Permite injeção de falhas em sistemas escritos em Java. A injeção é feita diretamente no *bytecode* (código Java compilado), não sendo necessário possuir o código-fonte do sistema a ser testado. A Figura 5 ilustra a tela inicial da ferramenta Jaca.

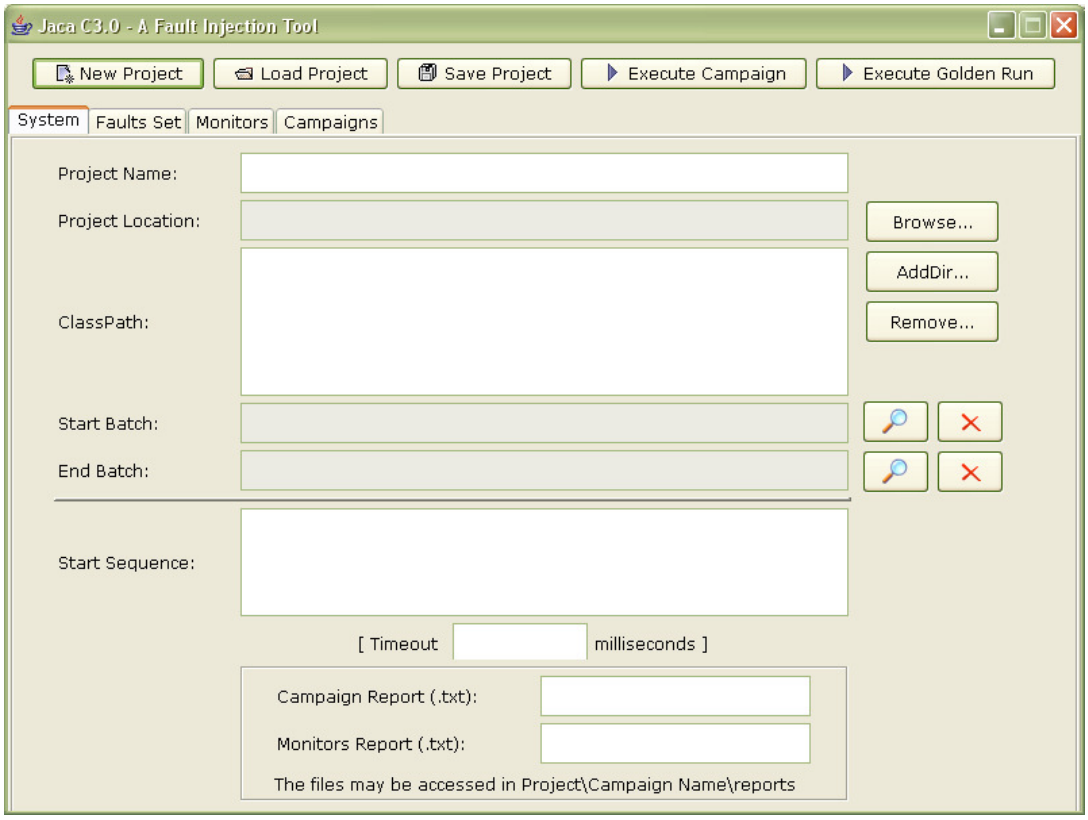

Figura 5. Ferramenta Jaca [11].

O usuário pode configurar a maneira como as falhas são injetadas especificando a classe Java e seu método (alvo da injeção), além da operação a ser realizada (substituição de valores, soma de valores, subtração de valores etc.). A implementação da ferramenta segue o padrão arquitetural *FaultInjection*, definido pelos mesmos autores [15]. A Figura 6 ilustra-o.

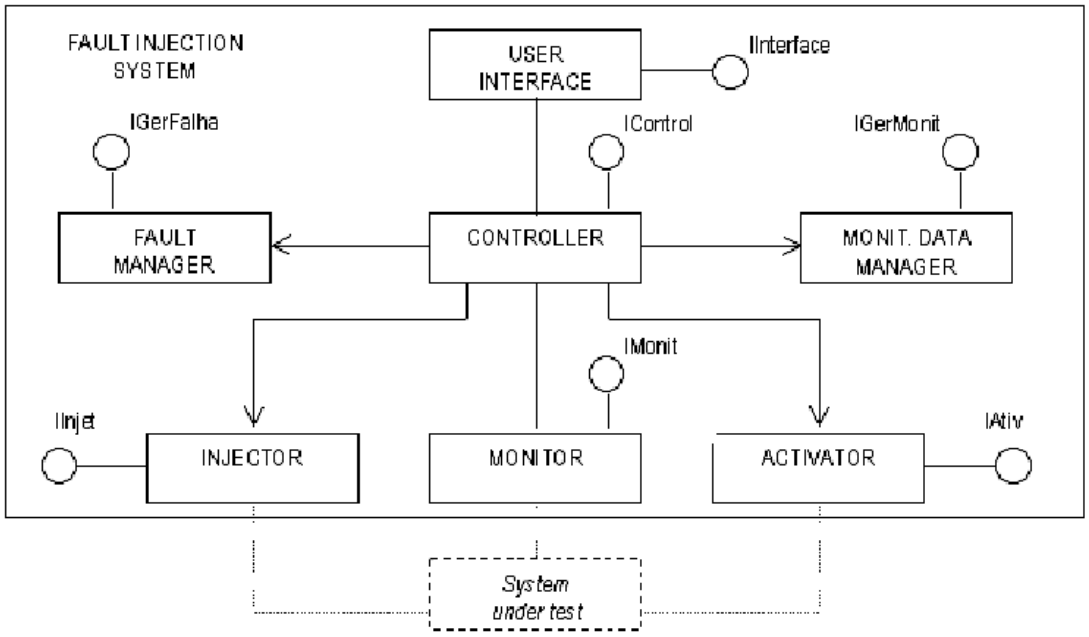

Figura 6. Padrão arquitetural FaultInjection [15].

O padrão especifica uma divisão de tarefas que uma ferramenta de injeção de falhas deverá seguir, separando os papéis de controlar interface gráfica, controlar os outros componentes, monitorar o sistema em testes etc.

A grande vantagem da ferramenta Jaca é a flexibilidade que ela oferece ao usuário de definir suas próprias falhas. Além disso, o padrão arquitetural FaultInjection também foi uma importante contribuição dos autores.

A maior limitação da ferramenta Jaca é que ela funciona apenas com software escrito em Java. Assim, não se mostra adequada para testes de Web Services, uma vez que a principal característica destes é ser independente de linguagem de programação.

3.3.2. WS-FIT

Looker *et al.* [17], [21], das Universidades de Durham e Leeds (Reino Unido), desenvolveram a ferramenta WS-FIT para injeção de falhas em Web Services. As falhas são injetadas na camada de rede, e permitem atuar em trechos específicos de mensagens SOAP.

WS-FIT funciona por meio de uma API SOAP instrumentada com código-gancho (*hook code*). Dessa forma, o ambiente de testes deverá ser modificado para permitir que as mensagens SOAP sejam interceptadas e modificadas antes de serem recebidas pelo destinatário. A Figura 7 ilustra esse cenário.

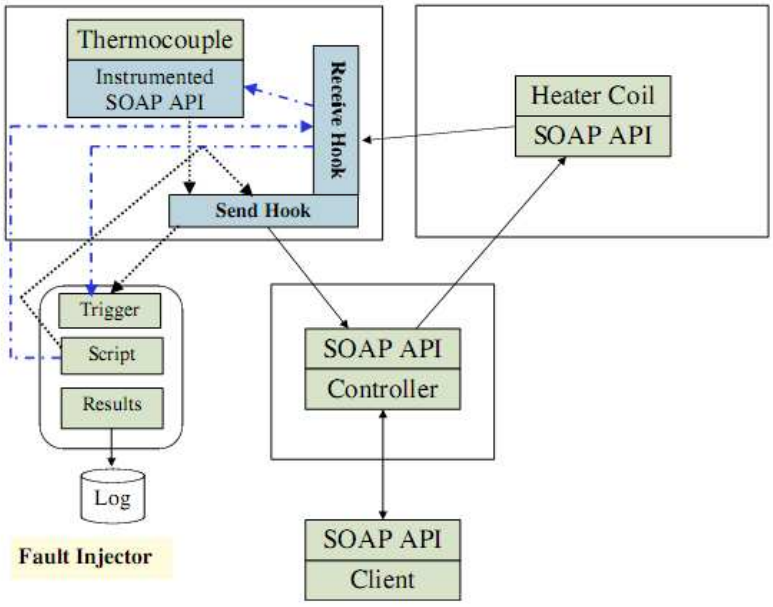

Figura 7. Arquitetura de testes da WS-FIT [21].

Uma vantagem da WS-FIT é que ela funciona por meio de scripts de injeção de falhas. Essa abordagem facilita a configuração da ferramenta e possibilita ao usuário ter mais controle sobre seu experimento.

Existem dois problemas fundamentais com a ferramenta WS-FIT. O primeiro é que ela não está disponível para ser baixada e usada, não sendo possível saber mais detalhes sobre seu funcionamento. O segundo problema é a necessidade da instrumentação da API SOAP com código-gancho. Essa necessidade impõe restrições ao seu uso em sistemas SOA que não ofereçam essa possibilidade.

3.3.3. wsrbench

Desenvolvida na Universidade de Coimbra, wsrbench [14] é voltada especificamente para testes de robustez em Web Services por meio de injeção de falhas. A ferramenta está disponível para uso em seu website[5].

[5] http://wsrbench.dei.uc.pt/

Não é necessário baixar a ferramenta para usá-la, pois ela é executada *online*, bastando ser acionada pelo usuário. Para tanto, este deverá apenas preencher um breve cadastro e informar o endereço do documento WSDL do serviço a ser testado, conforme Figura 8.

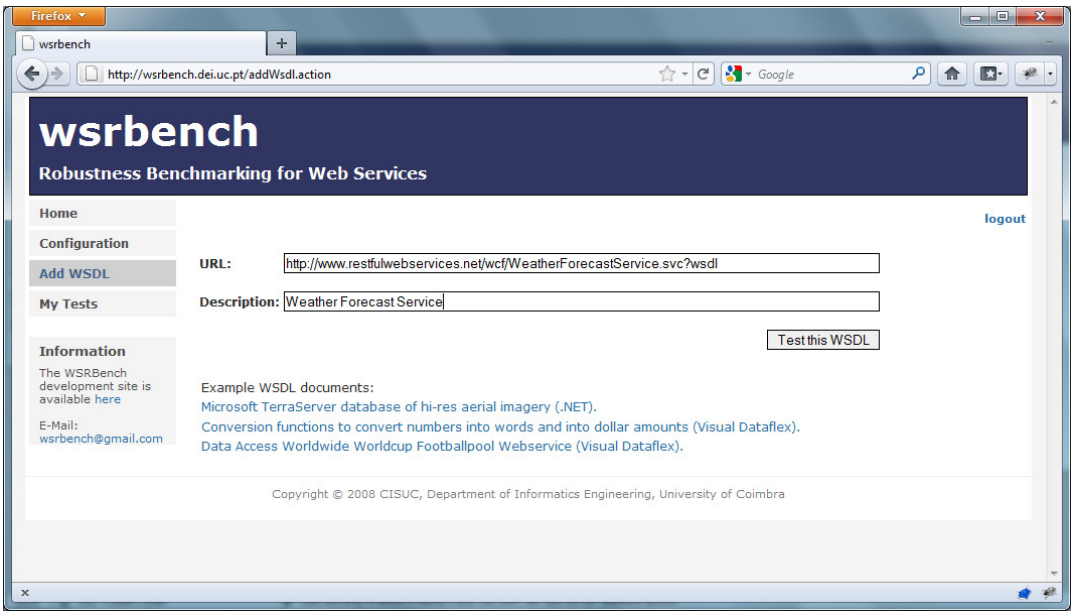

Figura 8. Tela da wsrbench[5].

A partir do momento em que o usuário dispara os testes, estes são iniciados e o usuário é informado por e-mail quando forem concluídos. A wsrbench opera em duas fases: na primeira, o serviço é chamado apenas com entradas válidas para se obter uma medida do funcionamento do serviço. Na segunda, são introduzidas falhas para se verificar se ocorrerão mudanças no comportamento do serviço. A Figura 9 ilustra a arquitetura resumida da wsrbench.

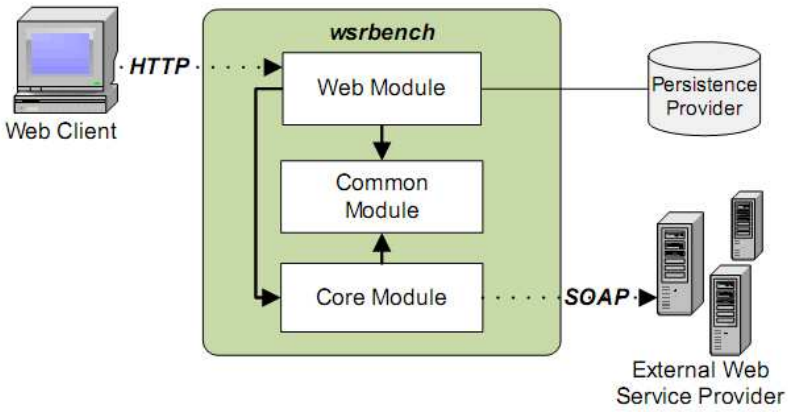

Figura 9. Arquitetura resumida da wsrbench [14].

O modelo de falhas utilizado foi baseado no modelo usado pelo método Ballista. Não há possibilidade de escolher quais falhas serão injetadas: existe um conjunto pré-definido delas, determinado pelos tipos de dados esperados pelas operações de cada serviço, conforme mostrado na Tabela B.

Tabela B. Falhas injetadas por wsrbench [35].

Tipo	Falha	Ação
String	StrNull	Substituir por valor nulo
	StrEmpty	Substituir por vazio
	StrPredefined	Substituir por valor predefinido
	StrNonPrintable	Substituir por caracteres não-imprimíveis
	StrAddNonPrintable	Adicionar caracteres não-imprimíveis
	StrAlphaNumeric	Substituir por caracteres alfanuméricos
	StrOverflow	Substituir por *string* muito grandes
Número	NumNull	Substituir por valor nulo
	NumEmpty	Substituir por vazio
	NumAbsoluteMinusOne	Substituir por -1
	NumAbsoluteOne	Substituir por 1
	NumAbsoluteZero	Substituir por 0
	NumAddOne	Adicionar 1
	NumSubtractOne	Subtrair 1
	NumMax	Substituir por valor máximo válido para o tipo
	NumMin	Substituir por valor mínimo válido para o tipo
	NumMaxPlusOne	Substituir por valor máximo válido para o tipo mais 1
	MumMinMinusOne	Substituir por valor mínimo válido para o tipo menos 1

Como principais vantagens, podemos citar sua facilidade de uso, seu alto grau de automatização e sua disponibilidade de acesso para qualquer pessoa; além disso, os autores

realizaram testes em vários Web Services disponíveis na Internet, obtendo interessantes resultados, com altas taxas de defeito encontradas [14].

Uma limitação chave da wsrbench é que ela não é extensível. O formato das mensagens de requisição a serem geradas é pré-definido e não pode ser alterado. Se o serviço a ser testado necessitar de qualquer detalhe específico para funcionamento, como uma mensagem SOAP especial (ex: no formato WS-Security – um padrão de recursos de segurança para Web Services[6]), não será possível usar a wsrbench.

3.3.4. soapUI

soapUI é uma ferramenta especializada em testes de Web Services, disponível para uso gratuitamente. Atualmente, ela é mantida pela empresa eviware e está disponível para download em seu *website*[7]. Conforme o nome sugere, sua principal característica é possuir uma interface gráfica para fácil acesso às funcionalidades, apesar de também poder ser executada em linha de comando. A Figura 10 ilustra a interface gráfica da ferramenta.

