Title:	**Testing the Fault-Tolerance of Networked Systems**
Authors:	Volkmar **Sieh**, Kerstin **Buchacker**
Published In:	International Conference on Architecture of Computing Systems ARCS 2002, Workshop Proceedings
Year:	2002
Pages:	95–105

Testing the Fault-Tolerance of Networked Systems

Volkmar Sieh & Kerstin Buchacker, Friedrich Alexander Universität, Erlangen, Germany

Abstract

This paper presents an extensible framework for testing the behavior of networked machines running the Linux operating system in the presence of faults. The framework allows injection of a variety of faults, such as faults in the computing core or peripheral devices of a machine or faults in the network connecting the machines. The system under test as well as the fault- and workload run on this system are configurable. The framework is supported by a graphical user interface for experiment control. We have tested the framework with a set of different fault injection experiments. The framework has proven to be stable and work as expected.

1 Introduction

This paper presents a framework capable of evaluating the dependability behavior of networked machines running the Linux operating system in the presence of faults. This paper focuses on the fault-injection and fault-tolerance testing aspect of the framework, which was first presented in [2].

The framework uses software fault injection to inject faults into a simulated system of Linux machines. The simulation environment is made available by porting the Linux operating system to a new "hardware" — the Linux operating system! Due to the *binary compatibility* of the simulated and the host system, any program that runs on the host system will also run on the simulated machine.

A process paired with each simulated machine injects faults via the `ptrace` interface. This interface allows complete control over the traced process, including access to registers and memory as well as to arguments and return values of input/output operations. Possible faults include hardware faults in computing core and peripheral devices of a single machine as well as faults external to machines, such as faults in external networking hardware. Of course the framework also allows to "inject" other types of faults, such as system configuration faults, site or environmental faults and interaction and operational faults, or to simply analyze the behavior and stability of the system under heavy load.

The rest of the paper is structured as follows. Section 2 gives a short overview of other work done in this area. Next Sec. 3 gives an outline of the framework. The next sections then give some information about the implementation of the simulated hardware (Sec. 4), and the steps necessary to ensure binary compatibility (Sec. 5). Sec. 6 presents the fault injector in detail. Section 7 explains how to use the framework and includes a small example of an actual fault injection experiment showing the functionality of the framework. The final section concludes the paper and presents an outlook on future enhancements of the framework.

2 Previous Work

This section gives an overview over hardware and software fault injection techniques as well as products, which allow simulation of a Linux machine.

Hardware fault injection is often destructive or damages the system under test, so it cannot be reused for more tests. Without an expensive setup of supporting machinery, hardware fault injection experiments are difficult to automate. Literature about hardware fault injection includes [8, 11, 14, 18, 22].

For many applications, simulation based or software implemented fault injection (SWIFI) techniques can be used, which do not have the drawbacks of hardware fault injection.

Faults based on a given fault model can be injected into a VHDL-model of a chip using simulation based techniques. This kind of simulation generally ensues a fairly high overhead, usually of several orders of magnitude if compared to the same chip implemented in hardware [9, 20].

Examples of SWIFI approaches are CrashMe [4] and Fuzz [15] for testing application programs, Ballista [12] for testing the behavior of the POSIX-API of an operating system. FERRARI [10], which can also be used to inject faults into the operating system, and MAFALDA [16] for embedded systems. A fault injector using the `ptrace` interface has been presented in [19]. Other tools using SWIFI include FIAT [1] and Xception [3].

The framework presented here differs from previous SWIFI and simulation work in several important aspects. It combines SWIFI with a simulation approach and therefore offers all the possibilities of the former. Since we simulate the hardware at a relatively high level (see [2]) not at the low level used by VHDL e.g. in [9, 20], the simulation is unusually fast (slowdown is less then one order of magnitude). Using SWIFI together with a simulated machine has the advantage, that once an experiment is set up, no user interaction is required. Furthermore the fault injection code is separated from the code for the simulated machine and runs as a separate process. Therefore, undesired interference and intrusion of the fault injection code on the simulated machine is avoided. Additional capabilities not offered by other simulation or SWIFI techniques include fault injection into a system of networked machines and use

of unmodified real world binaries and disk images on the simulated machines. As a result, the binaries used in our fault injection experiments are the exact same binaries later used in the real setting and the data on a simulated harddisk can be an exact image of the data on a real world harddisk. The tested system is therefore indeed the same system later running in the production environment.

2.1 Available Virtualization Software

The term *virtualization* as we understand it in this paper means running an operating system as a simple user process on top of the real operating system. In this and the following sections, the term *host* will always be used to designate the physical machine including its operating system. The terms *virtual*, or *user mode* (*UM*) will be used interchangeably when the simulated machine and operating system or processes running on this simulated machine are being referred to.

One way to implement virtualization is the emulation of virtual hardware in such a way, that an unmodified operating system can be installed onto this virtual hardware. Implementations like Bochs [21], SimOS [17] and Simics™ [23] also emulate the processor, whereas Plex86 [13], VMware™ [24] and Virtual PC™ [5] use the processor of the host system directly.

The second possibility to implement virtualization is the emulation of virtual hardware including the operating system running on this hardware. User Mode Linux (UML) [7] as well as our UMLinux are implementations of this approach.

