INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

WSInject: A Fault Injection Tool for Web Services – Technical Report 1.0

A. W. Valenti *W. Y. Maja* *E. Martins*
F. Bessayah *A. Cavalli*

Technical Report - IC-10-22 - Relatório Técnico

July - 2010 - Julho

WSInject: A Fault Injection Tool for Web Services
Technical Report 1.0

André Willik Valenti[1], Willian Yabusame Maja[1], Eliane Martins[1],

Fayçal Bessayah[2], Ana Cavalli[2]

[1] Instituto de Computação Universidade Estadual de Campinas (UNICAMP), Caixa Postal 6176
13083-970 Campinas-SP, Brazil.

{andre.valenti, willian.maja}@students.ic.unicamp.br
eliane@ic.unicamp.br

[2] IT/TELECOM SudParis – CNRS SAMOVAR, Evry, France.

{fayçal.bessayah, ana.cavalli}@it-sudparis.eu

Abstract

Service-Oriented Architecture (SOA) is a promising paradigm for developing distributed systems. It offers flexibility, dynamism and interoperability. Robustness is the dependability attribute which measures the degree to which a system operates correctly even in the presence of stressful environmental conditions or exceptional inputs, such as faults. Web Services are the main technology for implementing SOAs. Since Web Services are usually deployed on the internet – an unstable environment –, robustness problems might arise if the system is not properly tested. These problems might cause unexpected system behavior and must be avoided. On this context, we present WSInject, a fault injection tool for testing single and composed Web Services, which enables the user to evaluate the behavior of Web Services in the presence of faults. It is able to inject both communication and interface faults and was designed to be the minimally intrusive. This report describes WSInject's design and architecture, explains how to use it, compares it to similar tools and presents our preliminary experiments on a simple system.

Keywords: Dependability, Fault Injection, Robustness, Testing, Web Services.

1. Introduction

Service-Oriented Architecture (SOA) and Web Services technologies offer many advantages for the development of distributed systems. They are flexible, dynamic and interoperable. An increasing number of e-business, e-commerce, e-science and other applications are being developed based on Web Services compositions, a typical implementation of SOAs. In order to be a reliable solution, Web Services must be correctly tested. Robustness testing is an interesting approach for evaluating the behavior of a system in the presence of exceptional inputs and stressful environments. Failures exposed by applying this technique can be analysed so that system faults can be found and corrected.

Fault injection is a useful technique for robustness testing. It emulates the occurrence of real faults on a system by injecting artificial faults on it. The purpose of fault injection is to cause system failures, therefore revealing robustness problems and indicating directions to fix them.

The usual environment where Web Services run should be taken into account when developing a fault injection tool: they commonly work geographically distributed and use the Internet to exchange messages. On many cases, the tester has no access to the source code nor to the machine where the service is running. Thus, we believe that an important requirement for a fault injection tool to handle this kind of environment is to use a *black box* approach, meaning that only the Web Services interface should be considered, not its implementation (except for composed Web Services, for which their partner services should be considered). We also point out the requirements that such fault injectors be capable of handling composed Web Services properly and be able to inject communication and interface faults.

This report proposes **WSInject**, a work-in-progress tool implemented in Java for fault injection on both single and composed Web Services. It has a proxy-based approach that makes it minimally intrusive and able to intercept all messages exchanged by participants of Web Services compositions. It is also script-driven, a feature that simplifies the execution of test campaigns: by creating simple text-file scripts, the user specifies what faults should be injected and on which messages. More information about WSInject can be found on its website[1].

WSInject's fault library includes a diversity of communication and interface faults. **Interface faults** are those that affect contents of messages (e.g.: parameters corruption), while **communication faults** are those that affect communication between Web Services and their clients at a lower level (e.g.: loss of connection). This library allows users to select desired faults to be injected, in order to emulate as accurately as possible a set of real faults occuring on a system.

An initial validation of WSInject was made by testing **TravelReservationService**, a widely available Web Service composition that simulates a real-life travel itinerary reservation service. The rest of this report is organized as follows: section 2 explains how fault injection occurs in a Web Services environment; section 3 presents the architecture of WSInject; section 4 describes our preliminary experiments with the tool; and section 5 presents our conclusions and future work.

2. Fault Injection in Web Services Environment

Part of Web Services success is due to their interoperability. This is assured by a number of communication standards based on XML (eXtensible Markup Language). On

[1] http://robustweb.ic.unicamp.br/tools/wsinject

the context of WSInject, the standards of interest are WSDL, SOAP and BPEL [2] [3]. **WSDL** (Web Services Description Language) is a language for describing Web Services and defining their interfaces. **SOAP** is a protocol for exchanging messages between clients and services. **BPEL** (short for WS-BPEL, Web Services Business Process Execution Language) is an executable language for describing a business process as a composition of Web Services. A fault injection tool should be able to use information from SOAP messages to be able to inject interface faults on Web Services.

A Web Service can be either a single, atomic service executing an operation, or a composition of multiple, existing Web Services – called **partner services**. Web Services compositions are usually expressed in BPEL. Web Services compositions expressed in BPEL are called **BPEL processes**, and are deployed on a **BPEL engine**, a container for these types of processes.

As an example, a travel itinerary reservation company may use services from other companies to reserve a vehicle, a hotel and an airline for a client. In this scenario, each partner service executes an atomic process (reserving a vehicle, hotel or airline). A new process can be created by combining these processes into one that reserves the whole itinerary. This new process is a Web Service composition, and is seen by clients as a new, ordinary Web Service. In Web Service compositions, messages are exchanged both by the client and the composition, and by the composition and its partner services. A fault injection tool should be able to inject faults on both of these paths (client ⇔ composition, composition ⇔ partner services).

2.1. Related Work

There is a number of existing fault injection tools for Web Services. Some of the most widely known are **wsrbench**, proposed by Laranjeiro *et al.* [LARANJ 08], and **WS-FIT**, proposed by Looker *et al.* [LOOKER 04]. wsrbench is described as an online tool for robustness benchmarking of Web Services. It is a web-based fault injection system that connects to a specified Web Service on the internet, runs a series of tests on it, collects data about the results and presents them to the user. WS-FIT is described as a network level fault injection tool. It works on the network level from the OSI model [ZIMMER 80], although it is able to corrupt individual parameters at the application level. To operate, it requires inserting hook code in the SOAP API of a Web Service.

While wsrbench operates as a standard client of Web Services, WS-FIT works by instrumenting the environment on which a Web Service runs. A limitation of wsrbench is that SOAP requests are automatically generated and cannot be modified by the user. This means that Web Services that have special requirements for the request, such as WS-Security[4] headers, cannot be tested. Another wsrbench limitation is that, when testing

[2] http://www.w3.org/standards/webofservices/

[3] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[4] http://www.oasis-open.org/specs/index.php#wssv1.1

composed Web Services, it is not capable of injecting faults between compositions ad partner services; it tests composed services the same way as single services. WS-FIT, on the other hand, is able to test composed services by instrumenting a Web Service in order to test another one, which may be used to inject faults on partner services. While wsrbench is available to the public[5], WS-FIT is not.

WSInject works by intercepting messages exchanged between a client and Web Services, and between a BPEL process and partner services. WSInject eliminates some types of intrusivenesses on Web Services, clients and BPEL processes, such as hook code intrumentation found in WS-FIT. It only requires that clients and BPEL engines are configured to connect through a proxy, usually a very simple step.

2.2. Failure Classification

The major objective of fault injection is to cause failures on the system under test. Failures should be classified in some way that facilitates analysing results of experiments. A popular classification is the **CRASH scale** [KOOP 97], which defines 5 different **failure modes**: **C**atastrophic (OS crashes/multiple tasks affected), **R**estart (task/process hangs, requiring restart), **A**bort (task/process aborts, *e.g.*, segmentation violation), **S**ilent (no error code returned when one should be) and **H**indering (incorrect error code is returned).

