Detailed Description of Communication Faultloads to Improve the Usability of Fault Injection Testing Tools

Ruthiano S. Munaretti[1], Taisy S. Weber[1], Sérgio L. Cechin[1]

Institute of Informatics - Federal University of Rio Grande do Sul
P.O. Box 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

Abstract Message lost or delay, link and computer crashes, as well as network partitioning can lead a system to an inconsistent behavior. The identification and the precise reproduction of these faults may be hard to be done, specially during the test of complex communication systems. Fault injectors are tools that introduces faults in an application under test according to a faultload previously described by a test engineer. This controlled way to inject faults allows to evaluate the system dependability. Unfortunately, each fault injector presents a different and sometimes hard to learn fault description format, which is one of the main impairments to an efficient use of these tools. In this paper we present an environment to overcome this problem, that translates a high level description into faultload formats for different fault injectors. Furthermore, the environment generates all the commands needed to initiate a test campaign, thus reducing user flaws and improving fault injector usability.

1 Introduction

Computational systems are an essential support in almost all human activities. In order to attend the imposed requirements, they became complex in size and functionalities. For these reasons, a minimum level of service quality is crucial but hard to obtain. Dependability is an important concept, which indicates the ability of a system to avoid failures that are more frequent and more severe than is acceptable [1].

Failure-free communication is one of the main requirements of large computer systems. In these systems, messages flow through a computer network, which is composed of *nodes* (computers, servers, switches and routers) and *paths* (that interconnects any two nodes of the network). The occurrence of faults is unavoidable. Node or path crash as well as delay, drop and duplication of messages are examples of communication faults. The system must implement some form of fault tolerance based on redundancy to avoid that one of these unavoidable faults leads to system failure.

While faults are unavoidable, the probability of their occurrence may be considered low. For this reason, in an operational environment, the identification and reproduction of faults is a hard task, specially in distributed systems. Thus, the use of fault injectors permits the creation of complex fault scenarios to test the system. The injected faults are close to real ones and represent the most usual faults that can occur. Better than real faults, they can be introduced in a fully controlled way. Fault injection is specially useful to test the fault tolerance mechanisms built to improve the dependability of a system. If fault coverage is low, the system can suffer a failure. In this case, the injected fault permits to identify the failure mode of the system and estimate its robustness.

There are a significant amount of fault injectors reported in the literature [7]. However, from the point of view of test engineers, that are the users of fault injectors, the main issues are related to the classes of faults they can inject and to the formats available to describe faultloads. A faultload defines the faults that will be injected during the test campaign, as well as its configurations (moment of activation, duration, frequency of occurrence, distribution, location and many others). Appropriate faultloads must be provided by the user. Thus, the easiness to describe faultloads are directly related to the usability of the injector as a whole, influencing its effective use.

Unfortunately, it is still hard to describe a faultload with its detailed configuration for each specific fault injector. Thus fault injection may be completed dismissed during the validation phase of a system - a test engineer may prefer to use empirical techniques (for instance, an easy but potentially harmful disconnection of a network cable). Each injector demands a specific format that must be followed to describe a faultload. These formats vary from configuration files to script languages, from binary codes to XML. The time to learn each variation inhibits broader acceptance of fault injection and biases the usability of the tools. Moreover, if a test engineer wants to use two or more fault injectors to compare or complement the test results, he must write a specific faultload for each injector considering the details of each format and spending his precious time describing several times the same fault scenario.

Considering the above concerns, we propose an environment to describe faultloads at a high abstraction level. The environment interacts with the users producing faultloads for different communication fault injectors. Our approach targets fully functional systems and prototypes for testing them under fault conditions. The main goal is usability improvement through simplified and reusable descriptions of faultloads. For this reason, extensibility is a key point, in a way that new faultload formats for new fault injectors may be included, attending user needs. Finally, emphasis is given to the construction of complex faultloads, in order to support rich scenarios, such as sequence of faults, multiple faults and distributed faults.