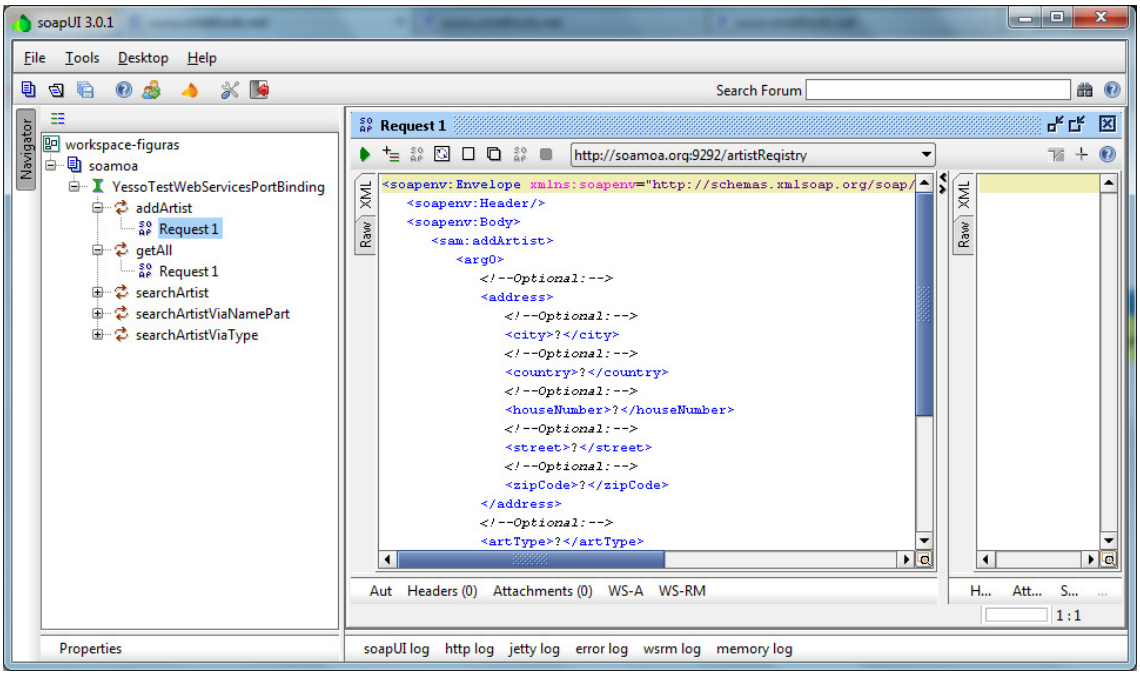

Figura 10. soapUI – interface gráfica[7].

[6] http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[7] http://soapui.org/

Assim como a wsrbench, soapUI também permite informar o endereço de um documento WSDL para que seus dados sejam processados e sejam geradas automaticamente chamadas às operações de um Web Service. A diferença é que o usuário poderá alterar a mensagem SOAP como desejar, para enviá-la ao serviço em seguida e observar sua resposta.

Destacamos sua facilidade de uso, sua disponibilidade para uso e sua alta popularidade. Além de fazer chamadas avulsas às operações de um Web Service, ela também possui facilidades para automatização de testes usando a interface de linha de comando.

Sua maior limitação é que, ao contrário das outras ferramentas citadas, soapUI não possui funcionalidades para injeção de falhas. *Grosso modo*, soapUI é tão somente uma interface para enviar e receber mensagens SOAP para e de um serviço. Atividades que não se enquadrem nessas possibilidades (ex.: alterar uma mensagem, atrasar a entrega de uma mensagem) deverão ser feitas de forma externa à ferramenta.

3.4. Considerações finais

A análise das ferramentas existentes permitiu-nos identificar algumas características interessantes em ferramentas de injeção de falhas em Web Services, além de limitações comuns a todas elas. Como limitações comuns, podemos citar duas:

1. Impossibilidade de testar adequadamente Web Services compostos
2. Ferramenta atua como cliente do serviço, não sendo possível usar o cliente já existente nos testes

Testes de Web Services compostos permitiriam ao usuário escrever casos de teste que explorem a comunicação entre os diversos serviços que compõem um serviço composto. No exemplo da Figura 3, verificamos um serviço de reserva de viagens, composto por um serviço de reserva de passagens aéreas, outro de reserva de hotéis e outro de reserva de veículos. Caso a ferramenta permita o teste somente de serviços simples, todo o sistema será visto como caixa-preta, e falhas poderão ser injetadas somente entre o cliente

e o serviço de reserva de viagens. A possibilidade de injeção de falhas entre o serviço composto e seus componentes aumentaria as possibilidades de se descobrirem defeitos.

A outra limitação é o fato de essas ferramentas atuarem como cliente dos serviços. Esse fato significa que o uso dessas ferramentas implica em excluir da arquitetura de testes o cliente do sistema. Dessa forma, a arquitetura de testes será significativamente diferente da original, podendo, por exemplo, acarretar em casos de teste resultando em falsos positivos ou negativos. Seria melhor que o cliente participasse dos testes e que a ferramenta fosse apenas um elemento a mais na arquitetura.

As seguintes características foram apontadas como desejáveis em ferramentas de injeção de falhas em Web Services:

1. Flexibilidade na escolha do modelo de falhas (ex: Jaca)
2. Baixa intrusividade
3. Funcionamento dirigido por *scripts* (ex: WS-FIT)
4. Possibilidade de uso tanto por interface gráfica como por linha de comando (ex: soapUI)

A flexibilidade para escolha do modelo de falhas permite que o usuário escolha o modelo mais adequado para o serviço em testes. Não havendo essa flexibilidade, o usuário ficará preso às possibilidades da ferramenta, como no caso da wsrbench, que não é capaz de testar serviços que usem WS-Security.

A baixa intrusividade é caracterizada pela necessidade de se alterar o mínimo possível de configurações na arquitetura de testes do usuário. Essa característica é importante, pois garante que os testes serão feitos refletindo-se o máximo possível a situação normal de operação do sistema. Isso evita, por exemplo, a geração de muitos casos de teste que causem defeitos durante os experimentos, mas não no sistema em produção.

O funcionamento dirigido por scripts oferece grande facilidade ao usuário para definir seus experimentos. Além disso, permite que ele defina o modelo de falhas a ser usado.

Finalmente, a possibilidade de usar a ferramenta com interface gráfica facilita a adaptação do usuário ao ambiente da ferramenta, enquanto a interface em linha de comando permite que o usuário automatize seus testes com mais facilidade.

4. WSInject

Com o objetivo de implementar uma ferramenta com as características apontadas no capítulo anterior, foi concebida a WSInject. Ela foi concebida por Valenti, Maja, Bessayah *et al.* [2], [33] e implementada por Valenti e Maja. Desenvolvida na linguagem Java, é uma ferramenta de injeção de falhas em tempo de execução, no nível HTTP, aplicável a Web Services simples e compostos. Seu funcionamento é baseado na interceptação de mensagens trocadas entre cliente e serviço. No caso de serviços compostos, ela também é capaz de interceptar mensagens trocadas entre o serviço composto e seus serviços parceiros.

A WSInject pode ser executada tanto em linha de comando quanto em modo de interface gráfica. A Figura 1 ilustra seu funcionamento no modo de interface gráfica. Cada linha da tabela na parte superior da tela representa um par de mensagens requisição/resposta. As áreas de texto (com fundo branco) na parte inferior da tela apresentam o conteúdo de uma mensagem antes e depois da injeção de falhas.

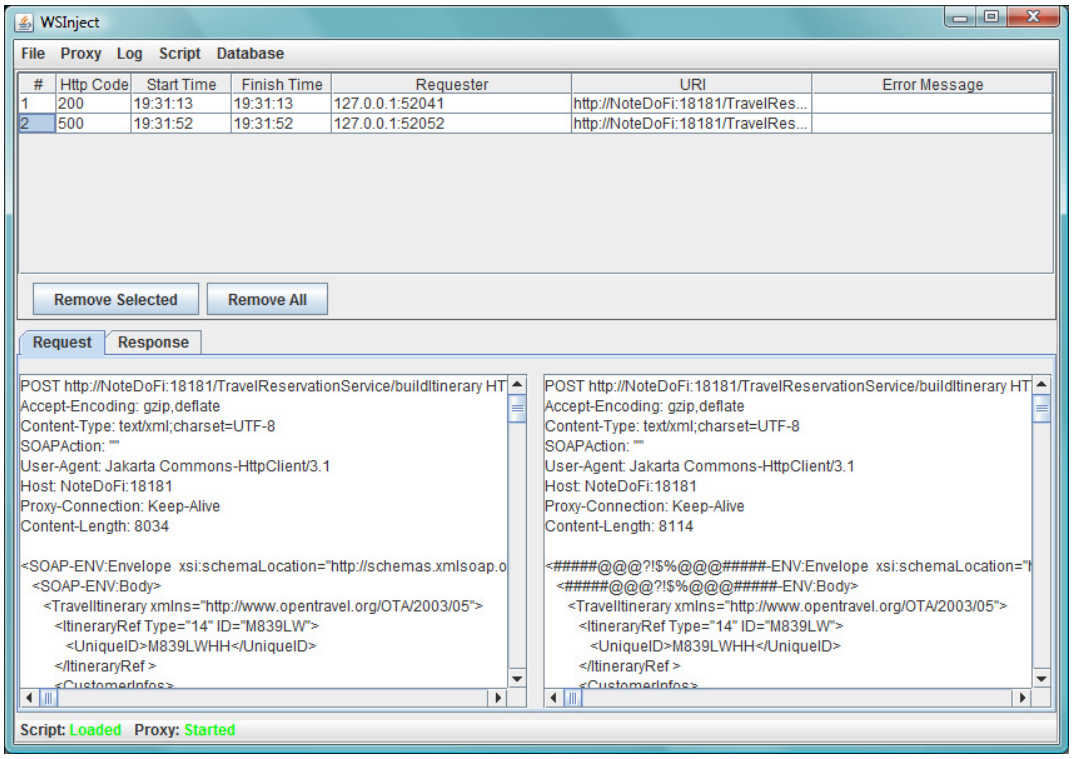

Figura 11. WSInject em funcionamento [33].

4.1. Visão geral

O funcionamento baseado na interceptação de mensagens é a principal característica da WSInject. Ferramentas de injeção de falhas em Web Services normalmente fazem o papel do cliente, substituindo-o na arquitetura de testes. A WSInject, por outro lado, trabalha em conjunto com o cliente, sendo este seu maior diferencial. As figuras a seguir ilustram essa diferença.

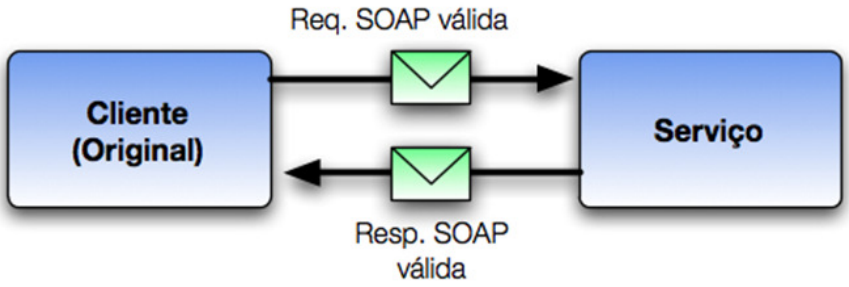

Figura 12. Arquitetura original do sistema (sem ferramenta de injeção de falhas).

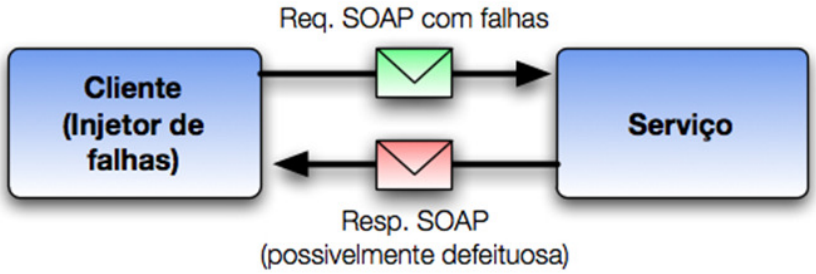

Figura 13. Substituição do cliente original por ferramenta de injeção de falhas.

Figura 14. WSInject atuando como injetor de falhas no momento da requisição.

Figura 15. WSInject atuando como injetor de falhas no momento da resposta.

A Figura 14 e a Figura 15 ilustram momentos diferentes da atuação da WSInject. Na primeira, as falhas são injetadas na requisição do cliente ao serviço. Na segunda, a injeção ocorre na resposta do serviço ao cliente. Essas duas injeções podem ser feitas isoladamente ou em conjunto, a critério do usuário.

Note que a WSInject não elimina o cliente da arquitetura de testes, apenas atua como mediadora da comunicação entre este e o serviço em testes. A WSInject comporta-se como um servidor *proxy* HTTP comum. O cliente deverá apenas ser configurado para se conectar por meio de um servidor *proxy* e não terá conhecimento de que suas mensagens serão interceptadas por um injetor de falhas – daí a baixa intrusividade. O teste do sistema como um todo, em contraste com o teste somente do serviço, apresenta duas importantes vantagens: 1) o teste é feito em um cenário muito próximo da situação real de uso do sistema; 2) além do serviço, é possível testar o cliente, que também pode ser uma fonte de falhas, erros e defeitos. O teste que pode ser realizado no cliente é do tipo caixa preta, pois as falhas são injetadas em sua interface, sem que se tenha acesso a seu código-fonte.

4.2. Implementação

A Figura 16 ilustra a arquitetura da WSInject.

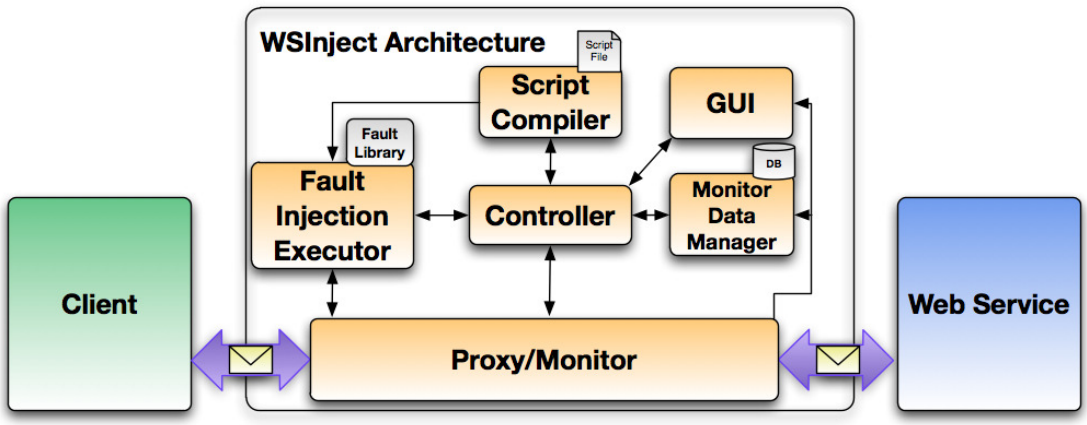

Figura 16. Arquitetura da WSInject [33].

Essa arquitetura é baseada no padrão de arquiteturas de injetores de falhas usado pela ferramenta Jaca, definido por Leme *et al*. [15].

Em uma situação de uso normal do sistema (sem injeção de falhas), o cliente se comunicaria diretamente com o Web Service. Em uma situação de injeção de falhas, a ferramenta estará presente entre o cliente e o serviço.

Os componentes mais importantes da arquitetura são **Proxy/Monitor** e **Fault Injection Executor**. Proxy/Monitor é o ponto de interceptação de mensagens SOAP e monitoramento da ocorrência de defeitos. Fault Injection Executor é o ponto onde as falhas são efetivamente injetadas.

Outros componentes importantes são **Controller**, **Script Compiler**, **Monitor Data Manager** e **GUI (Graphical User Interface)**. Controller é o ponto de partida da ferramenta, responsável por iniciar os outros componentes. Script Compiler é o componente que lê os scripts de injeção de falhas e converte-os para um formato que a ferramenta possa processar. Monitor Data Manager é responsável por armazenar as informações mantidas por Proxy/Monitor em um banco de dados. Finalmente, GUI é responsável por exibir graficamente essas mesmas informações (caso a ferramenta seja iniciada em modo gráfico).

4.3. Sistema de scripts

Na WSInject, são usados scripts para descrição das falhas a serem injetadas. Scripts são simples arquivos de texto contendo um ou mais **FaultInjectionStatement**s (comandos

de injeção de falhas). Cada FaultInjectionStatement é composto de um **ConditionSet** (conjunto de condições) e uma **FaultList** (lista de falhas).

Os FaultInjectionStatements funcionam como comandos do tipo *condição-ação*: ao interceptar uma mensagem, se ela satisfizer um conjunto de condições, uma lista de falhas é injetada nela. As condições são como métodos **boolean** de Java e as falhas são como métodos **void**. Condições não possuem ordem definida, por isso são agrupadas em um *conjunto*, enquanto falhas possuem ordem definida, por isso são agrupadas em uma *lista*.

A Tabela C mostra as condições disponíveis, a Tabela D mostra as falhas estáticas disponíveis na biblioteca de falhas da WSInject e a Figura 17 mostra um exemplo de script. Falhas estáticas são aquelas que produzem sempre o mesmo resultado. As falhas dinâmicas serão discutidas na seção seguinte.

Tabela C. Condições disponíveis [33].

Nome	Sintaxe	Descrição
ContainsCondition	**contains**(String stringPart)	Seleciona mensagens SOAP contendo a *string* especificada.
URICondition	**uri**(String uriPart)	Seleciona tanto as requisições enviadas à URI contendo a *string* especificada como as respostas a essas requisições.
MessageDestinationCondition	**isRequest**() **isResponse**()	Seleciona somente requisições, tanto de um cliente para um serviço como de um processo BPEL para um serviço parceiro. Seleciona somente respostas, tanto de um serviço para um cliente como de um serviço parceiro para um processo BPEL.
SoapActionCondition	**soapAction**(String soapAction)	Seleciona as requisições contendo uma determinada SoapAction nos cabeçalhos HTTP e as respostas a essas requisições. SoapAction é um cabeçalho HTTP usado por alguns Web Services para determinar a operação do serviço a ser chamada.

