UD 07.
LINUX: APPLICATIONS AND SERVICES

Computer Systems
CFGS DAW

Aarón Martín Bermejo
a.martinbermejo@edu.gva.es
2022/2023
Version:221124.1352

License

Nomenclature

Throughout this unit different symbols will be used to distinguish important elements within the content. These symbols are:

TABLE OF CONTENTS

UT 07. Linux: applications and services

An application usually refers to any program, any executable file that contains code and does a certain task. Specifically in Linux (and usually in other operating systems), applications are the ones that have an interface to interact with it and are managed by a user. Examples of applications could the file explorer, text editors or even the terminal.

1. How to install applications

As you probably know, one the biggest steps to climb when using Linux for regular users is installing applications. Although, as in Windows and MacOs installation of applications is performed usually and only through an installable package/tutorial really simple, in Linux we have (at least) three different ways to install applications:

- From a package
- From a compressed file[1]
- From the source code

Obviously, this last option is the most complex and requires the knowledge necessary to compile the application, knowledge that is beyond the scope of this module.

1.1 Packages

The simplest way is to use packages, something similar to Windows msi. A package in linux includes all the information for the installation and the configuration of the application, notifying to the system of the dependencies mandatory to be able to execute it correctly. Can contain all the files required itself, instructions about where to get them or just a list of dependencies to be solved by the package manager.

> The problem is that there is no single package format. In the market there are two main formats, .deb (used by distributions like Ubuntu or Debian) and .rpm (used by distributions like OpenSuse or Fedora). That makes the developers must generate at least two types of packages to distribute their applications… if they generate a package at all.

Actually the installation of these packages is very simple, they are simply downloaded, clicked on them and the installation program is automatically launched using an application manager.

The problem is that Linux systems are very open and very dynamic systems. The versions of the applications happen really quickly and the places from which to download are very varied. This is why it is convenient to use application managers, an idea existing in Linux for many years and that comes as something like today's application stores (App Store, Google Play, etc.).

1 We will study this method in the activities

1.2 Application managers

Application managers are the programs that allow users to install applications in an “easy way”. They usually manage these difficulties:

- **Dependencies**: each program makes use of different software libraries, work on specific platforms (like JAVA for instance), etc. Those things are called program dependencies, because the program cannot run without them. Application manager retrieve those dependencies and installs them previously to the installation of the program, to simplify the installation process. Specifically important is that the application manager installs those dependencies on the specific versions that the program needs which, if performed manually, would be pretty complicated.
- **Versioning**: each program can be updated to add new features, fix bugs, etc. Those updates generate new versions of the program. The application manager retrieves the versions of the program, manages the updates, installs the latest version or a specific version if required…

Usually each distribution comes with an application manager: apt (aptitude), synaptic… You can install different application managers in your operating system, like using snap inside ubuntu, which comes by default with apt.

Of course, each type of package is associated with its own manager (usually, working in terminal mode). Today the most common format is the store (in graphical mode), but possibly the most versatile is the classic command line (the ancestor of the store).

For example, in distributions with .deb packages, the manager is called apt-get, its classic desktop version is Synaptic, and the storage mode is called Application Center.

In case of knowing the name of the package the simplest method is the first one. For example, if we want to install the VLC player, the easiest way is to open the terminal and write:

```
sudo apt-get install vlc
```

Where:

- `sudo: allows us to execute applications in superuser mode. Obviously the installation of programs is not something that can do any user, so to be able to do this it is necessary to ask for superuser credentials.`
- `apt-get: name of the package manager application.`
- `install: option of the manager program that allows the installation. Obviously there are others for removal, updating, etc.`
- `vlc: package name`

If we run, the system prompts us for the password and proceeds to the installation.

> Linux is case sensitive, so it is not the same to write in uppercase than in lower case.

If we do not know the exact name of the package, it is better to use the app store / package manager. We can access from system tools menu. The application has a search engine to locate the

package that interests us and then select it. The dependencies necessary for installation will be automatically selected.

1.3 Problem of dependencies

Dependencies usually arise a kind of problem that it's very common and uncomfortable. Colloquially it's called "dependency hell", which usually comes for this different kind of problems with dependencies:

- **Too many dependencies**: a package that has too many dependencies and/or they are too heavy may require long times of downloading, large space usage…
- **Long chains of dependencies**: package A depends on package B → package B depends on package C → C on D → D on E… this kind of issue is different from too many dependencies, since that chain can't be solved easily by the package manager and usually the user is prompted to install manually that chain of dependencies.
- **Conflicting dependencies**: package A depends on package B 1.3 and C 1.1 → package B depends on package C 0.9. If those different versions can't coexist, there's a serious problem between them.
- **Circular dependencies**: probably the worst of them, installing A → A depends on B → B depends on C → C depends on A. Therefore, unsolvable dependency chain.

This is a really common problem. For instance, npm (which is a package manager for software projects) versions rely on specific versions of nodejs. But usually there are applications that rely on different versions of nodejs… and there's the problem.

There are more type of dependency issues and usually there are solutions to them (like an isolated installation).

1.4 Repositories

In Linux, a repository is a location from where the Linux system retrieves and installs updates and applications related to the Operating system. Basically, it's a list or a collection of software hosted in a server. That collection specifies versions of the programs that host, their dependencies and the files needed to install it.

Application managers make use of the repositories to retrieve the files needed to install a program, retrieve their dependencies and download everything needed to install the software. There's not only one software repository, there are many of them and you can add repositories to your operating system.