To be able to inject a variety of hardware faults precisely at the targeted fault locations, it is necessary to know in detail, how the virtual hardware is implemented. For the commercial products Simics™, VMware™and Virtual PC™the source code is not available to us. Therefore we decided not to use them as the core of our fault injection framework. For performance reasons we also did not want to use the tools with processor emulation (Bochs, SimOS). We did not consider using Plex86 as it is still in an experimental phase at this time.

The UML described in [7] has some similarities with the user mode port of Linux we present in this paper. One big difference is, that in [7] a design decision was made, to map *every* UML process onto a *separate* process in the host system. This has direct consequences for fault injection. With this approach, some information necessary for process management is not kept in the UML kernel, but in the kernel of the host system, where it will not be affected by faults injected into the virtual machine.

Another major difference between UMLinux and UML is, that we implement kernel memory protection whereas UML does not. UMLinux therefore behaves just like the real Linux operating system, which — like any modern operating system — has kernel memory protection. Kernel memory protection also has direct consequences for fault injection, since user processes corrupted by faults so that they would damage kernel memory and data structures, will be terminated by the kernel, which itself remains intact. Without kernel memory protection, the operating system will be corrupted, too.

3 Outline of Framework

The framework consists of several interacting processes. An abstract view of the framework is shown in **Fig. 1**. The
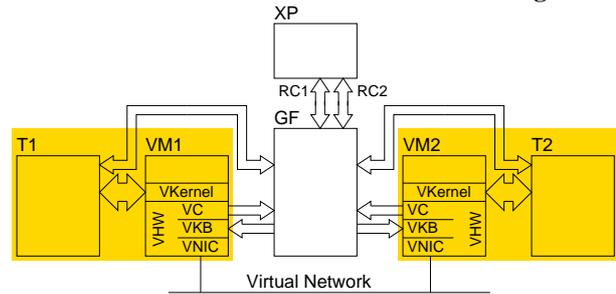

Figure 1   Outline of the Framework

figure shows two virtual machines (VM1 and VM2) each paired with its tracer (T), which is also the fault injector. Every box is a single process on the host. The processes making up a VM-T pair must run on the same host, but other than that there is no restriction. If several physical hosts are available, it is convenient to balance the load by starting VM-T pairs on different hosts. The close interaction of the tracer and the virtual machine is symbolized in the figure with a wide arrow.

The narrow arrows show how the graphical frontend (GF) interacts with the virtual machines. The output of each virtual console (VC) is sent to the frontend, where it can be viewed, saved or fed into Expect (XP), an automatic control program, as necessary. The frontend (or Expect) can simulate users sitting in front of the virtual machines typing away at the keyboard, since the virtual keyboard (VKB) is taking input from the frontend. The frontend can switch between the virtual machines, so a single instance is sufficient to control a complete virtual system under test consisting of several virtual machines. Expect opens a remote control connection (RC) to the frontend for every virtual machine to be controlled. Expect is configured from a file. Its purpose is to run lengthy experiments automatically without interaction from a human being.

4 Simulating the Hardware

The Linux operating system code is separated into hardware dependent and hardware independent code. Changes necessary for a port to another hardware only need to be made in the hardware dependent code parts.

We said in the introduction, that it is in fact a port of the Linux kernel to Linux. Thus, the virtual hardware is implemented using the facilities of the Linux kernel — system calls — and the CPU of the host machine. In modern operating systems, only the operating system kernel is allowed

to directly access hardware devices and execute privileged operations. All user processes must use the system calls the kernel provides to access the hardware devices they need.

We group the main hardware into the three categories *computing core* (memory management unit (MMU), random access memory (RAM), processor), *peripheral hardware* (keyboard, harddisks, cdroms, floppy drives, network interface) and *external hardware* (network cabling, power supply). The two other important hardware related categories which need to be handled by UMLinux are the *real time clock* (RTC) and *traps* (interrupts, exceptions, system calls).

For details concerning hardware simulation please refer to [2], this paragraph only gives a short overview. UMLinux accesses the real CPU directly. RAM is simulated with a memory-mapped file, the size of which is the size of the UMLinux machine's "physical RAM". Block devices, such as harddisks, floppy and cdrom drives are implemented as files. The console device is implemented as a socket and output to the console is sent to the frontend for display. The keyboard is implemented as a socket which takes input from the frontend. The network interfaces are also implemented as sockets. Power supply of a virtual machine is implemented simply by letting the simulator continue to run. The networking hardware and routing setup is implemented by a separate process, called UM networking process (UMNP). When the UMNP is running, the virtual machines are transparently integrated into the the local area network (LAN) to which the host machine is attached. Traps are implemented as signals.

5 Ensuring Binary Compatibility

To ensure the usefulness of our framework for real world problems, it is important that the simulator is binary compatible with the host system. Program binaries compiled for the host architecture should be able to run on the simulator without any modification. With binary compatibility, out of the box Linux distributions for the host system can be installed onto a UMLinux machine without any changes. We have installed of out-of-the-box Caldera OpenLinux, SuSE Linux and Debian distributions on the virtual machines without encountering any problems. Commercial applications for Linux can be subjected to fault injection using UMLinux as well, without the need to somehow acquire the source code and recompile the application before installation. Binary compatibility has the great advantage, that the program binaries used for fault injection experiments on UMLinux are the exact same program binaries later used in real world settings.

