tmp

u8-s1-vm-container

Sistemi Operativi

Unità 8: Altri Argomenti

Macchine Virtuali e Container

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

1. Necessità di isolamento
2. Macchine Virtuali
3. Container
4. Cloud
5. Layer di compatibilità

Necessità di isolamento (Motivazioni per le VM)

Diamo una serie di *motivazioni* per parlare delle *macchine virtuali*

Premessa 1: Le organizzazioni *comprano macchine molto potenti*

- Una server potente costa meno di tanti server piccoli
  Premessa 2: Ogni utente/dipartimento *ha bisogno un macchina dedicata*
- Un crash in una macchina non compromette l'altra
  - Esempi: Database, Web Server, eccetera...

Conseguenza: Si vuole *dividere* una macchine potente in più macchine *meno potenti*, dandone ognuna un suo servizio

Esempio.
Un organizzazione compra una *macchina potente*, e necessità di un *server Web, FTP e mail*

- Non vuole far girare i *3 software sulla stessa macchine*
- Un problema in uno solo, *può compromettere tutto* e bloccare l'intera azienda
    - Esempio: memory leak, disco pieno, ecc...
- Una vulnerabilità di sicurezza compromette tutti e 3 i sistemi

Soluzione: il server viene diviso in tre *Macchine Virtuali*, ad ognuna un suo compito
Questa è la tecnica usata *quasi sempre* nelle aziende IT moderne:

- I servizi sono sempre in VM dedicate
- Vengono eseguiti su server potenti dotati di *Hypervisor* (vedremo che cos'è)
- I *servizi Cloud* offrono la possibilità di usare VM (vedremo)

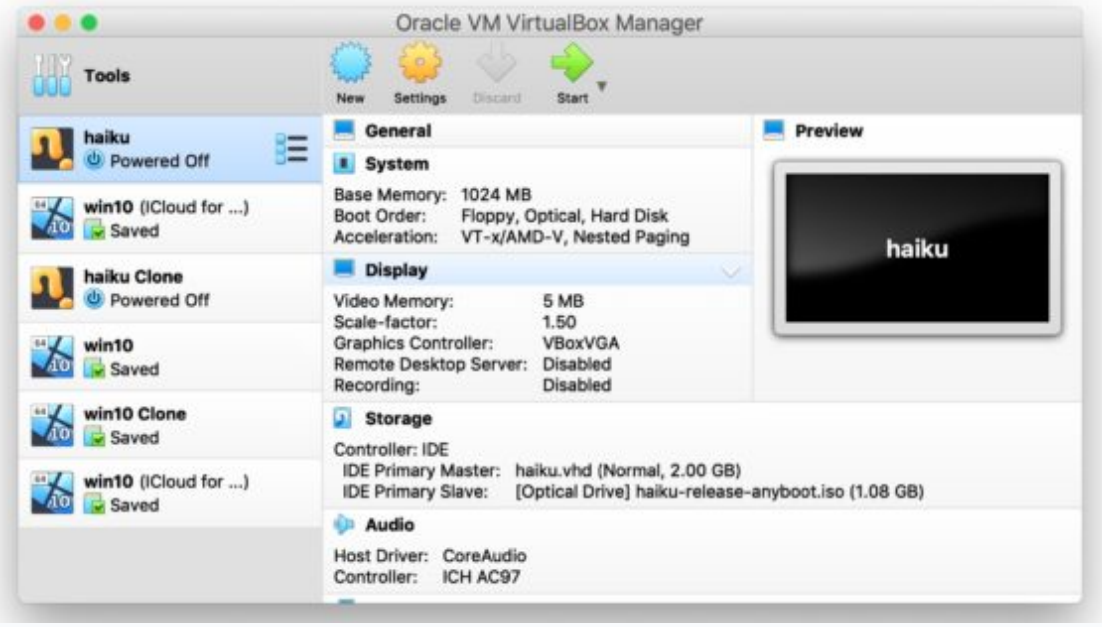
Macchine Virtuali

Definizione di VM e Hypervisor

Una *Macchina Virtuale (VM)* è un *ambiente virtuale che emula un sistema ad elaboratore* (quindi software)

Un *Hypervisor* è il software che rende possibile ciò, usando tecniche di *virtualizzazione*
Questi devono avere i seguenti requisitit teorico-tecnici:

- **Sicuri:** una VM non deve compromettere il sistema o accedere ad altre VM (ovvero non ho nessun *leak*, l'isolamento dev'essere proprio sicuro)
- **Affidabili:** una VM non deve essere *meno affidabile* di una macchina fisica (in certi casi si può raggiungere il caso in cui le *VM* sono ancora più affidabili!)
- **Efficienti:** una VM non deve essere *significativamente* meno veloce di una macchina fisica
  - Tante tecniche per arrivare a ciò (in particolare sia dal lato software che dal lato hardware)
  - La *"penalità"* può essere al più 5% ($\mathrm{VM} \ll \mathrm{HW}$).
  Per riassumere, un *Hypervisor permette di creare un sistema ad elaboratore virtuale*, con CPU, memoria e disco virtuali
- Eventualmente con accesso rete e dispositivi di I/O fisici o virtuali



Storia delle Macchine Virtuali

1. Il *concetto* nasce negli anni *'60*, nell'epoca dei mainframe
   Poco utilizzati fino ai primi anni 2000
- Gli *hypervisor erano lenti*, e non vi era grande necessità
- Si comprava *una macchina fisica per ogni servizio*
- Riassunto: esisteva, ma era rimasto sulla carta per limiti tecnologico-economici.

