JORRIT N. HERDER, HERBERT BOS, BEN GRAS, PHILIP HOMBURG, AND ANDREW S. TANENBAUM

roadmap to a failure-resilient operating system

Jorrit Herder holds an M.Sc. degree in Computer Science (cum laude) from the Vrije Universiteit in Amsterdam and is currently a Ph.D. student there. His research focuses on operating system reliability and security, and he is closely involved in the design and implementation of MINIX 3.

jnherder@cs.vu.nl

Herbert Bos obtained his M.Sc. from the University of Twente in the Netherlands and his Ph.D. from the Cambridge University Computer Laboratory (UK). He is currently an assistant professor at the Vrije Universiteit in Amsterdam with a keen research interest in operating systems, high-speed networks, and security.

herbertb@cs.vu.nl

Ben Gras has an M.Sc. in computer science from the Vrije Universiteit in Amsterdam and has previously worked as sysadmin and programmer. He is now employed by the VU in the Computer Systems Section as a programmer working on the MINIX 3 project.

beng@cs.vu.nl

Philip Homburg received a Ph.D. from the Vrije Universiteit in the field of wide-area distributed systems. Before joining this project, he experimented with virtual memory, networking, and X Windows in Minix-vmd and worked on advanced file systems in the Logical Disk project.

philip@cs.vu.nl

Andrew S. Tanenbaum is a professor of computer science at the Vrije Universiteit in Amsterdam. He has written 16 books and 125 papers and is a Fellow of both the ACM and the IEEE. He firmly believes that we need to radically change the structure of operating systems to make them more reliable and secure and that MINIX 3 is a small step in this direction.

ast@cs.vu.nl

IN RECENT YEARS, DEPENDABILITY AND security have become prime concerns for computer users. Nevertheless, commodity operating systems, such as Windows and Linux, fail to deliver a dependable and secure computing platform. The lack of proper fault isolation in the monolithic kernel of commodity systems means that a local failure can easily spread and corrupt other components. A single bug, say, a buffer overrun in a network driver, can overwrite crucial data structures, causing a subsequent, but unrelated, action to trigger a fatal exception. Recovery is usually not possible except by rebooting the computer.

While software is buggy by nature, device drivers are known to be especially failure-prone [1, 2]. It is irrelevant whether the failures are due to hardware glitches, improper device documentation, the arcane kernel programming environment, lack of quality control, limited testing, or code immaturity. The crucial point is that pieces of untrusted, third-party code, such as drivers and other extensions, run inside the kernel and can potentially take down the entire system. This property is inherent to the monolithic design used in commodity operating systems, and it cannot be solved through mere programming effort.

Our approach to dependability is to cope with imperfection and counter the more fundamental problem that driver failures threaten to take down the entire operating system. In particular, we have enhanced the MINIX 3 operating system with fault-resilience techniques to improve operating system dependability. We accept the fact that software is not perfect and probably never will be, and anticipate failures in device drivers and other critical operating system components. Our system is designed to withstand such failures and can often repair itself in a manner that is transparent to applications and without user intervention.

MINIX 3 has been under development for the past two years and is becoming increasingly mature. We have already reported on MINIX 3's multiserver architecture [3] and mechanisms to deal with dead device drivers [4]. In this article we loosely summarize what we have done to make MINIX 3 resilient against failures, where we stand now, and what is left for future work. An overview of the

highlights of MINIX 3's development and a tentative roadmap for future work are given in Figure 1. As the figure shows, we hope to release a thoroughly tested fault-resilient version of MINIX 3 early next year.

The remainder of this article is organized as follows. We start out with a short introduction to the recent history of MINIX 3. Then we give an overview of the fault-resilience mechanisms we have implemented thus far and perform a brief reality check. In the end, we discuss our current and future work that will eventually lead to the release of a fault-resilient version of MINIX 3 and then conclude.

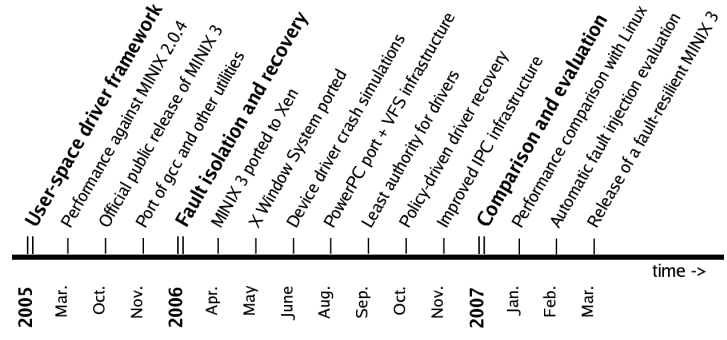

FIGURE 1: OVERVIEW OF THE HIGHLIGHTS OF MINIX 3'S DEVELOPMENT AND TENTATIVE ROADMAP FOR FUTURE WORK.

