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, it should outlive the process that created it. In this definition *a process* is not an operating system process, but instead a process is an 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 the customer's bank account. This process creates a record about the withdrawal that can be seen later in a bank app. In case of expression evaluation, the process is an execution of a CPU instruction. This process creates a data that is stored in CPU register.

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 upper levels of memory. These memory levels are often energy-independent and slower than lower ones. The presence of the memory hierarchy brings the need for moving portions of data to lower memory levels when a program needs to access a certain portion of 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 and programming lanuages manage moving data between the main memory and registers automatically, without the intervention from application developers. However, when an application needs to access data from a secondary storage, it should read and load data to the main memory by itself. Writing a boilerplate program code for loading and saving data from and to secondary storage adds more complexity to programs. Thus, it 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]–[4]. 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, bind to the same address and start listening on it. 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, a group of researchers from Innopolis University started porting Phantom OS to the Genode OS framework. The PVM in this port is implemented as a userspace Genode process, and all the Phantom kernel syscalls are implemented as functions provided by the Genode layer of the port. 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.

The functionality of restoration of sockets state was not in the scope of this port. Thus, even the inherent Phantom OS ability to restore listening sockets was lost in this port. My hypothesis is that it is possible to develop an enhancement to the networking stack of Phantom OS port. This enhancement should reside entirely in the Genode part of the port. The enhancement should provide client applications with an abstraction of a persistent socket, i.e. socket that can be used after any number of restarts of a host machine. To simplify the problem I will implement the mechanism working only for short-duration shutdowns that gets unnoticed by the remote side. However, I will consider a cooperative mechanisms that could allow handling longer shutdowns.

The original document describing TCP [5] suggests using one Finite State Machine (FSM) per socket to implement the protocol. Most implementations of the protocol, including the one used in Phantom-over-Genode (PoG) port, use this FSM technique. The theoretical challenge behind an 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. Thus, 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 goal 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 the persistence concept, brief overview of Phantom OS, the Genode framework and TCP, and a review of related work in creation of persistent networking stacks. Chapter 3 establishes requirements for the implemented software and discusses theoretical approaches that can be used to implement the software. Chapter 4 discusses relevant implementation details of the PoG port, approaches that were tried during the implementation, and reasons why some of them were discarded in favor of the others. Chapter 5 contains a description of the achieved results, description of experiments to evaluate them and future directions. Chapter 6 contains a brief summary of this paper.

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* [6].

Atkinson *et al.* [1], [6] 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 [7], [8]. 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. Ap-

plications 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 [9], 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 [10], and logical problems with code execution [11]. 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 [5]. However, the specification received updates later, which prescribe to retransmit packets with exponentially increasing delay [12] until the timeout of at least 100 seconds [13]. 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 [14], 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 [15]. 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 [16] 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 [17] 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 uninter-

rupted 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 receives 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. A drawback of this approach is that an operating system should provide mechanisms to allow such manipulations with address spaces.

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 the set of all possible states and transitions of FSM is 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.

Chapter 4

Implementation process

4.1 Components of Genode network stack

Before discussing the implementation details of developed persistent TCP stack it is necessary to consider how Genode network stack works. Figure 4.1 depicts components and libraries that are required to enable a network access for a client application. In this diagram Nic stands for Network interface controller. Nic component is a mediator for packets that are produced and consumed by applications and network drivers. As can be seen from the diagram, Nic interface provides two RCP interfaces: the Uplink interface is used by network drivers, and the Nic interface is used by client applications. Nic interface exposes network packets as Ethernet frames.

