

**Title:** Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects

**Authors:** Kerstin Buchacker, Volkmar Sieh

**Published In:** Proceedings Third IEEE International High-Assurance Systems Engineering Symposium, Boca Raton, Florida

**Year:** 2001

**Pages:** 95–105

# Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects

Kerstin Buchacker, Volkmar Sieh  
Institut für Informatik III  
Friedrich Alexander Universität Erlangen-Nürnberg  
Germany

{kerstin.buchacker, volkmar.sieh}@informatik.uni-erlangen.de

## 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 to inject 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 core of the framework is a User Mode Linux, which runs on top of a real world Linux machine as a single process and simulates a single machine. A second process paired with each virtual machine is used for fault injection. The framework will be supported by utility programs to automate testing and evaluate test results.*

## 1. Introduction

More and more computer systems nowadays are networked systems. Examples include large database systems in banking institutions, company e-commerce and web servers and systems providing IP-telephony to name but a few. Many of these systems fall into the high assurance category, since a failure of these systems will often mean great financial loss or even bankruptcy for those directly affected.

This paper presents a framework capable of evaluating the dependability behavior of networked machines running the Linux operating system in the presence of faults. The Linux operating system is usually employed in networked server environments, for example as web- or mailserver. We have tested the functionality of the framework with a first set of fault injection experiments.

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 virtual 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 framework will be used in the European DBench Project [6] for dependability benchmarking of Linux systems.

The rest of the paper is structured as follows. Section 2 gives a short overview of other work done in this area. Next an introduction to the implementation of the simulation core, the *UMLinux* is given in Sec. 3. The fault injector is presented in Sec. 4. Section 5 explains what is necessary to define a system under test. Section 6 shows how to conduct a testrun using the framework and includes a small example of an actual fault injection experiment showing the functionality of the framework. The last section concludes the paper and lists some future plans.

## 2. Previous Work

This section gives an overview over the hardware and software fault injection techniques as well as other products,

which might be used to construct a framework similar to the one proposed in this paper.

## 2.1. Fault Injection Techniques

Hardware fault injection techniques include pin level fault injection and electromagnetic, laser and heavy-ion radiation. Hardware fault injection is often destructive or damages the system under test so that it cannot be reused for more tests. Hardware fault injection experiments are difficult to automate without an expensive setup of supporting machinery. Literature about hardware fault injection includes [8, 11, 14, 17, 21].

For many applications, simulation based or software implemented fault injection (SWIFI) techniques can be used instead of hardware fault injection.

When the system under test is a chip, a VHDL-model of the chip in question can be generated, into which faults based on a given fault model can be injected using simulation based techniques [9, 20].

To test the stability of application programs, the approach taken by CrashMe [5] and Fuzz [15] is to feed the program random input and analyze its behavior. This approach is not directly applicable to server processes and operating systems, since these do not usually take direct input from the user. On the other hand, server processes and operating systems both usually communicate over the network or with other processes using certain standard protocols. Therefore faults can be injected into these communication links by simply feeding the process random input or incorrect protocol tokens instead of the correct protocol tokens. This form of fault injection may be used in tools employed in denial of service attacks [18].

Another approach is to feed random or at least erroneous input to the system call interface of an operating system. This approach is taken by Ballista [12] to test the behavior of the functions constituting the POSIX-API as defined by [1].

Yet another approach is to inject faults into the data and code segments of a running process as well as into the registers and parts of main memory it uses. In addition, the system calls the process under test makes can be intercepted and their return values changed. FERRARI [10] is a tool which implements this and works fine for application processes. When using this approach to inject faults into the operating system itself, automating testing is difficult, since the computer must usually be rebooted manually when the operating system crashes or hangs. In addition, in the latter case the fault injection process should not be running on the machine with the (possibly already corrupt) operating system, since this can interfere with the integrity of the

fault injection process and can thus lead to erroneous results. MAFALDA [16] is a tool which implements a similar approach and injects faults into application service requests, data and code segments of micro-kernels of embedded systems. A fault injector using the `ptrace` interface has been presented in [19]. Other tools using SWIFI include FIAT [2] and Xception [4].

