Fault-Injection and Dependability Benchmarking for Grid Computing Middleware

Sébastien Tixeuil[1], William Hoarau[1], Luis Silva[2]

[1] LRI – CNRS UMR 8623 & INRIA Grand Large,
Université Paris Sud XI, France
Email: {tixeuil,hoarau}@lri.fr
[2] Departamento Engenharia Informática, Univ. Coimbra,
Polo II, 3030-Coimbra, Portugal
Email: luis@dei.uc.pt

Abstract. In this paper we review several existing tools for fault injection and dependability benchmarking in grids. We emphasis on the FAIL-FCI fault-injection software that has been developed in INRIA Grand Large, and a benchmark tool called QUAKE that has been developed in the University of Coimbra. We present the state-of-the-art and we explain the importance of these tools for dependability assessment of Grid-based applications and Grid middleware.

1 Introduction

One of the topics of paramount importance in the development of Grid middleware is the impact of faults since their probability of occurrence in a Grid infrastructure and in large-scale distributed system is actually very high. So it is mandatory that Grid middleware should be itself reliable and should provide a comprehensive support for fault-tolerance mechanisms, like failure-detection, checkpointing-recovery, replication, software rejuvenation, component-based reconfiguration, among others. One of the techniques to evaluate the effectiveness of those fault-tolerance mechanisms and the reliability level of the Grid middleware it to make use of some fault-injection tool and robustness tester to conduct some experimental assessment of the dependability metrics of the target system. In this paper, we will present and review several software fault-injection tools and workload generators for Grid Services that can be used for dependability benchmarking in Grid Computing.

The ultimate goal of our common work is to provide some contributions for the definition of a dependability-benchmark for Grid computing and to provide a set of tools and techniques that can be used by the developers of Grid middleware and Grid-based applications to conduct some dependability benchmarking of their systems.

Dependability benchmarking must provide a uniform, repeatable and cost-effective way of performing experiments in different systems, mechanisms or components. Those three metrics can be achieved with the development of software tools that will

be used in the process and in the definition of the dependability benchmark. According to [1] a dependability benchmark should provide the following components:

- **Workload**: which represents the work the system must do during the execution of the benchmark;
- **Faultload:** represents a set of faults and stressful conditions to emulate real faults that experienced in the real systems;
- **Measures**: characterize the performance and dependability of the system under benchmark in the presence of the faultload when executing the workload;
- **Experimental setup and benchmark procedure**: describes the setup required to run the benchmark and the set of rules that should be followed during the benchmark execution.

In this paper we present the two tools that have been developed by the two partners of WP4 from CoreGrid (INRIA and Univ. Coimbra) and we explain some of our ongoing projects with these tools.

2 Related Works

2.1 Fault-injection

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 [2,3]. 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. The following table captures the main differences between the main solutions for distributed fault injection relatively to those criteria.

Criteria	**ORCHESTRA [5]**	**NFTAPE [6]**	**LOKI [7]**	**FAIL-FCI [This paper]**
High Expressiveness	no	yes	no	yes
High-level Language	no	no	no	yes
No Source Code Modification	yes	no	no	yes
Scalability	yes	no	yes	yes
Probabilistic Scenario	yes	yes	no	yes
Global-state-based Injection	no	yes	yes	yes

2.2 Dependability benchmarking

The idea of dependability benchmarking is now a hot-topic of research and there are already several publications in the literature. The components of a dependability benchmark have been defined in [1]. In [18] is proposed a dependability benchmark for transactional systems (DBench-OLTP). Another dependability benchmark for transactional systems is proposed in [19]. Both benchmarks adopted the workload from the TPC-C performance benchmark. While [18] used software-based faults, the work described on [19] considered a fault-load based on hardware faults. The Special Interest Group on Dependability Benchmarking (SIGDeB), created by the IFIP WG 10.4 in 1999, released a preliminary proposal with a set of standardized classes for the classification of dependability in transactional database systems [20]. The goal is

to help out the comparison of computer systems concerning four different dimensions: availability, data integrity, disaster recovery and security.

A dependability benchmark for operating systems was proposed by [21]. That benchmark was targeted for the study of the operating system robustness in the scenario of faulty applications. Another study about the behavior of the operating system in the presence of software faults in OS components was presented in [22].

The research presented in [23] addresses the impact of human errors in system dependability. In [24] is presented a methodology to evaluate human-assisted failure-recovery tools and processes in server systems. That methodology was illustrated with a case study of undo/redo recovery tool in email services. Another work was presented in [25] that focus on the availability benchmarking of computer systems. The authors propose a methodology, including single and multi-fault workloads, and they applied that methodology to measure the availability of software RAID systems in different operating systems.

Research work at Sun Microsystems defined a high-level framework that is targeted to availability benchmarking [26]. That framework decomposes availability in three main components: fault-maintenance rate, robustness and recovery. Within the scope of that framework, they have developed two benchmarks: one that addresses specific aspects of a system's robustness on handling maintenance events such as the replacement of a failed hardware component or the installation of software patch [27]; and another benchmark that is related to system recovery [28].