The Recent History of MINIX 3

As a base for MINIX 3 we used MINIX 2, which already ran some servers in user space but still had in-kernel device drivers. Starting in late 2003, we removed the drivers from the kernel, developed a user-space device driver framework, and officially released MINIX 3 in October 2005. Since then, the system has been downloaded over 100,000 times and a small but growing user community has formed to support MINIX 3. The official Web site (www.minix3.org) and newsgroup (comp.os.minix) are frequented by many enthusiasts who want to participate in our quest for a secure and dependable operating system.

The architecture of MINIX 3 is shown in Figure 2. All servers and drivers run as independent user-mode processes—each encapsulated in a private address space protected by the MMU hardware—on top of a tiny microkernel of under 4000 lines of executable code. The bottom half of the microkernel is responsible for programming the CPU and MMU, interrupt handling, and IPC. The in-kernel clock and system task provide an interface to kernel services, such as I/O and alarms, for the user-mode parts of the operating system. The most common servers provide file system services and process management functionality. A special server, called the *reincarnation server*, manages all servers and drivers and constantly monitors the system's well-being. With this design as a stable base we were able to achieve failure resilience, as we discuss below.

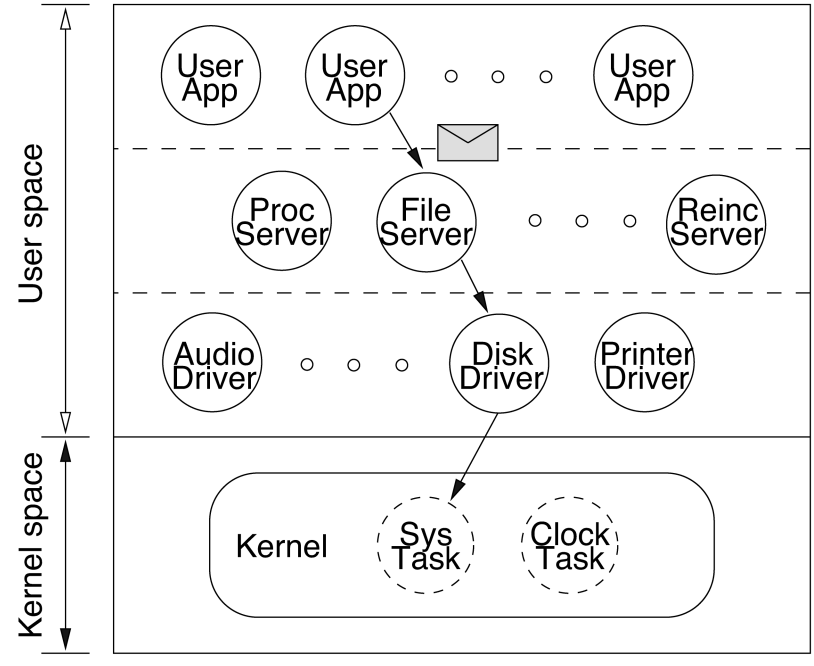

FIGURE 2: ARCHITECTURE OF MINIX 3. THE OPERATING SYSTEM IS COMPARTMENTALIZED IN USER SPACE, BUT COMPONENTS CAN INTERACT WITH EACH OTHER BY PASSING MESSAGES. THE POSIX READ() CALL, FOR EXAMPLE, IS TRANSFORMED INTO A REQUEST TO THE FILE SERVER, WHICH ASKS THE DISK DRIVER TO READ THE BLOCK FROM DISK, WHICH, IN TURN, ASKS THE IN-KERNEL SYSTEM TASK TO PERFORM PRIVILEGED OPERATIONS, SUCH AS WRITING TO THE DISK'S I/O PORTS.

MINIX 3 currently runs over 400 standard UNIX applications, including the X Window system, two C compilers, language processors, several shells, many editors, a complete TCP/IP stack that supports BSD sockets, a virtual file system infrastructure, and all the standard shell, file, text manipulation, and other UNIX utilities. The POSIX-compliant interface offered by MINIX 3 facilitates porting of common Linux and BSD applications. For example, porting an application to our system is often simply a matter of recompilation.

