Sistemas de paquetes

Desarrollo Colaborativo de Aplicaciones

Introducción

Distribuyendo *software*

Como desarrolladores de *software*, tenemos un objetivo claro:

Distribuyendo *software*

Como desarrolladores de *software*, tenemos un objetivo claro:

Nuestros programas los utilizan **usuarios**. Sin embargo…

Distribuyendo *software*

Como desarrolladores de *software*, tenemos un objetivo claro:

Nuestros programas los utilizan **usuarios**. Sin embargo…

¿cómo lo distribuimos?

Distribuyendo *software*

Un aspecto clave para el éxito es que nuestros programas sean **fáciles de instalar**.

pero no todo vale

Distribuyendo *software*

Debemos tratar de evitar instalaciones como en **Windows**:

- Desconoce componentes ya instalados.
- No tiene en cuenta dependencias.
- Permite la sobreescritura de archivos ya instalados (DLL Hell).
- La propia Microsoft ya puso *Windows Installer* como alternativa.

Objetivo

Debemos distribuir nuestro *software* a través de paquetes.

- Nos permiten controlar el proceso de instalación.
- El usuario es *agnóstico* al proceso.

¿Qué es un paquete?

Un **paquete** es un archivo que contiene:

1. **Archivos del software** (binarios, bibliotecas, datos)
2. **Metadatos** (nombre, versión, descripción)
3. **Información de dependencias** (qué necesita el software)
4. **Scripts de instalación** (acciones pre/post instalación)
5. **Checksums** (verificación de integridad)

¿Qué es un paquete?

> ⚠ **Advertencia**
>
> Un paquete **no es solo un archivo comprimido**, es un archivo comprimido + scripts de automatización

Tape ARchiver

TAR

- **TAR** = Tape ARchiver (archivador en cinta)
- Formato ampliamente usado en entornos UNIX
- Extensión: `.tar`
- Diseñado originalmente para **cintas magnéticas**
- Estándar POSIX.1 y POSIX.2

> **Importante**
>
> TAR **solo empaqueta**, no comprime

TAR

Ejemplo visual:

```
archivo1.txt (100 KB)
archivo2.txt (200 KB)
archivo3.txt (150 KB)
          ↓
archivo.tar (450 KB)
```

El tamaño total es la suma de tamaños

¿Qué hace TAR?

- Agrupa archivos en uno solo
- Preserva estructura de directorios
- Mantiene metadatos UNIX

Sintaxis básica de tar

```
1 tar    ...
```

Opciones principales:

Opción	Función
`-c`	**C**rear archivo
`-x`	E**x**traer archivo
`-t`	Lis**t**ar contenido
`-v`	**V**erbose (mostrar progreso)
`-f`	**F**ile (especificar archivo)

Ejemplos prácticos de TAR

```
# Crear un archivo tar
tar -cvf proyecto.tar proyecto/

# Listar contenido sin extraer
tar -tvf proyecto.tar

# Extraer archivo
tar -xvf proyecto.tar

# Extraer en directorio específico
tar -xvf proyecto.tar -C /tmp/
```

Tip

`-cvf` es el combo más común: **C**reate + **V**erbose + **F**ile

Metadatos UNIX en TAR

TAR preserva información importante:

```
1 $ tar -tvf archivo.tar
2 drwxr-xr-x usuario/grupo     0 2025-10-27 10:00 proyecto/
3 -rw-r--r-- usuario/grupo  1234 2025-10-27 09:30 proyecto/README.md
4 -rwxr-xr-x usuario/grupo  5678 2025-10-27 09:45 proyecto/script.sh
5 lrwxrwxrwx usuario/grupo    12 2025-10-27 09:50 proyecto/link -> archivo.tx
```

- **Permisos**: `-rw-r--r--`, `drwxr-xr-x`
- **Propietario/Grupo**: `usuario/grupo`
- **Timestamps**: Fecha y hora
- **Enlaces simbólicos**: `link -> archivo.txt`

TGZ, TBZ, TXZ

Problema: TAR no comprime

```
proyecto/ (100 archivos, 50 MB)
          ↓
proyecto.tar (50 MB)  ← Mismo tamaño!
```

Solución: Combinar el comando `tar` con distintos **compresores**.

