Using failure injection mechanisms to experiment and evaluate a grid failure detector

Sébastien Monnet, Marin Bertier

Using failure injection mechanisms to experiment and evaluate a grid failure detector

Sébastien Monnet[1] and Marin Bertier[2]

[1] IRISA/University of Rennes I, Sebastien.Monnet@irisa.fr
[2] IRISA/INSA, Marin.Bertier@irisa.fr

Abstract. Computing grids are large-scale, highly-distributed, often hierarchical, platforms. At such scales, failures are no longer exceptions, but part of the normal behavior. When designing software for grids, developers have to take failures into account. It is crucial to make experiments at a large scale, with various volatility conditions, in order to measure the impact of failures on the whole system. This paper presents an experimental tool allowing the user to inject failures during a practical evaluation of fault-tolerant systems. We illustrate the usefulness of our tool through an evaluation of a hierarchical grid failure detector.

Keywords

Failure injection, failure detection, performance evaluation, fault tolerance, grid computing

1 Introduction

A current trend in high-performance computing is the use of large-scale computing grids. These platforms consist of geographically distributed cluster federations gathering thousands of nodes. At this scale, node and network failures are no more exceptions, but belong to the normal system behavior. Thus grid applications must tolerate failures and their evaluation should take reaction to failures into account.

To be able to evaluate a fault-tolerant application, it is essential to test how the application reacts to failures. But such applications are often non deterministic and failures are not predictable. However, an extensive experimental evaluation requires execution reproducibility.

In this paper, we introduce a failure injection tool able to express and reproduce various failure injection scenarios. This provides the ability to extensively evaluate fault-tolerance mechanisms used by distributed applications. As an illustration, we use this tool to evaluate a failure detector service adapted to the grid architecture [7]. A failure detector is a well-known basic building block for fault-tolerant distributed systems, since most fault-tolerance mechanisms require to be notified about failures. These experiments are run over the Grid'5000 National French grid platform [1].

The remainder of this paper is composed as follows: Section 2 motivates the need of failure injection mechanisms. Section 3 describes our failure injection tool and explains

how to use it. Section 4 presents the failure detector we evaluate. Section 5 illustrates the usage of our failure injection tool for a practical evaluation of the failure detector. Finally, Section 6 concludes the paper.

2 Experimenting with various volatility conditions

2.1 System model

In this paper, we suppose that the grid architecture of the system implies a hierarchical organization. It consists of clusters of nodes with high-connectivity links, typically System Area Networks (SAN) or Local Area Networks (LAN), interconnected by a global network, such as Internet. In this context, we call *local group* each pool of processes running within the same SAN or LAN, and *global network* the network which connects the local groups.

Each local group is a finite set of processes that are spread throughout a local network. The distributed system is composed of a finite set of local groups. Every process communicates only by sending and receiving messages. All processes are assumed to have a preliminary knowledge of the system's organization.

We rely on the model of partial synchrony proposed by Chandra and Toueg in [12]. This assumption fit the behavior of a typical computing grid: nodes crashes are possible and messages may be delayed or dropped by routers during network congestion.

Note that the two levels of the grid hierarchy exhibit different properties for communications (latency, bandwidth, message loss rate).

2.2 Benefits of experimentation.

A theoretical evaluation of a system can be carried out using a formal proof of a system model, which can validate the system design. However, it relies on a formal model, which is generally a simplification of the reality (taking into account only *significant* parameters). A second type of evaluation uses extensive simulations [9,15,11], which as formal proof, generally run models of the design and not the implementation itself. Finally, experimentations on real testbeds can serve as a proof of concept. Such a practical evaluation can capture aspects related, for instance, to the node specifications or to specifics of the underlying physical network. In this paper we focus on experimental evaluations.

Experimenting large-scale distributed software is difficult. The tests have to be deployed and launched on thousands of geographically distributed nodes, then the results have to be collected and analyzed. Besides, the tests have to be *reproducible*. Achieving these tasks for a large-scale environment is not a trivial task.

