RAPPORT DE RECHERCHE

L R I

A LANGUAGE-DRIVEN TOOL FOR FAULT INJECTION IN DISTRIBUTED SYSTEMS

HOARAU W / TIXEUIL S

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

02/2005

Rapport de Recherche N° 1399

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

A language-driven tool for fault injection in distributed systems

William Hoarau Sébastien Tixeuil

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 present FAIL (for FAult Injection Language), a language that permits to elaborate complex fault scenarios in a simple way, while relieving the user from writing low level code. Besides, it is possible to construct probabilistic scenarios (for average quantitative tests) or deterministic and reproducible scenarios (for studying the application's behavior in particular cases). We also present FCI, the FAIL Cluster Implementation, that consists of a compiler, a runtime library and a middleware platform for software fault injection in distributed applications. FCI is able to interface with numerous programming languages without requiring the modification of their source code, and the preliminary tests that we conducted show that its effective impact at runtime is low.

Résumé

Dans un réseau constitué de plusieurs milliers d'ordinateurs, l'apparition de fautes est inévitable. Être capable de tester le comportement d'applications réparties dans un environnement où il est possible de contrôler les fautes (telles que le crash d'un processus) est une caractéristique importante pour la mise en œuvre de programmes fiables.

Dans ce rapport, nous présentons FAIL (pour FAult Injection Language), un langage qui permet d'élaborer des scénarios de fautes complexes de manière simple, tout en évitant à l'utilisateur d'écrire du code de bas niveau. De plus, il est possible de construire des scénarios probabilistes (pour des tests moyens quantitatifs) ou déterministes et reproductibles (pour tester le comportement de l'application dans des cas particuliers). Nous présentons également FCI (pour FAIL Cluster Implementation), qui comporte un compilateur, une bibliothèque d'éxecution et un intergiciel pour l'injection de fautes par programme dans les applications distribuées. FCI est capable de s'interfacer avec de nombreux langages de programmation sans requérir de modifier le code source des applications, et les tests préliminaires que nous avons menés montrent que son impact effectif sur le temps d'exécution est faible.

Chapter 1

Introduction

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

It is particularly difficult to study the functioning of large-scale distributed programs : it would be necessary to have a considerable number of computers and engineering power to execute the software in an actual situation, to measure the performances or to detect the defects. With the difficulty to set up such experiments and the fact that fault occurrences in such systems is not controllable nor predictable (it is also difficult to compare various solutions), two other approaches are possible: simulation and emulation. Simulation consists in using a single computer to simulate the behavior of a network of computers. It 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. Current platforms for emulation use, *e.g.*, one thousand powerful machines on a fast network, and every single machine emulates, using proper software, a dozen machines of lower power, each "believing" to be independent. Emulation also limits emulated machines' access to the network devices in order to reproduce the bigger system conditions. Such a system allows to control the programs execution environment, and to modulate it according to the network that we wish to emulate. 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 to be able to control those faults. Indeed, a fundamental result [3] 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 relative speeds makes 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.

In this paper, we concentrate on studying crash (with potential restart) faults in asynchronous distributed applications. However, we target platforms that are widely available (*e.g.* clusters or grids), but that might actually exhibit synchronous (or partially synchronous) behavior. To this purpose, we provide a scheme for distributed *software* fault injection. While there exist hardware methods (that consist in injecting faults into the actual hardware), they are difficult to set up, expensive, unstable, and can damage the actual hardware. On the other side, software methods (that consist in injecting faults using proper software) are easier to set up, may be reproduced, and do to risk to damage the hardware used for the experiment. In the context of cluster of grid platforms, software methods have the additional advantage to be compatible with emulation mechanisms.

Chapter 2

Related works

When considering solutions for software fault injection in distributed systems, there are several important parameters to consider. The main criteria 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 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.

We now review previous approaches relatively to those criteria.

2.1 ORCHESTRA

ORCHESTRA [2] is a fault injection tool. It allows the user to test the reliability and the liveliness of distributed protocols. A fault injection layer is inserted between the the tested protocol layer and the lower layers, and allows to filter and manipulate messages exchanged between the protocol participants. Messages can be delayed, lost, reordered, duplicated, modified and new messages can be spontaneously introduced into the tested system to bring it into a particular global state.