Since UMLinux directly runs unmodified user binaries compiled for the host system, without recompilation or code changes in those programs, every time they make a system call, they will trap directly into the host kernel instead of into the UM kernel. Therefore, all system calls of user programs running on top of UMLinux must be intercepted and redirected into the UM kernel. To both implement UM-Linux and ensure binary compatibility of UMLinux with the host system, it must be possible to convert a switch to real kernel mode to a switch to UM kernel mode, where the latter runs as a user process on the host system. This redirection/conversion of system calls is done by a second process (called tracer) paired with the UMLinux process and controlling it via the `ptrace` interface.

Another important aspect of binary compatibility are the access restrictions modern operating systems pose on user processes. In modern operating systems processes running in user space have no access to those parts of the RAM which contain the code and data constituting the operating system core (i.e. the kernel). This protects the kernel from corruption by user processes which inadvertently or maliciously try to write to addresses in the RAM, which contain kernel code or data. This access protection of the kernel space is supported by the hardware of most modern processors, which provide mechanisms to switch between kernel and user space.

On a UMLinux machine, both virtual kernel and user processes run inside a single real user process. To accomplish the separation of virtual kernel and user space we cannot use the same hardware mechanisms as the host machine's kernel, because these are not accessible to mere user processes. We must therefore use different hardware mechanisms available on the host machine.

The basic idea is quite simple. When a switch from kernel to user mode occurs in the UMLinux machine, all the memory pages belonging to the kernel are unmapped (removed) from the address space. Whenever a switch from user back to kernel mode is necessary (e.g. because a system call was made or an interrupt arrived), the pages containing the kernel code are remapped. The mapping and unmapping of the pages is triggered by a few lines of assembler code, most of the actual work is done by the host machine's MMU. Of course the code needed to trigger remapping the kernel pages must remain in main memory at all times and may not be changeable by user processes running on the UMLinux machine. This is enforced by the tracer.

6 Injecting Faults

This section explains how to inject hardware faults into UMLinux using a few typical faults as examples. The list of fault types given here is most likely incomplete, which is only due to the lack of imagination of the authors. Since the framework is easily extensible, anybody using it can implement and add their own faults.

The injection of hardware faults is event- or time-triggered. The exact time is configurable, as are the events, which may trigger the injection of a fault. We decided to concentrate the fault injection timing and control logic in the frontend so we can synchronize fault injection with other system events when investigating a networked system including several virtual machines.

We are aware of the fact, that the "faults" we inject are sometimes true faults (the cause of errors) and sometimes errors (effects of faults). E.g. when we inject a bit flip, we actually inject an error, not a fault, since a bit flip may be the effect of an alpha particle passing through a certain region of the chip. The classification of the faults and errors injected is closely tied to the question of fault representativeness.

The question of fault representativeness is exterior to our work. The framework presented here is a flexible and versatile fault injection tool, the user is free to choose the type and number of faults/errors to inject. The user may often want to test the survivability of a fault tolerant implementation against a certain type of fault, such as testing whether a system having software RAID tolerates more harddisk faults than a system without. Aspects of fault representativeness are extensively researched by [6].

The following sections categorize faults, explain how hardware faults are injected into UMLinux and give a definition of the fault load.

6.1 Fault Types

Following the hardware categories defined in Sec. 4 we define fault types of the same categories, i.e. computing core faults, peripheral faults, external faults, real time clock faults and interrupt/exception faults.

Hardware faults may be permanent or transient faults, can have a certain duration or appear intermittently. Permanent faults, such as a stuck memory bit or a bad block on a harddisk, remain in the system from their activation time until the end of the measurement run. Duration faults are like permanent faults, except for the fact that they disappear after a given duration. Transient faults, such as bit flips, are injected once and will disappear as soon as the bit is set to a new value. Intermittent faults are transient or duration faults which are injected again and again after a given interval.

6.2 Defining the Fault Load

The frontend (or Expect) needs to be configured to inject faults when a virtual machine is started. The description of the faults to be injected, the *fault load*, is configured per virtual machine using a simple textfile. For a large series of experiments, a script can generate a number of these files automatically. For each fault to be injected, the configuration file must contain at least the following information:

ID: (Autogenerated) Unique identification number.

Location: (User-supplied) The framework currently supports the following fault locations:
- RAM
- CPU registers
- Blockdevices (harddisks, cdroms, etc.)
    - failure of a number of consecutive blocks
    - device inaccessible
- Network interfaces
    - inability to send
    - inability to receive

Type: (User-supplied) The fault type may currently be one of
- permanent
- duration
- transient

Activation Time: (User-supplied) This is the time (in seconds, ≥ 0) at which the fault becomes active in the system. $t = 0$ is the time at which the virtual system boots.

Not all possible combinations of location and type are allowed. Depending on the fault location, additional information may be necessary to properly inject the fault (see Sec. 6.3).

6.3 Implementing the Fault Injector

To inject faults into the virtual hardware, those parts of the simulator implementing the hardware must be accessed. This can be achieved via the `ptrace` interface. All system calls can of course be modified to implement fault injection, but those system calls implementing the UMLinux hardware are most important for injecting (UMLinux) hardware faults.

To minimize the overhead, the process intercepting and diverting the system calls, the tracer, also handles the fault injection. Just like the Linux kernel, the tracer is implemented in the C programming language.

