Easy Fault Injection and Stress Testing with FAIL-FCI

William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles

LRI-CNRS 8623 et INRIA Grand Large
{hoarau,tixeuil}@lri.fr

Abstract. In a network consisting of several thousands computers, the occurrence of faults is unavoidable. Being able to test the behavior of a distributed program in an environment where we can control the faults (such as the crash of a process) is an important feature that matters in the deployment of reliable programs.
In this paper, we extend FAIL-FCI (for Fault Injection Language, and FAIL Cluster Implementation, respectively), a software tool that permits to elaborate complex fault scenarios in a simple way, while relieving the user from writing low level code. In particular, we show that not only we are able to fault-load existing distributed applications (as used in most current papers that address fault-tolerance issues), we are also able to inject *qualitative faults*, *i.e.* inject specific faults at very specific moments in the program code of the application under test. Finally, and although this was not the primary purpose of the tool, we are also able to inject specific patterns of workload, in order to stress test the application under test. Interestingly enough, the whole process is driven by a simple unified description language, that is totally independent from the language of the application, so that no code changes or recompilation are needed on the application side.

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 expected that Grid middleware is itself reliable and provides a comprehensive support for fault-tolerance mechanisms, such as 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 is 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 present a software that can be used both for software fault-injection and for stress testing of distributed applications, which are the basis for dependability benchmarking in Grid Computing.

In a network including several thousands machines, the appearance of faults is unavoidable. Some applications (for example peer to peer applications) involve a considerable number of users, e.g. to exchange files or to execute long calculations (SeTi@Home, Decrypthon, Xtremweb, Boinc, etc.). For those applications, the appearance and disappearance of participating machines are unpredictable, very frequent and occur eventually while the application is run. It is particularly difficult to study the functioning of large-scale distributed programs: it would be necessary to have a considerable number of computers and engineering power to execute the software in an actual situation, to measure the performances or to detect the defects. With the difficulty to set up such experiments and the fact that fault occurrences in such systems is not neither controllable nor predictable (it is also difficult to compare various solutions), two other approaches are possible: simulation and emulation. Simulation allows complete control of the runtime environment, but fails in imitating the actual behavior of all components in the system. Emulation consists in using a small network to reproduce the behavior of a large-scale network. However, it is not enough to emulate the machines used by the participants: it is also necessary to reproduce their behavior.

Testing the validity of fault-tolerant software and measuring the impact on performance of occurring faults requires being able to control those faults. Indeed, a fundamental result [5] shows that in an asynchronous distributed system (where the relative speeds of the processors are not known and unbounded), it is impossible to solve the consensus problem (all processors terminate agreeing on some initial value) when there is as little as one faulty process, even when the considered fault is as simple as a crash fault. The reason for this is that the decided value can depend on just one process and that in an asynchronous system, it is impossible to distinguish between a crashed process and a very slow one. When an application is run on a cluster, it is likely that machines will run roughly at the same speed (for example a one to ten ratio on the relative speeds of the processors makes it easy to solve the consensus problem), so the considered system is actually synchronous. Afterwards, when the application is then run at a larger scale (e.g. in an Internet-like setting) where the strong synchrony hypothesis does not hold any more, crucial issues related to fault-tolerance and asynchronous settings have been overlooked.

2 Related works

When considering solutions for software fault injection in distributed systems, there are several important parameters to consider. The main criterion is the usability of the fault injection platform. If it is more difficult to write fault scenarios than to actually write the tested applications, those fault scenarios are likely to be dropped from the set of performed tests. The issues in testing component-based distributed systems have already been described and methodology for testing components and systems has already been proposed [6,**?**]. However, testing for fault tolerance remains a challenging issue. Indeed, in available systems, the fault-recovery code is rarely executed in the test-bed as faults rarely get triggered. As the ability of a system to perform well in the presence of faults depends on the correctness of the fault-recovery code, it is mandatory to actually test this code. Testing based on fault-injection can be used to test for fault-tolerance by injecting faults into a system under test and observing its behavior. The most obvious point is that simple tests (e.g. every few minutes or so, a randomly chosen machine crashes) should be simple to write and deploy. On the other hand, it should be possible to inject faults for very specific cases (e.g. in a particular global state of the application), even if it requires a better understanding of the tested application. Also, decoupling the fault injection platform from the tested application is a desirable property, as different groups can concentrate on different aspects of fault-tolerance. Decoupling requires that no source code modification of the tested application should be necessary to inject faults. Also, having experts in fault-tolerance test particular scenarios for application they have no knowledge of favors describing fault scenarios using a high-level language, that abstract practical issues such that communications and scheduling. Finally, to properly evaluate a distributed application in the context of faults, the impact of the fault injection platform should be kept low, even if the number of machines is high. Of course, the impact is doomed to increase with the complexity of the fault scenario, e.g. when every action of every processor is likely to trigger a fault action, injecting those faults will induce an overhead that is certainly not negligible.