2. Tornano alla ribalta negli *anni 2000*

- Gli Hypervisor hanno fatto un *salto tecnologico*, diventando *efficientissimi*
  - Sono in grado di emulare l'*Hardware* come se fosse nativo: il lavoro degli Hypervisor viene facilitato proprio dall'architettura stessa
- I server sono diventati *molto potenti*, rendendo conveniente *dividerli* in più macchine di potenza intermedia
  - Abbiamo più core, memoria, in generale i computer sono molto più potenti

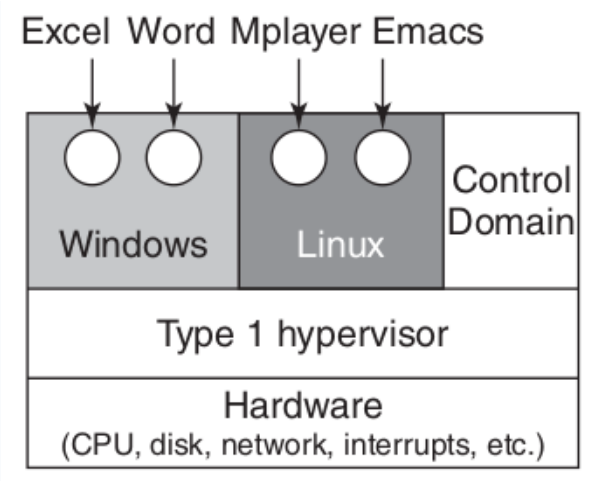
3. Oggi le *macchine virtuali* e gli *Hypervisor* sono una tecnologia pienamente matura:

- **Sicura**
- **Efficiente:** meno del 5% di penalizzazione rispetto a macchina fisica
- Tutte le aziende hanno *cluster* dedicati *a ospitare VM*
  - Un *team specializzato* gestisce il cluster e il software di virtualizzazione
  - I *team di sviluppo* (anche questo specializzato) installano i servizi su VM dedicate

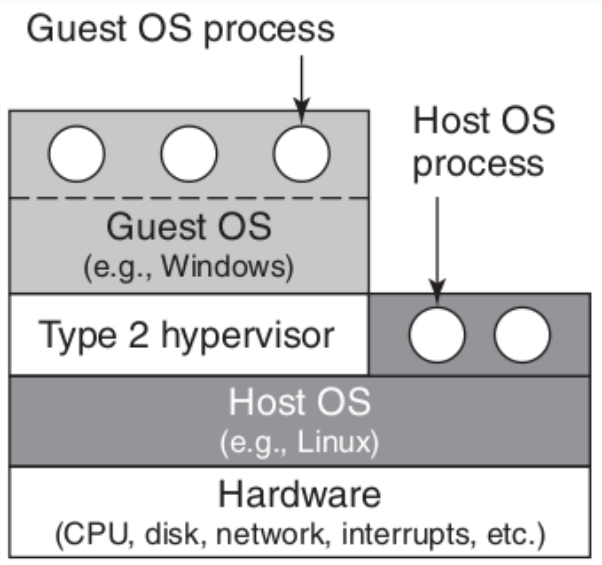
Tipologie di Hypervisor

Ci sono due tipologie di Hypervisor.

Hypervisor di Tipo 1
E' un *SO dedicato che serve solo a creare VM*
Efficienti perché hanno il controllo completo della macchina
Esempi: Xen, Microsoft Hyper-V, VMware ESXi



Hypervisor di Tipo 2
E' un *software eseguito in un normale SO* che permette di creare *Macchine Virtuale Meno efficienti* (dato che, come si vede nella figura, ho uno *"strato"* di software in più); ma ormai i SO offrono assistenza a Hypervisor di Tipo 2
Esempi: VMWare Player, Virtual Box, QEMU, Parallels



Problemi degli Hypervisor

Come visto prima, gli *Hypervisor* hanno il compito di emulare *sistemi ad elaboratori* dal lato software, mantenendo la stessa *affidabilità* ed *efficienza* dell'hardware. In particolare avremo i seguenti problemi:

- Ottimizzazione della CPU
- Ottimizzazione della Memoria
  Vedremo come verranno risolti questi problemi, sia dal lato *software* che dal lato *hardware*.

Ottimizzazione della CPU

Un Hypervisor permette di *emulare in software una CPU virtuale*, potenzialmente di *architettura diversa* rispetto alla macchina fisica

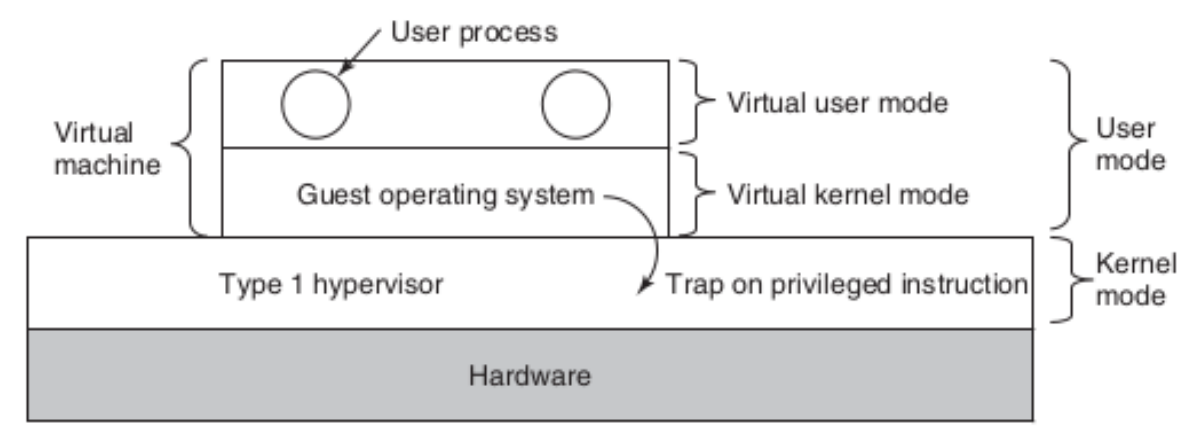
- **Esempio:** emulare ARM su CPU x86
- **Problema:** *molto lento!* Si deve implementare in software una CPU. In questo caso si ha $\mathrm{VM} \lll \mathrm{HW}$.

