Appunti di sistemi virtualizzati

Appunti di sistemi virtualizzati

Prime lezioni teoriche

La **Virtualizzazione** è quel processo che permette di astrarre le componenti hardware in modo da poter rendere disponibili ai software le loro risorse. Tramite questo processo possiamo installare sistemi operativi su hardware virtuale. L'insieme delle componenti hardware prende il nome di macchina virtuale. Ogni macchina fisica su cui è possibile ospitare una o più macchine virtuali si chiama **sistema host**. Con **sistema guest** invece definiamo l'insieme delle risorse hardware e software ospitate sopra il sistema hostante. Con **Hypervisor** definiamo il componente software che crea e manda in esecuzioni le Virtual Machine, inoltre ha il compito di astrarre le risorse hardware e di monitorare l'esecuzione delle VM.
Possiamo anche avere la **Virtualizzazione Desktop**, o anche detta *Type 2* che permette alla machcina virtuale di sfruttare le periferiche della machcina fisica. Inoltre questa virtualizzazione è utile per far funzionar eun software non compatibile con il sistema operativo dell'host.
La virtualizzazione avviene anche lato server, ma in un modo leggermente differente dove, utilizzando sistemi operativi dedicati, l'utente non opera direttamente sulla machcina fisica bensì si collega da remoto alla machcina virtuale

Virtualizzazione e Emulazione

Abbiamo detto quindi che con il termine virtualizzazione definiamo la possibilità di poter far eseguire uno o più sistemi operativi da un unico PC in un ambiente chiamato VM. Il sistema ospitato crede di essere per davvero su una macchina fisica, quindi esegue tutte le istruzioni come se non ci fosse un mediatore tra CPU e sistema hostato chiamato Hypervisor.
Con **Emulazione di processore** invece vediamo che il tutto si complica perchè, in questo caso, anche l'hardware viene emulato, traducendo ogni istruzione proveniente dal sistema guest affichè possa essere eseguita dall'host, rendendo il tutto più lento. L'emulazione però permette di emulare tutti i sistemi retro-compatibili con la macchina hostante.
Le nuove CPU prevedono dei *confini* che possono essere superati solo se le istruzioni machcina fornite sono di tipo **Privilegiato**. Infatti abbiamo due tipi di istruzioni, le **Privilegieate** e le **Generali**. Le prime possono essere eseguite solo dal kernel del sistema operativo, mentre le altre sono eseguite dallo *spazio utente*, come applicazioni. Ogni processore possiede almeno 4 livelli di protezione con indice da 0 a 3, dove 0 indica il più alto e 3 il più basso. Con livello 3 identifichiamo il livello delle istruzioni generali, ovvero tutte quelle usate scrivendo programmi in assembly; con livello 0 invece vediamo che le istruzioni eseguite sono potenzialmente pericolose e sono tipiche del sistema operativo. Tutto questo è necessario a capiere che andando a lavorare in un ambiente virtualizzato le cose si complicano, avendo la necessità di gestire due sistemi operativo contemporaneamente. Per questo quando andiamo a creare una machcina virtuale la viene aggiunto un livello di protezione alla CPU ancora più

interno che possiamo chiamare -1, dove vengono eseguite le **system call** fatte dalla macchina virtuale. Adesso che sappiamo le differenze a livello logico, possiamo fare la prima grande distinzione tra risorse virtuali, ovvero le **macchine virtuali e i container**.

- Con **macchina virtuale** vediamo che tramite la virtualizzazione dell'hardware utente possiamo installare un sistema operativo con una CPU virtuale dello stesso tipo di quella fisica. Le macchine virtuali vengono eseguite grazie agli *Hypervisor*. Possono esistere casi dove il sistema operativo tra hardware e hypervisor non esiste, rendendo l'hypervisor un surrogato del sistema operativo.
-
    - Si definisce virtualizzazione di **tipo 1 o bare-metal** quando l'hypervisor prende il posto del sistema operativo, questo è spesso utilizzato nella virtualizzazione server.
-
    - Si definisce virtualizzazione di **tipo 2 o hosted** quando l'hypervisor è un normale processo utente su di un sistema operativo ospitante, in questo caso parliamo di virtualizzazione *desktop*.
- Come abbiamo detto, la virtualizzazione hardware fornisce una macchina virtuale sulla quale si può installare un sistema operativo, ma questa machcian può assumere due tipi, la **Para-virtualization e la Full-virtualization**.
-
    - Nella **Para-virtualization** abbiamo una macchina virtuale con interfacce diverse da quella fisica, avendo la necessità di aggiungere all'hypervisor delle API kernel per utilizzare le istruzioni privilegiate tramite chiamate dette **Hypercall**, che somigliano molto alle *system call* ma fatte all'hypervisor.
-
    - Nella **Full-virtualization** abbiamo una macchina con la stessa interfaccia di una macchina fisica, quindi non è necessario dover modificare il sistema operativo guest. Questo metodo però comporta la chiamata diretta di istruzioni privilegiate alla CPU fisica che, in alcuni casi, possono provocare dei problemi con il sistema operativo host. Questo problema è risolvibile in due modi, tramite l'**Hardware-assisted o la Binary translation**. La prima comporta l'aggiunta di un nuovo livello CPU -1 per la gestione delle istruizioni del sistema guest. Questo livello è presente in tutte le CPU dal 2007. Inoltre crea anche delle tabelle di memoria per ogni macchina virtuale creata.
      Con Binary translation invece vediamo che le istruzioni vengono eseguite come se la CPU fosse in modalità debug, sospendendo via via l'esecuzione delle istruzioni.

