# Lezione 4 - L'interfaccia a Linea di Comando

30 novembre 2006

Alessio Checcucci

Riccardo Aldinucci

Q.It Universita' degli Studi di Siena

# 1 L'interfaccia a riga di comando

## 1.1 Le funzionalita' della riga di comando

UNIX fornisce, per l'espletamento dei normali compiti, una serie *tool*, ognuno dedicato ad uno specifico compito.

Un tool e' un semplice programma, di solito progettato per realiazzare un preciso scopo e a cui ci si riferisce in genere con il termine *comando*.

La "Unix tools philosophy" e' emersa durante la creazione del sistema operativo, dopo la "dirompente" invenzione della *pipe*.

La pipe permette all'output di un programma di essere inviato all'input di un altro. La filosofia e' quella di avere piccoli programmi che compiono un task particolare, invece di avere grandi programmi monolitici che compiono un gran numero di task. I task piu' complessi possono essere realizzati connettendo i tool fra di loro, appunto per mezzo delle pipe.

Tutti i tool base del sistema sono progettati per poter lavorare e cooperare in questo modo.

## 1.2 Il boot del sistema

Vediamo cosa accade ad una macchina con il sistema operativo GNU/Linux dal momento in cui viene accesa. Dopo la fase POST (Power On Self Test), il caricatore di boot del BIOS va a leggere il primo settore del disco e se qui viene trovato un elemento avviabile (caratterizzato da un particolare magic number che lo identifica), esso viene caricato in memoria ed eseguito.

In genere il codice contenuto in questo settore corrisponde al boot loader del sistema GNU/Linux. I piu' diffusi sono GRUB (GRand Unified Bootloader) e LILO (LInux LOder). A dispetto dei nomi questi sono strumenti molto potenti in grado di avviare un'enorme quantita' di sistemi operativi (ridirigendo la richiesta verso i boot loader di ogni singolo OS), a patto che si abbia la pazienza di configurarli. Il boot loader e' in genere costituito da due parti che vengono caricate in fasi successive. In genere (ma dipende dalla configurazione) il boot loader presenta una schemata con un menu' da cui scegliere quale OS avviare. Una volta scelto GNU/Linux il boot loader si premura di caricare il codice del kernel passandogli gli opportuni parametri che possono essere definiti o alla linea di comando o nel file di configurazione.

La procedura a partire dal kernel 2.6 prevede l'uso di un *initrd* (Initial Ramdisk). Ovvero il kernel viene caricato ed eseguito, al termine di questa fase viene caricato un certo file in un ramdisk, cioe' una porzione di memoria che viene gestita come un dispositivo a blocchi. Il file contiene l'immagine minimale di un system root filesystem Linux, che viene montato come root filesystem. In esso sara' presente uno script che viene eseguito. A seconda della distribuzione esso potra' compiere una serie piu' o meno grande di azioni; che per tutti si concludera' con l'unmount del filesystem sito sull'initrd e il mount del root filesystem del sistema presente sull'hard disk (in genere).

Fatto questo il kernel va a cercare se esiste il file /sbin/init o /bin/init e lo manda in esecuzione. Esso va ad avviare il processo init, ovvero il processo numero 1 di qualunque sistema Linux. Esso, come abbiamo visto, precedera' all'inizializzazione del sistema e carichera' il runlevel specificato nel suo file di configurazione (/etc/inittab). In ogni runlevel (tranne 0 e 6 che servono per arresto e riavvio del sistema), verranno lanciate e agganciate ai terminali seriali virtuali dei processi della famiglia getty che si occupano di presentare il login prompt sulla console o su un terminale seriale.
1:2345:respawn:/sbin/mingetty tty1

2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

A questo punto una volta che sia stato inserito nome utente e password viene invocato il programma login che si occupa della autenticazione per mezzo dei moduli PAM (dominio amministrativo *auth*). Per fare questo va a referenziare il file /etc/passwd estraendo lo *UID* dal corrispondente campo *nome utente*. A questo punto la password fornita dall'utente viene elaborata dalla funzione crypt(), la cui uscita viene confrontata con il secondo campo del record opportuno del file /etc/shadow. Se le due stringhe corrispondono la password fornita e' valida e l'utente puo' considerarsi autenticato, il dominio amministrativo *account* del modulo PAM andra' a verificare se l'account e' valido e se tutto e' a posto, verra' estratta dal file /etc/passwd la default shell dell'utente e verra' lanciata (eseguendo i propri file di configurazione), portando l'utente nella sua home directory (sempre estratta da /etc/passwd).

## 1.3 Le Unix shell

Le shell UNIX sono programmi che interpretano i comandi dell'utente, che sono o direttamente inseriti dall'utente stesso opppure possono essere letti da un file detto shell script. Gli script di shell sono interpretati e non compilati. La shell legge i comandi degli script una linea alla volta e va a cercarli nel sistema.

A parte passare i comandi al kernel, il principale compito di una shell e' quella di fornire all'utente il proprio ambiente di lavoro, che puo' essere configurato individualmente utilizzando gli opportuni file.

### 1.3.1 I tipi di shell

Cosi' come esistono molti linguaggi e dialetti differenti, ogni sistema UNIX offre una varieta' di tipi di shell:

- **sh** o Bourne Shell: e' la shell originale, ancora in uso nei sistemi UNIX e UNIX-like. E' una shell minimale, un piccolo programma con poche funzionalita'. Sebbene non sia la standard shell, essa e' disponibile su

ogni sistema Linux per compatibilita' con i programmi UNIX. Stephen Bourne scrisse la Bourne shell ai Bell Laboratories, da allora molti autori hanno prodotto specifiche versione della sh, ma l'hanno sempre inserita nei loro UNIX.

- **bash** o Bourne Again Shell: la shell standard GNU, intuitiva e flessibile. La piu' indicata per i principianti e allo stesso tempo ricca di potenti caratteristiche per gli utenti professionali. In Linux bash e' la shell standard per i comuni utenti. La bash e' anche conosciuta come *superset* della Bourne shell, attraverso una serie di add-on e plug-in. Questo significa che la Bourne Again shell e' retro compatibile con la sh: i comandi che funzionano nella Bourne Shell lavoreranno anche in bash. Comunque il contrario non e' sempre vero.

- **csh** o c shell: la sintassi di questa shell somiglia a quella del linguaggio C, che i programmatori spesso prediligono.

- **tcsh** o Turbo C shell: un *superset* della comune csh, che migliora la velocita' e l'interattivita' con gli utenti. Viene configurata con i file /etc/csh.cshrc, /etc/csh.login, /etc/csh.logout, ~/.tcshrc, ~/.cshrc, ~/.history, ~/.cshdirs, ~/.login, and ~/.logout.

- **ksh** o Korn shell: viene apprezzata da alcune persone con grossa esperienza UNIX. E' un *superset* della Bourne Shell. Non e' una shell free.

- **zsh**: il pacchetto ZSH contiene un interprete dei comandi che e' stato concepito per essere per la maggior parte simile alla Korn shell ma con vari miglioramenti.

- **pdksh**: e' un clone di pubblico dominio della Korn Shell, per la maggior parte aderente al modello ksh88.

- Microshell: sono shell concepite per essere di piccole dimensioni e con un set di caratteristiche molto ridotto. Possono essere utilizzate per l'avvio del sistema nel filesystem che viene creato nell'initial ramdisk e consentono poi di lanciare lo script (/linuxrc) che si occupa delle fasi successive, oppure possono essere utili su un rescue system su un dispositivo con poca capacita' di memoria e infine sono utili in situazioni di disaster recovery. Alcuni esempi sono: **ash**, **sash**, **nash**.