Tabela D. Falhas estáticas disponíveis [33].

Nome	Sintaxe	Descrição
FALHAS DE INTERFACE		
StringCorruptionFault	**stringCorrupt**(String fromString, String toString)	Substitui ocorrências de *fromString* por *toString*. Trata os dados apenas como *Strings*, ignorando completamente a sintaxe XML. Pode ser usada para substituir caracteres XML, como '<' e '>'.
XPathCorruptionFault	**xPathCorrupt**(String xPathExpression, String newValue)	Substitui todas as ocorrências de uma expressão XPath[8] pelo valor especificado. Pode ser usada para modificar atributos ou elementos.
MultiplicationFault	**multiply**(String xPathExpression, int multiplicity)	Multiplica parte de uma mensagem por um número específico de vezes. Exemplos: *multiply*("/", 2) duplica todo o conteúdo da mensagem. *multiply*("/Envelope/MyElement", 3) triplica somente o elemento "MyElement".
EmptyingFault	**empty**()	Esvazia a mensagem SOAP, entregando uma mensagem HTTP sem qualquer conteúdo.
CoerciveParsingFault	**coerciveParse**(int depth)	Cria uma sequência de *tags* de abertura de elementos XML sem fechá-los, permitindo criar uma árvore XML muito profunda (como <tag> <tag> <tag> <tag> <tag> <tag>...)
FALHAS DE COMUNICAÇÃO		
DelayFault	**delay**(int delayInMilliseconds	Atrasa a entrega de uma mensagem em um determinado número de milissegundos.
ConnectionClosingFault	**closeConnection**()	Fecha abruptamente a conexão entre cliente e *proxy*.

[8] http://www.w3.org/TR/xpath/

```
uri("Hotel"): stringCorrupt("Name", "Age"), multiply("/", 2);
uri("Airline"): stringCorrupt("Flight", "Might");
contains("caught exception") && isResponse(): empty();
```

Figura 17. Exemplo de script de injeção de falhas [33].

O exemplo acima possui três *FaultInjectionStatements*, um em cada linha de texto. O primeiro possui um *ConditionSet* de uma única condição: uma *URICondition* com um argumento "Hotel". Também especifica uma *FaultList* com duas falhas: *StringCorruptionFault* com os argumentos "Name" e "Age", e uma *MultiplicationFault* com os argumentos "/" e '2'. O segundo *FaultInjectionStatement* possui um *ConditionSet* com uma *URICondition* e uma *FaultList* com uma *StringCorruptionFault*. O último *FaultInjectionStatement* possui um *ConditionSet* com duas condições: uma *ContainsCondition* e uma *MessageDestinationCondition*, além de uma *FaultList* com uma *EmptyingFault*. Esse script descreve a seguinte campanha:

- Toda vez que a URI de uma chamada a Web Service ou a resposta a essa chamada contiver a string "Hotel":
    1. Substitua todas as ocorrências de "Name" por "Age".
    2. Duplique todo o conteúdo da mensagem SOAP.
- Toda vez que a URI de uma chamada a Web Service ou a resposta a essa chamada contiver a string "Airline":
    1. Substitua todas as ocorrências de "Flight" por "Might".
- Toda vez que uma mensagem contiver a string "caught exception" e for uma resposta a uma chamada a Web Service:
    1. Esvazie a mensagem.

4.4. Modelos de falhas e geração dinâmica de falhas

A seção anterior apresentou as falhas estáticas que a WSInject oferece. Essas falhas são injetadas diretamente pela WSInject e produzem sempre o mesmo resultado toda vez em que são acionadas. A ferramenta também oferece a possibilidade de usar **falhas**

dinâmicas. As falhas dinâmicas delegam a **geradores de falhas** a tarefa de produzir as falhas que serão injetadas. A Figura 18 ilustra esse cenário.

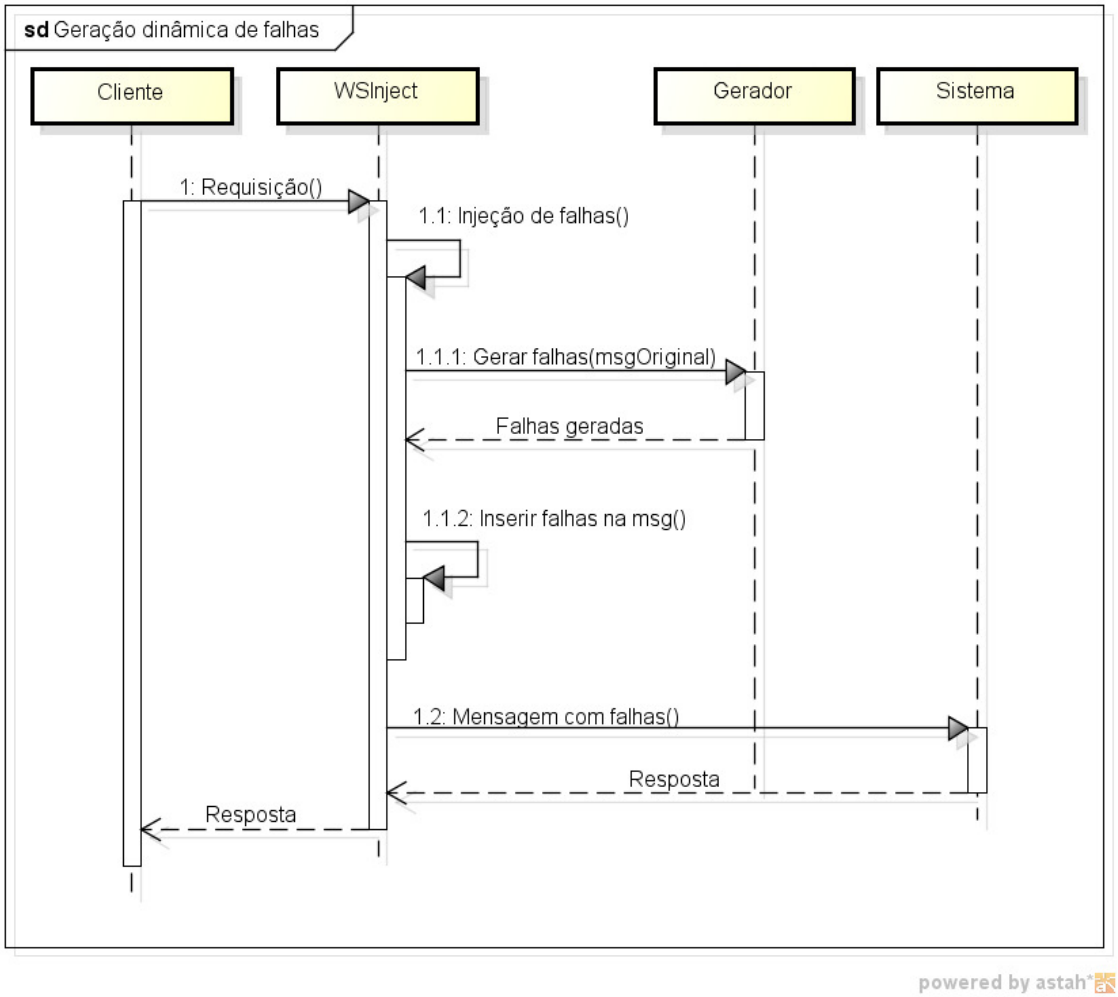

Figura 18. Geração dinâmica de falhas.

Atualmente, estão implementadas as seguintes falhas dinâmicas: *DynamicStringCorruptionFault*, *DynamicXPathCorruptionFault* e *DynamicFullMessageCorruptionFault*. As duas primeiras funcionam de maneira similar às suas versões estáticas. A terceira é usada para se ter controle total sobre a mensagem SOAP. As falhas dinâmicas, em vez de especificarem uma *string* para substituição do valor encontrado (como fazem as falhas estáticas), especificam um gerador. Um gerador de falhas é simplesmente uma classe Java que implementa a interface **FaultGenerator**,

definida pela WSInject. Essa interface descreve um método, **generateNewValue**, que gera o próximo valor a ser inserido na mensagem SOAP.

O usuário pode escrever seus próprios geradores e integrá-los à ferramenta em tempo de execução. Isso é possível na plataforma Java por meio de reflexão computacional. Para criar e usar seu próprio gerador de falhas, o usuário deve:

- Escrever e compilar uma classe Java que implemente a interface FaultGenerator;
- Incluir essa classe no *classpath* ao iniciar a WSInject;
- Escrever um script que use uma falha dinâmica e o seu gerador.

Dado que um modelo de falhas consiste na especificação de diversas falhas a serem injetadas em um sistema, o uso de um gerador dinâmico possibilita implementar um modelo de falhas de maneira executável pela WSInject. A WSInject já oferece dois modelos de falhas em sua biblioteca: *Fuzzing* e Ballista. A seguinte classe Java implementa o modelo *Fuzzing*:

```
public class Fuzzer implements FaultGenerator {
	private Random random = new Random();

	@Override
	public String generateNewValue(String oldValue) {
		int length = random.nextInt(1024);

		StringBuilder sb = new StringBuilder();

		for (int i = 0; i < length; ++i) {
			sb.append((char) random.nextInt(128));
		}

		return sb.toString();
	}

}
```

Figura 19. Classe geradora de falhas implementando o modelo *Fuzzing*.

O seguinte script pode ser usado para aplicar o modelo *Fuzzing* a todos os elementos XML em todas as mensagens de requisição:

```
isRequest():
      dynamicXPathCorrupt("//text()",
"br.unicamp.ic.robustweb.wsinject.addons.dynamicfaultgenerators.Fu
zzer");
```

Figura 20. Script de injeção de falhas dinâmicas.

O script acima usa caminho "//text()" (que corresponde ao conteúdo de todos os elementos do documento XML) e o gerador de falhas Fuzzer (localizado no pacote Java br.unicamp.ic.robustweb.wsinject.addons.dynamicfaultgenerators). Dessa forma, trocará o conteúdo de todos os elementos XML pelos valores gerados por chamadas ao método generateNewValue da classe Fuzzer.

4.5. Uso em composições de serviços

Além de permitir realizar testes do tipo caixa-preta em serviços compostos (enxergando-os como se fossem serviços simples), WSInject também permite a injeção de falhas entre uma composição e seus serviços parceiros. Para isso, é necessário apenas configurar o *BPEL Engine* para que se conecte por meio de um proxy. Esse cenário é ilustrado na Figura 21.

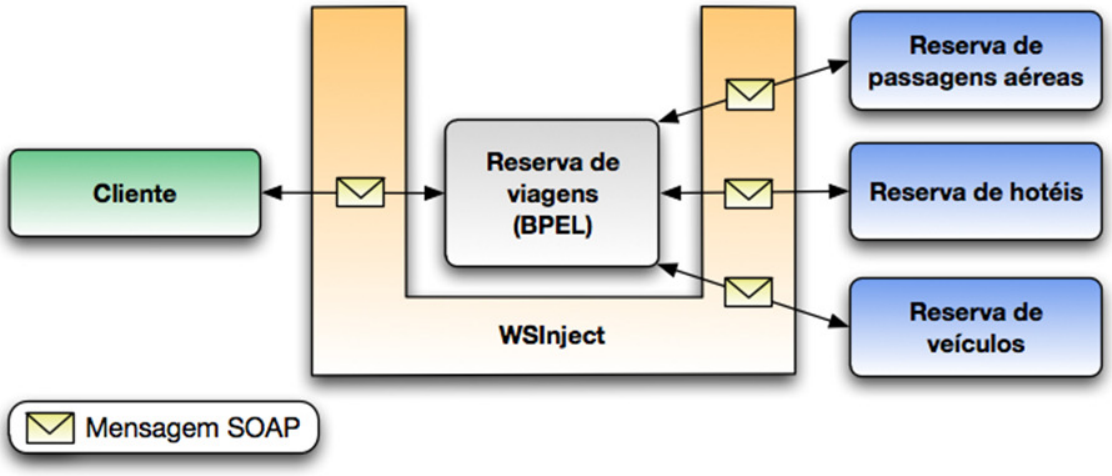

Figura 21. WSInject atuando em uma composição de serviços.

Nessa configuração, a WSInject intercepta tanto as mensagens trocadas entre o cliente e a composição quanto entre a composição e seus serviços parceiros. Dessa forma, pode injetar falhas entre quaisquer dois componentes – representados na figura como retângulos arredondados.

4.6. Limitações

A principal limitação da WSInject está na sua capacidade de emular falhas de comunicação. Apesar de disponibilizar as falhas de atraso e fechamento de conexão (*DelayFault* e *ConnectionClosingFault*), estas não emulam perfeitamente a situação em que um serviço está fora do ar. Isso ocorre porque a WSInject trabalha no nível HTTP (camada de aplicação). A WSInject não possui controle sobre as camadas inferiores (TCP, IP etc.). Numa situação real de serviço fora do ar, as tentativas de conexão efetuadas por essas camadas inferiores seriam mal-sucedidas. No caso da WSInject, as camadas TCP e IP sempre se comunicam corretamente, o que pode ser interpretado pelo cliente ou pelo serviço como uma conexão realizada com sucesso. A emulação realmente adequada de um serviço fora do ar seria possível somente trabalhando-se no nível IP (camada de rede), conforme proposto por Reinecke e Wolter [28].

A decisão de trabalhar apenas no nível HTTP foi tomada por dois motivos: 1) maior facilidade de se trabalhar somente nesta camada, tanto em termos de programação como em termos da praticidade de acesso ao conteúdo da mensagem SOAP; 2) bons resultados obtidos em outros trabalhos que adotaram essa abordagem [12], [14]. Ainda assim, um possível trabalho futuro seria a integração da WSInject com alguma ferramenta capaz de injetar falhas na camada IP. Menegotto e Weber [24] propõem que a injeção de falhas deve considerar as diversas camadas de comunicação.

Outra limitação está no seu modelo temporal de injeção de falhas: atualmente, a maioria das falhas somente pode ser injetada de forma **permanente**, ou seja, sempre que a condição especificada é satisfeita. Outras ferramentas permitem que as falhas sejam configuradas como **permanentes**, **intermitentes** ou **transientes** [9]. As falhas transientes são injetadas apenas uma vez na execução dos testes, enquanto as intermitentes são injetadas com distribuição de probabilidades especificada pelo usuário (uniforme, exponencial, normal etc.). Esse modelo temporal permite especificar falhas mais próximas

daquelas encontradas em sistemas reais. É importante destacar, no entanto, que essa limitação existe apenas para as falhas estáticas da biblioteca da WSInject, já que o uso de falhas dinâmicas permite ao usuário programar sua própria injeção de falhas, o que inclui o modelo temporal a ser usado.

4.7. Considerações finais

Este capítulo apresentou a WSInject, ferramenta de injeção de falhas em Web Services implementada neste trabalho. Ela foi baseada nas ferramentas já existentes estudadas, considerando seus pontos fortes e suas limitações. A Tabela E compara a WSInject com as ferramentas propostas nos trabalhos relacionados.

Tabela E. Comparativo das ferramentas.

	Jaca	WS-FIT	wsrbench	soapUI	WSInject
Nível de intrusividade	Médio	Alto	Médio	Baixo	Muito baixo
Falhas configuráveis	✓	✓			✓
Funcionamento automático			✓		
Ferramenta específica para injeção de falhas em Web Services		✓	✓		✓
Extensível					✓
Facilidades para testar serviços compostos					✓

A implementação da WSInject tornou possível a realização de experimentos utilizando diferentes modelos de falhas. Esses experimentos são apresentados no capítulo seguinte.

5. Experimentos

Neste capítulo, apresentamos os experimentos realizados com a WSInject com o objetivo de validá-la em diferentes cenários de injeção de falhas em Web Services. A ferramenta WSInject foi concebida e desenvolvida por Valenti, Maja e Bessayah, em conjunto. Já os experimentos foram conduzidos de forma individual. No presente trabalho, os experimentos realizados foram:

1. Processo BPEL: *TravelReservationService*
2. Sistema real
3. Integração WSInject + wsrbench

O experimento #1 foi realizado no início do desenvolvimento da WSInject, funcionando como prova de conceito de sua capacidade de injetar falhas, principalmente em composições de serviços. Esse experimento preliminar foi feito no *TravelReservationService*, uma aplicação de exemplo do NetBeans implementada como uma composições de serviços em BPEL.

O experimento #2 teve como intuito medir a eficácia da ferramenta quando aplicada a sistemas reais. Contamos com o apoio de uma empresa de desenvolvimento de software localizada na região de Campinas-SP, que gentilmente ofereceu seu sistema baseado em Web Services para que realizássemos nossos testes. Foi desenvolvido um cliente experimental simples para esta aplicação, com o objetivo de simular a presença de um cliente real na arquitetura de testes. Por questões de sigilo, alguns nomes relacionados ao sistema, serviço ou detalhes de implementação foram omitidos ou trocados.