I Contaiener

Con **container** definiamo la pacchettizzazione di applicazioni che permette di dividere le installazini delle applicazioni in partizioni isolate. Questo perchè in un container portiamo sia i file di libreria e sia tutte le infromazioni necessarie per l'esecuzione dell'applicazione. Per far eseguire i container abbiamo bisogno di un livello software che permetta l'esecuzione, quindi così come le macchine virtuali utilizzano degli hypervisor, così i container utilizzano i **container-runtime**. I container sono avviabili

solo se hanno lo stesso kernel della macchina hostante. Inoltre, i container possono decidere di lavorare sul file system della macchina ospitante.

Docker

I container possono essere gestite da applicazioni come **Docker**, basata su kernel Linux, che funziona da hypervisor. Un modello server di gestione delle richeiste utente è quello di utilizzare **Docker Swarm** in una rack server. Questo permette di avere un dispositivo manager che gestisce le richieste controllando il carico delle macchine che lavorano(worker) e, in caso ci sia una macchina con poco carico gli dice di creare un container per la gestione delle richieste(task). Quando un client richiede un servzio tramite un browser, in realtà al richiesta è fatta al manager che poi distribuisce le task in modo *invisibile*, non facendo notare al client quale worker lo sta eseguendo. Naturalmente, i worker devono essere aggiunti manualmente da persone in caso di overflow di richieste.

Kubernetes

È un applicativo software che permette la gestione dei container e la scalabilità dei worker. Si basa sui **pods**, quindi un contenitore di container cooperante che fingono di essere sulla stessa macchina tramite *localhost*. Il sistema di kubernetes è molto simile a quella di Docker, infatti abbiamo un manager che in questo caso si chiama **Control Plane** e poi abbiamo un **container-runtime** che è il runtime e quindi molto simile all'hypervisor. Ogni pod può comunicare tramite un **kube-proxy** che permette di avere una rete virtuale. Lo smistamento delel richieste degli utenti è analogo a quello di Docker, quindi lo manda al pod con meno lavoro. Inoltre ha la gestione automatica dei pod *rotti* iistanziandone di nuovi. È anche scalabile, ovvero decide quante pods utilizzare automaticamente in base al numero di richeiste. Questo metodo è chiamato **autoscaling** e ce ne sono di tre tipi:

- **autoscaling verticale** si basa sull'aumento dei container in ogni pod in base alla quantità di CPU e risorse utilizzate;
- **autoscaling orizzontale** aumenta il numero pod di tipo web-server;
- **cluster autoscaling** aumenta il numero di nodi tramite un'operazione che parte dal **Control Plane**. Tutto qeusto è scelto dall'applicazione Kubernetes presente nel cluster, anche perchè ogni criterio ha la necessità di aumentare le risorse del cluster stesso. Il problema viene quando dobbiamo accendere un cluster fisico, perchè è molto difficile da automatizzare solo tramite software. Per risolvere questo si può creare un cluster di macchine virtuali su cloud tramite un provider. Questo eprmette di gestire le machcine virtuali in modo automatico, senza doverele avere fisicamente. Di solito i servizi cloud offrono API che permettono la gestione di cluster e container. Non tutti lo offrono e per questo piò essere necessario customizzare le API tramite software come **Terraform** che permette di utilizzare macchine virtuali su sistemi cloud in modo indipendenti, ovvero nascondendo alcuni dettalgi ai provider.
  Possono esistere dei servizi cloud che offrono come servizio dei cluster Kubernetes già pronta, senza doverci occupare realmente della gestione delle macchine virtuali e quindi dei worker. Questo servizio si chiama **Kubernetes as a Service.** Possiamo avere anche un **Cluster as a Service** che permette di avere anche dei container istanziati oltre al cluster kubernetes. Però, il

container istanziato è sempre lo stesso e così avremo tante pods con un solo container che esegue lo stesso servizio. Nonostante ciò l'idea permette di spalmare il servizio su tutti i pods presenti.
Possiamo anche avere dei **Function as a Service** è il modo più facile per creare funzioni senza sapere cosa c'è dietro. In parole povere noi forniamo la nostra funzione al provider cloud e lui si occuperà di hostare la funzione.

