CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

Light-weight Packaging System for a Linux Distribution

Denis Shcherbakov
Software Engineering and Technology

May 2022
Supervisor: Ing. Pavel Troller, CSc.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Shcherbakov** Jméno: **Denis** Osobní číslo: **483823**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Odlehčený balíčkovací systém pro Linuxovou distribuci

Název bakalářské práce anglicky:

Light-weight Packaging System for a Linux Distribution

Pokyny pro vypracování:

Navrhněte a naprogramujte nástroj pro instalaci, update a odinstalaci volitelných programových balíčků pro Linuxovou distribuci. Dodržte následující pokyny:
- Systém si nebude udržovat centralizovanou databázi instalovaných balíčků.
- Balíčky budou vždy instalovány samostatně do adresáře /opt.
- V balíčku budou metadata, popisující důležité informace pro balíčkovací systém (závislosti na jiných balíčcích a jejich verzích a další dle potřeby).
- Jako vnější formát balíčku bude použit standardní formát komprimovaného taru, obsahující přímo adresáře balíčku + soubor/adresář s metadaty,

Nástroj vytvořte v jazyce C tak, aby spustitelný program byl pokud možno malý a nezávislý na mnoha méně obvyklých knihovnách. Cílem je aplikace i na různé embedded platformy.

Seznam doporučené literatury:

[1]: WELSH, M. et al.: Running Linux. O'Reilly Media; Fourth edition (December 15, 2002), ISBN: 0-59600-272-6.
[2]: BRODSKÝ, J. - SKOČOVSKÝ, L.: Operační systém Unix a jazyk C. SNTL - Státní nakladatelství technické literatury 1989, 368 s., ISBN: 80-03-00049-1.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Pavel Troller, CSc. katedra telekomunikační techniky FEL

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

Datum zadání bakalářské práce: **01.03.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **19.02.2024**

Ing. Pavel Troller, CSc. 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

Chtěl bych poděkovat vedoucímu své práce Ing. Pavlu Trollerovi, CSc. za odborné vedení bakalářské práce, za jeho cenné rady, podporu a čas, který mi věnoval, a zkušenosti, které jsem získal v rámci práce na tomto projektu.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 20. května 2022

....

Abstrakt / Abstract

Tato bakalářská práce si klade za cíl navrhnout a vyvinout decentralizovanou aplikaci správce balíčků v jazyce C pro operační systém Linux. Aplikace umožní uživatelům provádět všechny potřebné akce nad balíčky pro desktopové a embedded systémy.

Klíčová slova: decentralizovaný správce balíčků; Linux; jazyk C.

This bachelor thesis aims to design and develop a decentralized package manager application in C language for the Linux operating system. The application will allow users to perform all the necessary actions on packages for desktop and embedded systems.

Keywords: decentralized package manager; Linux; C language.

Contents /

1 Foreword **1**

2 Introduction **2**

2.1 Basic definitions 2

2.1.1 Package file 2

2.1.2 Dependency 2

2.1.3 Package management 2

2.1.4 Database 3

3 Goals and Structure **4**

3.1 Goals 4

3.2 Structure 4

3.2.1 Package Manager 4

3.2.2 Developer Tool 5

4 Analysis **6**

4.1 Directories and Files 6

4.2 System Characteristics 6

4.3 Data Structures 6

4.4 Access to a Repository 6

4.5 Implementation-Dependent Constants 6

4.6 Regular Expression 6

5 Design **7**

5.1 Why Are We Using the C Language? 7

5.2 Working with Directories 7

5.3 System Name Structure 8

5.4 Set Data Type 8

5.5 The libcurl API 9

5.6 implementation-Dependent Constants 10

5.7 Regular-Expression-Matching Types 10

5.8 Usage of Configuration Files . . 11

5.9 Easily Removable Packages . . 12

5.10 Parsing Command-Line Options 12

6 Implementation **13**

6.1 Integrated Development Environment 13

6.2 Package File 13

6.3 Package Manager 14

6.3.1 Initialization 14

6.3.2 CLI Options 15

6.3.3 Install from a Remote Server 16

6.3.4 Install from a Local Archive 18

6.3.5 Update 20

6.3.6 Remove 21

6.3.7 Search 21

6.3.8 Status 22

6.4 Developer Tool 23

6.4.1 Initialization 23

6.4.2 CLI Options 23

6.4.3 Create MetaData 23

6.4.4 Create Prepared Package . 24

6.5 Encountered Problems 24

6.5.1 Working with Memory . . . 24

6.5.2 Linking Libraries to the Project 25

6.5.3 Regular Expression cflags . 25

6.5.4 The Functionality of Standard Library. 25

7 Testing **26**

7.1 The Developer Tool 26

7.2 The Package Manager 26

8 Results **27**

8.1 Possibilities of Further Development 27

9 Conclusion **28**

References **29**

A User Manual **31**

B The Project's Repositories **33**

/ Figures

6.1 Flowchart of the main function 14
6.2 Flowchart of the install remote package function 16
6.3 Flowchart of the install local package function 18
6.4 Flowchart of the update package function 20
6.5 Flowchart of the remove package function 21
6.6 Flowchart of the search package function 21
6.7 Flowchart of the package status function 22

Chapter 1
Foreword

Before the existence of package management systems, there was a simple method of installing software on any Unix or Unix-like systems via an archive, commonly known as the **tarball**.

The **tarball**, or formally **tar** (“**T**ape **AR**chive”)[1], is an archive that contains multiple files and their parameters such as ownership, permissions, and directory organization.

After unpacking the archive and installing it using simple commands via Terminal, a user could start the application. But there were many difficulties for a user, for example, such as:

- There was no way to manage software after tarball was installed.
- It was difficult for a regular user to determine which version of the software is installing, as such information was never included in a tarball.
- Inability to upgrade the software because files were spread across the system upon installation.
- If the software requires dependencies to be installed, the user must install every dependency manually.

These factors have led to the emergence of today’s well-known package management systems or package managers and the creation and design of this project.

Chapter 2
Introduction

This work aims to design and develop a decentralized package manager application in the C language for the Linux operating system. The application will allow users to perform all the necessary actions on packages for desktop and embedded systems.

Here are described necessary definitions, which will be helpful for the reader to understand the range of problems that we will have to face in this work and overcome them in the future.

2.1 Basic definitions

This section explains basic concepts such as package, dependency, local database, and package management, which are essential for understanding the project's perspective.

2.1.1 Package file

The package file is an archive file that usually contains all of the program's necessary files, metadata, and instructions to implement software functionality.

The metadata consists of a manifest file and directory layouts. Files of a program that could be found inside the package are stored in either source code or binary executable files[2].

Usually, a package file is in pre-compiled binary format. Therefore installation will be quick, and no additional software compiling is required.

2.1.2 Dependency

Dependencies on their own are packages required by the primary package to function correctly.

Mismanaged dependencies may cause problems significant enough to extend, which can be lethal to a whole system.[3]

Therefore to resolve emerging issues, package management systems have been developed.