Performance measurements on a 2.2-GHz Athlon show that the overhead of our system is 5–10% compared to the base system (MINIX 2) with in-kernel device drivers [3]. User-mode Fast Ethernet runs at full speed, and our user-mode disk drivers show an average overhead of about 8% to perform disk I/O compared to in-kernel disk drivers. Since neither MINIX 2 nor MINIX 3 has been tuned for performance, we expect to find a somewhat higher overhead when the system is compared to Linux or FreeBSD. Nevertheless, MINIX 3 feels fast and responsive. For example, the boot time, as measured between exiting the multiboot monitor and getting the login prompt, is less than 5 seconds. At that point, a POSIX-compliant operating system is ready to use.

Crash simulation experiments show that our system can withstand failures and gracefully recover by restarting the driver rather than rebooting the entire computer [3]. For example, in one experiment, we used wget to retrieve a 512-MB file from the Internet while repeatedly killing the Ethernet driver every 4 seconds. The network transfer successfully completed in all cases, with a performance degradation of just 8%. Although these experiments prove the viability of our approach, manually killing a

driver to simulate a crash is not representative for many device driver failures. Therefore, we recently started to experiment with automatic fault injection, with promising results, as discussed next.

Achieving Fault Resilience

The key principles we used to make MINIX 3 failure resilient are fault isolation, defect detection, and run-time recovery. Fault isolation is required to prevent problems from spreading and limit the damage bugs can do. When a bug is properly caged it becomes easier to pinpoint the defect, and recovery may be possible. In the following we briefly discuss how we realized each principle in MINIX 3. As an aside, this model may have consequences for the accountability of software vendors [5], but in this article, we focus on the technical aspects of our design.

FAULT ISOLATION

Although we have fully compartmentalized the operating system in user space, as illustrated in Figure 2, isolation cannot be achieved by means of address-space separation alone. This is because servers and drivers need potentially dangerous mechanisms to communicate and share data in order to make the system work. Instead of granting such powers to all processes, we have carefully reduced the privileges of each according to the Principle Of Least Authority (POLA). Each device driver, for example, is loaded with a protection file that precisely lists its resources, including device memory, I/O ports and IRQ lines, and IPC capabilities. The reincarnation server ensures that the restriction policy is in place before the newly started driver gets to run.

Memory protection is realized by combining MMU and kernel protection. The MMU ensures that a process cannot directly access another process's memory. However, to prevent memory corruption in processes that need to share data, processes can grant access to precisely specified memory areas by sending a capability that is checked by the kernel when data is read or written. It has to be noted that DMA is still a potential danger, but this is a hardware problem and not a limitation of our system. Fortunately, I/O MMUs are becoming more common, and when we have the proper hardware we will solidify our defenses.

DEFECT DETECTION

The reincarnation server is the central component that guards all servers and drivers in the system. During system initialization the reincarnation server adopts all processes in the boot image as its children; servers and drivers that are started on the fly also become its children. Therefore, in line with the POSIX model, the reincarnation server will be notified by the process manager when a system process exits. Based on the exit status retrieved from the process manager, three cases can be distinguished: a process exit or panic, a CPU or MMU exception, or a user signal. Each of these cases is considered as a separate defect class.

In addition, the reincarnation server has three other ways to monitor the system for anomalies. When a driver's protection file specifies so, the reincarnation server periodically pings the driver and expects it to reply with a heartbeat message. Not responding is considered a defect and initiates the

recovery procedure. Furthermore, the reincarnation server acts as an arbiter in case of problems. For example, the network server can request replacement of an Ethernet driver that does not adhere to the multiserver protocol. Finally, the user can instruct the reincarnation server to dynamically update the system. In this way, when a bug or other vulnerability is found, the defective component can be replaced on the fly as soon as a patch is available.

RECOVERY PROCEDURE

When a server or driver is started, it can be associated with a (generic) shell script that governs its recovery procedure. When a defect has been detected, the reincarnation server looks up the malfunctioning process's recovery script from its internal tables and runs it. All relevant parameters, such as the component that failed, defect class, and failure count, are passed along so that the script can decide what to do. The simplest policy may log the error and shut down the malfunctioning component, but in many cases it is possible to replace it with a fresh copy. A sample policy script that uses a binary exponential backoff protocol in restarting failed components is shown in Figure 3.

```
component=$1            # args from reinc. server
reason=$2               # dynamic update = 6
repetition=$3           # current failure count

if [ ! $reason -eq 6 ]
then
    sleep $((1 << ($repetition - 1)))
fi
service restart $component
```

FIGURE 3: RECOVERY SCRIPT THAT USES A BINARY EXPONENTIAL BACKOFF PROTOCOL IN RESTARTING A FAILED COMPONENT TO PREVENT BOGGING DOWN THE SYSTEM IN CASE OF REPEATED FAILURES, UNLESS THE USER EXPLICITLY REQUESTED A DYNAMIC UPDATE [4].