TAR + Compresores

TAR puede enlazar automáticamente con compresores:

Compresor	Opción TAR	Extensiones
gzip	`-z`	`.tar.gz`, `.tgz`
bzip2	`-j`	`.tar.bz2`, `.tbz`
xz	`-J`	`.tar.xz`, `.txz`

TAR + Compresores

```
# Crear tar comprimido con gzip
tar -czvf proyecto.tar.gz proyecto/

# Crear tar comprimido con bzip2
tar -cjvf proyecto.tar.bz2 proyecto/

# Crear tar comprimido con xz
tar -cJvf proyecto.tar.xz proyecto/
```

Comparación de compresores

gzip (`.tgz`)

- Rápido
- Compresión media
- Más usado

bzip2 (`.tbz`)

- Medio
- Mejor compresión
- Más lento

xz (`.txz`)

- Mejor compresión
- Más lento
- Más memoria
- Moderno

Comparación de compresores

Tip

Si queremos escoger un compresor, normalmente **xz** da mejores resultados. Sin embargo, actualmente **gzip** es el estándar de facto y es compatible con la gran mayoría de sistemas de paquetes.

Fortalezas de TAR

- Empaquetar múltiples archivos
- Preservar permisos y metadatos
- Comprimir (con gzip/bzip2/xz)
- Portabilidad entre sistemas UNIX

Debilidades de TAR

1. **Verificar dependencias**
   - ¿Tengo las bibliotecas necesarias?
2. **Ejecutar scripts de instalación**
   - Crear usuarios del sistema, iniciar servicios, configurar el sistema…
3. **Gestionar actualizaciones**
   - ¿Qué versión tengo instalada?, ¿Cómo actualizo?

¿Por qué TAR no basta para distribuir *software*?

TAR es un excelente archivador, pero está muy limitado como instalador

Ejemplo de instalación

```
# Descargar
wget https://ejemplo.com/miapp-1.0.tar.gz

# Extraer
tar -xzf miapp-1.0.tar.gz
cd miapp-1.0

# ¿Y ahora qué?
# ¿Dónde lo instalo?
# ¿Qué dependencias necesito?
# ¿Cómo lo desinstalo después?
# ¿Cómo actualizo?
```

Si instalas con TAR, **debes averiguar muchas cuestiones manualmente**.

Cómo se debería instalar un paquete

> “Un buen paquete es invisible: Se instala sin preguntas, Funciona sin configuración, Y se desinstala sin dejar rastro.”

Cómo nos gustaría instalar un paquete

```
1 sudo apt install <paquete>
2 sudo apt remove <paquete>
```

La solución: Gestores de paquetes

Un **gestor de paquetes** es un sistema que:

1. **Empaqueta** software en un formato estándar
2. **Gestiona** dependencias automáticamente
3. **Rastrea** qué archivos pertenecen a qué paquete
4. **Verifica** integridad con checksums/firmas
5. **Actualiza** software de forma centralizada
6. **Desinstala** limpiamente sin dejar residuos

Ecosistemas de paquetes

Sistema	Formato	Distribuciones
APT/dpkg	`.deb`	Debian, Ubuntu, Mint
RPM/DNF	`.rpm`	Red Hat, Fedora, openSUSE
Pacman	`.pkg.tar.zst`	Arch Linux
Portage	Source-based	Gentoo

Ecosistemas de paquetes

> **Nota**
>
> En este tema, nos centramos en el sistema **.deb**, el formato más popular históricamente. En prácticas también veremos cómo hacerlo con **flatpak**.

Paquetes .deb

El sistema de paquetes `.deb`

- Sistema creado por `Debian`.
- Compatible con otras *distros* de *Linux*: Ubuntu, Linux Mint… etc.
- Actualmente es de los sistemas **más usados** para la distribución de paquetes.
- Su éxito radica en la **simplicidad** de gestión y la **claridad en el control**.