The approach presented in this paper, is based on the one described in the previous paragraph and therefore offers all its possibilities. Unlike the tools mentioned above, we do not inject the faults into the real operating system, but instead into a virtual machine under test. We can thus avoid the drawbacks associated with injecting faults into the real operating system — influencing the fault injection and logging software running the experiment. In our approach, the real operating system is never affected and does not crash or hang and the fault injection software is not affected by crashes of the virtual machine. Since the fault injection code is separated from the code for the virtual machine and runs as a separate process, undesired interference and intrusion of the fault injection code on the virtual machine is avoided. In addition, the framework presented here offers the possibility to not only inject faults into a single machine, but into a system of networked machines.

## 2.2. Available Methods

This section describes methods available to build the virtual machine into which the faults will be injected.

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 when the simulated machine and operating system or processes running on this simulated machine are meant.

A commercially available tool is VMware™ [24]. VMware™ emulates a virtual i386-processor with configurable main memory size and peripheral devices including network connection. Onto this virtual machine, basically any i386-compatible operating system may be installed, although only some (including Linux and several Microsoft Windows variants) are officially supported by VMware™. The virtual machine runs on top of another operating system as a simple user mode process supported by some kernel modules. To be able to inject a variety of faults precisely at the targeted fault locations, it is necessary to know in detail, how, for example, the simulated peripheral devices are implemented. Since VMware™ is a commercial product, the source code is not available to us. Therefore we decided not to use it as the core of our fault injection framework.

User Mode Linux (UML) [7] is a user mode port of the Linux kernel, which has many similarities with the user

mode port of Linux we present in this paper. The 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. Therefore, parts of the information any operating system must keep for each process will not be kept by the UML kernel, but by the kernel of the host system. When a context switch occurs, and another process gets its share of processor time, the register contents of the process which was running need to be saved in main memory, so they can be restored when the process is scheduled to run again. Of course, information kept by the host kernel will not be affected by faults injected into the virtual main memory of the UML kernel, with the result, that injected memory faults cannot affect all processes (as would be the case in a real world system). The same holds true for faults injected into the virtual processor or memory management unit. Faults injected into virtual peripheral devices are not affected by this design decision.

### 3. Outline of UMLinux Implementation

This section introduces the basic principles of the User Mode Linux implementation that is the core of the fault injection framework presented in this paper. To gain a good understanding of the topics covered in this and the following sections, it is helpful to be acquainted with operating system internals and the interaction between operating system and hardware [23]. Details about the Linux kernel may be found in [3].

The main difference between our user mode Linux (UMLinux) implementation and the one cited in the previous section is our design decision to have a *single* process per virtual processor of the UMLinux machine. Thus, for a virtual single processor machine, the complete UMLinux — virtual machine, operating system and all user processes included — runs as a single process on the host system, whereas for a virtual dual processor machine there will be two real processes.

#### 3.1. User Mode 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.

This section explains, how the virtual hardware, which the User Mode Linux is ported to, is implemented. 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*, *peripheral hardware* and *external hardware*. The two other important hardware related categories which need to be handled by UMLinux are the *real time clock* and *interrupts and exceptions*. The following sections give some details.

The computing core includes the *central processing unit (CPU)*, the *memory management unit (MMU)*, the *random access memory (RAM)* as well as cache memory, interrupt controller, and the buses needed to access memory and extension cards.

Access to parts of the computing core is direct for all programs and possible without the need for device drivers. To the programs running on the machine, the computing core is not visible as a separate device per se. This is also the case in UMLinux.

UMLinux makes no changes to the way processes use the CPU, i.e. there is no simulated CPU, instead the real CPU is accessed directly. Therefore, UMLinux is binary compatible with the host system, i.e. out of the box Linux distributions for the host system can be installed onto a UMLinux machine without any changes. Commercial applications for Linux can be subjected to fault injection using UMLinux 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.

UMLinux simulates RAM with a memory-mapped file, the size of which is the size of the UMLinux machine's "physical RAM". The size of the virtual machine's RAM can be configured separately for each virtual machine.