At IBM, the Autonomic Computing initiative is also developing benchmarks to quantify the autonomic capability of a system [29]. In that paper they have discussed the requirements of benchmarks to assess the self-* properties of a system and they proposed a set of metrics for evaluation. In [30] is presented a further discussion about benchmarking the autonomic capabilities of a system. The authors present the main challenges and pitfalls. In [31] is presented an interesting approach to conduct benchmarking of the configuration complexity. This is a valuable contribution since one of the main problems of current IT systems is the complexity of deployment and management. A benchmark for assessing the self-healing capabilities of a system was presented in [32]. Two metrics were introduced: (a) the effectiveness of the system to heal itself in the occurrence of some perturbations; (b) a measure of how autonomic that healing action was achieved. This paper has clear connections with the work we are conducting in the study of self-healing techniques for SOAP-based servers.

In [33] the authors present a dependability benchmark for Web-Servers (Web-DB). This tool used the experimental setup, the workload and the performance measures specified in the SPECWeb99 performance benchmark. Web-DB defined measures in the baseline performance, in the performance in presence of faults and some dependability measures, like autonomy, availability and accuracy.

The dependability benchmark tool that is presented in this paper is targeted to Grid and Web-Services. We are mainly interested in the study of potential software aging problems and the effectiveness of self-healing techniques like software rejuvenation.

3 Our proposal

3.1 FAIL-FCI

FAIL-FCI [15] is a recently developed tool from INRIA. 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. Both components are developed as part of the Grid eXplorer project [10] which aims at emulating large-scale networks on smaller clusters or grids.

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. The FCI platform is composed of several building blocks:

1. **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.
2. **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.
3. **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 approach is based on the use of a software debugger. Like the Mantis parallel debugger [11], 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 must be 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. 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). Using a debugger to trigger faults also permits to limit the intrusion of the fault injector during the experiment. Indeed, the debugger places breakpoints which correspond to the user-defined fault scenario and then runs the tested application. As long as no breakpoint is reached, the application runs normally and the debugger remains inactive.

3.2 QUAKE: A Dependability Benchmark Tool for Grid Services

QUAKE is a dependability benchmark tool for Grid and Web-Services. The following sub-sections present the QUAKE tool and the relevant metrics for dependability benchmark.

3.2.1- Experimental Setup and Benchmark Procedure

The QUAKE tool is composed by the following components presented in Figure 2. The main components are the Benchmark Management System (BMS and the System Under Test (SUT). The SUT consists of a SOAP server running some Web/Grid service. From the point of view of the benchmark the SUT corresponds to a web-based application server, a SOAP router and a web-service. That web-service will execute under some workload, and *optionally* will be affected by some fault-load. There are several client machines that invoke requests in the server using SOAP-XML requests. All the machines in the infrastructure are clock-synchronized using NTP. The application under test is not limited to a SOAP-based application: in fact, the benchmark infrastructure can also be used with other examples of client-server applications that use other different middleware technologies.

The Benchmark Management System (BMS) is a collection of software tools that allows the automatic execution of the benchmark. It includes a module for the definition of the benchmark, a set of procedures and rules, definition of the workload that will be produced in the SUT, a module that collects all the benchmark results and produces some results that are expressed as a set of dependability metrics. The BMS system may activate a set of clients (running in separate machines) that inject the defined workload in the SUT by making SOAP requests to the Grid Service. The execution of the client machines is timely synchronized and all the partial results collected by each individual client are merged into a global set of results that generated the final assessment of the dependability metrics. The BMS system includes a reporting tool that presents the final results in a readable and graphic format.

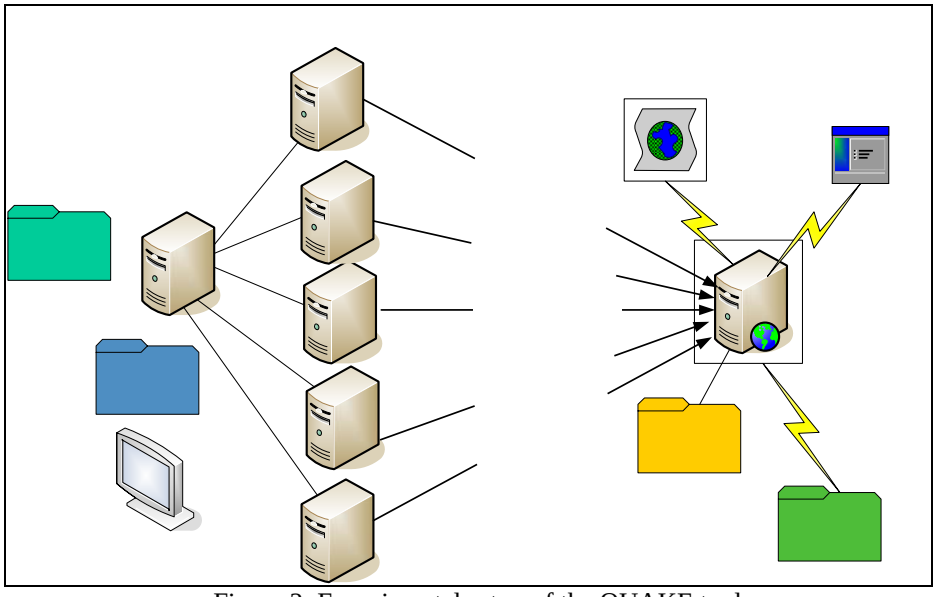

Figure 2: Experimental setup of the QUAKE tool.