Vieira *et al.* propose an adapted version of the CRASH scale for Web Services, the **wsCRASH** scale [VIEIRA 07]. wsCRASH failure modes are: Catastrophic (the application server used to run the Web Service under test becomes corrupted, or the machine crashes or reboots), Restart (the Web Service execution hangs and must be terminated by force), Abort (abnormal termination of the Web Service execution), Silent (no error is indicated by the application server) and Hindering (the error code returned is not correct or the response is delayed).

Vieira *et al.* state, however, that wsCRASH is only significant when testing a Web Service from a developer's point of viewIt is not possible to use the wsCRASH scale from a client's point of view [VIEIRA 07]. Since tests are run remotely, it is impossible to distinguish, for instance, between Catastrophic and Restart failures, because the client has no access to the application server where the Web Service is running. For this reason, they propose a simplified classification scale, **wsAS**, based on two easily observable failure modes: **Abort** (an unexpected exception is raised by the application server and sent to the Web Service client) and **Silent** (no response from the server within a timeout period).

3. WSInject Architecture

Figure 1 depicts WSInject architecture, designed to be simple and loosely coupled. It was based on a pattern of fault injection systems proposed by Leme *et al.* [LEME 00].

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

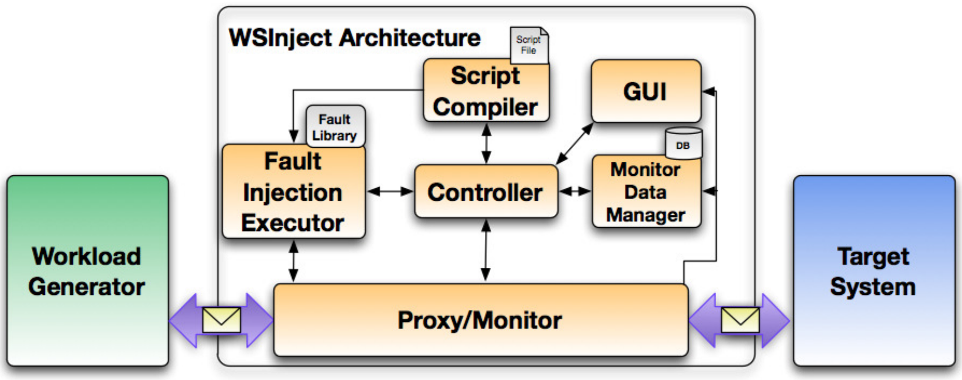


Figure 1: WSInject architecture.

Core WSInject components are **Proxy/Monitor** and **Fault Injection Executor**. Proxy/Monitor is the SOAP messages interception and failure monitoring point. Fault Injection Executor is the point where effective fault injection occurs. Other important components are **Controller**, **Script Compiler** and **Graphical User Interface (GUI)**. Controller is the starting point of the tool; it activates and starts other components. Script Compiler is the component that reads a fault injection campaign script and converts it into a processable format. GUI is responsible for showing data collected by the Proxy/Monitor. All these components are explained below in more details.

3.1. Proxy/Monitor

Proxy/Monitor is a 2-in-1 component that intercepts SOAP messages and monitors system behavior. User is able to select the port on which the proxy should be bound to. The client and the BPEL engine should then be configured to connect through a proxy on the selected port and on the IP address of the machine where WSInject is running (possibly the same machine as the client's or the BPEL engine's, *i.e.*, localhost). If the Target System presents any kind of failure (like crashing), Proxy/Monitor will keep track of this behavior.

More specifically, Proxy is a socket-based HTTP proxy, implemented using the java.net.Socket and java.net.ServerSocket classes. It intercepts every HTTP message exchanged by participants of Web Services communication, parses it, sends it to the Fault Injection Executor, receives the (possibly) modified message and finally sends it to its original destination. Non-SOAP HTTP messages are also intercepted, but these suffer no modification before being redirected to their original destination.

3.2. Script Compiler

Fault injection campaigns are described by scripts. Script Compiler is the component responsible for compiling a script and transforming it into a **CampaignDescriptor**. A CampaignDescriptor is an Abstract Syntax Tree (AST) that is

WSInject's internal representation of a script. It is part of the Fault Injection Executor component, more thoroughly explained later.

Scripts are simple text files containing one or more **FaultInjectionStatement**s. FaultInjectionStatements are composed of a **ConditionSet** and a **FaultList**. A ConditionSet consists of one or more **Condition**s and a FaultList is composed of one or more **Fault**s. FaultInjectionStatements work as *condition-action* statements: when a message arrives, if it matches a set of conditions, a list of faults is injected on it. Conditions are similar to boolean methods and faults are similar to void methods. Conditions have no defined order – hence being grouped in a *set*; faults do have a defined order – hence being grouped in a *list*.

Table A presents available conditions and Table B presents available faults to be injected (or "actions" to be taken). *Name/Class* is both the name of that condition or fault and its corresponding Java class on WSInject code. *Syntax* describes how that condition or fault is expressed on the script language.

Table A: Available conditions.

Name/Class	**Syntax**	**Description**
ContainsCondition	**contains**(String stringPart)	Matches SOAP messages containing the specified string.
URICondition	**uri**(String uriPart)	Matches request messages sent to a URI containing the specified string, and responses to those messages.
MessageDestinationCondition	**isRequest**() **isResponse**()	Matches request messages, either from a client to a service, or from a BPEL process to a service. Matches response messages either from a service to a client, or from a service to a BPEL process.
OperationCondition	**operation**(String operationPart)	Matches request messages sent to a Web Service operation whose name contains the specified string, and responses to those messages. *Note: this feature is still under development.*

Table B: Available faults.

Name/Class	Syntax	Description
INTERFACE FAULTS		
StringCorruptionFault	**stringCorrupt**(String fromString, String toString)	Replaces all occurences of fromString with toString. Works at String level. Ignores XML syntax (may be used to replace XML characters like '<' and '>').
XPathCorruptionFault	**xPathCorrupt**(String xPathExpression, String newValue)	Replaces all matches of an XPath[6] expression to the value specified. Can be used to modify either elements or attributes.
MultiplicationFault	**multiply**(String xPathExpression, int multiplicity)	Multiplies a part of a message by a number of times. For example, multiply("/", 2) duplicates the whole message contents, while multiply("/Envelope/MyNode", 3) triplicates only the MyNode XML element. *Note: this feature is still under development.*
EmptyingFault	**empty**()	Empties the SOAP message, delivering an HTTP message with no contents.
COMMUNICATION FAULTS		
DelayFault	**delay**(int delayInMilliseconds	Delays a message delivery by the specified number of milliseconds.
ConnectionClosingFault	**closeConnection**()	Closes the connection between client and proxy. *Note: this feature is still under development.*

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

Interface faults modify contents of SOAP messages, while communication faults affect the delivery of requests and/or responses. To emulate a message modification, user should simply choose the most appropriate interface fault for his/her needs. To emulate an unresponsive Web Service (i.e., network packet loss), user has two options: 1) use DelayFault to delay a response message (possibly by a very large amount of time); 2) use ConnectionClosingFault to abruptly close the conection between proxy and client without returning any HTTP answer to the client. Note that a more accurate emulation of unresponsive services/packet loss is not possible working at the HTTP level like WSInject does. According to Reinecke and Wolter [REIN 08], this would require working at the network level.

Conditions can be combined by using the '&&' (AND) operator, meaning a ConditionSet will only be satisfied when all individual conditions are satisfied. Faults can be combined by the ',' (comma) operator, meaning all of them will be injected, on the specified order. Figure 2 shows a sample script.