6.3.1 Computing Core Faults.

The following paragraphs treat computing core faults. RAM faults include transient bit-flip and permanent stuck-at faults. In addition to the information listed Sec. 6.2, a RAM fault is defined by

Address: (User-supplied) A valid address in the virtual machine's physical RAM (must fall on a word-boundary).

Bit: (User-supplied) The number of the bit which flips (between 0 and 31, where 0 is the least and 31 is the most significant bit of the word).

It is no problem to inject transient faults into the virtual machine. This is done by simply writing to the memory mapped file which is the virtual machine's RAM. The tracer first retrieves the contents of the word at the appropriate address using `word = ptrace(PTRACE_PEEKDATA, pid, (addr + MEM_BASE), 0)` Here and in the following paragraphs the variable `pid` contains the process identification number of the process simulating the UMLinux machine. The virtual machine's physical RAM is mapped beginning at (real) memory address `MEM_BASE`, the variable `addr` contains the (virtual) memory address to be retrieved. Its contents are returned in the variable `word`. `word` is changed to reflect the bit flip and written back to the virtual machine using `ptrace(PTRACE_POKEDATA, pid, (addr + MEM_BASE), &word)` When this manipulation is finished, the virtual machine, which was stopped

during this time, is allowed to continue. Transient faults can only affect processes (including the UM kernel) on the virtual machine when they read from memory, but will be overwritten by write accesses to the faulty part. Permanent faults, which do not disappear after a write access to the faulty part, can be implemented by remapping the affected pages or (on up-to-date Intel hardware) using the user debugging registers provided by Intel processors. Currently, only transient faults are implemented.

CPU faults include transient bit flips or permanent stuck-at faults in registers. An effect may be instructions skipped or wrong branches taken. In addition to the information listed Sec. 6.2, a CPU fault is defined by

Register: (User-supplied) A valid register of an Intel CPU (e.g. eax, eip, esp, eds).

Bit: (User-supplied) The number of the bit which flips (between 0 and 31, where 0 is the least and 31 is the most significant bit of the register).

Faults injected into the computing core will affect both the UM kernel and all UM user processes on the given virtual machine, just as would be the case on a real machine. To retrieve the current set of register contents from the virtual machine, the tracer uses `ptrace(PTRACE_GETREGS, pid, 0, &regs)` The register contents are now contained in the structure `regs`. Since the virtual machine is stopped at this time, it does not access its registers. The tracer changes the contents of the structure to reflect the bitflip, writes the changed register contents back to the virtual machine using `ptrace(PTRACE_SETREGS, pid, 0, &regs)` and allows the virtual machine to continue execution. Again, injecting transient faults is easily done in this way. To implement permanent faults, the register contents have to be checked and possibly modified after every single instruction executed by the UMLinux virtual machine, which will lead to a higher overhead. It is implemented by single stepping the simulator. Currently, only transient faults are implemented.

6.3.2 Peripheral Faults.

The following paragraphs treat peripheral faults. For parts of the following discussion, it is important to recall, that the virtual machine stops when entering into and returning from system calls. It does not continue until given a go-ahead by the tracer.

In addition to the information listed in Sec. 6.2, a block-device fault is defined by

Device: (User-supplied) A valid Linux block-device name (e.g. hda, hdb).

First Block: (User-supplied, only for blockwise failure) The number of the faulty block.

Number of Blocks: (User-supplied, only for blockwise failure) The number of faulty blocks following the first block.

Block-device access, such as harddisk, floppy or cdrom drive access is implemented using the system call sequence `open`, `lseek`, `read`, `close` for read access and `open`, `lseek`, `write`, `close` for write access. Thus every time an `open` system call is made, and at least one block-device-fault is active, the parameters of `open` must be checked, to see, if the faulty device is being accessed. Since blockdevices are implemented as files, we need to check the filename passed to `open`. The parameters to system calls are passed in defined CPU registers. The pointer to the filename specifically is passed in register ebx and we can retrieve its contents with `ptrace(PTRACE_GETREGS, ...)` as detailed above. The filename given is then compared to the names in all active fault descriptions referring to faults in block-devices. If a match is found, we set a flag to watch subsequent `lseek`, `write` and `read` calls. Since file access with these system calls uses filedescriptors instead of filenames, we need to remember the filedescriptor returned by a successful `open` call in register eax. In case the matching fault describes blockwise failures, we must look at the parameters to `lseek`, to see which block is accessed, and find out, if it falls into the range of defect blocks. If indeed all conditions are met, we annull the subsequent `read` or `write` call by doing a `getpid` instead (see [2]) and manipulate the return value to return an error. System calls return values in register eax, which can be accessed and manipulated with `ptrace(PTRACE_GETREGS, ...)` and `ptrace(PTRACE_SETREGS, ...)` as detailed above. The error returned from this real system call causes the virtual block-device-driver of the UMLinux machine to return an error to the virtual operating system.

The previous discussion shows that an active fault does not necessarily have an effect on the operating system at all, for example, when the defect blocks on the harddisk are never accessed. If the defect blocks are accessed, the fault is noticed by the operating system, but unless the blocks contain data important to the operating system (such as filesystem information), there may be no further visible effects.

In addition to the information listed Sec. 6.2, a network interface fault is defined by

Device: (User-supplied) A valid Linux network interface name (e.g. eth0).