The Benchmark Management System (BMS) is a collection of software tools that allows the automatic execution of the benchmark. It includes a module for the definition of the benchmark, a set of procedures and rules, definition of the workload that will be produced in the SUT, a module that collects all the benchmark results and produces some results that are expressed as a set of dependability metrics. The BMS system may activate a set of clients (running in separate machines) that inject the defined workload in the SUT by making SOAP requests to the Web Service. The execution of the client machines is timely synchronized and all the partial results collected by each individual client are merged into a global set of results that generated the final assessment of the dependability metrics. The BMS system includes a reporting tool that presents the final results in a readable and graphic format.

The results generated by each benchmark run are expressed as throughput-over-time (requests-per-second in a time axis), the total turnaround time of the execution, the average latency, the functionality of the services, the occurrence of failures in the Web-Service, the characterization of those failures (crash, hang, zombie-server), the correctness of the final results and the failure scenarios that are observed at the client machines (explicit SOAP error messages or time-outs).

From the side of the SUT system, there are four modules that also make part of the QUAKE benchmark tool: a fault-load injector, a configuration manager, a collector of logs with the benchmark results and a watchdog of the SUT system.

The configuration manager helps in the definition of the configuration parameters of the SUT middleware. It is absolutely that the configuration parameters may have a considerable impact in the robustness of the SUT system. By changing those parameters in different runs of the benchmark it allow us to assess the impact of those parameters in the results expressed as dependability metrics.

The SUT system should also be installed with a module to collect raw data from the benchmark execution. This log data will be then sent to the BMS server that will merge and compare with the data collected from the client machines. The final module is a SUT-Watchdog that detects when a SUT system crashes or hangs when the benchmark is executing. When a crash or hang is detected the watchdog generates a restart of the SUT system and associated applications, thereby allowing an automatic execution of the benchmark runs without user intervention.

There is another external module that will be used but is related with the software rejuvenation study. This module includes two main components:

(a) *An online Surveillance System*: this component is responsible for the online monitoring of some key parameters that may indicate the occurrence of software aging, namely: throughput (requests-per-second), average response time of requests and the memory usage at the server under test.

(b) *A Software Rejuvenation Agent*: this module applies some predictive techniques based on time-series analysis to indicate in advance the prob-

ability of software aging in the server system. It is used to trigger rejuvenation actions to study the impact of the self-healing capability in the system-under-test.

In this particular study, this predictive system does not trigger automatically a rejuvenation action: it is mainly used to study the likelihood of software aging in some packages of SOAP middleware. The rejuvenation actions in this study are triggered by a threshold metric based on the SLA parameters of the web-service. We are not only concerned in software aging that results in hang or crash situations. We are also considering the cases where the software aging manifests as a clear performance degradation that may violate the SLA for that service component, and surely result in profit-loss. In this case, we do apply a rejuvenation action.

3.2.2 Benchmark Procedure

The benchmark can run in three modes:

(a) *Learning mode*: in this mode the main performance metrics of the web-service application are collected. The idea is to take a "picture" of the web-service parameters while it is "young" and to compare it in a long-running execution where it may get "old". In other words, we get the baseline performance parameters and define a SLA for that service with thresholds. During the long-run execution we collect the same parameters and when they deviate from the baseline (as defined in the SLA) more than a certain threshold the SRA agent will trigger a rejuvenation action.

(b) *Workload-only*: in this mode the web-service will be tested under stress situations of workload to see its behavior and the likelihood of software aging. No faultload is injected in this mode, so the web-service will use all the available system resources in the SUT server. In this mode the web-service parameters are monitored in order to detect deviations from the SLA defined from the baseline parameters.

(c) *Workload and faultload*: in this mode, the behavior of the web-service will be studied when we apply simultaneous a defined workload and some particular faultload in the system resources of the server machine.

3.2.3 Workload

Since we want to study the response of a web-service in some stress situations we decided to include a list of different workloads that can be chosen at the beginning of each experiment. Currently, it is possible to use any of the following distributions:

WL1- Continuous burst with a maximum workload;
WL2- Steady-state distribution (N requests-per-second);
WL3- Burst for X minutes, quiet for Y minutes;
WL4- Surge load (fixed request rate and then a peak value in the load).
WL5- Ramp-up distribution: starts with an initial defined request-rate and increases at discrete intervals.

The client machines are all timely synchronized: they start executing at the same time and all of them use the same distribution for the request workload.

There are some other tools in the market that can be used for performance benchmarking of SOAP web-services, namely: SOAtest [34] and TestMaker [35]. However those tools are mainly targeted for testing the functionality of the SOAP applications and to collect some performance figures. QUAKE has some fundamental differences: it is targeted to study the dependability attributes, it includes a different approach for the workload distributions, a fault-load module that will be explained next and is used to evaluate the self-healing capabilities of a SOAP server that provides support for some Grid or Web-Services.

3.2.4 Faultload

The faultload injector does not inject faults directly in the software or in the SOAP messages like in the traditional fault injection tools. This injector is obviously oriented for the problem of software aging. We do not inject any software fault, any hardware fault or any operator fault. Instead, we just emulate resource exhaustion to accelerate the occurrence of software aging and to see the impact in the web-service under test. Other types of stressful condition can also be used in future experiments. The faultload is introduced by an external program that runs in the same server (SUT) and thus compete for the same system resources, consuming one or several of the following operating systems resources:

FL1- Consumption of memory using a ramp-up distribution;
FL2- Creation of a large number of threads;
FL3- Extensive usage of file-descriptors;
FL4- Consumption of database connections;

This list of system resources (the targets of our faultload) may be extended in future versions, but for the time being, those resources seemed to be the most critical ones in a web-service application server. The main point here is that the faultload we "inject" is representative of real scenarios.