The MMU is implemented using the `mmap` and `munmap` system-calls [22] which allow to map files into certain address ranges in main memory. Just as MMUs implemented by different hardware differ, the virtual MMU implemented by the UMLinux virtual hardware differs from existent hardware MMUs.

Peripheral devices mainly include storage devices such as harddisks, floppy and cdrom drives, but also input/output devices such as keyboard, mouse and console. Network cards are also grouped under peripheral devices.

For access to peripheral devices, special drivers are necessary. For user programs in a modern operating system, access to these device drivers (and thus to the devices themselves) is possible only through the operating system's system call interface.

Harddisks, as well as floppy and cdrom drives are implemented as files. The "drivers" to access these files are built using the `open`, `close`, `read`, `write` and `lseek` system calls [22] provided by the host Linux.

Keyboard and console device are implemented using an `xterm`.

The network interfaces can be implemented by simply using sockets. The `socket`, `bind` and `fcntl` [22] system calls are used to initialize the virtual network interface, `sendto` and `recvfrom` [22] are used to send to and receive from the interface respectively.

External hardware includes all hardware external to the machine, such as networking hardware (e.g. cabling, switches) or power supply. Often user processes have no access to external hardware at all (e.g. network cabling), sometimes it is possible to communicate with external hardware via special protocols (e.g. network routers).

Power supply of a UMLinux machine (or a group of UMLinux machines) is implemented simply by letting UMLinux process(es) continue to run.

The networking hardware and routing setup is implemented by a separate process, called UM networking process. This process can be configured to route IP packets between virtual and real machines, so that a login from any real workstation onto a UMLinux machine is possible.

Interrupts and exceptions (generically often called *traps*) are generated by the hardware, when certain events occur. An interrupt (asynchronous event), for example, may be generated by the network interface, when a new packet has arrived, or by the IO-system, when the data requested from the harddisk is ready. An exception (synchronous event) is generated by the CPU when some kind of error occurred, examples are the attempt to execute an illegal instruction or a register overflow during floating point calculations.

Traps also cause a switch from the user process which was running at the time the exception occurred to the real kernel. The real kernel then handles the trap on behalf of the user process.

Since we want UMLinux user processes to switch to the UM kernel (not the real kernel) on a trap, we implement traps with signals. `SIGSEGV` is used for a page fault, `SIGILL` for an illegal instruction trap, `SIGBUS` for a bus error and `SIGFPE` for a floating point exception. (For a list of available signals and their meanings see [13].)

The real time clock (RTC) is necessary in each modern operating system to keep track of processes' execution time, file access times, network timeouts etc. The system clock in UMLinux is implemented using the `gettimeofday` system call [22] and the `rdtsc` instruction.

### 3.2. User Mode Kernel

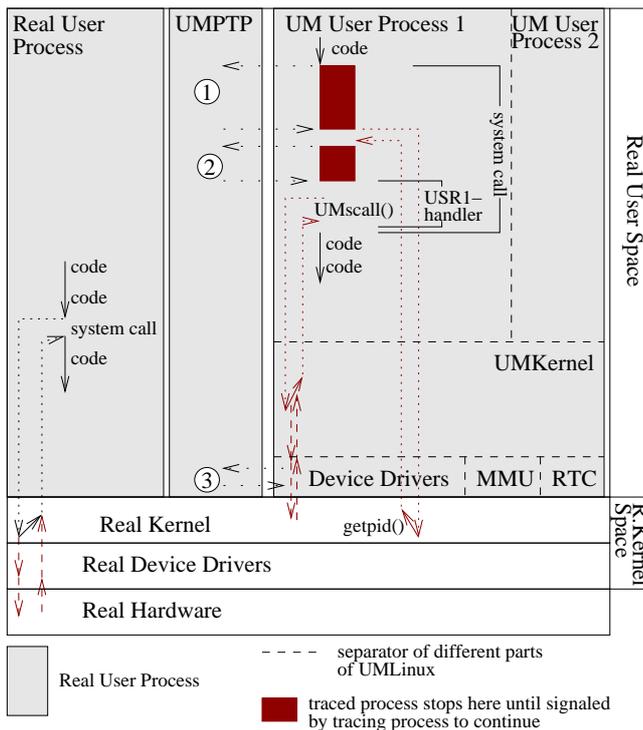
UMLinux directly runs user binaries compiled for the host system, without recompilation or code changes in the user programs. Because these binaries are unmodified, every time they make a system call, they will trap directly into the host kernel instead of into the UM kernel. Therefore, all system calls of user programs running on top of UMLinux must be intercepted and redirected into the UM kernel. To implement UMLinux, 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 paired with the UMLinux process and controlling it via the `ptrace` interface.

