9. Instalación Programas

9.1. Introducción

Introducción

Una de las tareas habituales que debe realizar el administrador de un sistema es la instalación (conviene distinguir entre tres términos similares: instalación, configuración y actualización, la primera indica que es la primera vez que se carga y prepara una aplicación para su uso; la segunda, que esa aplicación ya cargada, es adaptada para un sistema concreto; la tercera es que una aplicación instalada y configurada se está mejorando de alguna manera) del software. Evidentemente el tipo de software y su frecuencia de actualización dependerán en gran medida de las necesidades de sus usuarios.

En los sistemas multitarea y multiusuario, como lo es UNIX / LINUX, la instalación de aplicaciones se complica más que en otros sistemas por distintas causas:

- Puede haber usuarios diferentes con necesidades diferentes respecto de una misma aplicación.
- Las aplicaciones requieren más recursos del sistema, como disco y CPU, por lo que el administrador deberá asegurar que la instalación de la aplicación no pone en riesgo la funcionalidad del sistema. Esto se ve agravado al poder ejecutarse varias copias de la aplicación al mismo tiempo, además puede que algunas aplicaciones interfieran sobre otras, lo cual también debe ser comprobado.
- Por último, puede que la actualización de la aplicación requiera una del sistema, por lo que su uso sea incompatible.

Por eso las responsabilidades del administrador aumentan en el sentido de:

- Arrancar y parar el sistema cuando se requiera, de forma que la suspensión del mismo afecte lo menos posible a los usuarios.
- Comprobar que hay suficiente espacio en memoria y disco, para que el sistema funcione de forma óptima.
- Proteger al sistema de “incursiones” no autorizadas y de posibles acciones destructivas.
- Configurar el sistema para que el mayor número posible de usuarios tenga acceso óptimo a los recursos hardware y software.

Vistas las responsabilidades del administrador, ya estamos en disposición de describir como se instalan los paquetes de software.

Clásicamente, un programa se instala igual que lo podría hacer cualquier usuario: si nos dan los fuentes, compilándolos (ver la herramienta `make` en la siguiente sección) y produciendo los ejecutables para esa máquina en sitios predeterminados, para los que tendremos que actualizar el `path` del usuario. Además, todos los ficheros necesarios (fuentes, ejecutables y documentación sobre el software y su instalación) pueden (suele ser así) estar empaquetados y comprimidos (extensiones `tar` y `gz`), por lo que se suelen acompañar de una macro de instalación (que puede ser el propio `make`), por lo que el fichero `make` (por defecto `makefile`) puede llegar a ser muy complejo.

Por eso, en la siguiente sección, aunque no sea objetivo de este curso, se repasa la construcción de un programa UNIX. La desempaquetación y descompresión ya la hemos revisado en el capítulo de copias de seguridad.

Después, en un siguiente apartado, veremos como se instalan paquetes, sobre todo en sistemas Linux, a través de un nuevo método de instalación más estructurado y ordenado que es el RPM (Administrador de paquetes Red-Hat), que no sólo permite instalar aplicaciones, sino que a través de una base de datos, nos permitirá desinstalarla, actualizarla (sin perder nuestros ficheros de configuración) y verificar si esa aplicación está instalada en nuestro sistema o no.

9.2. Construcción

Construcción

Aunque, como hemos dicho, esta sección pertenezca a un curso de programación y no de administración UNIX, antes de poder instalar un programa conviene saber o repasar como realizaríamos nosotros un programa para que otros usuarios lo pudieran usar en sus sistemas. Para ello revisaremos herramientas de desarrollo de aplicaciones como son el compilador de C, la macro de compilación automatizada make y la programación modular a través de librerías.

