Chapter 1

Introduction

1.1 Background

Today, almost any enterprise application operates on data. A part of these data often should be *persistent*, that is, should outlive the process that created it. In this definition *a process* is not an operating system process, but instead some abstract operation that creates data.

The division between persistent and non-persistent data is not dichotomic. Several degrees of persistence exist – from temporary values during expression evaluations to long-stored records in a database. For example, when a customer uses a bank card to purchase goods, the process is the withdrawal of money from their bank account. This process creates a record about the withdrawal that can be seen later in a bank app. In the case of expression evaluation, the process is an execution of a CPU instruction.

This range of data persistence closely resembles the memory hierarchy of modern computers. Such similarity is for a reason: more persistent data are stored in slower and energy-independent upper levels of memory. The presence of the memory hierarchy brings the need for moving portions of data to lower memory levels when a program needs to access the data. On the other hand, the limited size of low memory levels forces programmers to move data that is no longer needed to the higher levels to make space for new data.

Modern operating systems manage moving data between the main memory and registers automatically, without the intervention from developers of applications for those systems. However, when an application needs to access data from the secondary storage, it should read and load data to the main memory by itself. Writing a program code for loading and saving data from and to secondary storage is a burden for developers of that applications.

To remedy this management of moving data between memory layers, Atkinson and Morrison [1] propose to use persistent support systems. These systems are a software for automatic management of physical memory layers. The systems provide its users with a sandbox to run programs where all data virtually have the same persistence.

1.2 Phantom OS

The idea of persistent support systems was developed later with changing focus to operating systems [2], [3]. The most modern of them up to date is Phantom OS. This system consists of a stateless kernel and a Phantom Virtual Machine (PVM). PVM hosts processes, which execution states persist across restarts of the host machine. Persistence is achieved with periodic snapshotting. In the context of Phantom OS, a snapshot is a memory dump of the PVM memory space. The system uses the latest snapshot as a recovery point on booting.

Even though Phantom OS is a working persistent support system, there is still a room for improvement. Currently, Phantom OS does not restore the state of the Transmission Control Protocol (TCP) stack. The operating system only handles sockets in the LISTEN state. If a socket was in this state when the snapshot was taken, the system will reopen it on boot and bind to the same address. Sockets in other states will become invalid. Any attempt to use these sockets will result in an error. The application that tries to use such socket will receive this error and will be responsible for its handling. Usually, that handling implies reestablishing a connection with a remote peer and restarting data transmission from the very beginning.

Because of the presence of these errors, existence in a persistent environment is not completely transparent for Phantom OS applications. Moreover, the common way of handling these errors, which was described above, results in inefficient network use, especially when power is off for a short period of time.

1.3 Problem statement

Recently, Phantom OS was ported to the Genode OS framework. The PVM was ported as a userspace Genode component, and all PVM syscalls are implemented as functions provided by other Genode parts. In particular, the network stack was implemented in the Genode as a Virtual File System (VFS) plugin. Due to this fact and the flexibility of the Genode VFS, changing implementation of the networking stack became easier in comparison to Phantom OS before this port. My hypothesis is that it is possible to develop an enhancement to the Phantom networking stack entirely in the Genode part of the port.

The original document describing TCP [4] suggests using a Finite State

Machine (FSM) to implement the protocol. Most implementations of the protocol, including the one used in Phantom-over-Genode (PoG), use this FSM technique. The theoretical challenge behind implementation of the enhancement described above is the integration of an external state machine into the Phantom persistence model. In this context, “external” means that the code of the state machine is not running as managed code inside PVM. The first aim of this paper is to develop a methodology for integrating such state machines into Phantom OS.

The second aim of this paper is to enhance PoG TCP stack to achieve two goals. The first is to reduce the number of errors that should be handled by applications supervised by the PVM. This means that as many errors as possible should be either prevented or handled at the Genode layer of the OS. The second goal is to make the network utilization more efficient. This means avoiding extra TCP transmissions, if they are not necessary.

The rest of this thesis is structured as follows: Chapter 2 contains a detailed description of persistence concept in and a review of related work. Chapter 3 describes implementation details of the PoG port, approaches that were tried during the implementation and reasons why some of them were discarded in favor of others. Chapter 4 contains description of achieved results, experiments to evaluate them and a future directions for networking in persistent systems.