3.2.5 Measures

The resulting measures are computed from the data that is collected by the BMS system in the several test-runs. They are grouped in four categories:

(a) Baseline performance measures;
(b) Performance measures in the presence of stress workload distributions;
(c) Performance measures in the presence of stress workload and injected fault-loads at the server side;
(d) Dependability related measures.

The baseline performance measures correspond to the sustained response that is obtained when the SUT is "young". These figures are obtained in short-term runs to avoid the possible occurrence of software aging. The **baseline measures** that we have considered are the following:

- **THR**: corresponds to the sustained throughput (requests-per-second);
- **ART**: average response time for a request;
- **CR**: this corresponds to the number of conforming requests that were performed with a response time lower than a maximum value (Max_RT). Any request to the web-service that exceeds this value may generate a "time-out" and thereby it is important to measure the number of CR (conforming requests) in every test-run;

The performance measures in the presence of workload are $\mathbf{THR_w}$, $\mathbf{ART_w}$ and $\mathbf{CR_w}$, while the performance measures in the presence of workload and faultload are $\mathbf{THR_{wf}}$ $\mathbf{ART_{wf}}$ and $\mathbf{CR_{wf}}$.

Independently from the performance measures, several system parameters are also collected during the benchmark runs to feed the knowledge base of the SRA agent, as explained before.

In addition to the performance-oriented measures, we also consider a set of dependability measures directly obtained by the QUAKE tool. Since we are particularly interested in studying software aging and the effectiveness of potential self-healing mechanisms we consider the following measures, taking into account the work published in [32] and [33]:

- **Integrity**: reports the number of errors that were found in the data of the web-service under test, in the presence of the workload and our faultload. At the end of each test-run we have a procedure for checking the integrity of the final state of the database. Since we are not injecting software/hardware faults it is expected that this metric will always report a value of 100%;
- **Availability**: represents the time the system is available to execute the workload of each test run. The watchdog module that was included in the QUAKE tool is responsible for the assessment of this metric. In this metric we distinguish downtime when the server is down due to a failure from the downtime that may be introduced in a rejuvenation operation. In this latter case, we assume that in production cases there is a cluster with a load-balancer that sends the incoming requests to another server while a SOAP-server is being rejuvenated. It is worth noting that the availability measure is related to the experimental conditions of the QUAKE tool and does not represent field availability (unconditional availability). As it is typically the case for dependability benchmarking, this availability measured is meant to be used for comparison purposes.
- **Autonomy**: this measure shows the need for manual intervention from the system manager to repair the SOAP server in the presence of a hang situation. Hang-type failures are much more expensive than clean crash-failures: while in the latter case the application server may recover autonomously with a restart

operation, the hang-scenario requires the manual intervention of a system manager to kill processes and clean some system resources.

- **Self-Healing Effectiveness**: this measure is only considered in some test-runs, where we want to study the effectiveness of the SRA to perform proactive rejuvenation of the SUT. More than the effectiveness of the technique, it measures the positive impact in the autonomy, availability and the performance of the web-service.

3.2.6 Benchmark Properties

As was explained in [36] a benchmark should offer the following properties:

- **Repeatable**: our QUAKE infrastructure obtains similar results when running several times with the same workload in the same SUT;
- **Portable**: as will be presented later, our QUAKE tool allows the comparison of different applications in this domain, that maybe implemented with different SOAP packages and even with other communication middleware;
- **Realistic**: the scenario portrait by the QUAKE tool represents typical Web-service applications and the harsh load conditions represented by QUAKE workload are actually common in this type of system. Furthermore, as the fault load does not inject any software or hardware faults (only consumes system resources), the QUAKE tool does not suffer from the representativeness difficulties that affect typical dependability benchmarks.

So, in summary the QUAKE tool is able to be used in other environments for testing traditional client-server and Grid-based applications where the subject of software aging and self-healing would be a point of concern. This section presented a general description of QUAKE.

4 Experimental Results

4.1 Fault Injection

We conducted a series of tests to validate our approach by injecting faults on actual distributed applications. We chose the XtremWeb platform [22] to perform our experiments. XtremWeb is a general purpose platform that can be used for high performance distributed calculus. The original XtremWeb application is written in Java, but we used FCI on the C++ version of the software, that is meant to be the most efficient version. XtremWeb participants are usually divided according to the kind of jobs they are doing: the *dispatcher* distributes jobs that are to be executed by the *clients*, each client querying and performing the actual work. The XtremWeb application that was run on the platform is *POV-Ray*, which creates three-dimensional, photo-realistic images using a rendering technique called ray-tracing. We used the

same cluster of computers as in the FCI overhead experiment, but only 35 cluster nodes were participating to the ray-tracing to the calculus.

We considered the task of calculating the same picture twice. Calculating a POV-Ray picture runs as follows: first, the dispatcher divides the image to be calculated into parts, each to be computed by a client, then, after each client has computed its image portion and sent it back to the dispatcher, the dispatcher selects a simple client to collect parts of the image and aggregate them into a single final image, that is finally sent back to the dispatcher.