Bash

Definiamo **CLI** o **command line interface** il terminale che ci permette di impartite comandi al sistema operativo sotto forma di sequenze di caratteri o stringhe. i comandi poi vengono interpretati dalla **Shell** che, a sua volta, è un programma. Possiamo dare ordini al terminale in due modi, tramite comandi o eseguibili. Gli eseguibili sono file scritti in un linguaggio macchina. I comandi invece sono letteralemnte codici interpretati presenti nel sistema che hanno delle funzioni ben specifiche.
Abbiamo diversi tipi di shell, ma la più utilizzata è il **bash** che è valido per la maggior parte dei sistemi operativi. In Linux vediamo che abbiamo una root chiamata */* a differenza di Windows che può essere una lettera come **C:**. Inoltre su Linux ogni volta che creiamo un utente viene creata una cartella per l'utente stesso e, questa cartella nel cmd è identificata dalla **tilde ~**.

Comandi bash

Nome	Descrizione
cd	change directory, quindi cambia la cartella inserendo il path (assoluto o relativo)
cp nomefile	copia il file selezionato e, dopo il nome, scrivere il path
rm nomefile	rimuove il file o cartella
man	concatenato ad un altro comando permette di leggere il manuale del comando
history	permette di vedere tutti i comandi fatti in precedenza
cat nomefile	visualizza in output il contenuto di un file
pwd	permette di vedere il percorso in cui ti trovi
ls	permette di vedere la lista dei file e cartelle nella cartella attuale
which nomefile	visualizza il percorso del file se esistente
mv nomefile destinazione	permette di spostare il file
touch	crea un file senza
vi oppure nano	permettono di creare e di scrivere all'interno di un file
head nomefile	mostra le prime 10 linee del file
tail nomefile	mostra le ultime 10 linee del file
find nomefile	cerca i file nel pc

Nome	Descrizione
grep	cerca tra le righe del file delle parole
read nomevariabile	legge dallo standard input e inserisce il valore nelle variabili che lo seguono
wc	conta il nome di parole in un file
alias comando='comando'	permette di mascherare il comando di destra con quello di sinitra. Per una migliore comprensione degli esercizi consiglio di fare alias ip='ip -c'. Per togliere questo alias basta scrivere unalias ip
sudo nome_applicazione &	avvio applicazione in background
sudo chmod +x nomefile	permette di dare l'accesso di eseguire il file come negli sh. usare u+x permette l'avvio all'utente loggato
;	usare un ; nella stessa riga permette di eseguire il comando a destra e sinistra in modo individuale
echo	permette di visualizzare a video un messaggio
chown	permette di cambiare il proprietario del file scrivendo il nome del nuovo propietario e il nome del file
export	dichiaro una variabile come globale
source	permette di eseguire i comandi dentro uno script senza creare una subshell, andando a lavorare nell'ambiente vero e proprio
unset	togli il valore all'iterno di una variabile
env	mostra tutte le variabili di ambiente

Nella shell possiamo dichiarare variabili che possono essere utilizzate in vario modo, ma ha una sintassi molto stretta che non permette di fare errori. La prima restrizione è nel modo di dichiarare le variabili dove, se mettiamo uno spazione tra la variabile e il valore come in `VAR=HELLO`. Adesso questa VAR è presenta nella nostra shell e può essere utilizzata anche negli script. Esistono anche variabili globali impostate dal sistema operativo in grado di fornire informazioni utili all'utente come la variabile **PATH, USER, PPID**.

