# Failure Resilience for Device Drivers

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum
*Computer Science Dept., Vrije Universiteit, Amsterdam, The Netherlands*
*{jnherder, herbertb, beng, philip, ast}@cs.vu.nl*

## Abstract

Studies have shown that device drivers and extensions contain 3–7 times more bugs than other code and thus are more likely to fail. Therefore, we present a failure-resilient operating system that can recover from dead device drivers and other critical components—primarily through monitoring and replacing malfunctioning components on the fly—transparent to applications and without user intervention. This paper focuses on the post-mortem recovery procedure. We explain the working of our defect detection mechanism, the policy-driven recovery procedure, and post-restart reintegration of components. Furthermore, we discuss the concrete steps taken to recover from network, block device, and character device driver failures. Finally, we evaluate our recovery mechanism using performance measurements, software fault-injection, and an analysis of the reengineering effort.

**Keywords:** Operating System Dependability, Failure Resilience, Device Driver Recovery.

## 1 INTRODUCTION

Perhaps someday software will be bugfree, but for the moment all software contains bugs and we had better learn to coexist with them. Nevertheless, a question we have posed is: "Can we build dependable systems out of unreliable, buggy components?" In particular, we address the problem of failures in device drivers and other operating system extensions. In most operating systems, such failures threaten to disrupt normal operation.

In many other areas, failure-resilient designs are common. For example, RAIDs are disk arrays that continue functioning even in the face of drive failures. ECC memories can detect and correct bit errors transparently without affecting program execution. Disks, CD-ROMs, and DVDs also contain error-correcting codes so that read errors can be corrected on the fly. The TCP protocol provides reliable data transport, even in the face of lost, misordered, or garbled packets. DNS can transparently deal with routing failures. Finally, *init* automatically respawns crashed daemons in the UNIX application layer. In all these cases, software masks the underlying failures and allows the system to continue as though no errors had occurred.

We have attempted to extend these ideas to the operating system internals. In particular, we want to tolerate and mask failures of device drivers and other extensions. Recovery from such failures is particularly important, since extensions are generally written by third parties and tend to be buggy [8, 38]. Unfortunately, recovering from driver failures is also hard, primarily because drivers are closely tied to the rest of the operating system. In addition, it is sometimes impossible to tell whether a driver crash has led to data loss. Nevertheless, we have designed an operating system consisting of multiple isolated user-mode components that are structured in such a way that the system can automatically detect and repair a broad range of defects [14, 9, 29], without affecting running processes or bothering the user. The architecture of this system is shown in Fig. 1.

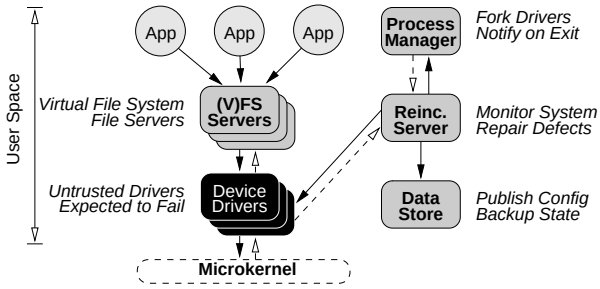


**Figure 1:** Architecture of our failure-resilient operating system that can recover from malfunctioning device drivers.

In this paper, we focus on the post-mortem recovery procedure that allows the system to continue normal operation in the event of otherwise catastrophic failures. We rely on a stable set of servers to deal with untrusted components. The *reincarnation server* manages all system processes and constantly monitors the system's health. When a problem is detected, it executes a policy script associated with the malfunctioning component to guide the recovery procedure. The *data store* provides naming services and can be used to recover lost state after a crash. Changes in the system are broadcast to dependent components through the data store's publish-subscribe mechanisms in order to initiate further recovery and mask the problem to higher levels. This is how failure resilience works: a failure is detected, the defect is repaired, and the system continues running all the time with minimum disturbance to other processes.

While our design can recover from failures in all kinds of components, including the TCP stack and simple servers, the focus of our research is to deal with buggy device drivers. Studies on software dependability report fault densities of 2–75 bugs per 1000 lines of executable code [3, 40], but drivers and other extensions—which typically comprise 70% of the operating system code—have a reported error rate that is 3 to 7 times higher [8], and thus are relatively failure-prone. For example, 85% of Windows XP crashes can be traced back to driver failures [38]. Recovery of drivers thus is an effective way to improve operating system dependability.

### 1.1 Contribution

We have built a failure-resilient operating system to improve operating system dependability. Our work is based on MINIX 3, which runs all servers and drivers as isolated user-mode processes, but is also applicable to other operating systems. This architecture allowed us to add mechanisms to detect and transparently repair failures, resulting in the design shown in Fig. 1. While several aspects of MINIX 3 have been published before [19, 20, 18], this is the first time we discuss the recovery of malfunctioning device drivers in detail.

The remainder of this paper is organized as follows. We first survey related work in operating system dependability (Sec. 2). Then we present our failure model (Sec. 3), discuss our isolation architecture (Sec. 4), introduce the defect detection mechanisms and policy-driven recovery procedure (Sec. 5). We illustrate our ideas with concrete recovery schemes for network, block device, and character device drivers (Sec. 6). We also evaluate our system using performance measurements, software fault-injection, and an analysis of reengineering effort (Sec. 7). Finally, we conclude (Sec. 8).