The reception script and the sending script are written in Tcl language and determine which operations are to be performed on received/sent messages. These scripts are specified with state machines. Transitions in these state machines are driven by the type of the message, its contents, the history of received messages or other information that was previously collected during the test (*e.g.* local time, number of received messages, etc.). The message modifications are specified using a user-defined script. The resulting message is passed to the next layer of the protocol stack.

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 to inject faults without requiring the modification of the protocol source code. However, the user has to implement his faults injection layer for the protocol he uses. 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. In summary, ORCHESTRA is well adapted for the study of network protocols, but is too complex to use and not expressive enough to study distributed applications.

2.2 NFTAPE

The NFTAPE project [8] 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. NFTAPE [8] provides mechanisms for fault-injection, triggering injections, producing workloads, detecting errors, and logging results. Unlike other tools, NFTAPE separates these components so that the user can create his own fault injectors and injection triggers using the provided interfaces. NFTAPE introduces the notion of *Lightweight Fault Injector* (*LWFI*). LWFIs are simpler than traditional fault injectors, because they don't need to integrate triggers, logging mechanisms, and communication support. This way, NFTAPE can inject faults using any fault injection method and any fault model. Interfaces for the other components are also defined to facilitate portability to new systems.

In NFTAPE, the execution of a test scenario is centralized. A particular computer, called the control host, takes all control decisions. this computer is generally separated from the set of computers that execute the test. It executes a script written in Jython (Jython is a subset of the Python language) which defines the faults scenario. All participating computers are attached to a process manager which in turn communicates with the control host. The control host sends commands to process managers according to the fault scenario. When receiving a command, the process manager executes it. At the end of the execution or if a crash occurs, the process manager notifies the control host by sending a message. All decisions are taken by the controller, which implies that every fault triggered at every nodes induces a communication with the controller. Then, according to the defined scenario, the controller sends a fault injection message to the appropriate process manager which can then inject the fault.

Although NFTAPE is modular and very portable, the choice of a completely centralized

decision makes it very intrusive (its execution strongly perturbs the system being tested) if a considerable amount of resources is not dedicated to the controller. Scalability is then an issue if the controller is to manage a large number of process managers, and NFTAPE is run on regular clusters or grids (where there is no overpowered node that can take care of controlling the fault injector). Furthermore, implementing tests can end up in being very difficult for the user (the user may have to implement every component).

2.3 LOKI

LOKI [1] is a fault injector dedicated to distributed systems. It is based on a partial view of the global state of the distributed system. The faults are injected based on a global state of the 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). Normally, injecting faults based on a global state of a distributed application leads to a high impact on its execution time. However, the technique used by LOKI allows to verify the validity of the injected faults while limiting their impact on the execution time.

In LOKI, every process of the distributed system is attached to a LOKI runtime to form a *node*. The LOKI runtime includes the code which manages the maintenance of the partial view of the global state, injects faults when the system arrives in a particular state, collects information about state changes and faults injections. LOKI proposes three execution modes: centralized, partially distributed and fully distributed. The definition of a scenario in LOKI is made by specifying the state machine used to maintain the local state, faults that are to be injected, and implementing a probe used to instrument the application in order to detect events (events are used to trigger faults). The user has to define fault identifiers and associate them to global states of the tested application, and then has to implement the probe by modifying the application source code. Calls to functions of the LOKI library must be inserted into the source code of the tested application to notify the LOKI runtime about events so that the appropriate state is reached. Also, to inject faults, the user has to implement the `injectFault()` function and insert it into the source code of the tested application.

LOKI is the first fault injector for distributed systems that allows to inject faults based on a global state of the system and to verify if these faults were correctly injected. However, it 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 doesn't provide any support for randomized fault injection.

Chapter 3

Our solution

Our solution consists in two main components, that are described separately. 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 [4] which aims at emulating large-scale networks on smaller clusters or grids.

The FAIL language allows to define 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 (see Figure 3.1) is composed of several building blocks:

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

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

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

Our approach is based on the use of a debugger to trigger and inject the 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.