The Nic RPC interface is almost never used directly. Instead most applications tend to use Berkeley sockets API, which include well-known functions socket(), bind(), connect(), listen, and others. This API is provided by libc. Genode's libc is based by FreeBSD libc and its functions are implemented by so-called libc plugins. Berkeley sockets API is implemented in the socket_fs plu-

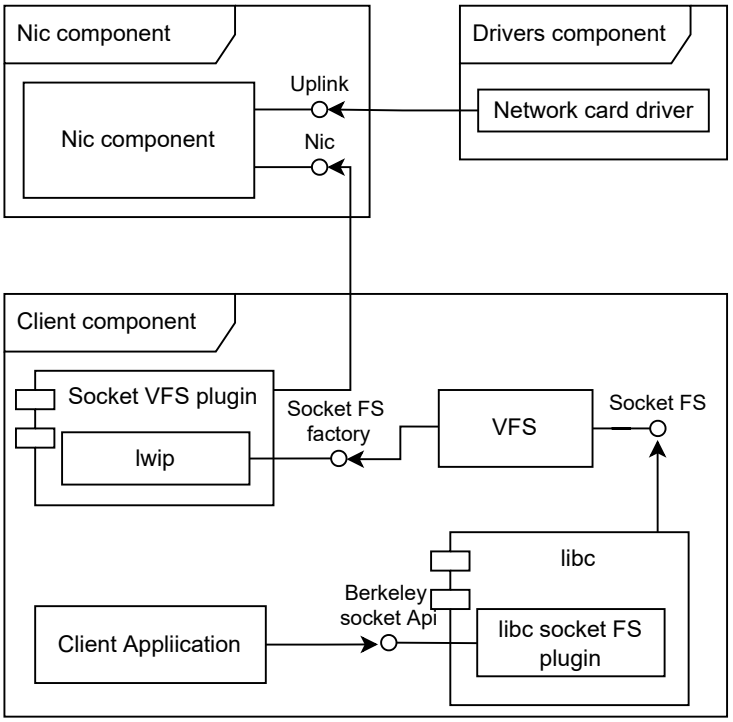

Fig. 4.1. Components of Genode network stack

gin. This plugin depends on Genode VFS. The plugin expects VFS to expose each socket as a set of virtual files.

The socket_fs plugin should be used in conjunction with a socket VFS plugin. Such VFS plugin generally is merely an adapter between a Nic session and port of some popular TCP stack. A port is denoted by "lwip" on the Figure 4.1. For now, two TCP ports available in Genode – lxip, which is a port of Linux TCP/IP stack, and lwip. Usually TCP stack is statically linked with a VFS plugin, and VFS plugins are shared libraries which are loaded in runtime.

Each Genode component has an initial configuration that is written as an XML "config" node. The node also may contain configuration for VFS, if a component needs one. Configurations for VFS and libc are located at "vfs" and "libc" sub-nodes of "config". Each VFS directory can have a plugin that will handle access to the directory and all its subdirectories and files. Plugins libraries are resolved in the runtime when a component initializes. To use

libc with a socket VFS plugin component should place a VFS plugin to some directory and pass path to this directory to the libc config. A minimal example configuration for Genode component that is able to use network is given in the Appendix A.1

I had a hypothesis that persistent TCP stack can be implemented as a VFS plugin wrapping another VFS plugin. The inner VFS plugin would provide a "real" TCP interface, and the outer plugin would supervise the inner one to save and restore its state as needed. I decided to name the plugin PTCP from "persistent TCP". PTCP uses lwip VFS plugin. PTCP is dynamically linked with the plugin, at the runtime it loads vfs_lwip shared library. This dynamic linkage was done by design, with the assumption that system that has components using PTCP can also have components that use plain lwip VFS plugin, since it is popular among Genode software.

4.2 Snapshot structure

The PoG port was not ready during the most of the time this work was in progress. However, the technique this port uses to achieve persistence was known in advance. The technique is snapshotting just like in plain Phantom OS. To use snapshotting I needed to decide what data should be included in a snapshot. I tried several different approaches with respect to the snapshot content. The content of a snapshot is discussed in detail in this section.