## 2 DEPENDABILITY CONTEXT

This work needs to be placed in the context of operating system dependability. Our design represents a special case of microreboots [6, 7], which promote reducing the mean time to recover (MTTR) in order to increase system availability. We apply this idea to drivers and other operating system extensions.

Several examples of microreboot are found in the context of operating systems. Solaris 10 [37] provides tools to manage UNIX services running in the application layer and can automatically restart crashed daemons. Nooks [38, 39] proposes in-kernel wrapping of components to isolate driver failures and supports recovery through shadow drivers that monitor communication between the kernel and driver. SafeDrive [43] combines wrapping of the system API with type safety for extensions written in C and provides recovery similar to Nooks. QNX [21] provides software watchdogs to catch and recover from intermittent failures in memory-protected subsystems. Paravirtualization [26, 12] supports driver isolation by running each driver in a dedicated, paravirtualized operating system and can recover by rebooting a failed virtual machine. In contrast to these approaches, we take the UNIX model to its logical conclusion by putting all servers and drivers in unprivileged user-mode processes and support microreboots through a flexible, policy-driven recovery procedure.

Numerous other approaches attempt to increase operating system dependability by isolating components. Virtual machine approaches like VM/370 [34], VMware [35], and Xen [2] are powerful tools for running multiple services in isolation, but cannot prevent a bug in a device driver from crashing the hosted operating system. User-mode drivers have been used before, but, for example, Mach [11] leaves lots of code in the kernel, whereas $L^4$Linux [15] runs the operating system in a single server on top of L4 [24]. Other designs compartmentalize the entire operating system, including GNU Hurd [5], SawMill Linux [13], and NIZZA [17]. Recently, user-mode drivers also made there way into commodity systems such as Linux [25] and Windows [27]. These systems differ from our work in that we combine proper isolation in user space with driver recovery.

Finally, language-based protection and formal verification can also be used to build dependable systems. OKE [4] uses instrumented object code to load safely kernel extensions. VFiasco [22] is an attempt at formal verification of the L4 microkernel. Singularity [23] uses type safety to provide software isolated processes. These techniques are complementary to our approach, since our user-mode servers and drivers can be implemented in a programming language of choice.

## 3 FAILURE MODEL

In our work, we are interested in the notion of a failure, that is, a deviation from the specified service [30], such as a driver crash. We are less interested in erroneous system states or the exact underlying faults. Once a failure has been detected, as described in Sec. 5.1, we perform a microreboot [6] of the failing or failed component in an attempt to repair the system. The underlying idea is that a large fraction of software failures are cured by rebooting, even when the exact failure causes are unknown.

Our system is designed to deal with intermittent and transient failures in device drivers. We believe that this focus has great potential to improve operating system dependability, because (1) transient failures represent a main source of downtime in software systems [14, 9, 29] and (2) device drivers are relatively failure-prone [8, 38]. Transient failures that can be handled by our design include failstop and crash failures, such as exceptions triggered by unexpected input; panics due to internal inconsistencies; race conditions caused by a specific configuration or unexpected hardware timing issues; and aging bugs that cause a component to fail over time, for example, due to memory leaks. While hard to track down, these kinds of problems tend to go away after a restart and can be cured by performing a microreboot, whereby the malfunctioning component is replaced with a fresh copy. In fact, our design can deal with a somewhat broader range of failures, since we can also start a patched version of the driver, if available.

Of course, there are limits to the failures our design can deal with. To start with, we cannot deal with Byzantine failures, including random or malicious behavior. For example, consider a printer driver that accepts a print job and responds normally, but, in fact, prints garbage. Such bugs are virtually impossible to catch in any system. In general, end-to-end checksums are required to prevent silent data corruption. Furthermore, algorithmic and deterministic failures that repeat after a microreboot cannot be cured, but can be found more easily through testing. Our design also cannot deal with performance failures where timing specifications are not met, although the use of heartbeat messages allows us to detect unresponsive components. Finally, our system cannot recover when the hardware is broken or cannot be reinitialized by a restarted driver. Nevertheless, it may be possible to perform a hardware test and switch to a redundant hardware interface, if available. More research in this area is needed, though.

In the remainder of this paper we focus on the recovery from transient driver failures, which, as argued above, represent an important area where our design helps to improve system dependability.

## 4 ISOLATION ARCHITECTURE

Strict isolation of components is a crucial prerequisite to enable recovery, since it prevents problems in one server or driver from spreading to a different one, in the same way that a bug in a compiler process cannot normally affect a browser process. Taking this notion to our compartmentalized operating system design, for example, a user-mode sound driver that dereferences an invalid pointer is killed by the process manager, causing the sound to stop, but leaving the rest of the system unaffected. Although the precise mechanisms are outside the scope of this paper, a brief summary of MINIX 3's isolation architecture is in place.

To start with, each server and driver is encapsulated in a private, hardware-protected address space to prevent memory corruption through bad pointers and unauthorized access attempts, just like for normal applications. Because the memory management unit (MMU) denies direct memory access, the kernel provides a virtual copy call that enables processes to copy data between address spaces in a capability-protected manner. The process that wants to grant access to its memory needs to create a capability describing the precise memory area and access rights, and pass an index to it to the other party.