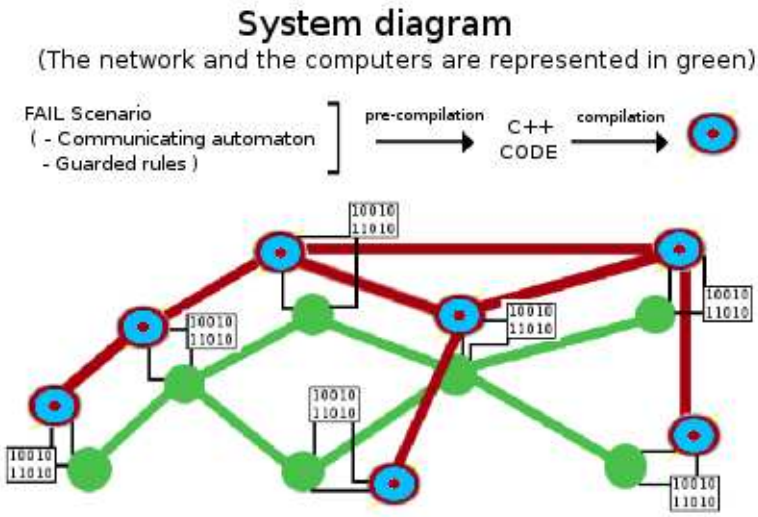

Figure 3.1: The FCI fault injection platform

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 *(i)* a state machine implementing the fault scenario, *(ii)* a module for communicating with the other daemons (*e.g.* to inject faults based on a global state of the system), *(iii)* a module for time-management (*e.g.* to allow time-based fault injection), *(iv)* a module for the instrumenting the tested application (by driving the debugger), and *(v)* 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

Criteria	ORCHESTRA	NFTAPE	LOKI	FAIL + FCI
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

Figure 3.2: Fault injection systems comparison

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.

In a nutshell, the essential differences between our approach and the aforementioned works are summarized in Figure 3.2.

3.1 The FAIL language

In FAIL, fault scenarios are described with named state machines that can be associated to one or several machines of the system. State machines are described using guarded commands like `guard -> action`. A guard is a predicate on the current state of the state machine, the time or the occurrence of events (*e.g.* exceptions being thrown, a particular function of the tested program being executed, a particular line of the source code - if it is available - of the tested program being reached, a particular message from another state machine being received, etc.). An action is simply a sequence of instructions, and can also bring the machine into a new state. When the predicate which corresponds to the guard is true, the corresponding rule is activatable. When an activatable rule is activated, the corresponding action is executed.

The explicit use of states for the state machines allows to ease programming by avoiding the use of too many variables. The intrusion of the faults injector is limited because all guards are not systematically evaluated. Only the guards that correspond to the current state of the machine may generate an event and are thus the only ones to be evaluated. The use of messages for communication between state machines allows to inject faults according to a global state of the system. Of course, if all faults are to be triggered according to a global state machine, the intrusion is likely to increase significatively.

Besides, the guards can depend on random variables to allow the injection of probabilistic faults. Several functions for random numbers generation are predefined, but the user can use other functions of an existing library if it is necessary to make probabilistic tests according to a specific plan.

The possible instructions in the action of a guarded command are the sending of messages to other state machines (*e.g.* to notify the arrival of the tested program at a particular state),

assignments of internal variables of the state machine and actions which permit the control of the tested program. These control actions are:

- **stop**: stops the execution of the program,
- **continue**: resume the execution of the program from where it was previously stopped (after a stop action),
- **halt**: stops definitively the execution of the program;
- **restart**: restarts the program in its initial state.

On one hand, stop and continue actions allow to add asynchronism to the system, by acting on the relative speeds of the participating processes of the distributed application. On the other hand, halt and restart actions make it possible to simulate definitive system failures (halt only) or with recovery (halt and restart).

3.1.1 Syntactic elements of the FAIL language

A FAIL automaton is defined using the constructor `Daemon`. Here is a simple example of an automaton written in the FAIL language:

```
Daemon Adv1 { ?ok -> !done, halt; }
```

In this example, `?ok` is a guard, `!done` an instruction and `halt` another instruction. The guard `?ok` means "upon receipt of message `ok`", the instruction `!done` means "I send `done` message" and the instruction `halt` means "the process executed on the local machine which is associated to the FAIL daemon stops definitively".

In a sequence of actions, the actions are separated by commas. The semicolon is used at the end of a rule. Every sequence of actions has to end with the target state the machine will be placed into when all actions are done (using the `goto` keyword). If at the end of a sequence of actions, there is no target state, then the target state is the current state of the automaton.

```
Daemon Adv2 { node 1: ?bye -> stop, goto 2;
              node 2: ?hello -> continue, goto 1; }
```

In this example, when the automaton receives the message `bye`, it interrupts the execution of the tested application. Then, when it receives the message `hello`, it resumes its execution. This mechanism is repeated until the end of the test.