Estructura interna de un .deb

Un archivo `.deb` es un **archivo AR** que contiene:

```
mi-paquete.deb
├── debian-binary          # Versión del formato (2.0)
├── control.tar.xz         # Metadatos del paquete
│   ├── control            # Información principal
│   ├── md5sums            # Checksums de archivos
│   ├── postinst           # Script post-instalación
│   ├── preinst            # Script pre-instalación
│   ├── prerm              # Script pre-desinstalación
│   └── postrm             # Script post-desinstalación
└── data.tar.xz            # Archivos a instalar
    └── usr/
        ├── bin/
        ├── lib/
        └── share/
```

Base de datos de dpkg

Todo paquete instalado se registra en:

```
/var/lib/dpkg/
├── info/
│   ├── flappybirddca.list          # Lista de archivos
│   ├── flappybirddca.md5sums       # Checksums
│   └── flappybirddca.postinst      # Scripts
├── status                          # Estado de paquetes
└── available                       # Paquetes disponibles
```

Nota

Gracias a esta lista, **dpkg** sabe qué desinstalar cuando queremos borrar un paquete.

Anatomía de `debian/`

Todos los paquetes de Debian se empiezan construyendo a partir de una carpeta clave: `debian/`

Anatomía de `debian/`

Una carpeta `debian/` es un **directorio de metadatos**.

Éste contiene lo siguiente:

- **Información del paquete**: control
- **Instrucciones de construcción** (rules)
- **Historial de versiones** (changelog)
- **Scripts de mantenimiento** (postinst, prerm, etc.)
- **Licencias y *copyright*** (copyright)
- **Configuración de debhelper** (compat)

Anatomía de `debian/`

El fichero `control`:

```
1  Source: flappybirddca
2  Section: games
3  Priority: optional
4  Maintainer: antoniorv6 <antoniorv6@correazo.es>
5  Build-Depends: debhelper-compat (= 13), g++, libgl1-mesa-dev, libx11-dev
6  Standards-Version: 4.6.0
7  Homepage: https://github.com/antoniorv6/flappy-bird-dca
8  Rules-Requires-Root: no
9
10 Package: flappybirddca
11 Architecture: any
12 Depends: ${shlibs:Depends}, ${misc:Depends}
13 Description: Flappy Bird DCA
14   En realidad es un Flappy Bird roto, pero vamos a hacer como que funciona.
```

Anatomía de `debian/`

Campos importantes:

Sección Source (construcción):

- **Build-Depends**: Dependencias necesarias para compilar
- **Standards-Version**: Versión de políticas Debian

Sección Package (instalación):

- **Architecture**: `any` (compilado) o `all` (*scripts*)
- **Depends**: Dependencias en tiempo de ejecución
- **Description**: Primera línea = resumen, resto = detalle

Anatomía de `debian/`

```
1 Package: flappybirddca
2 Architecture: any
3 Depends: ${shlibs:Depends}, ${misc:Depends}
4 Description: Flappy Bird DCA
5   En realidad es un Flappy Bird roto, pero vamos a hacer como que funciona.
```

Tip

¿Qué significan estas variables en las dependencias?

Anatomía de `debian/`

`debian/rules` es el `Makefile` de nuestro paquete. Por defecto, contiene:

```
1  #!/usr/bin/make -f
2
3  # Versión mínima usando debhelper
4  %:
5      dh $@
```

Anatomía de `debian/`

`debian/rules` es el `Makefile` de nuestro paquete. Por defecto, contiene:

```
1  #!/usr/bin/make -f
2
3  # Versión mínima usando debhelper
4  %:
5      dh $@
6
7  override_dh_auto_install:
8    $(MAKE) DESTDIR=$$(pwd)/debian/flappybirddca PREFIX=/usr APP_NAME=flappyb
```

Es posible sobreescribir procesos automáticos de `dh` para ejecutar nuestros *scripts* o reglas del `Makefile`.

Anatomía de `debian/`

debhelper (dh) automatiza:

- Compilación
- Instalación con DESTDIR
- Generación de scripts de mantenimiento
- Cálculo de dependencias
- Compresión de documentación

Anatomía de `debian/`

Ficheros adicionales:

- `preinst`: Script que ejecuta código ántes de la instalación del paquete.
- `postinst`: Script que ejecuta código después de la instalación del paquete.
- `prerm`: Script que ejecuta código ántes de la eliminación del paquete.