Herramientas usuales de compilación

Una vez que tenemos escrito el programa en C, debemos convertirlo en un programa ejecutable, al primero se le llama código fuente y al segundo código objeto. La herramienta que se utiliza en sistemas UNIX para realizar esta conversión es el cc (en sistemas `gnu` el *`gcc/g++`*). Este "compilador" (realmente es un `front-end` que lanza varios programas) realiza automáticamente toda la cadena de operaciones para producir un fichero ejecutable:

1. Llama al preprocesador `cpp` y produce un `programa.i` si el programa fuente se llama `programa.c`. Los ficheros cabecera para el preprocesador terminan en "`.h`".
2. Lo compila (`comp` es el compilador) produciendo un fichero en ensamblador `programa.s` o `programa.S`. El compilador siempre espera que se le dé un nombre de fichero que contenga un programa fuente de lenguaje C, para distinguir estos ficheros de código fuente del resto de ficheros del sistema, es obligatorio que todos terminen en "`.c`". Si no hacemos esto, el compilador nos responderá con algo parecido a: "`file not recognized: File format not recognized`". Otras terminaciones comunes son: `.C, .cc, y .cxx` para ficheros en lenguaje c++.
3. Ensambla (`as` es el ensamblador) el programa y produce un fichero objeto `programa.o`. No sólo se pueden usar ficheros objeto producidos desde código C, podemos utilizar cualquier otro lenguaje de bajo o alto nivel, desde ensamblador a PASCAL.
4. Lo enlaza (*linka*), `ld` es el *linker*, y produce el ejecutable `a.out`. En este punto es donde podemos utilizar varios ficheros objeto para producir el ejecutable.
5. Si usamos la opción `-g` estamos introduciendo una serie de elementos de compilación para que el programa ejecutable pueda ser usado con el depurador `gdb`.
6. Todos los ficheros intermedios son escritos en el directorio `/tmp` y después eliminados.

Como cualquier otro comando del sistema, el formato para ejecutarlo será `gcc [-opciones] y argumentos`. La forma más sencilla de ejecutarlo para producir un fichero ejecutable con nombre `a.out` sería la siguiente (en un sistema gnu):

```
gcc programa.c
```

Las opciones usualmente se colocan delante del nombre del fichero fuente y deberán estar separadas, por ejemplo, no es lo mismo poner `gcc -dr` que `gcc -d -r`. Las opciones están divididas en varios grupos: globales, del lenguaje, de alerta, de depuración, de optimización, de preprocesado, de ensamblador o linkador, de directorios y de dependencias del hardware.

A continuación aparecen las opciones más habituales (se pueden encontrar con `man gcc`):

Generales (controlan la cadena de ejecución de programas):

- `-c` No se llama al linker y se produce un código objeto.
- `-E` Ejecuta sólo el preprocesador y produce código fuente C.
- `-S` No se llama al programa ensamblador, por tanto genera sólo código ensamblador.
- `-o nombre` Se le indica el nombre del fichero ejecutable.

Del lenguaje:

- `-ansi` Compila siguiendo las reglas del ANSI C.

Del linker:

- `-l libreria` Incluye la librería *libreria*.

De directorios:

- `-Ldirectorio` Añade otro `directorio` de librerías donde se buscará lo dado por `-l` (anterior opción).
- `-Idirectorio` Añade el `directorio` a la lista de directorios donde encontar i*ncludes* del preprocesador.

De alerta:

- `-w` Inhibe los mensajes de *warning*. Éstos indicarán algo que no es un error pero que hay que tener en cuenta.

De depuración:

- `-g` Produce información para poder utilizar alguno de los depuradores (*debuggers*) del sistema como `gdb`.

Si el programa es ejecutado con el *debugger* del sistema: `gdb ejecutable`, su transcurso se podrá controlar con acciones típicas como: ejecución línea a línea, ejecución hasta un punto de ruptura, colocación de estos puntos, visualización de los valores de las variables, etc. Para más información se puede utilizar el comando interno `help`. También se puede utilizar el depurador para obtener información de los ficheros `core` del sistema. Cuando un programa se aborta en ejecución se produce un fichero de nombre `core` que puede ser analizado con `gdb core`.