2.3 Controlling volatility

In the context of large-scale, fault-tolerant distributed systems, one important aspect which needs to be controlled is *node volatility*. This section introduces a tool that provides the ability to inject failures according to pre-defined scenarii during experiments, in order to evaluate the quality of fault-tolerance mechanisms. More specifically, we illustrate how such a tool can be used in order to test a failure-detection service.

Failure injection requirements. The use of failure injection mechanisms provides the ability to test fault-tolerant mechanisms with different volatility conditions. This may validate that the service provided by the software is still available when particular types of failures occur. It also provides the ability to measure the overhead introduced by the fault-tolerant mechanism to support different kinds of failures.

The experimentations are run on a testbed that is assumed to be stable. As we want to experiment with controlled failures, we assume that there are no other unexpected failures during the test (in case of a real failure, the developer will have to re-launch his test). The test tool can help the developer to introduce controlled failures during the experimentation. In order to emulate some specific scenarios and to be scalable, the test tool has to provide a simple and efficient way to describe failures distributed across thousands of nodes.

The failure injection mechanisms should be able to take only *statistical parameters* and then compute failure schedules accordingly. This allows the tester to generate a failure scenario across thousands of nodes by giving only a few parameters. The failure injection mechanisms also need to be *highly customizable* allowing the user to specify groups of nodes that should fail simultaneously. More generally, they need to provide the availability to express *failure dependencies between nodes*. This should allow the tester to emulate correlated failures. Furthermore, an important feature of a failure injector (volatility controller) is *reproducibility*. Even if failures are computed using statistical parameters, one may want to replay an execution with the same set of failures, while varying other parameters (e.g. in order to tune the fault-tolerance algorithms). While experimenting various parameters of a fault tolerance feature or testing different fault tolerance mechanisms one may want to compare different solutions within the same context.

Scenarios.

Simple failure scheme. One simple way to describe a failure scheme is to assume that all the nodes have the same probability of failure and that they are independent (i.e the failure of a particular node does not depend on the failure of other ones). For instance, one may assume that the MTBF (Mean Time Between Failures) of a particular set of nodes may be one hour. The MTBF of a specific architecture can be easily observed. The developer may wish to run experiments with smaller MTBF values in order to stress the fault-tolerant mechanisms.

Correlated failures. As the underlying physical network system may be very complex, with hubs and switches, some failures may induce new ones. The crashes of some nodes may lead to the crashes of other nodes. By instance, while running a software on a cluster federation, a whole cluster may crash (Figure 1). This may be due to a power failure in one cluster room, for instance. While designing a fault-tolerant system for such an architecture, it is important to experiment its behavior while multiple failures occurs concurrently as it may happen in real executions (without failure injection mechanisms).

Accurate control. As the roles played by the different nodes in the system may not be strictly equivalent (some are more critical than others), the developer should be able to test some particular cases. For instance, one may want to experiment the simultaneous crash of two particular nodes, or the crash of a node when it is in a particular state. Typically, as illustrated by Figure 1, experimenting the failure of a node having manager capabilities may be really interesting as it may involve particular cases of the fault-tolerant algorithms.

2.4 Related work

Researchers working in the fault-tolerance area need to inject failures during their experiments. Most often this is done in an ad-hoc manner, by manually killing nodes or by introducing a few code statements into the tested system's source code, to make failures occur. The overhead for the tester is non negligible and usually it is neither scalable nor reproducible. The goal of our failure injection mechanism is precisely to automate this task, making it easy for the testers to inject failures at *large scale* and to *control* volatility conditions.

Many research efforts focus on failure injection. However, most of them are very theoretical or focus on the problem of failure prediction [18,17,6]. In this paper we do not address the issue of *when* a failure should be injected or *what* it will induce, but we provide a practical solution to *how* to inject it. The tester may use the results of these previous research works to feed our failure injectors.

Failure injection has also been studied for simulation and emulation. For instance, [3] provides a solution to test protocols under failure injection, but it relies on a fully *centralized* approach. Our work is intended to be used for tests running on real *distributed* architectures, with the full application code.

FAIL [14] (FAult Injection Language) defines a smart way to define failures scenarios. It relies on a compiler to trap application communications to emulate failures. Our work is integrated in the test environment, not at application level, thus it allows to inject failures even when the source code is unavailable.