Chapter 2

Literature Review

This chapter begins with an overview of the concept of persistence in general and orthogonal persistence in Section 1. Section 2 describes TCP, tries to establish requirements for a networking stack in persistent systems and discusses related works. The last section of this chapter describes Phantom OS, Genode OS framework and the current state of porting the former to the latter.

2.1 Persistence

2.1.1 Persistence definitions

The concept of *data persistence* was informally introduced in Section 1.1. The formal definition of this concept is presented below. The lifetime of a data is a time extent over which it can be used. This lifetime is commonly called *persistence* [5].

Atkinson *et al.* [1], [5] give a classification of application data by their persistence. This classification is presented in Table I. As it was said earlier, the support of levels 1-4 is usually provided by an operating system and a programming language. This is due to the fact that these data are stored at the main memory or lower memory levels. On the other hand, to use levels 5-8 applications designers should rely on some external component, such as database or file system.

Table I
Classification of data based on their lifetime

1.	Intermediate 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

2.1.2 Orthogonal persistence

As was stated in section 1.1 using inherent programming languages mechanisms and external mechanisms to access data interchangeably is a burden for programmers. Presence of different data formats in each of these storages makes this burden even heavier. As mentioned earlier, in response to this problem, [1] proposes to use persistent support systems, which act as a mediator between supervised applications and a transient environment. Atkinson *et al.* also summarize design requirements for such system, which they call *Principles of Orthogonal Persistence.* The authors define the following requirements for an orthogonally persistent support system:

1. The Principle of Persistence Independence.

Whether a program manipulates data that outlive 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.

   Any part of data 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 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.

A system that treats data according to these three principles is said to be *orthogonally persistent.* Two popular approaches exist to build such a system. The first one is to integrate interactions with databases or file systems into an existing programming languages. In this case a program syntax to access data stored in the memory should look the same as access to data in a database. But a semantics behind this syntax can vary. This idea is quite similar to work of modern Object-Relational Mapping (ORM) libraries [6], [7]. However, this approach is not really orthogonally persistent since most ORM libraries put certain limitations to what can be saved to the database. It also is sometimes problematic because a different meaning behind seemingly similar fragments of code confuses programmers.

The second approach is to build so-called *persistent worlds* [1]. These worlds usually have form of an operating system or a virtual machine. Applications residing in persistent worlds are written with managed code. From now on I will use the term *persistent applications* for resident applications of persistent worlds.

The fact that persistent applications are generally written with managed code enables a system to automatically manage states of the applications so that they have a consistent behavior across restarts. The examples of such systems are KeyKOS [8], Grasshopper OS [3], and Phantom OS.

Creation of persistent support systems that work with nonvolatile memory (NVM) as a main memory is an emerging research direction in field of persistent support systems. This approach seems to be more convenient for implementation of orthogonally persistent systems, but it still have unsolved problems. The two examples of such problems are problem of dealing with changed states of peripheral devices after restarts [9], and logical problems with code execution [10]. That is why creation of such system is not a trivial question.

2.2 Networking in persistent systems

The nature of persistent applications implies that their execution can be interrupted and then resumed after some time, when a host machine reboots. This creates a challenge when an application is designed to communicate with remote peers with a stateful network protocol. The challenge is to synchronize states of the communicating peers: while a local system is offline a remote peer can change a state of the connection indefinitely. Moreover, some persistent support systems, like Phantom OS, use snapshotting to create an illusion of uninterrupted execution. In case of such systems, the challenge is even more problematic, because a persistent peer can "forget" that it had sent or received some data. This happens if a record about data transfer was not embedded in a snapshot, for example due to an unexpected power loss.

The most popular transport layer protocol – TCP – is designed to provide a reliable packet delivery service. The reliability is achieved with periodic retransmissions and acknowledgements. When a remote peer does not acknowledge a packet TCP prescribes to retransmit it. The maximum number of retransmissions is not stated in the original TCP specification [4]. However, the specification received updates later, which prescribe to retransmit packets with exponentially increasing delay [11] until the timeout of at least 100 seconds [12]. This means, that the most TCP implementations will close a connection if their peer appears to be down for more than 100 seconds.