Otra herramienta interesante cuando se hacen grandes programas que están compuestos de varios ficheros es `make`. Ésta se utiliza con un fichero de comandos (`makefile`) donde decimos de que ficheros está compuesto nuestro programa, que librerías utiliza y como se pueden obtener los ejecutables. Hay que recordar que un programa en C puede estar compuesto de varios ficheros, obligatoriamente uno de ellos tendrá la función `main()` y los otros serán otras funciones ya compiladas con la opción `-c`. La sintaxis sería:

```
make [-f fichero de make] [-opciones] [objetivos]
```

La ventaja de utilizar el `make`, es que nos compilará sólo los ficheros que sean necesarios según la fecha del último cambio y que no tendremos que poner una complicada línea de comando para realizar la compilación o realizar una macro, sólo *make*, el cual leerá el fichero de órdenes `makefile` (también `GNUmakefile` y `Makefile`), donde encontrará como hacer la compilación.

El fichero de órdenes se compondrá de una serie de *targets* (objetivos) a cumplir (indicados para el usuario) y de la descripción de como realizarlos (indicación para la máquina), las líneas de realización siempre empiezan con el carácter tabulador (indicado abajo como tab), pudiendo haber varias para un solo objetivo.

También se puede indicar un objetivo especial que es el *clean* para borrar aquellos ficheros no deseados (este sólo se ejecutará si hacemos *make clean*).

```
programa : main.o   lib_uno.o   lib_dos.o
tab      gcc -o programa main.o lib_uno.o  lib_dos.o
main.o : main.c
tab      gcc -c main.c
lib_uno.o : lib_uno.c
tab      gcc -c lib_uno.c
lib_dos.o : lib_dos.c    incluido.c
tab      gcc -c lib_dos.c
clean:
tab      rm core
tab      rm *.o
```

En este ejemplo, el fichero `makefile` (nombre por defecto) estará compuesto de cinco objetivos: programa, *main.o*, *lib_uno.o, lib_dos.o y clean*. El primer objetivo es el fichero ejecutable `programa` cuyas dependencias aparecen a continuación de los ":". En la siguiente línea (empieza por tabulador) se dice a la máquina como se puede obtener, en este caso compilando e incluyendo los ficheros objeto. A continuación se indica como obtener estos objetivos secundarios, en este caso compilando con la opción `-c` para producir ficheros objeto. Se observa que el objetivo `lib_dos.o` tiene dos dependencias: `lib_dos.c` e `incluido.c`, ya que `lib_dos.c` tiene una instrucción de tipo `#include incluido.c`. Por último está el objetivo especial `clean` que se suele usar para borrar ficheros no deseados como los `core` o los `*.o`.

Otro objetivo también bastante utilizado es:

```
print: dependencias
tab    lp *.c
```

que sirve para imprimir los ficheros que interesen. Como norma general, se puede poner un objetivo que sea una combinación de comandos, que sólo se ejecutarán cuando se haga explícitamente `make objetivo`.

En algunos sistemas (no en nuestro caso) existen un par de comandos de ayuda a la hora de programar en c. El primero de ellos es `lint`, que chequea los errores de sintaxis y da algunos consejos sobre portabilidad del código. El otro es `cb` (embellecedor de c) que sangrará adecuadamente el programa y pondrá llaves si es necesario.

Programación separada (modular)

Hasta ahora hemos visto que todos nuestros programas estaban en un fichero que se edita, compila y ejecuta. Pero normalmente, en programas grandes, esto no se hace así, sino que se construye de forma modular en varios ficheros, aplicándose el principio de "divide y vencerás", ya que los módulos del programa serán más fáciles de entender y depurar (algo parecido a la división de un programa en funciones pero a otro nivel más abstracto). Con esta forma de trabajar conseguimos algunas ventajas:

1. Obviamente los módulos tienen una extensión menor que el programa completo. Por lo tanto, éstos serán más fáciles de manejar.
2. Cada módulo se puede compilar por separado lo cual será más rápido.
3. Cada módulo será más fácil de depurar por separado, ya que no se tendrán que tener en cuenta influencias externas.
4. La división del trabajo entre varios programadores es más sencilla y limpia de realizar.