The virtual network interface is implemented as a socket. To inject faults into the network interface, we need to manipulate the appropriate `sendto` or `recvfrom` calls, which implement sending and receiving via the network interface. Thus we need to be able to identify the virtual network interface by the (real) socket descriptor used in these calls. It is therefore necessary to build a table assigning socket descriptors to interface names. The necessary information is gained by looking at arguments and return values of `bind`, which is called once for each network interface during the hardware initialization phase of the boot process of the virtual machine. This system call is given a socket address (basically consisting of an IP address and a port number) and returns a socket descriptor. Our implementation of

the virtual network interface uses a fixed port number for each interface (which is read from a file having the same name as the interface). The implementation independent fault descriptions use interface names, not port numbers. To inject receive faults, we simply manipulate the return value of `recvfrom` to return an error. To inject send faults, we again annull the `sendto` call by doing a `getpid` instead and return an error.

Injecting permanent faults into peripheral devices does not incur a significant overhead, since the virtual machine is stopped at entry and exit of every system call anyway, so that the tracer can redirect the system call to the UM kernel if necessary. Additionally, only those system calls implementing UMLinux device drivers need to be examined more closely when peripheral faults are active. Those system calls redirected into the UMLinux kernel need not be manipulated to inject peripheral faults.

6.3.3 External Faults.

Some external faults are also possible. Depending on the setup, a power failure may affect a single or several machines. For all virtual machines affected by the virtual power failure, their tracers simply kill the corresponding processes by sending them a `SIGKILL`.

Defects in virtual external networking hardware can be implemented by configuring the UMNP to behave like faulty networking hardware. The UMNP must be started, to enable the transparent network connection of virtual and host machines or to have several virtual machines which are not in the same IP-subnet. By configuring the UMNP to not forward packets to/from a certain machine, a broken cable leading to that machine can be simulated. Examples for other possible defects include a working uplink but a broken downlink or missing or damaged network packets.

6.3.4 Interrupt/Exception Faults.

The tracer can intercept signals before they are delivered to the virtual machine as interrupts or exceptions. It is thus possible to inject "missing interrupt" faults, where an exception or an interrupt is not generated even though this should have been the case.

6.4 Controlling Fault Injection

The frontend (or Expect) reads the descriptions of the faults to inject into a certain virtual machine, the faultload, from a textfile.

As is shown in Fig. 1, the frontend passes the fault descriptions to the appropriate tracer for injection into the UM-Linux machine. In case the experiment is run automatically, the frontend is just an intermediary between Expect and the tracer.

The protocol used for the bidirectional communication between the tracer and the frontend is quite simple. **Figure 2** shows the transmission of the fault data from the frontend to the tracer. The left hand side shows the communica-

Figure 2   Fault Injection Protocol

tion, when a virtual machine is booted via the graphical user interface or Expect. The frontend starts the tracer using `fork`. Because we want to be able to boot a machine with faulty hardware, the tracer connects to the frontend and blocks soon after it is started (but before it starts the virtual machine) to receive the list of faults active at boottime. This list may be empty. After having received the list of initial faults, the tracer configures its communication socket to the frontend to use asyncronous IO. Now every time the frontend sends a fault description to the tracer, the latter receives an IO-interrupt and proceeds to retrieve the newly arrived data (right hand side of Fig. 2). The tracer itself does not need the timing information included in the fault description. All timing is done by the frontend. Once one or more faults are active, the tracer takes appropriate actions to inject the fault. New faults are activated as soon as the tracer receives new fault descriptions from the frontend. If the fault is a computing core fault, activating it is equivalent to injecting the fault, i.e. manipulating CPU register or RAM contents. If the fault is a peripheral fault, activating it is equivalent to setting appropriate flags to watch for access to the faulty hardware device.

The tracer can send information about the fault injection to the frontend. This can be used to find out, whether faulty blocks on a harddisk are accessed at all or whether the system actually did try to send data on a failed network interface. This information is necessary to tune the faultload and system setup so that it is possible to specifically target vulnerable locations, such as blocks on the harddisk containing data which is accessed often. We believe that it makes a difference, whether e.g. a faulty harddisk does not visibly affect the system because the faulty blocks are never accessed or even though they are accessed.

7   Using the Framework

The purpose of this section is to show how to use the fault injection framework we implemented. Section 7.1 explains what is necessary to run a workload on the system under test. The framework would not be complete without some means to collect the data from the experiments (Sec. 7.2).

Since the flexible framework allows a great variety of faults to be injected in a variety of systems, it is not possible to cover them all. We therefore describe a test setup in Sec. 7.3 and inject one of each processor, memory, network and harddisk fault. Sec. 7.4 assesses the overhead incurred through simulation.

7.1 Running the Workload

The framework presented here is aimed at testing networked server systems. The hardware and network configuration alone does not yet make a complete system. The missing part are the server processes, which are part of the system under test, and the client processes, which are necessary to generate a workload on the system under test.

The server processes need to be started on UMLinux machines. To this end, the appropriate startup and configuration files needed for the server processes must be installed on the virtual machines prior to the start of the experiment. When configured correctly, server processes, such as HTTP or DNS servers, are started automatically when a Linux system boots.

If the system under test is, for example, a database with an HTTP-frontend, the load would be generated by sending database queries to the system via the HTTP-frontend. The load generating processes are not subject to fault injection, because they are not part of the system under test. If these processes are started on virtual machines, these machines are also exempt from fault injection.