Solitamente ciò non avviene e si *ottimizza* l'uso della CPU

- La VM esegue le istruzioni direttamente sulla CPU fisica
  - Ovvero sono *eseguite davvero* sulla *CPU fisica*. Da questo ci sono ulteriori problematiche, tra cui la gestione delle *risorse globali*: su questo aiuterà il lato *Hardware*.
- *Necessaria cautela*, per evitare i *leak*.

Soluzione. (*Virtual Kernel Mode*)
Nei moderni Hypervisor, la VM esegue le istruzioni sulla *CPU fisica*

- Le CPU moderne permettono il *virtual kernel mode*
  - Permette di eseguire istruzioni in *kernel-mode*
  - Limitando i *privilegi*
  - Ovvero il *VM* crede di essere in *modalità kernel*, anche in realtà è limitata.
- Il kernel della VM esegue il suo codice in *virtual kernel mode*
  - Altrimenti potrebbe leggere tutta la memoria della macchina fisica! Che sarebbe pericolosissimo (esempi: leak, corruzione, accesso potenziale ai malintenzionati, eccetera...)



Ottimizzazione della Memoria

Abbiamo il problema analogo, solo che al posto della *CPU* occupiamo della *memoria*.

Un Hypervisor, se emula la CPU, *emula anche memoria in software*

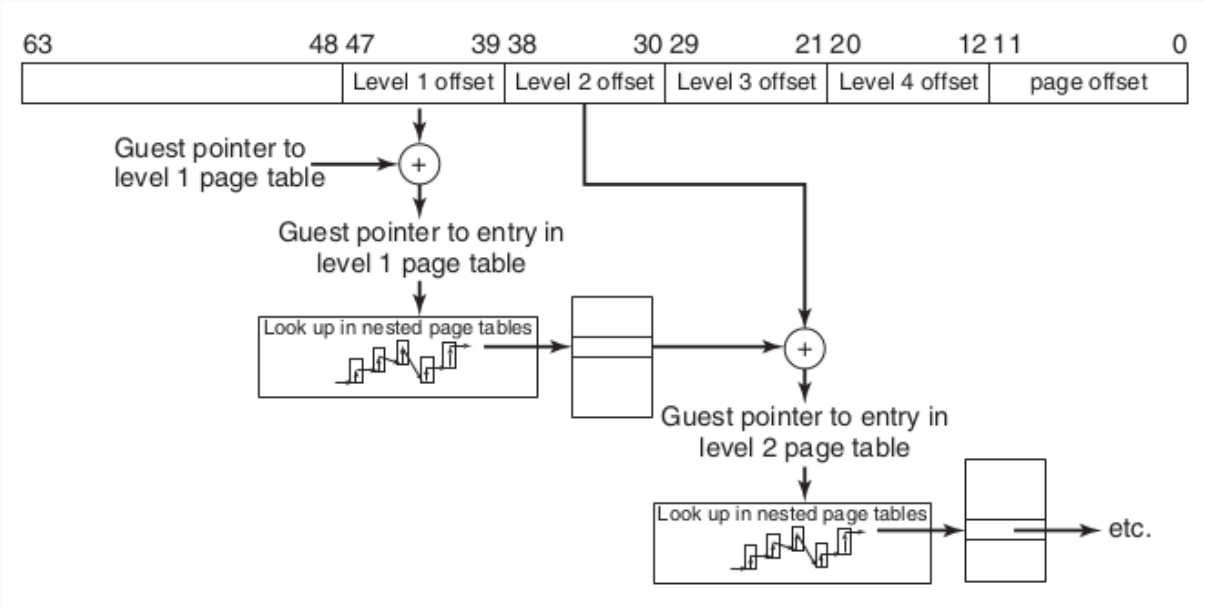
- Ogni volta che una VM accede a una *locazione di memoria*, l'*Hypervisor* esegue del codice per fornigli il risultato
- *Lentissimo*!

Gli Hypervisor moderni permettono alle VM di accedere *direttamente* a *porzioni di memoria fisica* (quindi come prima, usiamo veramente la *memoria fisica*)

- Necessari due livelli di *paginazione*
  - All'*interno della VM*: da *memoria virtuale di processo* a *memoria della VM*
  - Da *memoria della VM a memoria fisica*
- Abbiamo una composizione del tipo $(P)_{VM} : IV_{VM} \mapsto IF_{VM} \mapsto IF_{HW}$
  Serve cooperazione del sistema fisico e della CPU!

Soluzione. (*Page Table Annidate*)
Le CPU moderne supportano *Page Table annidate*

- *Primo livello:* mappa tra processo nella VM a memoria della VM ($IV_{VM} \mapsto IF_{VM}$)
- *Secondo livello:* mappa tra memoria della VM e memoria fisica ($IF_{VM} \mapsto IF_{HW}$)



Problema Attuale

In realtà ancora oggi c'è un altro problema ovvero la gestione dell'*Input/Output*: come faccio ad esporre direttamente l'*I/O* alla *VM*?

