Fault Injection in Distributed Java Applications

William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles
LRI – CNRS UMR 8623 & INRIA Grand Large
{hoarau,tixeuil}@lri.fr

Abstract

In a network consisting of several thousands computers, the occurrence of faults is unavoidable. Being able to test the behaviour of a distributed program in an environment where we can control the faults (such as the crash of a process) is an important feature that matters in the deployment of reliable programs.

In this paper, we investigate the possibility of injecting software faults in distributed java applications. Our scheme is by extending the FAIL-FCI software, and does not require any modification of the source code of the application under test, while retaining the possibility to write high level fault scenarios. As a proof of concept, we use our tool to test FreePastry, an existing java implementation of a Distributed Hash Table (DHT), against node failures.

Résumé

Dans un réseau constitué de plusieurs milliers d'ordinateurs, l'apparition de fautes est inévitable. Etre capable de tester le comportement d'un programme distribué dans un environnement où on peut contrôler les fautes (comme le crash d'un processus) est une caractéristique importante pour le déploiement de programmes fiables.

Dans cet article, nous étudions la possibilité d'injecter des fautes de manière logicielle dans des application Java réparties. Notre approche est basée sur le logiciel FAIL-FCI, et ne requiert pas de modification de l'application sous test, tout en conservant la possibilité d'écrire des scénarios de fautes de haut niveau. Comme preuve de faisabilité, nous utilisons notre outil pour tester FreePastry, une implantation Java d'une Table de Hashage Distribuée (DHT), en présence de défaillance de nœuds.

1. Introduction

In a network including several thousands machines, the appearance of faults is unavoidable. Some applications (for example peer to peer applications) involve a considerable number of users, *e.g.* to exchange files or to execute long calculations (SeTi@Home, Decrypthon, Xtremweb, Boinc, etc.). For those applications, the appearance and disappearance of participating machines are unpredictable, very frequent and occur eventually while the application is run.

It is particularly difficult to study the functioning of large-scale distributed programs: it would be necessary to have a considerable number of computers and engineering power to execute the software in an actual situation, to measure the performances or to detect the defects. With the difficulty to set up such experiments and the fact that fault occurrences in such systems is not neither controllable nor predictable (it is also difficult to compare various solutions), two other approaches are possible: simulation and emulation. Simulation allows complete control of the runtime environment, but fails in imitating the actual behavior of all components in the system. Emulation consists in using a small network to reproduce the behavior of a large-scale network. However, it is not enough to emulate the machines used by the participants: it is also necessary to reproduce their behavior.

Testing the validity of fault-tolerant software and measuring the impact on performance of occurring faults requires being able to control those faults. Indeed, a fundamental result [FLP85] shows that in an asynchronous distributed system (where the relative speeds of the processors are *not* known and unbounded), it is impossible to solve the consensus problem (all processors terminate agreeing on some initial value) when there is as little as one faulty process, even when the considered fault is as simple as a crash fault. The reason for this is that the decided value can depend on just one process and that in an asynchronous system, it is impossible to distinguish between a crashed process and a very slow one. When an application is run on a cluster, it is likely that machines will run roughly at the same speed (for example a one to ten ratio on the relative speeds of the processors makes it easy to solve the consensus problem), so the considered system is actually synchronous. Afterwards, when the application is then run at a larger scale (*e.g.* in an Internet-like setting) where the strong synchrony hypothesis does not hold any more, crucial issues related to fault-tolerance and asynchronous settings have been overlooked.

2. Related Works

When considering solutions for software fault injection in distributed systems, there are several important parameters to consider. The main criterion is the usability of the fault injection platform. If it is more difficult to write fault scenarios than to actually write the tested applications, those fault scenarios are likely to be dropped from the set of performed tests. The issues in testing component-based distributed systems have already been described and methodology for testing components and systems has already been proposed [Gosh, Rifle]. However, testing for fault tolerance remains a challenging issue. Indeed, in available systems, the fault-recovery code is rarely executed in the test-bed as faults rarely get triggered. As the ability of a system to perform well in the presence of faults depends on the correctness of the fault-recovery code, it is mandatory to actually test this code. Testing based on fault-injection can be used to test for fault-tolerance by injecting faults into a system under test and observing its behavior. The most obvious point is that simple tests (*e.g.* every few minutes or so, a randomly chosen machine crashes) should be simple to write and deploy. On the other hand, it should be possible to inject faults for very specific cases (*e.g.* in a particular global state of the application), even if it requires a better understanding of the tested application. Also, decoupling the fault injection platform from the tested application is a desirable property, as different groups can concentrate on different aspects of fault-tolerance.

