CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

Infrastructure for networked root filesystems of Linux-based embedded systems

Martin Škoudlil

May 2022
Supervisor: Ing. Michal Sojka, Ph.D.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Škoudlil**	Jméno: **Martin**	Osobní číslo: **492308**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Software**		

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Infrastruktura síťových kořenových souborových systémů pro Linuxové embedded systémy

Název bakalářské práce anglicky:

Infrastructure for networked root filesystems of Linux-based embedded systems

Pokyny pro vypracování:

Mounting the root file system over the network simplifies the development of Linux-based embedded systems. The NFS file system is well suited for this task, but running the NFS server requires root privileges to configure it and manipulate the files on the exported file system.
The goal of this work is to allow users without root privileges to control the NFS server and the files it serves and to build an efficient server infrastructure that facilitates embedded systems development.
1. Become familiar with the Linux network booting process and the novaboot tool.
2. Review userspace NFS servers such as unfs3 and Ganesha.
3. Extend the unfs3 server to be compatible with modern Linux systems (libtirpc) and integrate it with the server part of the novaboot tool.
4. Investigate the possibility of using NixOS to prepare cross-compiled NFS-mounted root file systems. This will require read-only NFS-mounting of the Nix store and extending the Nix daemon to support cross-compilation on the remote builder.
5. Evaluate the resulting infrastructure with Buildroot, Yocto, and NixOS distributions. Document the results thoroughly.

Seznam doporučené literatury:

1. Mounting the root filesystem via NFS (nfsroot):
https://www.kernel.org/doc/html/latest/admin-guide/nfs/nfsroot.html
2. Novaboot documentation:
https://github.com/wentasah/novaboot/blob/master/README.pod
3. Nix Manual: https://nixos.org/manual/nix/stable/
4. Nix Pills: https://nixos.org/guides/nix-pills/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Michal Sojka, Ph.D. vestavěné systémy CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **02.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

Ing. Michal Sojka, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgement / Declaration

Foremost, I would like to thank my supervisor Ing. Michal Sojka, Ph.D. for all the valuable information and the guidance he provided. I am also grateful for the moral support provided by my family.

I declare that I elaborated this thesis on my own and that I mentioned all the information sources that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating a final academic thesis.

In Prague 20.05.2022

. . . .

Abstrakt / Abstract

Tato bakalářská práce se zabývá použitím protokolu NFS pro připojování kořenových souborových systémů. Cílem je usnadnění vývoje větších vestavěných systémů založených na jádře Linux. K tomu je použit nástroj Novaboot, který integrujeme s NFS serverem běžícím v uživatelském prostoru (UNFS3). Výsledkem je možnost nabootovat cílový systém z lokálního obrazu, aniž by uživatel musel konfigurovat NFS, TFTP či DHCP servery.

Dále se v práci zabýváme Linuxovou distribucí NixOS a analyzujeme možnosti bootování této distribuce pomocí NFS a nástroje Novaboot. Zaměřujeme se na křížovou kompilaci a porovnáváme varianty vzdáleného kompilování potřebných balíčků.

Při testování jsme zjistili, že výsledný systém je použitelný, ale má drobné nedostatky.

Klíčová slova: bootování, embedded systémy, křížová kompilace, Novaboot, NFS, Nix remote builds, Nix daemon, NixOS, UNFS3

Překlad titulu: Infrastruktura síťových kořenových souborových systémů pro Linuxové embedded systémy

The aim of this bachelor thesis is the usage of the NFS protocol for mounting root filesystems. The goal is to facilitate the development of larger embedded systems based on the Linux kernel. To do this, we use the Novaboot tool, which we integrate with NFS server running in the userspace (UNFS3). The result is the ability to boot the target system from a local image without the user having to configure NFS, TFTP, or DHCP servers.

Furthermore, this thesis examines the Linux distribution NixOS and analyzes the possibilities of booting this distribution using NFS and the Novaboot tool. We focus on cross-compilation and compare options for remotely compiling the necessary packages.

In our testing, we found that the resulting system is usable but has minor shortcomings.

Keywords: booting, embedded systems, cross-compilation, Novaboot, NFS, Nix remote builds, Nix daemon, NixOS, UNFS3

Contents

Tables / Figures

Chapter 1
Introduction

Developing software for embedded devices is becoming more and more important as the Internet of Things devices gain popularity. This work will focus on the ones that are running Linux and their software is implemented primarily in userspace. Developers can download kernel sources, compile their own kernel, prepare the root filesystem, deploy it to the device, and boot the system on the device, all by themselves. This requires time and knowledge.

There are tools that simplify or even automate this process, such as Buildroot or Novaboot. Buildroot[1] can create simple boot images for many devices, but the booting and testing of them often require physical access to the device. Novaboot[2] is a tool that simplifies the deployment process and also allows you to deploy it to boards connected to the remote server. However, its main limitation is that out of the box it uses the TFTP protocol and loads the root file system into the RAM of the board. This inherits the problem that RAM is volatile storage, as each reboot will result in the loss of all changes made that were not backed up elsewhere. Novaboot can boot with NFS mounted root filesystem but requires extra configuration on the client-side, such as setting up the NFS server. This conflicts with the premise that if the server part of Novaboot is used, the clients can use Novaboot without any setup required. This work aims to extend Novaboot to allow access to the root filesystem over the network via the NFS protocol and management of its own NFS server.

Running operating systems with root filesystem mounted over NFS opens up the possibility of using more complex systems than what is provided by, e.g., Buildroot. Therefore, the other objective of this work is to investigate the possibility of using NixOS because it provides a way to manage multiple software stacks, even if some of them contain the same package but in a different version. NixOS also provides a way to offload builds onto a more powerful machine, which can be very useful because IoT devices are generally not very powerful.

[1] Buildroot – 2.2 or `https://buildroot.org/`
[2] Novaboot – 2.3 or `https://github.com/wentasah/novaboot`

Chapter 2
Background

This chapter introduces the requirements and shows existing programs and features that are used later in this work.

2.1 Requirements

These are the requirements for this work. Many of them were collected from my supervisor and some of them were created because this work would not be working without them.

R1: As a server administrator, I require that the NFS server can be configured without root privileges because giving users root privileges to users to configure the NFS server is a security risk.

R2: As a server administrator, I require that NFS support be enabled or disabled because I want to run the NFS servers only on some servers.

R3: As a user, I require that I can use Novaboot with the NFS server easier than I can currently while manually configuring everything.

R4: As a user, I require that the NFS server stores the ownership of files correctly, including root-owned files, as I intend to store the root filesystem in it and it contains different users.

R5: As a user, I require that I can cross-compile the NixOS image because the target devices, e.g., Raspberry Pi, are slow and I want to build the image on a fast machine with a different architecture, e.g., x86.

R6: As a user, I require that I have an easy way of booting NixOS with the root filesystem mounted over NFS because I do not want to configure it every time I boot the system.

R7: As a user, I want to remotely cross-build additional packages on NixOS booted over NFS, because the remote machine is faster and because the local device might not have enough resources, i.e., RAM, to build large packages.

2.2 Buildroot

"Buildroot[1] is a tool that simplifies and automates the process of building a complete Linux system for an embedded system, using cross-compilation" [1].

The user just needs to download the Buildroot repository, optionally select the default configuration from the list offered by running `make list-defconfigs` and apply it using `make <the-defconfig>`, then configure the build using `make menuconfig` and finally start the build by `make`.

Buildroot then builds a toolchain that will be used to build the system containing a bootloader, Linux kernel, and a root file system, depending on the configuration.

[1] `https://buildroot.org/`

2.3 Novaboot

"Novaboot[2] is a tool that automates the booting of operating systems on target hardware (typically embedded boards) or in Qemu" [2]. It is divided into the main (client) script and the server script. The main `novaboot` script is used by the client and controls the target device. The server-side part `novaboot-shell` is optional and is used to proxy all communication with the target hardware (2.1.C). Without the server-side part, the server can still be used to host boot images through TFTP (2.1.B).

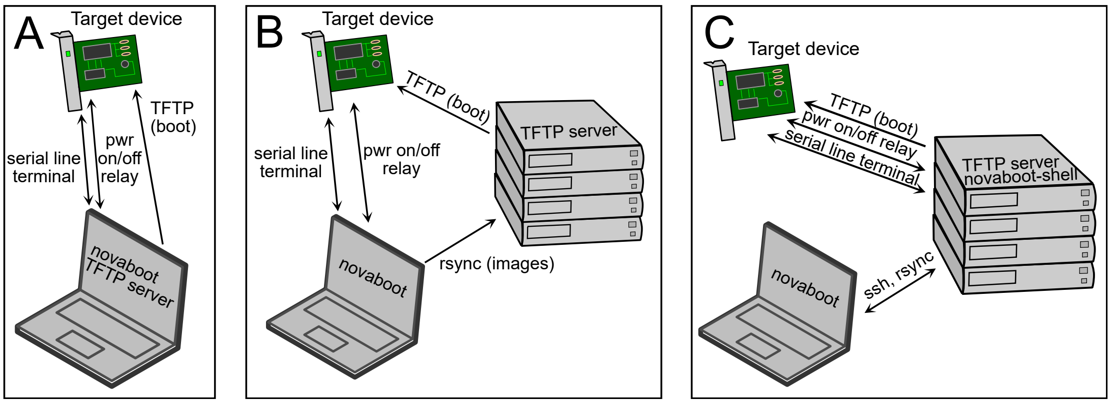


Figure 2.1. Typical Novaboot setups. Source [2]

2.3.1 The main script

The main script manages communication with the target hardware's bootloader either directly via serial line (2.1.A and 2.1.B) or through the server via SSH connection (2.1.C). Furthermore, it can toggle power to the device or reset it.