Several fault injectors for distributed systems already exist. Some of them are dedicated to distributed real-time systems such as DOCTOR [8]. ORCHESTRA [3] is a fault injection tool that allows the user to test the reliability and the liveliness of distributed protocols. ORCHESTRA is a "Message-level fault injector" because a fault injection layer is inserted between two layers in the protocol stack. This kind of fault injector allows injecting faults without requiring the modification of the protocol source code. However, the expressiveness of the faults scenario is limited because there is no communication between the various state machines executed on every node. Then, as the faults injection is based on exchanged messages, the knowledge of the type and the size of these messages is required. Nevertheless, those approaches do not fit the cluster and Grid category of applications.

The NFTAPE project [13] arose from the double observation that no tool is sufficient to inject all fault models and that it is difficult to port a particular tool to different systems. Although NFTAPE is modular and very portable, the choice of a completely centralized decision process makes it very intrusive (its execution strongly perturbs the system being tested). Finally, writing a scenario quickly becomes complex because of the centralized nature of the decisions during the tests when they imply numerous nodes.

LOKI [2] is a fault injector dedicated to distributed systems. It is based on a partial view of the global state of the distributed system. An analysis a posteriori is executed at the end of the test to infer a global schedule from the various partial views and then verify if faults were correctly injected (i.e. according to the planned scenario). However, LOKI requires the modification of the source code of the tested application. Furthermore, faults scenario are only based on the global state of the system and it is difficult (if not impossible) to specify more complex faults scenario (for example injecting "cascading" faults). Also, LOKI there is no support for randomized fault injection.

In [10] is presented Mendosus, a fault-injection tool for system-area networks that is based on the emulation of clusters of computers and different network configurations.

Finally in [12] is presented a fault-injection tool that was specially developed to assess the dependability of Grid (OGSA) middleware. However, the tool described in that paper is very limited since it only allows the injection of faults in the XML messages in the OGSA middleware, which seems to be a bit far from the real faults experienced in real systems.

Recently, the FAIL-FCI architecture [9] was proposed. This solution addresses most of the drawbacks of previous approaches, and is overviewed in the next section.

3 Overview of FAIL-FCI

In this section, we describe the FAIL-FCI framework that is presented in [9]. For further explanations, please refer to the original paper. First, FAIL (for Fault Injection Language) is a language that permits to easily described fault scenarios. Second, FCI (for FAIL Cluster Implementation) is a distributed fault injection platform whose input language for describing fault scenarios is FAIL. The FAIL language allows defining fault scenarios. A scenario describes, using a high-level abstract language, state machines which model fault occurrences. The FAIL language also describes the association between these state machines and a computer (or a group of computers) in the network.

The FCI platform is composed of several building blocks:

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

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

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

The FAIL-FCI approach is based on the use of a software debugger. Like the Mantis parallel debugger [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 are kept to a minimum to guarantee low overhead of the fault injection platform (in our approach, the debugger is only used to trigger and inject software faults). The tested application can be interrupted when it calls a particular function or upon executing a particular line of its source code. Its execution can be resumed depending on the considered fault scenario. With FCI, every physical machine is associated to a fault injection daemon. The fault scenario is described in a high-level language and compiled to obtain a C++ code which will be distributed on the machines participating to the experiment. This C++ code is compiled on every machine to generate the fault injection daemon. Once this preliminary task has been performed, the experience is then ready to be launched. The daemon associated to a particular computer consists in:

1. a state machine implementing the fault scenario,
2. a module for communicating with the other daemons (e.g. to inject faults based on a global state of the system),
3. a module for time-management (e.g. to allow time-based fault injection),
4. a module to instrument the tested application (by driving the debugger), and
5. a module for managing events (to trigger faults).