With this application, we designed a fault scenario using FAIL. The dispatcher is run on Computer 1 and is not subject to faults (it simply waits for clients to connect and feeds them jobs). Then, XtremWeb clients are run on the remaining 34 computers and are subject to fault events. The fault scenario that is run on every client is as follows:

1. Every 5 seconds, an XtremWeb client is likely to crash with probability *x*,
2. After a crash, every 5 seconds, a client may restart with probability 0.3,
3. A client may crash only once.

We carried out this test using various values for *x* (0.1, 0.3, 0.5, 0.7, and 0.9), and the experiment results are summarized in Figure 3.

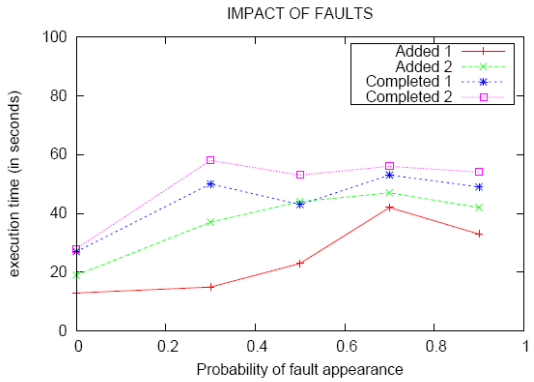

Figure 3: FCI fault-injection results

In this Figure, *Added 1* and *Added 2* correspond to the time when the dispatcher collected all image parts from all concerned clients for the first and second images, respectively. *Completed 1* and *Completed 2* correspond to the time when the dispatcher received the final first and second images, respectively. Also, the test for failure probability 0 serves as a reference test.

The outcome of these fault injections is consistent with what could be expected. First, for any fault appearance probability, the time needed to complete the second image is less than twice the time to complete the second image. This is due to our scenario where a machine may not fail twice, so that if it failed while calculating the

first image and recovered (this happens with probability 0.3), it may not fail again while calculating the second image.

When the probability *x* of fault appearance is low (under 0.5), the execution time of the image parts calculation is kept low, since clients are only responsible for a small percentage of the global result, so that the load of crashed clients can be carried out by someone else quickly. However, the full image completion time is high, because if a node crashes in that part, it is the only one responsible for carrying out the final image, so its failure has high impact. When the probability of fault appearance is high (more than 0.5), the execution time of the image parts calculation is very high since a large number of clients are likely to crash in this process. The final image completion time is high too, but the overhead of this part is not as high as when the fault appearance probability is low, simply because most faults occurred while calculating the images, so most clients will not fail again in the last phase of the calculus.

4.2 Dependability Benchmarking

In this section we present some experimental results taken with the QUAKE benchmark. To easily evaluate the correctness of the application after the execution of several benchmark runs we have developed a SOAP service that make access to a database and provide a simulated online e-banking application. This way we can easily check at the end of each run if the database contains corrected data and the at-most-once semantics has been followed in the execution of the SOAP service. This was the synthetic application that has been used in the experiments that will be described herein. Other Web-Services and Grid Services are currently under assessment by the QUAKE tool.

The testing infrastructure was composed by a cluster with 12 machines running Linux and Java 1.4. The SOAP service was running on a central node (dual-processor) of the cluster and we have use a TOMCAT-AXIS server running on top of Linux. As far as we know, most of the Java-based Web-Services and Grid-Services are currently using Tomcat-Axis so we were interested to evaluate the robustness of this middleware.

From those 12 machines, one was running the SUT application, other was dedicated to the BMS system, and the remaining 10 machines were running instances of the clients that were in practice the workload generators. For these results herein presented we have chosen the following parameters:

(a) default configuration parameters of the JVM, Tomcat and Axis;
(b) the Tomcat JVM was running with the implicit Java garbage-collector.
(c) in the overall, the client machines will send 1 million of SOAP requests;
(d) the request will follow the "*continuous-burst*" distribution;
(e) there are no retransmission of SOAP requests when a client gets a response error. This way there are no repeated messages and the "at-most-once" semantics is not violated;

(f) No fault-load is introduced in the SUT system. We ran the SOAP services in a dedicated server and all the operating system resources were available to the application. This means we are testing a Web-Service in a normal environment with any perturbations at the system-level.

The results of the first experiment are presented in Figure 4. The Figure presents the number of requests-per-second that is served by the SOAP service over the time axis. In this benchmark run, the client machines sent 1 million of requests to the SOAP service running in a dedicated machine with a dual-processor. We used the default configuration of Tomcat/Axis, that as matter of information allocates a Java JVM of 64Mb.

This first run produced impressive results: this test took 31 minutes, and only 73.740 of the 1 million requests were processed (about 7.37% of the total). The remaining requests were not processed by the server due to "out of memory". It was observed that the reason for this failure was directly related with the occurrence of memory leaks in the Tomcat/Axis middleware.

