Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR’S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Сетевые взаимодействия в ортогонально персистентных системах / Networking in orthogonally persistent operating systems**

Работу выполнил / Thesis is executed by	**Брисилин Антон Михайлович /Brisilin Anton Mihailovich**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Тормасов Александр Геннадьевич / Tormasov Alexander Gennadievich**	подпись / signature
Консультанты / Consultants		подпись / signature

Иннополис, Innopolis, 2021

Contents

List of Tables

List of Figures

Chapter 1

Literature Review

This chapter will begin with an overview of persistence concept in general, and in particular, orthogonal persistence in Section 1. Then, Section 2 will discuss how networking in persistent systems was implemented previously, and what are the requirements for a networking stack in such systems. The last section describes Phantom OS, Genode OS framework and current state of porting former to latter.

1.1 Persistence

1.1.1 Persistence definitions

Every program operates data. These data can be different in nature, and thus have different lifetimes. The lifetime of a data object is a time extent over which it can be used and is commonly called *persistence* [1].

Atkinson *et al.* [1], [2] give a classification of application data objects by their persistence, which is presented in Table I. Usually, support of levels 1-4 relies on programming languages themselves, and for levels 5-8 designers of persistent applications rely on some external component, such as database or file system.

Table I
Classification of data objects based on their lifetime

1.	Transient results in expression evaluation
2.	Local variables inside functions and code blocks
3.	Global variables and heap items
4.	Data that exists throughout a whole execution of a program
5.	Data that lasts for several executions of program
6.	Data that lasts for as long as a program is being used
7.	Data that outlives a program

1.1.2 Orthogonal persistence

Different ways of data interactions, namely using builtin programming language constructs and external tools available via some interfaces, bring an extra layer of complexity to programs. To answer this problem, [2] proposes to use persistent support systems, which act as a mediator between a persistent application and transient environment. Atkinson *et al.* also summarize requirements for such a system, saying that such a systems should have use of data not dependent of its persistence. They define following principles for such a system:

1. The Principle of Persistence Independence.

   Whether a program manipulates data that outlives it or not, the ways to use these data should be the same. There should be no significant difference in program syntax in either case.

2. The Principle of Data Type Orthogonality.

   Every data object should be allowed to have any level of persistence, irrespective of their type. There should be no special cases where objects of some type can not be persistent or transient.

3. The Principle of Persistence Identification.

   The choice of how to identify which objects are persistent and how To provide persistence to them objects is not related to the universe of discourse of the system. The mechanism for identifying persistent objects should not also be related to the type system.

Two popular approaches exist to build such a system.

One is to integrate interactions with databases or file systems into an existing programming languages, which is what most ORM libraries do [3], [4]. However, this approach is not really transparent for a programmer, and may cause errors.

Another approach is to build the so-called *persistent worlds* [2]. These worlds usually have form of operating systems, which provide required mechanisms to make state of userspace processes persistent across systems restarts. Examples of such systems are KeyKOS [5], Grasshopper OS [6], and Phantom OS.

A question may arise at this point, can an existent popular operating system, run with nonvolatile memory as a main memory, and if it will make it orthogonally persistent? The answer is no, because this approach raises two problems: problem of dealing with changed peripheral device states during restarts [7], and logical problems with code execution [8]. A persistent support system must satisfy orthogonal properties. Therefore building such a system is not trivial.

1.2 Networking in persistent systems

...This part is not done yet, because research on this is still in progress. However an outline was done for its contents

1. Definition of problems with currently supported network stack.

   Example of problems:

   - Lost connections and strategies to restore.
   - How stateful protocols can be changed?
   - Can we use some intermediary message broker to manage communication?

