It Wasn't My Fault: Understanding OS Fault Propagation Via Delta Execution

Cristiano Giuffrida[*]
Department of Computer Science
Vrije Universiteit, Amsterdam
giuffrida@cs.vu.nl

Lorenzo Cavallaro
Department of Computer Science
Vrije Universiteit, Amsterdam
sullivan@cs.vu.nl

Andrew S. Tanenbaum
Department of Computer Science
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Recent approaches to operating system (OS) crash recovery have attempted to design a high-coverage component-agnostic recovery infrastructure [1, 3]. To successfully recover from an otherwise-fatal crash, it is necessary to determine a safe execution point to resume operation consistently.

Common restart strategies attempt to bring back the OS from a faulty execution state to a safe state and selectively replay execution. Unfortunately, a faulty execution leading to a crash can result in many logical inconsistencies with complex dependencies among execution contexts (e.g. kernel threads) and OS subsystems that need to be tracked and rollbacked if necessary. Both heavyweight [3] and lightweight mechanisms [1] have been recently proposed for this purpose. Nevertheless, conclusions cannot be easily drawn on the effectiveness of these strategies, since very little is known about the way faults logically propagate throughout different OS subsystems and execution contexts in the general case.

To investigate the properties of faulty execution and analyze the effectiveness of recovery strategies in different fault scenarios, we propose fault injection experiments in a controlled environment. Prior fault injection work has mainly focused on analyzing the distance and the time it takes for a crash to occur once a fault is injected in a particular OS subsystem [2]. This is insufficient to analyze the behavior of the system during faulty execution in a fine-grained way and determine the ability to recover in a given fault scenario. In addition, in [2] individual subsystems were not properly isolated and there is no way to easily distinguish crashes caused by global memory corruption from crashes caused by logical inconsistencies among different OS subsystems.

Our approach, in contrast, is tailored to monitoring the faulty execution in realtime and analyze the logical propagation of different fault types throughout different subsystems and execution contexts. To this end, we break down the operating system into separate user-space components with well-defined boundaries and communication primitives (IPC calls based on message passing). When a fault is to be artificially injected in a component, we halt the execution of the component, fork a new replica, and inject the desired fault in the replica. From this moment on, the two replicas run in parallel and both interact with the environment using only IPC messages. The execution of both replicas, however, is confined in a controlled environment.

The first replica runs normally, but the incoming and outgoing IPC traffic is constantly monitored and synchronized with the faulty replica when necessary. The faulty replica, in contrast, runs in an emulated environment, in which all the outgoing IPC traffic is intercepted and incoming IPC messages are artificially crafted and injected. An IPC interceptor takes care of synchronizing the execution of the two replicas, by replicating the interactions with the environment of the first replica in the emulated environment of the faulty replica. At the same time, the faulty replica is continuously monitored to detect crashes and differences in the execution. To analyze the delta execution of the faulty replica, we use IPC messages as synchronization points and record state differences between the two replicas at each synchronization point. The evolution of the differences can give insights on how hard it is to recover from state inconsistencies or corruption at crash recovery time. At the same time, we monitor all the differences in the IPC traffic to detect and analyze any cross-subsystem fault propagation. We are also interested in classifying all the IPC interactions occurred in the delta execution to determine the ability to successfully recover. For example, idempotent IPC interactions during delta execution will not probably cause serious global inconsistencies. In other cases, the faulty execution may propagate throughout several subsystems or even reprogram the hardware, hampering the ability to recover transparently or even making recovery infeasible.

We expect our experimental analysis conducted on many fault scenarios to give important insights on the ability to recover from OS crashes and improve the dependability of modern operating systems. We are planning to conduct our analysis on a multiserver microkernel operating system to simplify the implementation, but we will also investigate how to replicate similar experiments on commodity operating systems. For example, recent work [3] has demonstrated the possibility to instrument commodity operating systems and transparently split the execution space into separate domains. We believe cross-domain communication could also be instrumented and intercepted to design a fault injection testbed similar to the one described here.

References

[1] Giuffrida, C., Cavallaro, L., and Tanenbaum, A. S. We Crashed, Now What? In *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HotDep'10)* (Oct 2010).

[2] Gu, W., Kalbarczyk, Z., Iyer, R. K., and Yang, Z. Characterization of Linux Kernel Behavior under Errors. In *DSN* (Los Alamitos, CA, USA, 2003), IEEE Computer Society.

[3] Lenharth, A., Adve, V. S., and King, S. T. Recovery Domains: an Organizing Principle for Recoverable Operating Systems. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), ACM.

[*] PhD student at Vrije Universiteit, Amsterdam