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

Figure 2: Sample script.

The example above has three FaultInjectionStatements, one on each text line. The first one has a ConditionSet of a single condition: a **URICondition** with a "Hotel" argument. It also has a FaultList of two Faults: **StringCorruptionFault** with "Name" and "Age" arguments and a **MultiplicationFault** with "/" and '2' arguments. The second FaultInjectionStatement has a ConditionSet with a URICondition and a FaultList with a StringCorruptionFault. The last FaultInjectionStatement has a ConditionSet with two conditions: a **ContainsCondition** and a **MessageDestinationCondition**; and a FaultList with an **EmptyingFault**. This script describes the following campaign:

- Whenever a URI of a Web Service call or its response contains the string "Hotel":
  1. Replace all text occurrences of "Name" by "Age".
  2. Duplicate the whole SOAP message.
- Whenever a URI of a Web Service call contains the string "Airline":
  1. Replace all text occurrences of "Flight" by "Might".
- Whenever a message contains the string "caught exception" and is a response to a Web Service caller:
  1. Empty the message.

Instead of manually writing the Script Compiler code, we used the JavaCC 5.0[7] tool to generate it automatically from a specification. This approach makes it easier to modify the script language when necessary, in order to, *e.g.*, create new types of conditions and faults for WSInject. JavaCC is a “compiler compiler”, which generates Java code from a language specification. The specification of our script language can be found at file gramar.jj, on package br.unicamp.ic.robustweb.wsinject.othercomponents.scriptcompiler. This file represents the script language grammar, depicted on Figure 3.

```
CampaignDescriptor → FaultInjectionStatement [CampaignDescriptor]

FaultInjectionStatement → ConditionSet : FaultList ;
[FaultInjectionStatement]

ConditionSet → Condition [&& ConditionSet]

Condition → operation(String) | contains(String) | uri(String) |
isRequest() | isResponse()

FaultList → Fault [, FaultList]

Fault → delay(Integer) | multiply(String, Integer) |
stringCorrupt(String, String) | xPathCorrupt(String, String) | empty() |
closeConnection()
```

Figure 3: Script language grammar.

3.3. Fault Injection Executor

Fault Injection Executor is the component in charge of effectively injecting faults. It processes the Abstract Syntax Tree (AST) produced by Script Compiler and injects faults where appropriate. For example, when a message should be corrupted, the Executor is the component which actually modifies the message; when the message should be delayed, the Executor is the component which actually inserts an emulated delay on the program execution. Fault Injection Executor code is called for all messages intercepted by the Proxy. For those that do satisfy the specified ConditionSet, Executor injects the appropriate faults. For those that do not, it takes no action.

Representing source code as ASTs is a common approach in the compilers field which facilitates the code processing. On WSInject, a CampaignDescriptor is an AST which is an exact representation of a fault injection script. Each element of the script

[7] https://javacc.dev.java.net/

corresponds to an AST node, while each AST node corresponds to a Java class on WSInject code. Figure 4 shows the AST corresponding to the script from Figure 2.

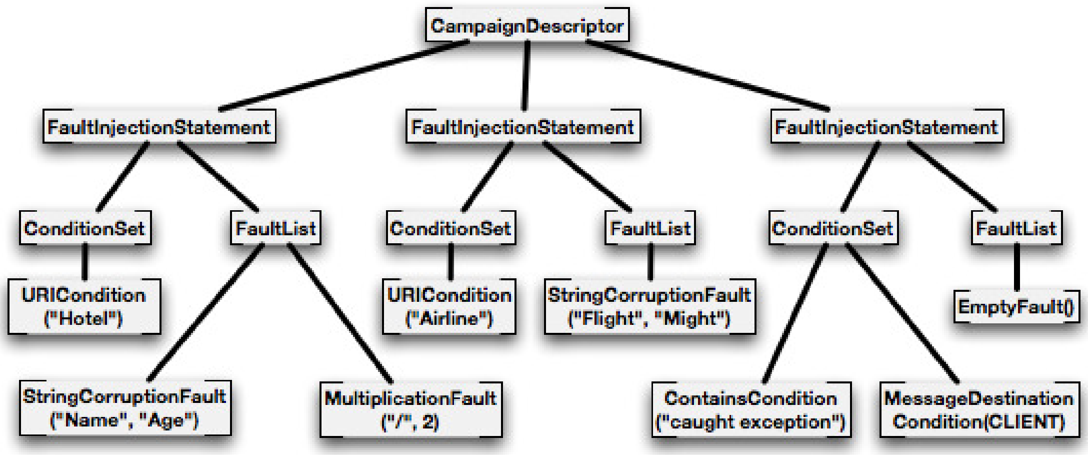
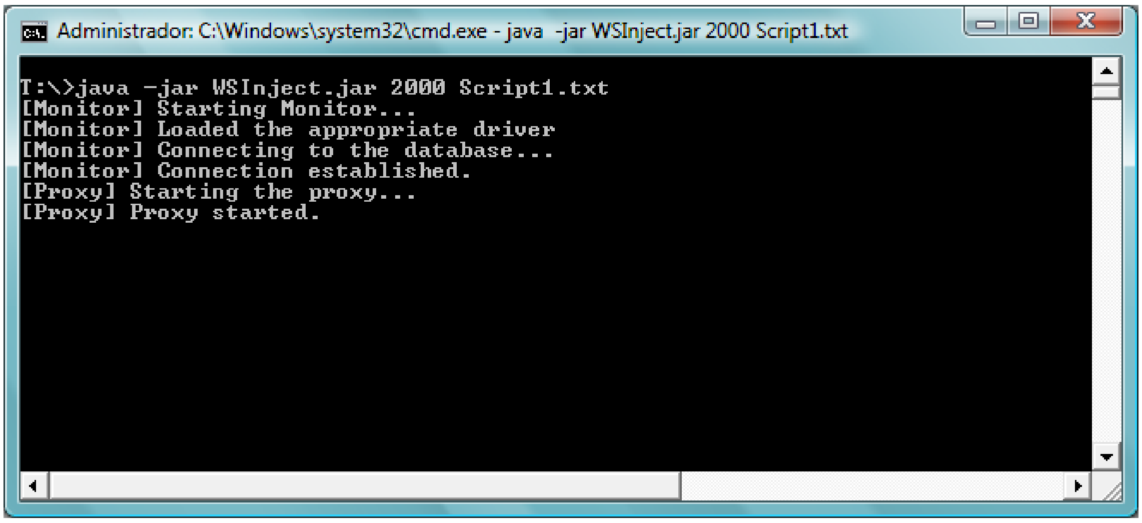


Figure 4: Abstract syntax tree generated from script of Figure 2.

3.4. Controller

Controller is the central component of WSInject. It starts the tool and activates other components when required. WSInject can be started in two modes: graphical user interface (GUI) or command-line interface (CLI). To activate the GUI mode, the main method of the Controller class should be called with no arguments. To start the CLI mode, the main method of Controller class should be called with arguments PORT and SCRIPT_PATH, which are respectively the port that the proxy should should be bound to, and the path to the script file to be loaded. Figure 5 shows the tool started on the command-line interface mode. GUI mode is described on the GUI component section.


Administrador: C:\Windows\system32\cmd.exe - java -jar WSInject.jar 2000 Script1.txt

```
T:\>java -jar WSInject.jar 2000 Script1.txt
[Monitor] Starting Monitor...
[Monitor] Loaded the appropriate driver
[Monitor] Connecting to the database...
[Monitor] Connection established.
[Proxy] Starting the proxy...
[Proxy] Proxy started.
```

Figure 5: WSInject started on command-line interface.