The previous part of this section explains why persistent support systems should restore state of network connections in restoration phase when the execution was interrupted by a power loss. Several approaches were proposed to mitigate this issue. The first approach, proposed by Ekwall, Urbán and Schiper [13], is to create a session layer protocol that will behave similar to the original TCP specification with respect of timeouts. That is, the protocol will retransmit packets infinitely reopening underlying TCP connections as they get closed.

The second approach was proposed by Zhang and Dao [14]. This article, as well as the previous one, proposes to use a session-layer protocol to create an abstraction of persistent connections, i.e. connections that outlive execution of a one peer. The authors propose to use a centralized notification service to transfer control messages. When this service detects that a process goes down the service notifies all peers communicating with the process. These processes switch their ends of connection to passive mode. In this mode blocking read operations are getting blocked, and writes are buffered. When the down process resumes its execution it registers in the notification service. The former peers of the process receive this notification and try to establish a connection to the continuation of the process. However, this approach can lead to loss of data that were in flight when a process comes down.

Creation of a session-layer protocol on top of TCP is not the only way to implement persistent connections. Zandy *et al.* in [15] propose a mechanism that hides disappearance of remote peer from applications that use TCP. The mechanism named *rock* employs heartbeat probes via a separate UDP control socket to detect if a remote peer is gone. When a process detects loss of connection with a remote peer the process repeatedly tries to reconnect to its peer by the last known physical address. If loss of a peer was caused by network error, this will eventually succeed. Otherwise, reconnection attempt will be timed out. However, rocks mechanism provide large timeout and it can be changed in per-connection basis, unlike TCP.

To avoid loss of in-flight data when process restarts a rock keeps a buffer sized as sum of local host send buffer and peer host receive buffer. The buffer is used in a similar way as send buffer is used in TCP: the packets in the buffer are used to perform retransmissions.

2.3 Phantom OS and Genode OS framework

2.3.1 Phantom OS

Phantom OS is a general-purpose operating system, developed in 2009-2011. The operating system consists of a stateless kernel and virtual machine, which executes userspace applications. Phantom OS applications are written in Phantom programming language. The execution of managed code is persistent, which makes Phantom OS a persistent support system. As was said earlier, persistence for the managed code is achieved using snapshotting. 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 embedded 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. Implementation of drivers as parts of kernel also complicate development and delivery of the drivers.

These are two reasons why it is desirable to replace Phantom kernel with a microkernel. The port of Phantom OS to Genode OS framework is being developed right now. It is implemented as a Genode component running the PVM. This means, that network capabilities are provided by the Genode level of the PoG port. That is why I will concentrate my attention on development of a network stack working in Genode.

2.3.2 Genode OS framework

The Genode OS Framework [16] is a toolkit for creating operating systems. It provides possibility to build a microkernel operating systems from set of existing components. Genode also provides an API to integrate various microkernels into it.

For now the supported kernels 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. Genode has a VFS with possibility of adding custom plugins to it. For instance, a part of process memory can be exposed as a file with the ram-fs plugin. TCP stacks are also implemented as VFS plugins that expose each socket as a set of files.

Genode's IPC mostly consists of a blocking remote procedure calls (RPC). The TCP stacks use the Network interface card (Nic) RPC interface as an input. VFS plugins are not persistent. Therefore, I need to implement saving and restoring of state of a TCP stack before it can be used by PVM.

Chapter 3

Methodology

This chapter starts with the analysis of requirements for the implemented software. In the second section of the chapter I will discuss possible approaches to adapt an external state machine to Phantom OS persistency model. The last section describes the implementation and design decisions in detail.

3.1 Requirements analysis

As previous chapters of this paper suggest, the goal of this paper is to create an implementation of a TCP suitable for a persistent system. More precisely, I target a port of Phantom OS to Genode OS framework. This section contains a list of requirements that I will follow while designing my own protocol implementation.

3.1.1 Transparency for client applications

Persistent support systems (PSSs) aim to make development of applications easier. The PSSs provide their clients with an abstraction of an uninterrupted execuiton. In a real world, however, a computer system may experience downtimes. The point of a PSS is to hide these downtimes from applications that use it. Such hiding of undesired effects is generally called *transparency*.

As discussed in Section 1.2, the current implementation of networking stack is not transparent for the applications residing in PVM. Application designers should foresee existence of these errors and write handlers for them. To make PoG port really orthogonally persistent these errors should be hadled at PSS level, not by clients.