Ogni file e directory ha dei permessi che limitano l'uso del file stesso. Esistono 3 tipi di utenti nel caso di **Unix**, l'**user** vero e proprio, il **group** e l'**others**. Gli attributi possibili per ogni file sono **read, write e execute** che hanno un valore ben distinto, ovvero 4, 2 e 1. Ogni file può avere un valore da 0 a 7 che indica quali attributi sono assegnati al fine tramite una somma. Genericamente assegneremo ai file tramite il comando chmod la possibilità di poterlo eseguire con `chmod 700 script.sh` dove ogni cifra indica il gruppo di appartenenza. Con il comando **ls -al** permette di avere come output tutti i file presenti nella nostra posizione e molto dettagli, tra cui i permessi. Noteremo subito che alcuni file hanno la *d* come primo carattere, indicando che il file è una directory. Se invece vediamo il - allora sappiamo che è un file generico. È possibile trovare al posto della **x** una **s** che indica un diverso tipo di esecuzione in base all'utente che lo esegue.

Ogni shell può avere una o più **subshell** che possono essere utilizzate in vari modi, come eseguire

script o processi in background. Queste subshell lavorano come figli della shell principale, ereditanto le caratteristiche e variabili di ambiente, ma non variabili create dall'utente. Il caso più comune di subshell è quello degli script dove, nella prima riga, andiamo a specifiare l'interprete del codice tramite la stringa **#!/bin/bash** che indica da quale cartella prendere l'interprete per eseguire lo script. Ogni volta che andiamo a far girare uno script questo viene avviato con subshell che, al termine dell'esecuzione, viene eliminata.

Wildcards

Con **wildcard** definiamo una serie di metacaratteri che vengono usati per sostituire una quantità di caratteri più o meno definita. Questi caratteri sono pochi ma molto importanti:

Nome	Descrizione
?	Rimpiazza un solo carattere in una stringa
*	Rimpiazza una qualunque serie di caratteri
[ ]	Permette di creare un range nel quale dovrà essere presente il carattere

Quest ultimo è il più importante che ha molte funzioni, dalla ricerca di un carattere alfabetico a quello di un numero, gli esempi più comuni sono

Nome	Descrizione
[abk]	Il carattere può essere solo uno dei 3 presenti nelle quadre
[1-7]	Il carattere può essere un numero tra 1 e 7
[c-f]	Il carattere può essere una lettere tra c e f
[[:digit:]]	Una sola cifra numerica
[[:upper:]]	Un solo carattere maiuscolo
[[:lower:]]	Un solo carattere minuscolo

Un esempio di wildcards concatenate può essere: **?[[:lower:][:lower:]]*** che indica una stringa formata da un carattere iniziale, 2 caratteri in munuscolo e poi una serie di caratteri indefinita.

Parameter Expansion

Espansione utile a gestire i parametri passati a uno script tramite apposite variabili

Nome	Descrizione
$#	numero di argomenti passati allo script
$0	il nome del processo in esecuzione
$1	primo argomento, ma con un altro numero possiamo identificarne N
$*	tutti gli argomenti passati a riga di comando concatenati con spazio per separarli

---

Nome	Descrizione
$@	restituisce il nome del processo o script in esecuzione

Valutazione aritmetica

Per i controlli su numeri interi dobbiamo utilizzare una particola sintassi che permette alla bash di poter identificare e quindi trattare i valori in modo diverso. Grazie agli operatori **(( ))** possiamo racchiudere all'interno operazioni solo se **sono le uniche operazioni da eseguire su una riga**. Per racchiudere solo una parte della riga dobbiamo utilizzare **$(( ))**. Queste espressioni posso contenere all'interno tutti gli operatori aritmetici e, in aggiunta, anche altre parentesi per modificare l'ordine delle operazioni.

Riferimenti Indiretti a Variabili

Analogomante a C possiamo avere dei simil puntatori che permettono di poter prendere il valore della variabile. Questo conceto è un po' strano ma tramite un esempio si può capire meglio:

```
varA=pippo
nomevar=varA
echo ${!nomevar}
stampa a video pippo
```

Questo accade perchè scrivendo **${!nomevar}** è come scrivere **${varA}**, qeusto perchè andiamo ad accedere al valore interno di una variabile.
Questo piccolo esercizio permette di capire il funzionamento intrinseco di questo operatore:

```
#!/bin/bash
echo "ho passato $# argomenti alla shell"
echo "il nome del processo in esecuzione e' $0"
echo "gli argomenti passati a riga di comando sono $*"
NUM=1
while (( "${NUM}" <= "$#" ))
do
    echo "arg ${NUM} is ${!NUM} "
    ((NUM=${NUM}+1))
done
```

dove tramite la sostituzione di **${!NUM}** otteniamo in realtà un numero tra 1 e 3. Con questo otteniamo che nella stampa finale verrà letto non il valore di NUM ma il valore di **${1}**, quindi il valore del primo parametro passato al programma.