- Questo problema rappresenta la *"ultima frontiera"* degli Hypervisor
- Situazioni applicabili: Reti neurali, per le *GPU*

Tecnologie Principali per le VM

- Per giocare: **VirtualBox** o **QEMU**
- Per installare VM su un server: **Kernel-based Virtual Machine (KVM)** + **Libvirt**
- Per un cluster di VM: **OpenStack**

Altre alternative possibili, non tutte *open source* e *free* (tipo **VMWARE**)

Container

Motivazioni per Container: Limiti delle VM

1. *Allocazione statica delle risorse*
   Una VM ha allocate *staticamente* una certa quantità di risorse della macchina fisica
   Esempio: un server con 16 core e $64GB$ di RAM
- Posso fare 3 VM con 5 core e $20GB$ di RAM ognuna
- Alcune risorse vanno mantenute per il funzionamento della macchina fisica: 1 core $4GB$ di RAM
  Questo può essere *inefficiente*
- Non sempre tutte le VM hanno necessità di 5 core! Si potrebbe allocare *"dinamicamente"* le risorse, compiendo una specie di *"overbooking"* in certi casi.

2. *Eccesso Relativo di Software*
   Con l'approccio *"una VM - un'applicazione"*, su tutte le VM gira un SO, che di fatto *esegue pochi processi*
- Quelle per cui è dedicata la VM
- **Inefficiente!** proliferazione di *SO che non fanno quasi niente!*
- Esempio: $10\ \text{VM} \implies 10\ \text{Ubuntu}$

Quando voglio avviare una *nuova applicazione*, devo:

- Creare una VM
- Installare il *SO*
- Avviare la mia applicazione
  Ci metto tanto tempo (di lavoro umano)

VM vs SO

Facciamo un parallelismo filosofico tra le *VM* e i *SO* (processi).

Ricordiamoci a cosa servono le **VM**:

- *Isolare* sistemi indipendenti
- *Controllare* che essi non si danneggino a vicenda
  Ma è simile allo scopo di un *processo* in un SO. Il **SO** serve a:
- *Isolare* processi diversi (con le tecniche della *memoria virtuale*)
- *Controllare* l'accesso alle risorse tramite utenti e privilegi
  Quindi l'idea per superare il problema iniziale è quello di usare i processi, al posto delle VM.

Problema.
Purtroppo, in un *SO*, *un'applicazione problematica* può *bloccare il sistema*, se:

- Usa al 100% la CPU
- Riempie il disco o la RAM
  Un'*applicazione* potrebbe *provocare problemi* a un'altra applicazione
- Satura le risorse di I/O rete, ecc...

- Se *modifica i suoi file di configurazione* (solo se eseguita come `root`)

Soluzione: Potrei avviare un processo che ha *risorse limitate*

- Il SO si occupa di limitare l'accesso a CPU/memoria/disco (quindi di creare delle specie di gabbie)
  Sarebbe *quasi* come una VM
- Un'applicazione che gira *senza poter influenzare le altre applicazioni!*
  I sistemi Linux forniscono queste funzionalità:
- Ovvero far girare processi con *privilegi limitati*. Vediamo di dare una definizione formale ad un costrutto del genere

Definizione di Container

Un *container* è un albero di processi che gira con privilegi limitati

- Non ha accesso completo alle risorse (disco, CPU, memoria, file, etc.)
- Pensa di essere l'*unico* (inseme di) *processo*(i) *in esecuzione*
  I container sono un'illusione: illudono un processo di avere poche risorse.
  Vedere: Containers as an illusion per approfondire il discorso

I processi di un container quindi non possono:

- *Vedere gli altri processi* della macchine
- *Vedere le risorse* che *non* gli sono state *assegnate*
  Ovviamente un container *non deve poter compromettere l'intera macchina*
- Si utilizzano varie funzionalità di Linux per raggiungere questi scopi

Vedremo in particolare le funzionalità Linux per:

- Isolare File System
- Isolare risorse della CPU e della Memoria
- Isolare i Namespace

Isolamento del File System

Linux permette di avviare un processo che vede solo un sotto albero del FS
Funzionalità `chroot`: cambia *radice del FS*

Permette di evitare che un processo (e i suoi figli) legga/modifichi file fuori dall'albero

Sintassi:

```shell
chroot /path/to/new/root command
```

Ovvero il processo **command** ha la radice in **/path/to/new/root** e vede solo questo sottoalbero.

Isolamento delle Risorse CPU-Memoria

E' possibile limitare quanta CPU e memoria un processo usa.

Funzionalità cgroup: offerta dalle System Call Linux

- Permettono di limitare:
  - Uso della CPU
  - Uso della memoria
  - Velocità di I/O
  - Traffico di rete

Ovvero, permettono di evitare che un processo sovraccarichi il sistema

I **cgroup** sono relativamente nuovi. *Stabili dal 2018* con una modifica al *Kernel Linux*. Vengono usati attraverso *uno pseudo file system*

- In **/sys/fs/cgroup**

Operazioni:

1. Creazione di un gruppo di processi:
   mkdir /sys/fs/cgroup/my-group
2. Limitazione delle risorse:
   echo 50000 100000 > /sys/fs/cgroup/cpu/my-group/cpu.max (ovvero scrivo su **cpu.max** la frazione)
   Significa che i processi del gruppo, in totale, non possono usare più del 50% del tempo CPU della macchina
3. Collocazione di un processo nel gruppo:
   echo 8764 > /sys/fs/cgroup/cpu/my-group/cgroup.procs (ovvero scrivo su **cgroup.procs** il processo a cui appartiene **cgroup**)

Isolamento dei Namespace

E' possibile creare processi che *non vedono le risorse globali della macchina fisica*, ovvero:

- Quali sono gli *altri processi in esecuzione*
- Le *interfacce di rete*
- I *dispositivi di I/O*
- Gli *utenti* e *gruppi* sulla macchina

Funzionalità Namespace: offerta dalle System Call Linux

- Vedi comandi **unshare** e **nsenter**

Container Engine

Queste funzioni del SO appena elencate sono *potenti*, ma *poco usabili*:

- Per usarle, *necessario conoscerle a fondo*
- *Errori nell'utilizzo* possono compromettere il sistema
- Non c'è sicurezza by default:
  - Necessario *togliere* privilegi ai processi
  Quindi devo fare tutto a mano... qual è la soluzione?