## 2 L'introduzione alla shell bash

La shell bash venne creata per l'utilizzo nel progetto GNU, di cui avrebbe dovuto essere la standard shell. La sua nascita risale al 1988, la versione originale fu scritta da Brian Fox assieme a readline. Dal 1993 la responsabilita' e' passata a Chat Ramey che ancora coordina lo sviluppo. In accordo con i principi GNU, la bash shell e' stata liberamente disponibile fin dalla versione 0.99 e e' stata

portata su ogni versione di UNIX ed e' rapidamente diventata la piu' popolare shell derivata dalla Bourne shell.

Al momento esistono tre major release della bash. Le versioni 1 e 2 sono ormai vecchie, mentre la versione 3 e' quella attualmente sviluppata.

Oltre ad essere compatibile con la Bourne shell, bash presenta tutta una serie di caratteristiche provenienti dalla korn e dalla C shell.

Uno degli aspetti che piu' attira gli utenti e' la capacita' di modifica della linea di comando, in questo modo e' possibile risalire la history dei comandi e modificare gli stessi in pochi keystroke.

L'altra caratteristica degna di nota e' il controllo dei job, che permette di far partire, arrestare e mettere in pausa qualsiasi numero di comandi nello stesso momento.

Oltre a questo ci sono un altro gran numero di caratteristiche che riguardano la personalizzazione e la programmazione della shell stessa.

## 2.1 Esecuzione dei comandi

Bash determina il tipo di programma che deve essere eseguito. I normali programmi sono comandi di sistema che esistono in forma compilata. Quando questi programmi vanno in esecuzione, viene creato un nuovo processo, perche' la bash crea una nuova copia di se stessa. Questo processo figlio eredita lo stesso ambiente (*environment*) del padre, solo il Process ID e' diverso. Questa procedura viene detta *forking*.

Dopo il processo di forking, l'address space del processo figlio viene sovrascritto con i dati del nuovo processo. Questo viene fatto per mezzo della system call *exec*.

Il meccanismo di *fork-and-exec* trasforma un vecchio comando in uno nuovo, mentre l'ambiente nel quale viene eseguito il nuovo programma resta lo stesso, inclusa la configurazione dei device di input e output, le variabili d'ambiente e la priorita'. Questo meccanismo viene utilizzato per creare tutti i processi UNIX, e nello stesso modo anche in Linux. Anche il primo processo del sistema, init, che ha un PID uguale a 1 viene avviato per fork durante la procedura di boot.

### 2.1.1 I comandi Built-In

Tutta una serie di comandi che ci troveremo ad utilizzare non fanno riferimento a file eseguibili, ma sono direttamente presenti all'interno della bash shell stessa. Essi sono i seguenti:

bash, :, ., alias, bg, bind, break, builtin, case, cd, command, continue, declare, dirs, disown, echo, enable, eval, exec, exit, export, fc, fg, for, getopts, hash, help, history, if, jobs, kill, let, local, logout, popd, pushd, pwd, read, readonly, return, set, shift, shopt, source, suspend, test, times, trap, type, typeset, ulimit, umask, unalias, unset, until, wait, while

## 2.2 Uso interattivo della shell

Quando la shell viene usata interattivamente, viene avviata una login session che dura finche' la shell non termina (usando il comando exit o logout). Durante la login session, l'utente digita delle linee di comando (*command line*), che non sono altro che linee di testo che terminano con un RETURN.

Per default il prompt della shell propone una linea di informazione seguita dal carattere $, naturalmente questo aspetto puo' essere completamente configurato.

### 2.2.1 Comandi, argomenti e opzioni

I comandi della shell consistono di una o piu' parole, separate da spazi o TAB. La prima parola della linea e' il *comando*. Il resto sono gli *argomenti* (detti piu' comunemente parametri) del comando, ovvero nomi di oggetti su cui il comando andra' ad operare.

Un'*opzione* e' un argomento di tipo particolare che fornisce al comando informazioni specifiche su quello che si suppone faccia. Le opzioni in genere consistono di un carattere dash "-" seguito da una lettera, questa e' comunque piu' una convenzione che una regola. A volte le opzioni hanno un proprio argomento.

## 2.3 File

Sebbene gli argomenti della linea di comando non siano sempre file, essi sono l'oggetto piu' importante in UNIX. Un file puo' contenere ogni genere di informazione e quindi avremo file di vario tipo, i piu' importanti sono: *regular files* (text files), *executable files* (programmi), *directories*.

### 2.3.1 La working directory

Specificare un path completo ogni volta che si voglia referenziare un file e' scomodo. A questo scopo esiste il concetto di working directory, che non e' altro che la directory in cui siamo in un certo momento. Quindi se viene specificato un pathname che non inizia con /, allora esso e' *relativo* alla working directory.

### 2.3.2 Tilde notation

Le home directory ricorrono molto spesso nei pathname, bash quindi ha un metodo per abbreviare il loro nome. E' sufficiente precedere il nome dell'utente con il carattere ~. Oltre a questo, un carattere ~ da solo rappresenta la propria home directory.

### 2.3.3 Cambiare working directory

Per cambiare working directory si utilizza il comando *cd*. Se non ci ricordiamo la working directory, esiste il comando *pwd*, e la shell stampera' il percorso.

Il comando cd lanciato senza argomenti ci porta nella home directory, mentre la forma cd - ci riporta nella directory precedente in cui eravamo.

## 2.4 Filename, wildcard e espansione dei pathname[1]

A volte un utente necessita di lanciare un comando su piu' di un file alla volta. Siccome i file sono cosi' importanti nel sistema, la shell implementa al suo interno una modalita' per specificare il pattern di un set di file senza doverne conoscere il nome. Per fare questo si usano speciali caratteri detti *wildcards*.

| **Wildcard** | **Matches** |
|---|---|
| ? | Ogni singolo carattere |
| * | Ogni stringa di caratteri |
| [set] | Caratteri nel set |
| [!set] | Caratteri *non* nel set |

Da notare che nei filesystem UNIX il carattere "." non ha un particolare significato, come nei sistemi Microsoft e VAX, ma viene trattato come un comune carattere.

Il costrutto *set*, puo' essere espresso in varie maniere:

| Expression | Matches |
|---|---|
| [abc] | a, b o c |
| [.,;] | punto, virgola o punto e virgola |
| [-_] | dash o underscore |
| [a-c] | a, b o c |
| [a-z] | Tutti i caratteri minuscoli |
| [!0-9] | Tutto tranne i numeri |
| [0-9!] | Tutti i numeri e ! |
| [a-zA-Z] | Tutte le lettere maiuscole e minuscole |
| [a-zA-Z0-9-_] | Lettere, numeri, underscore e dash |

Utilizzare i range nei set puo' essere comodo, ma non si deve fare troppo affidamento su quali caratteri verrano inclusi, sono sicuri da utilizzare solo i range per lettere maiuscole, lettere minuscole, numeri e ogni loro sottoinsieme, questo perche' i range non sono sono portatbili fra architetture (e quindi charset) diversi.

Il processo di fare il match di espressioni contenenti wildcard rispetto ai filename e' detto **globbing** o *wildcard expansion*. La cosa importante e' che i programmi saranno a conoscenza solo dei risultati dell'espansione, senza sapere da dove provengono.

Gli esempi con gli wildcard non sono altro che parte di un meccanismo detto *wildcard expansion*; cosi' com'e possibile utilizzare gli wildcard nella working directory, e' possibile adoperarli in ogni parte del pathname.