4.2.1 Access to TCP internal data structures

The first attempt to snapshot a TCP implementation concluded in saving the state of socket state machine. This is the second approach proposed in

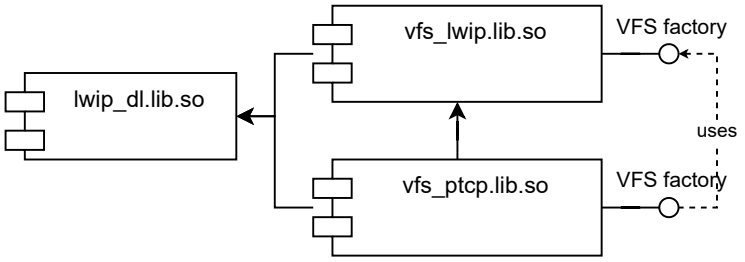

Fig. 4.2. Relationships between the libraries comprising PTCP

Section 3.2. For simplicity I decided to start with saving and restoring a specific VFS plugin – lwip. lwip represents a socket as a protocol control block (PCB). PCB is a structure that holds values that are required for protocol functioning. For example, TCP PCB encapsulates current state, SEQ number, ACK number, and several other fields. lwip keeps all PCBs in the few linked lists. References to these lists are located at static section of lwip binary.

The challenge was to access the lists from the PTCP plugin. PTCP can access the address space of the inner plugin but the problem was that lwip is statically linked to the vfs_lwip plugin. Moreover, all lwip symbols are erased during the linkage from the binary. The only symbol exposed from vfs_lwip is a function that returns a factory for lwip plugin instances. To overcome this issue I reconfigured vfs_lwip to include lwip as a shared library. With this reconfiguration I was able to access lwip internal structures from PTCP plugin. Both vfs_lwip and PTCP used the lwip shared library. The VFS pluigins existed in the same Genode component, so they used the same "instance" of lwip. Relationships between the libraries are presented at the Figure 4.2. There lwip_dl.lib.so denotes lwip in form of shared library.

After I manage to access the internal state of lwip in form of PCBs I started implementing saving it to persistent storage. A PCB is a C structure and I tried to save each PCB corresponding to socket field by field to a file.

However, in the process of implementing I discovered the two issues that made me abandon this approach. The first problem that I discovered was a rather complex structure of the PCBs. The tcp_pcb structure includes more than 60 members and some of them are not atomic values but rather nested structures or buffers. Also list of members of tcp_pcb can vary depending on compile-time config. The second issue was that despite my assumption sockets state was stored not solely on lwip PCB lists but also at the lwip VFS plugin level and even in libc. Therefore even if I would proceed with snapshotting lwip internal structures I would not achieve desired effect.

To mitigate the two issues above I decided to abandon the idea of saving state holding structures on the field-by-filed basis and instead proceed with saving and loading memory that belongs to them.

4.2.2 Approach with memory

As a second approach I tried to save memory pages that belong to network stack and restore them on component startup. Genode manages virtual memory with use of Allocator class instances. I have subclassed the base Allocator class and created a class that was able to track virtual addresses it allocated, save allocated regions to a file, and load them back to virtual memory. I applied this allocator so that it saved and restored heap dataspaces related to libc, lwip VFS plugin and lwip itself.

At this point I faced a problem related to restoration of heap space. If a heap space is prepopuilated before an allocator is passed to its user it will never use this space. The allocator will never return adresses belonging to the address range of the space in response to the user's allocation request. After I understood the problem I changed the implementation so that it overwrote

an address range *after* it was returned in response to user's request. That is the program detected that the TCP stack finished its initialization and then overwrote its address space with memory from previous boot of the system. However, this approach did not work by the two reasons.

The first reason was the presence of the kernel capabilities inside the address space. Capabilities are used throughout almost each part of the Genode 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. lwip VFS plugin should use eight more capabilities to use Nic RPC interface. Each allocated address range consumes one dataspace capability to use virtual memory service RPC interface. This ubiquity of capabilities and the fact that they become invalid after restart make straightforward saving and loading of component's memory to disk impossible. This approach becomes even more complicated if a snapshotted component uses shared memory. For example, Nic servers use zero-copy packet delivery mechanism to increase memory throughput. This mechanism implies establishment of shared memory between client and server components.