For extensibility purposes we adopted a *framework* approach. In an extensible environment, unpredictable cases may be considered in a straightforward manner. Considering the environment development, we assume the following premises:

Focus on communication faults: emphasis is given to communication fault, because these are the main sources of failures regarding message-based systems.

Fault injector independence: a faultload must describe the fault scenario the test engineer wants and must be the same for any fault injector that can be used in a test campaign. So, an unique high-level faultload will be converted automatically to specific formats appropriated to each injector that can be used.

Specification of rich faultloads: the environment supports the specification of multiple faults in a faulload with different activation triggers and probabilities. In that way, several classes of faults may be defined to executed into a specific fault injector during the test campaign.

Extensibility/flexibility: extensibility is widely supported. The environment may be easily adopted to describe several different fault scenarios to different fault injectors, according the requirements of the test engineer.

The remainder of the paper is organized as follows. Section 2 describes related

works. Section 3 details a model of the environment, while section 4 explains its architecture. Section 5 shows examples of faultloads generated for different fault injectors. Finally, section 6 concludes the paper.

2 Related Work

Injectors use several formats and approaches to permit the user creating fault scenarios. This section describes fault injectors that realizes some effort to make this task easier. With this background, comparisons may be done between the existing approaches and the proposed environment.

Related to classical tools, DOCTOR [5] is an environment for evaluation of real time distributed systems, that allows injection of hardware and communication faults. ORCHESTRA [2] is a tool for testing network protocols, implemented through a framework that intercepts communication messages. NFTAPE [11] allows the creation of new fault injectors using components and interfaces provided by the tool, bringing portability and extensibility to the description of test scenarios.

Considering more recent tools, MENDOSUS [8] implements fault injection into emulated networks, where faults may be injected directly into the components of the network under test. FIRMAMENT [3] injects faults into messages passing through the protocol stack inside the operating system kernel, which allows a direct packet manipulation. FAIL/FCI [6] is a tool that evaluates cluster applications, where the construction of test scenarios is based on a state machine approach.

To compare the injectors above we analyze the following parameters, based on the desirable requirements to faultload description, as well as on previous approaches [12]: *expressiveness* (how similar to real and complex fault scenarios the described faultloads can be), *difficulty* (how difficult is the creation of a faultload for a given injector) and *usability* (including issues like user documentation, availability of the tool and extensibility).

Starting with *expressiveness*, we consider that NFTAPE and DOCTOR present *low* expressiveness because both offer only few mechanisms for faultload description - only configuration files are accepted by them. MENDOSUS and ORCHESTRA expressiveness are *medium*. They provide configuration scripts that helps the specification of faultloads, but with limited fault models. FAIL/FCI and FIRMAMENT use scripts that specify rich sets of faultloads - for this reason, we consider that both present *high* expressiveness.

Following with *difficulty* and *usability*, FAIL/FCI has *low* difficulty and *medium* usability - its faultload description is intuitive but specific to grid environments. MENDOSUS presents *medium* difficulty and usability, also from its specific focus (related to emulated networks), as well as from the limited fault model used. FIRMAMENT, NFTAPE, ORCHESTRA and DOCTOR have both *high* difficulty and *low* usability, because the hard curve of learning (considering FIRMAMENT) and few available documentation and information (all other injectors). None of the analyzed injectors show *high* usability. Table 1 summarizes these comparisons.

Table1. Faultload description comparison

	Expressiveness	Difficulty	Usability
FAIL/FCI	High	Low	Medium
FIRMAMENT	High	High	Low
MENDOSUS	Medium	Medium	Medium
NFTAPE	Low	High	Low
ORCHESTRA	Medium	High	Low
DOCTOR	Low	High	Low

3 Model of the Environment

The environment model is based on *modularity* (independent units) and *extensibility* (for modifications without side effects). It provides a conceptual base for the architecture of the environment, turning its conception more precise. This model, defined previously [9], is divided into *elements* (figure 1). The fault injectors themselves are not components of the environment. They will be used later during the test campaigns. In the following paragraphs, we describe each element.