[1] I due meccanismi di espansione presenti in una quantita' di software Unix sono il globbing, che qui viene analizzato e le regular expression (espressioni regolari), che riportiamo in appendice.

### 2.4.1 Brace Expansion

Un concetto intimamente legato con il pathname expansion e' il *brace expansion*. Mentre la wildcard expansion fa riferimento file e directory che esistono, la brace expansion fara' l'espansione ad una stringa arbitraria della forma data: un *preamble* opzionale, seguito da *stringhe* separate da virgole all'interno di *parentesi graffe* e da un *postscript* opzionale. Per esempio b{ar,ed}s espandera in bars e beds.

Vale la pena di notare che questi non sono filename, le stringhe prodotte ne sono indipendenti, ogni istanza delle stringhe dentro le parentesi graffe sara' semplicemente combinata col preamble e il postscript.

## 2.5 Input e Output

Lo schema di I/O e' basato su due semplici idee:

- Lo UNIX file I/O prende la forma di sequenze di caratteri (byte) arbitrariamente lunghe.
- Tutto cio' che sul sistema produce o accetta dati e' trattato come un file.

### 2.5.1 Standard I/O

Per convenzione ogni programma UNIX ha un singolo modo per accettare l'input detto *standard input*, un solo modo per produrre output detto *standard output* e un singolo modo per produrre messaggi d'errore detto *standard error output* (meglio conosciuto come standard error).

Standard I/O fu il primo schema del genere e venne progettato in modo specifico per gli utenti interattivi ai terminali. Visto che la shell fornisce la user interface, non dovrebbe sorprendere che lo standard I/O sia progettato per adattarsi perfettamente ad essa.

Tutte le shell trattano lo standard I/O nello stesso modo, Ogni programma che venga invocato ha tre canali standard di I/O configurati per il teminale. In questo modo lo standard input e' la tastiera e gli standard output e error sono lo schermo o una finestra.

Quando si renda necessario e' possibile redirigere l'input e l'output.

E' possibile anche legare i programmi assieme per mezzo di una *pipeline*, nella quale lo standard output di un programma alimenta direttamente lo standard input di un altro.

Questo approccio, che e' basilare in UNIX, permette di utilizzare le utility presenti nel sistema operativo come mattoni per affrontare problemi complessi. Molti programmi di utilita' UNIX sono studiati per operare in questo modo: ognuno di essi fornisce una particolare operazione di filtraggio sul testo che hanno in ingresso.[2]

[2] Per un'utilita che non accettasse lo standard input se non viene specificato il file di input, basta sostituirlo col carattere -.

### 2.5.2 Redirezione dell'I/O

Con la sintassi:

*command* < *filename*

si realizza la redirezione dello standard input in modo da farlo provenire da un file. Le cose vengono sistemate in modo che *command* prenda lo standard input da un file invece che dal terminale.

In maniera simile:

*command* > *filename*

causa il *command* standard output ad essere ridiretto ad un file di nome *filename*.

I redirettori di input e output possono essere combinati, per esempio il comando:

*cat* <*file1* >*file2*

finira' per copiare file 1 in file 2.

### 2.5.3 Pipelines

E' possibile ridirigere l'output di un comando nello standard input di un altro comando invece che su un file. Il costrutto che fa questo e' detto **pipe**, e denotato dal carattere /. Una linea di comando che include due o piu' comandi connessi attraverso pipe e' detta *pipeline*.

Le pipe sono usate molto spesso con comandi come more o less, per poter accedere all'output di un comando una schermata alla volta.

Le pipeline possono diventare molto comlplesse e possono essere combinate con ogni tipo di redirettore di I/O.

E' chiaro come gli I/O redirector e le pipeline supportino la filosofia di base di UNIX. La notazione e' incredibilmente concisa, chiara e potente. Altro aspetto importante e' che il pipe elimina la necessita' di molti file temporanei per salvare l'output di un comando da dare poi in pasto a successivi comandi.

### 2.5.4 I/O e command line processing avanzato

Quanto visto finora copre il 95% di quello che un utente puo' fare come redirezione nel sistema. Ma bash supporta un grande insieme di redirectors:[3]

[3]Il redirector << *label* forza essenzialmente l'input di un comando ad essere lo standard input della shell, che viene letto finche' c'e' una linea che contiene solo *label*. L'input in mezzo viene chiamato *here-document*.

| Redirector | Function |
|---|---|
| cmd1 \| cmd 2 | Pipe, ridirige l'output di *cmd1* sull'input di *cmd2* |
| > file | Dirige lo standard output su *file* |
| < file | Prende lo standard input da *file* |
| >> file | Dirige lo standard output su *file* con modalita' append se il file esiste gia' |
| >\| file | Forza lo standard output su *file* anche se l'opzione noclobber e' attiva |
| n>\| file | Forza l'output su *file* dal descriptor *n* anche se l'opzione noclobber e' attiva |
| <> file | Utilizza *file* sia come standard input che come standard output |
| n <>file | Utilizza *file* sia come input che come output per il file descriptor *n* |
| << label | Here-document: Vedi nota |
| n > file | Ridirige il file descriptor n su *file* |
| n < file | Prende il file descriptor *n* da *file* |
| n >> file | Dirige il file descriptor *n* su *file* con modalita' append se il file esiste gia' |
| n >& | Duplica lo standard output sul file descriptor *n* |
| n <& | Duplica lo standard input dal file descriptor *n* |
| n >& m | Il file descriptor *n* sara' una copia dell'output file descriptor |
| n <& m | Il file descriptor *n* sara' una copia dell'input file descriptor |
| &> file | Dirige standard output e standard error su *file* |
| <&- | Chiude lo standard input |
| >&- | Chiude lo standard output |
| n >&- | Chiude l'output dal file descriptor *n* |
| n <&- | Chiude l'input dal file descriptor *n* |

### 2.5.5 File descriptor

I ridirettori visti dipendono dalla nozione di *file descriptor*. Questo e' un concetto di basso livello dello UNIX I/O con cui in genere solo i programmatori devono confrontarsi.

I file descriptor sono degli interi che iniziano da 0 che referenziano un particolare stream di dati associato con un processo. Quando un processo si avvia, di solito ha tre file descriptor aperti. Questi corrispondono ai tre *standard*: standard input (0), standard output (1) e standard error (2). Se un processo apre altri file per operazioni di input o output, ad essi vengono assegnati i successivi file descriptor disponibili.

### 2.5.6 L'I/O delle stringhe

Quello che ci interessa qui e' vedere l'I/O delle stringhe per mezzo dei comandi echo e read, che conferiscono alla shell le stesse funzionalita' di molti linguaggi di programmazione.

**Echo** Echo stampa i suoi argomenti sullo standard output. importanti opzioni sono:

- -e: abilita l'interpretazione delle escape sequences
- -E: disabilita l'interpretazione delle escape sequences se questa fosse la modalita' di default
- -n: Omette il newline a fine linea

Le escape sequences sono introdotte dal carattere "\" e servono per definire caratteri speciali e difficilimente riproducibili. Sono in tutto simili alle sequenze di escape del C.

**Read** L'altra meta' delle utilita' di I/O per le stringhe e' read, che permette di leggere dei valori e metterli in variabili di shell.

*read var1 var2 ...*

Questo comando prende una linea dallo standard input e la suddivide in valori in base all'internal field separator IFS.

## 2.6 Gestione dei processi

Il sistema operativo Unix ha costrutito la propria reputazione su una serie di concetti, semplici ma potenti. Unix ha guadagnato notorieta' come il primo sistema operativo per piccoli computer che ha dato ad ogni utente il controllo su piu' di un processo. Questa capacita' viene detta *user-controlled multitasking*.