Decoupling requires that no source code modification of the tested application should be necessary to inject faults. Also, having experts in fault-tolerance test particular scenarios for application they have no knowledge of favors describing fault scenarios using a high-level language, that abstract practical issues such that communications and scheduling. Finally, to properly evaluate a distributed application in the context of faults, the impact of the fault injection platform should be kept low, even if the number of machines is high. Of course, the impact is doomed to increase with the complexity of the fault scenario, *e.g.* when every action of every processor is likely to trigger a fault action, injecting those faults will induce an overhead that is certainly not negligible.

Several fault injectors for distributed systems already exist. Some of them are dedicated to distributed real-time systems such as DOCTOR [DOCTOR]. ORCHESTRA [ORCHESTRA] is a fault injection tool that allows the user to test the reliability and the liveliness of distributed protocols. ORCHESTRA is a "*Message-level fault injector*" because a fault injection layer is inserted between two layers in the protocol stack. This kind of fault injector allows injecting faults without requiring the modification of the protocol source code. However, the expressiveness of the faults scenario is limited because there is no communication between the various state machines executed on every node. Then, as the faults injection is based on exchanged messages, the knowledge of the type and the size of these messages is required. Nevertheless, those approaches do not fit the cluster and Grid category of applications.

The NFTAPE project [NFTAPE] arose from the double observation that no tool is sufficient to inject all fault models and that it is difficult to port a particular tool to different systems. Although NFTAPE is modular and very portable, the choice of a completely centralized decision process makes it very intrusive (its execution strongly perturbs the system being tested). Finally, writing a scenario quickly becomes complex because of the centralized nature of the decisions during the tests when they imply numerous nodes.

LOKI [LOKI] is a fault injector dedicated to distributed systems. It is based on a partial view of the global state of the distributed system. An analysis *a posteriori* is executed at the end of the test to infer a global schedule from the various partial views and then verify if faults were correctly injected (*i.e.* according to the planned scenario). However, LOKI requires the modification of the source code of the tested application. Furthermore, faults scenario are only based on the global state of the system and it is difficult (if not impossible) to specify more complex faults scenario (for example injecting "cascading" faults). Also, LOKI there is no support for randomized fault injection.

In [Mendosus] is presented Mendosus, a fault-injection tool for system-area networks that is based on the emulation of clusters of computers and different network configurations.

Finally in [OGSA] is presented a fault-injection tool that was specially developed to assess the dependability of Grid (OGSA) middleware. However, the tool described in that paper is very limited since it only allows the injection of faults in the XML messages in the OGSA middleware, which seems to be a bit far from the real faults experienced in real systems.

Recently, the FAIL-FCI architecture [FAIL] was proposed. This solution addresses most of the drawbacks of previous approaches, and is overviewed in the next section.

3. FAIL-FCI Overview

In this section, we describe the FAIL-FCI framework that is presented in [FAIL]. For further explanations, please refer to the original paper. First, FAIL (for FAult Injection Language) is a language that permits to easily described fault scenarios. Second, FCI (for FAIL Cluster Implementation) is a distributed fault injection platform whose input language for describing fault scenarios is FAIL.

The FAIL language allows defining fault scenarios. A scenario describes, using a high-level abstract language, state machines which model fault occurrences. The FAIL language also describes the association between these state machines and a computer (or a group of computers) in the network.

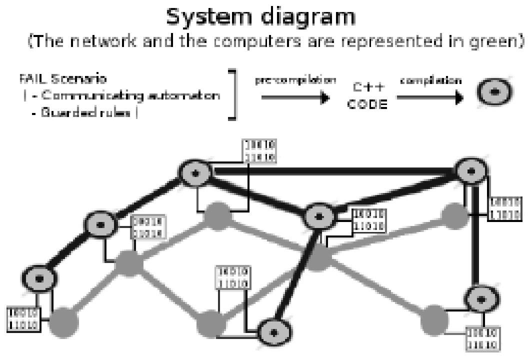
System diagram
(The network and the computers are represented in green)
FAIL Scenario
| - Communicating automaton
- Guarded rules |
pre-compilation
C++
CODE
compilation

Figure 1: the FCI Platform

The FCI platform (see Figure 1) is composed of several building blocks:

The FCI compiler: The fault scenarios written in FAIL are pre-compiled by the FCI compiler which generates C++ source files and default configuration files.