The second and the main issue with the raw memory saving approach is that layout of structures in the memory is undefined. The layout depends on order of allocation requests and presence of Address space layout randomization (ASLR). The interface of Allocator class does not allow the allocator determine the purpose of allocation, i.e. what structure will be placed in the newly created dataspase. These two issues explain why I decided to stop work in this direction and decided to shift focus to a completely different approach.

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.

4.2.3 Observer state machine

The main idea of the new approach was to build a tracker that will know the internal state of every socket without directly accessing those states. This idea is the third approach that was proposed in section 3.2. We can treat each socket as a graybox where the inputs are user-called functions and TCP packets from network. The outputs of this box are packets that are scheduled for sending at the Nic server and data chunks delivered to clients of TCP stack.

The libc transforms user calls to Berkeley socket API into file access operations. This operations are processed by Genode VFS. It delegates operations to its plugins. In our case all socket operations are processed by the PTCP plugin. To enable PTCP track VFS calls I created a wrapper around FS factory exposed by the PTCP inner plugin. This wrapper creates a new proxy filesystem that delegates processing all calls to FS returned by the inner plugin. The proxy FS is equipped with a tracking delegate that notifies the tracker about each accessed file. The tracker keeps a one metadata structure per each socket. When a tracking delegate notifies the tracker it searches the socket metadata that corresponds to the accesses file. Then the tracker updates the structure according to the new socket state. The tracker also keeps in account the result of file access. For example, when writing to a bind file finished with error, the state of a socket did no changed. Hence, no need to update the correspondent metadata structure.

To enable tracking and control for the arriving network packets I added a secondary Nic server that scans headers of incoming packets and notifies the