2.3.2 The server-side part

The `novaboot-shell` running on a server provides access to a device connected to said server and the device's configuration. When the server administrator configures the device, clients can connect to the server using the main script without the need to configure anything. Communication with the `novaboot-shell` is done through the SSH protocol.

It is possible to have multiple connected devices, each with a possibly different configuration. Each device has one designated Unix user on the server and the `novaboot-shell` runs as the Unix user's login shell. The client connected to the device is stored on the server as *Novaboot user*.

The `novaboot-shell` also provides a command `get-config` to provide configuration to the client `novaboot`.

2.4 Network File System

The Network File System protocol provides access to files shared across networks. The NFS protocol was initially developed by Sun Microsystems for their Unix system Solaris. The use of remote procedure calls (RPCs) allows the NFS protocol to be independent of operating systems, network architectures, and transport protocols [3]. The NFS protocol is often used in versions 3 and 4.

[2] `https://github.com/wentasah/novaboot`

The NFSv3 [4] is a stateless protocol provided over TCP or UDP and requires a supplementary MOUNT protocol to allow clients to attach directory trees to a specific point in their local file system.

The NFSv4 [5] is, in contrast, stateful and is only provided via TCP. It does not need the MOUNT protocol, as the NFS and MOUNT protocols are combined together into the single NFSv4 protocol. This version also provides a way to chain operations in a single request or delegate access to a file to a specific client.

There are multiple implementations of the NFS protocol available. We provide their overview in the following subsections.

2.4.1 Kernel NFS server

The simplest method to use NFS on Linux is to use the Kernel NFS server, which is already built into many popular Linux distributions, such as Ubuntu. The kernel NFS server is usually a kernel module but it can also be compiled directly into the kernel. The kernel NFS server supports many versions including the NFSv3 and NFSv4.

The server can be configured from the user space using the `/etc/exports` file or the `exportfs` command. Each line in the `exports` file contains a directory to be exported and clients that can access it. The client can be appended with options. The most notable options are the `ro` or `rw` which provide the client with read-only or read-write access, respectively, to the directory.

From the perspective of this work, the main drawback is that the kernel NFS server requires root privileges to export files and directories and provides only limited options for the mapping of user identifiers. They can be changed in the `exports` file via `<...>_squash` options.

2.4.2 User-Space NFSv3 Server

The User-Space NFSv3 Server (or UNFS3)[3] is a userspace implementation of the NFSv3 protocol. It is primarily developed and tested on Linux but should also work on other Unix systems [6]. The shared directories are specified in the `/etc/exports` file with the same rules as for the kernel NFS server but in contrast with the kernel NFS server, the UNFS3 does not support file locking so the client may have to mount with locking disabled [7]. UNFS3 should be started as a root for optimal experience but does also support running as an unprivileged user.

When stated as root, UNFS3 leaves file permission checks on the operating system by calling `seteuid/setegid` with the id provided by a client. However, when the UNFS3 is started as an unprivileged user, it manages all files as that user.

UNFS3 also allows multiple instances to run at the same time. This must be done by specifying different NFS and MOUNT ports for each instance or using the -u option, which assigns free random ports to the server. However, only one of the instances can be registered with rpcbind discovery service.

The implementation of UNFS3 is simple, so it would allow us to make changes or implement new features.

Currently, UNFS3 does not compile with newer glibc (2.26+), because it dropped support for Sun RPC and UNFS3 has to be rewritten to use the libtirpc library.

[3] `https://github.com/unfs3/unfs3`

2.4.3 NFS Ganesha

NFS-Ganesha[4] is another userspace server that uses the NFS version 3, 4, and 4.1 protocol in addition to 9P protocol. The main feature is that it supports many different storage mechanisms, such as VFS for exporting the same file system, the kernel NFS server is able to export, or distributed CEPH or GLUSTER, or can act as a proxy for different NFS servers. [8]

The NFS-Ganesha is configured through the `/etc/ganesha/ganesha.conf` configuration file which has different format than `/etc/exports` file. The configuration file consists of several blocks, each for a different filesystem. Since the documentation does not provide an example configuration it will take time to write even a working configuration. Also, some aspects can be configured in multiple places so it is hard to determine which one is active.

2.5 Faking file permissions

Neither UNFS3 nor NFS Ganesha natively supports running as an unprivileged user and retaining the correct file ownership information. This section presents some options on how to create a fake environment in which the NFS server thinks it has root privileges and can store correct ownership information.

2.5.1 Fakeroot

Fakeroot[5] runs a command in an environment where it appears to have root privileges for file manipulation [9]. It was developed to allow users to create Debian packages without the need to be root. Files created under the Fakeroot environment are saved on the file system owned by the user running the Fakeroot (actual owner). However, inside the Fakeroot environment, their ownership is managed by Fakeroot and can be any arbitrary user.

Fakeroot achieves this by using the `LD_PRELOAD` mechanism of the dynamic loader to wrap the file manipulation library functions. This means that Fakeroot cannot work with statically linked binaries.

Fakeroot lets us save the state of the faked environment so that it can later be restored later.

The main drawback for us is that Fakeroot does not enforce any permission checks on file operations inside the faked environment. This means that the faked unprivileged user can modify or even take the ownership of files owned by the fake root. Using Fakeroot with UNFS3 to serve root file systems would mean, that the non-root users on the target system could modify files owned by the root.

2.5.2 Fakeroot NG

Fakeroot NG[6] is very similar to Fakeroot but differs in that it intercepts syscalls instead of library calls. Fakeroot NG mocks system calls using `ptrace` system call. This allows Fakeroot NG to work with statically linked binaries, but it is slower [10].

[4] `https://github.com/nfs-ganesha/nfs-ganesha`
[5] `https://salsa.debian.org/clint/fakeroot`
[6] `https://fakeroot-ng.lingnu.com/`

2.5.3 User namespaces

User namespaces are a Linux kernel feature available from version 3.8 and allow us to isolate security-related identifiers and attributes, such as user or group identifiers (UIDs or GIDs, respectively) or capabilities. The primary usage of user namespaces is the separation among containers and from their host machine by docker or other container engines.

User namespaces can be created using `clone()` or `unshare()` system calls with the flag `CLONE_NEWUSER`.

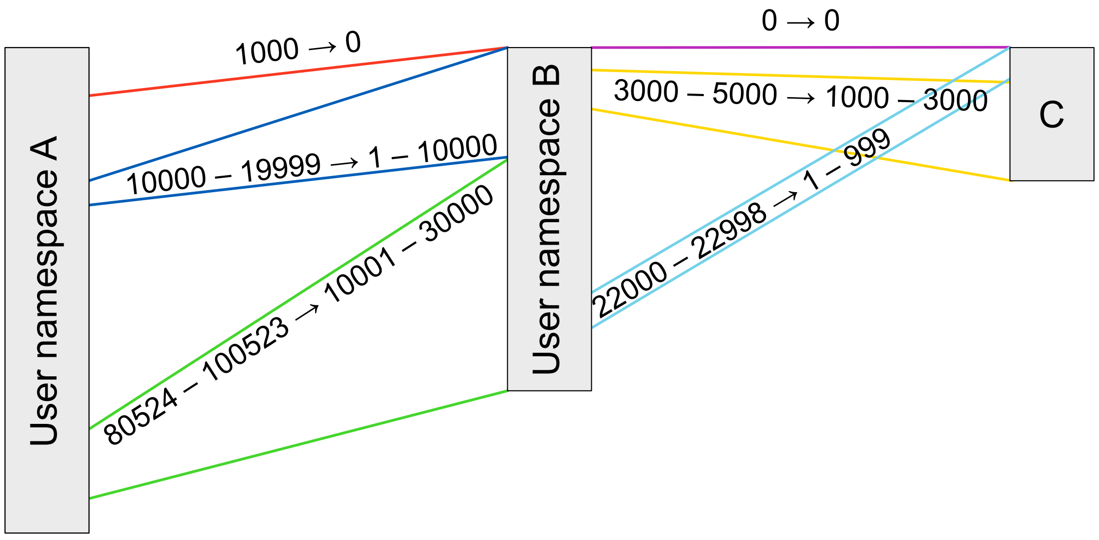


Figure 2.2. Example mapping by user namespaces.

Each ID (UID or GID) in a user namespace must be mapped to a unique ID in its parent user namespace. This can be seen in the image – for example, ID 50 in namespace C is equivalent to ID 22 049 in namespace B and 92 572 in namespace A.

2.6 Systemd services

"systemd is a suite of basic building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system." [11]

A systemd unit refers to any resource that the system knows how to operate on and manage, e.g., service, socket, device, mount. The resource is defined by a configuration file called *unit file*.

A systemd service is a type of systemd unit. The service is responsible for managing a program specified in the configuration – file ending with `.service` and must be located in predefined locations [12]. The service can be started either automatically on boot, user login or by other services or manually through `systemctl` command.

If the service name ends with `@`, it takes one argument that can be accessed inside the configuration file with `%i` specifier. The full name of the service is then `name@arg.service`.

2.7 Nix and NixOS

Nix[7] is a purely functional package manager. That means that each package is built by a pure function called Nix expression, and once they are built, they cannot change. Nix stores components (we will call them *packages*) in the Nix store [13].

NixOS is a Linux distribution based on the Nix package manager. All components of the distribution (kernel, installed packages, and system configuration files) are built by Nix from Nix expressions.

NixOS aims to be fully reproducible – all changes to system configuration should be done declaratively in `/etc/nixos/configuration.nix` file or in its imports. With the same configuration, anyone can build the exact same system [14].

2.7.1 Nix store

The Nix store is usually located in `/nix/store` directory, where each package has its own subdirectory, such as `/nix/store/b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z-firefox-33.1/`. The `b6gv...` hash is calculated using all its static dependencies, also called build inputs [13, 15].