This second process is called UM PTracer Process (UMPTP). Every time the UMLinux process makes a system call, returns from a system call or receives a signal it is stopped and UMPTP is notified. Via the `ptrace` interface, UMPTP then has the opportunity to analyze and change the virtual machine's register contents, values at specified memory addresses, arguments or return value of the system call made, or the signal received by the UMLinux process. It can let the UMLinux process continue with or without the signal received (or even with a different signal), return errors from system calls instead of the actual return value, and so on.

The actual redirection of a UM system call is implemented as explained below. Figure 1 shows a schematic representation of the different layers in any modern computer system: (from top to bottom) user space, kernel, device drivers, hardware. Real user processes are shown as grey rectangles. The same layering is visible in the virtual UMLinux machine, which is depicted as the real user process to the right. The "hardware" layer of the UMLinux machine is the real kernel, the device driver layer also includes the implementation of the virtual MMU and RTC.

When redirecting system calls care must be taken to distinguish between system calls made by UM device drivers and UM user processes. The former may not be redirected to the UM kernel, since they occur in a lower layer and implement the UM hardware access. Therefore, these system calls are allowed to continue without redirection (number 3 in Fig. 1). 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.

A system call occurring in a UM user process on the other hand is redirected. To accomplish this, the system call must first be annulled. This is done by saving the register contents and substituting a `getpid` system call instead. The `getpid` system call was chosen, because it does



**Figure 1. Redirection of System Calls in UMLinux**

not change the state of the process calling it (number 1 in Fig. 1). When it returns, the saved register contents are restored and the traced process is allowed to continue with a SIGUSR1 (number 2 in Fig. 1). The functionality of the system call originally called by the UM user process is now implemented by the signal handler for SIGUSR1. Solid arrows in Fig. 1 represent code executed, whereas dotted arrows show where the switches between user and kernel mode occur.

For comparison, a real user process executing a system call is shown to the left of Fig. 1.

### 3.3. 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 few system calls will incur very little 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 164 MB real RAM, the UMLinux machine only had 32 MB 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.

## 4. Injecting Faults into UMLinux

This section explains how to inject hardware faults into UMLinux using a few typical faults as examples. The lists of fault types given here are 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 time-triggered, the exact time may be chosen randomly or explicitly.

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 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.

#### 4.1. Fault Types

Following the hardware categories defined in Sec. 3.1 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.

#### 4.2. Implementation

This section explains, how the fault injection of hardware faults is implemented. We do not have a closer look at system configuration faults, as these are trivially injected by copying the faulty configuration files onto the system.

Generally the faults to be injected are read from a file by UMPTP when the UMLinux machine starts. A configured fault becomes active at the time given. Once one or more faults are active, the appropriate actions must be taken to inject the fault.

Memory faults include transient and permanent bit-flip and stuck-at faults. It is no problem to inject transient faults with UMLinux. This is done by simply writing to the memory mapped file. These faults will only affect the UMLinux process, when it is reading from memory, but will be overwritten by write accesses. Permanent faults can be implemented by remapping the affected pages or using the user debugging registers provided by Intel processors.

CPU faults include bit flips in registers, skipped instructions or wrong branches. Again, injecting transient faults is no problem, since a full copy of the register contents is passed to UMPTP via the `ptrace` interface and modified register contents can be passed back to the UMLinux process. To implement permanent faults, the register contents would have to be checked and possibly modified after every single instruction executed by the UMLinux process, which will lead to a higher overhead.