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

4 Demonstrating Fault Injection and Stress Testing with FAIL-FCI

In [9], the vast majority of experiments were made on a custom made distributed program, for which both source code and expertise were available. Moreover, tests only dealt with the overhead of the FAIL platform, and simply showed that this overhead was, for practical purposes, negligible.

In this section, we use FAIL-FCI to inject fault and stress test a readily available distributed application: XtremWeb [4]. The remaining of the section is organized as follows: Section 4.1 reviews the XtremWeb platform that we use for our tests. Section 4.2 describes the particular settings that we use for our experiments. Sections 4.3, 4.4, and 4.5 describe respectively how to use FAIL-FCI for quantitative fault injection, qualitative fault injection, and stress testing.

4.1 Overview of XtremWeb

XtremWeb is a general purpose platform that can be used for high performance distributed calculus. A list of tasks (or jobs) is described by the user and then distributed over the different available nodes of the system. The basic operating mode of XtremWeb is based on a participants community, *e.g.* it allows a High School, a University or a Company to setup and run a Global Computing or Peer to Peer distributed system for either a dedicated application or a whole range of applications. The original XtremWeb application is written in Java, but we used here the C++ version of the software, that is expected to achieve the most efficient results. The XtremWeb tool is divided into three modules:

the dispatcher centralizes, organizes and distributes the tasks,
the client proposes a set of tasks to the manager,
a set of workers regularly requests a work from the manager.

Like other distributed system platforms, the XtremWeb platform uses *(i)* remote resources (PCs, workstations, servers) connected to the Internet, or *(ii)* a pool of resources (PCs, workstations, servers) inside a LAN.

4.2 Technical Settings

Hardware Settings The experiments were performed on 30 machines running Linux 2.6.7 (except for the XtremWeb dispatcher and the XtremWeb client which were run on a different machine). Thirteen machines were equipped each with a 2083 MHz processor and 885 Mb RAM. Six machines were equipped each with two 1533 MHz processors and 885 Mb RAM. Eleven machines were equipped each with a 1533 MHz processor and 885 Mb RAM. The dispatcher and the client were run on a machine equipped with a 2995 MHz processor and 527 MB RAM. This last machine was running Linux 2.6.8. All machines were connected using a 100 Mbps Ethernet network.

XtremWeb Settings For all performed experiments, the XtremWeb dispatcher and client were placed on a single machine (`lri7-209`). The workload of the client does not really influence the dispatcher: indeed, the client and dispatcher almost run in a sequential way; the client first gives a list of jobs to the dispatcher at the beginning of the run, and the dispatcher notices the client when the jobs have been completed and results are available. The workers are each placed on a dedicated machine in the cluster (30 such machines).

Before a particular test starts, the dispatcher is started, as well as all workers. Then, the client is started (the staring time of the client is referred to as the *test begin time*). When the client exits (after receipt of an acknowledgement from the dispatcher), this time is referred to as the *test end time*.

The particular application that is run with XtremWeb is *POV-Ray*, which creates three-dimensional, photo-realistic images using a rendering technique called ray-tracing. For our purpose, a task consists in calculating a particular picture using POV-Ray. This operation is requested 40 times. When the dispatcher receives a task request from a worker, it sends all necessary information to perform the calculus of one picture.

4.3 Quantitative Fault Injection

We first design a probabilistic fault scenario, to quickly get a quantitative view of the fault tolerance capabilities of XtremWeb. We assume that both the dispatcher and the client are not subject to faults (*i.e.* some tasks can be submitted, and some results can be returned). XtremWeb workers are run on the remaining 30 machines that are subject to faults. The running time is the time between the client is started and the results are collected. The fault model is as follows: every x seconds, each of the XtremWeb workers may crash (and cease functioning) with probability y. Yet, we wish to ensure that there exists a particular worker that can not crash, in order to guarantee that the running time is always finite. The above scenario can be expressed in a surprisingly terse way using the FAIL language (with $x = 5$ and $y = 10\%$ here):

```
spyfunc main;

Daemon ADV1 {
node 1:
       before(main) -> continue, !ok(G1[1]), !go(G1), goto 2;
node 2:
}

Daemon ADV2 {
node 1:
       before(main) -> stop, goto 2;
node 2:
       ?ok -> continue, goto 4;
       ?go -> continue, goto 3;
node 3:
       always int x = FAIL_RANDOM(1,100);
       always time_g timer = 5;
       timer && x <= 10 -> halt, goto 4;
       timer && x > 10 -> continue, goto 3;
node 4:
}

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

Group G1 {
 size = 30;
 program = "WorkerStatic -i lri7-209";
 daemon = ADV2;
}
```

We now informally describe the aforementioned source code. First, two automata are defined: `ADV1` and `ADV2`, then automata `ADV1` is associated to one computer `P1` (that will execute dummy code), while `ADV2` is associated to 30 machines (that form the `G1` group), each executing the executable file `WorkerStatic` with the same parameters.