2.1.3 Package management

Package management is an organized method of automation of processes such as installing, upgrading, configuring, and removing packages for an operating system consistently[4].

The package manager resolves which dependencies the particular software needs to be installed on the system to work correctly.

Nowadays, most modern software installed on a Linux or Unix system can be found on the Internet or in so-called software repositories, which are grouping packages together and designed to be malware-free.

2.1.4 Database

After installation, the typical package manager stores all metadata of installed software in a local database. These metadata contain information about package dependencies and their versions to prevent software mismatches and missing prerequisites.

When a Linux operating system user performs a software upgrade, it needs to run synchronization with the repository to know what packages have got their newer version. Then the package manager could install an upgrade.

But in our project, we aim to achieve an approach that will not require the presence of a local database. Still, a package's metadata will be stored within its directory, which means the information will be distributed evenly; in other words - each package has its small database. And if the metadata is corrupted, a reinstallation process must be performed for that particular package.

Chapter 3
Goals and Structure

This chapter describes the goals we aim to achieve upon project completion and its basic structure's precise yet straightforward specifications.

3.1 Goals

Creating a decentralized package manager is the primary goal of the project, which will be written using C language and has these distinctive features:

- **C**ommand-**L**ine **I**nterface (**CLI**) for installing, updating, and removing (etc.) packages
- Absence of a central database, which will prevent unnecessary storage of data about installed packages and lowering their possibility to be corrupted as the single point of failure
- Custom, compiled packages as **tar** files, which will contain additional metadata; thus, it will help maintain installed packages on a system
- Using as less 3rd party libraries in the project as possible, hence it will allow us to get the result reasonably lightweight, and therefore its usage might be practical even on embedded systems

3.2 Structure

The project contains two independent applications.
First one is the package manager, and second is so called developer tool, both are described bellow.

3.2.1 Package Manager

This is the primary part of the project, used by regular users of CLI Linux environment, more concrete - Sinux distribution.

The package manager implements the following basic features:

- install a package
- update a package (or update all)
- remove a package
- search for a package in a repository
- get the status of an installed package

3.2.2 Developer Tool

So, we have the application for performing maintenance of packages on the system and a remote server (repository) from which we will download and then install desired packages. In the first place, how do we create a package compatible with our infrastructure and keep it up-to-date?

Developers will require a unique tool for this. It will collect all necessary data to create a metadata file and place it in a tar archive, which might be later uploaded to the repository by a developer.

Chapter 4
Analysis

This chapter describes the needed requirements and principles to fulfill the project design.

4.1 Directories and Files

Checking the existence of directories and files is essential for project development. Also, access to a particular directory would require the application to run under the root privilege.

4.2 System Characteristics

Using universal naming conventions about system and hardware information, such as the name of an operating system and its architecture, will help us build an application that will consistently utilize system data.

4.3 Data Structures

The C language is known for its outstanding performance, precise memory usage, easy Linux integration, and lack of complex data structures. Hence the requirement to develop a custom data type or use 3rd party solution.

4.4 Access to a Repository

The application has to have the ability to connect to the repository and retrieve needed data from it, to download and search for packages, whether as a registered or anonymous account, which will depend on a particular data server.

4.5 Implementation-Dependent Constants

Unified constants for implementation purposes will provide robustness, consistency, and convenience throughout the whole project.

4.6 Regular Expression

A **reg**ular **ex**pression (**RegEx**) is a powerful tool to match the input with a pattern. Based on that, it could quickly and easily give a result if it was used right.

In this work, we will use RegEx to extract all necessary data from the received output from the system or get info about packages on the server.

Chapter 5
Design

This chapter describes the necessary technologies required to develop and implement the aforementioned requirements.

5.1 Why Are We Using the C Language?

Embedded systems usually have not got installed such programming languages as `Python`, `Perl`, `Rust` etc., which use code interpreters to run written applications. Moreover, they generally consume more physical memory, RAM, and processor time to start a solution.

On the other hand, the `C` language environment requires the least amount of libraries to start an application. It is "natural" for any Unix-like operating system and is straightforward and easier.

5.2 Working with Directories

A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.[5]

Therefore, a way to distinguish between a regular file and a directory should exist. Such a solution exists in the form of a "pseudo-standard" library `dirent` that contains constructs that facilitate directory traversing and is usually portable between platforms.[6]

By including the following header file in the project, we will be able to distinct directories while traversing them to collect important information:

```
#include ⟨dirent.h⟩
```

Inside the `dirent` library exists a valuable structure that will help us determine an entity type, such as a directory, as mentioned above:

```
struct dirent
{
    ...
    // DT_DIR - the type of a directory.
    unsigned char d_type;
    ...
};
```

5.3 System Name Structure

The package that the package manager will download from the repository has to be tested on compatibility with the target system on which it will be installed. To do that, the application has to know at least two parameters: the kernel's name and CPU architecture.

By including the following header file in the project, we will be able to retrieve system-defined values of the machine's architecture and the operating system's name:

```
#include ⟨sys/utsname.h⟩
```

The header file declares the `utsname` structure[7] (basically the `uname()` functionality), which includes the following members which are necessary for the project:

```
// Name of this implementation of the operating system.
char sysname[]
// Name of the hardware type on which the system is running.
char machine[]
```

5.4 Set Data Type

When the application will search or collect data about packages it would require to specify uniqueness of their names, that is why set data type will be useful.

Also time complexity of set operations is reasonably fast; as for adding an elements it is `O(1)`, and for checking existence of an element in a set is `O(n)` in a worst case.

A simple implementation of a set has been chosen for purposes of this project that operates only with strings.

To use the library, `set.h` and `set.c` files from the GitHub repository[1] (under MIT license[2]) will be included in the project structure to use the library:

```
#include "set/set.h"
```

Initializing a set is required to allocate memory, and after usage, it has to be freed.

```
static __inline__ int set_init(SimpleSet* set);
int set_destroy(SimpleSet* set);
```

There are also realizations of some useful functions for the project, such as:

```
// Add element to set
int set_add(SimpleSet* set, const char* key);

// Remove element from the set
int set_remove(SimpleSet* set, const char* key);
```

[1] Tyler Barrus. A simple set implementation in C `https://github.com/barrust/set`
[2] MIT license `https://opensource.org/licenses/MIT`

```
    //  Check if key in set
    int set_contains(SimpleSet* set, const char* key);

    //  Return the number of elements in the set
    uint64_t set_length(SimpleSet* set);
```

5.5 The libcurl API

`libcurl` is a free, thread-safe, feature-rich, fast, thoroughly documented multiprotocol client-side file transfer library suitable for the project for accessing, retrieving, and downloading packages from a repository.[8]

After installing curl on the system, the header must be included in the code to use its functionality:

```
    #include ⟨curl/curl.h⟩
```

`libcurl` C API has synchronous and asynchronous interfaces for committing file transfers. For purposes of this work, synchronous version of `libcurl` interface will be used.