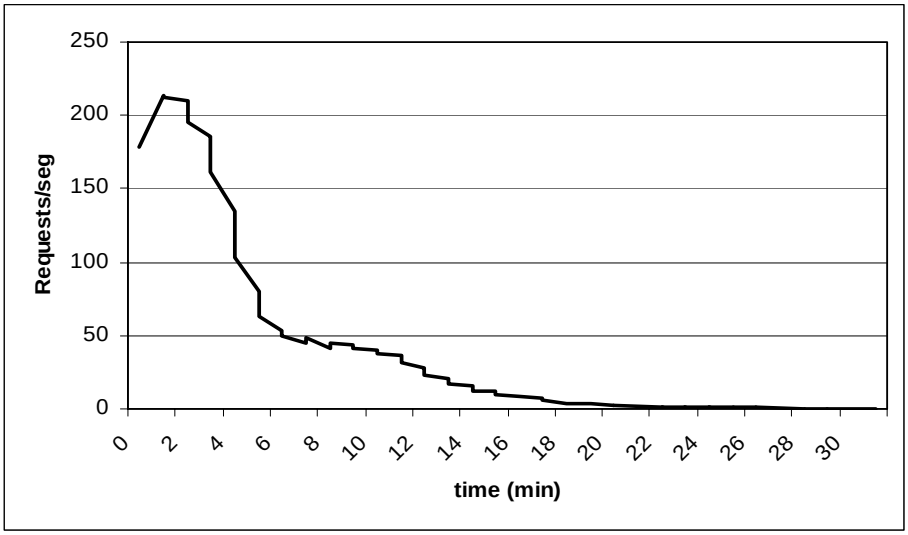

Figure 4: Results for the first test run, with default-configuration

More interesting that this result is the type of failure that happened in the SUT server: the Tomcat processes did not crashed, they were left in a completely hang-status that even the `shutdown` command of Tomcat was not able to restart the server. It was necessary to kill explicitly all the processes and restart the Tomcat server.

This would be that type of failures that would require a human intervention in a production system. These type of failures are very expensive to maintain since they

require human intervention. When the systems start growing in complexity the management will be almost virtually impossible [24]. The vision for autonomic computing defended by IBM researchers is entirely shared by the authors of this paper that recognize the strategic importance for creating self-healing Grid Services.

From the first test run was clear that the SOAP server was under-configured in terms of memory for the selected workload. So, in the second test run the memory of the Tomcat JVM was increased from 64Mb up to 1Gb. The results are presented in Figure 5.

This time the SOAP server did not crash and executed all the 1 million requests. The total turnaround the execution was 737 minutes. Those peaks that show up in the graphic have to due with the execution of the garbage collector at the server side. We can conclude from the graphic that the SOAP service is maintained running but the throughput (requests-per-second) drops heavily over time, which ends in the observation: the SOAP service does not crash, but it runs slower and slower over time.

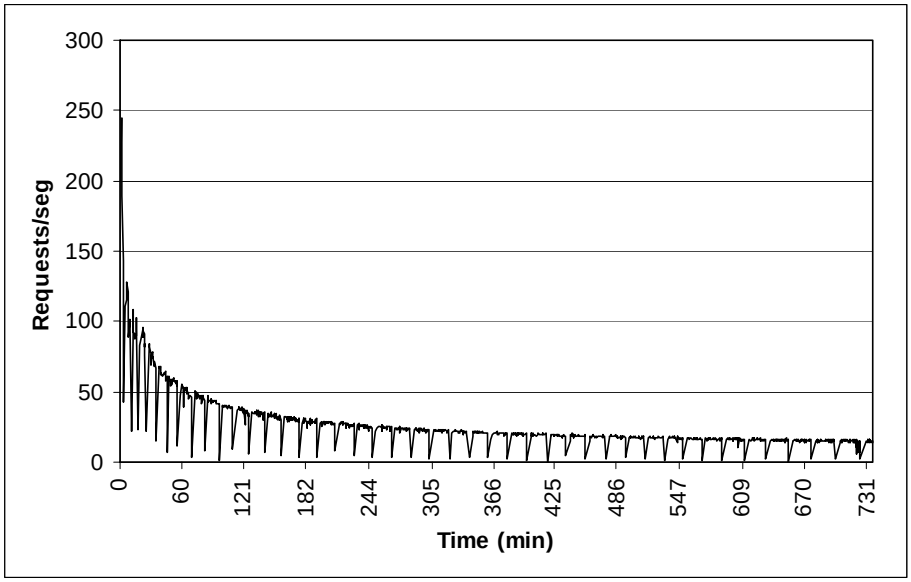

Figure 5: Results for the second test run, with a JVM of 1Gb.

Once again the reason for this performance drop-out has to due with memory leakage in the SOAP middleware. In the third test run we tried to observe the impact of the garbage collector so we changed the default configuration, optimized the `gc` parameters and ran it explicitly in a periodic basis. The results are presented in Figure 6.

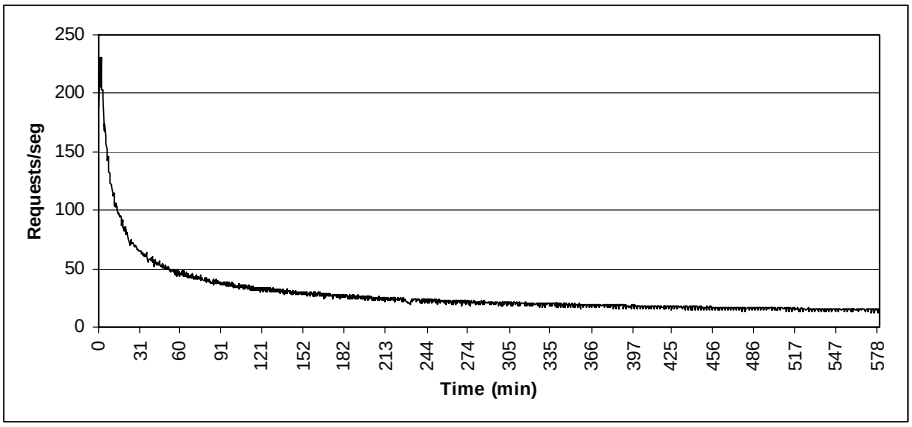

Figure 6: Results for the third test run, with a JVM of 1Gb but explicit GC.

