

A Fault-Injector Tool to Evaluate Failure Detectors in Grid-Services

Nuno Rodrigues¹, Décio Sousa¹, Luís Silva¹, Artur Andrzejak²

¹*CISUC - Centre for Informatics and Systems of the University of Coimbra
Polo II - Pinhal de Marrocos
3030-290 Coimbra, Portugal
luis@dei.uc.pt*

²*Zuse-Institute Berlin
Takustr. 7, 14195 Berlin, Germany
andrzejak@zib.de*



CoreGRID Technical Report
Number TR-0098

September 17, 2007

Institute on Architectural issues: scalability,
dependability, adaptability (SA)

CoreGRID - Network of Excellence
URL: <http://www.coregrid.net>

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-00426

A Fault-Injector Tool to Evaluate Failure Detectors in Grid-Services

Nuno Rodrigues¹, Décio Sousa¹, Luís Silva¹, Artur Andrzejak²

¹*CISUC - Centre for Informatics and Systems of the University of Coimbra
Polo II - Pinhal de Marrocos
3030-290 Coimbra, Portugal
luis@dei.uc.pt*

²*Zuse-Institute Berlin
Takustr. 7, 14195 Berlin, Germany
andrzejak@zib.de*

*CoreGRID TR-0098**

September 17, 2007

Abstract

In this paper we present a fault-injector tool, named JAFL (Java Fault Loader), which was developed with the target of testing the fault-tolerance mechanisms of Grid and Web applications. Along with the JAFL internals description, we will present some results collected from synthetic experiments where we used both our injector and fault detection mechanisms. With these results we expect to prove that our fault injection tool can be actively used to evaluate fault detection mechanisms.

* *This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).*

1 Introduction

Before the deployment of Grid applications or any Grid middleware it is necessary to be sure that the software is robust and reliable. The software modules that are most difficult to be tested are the ones related with failure-detection and recovery. To exercise these modules we propose the use of synthetic fault-injection. By using fault injection techniques we will be able to reproduce the occurrence of failures in a system and measure the latency and coverage of the built-in fault-detection mechanisms.

To reproduce valid failures, the fault injector tools must be developed based on today's most common failures. In surveys such as [6] or [7] we can see that the main causes of failure are divided in four big groups: software failures, operator errors, hardware failures and security violations. Since hardware failures and security violation are out of the scope of this paper, we will only discuss software failures and operator error issues, which, in their turn, account for 80% of system failures (software (40%), operator error (40%)).

After deeply analyzing these two groups (software and operator errors) we realized that operator errors occur mainly during system maintenance, software upgrades and system integration while the software failures normally occur due to system overload, resource exhaustion and complex fault-recovery routines. Since these are the most common causes of system failures, our work will be completely focused on them.

In this paper we present a fault-injector tool, named JAFL (Java Fault Loader) that was developed based on the previous issues and with the target of testing the fault-tolerance mechanisms of Grid and Web applications. Furthermore, we can say that this fault-injector was developed as an additional package of the QUAKE benchmarking tool [28].

In the next sections we will describe JAFL internals, we will expose some of the most common fault detection mechanisms and we will present some results that were collected from experiments where we used both our injector and some fault detection mechanisms. In the end of this paper we expect to prove that our fault injector can be helpful in the evaluation of the fault-detection mechanisms used in Web and Grid systems.

2 Fault Injector

There are three categories of fault injectors: hardware-implemented, simulator-based and software-implemented [1]. Our fault injector is included in the later category.

The most known software fault injection tools targeted for Grid systems are: Cecium [8], Doctor [9], Orchestra [10], NFTAPE [11], LOKI [12], Mendosus [13], FAIL-FCI [14] and OGSA [15]. Some of them are able to inject real failures in the system, applications or network while the others are simple simulator-based injectors.

While most of the software injection tools only consider the injection at low level (with the objective of emulating hardware failures such as processor or memory faults [2, 3]) or at software-level (with the objective of corrupting code or data [4, 5]), our fault-injector was based on a slightly different approach. Our goal is more targeted to the synthetic consumption of JVM and operating system resources and to the injection of human-operator errors. As we have said previously, these types of failures represent a considerable amount of failures that occur in real systems. Since our objective is to reproduce the most common failures, we developed our fault injector with the ability to inject the following faults: Memory consumption; CPU consumption; Thread consumption, Disk Storage consumption, I/O consumption, Database connections consumption, File handler consumption, Exception Throwing, Database Table Deletion and Database Table Lock. Later in this section we will explain each one of them.