Direct memory access (DMA) is a powerful I/O mechanisms that potentially can be used to bypass the memory protection offered by our system. On commodity hardware, we can deny access to the DMA controller's I/O ports, and have a trusted driver mediate all access attempts. However, this technique is impractical, since it requires manual checking of each device to see if it uses DMA. Modern hardware provides effective protection in the form of an I/O MMU [16]. To perform DMA safely a driver must first request the kernel to set up the I/O MMU by passing an index to a memory capability similar to the one described above. The overhead of this protection is a few microseconds to perform the kernel call, which is generally amortized over the costs of the I/O operation.

In addition, we have reduced the privileges of each component to a minimum according to the principle of least authority [33]. The privileges are passed to the trusted reincarnation server when a component is loaded through the *service* utility, which, in turn, informs the other servers, drivers, and microkernel so that the restrictions can be enforced at run-time. System processes are given an unprivileged user and group ID to restrict among other things file system access. For servers, we restrict the use of IPC primitives, system calls, and kernel calls. For device drivers, the same restrictions apply, and in addition, we restrict access to I/O ports, IRQ lines, and device memory such as the video memory.

## 5 RECOVERY PROCEDURE

In this section, we describe the general recovery procedure in the event of failures. The ability of our system to deal with dead drivers implies that we can dynamically start and stop drivers while the operating system is running. Starting a new device driver is done through the *service* utility, which forwards the request to the reincarnation server. The following arguments are passed along with the request: the driver's binary, a stable name, the process' precise privileges, a heartbeat period, and, optionally, a parametrized policy script—a shell script that is called after a driver failure to guide the recovery procedure. If all arguments make sense, the reincarnation server forks a new process, sets the process' privileges, and executes the driver's binary. From that point on, the driver will be constantly guarded to ensure continuous operation—even in the event of a failure. How the reincarnation server can detect defects and how the policy-driven recovery procedure works is described below.

### 5.1 Defect Detection

While a human user observes driver defects when the system crashes, becomes unresponsive, or starts to behave in strange ways, the operating system needs other ways to detect failures. Therefore, the reincarnation server monitors the system at run-time to find defects. The various inputs that can cause the recovery procedure to be initiated are:

1. Process exit or panic.
2. Crashed by CPU or MMU exception.
3. Killed by user.
4. Heartbeat message missing.
5. Complaint by other component.
6. Dynamic update by user.

At any point in time the reincarnation server has accurate information about the liveness of all servers and drivers, since it is the parent of all system processes. When a server or driver crashes, panics or exits for another reason, the process manager will notify the reincarnation server with a SIGCHLD signal, according to the POSIX specification. At that point it collects and inspects the exit status of the exitee, which leads to the first three defect classes. Since a process exit is directly reported by the process manager, immediate action can be taken by the reincarnation server.

In addition, the reincarnation server can proactively check the system's state. Depending on the configuration passed through *service*, the reincarnation server can periodically request drivers to send a heartbeat message. Failing to respond N consecutive times causes recovery to be initiated. Heartbeats help to detect processes that are 'stuck,' for example, in an infinite loop, but do not protect against malicious code. To prevent bogging down the system status requests and the consequent replies are sent using nonblocking messages.

Furthermore, the reincarnation server can be used as an arbiter in case of conflicts, allowing authorized servers to report malfunctioning components. How a malfunction is precisely defined depends on the components at hand, but in general has to do with protocol violations. For example, the file server can request replacement of a disk driver that sends unexpected request messages or fails to respond to a request. The authority to replace other components is part of the protection file that specifies a process' privileges.

Finally, faulty behavior also can be noticed by the user, for example, if the audio sounds weird. In such a case, the user can explicitly instruct the reincarnation server to restart a driver or replace it with a newly compiled one. As another example, latent bugs or vulnerabilities may lead to a dynamic update as soon as a patch is available. Since about a quarter of downtime is caused by reboots due to maintenance [41], such dynamic updates that allow patching the system on the fly can significantly increase system availability.

### 5.2 Policy-Driven Recovery

By default the reincarnation server directly restarts a crashed component, but if more flexibility is wanted, the administrator can instruct it to use a parametrized policy script that governs the actions to be taken after a failure. Policy scripts can be shared, but dedicated recovery scripts can be associated with individual servers and drivers as well. When a malfunctioning component is detected, the reincarnation server looks up the associated policy script and executes it. Input arguments are which component failed, the kind of failure as indicated in Sec. 5.1, the current failure count, and the parameters passed along with the script. The primary goal of the policy script is to decide when to restart the malfunctioning component, but other actions can be taken as well. Restarting is always done by requesting the reincarnation server to do so, since that is the only process with the privileges to create new servers and drivers.