Esistono dei software che si chiamano *Container Engine* (ovvero dei tool) che permettono di usare *in maniera semplice queste funzionalità*

- *Avviare container*: gruppi di processi isolati
- *Monitorarne* il funzionamento

Offrono comandi/API semplici per creare container. Popolari:

- **Linux Containers (LXC):** tra i primi a nascere nel 2008
- **Docker:** Nato nel 2013. Standard *de facto*

Principio di funzionamento: *eseguono processi con risorse limitate, che vivono in un file system limitato*

- Di *default*, i container hanno *privilegi minimi*
- Possibile configurarli per avere maggiori privilegi: e.g., accedere a porzioni del FS

Docker

Docker: Definizione

Si può installare *su ogni macchina Linux*

- Disponibile anche su MacOS e Windows (ma implementato tramite una VM)

Permette di avviare container a partire da una *Immagine*:

- E' un File System che *contiene il programma da eseguire*
- Ed *eventuali dipendenze*: librerie condivise, altri programmi, file di configurazione

La componente interna di Docker che permette di eseguire i container si chiama `containerd`

Docker: container e immagine e hub

Un *Container è una Immagine* in esecuzione:

- Un insieme di processi che può operare solo sui file presenti nell'immagine
- I file dell'immagine vengono copiati
- I processi possono creare nuovi file o modificare quelli esistenti
- Non può accedere ai file della macchina fisica
- I figli condividono la stessa *"gabbia"*

Esiste una libreria di immagini pre-costruite su **Docker Hub** (https://hub.docker.com/)

- Ognuna contiene un software installato con le sue dipendenze
- Può essere scaricata ed eseguita, creando un container
- Ogni immagine ha una versione identificata da un *tag*
  - *latest* indentifica l'ultima versione

E' anche possible *creare la propria immagine* col proprio software

Comandi di Docker

- **`docker pull <immagine>`**: scarica un **`immagine`** da Docker Hub
- **`docker ps`**: mostra i container in esecuzione
- **`docker run --name <nome> <immagine>`**: esegue un container da **`immagine`** e gli assegna il **`<nome>`**
  - Argomento **`-v pathLocale:pathContainer`**: permette al container di accedere **`pathLocale`** che viene montato in **`pathContainer`**
  - Argomenti **`--cpus <n>`**, **`--memory=<n><s>`**: limita le risorse dell'immagine
  - Argomento **`-d`**: processo in background (diventa un *demone*)
  - Argomento **`-e VAR=VAL`**: specifica delle variabili d'ambiente (come password, eccetera...)
- **`docker stop <nome>`**: termina il container identificato da **`nome`**
- **`docker logs <nome>`**: vedere l'output dell'immagine in esecuzione
- **`docker inspect <nome>`**: per vedere informazioni sull'immagine
  Molti altri comandi...

Necessari permessi di *superuser*, da fornire con **`sudo`**

Docker e file montati

Di default, un container *non* può accedere ai file della macchina fisica, ma solo a una copia di quelli dell'immagine

L'opzione **`-v pathLocale:pathContainer`** permette al container di accedere a **`pathLocale`**, che viene *montato* in **`pathContainer`** all'interno del FS del container

Esempio:

```shell
docker run --name nome \
    -v /home/martino:/opt/home-di-martino \
    immagine
```

Il container **nome** può accedere al path fisico **/home/martino** tramite il path **/opt/home-di-martino**

Docker e Rete

Ogni container ha un *indirizzo IP* in una *rete virtuale* che collega tutti i container

- Possibile comunicazione tra *container*
- Possibile comunicazione tra *macchina fisica e container* (esempio: Server Web)
- Possibile comunicazione tra *container e Internet* tramite Default Gateway virtuale

Docker e Risorse

Limitazione di CPU: **docker run --cpus 2 <immagine>**
Limitazione di memoria: **docker run --memory=512m <immagine>**

Docker: Esempio

Creazione di container per eseguire il DBMS *PostgreSQL*

- Un DBMS relazionale
- Un processo del database deve essere in esecuzione
- Vi si accede tramite rete e un protocollo dedicato

Scaricamento dell'immagine:
docker pull postgres

Avviamento del container:

```shell
docker run -d \
    --name some-postgres \
    -e POSTGRES_PASSWORD=mysecretpassword \
    -e PGDATA=/var/lib/postgresql/data/pgdata \
    -v /home/martino/db:/var/lib/postgresql/data \
    postgres
```

Opzioni usate:

- `-d`: fai partire il processo in background
- `-e VAR=VAL`: specifica variabili d'ambiente visibili nel container
  - Usato per password del DB e per specificare dove esso salva i dati
- `-v /home/martino/db:/var/lib/postgresql/data`: i dati sono salvati sulla macchina fisica in `/home/martino/db` ma nel container al path `/var/lib/postgresql/data`

Privilegi del container:
Il container `some-postgres` esegue l'immagine `postgres`.
Ha accesso alle risorse fisiche di:

- CPU e memoria senza limiti
- File System: solo `/home/martino/db`
- Rete: ha un indirizzo IP. Il server si mette in ascolto sulla porta di default 5432

Monitoraggio:
Se tutto è andato a buon fine, il container è in esecuzione. Si osserva con:

```shell
$ docker ps
CONTAINER ID  IMAGE    COMMAND                  CREATED       STATUS
PORTS    NAMES
c6320fa9eb9b  postgres "docker-entrypoint.s..." 50 seconds ago Up 49
seconds 5432/tcp some-postgres
```