`ADV2` runs as follows: the daemon first wait that the program has loaded, but before the `main` function is executed, the program is halted. The execution continues when the `ADV1` automata sends either the 'ok' or the 'go' message. Now, the `ADV1` simply send the 'ok' message to a particular automata in the `G1` group, and then a 'go' messages to all automata in the `G1` group. So, one automata in the `G1` group first receive a 'ok' message, moves to a new state (`node 4`), from which it simply runs the program, ignoring subsequent messages and events. So the corresponding worker process runs smoothly afterwards. In contrast, the other processes in the `G1` group receive the 'go' message. As a result, the state is changed (`node 3`) so that they now receive timer events (every five seconds). When the time expires, with 10% probability, the process under test crashes, while with 90% probability, the process continues its computation for another 5 seconds. Further details about the FAIL language can be found in [1].

We carried out this test using two values for x (5 and 10 seconds) and y varying from 10% to 90% with increments of 10%. The obtained results regarding the execution time of the total set of jobs are summarized in Figure 1. As can be seen in Figure 1, for some settings, the calculus did not terminate, due to a malfunction of the XtremWeb dispatcher (recall that this process was *not* purposely given crash order by the FAIL-FCI framework). So, we also collected information about the dispatcher failure during the tests, and these results are presented in Figure 2.

Before running the tests, one would expect that the two curves would increase, with an extra increasing gap between them. When there are no crashes, the time used to complete the execution of the tasks is approximately 25 seconds. Starting with a probability of failure of 40%, the results are as expected, but for lower probabilities, the rate of fault appearance does not significantly change the execution time. Also, when failures occur only every ten seconds, there is some kind of equilibrium (between 40% probability and 60% probability) where the execution time does not vary much. This equilibrium reflects the fact that if more failures occurred so far, it means that fewer failures are likely to appear (because there are fewer healthy machines yet) in the future.