Faults injected into the computing core will affect both the UM kernel and all UM user processes on the given UMLinux machine, just as would be the case on a real machine.

Peripheral hardware access, such as harddisk, floppy or cdrom drive access is implemented using the `open`, `close`, `read`, `write` and `lseek` system calls. Thus the arguments of these system calls must be checked to see, if

the faulty device is being accessed. If it is, the return value from a `read` or the data passed to a `write` must be modified according to the fault definition, for example to implement several defect blocks on a harddisk. An inaccessible harddisk can be implemented by modifying the return value of the `read` system call to return an error.

To inject faults into the network interface, the arguments and return values of the system calls mentioned in Sec. 3.1 can be modified. By modifying the return value of a `recvfrom`, for example, IP packets with faulty protocol information or wrong checksums can be generated.

Injecting permanent faults into peripheral devices does not incur such a high overhead as injecting permanent faults into the computing core, since the UMLinux process is stopped at every system call anyway, so that UMPTP can redirect the system call to the UM kernel if necessary. Additionally, only those system calls implementing UMLinux 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.

Depending on the setup, a power failure may affect a single or several machines. For all UMLinux machines affected by the virtual power failure, the UMPTP simply kills the corresponding UMLinux processes by sending them a `SIGKILL`.

Defects in virtual external networking hardware can be implemented by configuring the UM networking process to behave like faulty networking hardware. By configuring the UM networking process 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.

#### 4.3. Definition of the Fault Load

The UMPTP, which doubles as fault injector, needs to be configured to inject faults when the UMLinux machine is started. The description of the faults to be injected, the *fault load*, is configured using a simple textfile containing the fault descriptions. These files can be automatically generated from a master file for a series of experiments. For each fault to be injected, the configuration file must contain at least the information listed in the following paragraph.

Of course the location of the fault must be given. It must also be defined, whether the fault is of permanent or transient nature. The time of occurrence of the fault must be given. This is the point of time after system startup, at which the fault becomes active (if it is a permanent fault) or occurs once (if it is a transient fault).

Depending on the fault location, other information may be necessary to properly inject the fault. Fig. 2 gives an example of the definition of a harddisk fault affecting the virtual harddisk implemented by the file named "hda". The fault will become active 360 seconds after the start of the system. According to the fault description, block 302 and the two blocks following it will then become permanently defect.

## 5. Definition of the System Under Test

A useful fault injection framework must include a configurable virtual system under test to model the real system it represents as closely as possible. This includes configuring the virtual hardware as well as the workload running on top of this hardware. The following sections describe how the hardware setup and workload are defined within the proposed framework.

### 5.1. Description of Hardware Setup

The virtual computer system is made up of one or more machines possibly including networking hardware. For each virtual machine, the virtual hardware configuration must be defined. An example of such a machine definition is given in Fig. 3. It includes listing the files implementing the virtual harddisks, floppy and cdrom drives. For the virtual network interfaces, configuration files must be given, defining the virtual hardware address of the network interface (an Ethernet address in our case) and the real port number to be used for the socket created on the host system to implement the network interface. The size of the virtual memory is set here, too.

The next set of parameters is specific to `umlilo`, the Linux Loader of UMLinux. The path to the `umlilo` executable must be specified as well as the kernel to load and the device which is to be mounted as root filesystem. These are the same parameters passed to `lilo`, the Linux Loader of the real Linux from which `umlilo` was derived. Additional boot parameters can be given if necessary.

```
FAULT_HD
FAULT_PERMANENT
360 # time of occurrence
hda # device
302 # block
2 # num of consecutive blocks affected
```

**Figure 2. Configuration of a Harddisk Fault**

```
MACHINE
DISK = hda
DISK = hdb
CDROM = cdrom
NETCFG = eth0
MEMORY = 32

LILO = ../umlilo
ROOT = /dev/hda
KERNEL = ./vmlinux
BOOTPARAMS =

FAULTLOAD = faultload.txt
DIR = /home/umlunix/sim0
HOST = host05
END
```

**Figure 3. Virtual Machine Hardware Configuration**

The last set of parameters is needed for technical purposes. They give the name of the file containing the fault load definition, the working directory of the UMLinux process and the name of the host machine.