There are several possibilities to generate the client workload. Firstly, appropriate scripts or programs generating a client workload can be installed on a number of virtual machines and configured to automatically start when booting the machine. Secondly, when the UMNP is employed to provide a transparent connection between the virtual and the real world, the load generating clients can be started on real machines. And thirdly, the client workload can be generated with the frontend. Since the frontend is connected to the keyboard and console devices of all virtual machines, it is possible to simulate a user sitting in front of a virtual machine and using a web browser to generate HTTP requests. The user working with the frontend can do this interactively or start Expect to generate the client workload.

In this setup, it is useful to wait with running the workload, until the webserver is fully booted. Otherwise the clients may log connection timeouts which are not due to webserver-failures. How fast a virtual machine boots is of course dependent on the current load and process management decisions of the host machine, so even when the webserver is started before the client machines, the latter may have booted first. Concentrating the fault injection control in the frontend makes it possible to wait until the webserver has finished booting (this can be seen from console output) and then start the workload.

7.2 Collecting the Data

Data is collected from a number of different sources.

Virtual Machine Console Output: The frontend can be configured to log the console output of the virtual machines it supervises.

Virtual Machine Logfiles: Each Linux kernel, whether virtual or real, can be configured to log errors detected by the kernel. Server processes can also be configured to log their activities in different levels of detail. The web server Apache, for example, if configured accordingly, does not only log errors, but logs time, source and file requested for every single access.

Files on the Virtual Machine: When the system under test includes a database of some kind, the database files of the fault free and faulty runs can be compared directly.

Client Logfiles: The clients can be configured to log fault free and faulty server responses as well as timing information useful for calculating results.

Fault Injector: The fault injector can be configured to log when and how often the faulty device or part was accessed. This information can be used to target faults at specific locations for vulnerability testing.

Extensive logging may slow down the virtual system, just as is the case with a real system. Depending on the types of faults injected, some data sources may not be usable. Harddisk faults, for example, may corrupt the filesystem in such a way, that log- and other files on the affected virtual machine are no longer intact.

The data collected is examined to calculate the result measures of interest to the user. How this is done and what result measures can be calculated is highly dependent on the content and structure of the data collected. The general approach of the framework uses customizable scripts to examine the data and calculate result measures.

7.3 Example Experiment

The general setup of the example system is the following.

System under test: DB_WWW: web and database-server running Apache and MySQL. This virtual machine was equipped with two harddisks, one containing the operating system and binaries (HD1), the other containing the database and HTML-pages for the webserver (HD2). The contents of the database were automatically generated and consist of timestamped records each with a primary key. Two different databases were used. One contained about 2.7 million entries (DB1), the other started out empty and was filled and emptied again during the testrun (DB2).

DNS: domain name-server running bind.

Network: The virtual machines were connected by a virtual local network.

Processor, memory, harddisk and network faults were injected into DB_WWW.

The workload is generated by Perl-scripts running on an additional virtual machine (CLIENT). Two different workloads were used.

WL1 : 230 SELECT statements made via the webinterface accessing records evenly distributed throughout the database.

WL2 : A series of 100 INSERT, followed by 150 SELECT and 100 DELETE statements (all randomly generated and submitted via the webinterface) was repeated twice on different record sets.

The record sets to be read and written by the client were prepared in advance and known to the client, such that the client was able to recognize a faulty record returned by the server.

Four different experiments were conducted, one for each type of fault, as described in the following list. The results are summarized in the next paragraphs. The results were extracted from the client logfiles.

Memory Faults The faultload was a single transient bitflip of a randomly chosen bit in a randomly chosen byte of memory between 0 and 32MB. An equal percentage of runs was made with activation times of 150, 200 and 250 seconds. The workload and database used were WL2 and DB2. 282 single runs were conducted.

Processor Faults The faultload was a single transient bitflip of a randomly chosen bit in a randomly chosen register. An equal percentage of runs was made with activation times of 150, 200 or 250 seconds. The workload and database used were WL2 and DB2. 447 single runs were conducted.

Harddisk Faults The faultload consisted of permanent failures of 2000 consecutive blocks on the harddisk (HD2) containing the data. The activation time was 200, the start block was randomly chosen on the harddisk. The size of the harddisk was chosen, so that the data occupied nearly the complete disk. In this way we avoided that the faults injected only affect empty blocks on the harddisk which are never accessed. Furthermore we excluded a certain number of blocks located at the beginning of the harddisk from fault injection. Linux uses these blocks to keep information about the structure of the filesystem. Damaging these blocks will make the complete harddisk inaccessible, which is not what we intended in this experiment. The workload and database used were WL1 and DB1. 726 single runs were conducted.

Network Faults The faultload consisted of transient failures of the network device of DB_WWW. Both send and receive failures with durations from 5 to 40 seconds (with step of 5) were injected, with activation time

being 150 seconds. The workload and database used were WL2 and DB2. 109 single runs were conducted.

The server's behavior was viewed from the client's point of view and the errors observed were therefore classified into the following categories

- faulty response (faulty record data)
- delayed response
- server error response (SER)
- server crash or hang

The item SER corresponds to the HTTP-server returning some kind of error message, such as a "page not found" message or error messages from the database server which are passed on to the client via the HTTP-server. The last item is a server crash or hang from the clients point of view, i.e. the client is unable to evoke a response from the server until the end of that testrun. Not all of these possible behaviors were observed for each type of fault injected.