The performance drop-out is still clear but there was a clear improvement in the response of the SOAP service: the turnaround time of the run was reduced from 737 to 580 minutes.

One point of concern that we get even in this configuration scenario is the sharp decrease in the QOS level of the SOAP service: at the beginning it was able to sustain about 230 requests-per-second. At the end of the test run the throughput was less than 20 requests-per-second, so 10% of the initial throughput.

This observation led us to think: how can we improve the throughput level of the SOAP service and maintain it at acceptable levels? How can we provide some self-healing mechanisms to this SOAP server? How can we prevent the SOAP server to fail and be left in a hang-status?

With these questions in mind we start thinking about applying some software-rejuvenation technique to increase the throughput of that SOAP service. And the decision was to implement a preventive rebooting to avoid a zombie crash (hang status) of the server but also to avoid that the server would fall down into a lower level of throughput: when the throughput level decrease down to 20% of the initial throughput the watchdog produced a restart of the Tomcat/Axis server. This restart was done in a clean way: the SOAP server closed the service to new requests and all the on-going requests were finished before applying the shutdown-restart to the Tomcat. At the end of the test run the correctness of the application was successfully verified.

In Figure 7 we present the results of this test run. Those deep peaks in the throughput level correspond to a restart event. Every shutdown-restart of the Tomcat took between 14 to 16 seconds, in average.

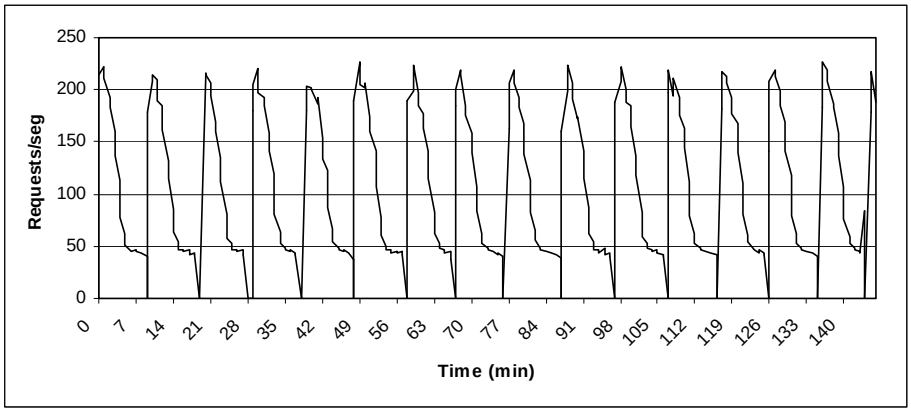

Figure 7: Results for the fourth test run with a preventive shutdown of the server

At first sight it seems that this technique would not produce interesting results, since it creates some seconds of downtime at the SOAP server. As can be seen in the Figure there was 15 preventive restarts and this may had resulted in 225 second of downtime in the overall. So this technique is not good from the point of view of availability metric.

But the result obtained in the turnaround metric is quite interesting: the total turn-around the test run was 146 minutes. This means the SOAP service was 5 times faster when compared with the second test run. It is clear that this "*wise-reboot*" technique is a potential technique to increase the sustained throughput level of the SOAP server and to avoid the zombie crashes of the server that would normally require human intervention.

There are more results taken with the QUAKE tool, but these small set of results is clear representative of the interest of using dependability benchmarking to assess the robustness of SOAP services and Grid services

5 Conclusions and Current Status

We reviewed several available tools for software fault injection and dependability benchmarking tools for grids. We emphasized on the FAIL-FCI fault injector developed by INRIA, and on the QUAKE dependability benchmark developed by the University of Coimbra.

The FAIL-FCI tool has so far only provided preliminary results on desktop grid middleware (XtremWeb) and P2P middleware (the FreePastry Distributed Hash Table). These results permitted to identify quantitative failure points in both tested middleware, as well as qualitative issues concerning the failure recovery of XtremWeb.

With the QUAKE tool we have been conducting the following experimental studies:

(a) Evaluate the robustness of some existing SOAP servers (Apache Axis, JBoss, gSOAP, MS.NET);
(b) Assess the reliability of different middleware for client/server applications;
(c) Evaluate the effectiveness of some mechanisms for software rejuvenation;
(d) Study the reliability of OGSA-DAI implementation;

After that last phase we are planning to assess the dependability level of further modules from the Globus middleware (GT4).
After the core of the tool set is properly packaged for clusters, our goal is to enable larger scale experiments by *(i)* using Grids instead of clusters, and *(ii)* integrate with emulation mechanisms.