Inoltre, con la funzione **${#var}** invece otteniamo il numero di caratteri della stringa contenuta nella variabile.

Exit status

Ogni comando che viene eseguito nella bash ha un valore di ritorno *nascosto* che, se richiamato, può dare molte informazioni all'utente. Questo valore viene restituito tramite la variabile **$?** che viene modificata ogni volta che un comando termina. Questi status sono diversi ma generalemnte quando si ha valore 0 vuol dire che il comando è andato a buon fine, un numero diverso da 0 è un errore. per poter vedere l'output basta fare il comando **echo $?**.

Command substitution

Questa funzione ha lo scopo principale di poter rendere disponibile l'output di un comando in fase di runtime di uno script. Questo è possibile grazie al quoting di un coomando dentro delle ` o melglio **backtick** che possiamo scrivere con il codice ascii alt+96. Tutto quello che viene scritto all'interno di queste backtick viene eseguito come comando e quindi può restituire un valore. Per esempio:

```
OUT=`./primo.exe`
echo "l'output del processo e' ${OUT}"
```

questo restituisce l'output del file primo.exe all'interno di uno script. All'interno delle backtick sono presenti tutte le espansioni per sostituire caratteri speciali e variabili. Se vogliamo evitare che le wildcards vengano utilizzate in un output basta far seguire un echo dai **doppi apici**. Se invece vogliamo che ci venga visualizzato tutto il contenuto di una stringa, con variabili, comandi e wildcards, allora dobbiamo usare l'**apice singolo**.

Espressioni condizionali

Queste espressioni sono identificate dal fatto che sono sempre circondate da **[[ ]]**. La peculiarità è che possono restituire solo true o false e, al loro interno, ci sono opportuni operatori per i controlli sulle condizioni di stringhe e numeri.

Condizioni su valori aritmetici

Nome	Descrizione
-eq	equal than
-ne	not equal
-le	lower equal
-lt	lower than
-gt	greater than
-ge	greater equal

Condizioni su stringhe

Nome	Descrizione
==	uguale
!=	diverso

Nome	Descrizione
<	ordine alfabetico minore
>	ordine alfabetico maggiore
<= >=	relativamente minore e maggiore uguale
-z	controlla che la stringa abbia lunghezza 0
-n	controlla che la stringa abbia lunghezza diversa da 0

Condizioni su fules

Nome	Descrizione
file1 -nt file2	vero se il primo file è più nuovo del secondo (newer than)
file1 -ot file2	vero se il secondo file è più nuovo del secondo (older than)
-d	controlla che il file sia una directory
-e	controlla se il file esiste
-f	controlla se il file esiste ed è regolare
-h	controlla se il file esiste ed è un link simbolico
-r	controlla se il file esiste ed è leggibile
-s	controlla se il file esiste ed ha una dimensione maggiore di 0
-t	controlla se il file descriptor è aperto nel terminale
-w	controlla se il file è scrivibile
-x	controlla se il file è eseguibile
-O	controlla se il file esiste ed è stato creato dallo user

Lettura da tastiera con comando READ

Il comando **READ** permette di prendere in input da tastiera delle sequenze di caratteri e di inserirle in variabili che seguono il comando stesso. Per scegliere un numero definito di caratteri da prendere utilizziamo l'attributo **-n** come in questo esempio : **read -n 5 Riga** che quindi legge i primi 5 caratteri da tastiera (o da un qualsiasi input) e li mette nella variabile Riga.

File Descriptor

Il **File descriptor** è un'astrazione che permette l'accesso ai file aperti da un processo. Questa astrazione permette di riconoscere ogni file tramite un interno che gli viene assegnato in una **file descriptor table**. Tramite il comando **exec** possiamo aprire i file e assegnarli a un file descriptor, così da poterli utilizzare a piacimento nei nostri script.

Apertura	Comando

Apertura	Comando
Solo Lettura	exec n< PercorsoFile
Scrittura	exec n> PercorsoFile
Aggiunta in coda	exec n>> PercorsoFile
Lettura e Scrittura	exec n<> PercorsoFile

```
# esempio:
# effettuo le letture dal file mioinput.txt aprendolo in lettura
exec {FD}< /home/vittorio/mioinput.txt
while read -u ${FD} StringaLetta ;
do
echo "ho letto: ${StringaLetta}"
done
```

Una volta aperto un file dobbiamo ricordarci di chiuderlo tramite il comando molto strano, ovvero **exec n>&-** dove n è il file descriptor. Bisogna rispettari questi spazi, altrimenti il comando non funzionerà

Ridirezionamento di Stream

PAG 111