Esto conlleva que se tengan que aplicar (conveniente no obligatorio) ciertos criterios a la hora de construir ese programa utilizando la estructura en árbol de directorios y ficheros:

1. Se puede utilizar un directorio (en vez de un fichero como antes) para contener los ficheros de los que va a estar constituido el programa.
2. Dentro de ese directorio general se puede crear varios subdirectorios donde se sepa que se va a encontrar lo que estamos buscando, como por ejemplo:
    1. Un directorio para los ficheros fuente.
    2. Un directorio para los ficheros de cabecera del preprocesado (normalmente "`include`"). Ver punto 4. Los includes del sistema están en el directorio `/usr/include`.
    3. Un directorio de librerías, normalmente "lib".
    4. Un directorio de ejecutables, normalmente "bin".
    5. Un directorio de documentación, normalmente "doc".
3. Los módulos tienen que ser construidos (división del programa) teniendo en cuenta principios semánticos (significado) y ofreciendo servicios a otros módulos externos (cajas negras), de tal manera que se garantice el perfecto funcionamiento de los mismos de forma aislada (gran parte de los inconvenientes del código dependiente se pueden solventar con la compilación condicionada que nos proporciona el preprocesador, referencias más amplias de él las podemos encontrar en el libro de Kernigham). También se deberá tener en cuenta que:
    1. Hay partes dependientes del hardware que deben ser señaladas como tal. De hecho, por definición no son transportables a otros sistemas y deben estar separadas del resto del programa.
    2. Otras dependerán de algo específico como llamadas a un sistema operativo concreto y deberán ser tratadas de la misma forma.
4. Los ficheros de cabecera `*.h` están destinados normalmente a contener las definiciones comunes a varios módulos. En ellos

suelen aparecer distintos tipos de información:
1. Definición de constantes.
2. Definición de tipos de datos.
3. Definición de prototipos de funciones.

Suele ser una mala práctica de programación incluir las propias definiciones (reservas de espacio) de variables.

Gestión de librerías en UNIX

Anteriormente se ha comentado que podemos incluir en nuestro programa, código objeto realizado en otros lenguajes o en el mismo C, así, la línea de compilación que veíamos en el anterior apartado podría complicarse:

```
gcc modulo1.c modulo2.c prepro.i programa.c objeto.o -o eje -lm
```

donde hemos incluido al compilar dos módulos de código fuente C, un fichero de preprocesado, el programa principal fuente, un código objeto (no necesariamente C) y una librería estática (podemos considerar una librería como un conjunto de ficheros que contienen código objeto, las propiedades de esos ficheros son asimiladas en la propia librería que los mantiene a través de un registro índice), en este caso la librería matemática (para ello deberemos haber usado en alguna parte del código un `#include math.h`).

Una pregunta que podemos hacernos es por qué hemos incluido esa librería. La respuesta es porque con el fichero de cabecera `math.h` sólo hemos incluido definiciones de constantes (por ejemplo `PI`), tipos de datos y prototipos de funciones, pero no el cuerpo compilado de estas funciones, que está contenido precisamente en esa librería (todas las librería estáticas empiezan por `lib` y tienen extensión "`.a`", en medio va el nombre de la librería que es el que se ultiza con `-l`, en este caso el nombre de la librería es `m` y el del fichero librería es: `libm.a`), ya que estas funciones no son de uso general. Ocurre lo contrario con las librerías de manejo de la entrada/salida y funciones comunes, que si que se incluyen por defecto al realizar los ejecutables, estas librerías son `crt0.o` y *`libc.a`*