First of all `libcurl` session must be initialized to get a handle, which further will be used as input to the numerous interface functions.

```
//  Start a libcurl easy session
CURL *curl_easy_init();
```

Second, by setting all the options for a handle in the upcoming transfer (most important among them is the URL itself), some callbacks have to be set as well that will be called from the library when data is available.

```
//  Set options for a curl easy handle
CURLcode curl_easy_setopt(CURL *handle, CURLoption option, parameter);
```

Third, after everything is setup, `libcurl` performs data transfer. It will then do the entire operation and won't return until it is done or failed.

```
//  Perform a blocking file transfer
CURLcode curl_easy_perform(CURL *easy_handle);
```

And last, after needed actions have been performed, we may get information about the transfer and then cleanup the session's handle to free initialized memory.

```
//  End a libcurl easy handle
void curl_easy_cleanup(CURL *handle);
```

5.6 implementation-Dependent Constants

Numerical values in code (“Magic Numbers”) can cause complicated problems if and when it becomes necessary to change a value.

From the point of view of portability, absolute values may cause more subtle problems. The type of a numeric value is dependent on the system’s implementation.[9]

Therefore, the project will use `limits.h` header, which defines various symbolic names. The names represent different limits on resources that the particular system imposes on applications.[10]

```
    #include ⟨linux/limits.h⟩
```

These symbolic names used in the project are described below:

```
    //  Maximum number of bytes in a filename.
    NAME_MAX
    //  Maximum number of bytes in a pathname.
    PATH_MAX
    //  Maximum length, in bytes, of a utility's input line
    _POSIX2_LINE_MAX
```

5.7 Regular-Expression-Matching Types

This work assumes operations with file names and complex lines of data from which the application will extract meaningful information about packages, for example, name, version, dependencies, etc.

Such actions would require writing a few intricate functions to handle similar tasks or applying a long-known method of processing a text - regular expressions.

In order to use them in the project, the header[11] will be included:

```
    #include ⟨regex.h⟩
```

The first step of using the `regex.h` library is to allocate memory for a regular expression and compile it by following the chosen pattern.

```
//  Compile the regular expression contained in the pattern
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

The compiled expression must be executed to detect a match and proceed with further actions over the result.

```
//  Compares the string with the compiled regular expression preg
//  Returns 0, if a match was found
int regexec(const regex_t *preg, const char *string,
    size_t nmatch, regmatch_t pmatch[], int eflags);
```

In the end, allocated memory for the regular expression must be freed to avoid unexpected memory leaks.

```
//  Free any memory allocated by regcomp() associated with preg
void regfree(regex_t *preg);
```

5.8 Usage of Configuration Files

Before the application connects to the repository server, the user must set its address (if we assume that the server may change it or else), but how?

There is a way of asking it every time the application starts. But a more practical way would be to use a configuration file that the user could customize.

For purposes of this work, we will be using the `libconfig` library. A low-footprint implementation (just 37K for the C library) will be well-suited memory-constrained embedded systems.[12]

To use the library from C code, the following preprocessor directive in source files will be included:

```
    #include ⟨libconfig.h⟩
```

First, initialize the essential `config_t` structure as an empty configuration.

```
//  Initializing the config_t structure as a new, empty configuration
void config_init (config_t * config)
```

Second, bind the initialized empty `config` to a file that contains a structured readable configuration defined by the documentation so that the read information will be parsed to the `config` structure.

```
//  Reading and parsing the file named filename
//  into the configuration object config
int config_read_file (config_t * config, const char * filename)
```

The most valuable functionality for the project is to check the presence of necessary configurations, such as the URL of a repository and the user's credentials, by using the function `config_lookup_string`.

```
//  Looking up the value of the setting in the configuration config
//  specified by the path
int config_lookup_string (const config_t * config, const char * path,
    const char ** value)
```

And finally, as you can already guess, all used memory must be deallocated. Thanks to the following function:

```
//  Destroying the config
//  deallocating all memory associated with the configuration
//  but does not attempt to deallocate the config_t structure itself.
void config_destroy (config_t * config)
```

5.9 Easily Removable Packages

The package manager has access only to the /opt64 directory, installing packages there, and not spreading their files all over the system's directories like /lib, /bin, /share, /include, etc. as other database-based package managers do. Therefore we do not need to be concerned that it would remove data that it should not.

5.10 Parsing Command-Line Options

The project will also use a universal command-line options parser for Unix-like systems. To be more precise, a GNU extension getopt_long[13]. It allows parsing of more readable, multicharacter options introduced by two dashes instead of one.

By including the header, it will allow using getopt_long() functionality:[14]

```
#include ⟨getopt.h⟩

int getopt_long(int argc, char * const *argv, const char *optstring,
                const struct option *longopts, int *longindex);
```

The getopt_long() call requires a string for short options (each option as a single letter) and an array describing the long options. The last element of the longopts array has to be filled with zeroes. Each element of the array is a structure:

```
struct option {
   char *name;      //  The option name
   int has_arg;     //  Expectancy of an argument
   int *flag;       //  Defines what a long option will return.
   int val;         //  Value to return
};
```

getopt_long() returns the option character when a short option is recognized. For a long option, the function returns val if flag is NULL and 0 otherwise. If all command-line options have been parsed, then it returns -1.

Chapter 6
Implementation

This chapter describes the tools used to create the project, the application's architecture, the structure of the package manager and the developer tool (including some parts of the code), and problems that emerged during the implementation process.

6.1 Integrated Development Environment

The development process can occur on any system; however, it will be more straightforward if the environment is similar to the target `Linux` system. Thus, making it easier development and further testing phase by eliminating excessive cross-compiling of the application for another system.

This project used the `Debian`[1] distribution of `Linux` and `CLion`[2], a cross-platform `IDE` for `C/C++`. The `IDE` uses a `CMake` build system that uses scripts called `CMakeLists` to generate build files for the specified environment.

6.2 Package File

Each package, either already installed or as a prepared archive, has to have a metadata file that can be found in the following directory:

```
"/opt64/⟨package_name⟩/sinmeta/data"
```

It has not such a complicated internal structure, so parsing the file would allow extracting needed data for further manipulations.