The initialization of WSInject with a fault injection campaign is described on the sequence diagram on Figure 6. First, the Controller asks the Script Compiler to compile the script file into a CampaignDescriptor, which represents the entire fault injection campaign. Controller then creates and configures a Fault Injection Executor, and passes it to the Proxy/Monitor. After these steps, WSInject is ready to identify desired messages and inject faults described on the script file. Final steps are to start the Proxy/Monitor, start the client (which will send SOAP messages) and stop the Proxy/Monitor after the experiment is completed.

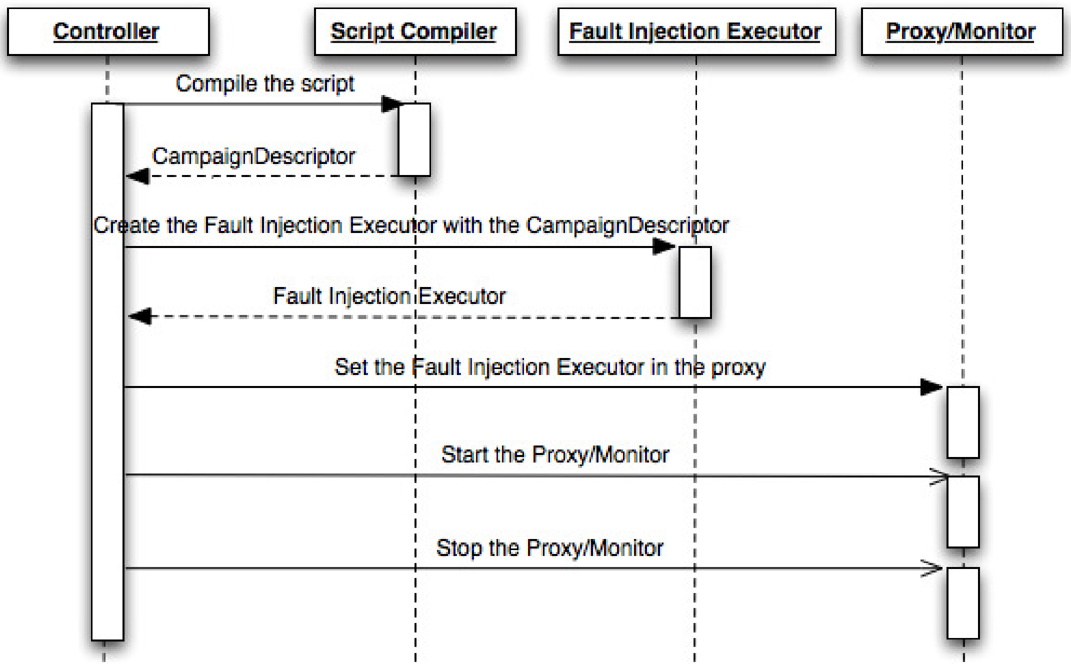
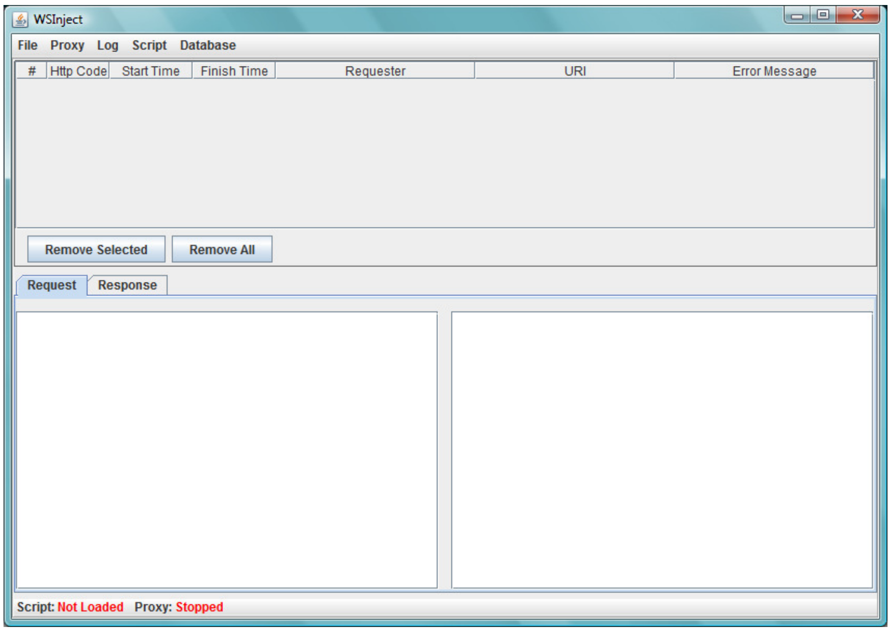


Figure 6: Initialization of WSInject's main components.

3.5 Graphical User Interface (GUI)

The GUI component is responsible for receiving user inputs and for showing SOAP messages to the user. User inputs include setting the proxy port, turning the proxy on/off and loading/unloading scripts. Request and response messages can be seen by clicking their respective tabs. The left and right white panels respectively show messages contents before and after fault injection. Figure 7 depicts WSInject started on graphical user interface mode.

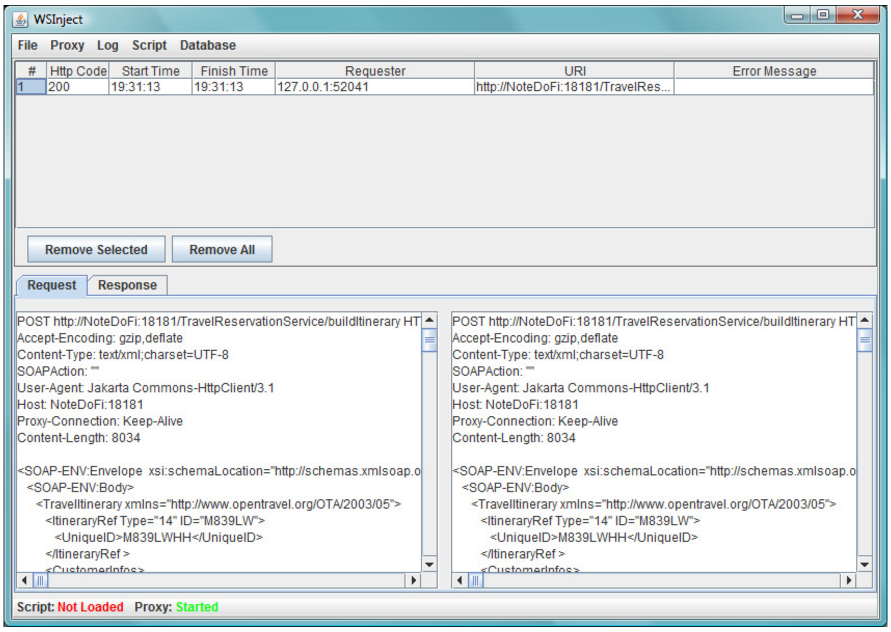


#	Http Code	Start Time	Finish Time	Requester	URI	Error Message

```
Script: Not Loaded   Proxy: Stopped
```

Figure 7: GUI (just started).

Figure 8 to Figure 11 show the GUI after the proxy is started, some messages are exchanged, some scripts are loaded and user has clicked on each message received – notice the selected cell on the "#" column, where the user has clicked.