2.1 Architecture

Our fault injector, as shown in Figure 1, is composed by 2 main modules: the Controller and the Injector.

The Controller is responsible for reading a configuration file where all the configurations for the next fault injection are stored. In this configuration file the user can define the fault load, the specific parameters for that fault load, the start time, the interval between injections and the end time (it can be infinite) of the next fault injection.

The Injector module has two main functions: receive the fault load parameters from the Controller module and deliver them to the respective fault load sub-module. There are ten sub-modules in the Injector module, each one representing one specific fault:

Memory Consumption: With this sub-module we are able to consume, along the time, a given amount of memory in the JVM. If we don't define an end point it will consume all the memory and throw an Out Of Memory Error.

CPU Consumption: By using various threads to do some hard calculations, this sub-module can increase substantially the CPU Load.

Thread Consumption: With this fault we are able to create various threads along the time and wait to see how the system responds.

Disk Storage Consumption: We can use this kind of fault to see how the system handles the lack of disk space. If we don't define an end point it will consume all the disk space.

I/O Consumption: This sub-module is responsible for doing read/write operations on our system. If we want to burst our system we can check our maximum read/write speed and configure the fault for those values.

Database Connections Consumption: With this fault we can consume various database connections and check whether our applications behave with the connection pool being filled up.

File Handler Consumption: By using this fault we can consume the file descriptors which are available for the user running the Java application.

Exception Throwing: This sub-module is able to throw some exceptions. We implemented this sub-module because after the occurrence of an exception the application can change its behavior.

Database Table Lock: With this sub-module we can emulate some database lock problems that occur, most of the times, during maintenance operations.

Database Table Deletion: This fault simulates a very common operator error which normally occurs when an erroneous backup is used to restore a database.

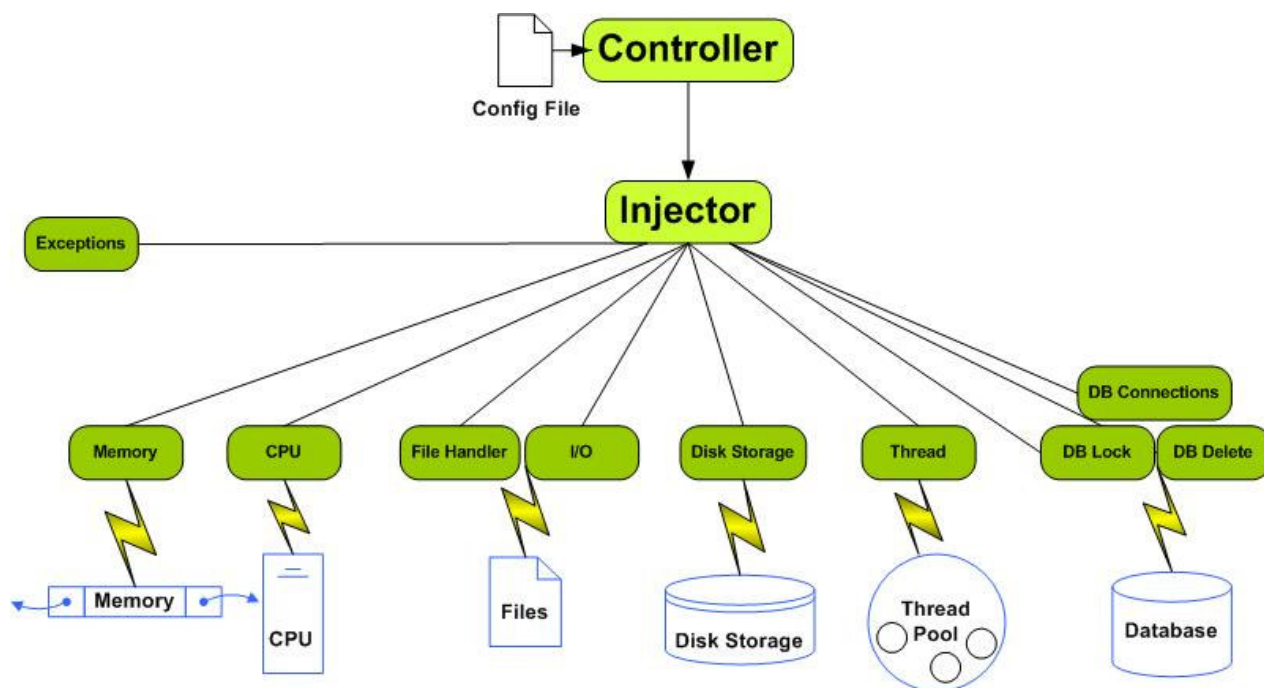


Figure 1 - JAFL Architecture

2.2 Usage

As JAFL is written in Java, it can be used in the following ways:

- Deployed in a Web/Grid container using the Java Technology
- Integrated with standalone applications
- Run as a standalone application

In Figure 2 we can see a sample scenario where JAFL can be used in the 3 ways. In the frontend server, JAFL is used to actively consume the Web/Grid container resources; in "node b" it is used to perturb a simple application running on that node and in "node e" it is running as a standalone application with the goal of consuming operating system resources. This sample scenario shows that JAFL can be very helpful to test the reliability of a Web/Grid application.

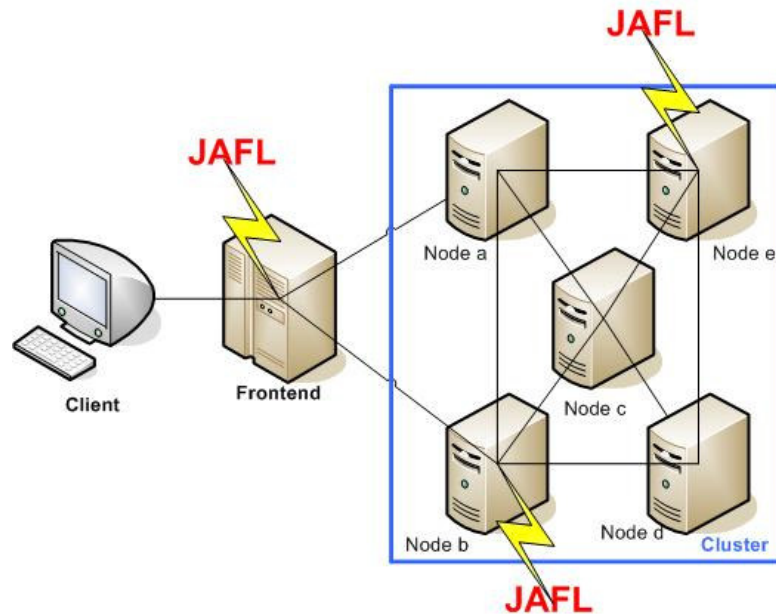


Figure 2 - JAFL Usage Scenario

3 Detectors

In the previous section we presented our fault injector tool. In this section we will present the failure detection mechanisms that we have chosen to detect the faults produced by the JAFL tool.

Fault-detection mechanisms are the responsible for measuring the health of an application. If they are accurate enough they can help the system operators to keep the system up and running. Otherwise, if they are not able to detect a failure and trigger an alarm, the system can become unavailable and probably nobody will notice that.

To guarantee good detection coverage we have chosen four types of failure detectors:

- System and Application Monitoring and Failure Detection Tools
- Component analyzers
- Log analyzers
- External Monitoring

Since each of these mechanisms has its own advantages and drawbacks, we will now describe them in detail.

3.1 System and Application Failure Detectors

This is the most common type of failure detectors. They are able to monitor the operating system resources (memory, CPU, etc), network interfaces, system services and others. With an accurate configuration they are also able to monitor applications and check the resources that they are consuming, their availability etc.

In the field of commercial solutions we have names such as HP Openview [18] or IBM Tivoli [19] to accomplish this task. Otherwise, if we prefer public domain solutions we can choose Zabbix [20], Nagios [21], Big Sister [22] and others.

In our study we adopted Zabbix, which is one of the powerful and easiest platforms to work with. It is very easy to deploy and to configure, provides a wide variety of features and stores all the data in a database, allowing external programs to collect data from there. It is composed by a Zabbix Server and Zabbix Agent. The Agent is deployed in the monitored host and sends all the information to the Server. The Server is then responsible for all the data analysis and alarm triggering.

3.2 Component Analyzers

One of the mostly known projects regarding component analysis is Pinpoint [16]. Pinpoint, developed in the University of Stanford, is a project which aim is to detect application-level failures in Web Applications by using path-analysis [17] and component interaction mechanisms. Having this project in mind, we instrumented a synthetic application (which will be used in our experiments) and we added a new detection layer to it. This detection layer was responsible for analyzing the components and detect the occurrence of Exceptions or high variations on the components execution latency. If any of these problems is detected, this detection layer will trigger an alarm and store a description of the problem on a specific database along with the respective timestamp.

