A Software-Implemented Fault Injection Toolkit for Dependency Analysis of Large Scale Distributed Applications

Guangxia Xu, Huawei Lu, Yunpeng Xiao, Yanbing Liu

1School of Software Engineering, Chongqing University of Posts and Telecommunications, Chongqing 400065, China, xugxia@163.com
2College of Computer Science, Chongqing University, Chongqing 400030, China, semiwinter@163.com
3Institute of Mobile Internet & Information Security, Chongqing University of Posts and Telecommunications, Chongqing, 400065, China, liuyb@cqupt.edu.cn

Abstract

A new software implemented fault injection toolkit named SIFIT is described in this paper which can be used to testify the dependability of large scale distributed applications on Unix-like platforms. This toolkit is composed by interfaces, services and tools. One of its innovations is that it has been also integrated with a general debugging tool so that it has the functionality of debugging and locating the weak components during the process of fault injection. It is assumed that the target application is composed of multiple components (programs) which cooperate for the result, and that its successful completeness is determined by a failure condition. SIFIT is capable of injecting transient and permanent faults with emulating error state incurred by hardware, software or network connection faults in the runtime environment of each program. It can also collect test results from all components and determine the soundness of the final result based on the failure condition.

Keywords: Software-Implemented Fault Injection Toolkit (SIFIT); Dependency Analysis; Distributed Applications

1. Introduction

One of the major problems in developing large scale distributed applications is how to evaluate their dependability [1, 2]. Fault and error injection is one of the most effective methods for testing fault tolerance algorithms/mechanisms with respect to their own specific inputs by speeding up the occurrence of errors and failures [3, 4]. It is accomplished by intentionally injecting faults to happen in real world. The effects of faults on the distributed system can be also uncovered while fault injection tool executes realistic programs. Fault injection method has become an attractive way of validating specific fault tolerance mechanisms and allowing the estimation of fault tolerant system measures [5, 6]. According to the way of injecting faults and errors into target, these methods can be classified into two categories which are hardware and software implemented fault injections. And network connection fault could be triggered by either hardware errors or software errors, so the fault injection methods are generally distinguished by hardware and software.

Hardware implemented fault injection method injects physical faults into the target system hardware. Several methods [6, 7, 8] have been proposed to cause actual hardware faults which may be close to a realistic fault model. However, these require specially designed hardware devices for fault injection which are generally dedicated to a specific target system. Simulation-based fault injection methods [9, 10] inject faults and errors into a simulation model of the target system. Although they have the advantage of perfectly controlling where and when faults are injected, enormous development effort and much time are required.

Software implemented fault injection method emulates the error state of the system hardware and software through special programs. This method is a cost effective and efficient method to require no extra hardware and to be able to be easily expanded for new fault types. FERRARI [11] inserts software trap instructions at the specific address of instructions where a fault should be triggered. The major function of FERRARI is to corrupt the memory image of the target process; if the target process arrives at the specific address of instruction, it alters the contents of the program counter or contaminates the values of registers used in the instruction. Therefore, a preprocessing job is required to get address information of the process through the parsing of its object
A Software-Implemented Fault Injection Toolkit for Dependency Analysis of Large Scale Distributed Applications
Guangxia Xu, Huawei Lu, Yunpeng Xiao, Yanbing Liu

code. MAFALDA [12] was developed to evaluate dependability of microkernel. It evaluates the robustness of the application interface by observing the behavior of the operating system when input parameters of a given type and the state of kernel are corrupted. XCEPTION [13] uses the advanced debugging and performance monitoring features included in most of the recent processors to inject more realistic faults by software, and to monitor the activation of the faults and their impact on the target system. Its major drawback is that the information about when and where exceptions have to be triggered to inject faults must be programmed on the processor debugging hardware before the target application starts.

Fault types specific in distributed systems have also been a concern in several fault injection tools. FIAT [14] was developed in a token ring based environment RT PC of Carnegie Mellon University, which enabled the corruption of the memory image of a task. This tool validates dependability of the distributed system of which architecture contains primary and backup nodes as hardware redundancy. SFI [15] is a tool developed to facilitate the validation of dependability mechanisms on a distributed real-time system called HARTS. It allowed combinations of fault types to be injected in the nodes of a distributed system. It is able to inject faults in the communication subsystems of the target systems through software.