The UM networking process is configured with several files, one for each of its virtual Ethernet interfaces. In addition to the same information as given in the configuration file for a virtual machine's networking interface, this file specifies the virtual hardware addresses and real socket addresses of all virtual machines reachable through this interface. The file also specifies the IP-address of the UM networking process in the virtual network of UMLinux machines, the virtual networks' mask and broadcast address. If the UM networking process is to provide a connection to the real world, it must be configured appropriately. The rest of the parameters specify the name of the UM networking process executable, its working directory and the host it should be started on. If faults are injected into external networking hardware, a file containing the fault load for the UM networking process must be provided.

### 5.2. Description of 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 copied

into the files which will become the virtual harddisks of the targeted UMLinux machines. This must be done prior to starting the experiments.

The clients can be programs or scripts which generate a load on the system under test. 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 should not be subject to fault injection, because they are not part of the system under test. If these processes are started on UMLinux machines, these machines also should be exempt from fault injection. When the UM networking process is employed to provide a connection between the virtual and the real world, the load generating clients can be started on real machines.

## 6. Running a Test

This section summarizes the general approach to conducting an experiment and gives a small example showing the functionality of the framework. This paper proposes a general, extensible and flexible framework which can be used to test the fault-tolerance of systems including OS and networking aspects. The aim of this section is to show that this framework works as required to conduct such tests. Although a library of preset test cases would be a nice add-on to the framework, this is by no means a requirement.

### 6.1. General Approach

Running a single test will consist of the steps summarized in the following list.

**Preparation** This step includes preparing all the files necessary for fault load and virtual system and network setup given in Sec. 4.2 and 5 as well as the files which will become the UMLinux machines virtual harddisks.

**Fault Injection** This step includes starting the system under test, the UM networking process and the load generating clients. Faults will be injected according to the fault load configured. The experiment is stopped when either a crash of the system under test is detected or a configured amount of time has passed.

**Evaluation** The evaluation step includes gathering the available data and extracting the wanted information from it.

In addition to a fault injection experiment, a fault free run may be necessary for purposes of comparison.

The preparation and fault injection steps have been treated in detail in the previous sections. The evaluation step is highly dependent on the system under test. The data for the evaluation step can be taken from the following locations:

**Server Logs** Each Linux Kernel, whether UM or real, can be configured to log errors detected by the kernel. Server processes can usually also be configured to log their activities in different levels of detail.

**Files** 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 Logs** The clients can be configured to log fault free and faulty server responses as well as timing information useful for calculating results.

Depending on the system under test, the following result measures may be of interest:

- data integrity in the presence of faults
- response time in the presence of faults
- vulnerability to certain types of faults
- reachability in the presence of faults

Which of these result measures are calculated and how exactly this is done is highly dependent on the system under test and cannot be generalized.

### 6.2. Example

The purpose of this section is to show the functionality of the fault injection framework we implemented. Since the flexible framework allows a great variety of faults to be injected, it is not possible to cover them all. We therefore chose a relatively simple test setup and one of each processor, memory, network and harddisk fault to inject.

The general setup of the example system is the following. The system consists of a web and database-server (DB\_WWW), a name-server (DNS) and a client workstation (CLIENT) all running Linux. DB\_WWW runs Apache as web-server and MySQL as database-server. DB\_WWW has 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 runs bind as name-server. Processor, memory, harddisk and network faults were injected into DB\_WWW. The workload is generated by Perl-scripts running on CLIENT. Two different workloads were used. The first (WL1) consisted of 230 SELECT statements via the webinterface accessing records evenly distributed throughout the database. The second (WL2) consisted of randomly generated read and write database accesses via the webinterface. A series of 100 INSERT, followed by 150 SELECT and 100 DELETE statements 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 recognise 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 logfile generated by the clients.

**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. The workload and database used were WL2 and DB2. 447 single runs were conducted.

