GNU/Linux

Processi di elaborazione

Procedura di inizializzazione del sistema

Il primo processo che viene avviato è **init**. Questi leggendo il file **/etc/inittab**, il quale a sua volta fa riferimento a gli script contenuti in **/etc/rc.d**, avvia i servizi e le shell.

E' importante capire che a seconda della modalità in cui si vuole portare il sistema operativo saranno attivati determinati gruppi di script. In questo modo è possibile attivare il sistema con o senza supporto della rete, spegnerlo o compiere altre operazioni.

Questi gruppi di operazioni sono detti *run-level* e ad ognuno di questi corrisponde una directory chiamata **rc*n*.d** dove la *n* indica il livello. Ognuna di questi directory contiene i collegamenti simbolici agli script di inizializzazione (raccolti nella directory **/etc/init.d**).

Ogni collegamento segue la convenzione di iniziare con una **K** o un a **S** (a seconda si voglia iniziare o terminare un servizio), un numero per darne la priorità all'avvio ed il nome dello script. Ad esempio:

```
k85bind9
```

termina il servizio di bind attraverso il comando

```
/etc/init.d/bind9 stop
```

Nel caso di RH 4 bisogna usare il comando chkconfig. Ad esempio:

```
chkconfig --level 345 tomcat on
```

permette di abilitare per i run-level 3,4 e 5 il servizio tomcat. Il comando

```
chkconfig --list
```

elenca lo stato dei servizi per ogni runlevel

Gestione dei processi

Lo stato dei processi può essere visualizzato mediante il comando ps che elenca i processi con le loro caratteristiche:

```
ps al
ps lax
```

e il programma pstree che permette di vedere graficamente la le dipendenze gerarchiche:

```
pstree –u –p
```

Altrettanto utile può essere usare il comando **grep** per selezionare i processi da vedere. Ad esempio:

```
ps auxww | grep ^postgres
```

mostrerà solo le righe che iniziano con l'utente postgres.

Un informazione fondamentale di un processo è il suo Process ID (PID) che identifica il processo e ne permette l'invio di segnali. Ad esempio volendo terminare un processo si può usare kill:

```
kill –s 9 1023
```

Un processo può essere avviato in primo piano ma può essere anche avviato sullo sfondo terminando la riga di comando semplicemente con uno spazio ed una &. Ad esempio:

```
make bzimage > ~/make.msg &
```

Molto spesso è altrettanto importante conoscere quale processo sta utilizzando un file. Ad esempio per poter *smontare* un file system che ci dà errore poichè occupato. Il comando può contenere l'opzione **–k** che ucciderà il processo bloccante:

```
fuser –um /dev/usbdrive
```

ritornerà il PID e l'utente che stanno utilizzando quel device. A questo punto posso usare il comando kill seguito dal PID.

Infine la visualizzazione in tempo reale dei processi può essere effettuata con il comando **top** mentre la memoria del sistema può essere visualizzata con **free**.

Pianificazione dei processi

Attraverso il demone **cron** è possibile pianificare operazioni ripetitive. Ogni utente ha un proprio file che elenca la periodicità e il comando da eseguire. Tali file sono generalmente memorizzati in

```
/var/spool/cron/crontabs
```

Ed hanno una sintassi de tipo:

```
...
#esegue lo script ciao.sh alle 1.45 del giorno 4 di ogni mese
45 1 4 * *     ciao.sh
...
#esegue lo script monitor.sh ogni 10 minuti tutti i giorni
*/10 * * * *   monitor.sh
...
```

Il singolo utente può creare o modificare l'elenco dei lavori da eseguire utilizzando l'apposito programma crontab:

```
crontab –e
```

modifica il file crontab dell'utente

```
crontab -l
```

elenca il contenuto del file crontab

Esistono inoltre altre possibilità: In **/etc/crontab** vengono eseguiti job specificando dopo la periodicità il nome dell'utente che esegue i comandi. In tale file, inoltre, si specifica quando eseguire gli script contenuti nelle comodissime directory:

```
/etc/cron.hourly
/etc/cron.dayly
/etc/cron.weekly
/etc/cron.monthly
```

in questo modo è sufficiente posizionare uno script in queste directory affinchè venga eseguito con la giusta cadenza.

L'esecuzione di tali script avviene utilizzando il comando di shell **run-parts** seguito dalla directory che contiene gli script

Infine la directory **/etc/cron.d** contiene ulteriori file di tipo crontab che il sistema eseguirà oltre a quello principale. In questo modo un'applicazione può posizionare lì il suo file crontab senza modificare quello di sistema.

Data e ora di sistema

Il comando date ci permette di impostare o conoscere la data e ora di sistema. Ad esempio:

```
date –d %d
```

ritorna il giorno (numero) del mese. Per completezza si hanno le seguenti direttive:

Direttiva	Descrizione
`%%`	Rappresenta un simbolo di percentuale singolo.
`%n`	Rappresenta un codice di interruzione di riga.
`%t`	Rappresenta una tabulazione orizzontale.
`%s`	Numero di secondi trascorsi dall'epoca di riferimento (1970.01.01 00:00:00 UTC).

%c	Data e ora corrispondente alla stringa **''%a %b %d %X %Z %Y**''.
%H	L'ora secondo il formato «00..23».
%I	L'ora secondo il formato «01..12».
%k	L'ora secondo il formato « 0..23».
%l	L'ora secondo il formato « 0..12».
%M	Minuti secondo il formato «00..59».
%S	Secondi secondo il formato «00..59».
%p	AM o PM.
%Z	Sigla del fuso orario o nulla se non è determinabile.
%r	Orario in 12 ore, secondo il formato «hh:mm:ss AM/PM».
%T	Orario in 24 ore, secondo il formato «hh:mm:ss».
%X	Orario corrispondente alla stringa **''%H:%M:%S**''.
%a	Giorno della settimana abbreviato.
%A	Giorno della settimana esteso.
%U	Settimana dell'anno, utilizzando la domenica come primo giorno, «00..53».
%w	Giorno della settimana, con lo zero corrispondente alla domenica, «0..6».
%b	Mese abbreviato.
%h	Come **'%**b'.
%B	Mese per esteso.
%m	Mese secondo il formato «01..12».
%y	Le ultime due cifre dell'anno, «00..99».
%Y	Anno per esteso.
%d	Giorno del mese secondo il formato «01..31».
%j	Giorno dell'anno secondo il formato «001..366».
%D	Data secondo il formato «mm/gg/aa».
%x	Rappresentazione locale della data in forma numerica.

Operazioni sul file-system

Il concetto di inode

A livello più basso i disco è composto da blocchi dati. Ogni file è memorizzato in un insieme di questi. L'**inode** è l'elemento del file system, identificato con un numero, che associa ad ogni file (directory comprese) i blocchi in cui è memorizzato.

Un inode contiene anche altre informazioni come la data di ultimo accesso, i permessi ed un contatore delle voci di directory che vi fanno riferimento (ovvero, un contatore di collegamenti fisici).

La **directory** è un file (e quindi è essa stessa un inode) che associa ai nomi dei file contenuit in essa il corrispondente inode. Tale associazione si chiama **collegamento fisico** o *hard link*.

E' possibile quindi che un singolo inode venga puntato da più nomi. Se ci troviamo nella directory radice, ad esempio, la voce . e .. punteranno allo stesso inode.

E' interessante notare che quando si cancella un file avvengono le seguenti operazioni:

- si cancella la voce dalla directory
- si scala il contatore dei collegamenti fisici all'inode.
- se questo è diventato zero l'inode è diventato libero ed il file è stato effettivamente cancellato.

Oltre ai collegamenti fisici esistono dei file, detti *symlink* (**collegamenti simbolici**), che sono file contenenti un riferimento ad un altro file. Ad esempio:

```
ln –s /bin/sh ./sh
```

crea un collegamento, chiamato sh, al programma /bin/sh nella directory corrente.

Attivazione di un file system

Per poter accedere ad un'unità di memorizzazione è necessario innestare il suo file system in quello globale. Tale operazione riguarda partizioni, supporti rimuovibile (cd-rom, floppy…) ecc.

Tale operazione avviene utilizzando il comando **mount**. Alcuni esempi:

```
mount –t vfat /dev/fd0 /mnt/floppy
```

monta un dischetto ms-dos sulla directory /mnt/floppy

```
mount –t iso9660 /dev/cdrom /mnt/cdrom
```

monta un cd-rom. Per riappropriarsi del supporto occorre smontarlo utilizzando **umount**:

```
umount /mnt/floppy
```

oppure

```
umount /dev/fd0
```

Una volta montato può essere utile conoscere le caratteristiche dei file system in utilizzo. Ad esempio il comando

```
df
```

mostra l'utilizzo di tutti i filesystem (numero di blocchi, spazio su disco utilizzato...) ed il punto di mount.
Il comando

```
du –c nome_directory
```

ritorna le dimensioni della directory.

Gerarchia del file system

Descriviamo di seguito la struttura essenziale di un file system secondo il documento FHS (*Filesystem hierarchy standard*)

`/bin`	Contiene gli eseguibili comuni a tutti gli utenti. La directory /sbin invece, dovrebbe raccogliere gli eseguibili di pertinenza dell'utente root
`/boot`	Contiene le immagini del kernel e i file di configurazione utili per il boot
`/dev`	Sotto Unix anche i dispositivi fisici sono visti come file. Vengono elencati in questa directory
`/etc`	Contiene i file di configurazione del sistema
`/home`	Contiene le cartelle dei singoli utenti
`/lib`	Contiene le librerie condivise essenziali ai moduli del kernel e ai programmi contenuti in /bin e /sbin
`/mnt`	E' il punto di innesto per inserire temporaneamente altri file system
`/opt`	Dovrebbe contenere gli applicativi aggiuntivi
`/root`	Cartella personale dell'utente root
`/temp`	Contiene i file temporarei
`/usr`	Contiene una gerarchia secondaria di dati statici e condivisibili. Al suo interno trovano posto /bin e /sbin dove risiederebbero i programmi di utilizzo meno comune. Altreattanto importante è /usr/share utile per condividere dati indipendenti dall'architettura
`/var`	Contiene i dati variabili. Più precisamente: /var/lock contiene i file che indicano che una certa risorsa è impegnata /var/log contiene i file di log di sistema e dei programmi /var/mail contiene le caselle di posta degli utenti /var/run contiene le informazioni che riguardano l'esecuzione dei processi /var/spool contiene le code di stampa e le alte code di elaborazione

Operazioni con le directory

Le directory sono file speciali che servono a contenere elenchi di file. Vediamo i principali comandi con a fianco degli esempi autoesplicativi:

mkdir	`mkdir dir1`	crea la directory dir1
	`mkdir –p dir1/dir2`	crea la directory dir1 poi dentro di essa la directory dir2
rmdir	`rmdir dir1`	rimuove dir1 se è vuota
pwd	`pwd`	indica il percorso della directory corrente
basename	`basename "d1/d2/ciao.txt" ".txt"`	ritorna il nome del file senza percorso e suffiso (opzionale). In questo caso "ciao"
dirname	`dirname "/di/d2/ciao.txt"`	estrae il percorso. In questo caso "/d1/d2"
ls	`ls –a`	mostra tutti i file (anche quelli nascosti che iniziano con il punto .)
	`ls –l`	mostra tutte le caratteristiche dei file nella directory
	`ls –I \*.html`	mostra tutti i file tranne quelli con estensione .html

file	`file ciao.txt`	cerca di ritornare il tipo di file
cksum	`cksum ciao.txt`	ritorna un valore di checksum del file
md5sum	`md5sum ciao.txt`	ritorna la firma md5 del file

Operazioni con i file

E possibile operare sui file attraverso i seguenti comandi:

cp	`cp pr* /home/tizio`	copia tutti i file che iniziano per pr in /home/tizio
	`cp –f –R /test/* /prova`	copia **il contenuto**, e le sottodirectory, di /test, in /prova sovrascrivendo i file con lo stesso nome
	`cp –R /test /prova`	copia la directory /test, e le sottodirectory, in /prova
	`cp –Rd /test /prova`	mantiene i **collegamenti simbolici** come tali
mv	`mv prova.txt /test/prova1.txt`	sposta prova.txt in /test/prova1.txt
rm	`rm –r –f /prova`	cancella tutti la directory /prova e tutte le directory in essa contenute senza chiedere conferma (-f)
tar	`tar cf /tmp/archivio.tar /usr`	Archivia in archivio.tar (-cf) il contenuto di /usr
	`tar f /tmp/archivio.tar /root`	Aggiunge a archivio.tar il contenuto di /root
	`tar xf archivio.tar`	Esplode l'archivio nella posizione corrente
	`tar cf /dev/fd0 –L 1440 –M /usr`	Archivia in un set di dischetti (-M) il contenuto di /usr
	`tar xf /dev/fd0 –L 1440 –M –p`	Recupera dal set di dischetti l'archivio corrente preservando i permessi (-p) nella posizione corrente
	`tar xvzf pippo.tar.gz`	esplode l'archivio compresso (-z) pippo.tar.gz nella posizione corrente visualizzando i file estratti (-v)
gzip	`gzip -9 *.txt`	Comprime tutti i *.txt sostituendo gli originali con i compressi *.txt.gz col massimo livello di compressione
	`gzip –d pippo.txt.gz`	Sostituisce pippo.txt.gz con pippo.txt decompresso

Spesso è molto importante effettuare ricerche di file o parole al loro interno:

grep	`grep –F –e ciao –i –n *`	Visualizza la riga e il numero di riga (-n) in cui presente la parola ciao in tutti i file della directory corrente ignorando maiscuole e minuscole (-i)
	`grep –F –e ciao –l`	Emette solo i nomi dei file per i quali la ricerca ha avuto successo
	`grep –q file.txt occorrenza`	Ritorna 0 se l'occorenza è presente nel file almeno una volta
find	`find / -name "lib*" –print`	Trova a partire da / tutti i file che iniziano con lib
	`find / -atime +3 –print`	Trova i file la cui data di accesso è più vecchia di 3 giorni
	`find / -size +5000k –print`	Trova i file più grandi di 5000 Kbyte

Proprietà di un file

Le proprietà di un file riguardano il proprietario ed i permessi su di esso. In particolare un file può avere permessi di lettura, scrittura ed esecuzione (RWX) distinti tra proprietario, gruppo, tutti gli utenti.

Tali permessi possono essere espressi attraverso una stringa oppure in notazione ottale. Ad esempio

-rwxr-xr-x

che equivale a

0755

ed indica che il file in questione può essere letto, eseguito,scritto dal proprietario e letto ed eseguito dal gruppo di appartenenza e dai restanti utenti.