2. Define requirements for protocol that are suitable for PSs

3. Say what will be focus of work

1.3 Phantom OS and Genode OS framework

1.3.1 Phantom OS

Phantom OS is a general-purpose operating system, developed in 2009-2011. This is an orthogonally persistent operating system, in which persistence is brought using snapshotting of virtual memory. The operating system consists of kernel and virtual machine, which executes all userspace programs written in Phantom bytecode. Managed code execution is persistent, whereas kernel state is transient, which means that it is not kept between reboots. It is assumed that it can be restored. Phantom is an experimental system, in a sense that it is developed as a proof-of-concept and is not ready for production use.

Phantom OS currently supports only i386 ISA, and it has all drivers builtin inside a kernel. It is problematic, because to support wide range of hardware OS needs drivers, which is often delivered by third-party developers. But in case when drivers are embedded in kernel error in driver can crash whole system.

These are two reasons why it is desirable to replace Phantom kernel with a microkernel. My colleagues from Innopolis University are now working on porting Phantom OS to Genode OS framework. I am going to use this port as soon as it will be ready.

1.3.2 Genode OS framework

The Genode OS Framework [9] is a toolkit for creating operating systems. It provides possibility to build a microkernel operating systems from set of existing components. Genode provides an API to integrate various microkernels into it.

For now the supported ones include Linux kernel, nova microhypervisor and several kernels from L4 kernel family. Supported kernels from the last family include L4ka::Pistachio, Fiasco.OC, formally verified seL4 microkernel, and L4/Fiasco. The framework also supports execution on a bare hardware on ARM and x86-64 ISA.

Components provided by Genode usually fall into five categories: device drivers, resource multiplexers, protocol stacks, applications and runtime environments.

Chapter 2

Methodology

This chapter starts with the analysis of requirements for implemented software. After that, the implementation is discussed in more detail in the dedicated section.

2.1 Requirements analysis

As previous chapters of this paper suggest, the goal of this paper is to create an implementation of a TCP/IP stack suitable for a persistent system. In particular, I use a port of Phantom OS to Genode OS framework as a target platform.

This section contains a list of requirements that a developer of such a stack should take into account when implementing their own port.

2.1.1 Transparent peer disappearance

One of the fields in which persistent systems are actively applied is IoT. In an intermittently powered environment, a system that can efficiently restore its state is preferable to a system that experiences a booting process that consumes a lot of energy.

In such cases, hosts can experience long downtimes. Standard TCP implementations are not applicable here because they perform retransmission of packets for only a limited amount of time, and then close connection.

To make the development of stable persistent applications easier a temporary shutdown should be transparent for the client app. That means that the network connections state should be the same when the powered-off host restarts. To achieve that persistent TCP implementation should either (1) backup and restore connections state or (2) make the disappearance of the remote peer invisible for application on the other end of the connection.

2.1.2 Rollbacks

The technique used in Phantom OS to provide persistence is memory snapshots of PVM memory space. They are taken in a live mode so that processes inside the VM continue to run. Then, when power is back on, the latest snapshot is restored by copying it to the main memory. The part of Phantom that is outside of VM is assumed to be stateless in the usual case, which is not the case in Genode port. That is why snapshotting and restoring of Genode part of the system is done with callbacks, which accept and provide blobs of data to be saved in persistent memory.

In the case of the intermittently powered Phantom system, this can cause trouble, when clients actively use networking. While the system performs a snapshot, the TCP stack can still proceed with receiving data. In this case, TCP on receiving side will acknowledge packets, as protocol prescribes. The issue is that acknowledged packets are getting deleted from the retransmission queue, and therefore not present at the sending site. However, they are not present in the snapshot on the receiver's side. In this case, after the snapshot is used to restore the state of the receiver, the receiver will request the packet (with duplicated ACK) that the sender is unable to retransmit. In this case, the sender's TCP will respond with RST and then close the connection.

Three solutions exist that may solve the problem above. One is to forbid sending acknowledgment packets if a received packet is not included in any snapshot. However, this will either dramatically increase the number of retransmissions in a network, which will result in reduced network performance or it will require doing snapshots very often. In such a case snapshot should be performed each 50ms. With a snapshot duration of around 10 seconds, this is not acceptable.