### 2.6.1 Process ID e Job numbers

UNIX assegna ad ogni processo un numero, detto process ID, quando esso viene creato. Al contrario il job number viene assegnato dalla shell. La differenza e' che i job number fanno riferimento ai processi che sono eseguiti sotto la shell, mentre i process ID si riferiscono a tutti i processi che girano sul sistema per tutti gli utenti. Il termine *job* fondamentalmente si riferisce alla linea di comando che e' stata invocata dalla shell.

Quando un processo viene fatto partire in background (cioe' senza essere connesso ad un terminale) la shell mostrera' il numero del processo e quello del job che lo referenzia:

*[alessio@garp man]$ gv &*
*[1] 10389*
*[alessio@garp man]$*
*[1]+ Done gv*

Quando il job termina, un messaggio verra' presentato, in cui viene mostrato se la fine e' stata corretta, altrimenti viene restituito l'exit status o il motivo della fine del job:

*[alessio@garp man]$ gv &*
*[1] 10408*
*[alessio@garp man]$ kill -9 %1*
*[1]+ Killed gv*

**Job control** Probabilmente nel lavoro quotidiano un utente non si trovera' spesso a dover trattare i Process ID, mentre sara' importante lavorare con i Job Numbers, perche' permettono il controllo dei job per mezzo degli appositi comandi forniti dalla shell.

Come potrebbe essere noto il modo piu' ovvio di controllare un job e' quello di creane uno in background, posponendo alla linea di comando il carattere **&**. Mentre il job e' in background e' possibile attendere che esso termini, portarlo in foreground oppure inviargli un segnale.

**Foreground e background** Il built-in *fg* porta un background job in foreground. Normalmente questo significa che prendera' il controllo del terminale o della finestra e quindi potra' accettare l'input. In altre parole esso si comportera' come se la linea di comando fosse stata lanciata senza &.

Se esiste solo un un job in esecuzione, fg puo' essere lanciato senza argomenti. Altrimenti sara' necessario specificare il job command name, preceduto dal segno *%* oppure utilizzare il job number preceduto sempre dal simbolo %, oppure il process ID preceduto dal simbolo %.

Il comando *jobs* elenca i job in esecuzione. L'opzione -l permette di associare anche i process ID. I termini %+ e %% fanno riferimento al job lanciato piu' di recente, mentre %- si riferisce a quello precedente. Se esistono piu' job con lo stesso comando essi possono essere referenziati con il numero di job, oppure se avessero argomenti diversi con la sintassi %?string.

**Sospendere un job** Cosi' com'e' possibile portare un job in foreground, allo stesso modo lo si puo' mettere in background. Per fare questo occorre sospendere il job, cosi' che la shell riacquisti il controllo del terminale.

Per sospendere un job e' sufficiente premere la combinazione *CTRL-Z* mentre esso e' in esecuzione, il job rispondera con:

*[alessio@garp man]$ gv ... (CTRL-z)*

*[1]+ Stopped gv*

Se si vuole rimettere il job in foreground e' sufficiente il comando fg. Altrimenti, e molto piu' usualmente, esso puo essere messo in background con il comando *bg*. Un job puo' essere stoppato anche con CTRL-Y, ma in questo caso viene bloccato solo quando tenta di leggere dell'input dal terminale.

**Segnali** Abbiamo visto che CTRL-Z permette di interrompere un job, mentre CTRL-C lo fa terminare definitivamente. Questi sono ambedue esempi di *segnali*.

Un segnale e' un messaggio che un processo manda ad un altro processo quando qualche evento anormale ha luogo o quando vuole che l'altro processo faccia qualcosa. La maggior parte delle volte il segnale viene inviato ad un sottoprocesso che il processo ha avviato. I segnali sono una modalita' di interprocess communication che si aggiunge a quella della pipeline gia' vista.

A seconda della versione di UNIX esistono molti segnali, di cui alcuni programmabili. Essi sono caratterizzati da un numero e da un nome, il comando

*kill -l* elenchera' tutti i segnali definiti.

**Tasti per il controllo dei segnali** Esistono tutta una serie di combinazioni di tasti *CTRL-<TASTO>* che permettono di inviare segnali al job corrente. I piu' importanti sono:

- CTRL-C - INT (interrupt)
- CTRL-Z - TSTP (terminal stop)
- CTRL-\ - QUIT (piu' forte di INT)

Esiste anche un "panic" signal, detto KILL, che e' possibile inviare ai processi quando anche CTRL-\ non funziona, ma non e' associato a nessun control key.

E' possibile personalizzare le combinazioni di tasti per inviare i segnali con le opzioni del comando *stty*.

**Kill** E'possibile utilizzare il built-in *kill* per mandare un segnale a qualsiasi processo che abbiamo creato, non solo il job in esecuzione. kill prende come argomento il process ID, il job number o il nome del processo.

Per default il segnale che invia e' TERM, che in genere ha lo stesso effetto di INT, ma e' possibile specificare un altro segnale indicandolo preceduto da un dash (-).

Il nome kill deriva dal fatto che il comportamento di default dei processi quando ricevono un segnale e' quello di morire. Pero' i processi stessi possono catturare i segnali (*trap*) e decidere cosa fare. Il segnale KILL non puo' essere trattato tramite trap, per cui il sistema operativo deve terminare il processo immediatamente e incondizionatamente.

I segnali TERM e QUIT sono stati concepiti per essere utilizzati prima di KILL, per dare una possibilita' al processo di fare un clean-up prima di morire. Quindi, utilizzare KILL come ultima risorsa!

## 2.7 Caratteri speciali e quoting

I caratteri <, >, |, &, *, ? abbiamo visto che hanno un significato speciale, ma non sono i soli. La seguente tabella cerca di essere abbastanza esaustiva, esistono peraltro dei caratteri che assumono particolari significati a seconda del contesto:

| Carattere | Significato |
|---|---|
| ~ | Home directory |
| # | Commento |
| $ | Espressione di Variabile |
| & | Background job |
| * | Wildcard per stringhe |
| ( | Inizio di una subshell |
| ) | Fine della subshell |
| \ | Escape del seguente carattere |
| \| | Pipe |
| [ | Inizio di character set wilcard |
| ] | Fine di character set wilcard |
| { | Inizio di un blocco di comandi |
| } | Fine di un blocco di comandi |
| ; | Separatore dei comandi della shell |
| ’ | Strong quote |
| “ | Weak quote |
| ‘ | Sostituzione di comandi (arcaico) |
| > | Redirezione dell’input |
| < | Redirezione dell’output |
| / | Separatore delle directory nel pathname |
| ? | Wildcard per il singolo carattere |
| ! | Not logico nella pipeline |

### 2.7.1 Quoting

Qualora si vogliano usare i caratteri speciali letteralmente, senza il loro significato particolare, occore ricorrere al *quoting*. Se un espressione viene inclusa in *single quotes (’)* tutti i caratteri all’interno degli apici vengono svuotati da ogni significato particolare.

Al contrario le stringhe in *double quotes (“)*sono soggette ad una parte dei passi che la shell compie quando processa la linea di comando, ma non tutti.

**Backslash-escaping** Un altro sistema per cambiare il significato di un carattere e’ quello di precederlo con un backslash (\). Precedere il carattere con un backslash permette anche di inserire apici e virgolette letterali all’interno di string con quotes.

Un’altra applicazione del backslash e’ quella di poter continuare a capo una riga di testo senza spezzarla con un newline, basta terminarla con il carattere \.

