Chapter 1

Introduction

1.1 Orthogonal persistence

Orthogonal persistence is a concept first introduced by Atkinson et al. in [1].

Atkinson in [2] proposes to use persistence support systems to create orthogonally persistent environments. Applications that work in such environments are called orthogonally persistent. Generally their development is easier for two reasons. First, programmers do not need to write boilerplate code for serialization and deserialization of data that they want to save into a persistent storage. Since these applications have a smaller codebase, they tend to be less error-prone: smaller codebase offers less space for mistakes.

Second, programmers of applications in persistent environments do not need to care about data loss in case of unexpected shutdown. For an application to be orthogonally persistent, it should have certain properties. The most important property is restart tolerance. That means that the application should not lose its state after the system is restarted. This property should hold for

an arbitrarily long time between system shutdown and turning on. The exact definition of orthogonal persistence can be found below in section **??**.

Up to the date, restoration of the internal state of a computer was actively researched [3]–[6]. This problem was successfully solved in several persistent operating systems, including EROS, Grasshopper OS and Phantom OS. These operating systems allow the development of an application that continues execution after restarts without data loss. However, these systems do not guarantee that a given operation will be performed exactly once. Sometimes unexpected power loss can lead to results of a computation being lost. In this case, the system restores its state to an older position in time. That leads to performing some operations several times.

1.2 Handling I/O in persistent operating systems

Any operating system has objects that do not belong to its internal state but instead represent some external resource. These objects are usually related to the system's input/output(I/O) interface. An example of such object can be an Unix file descriptor representing connection to some certain external device, for example a robotic arm.

Suppose the arm is initially at state S_a and the persistent application is aware of that. Now assume that the application issues a blocking request R for transition of arm to new state S_b and before this transition finishes an unexpected power loss occurs. In this case when the system goes back up the persistent application will issue the request R again. But then the arm device will transition to the state S_c. This can cause problems, because the application

does not expect this state. The same will happen even in case of absence of the pending request if something would interact with the robotic arm while the system is offline.

This simple example illustrates the need for special device drivers that will be aware of the transient nature of the environment and will set up I/O devices in accordance with expectations of persistent applications.

Network communication is a type of I/O operation. That means that everything said above about problems with objects representing I/O operations in general applicable to objects representing network operations – sockets.

1.3 Phantom OS

Phantom OS does not offer persistence of sockets for its applications. After each restart socket object becomes invalid - any operation on it results in error. The application that uses a socket is responsible for handling those errors and reopening a new socket, if it needs it. This behavior is not in accordance with the idea of orthogonal persistence. To achieve it, system restart should be transparent for an application, that is application should not know about any socket errors caused by restart.

Until recently, it was infeasible to develop a solution for persisting sockets in Phantom OS. This would require modifications to Phantom kernel, which is not that easy. However, recent work on porting Phantom OS to Genode Framework made it possible.

In this port the Genode part is transient. It replaces the Phantom kernel and handles all the syscalls from the Phantom Virtual Machine. Due to the modular nature of Genode, well-written documentation, and flexibility of the

framework one can easily change implementation of network-related syscalls. With these changes persistent applications running in PVM would not know about restart. This effectively provides a higher degree of persistence into Phantom OS.

The goal of my work is to make a mechanism that will allow managing network state in the Genode part of the port.

Bibliography cited

[1] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison, "Ps-algol: A language for persistent programming," in *Proc. 10th Australian National Computer Conference, Melbourne, Australia*, 1983, pp. 70–79.

[2] M. Atkinson and R. Morrison, "Orthogonally persistent object systems," *The VLDB Journal*, vol. 4, no. 3, pp. 319–401, 1995.

[3] C. R. Landau, "The checkpoint mechanism in keykos," in *[1992] Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, IEEE, 1992, pp. 86–91.

[4] A. Dearle, R. Di Bona, J. Farrow, *et al.*, "Grasshopper: An orthogonally persistent operating system," *Computing Systems*, vol. 7, no. 3, pp. 289–312, 1994.

[5] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014, pp. 1–3.

[6] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 575–585, 2015.