The major drawback of our implementation is that it is application specific.

3.3 Log analyzers

Log analyzers are applications that parse log files and extract information from them.

Two of the mostly known standalone rule-based log analyzers are Swatch [23] and Logsurfer [24]. These log analyzers are commonly used to do real-time analysis of the log files and check if certain regular expressions are found there. If a certain regular expression, (e.g. an error message) is found, the tools will throw output events that can be converted in alarms.

In our experiments we decided to use a tool developed by ourselves but with the same design as the ones presented before. We developed a tool that simply sweeps the log files and tries to locate keywords that can indicate any kind of error.

If the tool detects keywords such as "Exception", "Error" or "OutOfMemory" an alarm is triggered and the description of the problem is stored in an external database with the respective timestamp.

3.4 Remote Monitoring Tools

The external monitors are agents that run outside of the Grid platform and that are able to behave like real users. They can visit webpages, do shopping and use services.

These external agents are able to detect DNS problems, TCP problems, HTTP errors, content matching errors and high response times. If any of these problems is detected, the agent will store the timestamp and all the available information about the problem in a database.

Once again, this was an implementation from ourselves which was based on external monitoring tools such as [29] and [30].

4 Experimental Results

In this section, we will present two of the many results collected from experiments run in the CISUC cluster in Coimbra. We used eight nodes of the Cluster and they were divided by: Main Server, Zabbix Server, External Agent and normal clients.

Our main server was configured to expose TPC-W in Java [25]. TPC-W is a synthetic application written in Java which is composed by a set of Servlets that simulate an e-business like Amazon.com. We deployed TPC-W over Apache Tomcat (with 1024 MB JVM) [26] and we used MySQL [27] for data storage. As we have said before, we added a failure detection layer to the TPC-W application with the goal of detecting component failures. In the same machine we configured our log analyzer tool, which was constantly analyzing the Apache Tomcat logs and we configured the Zabbix Agent which was responsible for analyzing the operating system resources and monitor the Tomcat container activity.

In another node we deployed the Zabbix Server and a MySQL database. This database was very useful to synchronize all the failure detection times since all the detection mechanism store the information about the failures in this database.

The other single node was configured as an external monitoring agent. This agent executes transactions in the TPC-W Web application and checks if errors appear during those transactions.

The remaining nodes were used to inject workload in the application. This workload consists in various transactions according to the TPC distribution.

All the internal detectors (Zabbix, Component, and Log) were configured with a 15 seconds polling interval while the external agent was configured with 1 minute between each transaction.

After configuring our experimental framework we started our experiments. We configured our fault injector to inject failures in the Tomcat container and we obtained very interesting results.

4.1 Memory Consumption

In this experiment we used our fault injector to simulate memory leaks in the Tomcat container. To achieve this, we configured our fault injector to consume 1 MB of memory per second, we defined 120 seconds as our ramp-up time and we did not defined any stoppage time. In Figure 3 we can see the detectors reaction to the fault.