In contrast to previous work, we use failure mechanisms within a test tool, providing a simple way to deploy, run and fetch results of a test under various controlled volatility conditions.

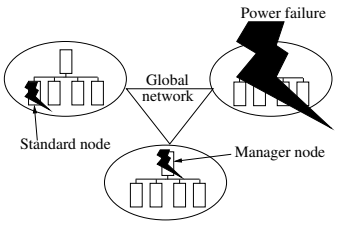

Fig. 1. Different kinds of failures

```
(00) <network analyze-class="test.Analyze">
(01)  <profile name="manager" replicas="1">
(02)       <!-- peer information -->
(03)   <peer base-name="peerA"/>
...
(11)   <bootstrap class="test.MyClass1"/>
(12)       <!-- argument -->
(13)   <arg value="x"/>
(14)  </profile>
(15)  <profile name="non-manager" replicas="20">
(16)   <peer base-name="peerB"/>
...
(23)    <bootstrap class="test.MyClass2"/>
(24)  </profile>
(25) </network>
```

Fig. 2. *JDF*'s description language

3 Our proposal: a flexible failure injection tool

3.1 JXTA Distributed Framework (*JDF*).

We are currently developing our failure injection mechanism within the JXTA Distributed Framework (*JDF* [19,4]) tool. The *JDF* project has been initiated by Sun Microsystems, and is currently being actively developed within the PARIS Research Group [2]. JDF is a tool designed to automate the tests of JXTA-based systems. In [4] we have specified that this kind of tool should provide the ability to control the simulation of nodes' volatility. In the remaining of this section we show how to provide this ability inside *JDF*. A detailed description of *JDF* can be found in [19].

JDF allows the user to describe his test through 3 configuration files. 1) a node file containing the list of nodes on which the test is run, 2) a file storing the names and paths of the files to deploy on each node, and 3) a XML file describing the node profiles, in particular, the Java classes associated and the parameters given to these classes.

JDF's XML description file allows the tester to describe his whole system through *profiles*. Figure 2 defines two profiles, one from line 01 to 14 and one from line 15 to 24. Then multiple nodes can share a same profile. The profile named *non-manager* on Figure 2 is replicated on 20 different nodes (thanks to the *replicas* attribute). The first experimentation phase consists of the creation of these files. This phase is called *basic configuration* thereafter.

3.2 *JDF* description language extension.

The first requirement to fulfill in order to use failure injection is to incorporate failure information is into the *JDF* test description language

To provide the ability to express failures dependencies (to represent correlated failures) we add a new XML tag: *failure*. This tag may have 2 attributes: 1) *grp* to indicate that all nodes having this profile are part of a same *failure group*; 2) *dep* to indicate that nodes having this profile depend, from a failure point of view, on nodes of another profile. The *grp* attribute allows to specify groups of nodes that should fail together (i.e. if one of them crashes, then all the set crashes). This can help the tester to simulate the failure of clusters, for instance. The *dep* attribute can be used to indicate that a node should crash if another one crashes (by instance to emulate the fact that the second node may serve as a gateway for the first one). For instance, in Figure 2, adding the line "*(17) <failure grp="1"/>*" in the *non-manager* profile will make all the *non-manager nodes* crash as soon as on of them crashes. Furthermore, if the line "*(18) <failure dep="manager"/>*" is added, all *non-manager nodes* will crash if the node having *manager* profile crashes.

3.3 Computing the failure Schedule.

We have developed a tool that generates a configuration file with volatility-related parameters (e.g. the global MTBF) which are given as an input to *JDF*. To do this, we introduce a new configuration file. In order to make the failure conditions reproducible,

this file contains the uptimes for all nodes (i.e. the failure schedule). It is generated using the XML description file, which is necessary in order to take into account failure dependencies. This phase is called *failure schedule generation* thereafter.