6 Acknowledgements

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

References

1. P.Koopman, H.Madeira. “Dependability Benchmarking & Prediction: A Grand Challenge Technology Problem”, Proc. 1st IEEE Int. Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems; Phoenix, Arizona, USA, Nov 1999
2. S Ghosh, AP Mathur, "Issues in Testing Distributed Component-Based Systems", 1st Int. ICSE Workshop on Testing Distributed Component-Based Systems, 1999
3. H. Madeira, M. Zenha Rela, F. Moreira, and J. G. Silva. “Rifle: A general purpose pin-level fault injector”. In European Dependable Computing Conference, pages 199–216, 1994.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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
9. N. Looker, J.Xu. “Assessing the Dependability of OGSA Middleware by Fault-Injection”, Proc. 22nd Int. Symposium on Reliable Distributed Systems, SRDS, 2003
10. http://www.lri.fr/~fci/GdX

11. 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.
12. G. Fedak, C. Germain, V. Néri, and F. Cappello. “XtremWeb: A generic global computing system”. Proc. of IEEE Int. Symp. on Cluster Computing and the Grid, 2001.
13. I.Foster, C. Kesselman, J.M. Nick and S. Tuecke. “Grid Services for Distributed System Integration”, IEEE Computer June 2002.
14. J. Kephart. “Research Challenges of Autonomic Computing”, Proc. ICSE05, International Conference on Software Engineering, May 2005
15. William Hoarau, and Sébastien Tixeuil. “A language-driven tool for fault injection in distributed applications”. In Proceedings of the IEEE/ACM Workshop GRID 2005, page to appear, Seattle, USA, November 2005.
16. G. Avarez and F. Cristian, "Centralized failure for distributed, fault-tolerant protocol testing," in Proceedings of the 17th IEEE Internationnal Conference on Distributed Computing Systems (ICDCS'97) May 1997.
17. E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool", Proceedings of the 24th International Symposium on Fault-Tolerant Computing, June 15-17, 1994.
18. M. Vieira and H. Madeira, “*A Dependability Benchmark for OLTP Application Environments*”, Proc. 29th Int. Conf. on Very Large Data Bases (VLDB-03), Berlin, Germany, 2003.
19. K. Buchacker and O. Tschaeche, “*TPC Benchmark-c version 5.2 Dependability Benchmark Extensions*”, http://www.faumachine.org/papers/tpcc-depend.pdf, 2003
20. D. Wilson, B. Murphy and L. Spainhower. “*Progress on Deining Standardized Classes of Computing the Dependability of Computer Systems*”, Proc. DSN 2002, Workshop on Dependability Benchmarking, Washington, D.C., USA, 2002.
21. A. Kalakech, K. Kanoun, Y. Crouzet and A. Arlat. “*Benchmarking the Dependability of Windows NT, 2000 and XP*”, Proc. Int. Conf. on Dependable Systems and Networks (DSN 2004), Florence, Italy, 2004.
22. J. Durães, H. Madeira, “*Characterization of Operating Systems Behaviour in the Presence of Faulty Drivers Through Software Fault Emulation*”, in Proc. 2002 Pacific Rim Int. Symposium Dependable Computing (PRDC-2002), pp. 201-209, Tsukuba, Japan, 2002.
23. A. Brown, L. Chung, and D. Patterson. "*Including the Human Factor in Dependability Benchmarks*", Proc. of the 2002 DSN Workshop on Dependability Benchmarking, Washington, D.C., June 2002.
24. A. Brown, L. Chung, W. Kakes, C. Ling, D. A. Patterson, "*Dependability Benchmarking of Human-Assisted Recovery Processes*", Dependable Computing and Communications, DSN 2004, Florence, Italy, June, 2004
25. A. Brown and D. Patterson, "*Towards Availability Benchmarks: A Case Study of Software RAID Systems*", Proc. of the 2000 USENIX Annual Technical Conference, San Diego, CA, June 2000
26. J. Zhu, J. Mauro, I. Pramanick. “*R3 - A Framework for Availability Benchmarking*”, Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003), USA, 2003.
27. J. Zhu, J. Mauro, and I. Pramanick, “*Robustness Benchmarking for Hardware Maintenance Events*”, in Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003), pp. 115-122, San Francisco, CA, USA, IEEE CS Press, 2003.

28. J. Mauro, J. Zhu, I. Pramanick. "*The System Recovery Benchmark*", in Proc. 2004 Pacific Rim Int. Symp. on Dependable Computing, Papeete, Polynesia, 2004.
29. S. Lightstone, J. Hellerstein, W. Tetzlaff, P. Janson, E. Lassettre, C. Norton, B. Rajaraman and L. Spainhower. "*Towards Benchmarking Autonomic Computing Maturity*", 1st IEEE Conf. on Industrial Automatics (INDIN-2003), Canada, August 2003.
30. A.Brown, J.Hellerstein, M.Hogstrom, T.Lau, S.Lightstone, P.Shum, M.P.Yost, "*Benchmarking Autonomic Capabilities: Promises and Pitfalls*", Proc. Int. Conf. on Autonomic Computing (ICAC'04), 2004
31. A. Brown and J. Hellerstein, "*An Approach to Benchmarking Configuration Complexity*", Proc. of the 11th ACM SIGOPS European Workshop, Leuven, Belgium, September 2004
32. A.Brown, C.Redlin. "*Measuring the Effectiveness of Self-Healing Autonomic Systems*", Proc. 2nd Int. Conf. on Autonomic Computing (ICAC'05), 2005
33. J. Durães, M. Vieira and H. Madeira. "*Dependability Benchmarking of Web-Servers*", Proc. 23rd International Conference, SAFECOMP 2004, Potsdam, Germany, September 2004. Lecture Notes in Computer Science, Volume 3219/2004
34. Parasoft SOAtest: http://www.parasoft.com
35. PushToTest TestMaker: http://www.pushtotest.com
36. J.Gray (Ed). "*The Benchmark Handbook*", Morgan-Kaufmann Publishers, 1993