Si può ottenere il suo IP con

```shell
$ docker inspect some-postgres
...
"IPAddress": "172.17.0.2",
...
```

Sulla macchina fisica:
I processi di `postgres` sono normali processi in esecuzione, ma con privilegi **molto limitati**

SHELL

```
$ ps fax
 443964 ?        Sl     0:05 /usr/bin/containerd-shim-runc-v2 -namespace moby ....
 443985 ?        Ss     0:01  \_ postgres
 444080 ?        Ss     0:00      \_ postgres: checkpointer
 444081 ?        Ss     0:00      \_ postgres: background writer
 444083 ?        Ss     0:00      \_ postgres: walwriter
 444085 ?        Ss     0:00      \_ postgres: logical replication launcher
...
```

I file dove il DB salva i suoi dati sono in

SHELL

```
$ sudo tree /home/martino/db/ -L 2
/home/martino/db/
└── pgdata
    ├── base
    ├── global
    ├── pg_commit_ts
...
```

Utilizzo:
Il DB si può usare installando il client **psql**, col comando:

SHELL

```
$ PGPASSWORD=mysecretpassword psql -U postgres -h 172.17.0.2
psql (12.12 (Ubuntu 12.12-0ubuntu0.20.04.1), server 15.1 (Debian 15.1-
1.pgdg110+1))
WARNING: psql major version 12, server major version 15.
         Some psql features might not work.
Type "help" for help.

postgres=#
```

Il DB salva i dati nella cartella fisica: **/home/martino/db**
Con 3 semplici comandi si è installato PostgreSQL!

Utilizzo odierno

L'utilizzo di *container* sta *prendendo il posto* dell'utilizzo delle *VM*.

- Più scalabile
- Costringe a separare codice da dati
- Ho lo stesso *livello di isolamento*, con *meno fatica*!

Nelle grandi aziende, si utilizzano *cluster di nodi* che eseguono container.
Esistono *software di orchestrazione di container* basati su Docker:

- **Kubernetes**: il più usato. Open-Source
- **OpenShift** e **OKD**: proprietari di *Red Hat*

Tecnologie Cloud

Scenario

Le tecnologie di VM e container permettono a un'azienda di *collocare i propri servizi in qualsiasi luogo del mondo* (ovvero è tutto *virtualizzato*)

Per molte aziende è conveniente *affittare una VM* da un'azienda specializzata, anziché comprare server fisici

- Avere server farm è *costoso*: necessario raffreddamento e sorveglianza
  - Economia di scala con data center grandi
  - Vedi: requisiti per *costruire un Data Center di livello 4, standard EN50600*
- Il personale specializzato è *poco e costa molto*!
  - Vedi: per costruire un data center, bisogna avere *più team specializzati* per progettare tutto, poi bisogna costruire il data center, poi un team per mantenere tutto, eccetera...
- Malfunzionamenti possono provocare *gravi danni economici*!

Cloud Provider

Conseguenza: sempre più spesso le aziende comprano servizi da *Cloud Provider*
Tra i più popolari cloud provider:

- **Amazon Web Services**
- **Google Cloud**
- **Microsoft Azure**
- **Aruba** (in Italia)
- Poi molti altri! (tipo dei provider ad-hoc per la *Pubblica Amministrazione*)

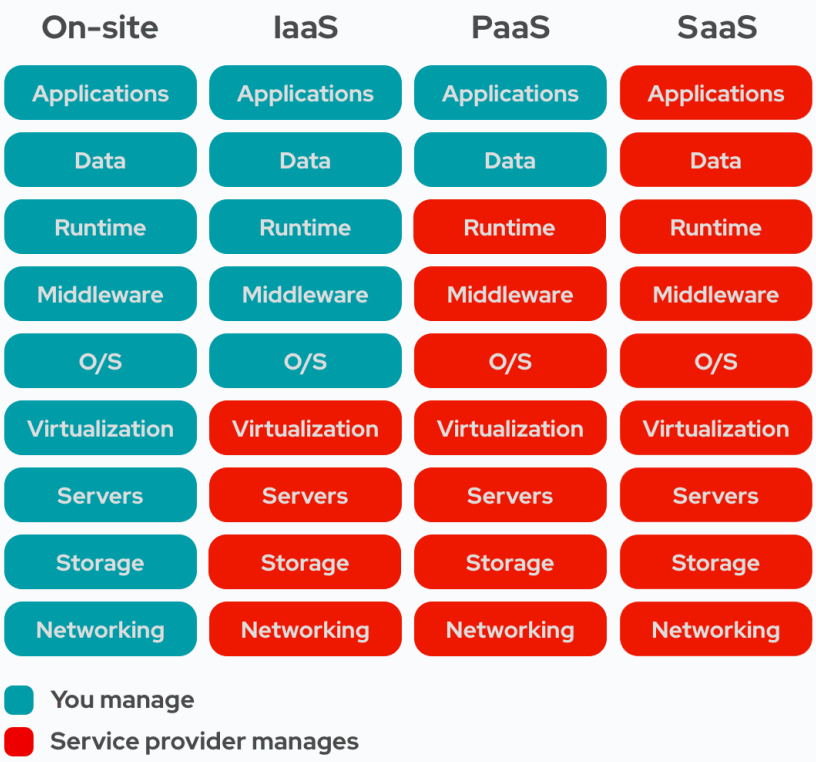
Pro: Il costo immediato è molto basso
Contro: Il costo a lungo termine è significativamente alto (in alcuni anni copri il costo immediato)

Vedremo i *pro/contro* in dettaglio ulteriormente.