The tool works as follows: it computes the first date using a given MTBF and the number of nodes (obtained from the XML description file), then it randomly chooses a node to which it assigns this first failure date. This operation is repeated until a failure date is assign to each node. Next, dependency trees are built using the information contained in the XML description file. The dependency trees are used to ensure that 1) in each failure group, nodes are assigned the smallest failure date of the group; 2) if the failure date of a node is greater than the failure date of a node on which it depends (the *dep* attribute), then the smallest date is assigned. This way, all dependencies expressed in the XML description file are satisfied.

Computing the failure schedule statically before launching the test allows the tester to easily reproduce failure conditions. The tool can be launched once to compute a failure schedule, and the same computed schedule can be used by multiple experiments.

3.4 Running experiments with failure injection.

Assuming that the standard *JDF* configuration files exist (i.e. the *basic configuration* phase has been done), the complexity overhead induced by the failure injection mechanisms to launch tests is very low.

To run a test by providing a MTBF value (*Simple failure scheme*) the tester has to launch a *JDF* script that will compute the failure dates before running his test (boxes A, C and E in Figure 3).

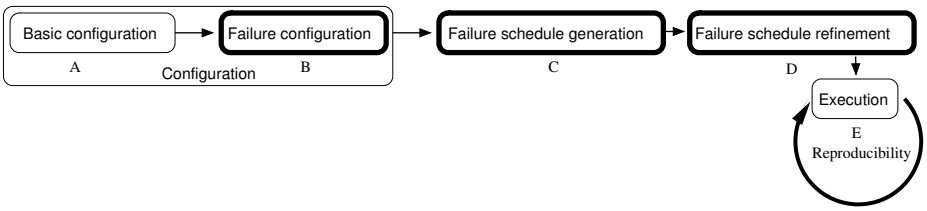

Fig. 3. Failure injection usage

The *failure schedule generation* phase consist in executing a script with the desired MTBF value and the *JDF* standard configuration files.

As a further step, to use correlated failures, the tester needs to use our *JDF* description language extension. In this case, one *Failure configuration* phase is required to add *failure* tags in the XML configuration file (Box B in Figure 3).

Finally, the tester may need an accurate control of the failures (i.e inject a failure on a specific node at a specific time). To do this, the failure schedule has to be explicitly edited (Box D in Figure 3).

Once the schedule is computed no extra step is needed to re-execute an experiment with the same failure conditions.

3.5 Run time failure injection.

At deployment time, a configuration file containing the failure schedule is sent to each node. At launch time, a *killer* thread is started. This thread reads the failure date (which is actually an uptime), then waits accordingly. If the application is still running at the failure date, this thread kills it, thereby emulating a failure. Note that all the application threads are killed, but the physical node itself remains up and running. If an application uses the TCP protocol, the node will answer *immediately* that no process is currently listening to this port. If the node were really down, the application would have to wait for a TCP time-out. In the case of the UDP protocol (as for the experiments presented in this paper), this side-effect does not exist. For applications using the TCP protocol, the thread *killer* should either trap messages or really shutdown the network interface.

4 A scalable failure detection service

We used the failure injection mechanisms previously described to evaluate a scalable failure detector adapted to hierarchical grids.

4.1 Unreliable failure detectors

Concepts. Since their introduction by Chandra and Toueg in [12], failure detectors are becoming a basic building block for fault-tolerant systems. A failure detector is one solution to circumvent the impossibility [13] of solving deterministically the consensus in asynchronous systems in presence of failure. The aim of failure detectors is to provide information about the liveness of other processes. Each process has access to a local failure detector which maintains a list of processes that it currently suspects of having crashed. Since a failure detector is unreliable, it may erroneously add to its list a process which is still running. But if the detector later realizes that suspecting this process is a mistake, it then removes the process from its list. Failure detectors are characterized by two properties: completeness and accuracy. Completeness characterizes the failure detector capability of suspecting incorrect process permanently. Accuracy characterizes the failure detector capability of not suspecting correct processes.

We focus on the $\Diamond P$ detector, named *Eventually Perfect*, it is one of failure detector classes, which enable to solve the consensus problem (i.e. it is not the weakest). This detector requires the following characteristics:

Strong completeness: there is a time after which every process that crashes is permanently suspected by every correct process.

