Self-stabilizing Device Drivers
(Extended Abstract)

Shlomi Dolev\(^1\) and Reuven Yagel\(^{1,2}\)

\(^1\) Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
\{dolev, yagel\}@cs.bgu.ac.il

\(^2\) Rafael Ltd. 3M, POB 2205, Haifa 31021, Israel

Abstract. This work presents approaches for designing the input-output device management components of self-stabilizing operating systems. As an example, we demonstrate the non-stability of the ATA standard protocol for storage devices. We state the requirements that an operating system and I/O devices should satisfy in order to become self-stabilizing. Then we suggest two solutions to satisfy these requirements. The first uses leases in order to guarantee progress from the I/O device side. The second assumes stabilization of the I/O device, and uses snapshots to perform consistency checks. By supplying an infrastructure for practical self-stabilizing systems, robust and dependable systems can be achieved.

Keywords: self-stabilizing systems, device driver failures, ATA interface standard.

1 Introduction

Device drivers are known to be a major cause of operating system failures [6,25,27]. This phenomena is often connected to a combination of reasons. First, drivers are usually loaded into the operating system kernel’s address space and are running in privileged processor modes where an error has a greater effect on the total system behavior. Additionally, usually essential system parts are designed, built, verified and tested with extra care while drivers are many times brought from the outside. The following are techniques which are used to deal with these failures: (a) reducing the driver’s access to system’s resources [34,13], (b) containment of errors in realtime through kinds of virtualization [27,2], (c) using typed languages [13,28], and (d) static analysis of the drivers’ code, and of their resource usage [130]. Applying such techniques helps improve the system’s robustness, but the bottom line is that in systems running for a long period of time, errors (e.g., soft errors [17]) in device drivers accumulate and lead to undesired behavior.

\(^*\) Partially supported by Rafael, Israeli Ministry of Science, Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences and Lynn and William Frankel Center for Computer Sciences.
Device drivers are programs which are practically an essential part of any operating system. They serve as an adaptation layer by managing the various operation and communication details of I/O devices. They also serve as a translation layer providing consistent and more abstract interface between other programs and the hardware device resources (and sometimes they also add extra services not provided by the hardware devices). Devices usually contain a controller which is the electronic part with which drivers communicate. The communication is carried out via the system bus, and is usually done through some standard protocols and interfaces e.g., ATA and SCSI for disk drives. In [23] it is stated that “a modern Seagate drive contains roughly 400,000 lines of code”. In [34] it is noted that “modern disk controllers often have many megabytes of memory inside the controller”. The complexity of today’s I/O devices emphasize the need for robust device drivers.

In this work we suggest enhancing the robustness of device-drivers by designing them to be self-stabilizing. Generally, a system is self-stabilizing [8,9] if it can be started in any possible state and subsequently it converges to a desired behavior. A state of a system is an assignment of arbitrary values to the system’s variables. Building a system, and specifically device drivers, in this way, ensures that errors will be contained autonomously by each driver, leading eventually to correct behavior of the whole system.

Self-stabilizing Operating System (SOS). In order to have a full self-stabilizing system, the other system parts must also be self-stabilizing. This work uses building blocks from our previous work [10] where simple self-stabilizing process schedulers are presented, and from [11] where various memory management schemes are suggested. We also rely on [7], which addresses self-stabilization of the microprocessor. Thus, based on the idea of fair composition [9], once the microprocessor stabilizes and starts fetching and executing instructions, the system’s kernel converges to a legal behavior, in which other programs are executed infinitely often to fulfill the system’s goal.

Related work. Extensive theoretical research has been done towards self-stabilizing systems [8,9,31] and recovery-oriented/autonomic-computing/self-repair, e.g., [15,24,32]. Fault tolerance properties and robustness of operating systems (e.g., [27,22]) were also extensively studied.

As explained earlier, robustness of device drivers is of great importance in system design. It is stated in [6] and [21] that about 70% percent of operating system code is devoted to device drivers. It is stated in [25] that “In Windows XP, for example, device drivers cause 85% of reported failures”. Moreover, in [6] it is claimed that for some cases in Linux “the error rate for drivers is almost seven times higher than the error rate for the rest of the kernel”. Here we shortly survey various efforts in this field.

Device driver isolation and monitoring. The micro-kernel system architecture (pioneered in the Mach system [1]), suggests achieving minimal trusted computing base (TCB) by removing as much as possible from the kernel. For example,