The Nix store is managed by so-called *garbage roots*. The roots are links that point to the packages used. Nix does not automatically remove packages once they are no longer needed. Instead, it is required to start the garbage collection operation.

2.7.2 Nixpkgs

The Nix Packages collection (Nixpkgs) is a set of over 80,000 packages for Nix package manager [16]. Nixpkgs contains functions that can source and build these packages and also contains targets to build NixOS. Nixpkgs will be modified later in this work to create a custom NixOS system (see 3.5) or to simplify remote building on systems with different architecture (see 4.5.2).

2.7.3 Nix configuration file

The `nix.conf` file is located in `/etc/nix/` directory on most systems [17]. The file can only be changed directly if the user is not on NixOS and is using Nix package manager on other operating system. On NixOS it must be configured in `configuration.nix` file or one of its dependencies by setting `nix.<nameOfConfigInCamelCase>` to the requested value when using NixOS 21.11 or older or `nix.settings.<name-of-config-in-kebab-case>` for newer[8] versions of NixOS.

2.7.4 Remote Builds

Because each build is well defined in terms of its inputs and build steps, the build will always have the same output if built on a machine with the same platform (see 2.7.6). Nix thus allows us to offload builds from, e.g., a slow laptop or split them among powerful servers [13].

In order to perform the build on a remote machine, Nix uploads all of the build dependencies to the build server's Nix store, then performs the build and downloads the result back to the clients' Nix store.

To use remote builds, the user needs to have ssh access to the remote machine and be a trusted user on the remote machine. If the remote machine is specified as a builder

[7] `https://nixos.org/`

[8] At the time of writting that means only unstable 22.05pre version.

in `nix.conf` (2.7.3), it will automatically be used to offload parts of the build. Another way is to specify the builder as an option `--builders 'ssh://remote-machine'` in the build command if the local user is trusted [18].

2.7.5 Nix daemon

Nix daemon is an essential part of multi-user installations of Nix. The daemon is used automatically by other nix commands to allow non-root users to perform builds or other operations on the Nix store. Usually, communication with the daemon is done using a unix socket in `/nix/var/nix/daemon-socket/socket` but can be changed by setting the `NIX_REMOTE` environment variable. Later it the work the nix demon will be used to connect to a remote machine using the ssh-ng protocol, which supports the same actions as the Nix daemon (see 3.4.4).

2.7.6 Cross compilation

Nix also provides mechanisms to allow cross-compilation to other platforms. Internally Nix uses 3 different platforms: build, host and target, but provides convenient access for only two, the build stored as `system` or `localSystem` and the host specified by `crossSystem`. The build platform represents the platform that will create a package, the host will be able to use the package, and the package will then be able to create outputs for the target platform.

Cross-compilation can start in many ways. When using `nix-build` or `nix build` the host platform can be specified like `--argstr system x86_86-linux` and defaults to the same platform on which the command is run. The host platform can be specified as `--argstr crossSystem aarch64-multiplatform` and is the same as the build platform by default.

2.7.7 Nix booting stages

The NixOS separated the booting process into two stages, stage 1 and stage 2, each with its own init script.

The stage 1 init script is run from initrd and is used to mount the filesystem.

The stage 2 init script is located in the filesystem, manages all other setups that need to be done, and finishes with starting systemd.

Chapter 3
Analysis

In this chapter, we analyze the usability of the NFS server options, analyze the creation of user namespaces, check what needs to be changed in Novaboot-shell during integration with UNFS3 and compare possible options for NixOS' remote builds and its cross-compilation.

3.1 Choosing NFS server

In this section, we determine which of the NFS server solutions from the following subsections suits our needs.

3.1.1 Kernel NFS server

Pros: The kernel server is a good choice for simple setups, such as when you are hosting the root filesystem on your machine.

Cons: The problem occurs when you try to host the filesystem on the public server, as you often do not have permissions to configure the kernel NFS server and as per requirement R1 the server must be configurable without root privileges.

Even if the server administrator configured the exported folder for you, it can be a huge security vulnerability, as you could use `S_ISUID` permission bit on files created through NFS export and impersonate other users. This can be prevented by specifying the `all_squash` option, but all files inside the export will be owned by one `UID:GID` pair. This is a very large limitation, as this pair cannot be `root:root` otherwise we would still have the `S_ISUID` vulnerability.

3.1.2 NFS Ganesha

The Ganesha is great if we wanted to use distributed systems, but it would not provide us with any benefits in the context of this work. As NFS Ganesha is also harder to configure, we will not use it.

3.1.3 UNFS3

UNFS3 on its own behaves similarly to the kernel NFS server.

Pros: UNFS3 does not need to be started as root to export the filesystem with `all_squash` or if the filesystem does not need to make changes to files not owned by the user running the server, e.g., read-only Nix store (see TODO link).

The source code for UNFS3 is available on github[1] with a license that allows modifications.

Cons: Running UNFS3 unprivileged prevents the NFS Client from creating or making changes to the root-owned files.

[1] `https://github.com/unfs3/unfs3/`

3.1.4 UNFS3 with a Database

We can implement a database to store file ownership and permission information. The user running UNFS3 will own all files on the server, but the client that accesses the NFS storage will see them with ownership and permissions stored in the database. This will also require implementing the permission check in UNFS3, as currently, UNFS3 can offload them onto the Linux kernel as the permissions are the same on the server as the client.

3.1.5 Wrapped UNFS3

Because UNFS3 is a userspace program, it can be started in the custom environment that alters the permissions and / or ownership it sees. These environments are considered:

- Fakeroot (Section 2.5.1)
- Fakeroot NG (Section 2.5.2)
- User namespaces (Section 2.5.3)

Fakeroot or Fakeroot NG would be a good choice, as they store files as a single user on the server but do not enforce permission checks (see Section 2.5.1).

Another option is to run UNFS3 in the user namespace. Namespaces require that each user is mapped to a unique user in the parent namespace. An unprivileged user can only create a user namespace with a single user – current user mapped to the root, which provides the same functionality as `all_squash` but all files will appear as owned by the root. There are ways to create a namespace with more users as discussed next in the Section 3.2.

3.1.6 Final choice

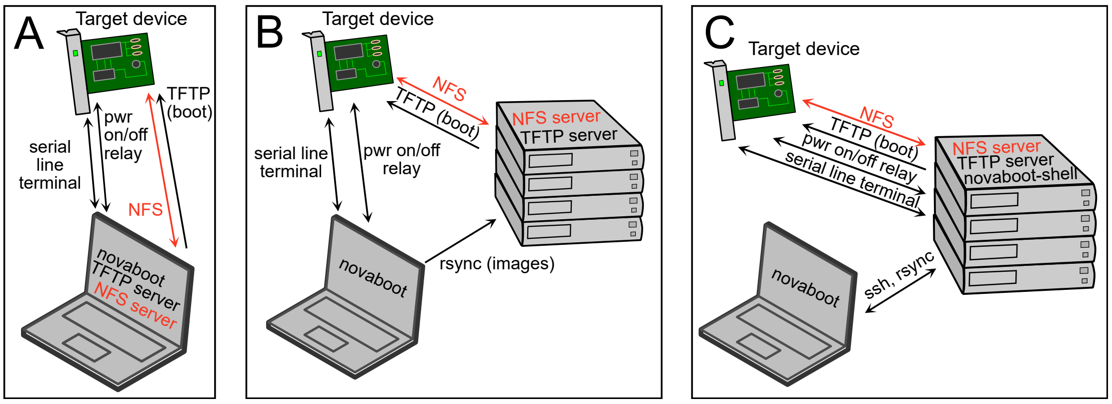


Figure 3.1. Novaboot setups with NFS server.

For the setup 3.1.A, I recommend using the kernel NFS server as you already have root access to your machine.

For the setups 3.1.B and 3.1.C, UNFS3 with the database would be a great choice, but UNFS3 inside the user namespace would behave nearly identically if given some large enough range of unused UIDs and GIDs. So using the namespace is a better option, as extending UNFS3 with a database requires time to implement.

3.2 Creation of user namespaces

If we use the user namespaces, the NFS server can run without root privileges but still provide the NFS clients ability to manipulate root-owned files. Creating user namespaces that contain more than one user requires root privileges. However, there are two options for an unprivileged user to create such namespace.

The first option is the usage of `newuidmap` and `newgidmap` from a widely spread package *shadow-utils*[2] which manages accounts and password files. These commands check if the requested namespace mapping is from a range of IDs assigned to the user and create the mapping for them. The assigned ranges are configured in the files `/etc/subuid` and `subgid`.

Another option is to create a program that will be owned by root and have `S_ISUID` permission flag, which will cause the program to start as root even when started as unprivileged. The program will then take the mapping from a trusted source and start specified command in a new user namespace according to the mapping.

We choose the first option, as the security is left on the package. However, we will implement a tool that will simplify the creation of the user namespace with the commands `newuidmap` and `newgidmap`.

3.3 Novaboot server with NFS

In this section, we focus on:

- toggling of the NFS support in Novaboot-shell,
- the integration of the NFS server with Novaboot-shell in order to serve the root filesystem over NFS, and
- copying of the root filesystem to the server.

This section assumes the setup with the novaboot server depicted in Fig. 3.1.C and will explain what needs to be considered to extend Novaboot with the automated NFS server functionality.

3.3.1 Enabling and disabling the NFS support

In order for the NFS support to be toggle-able as per requirement R2, we check for presence of a specific configuration file `.novaboot-nfs` (see Section 4.3.1). NFS support is only active if the file is present.

3.3.2 Integrating UNFS3 with novaboot-shell

The device needs to know the address, port, and exported path of the NFS server to mount the filesystem. Luckily, the `novaboot-shell` already provides a mechanism to pass the configuration to the main novaboot script – the command `get-config`. We could provide the NFS configuration in a single entry since in most cases, the user will mount the root file system in the kernel as `nfsroot=address:/path,options`, but we opt to provide it separately with two entries (`--nfsroot` and `--nfsopts`). This lets us use them in the `mount` call because it requires the options to be provided separately in the `-o` argument.