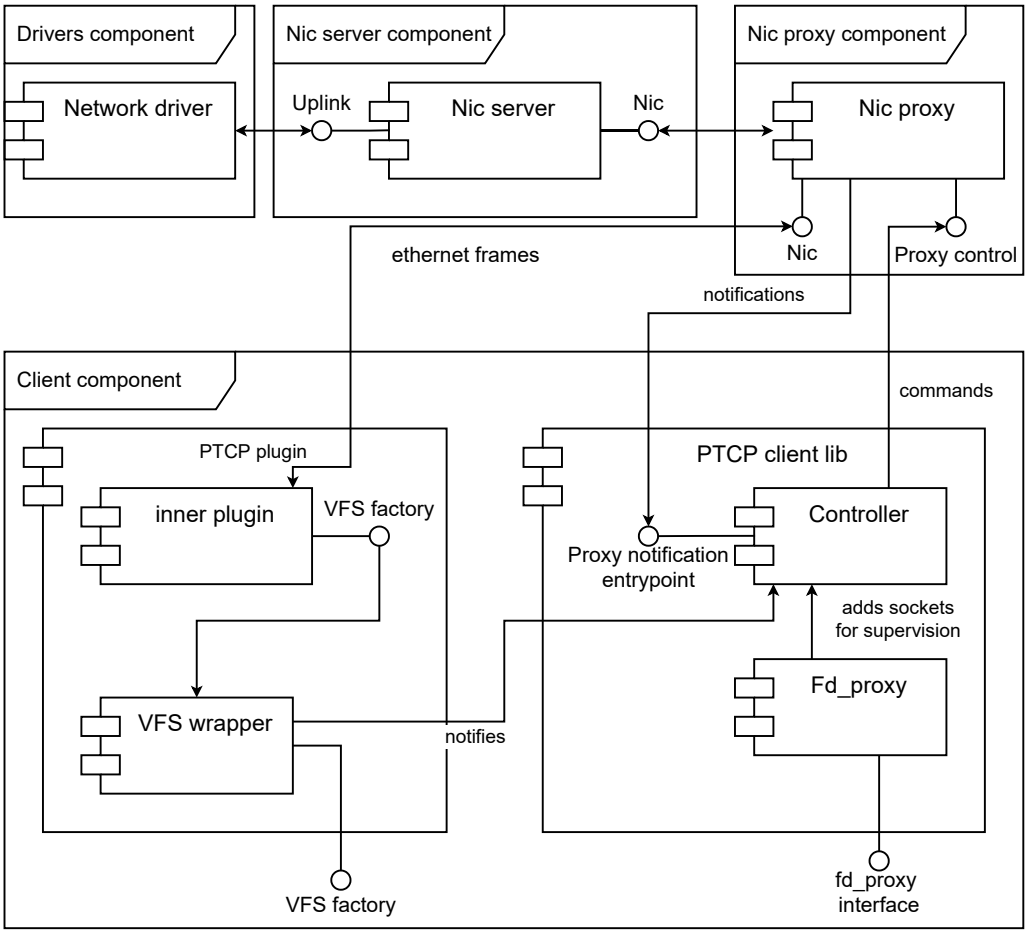

Fig. 4.3. Structure of software components involved in snapshotting of TCP stack

tracker. The tracker is the same one that was discussed above. The presence of tracker allows PTCP to intercept incoming packets without changing code of the inner plugin.

To restore state of the sockets I added a client library that is used on component startup. During initialization the library reads a snapshot, reopens each socket present in the snapshot, and brings them to the last known state. Restoration of the state is performed with the sequence of calls to Berkeley socket API, and RPC calls to the proxy Nic server. The proxy Nic is responsible for restoration of those parts of socket state that can not be set by user. For example, these parts include sequence and acknowledgements numbers. To

restore those values the Nic proxy acts similar to the NAT mechanism in routers. But instead of changing source and destination ports values in TCP packet headers the Nic proxy changes headers so that the remote side of the connection would recognise the newly opened socket as the old one. The idea of proxy was highly inspired by the use of Linux kernel packet filters in [16]. The complete architecture of network stack with tracking and restoring of TCP sockets states is presented on Figure 4.3.

4.3 Sockets naming

To use sockets an application should have a way of distinguishing between them. An application in a persistent environment expects that it will be able to use objects using the same identifiers before and after restart. This explains a need in a persistent socket handles for PTCP.

Each new socket created with the Berkeley socket API has a unique identifier. This identifier is an integer number that corresponds to a socket file descriptor. In Genode these file descriptors are stored inside the libc. They are assigned to the files in the order of opening in a sequential manner. Neither libc no Berkeley sockets API have a way of expressing desired file descriptor when a socket is opened. That is why it is impossible to guarantee that sockets will have the same libc file descriptors before and after restart.

To workaround this problem I created a proxying file descriptor mechanism. This mechanism is merely a map between a persistent file descriptor and a libc one. The persistent handles are saved together with socket metadata blocks used in a restoration phase. This allows to restore state of each socket and prepopulate the map with new libc file descriptors before returning

control to an application. The drawback of this approach is that applications can not use Berkeley socket API with the persistent socket descriptors until the proxying mechanism is not integrated to the libc.

4.4 Snapshot consistency

Saving state of a TCP stack to persistent memory is not an atomic operation. It is not desirable to stop execution of the entire component until it is finished. That is why forming of snapshot structure and writing to a file is executed in a separate thread. However, this creates a problem of making a snapshot consistent. For example, while a socket metadata structure is being written to persistent storage the socket might receive a packet. The controller will intercept the packet and change the metadata structure accordingly. This modification would make the structure corrupted.

To avoid this problem I created a critical section inside a snapshotting function. When execution flow enters this section, it acquires the mutex that blocks all calls to PTCP VFS plugin and suspends the Nic proxy. The function suspends the Nic proxy via the Proxy control RPC interface. The Nic proxy handles RPC requests and incoming packets in the different threads. That allows to suspend packet receiving thread until snapshot structure will be formed. When the process of forming is complete, PTCP calls an RPC method that resumes execution of the packet processing thread in the Nic proxy. Then it releases the PTCP VFS plugin mutex.