Un altro aspetto importante è la possibilità di un file di assumere in esecuzione i privilegi del proprietario qualora venga eseguito da un utente diverso. In pratica se il proprietario di un file eseguibile è l'utente root ed è settato il bit **SUID** allora tale eseguibile, anche se lanciato da un utente diverso da root, avrà gli stessi privilegi di root.

chown	`chown tizio:user miofile`	cambia il proprietario del file miofile in tizio ed il gruppo in user
	`chown –r tizio:user miadir`	agisce ricorsivamente all'interno della directory
	`chown –r –from=tizio caio miadir`	trasferisce la proprietà di ciò che era di tizio a caio nella directory miadir e nelle sottodirectory
chmod	`chmod +x script.sh`	rende eseguibile uno script
	`chmod u+s eseguibile`	da gli stessi privilegi del proprietario al file quando viene mandato in esecuzione

In ogni caso un comando, o uno script, può essere eseguito con i privilegi di root facendolo precedere dal comando **sudo**. Gli utenti che possono invocare tale comando sono elencati in **/etc/sudoers**.

VI

Modalità di funzionamento di VI

VI è un editor di testo piuttosto diffuso sotto UNIX e quindi anche sotto GNU/Linux. Ha due modalità di funzionamento:

- **command mode**, una modalità in cui l'editor accetta comandi per modificare il testo. VI si trova in questa modalità appena caricato.

- **insert mode**, utilizzando il comando **i** oppure **a** si entra in insert mode e si può inserire il testo. Premendo il **tasto escape** si torna in command mode.

Comandi

La sintassi dei comandi di VI è la seguente:

[ripetizioni] comando [dove]

Per *ripetizioni* si intende un numero da 0 a 9 che indica quante volte ripetere il comando. Ad esempio:

23x

Cancella 23 caratteri a partire da quello sotto il cursore. Il parametro *dove* permette di specificare la parte di documento (ad esempio il numero di linee) che il comando interesserà.

a	Immette in insert mode facendoci scrivere **dopo** il cursore
i	Immette in insert mode facendoci scrivere **prima** del cursore
h j k l	Muove il cursore rispettivamente a sinistra, sotto, sopra, destra (spesso funzionano anche le freccette)
u	Undo l'ultimo cambiamento al file
w w nomefile w >> nomefile wq	Scrive il buffer Scrive il buffer con il nuovo nome nomefile Scrive in coda il buffer ad un file di nome nomefile Scrive il buffer ed esce
q q!	Esce Esce senza salvare

Cancellare, copiare e incollare

dd	Cancella la riga corrente e inserisce la riga nel buffer (se non selezionato precedentemente lo mette in quello di default
D	Cancella a partire dal cursore fino alla fine della riga
p	Incolla l'intera riga a partire dall'inizio
P	Incolla **prima** del cursore
yy	Copia l'intera riga a partire dall'inizio

Programmazione della Bourne Shell

Avvio

Al momento dell'avvio la shell esegue, solitamente, il contenuto dei file

```
/etc/profile
```

reltativamente a tutti gli utenti. E

```
~/.profile
```

relativamente ad ogni singolo utente. In questo modo è possibile dare delle impostazioni generali e delle impostazioni personalizzate per ognuno. Il file /etc/profile può essere particolarmente utile per indicare nuove variabili d'ambiente del tipo:

```
JAVA_HOME=/usr/java/j2se_1.4
export JAVA_HOME
```

e aggiungere percorsi alla variabile d'ambiente PATH

```
PATH=${PATH}:/usr/java/j2se_1.4/bin
```

Convenzioni fondamentali

Gli script della shell sono geralmente dei file eseguibili .sh che hanno in testa la seguente specificazione:

```
#!/bin/sh
```

Per essere eseguibili devono avere il permesso di esecuzione:

```
chmod +rx script
```

Volendo rendere pubblico a tutti gli utenti lo script possiamo copiarlo in

```
/usr/local/bin
```

in questo modo chiunque potrà usarlo chiamandolo semplicemente per nome.

La shell tende a rimpiazzare variabili o simboli speciali con quello che rappresentano. Per evitare questo comportamento ci sono tre modi (*quoting*):

- backslash \ : preserva il significato letterale del carattere

Esempi

```
ls –l \*
```

tenta di visualizzare le caratteristiche del file *. Se **non si usasse** \ visualizzerebbe tutti i file

```
… prima r\
iga…
```

spezza una riga singola su due righe

- apice singolo ' : mantiene il valore letterale dell'intera stringa

Esempi

```
echo 'Attenzione $0 resta "inalterato"'
```

```
Attenzione $0 resta “inalterato”
```

- apice doppio **“** : effettua la sostituzione quando incontra **$** per le variabili e ` (backtick alt-96) per il risultato di un comando

Esempi

```
echo “Attenzione \$0 contiene:\” $0\” ”

Attenzione $0 contiene “-bash”

echo “La directory contiene i seguenti file temporanei: `ls *.tmp`”

La directory contiene i seguenti file temporanei: prova.tmp ...
```

Parametri e variabili

All’interno di uno script è possibile definire variabili, usare variabili di shell come $PATH e leggere i parametri passati dopo il nome dello script:

- $0 rappresenta il percorso completo dello script
- I parametri $1, $2…${10}… rappresentano i parametri a linea di comando
- il numero di parametri è dato da $#
- $@ è un enumerazione dei parametri passati allo script
- $* una singola stringa contenente tutti i parametri passati allo script
- {***n*:-*default***} impone al parametro un valore di default se non è settato

L’ultimo parametro si può ottenere nel seguente modo:

```
args=$#
lastarg=${!args}
```

Utilizzando il comando shift è possibile scorrere di una posizione i parametri. In questo modo $n doventa $n-1 ma $0 resta tale:

```
until [ -z  “$1” ]
do
        echo –n “$1 ”
        shift
done

exit 0
```

Molto spesso invece dei semplici parametri posizionali si desiderebbe utilizzare gli switch tipici dei comandi tradizionali. Ad esempio:

```
scriptname –abc –e /usr/local
```

Tutto ciò può essere facilmente raggiunto usando getopt. Questo costrutto utilizza due variabili implicite:

- **$OPTIND** (OPTion INDex): il puntatore degli argomenti
- **$OPTARG** (OPTion ARGument): l’argomento opzionale dopo lo switch

Vediamo un esempio che ci aiuta a capirne l’utilizzo:

```
#inizia un ciclo while paticolare per analizzare le opzioni a riga di comando
while getopts “:mnoq:r” Option
do
        case $Option in
        m)      echo “opzione –m”;;
        n | o)  echo “opzione -$Option”;;
        q)      echo “opzione –q con argomento \”$OPTARG\” “;;
        r)      echo “opzione –r”;;
        *)      echo “opzione non implementata”;;
        esac
done
```

```
shift $(($OPTIND – 1))
# con questa operazione $1 punta al primo argomento NON preceduto da una opzione
# presente sulla linea di comando
```

Alcune osservazioni:

- Il pattern ":mnoq:r" significa che lo script accetta un primo parametro NON preceduto da opzione (**:**...) e che q deve essere seguito da un argomento (...**q:**...) altrimenti non viene riconosciuto e ricade nella opzione finale di default.

Globbing

Il processo che traduce parametri, variabili ecc. nel loro risultato finale si dice **espansione (Globbing)**. Alcuni esempi:

```
Mkdir /usr/{uno,due,tre}
```

Crea le directory

```
/usr/uno
/usr/due
/usr/tre
```

La tilde (alt 126) si riferisce all'utente:

```
Cd ~tizio
```

Sposta nella directory personale dell'utente tizio.

Per far riferimento ad una variabile si antepone il **$**. In alcuni casi (vedi il caso di parametri posizionali con più cifre decimali) è comodo utilizzare la forma:

${nome_variabile | nome_parametro**}**

Infine possiamo effettuare il riferimento indiretto di una variabile mediante un'altra variabile che ha come valore il suo nome. Ad esempio:

```
a=lettera
lettera=z

eval a=\$$a
echo "$a"      #stampa z
```

Espressioni matematiche

Possiamo assegnare anche espressioni matematiche. In questo caso dobbiamo però racchiuderle tra doppie parentesi:

$((espressione**))**

o parentesi quadre

[espressione**]**

Le espressioni matematiche sono rifrerite a valori interi; per la precisione sono *signed long* a 32 bit. Gli operatori utilizzabili sono +,-,*,/,**(elevamento a potenza) e % (modulo, cioè il resto di una divisione intera).

Inoltre è possibile ottenere numeri random utilizzando la variabile di sistema **$RANDOM** che ritorna un intero nell'intervallo [0 – 32767].

Se si usano valori decimali essi verranno interpretati come stringhe. Quindi per effettuare calcoli in virgola mobile è necessario usare **bc**.

Volendo assegnare ad una variabile un valore esadecimale o ottale è necessario usare la seguente espressione:

```
let "oct = 032"       #26 in decimale
let "hex = 0x32"      #50 in decimale
```

Esecuzione di comandi

E' possibile effeettuare la sostituzione non solo con parametri o variabili ma anche con il risultato derivante dall'esecuzione di un comando. Alcuni esempi:

```
ELENCO=`ls`
```

Assegna alla variabile elenco il listing della directory corrente.

```
Rm $(find / -name "*.tmp")
```

Elimina da tutto il file system i file *.tmp a partire da una lista generata dal comando find.

Oltre alla forma $(…) si possono usare gli apici inversi (backtick). Nel caso in cui ci siano comandi annidati, nel comando più interno occorre far precedere il backtick da una barra rovesciata:

```
`comandoExt\`comandoInt\`comandoExt`
```

Facciamo notare che la shell effettua la **suddivisione in parole** di ogni possibile globbing che non avvenga all'interno dei doppi apici. Ad esempio:

```
$pippo = "b* d*"
echo $pippo
echo "$pippo"
echo '$pippo'
```

Produrrà i seguenti risultati:

```
bin boot dev
b* d*
$pippo
```

Anche i **percorsi** subiscono l'espansione. Infatti, dopo l'espansione in parole la shell cerca i caratteri *** ? [**intervallo..caratteri**] [**elenco caratteri**]** e li utilizza come modello di sostituzione.

In particolare il simbolo **?** indica un carattere mentre ***** una stringa di lunghezza qualsiasi (anche nulla).
Infine si ricordi che per eseguire uno script esterno si scrive:

. *file_script* [argomenti]

Redirezioni dei comandi

Si possono definire degli alias dei comandi più comuni. Si ricordi che non è possibile usare gli alias all'interno degli script tuttavia possono essere inseriti in **/etc/profile** in modo da essere usati ad ogni avvio della shell.

Ad esempio:

```
alias rm="rm –i"
```

La redirezione può avvenire sia sovrascrivendo un file:

```
Ls > ./dir.txt
```

Che aggiungendo in coda:

```
Ls >> ./dir.txt
```

Lo **stderr** può essere rediretto mediante

```
2>
```

Mentre possiamo redirigere sia lo **stdout** che lo **stderr** verso il medesimo file mediante l'operatore

```
&>
```

Al trettanto interessante la possibilità di redirigere il contenuto di un file allo **standard input** di un programma. Ad esempio:

```
sort < ./elenco
```

produce in output l'elenco ordinato delle righe del file ./elenco.

Anche i blocchi di istruzioni possono essere utilizzati per la redirezione dell'I/O. Ad esempio

```
FILE=/etc/fstab

{
        read line1
        read line2
} < $FILE

echo "$line1\n$line2"
```

legge e, successivamente, stampa le prime due linee del file fstab. Si noti come le variabili definite nel blocco non siano locali al blocco. Allo stesso modo è possibile inviare lo stdout prodotto in un blocco ad un file:

```
{
        echo ciao
} > ciao.txt
```

E' possibile creare delle pipeline in cui **l'output del n-esimo** programma diventa **l'input dell'n+1-esimo**. Ad esempio:

```
cat prova.txt | /usr/bin/unix2dos | lpr
```

converte il contenuto del file prova.txt secondo la modalità del DOS e lo invia alla coda di stampa.

Un comando che può essere utilizzato all'inizio di una pipe per sveltire operazioni interattive è il comando **yes**. tal comando invia sullo stdin una sequenza di y seguiti da LF. Ad esempio:

```
yes | rm –r directory
```

equivale a

```
rm –rf directory
```

E' abbastanza evidente la capacità distruttiva di questo comando se non si sa quello che si sta facendo!

Infine, utilizzando l'operatore –, è possibile sostituire ad un nome di file lo stdin (o lo stdout). Si noti che questo non è un operatore della shell ma è un opzione di utilities come tar, cat...

Nel caso di tar, se le opzioni non ci soddisfano del tutto, invece di elencare i nomi dei file possiamo redirigere un elenco che soddisfi opportune condizioni. Ad esempio:

```
BACKUPFILE=backup-$(date +%m-%d-%Y)

#se non specifico un nome tra i parametri il nome sarà backup-MM-DD-YYYY.tar.gz
archive=${1:-$BACKUPFILE}

tar cvf -  `find . –mtime –1 –type f -print` > archive.tar
gzip $archive.tar
```

effettua un tar zippato di tutti i file che sono cambiati nelle ultime 24 ore

Here document

Attraverso il cosidetto here document è possibile specificare lo stdin di un comando all'interno di uno script di shell. La sintassi prevista è la seguente:

comando <<-DELIMITATORE

linee di input

DELIMITATORE

il segno – serve per consentire la tabulazione al corpo del documento in modo da presentare un aspetto più gradevole. Consideriamo ad esempio l'invio di un mail:

```
#rinveniamo il percorso dell'eseguibile netcat
NC="$(which nc)"

$NC –w 5 "smtp.server.it" "25" &> /dev/null <<-EOF
        HELO $(hostname)
        MAIL FROM: c@b.it
        RCPT TO: d@b.it
        DATA
        Subject: prova
        To: d@b.it
        From: c@b.it

        ciao, ciao

        .
        EOF

exit 0
```

I parametri che abbiamo utilizzato per sodisfare il protocollo SMTP potevano essere importati da un file esterno allo script utilizzando il comando **source** (abbreviato con il puntino **.**). Ad esempio:

. $HOME/mail.conf

richiamerà in quel punto dello script il file mail.conf

Valutazione dei comandi

Il comando **eval** combina gli argomenti in una espressione, o lista di espressioni, e li valuta espandendo ogni variabile contenuta in essa.