```
//  Name of the package
PKG=⟨package_name⟩
//  Package's verion, for example: 1.0.0
VER=⟨package_version⟩
//  List of dependencies, for example: test2--1.0.3,test1--1.0.0
DEP=⟨dependency_1⟩--⟨dep1_version⟩,⟨dependency_2⟩--⟨dep2_version⟩,...
//  Required architecture. x86_64, ARM, etc.
ARC=⟨required_architecture⟩
//  Required OS. In our case it has to be Linux
SYS=⟨requred_operating_system⟩
//  Flag for updating libraries after installation. Y or N
LIB=⟨flag_for_updating_libraries⟩
```

[1] Debian GNU/Linux `https://www.debian.org/`
[2] CLion `https://www.jetbrains.com/clion/`

6.3 Package Manager

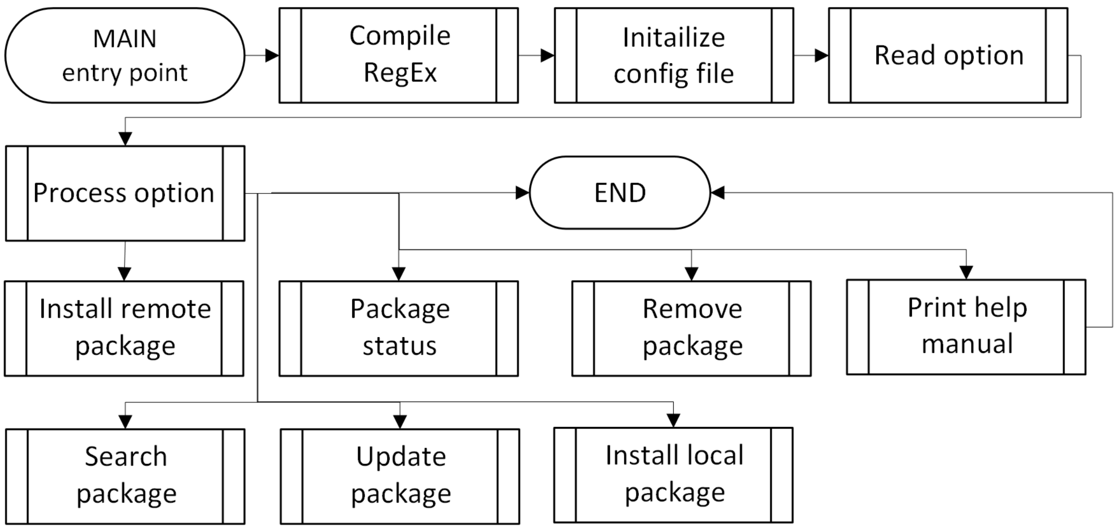

Figure 6.1. Flowchart of the main function.

6.3.1 Initialization

At the very start of the application, we initialize a configuration file and system information and compile a regular expression, which the program will use later.

1. Initialization of the configuration file:

```
config_t config;
char* configFile = "sinpm.conf";

config_init(&config);

if (config_read_file(&config, configFile) == CONFIG_FALSE) {
    printf("%s:%d - %s\n", config_error_file(&config),
            config_error_line(&config), config_error_text(&config));
    config_destroy(&config);

    exit(EXIT_FAILURE);
}
```

Then the program will check presence of repository address, login, and password.

2. Initialization of the system information and writing it to an auxiliary structure:

```
struct utsname systemInfo;
...
systemInfo = getSystemInfo();
PackageInfo* packageInfo = malloc(sizeof(PackageInfo));
...
strcpy(packageInfo->architecture, systemInfo.machine);
strcpy(packageInfo->operatingSystem, systemInfo.sysname);
```

3. Compiling a regular expression:

```
regex_t* regexCompiled;
...
int regexCompile() {
    char* regexString =
    "^(.*?)--([0-9]+\\.[0-9]+\\.[0-9]+)--(.*?)--(.*)\\.(.*)$";

    int comp = regcomp(regexCompiled, regexString, REG_EXTENDED);
    if (comp != 0) {
        printf("Could not compile regular expression.\n");
        regfree(regexCompiled);
        return 1;
    }
    return 0;
}
```

6.3.2 CLI Options

There are a couple of input command-line options:

- Install from a remote source is downloading a package from a repository server

  `--install` or `-i` option requires an argument.

- Install from a local source is installing a local archive with suitable requirements

  `--extract` or `-e` option requires an argument.

- Update specified package, if it exists on the system, or update all packages

  `--update` or `-u` option. Argument is optional.

- Remove specified package if it exists on the system

  `--remove` or `-r` option requires an argument.

- Print information about an installed package, if it exists on the system, or all installed packages, such as name and version

  `--status` or `-t` option. Argument is optional.

- Search for a package on a repository server

  `--search` or `-s` option requires an argument.

- Print a short help manual

  `--help` or `-h` option. Argument is optional.

6.3.3 Install from a Remote Server

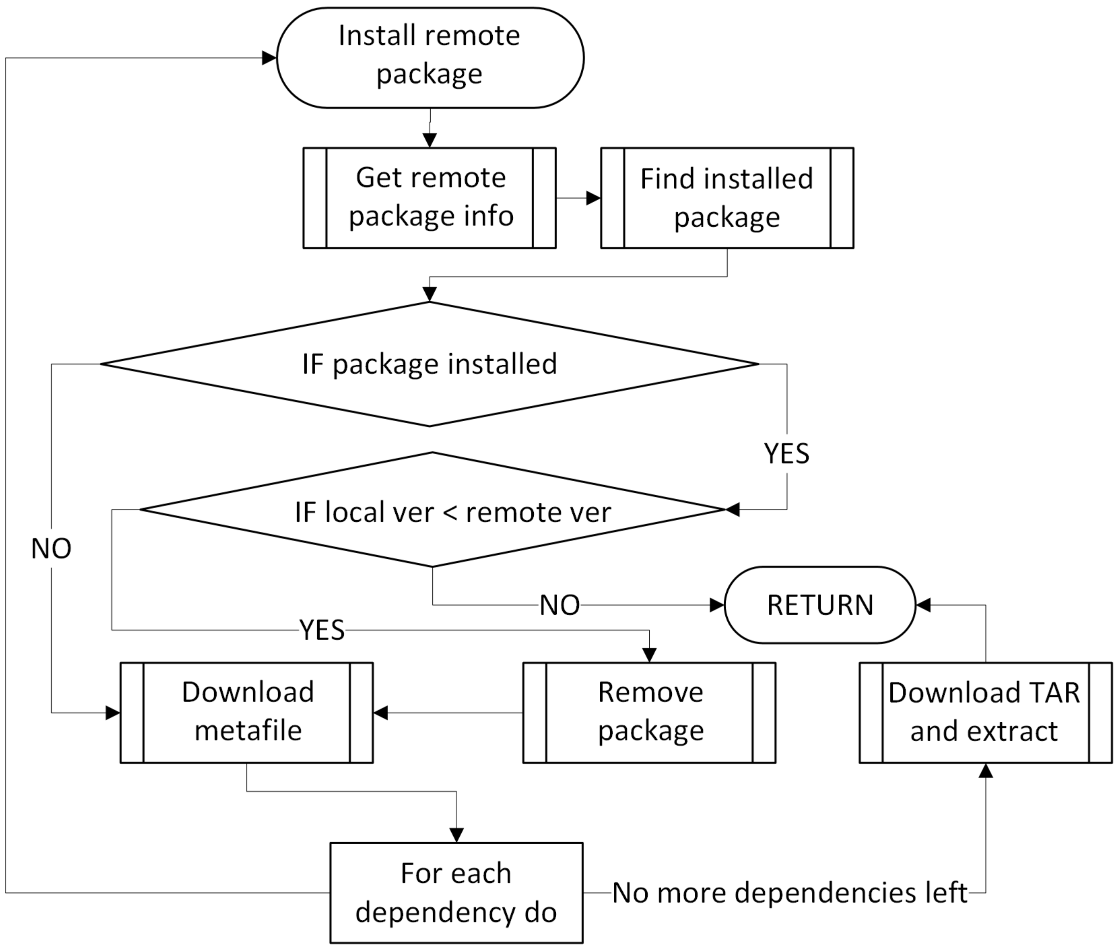

Figure 6.2. Flowchart of the install remote package function.

The initial call of the remote install function from the `main.c` accepts two arguments: the `PackageInfo` auxiliary structure previously filled with system information and the second is a boolean recursive flag.

The flag defines whether it should or should not ask to update a package, primarily inside a recursive installation call.