The other solution is to keep a buffer in a persistent memory that will store all packets that a user of TCP had not consumed yet. In this case, persistent memory can be a regular file or memory space that will fall into a snapshot for sure.

The third approach is to have a dedicated broker inside a network that will function similarly to the buffer in the second case. The broker can save all acknowledged packets in a network, and flush its queue when it detects that receiving host has done a snapshot. ICMP messages or some similar mechanism will be useful to notify the broker about the end of the snapshot process.

2.1.3 Compatibility with existing protocols

The third design goal is not directly related to providing persistent I/O abstraction to the clients of a network stack. Since snapshotting is a very popular technique for implementing persistent systems, and Genode is actively used to develop various operating systems, it is probable that some other developers might want to use persistent TCP in their software.

That means that result of this work should be a standalone application that works without Phantom and is flexible enough to support all use cases that may arise. Fortunately, Genode's component-based architecture helps greatly with it.

Another thing that should be addressed in the design of the API of the system, is easy migration from non-persistent TCP stacks already available in Genode, namely lwip and lxip. To do this I intend to design API very similar to regular sockets or even exactly the same, but with different call semantics. I also plan to add a Virtual File System (VFS) plugin that will expose sockets as regular files because this is the way how Genode developers suggest integrating sockets into the Genode ecosystem since version 18.11. With such implementation, even relinking of existing binary will be not needed. The only thing user would need is to change the configuration of the init Genode component.

2.2 Implementation process

I have chosen to implement a persistent TCP stack in form of a Genode component that will serve as a plugin for a VFS. The users of the TCP stack should add signal handlers to their components for signals of type SIG_SNAP and SIG_RESTORE. When SIG_SNAP is received, it is passed to the persistent TCP VFS plugin which gathers data that should be saved into persistent memory and calls filesystem service to save socket snapshot structure into persistent memory.

Symmetrically, when SIG_RESTORE is received, the component will access the previously saved file and read information about sockets state at the snapshot moment.

The exact structure of the data needed for the connection restore process and some conditions for successful persistence of sockets will be discussed in the following sections of this chapter.

2.2.1 Snapshot structure

As discussed earlier, I decided to make TCP stack persistent using periodic snapshotting. To be useful in process of restoring the state, the snapshot should indeed have some data stored in it. The content of the snapshot is discussed in detail in this section.

When deciding about snapshot structure, two distinct approaches exist.

The first one is to make a full-memory snapshot, saving each page of virtual memory into persistent memory. To make it consistent, all threads that use these memory pages should be stopped, or writing to the pages should be prohibited. One of the disadvantages of the snapshot is that is not userspace friendly, since memory pages are usually managed in the kernel. The last downside is its relatively big size, in terms of hundreds of megabytes.

The second approach is to use selective snapshotting. In this case, when the component forms a snapshot, it should first read some set of state variables from memory then serialize them together. After that, the serialized structure is persisted. The benefits of this approach are easier implementation, the smaller size of the snapshot structure, which consequently leads to the faster snapshot/restore process.