The simplest policy script immediately tries to restart the failed component, but the policy-driven recovery process can use the information passed by the reincarnation server to make decisions about the precise recovery steps taken. For example, consider the generic policy script in Fig. 2. Lines 1–4 process the arguments passed by the reincarnation server. Then, lines 6–10 restart the

*a generic recovery script*

```
1  component=$1            # component name
2  reason=$2               # numbers as in Sec. 3.1
3  repetition=$3           # current failure count
4  shift 3                 # shift to parameters
5
6  if [ ! $reason -eq 6 ]; then
7    sleep $((1 << ($repetition - 1)))
8  fi
9  service restart $component
10 status=$?
11
12 while getopts a: option; do
13 case $option in
14 a)
15   cat << END | mail -s "Failure Alert" "$OPTARG"
16     failure: $component, $reason, $repetition
17     restart status: $status
18 END
19 ;;
20 esac
21 done
```

**Figure 2:** A parametrized, generic recovery script. Binary exponential backoff is used before restarting, except for dynamic updates. Optionally, a failure alert is sent.

component, possibly after an exponentially increasing delay to prevent bogging down the system in the event of repeated failures. Note that the backoff protocol is used only for unexpected failures, and not for dynamic updates that are requested explicitly. Finally, lines 12–21 represent an extension that sends an e-mail alert when the parameter '-a' is passed.

Using shell scripts provides a lot of flexibility and power for expressing policies. For example, a dedicated policy script can support the administrator to recover from failures in the network server. Such a failure closes all open network connections, including the sockets used by the X Window System. Among other things, recovery requires restarting the DHCP client and X Window System, which can be specified conveniently in a dedicated policy script. As another example, when a required component cannot be recovered or fails too often, the policy script may reboot the entire system, which clearly is better than leaving the system in an unusable state. At the very least, the policy script can log the failing component and its execution environment for future inspection. The ability of our system to pinpoint precisely the responsible components might have consequences for software vendors, which may help in building more dependable systems [42].

### 5.3 Post-Restart Reintegration

A restart of an operating system process is similar to the steps taken when a new component is started through the *service* utility, although the details differ. Some extra work needs to be done, mainly because the privileges and capabilities of dependent components that refer to the restarted component need to be reset. In general, simple update requests suffice, but these issues illustrate that interprocess dependencies complicate the recovery. For example, our design uses unique IPC endpoints, so that messages cannot be wrongly delivered during a failure. As a consequence, a component's endpoint changes with each restart, and the IPC capabilities of dependent processes must be updated accordingly. This update is done well before the dependent components learn about the restart, as discussed next.

Once a component has been restarted it needs to be reintegrated into the system. Two steps need to be distinguished here. First, changes in the operating system configuration need to be communicated to dependent components in order to initiate further recovery and mask the problem to higher levels. This information is disseminated through the data store, a simple name server that stores stable component names along with the current IPC endpoint. The reincarnation server is responsible for keeping this naming information up to date. The data store implements a publish-subscribe mechanism, so that components can subscribe to naming information of components they depend on. This design decouples producers and consumers and prevents intricate interaction patterns of components that need to inform each other. For example, the network server subscribes to updates about the configuration of Ethernet drivers by registering the expression '$\mathsf{eth}.*$'.

Second, a restarted component may need to retrieve state that is lost when it crashed. The data store also serves as a database server that can be used by system processes to store privately a backup copy of certain data. By using the data store, lost state can be retrieved after a restart. Authentication of restarted components is done with help of the naming information that is also kept in the data store. When private data is stored, a reference to the stable name is included in the record, so that authentication of the owner is possible even if its endpoint changes. In our prototype implementation, state management turns out to be a minor problem for device drivers, but is crucial for more complex services. In fact, none of our device drivers currently uses the data store to backup internal state. However, all mechanisms needed to recover from failures in stateful components, such as servers, are present. Transparent recovery of servers is part of our future work.

# 6 DRIVER RECOVERY SCHEMES

Although our design in principle can handle both server and driver failures, our current focus is to reincarnate dead drivers, since such failures are more common [8, 38]. If a driver failure cannot be handled at the server level, it will be reported to the application that made the I/O request, which, in turn, notifies the user of the problem, if it cannot recover. Different recovery schemes exist depending on the type of device, as summarized in Fig. 3. Full, transparent recovery without bothering the user is possible for both network and block device drivers.

| Driver | Recovery | Where | Section |
|---|---|---|---|
| Network | Yes | Network server | Sec. 6.1 |
| Block | Yes | File server | Sec. 6.2 |
| Character | Maybe | Application | Sec. 6.3 |

**Figure 3:** Driver recovery schemes. Only network and block device drivers allow transparently recovery.

The recovery schemes discussed here pertain not only to failures. Our design also allows the administrator to dynamically update the driver—even when I/O is in progress. In this case, the reincarnation server first requests the driver to exit by sending it a SIGTERM signal, followed by a SIGKILL signal, if the driver does not comply. The steps that are taken after the reincarnation server caused the driver to exit are similar to those for a failure. Most other operating system cannot dynamically replace active drivers on the fly like we do.

## 6.1 Recovering Network Drivers

If a network driver fails, full recovery transparent to the application is possible. We have implemented support for Ethernet driver recovery in MINIX 3's network server, INET. If the application uses a reliable transport protocol, such as TCP, the protocol handles part of the recovery. If data is lost or corrupted, the network server (or its peer at the other end of the connection) will notice and reinsert the missing packets in the data stream. If an unreliable protocol, such as UDP, is used, loss of data is explicitly tolerated, but if need be, application-level recovery is possible, as illustrated in Fig. 4.