For a successful integration with Novaboot, we need to run multiple instances of UNFS3 at the same time. This is because the `get-config` command can be run without

[2] `https://github.com/shadow-maint/shadow`

the Novaboot user having exclusive access to the device, and thus multiple users can use in at the same time. Running multiple instances allows us to easily separate the data of each Novaboot user, without the need to change the `exports` file during the NFS server's runtime.

UNFS3 is a daemon that forks in the background, and we run it as a service under the systemd user instance (see Section 2.6) of the unix user assigned to the device. The accessing Novaboot user will be passed in the service argument. This approach also benefits from the ability to stop all instances of UNFS3 when the device is not used.

3.3.3 Copying root filesystems to Novaboot server

To use the files on the board, the user first needs to get them on the server hosting the NFS root. These are some options:

- access all files using rsync
- access all files using NFS
- upload tar and unpack it on the server

The first option is rsync; since it is already used to upload files to be accessed through the TFTP server, it would seem like a great option. However, rsync should be restricted only to the NFS share, and if the NFS server is run in a user namespace, the rsync also has to run in it.

The second option is to allow access to files through NFS; it would require the NFS server to accept connections anywhere from the Internet. This can be overcome by exporting the share only to localhost and *local forwarding* the NFS connection through SSH.

The last and the chosen option is the most restrictive since it does not allow the user to download the filesystem back, and if the user needs to make changes, they need to upload the modified tar to replace the whole filesystem. We choose this option because it is the easiest to implement and does not have security issues.

3.4 NixOS

This section introduces why we chose NixOS as the operating system for the development boards and we show how to boot it via NFS and how to use remote builds.

3.4.1 Background

We chose NixOS because it supports declarative and reproducible system configuration, and thus building from the same configuration always produces the same system. NixOS also supports having multiple versions of the same package, which is useful for systems with complex software stacks.

3.4.2 NixOS through NFS

Currently, NixOS can be configured to have its root filesystem hosted through NFS in two ways:

- in `configuration.nix` using the `fileSystems` attribute,
- by specifying `nfsroot` parameter on the kernel command line and skipping the filesystem mounting phase in NixOS stage 1.

The first solution aligns with Nix's ideology of a declarative system. However, the system requires rebuilding each time something changes in the configuration of the NFS, e.g., the server's IP address or in our case more importantly, the port number.

The second solution requires that the kernel is compiled with the support for mounting a root filesystem over NFS.

My proposed option is a combination of both solutions. We can modify stage 1 init script to support new command line arguments, most notably `nfsPrefix` and `nfsOptions`.

3.4.3 Remote builds

When we want to build large packages (e.g. Firefox, LLVM, rustc) for an embedded device that does not have enough memory or computation power, we need to use the remote builds.

The first option is to use standard remote builds, which use dependencies available locally and copy them to the build server. After the build of each package is done on the server they are also copied back to the local device.

The main drawback is the inefficiency in using the Nix store, which needs to copy files between the device's store and the server's store over the network when, in fact, the device's store is hosted on the server. We want to investigate whether there is a better option.

The next option is the usage of the Nix daemon, which is discussed next in Section 3.4.4.

3.4.4 NixOS – remote daemon

Because we host the root filesystem with the Nix store on the same server as we do remote build, there is a lot of unnecessary copying of files over the network, as seen in Fig. 3.2 when the files could be accessed locally on the server. This can be solved by the Nix daemon.

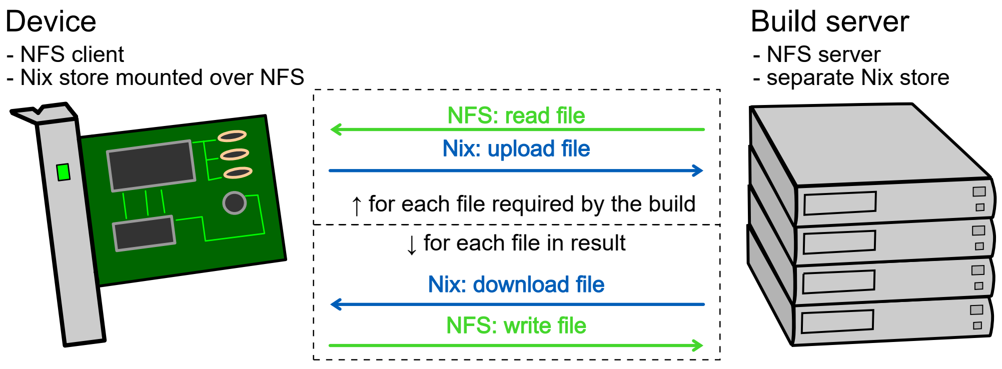


Figure 3.2. Showcase of redundant file accesses during remote build with NFS.

Reducing file transfers can be done using the Nix daemon. If the Nix store of the build server and the device is the same and the device has access to the server's Nix daemon, the Nix daemon accesses the files locally, and the device only transfers files that it actually uses.

However, it comes with a limitation: The build server does not know which paths in the Nix store are still being used by the device. These paths could then be garbage collected because the server thinks that they are not used. The solution is to manually create garbage roots for them on the server.

3.4.5 Remote cross-builds

Sometimes we want the remote machine to have a different architecture than the device (Requirement R7). But the remote machine can only build packages for its own platform.

The solution is to request the cross-compiled version of the package. This can be done manually for each build, or we can change the default values in Nixpkgs.

3.5 Novaboot and NixOS

NixOS has several Nix expressions for building bootable images but none of them can be easily used with Novaboot. Closest to our target are tarballs designated for network booting, but they rely on manual modification of the system after it is built. However, they can be used as a template for the new Novaboot image.

The Novaboot image is configured to boot over NFS and contains all necessary files.

Chapter 4
Implementation

In this chapter we will implement changes to UNFS3 and how to integrate with it with `Novaboot-shell`, create `userns` tool to help with creating user namespaces, show how to use remote builds with remote Nix daemon and how to create a working image of NixOS that can be used with Novaboot.

4.1 Changes to UNFS3

This section focuses on compiling UNFS3 with a new glibc library, running multiple instances of UNFS3, or fixing changes to the modification time on a symlink.

As mentioned in Section 2.4.2, the UNFS3 does not currently compile with newer glibc. There is an attempt to port the program to use the libtirpc library, but it contains several bugs that prevented UNFS3 from running correctly. These bugs include missing calls to `listen()` after creating TCP sockets, using old functions that do not account for the IPv6 protocol, or not creating entries in rpcbind. I address them in my pull request[1] which is now merged into master.

4.1.1 Port file

The next change is to allow multiple instances to be run at the same time. It is necessary to improve operation without the rpcbind service as the service can only store information for one instance at a time. Because the rpcbind service is responsible for providing port numbers, another method is required to extract the port numbers. We solve this by introducing a new option `-P <file>` which creates the file listing all ports used (example below in Listing 4.1.1). The syntax of the file allows it to be sourced by a shell script or used in the `EnvironmentFile` directive in systemd units.

```
NFS_UDP=52460
NFS_TCP=54172
MOUNT_UDP=52460
MOUNT_TCP=54172
```

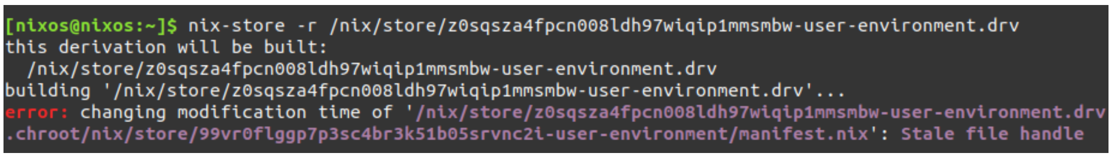
Listing 4.1.1. Example port file created by option `-P <file>` in UNFS3.

4.1.2 Modifying time on symlink

When the client tries to change the time on a symbolic link, the error depicted in Fig. 4.1 happens. This is because the UNFS3 changes time with `utime()`. This function cannot change the time on the link and instead dereferences the link. However, the link is valid only in the client system and not on the server, and thus `utime()` fails with *"No such file"* error, and UNFS3 returns *"Stale file handle"*.

The problem has been solved by replacing the `utime()` function with `lutimes()`.[2]

[1] UNFS3 PR#20 `https://github.com/unfs3/unfs3/pull/20`
[2] `https://github.com/skoudmar/unfs3/commit/9f16aa438337548a3039f45099bce5f9f44eb3ba`



```
[nixos@nixos:~]$ nix-store -r /nix/store/z0sqsza4fpcn008ldh97wiqip1mmsmbw-user-environment.drv
this derivation will be built:
  /nix/store/z0sqsza4fpcn008ldh97wiqip1mmsmbw-user-environment.drv
building '/nix/store/z0sqsza4fpcn008ldh97wiqip1mmsmbw-user-environment.drv'...
error: changing modification time of '/nix/store/z0sqsza4fpcn008ldh97wiqip1mmsmbw-user-environment.drv
.chroot/nix/store/99vr0flggp7p3sc4br3k51b05srvnc2i-user-environment/manifest.nix': Stale file handle
```

Figure 4.1. Error state file handle when changing modification time on a symlink in NixOS.

4.2 User namespace creator

User namespace creator or `userns` is a tool that simplifies the creation of user namespaces with commands `newuidmap` or `newgidmap`. The tool will request the maximum size mapping that is available according to the first range of identifiers available to the user in the `/etc/subuid` or `/etc/subgid` files.

The tool is used as `userns <command>` where the command is a list of arguments that will be passed to `execvp`, the first being the name of the requested program. An example use of the program is shown below in Listing 4.2.1.