## 2.8 L’editing della comand line e la history

La modifica della linea di comando e’ sempre stata una delle caratteristiche piu’ desiderate dagli utenti, che spesso si trovavano a commettere degli errori di battitura e a perdere intere linee con la Bourne shell e la C shell. Bash

permette l'editing della linea di comando in piu' modalita', simili al modo in cui operano gli editor *vi* e *emacs*. Oltre a questo esiste un meccanismo detto *fc* (fix command) che permette di utilizzare il proprio editor preferito per modificare le linee di comando. Oltre a questo bash implementa anche il meccanismo di *history* che deriva dalla C shell.

Bash quando si avvia parte interattivamente in *emacs-mode* come default. Se l'editing non fosse abilitato, e' possibilile renderlo operativo con il comando set:

*set -o emacs* oppure *set -o vi*

Oppure e' possibile impostare una variabile *readline* nel file *.inputrc*.

Le due modalita' vi ed emacs emulano un piccolo set delle capacita' di questi editor. Mentre per un completo controllo con un editor e' necessario utilizzare fc.

### 2.8.1 L'History file

Tutte le utilita' di bash relative alla history dipendono da una lista che registra i comandi man mano che sono inseriti. Appena un utente fa login oppure apre una shell interattiva, bash legge la lista iniziale della history dal file *.bash_history* nella home directory.

Da questo momento ogni sessione interattiva mantiene la propria lista di comandi. Quando si esce dalla shell, la lista viene salvata in *.bash_history*. Questo file puo' essere nominato come si vuole, impostando la variabile di ambiente *HISTFILE*.

### 2.8.2 Emacs editing mode

Prenderemo in considerazione solo la modalita' emacs di modifica della linea di comando. Peraltro' essa e' utilizzata dalla stragrande maggioranza degli utilizzatori bash. Gli utenti di emacs troveranno questa modalita' molto familiare, sebbene con qualche limitazione rispetto all'editor originale.

**Basic commands** La modalita' Emacs usa i control key per la maggior parte della operazioni di editing. I comandi control-key base sono i seguenti:

| Comando | Descrizione |
|---|---|
| CTRL-B | Spostamento indietro di un carattere |
| CTRL-F | Spostamento avanti di un carattere |
| DEL | Cancella un carattere indietro (in genere associato a BACKSPACE) |
| CTRL-D | Cancella un carattere avanti |

Naturalmente per lo spostamento sulla linea e' possibile utilizzare anche i tasti freccia destra e sinistra, ma nel caso su qualche sistema non dovessero funzionare, abbiamo a disposizione anche i control key.

In modalita Emacs il *point (o dot)* e' un carattere immaginario a sinistra del carattere su cui e' il cursore, occorre sempre pensare a movimenti avanti e indietro a "destra del punto" e a "sinistra del punto" rispettivamente.

**Word commands** Questi comandi permettono di operare con un numero di keystrokes minore, perche' operano su parole invece che su singoli caratteri. Una parola e' una sequenza di uno o piu' caratteri alfanumerici. I comandi per le parole sono in genere combinazioni di tasti con ESC. In genere la lettera che definisce il comando e' la stessa sia per caratteri che per parole, quello che cambia e' il metacarattere.

| Comando | Descrizione |
|---|---|
| ESC-B | Spostamento indietro di una parola |
| ESC-F | Spostamento avanti di una parola |
| ESC-DEL | Cancella una parola indietro (in genere associato a BACKSPACE) |
| ESC-CTRL-H | Cancella una parola in avanti |
| ESC-D | Cancella ("Kill") una parola in avanti |
| CTRL-Y | Recupera ("Yank") l'ultima killed item |

Vale la pena di notare come "Kill" abbia lo stesso significato di "delete"; esso e' il termine standard usato dalla libreria *readline*.

**Line commands** Esistono modalita' ancora piu' efficienti di muoversi sulla linea di comando:

| Comando | Descrizione |
|---|---|
| CTRL-A | Spostamento all'inizio della linea |
| CTRL-E | Spostamento alla fine della linea |
| CTRL-K | Cancella in avanti fino alla fine della linea |

**Interagire con l'History file** L'emacs-mode ha molti comandi per muoversi all'interno dell'history file:

| Comando | Descrizione |
|---|---|
| CTRL-P | Spostamento alla linea precedente |
| CTRL-N | Spostamento alla linea successiva |
| CTRL-R | Ricerca all'indietro |
| ESC-< | Spostamento alla prima linea dell'history file |
| ESC-> | Spostamento all'ultima linea dell'history file |

Lo spostamento di linea in linea all'interno della history puo essere fatto anche utilizzando i tasti freccia alto e basso.

Il comando CTRL-R risulta essere molto potente. E' in pratica un motore di ricerca nel file della history. Appena lanciato ci propone un prompt del tipo: *(reverse-i-search)'':*

via via che viene digitata una stringa, viene avviato un meccanismo di ricerca nell'history file per quella sottostringa. Naturalmente piu' sottostringhe possono corripondere e quello che viene restituito e' il primo match. Per proseguire la ricerca deve essere premuto ancora CTRL-R. Una volta individuato il comando che ci interessa e' possibile portarlo sulla linea di comando con i tasti freccia, oppure premere invio ed eseguirlo direttamente.

**Completamento testuale** L'utilita' di *textual completion* e' di gran lunga una delle piu' utilizzate, anche se poco conosciuta a fondo.

Il fondamento di questa utilita' e' semplice: un utente deve digitare solo la porzione di filename, username, funzione, ecc. necessaria ad identificare l'entita' senza ambiguita'.

Ci sono tre comandi relativi al completamento testuale. Il piu' importante e' TAB. Quando viene digitata una parola seguita da TAB, bash cerchera' di completarne il nome. Possono accadere quattro cose:

- Se non c'e' nulla il cui nome inizi con quella parola, la shell emettera' un beep e nient'altro succedera'.

- Se nel search path c'e' un nome di comando, di funzione o un filename che la stringa identifica univocamente, la shell digitera' la parte mancante seguita da uno spazio. Il completamento dei comandi viene fatto solo all'inizio riga.
- Se c'e' una directory che la stringa identifica univocamente, la shell completera' il filename seguito dallo slash.
- Se ci sono piu' modi per effettuare il completamento, la shell completera' fino al piu' lungo pattern comune. I comandi nel search path hanno la precedenza rispetto ai filename. La pressione di un doppio TAB comporta che bash mostri tutti i possibili completamenti (la stessa cosa viene fatta da ESC-?).

Vale la pena di notare che se ci sono funzioni o comandi che soddisfano la stringa inserita, la shell li espande per primi e ignora ogni file nella directory corrente. E' peraltro possibile forzare il completamento di un particolare tipo.

E' possibile anche completare altre entita' dell'ambiente. Per esempio se il testo che deve essere completato e' preceduto da $, allora la shell cerchera' di completare con un nome di variabile d'ambiente. Se il testo e' preceduto da una ~, viene tentato il completamento ad uno username, se e' preceduto da @, allora il completamento e' ad un hostname.