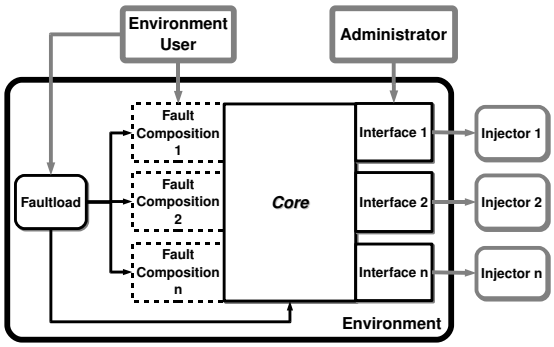

Figure1. Environment model

Core: Includes operations supported by the model to define the *type* of injected faults. The core is composed by: *Node*, *Path*, *Packet* and *Network*. In the core, one or more *primitive faults* may be selected to be injected later during test runtime.

Fault Composition: Defines a set of primitive faults, that may be used as a single fault. It allows the creation of new fault types, based on primitive faults already available on the Core. Fault composition is optional (as dotted lines in figure 1).

Interface: Realizes a mapping between a faultload created by a user, and a specific faultload (related to an injector and following the syntax and semantic defined by this injector). To manage this task, we define another type of user, called Administrator. The main advantage of this approach is the separation of concerns: while issues of

configuration are delegated to special people, a test engineer may focus his attention to fault injection, improving the quality of the test campaign.

Faultload: Through faultloads, the environment users (the test engineers) describe which faults will be injected later in the test campaign, as well as their configurations.

From this point onwards we start presenting the new contributions of this paper. The current work defines a Java subset language, as well as additional libraries, for faultload description. In this subset, all the known constructions and operations used in Java are allowed, among other additional features. Table 2 describes the main functionalities supported in this subset. The advantage of using a Java subset instead another faultload description language are easy to see - it eliminates the learning curve.

Table2. Main functionalities of the environment's language

Variable Declarations	Permits the declaration of variables of several types (such as *numeric/alfanumeric*). Syntax: `<type> <name> [=<value>];`
Conditional Statements	Defined through an *if* command (such as Java language). Syntax: `if (<expr>) { <commands> }`
Loop Statements	Used for operations that must be its execution repeated, according to an expression. Syntax: `while|for (<expr>) { <commands> }`
Topology Components	Support of additional classes, related to the topology previously defined. Syntax: `Node|Path|Packet <name> [=<value>];`

4 Architecture

In order to assert an appropriate level of usability [4, 10], a computational environment must be easy to use. At the same time, this environment must be also powerful enough to allow several levels of operation from final users. Finally, extensibility and modularity are also important requirements to consider in developing this kind of system. In this context, the environment proposed in this paper has a modular architecture. The approach used here is based on the model described in the previous section, with emphasis on extensibility. Figure 2 presents a general diagram of the architecture.

With this general architecture in mind, the following items explains the internal function of each component as well as the interaction between them.

Input represents the main input of the environment and is responsible for its configuration and execution. This input encompasses three phases: *(1) Faultload* specifies the faults that will be injected in a test campaign; *(2) Compiler Call* contains the set of calls needed to convert the general faultload into specific faultloads for each fault injector used, and *(3) Injector Call* provides all the mechanisms for an appropriate execution of fault injectors during the test campaign.

Topology allows the specification of a network topology that will be used in a test campaign. The main components in a topology are *nodes* and *paths*.