I have chosen the second approach. Currently, Genode does not support

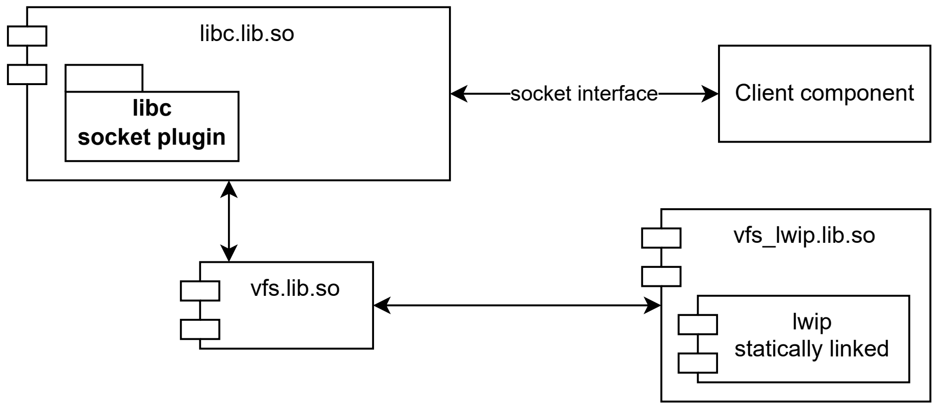

Figure 2.1
Control path of socket API

full-memory snapshotting, and implementing it from scratch is almost impossible. Though it may be possible to mimic a full memory snapshot for a single component by persisting all its dataspaces, it poses additional challenges. For example, a dataspace might be shared with other components that are not persisted. In this case persistent component expects its sibling to write something to the dataspace they used for memory-mapped communication earlier, and the sibling is unaware of any communications before power loss.

To snapshot socket state selectively, I was needed to decide on which data needs to be saved.

Figure 2.1 shows all libraries that are related to the socket interface. To find out, which of them contain state that needs to be snapshotted I have studied their source codes and how their clients should use them.

Libc is the library that provides a libc runtime and API for client applications. Part of the standard libc API is a socket API. It includes well-known functions socket(), bind(), listen(), and others. Genode libc implementation is based upon a FreeBSD libc with some adjustments. In this library socket API is implemented as a libc plugin. Libc plugin is a logically independent part of code, responsible for a single API. In this case, it is sockets API.

Libc socket plugin maintains a single Socket_fs::Context structure per each created socket. This structure encapsulates a socket state variable _state, which is used to determine which calls on the socket are allowed in the current state. The Context also has _fd_flags variable, which represents flags set on socket. To make the behavior of the plugin consistent after restore, the _state and _fd_flags variables should be restored.

Internally the libc plugin uses VFS to provide its functionality. VFS is implemented in form of a shared library, vfs.lib.so. This library is not interesting from point of snapshotting, as it does not contain any state itself. The library in our case just a proxy and a dynamic loader for VFS plugins.

State of vfs_lwip.lib.so should certainly be snapshotted, as this library contains information required for operation of the TCP itself. Information is stored in form of Protocol Control Blocks (PCBs). PCB is a structure used at a protocol level that holds all the data needed for the protocol stack socket uses. More details about PCBs can be found at [10]. Each socket corresponds to exactly one PCB. Fortunately, it seems to be enough to save the states of all PCBs and then restore them.

2.2.2 Snapshot location

The goal of this paper is to design a persistent networking stack suitable for Phantom OS. Due to this, it is possible to include a snapshot of the TCP stack into the Phantom VM snapshot structure. However, I decided that a persistent network stack can be useful not only for the Phantom OS but also for other persistent systems. These systems can use other mechanisms for persistence. Due to this, I decided to make an abstract interface that has functions for saving and loading binary data from persistent memory.

For simplicity, the only implementation of this service saves its state directly to the hard drive volume attached to VM. An implementation that saves and loads data using the Phantom snapshotting mechanism is currently in progress.

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] П. А. Аннин, “Краткий обзор room persistence library,” in *Актуальные направления научных исследований: перспективы развития*, 2018, pp. 146–147.

[4] R. Copeland, *Essential sqlalchemy.* " O’Reilly Media, Inc.", 2008.

[5] A. C. Bomberger, N. Hardy, A. Peri, *et al.*, “The keykos nanokernel architecture,” in *In Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[6] 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.

[7] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, “Peripheral state persistence and interrupt management for transiently powered

systems," in *NVMW 2018-9th Annual Non-Volatile Memories Workshop*, 2018, pp. 1–2.

[8] 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.

[9] N. Feske, "Genode. operating system framework 18.05. foundations," *Genode Labs*, 2015.

[10] W. R. Stevens and G. R. Wright, *TCP/IP Illustrated: volume 2.* Addison-wesley, 1996.