That means that the state of TCP connections should be the same before a shutdown and after a reboot. To achieve that a persistent TCP implementation should either (1) backup and restore connections state or (2) make the disappearance of a remote peer invisible for application on the other end of the connection. Of course, restoration of the state should be done before returning control to applications.

(1) can be achieved with reestablishing connections on startup. However, the remote peer should be ready to the reestablishment request. This is problematic in case when a persistent host plays role of a server in a client-server communication. (2) can be achieved with use of an message broker that will mimic an alive host with full TCP input queue. Thus it will keep the connection alive but prohibit sending packets to the channel. Another way to achieve (2) is to send a control message to the remote socket. The remote TCP in this case should efficiently do the same thing as a message broker did. It should mimic live connection that cannot accept data. The drawback of this approach is that to exchange control messages both communicating peer should have enhanced TCP stack.

3.1.2 Robustness to rollbacks

Phantom OS uses full-memory snapshots of PVM memory space to provide persistence for its clients. Snapshots are taken in a live mode so that processes inside the VM continue to run. Due to the two previous facts, computational operations of PVM applications can be executed more than once. For example, assume that an application receives packets A, B, and C. Assume that after packet A is received, snapshot is performed. While it was on progress, the application receives packet B. After the snapshot was made, applications reveives packet C and immediately after this machine goes down. After the snapshot would be restored the application will have no knowledge that it already received packets B and C. Thus the application will send ACK packet for A, which will result in error because the remote peer does not expect this ACK. Moreover, it is unable to retransmit this packet, since it is not exist in remote TCP's retransmission queue, because it was already acknowledged when the application received B and C.

Phantom OS snapshot mechanism takes into account that snapshot is not a atomic process. Thus, in usual case it would handle reception of packet B properly, i.e. the client application would know that B was received. But in case of PoG the TCP stack is not stateless like Phantom kernel but still transient. Therefore, if TCP receives a packet but does not passes it to the application before restart, this packet may be lost. The remote peer expects that the application processed it but the application never received it because TCP did not finish delivering it to the application.

Three solutions exist that may solve the problem above. The first is to refrain from 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 case a snapshot should be performed each 50ms to avoid extra network congestion at all. However, in PoG a snapshot runs for around 10 seconds. This approach is not acceptable with such a long snapshot.

The second solution is to keep in persistent memory a buffer that will store all packets that TCP clients had not consumed yet. In this case persistent memory can be a regular file or snapshotted memory space.

The third approach is to have a message broker inside a network that will function similarly to the buffer in the second approach. 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 stateless network protocol could be useful to notify the broker about the end of the snapshot process.

3.1.3 Usefullness

Phantom OS is already capable of restoring sockets in CLOSED and LISTEN states. For a Genode implementation of a persistent TCP to be useful it should work for the same or bigger set of states. Even if the developed software will only support the same set of states the fact that the software is implemented as a Genode component will greatly improve maintainablility of the software.

3.1.4 Compatibility with existing protocols

The last 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 reason to make persistent TCP implementation a standalone application is that the PoG port is not yet ready for me to experiment 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 that are 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.

3.2 On persisting of finite state machines

Most TCP implementations implement logical state of the sockets as finite state machines. Since the goal of the paper is to create a persistent TCP implementation, it might be useful to consider how an arbitrary FSM can be snapshotted to restore it later to the same state from the snapshot.

The most straightforward and universal way to achive the desired outcome is to use a full-memory snapshot of a running process. Full-memory snapshot should include stack and address space. Also it should include current context, i.e. general-purpose registers and the program counter. Obtained snapshot should be stored in a persistent memory and loaded to the apropriate memory on restore. In principle, this approach would work for any running process.

The second approach is to directly access current state of the FSM. Usually FSM are implemented on C and C++ as structs or classes holding current state as an value of an enumerable type. If it is possible to access these structures directly, simple saving and loading them would work. In some cases, for example sockets, there are several state machines: one per socket. In such cases FSMs should be registered at the supervisor that controls inputs and stores FSM structures.