The source code for the tool can be found at Gitlab repository A.3.

```
$ id
uid=1000(martin) gid=1000(martin)

$ userns id
uid=0(root) gid=0(root)

$ userns chmod +x,-w script
```

Listing 4.2.1. Example usage of the `userns` tool.

4.3 Integration of UNFS3 with Novaboot-shell

This section describes changes to the `novaboot-shell` to allow it to manage the UNFS3 and allow Novaboot users to host their root filesystem.

4.3.1 Novaboot NFS configuration file

The entries of this configuration file `.novaboot-nfs` are used to generate `exports` file for NFS server and to tell the server's address to the device in `get-config`. The presence of this file in the home directory of the unix user enables the NFS support for the device.

The file consists of lines that contain variable assignments. Empty lines or lines starting with `#` are ignored and can be used to comment on the configuration file. Other lines have form of `<name>=<value>`.

In order to correctly create the `exports` file used by UNFS3 to allow the Novaboot target device to access the NFS server, Novaboot-shell needs to know the device's address – it has to be configured in the variable named `allowed_clients`. The entry accepts IP address, hostname, or network. Although for security reasons only one device should be matched, allowing the entire network might be needed, e.g., if the device obtains a dynamic IP address through DHCP.

The entry `server_addr` makes it possible for the device to know the IP address of the server as the device is sending the mount request to it. The server passes this

```
allowed_clients=172.17.0.0/24
server_addr=172.17.0.2
```

Listing 4.3.1. Example of `.novaboot-nfs` configuration file.

information to the Novaboot client via the `get-config` command, the entry accepts anything that can be correctly resolved by the device. However, if the Linux kernel is used to mount the root filesystem, `server_addr` should contain an IPv4 address [19].

4.3.2 UNFS3 systemd service

An execution of the UNFS3 server and the creation of its `exports` file are managed with the systemd service – called `novaboot-unfsd@.service`. It it considered fatal error to start this service without the `.novaboot-nfs` (see Section 4.3.1) file present since the created `exports` file needs to know the `allowed_clients` value.

The service can be started using the command shown in Listing 4.3.2. The unit file for the service is seen in Listing 4.3.3.

```
systemctl --user start novaboot-unfsd@<username>.service
```

Listing 4.3.2. Command to start the UNFS3 systemd service

4.3.3 The core integration of Novaboot-shell with UNFS3

The Novaboot-shell needs to start UNFS3 and then extract the ports used.

In view of the fact that we need to pass the NFS configuration through the command `get-config` which allows the novaboot client to learn the configuration needed to boot the target device properly, and the fact that the command `get-config` is the first command called by the main Novaboot script as seen in Figure 4.2, the UNFS3 server must start there. This is achieved using the command in Listing 4.3.2.

Once UNFS3 is started it will create a file containing used ports as seen in Listing 4.1.1. The ports with other options are then returned in the response of the `get-config` command as the `--nfsopts` switch to the main Novaboot client. The server address from `server_addr` in `.novaboot-nfs` file is then concatenated with exported path and returned as `--nfsroot` switch. An example response of the novaboot-shell to the get-config command is shown in Listing 4.3.4.

```
--nfsroot=10.0.0.1:/home/rpi/nfsroot/<username>/root
--nfsopts=v3,tcp,port=54172,mountport=54172
```

Listing 4.3.4. New switches from `get-config`

```
[Unit]
Description=UNFS3 daemon running as user %u and NB_USER %i
AssertPathExists=%h/.novaboot-nfs

[Service]
Type=forking
EnvironmentFile=%h/.novaboot-nfs
Environment="NB_UNFSD_EXPORT_FILE=%h/.cache/unfsd/%i.export"
Environment="NB_UNFSD_PORT_FILE=%h/.cache/unfsd/%i.ports"
Environment="NB_NFS_EXPORT_PATH=%h/nfsroot/%i/root"

ExecStartPre=mkdir -p %h/.cache/unfsd %h/nfsroot/%i/root

# generate an export file for the unfsd
ExecStartPre=sh -c 'echo "\
# generated by novaboot-unfsd@.service\n\
$NB_NFS_EXPORT_PATH/  ${allowed_clients:?}(rw,no_root_squash,insecure)\
" > $NB_UNFSD_EXPORT_FILE'

# Start unfsd
#   -p: do not register with portmapper
#   -u: use random ports
ExecStart=unfsd -p -u -e ${NB_UNFSD_EXPORT_FILE} -P ${NB_UNFSD_PORT_FILE}

# Remove the export and port file when the service is stopped
ExecStopPost=rm -f ${NB_UNFSD_EXPORT_FILE} ${NB_UNFSD_PORT_FILE}
```

Listing 4.3.3. Unit file for `novaboot-unfsd@.service`.

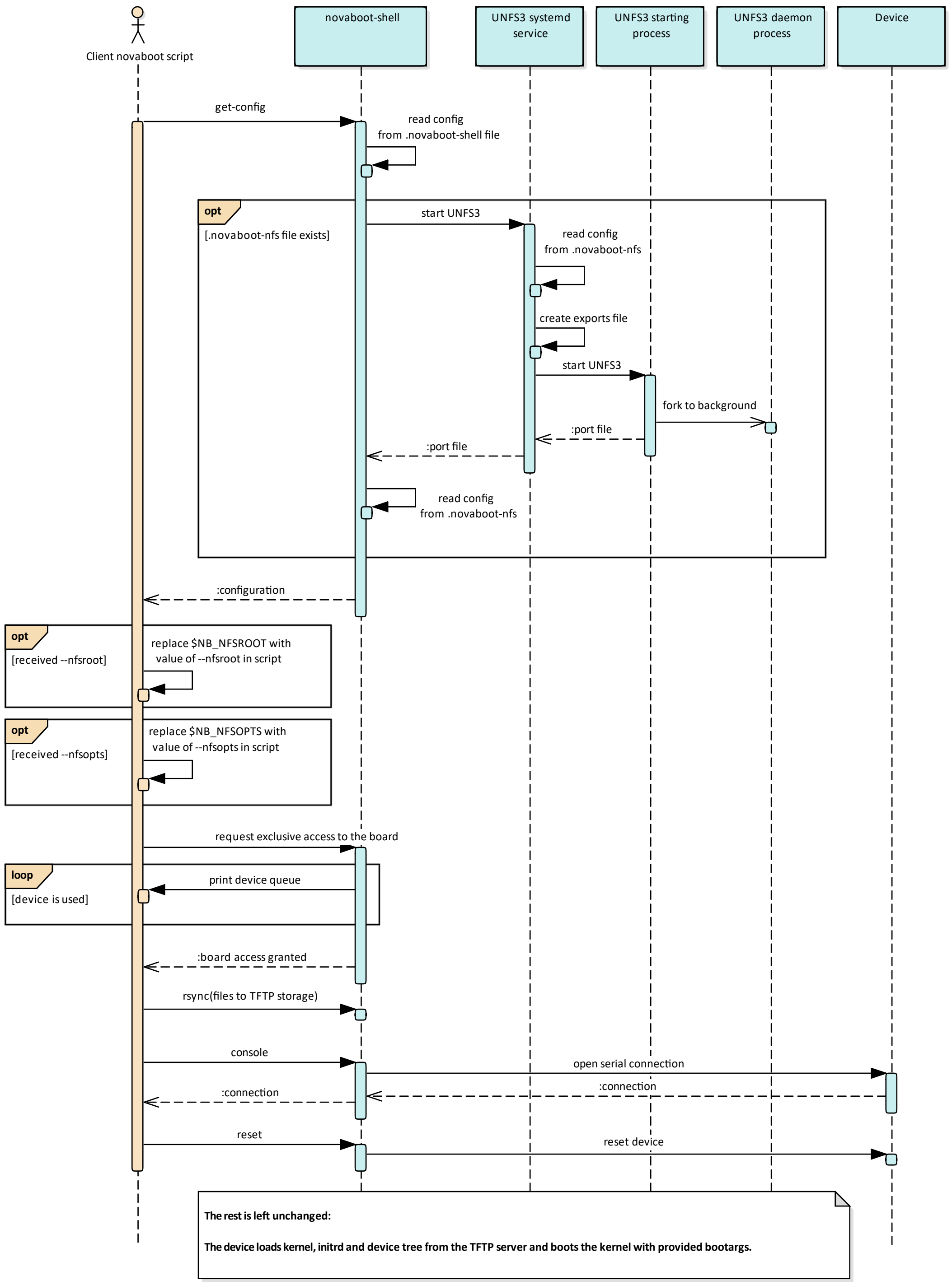

Client novaboot script
novaboot-shell
UNFS3 systemd service
UNFS3 starting process
UNFS3 daemon process
Device
get-config
read config from .novaboot-shell file
opt
[.novaboot-nfs file exists]
start UNFS3
read config from .novaboot-nfs
create exports file
start UNFS3
fork to background
:port file
:port file
read config from .novaboot-nfs
:configuration
opt
[received --nfsroot]
replace $NB_NFSROOT with value of --nfsroot in script
opt
[received --nfsopts]
replace $NB_NFSOPTS with value of --nfsopts in script
request exclusive access to the board
loop
[device is used]
print device queue
:board access granted
rsync(files to TFTP storage)
console
open serial connection
:connection
:connection
reset
reset device
The rest is left unchanged:
The device loads kernel, initrd and device tree from the TFTP server and boots the kernel with provided bootargs.

Figure 4.2. `novaboot-shell` communication sequence diagram.

4.4 NixOS image for Novaboot

The goal is to create a NixOS image that can be directly booted with Novaboot. To do that, we need to create a new Nix expression that builds an image, which:

- can easily be deployed to the Novaboot server,
- have its root filesystem mounted over NFS,
- reads the information about the NFS server and options from the kernel command line, and
- contains a pre-configured Novaboot script with its dependencies.