Eventual strong accuracy: there is a time after which correct processes are not suspected by any correct process.

Utility. A failure detector $\Diamond P$ provides the ability to solve the consensus, but it does not contradict the impossibility of Fischer, Lynch and Paterson, then it is impossible to implement it in asynchronous systems.

A failure detector has several advantages from a theoretical and a practical point of view. The first one is to abstract synchronism matter: algorithms that use a failure detector depends on failure only. The hypotheses, in terms of completeness and accuracy, describe how a failure detector detects other processes failures. These hypotheses are more natural than temporal ones but also useful.

In a practical way, the need to detect failures is a common denominator among the majority of distributed reliable applications. In fact an application must know if one of these processes has crashed: to be able to replace it in case of replication or more generally to avoid waiting infinitely its result. From this perspective, a failure detector is a specific service which provides the ability to guarantee the application vivacity. This service can be shared by several applications and then its cost is amortized.

4.2 GFD (GRID Failure Detector)

Properties. The aim of our failure detector is to propose a shared and moreover scalable detection service among several applications. In this implementation we dissociate two aspects: a basic layer which computes an estimation of the expected arrival date to provide a short detection time and an adaptation layer specific for each application. This adaptation layer guarantees the adequacy between the detection quality of service and the application needs. This architecture provides the ability to generate only one flow of messages to provide adapted detection information for all applications.

The second specificity is the hierarchical organization of the detection service in order to decrease the number of messages and the processor load [8]. It comprises two levels: a local and a global one, mapped upon the network topology. The system is composed of local groups, mapped upon SANs or LANs, bound together by a global group. Each group is a detection space: every group member watches all the other members of its group. Every local group designates at least one mandatory which will participate to the global group.

This organization implies two different failure detector types. This distinction is important since a failure does not have the same interpretation in the local context as in the global one. A local failure corresponds to the crash of a host, whereas in the global context a failure represents the crash of an entire local group. In this situation, the ability to provide different qualities of service to the local and the global detectors is a major asset of our implementation. Therefore a local group mandatory has two different failure detectors, one for the local group and one for global group.

In a local group, the failure detector uses IP-Multicast for sending periodicals *"I am alive"* messages. In SANs and LANs, IP-Multicast can be used with the broadcast property. Therefore a host only sends one message to communicate with all the other hosts. Failure detectors in a global group use UDP in order to be more compatible with the general network security policy.

5 Experimentations

We use our tool to inject failures and measure the time it takes to detect them with our failure detection service. Reproducibility is used to perform multiple experiments with

the same set of failure while tuning the failure detector. The correlated failure feature is used to experiment the global level of the failure detector's hierarchy.

5.1 Experimental setup.

For all the experiments, we used the Grid'5000 platform [1], which gathers 9 clusters geographically distributed in France. These clusters are connected together through the Renater Education and Research National Network (1 Gb/s). For our preliminary experiments, we used 64 nodes distributed in 4 of these sites (Rennes, Lyon, Grenoble and Sophia). In these 4 sites, nodes are connected through a gigabit Ethernet network (1 Gb/s). This platform is hierarchical in terms of latency: a few milliseconds among the clusters, around 0.05 within each cluster.

As our failure detector is hierarchical, with a *local* and a *global* level, the 64 nodes are partitioned into 4 *local groups*, one in each cluster. Within each local group, a special node (mandatory) is responsible for the failure detection at global level (i.e cluster failures).

Even if our algorithms do not require a global clock assumption, for measurements purposes, we assume a global clock. Each node runs a *ntp* (*network time protocol*) client to synchronize its local clock and we assume that the local clock drifts are negligible (as the test period is short, of the order of a few tens of minutes). This assumption stands only for measurements purposes.

Performance metrics. The most important parameter of the failure detector is the **delay between heartbeats**. It defines the time between two successive emissions of an "*I am alive*" message. The failure injection is essentially characterized by the **MTBF** (*Mean Time Between Failure*) and possibly by the correlation between failures described in the test files. The experimental results are essentially expressed in terms of **detection latency**. It corresponds to the elapsed time between a node crash and the moment when the other nodes start suspecting it permanently.