Strutture di controllo

Elenchiamo le principali strutture di controllo utilizzando esempi di script. Il comando **for** esegue una scansione di elementi in una lista ed in corrispondenza di questi esegue un comando. Ad esempio:

```
for i in 1 2 3 4 5
do
        echo $i
done
```

stamperà 1 2 3 4 5. Se chiamo stamp.sh il seguente script:

```
#!/bin/bash
for i in $@
do
        echo $i
done
```

e lo richiamo nel seguente modo:

```
./stampa.sh a b "c d"
```

si ha il seguente output

```
a
b
c d
```

Come lista sulla quale iterare è possibile inserire una variabile che funga da lista come la seguente:

```
lista="uno
due
tre
quattro"
```

Si noti che a partire da questa si può ottenere un vettore i cui elementi possono essere indirizzabili mediante l'operatore [ ]. Ad esempio:

```
vettore={$lista}              #ottengo il vettore
num_elementi=${#vettore[*]}   #ne ottengo il numero degli elementi
echo "${vettore[1]}"          #stampa il secondo elemento
```

Se la lista fosse stata un elenco di elementi separati da spazi (e non da newline) occorreva generare la lista all'interno del for:

```
lista="uno due tre quattro"

for number in `echo $lista`
do
        echo "$number"
done
```

Un altra interessante possibilità è sostituire ai parametri posizionali gli elementi che stiamo scorrendo con il for:

```
for persone in "Carlo 35" "Maria 33" "Sabina 21"
do
        set -- persone
        echo "$1  età $2"
done
```

Il for permette anche di iterare sul contenuto di una directory (PWD se non la specifichiamo). Ad esempio:

```
for file in [jx]*
do
        rm –f $file #rimuove tutti i file che iniziano per j o per x
done
```

Il comando **while** esegue un gruppo di comandi fino a che una certa condizione continua a dare risultato true. Ad esempio:

```
RISPOSTA="continua"
while [ "$RISPOSTA" != "fine" ]
do
        echo "usa la risposta fine per terminare"
        read risposta
done
```

Il comando **case** permette di eseguire una scelta nell'esecuzione di varie liste di comandi in base al confronto con un modello:

```
case $1 in
        -a | -A | --alpha)    echo "alpha"   ;;
        -b)                   echo "bravo"   ;;
        *)                    echo "opzione sconosciuta"    ;;
esac
```

viene controllato il primo argomento dello script con vari modelli dopo di che si esegue il comando corrispondente. La star l'opzione di default. Vediamo un esempio più interessante riferito ad un ipotetico script di inizializzazione di un serivizio:

```
#!/bin/sh
case $1 in
        start)
                echo –n "Avvio del servizio Pippo: "
                /usr/sbin/pippo &
                echo
                ;;
        stop)
                echo –n "Disattivazione del servizio Pippo:"
                killall pippo
                echo
```

```
                ;;
        restart)
                killall –HUP pippo
                ;;

        *)      echo “Utilizzo: pippo {start|stop|restart}”
                exit 1
esac

exit 0
```

Il comando **if** permette di effettuare scelte condizionali in funzioni di un parametro. La sua struttura generale è la seguente:

if *condizioni*
then
        comandi
[elif *condizioni*
then
        comandi]
...
[else
        comandi]
fi

Il caso particolare in cui ci sono solo due condizioni da verificare

```
if ! mkdir deposito
then
        echo “impossibile creare la directory deposito”
else
        echo “creata la directory deposito”
fi
```

l’esito dell’operazione viene negato da ! in modo da segnalare l’errore se il sistema non riesce a creare la directory.

Vediamo ora una funzione:

```
function verifica() {
        if  [ -e “/var/log/packages/$1” ]
        then
                return 0
        else
                return 1
        fi
}

if verifica nome_file
then
        echo “nome_file esiste”
else
        echo “nome_file non esiste”
fi
```

notiamo alcune cose:

- la funzione ritorna 0 per indicare true.
- $1 nella funzione indica il primo argomento della chiamata di funzione. In questo caso il valore passato è nome_file
- la verifca dell’esistenza di un file usa il costrutto [-e “nome file”]

Si noti, quindi che le **[ ]** sono un’abbreviazione del comando **test**. L’elenco degli argomenti da usare è il seguente:

$#	numero degli argomenti della linea di comando	$# = 0 è vera se non ci sono argomenti
-e	esistenza del file di qualunque tipo	-f (file che non sia un device o una directory) -b (file come block device: floppy,dischi...) -c (file come character device: keyboard...) -p pipe -s socket

-d	esistenza di una directory	
-h	esistenza di un collegamento simbolico	
-r	esistenza di un file leggibile	
-w	esistenza di un file scrivibile	
-x	esistenza di un file eseguibile	
-s	esistenza di un file di dimensione maggiore di zero	
-z stringa	vero se la stringa ha lunghezza zero	
-n stringa	vero se la stringa ha lunghezza maggiore di zero	
str1 = str2	vero se sono uguali	
str1 \< str2	vero se str1 lessicograficamente precedente	bisogna fare l'escaping usando \ prima di <
op1 –eq op2	vero se gli operandi sono uguali	
op1 –lt op2	vero se op1 è inferiore al secondo	
expr1 –a expr2	vero se entrambe le espressioni danno risultato vero	
expr1 –o expr2	vero se almeno un'espressione da risultato vero	
file1 –ot file2	vero se file1 è più vecchio di file2	

Gestione delle stringhe

Esaminiamo direttamente attraverso un esempio alcune funzioni sulle stringhe:

```
st=abcABC123ABCabc
#  0123456789...      l'indexing delle funzioni che non utilizzano expr o awk è zero-based

#lunghezza della stringa
lunghezza=${#st}      # lunghezza = 15
lunghezza=`expr length $st`

#lunghezza della sottostringa (espressione regolare) corrispondente all'inizio della stringa
lunghezza=`expr match "$st" 'abc[A-Z]*.2'`  #lunghezza=8 (effettua il match con abcABC12)

#posizione del primo carattere nella sottostringa che corrisponde ``
posizione=`expr index  "$st" 1c`     #posizione=3 poichè 'c' corrisponde prima del '1' nella stringa

#estrazione di una sottostringa
#in questo caso è zero-based!!!

st1=${st:0}    #st1=abcABC123ABCabc
st1=${st:7}    #st1=23ABCabc
st1=${st:7:3}  #st1=23A
st1=${st:(-4)} #st1=Cabc      parte da destra

#in quest'altro invece è 1-based!!!
st1=`expr substr $st 1 2`     #st1=ab

#rimozione di una sottostringa utilizzando un espressione regolare

st1=${st#A*c}  #st1=123ABCabc        rimuove la  corrispondenza più corta partendo dalla testa
st1=${st##A*c} #st1=abc              rimuove quella più lunga
st1=${st%b*c}  #st1=abcABC123ABCa    rimuove la  corrispondenza più corta partendo dalla coda
st1=${st%%b*c} #st1=a                rimuove quella più lunga

#rimpiazzo di una sottostringa all'interno della stringa con un altra sottostringa

st1=${st/abc/xyz}     #st1=xyzABC123ABCabc prima occorrenza a partire da sinistra
st1=${st//abc/xyz}    #st1=xyzABC123ABCxyz tutte le occorrenze a partire da sinistra
st1=${st/#abc/XYZ}    #st1= XYZABC123ABCabc prima occorrenza a partire da sinistra
st1=${st/%abc/ XYZ }  #st1=abcABC123ABCXYZ prima occorrenza a partire da destra
```

Variabili interne

Attraverso le vairabili **$EUID** E **$UID** E' possibile sapere chi ha lanciato lo script. L'$UID si riferisce all'utente vero e proprio mentre $EIUD si riferisce all'utente effettivo. Ad esempio mi loggo come utente poi passo a root utilizzando il comando su. In questo caso si avrà:

$UID = utente
$EUID = root

quindi per controllare che si tratta di root dall'inizio si ha:

```
if [ "$UID" –ne "0" ]
then
       echo "Non sei root"
       #esce con un codice di errore che indica di non essere lanciato da root
       exit 67
fi
```

La variabile **$IFS** contiene i caratteri usati come delimitatori di argomenti. Generalmente è posta uguale allo spazio, al tab o al newline. Volendo fare il parsing di una stringa i cui delimitatori siano un alro carattere si può sfruttare momentaneamente la shell impostando $IFS al carattere separatore. Ad esempio:

```
#!/bin/bash
output()
{
       for arg
       do echo "[$arg]"
       done
}

#adotto : come separatore
IFS=:

var=":a::b:c:::"
output arg
```

Produrrà il seguente output:

```
[]
[a]
[]
[b]
[c]
[]
[]
[]
```

E' possibile, inoltre avere informazioni sulla macchina su cui si sta eseguendo lo script. In particolare si hanno le seguenti variabili:

$HOSTNAME	nome dell'host
$OSTYPE	nome del sistema operativo
$MACHTYPE	hardware del computer (ad es. i686)

Ancora più interessante è conoscere le informazioni relative ai processi che interessano la storia dello script.

- **$PPID** è il Process ID del padre del nostro script.
- **$$** rappresenta il Process ID dello script stesso.
- **$!** è il Process ID dell'ultimo processo in background lanciato dallo script. All'interno di uno script, infatti, è possibile lanciare un processo in background come dalla shell utilizzando l'**&** ad esempio:

  ```
  ...
  sleep 100 && echo "mi sono svegliato" &
  ...
  ```

  vedremo come sfruttare questa variabile nel seguente paragrafo.

- Infine **$?** è l'exit status dell'ultimo comando eseguito o dello script stesso.

  ```
  echo ciao
  echo $?       #ritornerà 0

  lsfnrioej
  echo $?       #ritornerà un valore diverso da 0 poichè è un comando che non esiste
  ```

Gestione del tempo all'interno di uno script

Grazie alla variabile **$SECONDS** è possibile sapere da quanti secondi lo script è in esecuzione. Un'altro caso interessante è il timeout di una porzione dello script.

In pratica si può assegnare un tempo massimo di esecuzione (in secondi) di una porzione di uno script inviando, dopo un certo periodo di tempo, un segnale di interruzione che un'apposita routine intercetterà:

```
#!/bin/bash
TIMELIMIT=3

PrintAnswer ()
{
	if [ "$a" = TIMEOUT ]
	then
		echo $a
	else
		echo "hai risposto $a"
		#uccide l'ultimo processo lanciato in background (cioè il timer)
		kill $!
	fi
}

TimerOn()
{
	#mette in background il comando &
	#che aspetterà un certo tempo TIMELIMIT
	#e una volta sveglio manderà un segnale –s 14 al processo dello script stesso $$

	sleep $TIMELIMIT && kill –s 14 $$ &
}

#intercetta il segnale 14
Int14Vector()
{
	answer="TIMEOUT"
	PrintAnswer
	exit 14
}

#inizio del "programma" vero e proprio
echo "come ti chiami?"

#parte il cronometro
TimerOn

#effettuo l'input da console
read a

#se tutto va bene stampo la risposta e uccido il cronometro
PrintAnswer
```

Ricorsione

La possibilità di avere funzioni all'interno di uno script con variabili locali permette la possibilità di sviluppare semplici algoritmi ricorsivi:

```
#se non vengono passate nomi di directory prende la directory corrente
[  $# -eq 0 ] && ds=`pwd` || ds=$@

lista () {
	for el in $1/*
	do
		#se è una directory esplorala
		[ -d "$el" ] && lista $el
		#in ogni caso stampa il nome dell'elemento listato
		echo $e1
	done
}

for d in $ds
do
	if [ -d $d ]
	then
		lista $d
	else
		echo $d
	fi
```

```
done

exit 0
```

Rendere gli script più robusti

Uno script che si comporti come un comando di shell deve presentare alcune accortezze. Innanzitutto deve controllare la presenza di parametri nel giusto numero. A tale spopo si può incorporare il seguente frammento di codice

```
if [ $# -ne $number_of_expected_args ]
        echo “Usage: `basename $0` parameters”
        exit 1
fi
```

In questo caso $0 ritornerebbe il nome completo di percorso dello script. Utilizzando basename abbiamo solo il nome dello script.

Trattamento del testo

Definizione di file testo

Un file di testo è una sequenza di caratteri e simboli separati da un codice di interruzione di riga detto genericamente **newline**. Tale codice varia a seconda del sistema operativo:

- Unix 0x0A <*LF*>
- DOS 0x0D 0x0A <*CR*><*LF*>
- Mac 0x0D <*CR*>

dove <*LF*> sta per *line feed*, ovvero avanzamento carta, e <*CR*> sta per *carriage return*, ovvero ritorno di carrello.

Output di testo

Il programma più semplice è **cat** che riemette il testo così come è. Può essere usato come comando inziale di una pipeline o per effettuare l'append di un file ad un altro:

```
cat pippo.txt >> paperino.txt
```

aggiunge in coda a paperino.txt il file pippo.txt. Presenta, inoltre, alcune opzioni interessanti:

-b numera le righe emesse che non sono vuote
-s sostituisce le righe vuote multiple con una sola

Il comando **head** emette le prime righe di un file (**-n**) o primi bytes (**-b**) come specificato:

```
head –n 10 paperino.txt
```

Allo stesso modo lavora **tail** partendo dalla fine. Può essere particolarmente utile per tagliare un log file periodicamente:

```
tail –n 50 log.txt > log.tmp
mv log.tmp log.txt
```

Elaborazione del testo

Il comando **split** effetua la suddivisione in file più piccoli che avranno il prefisso secondo una sequenza ordinata del tipo aa, ab ac...

```
split –l 10 pippo.txt
```

suddivide il file pippo.txt in file di testo di, al massimo, 10 righe. Altrettanto si può fare per i byte utilizzando l'opzione **–b**.

Il comando **sort** può effettuare ordinamenti di file strutturati in record (righe) composte da campi. I campi sono delimitati da un separatore. Ad esempio dato un file così strutturato:

```
...
Paperino:Paperopoli:313
Topolino:Topolinea:113
Minnie:Topolinea:2345
...
```

in cui si singoli campi sono separati da un : (opzione **–t**) si può ottenere un file ordinato.txt (**-o**) usando il valore del secondo campo come chiave (**-k**) col seguente comando:

```
sort –t : -k 2,2 da_ordinare.txt –o ordinato.txt
```

Se si tratta di numeri si può utilizzare un ordinamento di tipo numerico (opzione **–n**) mentre con l'opzione **–u esclude le chiavi duplicate**. Infine, con l'opzione **–f non distingue tra maiuscole e minuscole**.

Un file ordinato può essere utile per effettuarvi ricerche di tipo binario all'interno. Il comando **look** ritorna la riga del file contenente la stringa indicata a partire dalla prima colonna:

```
look Paperino ordinato.txt
```

Il comando **cut** permette di estrarre porzioni di campi da un file in cui i singoli campi possono essere sia a lunghezza fissa che variabile utilizzando un delimitatore.

Se i record sono a lunghezza fissa, non viene cioè usato un separatore, si può utilizzare l'opzione **-b** per estrarre intervalli di byte:

```
cut –b 1-10,21 lunghezza_fissa.txt
```

emette righe contenenti i primi 10 byte e il 21-esimo byte del file originario. Se specifico un delimitatore invece:

```
cut –d ":" –f  1,2 ordinato.txt
```

emette il primo ed il secondo campo del file ordinato.txt (secondo l'esempio il nome del personaggio e la città).

Altrettanto interessante il comando **paste** che fonde due file unendoli riga per riga (separate da un carattere a scelta specificato nell'opzione **-d**). Ad esempio dati i due file ordinato.txt:

```
...
Paperino:Paperopoli:313
Topolino:Topolinea:113
Minnie:Topolinea:2345
...
```

e ordinato2.txt:

```
...
papero:M
topo:M
topo:F
...
```

l'effetto del comando:

```
paste –d : ordinato.txt ordinato2.txt > ordinato3.txt
```

sarà il seguente file ordinato3.txt:

```
...
Paperino:Paperopoli:313:papero:M
Topolino:Topolinea:113:topo:M
Minnie:Topolinea:2345:topo:F
...
```

Il comando **tr**, invece, permette di lavorare sui singoli byte di un file. Alcuni esempi chiariranno meglio:

```
tr abc def < primo.txt > secondo.txt
```

I caratteri a, b e c vengono rispettivamente sostituiti da d, e ed f nel file primo.txt che viene riemesso nel file secondo.txt.

```
tr '[:lower:]' '[:upper:]' <  primo.txt > secondo.txt
```

rende tutto in maiscolo.

```
tr -d '\r' < primo.txt > secondo.txt
```

toglie il carriage return dal file primo.txt producendo il file secondo.txt. Infine

```
tr -s 'n' < primo.txt > secondo.txt
```

toglie righe vuote multiple.