The recovery procedure starts when the process manager informs the reincarnation server about the exit of one of its children, as discussed in Sec. 5.1. The reincarnation server collects the pieces of the dead driver and looks up the driver's details in its internal tables. It then runs the associated policy script to restart the failed driver. Because the network server is subscribed to information about Ethernet drivers, it is automatically notified by the data store if the configuration changes. If

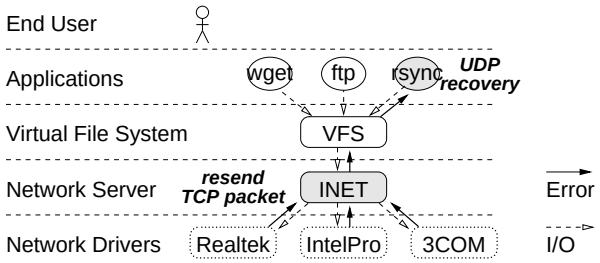


**Figure 4:** Network driver recovery is done by the network server or application, transparent to the user.

the network server tries to send data in the short period between the driver crash and the consequent restart, the request fails and is postponed until the driver is back. Upon notification by the data store, the network server checks its tables to find out if the driver update concerns a new driver or a recovered one, and starts its internal recovery procedure. This procedure closely mimics the steps that are taken when the driver is first started. The Ethernet driver is reinitialized to put it in promiscuous mode and told to resume I/O. The Ethernet driver is stateless and does not need to retrieve lost state from the data store—although it could do so, were that necessary.

## 6.2 Recovering Block Drivers

We now zoom in on block device driver crashes and describe the recovery procedure as implemented by the native MINIX 3 file server, MFS. If a disk driver (such as the hard disk, RAM disk, or floppy driver) crashes, transparent recovery without loss of data is possible, since disk block I/O is idempotent. If I/O was in progress at the time of the failure, the IPC rendezvous will be aborted by the kernel, and the file server marks the request as pending. Then, the file server blocks and waits until the disk driver has been restarted and pending I/O requests can be retried, as illustrated in Fig. 5.

In contrast to other drivers, disk driver recovery is currently not policy driven, since that would require reading the recovery script from disk. Instead, the reincarnation server directly restarts failed disk drivers from a copy in RAM.[1] Like the other drivers in our system, disk drivers are stateless and do not need to retrieve lost state from the data store, although they could were that necessary. Once the driver has been restarted, the reincarnation server publishes the new IPC endpoint in the data store and the file server is notified. At this point, the file server updates its device-driver mappings, and reini-

[1] If policy-driven recovery is needed for disk drivers, the system can be configured with a dedicated RAM disk to provide trusted storage for crucial data, such as the driver binaries, the shell, and policy scripts. We have written a small 450-line RAM disk driver for this purpose.

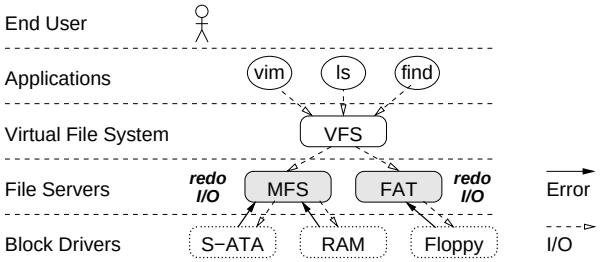


**Figure 5:** Block device driver recovery is done by the file server, transparent to applications and users.

tializes the disk driver by reopening minor devices, if needed. Finally, the file server checks whether there are any pending I/O requests, and if so, it reissues the failed disk operations. At this point the file server resumes normal operation and continues to handle pending I/O requests from other user applications.

As an aside, our design is meant to provide recovery in the event of driver crashes, but is not designed to detect data corruption, as discussed in Sec. 3. While the TCP protocol automatically eliminates this problem for network driver crashes, disk driver crashes can potentially corrupt the data on disk. However, our design may be combined with end-to-end checksums, as in IRON file systems [32], to prevent silent data corruption.

### 6.3 Recovering Character Drivers

Character device drivers cannot be transparently recovered, since it is impossible to tell whether data was lost. Deciding which part of the data stream was successfully processed and which data is missing is an undecidable problem. If an input stream is interrupted due to a device driver crash, input might be lost because it can only be read from the controller once. Likewise, if an output stream is interrupted, there is no way to tell how much data has been output to the controller, and full recovery is also not possible. Therefore, such failures are reported to the application layer where further recovery might be possible, as illustrated in Fig. 6.

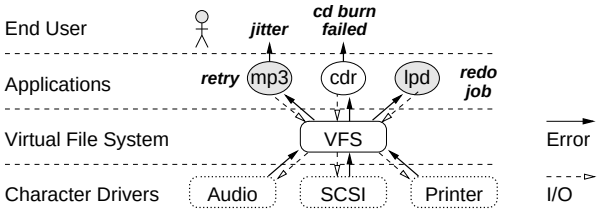


**Figure 6:** Recovery for character device drivers. Errors are always pushed up, but need to be reported the user only if the application cannot recover.

For historical reasons, most applications assume that a driver failure is fatal and immediately give up, but our prototype in fact supports continuous operation when applications are made recovery-aware. For example, it may be possible to modify the printer daemon such that it automatically reissues failed print requests (without bothering the user). While transparent recovery is not possible—duplicate prints may result in this case—the user benefits from this approach. As another example, an MP3 player could continue playing a song after a driver recovery at the risk of jitter. Only when the application layer cannot handle the failure, the user needs to be informed. For example, continuing the CD or DVD burn process if SCSI driver fails will produce a corrupted copy, so the error must be reported to the user.