The incoming packets that arrived while Nic proxy was suspended are buffered by a Nic server component. To avoid overflow of the buffer I tried to make critical section as short as possible. Due to this the writing of memory to

the disk is not within the critical section. Instead, the critical section contains only copying values that will be written to a thread-local storage. This storage is used solely by the snapshot thread and thus is not modified concurrently. After snapshot is formed it is safely written to the persistent memory file. For now this persistent memory is just a file, but in principle it can be anything, even a part of Phantom OS snapshot.

4.5 Integration with Phantom OS

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 be implemented entirely in Genode and 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.

Phantom OS should notify PTCP about snapshot so that their shapshots would be synchronized. The component that hosts Phantom VM implements Snapshot_notifier RPC interface. This interface allows an application to suscribe to snapshot notifications. When PoG begins a snapshot the application receives the notification and starts snapshot process of itself. After the

snapshot of PTCP is finished it is ready to process new packets. Restoration of the PTCP state is performed when its component initializes.

The duration of PTCP snapshot is generally shorter than one of the PoG port. This might create issues with snapshot consistency. However, it is hard to reason about this issue since PTCP was not tested together with PoG port.

Chapter 5

Evaluation and Discussion

This chapter begins with the brief description of achieved results. The second section discusses of how the requirements formulated in section 3.1 were implemented in PTCP. Then the chapter discusses testing scenarios created to test if the implemented mechanism works. After that the results of tests are discussed. The end of the chapter contains future directions in a field of network stack of PoG port.

5.1 Results

During this work I created a PTCP mechanism in form of Genode VFS plugin, Genode Nic proxying component and a shared library. The mechanism can be used to restore states of TCP sockets if execution of a program was interrupted by a shutdown. In particular, this can be useful for Phantom OS port to Genode OS framework.

The current limitations of the PTCP mechanism is that it is only works with TCP sockets in closed and listen states. However, this limitation is only due to the lack of time. In principle, nothing blocks addition of code for handling

other states. With this limitation, however, PTCP is still useful with the PoG port, because its capabilities are the same as ones of the Phantom OS kernel TCP stack. The PoG port aims to replace the Phantom kernel with Genode components. Hence, PTCP can be a suitable replacement for the TCP stack previously implemented as a part of Phantom kernel. Furthermore, PTCP is easier mantained, because it is independent and its codebase is quite small.

5.2 Meeting the requirements

5.2.1 Transparency for client applications

PTCP is not completely transparent for client applications. To use it, application should use a PTCP client library and add function call that will initialize the library at component startup. This lack of transparency is a result of compromise between problem of creation of persistent socket names and unwillingness of modifying Genode libc. I do not want to modify libc because it belongs to a trusted computing base. Any change in it should be done with caution and should be verified.

If an application does not need the persistent sockets functionality and instead wants to use PTCP as a regular TCP stack then PTCP acts as any other TCP stack available in Genode. That is, PTCP VFS plugin is added as a backing plugin for socket directory and path to the directory is passed to libc.

5.2.2 Robustness to rollbacks

PTCP fails to properly process rollbacks. In case of unexpected shutdown packets that were received by PTCP but were not delivered to applications will

be lost. However, the architecture including a proxying Nic component allows PTCP to replay the packets that were in reception queue when a host machine came down. However, in this case PTCP should somehow track what packets are in a send queue and this seems to be challenging.

5.2.3 Usefullness

The usefullness requirement was satisfied by PTCP. For now the developed prototype already can serve as a replacement for Phantom OS TCP stack. I will not stop on this requirement in detail, as it already was discussed in the Section 5.1.

5.2.4 Compatibility with the existing protocols

This requirement, in essense was about using TCP or some other well-known protocol. This requirement holds in PTCP. The TCP stack uses TCP protocol and can be used by Genode components exactly as they use any other TCP stack.

5.3 Testing scenarios

I created two test cases that emulate behavior of a persistent system using PTCP. These cases are quite simplistic and only designed to check correctness of work of the developed software.