Anatomía de `debian/`

Ficheros adicionales:

- `changelog`: Histórico de versiones del paquete.
- `copyright`: Fichero de licencia de la aplicación.

Anatomía de `debian/`

`debhelper` tiene *scripts* para incrementar la versión de los paquetes y actualizarlos en el `changelog`

```
1 dch -i  # Incrementa versión y abre editor
2 dch "Fix memory leak"  # Añade entrada
```

De código fuente a paquete

Para crear un paquete .deb, debemos tener claro el proceso que hará el instalador de paquetes.

De código fuente a paquete

El instalador realiza los siguientes pasos:

1. Extrae los metadatos de `control.tar.xz`.
2. Verifica dependencias.
3. Ejecuta *scripts de preinstalación*.
4. **Mapea la estructura de directorios de `data.tar.xz`.**
5. Copia los ficheros del paqute siguiendo ese *mapping*.
6. Ejecuta *scripts de postinstalación*.

De código fuente a paquete

El paso 4 es importante, ya que implica que el paquete **contiene una réplica del sistema de directorios de Linux** para mapear los ficheros del programa.

De código fuente a paquete

Es decir, dentro del paquete deberá haber una carpeta (con el nombre del mismo paquete) que **contenga la jerarquía FHS de Linux**.

De código fuente a paquete

Esta estructura **no es arbitraria**

- Un ejecutable **debe** ir en `/usr/bin/`
- Una biblioteca **debe** ir en `/usr/lib/`
- La documentación **debe** ir en `/usr/share/doc/`
- Los iconos **deben** ir en `/usr/share`

De código fuente a paquete

Para empaquetar nuestro código fuente, necesitamos un `Makefile`.

De código fuente a paquete

Para empaquetar nuestro código fuente, necesitamos un `Makefile`.

> El sistema de empaquetado `.deb` **exige que haya un *upstream Makefile*** para empaquetar, ya que lo usará durante el proceso de construcción del paquete.

De código fuente a paquete

Un Makefile para empaquetado debe contener tres reglas:

- `all` para la compilación/construcción del binario ejecutable.
- `clean` para borrar ficheros intermedios de compilación/construcción.
- `install` para crear el *staging* intermedio.

Tip

Estas condiciones permiten que *cualquier sistema* pueda empaquetar tu software

De código fuente a paquete

```
1 PREFIX ?= /usr
2 DESTDIR ?=
3
4 install:
5     install -D -m 0755 hello $(DESTDIR)$(PREFIX)/bin/hello
6     install -D -m 0644 hello.1 $(DESTDIR)$(PREFIX)/share/man/man1/hello.1
```

De código fuente a paquete

- `PREFIX`: Dónde se instala normalmente (`/usr`)
- `DESTDIR`: Directorio temporal para construcción del paquete

De código fuente a paquete

DESTDIR permite “instalación en seco” sin tocar tu sistema

De código fuente a paquete

DESTDIR permite “instalación en seco” sin tocar tu sistema

> **Importante**
>
> Nunca debemos insertar *manualmente* en `usr/bin`, ya que cualquier fallo, error o problema contaminará el sistema. Asímismo, las eliminaciones también deberán ser *manuales*.

De código fuente a paquete

DESTDIR permite “instalación en seco” sin tocar tu sistema

> ⚠ **Advertencia**
>
> Esto no quiere decir que nuestro programa no acabe con un ejecutable en `usr/bin`, pero será el gestor de paquetes quien controle esa instalación, no nosotros.

De código fuente a paquete

Como ya hemos dicho, el *software* se debe ubicar en una carpeta `<mi_paquete>` dentro del directorio `debian/` y debe respetar el **HFS**.