Sed

Sed è un editor di testo non interattivo che riceve l'output da file o da stdin è lo riversa sullo stdout (per salvarlo su file è necessario effettuare una redirezione >). Vediamo alcuni esempi:

```
sed “3p” “file.txt” > out.txt               #stampa la terza riga di file.txt

sed “/window/d” “file.txt” > out.txt        #cancella tutte le linee contenenti window

sed “s/uPortal/sdf/g” “file.txt” > out.txt  #sostituisce tutte le occorrenze di uPortal con sdf
```

quando nel pattern da ricercare o da sostituire è presente una slash (/) è sufficiente sostituirla con la sequenza di escape backslash slash (\/). Ad esempio il pattern **http://server1.com** diventa **http:**\/\/**server1.com**

PERL

Convenzioni fondamentali

Perl è un linguaggio di script parzialmente interprestato il cui engine (l'eseguibile perl) si trova solitamente in

```
/usr/bin
```

Può essere utile, tuttavia, aggiungere dei collegamenti simbolici nelle seguenti directory senza fidarsi troppo della variabile d'ambiente **PATH** usata per la ricerca dei percorsi degli eseguibili

```
/bin/perl
/usr/bin/perl
/usr/local/bin/perl
```

Ogni script inizia per convenzione con la riga:

```
#!/usr/bin/perl
...
```

I file di script, infine, hanno generalmente estensione .pl e vanno resi eseguibili mediante il comando:

```
chmod +x program_file.pl
```

Variabili e tipi di dati

Il concetto fondamentale del Perl è che gli elementi che si indicano hanno valore in funzione al contesto in cui si trovano.

Ad esempio un array può essere visto come:

- una lista di elementi
- il numero degli elementi contenuti
- una stringa contenente tutti i valori degli elementi contenuti

I tipi di dati più importanti sono:

- stringhe
- valori numerici
- riferimenti
- liste

Non esiste un tipo boolean. L'unica cosa che c'è è il risultato di una espressione. Dal punto di vista logico-booleano vengono considerati *false* i seguenti valori:

- indefinito – equivale ad una variabile non dichiarata
- "" – la stringa vuota
- 0 – il valore numerico zero
- "0" – la stringa corrispondente al valore numerico zero

Qualsiasi altro valore viene trattato equivalente a *true*.

Le variabili scalari sono costitutite, a differenza degli array, da un solo valore. Il loro nome e l'assegnazione di un valore segue la regola:

```
$variabile_scalare = valore
```

Si noti che al contrario della Bourne Shell **il dollaro si usa sempre** e non solo quando si deve leggere il valore della variabile.

Il valore può essere ad esempio una costante. Esistono costanti numeriche e costanti stringa. Queste ultime sono racchiuse da o:

Delimitatore	Comportamento dell'interprete
doppi apici "	1) le variabili vengono interpolate 2) \ viene intrepretato come prefisso di escape
apici '	1) le variabili **non** vengono interpolate 2) \ può essere usato solo nel caso \' e \\

Per evitare ambiguità nel caso di una variabile all'interno dei doppi apici **non seguita da spazi o punteggiatura** si deve delimitare il nome della variabile con le aprentesi graffe {}.

Per capire meglio consideriamo il seguente script:

```
#!/usr/perl

$primo =”Ciao”;
$secondo = “Mondo”;

print “$primo $secondo! \n”;
print ”$primo ${secondo}cane \n”;

Ciao Mondo!
Ciao Mondocane!
```

Array e liste

Le liste sono sequenze di elementi scalari di valore costante, di qualunque tipo, separati da una virgola:

```
( “uno”, “due”, “tre” )
```

Una lista può inizializzare un array mentre se viene assegnata ad una variabile scalare tale variabile assumerà il valore dell'ultimo elemento:

```
$miavar = ( “uno”, “due”, “tre” );  -> $miavar == “tre”
```

Le liste hanno un inice attraverso il quale si accede ai loro elementi:

```
$ind = 1;
$miavar = ( “uno”, “due”, “tre” )[$ind]; -> $miavar == due
```

Gli elementi non scalari all'interno di una lista vengono interpolati incorporando tutti gli elementi che contengono in quel punto. Gli elementi nulli vengono ignorati.

Gli array sono indicati nella loro totalità con la @

```
@nome = (“uno”, “due”, tre”);
```

Il singolo elemento dell'array è indicato col $:

```
$nome[indice]
```

Vediamo degli esempi:

```
# la variabile $elementi riceve il numero di elementi dell’array @nome
$elementi = @nome;

#l’intero array diventa una stringa
$elementi = “@nome”;
```

Infine si noti che

- $[rappresenta il primo elemento dell'array
- $# rappresenta l'ultimo elemento dell'array
- $[indice_iniziale..indice_finale] rappresenta un sottoinsieme, estremi compresi, dell'array.

Un importante array di sistema predefinito è **@argv** che **rappresenta gli argomenti della linea di comando**. Ricordiamo, infine, che il nome del programma è ottenibile mediante la variabile predefinita **$0**.

Array associativi

L'array associativo è un insieme di coppie chiave-valore i cui elementi valore sono ottenibili mediante la chiave. Può essere dichiarato mediante la seguente:

```
%array_associativo = (chiave, valore, chiave, valore,…)
%array_associativo = (chiave => valore, chiave => valore, …)
```

Il singolo elemento sarà rinvenuto come:

```
$(array_associativo){chiave}
```

Un sotto insieme di un array associativo è un array:

```
@array_associativo{chiave1, chiave2,…chiave5}
```

Un array associativo predefinito particolarmente importante è **%ENV** per la lettura delle variabili d'ambiente. Ad esempio:

```
print "PATH: $ENV{PATH}\n";

PATH: /usr/local/bin:/bin:/usr/X11R6/bin
```

Operatori ed espressioni

Gli operatori matematici sono i consueti tranne la presenza dell'esponente indicato con ** e l'operatore modulo (resto di una divisione intera) indicato con %.

Per quanto riguarda le stringhe si usa il punto . per concatenare le stringhe. Quindi se abbiamo

```
$var1 = "canar";
$var2 = "it";
Print "$var1.$var2";
```

Il risulato sarà:

```
canarit
```

Per confrontare l'uguaglianza tra due stringhe si usa l'operatore **eq** per == e **ne** per !=.

Strutture di controllo

La struttura più particolare è:

```
Condizione ? espressione1 : espressione2
```

In cui se la condizione si avvera allora viene valutata l'espressione 1 altrimenti si valuta la seconda.

Ci sono poi le classiche in cui bisogna usare le parentesi {}terminate da ;

```
if( condizione ) {
 …
} elseif ( condizione ) {
…} else {
…
};
```

L'istruzione **unless** viene interpetata come una IFNOT. Per capire l'istruzione **while** e l'istruzione **last** per uscire da tale ciclo vediamo il seguente script per il calcolo del fattoriale:

```
#!/usr/bin/perl

$numero = $ARGV[0];
$cont = $numero –1;
#ciclo senza fine
while (1) {
	$numero *= $cont;
	$cont--;
	if( !$cont ) {
		last;
	};
};
print “Il fattoriale è $numero. \n”;
```

Ancora meglio si poteva fare utilizzando la condizione di terminazione alla fine all’interno di un costrutto **do while**.

```
#!/usr/bin/perl

$numero = $ARGV[0];
$fattoriale = 1;
#ciclo senza fine
do {
	$fattoriale *= $cont;
	$cont--;
} while($cont);
print “Il fattoriale è $numero. \n”;
```

Utilizzando un ciclo **for** invece:

```
#!/usr/bin/perl

$numero = $ARGV[0];
for( $cont = 1; $cont < $ARGV[0]; $cont++) {
	$numero *= $cont;
};
print “Il fattoriale è $numero. \n”;
```

Infine esiste la possibilità di scorrere una lista. In pratica la varibile scalare dichiarata come primo argomento del **foreach** assume i valori della lista. **Quindi bisogna ricordarsi delle ( )**! Nel caso del fattoriale costruendo al volo una lista si ha:

```
#!/usr/bin/perl

$numero = $ARGV[0];
foreach  $cont  (1 .. ($ARGV[0]-1) ) {
	$numero *= $cont;
};
print “Il fattoriale è $numero. \n”;
```

Se vogliamo stampare gli elementi di un array scorrendoli uno ad uno per aggiungere un \n ad esempio si può fare cosi:

```
#!/usr/bin/perl

@a = (“uno”,”due”,”tre”);
foreach  $elemento  (@a) {
	print “$elemento\n”;
};
```

Input/Output

Le funzioni del Perl sono **operatori unari** che intervengono sull’argomento posto immediatamente alla loro destra. In questo modo si ha:

```
Print (1+2)+4;
```

```
3
```

Fatta questa premessa vediamo **l’esecuzione di un comando di sistema** e la sua successiva elaborazione

```
#!/usr/bin/perl

#riceve l'elenco dei file in un'unica stringa
$elenco = `ls *.pl`;

if( $? == 0) {
        print "$elenco\n";
} else {
        print "non ci sono programmi Perl\n";
};
```

Si noti che **$?** è il risultato del comando. Se è terminato correttamente generalmente è 0. Mentre la valutazione di un espressione al fine di lanciare un comando viene eseguita mettendo il comando tra due backtick (apostrofi inversi) ` comando `.

Si può utilizzare, inoltre, l'operatore stringa valutazione comando **qx**|`` che ha la forma:

```
qx delimitatore_sinistro comando delimitatore_destro
```

Ad esempio potevamo scrivere:

```
…
$elenco =  qx(ls *.pl);
…
```

La gestione di un file passa attraverso la gestione del corrispondente filehandle, in sostanza un numero, indicato per convenzione in maiuscolo.

Oltre ai filehandle predefiniti come <STDIN>, <STDOUT>,<STDERR> è possibile ottenere associare un filehandle ad un file chiamando la funzione open():

```
open(HANDLE_NAME, "espressione");
```

per espressione non si intende solo il nome del file. Innazitutto prima del nome del file può essere specificata la modalità di apertura:

<	input
>	output (riscrittura)
>>	append
+	in aggiunta ai precedenti indica modalità lettura/scrittura

Altri casi di espressioni più interessanti coinvolgono l'uso del simbolo di pipe **|**

```
open(ARTICLE, "caesar <$article |");
```

associa l'handle ARTICLE al risultato decrittato del file il cui nome è contenuto dalla variabile $article.

```
open(EXTRACT, "|sort >/tmp/Tmp$$");
```

in cui $$ è la variabile predefinita relativa al proprio process id (PID). L'output del comando viene legato all'handle EXTRACT.

La stampa a video delle righe di un file di testo è semplice poiché ogni riga viene acquisita con il simbolo di \n compreso al momento in cui viene valutata (sia come assegnamento):

```
$riga = <HANDLE>;
```

Sia all'interno di una espressione condizionale come il while( ).

Comunque sia, viene riversata nella variabile predefinita **$_** . Una volta raggiunto il fine file ritorna un valore indefinito quindi valutabile come una condizione false.

```
#!/usr/bin/perl

while(<STDIN>) {
        print $_;
};
```

Per scrivere è sufficiente:

```
Print FILE_HANDLE “espressione”;
```

Se il filehandle è contenuto nell’elemento di un array è necessario inserirlo in un blocco

```
Print {$elenco_file[$i]} “espressione”;
```

Infine è necessario chiudere il file mediante close:

```
Close FILE_HANDLE;
```

E’ possibile, inoltre, effettuare un locking dei file. Per praticità facciamo riferimento al seguente esempio:

```
Use Fcntl ‘:flock’;   #importa le costanti di lock per il sistema
…
open (ELENCO, “>> /home/canar”);
flock (ELENCO; LOCK_EX);      #lock del file
…
#elaborazione del file
…
flock(ELENCO, LOCK_UN);       #unlock del file
```

Si può anche creare una lista a partire da un filehandle in cui ogni elemento è una riga:

```
#!/usr/bin/perl

@miofile = <STDIN>;
print @miofile;
```

Quando il contenuto all’interno delle parentesi angolari non viene riconosciuto Perl interpreta ciò come un modello per indicare nomi di file (**globbing**):

```
#!/usr/bin/perl

@elenco = <*.pl>;
print “@mioelenco\n”;
```

Si può anche usare

```
@elenco = glob( “*.pl”);
```

Gestione delle stringhe

In Perl la gestione delle stringhe è particolarmente raffinata in quanto sono definiti dei delimitatori che consentono di intervernire sulla stringa in maniera ogni volta diversa.

Si possono usare dei delimitatori tradizionali o una forma che consente di usare un tipo di delimitatore a piacere.

q|’’	‘Stringa’ q|Stringa che “contiene” ‘vari apici’.| q{$miavar non è interpolata}	Stringa Stringa che “contiene” ‘vari apici’. $miavar non è interpolata
qq|””	“Stringa che $interpola” qq|Stringa che “contiene” ‘vari apici’.| qq|Stringa che $interpola|	Stringa che *valore* Stringa che “contiene” ‘vari apici’. Stringa che *valore*
qx|``	qx|ls $opzioni|	ls valore
qw	@lista =qw/ciao come stai/	@lista = (ciao,come,stai)

Confronto e sostituzione

Per effettuare il confronto su una stringa si può usare il modello di confronto. Nella stringa di confronto viene effettuata anche l'interpolazione.

Viene indicato con **m|//** ed ha due forme possibili

/stringa/modificatori

m|stringa|modificatori #dove | è il delimitatore

I modificatori permettono di effettuare ulteriori azioni durante il confronto:

i	ingnora la differenza tra maiuscole e minuscole
x	ignora eventuali spazi e commenti
g	trova tutte le occorrenze
o	interpreta il modello solo la prima volta

L'operazione di confronto con un modello utilizza l'operatore =~ come si può vedere negli esempi:

```
#!/usr/bin/perl

$frase = 'Ciao, come stai?';
if ( $frase =~ /ciao/i ) {
        print "Trovato il ciao";
};
```

Troverà il ciao in virtù del modificatore **i**.

```
#!/usr/bin/perl

$lista =  `ls`;
if ( $elenco =~ /.*\.pl/ ) {
        print "Trovato un file .pl\n";
};
```

In questo caso il comando ls viene interpretato. Si ha un listing dei file nella directory corrente.

L'operatore cerca un file una stringa del tipo qualsiasicosa.pl esemplificata da *\.pl. Si utilizza il punto all'inizio perché …

Per trovare il punto pl si usa **\.** Infatti il punto ha un significato speciale nelle espressioni regolari in Perl quindi bisogna usare la \ per poterlo confrontare normalmente.