Because NixOS is only officially supported on a handful of embedded devices such as Raspberry Pi[3], I have chosen to create a target for the Raspberry Pi 4 board. Modifying it should be easy enough to support other devices in the future.

4.4.1 Modifying the NixOS' stage 1 init

This subsection will describes the implementation of kernel command line arguments `nfsPrefix` and `nfsOptions` providing NFS server configuration without the need to rebuild the NixOS image (see Section 3.4.2).

- `nfsPrefix` is used as a prefix to the path provided in the attribute `fileSystems` and allows us to provide the address and even part of the path.
- `nfsOptions` is a comma-separated list of options to be used during mounting that is appended to the options provided in the `fileSystems` attribute.

In NixOS the *stage 1 init* script is stored in the initial RAM file system image (initrd) manages parsing of the kernel command line arguments and mounting of the filesystem.

We extend the script in such way that while reading the kernel command line, if the script finds the `nfsPrefix` or `nfsOptions` it will save it to a variable. During the mounting phase, if it tries to mount the NFS filesystem, it will apply these variables if they are set and then mounts the filesystem.

The exact changes can be seen at this link[4] or in my Nixpkgs repository A.5.

```
fileSystems."/" = {
    device = "//root/";
    fsType = "nfs";
    options = [ "v3 "tcp" ];
};
```

Listing 4.4.1. Example NixOS configuration specifying that root filesystem will be mounted through NFS. In Novaboot NixOS image this generated internally (see Section 4.4.2).

If the Linux kernel is booted with the following options (perhaps passed by the novaboot client script):

```
nfsPrefix=10.0.0.1:/home/martin nfsOptions=port=54172,mountport=54172
```

Listing 4.4.2. Example usage of `nfsPefix` and `nfsOptions`.

[3] Popular embedded device capable of running Linux.
[4] `https://github.com/skoudmar/nixpkgs/commit/312236bbe4838785028cf5d43e00f705687717c5`

The stage 1 script will merge configuration from Listing 4.4.1 and options from Listing 4.4.2 and mount the NFS root file system from `10.0.0.1:/home/martin//root/` with options `v3,tcp,port=54172,mountport=54172`

4.4.2 Creating the filesystems entry for NixOS

This subsection will describe how the NixOS expression for Novaboot tarball creates filesystems configuration.

By default the `fileSystems` entry is created that the NixOS expects the `nfsPrefix` to provide full path and the `nfsOptions` to set all necessary options.

When the user specifies the filesystem manually as in Listing 4.4.1 the must not start with single '/' otherwise the stage 1 script will prefix the path with `/mnt-root`. Because of that the new the configuration option `novaboot.nfs.server.rootPath` correctly parses the paths staring with single '/'.

Mounting options can be specified in `novaboot.nfs.server.options`. There are more configuration options available in the file `nixos/modules/installer/cd-dvd/system-tarball-novaboot.nix`.[5]

4.4.3 Cross-compiling and using the image

The created Nix expression to build the tar can be found in the Nixpkgs repository A.5 in the file `nixos/modules/installer/cd-dvd/system-tarball-novaboot.nix`.[6] The archive can be built using my cross-system repository A.6.

```
git clone https://github.com/skoudmar/cross-system.git .
nix-build -A aarch64-linux.novaboot
```

The tar will be stored in the `result/tarball` directory. On the user's computer, only the `boot` directory of the archive needs to be extracted, as it contains a Novaboot script (`boot-rpi4`), Linux kernel, and the initial RAM drive image (`initrd`). The archive needs to be uploaded to the Novaboot server using rsync and then extracted.

```
# Upload the tar
rsync --rsync-path=rsync-nfsroot <the-tar> <server>:.

# Extract the tar on the server
ssh <server> untar <the-tar-name>.tar

# Extract the boot directory of the tar locally
tar xf <the-tar> boot

# Run the novaboot script
boot/boot-rpi4 -i --ssh=<server>
```

4.5 Remote builds on NixOS

There are two ways to use remote builds. First, the standard way with separate Nix stores for client and build server, where Nix copies files between stores as documented by Section 2.7.4. The other is to access the Nix daemon on the remote build machine

[5] `https://github.com/skoudmar/nixpkgs/blob/Bachelor_thesis/nixos/modules/installer/cd-dvd/system-tarball-novaboot.nix`

[6] See note 5

which has direct access to the Nix store mounted via NFS as depicted in Fig 4.3. In the following, we describe the latter option.

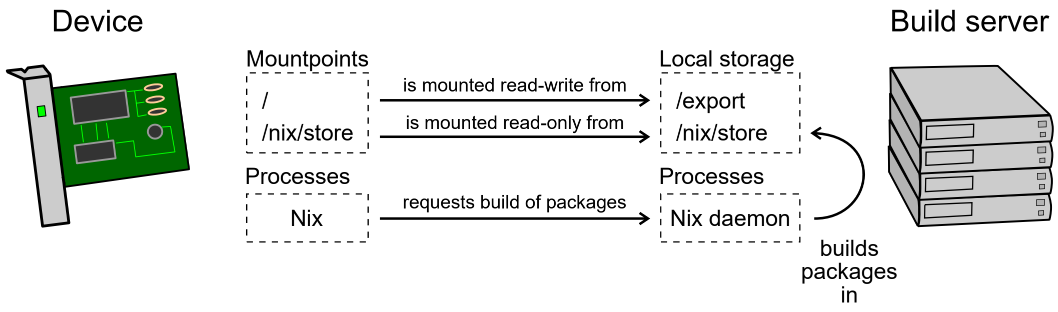


Figure 4.3. Remote Nix daemon deployment diagram.

4.5.1 Using nix daemon and NFS

Prerequisites:

- The current user has SSH access to the build server.
- Nix daemon is running on the server.
- The server exports `/nix/store` path through NFS.

NixOS stores all programs in a Nix store. Consequently, changing to a different store that does not have the programs would cause NixOS to break. Before mounting the remote Nix store on the target device, the user needs to upload their profile by the command in Listing 4.5.1 or they need to set their profile link `~/.nix-profile` and `~/.nix-defexpr/channels` to valid profile and channels on the server. If the `~/.nix-profile` link does not exist, it will be created if you install any package to your environment, e.g., `nix-env -i hello`.

On NixOS it is also required to copy `/run/current system` as it contains programs and configuration files used by the system.

```
nix-copy-closure --to <server> ~/.nix-profile ~/.nix-defexpr/channels*
nix-copy-closure --to <server> /run/current-system
```

Listing 4.5.1. Copying Nix profile, channels and system to remote server.

Now, the user should mount the remote Nix store by using the commands in Listing 4.5.2. Setting the environment variable `NIX_REMOTE` to the server tells Nix to use the Nix daemon on the server. It is important to specify the `ssh-ng` protocol and not just `ssh` because the legacy store used by `ssh` does not support all the required functions, such as `addToStore` for adding non-store paths and files that need to be copied from the local device.

```
mount <server>:/nix/store /nix/store -o ro,remount

export NIX_REMOTE=ssh-ng://<server>
```

Listing 4.5.2. Mounting NFS exported remote Nix store.

The user can now use the remote Nix store and operate with it as if it was a local store. This method works even when the server has a different architecture but only if the package is in the Nix cache and the server is only downloading it. However, here are some limitations.

Build commands do not create a link to the result and its *gc root* since this is done by the server and it does not have access to the user's device. The creation of the result link and gc root must be done manually. Fortunately for us `nix-build` command prints the path to the result on the console so that we can just copy it and create the result link and the *gc root* as in Listing 4.5.3. We recommend user to create a directory on the server such as `~/.nix-gcroots` to be a personal collection of packages used and then create links in it to serve as *gc roots*, but any other directory would work just fine.

```
# Create the result link
ln -s <copiedPath> result

# Create the gc root on server
ssh <server> nix-store -r <copiedPath> --add-root ~/.nix-gcroots/<name>
```

Listing 4.5.3. Creating missing result link and gc root.

Another limitation to consider is that the server will fail to build packages if the package is for another platform. This can be solved by manually specifying that the package should cross-compile.

```
nix-build '<nixpkgs>' -A hello --argstr system 'x86_64-linux' \
    --arg crossSystem 'builtins.currentSystem'
```

Listing 4.5.4. Command to cross-compile the `hello` package on a remote machine using remote Nix daemon.

4.5.2 Patching the Nixpkgs

To make it easier to use remote builds with remote machines of different architectures, I have created a patch for Nixpkgs in the cross-system repository A.6 to be used with my modified version of Nixpkgs A.5. The purpose of this patch is to make it easier to cross-compile packages – the command 4.5.4 is equivalent to command 4.5.5 with patched Nixpkgs.

```
nix-build '<nixpkgs>' -A hello
```

Listing 4.5.5. Command to cross-compile the `hello` package on a remote machine using remote Nix daemon with patched Nixpkgs.

The patch should be used when building the NixOS image. The built system will then contain the patched Nixpkgs and they can be used to request cross-compilation of the requested package on a remote server.

To use the patched Nixpkgs, a user must set the path to them must be set in `NIX_PATH` environment variable or must be provided to the build commands by option `-I`.

The patch sets the default value of attribute `localSystem` to the system that will apply the patch and changes the default value of `crossSystem` from being the same as `localSystem` to the system of the machine currently used. The default values used are taken from the new file `system.nix` located at the root of the Nixpkgs repository.

Chapter 5
Evaluation

This chapter will focus on the evaluation of functionality and performance.