The memory faults injected had no immediately visible effect on the server. We believe this is due to the fact that only a single fault was injected per testrun. We also did not try to target sensitive parts of the memory explicitly, since (apart from the memory location of the kernel) it is not possible to tell a priori where i.e. the database- or webserver executables are located in memory at a certain time during the testrun. This behavior has also been observed on real machines with a defect RAM, were the only visible errors occurring once in a while were some defect files on the harddisk (the reason for this being the fact that Linux buffers disk I/O, so a memory fault in one of the I/O buffers will affect what is written to disk).

Figure 3 shows the percentage of different types of behavior observed in the example setup for the processor and harddisk faults. For 86.1% of the testruns injecting pro-
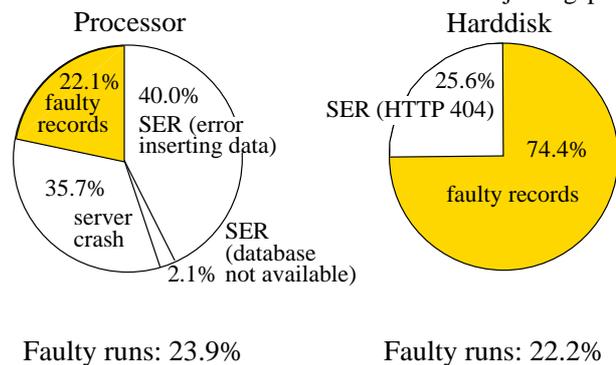
Processor   Harddisk

Faulty runs: 23.9%   Faulty runs: 22.2%

Figure 3   Results of the Experiments (Faulty Runs Only)

cessor faults, the client could not observe a faulty server behavior. For the 23.9% of testruns with observable faulty behavior, the distribution is shown in the left part of Fig. 3. It is possible for several different errors to occur during a single testrun. The client completely lost connection with the server and could not regain it during this testrun (35.7% of the faults). For the client it is impossible to tell, whether the lost connection is due to an operating system crash of

the server or crash of the webserver daemon only. In 40.0% of the errors observed, the server returned an error message, saying, that it could not insert the data into the database. In 22.1% of the cases no error message but a faulty record were returned. In the last 2.1% the web server returned the message, that it was unable to connect to the database.

Harddisk faults, since confined to HD2, could not affect the operating systems or database- and webserver binaries. Accordingly the clients never lost connection to the webserver, instead the webserver returned error messages when the data the client requested was inaccessible. Of the testruns performed, 77.8% terminated without errors. The percentages of the errors observed in the other 22.2% of the testruns are shown in the right part of Fig. 3. In about a quarter of the cases the web server returned an HTTP 404 "page not found" error, the rest of the time faulty records were returned without any error messages.

The experiment shows that the worst case, i.e. undetectable errors, happens in, as we believe, a non-negligible percentage of the testruns. In the experiment setup, the clients knew which response to expect from the server and could therefore identify the faulty records returned. This is usually not the case in a real world system unless special fault detection measures are taken.

Network faults only led to delayed server responses being observed by the clients. This is due to the fact, that the HTTP-exchange between client and server is layered on the fault-tolerant Transmission Control Protocol (TCP). The latter hides the retransmissions occurring due to the network failure from the application layer and the client only records a higher response time. The durations of the network faults were obviously not long enough to lead to TCP-timeouts. **Fig. 4** shows how the response times of the delayed responses (y-axis) relate to the fault duration (x-axis). The response times are sometimes much higher than the actual fault duration. This is due to the backoff and retry mechanism of TCP, which backs off for an increasing amount of time after an unsuccessful retry before trying again to connect.
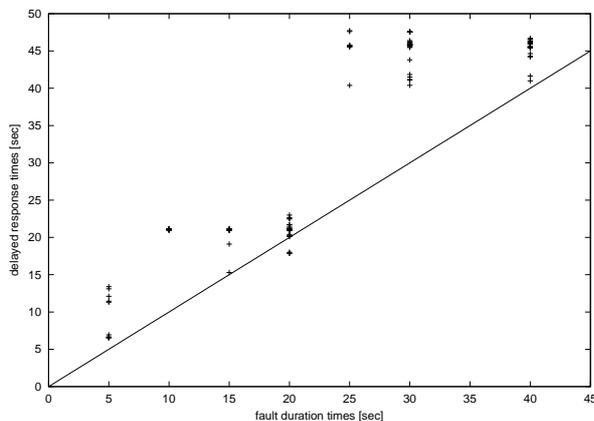
Figure 4   Network Delay Times

7.4   Overhead

As with any kind of simulation compared to the real thing, there is a performance penalty associated with running processes on UMLinux instead of on a real Linux machine.

A few a priori statements about the overhead incurred can be derived from the implementation of UMLinux. Since overhead is only incurred when a system call is made, compute intensive programs which make no system calls will incur no overhead. On the downside, system call intensive applications, such as creating a disk image, will incur a relatively high overhead.