Otra pregunta que surge es: ¿Puedo yo crear mis propias librerías?. La respuesta es sí. Existe el comando `ar` que nos permite crearlas (añadir módulos), modificarlas o eliminarlas (quitar módulos). Como cualquier comando, la sintaxis del mismo incluye opciones y argumentos:

```
ar -[opciones] [módulos] librería [ficheros]
```

Las opciones más habituales son:

Opción	**Significado**
d	Borrar módulos a través de los ficheros indicados
m	Cambia de orden (mueve) un módulo en la librería
p	Pinta en pantalla un módulo a través de su fichero
q	Añade de forma rápida módulos (sin registro índice) al final
r	Reemplaza módulos a través de su fichero
t	Muestra el contenido de la librería
x	Extrae módulos a través de su fichero
o	Preserva la fecha original del módulo en la extracción
s	Crea o actualiza el registro índice
u	Reemplaza teniendo en cuenta la fecha
Modificador	
a	Lo coloca detrás de un módulo existente
b, i	Añade delante de un módulo existente
c	Crea una librería
v	Modo "verbose"

De esta manera si tenemos una librería que se llama *libre.a* (la librería sería `re`) y tres módulos *mod1.o*, *mod2.o* y *mod3.o* podemos hacer por ejemplo:

Ejemplo	**Acción**
ar c libre.a	Crea la librería
ar r libre.a mod1.o	Añade el módulo y crea la librería si no existe
ar tv libre.a	Muestra el contenido de la librería
ar q libre.a mod2.o mod3.o	Coloca al final de forma rápida el módulo
ar s libre.a	Actualiza el registro índice
ar x libre.a mod3.o	Extrae el tercer módulo

A la hora de usar la librería creada tenemos que tener en cuenta las siguientes reglas:

1. Las librerías se buscarán en los directorios por defecto que son `lib` y `/usr/lib`. Si no ponemos nuestra librería ahí, tendremos que utilizar la opción del compilador `-L` para indicarlo.
2. Lo mismo tendremos que hacer con los ficheros incluidos de cabecera (`include` y `/usr/include`), en este caso la opción es `-I`.
3. Todas las librerías que creemos empezarán con la palabra `lib` a la que seguirá el nombre propio de la librería con una extensión `.a`.

4. Al compilar tendremos que invocar al enlazador con la opción `-l` para que incluya la librería creada (en el ejemplo `-lre`).

Existe un comando relacionado con las librerías, `nm`, para ver el contenido de sus módulos. La sintaxis es:

```
nm -[opciones] [ficheros]
```

donde el fichero puede ser un módulo o una librería, en este último caso se puede usar la opción `-s` para ver el índice.

Existe otro tipo de librerías que son las dinámicas (las anteriores eran estáticas) que se son el equivalente DLL del mundo Windows. Estas librería no se incluyen en el ejecutable, están residiendo en memoria y pueden ser compartidas por varias aplicaciones, con lo cual ahorramos espacio en memoria. Otra ventaja de estas librerías es que su actualización supone una actualización implícita de las aplicaciones que las usan y una desventaja es que si queremos activar un servicio (aplicación) ya no será suficiente con cambiar el ejecutable si no que lo deberemos hacer también con las librerías que usan éstos tanto estáticas como dinámicas.

Las librerías dinámicas tienen la extensión .so (las estáticas .a) y se ubican en los mismos sitios: `/lib`, `/usr/lib` y `/usr/local/lib`. Para su uso deberemos utilizar la opción del compilador `-shared`.