The repository is nothing, but a collection of software, and this collection is hosted on a remote server and is downloaded and installed for either installing or updating software packages on the Linux system. In this article, we will talk about the standard and non-standard repositories, but before that let us peep into the syntax first.

Important repositories are the next ones:

- **Main**: When you install Ubuntu, this is the repository enabled by default. The main repository consists of only FOSS (free and open source software) that can be distributed freely without any restrictions. Software in this repository are fully supported by the Ubuntu developers. This is what Ubuntu will provide with security updates until your system reaches end of life.
- **Universe**: This repository also consists free and open source software but Ubuntu doesn't guarantee of regular security updates to software in this category. Software in this category are packaged and maintained by the community. The Universe repository has a vast amount of open source software and thus it enables you to have access to a huge number of software via apt package manager.
- **Multiverse**: Multiverse contains the software that is not FOSS. Due to licensing and legal issues, Ubuntu cannot enable this repository by default and cannot provide fixes and updates.

You can add those repositories to your machine. There are also other public repositories and even private ones. All of them are available to add them so your package manager is able to retrieve programs from those repositories.

1.5 AppImage

AppImage is another way to “install” applications that tries to simplify the process of it. Basically, AppImage is a format for distributing portable software on Linux without needing superuser permissions, because basically it’s not “installed”. Also it tries to be distribution-agnostic, portable and isolated. That means:

- **Distribution-agnostic**: Developers pack the application in AppImage, the app image can be used in any linux distribution instead of looking the specific package/files for the distribution you need.
- **Portable**: contains all the files that are required for the application to be used and they are mounted using FUSE, which means that it’s not really installed.
- **Isolated:** instead of putting all the files required for the application, it contains all of them so no modification of the OS or the user files.
- **No super user permissions:** because of being portable and isolated, it does not require the super user permissions as it does not need to modify OS files/folders or nothing similar.

> 📣 Not all applications can be packaged as an AppImage, since they need to make usage of extra things outside the package such as OS folders, creating specific folders somewhere, etc.

AppImages just need permission to be executed, nothing else. That’s why they are so easy for regular users, since they only have to download the AppImage application and execute it.

1.6 Snap

Snap is a software packaging and deployment system developed by Canonical for operating systems that use the Linux kernel and the systemd init system. It works among a range of Linux distributions contrary to the way of packaging applications for specific distributions.

The packages, called snaps, are self-contained applications running in a sandbox with mediated access to the host system.

1.6.1 Format

As the snap documentation site specifies[2]:

> "A snap is a SquashFS file carrying content alongside metadata to tell the system how it should be manipulated. When installed, the SquashFS file for the snap is mounted read-only at the following location:
>
> /snap/<snap name>/<revision>/
>
> [...]
>
> Applications declared in the snap become commands at:
>
> /snap/bin/<snap name>[.<app name>]
>
> That snap command file is not the actual application, but rather a link to the real application under the isolation and confinement rules of the snap's default restricted environment, plus any allowances granted to it via the interface system."

That format implies certain things:

- snaps or snap's packages are basically files using SquuasFS filesystem that contain all the files needed for the application (the app itself, dependencies, data files, etc) plus metadata.
- Installation of a snap means deploying the files to a read-only folder. That allows snaps to be stable, non-editable installation with no possibilities of external intervention on their files causing mutations or errors. They are always clean as they were defined.
- Their "executable" is a link to the real application, which will be confined in a sandbox environment, which we will talk about it later on.

1.6.2 Snapd, snap, snapcraft, snap store

The components of the snap package and deployment system are:

- snap: as said before, snaps are the packages of the system. They contain all the data needed for the application to be installed and run.

2 https://snapcraft.io/docs/snap-format

- snapd: the daemon[3] required to run snaps (download them from the store, mount them into place, confine them, run apps out of them, etc.). Snapd also includes the snap command, used to communicate with snapd (for the user to request a new snap be installed, etc.)
- snapcraft: the command/program used to build snaps and optionally upload them to the snap store
- snap store: the snap default repository where the snaps are catalogued, managed and published.

1.6.3 Confinement

Snap applications run in a sandbox environment. That means they are "trapped" inside a heavily controlled environment that can only access to the resources inside the box or to external resources only when allowed.

That isolation is called **confinement** and snap defines three levels of confinement:

- **Strict**: almost all snaps use this. Snaps under strict confinement run in complete isolation, not being allowed to access external resources such as: your files, network, external processes, external devices like audio/video etc.
- **Classic**: snaps under classic confinement have access as classic packages have. Meaning that they will be under the Linux resource protection system[4] but they will have access to external resources like files, devices, etc. Snaps published with classic confinement require manual approval to be published and to be installed they require a "--classic" command line argument to ensure users are sure about it.
- **Devmode**: basically a strictly confined snap but with access to everything for development purposes. Cannot be published to stable channels for instances and it's meant only to develop or test snaps.

Talking about confinement arises the next question: but if snaps are trapped and don't have access to anything, how do they make use of external resources if needed? The answer is what's called **interfaces.**

An interface gives access to restricted resources from outside the sandbox, such as network access, sound, video, desktop access, etc. Those interfaces need to be connected to be active, which can happen at installation or during execution. Those connections are explicit to the users, which even sometimes they need to be manually connected by them. For instance, these are the interfaces required by vlc:

```
$ snap connections vlc
Interface         Plug                  Slot              Notes
camera            vlc:camera            -                 -
desktop           vlc:desktop           :desktop          -
desktop-legacy    vlc:desktop-legacy    :desktop-legacy   -
home              vlc:home              :home             -
mount-observe     vlc:mount-observe     -                 -
```

3 We will talk about daemons/services later on this unit.
4 To be studied in next units