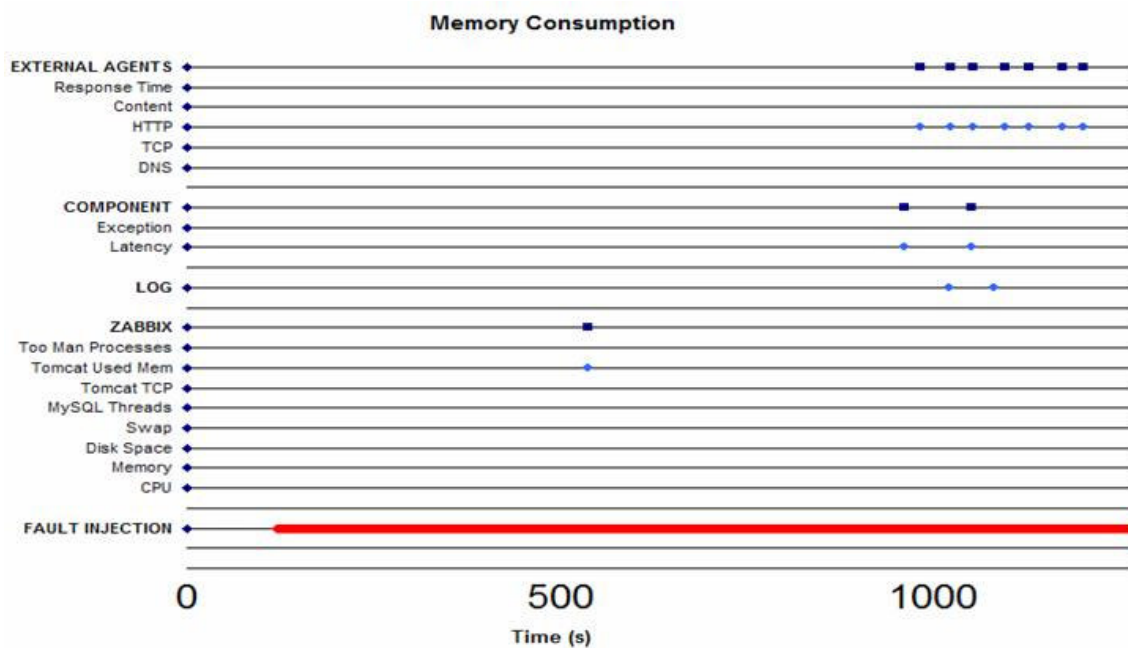


Figure 3 - Memory Consumption

By looking carefully at the Figure 3 we can see that the red line represents the injection period, the dark blue dots represent the mechanism that detected the failure and the light blue dots represent the activated trigger.

If we check the detection points along the time we can see that the Zabbix Agent spotted the problem at 540 seconds. Since Zabbix was configured to trigger an alarm when the JVM reaches 90% of its maximum heap space, this was as expected result. At this point, the JVM Memory is almost full and the application starts getting unstable. This instability is immediately noticed by the components when they understand that the request latencies are getting very high. After the components, also the external monitors detected the problem when they caught some HTTP timeout errors. By last, the Log analyzer tool detected an error in the log file. Therefore, we can say that the errors detected in the log files were OutOfMemory errors.

Since here, the service gets completely unavailable and only the external agents are able to detect the HTTP timeouts.

With this experiment we were able to test the capabilities of both our fault injector and our four detection mechanisms. Figure 3 can prove that both of them produced the expected results.

4.2 Table Deletion

With this experiment we have seen how the system handles a table deletion operator error and what detectors are able to detect this kind of problem. Since TPC-W uses various database tables, we configured our fault injector to delete each one of those tables from 10 to 10 seconds. The objective of this table deletion fault is to simulate an erroneous database backup.

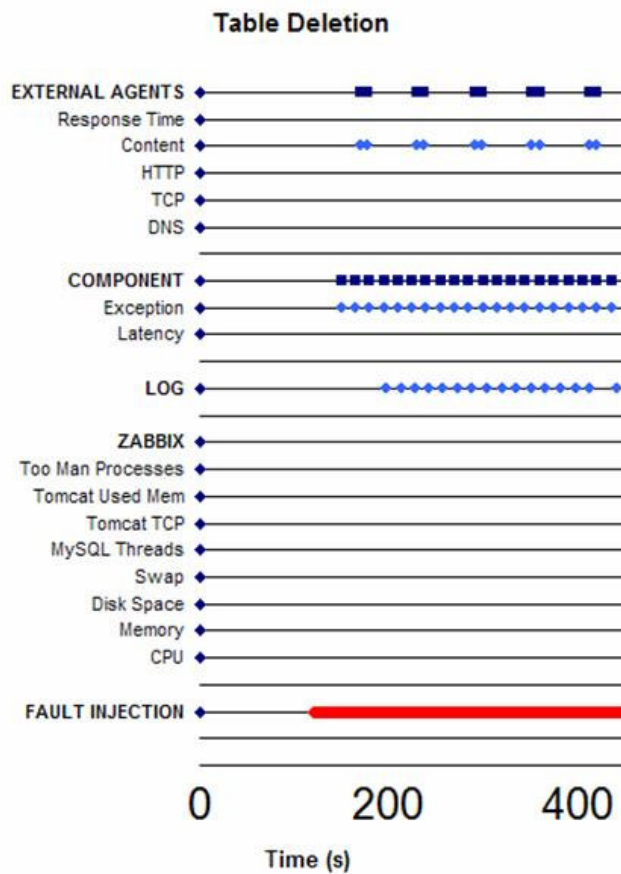


Figure 4 - Table Deletion

If we look at Figure 4 we can see that the component detector was the quickest to spot the failure. This time, instead of detecting high latency in the components execution, it has detected an exception at component level. Since Tomcat logs exceptions to log files, the Log Analyzer tool was also able to catch these exceptions; but lately.