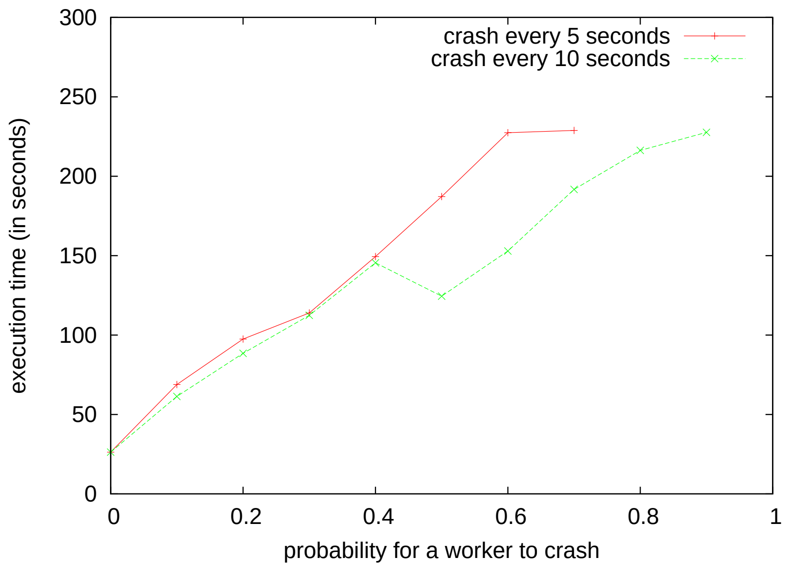

Fig. 1. Impact of workers crash on execution time

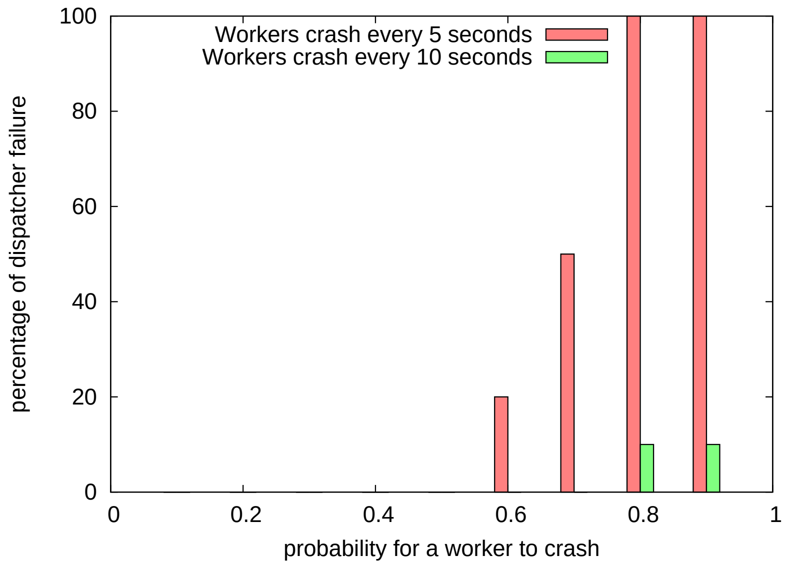

Fig. 2. Impact of workers crashes on dispatcher failure

When some tests did not finished, we detected in these cases that the dispatcher was still running but was not available anymore (*i.e.* workers could not communicate with the dispatcher to notify they completed their task). Figure 2 shows that starting from a 70% probability for a worker to crash every five seconds, the dispatcher ends up failing in 50% of the runs. Also, from a probability of 80% for a worker to crash every five second, the dispatcher *always* fail. This failure of the dispatcher probably reveals a bug that would extremely rarely occur in a real cluster, these fault rates being pretty extreme: every 5 seconds, 80% of the nodes crash!

4.4 Qualitative Fault Injection

The quantitative evaluation that was presented in Section 4.3 could also be handled, although if a more tedious and cumbersome way, through proper scripting of the distributed application. In this section, we go one step further and provide qualitative evaluation of the faults that could potentially hit the system. In more details, we are interested here in which part of the XtremWeb clients the fault occur. In particular, we consider the following four possible logical states for a particular XtremWeb worker:

1. *job received*: the XtremWeb worker has received a job to perform from the XtremWeb dispatcher,
2. *after calculus*: the XtremWeb worker has finished to perform its task,
3. *job finished*: the XtremWeb worker has notified the XtremWeb dispatcher that it completed its job,
4. *job completed*: the XtremWeb worker has sent the XtremWeb dispatcher the results of the completed task.

Our goal in this series of tests is to fix the number of workers (30) and the crash probability (40%), but a worker may only fail at precise points in its program code: the points that correspond to entering the four states mentioned above. The corresponding FAIL program (*i.e.* fault scenario) is as follows (considering that faults would only occur when the worker is in the state *job completed*):

```
spyfunc main;
spyfunc Protocol::DataSaved;
spyfunc Protocol::release;

Daemon ADV1 {
node 1:
       before(main) -> continue, !ok(G1[1]), !go(G1), goto 2;
node 2:
}

Daemon ADV2 {
node 1:
       before(main) -> stop, goto 2;
node 2:
       ?ok -> continue, goto 5;
       ?go -> continue, goto 3;
node 3:
       always int x = FAIL_RANDOM(1,100);
       before(Protocol::DataSaved) && x <= 40 -> continue, goto 4;
       before(Protocol::DataSaved) && x > 40 -> continue, goto 3;
node 4:
       before(Protocol::release) -> stop, goto 5;
node 5:
}

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

Group G1 {
 size = 30;
 program = "WorkerStatic -i lri7-209";
 daemon = ADV2;
}
```

As in Section 4.3, there are two automata `ADV1` and `ADV2` that are dispatched in the same way as before. The same trick to get at least one working worker is also used (using the 'ok' and 'go' messages). the key difference is the use of breakpoints to get back control over the processes when a particular function is reached. In this scenario, the methods *DataSaved* and *release* of the class *Protocol* are watched. The state *job completed* is reach after the call to the method *DataSaved* has completed and just before the call of the method *release*. Note that the *release* method is called often and in various contexts in the XtremWeb worker code, but only corresponds to the *job completed* state after the *DataSaved* method has been executed.

The obtained results are summarized in Figure 3. In this Figure, the category *without fault* refers to the test without injecting faults (for comparison purpose). For every of the four aforementioned possible states of the workers, two kinds of faults are considered:

1. suspending the process (using `stop` in the FAIL language) to simulate an overloaded machine,
2. crashing the process (using `halt` in the FAIL language).