An accepted benchmark for Linux systems is compilation of a standard kernel from scratch. Measurements have shown, that a standard kernel compilation which took about 6:00 minutes on the host machine, took 28:40 minutes on the virtual UMLinux machine running on that same host. The host machine was a Pentium II 350 with 128MB real RAM, the UMLinux machine only had 32MB virtual RAM. The virtual hard disk file and the directory used for kernel compilation on the real host were on the same local real harddisk. The overhead measured is much lower than for traditional simulation environments, where it is usually around several orders of magnitude.

The size of the virtual system which can be simulated within a preset amount of calendar time depends heavily on the computing power available. In an ideal environment, where each virtual machine is simulated on a separate real host, there is no limit to the size of the virtual system under test which can be implemented. The RAM and harddisk of the virtual machine should ideally be a little smaller than that of the host machine. Other than with SWIFI methods working with the real hardware and operating systems, virtual systems consisting of several UMLinux machines can still be simulated even when only a *single* host is available to run the simulation, albeit at a much slower speed due to the high load on that host.

8   Conclusion and Future Work

This paper presented an extensible framework for running fault injection experiments using User Mode Linux. The approach enables to setup user configurable virtual networked computer systems which can then be subjected to fault injection. Different fault tolerance strategies of the setup can be tested or compared. The Linux operating system and application programs can be tested for availability of fault tolerance capabilities. To the knowledge of the authors such a simple yet flexible and powerful fault injection framework has not been presented before.

We are currently working on enhancing UMLinux to include virtual *multi-processor* machines. Future work will certainly include smoothing some of the rough edges of the prototype and further improving the frontend, e.g. to allow interactive injection of faults. We also plan to include a small library of examples for some typical setups (including

Expect-scripts for automatic experiments). These can serve as a starting point for customization of a user defined setup and ease creation of new system setups.

Acknowledgment

References

[1] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, 1990.

[2] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings Sixth IEEE International High-Assurance Systems Engineering Symposium*, pages 95–105, 2001.

[3] J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. In *5th International Working Conference on Dependable Computing for Critical Applications*, pages 135–149, 1995.

[4] G. J. Carrette. Crashme. URL: http://people.delphi.com/gjc/crashme.html, 1996.

[5] Conntectix Corporation. Virtual PC. URL: http://www.connectix.com/, 2001.

[6] DBench - Dependability Benchmarking (Project IST-2000-25425). Coordinator: Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique, Toulouse, France; Partners: Chalmers University of Technology, Göteborg, Sweden; Critical Software, Coimbra, Portugal; Faculdade de Ciencias e Technologia da Universidade de Coimbra, Portugal; Friedrich-Alexander Universität, Erlangen-Nürnberg, Germany; Microsoft Research, Cambridge, UK; Universidad Politechnica de Valencia, Spain. URL: http://www.laas.fr/DBench/, 2001.

[7] J. Dike. A user-mode port of the Linux kernel. In *5th Annual Linux Showcase & Conference, Oakland, California*, 2001.

[8] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings of the 19th IEEE International Symposium on Fault Tolerant Computing*, pages 340–347, 1989.

[9] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proceedings of the 24th IEEE International Symposium on Fault Tolerant Computing*, pages 66–75, 1994.

[10] G. Kanawati, N. Kanawati, and J. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proceedings of the 22th IEEE International Symposium on Fault Tolerant Computing*, pages 336–344, 1992.

[11] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *5th International Working Conference on Dependable Computing for Critical Applications*, 1995.

[12] N. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th IEEE International Symposium on Fault Tolerant Computing*, pages 230–239, 1998.

[13] K. Lawton. Plex86. URL: http://www.plex86.org/, 2001.

[14] H. Madeira, M. Rela, and J. G. Silva. RIFLE: A general purpose pin-level fault injector. In *1st European Dependable Computing Conference*, pages 199–216, 1994.

[15] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revised: A re-examination of the reliability of UNIX utilities and services. Computer Science Technical Report 1268, University of Wisconsin-Madison, 1995.

[16] M. Rodríguez, F. Salles, J. C. Fabre, and J. Arlat. MAFALDA: Microkernel assessment by fault injection and design aid. In *3rd European Dependable Computing Conference*, pages 208–217, 1993.

[17] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The simos approach. *IEEE Parallel and Distributed Technology*, Fall, 1995.

[18] J. R. Sampson, W. Moreno, and F. Falquez. A technique for automatic validation of fault tolerant designs using laser fault injection. In *Proceedings of the 28th IEEE International Symposium on Fault Tolerant Computing*, pages 162–167, 1998.

[19] V. Sieh. Fault-injector using UNIX ptrace interface. Internal Report 11/93, IMMD3, Universität Erlangen-Nürnberg, 1993.

[20] V. Sieh, O. Tschäche, and F. Balbach. VERIFY: Evaluation of reliability using VHDL-models with integrated fault descriptions. In *Proceedings of the 27th IEEE International Symposium on Fault Tolerant Computing*, pages 32–36, 1997.

[21] Source Forge. Bochs IA-32 Emulator Project. URL: http://bochs.sourceforge.org/, 2001.

[22] A. Steininger and C. Scherrer. On finding an optimal combination of error detection mechanisms based on results of fault injection experiments. In *Proceedings of the 27th IEEE International Symposium on Fault Tolerant Computing*, pages 238–247, 1997.

[23] Virtutech Inc. simics. URL: http://www.simics.com/, 2001.

[24] VMware Inc. VMware. URL: http://www.vmware.com/, 2001.