Servizi offerti dai Cloud Provider

Diverse tipologie di servizi offerti dai cloud provider

- **IAAS (Infrastructure As A Service)**: possibilità di creare e utilizzare *VM* o *container*
- **PAAS (Platform As A Service)**: il cloud provider offre una *piattaforma di sviluppo*. L'utente scrive solo l'applicazione
  - **Esempio 1:** Database SQL remoto in Cloud
  - **Esempio 2:** servizio di hosting per siti web dinamici: supporto a hosting HTML, esecuzione server-side di PHP e SQL
- **SAAS (Software As A Service)**: l'utente/azienda compra una *subscription* a un *servizio completo*
  - **Esempio:** un'azienda compra un abbonamento a Microsoft Teams



Prospettive Odierne delle Tecnologie Cloud

Sempre più spesso aziende ed enti pubblici fanno ricorso a Cloud Provider per IAAS/PAAS/SAAS; tuttavia ancora oggi queste tecnologie sono ancorai in *fase di discussione*.

Vantaggi:

- Minore costo iniziale
- Maggiore affidabilità (infatti, *Google*, *Amazon* sono affidabili)

Svantaggi:

- *Vendor Lock-in* (ovvero praticamente sono *"legato"* all'ente fornitore, ho bisogno di tutele legali)
- Perdita di *Know How* (in particolare della *sovranità dei dati*: ripercussione particolare sui *problemi geopolitici*, come conflitti internazionali, leggi che tutelano dati (GDPR), eccetera...)
- Costo elevato nel lungo termine (in *2 anni* avrei coperto il costo immediato per *un server*)

Layer di compatibilità

Torniamo indietro con queste tecnologie di virtualizzazione. In particolare siamo sul *livello utente*.

VM e Software

Le VM permettono di avere un sistema ad elaboratore *virtuale*

- Su cui installare *un SO a piacere*
- Esempio: VM con Linux su PC Windows

Spesso per l'utente, la VM serve solo a usare un *software* scritto per un SO diverso

- Esso può girare solo su $(Architettura, SO)$ *per cui é stato compilato*
- Non é possibile usare su altro SO, anche se stessa $Architettura$. *Le System Call sono diverse!*
  Tuttavia le *macchine virtuali* sono un po' *"esaggerate"* per questa casistica. Vediamo un'altra tecnica di virtualizzazione, ovvero i *Layer di Compatibilità*.

Definizione di Layer di Compatibilità

Un *Layer di compatibilità* é un *software* che permette di eseguire un programma scritto per un SO diverso: sostanzialmente implementa delle *System Call*

- Ma compilato su stessa $Architettura$
  Implementa le **System Call** di un altro SO, tramite quelle del SO corrente.
- **Esempio:** Win32 `ReadFile` ⇒ POSIX `read`

Funzionamento *complesso* e problematico

- Esistono meccanismi *non-mappabili*
- Gestione di I/O complessa: dipende da SO e da driver
- Le System Call diverse hanno una semantica diversa
- Molto difficile, comunque possibile

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

Layer di Compatibilità a Livello API e ABI

Layer di compatibilità a livello Application Programming Interface (API): Richiede ricompilazione del software

- Basato su una libreria software implementa le System Call di un SO tramite *quelle di un altro*
- Si fa una specie di *"massaggio degli argomenti"* delle System Call

Layer di compatibilità a livello Application Binary Interface (ABI): *NON* richiede ricompilazione del software

- Il programma usa le System Call del proprio SO. Il Layer *le intercetta* e *invoca quelle del SO corrente*

Tecnologie Principali

1. **CYGWIN**
   Permette di usare programmi che usano System Call **POSIX** su Windows
- A livello **API**
- Richiede ricompilazione

Nota: POSIX $\neq$ Linux

Cygwin é semplicemente un'altro ambiente per compilare ed eseguire programmi POSIX che usano le System Call e Librerie POSIX

2. **WINE**
   Permette di usare programmi per **Windows** su Linux e MacOS
- A livello **ABI**
- Non richiede ricompilazione
  - Non sarebbe possibile con software closed-source Windows

Molto matura e usata:

- Funzionano anche programmi con interfaccia grafica
- Alcuni programmi complessi invece non si possono usare

3. **WSL 1**
   Permette di usare programmi per **Linux** su Windows
- A livello **ABI**
- Non richiede ricompilazione

Nota. (*WSL 2*)
Invece la *WSL 2* è una VM minimale con un vero kernel

- **NON** é un Layer di compatibilità
- Più flessibile, ma più lenta

Domande

Quale tra questi non è una motivazione per l'uso di VM?
• Maggiore sicurezza **• Maggiore affidabilità** **• Maggiore velocità della memoria**
Risposta: *Maggiore velocità della memoria*

Una macchina fisica sta eseguendo una VM. Quanti kernel sono in esecuzione?
• Nessuno **• 1** **• 2** **• 3**
Risposta: *2*

Una VM può usare direttamente la memoria fisica della macchina?
• Mai **• Sempre** **• Se la CPU lo permette**
Risposta: *Mai*

Una macchina fisica sta eseguendo una container. Quanti kernel sono in esecuzione?
• Nessuno **• 1** **• 2** **• 3**
Risposta: *1*

Cosa è un container?
• Un FS isolato **• Un namespace** **• Un gruppo di processi con privilegi limitati**
Risposta: *Un gruppo di processi con privilegi limitati*

Un container può accedere al File System della macchina ospitante?
• Sempre **• Mai** **• Dipende da come è stato creato**
Risposta: *Dipende da come è stato creato*