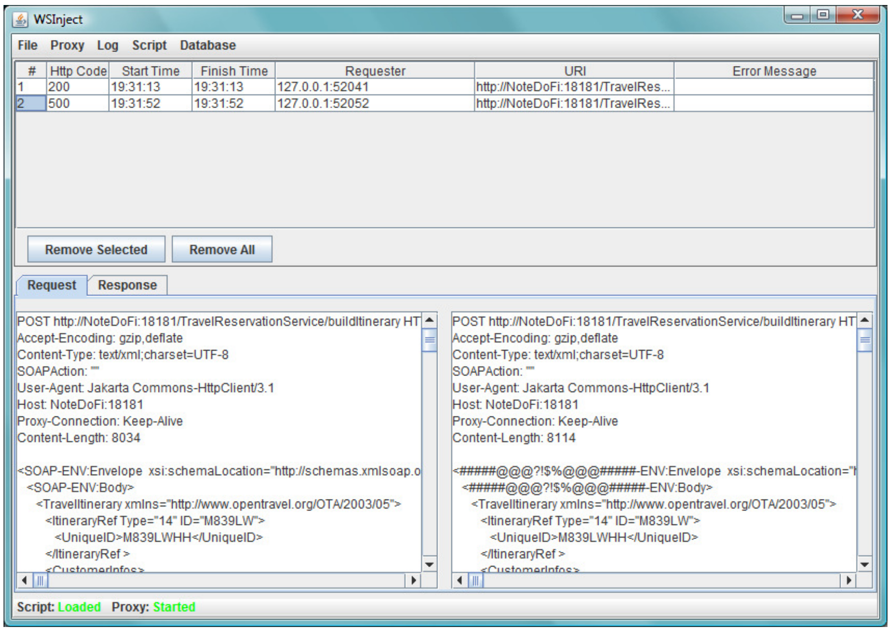
#	Http Code	Start Time	Finish Time	Requester	URI	Error Message
1	200	19:31:13	19:31:13	127.0.0.1:52041	http://NoteDoFi:18181/TravelRes...	

```
POST http://NoteDoFi:18181/TravelReservationService/buildItinerary HT
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
User-Agent: Jakarta Commons-HttpClient/3.1
Host: NoteDoFi:18181
Proxy-Connection: Keep-Alive
Content-Length: 8034

<SOAP-ENV:Envelope  xsi:schemaLocation="http://schemas.xmlsoap.o
  <SOAP-ENV:Body>
    <TravelItinerary xmlns="http://www.opentravel.org/OTA/2003/05">
      <ItineraryRef Type="14" ID="M839LW">
        <UniqueID>M839LWHH</UniqueID>
      </ItineraryRef >
```

```
POST http://NoteDoFi:18181/TravelReservationService/buildItinerary HT
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
User-Agent: Jakarta Commons-HttpClient/3.1
Host: NoteDoFi:18181
Proxy-Connection: Keep-Alive
Content-Length: 8034

<SOAP-ENV:Envelope  xsi:schemaLocation="http://schemas.xmlsoap.o
  <SOAP-ENV:Body>
    <TravelItinerary xmlns="http://www.opentravel.org/OTA/2003/05">
      <ItineraryRef Type="14" ID="M839LW">
        <UniqueID>M839LWHH</UniqueID>
      </ItineraryRef >
```

```
Script: Not Loaded   Proxy: Started
```

Figure 8: GUI after first SOAP message is intercepted. No script was loaded yet, therefore no faults were injected. Showing request message.



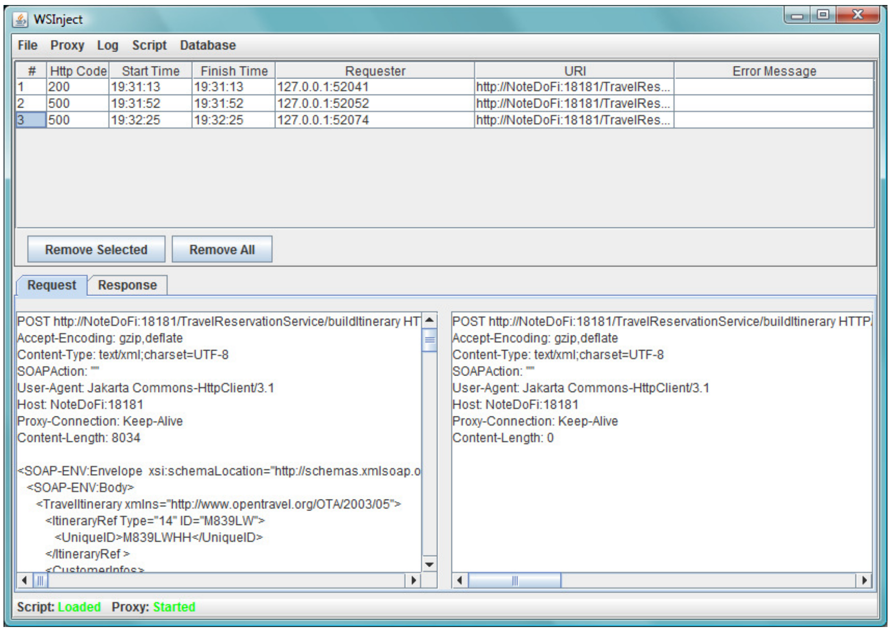
#	Http Code	Start Time	Finish Time	Requester	URI	Error Message
1	200	19:31:13	19:31:13	127.0.0.1:52041	http://NoteDoFi:18181/TravelRes...	
2	500	19:31:52	19:31:52	127.0.0.1:52052	http://NoteDoFi:18181/TravelRes...	

```
POST http://NoteDoFi:18181/TravelReservationService/buildItinerary HT
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
User-Agent: Jakarta Commons-HttpClient/3.1
Host: NoteDoFi:18181
Proxy-Connection: Keep-Alive
Content-Length: 8034

<SOAP-ENV:Envelope  xsi:schemaLocation="http://schemas.xmlsoap.o
  <SOAP-ENV:Body>
    <TravelItinerary xmlns="http://www.opentravel.org/OTA/2003/05">
      <ItineraryRef Type="14" ID="M839LW">
        <UniqueID>M839LWHH</UniqueID>
      </ItineraryRef >
      <CustomerInfos>
```

```
POST http://NoteDoFi:18181/TravelReservationService/buildItinerary HT
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
User-Agent: Jakarta Commons-HttpClient/3.1
Host: NoteDoFi:18181
Proxy-Connection: Keep-Alive
Content-Length: 8114

<#####@@@?!$%@@@#####-ENV:Envelope  xsi:schemaLocation="h
  <#####@@@?!$%@@@#####-ENV:Body>
    <TravelItinerary xmlns="http://www.opentravel.org/OTA/2003/05">
      <ItineraryRef Type="14" ID="M839LW">
        <UniqueID>M839LWHH</UniqueID>
      </ItineraryRef >
      <CustomerInfos>
```

Script: Loaded   Proxy: Started

Figure 9: GUI after a script with a StringCorruptionFault is loaded and second SOAP message is intercepted. Showing request message. Notice the corruption of text containing the string "SOAP".