We did not collect information about possible dispatcher failures, since no crashes were observed (this was expected, because the probability of crashes was 40%).

It was expected that injecting `stop` faults would induce worse performance than injecting `halt` faults (because in the first case, the other end of the TCP connexion, *i.e.* the dispatcher, is not notified by the network layer that something bad happened, while in the second case, it usually is). This was confirmed by the

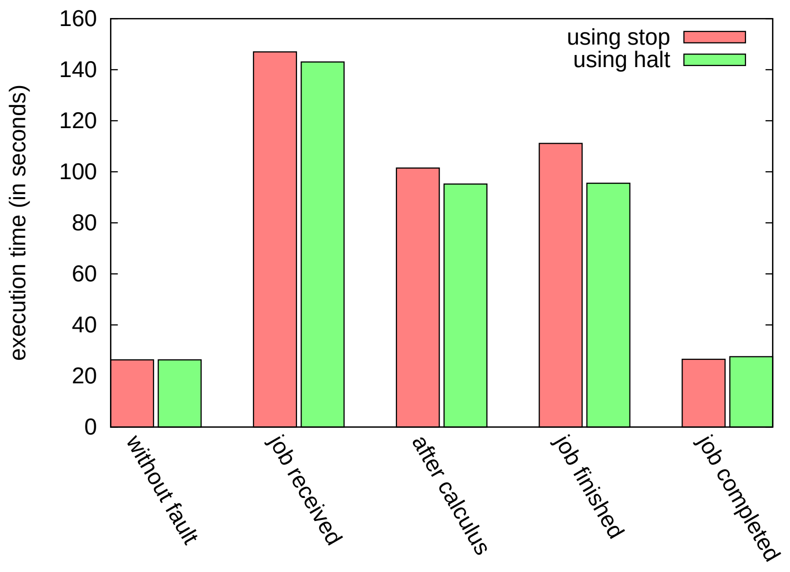

Fig. 3. Impact of the state of a worker when crashing

results we obtained. We also expected that the later the injection (but yet before the results are sent to the dispatcher), the more time it would take to complete the calculus. However, and surprisingly, if the workers crash before even starting a calculus, the performance is worse than if it crashes after the computation. This behavior is probably due to an misconception in the XtremWeb dispatcher, that does not expect failures just after the job was sent (at that time, it is probably not watching the TCP connexion with the client, while it is when the job is near to completion). We also remark that if a worker crashes after a job is completed the worker notified the controller that the results are available), then the performance is almost the same as if no faults were injected.

4.5 Stress Testing

Sections 4.3 and 4.4 showed how FAIL-FCI can be used to obtain failure resilience capabilities of distributed applications using a unified approach for both quantitative and qualitative analysis. We now show that the same tool can be set up to handle stress testing as well. For this purpose, we use a slightly different scenario. The set of tasks is the same as before, and the XtremWeb client and dispatcher are still on the same machine. The *test begin time* is the time when both the XtremWeb client and dispatcher are up and running, waiting for workers to perform the tasks. Then, a particular XtremWeb worker is launched into action with probability y every x seconds. When the client exits (after having received the acknowledgement from the dispatcher), the current time is taken as the *test end time*.

The corresponding scenario written using the FAIL language is as follows (considering that $x = 1$ and $y = 10\%$):

```
spyfunc main;

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

Daemon ADV2 {
node 1:
       before(main) -> stop, goto 2;
node 2:
       ?go -> stop, goto 3;
node 3:
       always int x = FAIL_RANDOM(1,100);
       always time_g timer = 1;
       timer && x <= 10 -> continue, goto 4;
       timer && x > 10 -> stop, goto 3;
node 4:
}

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

Group G1 {
 size = 30;
 program = "WorkerStatic -i lri7-209";
 daemon = ADV2;
}
```

We performed tests varying x from 1 to 9 seconds (with increments of 2 seconds), and varying y from 10% to 100%. The obtained results regarding the global execution time are summarized in Figure 4. We did not collect information about dispatcher failure, since none appeared.