Besides the component and log analyzers, also the external agents were able to catch this failure. They understood that the content of the page that they were requesting was different from the expected one. By analysing the content, they have seen that the exception

Once again our injector was able to accomplish its objective: triggering the detection systems.

5 Conclusions and Future Work

Using fault injectors to test the robustness of an application is one of the most common practices in the process of development. In this paper we presented a new fault injection tool, named JAFL, which is slightly different from the common injectors. JAFL was developed to consume operating system resources and simulate operator errors. After choosing some detection mechanisms, we have run various experiments where we used JAFL to inject failures in a synthetic application and we observed that most of those failures were detected by our detection mechanisms. This is a very positive result since our objective was to develop a tool capable of triggering the various fault detection mechanisms used in Web/Grid services. In the future we want to add more features to the JAFL tool and turn it available to the community. Our objective is to allow the community to use JAFL in their own applications.

References

- [1] H. Ammar, B. Cukic, C. Fuhrman, and A. Mili. A Comparative Analysis of Hardware and Software Fault Tolerance. *Annals of Software Engineering*, 10:103 - 150, 2000.
- [2] Aidemark, J.; Vinter, J; Folkesson, P.; Karlsson, J. "GOOFI - A Generic Fault Injection Tool". Proc DSN'01, Gothenburg, Sweden, 2001, pp. 83-88
- [3] Avresky, D.R.; Tapadiya P.K; "A method for Developing a Software-Based Fault Injection Tool"
- [4] Eliane Martins, Cecilia M.F. Rubira, Nelson G.M Leme, "Jaca: A reflective fault injection tool based on patterns". Proc DSN'02
- [5] Martins, E.; Rosa, A. "A Fault Injection Approach Based on Reflective Programming". Proc DSN'00
- [6] Soila Pertet and Priya Narasimhan. Causes of Failure in Web Applications. December 2005
- [7] David Openheimer, Archana Ganapathi and David A. Paterson. Why do Internet services fail, and what can be done about it? In 4th USENIX Symposium on Internet Technologies and Systems, 2003
- [8] G. Avarez and F. Cristian, "Centralized failure for distributed, fault-tolerant protocol testing," in Proceedings of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS'97) May 1997.
- [9] S. Han, K. Shin, and H. Rosenberg. "Doctor: An integrated software fault injection environment for distributed real-time systems", Proc. Computer Performance and Dependability Symposium, Erlangen, Germany, 1995.
- [10] S. Dawson, F. Jahanian, and T. Mitton. Orchestra: A fault injection environment for distributed systems. Proc. 26th International Symposium on Fault-Tolerant Computing (FTCS), pages 404-414, Sendai, Japan, June 1996.
- [11] 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.
- [12] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In Proc. of the Int.Conf. on Dependable Systems and Networks, June 2000.
- [13] 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
- [14] William Hoarau, and Sbastien Tixeuil. "A language-driven tool for fault injection in distributed applications". In Proceedings of the IEEE/ACMWorkshop GRID 2005, page to appear, Seattle, USA, November 2005.
- [15] N. Looker, J.Xu. "Assessing the Dependability of OGSA Middleware by Fault-Injection", Proc. 22nd Int. Symposium on Reliable Distributed Systems, SRDS, 2003
- [16] E. Kiciman and A. Fox. Detecting Application-Level Failures in Component-based Internet Services. *IEEE Transactions on Neural Networks*, Vol. 16, Issue 5, 2005
- [17] M.Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox and E. Brewer. Path-based failure and evolution management. In The 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04), San Francisco, CA, March 2004
- [18] HP Openview - www.managementsoftware.hp.com/,
- [19] Tivoli - <http://www-306.ibm.com/software/tivoli/products/monitor/>
- [20] Zabbix - <http://www.zabbix.org/>
- [21] Nagios - <http://nagios.org/>
- [22] Big Sister - <http://bigsister.graeff.com>
- [23] Swatch - <http://swatch.sourceforge.net>
- [24] Logsurfer - <http://www.dfn-cert.de/eng/logsurf/index.html>
- [25] TPC-W in Java - <http://www.ece.wisc.edu/~pharm/tpcw.shtml>
- [26] Apache Tomcat - <http://tomcat.apache.org/>
- [27] MySQL - <http://www.mysql.com/>
- [28] Quake Benchmarking tool - Presented in the CoreGRID Industrial Conference, CIC'2006
- [29] Site24x7 - <http://site24x7.com>
- [30] Gomez - <http://www.gomez.com>