Many achievements are mounted nowadays, but there is still lack of research on general model of fault injection, and the common characteristics of popular fault injection tools could be summarized as: 1) the oriented target systems are mainly focused on IA-32, the research on IA-64 is still limited; 2) the patterns that faults injected are generally fixed, and new injection pattern is not allowed; 3) current tools are lack of automation in some degree, and the time-consumption could not be well controlled.

Focus on the This paper proposes a new toolkit called SIFIT which is capable of injecting software design or implementation faults into distributed applications in addition to hardware faults in single nodes. There are several fault injections tools in this toolkit and new injection patterns can be easily extended, meanwhile several debugging tools are also integrated in this toolkit. The objectives of SIFIT are to simplify the measurement of dependability parameters and the validation of dependability mechanisms in distributed systems, and to provide easily extended local fault injection patterns to support testing of dependability mechanisms on distributed Unix-like platforms. This toolkit can be also adapted to other distributed application systems such as clouds platforms and other COTS (Commercial Off-The-Shelf) operating systems with few configurations changes. Several studies including Ballista project [16] have addressed the assessment of COTS operating system in the presence of faults [17]. We focus on the validation of the robustness of dependable distributed applications which can be composed of several software components running on different nodes to cooperate with each other to perform a specific functionality of the application on condition that the robustness and coverage of the operating system can be known priori. SIFIT is designed to emulate error states of the runtime environment of the target application without effecting on the state of the operating systems. Faults are injected with minimum interference with the target application, i.e, the target application is not modified.

The remaining parts of the paper are organized into four sections. Section 2 describes the assumptions and faults models. The design and implementation details of SIFIT are described in Section 3. Section 4 describes the experiments performed on a single program and some distributed dependable applications which adopt two representative software fault tolerance techniques such as recovery block (RB), N version programming (NVP). Finally, conclusions are provided in Section 5.

2. Fault Models and Assumptions

2.1. Hardware Faults

This section describes fault models and assumptions used in this paper. Users set fault attributes to happen in real circumstances including fault type, fault location, fault latency and fault duration. It is assumed that the origin of a fault can be physical defect of hardware or due to design or implementation of hardware or software. Hardware faults are classified by their duration as follows [18]: permanent, also known as a black-hole, or transient and intermittent, which is treated as a Byzantine one. Permanent faults are caused by irrecoverable component damages such as exhaustion. They can only be tolerated by using hardware resource redundancy. SIFIT emulate these faults with
keeping their error states during the whole mission time of the target application. Byzantine ones, on the other hand, are triggered by environmental conditions such as voltage fluctuation, or they occur due to unstable hardware or varying hardware states. These faults usually do not cause any lasting damage in the affected component, although they can cause the system to change to an erroneous state temporarily. We assume that they alter the state of the runtime environment of the target application once or any other times. And the fault intervals allow users to set latency time which is the predetermined amount of time elapses for faults to be injected. It is important to allow users to determine the fault trigger condition. The fault type and fault location defines exactly what is corrupted and how is that corruption performed. Table 1 summarizes fault types which SIFIT supports according to three major hardware resources as follows:

<table>
<thead>
<tr>
<th>Processor/Register</th>
<th>Address of source and destination (HAF)</th>
<th>Base address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Single bit (HMSF)</td>
<td>Text/Code segment</td>
</tr>
<tr>
<td></td>
<td>Single page (HMPF)</td>
<td>Data segment</td>
</tr>
<tr>
<td></td>
<td>Bus error (HBF)</td>
<td>Heap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td>Device</td>
<td>Device down (HDD)</td>
<td>Open – Close</td>
</tr>
<tr>
<td></td>
<td>Data corruption (HDC)</td>
<td>Read – Write</td>
</tr>
</tbody>
</table>