Ruta	Contenido
`/usr/bin/`	Ejecutables de usuario
`/usr/lib/`	Bibliotecas compartidas
`/usr/share/`	Archivos ind. de arquitectura
`/usr/share/doc/`	Documentación

De código fuente a paquete

Ejemplo de `Makefile`:

```
1  APP_NAME := game
2  PREFIX ?= /usr/local
3  BINDIR := $(PREFIX)/bin
4  LIBDIR := $(PREFIX)/lib/$(APP_NAME)
5  DATADIR := $(PREFIX)/share/$(APP_NAME)
6
7  install: $(BIN_DIR)/$(APP_NAME)
8      install -d $(DESTDIR)$(LIBDIR)
9      install -m 0755 $(BIN_DIR)/$(APP_NAME) $(DESTDIR)$(BINDIR)/$(APP_NAME)
10     install -d $(DESTDIR)$(DATADIR)/assets
11   cp -r $(ASSETS_DIR)/* $(DESTDIR)$(DATADIR)/assets/
```

De código fuente a paquete

¿Por qué usamos `install` en vez de `cp`?

De código fuente a paquete

Para crear la carpeta `debian/` y el `.tgz` de origen, podemos usar la herramienta `dh_make`

```
1 dh_make --createorig --single --yes
```

De código fuente a paquete

Para crear la carpeta `debian/` y el `.tgz` de origen, podemos usar la herramienta `dh_make`

```
1 dh_make --createorig --single --yes
```

> ⚠ **Advertencia**
>
> `dh_make` trata siempre de buscar el **nombre del paquete y la versión** en el directorio de origen. Por lo tanto, lo suyo es que la carpeta donde se va a crear el paquete se llame `<nombre_paquete>-<version>`

De código fuente a paquete

A partir de aquí, deberemos modificar los ficheros de:

1. `control` con los metadatos del proyecto.
2. `rules` para utilizar nuestro `Makefile` con las variables personalizadas.
3. `changelog` con la información pertinente de las actualizaciones.
4. `preinst`, `postinst`, `prerm` por si necesitamos que se ejecuten *scripts* adicionales a la instalación.

De código fuente a paquete

Una vez montada la configuración de `debian/` construimos nuestro paquete con:

```
1 dpkg-buildpackage -us -uc -b
```

- `-us`: No firmar .changes
- `-uc`: No firmar .buildinfo
- `-b`: Solo binario (no .dsc)

De código fuente a paquete

Flujo de construcción del paquete:

De código fuente a paquete

De forma simplificada:

1. Se buscan e instalan las dependencias en el SO.
2. Se llama a `clean` con el proceso automático `dh_auto_clean`.
3. Se construye el proyecto con `dh_auto_build`.
4. Se crea el *staging* del paquete con `dh_auto_install`.
5. Se generan *scripts de mantenimiento*.
6. Se empaqueta la app en el fichero `.deb`.

De código fuente a paquete

> ⚠ **Advertencia**
>
> Ten en cuenta que `dh`, en sus funciones automáticas, llamará a `make clean`, `make all` y `make install` por defecto, sin ningún parámetro. Si quieres modificar ese comportamiento, debes actualizar el fichero `rules`.

De código fuente a paquete

> **Nota**
>
> Si todo funciona adecuadamente, el fichero `.deb` se habrá generado en el **directorio padre** de tu proyecto. Este comportamiento **no se puede modificar**, pero sí puedes automatizar su localización en el mismo `Makefile`.

De código fuente a paquete

Para instalar tu paquete, utiliza:

```
1 dpkg -i <tu-paquete>.deb
```

De código fuente a paquete

Si quieres verificar que tu paquete cumple con los estándares de Debian, es conveniente que uses la herramienta de `lintian`.

```
1 lintian <mi-paquete>.deb
```

¿Esto funciona con `sudo apt install`?

¿Esto funciona con `sudo apt install`?

Nota

`apt install` y `apt-get install` llaman internamente a `dpkg -i` si el paquete es un `.deb`

Más allá del .deb

Hasta ahora, hemos visto cómo gestionar los paquetes `.deb` como instalables en distros Linux.

¿Pero eso cubre todas las distros de Linux?

Más allá del .deb

En sistemas derivados de **Red Hat**, como **Fedora** u **openSUSE** usan un formato de archivos diferente: el RPM.

> ⚠ **Advertencia**
>
> No vamos a ver en la asignatura cómo crear un paquete RPM, ya que existen herramientas de conversión para evitar producir varias ediciones de un paquete. Si, embargo, ten en cuenta que producir un paquete puede ser, en ocasiones, un proceso con coste **exponencial**.