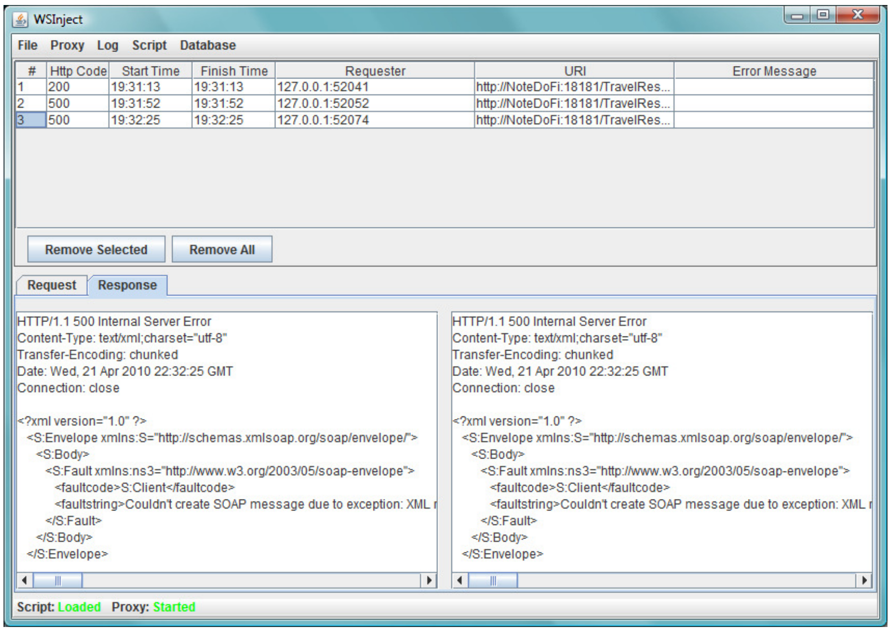
#	Http Code	Start Time	Finish Time	Requester	URI	Error Message
1	200	19:31:13	19:31:13	127.0.0.1:52041	http://NoteDoFi:18181/TravelRes...	
2	500	19:31:52	19:31:52	127.0.0.1:52052	http://NoteDoFi:18181/TravelRes...	
3	500	19:32:25	19:32:25	127.0.0.1:52074	http://NoteDoFi:18181/TravelRes...	

```
POST http://NoteDoFi:18181/TravelReservationService/buildItinerary HT
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
User-Agent: Jakarta Commons-HttpClient/3.1
Host: NoteDoFi:18181
Proxy-Connection: Keep-Alive
Content-Length: 8034

<SOAP-ENV:Envelope  xsi:schemaLocation="http://schemas.xmlsoap.o
  <SOAP-ENV:Body>
    <TravelItinerary xmlns="http://www.opentravel.org/OTA/2003/05">
      <ItineraryRef Type="14" ID="M839LW">
        <UniqueID>M839LWHH</UniqueID>
      </ItineraryRef >
      <CustomerInfos>
```

```
POST http://NoteDoFi:18181/TravelReservationService/buildItinerary HTTP/
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
User-Agent: Jakarta Commons-HttpClient/3.1
Host: NoteDoFi:18181
Proxy-Connection: Keep-Alive
Content-Length: 0
```

Script: Loaded   Proxy: Started

Figure 10: GUI after a script with an EmptyingFault is loaded and third message is intercepted. Showing the <u>request</u> message.



#	Http Code	Start Time	Finish Time	Requester	URI	Error Message
1	200	19:31:13	19:31:13	127.0.0.1:52041	http://NoteDoFi:18181/TravelRes...	
2	500	19:31:52	19:31:52	127.0.0.1:52052	http://NoteDoFi:18181/TravelRes...	
3	500	19:32:25	19:32:25	127.0.0.1:52074	http://NoteDoFi:18181/TravelRes...	

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml;charset="utf-8"
Transfer-Encoding: chunked
Date: Wed, 21 Apr 2010 22:32:25 GMT
Connection: close

<?xml version="1.0" ?>
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <S:Fault xmlns:ns3="http://www.w3.org/2003/05/soap-envelope">
        <faultcode>S:Client</faultcode>
        <faultstring>Couldn't create SOAP message due to exception: XML r
      </S:Fault>
    </S:Body>
  </S:Envelope>
```

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml;charset="utf-8"
Transfer-Encoding: chunked
Date: Wed, 21 Apr 2010 22:32:25 GMT
Connection: close

<?xml version="1.0" ?>
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <S:Fault xmlns:ns3="http://www.w3.org/2003/05/soap-envelope">
        <faultcode>S:Client</faultcode>
        <faultstring>Couldn't create SOAP message due to exception: XML r
      </S:Fault>
    </S:Body>
  </S:Envelope>
```

Figure 11: GUI showing the response for the third message.

3.6 Monitor Data Manager

The Monitor Data Manager is responsible for storing and retrieving data about messages intercepted by the Proxy/Monitor and also the log of WSInject. This component works with a local Apache Derby database[8]. If no existing database is found when the tool is started, one is automatically created.

4. Experiments

We have conducted preliminary experiments to verify WSInject's ability to inject faults into a composition of Web Services. For these initial experiments, only corruption faults were tested. Our testbed was based on free tools available for the Java community, so the experiments could be made as reproducible as possible.

4.1. Platform Description

An applications server is a piece of software that allows deployment of web applications and components, such as Web Services. Once deployed, Web Services are operational and available for users to call them. On our experiments, we used the GlassFish

[8] http://db.apache.org/derby/

v2.1 application server[9]. Web Services can be deployed directly on GlassFish. BPEL processes can be deployed on GlassFish after the installation of a BPEL engine. On our experiments, we used the OpenESB V2 BPEL engine[10]. We also used the NetBeans 6.5.1 IDE[11], which is able to manage GlassFish and OpenESB. NetBeans also includes useful samples of Web Services and SOA applications.

4.2. Test Setup and scenario

Our testbed was composed of WSInject, soapUI[12] (a Web Services testing tool, used to generate SOAP requests), GlassFish (the web application server), BPEL process (the descriptor of Web Services composition) and the partner Web Services (which implement individual system features). We tested the Travel Reservation Service (TRS) from Netbeans 6.5.1 SOA examples. TRS is a Web Services composition that simulates a real-life organization that manages airline, hotel and vehicle reservations by calling partner Web Services. These partner Web Services are *VehicleReservationService*, *AirlineReservationService* and *HotelReservationService*. They are orchestrated by a BPEL process to build a complete travel itinerary.

In our experiments, soapUI was the client of TRS. It was responsible for originating requests that activated the BPEL process, which in turn called its partners services to make reservations. soapUI and GlassFish were configured to make connections through WSInject proxy. All comunication between client, BPEL process and partner services was intercepted by WSInject, which was able to inject faults into any SOAP message exchanged in our test setup. Figure 12 depicts the complete test setup and scenario.

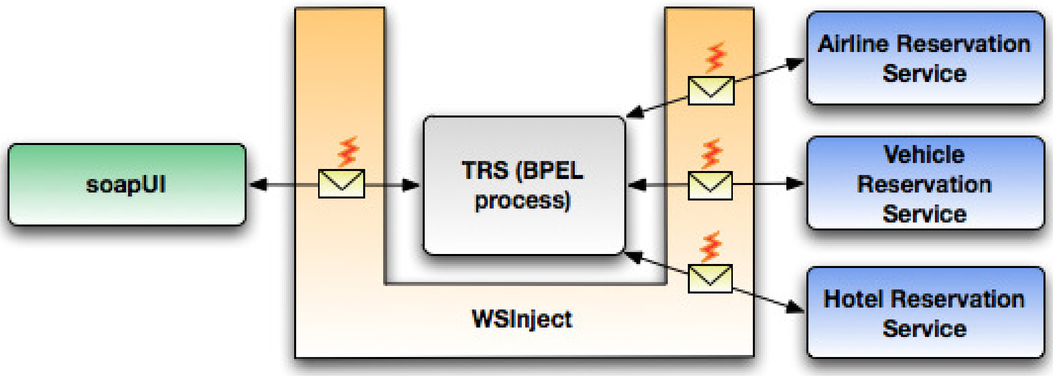

soapUI
TRS (BPEL process)
WSInject
Airline Reservation Service
Vehicle Reservation Service
Hotel Reservation Service

Figure 12: Test setup and scenario for our experiments.