The first testcase checks that TCP sockets in various states are restored correctly. The program creates six TCP sockets. Two of these sockets remain in the closed state. The two of them are bound to local ports to be later used

in listen() function. The rest two are bound to addresses and then their states set to the listen state. All sockets are bound to the different ports.

After this initialization the test scenario remembers which socket has which persistent descriptor. Then the scenario saves the mapping to persistent memory and ensures that PTCP have made its snapshot. Then a restart is scheduled. After the restart the test case loads mapping between sockets and descriptors. At this point PTCP should have its state restored, because restoration takes place before passing control to client application. After the mapping is loaded the test case checks that each socket is in valid state. The checks consists of moving each socket to listen state with Berkeley socket API and ensuring that socket really listens by connecting to the machine from another machine in the same subnet. If any socket returns any error in process of verification or initialization, the test case considered to be failed.

The second test case is designed to check that PTCP can correctly process sending and receiving packets. This test case works similarly to the previous one – it also consists of initialization and verification phases separated by a restart. The initialization part includes setup of a listening socket. When the socket is open, a remote machine establishes connection to the PTCP client application and expects to receive 100 packets with a certain payload. After connection establishment the server machine sends packets. It performs snapshot after each packet is sent. When it sends 50th packet, the server machine shuts restarts. After the restart the test case on the server reads number of sent packets and proceeds sending the payload starting from 51th packet. The point of this test is to check the work of the Nic proxy mechanism. The fact that the old server socket does not exists anymore should not be noticed by the client process. Unfortunately, for now this test always fails because the proxy was not

implemented.

5.4 Future directions

PTCP only works with changing of state of only one machine. Therefore, the state of a connection can be restored only when the socket at the remote site exists. This limits a possible time when PTCP can restore sockets. In most cases the time that a host can be down is around of 30 seconds. This is quite small amount of time and. To make it larger PTCP should be a collaborative mechanism. Communicating PTCPs may employ exchange control messages, as proposed by [16], for example by using a separate UDP socket or ICMP messages.

Another issue that might be addressed is change of interface's IP address after restart. Throughout this paper, I discussed PTCP mechanism under the assumption that IP address does not changed. However, it is not always the case in the real world. In case of change of IP address the remote peer will not be able to consider newly created socket as a continuation of the previously existing one. One can implement a proxying NAT-like server similar to the Nic proxy discussed above. However, this server should live outside the persistent machine and have static address.

Chapter 6

Conclusion

During this thesis I have described and implemented the PTCP mechanism for providing an abstraction of persistent TCP sockets for the client applications. The mechanism was initially intended to be used together with the port of Phantom OS to Genode OS Framework. However, the mentioned port was not finished before this work. That is why I narrowed my scope to implement software that can be used with Genode only. I nevertheless believe that the PTCP can be easily integrated to the upcoming PoG port. It is used in the same way as any other Genode TCP stack therefore switching to it should not be a problem.

However, there are still limitations in the current version of the PTCP. It functionality can be improved, as described in the Chapter 3. It can include support for sockets in established state, as well as handling of longer shutdowns with some sort of cooperation mechanism.

Appendix A

Implementation

This section contains source code fragments used in discussion of implementation.

A.1 Example configuration for Genode component with networking capabilities

```
<start name="test" caps="100" priority="-1">
  <resource name="RAM" quantum="50M"/>
  <config>
    <vfs>
      <dir name="socket"> <ptcp dhcp="yes"/> </dir>
    </vfs>
    <libc socket="/socket"/>
  </config>
</start>
```

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] M. P. Atkinson and M. J. Jordan, *A review of the rationale and architectures of pjama-a durable, flexible, evolvable and scalable orthogonally persistent programming platform.* 2000.

[5] P. John, "Transmission control protocol," *RFC 793*, 1981.

[6] 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.

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

[8] R. Copeland, *Essential sqlalchemy.* " O'Reilly Media, Inc.", 2008.

[9] 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.

[10] 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.

[11] 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.

[12] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing tcp's retransmission timer," Tech. Rep., 2011.

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

[14] 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.

[15] 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.

[16] 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.

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