The FCI library: The files generated by the FCI compiler are bundled with the FCI library into several archives, and then distributed across the network to the target machines according to the user-defined configuration files. Both the FCI compiler generated files and the FCI library files are provided as source code archives, to enable support for heterogeneous clusters.

The FCI daemon: The source files that have been distributed to the target machines are then extracted and compiled to generate specific executable files for every computer in the system. Those executables are referred to as the FCI daemons. When the experiment begins, the distributed application to be tested is executed through the FCI daemon installed on every computer, to allow its instrumentation and its handling according to the fault scenario.

The FAIL-FCI approach is based on the use of a software debugger. Like the Mantis parallel debugger [Mantis], FCI communicates to and from `gdb` (the Free Software Foundation's portable sequential debugging environment) through Unix pipes. But contrary to Mantis approach, communications with the debugger are kept to a minimum to guarantee low overhead of the fault injection platform (in our approach, the debugger is only used to trigger and inject software faults).

The tested application can be interrupted when it calls a particular function or upon executing a particular line of its source code. Its execution can be resumed depending on the considered fault scenario.

With FCI, every physical machine is associated to a fault injection daemon. The fault scenario is described in a high-level language and compiled to obtain a C++ code which will be distributed on the machines participating to the experiment. This C++ code is compiled on every machine to generate the fault injection daemon. Once this preliminary task has been performed, the experience is then ready to be launched. The daemon associated to a particular computer consists in:

1. a state machine implementing the fault scenario,

2. a module for communicating with the other daemons (*e.g.* to inject faults based on a global state of the system),

3. a module for time-management (*e.g.* to allow time-based fault injection),

4. a module to instrument the tested application (by driving the debugger), and

5. a module for managing events (to trigger faults).

FCI is thus a Debugger-based Fault Injector because the injection of faults and the instrumentation of the tested application is made using a debugger. This makes it possible not to have to modify the source code of the tested application, while enabling the possibility of injecting arbitrary faults (modification of the program counter or the local variables to simulate a buffer overflow attack, etc.). From the user point of view, it is sufficient to specify a fault scenario written in FAIL to define an experiment (See subsequent section). The source code of the fault injection daemons is automatically generated. These daemons communicate between them explicitly according to the user-defined scenario. This allows the injection of faults based either on a global state of the system or on more complex mechanisms involving several machines (*e.g.* a cascading fault injection). In addition, the fully distributed architecture of the FCI daemons makes it scalable, which is necessary in the context of emulating large-scale distributed systems. FCI daemons have two operating modes: a random mode and a deterministic mode. These two modes allow fault injection based on a probabilistic fault scenario (for the first case) or based on a deterministic and reproducible fault scenario (for the second case).

4. The New FAIL-FCI Library Internal Structure

The previous implementation of the FAIL-FCI library was monolithic and hardly extensible [FAIL]. Also, it could only handle native programs. If distributed Java applications were to be tested, the scenarios would have to be written to target the Java virtual machine (on which Java programmers usually have little knowledge of the internals) instead of the Java program (on which control and knowledge is usually greater).

As a result, we restructured the core of the library to make it more modular and extensible, and to facilitate the handling of programming languages that do *not* generate native programs. The new internal structure of the FAIL-FCI library can be divided in three main parts: the *kernel*, the *external interface*, and the *network structure description*.