9.3. RPM

Instalación RPM

*U*na de las utilidades más potentes e innovadoras de la empresa Red-Hat es el gestor de paquetes (RPM), que no sólo se usa en sistemas Linux Red-Hat (Suse, Mandriva, ...), sino que se ha extendido a otros sistemas Linux y UNIX. En la siguiente tabla aparecen los sistemas y arquitecturas que cuentan con la utilidad RPM:

Linux: Sparc/Intel/PowerPC/Alpha/m68k/SGI	OS/2
Solaris - Sparc/Intel, solaris2.4	Hewlett-Packard HP-UX 10.20
SCO OpenServer 5.0.2	osf3.2
SunOS 4.1.3	sinix5.42
HP-UX 9.04	AIX 3.2.5
AIX 4.1.4	cygwin-B20
LynxOS 3.0.1	MachTen
IRIX	ncr-sysv4.3
FreeBSD	NetBSD
Mint	AmigaOS (with GeekGadgets)
Beos (with GeekGadgets)	

Básicamente un paquete es un conjunto de ficheros y metadatos (macros de ayuda, atributos de ficheros e información del paquete) usados para instalar y borrar una aplicación. Esta aplicación puede venir dentro del paquete en dos modalidades:

- Ficheros binarios, donde el paquete se usa básicamente para encapsular la aplicación.
- Ficheros fuente, donde a parte de los programas fuente existen macros para producir los ejecutables binarios.

Los paquetes RPM se pueden usar desde la línea de comando con una estructura de tipo:

```
rpm [opción] nombre_de_paquete
```

Las dos fuentes básicas de paquetes RPM serán por un lado la distribución (usualmente en CD-ROM), en el caso de RedHat en el directorio RedHat/RPMS, y la dirección de RPM: http://www.rpm.org donde se encuentra la herramienta (buscador de web) http://rpmfind.net/ a la que daremos el nombre del paquete o aplicación buscada y nos mostrará una lista con las versiones disponibles. Esta versión suele ser parte del nombre del paquete, que está compuesto por: `nombre-versión-publicación.arquitectura.extensión (rpm)`. Los números de versión están divididos en tres partes: el número mayor, el número menor y la revisión. Un aumento en el primer número implica grandes cambios y siempre merecerá la pena actualizar el software si está disponible. El número menor indica la estabilidad del producto, los números pares son estables, los impares indican versiones en desarrollo. El último número indica el nivel de actualización alcanzado por la versión, en los pares avanzará poco a poco y en las impares se pueden dar hasta dos revisiones por semana.

Actualmente (enero 2006) está operativa la versión 4.4.2, (se puede comprobar la nuestra con: rpm -version), que nos permitirá operar en cinco modos (excluyendo el modo de construcción, más orientado a programadores/distribuidores que a administradores y otros menores como las firmas de los paquetes):

- **Instalación** (opción `-i`). La sintaxis de este modo es:

  `rpm -i --opciones paquetes`

  Entra las opciones de instalación se pueden destacar:

  - `-vv`, para dar información completa (verbose).
  - `-h`, muestra “#” para saber que el programa sigue funcionado.
  - `--percent`, para dar un porcentaje en vez de “#”.
  - `--test`, para saber que ocurrirá con la instalación sin hacerla realmente.
  - `--force`, forzará a que RPM instale el paquete a pesar de que haya conflictos de instalación.

  El origen del paquete a instalar puede ser tanto un fichero local como alguna dirección de ftp (normalmente `rpmfind.net`), como por ejemplo: ftp://ftp.rpmfind.net/linux/redhat/redhat-5.2/updates/i386//glint-2.6.3-1.i386.rpm, para conseguir la versión 2.6.3 de `glint` para Linux basados en Intel.

  Cuando se realiza la instalación, el programa RPM ejecutará varios pasos:

  - Comprobará la dependencia con otros paquetes y también si hay conflictos (instalar un software más antiguo que el existente).
  - Preservará ficheros de configuración.
  - Instalará los archivos del paquete en los directorios pertinentes.
  - Configurará el software y
  - Actualizará su base de datos (registro de programas instalados) usualmente en `/var/lib/rpm`.

- **Desinstalación** (opción `-e`). La sintaxis de este modo será:

  `rpm -e --opciones paquetes`

  Las dos opciones de borrado más útiles son:

  - `-vv`, para dar información completa (verbose) y
  - `--test`, para saber que ocurrirá con la desinstalación sin hacerla realmente, normalmente se utilizan juntas.