Oltre ad effettuare un semplice confronto è possibile effettuare la sostituzione di ciò che si è trovato mediante il modello di sostituzione **s**.

La forma è la seguente:

s|modello|rimpiazzo|modificatori

Alcuni esempi cercherannodi chiarire:

```
$path =~ s|/usr/bin|/usr/local/bin|
```

sostituisce in $path /usr/bin con /usr/local/bin. Analogamente usando delimitatori destri e sinistri:

```
$path =~ s{/usr/bin}{/usr/local/bin}
```

Un'altra possibilità interessante è la definizione di un modello di **traslazione caratteri tr| y** che sostituisce una serie di caratteri in un'altra. Ad esempio:

```
$miavar =~ tr/A-Z/a-z/;
```

converte in minuscolo il contenuto in maiuscolo della stringa (a parte le vocali accentate).

```
$contatore = ( $miavar =~ tr/0-9// );
```

conta i caratteri numerici contenuti in $miavar.

Espressioni regolari

Un'espressione regolare è rinchiusa tra / , sebbene questa sia più una convenzione che una regola! E' comune vedere un'espressione regolare rinchiusa tra | o # o qualche altro carattere, per evitare di mettere il simbolo \ davanti a tutti i / in regexp.

Un'espressione regolare definisce un pattern che verrà cercato. Nelle espressioni regolari, i caratteri alfanumerici (dalla **a** alla **z**, dalla **A** alla **Z**, da **0** a **9** e **_**) hanno corrispondenza univoca, mentre altri caratteri hanno caratteristiche speciali.

Un \ seguito da un carattere non alfanumerico corrisponde esattamente a quel carattere; in particolare, \\ corrisponde a \ .

^ (solo all'inizio dell'espressione) fa in modo che il testo corrisponda al pattern **solo** se il pattern è all'**inizio del testo**.

$ (solo alla fine dell'espressione) fa in modo che il testo corrisponda al pattern **solo** se il pattern è alla **fine del testo**.

Qualche esempio chiarirà un po' le cose.

```
"bla" =~ /\//
```

falso, poichè cerca in "bla" il carattere / (abbiamo una backslash "\"che introduce una slash "/"

```
"blh" =~ /\$/
```

falso, poichè cerca in "bla" il carattere $

Quindi per cercare un carattere speciale (o, se volete, riservato) bisogna chiarire tramite il simbolo \ che si ricerca proprio quel carattere, e non il suo valore. E la stessa cosa per la ricerca con grep ma anche con editor (vi, ad esempio)

```
"bla" =~ /la/                vero
"bla" =~ /^la/               falso (infatti il pattern la non si trova essattamente all'inizio)
"bla" =~ /^bl/               vero (bl si trova esattamente all'inizio)
"blh" =~ /h$/                vero
"bla" =~ /ls$/               falso
"bla" =~ /^bla$/             vero
```

Altri caratteri speciali sono:

. cerca ogni singolo carattere

\t cerca una tabulazione

\s cerca uno spazio o una tabulazione

\S cerca ogni carattere che non sia una tabulazione

\n cerca una "newline"

\w cerca ogni singola lettera, cifra ma anche _

\W cerca ogni cosa che non sia una lettera, una cifra o _

\d cerca ogni singola cifra da 0 a 9

\D cerca ogni carattere che non sia una cifra

```
"bla" =~ /b.a/               vero
```

```
"bla" =~ /b.la/              falso
"bla" =~ /b\w.h$/            vero
"bla" =~ /\w\D/              vero
"bla" =~ /\d/                falso (non ci sono cifre)
```

[caratteri] -- cerca ogni carattere che sia tra [ ]

Inoltre, un campo di ricerca può essere specificato con **-** , ad esempio [a-m] cerca le lettere tra la a e la m. Se il primo carattere tra [ ] è ^ , il significato è **negato**, e verrà cercato tutto ciò che **NON è quello che è compreso tra [ ]**

[-.0-9] cerca esattamente un **-** un **.** o una **cifra**

[^\@ \t] cerca qualsiasi carattere che non sia un @, una tabulazione o uno spazio. Da notare che il \ davanti a @ è opzionale, poichè il simbolo @ non ha particolari significati in questo contesto; ma funziona anche con \@

Per determinare il numero **minimo** e **massimo** di volte che un determinato elemento **deve ricorrere consecutivamente** utilizziamo i **quantificatori**:

( ) raggruppa più elementi in un pattern da cercare una volta

{min,max**}** dopo un raggruppamento, dopo un carattere o metacarettere indica quante volte deve essere ripetuto il patter in questione.

|(\d{2,4})| è soddisfatta da 70 e 1970

***** Corrisponde all'intervallo {0, } ovvero da un minimo di 0 volte ad un massimo indefinito

+ Corrisponde all'intervallo {1, }

? Corrisponde all'intervallo {0, 1}

s Opera una sostituzione

tr Opera una traduzione nella forma 'tr [a-z] [A-Z]' (ovvero rende maiuscoli i caratteri minuscoli e viceversa)

```
"bla" =~ /c*k*z?b+.l/
```
Vero, visto che la **c**, la **k** e la **z** non ci sono, la **b** appare una volta e la **l** anche

```
"ccckkzbbb8lZOINX" =~ /c*k*z?b+.l/
"blahbla" =~ /ah(EEK)?bl/
"blahEEKbla" =~ /ah(EEK)?bl/
"blahEEKEEKbla" =~ /ah(EEK)?bl/
"blahEEKEEKbla" =~ /ah(EEK)+bl/
```
Vero
Vero
Vero
Falso
Vero

```
/^([^\@ \t]+\@[^\@ \t])+\s+([-.0-9]+)$/
```

L'ultimo esempio corrisponde ad una riga che:

1) inizia (simbolino ^) con un numero diverso da 0 di caratteri che non sono nè @ nè spazi nè tabulazioni **([^\@ \t]+**
2) poi una @, **\@**
3) poi altri caratteri che non sono @ nè spazi nè tabulazioni **[^\@ \t])**
4) ,poi alcuni spazi e tabulazioni **\s+**
5) poi qualsiasi mistura di '**-**', '.' e cifra **([-.0-9]+)** alla fine (simbolino **$**)

In parole povere, qualcosa di simile ad un indirizzo e-mail seguito da spazi e qualcosa che si può ragionevolmente pensare come un numero!

```
/^\s*(\d+)\.(\d+)\.(\d+)\.(\d+)\s*$/
```

Questo invece potrebbe corrispondere ad un indirizzo IP.

Inoltre, se il pattern ha dei **sub-pattern** (), questi sono assegnati a variabili numerate, **$1** per la prima, **$2** per la seconda ecc. Ad esempio l'istruzione Perl seguente:

```
"129.199.129.13" =~ /^\s*(\d+)\.(\d+)\.(\d+)\.(\d+)\s*$/;
```

implica l'assegnamento delle seguenti variabili

$1 è 129,
$2 è 199,
$3 è 129,
$4 è 13

Un altro esempio intressante è l'estrazione della data da una stringa se questa ha il formato data:

```
if( $miadata =~ m|Data:\s+(\d\d)/(\d\d)/(\d{2,4})| ) {
        $giorno = $1;
        $mese = $2;
        $anno = $3;
}
```

Inoltre è possibile far riferimento ad una corrispondenza parziale contenuta in un raggruppamento. Si utilizza il metacarattere **\n** dove **n** corrisponde ad un **numero** ed indica **n-esimo** raggruppamento. Ad esempio:

```
(0|0x0)\d*\s\1\d*
```

In cui \1 corrisponde al primo raggruppamento cioè

```
(0|0x0)
```

quindi un espressione del genere è soddisfatta dalla sequenza

0x0123 0x0456

ma non da

0x0123 0456

poiché si fa riferimento alla corrispondenza che è stata trovata all'interno della stringa stessa non al modello. Per capire è come se l'espressione

```
(0|0x0)\d*\s\1\d*
```

Sia stata trasformata run-time in

```
(0|0x0)\d*\s(0x0)\d*
```

In quanto si è trovata all'inizio delle due stringhe (quella soddisfatta e quella no) la corrispondenza 0x0.
Invece se avessimo scritto l'espressione regolare:

```
(0|0x0)\d*\s(0|0x0)\d*
```

Entrambe le stringhe avrebbero passato il test.

Notiamo, infine, l'utilizzo del simbolo "|" all'interno di un raggruppamento. Tale simbolo implica una pluralità di scelte, che si escludono a vicenda, che rendono vera l'espressione. In questo caso si ha un matching sia se c'è lo 0 si se c'è 0x0.

Come abbiamo già visto un pattern con sostituzione è scritto nella forma:

$variabile =~ s/pattern/replacement/;

Il pattern è, come prima, un'espressione regolare, ed il replacement è una normale stringa, tranne per il fatto che le variabili sono interpolate al suo interno.

Nella maggior parte dei casi, si può aggiungere il carattere 'g' dopo il replacement, in modo che le parole che corrispondono al pattern siano tutte cambiate, non solo la prima.

```
$waitops{$c} =~ s/:${oldnick}:/:${newnick}:/g
```

Le espressioni regolari lavorano con la funzione **split**, che prende come argomento un'espressione regolare , una variabile scalare e, volendo, un secondo scalare che specifica il numero massimo di campi in cui splittare il pattern.

```
@array = split(/pattern/, expression);
```

Nel caso più tipico, si può voler splittare una linea in parole, come segue:

```
@words = split(/\s+/, $some_line);
```

se $some_line era "a b c", ora @words è ("a", "b", "c")

oppure

```
($val1, $val2) = split(/\s+/, $whatever);
```

setta $val1 con il primo valore di $whatever e $val2 con il secondo. Eventuali altri valori (terzo, quarto ecc.) vengono scartati

Networking

Interfacce di rete

Un interfaccia di rete può essere un collegameno punto-punto, un loopback virtuale come 127.0.0.1 o una scheda Ethernet. L'interfaccia di loopback può essere attivata nel seguente modo:

```
ifconfig lo 127.0.0.1
```

L'interfaccia di una scheda ethernet invece è livemente più complessa:

```
ifconfig eth0 192.168.1.1 netmask 255.255.255.0
```

inoltre una scheda di rete può avere più indirizzi ip. A tale scopo è necessario considerare delle schede virtuali che vengono indicate secondo la convenzione seguente:

interfaccia_reale:n_iterfaccia_virtuale

il numero di interfaccia virtuale inizia da 0

```
ifconfig eth0:0 192.168.1.2 netmask 255.255.255.0
```

avremo quindi una eth0 ed una eth0:0 con indirizzo IP diverso ma tutte e due riferite alla medesima interfaccia fisica. Il comando ifconfig presenta i seguenti parametri:

up | down	abilita | disabilità l'interfaccia esplicitamente
mtu	Stabilisce l'mtu per l'inerfaccia per una fast Ethernet può andare da 1000 a 2000
pointopoint [indirizzo]	Il traffico dell'interfaccia si dirige solo all'indirizzo specificato
allmulti	Abilità la modalità promiscua di rete per monitorare tutti i pacchetti che passano

Instradamento

L'instrandamento del traffico di rete è gestito dall'elaboratore mediante un'apposita tabella di routing. Il comando route permette di aggiungere percorsi verso reti o host a tale tabella. Per semplicità vediamo alcuni esempi concreti:

```
route add –net 127.0.0.0 netmask 255.0.0.0 dev lo
```

una ipotetica rete 127.0.0.0 viene raggiunta passando per l'interfaccia di loopback

```
route add –net 192.168.1.0 netmask 255.255.255.0 dev eth0
```

la rete 192.168.1.0 viene raggiunta mediante l'interfaccia eth0

```
route add –host 192.168.7.1 dev eth0:0
```

l'host 192.168.7.1 viene raggiunto mediante eth0:0

```
route add –net 192.168.2.0 netmask 255.255.255.0 gw 192.168.1.2
```

la rete 192.168.2.0 viene raggiunta mediante 192.168.1.2 per il quale è stato già stato definito un instradamento

```
route add –net default gw 192.168.7.1 dev eth0:0
```

qualsiasi rete, all'infuori di quelle precedentemente specificate va raggiunta attraverso 192.168.7.1

Risoluzione dei nomi

La risoluzione dei nomi in indirizzi IP avviene mediante il file statico hosts ed un server dns. L'ordine di ricerca è dato da **/etc/host.conf** :

```
order hosts,bind
multi on
```

La seconda riga indica che ad un singolo nome di dominio possono corrispondere più IP.

Il file **/etc/hosts** elenca gli host nel seguente modo:

```
#necessario per il loopback IPv4
127.0.0.1      localhost.localdomain        localhost
#elenco degli host
1.2.3.4        p1.prova.com                 p1
```

La risoluzione dei nomi avviene generalmente mediante DNS. I sever DNS vengono elencati in **/etc/resolv.conf**:

```
nameserver 192.168.1.1
nameserver 192.168.2.15
```

Il server DNS che andiamo a considerare è quello relativo al pacchetto **BIND** (in cui il demone che effettua il servizio si chiama **named**).

Per prima cosa si indicano nel file **/etc/named.conf** o **/etc/bind/named.conf** di quali zone (domini o sottodomini) si occupa compresi gli indirizzi di altri server DNS per risolvere richieste che non riesce a gestire da solo:

```
options {
        //directory in cui sono contenuti i file di zona
        directory "/etc/bind/zone";
        //eventuale server DNS per risolvere i nomi sconosciuti
        forwarders {
                111.112.123.114;
        };

//cache per la risoluzione dei nomi
//il file named.root contiene gli indirizzi dei root server di Internet
zone "." {
        type hint;
        file "named.root";};

//risoluzione inversa degli indirizzi locali
zone "0.0.127.in-addr.arpa" {
        type master;
        file "127.0.0";
        };

//risoluzione inversa degli indirizzi della rete locale
zone "1.168.192.in-addr.arpa" {
        type master;
        file "192.168.1";
        };

//risoluzione dei nomi a dominio prova.com
zone "prova.com" {
        type master;
        file "prova.com"}
```

Per semplicità iniziamo proprio dall'ultimo cioè prova.com che definisce tutti i nomi del dominio prova.com:

```
;dominio classe    start of autority server primario       contatto
@       IN              SOA             server1.prova.com.    root.server1.prova.com.(
                                         2004080300
                                        28800
                                        7200
                                        604800
                                        86400 )

;indirizzo dell'host server1
server1.prova.com.     A       192.168.1.1

;alias per www
www.prova.com.         CNAME   server1.prova.com.

;name server
@       IN      NS      server1.prova.com.

;mail server
@       in      mx      10      server1.prova.com.
```

Facciamo notare alcune cose:

- @ indica il dominio per il quale tale file è stato interrogato quindi è come se scrivessimo prova.com
- i nomi delle macchine completi di dominio (i FQDN, fully qualified domain name, per intedenderci) devono essere sempre terminati da un punto "." Altrimenti viene aggiunto il nome di dominio.
- è possibile delegare sottodomini indicando il nome completo del sottodominio seguito da NS e l'indirizzo IP dell'host o il nome se risolvibile. Ad esempio:

```
mc.prova.com.  IN     NS     123.124.125.126
```

Il file di risoluzione inversa, 192.168.1, per la classe da noi gestita è

```
1.1.168.192.in-addr.arpa.           PTR    server1.prova.com.
```

Xinetd

Xinetd è un demone che funge da wrapper di servizi. In questo modo è possibile decidere chi e come può accedervi permettendo anche un'ottimizzazione di risorse in termini di socket aperti.