O experimento #3 consistiu em integrar a WSInject com a ferramenta wsrbench, desenvolvida em Coimbra. Esse experimento teve como objetivo demonstrar a capacidade de extensão e integração da WSInject, abrindo outras possibilidades de uso, mesmo quando a outra ferramenta não oferece nenhuma facilidade para integração ou extensão.

Os experimentos estão detalhados nas seções a seguir.

5.1. Experimento #1: TravelReservationService (TRS)

TravelReservationService (TRS) é uma aplicação demonstrativa disponível na ferramenta de desenvolvimento NetBeans[9]. Essa aplicação simula um sistema de reserva de viagens composto por três serviços: reserva de passagem aérea, reserva de hotel e reserva de veículo para aluguel. Escolhemos essa aplicação pelo fato de ela estar disponível abertamente para uso. Esta foi a única aplicação baseada em BPEL que encontramos disponível para uso na internet. Apesar de simples, ela serve ao propósito dos testes preliminares com a WSInject.

Os experimentos com o TRS tiveram o intuito de verificar a habilidade da WSInject de injetar falhas em Web Services compostos. Nesse experimento, somente falhas de corrupção foram aplicadas. Nossa plataforma de testes foi baseada em ferramentas abertas à comunidade Java, de modo a tornar os testes facilmente repetíveis.

Um servidor de aplicação é um sistema de software que permite implantar aplicações web e componentes como Web Services. Após implantados, os Web Services ficam disponíveis para serem chamados pelos usuários. Em nosso experimento, usamos o servidor de aplicação GlassFish v2.1[10]. Web Services podem ser diretamente implantados no GlassFish. Já processos BPEL podem ser implantados no GlassFish após a instalação de um *BPEL engine*. Em nosso experimento, usamos o OpenESB V2[11]. Também usamos o ambiente NetBeans 6.5.1, capaz de gerenciar o GlassFish e o OpenESB. A Figura 22 ilustra a arquitetura de testes utilizada.

[9] http://www.netbeans.org/
[10] https://glassfish.dev.java.net/
[11] https://open-esb.dev.java.net/

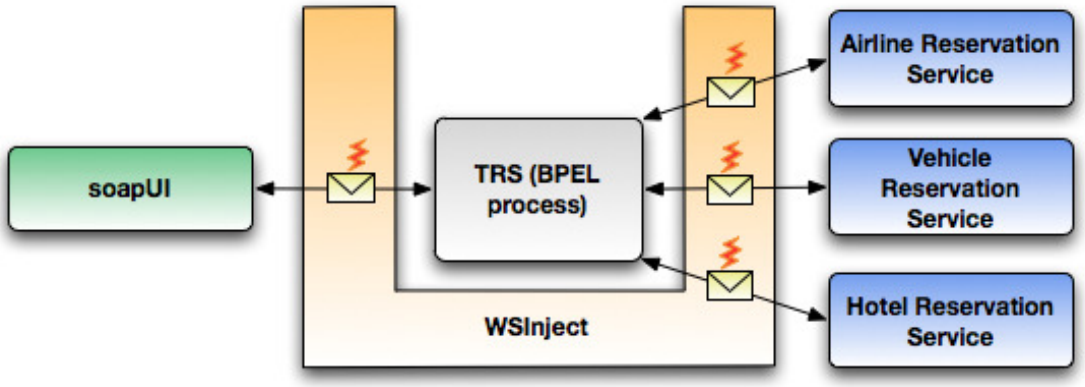

Figura 22. Arquitetura de testes do experimento com TRS.

Nossa arquitetura de testes foi composta pela ferramenta soapUI (usada para gerar requisições SOAP), a WSInject, o processo BPEL e os serviços parceiros (que implementam as funcionalidades individuais do sistema). Os serviços parceiros do TRS são *VehicleReservationService*, *AirlineReservationService* e *HotelReservationService*. Eles são orquestrados pelo processo BPEL para se obter uma reserva de viagem completa.

A ferramenta soapUI atuou como cliente do TRS, sendo responsável por originar requisições que ativaram o processo BPEL. Este, por sua vez, chamou os serviços parceiros para fazer as reservas. Tanto soapUI como GlassFish foram configurados para conectarem-se por meio do *proxy* da WSInject. Dessa forma, toda a comunicação entre cliente, processo BPEL e serviços parceiros foi interceptada pela WSInject, possibilitando que esta injetasse falhas em qualquer mensagem SOAP do sistema em testes.

5.1.1. Carga de trabalho e carga de falhas

Dois experimentos de corrupção de mensagem foram realizados. Consistiram em injetar falhas de corrupção em todas as mensagens trocadas no sistema, tanto entre cliente e processo BPEL como entre processo BPEL e serviços parceiros. O primeiro experimento consistiu na corrupção de todos os valores do tipo inteiro. O segundo consistiu na corrupção de todos os valores do tipo *string*. Isso foi aplicado tanto ao conteúdo de elementos (ex: `<elemento> 1234 </elemento>`) como atributos (ex: `<elemento atributo="1234" />`). Os experimentos foram baseados no modelo de falhas Ballista

do modo como foi implementado na ferramenta wsrbench, descrito na Tabela B do Capítulo **3. Trabalhos Relacionados e Modelos de Falhas**.

No NetBeans, o TRS já vem com casos de teste pré-definidos: *hasAirline*, *hasHotel*, *hasVehicle* and *hasNoReservations*. São testes funcionais para verificar a operação correta do sistema. A requisição SOAP do caso de teste *hasNoReservations* (também chamado de *TestCase1* em algumas versões do NetBeans) foi usada para ativar o TRS nos experimentos. A Figura 23 mostra o conteúdo da requisição deste caso de teste.

```
<SOAP-ENV:Envelope
  xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <TravelItinerary xmlns="http://www.opentravel.org/OTA/2003/05">
            <ItineraryRef Type="14" ID="M839LW">
                <UniqueID>M839LWNN</UniqueID>
            </ItineraryRef >
            <CustomerInfos>
                <CustomerInfo RPH="01">
                    <Customer>
                        <PersonName>
                            <NamePrefix>Mr.</NamePrefix>
                            <GivenName>Robert</GivenName>
                            <MiddleName>Anthony</MiddleName>
                            <Surname>Jones</Surname>
                        </PersonName>
                        <Telephone PhoneNumber="2241630" AreaCityCode="302"
                                Extension="5574"/>
                        <Email>rajones@somewhere.org</Email>
                        <Address FormattedInd="true">
                            <StreetNmbr PO_Box="P.O. Box 77">
                                1452 S. 13th St. N.W.</StreetNmbr>
                            <CityName>Westfinster</CityName>
                            <PostalCode>90210</PostalCode>
                            <StateProv>NY</StateProv>
                            <CountryName>US</CountryName>
                        </Address>
                    </Customer>
                    <AgencyAcctNumber Type="5" ID="UOTWT0122321"/>
                </CustomerInfo>
            </CustomerInfos>
            <ItineraryInfo>
                <ReservationItems ChronoOrdered="true"/>
                <Ticketing TicketType="eTicket" eTicketNumber="30465876954325"
                        PlatingCarrier="UAL"/>
                <ItineraryPricing ItemRPH_List="01 02 03 04">
                    <Cost AmountAfterTax="745.00" CurrencyCode="USD"/>
                </ItineraryPricing>
                <SpecialRequestDetails>
                    <SpecialServiceRequests>
                        <SpecialServiceRequest FlightRefNumberRPHList="01"
                                TravelerRefNumberRPHList="01" SSRCode="VGML">
                            <Airline Code="UA">United Airlines</Airline>
                        </SpecialServiceRequest>
                    </SpecialServiceRequests>
                </SpecialRequestDetails>
            </ItineraryInfo>
            <TravelCost>
                <FormOfPayment>
                    <PaymentCard CardNumber="3151002645983754"
                            ExpireDate="1205" CardType="MC">
                        <CardHolderName>ROBERT A. JONES</CardHolderName>
                    </PaymentCard>
                </FormOfPayment>
                <CostTotals AmountAfterTax="1957.92" CurrencyCode="USD"/>
            </TravelCost>
            <UpdatedBy>
                <Access ID="U09932147"/>
            </UpdatedBy>
        </TravelItinerary>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Figura 23. Requisição do caso de teste hasNoReservations.

Nos experimentos de corrupção, a carga de trabalho consistiu no envio de 22 mensagens como a da Figura 23. 7 delas foram usadas no experimento com *strings* e 15 no experimento com inteiros. A carga de falhas consistiu em uma falha por mensagem enviada. Cada uma das 22 execuções foi feita três vezes. Em cada execução, um script foi carregado na WSInject e uma requisição foi enviada a partir da ferramenta soapUI, que ativou o processo de interceptação de mensagens e injeção de falhas.

Além dos experimentos acima, foram realizados outros, usando *EmptyingFault*, *MultiplicationFault* e *DelayFault*. Cada um foi executado 8 vezes – 2 por chamada ao processo BPEL e 2 por chamada a cada serviço parceiro (*AirlineReservationService*, *VehicleReservationService* e *HotelReservationService*). Também foi realizado um experimento de corrupção estrutural do XML usando *StringCorruptionFaults*, que inverteu as *tags* XML de abertura e fechamento. Este último experimento foi executado 10 vezes.

5.1.2. Resultados do experimento #1

A corrupção de inteiros produziu respostas da composição de serviços indicando sucesso na operação. Esse não foi o resultado esperado, pois dados inválidos foram enviados. Por exemplo, o valor -1 é um valor inválido para a data de validade de cartão de crédito. O sistema deveria ter validado esses dados e enviado uma mensagem de erro ao cliente. Isso mostra que a falha não foi detectada e provavelmente seria propagada ao banco de dados em um sistema real.

A corrupção de *strings* produziu o mesmo resultado, já que o uso de *strings* vazias, aleatórias e alfanuméricas não causou nenhuma mensagem de erro enviada ao cliente. Além disso, quando *strings* vazias eram enviadas, a conexão foi mantida aberta por um longo período de tempo, sem o cliente receber qualquer resposta. A injeção de caracteres não-imprimíveis gerou erros de *parsing* do XML, e o servidor retornou um erro informando essa ocorrência.

A injeção de *EmptyingFaults* nos serviços parceiros fez com que um erro HTTP 500 fosse enviado ao cliente, indicando a ocorrência de um erro interno do servidor e causando a desconexão do cliente. A injeção de *MultiplicationFaults* não afetou o sistema, que ignorou as repetições de elementos dentro da mensagem. Na injeção de *DelayFaults* de mais de 20 segundos nas requisições do TRS aos serviços parceiros (como na chamada ao *VehicleReservationService*), o TRS travou até o momento em que o GlassFish atingiu seu *timeout* de 2 minutos. Por outro lado, ao atrasar as respostas de confirmações de reservas (como aqueles retornadas pelo *VehicleReservationService* ao TRS), o GlassFish enviou mensagem indicando que houve um erro no envio da mensagem de cancelamento. Isso pode indicar a existência de um *bug* na implementação do cancelamento de reservas, o que seria um problema grave em um sistema real.

5.2. Experimento #2: Sistema real com clientes em Python

Este experimento consistiu em testar uma aplicação comercial baseada em Web Services, com o objetivo de validar o uso da WSInject em sistemas reais.

A aplicação testada é um sistema organizador de informações sobre ativos de software. Exemplos de ativos de software são código-fonte, bibliotecas e documentação. O sistema auxilia empresas desenvolvedoras de software a organizarem esses elementos para facilitar a atividade de desenvolvimento.

De forma simplificada, o sistema pode ser visto como um CRUD de ativos de software. CRUD significa *Create*, *Retrieve*, *Update*, *Delete* (criar, recuperar, atualizar, remover) e corresponde às operações básicas realizadas em um banco de dados ou em um sistema de informação. Para cada ativo, são armazenadas informações como nome, versão, data de criação e usuário responsável por sua inclusão no sistema.

O sistema é dividido em três camadas:

- **Banco de dados**: armazena todos os dados da aplicação;
- **Núcleo**: fornece uma API (*Application Programming Interface*) de acesso a esses dados;
- **Interface com usuário**: usa a API do núcleo para fornecer uma visão dos dados ao usuário.

O sistema oferece atualmente duas interfaces diferentes: sistema web e Web Service. A Figura 24 ilustra essa arquitetura.

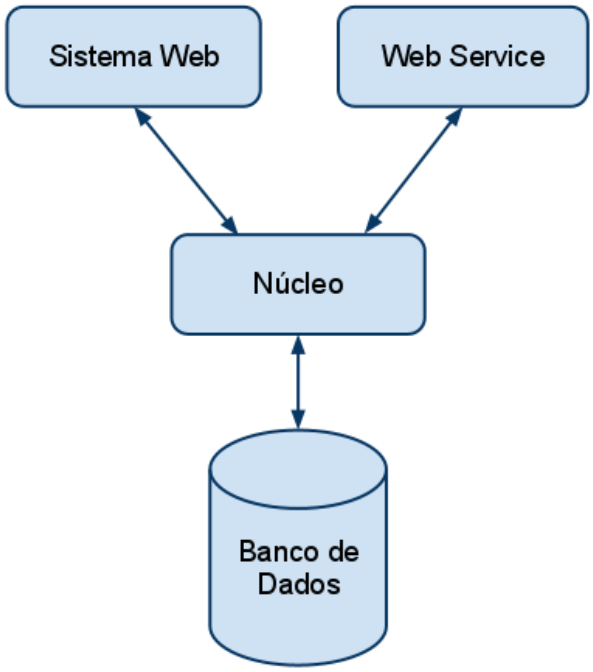

Figura 24. Arquitetura do sistema real em teste.

O uso mais comum da aplicação é feito pelo sistema web, acessível por meio de uma URL em um navegador. A exposição das funcionalidades em Web Service tem como finalidade permitir a integração da aplicação com outros sistemas de software. Integrações entre sistemas podem ser fontes de falhas [30], o que torna esta aplicação bastante interessante de ser testada.

Tanto o sistema web quanto o Web Service fornecem meios para se fazer CRUD dos dados, sendo que o sistema web oferece uma interface gráfica para essa finalidade e o Web Service oferece operações SOAP (adicionarAtivo, removerAtivo etc.). Ambos rodam sobre o *web container* Apache Tomcat/5.5.17[12]. Um *web container* é um sistema de software que permite implantar sistemas web e Web Services, da mesma forma que um servidor de aplicação (como o GlassFish do experimento anterior).

[12] http://projects.apache.org/projects/tomcat.html

O Web Service da aplicação exige autenticação do cliente usando WS-Security. Este padrão descreve, entre outras coisas, mecanismos para autenticação em Web Services, assinatura de mensagens SOAP e criptografia de mensagens SOAP. Toda requisição SOAP feita ao serviço em questão deve obrigatoriamente conter um cabeçalho informando usuário e senha, como no seguinte exemplo:

```
   <soapenv:Header>
      <wsse:Security soapenv:mustUnderstand="1"
xmlns:wsse="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-
1.0.xsd">
         <wsu:Timestamp xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd">
         </wsu:Timestamp>
         <wsse:UsernameToken wsu:Id="UsernameToken-1"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd">
            <wsse:Username>usuario</wsse:Username>
            <wsse:Password Type="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-username-token-
profile-1.0#PasswordText">senha</wsse:Password>
         </wsse:UsernameToken>
      </wsse:Security>
   </soapenv:Header>
```

Figura 25. Exemplo de cabeçalho WS-Security.

5.2.1. Clientes do serviço

Para ser testado, o serviço oferecido pela aplicação precisaria de um sistema que atuasse como seu cliente, enviando requisições SOAP – a carga de trabalho – e recebendo as respostas. Para essa finalidade, implementamos dois clientes, na linguagem Python, usando a biblioteca suds para efetuar chamadas a Web Services[13]. O conjunto composto pelo serviço em testes mais o sistema cliente constituiu o sistema em testes como um todo.

Esses clientes foram responsáveis pela geração da carga de trabalho, enquanto a WSInject foi responsável pela geração da carga de falhas. É importante ressaltar que

[13] https://fedorahosted.org/suds/

poderíamos ter usado alguma ferramenta como soapUI para geração da carga de trabalho. No entanto, preferimos desenvolver nossos próprios clientes para aproximarmos nossos experimentos de um teste de sistema completo baseado em Web Services. Nos clientes, foi implementado um mecanismo simples de tratamento de exceções para que eles pudessem executar seu código até o final, mesmo que o servidor disparasse alguma exceção. Os clientes são detalhados a seguir.

Cliente 1 – Dados do usuário

O cliente 1 faz chamadas às seguintes operações de apoio oferecidas pelo serviço: recuperarDadosDoUsuario e atualizarDadosDoUsuario. Esses dados consistem em informações de cadastro, como nome e endereço de email. Para preenchimento desses campos, foram gerados dados aleatórios. Esse cliente foi o primeiro usado nos testes por ser mais simples. Numa execução normal, sem injeção de falhas, o cliente 1 simplesmente executa uma sequência de atualizarDadosDoUsuario e recuperarDadosDoUsuario. A Figura 26 ilustra o fluxo de execução deste cliente.

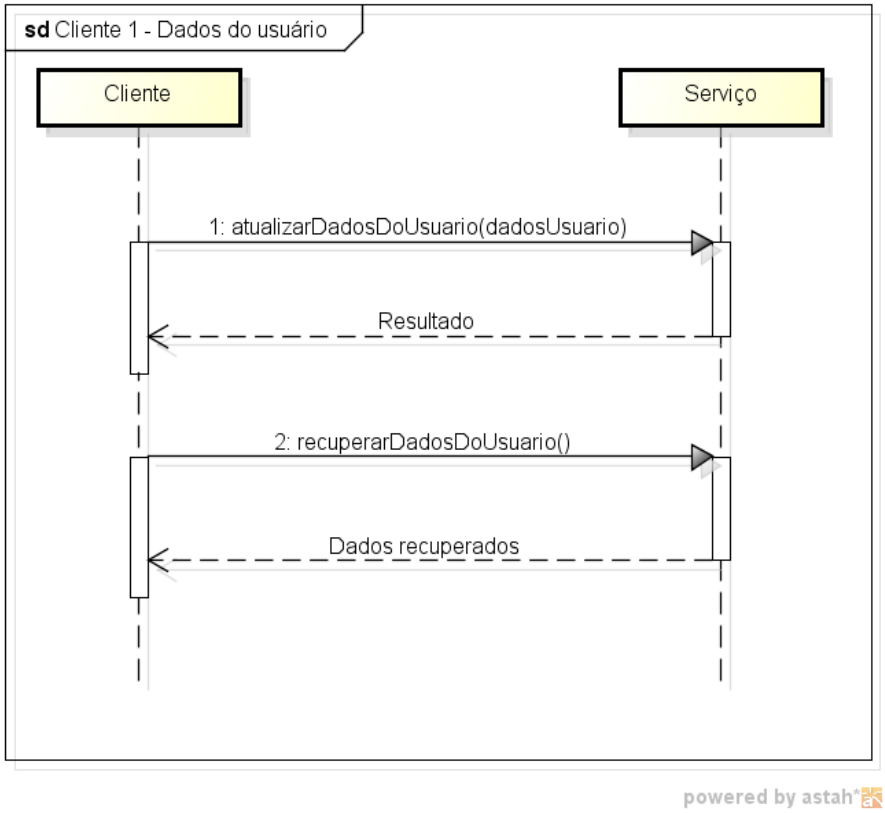

Figura 26. Cliente 1 do serviço.

A seguir, são apresentados os formatos esperados pelas requisições SOAP. A recuperação de dados consiste num simples pedido, sem nenhum dado. A atualização consiste no fornecimento de três dados de entrada, a serem gravados pela aplicação.

Formato da mensagem Recuperar Dados

```
<recuperarDados />
```

Formato da mensagem Atualizar Dados

```
<atualizarDados>
     <nome> nome </nome>
     <email> email </email>
     <senha> senha </senha>
</atualizarDados>
```

Cliente 2 – CRUD de ativos

O cliente 2 consistiu na execução das operações principais oferecidas pelo Web Service, as que realizam CRUD de ativos de software: criação de um novo ativo, recuperação do ativo, atualização do ativo e remoção do ativo. Os dados preenchidos em cada ativo também foram gerados aleatoriamente. A Figura 27 ilustra os fluxos de execução do cliente.

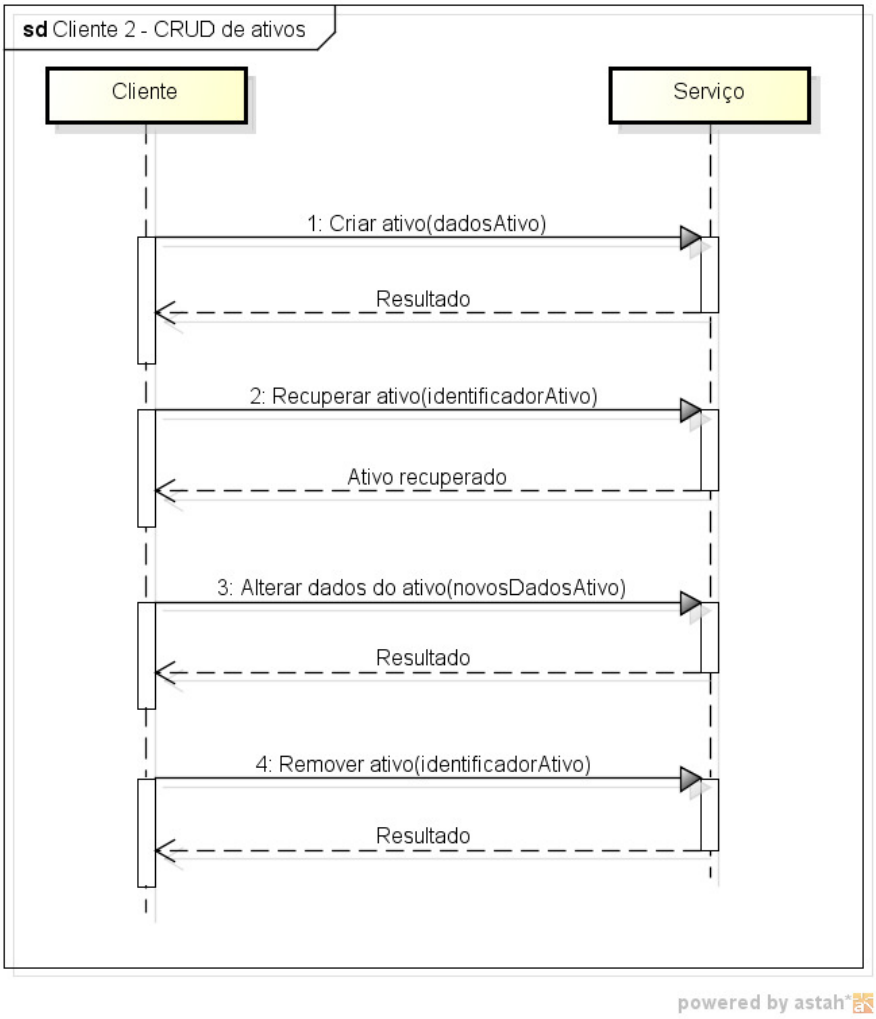

Figura 27. Cliente 2 do serviço.

Em sua execução normal, sem injeção de falhas, o cliente 2 executa uma sequência de criar um novo ativo no servidor, recuperá-lo, alterar seus dados, gravá-lo novamente no servidor e, finalmente, removê-lo do servidor.

São apresentados a seguir os conteúdos esperados pelas requisições SOAP. Na recuperação ou remoção, o conjunto de dados {nome, versao} identifica univocamente um ativo no sistema.

Formato das mensagens Criar / Alterar

```
<ativo>
     <nome> nome </nome>
     <versao> versao </versao>
     <descricao> descricao </descricao>
     <classificacao> classificacao </classificacao>
     <categoria> categoria1 </categoria>
     <categoria> categoria2 </categoria>
     <categoria> categoria3 </categoria>
     ... <!-- mais categorias -->
</ativo>
```

Formato das mensagens Recuperar e Remover

```
<ativo>
     <nome> nome </nome>
     <versao> versao </versao>
</ativo>
```

5.2.2. Descrição geral dos experimentos

Os experimentos foram realizados em duas etapas: pré-análise e execução completa. A pré-análise consistiu em avaliar brevemente os vários possíveis casos de teste, com o objetivo de selecionar os mais promissores a serem usados nos experimentos e descartar aqueles que produzem sempre a mesma resposta de validação do servidor, sendo desnecessários (pois não causam defeitos). A execução completa é aquela feita a partir do resultado da pré-análise, somente com os casos de teste selecionados. Seu objetivo é exercitar muitas vezes os mesmos fluxos de execução do sistema para obter um resultado mais confiável.

Mais especificamente, a pré-análise teve o intuito de identificar algumas situações em que o sistema sempre se comporta de maneira robusta, evitando, assim, a injeção de falhas desnecessárias. Outro objetivo da pré-análise é a seleção dos parâmetros mais interessantes da mensagem SOAP para serem alterados.

Após a pré-análise, foram feitas execuções completas. Estas consistiram na execução dos clientes de modo a serem injetadas falhas em 500 mensagens. No modelo Ballista, por exemplo, que possui 7 falhas para o tipo de dados *string*, cada caso de teste foi exercitado aproximadamente 70 vezes.

5.2.3. Arquitetura de testes

Os testes foram realizados usando-se a arquitetura de testes padrão da WSInject: cliente envia requisições ao serviço e recebe suas respostas, sendo as requisições e as repostas interceptadas pela ferramenta de forma transparente. As falhas foram injetadas somente na requisição para o serviço, não na resposta para o cliente. O motivo disso é que o objetivo era testar somente o serviço, não o cliente. A Figura 28 ilustra esse cenário.

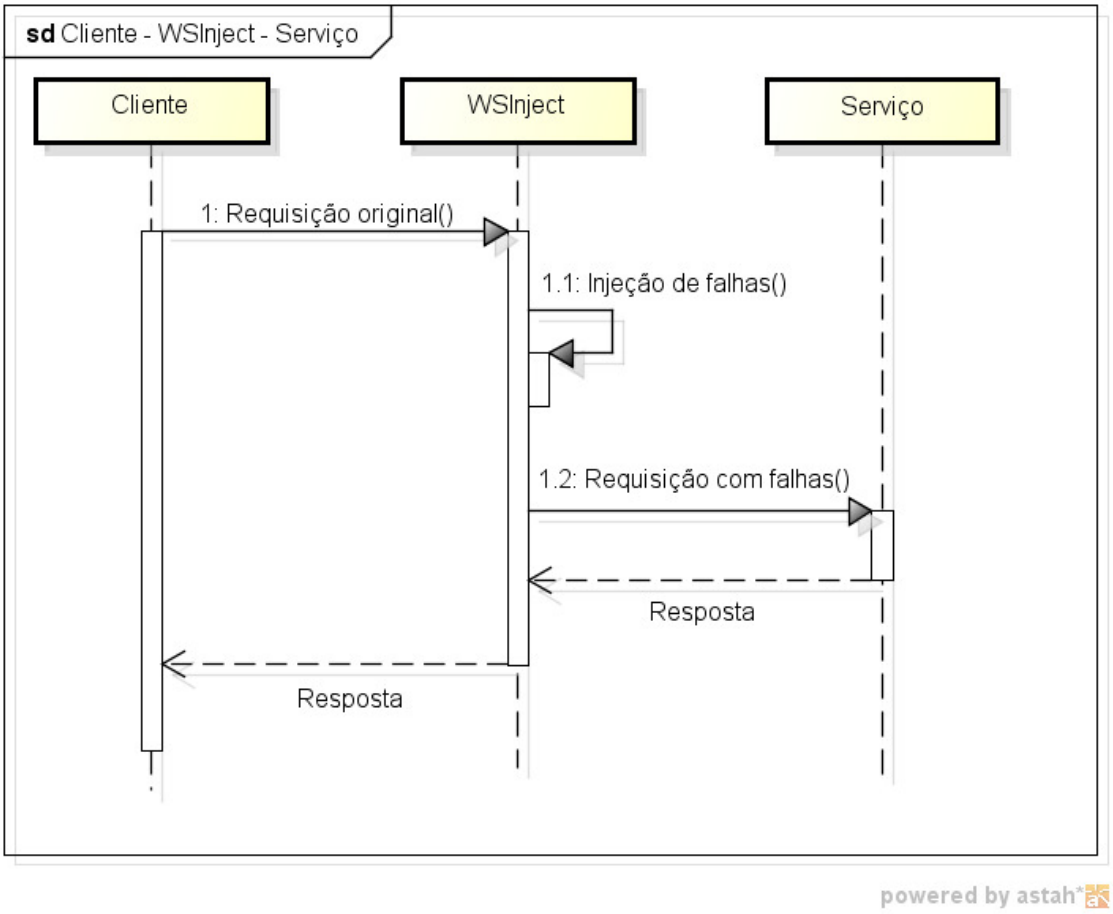

Figura 28. Arquitetura de testes do experimento #2.

5.2.4. Aplicação dos modelos

Cada um dos dois clientes foi testado com cada um dos três principais modelos de falhas descritos no capítulo **3**: *Fuzz*, Ballista e modelo de falhas XML.

Os três modelos utilizados foram implementados por meio de *scripts* da WSInject. Os modelos *Fuzz* e Ballista foram implementados diretamente por meio de geradores dinâmicos, e foram incorporados à biblioteca de modelos de falhas da ferramenta. O modelo XML foi implementado usando-se uma combinação de *scripts* estáticos. Futuramente, ele também poderá ser incorporado à biblioteca. Para mais detalhes, consulte a seção **4.4**.

5.2.5. Pré-análise – modelos *Fuzz* e Ballista

A primeira pré-análise foi realizada com o cliente 1 e o modelo *Fuzz*. Dentre as operações do Web Service chamadas pelo cliente 1 (recuperarDados e atualizarDados), recuperarDados envia apenas um elemento XML vazio. Dessa forma, não seria possível alterar nenhum parâmetro dessa mensagem. recuperarDados, por outro lado, oferece três parâmetros: nome, email e senha. A alteração de nome e email pode ser feita sem nenhum problema. No entanto, a senha do usuário é a mesma que deve ser informada no cabeçalho WS-Security. Assim, se ela for modificada, cria-se o inconveniente de ser necessário modificar o cabeçalho WS-Security também. Dessa forma, decidimos por trabalhar somente com os parâmetros nome e email, ambos do tipo *string*. Executamos casos de teste que alteram somente nome, somente email e ambos ao mesmo tempo.

O modelo *Fuzz* foi implementado de forma a gerar *strings* totalmente aleatórias, contendo caracteres com código ASCII de 0 a 127, com tamanho variando aleatoriamente entre 0 e 500. Alterando-se somente o parâmetro nome, foram feitas 21 execuções do cliente, e as respostas enviadas pelo serviço estão mostradas na Tabela F.

Tabela F. Pré-análise: alteração de nome, com modelo *Fuzz* completo.

Resposta obtida	Número de requisições
[com.ctc.wstx.exc.WstxLazyException] Illegal character entity: expansion character not a valid XML character	17
Problems creating SAAJ object model	3

[com.ctc.wstx.exc.WstxLazyException] Invalid character reference: null character not allowed in XML content	1
TOTAL	**21**

Todas as mensagens SOAP foram rejeitadas pelo servidor. Pela análise das mensagens de resposta e do conteúdo das requisições após a injeção de falhas, notamos que esses resultados foram obtidos porque a requisição continha caracteres de controle ASCII. O servidor informou *Problems creating SAAJ object model* quando este caractere estava na primeira posição; *Illegal character entity* quando o caractere estava em uma posição diferente da primeira; e *Null character* quando algum desses caracteres era o ASCII 0. A especificação do padrão XML recomenda que não sejam usados esses valores [38]. Percebemos que respostas diferentes podem indicar o mesmo problema causador, o que torna uma classificação automática dos resultados bastante difícil.

Concluiu-se que os caracteres de controle são barrados pela validação de dados no servidor. A geração de mensagens contendo esses caracteres produziria pouco ou nenhum efeito. Assim, decidimos remover esses caracteres do domínio de caracteres aleatórios concatenados para gerar as *strings* de saída. Após essa alteração, o experimento foi repetido e o resultado está descrito na tabela seguinte.

Tabela G. Pré-análise: modelo *Fuzz* sem caracteres de controle.

Resposta obtida	Número de requisições
OK	6
PersistenceException: ORA-12899: value too large for column "NOME_DO_BANCO"."ATIVO"."NOME" (maximum: 250)	15
TOTAL	**21**

Notamos que algumas requisições foram atendidas corretamente, mas a maioria produziu a resposta “valor muito grande para a coluna do banco de dados”. Isso ocorreu porque o gerador utilizado gerava *strings* com tamanho que poderia ultrapassar os 250 caracteres, e esse era o limite que o banco de dados especificava para o dado esperado.

Percebemos que *strings* com mais de 250 caracteres também são corretamente barradas e validadas pelo servidor. Destaca-se, ainda, que a aplicação informou ao usuário, além do número máximo de caracteres permitido, o **Sistema Gerenciador de Bancos de Dados (SGBD)** utilizado – Oracle, identificável por meio do código de erro –, o **nome do banco de dados**, o **nome da tabela** e o **nome da coluna**. Essas informações são internas do sistema e não deveriam chegar a conhecimento de usuário, representando uma pequena, mas existente, brecha de segurança.

Após o modelo de falhas ser restringido a gerar *strings* com no máximo 250 caracteres, o resultado obtido foi o seguinte:

Tabela H. Pré-análise: modelo *Fuzz* sem caracteres de controle e com 250 caracteres.

Resposta obtida	Número de requisições
OK	20
PersistenceException: ORA-01407: cannot update ("NOME_DO_BANCO"."ATIVO"."NOME") to NULL	1
TOTAL	**21**

Encontramos mais uma situação em que o servidor valida corretamente o dado de entrada, rejeitando uma *string* nula para preencher o campo nome. Novamente, foi informado SGBD, banco de dados, tabela e coluna. Essa validação, ao contrário das anteriores, não interfere negativamente nos testes, pois a ocorrência de *strings* nulas mostrou-se bastante rara no decorrer dos experimentos.

Os mesmos testes foram repetidos com a modificação somente do parâmetro email e com a modificação de nome e email simultaneamente. O resultado obtido foi equivalentes ao obtido com a modificação somente do parâmetro nome: somente conseguimos exercitar adequadamente as operações do serviço quando implementamos as restrições no modelo *Fuzz*.

A pré-análise foi realizada também com o modelo Ballista. Dado que ele possui uma natureza muito similar à do modelo *Fuzz* (alteração de valores de parâmetros) e que as operações do Web Service exercitadas foram as mesmas, obtivemos novamente um resultado equivalente, indicando que *strings* contendo caracteres de controle ou com mais

de 250 caracteres são barradas pela validação no servidor. Conclui-se que as validações realizadas são iguais, independentemente do modelo de falhas utilizado.

Uma pré-análise bastante similar foi realizada com o cliente 2. Os parâmetros disponíveis para injeção são nome, versão, descrição, classificação e categoria. Injeções nesses parâmetros indicaram que a alteração de nome, versão e descrição poderia ser feita livremente, enquanto no parâmetro classificação não era permitido escrever caracteres como espaço, vírgulas, pontos etc., e que o parâmetro categoria precisaria conter um dentre alguns valores pré-definidos pelo servidor. Dessa forma, para evitar que muitos casos de teste fossem rejeitados por essas validações no servidor, optamos pela injeção de falhas somente nas combinações dos parâmetros nome, versão e descrição, ou seja: {nome}, {versão}, {descrição}, {nome, versão}, {nome, descrição}, {versão, descrição} e {nome, versão, descrição}.

Primeiramente, injetamos falhas do modelo *Fuzz* e do modelo Ballista sem implementar as restrições de limite no domínio de caracteres nem no tamanho da *string*. Obtivemos os mesmos resultados do cliente 1, com o servidor validando esses dados e rejeitando valores nessas condições.

O resultado da pré-análise é, portanto, que o modelo *Fuzz*, para ser bem utilizado nessas operações do Web Service, deve ser limitado a 250 caracteres e limitado à geração de caracteres ASCII que não sejam de controle.

O modelo *Fuzz* é totalmente livre, enquanto o modelo Ballista possui falhas específicas para geração de caracteres de controle e para *strings* muito grandes. Para o modelo Ballista, decidimos limitar caracteres de controle e tamanho de *strings* somente naquelas falhas que não especificavam nada com relação a essas características. Assim, a falha OVERFLOW continuou gerando *strings* grandes e a falha NON_PRINTABLE continuou gerando caracteres de controle, mas as outras não.

5.2.6. Execução do modelo *Fuzz*

Após a pré-análise, partimos para a execução completa do experimento nos dois clientes, usando o modelo *Fuzz*. No cliente 1, foram enviadas mensagens para as operações recuperarDados e atualizarDados, sendo injetadas falhas somente nesta última. As alterações foram feitas em três etapas: somente no parâmetro nome; somente no parâmetro

email; e em ambos. No cliente 2, foram enviadas mensagens para as operações de salvar ativo, recuperá-lo, alterá-lo e removê-lo. A injeção de falhas foi feita somente na mensagem de alteração de ativo. Cada etapa consistiu em uma combinação dos parâmetros nome, versão e descrição. A Tabela I mostra o resultado obtido com as injeções de falhas nos dois clientes.

Tabela I. Resultado da execução do modelo *Fuzz*.

Resposta obtida	Número de requisições
OK	4973
PersistenceException: ORA-01407: cannot update ("NOME_DO_BANCO"."ATIVO"."COLUNA") to NULL	27
TOTAL	**5000**

No cliente 1, essas respostas foram obtidas para as requisições à operação atualizarDados. No cliente 2, essas respostas foram obtidas para as requisições à operação alterarAtivo. As respostas do servidor para todas as mensagens de criar, recuperar e remover ativos do cliente 2 foram "*OK"*. Analisando o fluxo de execução do cliente 2, percebemos que essa resposta "*OK*" não está correta. O cliente tentou remover um ativo usando nome e versão informados por ele no momento da alteração. Esses dados, no entanto, são **diferentes daqueles recebidos previamente pelo servidor**, pois na alteração houve injeção de falhas e na exclusão, não. Dessa forma, nenhum ativo foi de fato removido, e o serviço informou incorretamente que a operação de remoção foi executada com sucesso.

Resultados

Nenhum defeito de robustez foi encontrado. As operações testadas mostraram-se robustas nos testes realizados com o modelo *Fuzz*. Porém, um defeito funcional foi encontrado com a execução do cliente 2: a operação removerAtivo informou "sucesso" mesmo quando não feita nenhuma remoção.

O modelo *Fuzz* não se mostrou tão adequado para uso em XML, pois baseia-se fortemente na presença de caracteres especiais, como caracteres de controle, que muitas vezes não são suportados em mensagens SOAP. O serviço chamado implementou

corretamente a especificação do padrão e rejeitou a presença desses caracteres. Para que o experimento pudesse continuar, foi necessário eliminar os caracteres de controle do domínio dos valores a serem gerados, e isso prejudicou o potencial do modelo de causar defeitos.

Vale ressaltar que a aplicação enviou como resposta algumas mensagens de erro contendo nome de SGBD, nome do banco de dados, nome da tabela e nome da coluna que estava sendo alterada. Essas informações são úteis para os desenvolvedores na atividade de depuração, mas nunca devem se tornar visíveis ao usuário, apresentando-se como uma possível brecha de segurança.

5.2.7. Execução do modelo Ballista

Este modelo é ligeiramente mais complexo de ser implementado do que o *Fuzz*, pois suas falhas possuem características mais específicas. Outra diferença é que algumas falhas do modelo Ballista dependem do valor original do parâmetro para serem geradas, ao contrário do *Fuzz*, que produz valores totalmente aleatórios.

Os experimentos com o modelo Ballista seguiram a mesma linha do modelo *Fuzz*: experimentos com alteração isolada de cada um dos parâmetros e experimentos com alterações de mais de um parâmetro simultaneamente. O resultado geral obtido é mostrado na Tabela J.

Tabela J. Resultado da execução do modelo Ballista.

Resposta obtida	Número de requisições
OK	1056
PersistenceException: ORA-01407: cannot update ("NOME_DO_BANCO"."ATIVO"."COLUNA") to NULL	1241
Illegal character entity: expansion character not a valid XML character	1576
Problems creating SAAJ object model	1052
[com.ctc.wstx.exc.WstxLazyException] Invalid character reference: null character not allowed in XML content.	58
Versão de ativo duplicada	17
TOTAL	**5000**

No caso do cliente 2, assim como no experimento com o modelo *Fuzz*, novamente o serviço respondeu "*OK"* para a solicitação de remoção quando os dados informados eram inexistentes no banco de dados do servidor, caracterizando-se, assim, como um defeito funcional.

Em alguns casos, o servidor respondeu que a versão do ativo estava duplicada, indicando que foi feita uma tentativa de inserir um ativo com mesmo nome e mesma versão. Isso não é permitido pela aplicação e foi validado adequadamente.

Resultados

O modelo Ballista, assim como o *Fuzz*, baseia-se na presença de caracteres especiais, embora em menor escala. Ao contrário do que foi feito no experimento com o modelo anterior, dessa vez não eliminamos totalmente os caracteres de controle ou as *strings* muito grandes. Os casos de teste que tinham a especificação de possuírem essas características mantiveram-se inalterados.

Assim como com o modelo *Fuzz*, o Ballista também não encontrou nenhum defeito de robustez na aplicação, mas encontrou um defeito funcional com a execução do cliente 2 (remoção de ativo indicando "*OK*" quando na verdade não removeu) e a brecha de segurança (informações sobre o banco de dados chegando ao usuário). Além disso, novamente notam-se várias respostas diferentes para indicar o mesmo problema: *Illegal character entity*, *Problems creating SAAJ object model* e *Invalid character reference*. Todas essas respostas indicam a existência de um parâmetro com caracteres inválidos.

5.2.8. Pré-análise do modelo XML

Diferentemente dos anteriores, este modelo baseia-se na alteração da **estrutura** do documento XML, não na alteração ou inserção de dados no **conteúdo** da mensagem. O modelo de falhas XML foi implementado por meio de falhas estáticas oferecidas pela WSInject (*MultiplicationFault*, *CoerciveParsingFault* e StringCorruptionFault).

Os modelos *Fuzzing* e Ballista trabalhavam sorteando o tamanho das *strings* produzidas e também seu conteúdo. O modelo XML não possui qualquer variação ou sorteio, produzindo sempre a mesma mensagem para cada uma de suas falhas. Na pré-análise, cada uma das quatro falhas do modelo foi executada 100 vezes em cada cliente, tendo produzido sempre o mesmo resultado.

Injeção de elemento

Consiste em mover um elemento B que seja irmão do elemento A para tornar-se filho do elemento A. Três casos de teste por cliente foram executados. No caso do cliente 1, foi feita uma alteração de:

```
<nome> nome </nome>
<email> email </email>
```

Para cada uma das seguintes variações:

```
<nome>
    nome
    <email> email </email>
</nome>


<nome>
    <email> email </email>
</nome>


<nome>
    <email> email </email>
    nome
</nome>
```

Para o cliente 2, foi feita uma alteração similar entre os parâmetros nome e versão. O resultado obtido em todas as execuções foi o mesmo: *Unmarshalling Error: unexpected element*. Essa resposta indica que o servidor validou o XML recebido de acordo com a estrutura esperada. Assim, a falha foi tolerada pelo sistema.

Elementos muito profundos

Consiste em gerar um XML contendo uma árvore de elementos muito profunda, similar a:

```
<x>
     <x>
          <x>
               <x>
```

```
                    ...
                    ...
                </x>
            </x>
        </x>
    </x>
```

Essa falha procura causar defeitos como falta de memória no servidor, devido a uma possível alocação de uma estrutura de dados muito grande.

Para a árvores XML com profundidade até aproximadamente 1390 elementos, o serviço mostrou-se robusto, emitindo a mensagem: *Unmarshalling Error: unexpected element*. A partir de um valor próximo de 1400, o servidor passou a retornar uma página HTML de erro interno do servidor, indicando a ocorrência de StackOverflowError (estouro de pilha).

Essa página HTML foi uma resposta inusitada, pois se esperava que a resposta a uma requisição SOAP também estivesse no formato SOAP. Isso mostra que o defeito apresentado pelo servidor ultrapassou a camada do protocolo SOAP e alcançou a camada HTTP.

Além do estouro de pilha no servidor, qualquer requisição feita ao serviço logo após a ocorrência do defeito era respondida com uma mensagem SOAP contendo a seguinte informação: "*ERRO: Uma requisição já está em andamento e foi solicitada a criação de uma nova. Favor finalizar a primeira antes"*. Isso mostra que a falha causou uma mudança de estado interno do sistema, que perdurou até que uma nova requisição fosse feita. Após essa resposta ser dada à primeira requisição subseqüente, as próximas foram respondidas corretamente.

Em resumo, a falha de elementos muito profundos, quando injetada com 1400 ou mais elementos no serviço testado, causou um defeito de robustez todas as vezes em que foi injetada.

Repetição de muitos elementos

Esta falha consiste na replicação de elementos em um documento XML. Exemplo:

```
<ativo>
     <nome> nome </nome>
     <nome> nome </nome>
     <nome> nome </nome>
     <nome> nome </nome>
     <nome> nome </nome>
     <nome> nome </nome>
     <nome> nome </nome>
     ...
     <versao> versao </versao>
</ativo>
```

Para uma injeção de 5.000 a 50.000 elementos, nenhum defeito foi encontrado. O servidor simplesmente retornou OK e ignorou a presença dos elementos repetidos. A única diferença observável foi a demora no tráfego dos dados pela rede. Ao aumentar o número de repetições para 500.000 elementos, a demora foi tão grande que o cliente apresentou um *timeout*. O servidor, no entanto, continuou em comunicação com a WSInject, e retornou OK, indicando sucesso na execução da operação, mesmo com a injeção de 500.000 elementos repetidos.

Fomos impedidos de aumentar ainda mais o número de elementos por causa da ocorrência de *timeouts*. A ferramenta soapUI possui um recurso de usar compressão de dados na mensagem enviada ao servidor. Isso reduz drasticamente o tempo de envio da mensagem e viabiliza a execução de cenários como esse. É um recurso que poderá ser implementado futuramente na WSInject.

Troca de caracteres válidos por inválidos

Essa falha consiste em alterar alguns caracteres específicos do padrão XML por outros caracteres. Tal alteração tem o objetivo de avaliar o que acontece quando o *parser* XML do servidor depara-se com um documento mal-formado. A Tabela K resume os casos de teste executados e as respostas obtidas do servidor. As colunas “De” e “Para” indicam a mudança que foi feita.

Tabela K. Pré-análise do modelo XML: caracteres inválidos.

De	Para	Resposta obtida
<el>	#el>	Error reading XMLStreamReader
<el>	<el#	Couldn't parse stream
</	<	Fault occurred while processing
"	""	Couldn't parse stream
<el>	<>	Problems creating SAAJ object model

Todas essas alterações tornaram a mensagem XML igualmente mal-formada e inválida. Apesar disso, para 5 tipos de alteração, foram emitidas 4 mensagens de erro diferentes.

Resultados

Todas as falhas do modelo XML apresentaram resultados idênticos nos dois clientes. Um defeito de robustez foi encontrado, revelando falhas na implementação do *parser* XML. Uma análise da página HTML de erro recebida indicou que o componente responsável por essa atividade de *parsing* foi o Apache Xerces[14].

O resultado da execução da falha *troca de caracteres válidos por inválidos* reforça a observação de que um mesmo problema pode causar diversas mensagens de erro diferentes, ficando a cargo do testador classificar essas mensagens adequadamente.

5.2.9. Execução do modelo de falhas XML

Durante a pré-análise, a falha *elementos muito profundos* foi a única que causou defeitos no serviço. Além desta, a falha *troca de caracteres válidos por inválidos* chamou a atenção por ter produzido uma variedade grande de respostas para falhas equivalentes. Optamos, assim, por incluir somente essas duas falhas na execução completa.

A injeção de elementos com nível de profundidade a partir de 1400, para qualquer parâmetro escolhido e para ambos os clientes, produziu o resultado mostrado obtido na pré-análise: a primeira mensagem contendo elementos muito profundos causou um estouro de pilha no servidor. A próxima mensagem recebeu a resposta de que não poderia ser atendida porque já havia uma solicitação pendente. A terceira mensagem causou novamente o estouro de pilha, e assim por diante, em ciclo. A Tabela L ilustra esses resultados.

[14] http://xerces.apache.org/

Tabela L. Modelo XML: injeção de elementos muito profundos.

Resposta obtida	Número de requisições
java.lang.StackOverflowError	250
Uma requisição já está em andamento e foi solicitada a criação de uma nova. Favor finalizar a primeira antes.	250
TOTAL	**500**

O resultado da injeção das alterações de caracteres válidos para inválidos é mostrado na Tabela M.

Tabela M. Modelo XML: injeção de caracteres inválidos.

De	Para	Resposta obtida	Número de requisições
<el>	#el>	Error reading XMLStreamReader	100
<el>	<el#	Couldn't parse stream	100
</	<	Fault occurred while processing	100
"	""	Couldn't parse stream	100
<el>	<>	Problems creating SAAJ object model	100
TOTAL			**500**

Resultados

A execução completa das falhas selecionadas do modelo XML produziu resultado idêntico ao observado anteriormente, na pré-análise. Atribuímos essa igualdade de resultados ao fato de que, no modelo XML, as falhas são muito mais definidas do que nos outros modelos. O *Fuzz*, por exemplo, determina apenas que uma mensagem deve ser gerada aleatoriamente. O Ballista determina algumas alterações a serem feitas na mensagem, sendo algumas delas mais definidas e outras mais livres, como a inclusão de caracteres aleatórios.

5.2.10. Resultados do experimento #2

O modelo de falhas XML mostrou-se o mais adequado, tendo encontrado um defeito de robustez (estouro de pilha no servidor). Os outros modelos testados encontraram uma pequena brecha de segurança e um defeito funcional. Essa melhor atuação do primeiro

modelo deve-se ao fato de ele estar mais próximo de Web Services do que os outros, já que é baseado no padrão XML.

O defeito de robustez encontrado mostra que casos de teste simples podem ser capazes de causar efeitos indesejáveis, mesmo em software bastante utilizado, como é o caso do Apache Xerces. A possibilidade de causar esse defeito poderia ser explorada por atacantes com o intuito de causar situações de negação de serviço (*DoS – Denial of Service*) em sistemas em produção, visto que o serviço torna-se inoperante na próxima requisição.

Ainda sobre esse defeito, é importante ressaltar que a falha causadora deste é plausível de ocorrer em uma situação real. O Web Service testado tem o intuito de permitir a integração com outros sistemas, e não há como conhecer todos os sistemas que podem vir a ser integrados a ele. Estes podem apresentar comportamentos inesperados e produzir uma requisição similar àquela produzida no experimento, causando o mesmo defeito. A negação de serviço observada após a ocorrência do defeito pode ser um problema sério caso o sistema precise oferecer alta disponibilidade.

Apesar do defeito de robustez mencionado, os testes indicaram, no geral, um bom nível de robustez do serviço. Ele comportou-se de forma robusta em todas as outras situações, principalmente na aplicação dos modelos *Fuzz* e Ballista. É importante destacar que isso não significa necessariamente a ausência de falhas no serviço. Tampouco significa que tais modelos não sejam capazes de causar defeitos em sistemas baseados em Web Services, e o trabalho de Laranjeiro *et al*. mostra bons resultados com o modelo Ballista [14]. O resultado obtido deve ser visto apenas como **indício** de que o sistema é robusto em boa parte das situações. Para sistemas que devam obrigatoriamente oferecer alta robustez e disponibilidade, uma medida mais precisa dessas características poderia ser obtida por meio de técnicas mais apropriadas, conforme proposto no trabalho de ReSIST Team [29].

5.3. Experimento #3: WSInject com wsrbench

Este experimento teve como objetivo aplicar ao sistema do experimento #2 uma ferramenta já existente, a wsrbench [14]. Tentativas de aplicação direta da wsrbench no serviço não foram bem-sucedidas, pois o serviço requer o uso de WS-Security e a wsrbench não oferece esse recurso. Não tínhamos acesso à implementação da wsrbench e ela não oferece facilidades para extensões, portanto não havia a possibilidade de realizarmos

qualquer modificação diretamente nela. Apesar disso, com o uso da funcionalidade de injeção de falhas oferecida pela WSInject, fomos capazes de realizar uma integração entre as duas ferramentas, conforme será explicado nesta seção.

A wsrbench funciona em duas etapas:

- Primeira etapa:
  - Usuário entra com a URI do WSDL (arquivo que descreve o serviço a ser testado). Exemplo: http://example.com/service?WSDL.
  - wsrbench baixa o arquivo WSDL.
  - wsrbench processa o arquivo WSDL e procura dentro dele a URI do serviço a ser testado. Normalmente, o serviço localiza-se na mesma URL (parte que antecede o '?'). Exemplo: http://example.com/service.
- Segunda etapa:
  - wsrbench inicia os testes, enviando requisições SOAP para o serviço na URL previamente descoberta (http://example.com/service).
  - wsrbench analisa as respostas recebidas.

A URI do Web Service fica localizada dentro do elemento `<soap:address>` do WSDL. Exemplo:

```
<soap:address location="http://example.com/service" />
```

A Figura 29 ilustra uma tentativa de aplicação direta da wsrbench ao serviço.

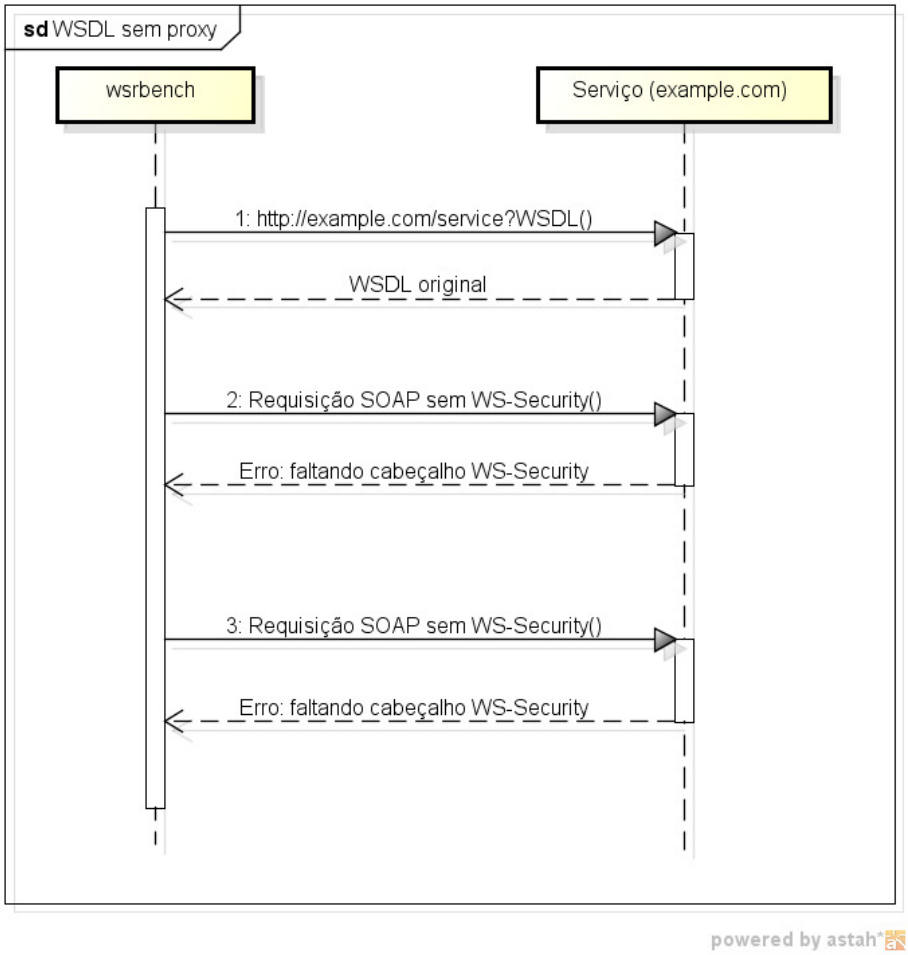

Figura 29. wsrbench aplicada ao serviço em teste sem uso de proxy

Notamos que a falta do cabeçalho WS-Security causa sempre a mesma mensagem de erro, de forma que a wsrbench não é bem-sucedida na tentativa de executar seus testes. Precisamos, portanto, inserir o cabeçalho na mensagem de alguma forma.

A WSInject é capaz de alterar o conteúdo de mensagens SOAP, podendo, portanto, ser usada para realizar essa tarefa e possibilitar a aplicação da wsrbench ao serviço. No entanto, uma restrição da WSInject é que o cliente precisaria ser configurado para se conectar por meio de um proxy. Isso não seria possível no caso da wsrbench, já que não tínhamos acesso a suas configurações.

Gostaríamos então de remover essa restrição, fazendo com que a WSInject funcione como um **proxy transparente**. Isso eliminaria a necessidade de se configurar a wsrbench. Conseguimos atingir esse objetivo efetuando uma simples alteração no elemento `<soap:address>` do WSDL:

```
<soap:address location="http://201.20.80.1:8081/service" />
```

(sendo 201.20.80.1:8081 o endereço onde está localizada a WSInject)

Essa alteração faz com que o cliente se conecte à WSInject em vez de conectar-se diretamente ao serviço. Para realizarmos essa alteração no WSDL, usamos uma funcionalidade já disponível na WSInject, a corrupção de mensagens. A Figura 30 ilustra esse cenário.

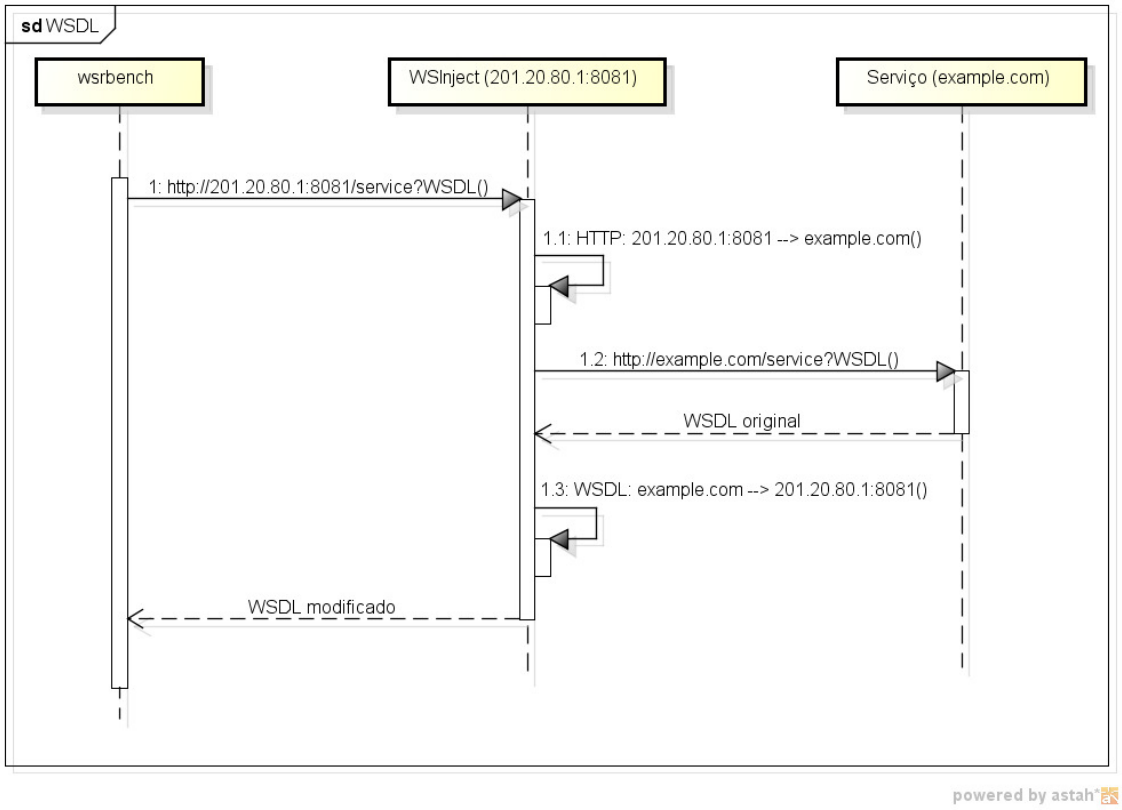

Figura 30. Primeira etapa da wsrbench com a WSInject – WSDL

Após o WSDL ser entregue à wsrbench, temos o seguinte cenário:

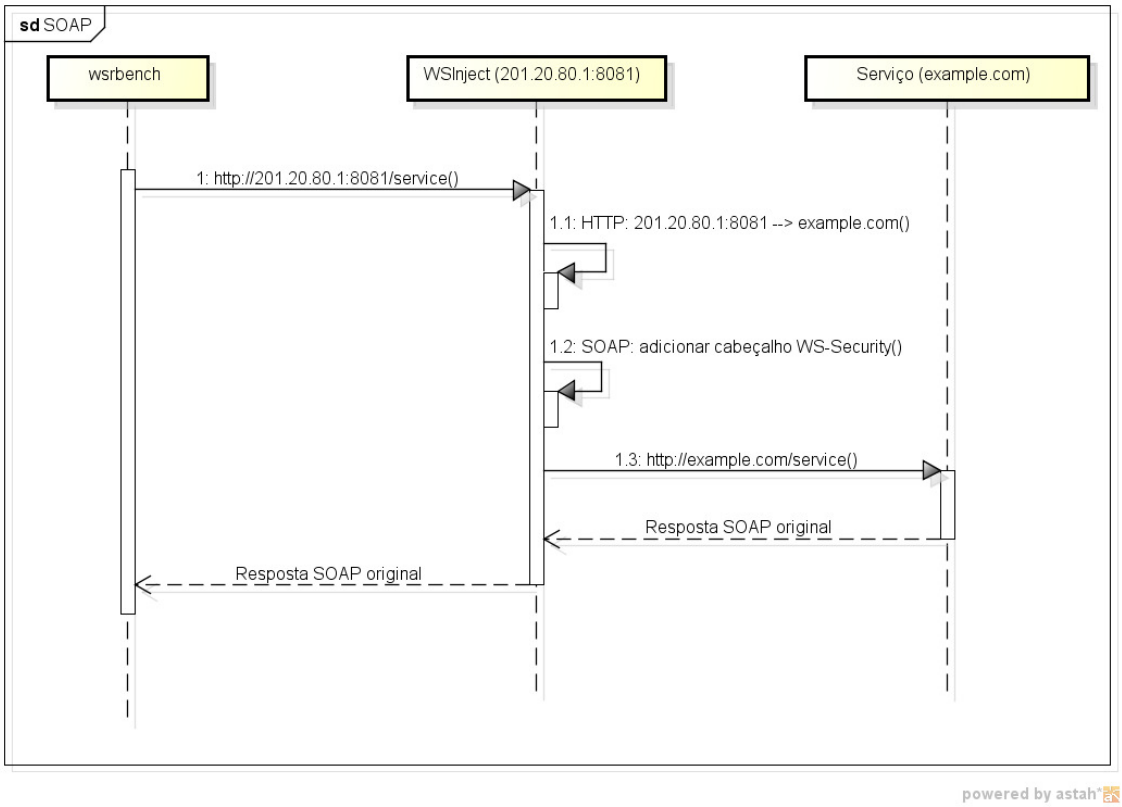

Figura 31. Segunda etapa da wsrbench com a WSInject – SOAP

As atividades 1.1 de ambas as figuras indicam alterações nas requisições HTTP. A WSInject recebe uma requisição como se fosse direcionada para ela própria, precisando redirecioná-la para o serviço real. Atualmente, a ferramenta não possibilita que o usuário defina suas próprias alterações em mensagens HTTP. Dessa forma, para a realização deste experimento, foi feita uma simples prova de conceito, com implementação diretamente no código (*hard-coded*).

Este experimento subdividiu-se em dois. No primeiro, apenas adaptamos as requisições da WSInject para funcionarem no serviço em testes. No segundo, além disso, fizemos mais uma alteração na mensagem com o intuito de melhorar o resultado obtido. Ambos estão detalhados nas seções a seguir.

5.3.1. Simples inclusão do cabeçalho WS-Security pela WSInject

A presença da WSInject na arquitetura de testes permitiu à wsrbench atuar com sucesso na injeção de falhas no serviço. A wsrbench realizou **336** requisições SOAP, das

quais 335 foram respondidas de forma robusta e apenas uma apresentou problemas, disparando a exceção NullPointerException. A maioria das respostas indicou problemas na transformação do valor passado por parâmetro em um valor válido – um comportamento correto. Outras respostas consistiram em um conjunto de dados vazio, indicando que uma busca por objetos armazenados (ativos, relacionamentos etc.) foi realizada e nenhum objeto correspondeu aos valores passados como parâmetro – um comportamento igualmente correto. A Tabela N resume essas informações.

Tabela N. Resultados obtidos com a simples inclusão do cabeçalho WS-Security.

Resposta obtida	Número de requisições
NullPointerException	1
Conjunto de dados vazio	40
Unmarshalling Error: Not a number: ?	67
Unmarshalling Error: Not a number: *(outros)*	164
Problems creating SAAJ object model	64
TOTAL	**336**

Como visto na tabela, 67 requisições (20% do total) causaram erros de transformação de XML em dados Java (*Unmarshalling Error*) devido à presença do caractere '?'. Observando-se o conteúdo das requisições, encontramos diversas vezes o seguinte padrão:

```
<ativo>
     <identificador>?</identificador>
     <nome>?</nome>
     <versao>?</versao >
</ativo>
```

O serviço espera como entrada valores numéricos para o parâmetro identificador. O caractere "ponto de interrogação" não pode ser convertido em valor numérico. Além disso, nome e versão possivelmente também não aceitem esse caractere. Esses casos de teste não tiveram sucesso, pois o serviço identificou-os como entradas inválidas e não prosseguiu com o processamento. Se pudéssemos remover esses elementos da requisição, poderíamos

aumentar a chance de esses casos de teste causarem defeitos. O experimento seguinte consistiu exatamente nisso.

5.3.2. Remoção dos elementos contendo o caractere '?'

Neste caso de teste, além da inclusão do cabeçalho WS-Security, removemos todas as ocorrências dos três elementos citados anteriormente, usando uma simples corrupção de *strings.* O novo experimento produziu os seguintes resultados:

Tabela O: Resultados obtidos com a injeção de uma falha extra pela WSInject.

Resposta obtida	Número de requisições
NullPointerException	18
Conjunto de dados vazio	56
Fault occurred while processing.	17
Cannot find relationship type with name: *(...)*	16
FAILURE – ASSET NOT FOUND	1
Unmarshalling Error: Not a number: *(...)*	164
Problems creating SAAJ object model	64
TOTAL	**336**

Notamos um aumento bastante significativo no número de defeitos que conseguimos produzir. Para 336 requisições SOAP enviadas, verificamos 18 ocorrências de NullPointerException (5,4%), contra 1 ocorrência (0,3%) no teste anterior. O número de respostas contendo um conjunto de dados vazio – uma resposta correta – passou de 40 para 56. Surgiram ainda novas mensagens indicando erros no processamento dos dados passados como entrada: "Fault occurred while processing", "Cannot find relationship type" e "FAILURE – ASSET NOT FOUND".

5.3.3. Resultados do experimento #3

A WSInject integrou-se com sucesso à wsrbench, permitindo adaptar as requisições de uma ferramenta já existente para atuar em serviços em que esta originalmente não seria capaz de atuar. A WSInject possuía a restrição de que o cliente precisaria ser configurado se conectar por meio de proxy, e nesse experimento conseguimos eliminar essa restrição. Com isso, aumentamos a abrangência de nossa ferramenta, tornando-a capaz de atuar em

serviços, clientes e outras ferramentas, mesmo quando estes não possam ser configurados para se conectarem por meio de um *proxy* (como era o caso da wsrbench).

Além de viabilizar o uso da wsrbench com serviços que exigem WS-Security (ou qualquer outra característica especial na mensagem SOAP), a inclusão de mais uma modificação na mensagem – remoção dos elementos contendo '?' – potencializou a ação da wsrbench. Com a modificação extra, novos defeitos foram encontrados nos serviços testados, passando de 0,3% para 5,4%.

A wsrbench funciona de maneira extremamente automática: basta informar a localização do documento WSDL e ela já é capaz de testar todas as operações do serviço informado. No experimento atual, a wsrbench exercitou todas as operações do Web Service do sistema real, encontrando, dessa forma, um grande número de defeitos. Isso não ocorreu no experimento #2 porque a WSInject atualmente não possui esse grau de automatização: as chamadas a operações do serviço devem ser feitas por um cliente separado, já que a ela não exerce a função de cliente.

5.4. Considerações finais

Os experimentos foram realizados com sucesso e demonstraram diferentes usos e funcionalidades da ferramenta. O experimento #1 ilustrou o uso da WSInject com serviços compostos e algumas injeções de falhas de comunicação (como *DelayFault*). O experimento #2 ilustrou a aplicação da ferramenta em um sistema comercial, sendo este o uso mais próximo daquele que ocorreria em um teste real de sistema. Finalmente, o experimento #3 demonstrou a capacidade de extensão da WSInject por meio da integração com outras ferramentas.

Muitos resultados obtidos no experimento #2 indicam que uma análise manual de resultados de experimentos é fundamental para a correta classificação dos defeitos. Além disso, é importante destacar que essa classificação é fortemente dependente do contexto e da especificação da aplicação em testes. Por exemplo, julgamos como defeito a resposta "*OK*" quando um pedido de remoção de ativo não efetuou de fato a remoção. No entanto, em vez disso, tal resposta também poderia ser interpretada como "nenhum problema ocorreu". Além disso, as mensagens de erro variam entre diferentes bibliotecas de Web Services e entre diferentes linguagens de programação. Uma possível classificação

automática de resultados seria bastante trabalhosa – e não necessariamente desejável. Alguns resultados dependem da interpretação do testador, principalmente quando não houver uma especificação formal do sistema.

O experimento #3 mostra que é possível integrar a WSInject com outras ferramentas de injeção de falhas e que essa integração pode trazer bons resultados. A wsrbench, em particular, executa testes de forma automática em todas as operações do Web Service, o que permite encontrar um grande número de defeitos.

É importante ressaltar que os resultados desses experimentos foram obtidos em seus respectivos contextos. Testes realizados em outros Web Services poderão apresentar resultados diferentes. Os experimentos e resultados mostrados neste capítulo devem ser visto como uma base, a partir da qual outros experimentos em diferentes contextos poderão ser realizados.

6. Conclusões e Trabalhos Futuros

Esta dissertação apresentou um estudo sobre testes de robustez em Web Services por meio da aplicação da técnica de injeção de falhas. Foram analisados trabalhos similares propondo modelos de falhas e ferramentas de injeção de falhas. Foi apresentada a WSInject, ferramenta desenvolvida para auxiliar os testadores em sua atividade, com as vantagens de possuir baixa intrusividade e de ser flexível com relação ao modelo de falhas utilizado. Foram apresentados experimentos realizados com a WSInject, que mostraram seu potencial para encontrar defeitos de robustez em sistemas baseados em Web Services. No decorrer deste trabalho, foi publicado um artigo apresentando a ferramenta e alguns experimentos preliminares [2], além de um relatório técnico contendo informações detalhadas sobre seu uso e implementação [33].

Outras contribuições deste trabalho foram:

- Análise e comparação de ferramentas de injeção de falhas já existentes;
- Criação do modelo de falhas XML;
- Possibilidade de testar o cliente de um Web Service, que deverá ser explorada em trabalhos futuros;

Foram realizados experimentos com um sistema demonstrativo, meramente experimental (TRS do NetBeans), e com um sistema real (de organização de ativos de software). Este último foi testado em dois momentos: primeiramente apenas com a WSInject e, em seguida, com a WSInject em conjunto com a ferramenta wsrbench. A wsrbench não pôde ser aplicada diretamente a esse sistema devido à incompatibilidade com o padrão WS-Security. Por esse motivo, a WSInject foi utilizada para interceptar as mensagens da wsrbench e fazer as adaptações necessárias, viabilizando os testes.

Os experimentos indicaram que a WSInject é capaz de detectar problemas tanto em sistemas experimentais quanto em sistemas reais. A liberdade de escolha de modelo de falhas permitiu que vários modelos fossem testados e seus resultados, comparados. O uso de falhas dinâmicas permite que a ferramenta seja facilmente estendida por meio de geradores dinâmicos de falhas. Novos geradores podem ser implementados pelo usuário, que poderá, assim, definir seu próprio modelo. Além disso, a WSInject integrou-se com sucesso a outra ferramenta, wsrbench, possibilitando outras formas de utilização.

Os experimentos indicaram que, na atividade de injeção de falhas em Web Services, a análise e classificação das respostas obtidas é uma tarefa difícil. Isso ocorre porque não existe uma especificação bem definida que determine as respostas possíveis de um Web Service. Esse cenário é bastante diferente do que ocorre, por exemplo, com o HTTP, que possui códigos de erro padronizados (erro 403, 404, 500 etc.). Para requisições contendo falhas equivalentes, várias respostas diferentes foram recebidas. Isso dificulta a classificação dos resultados, principalmente de forma automática. Isso foi observado principalmente no defeito de robustez encontrado: após ser injetada a falha causadora, o servidor enviou como resposta uma mensagem no formato HTML, apesar de o padrão Web Service especificar que ela deveria ser dada no formato SOAP.

Além disso, em alguns casos, a análise desses resultados é uma atividade subjetiva. Se não houver uma especificação formal e detalhada do sistema, algumas respostas do serviço podem ser consideradas corretas ou incorretas, dependendo da perspectiva e da interpretação do testador. Por exemplo, como classificar uma exceção retornada pelo serviço? A princípio, parece ser uma resposta incorreta e possivelmente não robusta, mas essa exceção pode ser um mero indicador de erro no processamento da mensagem (por exemplo, no *parsing* do XML), o que seria um comportamento correto. Conclui-se que as respostas devem ser sempre analisadas com cuidado e que se deve levar em consideração o contexto em que elas são geradas.

Como trabalhos futuros, podemos citar diversas melhorias a serem realizadas na WSInject, principalmente na funcionalidade *proxy transparente*, utilizada no experimento #3. A prova de conceito realizada levou à obtenção de resultados positivos, e a implementação de melhorias nessa funcionalidade permitirá que ela seja aplicada adequadamente a qualquer serviço. Além disso, deverá ser estudada a possibilidade de permitir o uso da WSInject de forma mais automática. Isso pode ser feito por meio da interface em linha de comando, mas exige a criação de *scripts* externos. Tal configuração poderia ser feita dentro da própria ferramenta.

Outra funcionalidade interessante a ser implementada seria o uso de compressão de dados no envio de mensagens. Conforme mencionado na pré-análise do modelo XML no experimento #2, isso tornaria mais praticável a aplicação de alguns tipos de falhas, como a repetição de muitos elementos (<el> </el> <el> </el> <el> </el> ...). Essas falhas

costumam produzir saídas muito grandes, que, ao serem comprimidas, tornam-se muito pequenas, devido à grande redundância. Isso facilitaria a aplicação dessas falhas, dado que mensagens muito grandes costumam causar *timeouts*.

Considerando-se os fatores carga de trabalho e carga de falhas, novos experimentos poderão ser realizados. Os experimentos deste trabalho priorizaram a carga de falhas. A carga de trabalho apenas serviu como base para a injeção de falhas. Outros experimentos poderão usar uma combinação dos dois fatores para procurar obter um melhor resultado – abordagem considerada promissora por ReSIST Team [29].

A biblioteca de modelos de falhas da WSInject também poderá ser estendida para contemplar os modelos de falhas usados nos trabalhos de Xu *et al.* [40], Almeida e Vergílio [1] e Silveira e Melo [32]. Apesar de o foco do nosso trabalho ser em robustez e o foco destes trabalhos ser em testes funcionais, um estudo mais aprofundado poderia ser realizado para encontrar pontos de convergência entre as abordagens, chegando-se a um modelo capaz de causar defeitos de robustez.

Ainda nessa linha, uma análise mais completa de outros modelos de falhas poderia ser feita, além de uma possível combinação desses modelos. Um número maior de sistemas poderia ser testado e um número maior de experimentos poderia ser realizado, com o objetivo de determinar quais modelos de falhas tendem a apresentar melhores resultados em quais tipos de sistemas.

7. Referências

[1] Almeida, L. F. A., Vergílio, S. R. Explorando Teste Baseado em Perturbação no Contexto de Web Services. *Dissertação de Mestrado, Universidade Federal do Paraná*, 2009.

[2] Bessayah, F., Cavalli, A., Maja, W., Martins, E., Valenti, A. W. A fault injection tool for testing web services composition. *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, Springer-Verlag, 2010.

[3] Canfora, G., Di Penta, M. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, vol. 8, 2006.

[4] Canfora, G., Di Penta, M., Esposito, R., Villani, M. L. A Lightweight Approach for QoS-aware Service Composition. *International Conference On Service Oriented Computing*, 2004.

[5] Canfora, G.; Di Penta, M. Service-Oriented Architectures Testing: A Survey. *International Summer Schools on Software Engineering 2006 – 2008*, *LNCS 5413*, Springer-Verlag Berlin Heidelberg, 2009.

[6] DeVale, J. P., Koopman P. J., Guttendorf D. J.. The Ballista software robustness testing service. *Testing Computer Software Conference (Bethesda, MD)*, junho/1999.

[7] Koopman, P. J., DeVale, J. P. Comparing the Robustness of POSIX Operating Systems – Main conference slides. *Proceedings of FTCS'99*, Madison, Wisconsin, EUA, junho/1999.

[8] Fugini, M. G., Pernici, B., and Ramoni, F. Quality Analysis of Composed Services through Fault Injection. *BPM 2007 Workshops, LNCS 4928*, 2008.

[9] Han, S., Shin, K. G., Rosenberg, H. A. DOCTOR: an integrated software fault injection environment for distributed real-time systems. *Computer Performance and Dependability Symposium Proceedings,* abril/1995.

[10] IEEE Standard Glossary of Software Engineering Terminology. http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html. Acesso em 01/08/2011.

[11] Martins, E., Rubira, C. M. F., Leme, N. G. M. JACA Software Fault Injection Tool. http://www.ic.unicamp.br/~eliane/JACA.html. Acesso em 18/06/2011.

[12] Jensen, M., Gruschka, N., Herkenhöner, R., Luttenberger, N. SOA and Web Services: New Technologies, New Standards – New Attacks. *Fifth European Conference on Web Services (ECOWS'07)*, 2007.

[13] Koopman, P., DeVale, J. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Transactions on Software Engineering*, Vol. 26, No. 9, Setembro/2000.

[14] Laranjeiro, N., Canelas, S., Vieira, M. wsrbench: An On-Line Tool for Robustness Benchmarking. *IEEE International Conference on Services Computing*, julho/2008.

[15] Leme, N. G. M.; Martins, E.; Rubira, C.M.F. A Software Fault Injection Pattern System. *Technical Report of Institute of Computing*, UNICAMP, 2002.

[16] Looker, N, Xu, J. Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection. *Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*, 2004.

[17] Looker, N., Munro, M., Xu, J. Assessing Web Service Quality of Service with Fault Injection. *Workshop on Quality of Service for Application Servers, SRDS*, Brasil, 2004.

[18] Looker, N., Munro, M., Xu, J. Increasing Web Service Dependability Through Consensus Voting. *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, 2005.

[19] Looker, N., Munro, M., Xu, J. Simulating Errors In Web Services. *International Journal of Simulation*, Vol. 5, No. 5, 2004.

[20] Looker, N., Munro, M., Xu, J. Testing Web Services. *Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems*, Oxford, 2004.

[21] Looker, N., Munro, M., Xu, J. WS-FIT: A Tool for Dependability Analysis of Web Services. *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, 2004.

[22] Martins, E., Rubira, C. M. F., Leme, N. G. M. Jaca: A reflective fault injection tool based on patterns. *International Performance & Dependability Symposium (IPDS), joint with DSN*, Washington, EUA, junho/2002.

[23] Mendes, N., Moraes, R., Martins, E., Madeira, H. Jaca Tool Improvements for Speeding Up Fault Injection Campaigns. *13ª. Sessão de Ferramentas. 20º. Simpósio Brasileiro de Engenharia de Software. SBES, Florianópolis*, 18-20/outubro/2006.

[24] Menegotto, C. C., Weber, T. S. Injeção de falhas de comunicação em aplicações multiprotocolo. *Dissertação de Mestrado, Universidade Federal do Rio Grande do Sul*, 2009.

[25] Miller, B. Fuzz Testing of Application Reliability. http://pages.cs.wisc.edu/~bart/fuzz/. Acesso em 01/08/2011.

[26] Offutt, J., Xu, W. 2004. Generating Test Cases for Web Services Using Data Perturbation. *SIGSOFT Softw. Eng. Notes 29, 5*, setembro/2004.

[27] Postel, J. Transmission Control Protocol – DARPA Internet Program – Protocol Specification. *RFC 793*, 1981. http://tools.ietf.org/html/rfc793. Acesso em 09/06/2011.

[28] Reinecke, P., Wolter, K. Towards a Multi-Level Fault-Injection Test-bed for Service-Oriented Architectures: Requirements for Parameterisation. *SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*, 2008.

[29] ReSIST Team. Resilience-Building Technologies: State of Knowledge. Deliverable D12, setembro/2006. http://www.resist-noe.org/outcomes/outcomes.html. Acesso em 01/08/2011.

[30] Ribarov, L., Manova, I., Ilieva, S. Testing in a Service-Oriented World. *Proceedings of the International Conference on Information Technologies (InfoTech-2007)*, 2007.

[31] SeCSE Team. State of the Art on Service Testing, 2004. *http://www.secse-project.eu/?page_id=80*. Acesso em 01/08/2011.

[32] Silveira, P., Melo, A. C. Exploring XML Perturbation Techniques for Web Services Testing. *Proceedings of the 9th International Conference on Web Engineering (ICWE '9)*, Springer-Verlag Berlin Heidelberg, 2009.

[33] Valenti, A. W., Maja, W. Y., Martins, E., Bessayah, F., Cavalli, A. WSInject: A Fault Injection Tool for Web Services Technical Report 1.0. *Instituto de Computação, UNICAMP*, julho de 2010. http://www.ic.unicamp.br/~reltech/2010/abstracts.html. Acesso em 01/11/2011.

[34] van Moorsel, A. *et al.* State of Art. Amber: Assessing, Measuring, and Benchmarking Resilience. Deliverable nº D2.1, versão 1.0, abril/2008. http://www.amber-project.eu/. Acesso em 01/08/2011.

[35] Vieira, M., Laranjeiro, N., Madeira, H. Benchmarking the Robustness of Web Services. *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC '07).* IEEE Computer Society, Washington, DC, EUA, 2007.

[36] Weber, T. S. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. Curso de Especialização em Redes e Sistemas Distribuídos. Instituto de Informática, UFRGS, 2002. http://www.inf.ufrgs.br/~taisy/disciplinas/textos/Dependabilidade.pdf. Acesso em 01/08/2011.

[37] W3C Working Group. Web Services Glossary. http://www.w3.org/TR/2004/NOTE-ws-gloss/. Fevereiro/2004. Acesso em 19/06/2011.

[38] W3C Recommendation. Extensible Markup Language (XML). http://www.w3.org/TR/REC-xml/. Novembro/2008. Acesso em 19/06/2011.

[39] World Wide Web Consortium. http://www.w3.org/. Acesso em 11/03/2011.

[40] Xu, W., Offutt, J., Luo, J. Testing Web Services by XML Perturbation. *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE '05).* IEEE Computer Society, Washington, DC, EUA, 2005.