Each automaton is associated to a computer of the system. This association is made using the keyword `Computer`.

```
Computer p1 { daemon = Adv1; }
```

In this example, `Adv1` is associated to the computer which has the identifier `p1`.

In practise, the applications to be tested run on a large number of computers, it is thus necessary to offer the possibility for the user to associate a automaton to a group of computers to facilitate the writing of the scenario. The definition of this group is made using the `Group` keyword.

```
Group g1 { size = 100; daemon = Adv1; }
```

In this example, `Adv1` is associated to 100 computers of the system which are individually accessible through a named table of identifiers `g1`.

The user has access to several constants, variables and predefined functions. They are all prefixed with `FAIL_` and are composed with capital letters and "_". It is possible to access the global time (in seconds) `FAIL_GLOBAL_CLOCK`, the local time (in milliseconds) `FAIL_LOCAL_CLOCK`, a table `FAIL_COMPUTERS` containing the identifiers of all used computers in the system, a function `FAIL_SIZE` taking a table of computers identifiers and returning its size, a function `FAIL_RANDOM(min, max)` returning a uniform random integer taken from the interval [min,max], and a function

```
FAIL_RANDOM_TABC(tab_comp, nb_comp)
```

taking table of computers identifiers `tab_comp` and an integer `nb_comp` and returning a table of identifiers whose size is `nb_comp` and whose elements are chosen randomly among those of `tab_comp`.

The guards are conjunctions of predicates with boolean values. The conjunction is denoted by the operator `&&`. The various predicates which can be part of a guard are the receipt of a message, the test of equality, inequality or difference of a variable and a value, the test of a function call by the tested program or the test of the arrival of the tested program in a particular line of its source code.

```
Daemon Adv3 { int rand = FAIL_RANDOM(1,2);
                         // definition of a random variable.
                         time_g timer = FAIL_GLOBAL_CLOCK + 20;
                         timer && rand == 1 -> halt; }
```

In this example, at the end of 20 seconds of global time, the automaton stops definitively the tested process with a probability of 0.5.

3.1.2 Example

In this example, a distributed program which requires 500 machines, will have to cope with the following fault scenario: every minute, some computers (randomly chosen) will be potentially faulty (they cease to execute the application). The number of potentially faulty computers ranges between 1 and 4. The potentially faulty computers must fail before calling the `send_value` function or when the tested program arrives at the line 666 of the `matrix.c` file.

```
spyfunc send_value ; // the "send_value" function can be used in a
guard
Daemon dem_root { // the "dem_root" state machine
   // "timer" contain the global time plus 2 minutes
   // (the global time is in seconds)
   // The keyword "always" allows to evaluate a variable each time
   // there is recursion on a node.
   // (This automaton does not have nodes, thus the variables are
   // recomputed at the end of the execution of the command actions).
  always time_g timer =  FAIL_GLOBAL_CLOCK + 120;
   // "rand" ranges uniformy from 1 to 4
  always int rand = FAIL_RANDOM (1, 4);
   // "comp_to_fail" contains the random choice of "rand" computers.
  always tabc comp_to_fail = FAIL_RANDOM_TABC (All_comp, rand);
  timer -> !crash(comp_to_fail); // when timer expires, send message
           // "crash" to all computers in "comp_to_fail"
}

Daemon dem_other { // the "dem_other" state machine
   // "kill_line" refers to a line in the source code that can be used
   // in a guard.
  ln kill_line = "matrix.c":666;
  node 1 :              // Upon receipt of message
    ?crash -> goto 2;  // "crash", go to node 2.
  node 2 : // When the tested program calls "send_value" or arrives
           // at line 666 of "matrix.c", permanently stop its execution
           // and go to node 1.
    kill_line -> halt, goto 1;
    before(send_value) -> halt, goto 1;
}

Computer comp_root {    // the "comp_root" computer
  program = "test 500"; // "test 500" is the command line to be
                        // executed
  daemon = dem_root;    // computer "comp_root" is driven by
                        // "dem_root"
}

Group group_others {    // the "group_others" group
  size =  499;          // this group has 499 machines
  daemon = dem_other;   // computers in this group are driven by "dem_other"
}
```

If there are no processes launched on a machine that executes the "halt" action, then this action does not have any effect. The expression `timer -> !crash(comp_to_fail)` is a guarded command. When the moment defined by `timer` is reached, the action is exe-

cuted. Here, the action consists in a sending of the message `crash` to the set of machines `comp_to_fail` (which was previously calculated).