  Cuando de realiza la desinstalación de un paquete, RPM ejecuta varias acciones:

  - Comprueba si otros paquetes son dependientes del que se quiere borrar, si esto ocurre no se producirá el borrado hasta que se indique.
  - Guarda una copia del fichero de configuración.
  - Realiza el borrado efectivo de los ficheros y
  - Actualiza la base de datos o registro.

- **Actualización** (opción `-U`). La sintaxis de este modo será:

  `rpm -U --opciones paquetes`

  Las opciones serán las mismas que las opciones de instalación. Primero se realizará la instalación de la nueva versión y después se borrarán las anteriores, teniendo en cuenta la configuración existente. Si la configuración no es compatible con la nueva versión, se advertirá con un mensaje como : `saving /etc/paquete.conf as /etc/paquete.conf.rpmsave`, debiendo hacerse posteriormente los ajustes necesarios entre las dos versiones de configuración.

- **Consulta** (opción `-q`). La sintaxis de este modo será:

  `rpm -q opciones paquetes`

  Las opciones son:

  - `name`, para dar el nombre del paquete.
  - `a` para dar la lista de todos los paquetes instalados en el sistema.
  - `f fichero`, para dar el paquete asociado a un fichero.
  - `i`, para dar información sobre el paquete y
  - `l`, da la lista de ficheros asociados al paquete.

- **Comprobación** (opción `-V`). La sintaxis de este modo será:

  `rpm -V paquete`

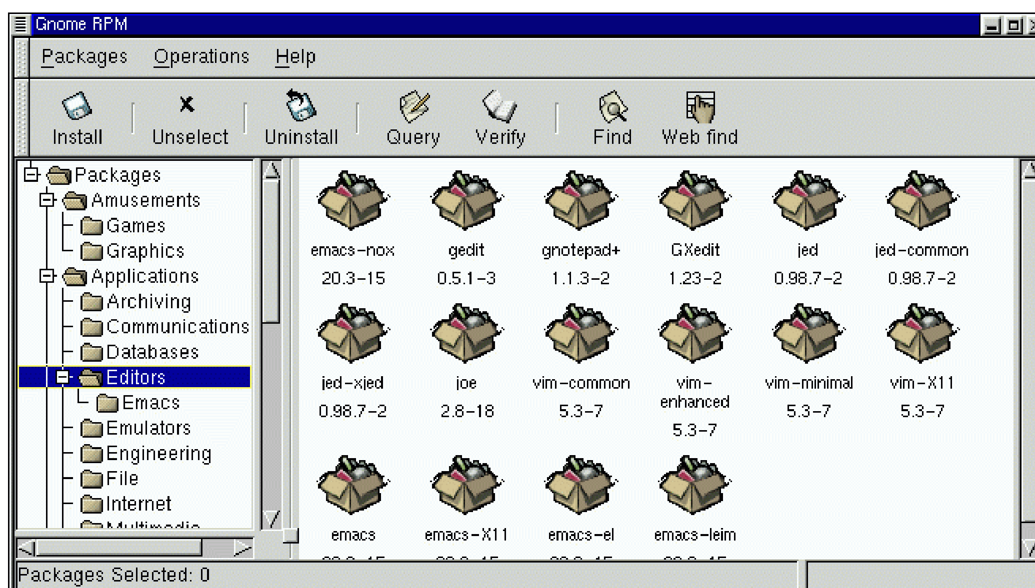
En este modo se comprobarán los ficheros existentes de un paquete con los originales en varios aspectos (presentados abajo), si no hay discrepancias no se mostrará nada, si las hay se presentarán los siguientes códigos:

- `c`, configuración.
- 5, verificación de suma para comprobar si el contenido.
- S, tamaño.
- L, enlaces.
- T, fecha.
- D, atributos.
- U, usuario.
- G, grupo.
- M, modo.