**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. The workload and database used were WL2 and DB2. 282 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 (excluding the first 2000 blocks). 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 (with step of 5) were injected, with activation time being 150. 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)

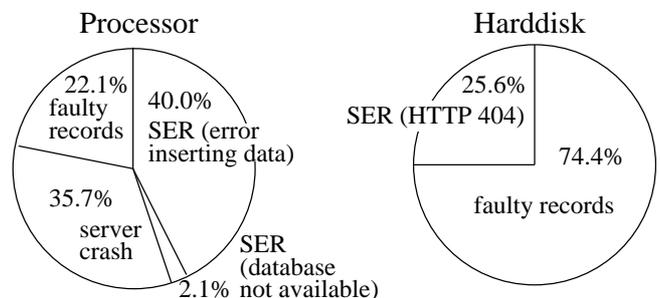
- delayed response
- server error response (SER)
- server crash or hang

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 the testrun. Not all of these possible behaviors were observed for each type of fault injected.

Figure 4 shows the percentage of different types of behavior observed in the example setup for the processor and hard-disk faults.

Concerning the processor faults, we observed, that two server behaviors dominated. Either, the processor fault did not visibly affect the server (86.1 % of the testruns were faultfree), or the client completely lost connection with the server and could not regain it during this testrun (11.1 % of the testruns). 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. During the rest of the testruns some errors were observed, but the testrun could be completed. The percentages of the single errors observed are shown in the left part of Fig. 4.

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 webserver or database-server 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



**Figure 4. Results of the Experiments (Faulty Runs Only)**

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).

Harddisk faults, since confined to HD2, could not affect the operating systems or web- and databaseserver 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 faults observed are shown in the right part of Fig. 4.

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. 5 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.

## 7. 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.

Future work will certainly include smoothing some of the rough edges of the prototype and including user friendly graphical interfaces to aid in the preparation of experiments. We also plan to include a kind of library of example setups for some typical setups. These can serve as a starting point for customization of a user defined setup. This should make it easier to create new system setups.

## References

[1] I. S. 1003.1b 1993. IEEE Standard for Information Technology — Portable Operating System Interface (POSIX)

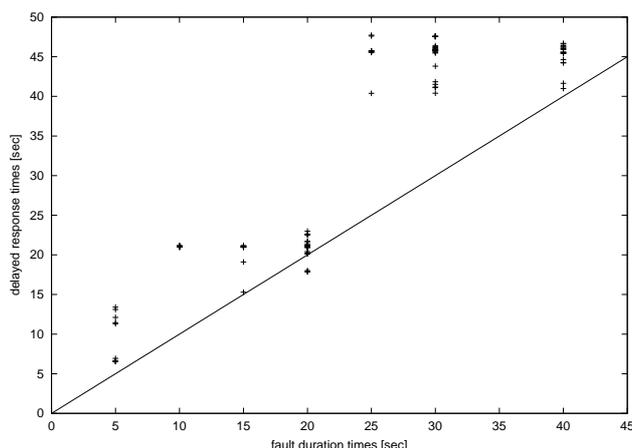


Figure 5. Network Delay Times

— Part 1: System Application Program Interface (API) — Amendment 1: Realtime Extension [C Language], 1994.

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

[3] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O’Reilly & Associates, Inc., 2000.

[4] 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.

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

[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 Tecnologia da Universidade de Coimbra, Portugal; Friedrich-Alexander Universität, Erlangen-Nürnberg, Germany; Microsoft Research, Cambridge, UK; Universidad Politecnica de Valencia, Spain. URL: <http://www.laas.fr/DBench/>, 2001.

[7] J. Dike. A user-mode port of the Linux kernel. In *4th Annual Linux Showcase & Conference, Atlanta*, 2000.

[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] Linux Programmer’s Manual. Included in any Linux distribution, 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] 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.
- [18] J. Scambray, S. McClure, and G. Kurtz. *Hacking Exposed, 2nd Edition*. McGraw-Hill, New York, 2000.
- [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] 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.
- [22] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing Company, Reading MA, 1992.
- [23] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 2001.
- [24] VMware Inc. VMware. URL: <http://www.vmware.com/>, 2001.