Processor: Processor faults are emulated with the effect of faults occurred in internal functional units such as ALU, or internal data and control flow of processor. Buffer registers, base register and program counter are considered as hardware resources where faults can occur. Permanent injection of these faults consumes too much time to be practical, even though both permanent and transient fault duration are provided. HAF fault signifies what can be occurred in buffer registers to make a wrong address reference. The contents of data field in buffer registers are corrupted under HOF faults. When a HCF fault occurs, the content of program counter becomes corrupted, i.e., the control sequence of the target program is disturbed. HMAF fault is one that occurs in base register when it contains address for memory allocation or segment reallocation, as a result, leads to a wrong memory access and allocation.

Memory: A memory error can be injected as a single bit, a single byte, a single page or bus error. The location where faults are to be injected can either be specified explicitly or randomly from the range of target segment.

Device: Device faults including communication faults are treated by emulating high-level error state incurred by these faults. It is accomplished by wrapping several generic system calls related with device I/O operations such as open, read, write and close. HDD fault is what prevents the failed device from responding arriving messages or requests from the target program. When these faults occur, all requests or messages to the device get lost. HDC fault indicates what corrupts the data content of the device, but let the device respond normally. Messages or requests are altered or delayed on condition that HDC faults occur. These faults can be manipulated with a manner similar to memory faults.

2.2. Software Faults

Software faults are caused by incorrect specification, design or implementation of a program. Both experimental and real-life dependable distributed applications have been adopting design diversity to tolerate software faults. For examples, RB and NVP are commonly used to tolerate software faults. SIFIT has provision to validate fault tolerance capabilities of these software redundancy architectures for distributed dependable applications. We assume that the target distributed application is composed of several components (programs) to cooperate with each other to perform specific functionality. Since SIFIT can inject faults into just one program, it requires the same number of instances of SIFIT to test dependability of the target application. A client-server model is adopted for the coordination of multiple SIFIT instances that inject different faults into each component and measure dependability of the applications.

3. Designs and Implementation
3.1. Architecture Design

The purpose of design of SIFIT is to provide a combination toolkit of multiple existed fault injection tool and debugging tools, and provide a general interface for testing and debugging. And faults on different layers of targeting test applications could be simulated and injected into the system. In order to accelerate the paralleled development and decrease the components coupling of the toolkit, the definition of service is proposed in this paper. A service is a set which contains several specified functions. And SIFIT is composed by services, interfaces and fault injection tools, as shown in Figure 1.

![Figure 1. The Composition of SIFIT](image)

Based on the basic principle of fault injection, the fault injection toolkit should be composed of monitoring and fault injection, the monitoring service is taken to configure the testing parameter and analyze the collected testing results. The fault injection service is taken to inject determined faults into specified components of the testing target by using corresponding tool. And in our design, we also introduce the debugging service into our toolkit, which is taken to verify the exact failures.

Meanwhile current processors have debugging registers as well as performance monitoring mechanism. For examples, a large number of processors including HP SFU (Special Function Unit), Pentium Processor, Power PC, Alpha AXP, MIPS architectures provide these mechanisms. The prototype of SIFIT has been developed on Linux kernel 2.6.22 and GDB 7.2.

3.2. Implementation of service

SIFIT provides rich functionality of both program debugging and fault injection through the hardware debugging and monitoring facilities, in order to make SIFIT more portable and general, we divided the three services into several functions, or modules.

The monitoring service is composed by controller, user interface (UI), strategies set, communicator and analyzer. As SIFIT requires coordinating multiple instances of fault injection services and collecting test results from all components, the monitoring service takes the responsibility of these jobs.

Controller module is the core of SIFIT, due to fault injection platform using C/S mode, controller module is divided into two parts running on the test node and the target node, their operating environments are quite different, but the basic function is that they are under the control of the message in the system in order to trigger the module to pass, and effectively exchange the data between modules.