## 7 EXPERIMENTAL EVALUATION

We have evaluated our recovery mechanisms in various ways. First, we discuss the performance overhead introduced by our recovery mechanisms. Then, we briefly report on the results of software fault-injection. Finally, we quantify the reengineering effort needed to prototype our recovery mechanisms in MINIX 3.

### 7.1 Performance Overhead

To determine the performance overhead introduced by our recovery mechanisms we simulated driver crashes while I/O was in progress and compared the performance to the performance of an uninterrupted I/O transfer. We performed this test for both block device and network drivers. The crash simulation was done using a tiny shell script that first initiates the I/O transfer, and then repeatedly looks up the driver's process ID and kills the driver using a SIGKILL signal. The test was run with varying intervals between the simulated crashes. The recovery policy that was used for these tests directly restarts the driver without introducing delays. After the transfer we verified that no data corruption took place. In all cases, full recovery transparent to the application and without user intervention was possible.

We first measured the overhead for the recovery of network drivers using the RealTek 8139 Ethernet driver. In this case, we initiated a TCP transfer using the *wget* utility to retrieve a 512-MB file from the Internet. We ran multiple tests with the period between the simulated crashes ranging from 1 to 15 seconds. In all cases, *wget* successfully completed, with the only noticeable difference being a small performance degradation as shown in Fig. 7. To verify that the data integrity was preserved we compared the MD5 checksums of the received data

the original file. The mean recovery time for the RealTek 8139 driver failures is 0.48 sec, which is, in part, due to the TCP retransmission timeout. The uninterrupted transfer time is 47.41 sec with a throughput of 10.8 MB/s. The interrupted transfer times range from 47.85 sec to 62.98 sec with a throughput of 10.7 MB/s and 8.1 MB/s, respectively. The loss in throughput due to Ethernet driver failures ranges from 25% to just 1% in the best case.

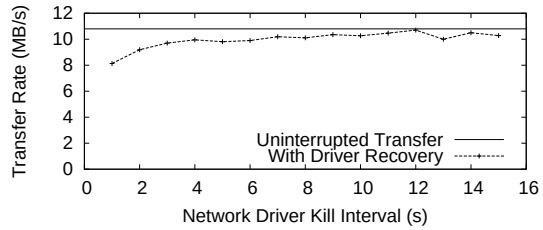


**Figure 7:** Networking throughput when using *wget* retrieve a 512-MB file from the Internet while repeatedly killing an Ethernet driver with various time intervals.

We also measured the overhead of disk driver recovery by repeatedly sending a SIGKILL signal to the S-ATA hard disk driver while reading a 1-GB file filled with random data using *dd*. The input was immediately redirected to *sha1sum* to calculate the SHA-1 checksum. Again, we killed the driver with varying intervals between the simulated crashed. The data transfer successfully completed in all cases with the same SHA-1 checksum. The transfer rates are shown in Fig. 8. The uninterrupted disk transfer completed in 31.33 sec with a throughput of 32.7 MB/sec. The interrupted transfer shows a transmission time ranging from 83.06 sec to 34.73 sec, with a throughput of 12.3 MB/s and 30.5 MB/s, for simulated crashes every 1 and 15 sec, respectively. The performance overhead of disk driver recovery ranges from 62% to about 7% in this test. The higher recovery overhead compared to the previous experiment is due to the much higher I/O transfer rates.

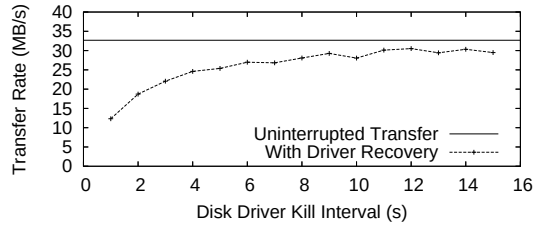


**Figure 8:** Throughput of uninterrupted disk transfers and throughput in case of repeated disk driver failures while reading a 1-GB file filled with random data from disk.

## 7.2 Fault-Injection Testing

To test the capability of our system to withstand and recover from driver failures, we also simulated failures in our drivers by means of software fault-injection. We based our experiments on two existing fault injectors that mutate binary code [31, 38, 39]. This was shown to be most representative for real failures [28]. In our experiments we used the following seven fault types: change source register, change destination register, garble pointer, use current register value rather than paramater passed, invert termination condition of a loop, flip a bit in an instruction, or elide an instruction. These faults emulate programming errors common to operating system code according to earlier studies [36, 10].

In one particular experiment we targeted the RTL8029 Ethernet driver. We ran over 12,500 fault-injection trials that each injected 1 randomly selected fault into the running driver. After each trial we used the *daytime* service to check the driver's status. The distribution of the number of faults needed to prevent the driver from doing its work and cause it to crash it is plotted in Fig. 9. While driver disruption usually happens after only a few faults, the number of faults needed to induce a crash shows a long tail to the right. The experiment led to 347 detectable crashes that initiated the recovery procedure: 226 exits due to an internal panic, 109 kill signals due to a CPU or MMU exceptions, and 12 restarts due to missing heartbeats.