I file che attuano il controllo di accesso sono **/etc/host.allow** e **/etc/host.deny**. Attenzione se i due file non sono affatto configurati tutti i servizi sono disponibili a tutti.

Ad esempio, per limitare l'accesso al servizio telnet solo da parte della macchina 192.168.1.2 si configurano /etc/host.allow nel seguente modo:

```
in.telnetd : 192.168.1.2
```

e per sicurezza /etc/host.deny :

```
ALL : ALL
```

Vediamo un esempio di file di configurazione **/etc/xinetd.conf** per un servizio:

```
…
service telnet {
        socket_type = stream
        protocol = tcp
        wait = no
        user = root
        server = /usr/sbin/in.telnetd
}
…
```

OpenSSL

La tecnologia OpenSSL permette la creazione di certificati digitali e chiavi pubbliche e private da utilizzare per sistemi di comunicazione sicura. Un certificato digitale è composto dalle seguenti parti:

- la versione dello standard X.509
- il numero di serie rilasciato dall'autorità di certificazione (*Certification Authority* CA)
- il nome distintivo (*Distinguisched Name* DN) dell'autorità di certificazione
- Il periodo di validità del certificato
- il nome distintivo (*Distinguisched Name* DN) del soggetto formato nel seguente modo
    - Country (C): nazione. Es. C=IT
    - State (ST): stato o provincia. Es. ST=MC
    - Locality (L): località. Es. L=Macerata
    - Organization (O): organizzazione. Es. O=UNIMC
    - Organizational Unit (OU): dipartimento all'interno dell'organizzazione. Es. OU=SdF
    - Common Name (CN): nome. Es. Carlo Alberto Bentivoglio
- la chiave pubblica del del titolare del certificato

Generalmente il certificato è codificato in formato PEM che utilizza solo l'ASCII a 7 bit. L'iter consiste nel generare una chiave privata a partire da un file di byte casuali:

```
dd if=/dev/random of=casuale bs=1b count=1k
openssl genrsa –rand casuale –out pk.pem 1024
```

la chiave può essere crittografata mediante il seguente comando:

```
openssl rsa –des3 –in pk.pem –out protected_pk.pem 1024
```

Tuttavia questo ultimo passo non è utile in quanto la chiave potrebbe essere usata da un servizio che ogni volta dovrebbe chiederci la passphrase per poterla usare.

A questo punto o si produce una richiesta di certificazione o si produce un certificato fittizio. Nel secondo caso:

```
openssl req –new –x509 –key pk.pem –out certificato.pem
```

Nel caso di Apache bisogna allegare in chiaro la chiave privata oltre al certificato. Per far prima si possono saltare tutti i passaggi e produrre tutto insieme:

```
openssl req –new –x509 –nodes –out cerificato.pem –keyout certificato.pem
```

A questo punto nella directory in cui si posizionerà il certificato è necessario mettere un link simbolico al certificato stesso avente per nome il valore del *digest* del certificato seguito dall'estensione **.0**:

```
ln –s certificato.pem `openssl x509 –hash –noout –in certificato.pem`.0
```

OpenSSH

Secure Shell è una tecnologia che consente di effettuare comunicazioni cifrate tra un client ed un server al fine di lavorare a distanza sul server (come un telnet).

Per prima cosa bisogna creare le chiavi e il certificato che il server utilizzerà al momento della connessione del client. Tali chiavi saranno 3 coppie in quanto SSH può funzionare utilizzando 2 tipi di protocollo in cui, il più sicuro, può usare due tipi di algoritmi crittografici diversi:

```
ssh-keygen –t rsa1 –f  /etc/ssh/ssh_host_key  -N ‘’
ssh-keygen –t rsa –f  /etc/ssh/ssh_host_rsa_key  -N ‘’
ssh-keygen –t dsa –f  /etc/ssh/ssh_host_dsa_key  -N ‘’
```

l'opzione **–N** serve a specificare òa parola d'ordine da usare per accedere alla chiave privata. Ora, poichè tale chiave deve essere usata da un servizio, non ha senso cifrarla. Inoltre ogni comando crea due chiavi: una privata ed una pubblicata il cui file avrà estensione **.pub**.

Il servizio SSH viene offerto tramite il demone **sshd** avviato durante l'inizializzazione del sistema in modo indipendente da xinetd per motivi di performance. Il file di configurazione

```
/etc/ssh/sshd_config
```

specifica la porto di ascolto, la collocazione delle chiavi, le modalità di autenticazione ecc. Al fine di verificare il corretto funzionamento del sistema può essere utile avviare sshd in modalità diagnostica:

```
sshd –e –d 2>&1 | less
```

Lato client si usa il programma ssh il cui file di configurazione globale è:

```
/etc/ssh/ssh_config
```

L'utilizzo di ssh è semplice ad esempio:

```
ssh –l tizio www.canar.org
```

collega l'utente tizio al sistema www.canar.org.

Per copiare file tra elaboratori si può utilizzare il comando scp che utilizza ssh in modo trasparente:

```
scp root@canar.org:/etc/profile .
```

copia il file etc/profile dall'elaboratore canar.org (impersonando l'utente root) nella direcrtory corrente (si noti l'uso del punto).

Esiste, infine, la possibilità di stabilire un collegamento FTP utilizzando sftp che è un client ftp simile a quello tradizionale. Per funzionare è necessario che nel file di configurazione del sistema remoto /etc/ssh/sshd_config ci sia la seguente riga:

```
Subsystem      sftp    /usr/lib/sftp-server
```

Ad esempio:

```
sftp root@canar.org:/usr
```

ti collega come root al nodo canar.org specificando la directory di partenza /usr. A questo punto si possono utilizzare i seguenti comandi:

cd, ls, rename, mkdir, rm, ,rmdir, pwd	operazioni sul sistema remoto
get, put	trasferimento file
l**cd**,l**ls**,l**mkdir**,l**pwd**,	operazioni sul sistema locale
! [comando [argomenti]]	esegue localmente il comando specificato

Protocolli della famiglia TCP/IP e filtraggio dei pacchetti

Una connessione client/server avviene a partire da un indirizzo IP ed una porta non privilegiata (da 1024 a 65535) verso un l'IP del server in ascolto sulla porta relativa ad un servizio (l'HTTP ascolta sulla porta 80).

L'istaurarsi di una connessione TCP avviene attraverso fasi differenti in cui vengono usati degli indicatori speciali all'interno dei pacchetti per attribuire loro un significato speciale.

In particolare, quando un pacchetto contiene il bit **SYN** attivo, si tratta di un tentativo di iniziare una nuova connessione. In questo modo è possibile capire chi sta iniziando la comunicazione. In un certo senso è possibile capire il verso che tale comunicazione presenta.

Oltre ai protocolli TCP e UDP è molto importante saper filtrare il protocollo ICMP. Tale protocollo serve per inviare messaggi che riguardano il funzionamento della rete ma viene spesso utilizzato per scopi fraudolenti dai malintenzionati.

Bisongna fare attenzione a non bloccare i messaggi di tipo 3 (destination-unreachable) utile per il traffico TCP e UDP. Può essere utile conoscere le principali tipologie di messaggi per abilitare tali funzionalità in punti precisi dei nostri filtri:

Tipo	Nome	Utilizzatore
0	echo-reply	Messaggio di replica ad un ping
3	destination-unreachable	Traffico TCP e UDP
5	Redirect	Instradamento dei pacchetti
8	echo-request	Messaggio di richiesta del programma ping
11	Time-exceeded	Programma traceroute

Nello schema che utilizzeremo si può intervenire in tre punti diversi per intercettare i pacchetti:

- INPUT: pacchetti in arrivo attraverso una data interfaccia
- OUPUT: pacchetti in uscita attraverso una data interfaccia
- FORWARD: pacchetti in transito da un interfaccia ad un'altra

Ad esempio ipotizzando una macchina che faccia da router tra due reti una interna ed una esterna:

- intercettando i pacchetti ICMP di tipo 8 in uscita impediamo il ping verso l'esterno (anche dal router stesso)
- intercettando i pacchetti ICMP di tipo 8 in ingresso impediamo il ping verso il router e la rete sull'altra scheda di rete
- intercettando i pacchetti ICMP di tipo 8 in transito impediamo il ping tra macchine connesse alle due reti ma non da e verso il router

l'effetto dell'intercettazione può esprimersi mediante le seguenti azioni:

- ACCEPT: vengono lasciati passare
- DROP: vengono bloccati
- REJECT: vengono bloccati ma viene mandato un messaggio ICMP all'origine per segnalare un rifuto

A questo punto è necessario vedere sommariamente la sintassi del commando iptables che utilizzeremo per impostare i filtri sulle schede di rete. La sintassi base è:

iptables [-t tabella] opzione_di_comando punto_di_controllo [regola] [-j obiettivo]

Vediamo uno per uno i parametri:

- **-t**: indica il contesto in cui si applicano le regole di iptables. Il contesto di default è proprio **filter** che è quello relativo al firewall. Mentre **nat** permette di effettuare il network address translation.

- **opzione_di_comando**: specifica che azione svolgerà il comando. Ad esempio –F cancellerà tutte le regole nel punto di controllo, -A aggiungerà una regola in quel punto.

- **punto di controllo**: specifica il punto sull'interfaccia in cui la regola viene applicata. Si noti che:

  INPUT: indica il traffico prima di arrivare al router
  OUTPUT indica il traffico prima di lasciare il router verso l'esterno
  FORWARD: indica il traffico che attraversa il router

- **regola**: individua un flusso di pacchetti che attraversano il punto di controllo in oggetto specificandone la provenienza e la destinazione mediante numeri IP e porte, le interfacce di rete in ingresso e uscita:

-s [!] indirizzo[/maschera]	Specifica l'indirizzo d'origine dei pacchetti
-d [!] indirizzo[/maschera]	Specifica l'indirizzo di destinazione dei pacchetti
-p tcp | udp | icmp	Specifica il protocollo
--sport porta | porta_iniziale:porta_finale	Specifica la porta sorgente
--dport porta | porta_iniziale:porta_finale	Specifica la porta destinazione
-i [!] interfaccia	Specifica l'interfaccia di rete dal quale si ricevono i pacchetti
-o [!] interfaccia	Specifica l'interfaccia di rete alla quale si inviano i pacchetti

Ad esempio

```
iptables –A INPUT –p tcp –s ! 192.168.0.0/16 –d 192.168.0.0/16 –dport 80 –j REJECT
```

impedisce alle macchine non appartenenti alla rete 192.168.*.* di accedere ai servizi sulla porta 80 (HTTP) offerti dalle macchine di quella sottorete

- **obiettivo**: indica cosa fare di quei pacchetti: ACCEPT,DROP o REJECT

Oltre a funzioni di controllo si possono effettuare operazioni di trasformazione di porte e indirizzi (NAT/PAT). In questo modo è possibile abilitare una rete locale con indirizzi privati all'esplorazione di Internet oppure si può permettere l'accesso dall'esterno ad un servizio posto dietro un firewall.

Queste trasformazioni avvengono mediante il comando iptables utilizzando la seguente sintassi:

iptables –t nat opzione_di_comando punto_di_intervento regole –j obiettivo_di_trasformazione

Vediamo in particolare i parametri che cambiano

- **punto di intervento**: indica dove tale trasformazione avverrà. In particolare si ha

  PREROUTING: posizione ideale che precede l'istradamento da parte dell'elaboratore. In questa posizione si modificano gli indirizzi di destinazione
  POSTROUTING: posizione ideale successiva all'istradamento in cui si modificano gli indirizzi di origine

- **obiettivo di trasformazione**: indica come modificare l'origine o la destinazione del pacchetto

Vediamo alcuni esempi per chiarire. Il primo consiste nel permettere agli elaboratori di una rete privata di uscire su una rete pubblica:

```
iptables –t nat –A POSTROUTING –o eth0 –j SNAT –to 1.2.3.4
```

gli elaboratori della rete interna privata quando escono su Internet avranno l'indirizzo IP pari a 1.2.3.4. Per semplicità si sarebbe potuto usare –j MASQUERADE.

Il secondo esempio riguarda l'esposizione di un servizio in funzione sulla rete interna ma accessibile esternamente:

```
iptables –A FORWARD –i eth0 –o eth1 –p tcp –d 192.168.7.7 –dport 8080 –m state –state
NEW,ESTABLISHED,RELATED –j ACCEPT
```

permette alle connessioni con stato NEW,ESTABLISHED e RELATED di entrare se dirette alla porta 8080 della macchina 192.168.7.7. Ovviamente tali richieste devono essere dirottate dal firewall all'elaboratore privato:

```
iptables –t nat –A PREROUTING –p tcp –dport 80 –i eth0 –j DNAT –to 192.168.7.7:8080
```

le richieste fatte al firewall sulla porta 80 vengono dirottate verso la macchina della rete interna 192.168.7.7 sulla porta 8080. Poiché l'HTTP è simmetrico occorrerà modificare in uscita il flusso relativo alla porta 8080 in 80

```
iptables –t nat POSTROUTING –p tcp –sport 8080 –i eth1 –j SNAT –to 1.2.3.4:80
```

in questo modo il flusso che viene inviato da 192.168.7.7 sulla porta 80 apparirà come un flusso inviato da 1.2.3.4 sulla porta 80

Firewall con iptables

Consideriamo il caso più semplice di un elaboratore, dotato di due schede di rete, che funge da router verso l'esterno (usando eth0) per una piccola rete locale (usando eth1). Per prima cosa bisogna caricare i moduli relativi al firewall:

```
/sbin/depmod –a
/sbin/insmod ip_tables
/sbin/insmod ip_conntrack
/sbin/insmod ip_conntrack_ftp
/sbin/insmod ip_conntrack_irc
/sbin/insmod iptable_nat
/sbin/insmod ip_nat_ftp
```

Qindi è necessario abilitare il *forwarding* attraverso le schede:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

tornando al nostro script resettiamo le tabelle

```
#accettiamo tutto in ingresso su tutte le schede in assenza di regole
iptables –P INPUT ACCEPT
#ripuliamo la tabella relativa alle regole in ingresso
iptables –F INPUT
#idem per le regole in uscita
iptables –P OUTPUT ACCEPT
iptables –F OUTPUT
#impediamo per ora un forward completo tra schede
iptables –P FORWARD DROP
iptables –F FORWARD
#ripuliamo la tabella del nat
iptables –t nat –F
```

definiamo finalmente le regole per effettuare forwarding e NAT

```
#inoltra i pacchetti dall'interfaccia locale (eth1) a quella su Internet (eth0)
iptables –A FORWARD –i eth1 –o eth0 –j ACCEPT
#inoltra I pacchetti dall'esterno all'interno solo se appartengono a connessioni in atto
#(viene escluso lo stato NEW)
iptables –A FORWARD –i eth0 –o eth1 –m state –state ESTABLISHED,RELATED  –j ACCEPT
#attiva il masquerading degli indirizzi interni con l'IP pubblico del router
iptables –t nat –A POSTROUTING –o eth0 –j MASQUERADE
```

Si può notare che le regole non agiscono in ingresso, quindi il sistema è facilmente sottoponibile ad attacchi esterni che sfruttano falle del protocollo. Dobbiamo quindi modificare pesantemente lo script.