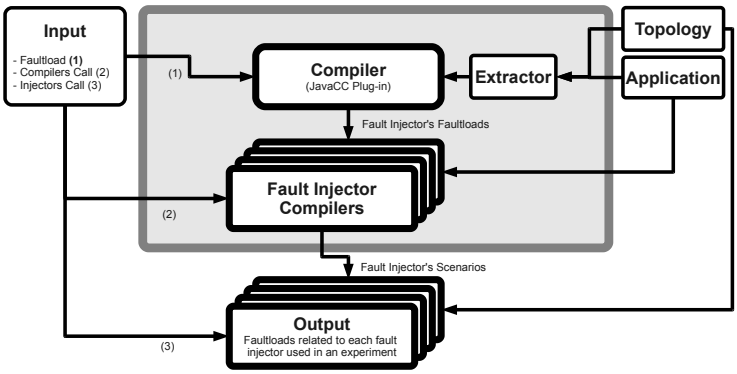

Figure2. Environment architecture

Application indicates the target application that will be tested in a test campaign. The target application is indicated by: *(1) Call Name* (name of the application); *(2) Configuration Parameters* (all the required parameters to execute the target application); and *(3) Topology* (identification of where the target application will be executed, considering the *topology* defined for the test campaign).

Extractor is responsible to *extract* relevant information from the *Topology* and *Application* components. This extra information is used to complete the generation of faultloads, as well as calls for compilers and fault injectors (such as the *name* of each component in a given topology, as well as their *types*.

Compiler converts the high-level faultload described by the user using the Java subset language to a faultload format closer to the descriptions used by the fault injectors. The Intermediate Language used to represent these faultloads is simpler and less structured than the Java subset language described previously, following the main purpose to gain performance on the compilation process. This language has only a Symbol Table (for data storage) and six commands: INSERT (adds a value into the Symbol Table), RUN (runs a specified target application), RAND (generates a random number, based on given seed), EVAL (evaluates a given expression), GOTO (goes to a given line into the code), DROP (drops a given packet) and DELAY (delays a given packet).

Fault Injector Compilers translate the faultloads previously created to each specific faultload format needed. For each fault injector available to be used must exist a compiler in the environment. New compilers for new fault injector can be created by the *administrator*. The main goal is to realize the mapping between the functionalities provided by the environment and that of the fault injectors. The translation uses mapping and triggering scripts. The mapping script has the following syntax: "Intermediate Language Command" : "Fault Injector Command", where the first one represents a command of the intermediate language, while the second one indicates what command or sequence of commands of the fault injector is associated with it. The triggering scripts are used for extra features provided by fault injectors, that is not directly treated by the environment. It is possible to define triggers in the following points: *before faultload* (for initialization purposes), *before line* (considering a line of

the intermediate language), *after line* (activated at the end of the current line) and *after faultload*.

Output consists of faultloads and commands that will be used to activate a fault injection test campaign. A complete fault scenario related to each fault injector, the activation commands for a fault injection campaign and also information about topology, when needed for a specific fault injector, are the output of the environment. This fault scenario embraces not only the faultload, but all configuration parameters required for the execution of a fault injector in a test campaign. Following this phase, the test engineer can finally start a test campaign.

It is worth to mention that the proposed environment does not replace the fault injection environment where the test campaign will finally run. It complements the last one allowing to describe the desired faultload at a high abstraction level and independently from a given fault injector format. It also allows that the same faultload description can be translated to formats appropriated for different fault injectors without the need to learn their specific formats and configuration details. One can argue that the administrator must still know the fault injector details to developed the needed fault injector compiler. It is true, but the administrator just do that for a new fault injector the test engineer wants to use and just one time. The description of faultloads is instead a more frequent task; tests must be repeated several times with different faultloads and workloads, and a significative amount of different faultloads is needed to cover the most common fault scenarios associated to real networks.

5 Case Example

This section describes the generation of faultloads using the proposed environment. In this execution, from a general faultload we generate three different faultloads for three fault injectors that use different and incompatible formats: FAIL/FCI, MENDOSUS and FIRMAMENT. A format accepted as correct by MENDOSUS for example is not recognized as valid by FIRMAMENT. The main goal here is to illustrate the concepts explained in the previous sections. Table 3 illustrates an example of a faultload described with the input language that was defined for the environment, a Java subset language. In this faultload, a crash will be injected during a test campaign with a probability of 5% (represented from the 0.05 number). This high level description is converted to an intermediate language closer to that used by the injectors.