Esiste la possibilita' di forzare il tipo di completamento con i seguenti comandi:

| Comando | Descrizione |
|---|---|
| TAB | Cerca di effetture un completamente generale del testo |
| ESC-? | Lista di tutti i possibili completamenti |
| ESC-/ | Cerca di completare un filename |
| CTRL-X / | Lista di tutti i completamenti di filename |
| ESC-~ | Cerca di completare uno username |
| CTRL-X ~ | Lista di tutti i completamenti di username |
| ESC-$ | Cerca di completare una variabile |
| CTRL-X $ | Lista di tutti i completamenti di variabile |
| ESC-@ | Cerca di completare un hostname |
| CTRL-X @ | Lista di tutti i completamenti di hostname |
| ESC-! | Cerca di completare un comando |
| CTRL-X ! | Lista di tutti i completamenti di comando |
| ESC-TAB | Cerca il completamento dal precedente comando della history list |

**Altri comandi** Altri comandi interesanti possono essere:

| Comando | Descrizione |
|---|---|
| CTRL-L | Pulisce lo schermo, mettendo la linea corrente ad inizio schermo |
| CTRL-U | Cancella (kill) la linea dall'inizio al point |
| ESC-C | Trasforma la parola dopo il punto con la prima lettera maiuscola |
| ESC-U | Cambia la parola dopo il punto a tutte maiuscole |
| ESC-L | Cambia la parola dopo il punto a tutte minuscole |
| ESC-. | Inserisce l'ultima parola del precedente comando dopo il punto |

### 2.8.3 Il comando fc

*fc* e' un built-in della shell, esso permette di esaminare i comandi piu' recenti, modificarli con l'editor preferito e lanciare vecchi comandi modificati senza doverli battere di nuovo. Esso utilizzera' come editor (a meno che non sia stato specificato in linea di comando): $FCEDIT, poi $EDITOR e infine vi.

L'opzione -l permette di avere la lista di comandi precedenti. Gli argomenti possono essere passati come stringhe o numericamente, essi vengono trattati:

- Se ne sono specificati due, servono come primo e ultimo comando che deve essere mostrato.
- Se ne e' specificato uno, esso si riferisce allo specifico comando.
- Con un solo argomento stringa, viene cercato il piu' recente comando che inizia con la stringa.
- Senza nessun argomento, vengono mostrati gli utimi 16 comandi della history.

Bash ha anche il comando *history* per mostrare tutto lo storico.

Vale la pena di notare come dopo l'editing, fc lanci i comandi che ha editato. Questo puo' essere molto pericoloso, se si fossero editati comandi multipli.

### 2.8.4 History expansion

History expansion e' un meccanismo primitivo per richiamare e modificare comandi. La modalita' per richiamare i comandi e' tramite l'uso di un *event designator*.

| Comando | Descrizione |
|---|---|
| ! | Inizia una history substitution |
| !! | Si riferisce all'ultimo comando |
| !*n* | Si riferisce alla *n-esima* linea di comando |
| !-*n* | Si riferisce al linea corrente meno *n* |
| !*string* | Si riferisce all'ultimo comando che inizia con *string* |
| !?*string*? | Si riferisce all'ultimo comando che contiene con *string* |
| ^*string1*^ *string2* | Ripete l'ultimo comando e sostituisce *string1* con *string2* |

E' possibile referenziare certe parole nel comando precedente per mezzo di un *word designator* (un carattere opreceduto da !):

| **Designator** | **Descrizione** |
|---|---|
| 0 | La prima (*0-esima)* parola in una linea |
| *n* | L'*n-esima* parola in una riga |
| ^ | Il primo argomento (la seconda parola) |
| $ | L'ultimo argomento di una riga |
| % | La parola che fa il match con l'ultima *?string* cercata |
| *x-y* | Un range da *x* a *y* |
| * | Tutte le parole tranne la prima |
| *x** | Sinonimo di *x-$* |
| x- | Tutte le parole da *x* alla penultima |

# 3 L'editor vi

*Vi* e la sua piu' recente evoluzione *Vim* (vi improved) e' un editor che puo' lavorare con ogni genere di file di testo. Sebbene ad un primo impatto esso non sembri particolarmente amichevole, si tratta di un programma potente e indispensabile per le attivita' dell'amministratore, soprattutto perche' in genere e' l'unico editor disponibile nelle situazioni critiche.

*Vi* riconosce tre modalita' di esecuzione: *command mode, edit mode* e *ex mode.*

Una volta avviato *vi* si trova in command mode, in questa modalita' l'utente puo' eseguire una serie di comandi per modificare il testo. Per esempio il comando *i* (insert) (oppure *a, o*) permette di iniziare ad inserire del testo, questo fara' passare l'editor in **edit mode**.

Il Vi tradizionale permetteva lo spostamento tramite tasti freccia (o piu' comunemente con le lettere h, j, k, l) solo in command mode. Vim in questo senso e' molto piu' flessibile, ma dipende dalla configurazione (file *.vimrc*).

Una volta che ci troviamo in edit mode e' sufficiente premere il tasto *esc* per tornare in **command mode**.

I comandi per gestire i file dall'interno di Vi vanno invece inseriti in **ex mode**. L'ex mode viene attivato inserendo il carattere : durante il command mode. Il cursore viene spostato sull'ultima linea dello schermo, permettendo cosi' di digitare i comandi. Il tipico comando e' quello per salvare ed uscire *:wq*.

I comandi vi vanno immediatamente in esecuzione appena li si digita, fuorche' i comandi preceduti da : che prevedono l'introduzione di un RETURN.

Una breve, e assolutamente non esaustiva reference, puo' essere trovata nei seguenti paragrafi:

**Quitting** exit, saving changes **:x**
quit (unless changes) **:q**
quit (force, even if unsaved) **:q!**

**Inserting text** insert before cursor, before line **i , I**
append after cursor, after line **a , A**
open new line after, line before **o , O**
replace one char, many chars **r , R**

**Motion** left, down, up, right **h , j , k , l**
next word, blank delimited word **w , W**
beginning of word, of blank delimited word **b , B**
end of word, of blank delimited word **e , E**
sentence back, forward **( , )**
paragraph back, forward **f , g**
beginning, end of line **0 , $**
beginning, end of file **1G , G**
line n **n G or :n**
forward, back to char c **fc , Fc**
top, middle, bottom of screen **H , M , L**

**Deleting text** Quasi tutti i comandi di cancellazione vengono eseguiti battendo *d* seguito da un *motion*. Altri comandi di cancellazione sono:
character to right, left **x , X**
to end of line **D**
line **dd**
line **:d**

**Yanking text** Quasi tutti i comandi di copiatura vengono eseguiti battendo *y* seguito da un *motion*. Altri copiatura di incollatura sono:
line **yy**
line **:y**

**Changing text** Il comando di change e' una cancellazione che lascia l'editor in insert mode. Viene seguito battendo un carattere *c* seguito da una *motion*. Altri comandi di change sono:
to end of line **C**
line **cc**

**Putting text** put after position or after line **p**
put before position or before line **P**

**Buffers** I named buffers possono essere specificati prima delle cancellazioni, change, yank o put. Il prefisso generico ha la forma *"c* dove c puo' essere qualsiasi lettera minuscola. Essi possono poi essere reinseriti nel testo con un appropriato comando di put.

**Markers** I named markers possono essere impostati su ogni linea di un file. Ogni lettera minuscola puo' essere un marker. I marker possono anche essere utilizzati per limitare i range.
set marker c on this line **mc**
goto marker c **'c**
goto marker c first non-blank **'c**

**Search for Strings** search forward **/string**
search backward **?string**
repeat search in same, reverse direction **n , N**

**Replace** Le funzioni di search e replace sono realizzate con il comando *:s*. Esso viene in genere utilizzato sui range con il comando *:g*.
replace pattern with string **:s/pattern /string /flags**
flags: all on each line, confirm each **g , c**
repeat last :s command **&**

**Regular Expressions** any single character except newline **.** (dot)
zero or more repeats *****
any character in set **[...]**
any character not in set **[^ ...]**
beginning, end of line **^ , $**
beginning, end of word \< , \>
grouping **\(: : : \)**
contents of n th grouping **\n**

**Counts** Praticamente ogni comando puo' essere preceduto da un numero che specifica quante volte debba essere eseguito.