Innanzitutto applichiamo la strategia seguente:

- bloccare gli accessi e il transito dei pacchetti come politica predefinita

- effettuare l’azzeramento di tutte le regole
- definire le regole di reject
- definire le regole di accept
- attivare dei kernel flag per rendere più sicuro il sistema

quindi avremo

```
#rifiutiamo tutto in ingresso su tutte le schede in assenza di regole
iptables –P INPUT REJECT
#ripuliamo la tabella relativa alle regole in ingresso
iptables –F INPUT
#idem per le regole in uscita
iptables –P OUTPUT REJECT
iptables –F OUTPUT
#impediamo per ora un forward completo tra schede
iptables –P FORWARD DROP
iptables –F FORWARD
#ripuliamo la tabella del nat
iptables –t nat –F
```

Infine procediamo con le ultime precauzioni:

```
#disabilita il ping
echo “1” > /proc/sys/net/ipv4/icmp_echo_ignore_all
#disabilita Smurf IP Denial-of-service
echo “1” > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
#disabilita i pacchetti Source Routed
echo “0” > /proc/sys/net/ipv4/conf/all/accept_source_route
#disabilita la redirect ICMP
echo “0” > /proc/sys/net/ipv4/conf/all/accept_redirects
#Kill timestamp
echo “0” > /proc/sys/net/ipv4/tcp_timestamps
#abilita i SYN cookies per evitare gli attacchi SYN flood
echo “1” > /proc/sys/net/ipv4/tcp_syncookies
```

Un approccio più moderno è il seguente:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -F
iptables -t nat -F
iptables -P INPUT ACCEPT
iptables -P OUTPUT ACCEPT
iptables -P FORWARD DROP
iptables -t nat -I POSTROUTING -s 10.1.3.0/24 -j MASQUERADE
iptables -I FORWARD -s 10.1.3.0/24 -i eth2 -j ACCEPT
iptables -I FORWARD -d 10.1.3.0/24 -i eth0 -j ACCEPT
iptables -A INPUT -p tcp -s ! 193.205.112.0/20 -i eth0 -j DROP
iptables -A INPUT -p tcp -s ! 10.1.0.0/16 -i eth2 -j DROP
```

Strumenti di verifica

Lo strumento più immediato è vedere le porte e i servizi in ascolto su di esse. A tal fine si può usare il comando:

```
nmap $nome_server
```

Per sapere quali connessioni sono attive si può utilizzare **netstat**.

Network File System

Un server Unix può mettere in condivisione una parte del suo file system permettendo ad un altro sistema di *montarlo* all'interno del proprio come avviene con un floppy. Tale possibilità è offerto dal servizio di rete **NFS** che a sua volta si avvale delle RPC.

Quindi, per potersi avvalere di questo servizio abbiamo bisogno che siano attivi i seguenti demoni sulle porte TCP e UDP che generalmente sono:

- Portmapper (111)
- mountd (...)
- nfsd (2049)

Al fine di conoscere su quali porte siano in ascolto **esattamente**, in modo da abilitare opportunamente il firewall, è sufficiente digitare:

```
rpcinfo –p
```

che ci elencherà protocollo, porta e demone in ascolto. Si noti che **ogni volta** che si stoppa il servizio **è necessario controllare su quali porte mountd si pone in ascolto** in modo da riconfigurare il firewall.

Una volta configurato il firewall bisognerà configurare hosts.deny e hosts.allow, presenti nella directory /etc, in modo che le rpc le possano fare solo le macchine da noi desiderate:

in **hosts.deny** avremo:

```
ALL:ALL
```

in **hosts.allow**

```
portmap: 193.205.123.0/255.255.255.0
```

A questo punto possiamo indicare esplicitamente cosa condividere, con chi e come andando a modificare il file **/etc/exports** nel seguente modo:

directory client(opzioni)

ad esempio:

```
/usr    *.canar.org(ro,no_root_squash)
/usr    193.205.123.57(ro,root_squash)
/usr    193.205.123.53(rw)
```

La prima direttiva concede solo la lettura della directory /usr a tutte le macchine del dominio canar.org. permettendo all'utente root di essere effettivamente se stesso senza essere cambiato nell'utente nobody.

La seconda è simile alla prima ma esclude i privilegi dell'utente root. Infine la terza permette di leggere e scrivere.

Lato client il montaggio di uno share di rete deve avvenire a partire da una directory che già esiste sulla macchina client. Ad esempio:

```
mount –t nfs server.canar.org:/usr/database /usr/data
```

monterà nella directory /usr/data la porzione di file system del server NFS a partire da /usr/database.

Lo smontaggio avviene nel modo consueto:

```
umount /usr/data
```

Per verificare quali share presenta un server si usa:

```
showmount –e server.canar.org
```

Per sapere chi sta utilizzando il servizio:

```
showmount -a
```

POP3

Il protocollo POP3 è attivo sulla porta 110 ed utilizza i seguenti comandi:

`USER` *nomeutente*
`PASS` *password*

Una volta stabilita la connessione con il server ci troviamo nella fase di autenticazione.

`APOP` *nomeutente digest*

Come metodo alternativo di autenticazione dell'utente, il protocollo POP-3 prevede anche l'invio criptato della password, come vedremo più in dettaglio negli esempi. Questa funzionalità è prevista come opzionale, e solo alcuni server la supportano.

`STAT`

Ritorna il numero di messaggi presenti nella mailbox e la lunghezza totale in byte.

`LIST`

Ritorna l'elenco dei messaggi giacenti. L'elenco prevede una riga per ogni messaggio, e viene terminato con una riga contenente un solo punto. Ogni riga presenta il numero del messaggio e la sua lunghezza (indicativa) in bytes.

`RETR` *n*

Ritorna il messaggio numero *n*. Al termine del messaggio il server ritorna una riga contenente un solo punto; le righe del messaggio che dovessero contenere almeno un punto iniziale vengono fatte precedere da un altro punto per evitare ambiguità.

`TOP` *n r*

Ritorna l'intestazione del messaggio e le prime *r* righe del corpo del messaggio numero *n*. Questo comando è opzionale, ma pare che tutti i server lo abbiano implementato.

`DELE` *n*

Marca il messaggio numero *n* per essere cancellato una volta chiusa in modo regolare la connessione.

`RSET`

In caso di ripensamento, questo comando rimuove la marcatura dai messaggi marcati per la cancellazione con il comando `DELE`.

`QUIT`

Il server esegue la cancellazione dei messaggi marcati per cancellazione, e quindi chiude la connessione con il client.

Il comando APOP permette di inviare la password criptata. La dinamica di questa procedura consiste in un una prima fase in cui il server invia un *timestamp* che sarà usato dal client per criptare la password da inviare. Un esempio di interazione può essere il seguente:

```
$telnet popmail.libero.it 110
+OK POP3 PROXY server ready (6.0.012) <C1ADC15AA12@pop2.libero.it>
$APOP umberto-salsi 3c7450bfa9b8148d75c3df9677a606a1
+OK authentication successful
$STAT
+OK 0 0
$QUIT
+OK POP3 server closing connection
```

Con il valore **`C1ADC15AA12`** ho ottenuto il digest **`3c7450bfa9b8148d75c3df9677a606a1`**

```
$ echo -n "<C1ADC15AA12@pop2.libero.it>pippo" | md5sum 3c7450bfa9b8148d75c3df9677a606a1
```

In cui pippo è la password e l'opzione –n serve a togliere l'a-capo automatico dalla stringa che deve subire l'operazione di hash md5

Struttura di un messaggio di posta elettronica

Prendiamo ad esempio il seguente messaggio:

```
Message-ID: <3C0D5E3F005FC9EA@ims2c.libero.it>
      (added by postmaster@libero.it)
Date: Wed, 26 Dec 2001 21:19:45 CET
In-Reply-To: <3C28E260.6080703@qui.linux.it>
References: <20011225193905.6E7AA2B6E8@qui.linux.it>
      <3C28E260.6080703@qui.linux.it>
From: Umberto Salsi <umberto-salsi@libero.it>
To: dodo@websic.com, mono@qui.linux.it
Subject: Re: Ho finito l'articolo per il PLUTO!

  Anch'io ho finito la formattazione dell'articolo: adesso mi sembra che
  appaia bene un po' in tutti i browser. Anche la stampa e' ok. Direi che ci siamo: si
  pubblica? Ciao,
  - Umb
```

L'**intestazione** contiene le informazioni tecniche necessarie per l'invio e il riconoscimento del messaggio.

In generale ogni riga contiene un certo **campo di intestazione** costituito dal **nome del campo** (come ad esempio `Subject`), dal carattere ":", e quindi dal **corpo del campo** che si estende fino alla fine della riga (come ad esempio `Re: Ho finito l'articolo per il PLUTO!`).

Il corpo di alcuni campi può estendersi anche su più righe: le righe di continuazione devono obbligatoriamente cominciare con uno spazio o con un codice di tabulazione HT; per ricostruire la riga intera originale, la coppia CR+LF della riga precedente e il primo carattere di spaziatura della riga di continuazione devono essere interpretati dai programmi come un unico carattere di spaziatura.

Nel nostro esempio il campo `Message-ID` e `References` hanno una riga di continuazione. Viceversa, si possono spezzare i corpi dei campi della intestazione solo in corrispondenza degli spazi.

La **riga di separazione** tra l'intestazione e il corpo del messaggio deve essere vuota, cioè non deve contenere caratteri salvo che la coppia CR+LF.

Basterebbe uno spazio, invisibile, per contrassegnare questa riga come continuazione della precedente, falsando la corretta interpretazione della intestazione del messaggio.

Il **corpo** del messaggio contiene il messaggio in prosa. L'RFC 822 non impone alcuna particolare limitazione al contenuto del corpo, né per quanto riguarda il numero di righe, né per quanto riguarda la lunghezza del corpo. L'RFC 2822 impone un limite massimo di 998 caratteri per riga, ma raccomanda non più di 78 caratteri esclusi CR+LF per favorire la visualizzazione sotto le diverse interfacce utente.

La filosofia generale sottostante ai protocolli definiti negli RFC è perciò la seguente: siate permissivi in ciò che i vostri programmi sono in grado di poter interpretare, e siate rigorosi in ciò che essi producono.

Ritorniamo all'intestazione, e descriviamo i principali campi che vi possono apparire. L'RFC 822 dice che questi campi possono apparire scritti indifferentemente in lettere maiuscole o in lettere minuscole, tuttavia raccomanda che, quando si formatta un messaggio, venga rispettata l'ortografia suggerita nel documento, e che anche qui rispetteremo.

L'RFC 822 prevede anche un meccanismo generale per inserire commenti in prosa nel corpo di un campo di intestazione, commenti che il software dovrebbe ignorare. Sono regole un po' ingarbugliate. Qui vediamo solo il tipo di commento che appare più frequentemente, e cioè il nome proprio della persona associato al suo indirizzo email. Ecco alcune delle forme possibili:

```
tizio@qualcosa.it
Tizio Tiziani <tizio@qualcosa.it>
tizio@qualcosa.it (Tizio Tiziani)
```

Nel seguito, ovunque dirò che è possibile inserire un indirizzo email, si può inserire una qualsiasi di queste soluzioni.

`From:` riporta l'indirizzo (o gli indirizzi separati da virgole) del mittente (o dei mittenti) del messaggio. E' raro che un messaggio venga redatto a quattro mani, per cui qui di solito si troverà un solo indirizzo.

Tuttavia nel caso di più indirizzi deve apparire anche un campo `Sender:`, che riporta l'indirizzo del mittente effettivo del messaggio.

`To:` l'indirizzo (o gli indirizzi separati da virgole) del destinatario (o dei destinatari) del messaggio.

`Cc:` il campo copia-carbone, ovvero "per conoscenza", funziona esattamente come il precedente. La differenza è puramente etica.

`Bcc:` il campo blind carbon copy (copia carbone cieca) funziona esattamente come i campi `To` e `Cc`. L'unica differenza è che questo campo non viene mai trasmesso con il messaggio, per cui gli indirizzi in esso contenuti non appaiono ai destinatari e rimangono quindi riservati.

`Reply-To:` contiene l'indirizzo, o gli indirizzi, ai quali vogliamo siano dirette eventuali risposte al messaggio.

`Message-ID:` contiene l'identificativo univoco del messaggio. Si tratta di una stringa che il server SMTP calcola basandosi tipicamente sul nome di dominio, sul tempo e su un numero progressivo. Questo permette anche di tracciare gli spostamenti del messaggio (come registrati dei log file dei server) e diagnosticare eventuali problemi nella propagazione.

`In-Reply-To:` riporta l'ID del messaggio o dei messaggi dei quali questo messaggio costituisce la replica.

`References:` riporta l'elenco degli ID dei messaggi di cui questo messaggio costituisce il seguito, anche se non costituisce necessariamente una replica a tutti questi. Dall'esame di questi ID, i programmi client possono ricostruire il corretto albero di lettura ("thread") e presentare i messaggi in una forma strutturata che ne evidenza le correlazioni.

`Subject:` è l'oggetto del messaggio.

`Date:` contiene la data in cui il messaggio è stato finito di scrivere, che spesso coincide con la data di invio. Se la data manca, il server SMTP l'appone automaticamente. Tipicamente questa data viene apposta dal programma client del vostro computer, quindi una raccomandazione: controllate bene l'orologio, la data e il fuso orario del vostro PC se non volete fare la figura degli eternauti.

`X-abcdefg:` i campi che iniziano con i caratteri "`X-`" si considerano sperimentali, vengono trasmessi regolarmente, ma di norma vengono ignorati dai sistemi. Qualcuno li usa per inserire informazioni, altri ci mettono la pubblicità, altri ancora tentano di inserire la fotografia dell'autore del messaggio (`X-Face`). Da usare con fantasia.

Protocollo SMTP

Il protocollo SMTP viene usato per inviare la posta tra Mail-agent. Utilizza la porta 25. Per capirne il funzionamento vediamo il classico “giochetto”:

```
$ telnet mail.libero.it smtp
Trying 195.123.94.65...
Connected to localhost.localdomain.
Escape character is '^]'.
220 smtp3.libero.it ESMTP Service (6.0.032) ready
HELO localhost
250 smtp3.libero.it
MAIL FROM:<umberto-salsi@libero.it>
250 MAIL FROM:<umberto-salsi@libero.it> OK
RCPT TO:<dodo@websic.com>
250 RCPT TO:<dodo@websic.com> OK
RCPT TO:<mono@qui.linux.it>
250 RCPT TO:<mono@qui.linux.it> OK
DATA
354 Start mail input; end with <CRLF>.<CRLF>
Date: Wed, 26 Dec 2001 21:19:45 CET
In-Reply-To: <3C28E260.6080703@qui.linux.it>
References: <20011225193905.6E7AA2B6E8@qui.linux.it> <3C28E260.6080703@qui.linux.it>
From: Umberto Salsi <umberto-salsi@libero.it>
To: dodo@websic.com, mono@qui.linux.it
Subject: Re:

Ho finito l'articolo per il PLUTO! Anch'io ho finito la formattazione dell'articolo: adesso
mi sembra che appaia bene un po' in tutti i browser. Anche la stampa e' ok. Direi che ci
siamo: si pubblica? Ciao,
Umb
.
250 <3C0D5E3F005FC9EA> Mail accepted
```

```
QUIT
221 smtp3.libero.it QUIT
Connection closed by foreign host.