Once a component has been restarted it needs to be reintegrated into the system. First, the reincarnation server updates the corresponding name server entry in the *data store*, which uses a publish/subscribe mechanism to inform dependent components about the new system configuration. For example, the file server will be notified when a disk driver is restarted and its new IPC endpoint is published in the data store. At this point, the file server can reinitialize its own tables and can request the driver to reinitialize itself. If the restarted component lost state during its crash, it can, in principle, retrieve a backup made by the crashed component from the data store. In our current prototype implementation, however, all drivers are stateless or can be reinitialized from the server level. Recovery of stateful components is not used by our prototype implementation, but the mechanisms required to do so are in place.

Reality Check

Although our system has been designed to recover from failures in both servers and drivers, there are limits to what we can do. Since the core operating system servers maintain a lot of state, recovery is currently not supported. For example, the process server keeps track of process IDs, child-parent relationships, alarms, and more. Although a crash does not take down the entire system, all user programs will be seriously hampered. Nevertheless, our approach deals with an important class of problems, since 70% of the operating system typically consists of driver code, with reported error rates 3–7 times higher than those of ordinary code [2].

The assumption underlying our recovery procedure is that failures are transient and can be repaired by replacing malfunctioning components. For example, rare timing causing an exception, software aging from memory leaks, and the like may bring down a component, but in many cases a restart will cure the problem. Moreover, our design not only helps when disaster strikes but also opens the possibility for ante-mortem updates. At any point in time the user can update the system by requesting the reincarnation server to replace a component under suspicion with a new one. This feature helps system administrators to keep the system in good shape without system downtime. It may also be useful in embedded systems that need to automatically replace components when new versions are available.

Work for the Near Future

The general fault-resilience mechanisms presented here are currently implemented in MINIX 3, but more work needs to be done, as shown in Figure 1. In particular, a better performance assessment and a more thorough evaluation of MINIX 3's ability to recover from failures are needed. We have already studied the performance of MINIX 3 compared to MINIX 2 and have concluded that the transformation of in-kernel drivers into user-space drivers resulted in a performance overhead of about 5–10%. We are currently investigating how MINIX 3 compares to other UNIX-like operating systems such as Linux and FreeBSD. Preliminary results show that the overhead is somewhat higher, but we do not have the precise numbers yet. However, because MINIX 3 is not optimized for performance—in contrast to the other systems—it will be hard to tell to what extent the overhead is due to MINIX 3's multiserver design or to the differences in, for example, compiler quality, memory management algorithms, and file system implementation.

In addition, we are working on a better evaluation of MINIX 3's ability to survive failures in critical operating system components and transparently repair the system. Our current focus has been to reincarnate dead device drivers, but recovery from failures in stateful components has our interest as well. Furthermore, we have mostly tested the system's failure resilience by manually killing components, but this approach is not representative for failures that are caused by, say, programming bugs. Therefore, we recently ported the fault injection tool used by Nooks [6] to MINIX 3, which allows us to inject more representative faults by mutating the driver binaries. This method already proved its value, as we discovered a small number of bugs in the core components, which we fixed. More importantly, the results thus far indicate that our system is indeed capable of surviving and recovering from common failures.

Summary and Conclusion

In this article, we briefly described the recent history of our work on MINIX 3 and we showed how the modular design of MINIX 3 can be exploited to achieve failure resilience within the operating system. A timeline with the highlights of MINIX 3 thus far and a tentative roadmap for future work was presented. Although more development and testing is needed, the principles discussed here show that it is possible to improve operating system dependability by revisiting design choices that were made decades ago. All in all, we believe that MINIX 3 has serious potential to claim a niche in the operating system market—for example, on moderately powerful embedded systems where security and dependability are at stake, such as mobile phones, set-top boxes, and medical appliances.

REFERENCES

[1] T.J. Ostrand and E.J. Weyuker, "The Distribution of Faults in a Large Industrial Software System," *Proc. 2002 ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, ACM, pp. 55–64, 2002.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating System Errors," *Proc. 18th ACM Symp. on Operating System Principles*, pp. 73–88, 2001.

[3] J.N. Herder, H. Bos, B. Gras, P. Homburg, and A.S. Tanenbaum, "Reorganizing UNIX for Reliability," *Proc. 11th Asia-Pacific Computer Systems Architecture Conference*, pp. 81–94, 2006.

[4] J.N. Herder, H. Bos, B. Gras, P. Homburg, and A.S. Tanenbaum, "Who's Afraid of Dead Device Drivers," Technical Report IR-CS-D29, Vrije Universiteit, Amsterdam, 2006.

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

[6] M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy, "Recovering Device Drivers," *Proc. 6th Symp. on Operating System Design and Implementation*, pp. 1–15, 2004.