**Ranges** I range possono precedere la maggior parte dei comandi *":"*, facendo in modo che siano eseguiti su una o piu' linee. I range in genere sono combinati con il comando :s per fare sostituzione su piu' righe.
lines n-m **:n ,m**
current line **:.**
last line **:$**
marker c **:'c**
all lines **:%**
all matching lines **:g/pattern /**

**Files** write file (current file if no name given) **:w file**
read file after line **:r file**
next file **:n**
previous file **:p**
edit file **:e file**
replace line with program output **!!program**

**Varie** toggle upper/lower case **~**
join lines **J**
repeat last text-changing command **.**
undo last change, all changes on line **u , U**

# 4 I comandi principali di un sistema GNU/Linux

Vedi Bibliografia.

# 5 Appendice A - Regular Expressions (cenni)

L'espressione regolare è un modo per definire la ricerca di stringhe attraverso un modello di comparazione. Viene usato da diversi programmi di servizio, ma non tutti aderiscono agli stessi standard. In questo capitolo si vuole descrivere lo standard POSIX al riguardo.

Per studiare la grammatica delle espressioni regolari, occorre abbandonare qualunque resistenza, tenendo presente che l'interpretazione di queste espressioni va fatta da sinistra a destra; inoltre, ogni simbolo può avere un significato differente in base al contesto in cui si trova.

Un'espressione regolare, come definita nello standard POSIX 1003.2, può essere espressa attraverso due tipi di grammatiche differenti: le espressioni regolari di base, o elementari, identificate dall'acronimo BRE (Basic regular expression), e le espressioni regolari estese, identificate dall'acronimo ERE (Extended regular expression). La grammatica delle espressioni regolari tradizionali degli ambienti Unix viene identificata dall'acronimo SRE (Simple regular expression); in generale, per fare riferimento a espressioni regolari non meglio definite, si usa anche soltanto l'acronimo RE.

Un'espressione regolare è una stringa di caratteri, che nel caso più semplice rappresentano esattamente la corrispondenza con la stessa stringa. All'interno di un'espressione regolare possono essere inseriti dei caratteri speciali, che permettono di rappresentare delle corrispondenze in situazioni più complesse. Per fare riferimento a tali caratteri in modo letterale, occorre utilizzare delle tecniche di protezione, che variano a seconda del contesto.

La corrispondenza tra un'espressione regolare e una stringa, quando avviene, serve a delimitare una sottostringa che può andare dalla dimensione nulla fino al massimo della stringa di partenza. È importante chiarire che anche la corrispondenza che delimita una stringa nulla può avere significato, in quanto identifica una posizione precisa nella stringa di partenza. In generale, se sono possibili delle corrispondenze differenti, viene presa in considerazione quella che inizia il più a sinistra possibile e si estende il più a destra possibile.

**Ancoraggio iniziale e finale** In condizioni normali, un'espressione regolare può individuare una sottostringa collocata in qualunque posizione della stringa

di partenza. Per indicare espressamente che la corrispondenza deve partire obbligatoriamente dall'inizio della stringa, oppure che deve terminare esattamente alla fine della stringa stessa, si usano due ancore, rappresentate dai caratteri speciali ^ e $, ovvero dall'accento circonflesso e dal dollaro.

Per la precisione, un accento circonflesso che si trovi all'inizio di un'espressione regolare identifica la sottostringa nulla che si trova idealmente all'inizio della stringa da analizzare; nello stesso modo, un dollaro che si trovi alla fine di un'espressione regolare identifica la sottostringa nulla che si trova idealmente alla fine della stringa stessa.

**Delimitazione di una o più sottoespressioni** Una sottoespressione è una porzione di espressione regolare individuata attraverso dei delimitatori opportuni. Per la precisione, si tratta di parentesi tonde normali nel caso di espressioni regolari ERE (estese), oppure dei simboli \( e \) nel caso di espressioni regolari BRE.

La delimitazione di sottoespressioni può servire per regolare la precedenza nell'interpretazione delle varie parti dell'espressione regolare, oppure per altri scopi che dipendono dal programma in cui vengono utilizzate. In generale, dovrebbe essere ammissibile la definizione di sottoespressioni annidate.

**Corrispondenza con un carattere singolo** In un'espressione regolare, qualsiasi carattere che nel contesto non abbia un significato particolare, corrisponde esattamente a se stesso.

Il carattere speciale . (il punto), rappresenta un carattere qualunque, a esclusione di <NUL>.

È possibile definire anche la corrispondenza con un carattere scelto tra un insieme preciso, utilizzando una notazione speciale, ovvero un'espressione tra parentesi quadre:

- [elenco_corrispondente]
- [^elenco_non_corrispondente]

**Corrispondenze multiple** Alcuni caratteri speciali fungono da operatori che permettono di definire e controllare il ripetersi di un modello riferito a un carattere precedente, a una sottoespressione precedente.

In tutti i tipi di espressione regolare, l'asterisco (*) corrisponde a nessuna o più ripetizioni di ciò che gli precede. Nel caso di espressioni regolari ERE si possono utilizzare anche gli operatori + e ?, per indicare rispettivamente una o più occorrenze dell'elemento precedente, oppure zero o al massimo un'occorrenza di tale elemento.

Le espressioni regolari BRE e ERE permettono l'utilizzo di un'altra forma più precisa e generalizzata per esprimere la ripetizione di qualcosa.

- {n} : si intende indicare la ripetizione di n volte esatte l'elemento precedente

# 6 Appendice B - I package manager

Ogni distribuzione presenta all'amministratore un gestore per i propri pacchetti, che oltre ad occuparsi dell'installazione ordinata degli stessi, vanno ad aggiornare un file (o un database transazionale) nei casi piu' evoluti, che permette di gestire oltre a installazione e disinstallazione, le revisioni, delle informazioni ed il roll back di installazioni non andate a buon fine.

Questi sistemi sono molto potenti e ordinati rispetto ai corrispondenti dei sistemi operativi propietari. Oltre ad ogni package manager viene anche fornito uno strumento grafico che permette di mascherare alcune delle complessita'.

La maggior parte delle distribuzioni si e' orientata verso due tipi di package manager (sebbene altre come Slackware ne utilizzino altri):

- dpkg - Il package manager creato da Debian
- rpm - Il package manager creato da RedHat

Oltre ai package manager, esistono dei tool che ad essi si interfacciano e che permettono di fare ricerche e scaricare pacchetti dalla rete e successivamente installarli. Naturalemente l'amministratore dovra' configurare (o chi ha creato la distribuzione l'ha fatto per lui) i repository in cui cercare e importare le opportune chiavi crittografiche (per es con il comando rpm –import http://download.fedora.redhat.com/pub/fedora/linux/ GPG-KEY-fedora) che permettono di verificare l'autenticita' dei pacchetti. Quelli utilizzati dai precendenti package manager sono:

- apt
- yum

**Debian dpkg** Dpkg è un sistema di gestione pacchetti delle distribuzioni GNU\Debian e derivate, di seguito i principali usi.