5.1 UNFS3's compilation

The UNFS3 compilation was fixed and now UNFS3 successfully compiles and even supports the IPv6 protocol. Compilation can be tested by downloading the git repository (either upstream or my fork A.2) and using the following commands.

```
./bootstrap
./configure
make
```

These options were tested and all of them :

- `-u` to use random ports
- `-d` to run in foreground
- `-e file` to specify the custom export file
- `-P file` to generate the port file (only in my repository)
- `-t` to only provide TCP connection
- `-p` to not register used ports with rpcind/portmapper
- `-n port` to specify exact port for NFS protocol
- `-m port` to specify exact port for MOUNT protocol

5.2 Booting a Buildroot-based system with Novaboot

The goal of these tests is to determine whether Novaboot can successfully boot the system built by Buildroot.

These tests are already prepared in the integration repository A.1 and will assume the repository is used.

5.2.1 Booting on a virtual device (QEMU)

The system is already preconfigured in the `build` directory. Build the system using Buildroot running `make -C build`.

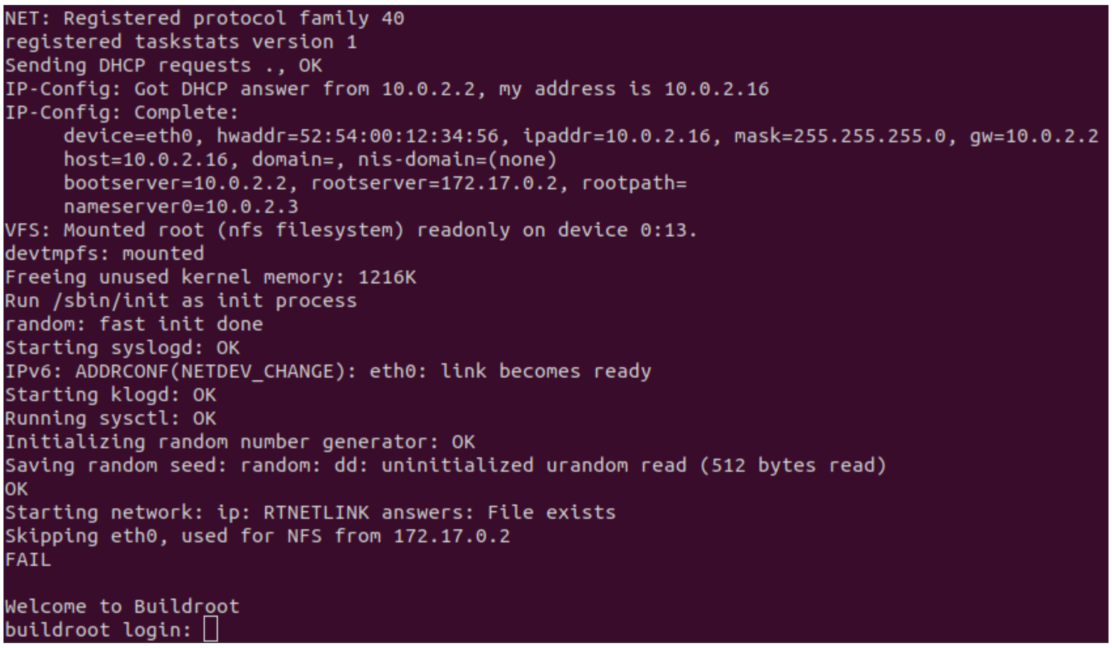
This test will use the `novaboot-server` docker container from the integration repository. The image is built by running `make docker-image` and is then started by `make run-docker`.

In the `novaboot-client/QEMU-buildroot-system` there are two deployment scripts `tftp-boot` and `nfs-boot`. The `tftp-boot` can be run without additional commands by running:

```
./tftp-boot -i --ssh=novaboot@172.17.0.2
```

To boot the system using the NFS run the folowing commands:

```
rsync --rsync-path=rsync-nfsroot -L rootfs.tar novaboot@172.17.0.2:.
ssh novaboot@172.17.0.2 untar rootfs.tar
./nfs-boot -i --ssh=novaboot@172.17.0.2
```



```
NET: Registered protocol family 40
registered taskstats version 1
Sending DHCP requests ., OK
IP-Config: Got DHCP answer from 10.0.2.2, my address is 10.0.2.16
IP-Config: Complete:
     device=eth0, hwaddr=52:54:00:12:34:56, ipaddr=10.0.2.16, mask=255.255.255.0, gw=10.0.2.2
     host=10.0.2.16, domain=, nis-domain=(none)
     bootserver=10.0.2.2, rootserver=172.17.0.2, rootpath=
     nameserver0=10.0.2.3
VFS: Mounted root (nfs filesystem) readonly on device 0:13.
devtmpfs: mounted
Freeing unused kernel memory: 1216K
Run /sbin/init as init process
random: fast init done
Starting syslogd: OK
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
Starting klogd: OK
Running sysctl: OK
Initializing random number generator: OK
Saving random seed: random: dd: uninitialized urandom read (512 bytes read)
OK
Starting network: ip: RTNETLINK answers: File exists
Skipping eth0, used for NFS from 172.17.0.2
FAIL

Welcome to Buildroot
buildroot login: 
```

Figure 5.1. Terminal output after booting Buildroot system in QEMU with NFS.

As can be seen by the image 5.1 Novaboot can successfully boot the QEMU device with the Buildroot system. Even though the system says the device is mounted read-only, all changes are correctly written to the NFS server.

5.2.2 Booting on Raspberry Pi 4

The Raspberry Pi 4 system can be built similarly by the preconfigured target in the `build_rpi4` directory.

User should prepare SD card with single FAT partition. Download `start4elf` and `fixup4.dat` from the `boot` directory of the official Raspberry Pi firmware repository[1] to the partition. Also create `config.txt` file as seen at 5.2.1. At last copy `u-boot.bin` from the `build_rpi4/image` directory.

```
kernel=u-boot.bin
arm_64bit=1
enable_uart=1
```

Listing 5.2.1. File `config.txt` for the Raspberry Pi 4.

The server can be configured by following the directions of the `novaboot-server/rpi/README.md` file.

To connect to the server, use where `<server>` is replaced with the address of the server:

```
rsync --rsync-path=rsync-nfsroot -L rootfs.tar rpi@<server>:.
ssh rpi@<server> untar rootfs.tar
./nfs-boot -i --ssh=rpi@<server>
```

[1] https://github.com/raspberrypi/firmware

Listing 5.2.2. The `.novaboot-nfs` configuration file for QEMU target.

The test shows that the Raspberry Pi can also use a Buildroot system that is deployed by Novaboot and hosted on a Novaboot-operated NFS server.

5.3 Bootnix NixOS with Novaboot and UNFS3 server

When hosting the entire NixOS, the system requires to differentiate user and group identifiers, so the only option is to run UNFS3 in the user namespace or as a root with `no_root_squash`.

5.3.1 Failing to start Nix-daemon

The systemd Nix daemon socket target can fail to start with one of the following messages.

```
Failed to create listening socket (/nix/var/nix/daemon-socket/socket):
    File name too long

Failed to create listening socket (/nix/var/nix/daemon-socket/socket):
    Input/output error
```

Listing 5.3.1. Error messages in journal for `nix-daemon.socket`.

The “File name too long” error is caused when the filename of the socket is larger than 108 characters. This length includes the length of the exported and length of the actual name. The solution is simple; use a shorter exported path on the server.

The IO error is caused because there already exists a socket from the previous boot and UNFS3 reports `NFS3ERR_IO` instead of `NFS3ERR_EXIST`.

The Nix-daemon service can be started without the socket unit manually by typing the command 5.3.2.

```
systemctl start nix-daemon.service
```

Listing 5.3.2. Start Nix daemon service

5.3.2 Failing to use Nix daemon

The Nix may fail to use the Nix daemon with the error message in Listing 5.3.3 is printed.

```
error: could not set permissions on '/nix/var/nix/profiles/per-user' to
755: Operation not permitted
```

Listing 5.3.3. Error message printed when Nix does not use Nix daemon.

This issue can be prevented by manually setting the `NIX_REMOTE` environment variable as shown in Listing 5.3.4.

```
export NIX_REMOTE=daemon
```

Listing 5.3.4. Setting `NIX_REMOTE` to tell Nix to use Nix daemon

5.4 Booting NixOS with Novaboot and kernel NFS server

During the evaluation, the only problem I encountered was the long name for the socket described in Section 5.3.1. The test were run with `exports` file depicted in Listing 5.4.1. The NixOS runs

```
/export *(rw,no_root_squash,insecure)
```

Listing 5.4.1. Exports file used with the kernel NFS server for hosting the root filesystem.

5.5 Remote Nix daemon

Accessing the Nix store through NFS works without any problems with all of the tested NFS servers:

- UNFS3 running as a root
- UNFS3 in a user namespace
- UNFS3 running unprivileged
- Kernel NFS server

```
/export *(ro,no_root_squash,insecure)
```

Listing 5.5.1. Exports file used with the kernel NFS server for hosting the Nix store.

```
/export ::/0(ro,no_root_squash,insecure)
```

Listing 5.5.2. Exports file used with the UNFS3 for hosting the Nix store.

Fetching packages:

The fetching can be tested by requesting a package that is in the Nix cache, for example, hello package.

```
nix-build '<nixpkgs>' -A hello
```

The link to the result must be created manually with steps in 4.5.3.

Building packages:

The building must be manually triggered because when using the remote daemon, Nix appears to ignore the option `--substitutes ''`. The test will be conveyed in Nix repl, the interactive nix console, and will use overriding to modify the package.