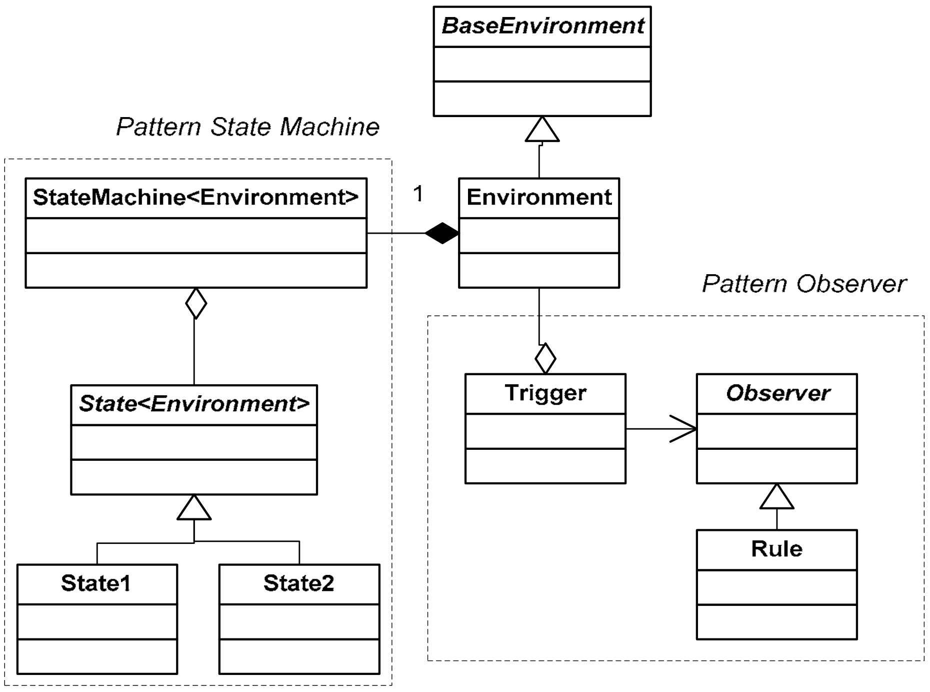

Figure 2: Kernel Architecture

The kernel architecture is described in Figure 2. The main class is *Environment*, that takes inputs from the *Trigger* class, and that controls the *StateMachine* class. The *Trigger* class models events that are likely to occur during the execution of the distributed applications (breakpoint reached in one of the program, message received from another machine, etc.). The *StateMachine* class refers to the automata that describes the actions that are taken by the FAIL-FCI deamon when some events are triggered. The relation between events (the *Trigger*) and actions (the *StateMachine*) is carried out by the *Environment*.

The actual actions that are taken by the *Environment* are handled by three different classes (see Figure 3). The *TimerController* class permits to setup timeout events, the *TimerController* permits to manage communication between FAIL-FCI deamons, and the *ProgramController* handles the program under test through a debugger. The *ProgramControllerGDB* permits to manage native programs through the *gdb* debugger, while the *ProgramControllerJDB* takes care of Java programs through the Java debugger. Further extensions for other languages and systems are straightforward with the new library structure.

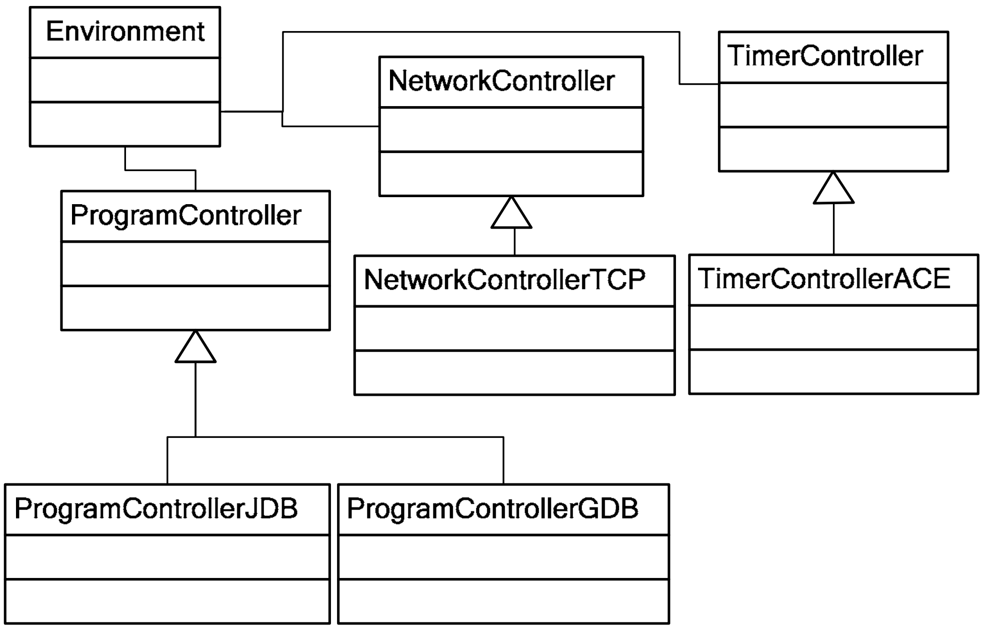

Figure 3: The External Interface

The network structure is generated by the FAIL compiler and resides in a XML file. This XML file permits to address individual machines as well as group of machines (those are handled through the *Node* and *GroupNode* classes, see Figure 4). The automata that drive the distributed components of the applications can have states that include integer or floating point variables. In order to properly handle the initialization of those variables, the *Value* class hierarchy has been added.