When an automaton consists of several guarded commands, it is possible that several guards become activatable at the same moment (for example if they depend on the reception of the same message). In this case, only one guarded command is executed. The choice of this guarded command depends on the execution mode of the automaton (there are two modes). In the *random* mode, the criterion is to randomly choose among activatable commands. In the *deterministic* mode, the first activatable command appearing in the automaton definition is executed. This second mode is necessary, albeit less intuitive, if reproducibility of the tests is to be ensured.

3.2 The FCI platform

The FCI platform is the first fault injection platform that uses FAIL as its input language. We now describe the implementation of the runtime library and address some deployment issues. Finally, we run some preliminary tests to investigate the potential impact of FCI on the tested applications.

3.2.1 The FCI library

The FCI library supplies to the automatons generated from the user's scenario basic building blocks that are necessary for the execution. This library consists in a set of C++ classes which use the standard C++ library and the ACE library (Adaptive Communication Environment) [7, 6], that is available on a large set of system platforms (including MS Windows and various Unixes). The classes of the FCI library can be partitioned in three main categories:

Event management The FCI library use the ACE library to manage events. It supplies a mechanism for events multiplexing. An ACE class called`Reactor` [5] is used to wait for particular events to occur. When an object is interested by a particular event (*e.g.* meter, network connection, etc.), it registers itself with the appropriate `Reactor` object which also starts waiting. When one of the expected events occurs, the `Reactor` activates the object which were interested by it. This mechanism allows to wait for several events of different types with a minimal impact on the functioning of the tested application. Indeed, when the `Reactor` is waiting for events, the FCI daemon remains sleeping.

Application control The FCI library uses `gdb` software to control the tested application. This mechanism allows to observe and to control the behavior of the application without modifying its source code and with low intrusiveness[1]. Also, having the source

[1]The written application should however be written to be "debugger friendly", *i.e.* properly resume blocking operations when interrupted (the `gdb` debugger makes use of signals to control the execution of the debugged process).

code of the application is not necessary. For full features, at least the application compiled in debug mode is needed. However, if only a "release" application is available, automatas that do not use knowledge of the source code (*e.g.* using timers, messages, etc.) can be implemented. Furthermore, `gdb` works with many different programming languages, which allows to use our tool for numerous existing applications.

Daemon communication The FCI daemons are themselves a distributed application. We use the ACE library to connect and communicate with distant machines. Besides, a configuration file (which is automatically generated) allows to deploy the whole application.

3.2.2 Deployment

The compilation of a FAIL scenario (*e.g.* `file.fl`), using the FCI compiler, produces a dedicated directory. This directory contains the generated C++ files corresponding to the fault scenario, the source code of the FCI runtime library, and Perl language scripts used for deployment. The "init" script copies appropriate FCI source code to every participating machines (that are specified using a configuration file) and compiles them into FCI deamons executable files specific to every platform. The "exec" script spawns every daemon. When all daemons are initialized, either each daemon executes a command (if such command was provided by the user) or wait for some other daemon to connect. A distinguished daemon, the *entry point*, takes care of waiting for all other daemons to be properly initialized. The entry point daemon need not (but can) participate to the experiment. When the experiment is over, the "stop" script is to be executed to shutdown all daemons on participating machines.

3.2.3 Preliminary tests

Due to the still early stage of development of FCI, only simple preliminary tests were carried out. Those tests are aimed at determining the impact of our solution on the execution time of a distributed application, rather than actually evaluating distributed applications in a faulty environment. The application being used for the tests consists in a server and several clients. The tests were led on 61 machines performing under Linux 2.6.7. Thirty machines were equipped each with a 2083 MHz processor and 885 Mb RAM (the server was run on one of those machines). Six machines were equipped each with two 1533 MHz processors and 885 Mb RAM. Eighteen machines were equipped each with a 1533 MHz processor and 885 Mb RAM. Seven machines were equipped each with a 1800 MHz processor and 504 Mb RAM. All machines were connected using a 100 Mbps Ethernet network. We considered two possible kinds of clients (that were executed each on 60 machines):

1. *the sleepy client* executes 60 times a loop where it sleeps for 10 seconds and then performs a TCP connection to the server and wait for some response,

2. *the looping client* executes 60 times a loop where it performs a calculus (an addition within a inner loop) and then performs a TCP connection to the server and wait for

Client	Ref.	Empty	Time	Func.	1->all	all->1
sleepy	0	0.01%	0.02%	0.03%	0.08%	0.10%
looping	0	0.00%	0.31%	0.08%	0.18%	0.17%

Figure 3.3: FCI Overhead

some response.