Table3. A sample of high-level faultload, followed by the representation in intermediate language

Java Faultload	Intermediate Language

`import env.Node;`
 	
`public class Client {`
`  public static void main(String[] args) {`
`  Node n = new Node("n1");`
`  n.setApp("/opt/ClientProgram");`
`  n.crash(0.05);`
`  }`
`}`
	`1 INSERT n "n1"`
	`2 INSERT app "/opt/ClientProgram"`
	`3 INSERT crash "95"`
	`4 INSERT total "100"`
	`5 RUN app`
	`6 RAND value total`
	`7 EVAL "value - total < 0" ? 8 ! 6`
	`8 DROP "n1"`
	`9 GOTO 6`

A set of triggers for each fault injector used is defined, as well as the mapping scripts that will be used on the compilation phase. Table 4 summarizes these scripts, with one example for each fault injector.

Table4. Triggers and mapping scripts related to fault injectors

	Triggers	Mapping Scripts
FAIL/FCI	BeforeFaultload : "Daemon **$x**Logic {" BeforeLine : "node **$x**" AfterLine : ", goto **$x;**" AfterFaultload : "}"	INSERT(x,y) : "int **$x $y**" RUN(x) : "Computer **$x** { program = '**$x**'; daemon = '**$x**Logic'; }" RAND(x,y) : "FAIL_RANDOM(**$x, $y**)" EVAL(x?t!f) : "**$x** -> **$t**" DROP(x) : "stop"
MENDOSUS	BeforeFaultload : "set_host **$x $y**" BeforeLine : "" AfterLine : "" AfterFaultload : ""	INSERT(x,y) : "" RUN(x) : "set_command **$i** '**$x**' EXEC **$i $x**" RAND(x,y) : "poisson **$x**" EVAL(x?t!f) : "" DROP(x) : "set_fault **$x** fail_host_power_off TRANSIENT"
FIRMAMENT	BeforeFaultload : "MAIN:" BeforeLine : "" AfterLine : "" AfterFaultload : "END:"	INSERT(x,y) : "SET **$y $x**" RUN(x) : "" RAND(x,y) : "RND **$y $x**" EVAL(x?t!f) : "JMP **$x $t**" DROP(x) : "DRP"

Finally, the environment translate the faultload to appropriated formats using the representation in intermediate language, the triggers and the mapping scripts. Table 5 illustrates the description of a crash fault, that occurs with the probability of 5% in three different but equivalent formats. Each format is syntatic and semantic well-formed for a specific injector: FAIL/FCI, MENDOSUS or FIRMAMENT. This example shows that even considering simple faultloads, the formats accepted by each injector differ highly, as well as the semantic associated with fault types into each fault injector.

Receiving the faultloads generated by the environment (table 5), the test enginner can now start test campaigns with one of more fault injectors or he can describe and generated new faultloads using the environment. Comparing the input description of 3 with the faultloads of (table 5) we can see that even for this simple example the proposed solution allows for better readability of faultloads.

6 Final Remarks

This paper presents an environment for detailed description of communication faultloads. We detail its model, as well as its architecture and the internal components. Finally we show an example of faultload translation for three different fault injectors.

Considering usability issues, the environment proposes a language for faultload description at a high abstraction level and a method for translating it to formats that the injectors accept. Related to requirements mentioned in section 2, the environment has *high* expressiveness, *low* difficulty and *high* usability. These are possible using simplified high level descriptions, which involves few language constructions and, at the same time, allows to describe powerful and detailed fault scenarios, as illustrated in section 5.