The fault injection service provides the functionality to inject faults into the runtime environment of each component during reciprocal actions with system resources. The fault injection service is run on the target, which exactly executes fault injected tasks. Fault injection service performs three major steps. First, it establishes fault model by receiving an input file which contains the attributes of faults given by the user. The attributes of faults include fault type, fault location, fault latency and fault duration which have already been described in Section 2. Next, it injects faults into the system according to the model while running the target program. Finally, it collects and measures the responses of the system for every injected fault and then records the results in an output file.

In order to achieve the original tool integrated into a distributed architecture, uniform test interface is provided, the idea we put forward the development of each component-based fault injection tool. Each of the tools is composed of three parts: component behavior, component attributes, component interface. Component behavior described tools can realize the function, specifically refers to the principles of simulation-based techniques into a certain target a certain level of failure. Component
represents a tool of state property, Tool ID and description of specific information by uniquely identification of the fault injection tool. Component interface, the method used for the external module, the interface in the next section will detail the implementation strategy. Based on this standard to develop the fault injection tool, you can add this article to support the fault injection platform, enhanced this development platform scalability.

The procedure of fault injection on a single node is as follow: First, it loads a target program and sets a breakpoint at the main function of the program. When the program stops at main function, it reads an input file containing fault attributes given by the user. According to the fault attributes, it performs preprocessing. During the preprocessing period, it extracts the necessary instructions where faults have to be injected in order to minimize the number of software traps. Next, SIFIT sets breakpoints on all the instruction selected in the preprocessing stage and continues the target program. The program is executed at full speed until it is interrupted by software traps, and then faults are injected into the system state. After a series of stages, fault injection process returns to the initial stage and repeats the above stages according to the number of faults to be injected. Each instance of SIFIT injects faults and measures the dependability of each component. It injects faults which can occur in processor, memory and device 170 operations. Memory faults are injected into the logical memory areas of the target program which include data segment, text segment and stack area. The location where a memory fault has to be injected can be specified either explicitly by the user or randomly by SIFIT. Randomly selected faulty memory locations can be anywhere in the local memory areas of the process. SIFIT first extracts the addresses of all instructions which are going to access faulty memory locations through the preprocessing stage. The fault duration of memory fault can be either permanent or transient. In case of transient memory faults, the target program stops when it reaches the specific addresses, and corrupts the value of the memory only once.

The debugging service is used to verify the test under certain circumstance. Debugging service is a part of GDB which provides process debugging and monitoring facilities including interactive user operations, while SIFIT also has provisions for distributed dependable applications.

3.3. Implementation of interface

In order to enhance the flexibility of fault injection tools, we encapsulated the dynamic link library of the fault injection tools, providing its external interfaces using the four functions: initialization (fnInit); query for tool information (fnGetModInfo); receive messages (fnProcPkg); and end of operation (fnTerm).

Function fnInit is an abstraction which is used to trigger a specified fault injection.

The most used function is fnGetModInfo, it returns specific information of the fault injection tool, such as tool ID, tool description, etc.

Function fnProcPkg is responsible for passing specified messages to the fault injection tool in the message queue to be processed.

Function fnTerm is the interface of termination an operation of fault injection. It is responsible for cancelling the processes and threads, releasing relevant resources that the tools taken during the injection process.

3.4. Design of automated fault injection tools

Traditional fault injection tools are mainly operated manually, this method has many problems, and we take the advantages of the automated testing framework developed by IBM STAF [11] to implement our own automated fault injection tools.

As the automated testing process is generated by simulated manual processes, based on the process of fault injection, we extract the consistency of the work, by means of the procedure call services, the file system service, queue service, which are provided by STAF, an automated test system could be achieved. It could be divided into three modules, dynamic deployment, testing task execution, and task monitoring separately.