Alien

`alien` es una herramienta de terminal que nos permite convertir ficheros `tgz`, `deb` y `rpm` entre ellos.

Alien

`alien` es una herramienta de terminal que nos permite convertir ficheros `tgz`, `deb` y `rpm` entre ellos.

- Usa permisos de superusuario.
- Recibe un paquete como parámetro (`.tgz`, `.deb` o `.rpm`).
- Recibe un argumento: `-t`, `-r` y `-d`.
- Este argumento es el **objetivo** del comando.

Alien

> **Nota**
>
> También se puede ejecutar con la opción de `-c` para que `alien` trate de convertir los *scripts* de instalación.

CD con GitHub Actions

Recordando qué es CD

El CD es una práctica de automatización del despliegue de la aplicación tras producir una nueva versión.

Recordando qué es CD

El CD es una práctica de automatización del despliegue de la aplicación tras producir una nueva versión.

> **Nota**
>
> Recuerda que en el Tema 2 vimos las diferentes formas en las que gestionamos el versionado de un proyecto *software* a través de las distintas estrategias de integración con versionado de *software*. Esto siempre ejecutará el **CD** de nuestro sistema.

GitHub Actions

Podemos aprovechar las **GitHub Actions** para integrar CD en nuestro flujo de trabajo.

- Integración nativa con GitHub (sin configuración adicional).
- Muy flexible y personalizable con YAML.
- Gran ecosistema de acciones públicas reutilizables.
- Permite realizar todo el proceso de CI/CD con facilidad.

Workflows

En GitHub Actions definimos una serie de workflows que son los responsables de definir un conjunto de acciones automatizadas para realizar un proceso.

> Nuestros workflows se definen en la carpeta `.github/workflows/` de nuestro proyecto.

```yaml
name: Build Debian Package
on:
  push:
    tags:
      - 'v*.*.*'
jobs:
  build-deb:
    runs-on: ubuntu-latest
    steps:
      - name: Build Debian package with make dist
        run: |
          make dist
          echo "Build completed!"
```

Jobs: Grupo de fases que se ejecutan en una misma máquina y con un objetivo concreto.

github/workflows/push_validation.yaml

```yaml
name: Build Debian Package
on:
  push:
    tags:
      - 'v*.*.*'
jobs:
  build-deb:
    runs-on: ubuntu-latest
    steps:
      - name: Build Debian package with make dist
        run: |
          make dist
          echo "Build completed!"
```

Steps: Conjunto de acciones, como correr un test o instalar una dependencia.

github/workflows/build-dist.yaml

```yaml
name: Build Debian Package
on:
  push:
    tags:
      - 'v*.*.*'
jobs:
  build-deb:
    runs-on: ubuntu-latest
    steps:
      - name: Build Debian package with make dist
        run: |
          make dist
          echo "Build completed!"
```

Action: Paso reutilizable publicado por la comunidad o creado a medida.

github/workflows/build-dist.yaml

```yaml
name: Build Debian Package
on:
  push:
    tags:
      - 'v*.*.*'
jobs:
  build-deb:
    runs-on: ubuntu-latest
    steps:
      - name: Build Debian package with make dist
        run: |
          make dist
          echo "Build completed!"
```

`Triggers:` Indica cuando debe ejecutarse este flujo de trabajo.

github/workflows/build-dist.yaml

```yaml
name: Build Debian Package
on:
  push:
    tags:
      - 'v*.*.*'
jobs:
  build-deb:
    runs-on: ubuntu-latest
    steps:
      - name: Build Debian package with make dist
        run: |
          make dist
          echo "Build completed!"
```

Los `triggers` son *el núcleo* de un buen CI/CD. Nos dan un control detallado de cuando y cómo debe hacerse la ejecución.

```yaml
on:
  push:
    branches: [main]      # Cuando se haga un push a main
  pull_request:
    branches: [main]      # Cuando se haga una pull request a main
  schedule:
    - cron: '0 0 * * 0'   # Cada domingo
  workflow_dispatch:      # Ejecución manual desde la interfaz
```

CD con GitHub Actions

GitHub Actions permite generar **artefactos** durante el proceso de ejecución.

CD con GitHub Actions

GitHub Actions permite generar **artefactos** durante el proceso de ejecución.