- dpkg -i sintassi: dpkg -i nomepacchetto.deb Installa un pacchetto in formato .deb
- dpkg -r sintassi: dpkg -r nomepacchetto Disinstalla dal sistema il pacchetto specificato
- dpkg -s sintassi: dpkg -s nomepacchetto Mostra informazioni dettagliate su un pacchetto
- dpkg -l sintassi: dpkg -l Restituisce la lista dei pacchetti installati su un sistema

**Debian APT** Apt è una funzionalità della shell disponibile nelle distribuzioni Debian e derivate. Permette di prelevare pacchetti dalla rete e di installarli automaticamente, con l'utilizzo di pochissimi comandi. Vediamo ora i suoi usi.

- apt-cache search sintassi: apt-cache search nomepacchetto Cerca un pacchetto nella lista
- apt-cache show sintassi: apt-cache show nomepacchetto Serve per ottenere informazioni complete riguardo il pacchetto che si specifica.
- apt-get install sintassi: apt-get install nomepacchetto Usate questo comando per installare un pacchetto. Se il pacchetto è già installato nel sistema verrà effettuato, se disponibile, il suo aggiornamento
- apt-get remove sintassi: apt-get remove nomepacchetto Disinstallare un pacchetto.
- apt-get update
- apt-get update Aggiornare la lista dei pacchetti che è possibile installare sul proprio sistema. E' bene effettuare questo comando frequentemente, per mantere l'elenco aggiornato.
- apt-get upgrade sintassi: apt-get upgrade Aggiorna tutti i pacchetti possibili del sistema.

**RedHat rpm** Quella che segue e' una breve guida:

- L'opzione -q introduce una richiesta di informazioni.

| **Comando** | **Descrizione** |
|---|---|
| rpm -qpi *archivio_rpm* | Mostra una descrizione del contenuto dell'archivio RPM. |
| rpm -qpl *archivio_rpm* | Mostra l'elenco dei file contenuti nell'archivio RPM e dove verrebbero collocati se lo si installa. |
| rpm -q*a* *archivio_rpm* | Mostra l'elenco dei pacchetti RPM installati, così come sono stati registrati nel sistema RPM. |
| rpm -qf *archivio_rpm* | Determina il nome del pacchetto da cui proviene il file indicato come argomento. |

- L'opzione -i introduce una richiesta di installazione di un pacchetto.

| Comando | Descrizione |
|---|---|
| rpm -i *archivio_rpm* | Installa il pacchetto contenuto nell'archivio indicato se non si verificano errori. |
| rpm -i *uri_ftp_archivio_rpm* | Installa il pacchetto contenuto in un archivio identificato dall'URI indicato se non si verificano errori. |
| rpm -ivh *archivio_rpm* | Installa il pacchetto contenuto nell'archivio indicato, se non si verificano errori, mostrando qualche informazione e una barra di progressione. |
| rpm -i *–nodeps archivio_rpm* | Installa il pacchetto contenuto nell'archivio indicato, senza verificare le dipendenze tra i file. |
| rpm -i *–replacefiles archivio_rpm* | Installa il pacchetto contenuto nell'archivio indicato, senza verificare se vengono sovrascritti dei file. |
| rpm -i *–ignorearch archivio_rpm* | Installa il pacchetto contenuto nell'archivio indicato, senza verificare l'architettura dell'elaboratore. |
| rpm -i *–ignoreos archivio_rpm* | Installa il pacchetto contenuto nell'archivio indicato, senza verificare il tipo di sistema operativo. |

- Le opzioni -U e -F introducono una richiesta di aggiornamento di un pacchetto.

| Comando | Descrizione |
|---|---|
| rpm -U *archivio_rpm* | Aggiorna o installa il pacchetto contenuto nell'archivio indicato, se non si verificano errori. |
| rpm -Uvh *archivio_rpm* | Aggiorna o installa il pacchetto contenuto nell'archivio indicato, se non si verificano errori, mostrando qualche informazione e una barra di progressione. |
| rpm -F *archivio_rpm* | Aggiorna il pacchetto contenuto nell'archivio indicato, solo se risulta già installata una versione precedente. |
| rpm -*F modello_archivi_rpm* | Aggiorna i pacchetti contenuti negli archivi indicati, che risultano già installati nelle loro versioni precedenti. |

- L'opzione -e introduce una richiesta di eliminazione di un pacchetto installato.

| Comando | Descrizione |
|---|---|
| rpm -e *nome_del_pacchetto_installato* | Elimina (disinstalla) il pacchetto. |

- L'opzione -V introduce una richiesta di verifica di un pacchetto installato.

| Comando | Descrizione |
|---|---|
| rpm -V *nome_del_pacchetto_installato* | Verifica che il pacchetto indicato risulti installato correttamente. |
| rpm -Vf *file* | Verifica il pacchetto contenente il file indicato. |
| rpm -Va | Verifica tutti i pacchetti. |
| rpm -Vp *archivio_rpm* | Verifica la corrispondenza tra l'archivio RPM indicato come argomento e quanto installato effettivamente. |

Alle volte, quando si installano o si vogliono eliminare dei pacchetti si incontrano dei problemi, perché il programma rpm impedisce di fare ciò che potrebbe essere dannoso e sembra originato a causa di un errore. A questo proposito vale la pena di conoscere alcune opzioni speciali:

| Comando o opzione | Descrizione |
|---|---|
| –oldpackage | Permette di aggiornare un pacchetto utilizzando una versione precedente a quella che appare essere già installata. |
| –replacefiles | Permette di installare o aggiornare un pacchetto quando questo fatto implica la sostituzione di file già esistenti che appartengono ad altri pacchetti. |
| –replacepkgs | Permette di installare un pacchetto anche quando questo risulta già installato. |
| –force | È l'equivalente delle opzioni –oldpackage, –replacefiles e –replacepkgs, messe assieme. |
| –nodeps | Installa, aggiorna o disinstalla senza curarsi delle dipendenze da file o da altri pacchetti. |
| rpm –setperms -a | Verifica ed eventualmente corregge i permessi dei file di tutti i pacchetti installati. |
| rpm –setugids -a | Verifica ed eventualmente corregge la proprietà dei file di tutti i pacchetti installati. |

### RedHat Yellowdog updater modified (YUM)

- yum - display brief help
- yum check-update - update headers and display any updates the current system needs
- yum update - check for updates and apply them interactively
- yum -y update - check for updates and apply them with
- yum update <package> - check for updates and upgrade the specified package(s) only
- yum info - similar output to a rpm -qai
- yum info <package> - information about a specific package
- yum list - lists all available packages
- yum list <package> - list individual package(s)
- yum list installed - list all installed packages
- yum list available - list all packages not installed
- yum list update - list all packages that need to be upgraded

- yum list extras - list all installed packages that are not available from any of the defined yum resources (defined in the /etc/yum.conf file)
- yum clean - delete any rpms in the yum cache and remove any unneeded headers
- yum install <package> - install the package
- yum remove <package> - delete the package
- yum provides <file> - find out what package provides a particuar file
- yum search <string> - searches for packages containing the string in their name or header info

# References


[1] Appunti di informatica libera - Appunti Linux Copyright © 1997-2000 Daniele Giacomini Appunti di informatica liberaCopyright © 2000-2006 Daniele Giacomini Via Morganella Est, 21 – I-31050 Ponzano Veneto http://na.mirror.garr.it/mirrors/appuntilinux/HTML/a2.htm

[2] By Cameron Newham - Learning the bash Shell, Third Edition (O'Reilly)

[3] Vi Reference Card - http://www.eggfaq.com/docs/vi.pdf

[4] Vi Editor Reference - http://www.washington.edu/computing/unix/viref.html

[5] GNU/Linux Command-Line Tools Summary - http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/index.html

# Contents