Other issues related to usability refers to the target public - although usability is generally related to final users, developers and system architects must also be considered to improve their levels of productivity. The environment is developed with this idea

Table5. Specification of a faultload (crash with 5% probability) into fault injectors

FAIL/FCI	`Daemon ClientLogic {`
	`  node 1: int prob = FAIL_RANDOM(1,100);`
	`  prob <= 5 -> stop, goto 2;`
	`  node 2:`
	`}`
	`Computer Client {`
	`  program = "/opt/ClientProgram";`
	`  daemon = "ClientLogic";`
	`}`

MENDOSUS	`// Configuration of topology:`
	`set_host Client 10.0.0.1`
	`// Target system commands:`
	`set_command 0 "/opt/ClientProgram"`
	`EXEC 0 Client`
	`// Faults that will be injected:`
	`set_fault Client ft_host_power_off`
	`TRANSIENT poisson 0.05`

FIRMAMENT	`; R0 = 100; // 100%`
	`SET  100 R0`
	`; R1 = RND(R0)`
	`; (sorts a number, with seed 100)`
	`RND  R0  R1`
	`; R0 = 95`
	`; (looking for 5% of probability)`
	`SET  95  R0`
	`; R0 = R0 - R1;`
	`SUB  R1  R0`
	`; if R0 is negative, drops packet (crash)`
	`JMPN R0  DROP`
	`DROP: DRP`

in mind - all descriptions are at a high abstraction level, such as in programming languages. Besides this, the environment separates concerns affecting configuration and description, which organizes all the parameters and commands related to a test campaign.

To achieve better usability, the environment attends properly the usability metrics previously defined [4, 10]. The language proposed to describe faultloads is both *easy to learn* and *easy to remember* through simplified scripts that use constructions commonly found in programming languages. This simplicity also leads to *efficient* and *relatively error-free* faultload description, as showed in the previous section. Finally, considering the target public (developers, test engineers and system architects), the environment is *pleasant to use* because it does not impose another fault description language but uses constructions next to the world of developers.

In comparison to similar approaches (described in section 2), the environment implements a more detailed fault model than MENDOSUS and FAIL/FCI. Another feature the environment offers is the independence of a specific network infrastructure or system type, differently of the fault model of DOCTOR (with focus on real-time distributed systems) and of ORCHESTRA (that focuses on network protocols). Finally, the environment uses a simplified approach in all of its constructions, feature not found in the faultload description formats defined by FIRMAMENT and NFTAPE.

References

1. A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. In *IEEE Transactions on Dependable and Secure Computing*, volume 1, pages 11–33, 2004.
2. S. Dawson, F. Jahanian, and T. Mitton. Fault Injection Experiments on Real-Time Protocols using ORCHESTRA. In *High-Assurance Systems Engineering Workshop*, pages 142–149. IEEE Proceedings, 1996.
3. R. J. Drebes. FIRMAMENT: Um Módulo de Injeção de Falhas de Comunicação para Linux. Master's thesis, Federal University of Rio Grande do Sul, Porto Alegre, 2005.
4. X. Ferré, N. Juristo, H. Windl, and L. Constantine. Usability basics for software developers. *IEEE Software*, pages 22–29, 2001.
5. S. Han, K. Shin, and H. Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems. In *Int. Computer Performance and Dependability Symposium (IPDS'95)*, pages 204–213, Erlangen, Germany, 1995. IEEE Computer Society Press.
6. W. Hoarau and S. Tixeuil. A Language-Driven Tool for Fault Injection in Distributed Systems. In *Proc. of the 6th IEEE/ACM Intl. Workshop on Grid Computing*, pages 194–201, Grand Large, France, 2005.
7. Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
8. X. Li, R. Martin, K. Nagaraja, T. D. Nguyen, and B. Zhang. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, 2002.
9. R. S. Munaretti and T. S. Weber. Modelo de um Ambiente para Descrição de Cenários Detalhados de Falhas. In SBC, editor, *IX Workshop de Testes e Tolerância a Falhas - WTF2008*, pages 71–84, Rio de Janeiro, RJ, 2008.
10. J. Nielsen. Iterative user-interface design. *IEEE Computer*, pages 32–41, 1993.
11. D. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: a Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91–100. IEEE Proceedings, 2000.
12. S. Tixeuil, W. Hoarau, and L. Silva. An Overview of Existing Tools for Fault Injection and Dependability Benchmarking in Grids. In *Second CoreGRID Workshop on GRID and Peer to Peer Systems Architecture*, Paris, France, 2006. LIAFA.