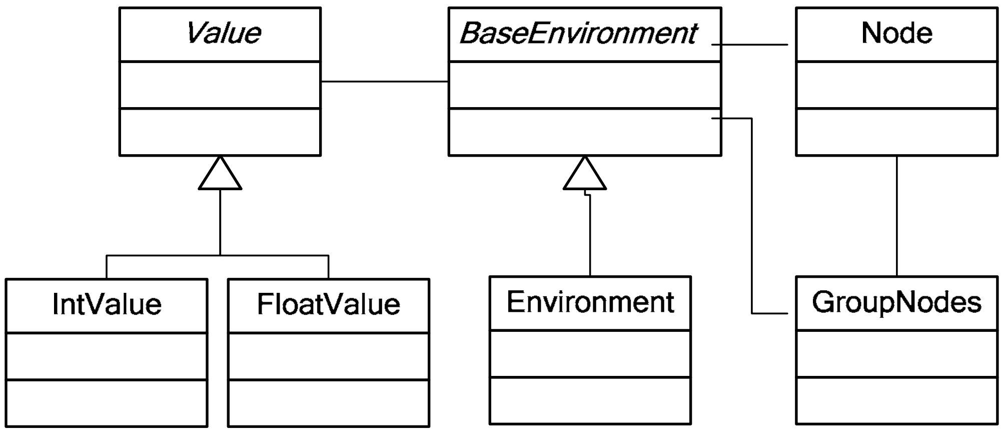

Figure 4: Network Structure

5. Preliminary Results

Pastry [Pastry] is a generic, scalable and efficient substrate for peer-to-peer applications. Pastry nodes form a decentralized, self-organizing and fault-tolerant overlay network within the Internet. Pastry provides efficient request routing, deterministic object location, and load balancing in an application-independent manner. Furthermore, Pastry provides mechanisms that support and facilitate application-specific object replication, caching, and fault recovery.

FreePastry [FreePastry] is an open-source implementation of Pastry intended for deployment in the Internet.

Using FAIL-FCI with our Java extension, we aim at testing the failure resilience capabilities of FreePastry. In particular, we wish to gradually inject more faults to evaluate the failure point of the FreePastry implementation, according to various criteria. Unlike previous approaches [PastryFT] that were conducted through *simulations*, our approach uses the *actual distributed java application*.

For this purpose, we write using FAIL a generic scenario, where each node has probability *x* to crash every *y* seconds. The FAIL source code is as follows:

```
spyfunc main;

Daemon ADV1 {
node 1:
      before(main) -> !start(G1),
            continue, goto 2;

node 2:
}

Daemon ADV2 {
node 1:
      before(main) -> stop, goto 2;

node 2:
      ?start -> continue, goto 3;

node 3:
      always time_g timer = 5;
      always int random =
            FAIL_RANDOM(0,100);
      timer && random <= 10 ->
            halt, goto 4;
      timer && random > 10 ->
            continue, goto 3;
node 4:
}

Computer P1 {
  program = "dummy";
  daemon = ADV1;
}

Group G1 {
  size = 30;
  program = "-classpath FreePastry-1.4.1.jar: DHT_Simple.jar
DistSimple 5000 lri7-209 5000";
  daemon = ADV2;
}
```

We now informally describe the aforementioned source code. First, two automata are defined: *ADV1* and *ADV2*, then automata *ADV1* is associated to one computer *P1* (that will execute dummy code), while *ADV2* is associated to 30 machines (that form the *G1* group), each executing a *.jar* file with the same parameters. *ADV2* runs as follows: the deamon first wait that the program has loaded, but before the *main* function is executed, the program is halted. The execution continues when the *ADV1* automata sends the 'start' message. Then, every five seconds, a timer event is set up. When the timer expires, with 10% probability, the process under test crashes, while with 90% probability, the process continues its computation. Further details about the FAIL language can be found in [FAILWeb].

Experimental Setting.

For our preliminary experiments, the tests were led on machines performing under Linux 2.6.7. Six machines were equipped each with a 2083 MHz processor and 885 Mb RAM, Six machines were equipped each with two 1533 MHz processors and 885 Mb RAM. Seven machines were equipped each with a 1800 MHz processor and 504 Mb RAM. All machines were connected using a 100 Mbps Ethernet network. All machines were running the 1.4.1 version of FreePastry. All tests were conducted 100 times and the obtained results are averaged over all tests.