5.5.1 Missing Nix store paths

During testing, if the user encounters that some paths are missing from the Nix store, this is because they were present before mounting the remote Nix store. The fix is simple, unmount the remote Nix store and use `nix-copy-closure` to copy the missing path.

```
$ nix repl '<nixpkgs>'

# This function modifies name of the package
nix-repl> overrideFcn = (oldAttrs: { name = "hello-world"; })

nix-repl> :b hello.overrideAttrs (oldAttrs: rec {name}

# This fails because it requires that the build machine has
# the same architecture as the local device.
nix-repl> :b hello.overrideAttrs overrideFcn

nix-repl> args = {
        system="x86_64-linux";
        crossSystem = builtins.currentSystem;
}
nix-repl> crossPkgs = (import <nixpkgs> args)

# This will successfully build the package
nix-repl :b crossPkgs.hello.overrideAttrs overrideFcn
```

Listing 5.5.3. Using Nix repl to verify package cross-compilation.

5.6 Performance test of different NFS servers

The goal of this test is to compare the file access speed between different NFS servers.

The test will be run on a Raspberry Pi 4 with NixOS 22.05pre started with Novaboot. The remote machine is an 8-core AMD64 machine running Ubuntu 20.04 with multi-user installation of Nix 2.8.1. The exported directory with the root filesystem is on SSD. The Raspberry Pi is in the same 1 Gbit/s local network as the server and configured to use a TCP connection.

NFS servers include UNFS3 running as root or in the user namespace and kernel NFS servers with NFS versions 3 and 4.

`nix-instantiate` command generates store derivations from the Nix expressions. In the test, the Nixpkgs will be evaluated to produce store derivations for the `hello` package.

The test is run by `hyperfine`[2] 100 times. The script in the preparation flushes the caches of both the local machine and the NFS server. Remounting did not have any effect on the times, so it is skipped.

```
# prepare.sh file
ssh root@server 'sync; echo 3 > /proc/sys/vm/drop_caches'
sync
echo 3 > /proc/sys/vm/drop_caches
```

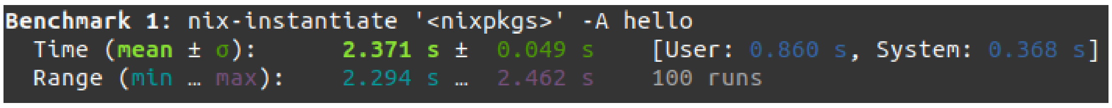
```
hyperfine --prepare ./prepare.sh --runs 100 \
    "nix-instantiate '<nixpkgs>' -A hello"
```

Listing 5.6.1. Performance test command.

[2] `https://github.com/sharkdp/hyperfine`

NFS server	Mean [s]	Std. dev. [s]	Min [s]	Max [s]
kernel NFSv4	2.488	0.041	2.433	2.667
kernel NFSv3	2.073	0.046	2.003	2.175
root UNFS3	2.155	0.048	2.061	2.239
namespace UNFS3	2.371	0.049	2.294	2.462
local SD card	1.765	0.004	1.753	1.775

Table 5.1. Performance test of NFS servers – 100 iterations.



```
Benchmark 1: nix-instantiate '<nixpkgs>' -A hello
  Time (mean ± σ):       2.371 s ±  0.049 s    [User: 0.860 s, System: 0.368 s]
  Range (min … max):     2.294 s …  2.462 s    100 runs
```

Figure 5.2. Screenshot of performance testing UNFS3 in user namespace.

As we can see at 5.1, the best is the kernel server operating with NFS version 3. Running UNFS3 unprivileged in the user namespace resulted in 10 % increase in run time.

Chapter 6
Conclusion

The goal of this work was to extend the Novaboot server part with the NFS server. This was done using the UNFS3 server, which runs without root privileges as requested. Changes made in the pull request to UNFS3 were merged into the master, and the pull request to Novaboot is soon to be merged as well.

The other goal of this work was to cross-compile the NixOS distribution and configure it for usage with the Novaboot tool. This was done successfully and the resulting system can be compiled with the cross-system GitHub repository A.6. We have also analyzed options for the remote building of packages and found that the remote Nix daemon suits our needs.

During the evaluation, we have determined that NixOS with its filesystem hosted by kernel NFS server, works without any issues, and with UNFS3 there are only minor issues that have simple workarounds.

6.1 Future work

Improving the experience by providing a better way to modify files hosted on a Novaboot-managed UNFS3 server will be the next step. This work already lists some possibilities for how it can be done in section 3.3.3.

Fixing UNFS3 issues uncovered while evaluating the usage of NixOS is also on the list.

Appendix A
Git repositories

All of my repositories contain a tag `Bachelor_thesis` that points to the last commit before submitting the thesis.

A.1 Integration repository

This repository contains configurations, builders, and steps used to evaluate the correct integration with UNFS3 with Novaboot. Buildroot system and NixOS are used with QEMU and Raspberry Pi 4.

`https://gitlab.fel.cvut.cz/skoudmar/novaboot-nfs-integration`

A.2 UNFS3

This repository contains UNFS3 source code. Many of my changes have already been merged into the upstream master branch. At the time of publication, my fork differs from upstream by having the fix for the problem 4.1.2 and has also introduced a port file.

Upstream: `https://github.com/unfs3/unfs3`

My fork: `https://github.com/skoudmar/unfs3`

A.3 User namespace creator

This repository contains the `userns` tool described in 4.2.

`https://gitlab.fel.cvut.cz/skoudmar/user-namespace-creator`

A.4 Novaboot

This repository contains the source code of Novaboot 2.3. My fork is used to work on the pull request `#10`.

Upstream: `https://github.com/wentasah/novaboot`

My fork: `https://github.com/skoudmar/novaboot`

A.5 Nixpkgs

This repository contains the Nixpkgs. My fork adds the Novaboot tarball NixOS target at 4.4 and modifications to the stage 1 init script at 4.4.1.

Upstream: `https://github.com/NixOS/nixpkgs`

My fork: `https://github.com/skoudmar/nixpkgs`

A.6 Cross-system

This repository publishes the internal NixOS targets from Nixpkgs. My fork adds a build target for Novaboot tarball with configuration.

Upstream: `https://github.com/samueldr/cross-system`

My fork: `https://github.com/skoudmar/cross-system`

Appendix B
Contents of the attached archive

This archive contains the following git repositories:

- **Integration repository** - novaboot-nfs-integration-master.tar.gz
  - Archive contains all submodules
  - This archive is tar.gz file because zip does not support symbolic links.
- **UNFS3** - unfs3-master.zip
- **User namespace creator** - user-namespace-creator-master.zip
- **Novaboot** - novaboot-master.zip
- **Nixpkgs** - nixpkgs-novaboot.tar.gz
  - This archive is tar.gz file because of better compression method.
- **Cross-system**- cross-system-master.zip

References

[1] Buildroot Association. *The Buildroot user manual* [online]. [cit. 2022-04-24]. Available from `https://buildroot.org/downloads/manual/manual.html`.

[2] Sojka, Michal. *Novaboot*. [cit. 2022-04-26]. Available from `https://github.com/wentasah/novaboot/blob/master/README.md`.

[3] Sun Microsystems, Inc. *NFS: Network File System Protocol specification* [RFC 1094]. Available from DOI 10.17487/RFC1094. Available also from `https://www.rfc-editor.org/info/rfc1094`.

[4] Staubach, Peter, Brian Pawlowski, and Brent Callaghan. *NFS Version 3 Protocol Specification* [RFC 1813]. Available from DOI 10.17487/RFC1813. Available also from `https://www.rfc-editor.org/info/rfc1813`.

[5] Haynes, Thomas, and David Noveck. *Network File System (NFS) Version 4 Protocol* [RFC 7530]. Available from DOI 10.17487/RFC7530. Available also from `https://www.rfc-editor.org/info/rfc7530`.

[6] *UNFS3* [online]. [cit. 2022-05-12]. Available from `https://github.com/unfs3/unfs3`.

[7] *UNFS3 manual* [online]. [cit. 2022-05-12]. Available from `https://github.com/unfs3/unfs3/blob/master/unfsd.8`.

[8] *NFS-Ganesha* [online]. [cit. 2022-05-12]. Available from `https://github.com/nfs-ganesha/nfs-ganesha/wiki`.

[9] Dassen, J.H.M., Joost Witteveen, and Clint Adams. *Fakeroot* [online]. [cit. 2022-04-30]. Available from `http://manpages.ubuntu.com/manpages/trusty/man1/fakeroot-tcp.1.html`.

[10] Shemesh, Shachar. *Fakeroot NG* [online]. [cit. 2022-01-09]. Available from `https://fakeroot-ng.lingnu.com/`.

[11] *systemd* [online]. [cit. 2022-05-02]. Available from `https://systemd.io/`.

[12] *ArchWiki - systemd* [online]. [cit. 2022-05-02]. Available from `https://wiki.archlinux.org/title/systemd`.

[13] Dolstra, Eelco. *The purely functional software deployment model*. Utrecht University, 2006. ISBN 90-393-4130-3. PhD. thesis.

[14] NixOS Contributors. *NixOS* [online]. [cit. 2022-04-24]. Available from `https://nixos.wiki/wiki/NixOS`.

[15] NixOS Contributors. *How Nix works* [online]. [cit. 2022-04-24]. Available from `https://nixos.org/guides/how-nix-works.html`.

[16] NixOS Contributors. *Nixpkgs 21.11 manual* [online]. [cit. 2022-05-15]. Available from `https://nixos.org/manual/nixpkgs/stable/`.

[17] NixOS Contributors. *Nix manual: nix.conf* [online]. [cit. 2022-05-15]. Available from `https://nixos.org/manual/nix/stable/command-ref/conf-file.html`.

[18] NixOS Contributors. *Nix manual: Remote Builds* [online]. [cit. 2022-04-25]. Available from `https://nixos.org/manual/nix/stable/advanced-topics/distributed-builds.html`.

[19] Torvalds, Linus. *Linux kernel: linux/net/ipv4/ipconfig.c* [online]. [cit. 2022-05-14]. Available from `https://github.com/torvalds/linux/blob/master/net/ipv4/ipconfig.c`.