5.2 Preliminary tests.

We started by evaluating the failure injection mechanisms alone. The goal is to assess its ability to inject failures according to a given MTBF following an exponential distribution. To do this, we launch 20 times a test with a MTBF value set to one minute, with no failure dependencies. Before each test, the failure schedule is recomputed in order to obtain mean failure dates. Figure 4 shows that the average number of alive nodes decrease as the time elapses. The experimental results are close to the theoretical ones obtained using an exponential distribution.

In a second experiment, we evaluated the ability of our failure injector to correctly generate *correlated failures*. There again we assume the failures follow the same exponential distribution. Besides, we add a *failure dependency*: all members of *local group 1* (located in Rennes) depend on their mandatory (i.e. they must fail if the mandatory fails). This results in a gap on Figure 5 when *Rennes' mandatory* is killed, as all nodes

in Rennes fail concurrently. After this correlated failure happens, the slope of the curve is smaller. This is due to the fact that the dependency made some failures happen sooner.

We can conclude that the failure injector is able to inject failures according to a given MTBF and may also take into account correlated failures.

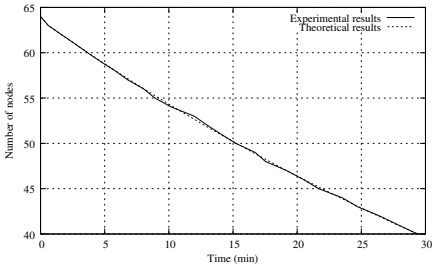

Fig. 4. Failure injection according to MTBF

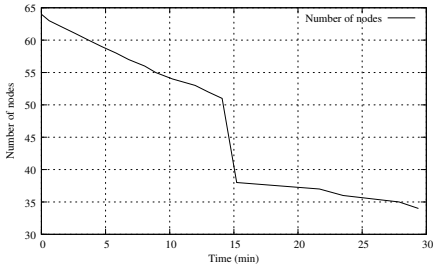

Fig. 5. Correlated failures

5.3 Experimenting with the failure detector

Tradeoff: detection time versus network stress. The failure detector is hierarchical: it provides failure detection in local groups (i.e clusters) and between these groups. We first evaluate the detection time at local level (within local groups) according to the *delay between heartbeats* of the failure detector. To do this evaluation, we set a MTBF of 30 seconds with no failure dependency, and no mandatory failures. During each run of 10 minutes, 18 nodes are killed. Figure 6 shows for each delay between heartbeats the average failure detection time in local groups. The results are very close to what we expected: theoretically, the average detection time is $(delay\_between\_heartbeats/2) + latency$ (and the maximum detection time is almost $(delay\_between\_heartbeats) + latency$). On the other hand, as the delay between heartbeats decreases, the number of messages increases, as shown by figure 7. For a fixed accuracy of the failure detection, there is a tradeoff between detection time and network stress. This is why, through adapters, our failure detector allows multiple applications to share the same heartbeat flow to minimize the network load.

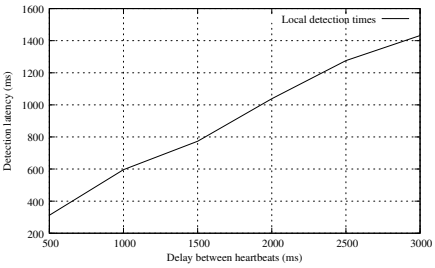

Fig. 6. Local detection times

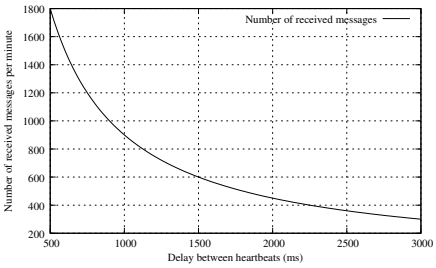

Fig. 7. Network stress

Correlated failures. The aim of this second experiment is to evaluate the detection time at the global level. At this level, the failure detection is done through the local group mandatories. When the failure of a mandatory is detected in a group, a new one is designated to replace it with an average measured nomination delay of *156ms*. Thus, to experiment failure detection at global level, we need to use correlated failures in order to induce the crash of whole local groups. We emulate the failure of sites by introducing a failure dependency between the members of a group (i.e nodes in one site) and their mandatory. By instance, for the Rennes cluster, we add: *<failure dep="RennesInitialMandatory"/>* in the profiles of Rennes' non-initially mandatory nodes.