$
```

In particolare il protocollo è formato dai seguenti comandi:

`HELO` *`hostname`*

Il primo comando che usiamo dichiara al server semplicemente il nome del nostro computer. Questo nome apparirà nelle intestazioni `Received`, per cui è meglio evitare di scegliere nomi imbarazzanti. Il server ci risponde semplicemente col suo nome, e il comando si esaurisce qui.

`MAIL FROM:<`*`mittente`*`>`

Qui dichiariamo il nostro indirizzo di posta elettronica. Di solito il server fa i suoi controlli per verificare che sia valido. Inoltre è a questo indirizzo che verranno segnalati eventuali disguidi verificatisi durante il recapito, per cui se siamo interessati a sapere che ne è stato del nostro messaggio, qui è meglio mettere il nostro vero indirizzo. Il server ci rassicura con il suo OK, e possiamo andare avanti.

`RCPT TO:<`*`destinatario`*`>`

Con questo comando informiamo il server dell'indirizzo del destinatario del messaggio. Anche qui il server può fare le sue verifiche preventive sull'indirizzo indicato. Nel nostro caso il server ci risponde con un altro bell'OK. Possiamo continuare e specificare altri destinatari, nel caso volessimo recapitare lo stesso messaggio a più persone, come avviene nel nostro esempio.

`DATA`

Questo comando istruisce il server che stiamo per trasmettere il messaggio vero e proprio. Il server si premura di ricordarci che dovremo terminare il messaggio scrivendo una linea costituita da un solo punto. Quello che non può precisare in questa nota telegrafica è che tutte le righe del messaggio che dovessero cominciare con un punto, dovranno essere fatte precedere da un altro punto. Si tratta della stessa convenzione utilizzata dal server POP-3 che abbiamo già visto, ma questa volta applicata al contrario: siamo noi a dover garantire il rispetto di questa regola. Ovviamente, i programmi di posta elettronica faranno altrettanto. Nel nostro messaggio di esempio non ci sono righe che cominciano con un punto, per cui qui non vediamo in azione questo accorgimento. Completato il messaggio, inviata la riga di chiusura costituita da un solo punto, il server ci risponde che il messaggio è stato accettato per il recapito.

`QUIT`

Siccome non abbiamo altro da fare in questo collegamento, non ci rimane che salutare e chiudere la connessione con un rispettoso QUIT.

Codifica dei messaggi di posta elettronica

La necessità di una codifica universale ha fatto nascere lo standard ISO-10646. Un implementazione di tale standard è costituito dal set di caratteri UNICODE. Una possibile codifica del set Unicode è costituita dalla codifica UTF-8 che può utilizzare da 1 a 3 bytes a seconda del tipo di carattere.

Poiché tale codifica sfrutta 8 bit ed i sistemi di posta elettronica utilizzano 7 bit si è sviluppata la codifica UTF-7 per il testo delle e-mail(RFC2152). In pratica il carattere + viene utilizzato per indicare l'inizio di una codifica base64 modificata.

Il problema di trasmettere file binari, invece, si risolve mediante la codifica uuencode che, oltre a codificare il file utilizzando un set di 64 caratteri ASCII, aggiunge un intestazione (begin) contenente le informazioni sul file.

Altrimenti si può usare la codifica Base64 che, insieme alla codifica MIME, viene utilizzato per trasmettere i file binari o di tipo multimediale in genere. Vediamo alcuni esempi interessanti:

Inviare un messaggio HTML. Capita spesso al sistemista e al programmatore di dover generare email in modo automatico, per esempio per segnalare ad un utente o a un cliente informazioni statistiche degli accessi al suo sito WEB, oppure per segnalargli il superamento della quota per la sua mailbox. Per ottenere un risultato più elegante potremmo inviargli un documento HTML, che fa la sua bella figura. Ecco come confezionare il messaggio:

```
MIME-Version: 1.0
Content-Type: text/html
From: root@webfarm.it
To: tizio@qualcosa.it
Subject: AVVISO di superamento quota disco
<HTML><BODY bgcolor="#ff3030"><H1>ATTENZIONE!</H1> <P>La tua mailbox ha superato la
dimensione massima consentita di <B>10 MB</B>. Per favore, provvedi prima possibile a
svuotarla, altrimenti per motivi tecnici saremo costretti a <B>cancellare tutto il contenuto
entro 24 ore!</B></P> <P>Per maggiori informazioni sulle condizioni del servizio, leggi il <A
href="http://webfarm.it/contratto">contratto di fornitura</A>. </BODY></HTML>
```

Ho evidenziato in grassetto le due righe chiave di tutto il meccanismo: l'intestazione `MIME-Version: 1.0` è obbligatoria e indica che il messaggio è in formato MIME versione 1.0.

La riga seguente `Content-Type: text/html` indica che il corpo del messaggio è di tipo testo e di sottotipo HTML, pertanto la dicitura `text/html` è il tipo MIME di questo messaggio. Siccome non abbiamo specificato diversamente, il messaggio si considera codificato a 7 bit usando il charset ASCII.

Una volta scritto il messaggio in un file di testo, per inviarlo basta il comando:

```
cat lettera.txt | sendmail -t
```

Inviare un messaggio con una immagine. Usando i vari tipi MIME registrati, potremo inviare anche altri tipi di informazioni, per esempio la nostra solita figura:

```
MIME-Version: 1.0
Content-Type: image/png
Content-Transfer-Encoding: base64
From: root@webfarm.it
To: tizio@qualcosa.it
Subject: Una bella figurina per te!

iVBORw0KGgoAAAANSUhEUgAAABAAAAAQAQAAAAA3iMLMAAAANUlEQVR42mMQ
62YoW81QNptBYDXD//8MR9wZrl5nuBLOcD0cxAaK7JZimF3FsNqKYXM9kA0A vSoSfMoIiCAAAAAASUVORK5CYII=
```

Messaggio a più parti alternative. E' sempre più frequente trovare in circolazione questo tipo di messaggi dopo l'affermarsi del client di posta Outlook Express di Microsoft.

Questo programma viene configurato per default con l'opzione MIME (l'altra possibilità mi pare che sia l'Uuencode), e in questo caso il programma formatta tutti i messaggi con un formato MIME di tipo `multipart/alternative`.

Il concetto alla base di questo tipo MIME è il seguente:

non tutti i client di posta hanno le stesse possibilità di visualizzazione. Per esempio, alcuni sono in grado di mostrare un documento HTML, altri possono mostrare solo testo. Il formato a più parti alternative permette di inserire lo stesso contenuto in più forme diverse, lasciando al client la possibilità di scegliere il formato preferito tra quelli disponibili.

Come esempio ripropongo l'email di notifica spazio esaurito che prima avevamo scritto solo come HTML. Qui lo scriveremo anche come testo ASCII:

```
MIME-Version: 1.0
Content-Type: multipart/alternative;
       boundary="ZQZQZQ"
From: root@webfarm.it
To: tizio@qualcosa.it
Subject: AVVISO di superamento quota disco

Questo e' un messaggio in piu' parti alternative. Se stai leggendo queste righe, significa
che il tuo client di posta non supporta il formato MIME. Cio' e' un bene, cosi' puoi vedere
come funzionano veramente le cose. :-)

--ZQZQZQ
Content-Type: text/plain

ATTENZIONE! La tua maibox ha superato la dimensione massima consentita di 10 MB. Per favore,
provvedi prima possibile a svuotarla, altrimenti per motivi tecnici saremo costretti a
cancellare tutto il contenuto entro 24 ore! Per maggiori informazioni sulle condizioni del
servizio, leggi il contratto di fornitura (http://webfarm.it/contratto).

--ZQZQZQ
Content-Type: text/html
```

```
<HTML><BODY bgcolor="#ff3030"><H1>ATTENZIONE!</H1> <P>La tua mailbox ha superato la
dimensione massima consentita di <B>10 MB</B>. Per favore, provvedi prima possibile a
svuotarla, altrimenti per motivi tecnici saremo costretti a <B>cancellare tutto il contenuto
entro 24 ore!</B></P> <P>Per maggiori informazioni sulle condizioni del servizio, leggi il <A
href="http://webfarm.it/contratto">contratto di fornitura</A>. </BODY></HTML>

--ZQZQZQ–
```

Nella **intestazione** abbiamo la solita linea `MIME-Version`, e poi la linea `Content-Type`. Questa volta il tipo è `multipart/alternative`.

Subito dopo il tipo ho aggiunto in una riga di continuazione il parametro `boundary="ZQZQZQ"`; questo parametro serve a dichiarare la stringa di separazione delle varie parti che costituiscono il corpo del messaggio.

Questa stringa deve essere univoca, e per evitare collisioni i programmi tendono a renderla molto lunga e con una sequenza casuale di cifre, lettere e altri simboli.

Ogni **sezione MIME** nel corpo inizia con la **riga di separazione**, costruita con due segni - (meno) seguiti dalla stringa di boundary che ho specificato nell'intestazione.

Subito dopo c'è l'intestazione MIME che dichiara il tipo dei dati di questa sezione, una riga vuota, e quindi i dati stessi della sezione. La sezione termina dove compare la successiva riga di separazione.

Al **termine** del messaggio bisogna inserire la riga di separazione seguita da altri due segni ``-" (meno).

La cosa più interessante è che ogni sezione MIME nel corpo del messaggio assume la stessa forma di un messaggio, con la sua intestazione, la riga vuota, e il corpo. In effetti MIME consente dichiarazioni ricorsive: ogni sezione può essere del tipo multiparte e contenere quindi un altro messaggio MIME.

Riguardo all'uso di font alternativi nella intestazione, il MIME consente l'uso di vari charset, incluso ISO e UTF-8: i caratteri non ASCII dovranno essere avvolti dentro a una particolare sequenza di caratteri come in questo spezzone di esempio:

```
Subject: =?ISO-8859-1?Q?Propriet=E0?= del formato MIME
```

A causa della presenza della lettera accentata "à", che non fa parte del charset ASCII, il client di posta ha codificato l'oggetto del messaggio specificando il charset ISO-8859-1, ha circondato la parola in questione con una serie di strani caratteri, e quindi ha codificando la lettera accentata con il suo valore esadecimale `E0`.

E' possibile anche usare un charset non ASCII nel corpo del messaggio introducendo un parametro apposito nel campo di intestazione `Content-Type`:

```
ContentType: text/plain; charset="ISO-8859-1"
```

Con queste ultime annotazioni telegrafiche chiudo il discorso sul formato MIME. Ricordo solo che lo stesso formato viene utilizzato anche dal protocollo HTTP per il WEB: in quel caso il canale è sempre pulito a 8 bit per byte

Appendice 1: Segnaccento obbligatorio

Quello che segue è la norma UNI 6015 sull'uso degli accenti. Il testo è stato ottenuto da *Scienza, tecnologia e arte della stampa e della comunicazion*e, *Preparazione del manoscritto*
http://www.apenet.it/grafica/libri/Grafica/Grafica01/1206.html

Segnaccento obbligatorio nell'ortografia della lingua italiana (Uni 601567):

1. *Scopo*
La presente unificazione ha lo scopo di stabilire le regole ortografiche per il segnaccento nei testi stampati in lingua italiana, quando esso sia obbligatorio.

2. *Definizione*

2.1 Il segnaccento (o segno d'accento, o accento scritto) serve a indicare esplicitamente la vocale tonica, per esempio: *andr*à, *colp*ì, *tem*é, *virt*ù.
2.2. Il segnaccento può essere grave ` o acuto ´ .

3. *Uso*
Il segnaccento è obbligatorio nei casi seguenti:

3.1. Su alcuni monosillabi, per distinguerli da altri monosillabi che si scrivono con le stesse lettere ma senza accento:
ché («poiché», congiunzione causale) per distinguerlo da *che* (congiunzione in ogni altro senso, o pronome);
dà (indicativo presente di dare) per distinguerlo da *da* (preposizione) e *da'* (imperativo di dare);
dì («giorno») per distinguerlo da *di* (preposizione) e *di'* (imperativo di dire);
è (verbo) per distinguerlo da *e* (congiunzione);
là (avverbio) per distinguerlo da *la* (articolo, pronome, nota musicale);
lì (avverbio) per distinguerlo da *li* (articolo, pronome);
né (congiunzione) per distinguerlo da *ne* (pronome, avverbio);
sé (pronome tonico) per distinguerlo da *se* (congiunzione, pronome atono);
sì («così», o affermazione) per distinguerlo da *si* (pronome, nota musicale);
té (pianta, bevanda) per distinguerlo da *te* (pronome).

3.2. Sui monosillabi: *chi*ù, *ci*ò, *di*è, *f*é, *gi*à, *gi*ù, *pi*è, *pi*ù, *pu*ò, *sci*à.

3.3. Su tutte le parole polisillabe su cui la posa della voce cade sulla vocale che è alla fine della parola, per esempio: *piet*à, *lunedì*, *far*ò, *autogr*ù.

4. *Forma*

4.1. Il segnaccento, nei casi in cui è obbligatorio, è sempre grave sulle vocali: a, i, o, u.
4.2. Sulla e, il segnaccento obbligatorio è grave se la vocale è aperta, è acuto se la vocale è chiusa:
è sempre grave sulle parole seguenti:

ahimè e *ohim*è, *caff*è, *canap*è, *cio*è, *coccod*è, *diè* e *gil*è, *lacch*è, *pi*è, *t*è; inoltre sulla maggior parte dei francesismi adattati, come *beb*è, *cabar*è, *pur*è, ecc. e sulla maggior parte dei nomi propri, come *Giosu*è, *Mos*è, *No*è, *Salom*è, *Tigrè;*

- è acuto sulle parole seguenti:
ché («poiché») e i composti di *che (affinch*é, *macch*é, *perch*é, ecc.), *fé* e i composti *aff*é, *autodaf*é, i composti di *re* e di *tre (vicer*é, *ventitr*é), i passati remoti *(cred*é, *tem*é,

ecc., escluso *di*è), le parole *merc*é, *n*é, *scimpanz*é, *s*é, *test*é.

4.3. Anche per la *o* si possono distinguere i due timbri (aperto o chiuso) con i due accenti (grave ed acuto) ma solo in casi in cui l'accento è facoltativo, per esempio: *còlto* (participio passato di *cogliere*), e *cólto* («istruito»).