This operation is then repeated several times and averaged over all clients. The description of the tests we made and the results that we obtained are as follows:

First, we ran the application without using FCI, to obtain a reference execution time of the application. The execution time we measured was 600.37 seconds with the sleepy clients, and 554.02 seconds with the looping clients.

Then, we ran the application using FCI, by specifying an empty fault scenario. Every program was also launched through a FCI daemon, the objective was to measure the extra cost induced by our architecture. The execution's time of the application we measured was 600.42 seconds with the sleepy clients, and 554.03 seconds with the looping clients.

Then, we launched the application using FCI and time-based triggers (the FCI daemon getting control of the application every 1 second, this duration corresponding to the granularity of timing events in ACE). On the simulations we performed, each daemon was activated 556 times on average. However, no faults were injected and no guarded commands were defined. This allowed us to verify the impact of this type of triggers on the performance. The execution's time of the application we measured was 600.47 seconds with the sleepy clients, and 555.75 seconds with the looping clients.

Then we ran the application using FCI and triggers based on functions calls in the tested application (in our case, when the tested application tried to call the function `connect_and_read` for the reception of messages (this function being called 60 times), the FCI daemon gets control of the application). Still we injected no faults and performed no actions. The measured execution time was 600.58 seconds with the sleepy clients, and 554.48 seconds with the looping clients.

Then, we launched the application using FCI and using triggers based on function calls (60 function calls are made by each process), but this time we executed actions when a particular function was called by the tested application. We considered to kinds of actions:

1. *one to all*: one daemon sends a message to every other daemon in the system,

2. *all to one*: every daemon sends a message to the same designated daemon.

The execution time we obtained was 600.83 (for the one to all scheme) and 600.95 (for the all to one scheme) seconds with the sleepy clients, and 555.01 (for the one to all scheme) and 554.97 (for the all to one scheme) seconds with the looping clients.

Overall, the preliminary tests that we ran show that the event-driven model of our fault injector limits its impact (*i.e.* calculated overhead) on the execution time of the application

to less than 0.31% for the considered cases (see Figure 3.3). The highest impact is reached with the looping clients, which is explained by the interruption of the normal program flow. Also, the higher impact of the time based trigger can be explained by the higher number of interruptions (556 on average versus 60).

Chapter 4

Concluding Remarks

In this article, we presented a new language (FAIL) and tool (FCI) for software fault injection in distributed applications. Our solution allows to build complex faults scenario in a simple way while preserving the user of writing low-level code. It is possible to generate probabilistic scenarios (for average quantitative tests) or deterministic and reproducible scenarios (for studying the application's behavior in particular cases). Lastly, it is possible to work with applications written in many programming languages without requiring modification of their source code and the preliminary tests which we made show that FCI effective impact at runtime is low.

We are now investigation using our fault injector in larger systems, typically by using emulation systems, within the Grid eXplorer project framework. 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.

For typical fault scenarios in large scale systems, previous studies show that the average time between two faults will be on the order of minutes. In this context and with the results we obtained from the preliminary tests, FCI is expected to be scalable, at least when faults do not require to maintain the whole global state during the whole execution (note that probabilistic and cascading faults fall in this category).

The source code of FCI and the source code of the tests carried out in this article can be downloaded at `http://www.lri.fr/~hoarau/fail.html`.

Bibliography

[1] 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.

[2] 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.

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

[4] `http://www.lri.fr/~fci/GdX`.

[5] D.C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.

[6] D.C. Schmidt and S.D. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns.* Addison-Wesley Longman, 2002. ISBN 0-201-60464-7.

[7] D.C. Schmidt and S.D. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks.* Addison-Wesley Longman, 2003. ISBN 0-201-79525-6.

[8] 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.