The fault-injection experiments demonstrated that our design can successfully recover from common, transient failures and provide continuous operation in more than 99.9% of the recovery attempts. We ran many experiments and in only a very small number of cases was the network card confused and could not be reininitialized by the restarted driver. Instead a low-level BIOS reset was needed. If the card had a 'master reset' command the problem could be solved by our driver, but our Realtek RTL8029 PCI card did not have this.

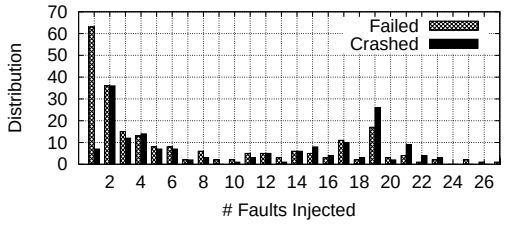


**Figure 9:** Distribution of number of faults needed to induce a failure or a crash. The long right tail for crashes is not shown. Data based on over 12,500 randomly injected common faults.

### 7.3 Reengineering Effort

An important lesson that we have learned during our prototype implementation is that the recovery procedure requires an integrated approach for optimal results, meaning that certain components need to be recovery-aware. As a metric for the reengineering effort we counted the number of lines of executable code (LoC) that is needed to support recovery. Blank lines, comments, and definitions in header files do not add to the code complexity, so these were omitted in the counting process. Line counting was done using the *sclc.pl* Perl script [1]. Fig. 10 summarizes the results.

Fortunately, the changes required to deal with driver failures are both very limited and local. The reincarnation server and *service* utility's logic to dynamically start servers and drivers is reused to support recovery. Most of the new code relates to defect detection and execution of the recovery scripts. The virtual file system and file server mostly stay the same, with changes centralized in the device I/O routines. Furthermore, the recovery code in the network server represents a minimal extension to the code needed to start a new driver. Finally, the process manager and microkernel are not affected at all.

Most importantly, the device drivers in our system are hardly affected. In general, only a minimal change to reply to heartbeat requests from the reincarnation server is needed. For most drivers this change comprises exactly 3 lines of code in the shared driver library to handle the new heartbeat request type and send the corresponding reply. Device-specific driver code does not need to be changed in general. For a few drivers, however, the code to initialize the hardware had to be modified in order to support reinitialization. The changes required to enable device driver recovery thus are negligible compared to the total amount of driver code that potentially can be guarded by our design.

| Component | Total LoC | Recovery LoC | % |
|---|---|---|---|
| Reinc. Server | 2,002 | 593 | 30% |
| Data Store | 384 | 59 | 15% |
| VFS Server | 5,464 | 274 | 5% |
| File Server | 3,356 | 22 | <1% |
| RAM Disk Driver | 454 | 0 | 0% |
| Network Server | 20,019 | 124 | <1% |
| RTL8029 Driver | 2,769 | 3 | <1% |
| Process Manager | 2,954 | 0 | 0% |
| Microkernel | 4,832 | 0 | 0% |
| Total | 39,011 | 1,072 | - |

**Figure 10:** Statistics on the total code base and reengineering effort specific to recovery expressed in lines of executable code (LoC). Most components are written purely in C, except for the microkernel, which includes some assembly code.

### 7.4 General Applicability

The ideas put forward in this paper are generally applicable, and may, for example, be used in commodity operating systems. While the degree of isolation provided by our prototype platform, MINIX 3, enabled us to implement and test our ideas with relatively little effort, we believe that other systems can also benefit from our ideas. Especially, since there is a trend towards isolation of untrusted extensions on other operating systems. For example, user-mode drivers have been successfully tested on Linux [25] and adopted by Windows [27]. If the drivers are properly isolated, these systems can build on the principles presented here in order to provide policy-driven recovery services like we do.

## 8 CONCLUSION

Our research explores whether it is possible to build a dependable operating system out of unreliable, buggy components. We have been inspired by failure-resilient designs that mask failures so that the system can continue as though no errors had occurred, and attempted to extend this idea to device driver failures. Recovery from such failures is particularly important, since device drivers form a large fraction of the operating system code and tend to be buggy.

In this paper, we presented an operating system architecture in which common failures in drivers and other critical extensions can be transparently repaired. The system is constantly monitored by the reincarnation server and malfunctioning components may be replaced in a policy-driven recovery procedure to masks failures for both users and applications. We illustrated our ideas with concrete recovery schemes for failures in network, block device, and character device drivers.

We also evaluated our design in various ways. We measured the performance overhead due to our recovery mechanisms, which can be as low as 1%. Fault-injection experiments proved that our design can handle realistic failures and provide continuous operation in 99.9% of recovery attempts. Source code analysis showed that the reengineering effort needed is both limited and local. All in all, we believe that our work on failure resilience for device drivers represents a small step towards more dependable operating systems.

## 9 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their suggestions that improved this paper. This work was supported by Netherlands Organization for Scientific Research (NWO) under grant 612-060-420.

# REFERENCES


[1] B. Appleton. Source Code Line Counter (sclc.pl), Last Modified: April 2003. Available Online.