[9] https://glassfish.dev.java.net/

[10] https://open-esb.dev.java.net/

[11] http://www.netbeans.org/

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

Figure 13 shows a sequence diagram that describes how the Proxy/Monitor and Fault Injection Executor components act to indentify and inject faults on messages exchanged by soapUI, TRS and partner Web Services. All incoming and outcoming messages from TRS (1, 4, 7, 10) pass through the Proxy/Monitor. Before they are delivered to their destination, messages are sent to the Fault Injection Executor (2, 5, 8), which analyses them and injects desired faults on the appropriate ones. The rectangle in the middle highlights the sequence representing messages exchanged between TRS and partner services to generate the travel itinerary.

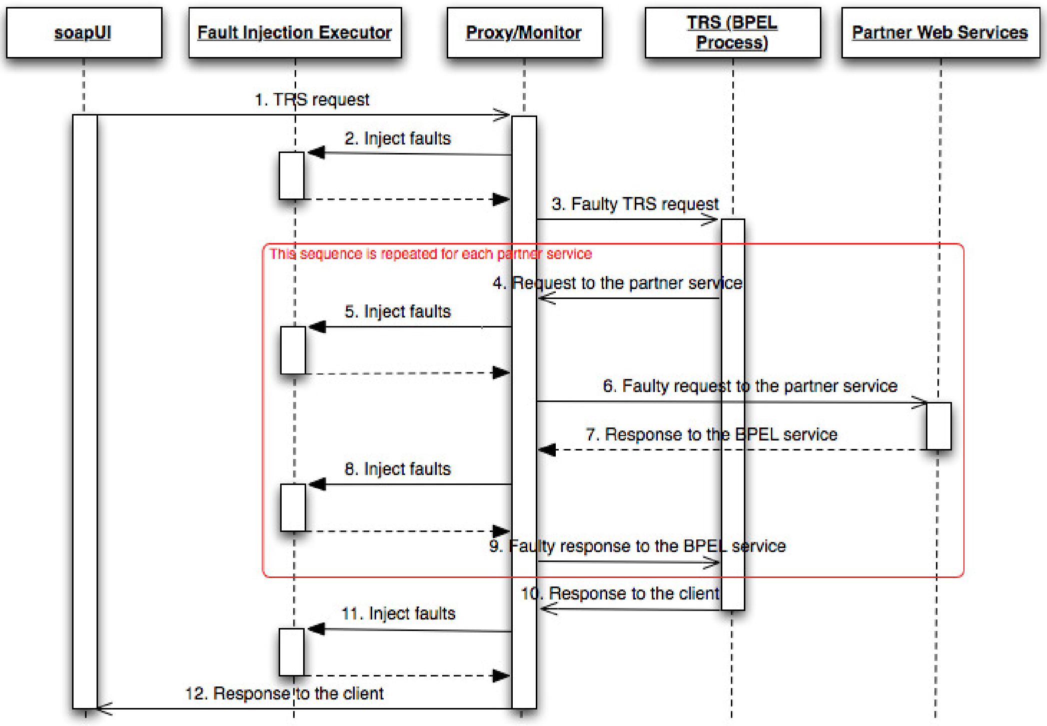

soapUI
Fault Injection Executor
Proxy/Monitor
TRS (BPEL Process)
Partner Web Services
1. TRS request
2. Inject faults
3. Faulty TRS request
This sequence is repeated for each partner service
4. Request to the partner service
5. Inject faults
6. Faulty request to the partner service
7. Response to the BPEL service
8. Inject faults
9. Faulty response to the BPEL service
10. Response to the client
11. Inject faults
12. Response to the client

Figure 13: Sequence diagram showing the interception and fault injection on each message passing through the Proxy/Monitor component of WSInject.

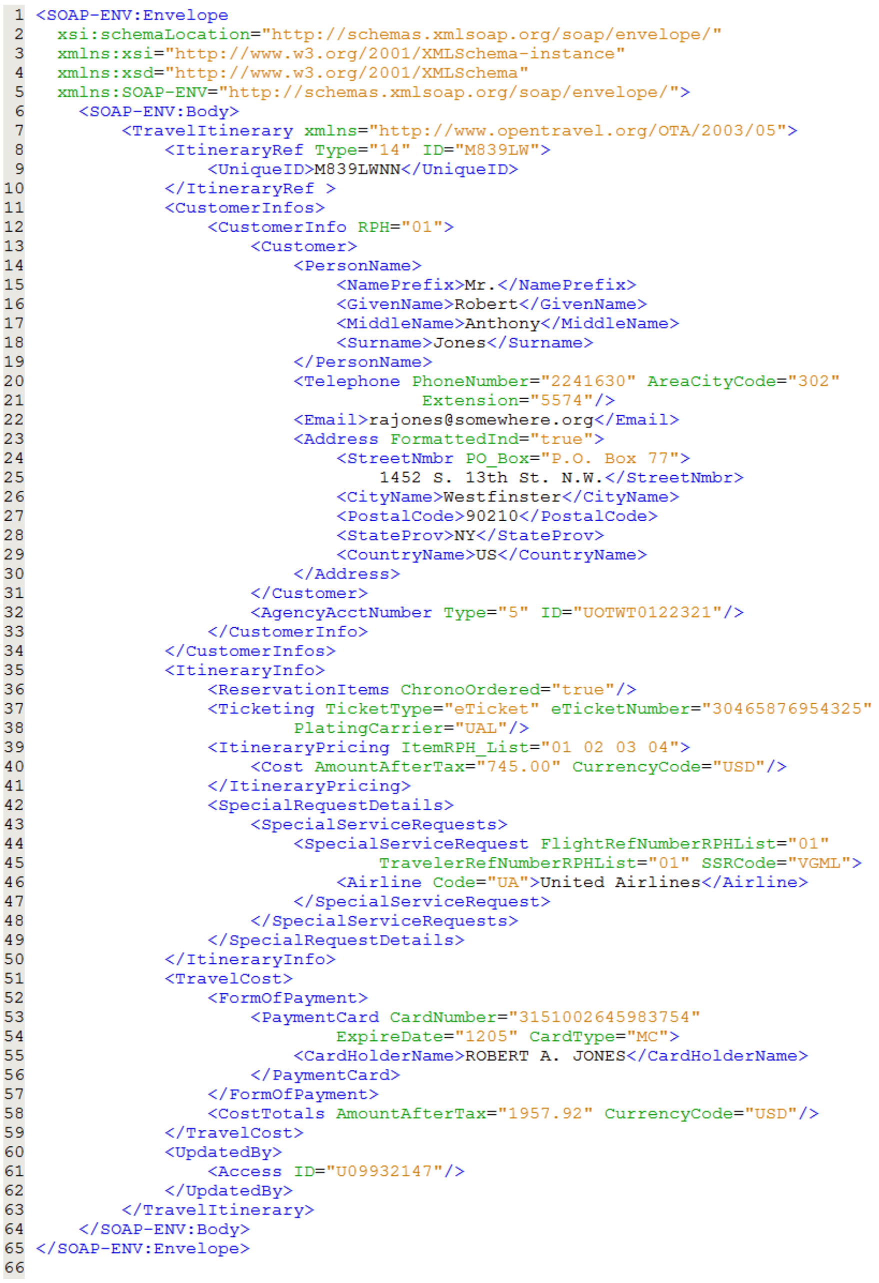
4.3. Workload and Faultload

Two parameter corruption experiments were conducted, consisted of injecting corruption faults on all SOAP messages exchanged on our test setup – both by the client and the BPEL process, and by the BPEL process and partner services. Integer and string values of the SOAP messages were corrupted, both from XML elements (like `<element>` **`1234`** `</element>`) and from XML attributes (like `<element attribute=`**`"1234"`** `/>`). These were the **integer corruption experiment** and **string corruption experiment**. Each of them was composed of many campaigns. Campaigns consisted of corrupting all integer or all string values from SOAP messages. Each of these campaigns corresponded to an entry from the list of integer and string corruption faults from Vieira *et al.* [VIEIRA 07], shown on Table C below.