Además de las opciones más usuales vistas para cada modo de utilización de RPM, existen otras de carácter general, como:

- `--help`, que nos dará información sobre el uso de RPM (también se puede hacer man rpm).
- `--version`, que nos dará la versión de RPM que estamos usando.
- `--showrc`, que nos indicará el proceso de inicializacón y
- `--rebuilddb` para reconstruir la base de datos.

Por último, existe una herramienta gráfica llamada `glint` que permite ejecutar los modos de acción de RPM de forma mas amigable. Si disponemos de una versión de Red-Hat con un entorno gráfico Gnome, ya no necesitaremos instalar `glint`, ya que desde el propio menú contamos con la aplicación gráfica (ver figura posterior).



9.4. DEB

Instalación DEB

La otra familia de distribuciones basada en Debian: Debian, Ubuntu, Knopixx, Gentoo, ... utiliza otro tipo de instalación de paquetes similares a los RPM pero distintos en su funcionamiento interno, llamados paquetes DEB. Hay que tener en cuenta que en la mayoría de los sistemas pueden convivir los dos tipos de instalación perfectamente.

Existen dos formas de usar estos paquetes:

- **`dpkg`** y **`dpkg-deb`** que sustituyen al comando rpm para la instalación de paquetes Debian. O mejor:
- La familia **`apt`** (Advanced Packaging Tool). Especialmente el comando **`apt-get`** que se encarga de la instalación y actualización del software automáticamente (se debe editar el fichero `/etc/apt/sources.list` para indicar al sistema las ubicaciones de dónde se puede encontar el software) sin necesidad de "saber" dónde está. El contenido de este archivo,

normalmente sigue este formato:

```
deb http://host/debian distribución sección1 sección2 sección3
deb-src http://host/debian distribución sección1 sección2 sección3
```

La primera línea para código ejecutable y la segunda para código fuente. Por defecto aparece lo siguiente (las líneas de código fuente deshabilitadas):

```
# See sources.list(5) for more information,
especialy
# Remember that you can only use http, ftp or file
URIs
# CDROMs are managed through the apt-cdrom tool.
deb http://http.us.debian.org/debian stable main
contrib non-free
deb http://non-us.debian.org/debian-non-US
stable/non-US main contrib non-free
deb http://security.debian.org stable/updates main
contrib non-free

# Uncomment if you want the apt-get source
function to work
#deb-src http://http.us.debian.org/debian stable
main contrib non-free
#deb-src http://non-us.debian.org/debian-non-US
stable/non-US main contrib non-free
```

Si tuvieramos dudas de que servidor usar lo podríamos escoger con el comando **`netselect`** que nos dará el de menor ping. Incluso podríamos añadir el dispositivo CDROM en vez de la red con `apt-cdrom add`.

Las opciones más frecuentes de apt-get son:

install paquete. Para instalar un paquete nuevo. Si hubiera alguna dependencia de otro paquete también se instalará automáticamente.

remove paquete. Para eliminar un paquete que no sea necesario, incluyendo sus dependencias.

upgrade. Para actualizar los paquetes instalados en el sistema. Se deberá hacer periódicamente.

dist-upgrade. Para actualizar la distribución entera. Normalmente se hace desde CDROM.

clean y autoclean. Cuando instalamos paquetes estos son guardados de forma local en /var/cache/apt/archives/ . Este depósito puede crecer con el tiempo y es necesario su ajuste. Utilizaremos clean para borrar todos los archivos (salvo los bloqueados) y autoclean para hacer una autolimpieza consistente en borrar versiones anteriores de los paquetes, ya sea localmente (existen dos guardadas, se borrará la más antigua) o en red (si existe una nueva versión en red será borrada la local).

Para usar algunas de estas opciones antes debemos saber el nombre del paquete con el que queremos trabajar. Esto se hace con otro comando de la familia que es **`apt-cache`**:

`search nombre.` Con esta opción buscaremos información sobre nombre.

`show.` Suministra información de un paquete en concreto que hayamos podido encontrar con la opción anterior.