Es posible recuperarlos y publicarlos en una *release*.

CD con GitHub Actions

```yaml
name: Build Debian Package

on:
  workflow_dispatch:
  push:
    tags:
      - 'v*.*.*'  # Se activa con etiquetas como v1.0.0, v2.1.3, etc.

jobs:
  build-deb:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4
        with:
          fetch-depth: 0  # Necesario para obtener todo el historial y las

      - name: Install build dependencies
```

CD con GitHub Actions

> ⚠ **Advertencia**
>
> Este código solamente se ejecuta si hacemos un **push** con una etiqueta.

> **Tip**
>
> Recuerda que, para poner una etiqueta, tienes el comando de `git tag`. O, cuando creas una *release* añadir la nueva etiqueta.

Flujo recomendado de versionado

1. Desarrollar nueva feature
2. Actualizar `debian/changelog`: `dch -i`
3. Construir paquete
4. Probar exhaustivamente
5. Etiquetar en git

Sistemas modernos de paquetería

Ventajas de .deb y .rpm

1. **Eficiencia de espacio**: Las bibliotecas se comparten entre aplicaciones
2. **Seguridad (relativa)**: Mantenidos y probados por la distribución
3. **Sistema de dependencias**: Instalación automática de requisitos
4. **Integración perfecta**: Siguen los estándares del sistema

Problema #1: Seguridad comprometida

Instalación con privilegios de superusuario

- Los paquetes se instalan con todos los permisos del sistema
- Acceso total durante la instalación
- Los paquetes de terceros pueden contener malware
- No hay sandbox ni aislamiento

Problema #1: Seguridad comprometida

Precaución

Un paquete malicioso tiene control total sobre tu sistema durante la instalación

Problema #2: El infierno de dependencias

El problema de las bibliotecas compartidas

- Solo UNA versión de cada biblioteca en el sistema
- Todas las aplicaciones deben usar la misma versión
- Conflictos entre versiones antiguas y nuevas

Problema #2: El infierno de dependencias

Consecuencias:

- Instalación de paquetes antiguos puede romper el sistema
- Actualizar una biblioteca puede hacer que otras apps dejen de funcionar
- “Dependency hell” = sistema roto

Problema #3: 13000 paquetes

Complejidad de mantenimiento:

- Crear paquetes para cada arquitectura (x86, ARM, 32/64 bits)
- Empaquetar para cada versión de cada distribución
- Resultado: **decenas de paquetes** para una sola aplicación

Problema #3: 13000 paquetes

> **Tip**
>
> Las distribuciones, por lo general, empaquetan ellas mismas el software, no los desarrolladores originales.

Estos problemas han motivado a la comunidad de desarrollo a crear **nuevos sistemas de paquetes** capaces de aplacarlos.

Flatpak: la solución moderna

Filosofía: Empaqueta la aplicación + dependencias **en un solo bundle**

Flatpak: la solución moderna

Características principales:

- Usa **runtimes** compartidos (ej: KDE runtime, GNOME runtime)
- Solo para aplicaciones gráficas
- Repositorio principal: **Flathub**
- Instalable con GNOME Software, Discover, o línea de comandos

Ventajas de Flatpak

1. **Más seguro**:
   - Instalación sin privilegios de administrador (*sandbox*).
2. **Universal**:
   - Un paquete funciona en todas las distribuciones.
3. **Actualizaciones independientes**:
   - No dependes de tu distro para actualizaciones.
4. **Sin infierno de dependencias**:
   - Apps antiguas no rompen tu sistema.

Ventajas de Flatpak

Importante

Por desgracia, Flatpak es más lento de empaquetar y, por lo general, **los paquetes consumen mucho más espacio en disco**

Snap: La alternativa de Canonical

Similar a Flatpak, pero con diferencias clave:

- **Auto-actualización** automática
- **Containerización** completa
- Soporte para CLI y herramientas de servidor
- Canales de actualización (stable, beta, edge)

Snap: La alternativa de Canonical

Importante

Por desgracia, *snap* solo se puede usar a través de un servidor propietario: la *Snap Store* y tiene problemas con la personalización. Por otro lado, también es lento para arrancar.