Table C: Different types of corruption applied on experiments.

Type	Test Name	Parameter Mutation
String	StrNull	Replace by null value
	StrEmpty	Replace by empty string
	StrPredefined	Replace by predefined string
	StrNonPrintable	Replace by string with nonprintable characters
	StrAddNonPrintable	Add nonprintable characters to the string
	StrAlphaNumeric	Replace by alphanumeric string
	StrOverflow	Add characters to overflow max size
Number	NumNull	Replace by null value
	NumEmpty	Replace by empty value
	NumAbsoluteMinusOne	Replace by -1
	NumAbsoluteOne	Replace by 1
	NumAbsoluteZero	Replace by 0
	NumAddOne	Add one
	NumSubtractOne	Subtract 1
	NumMax	Replace by maximum number valid for the type
	NumMin	Replace by minimum number valid for the type
	NumMaxPlusOne	Replace by maximum number valid for the type plus one
	MumMinMinusOne	Replace by minimum number valid for the type minus one
	NumMaxRange	Replace by maximum value valid for the parameter
	NumMinRange	Replace by minimum value valid for the parameter
	NumMaxRangePlusOne	Replace by maximum value valid for the parameter plus one
	NumMinRangeMinusOne	Replace by minimum value valid for the parameter minus one

TRS system comes with predefined test cases on NetBeans – *hasAirline*, *hasHotel*, *hasVehicle* and *hasNoReservations*. These are functional tests to verify the correct behavior of the system. The SOAP request from the *hasNoReservations* test case (also named *TestCase1* on some versions of NetBeans) was used to activate the TRS on all fault injection experiments. Figure 14 shows the *hasNoReservations* test case request message.



```
<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>

```

Figure 14: Original request message from the hasNoReservations test case.

For the parameter corruption experiments, the faultload consisted of 22 different fault injection campaigns, 7 for the string experiment and 15 for the integer experiment. For each campaign, a single SOAP request message (depicted on Figure 14) was sent. The workload also consisted of 22 messages, 7 for the string experiment and 15 for the integer experiment. An initial execution of each experiment was made, followed by two repeated executions. One fault injection script was created for each campaign. The execution of each campaign consisted of manually loading a script into WSInject and a sending a request message from soapUI, which activated the message interception and fault injection process.

Besides the experiments mentioned above, other ones using EmptyingFault, MultiplicationFault and DelayFault were conducted. Each one was executed 8 times – 2 for TRS calls and 2 for calls to each partner service (AirlineReservationService, VehicleReservationService and HotelReservationService). Finally, a structure corruption experiment consisted of using StringCorruptionFaults to invert opening and closing XML tags on SOAP messages. This structure corruption experiment was executed 10 times.

4.4. Results

Corruption of integer values produced normal responses from the Web Services composition. This was not the expected result, since invalid data were sent to the service composition, *e.g.*, value -1 for credit card expiration date. TRS should have detected these incorrect values, and it didn't. This demonstrates that faults have gone unnoticed, meaning they would probably be propagated to a database in a real system. Considering the wsAS scale, this can be classified as a *Silent* failure.

Corruption of string values produced *Silent* failures as well, since using empty, random and alphanumeric strings produced no error messages to the client. Also, when a request was sent with empty strings, the connection was kept open during a long period of time and no response was sent to the client. Injecting non-printable characters produced errors in the XML parser and an error code was returned to the client.

For all injections of EmptyingFault between TRS and one of its partner services, the client received an HTTP 500 error code. This means that an internal server error occurred and the client was disconnected from the service (i.e., an *Abort* failure). Applying MultiplicationFaults to the whole SOAP message did not affect the TRS system: partner services considered only the first SOAP envelope and ignored the replicated ones. For message delaying, when we applied a DelayFault of more than 20 seconds on a request (such as the invocation of reserveVehicle provided by the VehicleReservationService), TRS hanged until the GlassFish server timeout was reached (2 minutes). On the other hand, when we delayed reservation confirmations (such as those returned by VehicleReservationService), GlassFish sent an error message indicating that there was an instantiation error when sending the cancellation message. This indicates the existence of a bug in the implementation of the cancellation process.

5. Conclusion and Future Work

We presented WSInject, a tool for fault injection on single and composed Web Services. Its major features are the ability to work with Web Services compositions, the script system and its proxy-based approach. On our preliminary experiments described on this report, WSInject was able to inject desired faults and cause system failures on an application based on Web Services composition.

For future work, further analysis on OSI model layers should be done. WSInject uses an HTTP proxy for intercepting SOAP messages. This has some implications. For instance, when a DelayFault is injected, a client will successfully connect to the proxy, and the proxy will take a long time to respond. This creates a different situation than trying to connect directly to an unresponsive service: the latter would not respond at all (not even at TCP level), while a proxy always responds at TCP level, even when injecting DelayFaults.

Another interesting feature to be added in the future is to allow users to select out-of-the-box experiments/campaigns from a predefined set of options, like the *string corruption experiment* and *integer corruption experiment* presented. Currently, user is required to create his/her scripts (one per campaign), load each of them manually into WSInject and run the Web Services client. Running a complete experiment may require various manual setups. There is already a preliminary alternative to this completely manual process, namely, the command-line interface mode. It was designed to allow the user to start the fault injection process by means of an external script – such as a shell script – and speed up this process. On the other hand, being able to configure full experiments directly on WSInject GUI would be more desirable.

6. References

1. **[LARANJ 08]** Laranjeiro, N., Canelas, S., and Vieira, M. 2008. wsrbench: An On-Line Tool for Robustness Benchmarking. In *Proceedings of the 2008 IEEE international Conference on Services Computing - Volume 2* (July 07 - 11, 2008). IEEE Computer Society, Washington, DC, 187-194. DOI= http://dx.doi.org/10.1109/SCC.2008.123

2. **[LEME 00]** Leme, N., Martins, E., and Rubira, C. 2000. Um Sistema de Padrões para Injeção de Falhas por Software. In *II Workshop de Testes e Tolerância a Falhas* (July 15 - 16, 2000). WTF. SBC - Sociedade Brasileira de Computação, Curitiba, PR, 100-105.

3. **[LOOKER 04]** Looker, N., Munro, M., and Xu, J. 2004. WS-FIT: A Tool for Dependability Analysis of Web Services. In *Proceedings of the 28th Annual international Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02* (September 28 - 30, 2004). COMPSAC. IEEE Computer Society, Washington, DC, 120-123.

4. **[KOOP 97]** Koopman, P., Sung, J., Dingman, C., Siewiorek, D., and Marz, T. 1997. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems* (October 22 - 24, 1997). SRDS. IEEE Computer Society, Washington, DC, 72.

5. **[REIN 08]** Reinecke, P., and Wolter, K. 2008. Towards a multi-level fault-injection test-bed for service-oriented architectures: Requirements for parameterisation. In *SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems* (October 5th, 2008). AMBER, Naples, Italy.

6. **[VIEIRA 07]** Vieira, M., Laranjeiro, N., and Madeira, H. 2007. Benchmarking the Robustness of Web Services. In *Proceedings of the 13th Pacific Rim international Symposium on Dependable Computing* (December 17 - 19, 2007). PRDC. IEEE Computer Society, Washington, DC, 322-329. DOI= http://dx.doi.org/10.1109/PRDC.2007.24

7. **[ZIMMER 80]** Zimmermann, H. 1980. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. In *IEEE Transactions on Communications*, vol. 28, no. 4 (April, 1980). IEEE Computer Society. 425 - 432.