[2] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th Symp. on Oper. Syst. Prin.*, pages 164–177, 2003.

[3] V. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *Comm. of the ACM*, 21(1):42–52, Jan. 1984.

[4] H. Bos and B. Samwel. Safe Kernel Programming in the OKE. In *Proc. of the 5th IEEE Conf. on Open Architectures and Network Programming*, pages 141–152, June 2002.

[5] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin*, 1994.

[6] G. Candea, J. Cutler, and A. Fox. Improving Availability with Recursive Microreboots: A Soft-State System Case Study. *Performance Evaluation Journal*, 56(1):213–248, .

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–A Technique for Cheap Recovery. In *Proc. 6th Symp. on Oper. Syst. Design and Impl.*, pages 31–44, .

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proc. 18th Symp. on Oper. Syst. Prin.*, pages 73–88, 2001.

[9] T. C. K. Chou. Beyond Fault Tolerance. *IEEE Computer*, 30(4), Apr. 1997.

[10] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults – Based on Field Data. In *Proc. 26th IEEE Int'l Symp. on Fault Tolerant Computing*, June 1996.

[11] A. Forin, D. Golub, and B. Bershad. An I/O System for Mach 3.0. In *Proc. 2nd USENIX Mach Symp.*, pages 163–176, 1991.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proc. 1st Workshop on Oper. Sys. and Arch. Support for the On-Demand IT InfraStructure*, Oct. 2004.

[13] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. 9th ACM SIGOPS European Workshop*, pages 109–114, Sept. 2000.

[14] J. Gray. Why Do Computers Stop and What Can Be Done About It? In *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[15] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of $\mu$-Kernel-Based Systems. In *Proc. 6th Symp. on Oper. Syst. Prin.*, pages 66–77, Oct. 1997.

[16] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O Architecture for Microkernel-Based Operating Systems, July 2003. Technical Report. TU Dresden.

[17] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proc. 1st Conf. on Collaborative Computing*, Dec. 2005.

[18] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Reorganizing UNIX for Reliability. In *Proc. 11th Asia-Pacific Comp. Sys. Arch. Conf.*, Sept. 2006.

[19] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. 6th European Dependable Computing Conf.*, Oct. 2006.

[20] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *ACM SIGOPS Operating System Review*, 40(3), July 2006.

[21] D. Hildebrand. An Architectural Overview of QNX. In *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Apr. 1992.

[22] M. Hohmuth and H. Tews. The VFiasco Approach for a Verified Operating System. In *Proc. 2nd ECOOP Workshop on Prog. Lang. and Oper. Sys.*, July 2005.

[23] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber. Sealing OS Processes to Improve Dependability and Safety. In *Proc. 2nd EuroSys Conf.*, 2007.

[24] J. Liedtke. On $\mu$-Kernel Construction. In *Proc. 15th Symp. on Oper. Syst. Prin.*, pages 237–250, Dec. 1995.

[25] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[26] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. 6th Symp. on Oper. Syst. Design and Impl.*, pages 17–30, Dec. 2004.

[27] Microsoft Corporation. Architecture of the User-Mode Driver Framework. In *Proc. 15th WinHEC Conf.*, May 2006.

[28] R. Moraes, R. Barbosa, J. Dures, N. Mendes, E. Martins, and H. Madeira. Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent? In *Proc. 6th Eur. Dependable Computing Conf.*, pages 53–64, Oct. 2006.

[29] B. Murphy and N. Davies. System Reliability and Availability Drivers of Tru64 UNIX. In *Proc. 29th Int'l Symp. on Fault-Tolerant Computing*, June 1999. Tutorial.

[30] V. P. Nelson. Fault-Tolerant Computing: Fundamental Concepts. *IEEE Computer*, 23(7):19–25, July 1990.

[31] W. T. Ng and P. M. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proc. 29th Int'l Symp. on Fault-Tolerant Computing*, pages 76–83, June 1999.

[32] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proc. 20th Symp. on Oper. Sys. Prin.*, pages 206–220, Oct. 2005.

[33] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proc. of the IEEE*, 63(9), Sept. 1975.

[34] L. Seawright and R. MacKinnon. VM/370—A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[35] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. USENIX Ann. Tech. Conf.*, pages 1–14, 2001.

[36] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability – A Study of Field Failures in Operating Systems. In *Proc. 21st Int'l Symp. on Fault-Tolerant Computing*, June 1991.

[37] Sun Microsystems. Predictive Self-Healing in the Solaris 10 Operating System, June 2004. Available Online.

[38] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering Device Drivers. In *Proc. 6th Symp. on Oper. Syst. Design and Impl.*, pages 1–15, Dec. 2004.

[39] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Trans. on Comp. Syst.*, 23 (1):77–110, 2005.

[40] T.J. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *Proc. Symp. on Software Testing and Analysis*, pages 55–64, July 2002.

[41] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proc. 6th Pacific Rim Symp. on Dependable Computing*, pages 178–185, Dec. 1999.

[42] A. R. Yumerefendi and J. S. Chase. The Role of Accountability in Dependable Distributed Systems. In *Proc. 1st Workshop on Hot Topics in System Dependability*, June 2005.

[43] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proc. 7th Symp. on Oper. Sys. Design and Impl.*, pages 45–60, Nov. 2006.