It was expected that the shape of the curves would have a “U” form for at least the test with $x = 1$ (getting workers to the job every second): if few workers arrive at the same time, the performance is low, if several workers arrive at the same time, this is manageable by the dispatcher and the performance is good, if many workers arrive at the same time, the dispatcher would be more overloaded and the overall performance would be worse that with fewer workers. In fact, all curves are decreasing, which means that the more workers you get, the faster is the completion of the calculus. It also means that the C++ version of XtremWeb

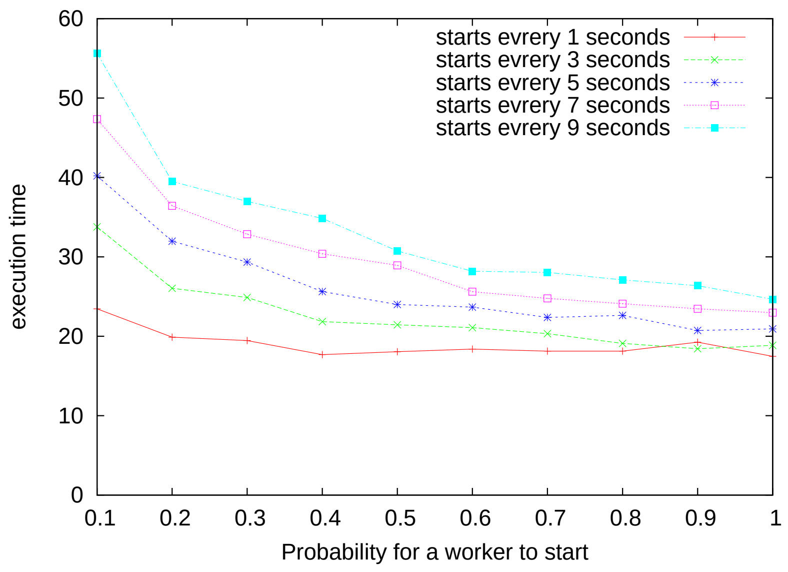

Fig. 4. Stress testing

can handle 30 new workers arriving at the same time with no problems (this is the case where $y = 100\%$).

5 Concluding Remarks

We proposed a unified approach for fault injection and stress testing distributed applications. Fault injection can be made using a *quantitative* approach (as in most related studies) as well as the more original *qualitative* approach, where precise faults are inserted at precise logical states of the application under test. Although the set of possible fault injection is extremely large, the language that describes the faults scenario is high level and independent from the language used in the application. This enables decoupling between the application programmers and the test specifiers, so that expertise is used in the proper domain.

As a proof of concept, we also showed that the same specification language and fault injection tool could also be used as a stress test platform. While the preliminary tests we performed caused no problem, they actually raised a number of interesting open questions. The main one relates to the use of FAIL-FCI at a larger scale (for purpose of stress testing). We are currently investigating using our fault injector in larger systems, typically by using emulation schemes, within the Grid eXplorer [7] project platform. Further studies are needed to see the effect of correlated faults injection (such as those occurring when a virus is spread throughout the network). Finally, extra development is needed to integrate FCI with self-distributing applications (such as those based on MPI), since our current implementation assumes that distributed applications are launched through a `ssh`-like mechanism.

References

1. `http://www.lri.fr/~hoarau/fail.html`.
2. 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.
3. S. Dawson, F. Jahanian, and T. Mitton. Orchestra: A fault injection environment for distributed systems. In *In 26th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 404–414, Sendai, Japan, June 1996.
4. G. Fedak, C. Germain, V. Néri, and F. Cappello. Xtremweb: A generic global computing system. In *Proceedings of IEEE Int. Symp. on Cluster Computing and the Grid*, 2001.
5. M. Fisher, N.A. Lynch, and M.J. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 1985.
6. S. Ghosh and A. Mathur. Issues in testing distributed component-based systems, 1999.
7. `http://www.lri.fr/~fci/GdX`.
8. S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems, 1995.
9. W. Hoarau and Sbastien Tixeuil. A language-driven tool for fault injection in distributed applications. In *In Proceedings of the IEEE/ACM Workshop GRID 2005*, November 2005. also available as LRI Research Report 1399, february 2005, at `http://www.lri.fr/~hoarau/fail.html`.
10. X. Li, R. Martin, K. Nagaraja, T. Nguyen, and B. Zhang. Mendosus: A san-based fault-injection test-bed for the construction of highly available network services, 2002.
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. J.Xu. N. Looker. Assessing the dependability of ogsa middleware by fault-injection. In *Proc. 22nd Int. Symposium on Reliable Distributed Systems*. SRDS, 2003.
13. D.T. Stott and al. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *In Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91–100, March 2000.