Influence of the Periodicity of Crashes.

In this test, we fix for each node the probability to actually crash to 5%. We vary the time between possible failures for all nodes from 1 second to 5 seconds. We ran the test on a network composed of 19 machines. In each case, the FreePastry network was able to reconfigure itself, so that killed nodes were properly removed from the distributed structure.

Influence of the Probability of Crashes.

In this test, we fix for each node the periodicity of possible crash to 10 seconds. We vary the probability of crash from 10% to 100% with an increment of 10%. We ran the test on a network composed of 19 machines. In each case up to 90% (included), the FreePastry network was able to reconfigure itself, so that killed nodes were properly removed from the distributed structure. At 100% probability, all nodes would crash so it not surprising that the FreePastry network could not handle this case.

Influence of the Number of Nodes.

In this test, we fix for each node the periodicity of possible crash to 10 seconds, and the crash probability to 50%. We vary the number of nodes (and thus machines) from 19 to 35. It turns out that up to 34 nodes, the Pastry network is able to reconfigure itself. However, at 35 nodes, in 12% of the computations, the FreePastry network was not able to reconfigure and ended up in creating at least two separate networks.

Overall, it turns out that FreePastry is generally able to handle the crash of participating processes, restructuring so that the Distributed Hash Table is still maintained in spite of induced dynamicity. However, in our setting, there seems to be a failure point when the number of possibly failing processes increases. We expect to further analyze this hypothesis using a larger number of machines.

6. Conclusion

We extended the FAIL-FCI platform to support distributed Java application in the same way as native applications were supported. As a result, our new tool permits to inject faults in the actual applications, according to high level scenarios, without having to modify the source code of the Java programs or the java virtual machine.

As a proof of feasibility, we used a widely available distributed Java application (FreePastry) and used FAIL to specify a generic fault scenario that was then automatically handled by the FAIL-FCI infrastructure. Preliminary results that we obtained show that the periodicity and the probability of the faults are irrelevant (*i.e.* The FreePastry network is able to recover in any case), but that their number does matter (when the number of possibly failing nodes augments, the network may be partitioned into several networks). Further studies are needed, using wider test beds and different kinds of distributed Java applications.

7. References

[DOCTOR] S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems, 1995.

[FAIL] W. Hoarau and Sébastien Tixeuil. A language-driven tool for fault injection in distributed applications. In Proceedings of the IEEE/ACM Workshop GRID 2005, Seattle, USA, November 2005.

[FAILWeb] http://www.lri.fr/~hoarau/fail.html

[FLP85] M. Fisher, N.A. Lynch, and M.J. Paterson. Impossibility of consensus with one faulty process. Journal of the ACM, 1985.

[FreePastry] http://freepastry.rice.edu

[Gosh] S. Ghosh and A. Mathur. Issues in testing distributed component-based systems, 1999.

[LOKI] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In In Proc. of the Int.Conf. on Dependable Systems and Networks, June 2000.

[MANTIS] S. Lumetta and D. Culler. The mantis parallel debugger. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 118–126, Philadelphia, Pennsylvania, May 1996.

[Mendosus] X. Li, R. Martin, K. Nagaraja, T. Nguyen, B.Zhang. "Mendosus: A SAN-based Fault-Injection Test-Bed for the Construction of Highly Network Services", Proc. 1st Workshop on Novel Use of System Area Networks (SAN-1), 2002.

[NFTAPE] D.T. Stott and al. 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, March 2000.

[OGSA] N. Looker, J.Xu. "Assessing the Dependability of OGSA Middleware by Fault-Injection", Proc. 22nd Int. Symposium on Reliable Distributed Systems, SRDS, 2003

[ORCHESTRA] S. Dawson, F. Jahanian, and T. Mitton. Orchestra: A fault injection environment for distributed systems. In In 26th International Symposium on Fault-Tolerant Computing (FTCS), pages 404–414, Sendai, Japan, June 1996.

[Pastry] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks", SIGOPS European Workshop, France, September, 2002.

[PastryFT] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach, "Security for structured peer-to-peer overlay networks". In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, MA, December 2002.

[Rifle] Henrique Madeira, Mario Zenha Rela, Francisco Moreira, and Joao Gabriel Silva. Rifle: A general purpose pin-level fault injector. In European Dependable Computing Conference, pages 199–216, 1994.

8. Acknowledgement

This work was supported in part by the FRAGILE project of the ACI "Sécurité et Informatique".