4. Experiments
This section presents empirical results on the evaluation of the impact of faults in applications on Linux system. The main objective of this section is to show the experimental results about fault impact and characteristics that can be obtained through SIFIT. Experiments are performed on four PCs connected with Ethernet, running Linux with kernel 2.6.22 operating system. Matrix multiplication programs are chosen as the target of these experiments. Even though the user can set attributes of faults to be injected and the number of fault/error injection runs, these experiments select random fault type and random fault location. Results for 8000 runs are presented in this section. Whether a program succeeds or not is decided by comparing with that of the component without artificial faults or by acceptance test. We gather the results of all experiments and classify the fault impact into three main classes. Task Stop is the class for an abnormal termination of the program due to illegal runtime environment such as corrupted address or illegal instruction code. This class makes the target program terminated or suspended abnormally. SIFIT reports the reason why the target program has been halted. Invalid Output occurs when the faulty task completes its execution without the detection of the faults, but generates invalid and unacceptable output. The faults that belong to this class will go undetected by control flow monitor, but can be detected by duplication or bounds checking. SIFIT automatically find the failure of the component by help of result comparison mechanism. No Error is the class to signify that the program successfully completes its execution and generated valid outputs.

Figure 2. Results of Matrix Multiplication

Figure 2 shows the results for the matrix multiplication application. A thousand transient errors from every fault model are injected into the application while multiplying 20x20 matrices. 43.30% of the injected transient faults do not affect the program output because they remain latent in the course of the experiment, or are overwritten before they were used in subsequent computations. 39.20% of injected faults are detected by the built-in error detection in the system, so they pertain to Task Stop class. 17.5% of the errors are not detected by fault detection mechanism.

Figure 3 shows kinds and percentages of errors detected in the above experiment. The responses which belong to No Error are not taken into consideration. Task Stop includes segment violation, bus error and others. Segment violations are detected by memory access exceptions that occur when a data memory access or instruction prefetching fails to complete normally. The majority of these errors modify operand values that specify the memory address of data directly or indirectly. Bus error is due to the access of misaligned memory address that occurs when load, store or exchange instruction attempts to access the memory location which is not allowed to the process. Others include process hang-up and arithmetic exceptions.

Table 2. Results of Distributed Applications

<table>
<thead>
<tr>
<th>Adjudicator</th>
<th>Two components</th>
<th>Three components</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVP</td>
<td>succeed 446 fail 554</td>
<td>succeed 580 fail 420</td>
</tr>
<tr>
<td>RB</td>
<td>378 622</td>
<td>1000 0</td>
</tr>
</tbody>
</table>

Two representative software fault tolerance techniques, i.e., NVP and RB, are adopted as target software architectures. These architectures consist of a set of components, each one implementing the same function, plus some mechanism called adjudicator that obtains a single result to be used as the output of the target application. Three components are for RB and NVP. Adjudicators are usually based on voting for NVP, acceptance tests for RB. Each component in this experiment implements a different
sorting algorithm. We inject faults into either an adjudicator or a combination of components. When random faults are injected in the target application, Tab. 2 shows the experimental result.

5. Conclusions

We have presented a new software implemented fault injection tool SIFIT to test fault tolerance of dependable applications. It supports a general processor fault model with a wide range of fault triggers and fault sources including the internal processor functional units, memory cells and I/O operations. SIFIT is capable of injecting transient and permanent hardware faults with emulating error status incurred by hardware faults on the runtime environment of each program. Furthermore, it covers the ability to incorporate a distributed application. This functionality is necessary in that almost all techniques known for software fault tolerance are organized into several components to cooperate to produce the output of the whole application. SIFIT has been built upon the program debugging utility GDB. Therefore, user can make use of both facilities of fault injection and debugging simultaneously. Currently, we are working on developing methodologies for forming a unified dependable system design package by incorporating fault injection mechanisms with fault detection/tolerance mechanisms.

6. Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grant No. 60973160); the Science & Technology Research Project of Chongqing Education Committee of China (Grant No. KJ100506); the Natural Science Foundation of Chongqing University of Posts and Telecommunications (Grant No. K2008-27); the Open Foundation for Key Laboratory of E-commerce and Modern Logistics (under Chongqing Municipal Government) of Chongqing University of Posts and Telecommunications (Grant No. ECML1015); the Key Research Subject of The Network Social Development Issue Research Center of Chongqing University of Posts and Telecommunications, Key Research Institute of Humanities and Social Sciences of Chongqing (Grant No. 2010skjd12)

7. References