```
void installRemotePackage(PackageInfo* packageInfo, int isRecursive);
```

1. Get the last version of the package from a remote repository:

```
void fillPkgInfoRemoteVersion(PackageInfo* packageInfo);
```

2. Get version of installed package:

```
PackageInfo* getInstalledPackageInfo(char* packageName);
```

3. If the local version of the package exists and it is less than the remote version, the application will remove(6.3.6) the package:

```
if (localInfo != NULL && isRecursive == 0
    && compareVersions(localInfo->version, packageInfo->version) > 0) {
    ...
    removePackage(packageInfo->packageName);
}
```

4. Download auxiliary metafile to get complete information about the package:

```
FtpFile metaFile;
...
metaFile = ftpDownload(packageInfo->packageName,
                        packageInfo->version, ".meta");
```

5. Recursively install its dependencies:

```
if (packageInfo->dependenciesCount > 0) {
    for (int i = 0; i ⟨ packageInfo-⟩dependenciesCount; ++i) {
        installRemotePackage(&packageInfo->dependencies[i], 1);
    }
}
```

6. And finally, download and extract the package's archive:

```
FtpFile tarFile;
...
tarFile = ftpDownload(packageInfo->packageName,
                        packageInfo->version, ".tar");
extractArchive(tarFile.fullName);
```

6.3.4 Install from a Local Archive

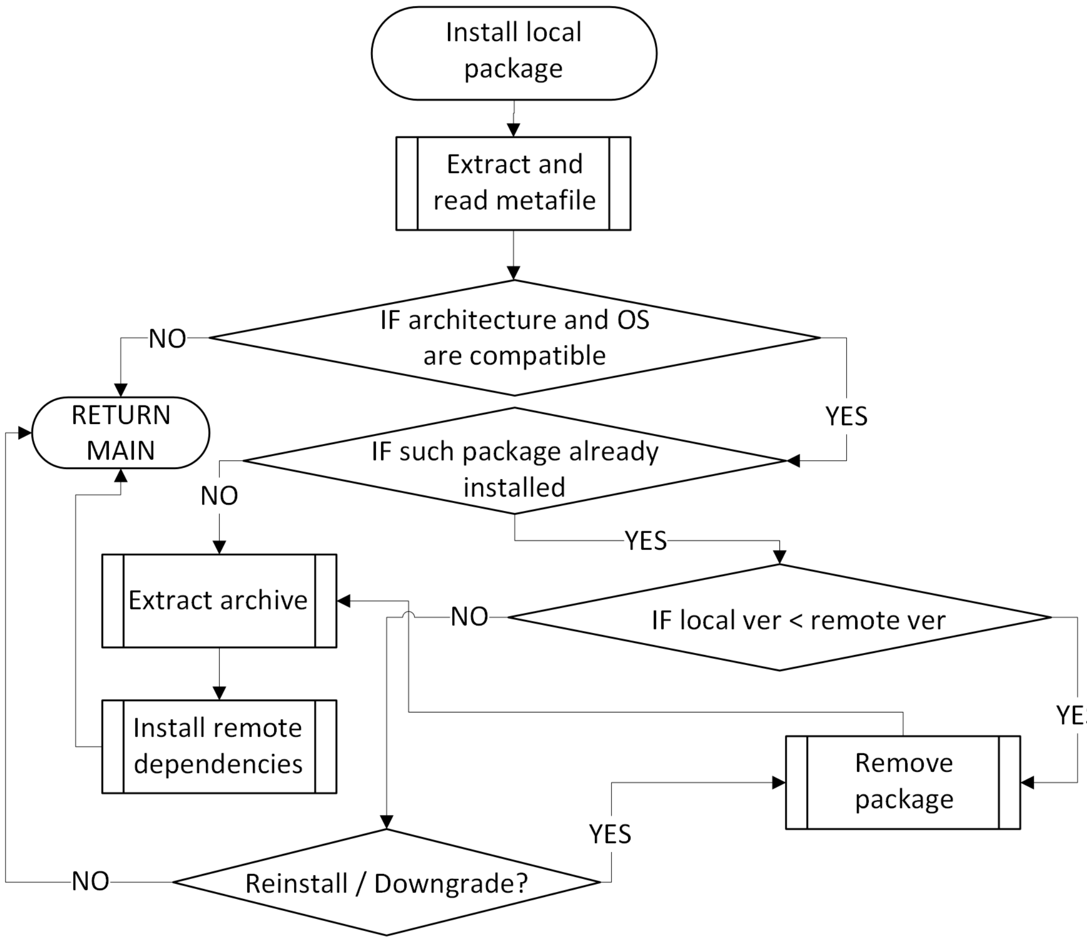
Install local package
Extract and read metafile
IF architecture and OS are compatible
NO
YES
RETURN MAIN
IF such package already installed
NO
YES
Extract archive
IF local ver < remote ver
NO
YES
Install remote dependencies
Remove package
YES
Reinstall / Downgrade?
NO

Figure 6.3. Flowchart of the install local package function.

The application will start installing a local package after entering an absolute path of a package archive to the `--extract` option:

```
void installLocalPackage(char* absolutePath);
```

1. In the beginning, we will untar a single metafile from the archive to the `/tmp/sinpmtmp/` directory:

```
char command[PATH_MAX + 10] = "tar -xvf ";
strcat(command, absolutePath);
strcat(command,
        " --one-top-level=/tmp/sinpmtmp
        --strip-components=2 --wildcards\"*/sinmeta/data\"");

int code = system(command);
if (code != 0) {
    perror("Error extracting metadata!\n");

    exit(EXIT_FAILURE);
}
```

2. Then read that file:

```
FILE* file = fopen("/tmp/sinpmtmp/data", "r");
...
PackageInfo* pkgInfo = malloc(sizeof(PackageInfo));
fillPkgInfoFromFile(file, pkgInfo);
```

3. Compare OS and architecture:

```
if (strcmp(systemInfo.sysname, pkgInfo->operatingSystem) != 0) {
    ...
    exit(EXIT_FAILURE);
}
if (strcmp(systemInfo.machine, pkgInfo->architecture) != 0) {
    ...
    exit(EXIT_FAILURE);
}
```

4a. Then check if the package is already installed, and if not, then extract the archive and install its dependencies:

```
PackageInfo* installedPackage =
                        getInstalledPackageInfo(pkgInfo->packageName);
if (installedPackage == NULL) {
    extractArchive(absolutePath);

    if (pkgInfo->dependenciesCount != 0) {
        for (int i = 0; i ⟨ pkgInfo-⟩dependenciesCount; ++i) {
            installRemotePackage(&pkgInfo->dependencies[i], 1);
        }
    }
    ...
    return;
}
```

4b. If the package is already installed - remove it before extracting:

```
removePackage(installedPackage->packageName);
```

6.3.5 Update

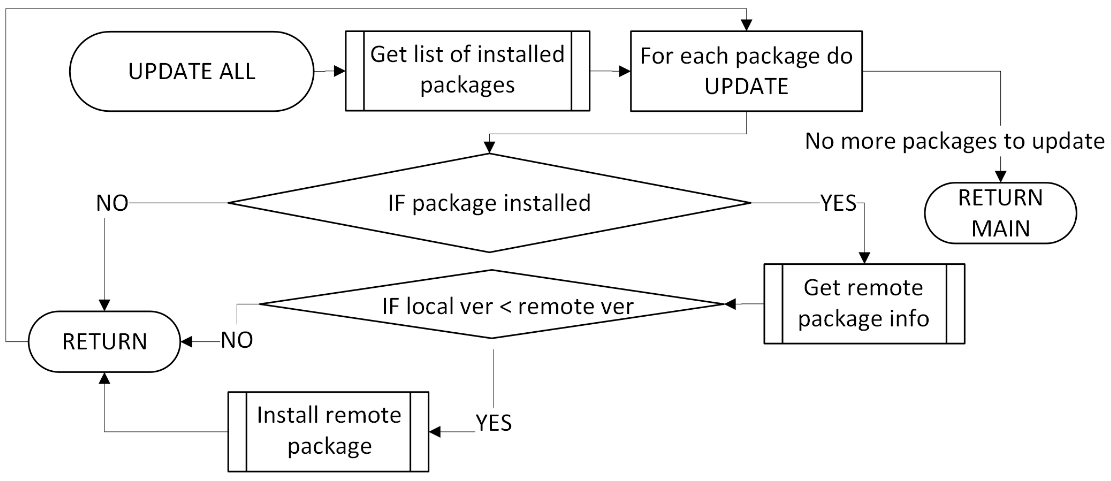
UPDATE ALL
Get list of installed packages
For each package do UPDATE
No more packages to update
RETURN MAIN
IF package installed
NO
YES
Get remote package info
IF local ver < remote ver
RETURN
NO
YES
Install remote package

Figure 6.4. Flowchart of the update package function.

Update all functionality updates every package on the system:

```
void updatePackage(char* packageName);
```

1. Get a local version of the installed package:

```
PackageInfo* localInfo = getInstalledPackageInfo(packageName);
if (localInfo == NULL) {
    printf(
        "There is no such package [%s] in the system!\n", packageName
    );
    return;
}
```

2. Then get a remote version:

```
char localVersion[NAME_MAX];
strcpy(localVersion, localInfo->version);
fillPkgInfoRemoteVersion(localInfo);
```

3. And if the remote version is newer than the local, do the installation process mentioned above(6.3.3):

```
if (compareVersions(localVersion, localInfo->version) > 1) {
    installRemotePackage(localInfo, 0);
}
```

6.3.6 Remove

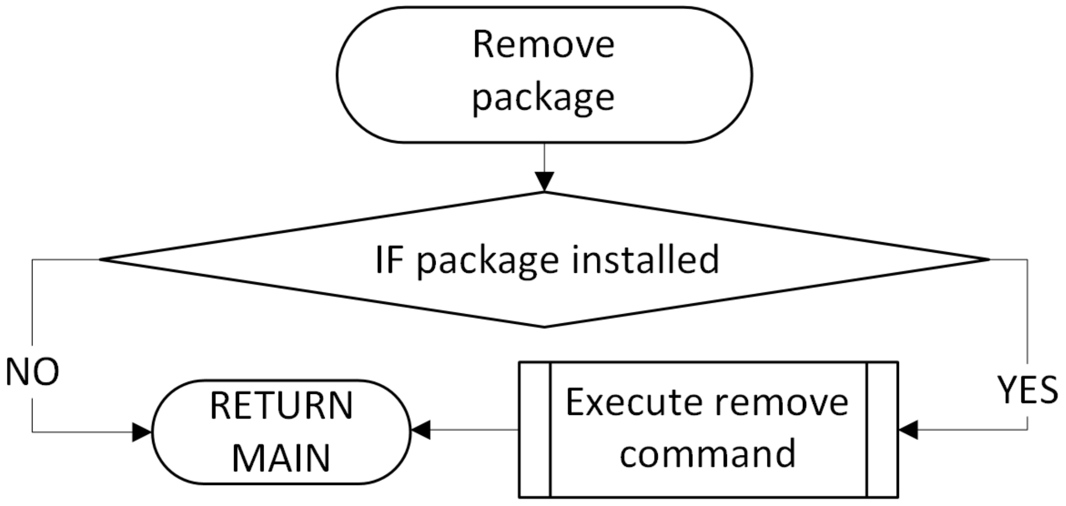

Figure 6.5. Flowchart of the remove package function.

To remove a package, the program has to check if it exists on the system

```
PackageInfo* info = getInstalledPackageInfo(packageName);
if (info == NULL) {
    printf(
    "Package [%s] has not been installed. Nothing to remove here!\n",
    packageName
    );
    return;
}
```

Then by opening the corresponding directory, it will remove it by executing the system command:

```
"rm -rf /opt64/⟨packageName⟩"
```

6.3.7 Search

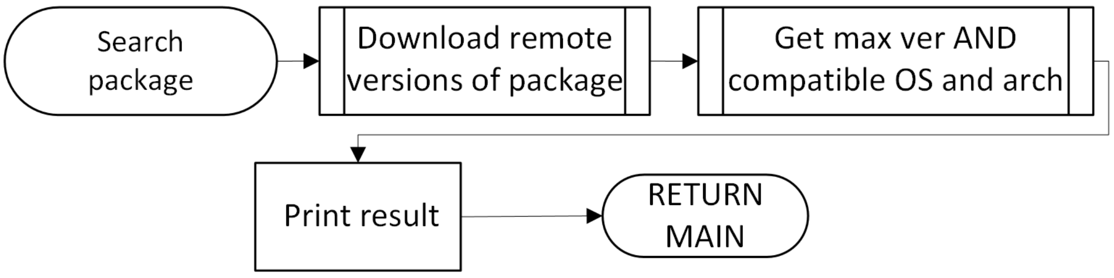
Search package
Download remote versions of package
Get max ver AND compatible OS and arch
Print result
RETURN MAIN

Figure 6.6. Flowchart of the search package function.

Search performs via function call:

```
void searchPackage(char* packageName);
```

1. The process starts by getting a list of packages with the specified name from the repository server:

```
StringList* list = ftpGetNames(packageName);
if (list->fileCount == 0) {
    ...
    printf("Package [%s] not found!\n", packageName);
    ...
    return;
}
```

2. Then it checks the version and compatibility with architecture and OS:

```
char* version = getMaxVersion(list);
if (version == NULL) {
    printf(
    "Package [%s] has not got compatible version for your system!\n",
    packageName
    );
    ...
    return;
}
```

And then, it prints the information to the console.

6.3.8 Status

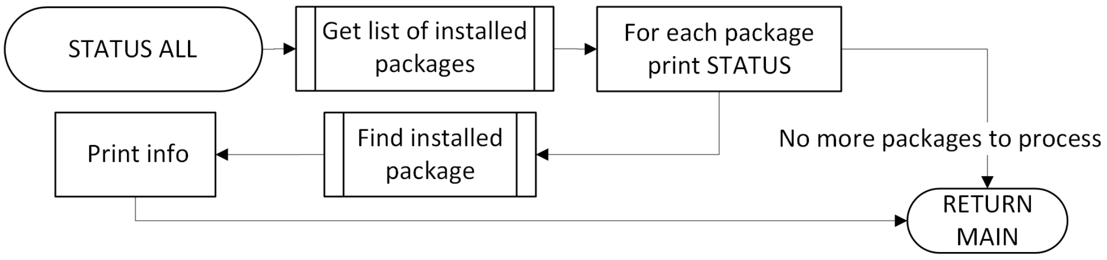
STATUS ALL
Get list of installed packages
For each package print STATUS
No more packages to process
Find installed package
Print info
RETURN MAIN

Figure 6.7. Flowchart of the package status function.

The status of a package will be printed by calling the function:

```
void printPackageStatus(char* packageName);
```

1. It checks the existence of the package:

```
DIR* directoryStream = opendir(path);
if (directoryStream == NULL) {
    printf("PACKAGE: [%s] HAS NOT BEEN INSTALLED!\n", packageName);

    return;
}
```

2. And prints it following the format:

```
PackageInfo* info = getInstalledPackageInfo(packageName);
int width = 30;
printf("PACKAGE: %-*sVERSION: %s\n", width, packageName, info->version);
```

6.4 Developer Tool

This section describes the functionality of the developer tool application.

6.4.1 Initialization

At the beginning of the application, we will do the same compilation process of a regular expression as we did in the package manager's initialization section(6.3.1)

But here, in the developer tool, we will use a different pattern to catch matches:

```
char* regexString = "(\\/opt64\\/(.+?)\\/)";
```

6.4.2 CLI Options

There are a couple of input command-line options:

- Create a metadata file for a newly compiled package
  `--meta` or `-m` option requires an argument.

- Archive a package with a prepared metadata file and create a same-named copy of the metadata for the repository server.
  `--tar` or `-t` option requires an argument.

- Print a short help manual
  `--help` or `-h` option. Argument is optional.

6.4.3 Create MetaData

Creating of a metadata file will start by calling the function:

```
void createMetaFile(char* packageName);
```

1. Foremost, we will check the existence of the package's directory and metadata file.

```
//  path = "/opt64/⟨packageName⟩"
int pkg = checkDirExistence(path);

//  path = "/opt64/⟨packageName⟩/meta/data"
metaDataFile = fopen(path, "r");
```

2. And then collect all key and corresponding values to a string that will be written into the metadata file:

```
//  Package name
strcat(metaLine, "PKG=");
strcat(metaLine, packageName);
...
fputs(metaLine, metaDataFile);
```

After these actions, the metadata file will be placed to the directory:

```
/opt64/⟨packageName⟩/meta/
```

6.4.4 Create Prepared Package

Archiving a package with a prepared metadata file and creating a same-named copy of the metadata will start by calling the function:

```
void createTarFile(char* packageName);
```

1. First, we will check the existence of the package's directory and metadata file existence and validity and read its data:

```
//  path = "/opt64/⟨packageName⟩"
int pkg = checkDirExistence(path);

//  path = "/opt64/⟨packageName⟩/meta/data"
metaDataFile = fopen(path, "r");
int isValid = isMetaFileValid(metaDataFile, pkgInfo);
```

2. Then copy the metadata file and archive the package, both with the determined name structure:

```
/tmp/sinpmtmp/archived/
    ⟨name⟩--⟨version⟩--⟨architecture⟩--⟨operating_system⟩.meta
    ⟨name⟩--⟨version⟩--⟨architecture⟩--⟨operating_system⟩.tar
```

Now, these two files are ready to be uploaded to a repository.

6.5 Encountered Problems

Some of the problems that emerged during the design and development of the project and their solutions are described in this section.

6.5.1 Working with Memory

- Freeing a memory recursively

  One of the auxiliary structures used in the project was:

  ```
  typedef struct PackageInfo {...} PackageInfo;
  ```

  Which contains a pointer to an array of similar `PackageInfo's`:

  ```
  struct PackageInfo* dependencies;
  ```

  It was not so evident at first glance how to handle this type of memory deallocating. Still, after a thorough analysis, the found solution was to use an extra recursive function in addition to the process of freeing the main structure's memory.

- Double memory allocation

  While doing the `updatePackage()` procedure, it was causing a memory leak, but the `free()` function was in place, and it was hard to know why leaks were appearing at all.

  It turned out that there were two calls of the `malloc()` function in a row. To solve that, a heuristic approach was implemented into the `installRemotePackage()` process:

```
//  Prevent malloc'ing memory twice for updatePackage()
if (packageInfo->dependenciesCount == 0
    && strlen(packageInfo->rawDependencies) > 0) {
    fillDependenciesAndCount(
        packageInfo->rawDependencies, packageInfo
    );
}
```

6.5.2 Linking Libraries to the Project

Some libraries, such as `libcurl`, could be easily installed and then linked, while others are not so easily handled, for example, `libconfig`.

It took some time to research it and finally found a working solution for adding them to `CMakeLists.txt` correctly.

6.5.3 Regular Expression cflags

Because the default regular expression type for processing a pattern was the basic regex, its execution could not find any matches and was very confusing. And the solution to that was to specify a `cflag`, more concrete to set it as `REG_EXTENDED`.

6.5.4 The Functionality of Standard Library.

Both the package manager and the developer tool require a user's input which may cause lots of problems on its own.

But the problem emerged in a different domain. The well-known `scanf()` function reads user input. Sometimes we want to get a user's input multiple times. And for some reason, the function above was rejecting to work correctly by only reading the first input but ignoring further.

The reason was that when a user enters a string and presses the `ENTER` button, the function consumes the string, but the new line character generated by pressing the `ENTER` button remains in the buffer. Hense the next call of `scanf()` never waits for a user's input because it consumes a `\n` character.

A simple solution to that was adding an extra space character before input:

```
scanf("%c", &answer);   --->    scanf(" %c", &answer);
                                       ^^^ - here is an additional
                                             space character
```

Chapter 7
Testing

This chapter describes the testing process of the implementation part of the project.

7.1 The Developer Tool

Unfortunately, most packages nowadays have lots of dependencies and dynamically linked libraries, and because of that, software developers must manually create corresponding metadata files for the very first applications. Hence there was no possibility of preparing an actual test environment.

However, due to circumstances beyond my control, it resulted in no real opportunities to test the developer tool, but it also was not the primary goal of this thesis.

7.2 The Package Manager

On the bright side, on the Linux virtual test machine, an `FTP` server has been installed and configured to the needs of testing purposes. Also, there have been prepared a lot of dummy packages with test metafiles (with the structure described above 6.2), but not with binaries or dynamically linked libraries since we were testing the correctness and availability of external data.

On the aforementioned `FTP` server, we have the root directory, which contains directories with corresponding names of test packages, each of them has at least two files with the same name but different extensions (`*.tar` and `*.meta`) describing its name, version, operating system, and architecture. So, the directory can have different versions, OS, and architecture implementations for one application.

With this infrastructure, the package manager has passed tests for the correctness of searching, downloading, installing, updating, removing, and determining package compatibility with the test system, given that an internet connection was stable throughout the whole test phase.

Chapter 8
Results

During the course of this work, many problems have emerged. A significant part of them was related to the problem of understanding the essential processes of the C language. Other parts belonged to the field of design and testing, sometimes by trials and errors.

But it allowed me to achieve the primary goal of this thesis and learn many new skills in low-level programming in C, which I did not know beforehand.

8.1 Possibilities of Further Development

What functionality of the project could be improved in the future:

- While installing the package's dependencies, we could avoid double-checking already installed dependencies in a recursive call. It may be accomplished by downloading metafiles and checking them in advance or adding them in a set structure and checking it each time.
- Add possibility to download package of a different than the last version.
- Add authorization for anonymous/registered users to access a repository.
- Improve search function by searching not as a complete match of a name, but providing that a user can write it with a mistake or the package's name may not be entirely written.
- Add functionality to the developer tool for uploading a prepared package to the repository.

Chapter 9
Conclusion

The main goal of our project was to design and implement an application for the installation, updating, and removal of packages for a Linux distribution.

Throughout many project development steps, we've been analyzing, designing, implementing, and testing.

The result concludes two applications that allow manipulating packages on the system, the package manager for their maintenance and the developer tool for their creation.

By completing the goals of this thesis, we developed the project as a deployable solution for its further integration into a Linux system without any significant risks. And there is always exists room for improvement in the project.

References

[1] FreeBSD. *Freebsd Manual Pages*. May 20, 2004.
`https://www.freebsd.org/cgi/man.cgi?query=tar&apropos=0&sektion=5&manpath=FreeBSD+7.0-RELEASE&arch=default&format=html`.

[2] archlinux.org. *Configuration File*. September 4, 2021.
`https://archlinux.org/pacman/makepkg.conf.5.html`.

[3] wikipedia.org. *Dependency hell*. March 16, 2022.
`https://en.wikipedia.org/wiki/Dependency_hell`.

[4] debian.org. *What Is A Package Manager?* 2016.
`https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html`.

[5] Machtelt Garrels. *Introduction to Linux. A Hands on Guide*. 2008.
`https://tldp.org/LDP/intro-linux/html/sect_03_01.html`.

[6] wikibooks.org. *C Programming/POSIX Reference/dirent.h*. April 16, 2020.
`https://en.wikibooks.org/wiki/C_Programming/POSIX_Reference/dirent.h`.

[7] IEEE/The Open Group. *sys/utsname.h — system name structure*. 2017.
`https://man7.org/linux/man-pages/man0/sys_utsname.h.0p.html`.

[8] curl.se. *libcurl - the multiprotocol file transfer library*. April 27, 2022.
`https://curl.se/libcurl/`.

[9] Mats Henricson, and Erik Nyquist. *Programming in C++, Rules and Recommendations*. 1992.
`https://www.doc.ic.ac.uk/lab/cplus/c++.rules/chap10.html`.

[10] IEEE/The Open Group. *dirent.h — format of directory entries*. 2017.
`https://man7.org/linux/man-pages/man0/limits.h.0p.html`.

[11] GNU. *regex.h - regular-expression-matching types*. March 22, 2021.
`https://man7.org/linux/man-pages/man3/regcomp.3.html`.

[12] Mark A. Lindner. *libconfig - A Library For Processing Structured Configuration Files*. June 20, 2021.
`https://hyperrealm.github.io/libconfig/libconfig_manual.html`.

[13] wikipedia.org. *getopt*. February 14, 2022.
`https://en.wikipedia.org/wiki/Getopt`.

[14] GNU. *getopt.h — command option parsing*. August 27, 2021.
`https://www.man7.org/linux/man-pages/man3/getopt.3.html`.

Appendix A
User Manual

The user manual for the package manager:

```
-u [ARGUMENT], --update [ARGUMENT]
    * If [ARGUMENT] provided, then the package with name [ARGUMENT]
        is going to be updated, if such package exists on the server
    * Updates all packages if no [ARGUMENT] provided
    * [ARGUMENT] is a name of a package

-i ARGUMENT, --install ARGUMENT
    * If ARGUMENT provided, then the package named ARGUMENT
        is going to be installed, if such package exists on the server
    * ARGUMENT is required here
    * ARGUMENT is a name of a package

-e ARGUMENT, --extract ARGUMENT
    * If ARGUMENT provided, then the named archive
        is going to be installed, if such package does not exist
        in the system or has an older version
    * ARGUMENT is required here
    * ARGUMENT is a path to a *.tar archive

-r ARGUMENT, --remove ARGUMENT
    * If ARGUMENT provided, then the package named ARGUMENT
        is going to be removed, if such package exists in the system
    * ARGUMENT is required here
    * ARGUMENT is a name of a package

-t [ARGUMENT], --status [ARGUMENT]
    * If [ARGUMENT] provided, then info about the package
        with name [ARGUMENT] is going to be printed out,
        if such package exists in the system
    * Prints info about all packages if no [ARGUMENT] provided
    * [ARGUMENT] is a name of a package

-s ARGUMENT, --search ARGUMENT
    * If ARGUMENT provided, then the info of package named ARGUMENT
        is going to be fetched from the server,
        if such package exists there
    * ARGUMENT is required here
    * ARGUMENT is a name of a package
```

The user manual for the developer tool:

```
-m ARGUMENT, --meta ARGUMENT
    * ARGUMENT is a name of a package
    * If ARGUMENT provided, then new META file will be created
        if such package exists in the system
    * ARGUMENT is required here

-t ARGUMENT, --tar ARGUMENT
    * ARGUMENT is a name of a package
    * If ARGUMENT provided, then TAR and META files will be created
        corresponding to architecture and OS of the machine
        if such package exists in the system
    * ARGUMENT is required here
```

Appendix B
The Project's Repositories

The following links contain the project repositories on the `CTU FEL GitLab`:

- The Package Manager
  `https://gitlab.fel.cvut.cz/shcheden/bac/-/tree/master`

- The Developer Tool
  `https://gitlab.fel.cvut.cz/shcheden/bac-dev/-/tree/master`