The last approach requires knowing of internal functioning of an FSM. When all states and transitions of FSM are known in advance, it is possible to implement FSM with the same transitions. When the tracking FSM is provided with the same inputs as the target one states of both FSMs should be in sync. Assessing state of the tracking FSM will allow observer to determine state of the target FSM. Moreover, tracking of inputs greatly helps in restoration phase. To restore state of the target FSM it is enough to provide it with the same inputs once again. This approach is useful when an FSM does not have a centralized state or when it cannot be easily accessed.

3.3 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.

3.3.1 Snapshot structure

As discussed earlier, I decided to make a 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 snapshot consistent, all threads that use these memory pages should be stopped, or writing to the pages should be prohibited. One of the disadvantages of this type of snapshot is that it is not

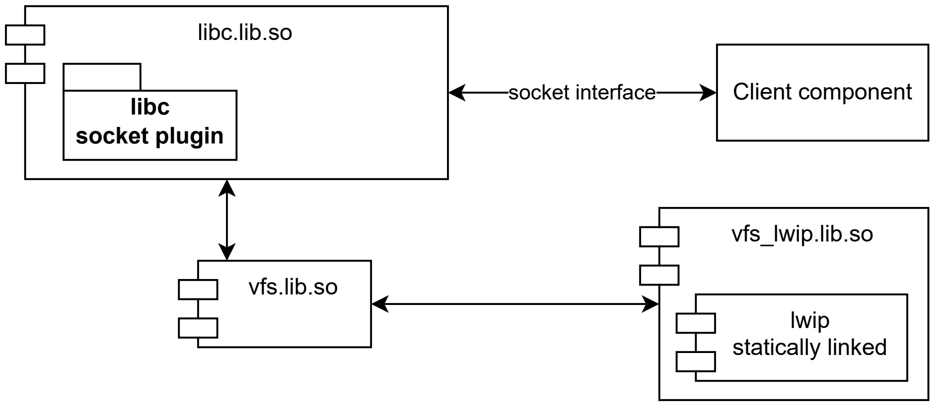

Fig. 3.1. Control path of socket API

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 certain 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, and the smaller size of the snapshot structure. The smaller snapshot size leads to the faster snapshot/restore process.

I have chosen the second approach. Currently, Genode does not support 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 needed to decide which data needs to be saved.

Figure 3.1 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.

The 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 is 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 [17]. Each socket corresponds to exactly one PCB. Fortunately, it seems to be enough to save the states of all PCBs and then restore them.

3.3.2 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.

/subsectionApproach with memory As a first approach I tried to save memory pages that belong to network stack and restore them on booting. At first I tried to snapshot the whole virtual address space of the component that uses network stack. However, this was problematic due to the nature of data inside a components' virtual memory. The problem was with kernel capabilities – they are used throughout almost each part of the OS framework. In fact, the component can not exist without using at least one capability, and can not execute any code without using at least three - namely Parent, CPU and RAM capabilities. This ubiquity of the capabilities and the fact that they become invalid after restart make straightforward dumping of full component's address space to disk impossible.

The idea of using a proxy that will mask sequence numbers was inspired by proposal of using packet filters (see rocks-racks).

Chapter 4

Evaluation and Discussion

. . .

Chapter 5

Conclusion

. . .

Bibliography cited

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

[2] 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.

[3] 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.

[4] P. John, “Transmission control protocol,” *RFC 793*, 1981.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] V. Paxson, M. Allman, J. Chu, and M. Sargent, “Computing tcp’s retransmission timer,” Tech. Rep., 2011.

[12] R. Braden, *Rfc1122: Requirements for internet hosts-communication layers*, 1989.

[13] R. Ekwall, P. Urbán, and A. Schiper, “Robust tcp connections for fault tolerant computing,” in *Ninth International Conference on Parallel and Distributed Systems, 2002. Proceedings.*, IEEE, 2002, pp. 501–508.

[14] Y. Zhang and S. Dao, “A" persistent connection" model for mobile and distributed systems,” in *Proceedings of Fourth International Conference on Computer Communications and Networks-IC3N’95*, IEEE, 1995, pp. 300–307.

[15] V. C. Zandy and B. P. Miller, “Reliable network connections,” in *Proceedings of the 8th annual International Conference on Mobile Computing and Networking*, 2002, pp. 95–106.

[16] N. Feske, *Genode. Operating system framework. Foundations. 21.05.* Genode Labs, 2021.

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