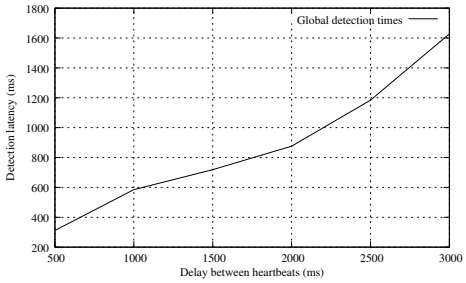

Fig. 8. Global detection times

During a 10 minutes run, 3 out of 4 mandatories are killed. Figure 8 shows the average failure detection times according to the delay between heartbeats. The results are similar to the ones obtained in local groups. The irregularity comes from the fact that less failures occur than for the previous tests. It is important to note that the correlated failures feature is mandatory to perform these measurements as a whole site should fail.

6 Conclusion and future work

In grid environments, building and evaluating fault-tolerant softwares is a hard task. In this paper, we present a test environment providing failure injection features, allowing the developer to control volatility without altering the application code. To illustrate this tool, we evaluate a hierarchical failure detection service. First, our experiments have show that our failure injection tool is able to provide accurate volatility control in a reproducible manner. This allowed us to evaluate a hierarchical failure detection service by emulating independent and correlated failures. In each of these two cases, we have run multiple experiments for different configurations of the failure detector. The results show that the faults are efficiently detected. To the best of our knowledge, no failure detectors have been experimented in the past using automated failure injection on grid platforms.

We plan to further enhance our test environment by adding support for message loss injection. This can be done through network emulation tools like Dummynet [16] or NIST Net [10]. The failure description language will be extended accordingly, in

order to incorporate message loss description. Furthermore we will use this test environment to evaluate the fault tolerance mechanisms of higher-level grid service (e.g. the JUXMEM [5] data sharing service).

References

1. Grid'5000 project. `http://www.grid5000.org`.
2. The PARIS research group. `http://www.irisa.fr/paris`.
3. Guillermo A. Alvarez and Flaviu Cristian. Centralized failure injection for distributed, fault-tolerant protocol testing. In *International Conference on Distributed Computing Systems*, pages 0–10, 1997.
4. Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. Going large-scale in P2P experiments using the JXTA distributed framework. In *Euro-Par 2004: Parallel Processing*, number 3149 in Lect. Notes in Comp. Science, pages 1038–1047, Pisa, Italy, August 2004. Springer-Verlag.
5. Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation: Practice and Experience*, (17), September 2006. To appear. Available as RR-5467.
6. Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, 1993.
7. Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 354–363, Washington, DC, June 2002.
8. Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks*, San Francisco, CA, USA, june 2003.
9. A Collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC. The ns manual (formerly ns notes and documentation). `http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf`, 2003.
10. Mark Carson and Darrin Santay. NIST Net - a Linux-based network emulation tool. 2004. To appear in special issue of Computer Communication Review.
11. Henry Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–441, Brisbane, Australia, 2001.
12. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996.
13. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, apr 1985.
14. William Hoarau and Sébastien Tixeuil. Easy fault injection and stress testing with fail-fci, January 2006.
15. Marc Little and Daniel McCue. Construction and use of a simulation package in c++. Technical Report 437, University of Newcastle upon Tyne, June 1993.
16. Luigi Rizzo. Dummynet and forward error correction. In *1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998. FREENIX track.
17. Jeffrey Voas, Frank Charron, Gary McGraw, Keith Miller, and Michael Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73–83, 1997.
18. Jeffrey Voas, Gary McGraw, Lora Kassab, and Larry Voas. A 'crystal ball' for software liability. *Computer*, 30(6):29–36, 1997.
19. JXTA Distributed Framework. `http://jdf.jxta.org/`, 2003.