Quale tra questi non è un servizio offerto dai Cloud Provider?
• Esecuzione di VM **• Abbonamento a database remoto** **• Licenze di software da eseguire su PC**
Risposta: *Licenze di software da eseguire su PC*

Quali delle seguenti affermazioni é vera? Un layer di compatibilità:
• é una VM **• é un insieme di container**
• permette di esguire programmi compilati su un'architettura diversa
• permette di esguire programmi compilati su un SO diverso
Risposta: *permette di eseguire programmi compilati su un SO diverso*

u8-s2-socket

Sistemi Operativi

Unità 8: Altri Argomenti

Rete e socket in Linux

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

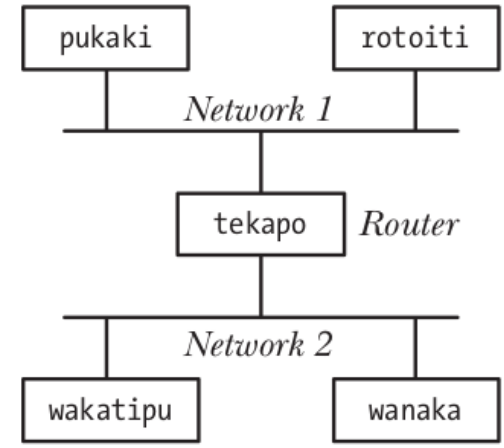
1. Lo stack di rete TCP/IP in Linux
2. I Socket
3. Funzioni e System Call per i Socket
4. Comandi per Networking in Linux

Lo stack di rete TCP/IP in Linux

Definizione di Internet

Internet è un l'*insieme di nodi e apparati di rete* che permettono una *comunicazione mondiale*

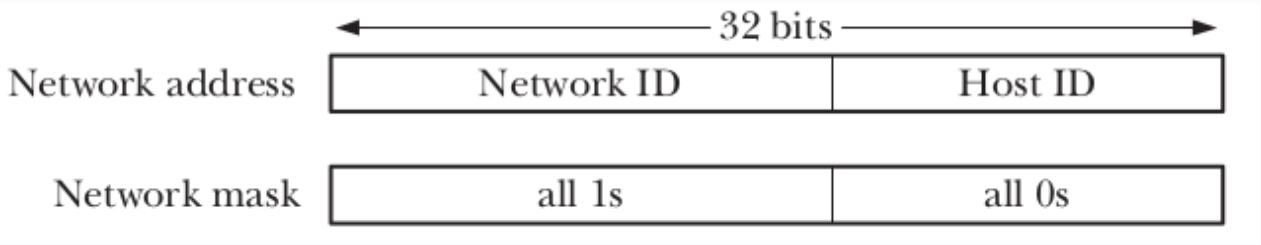
- Internet è l'unione di tante *Network*
- Collegate tramite *Router* (serve per *distinguere traffici di rete*)
- Ogni nodo è identificato da un *Indirizzo IP*



Indirizzi IP

Un indirizzo IP identifica univocamente un nodo in Internet

- Numero su 32 bit. Sono *pochi!* Con dei calcoli si trova che $2^{32} \ll 7 \cdot 10^9$
- Composto da una *parte di network e una di host*
  - La *netmask* delimita le *due parti*
  - Necessario per la trasmissione di pacchetti tramite Ethernet
    - L'indirizzo IP $127.0.0.1$ identifica per convenzione il *Local Host*
  - Ovvero serve per mandare un pacchetto a se stesso



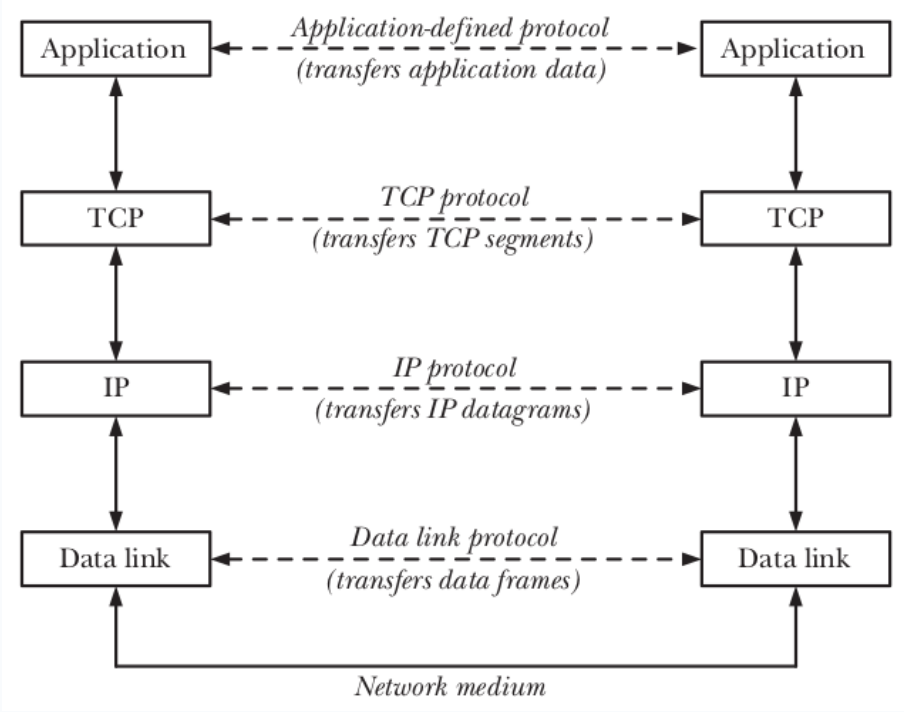
Protocolli Rete

I protocolli formano una *Pila*:

- Il livello N usa i servizi del livello $N-1$
- Li migliora e li offre al livello $N+1$
- Il livello N parla col suo omologo su un altro nodo

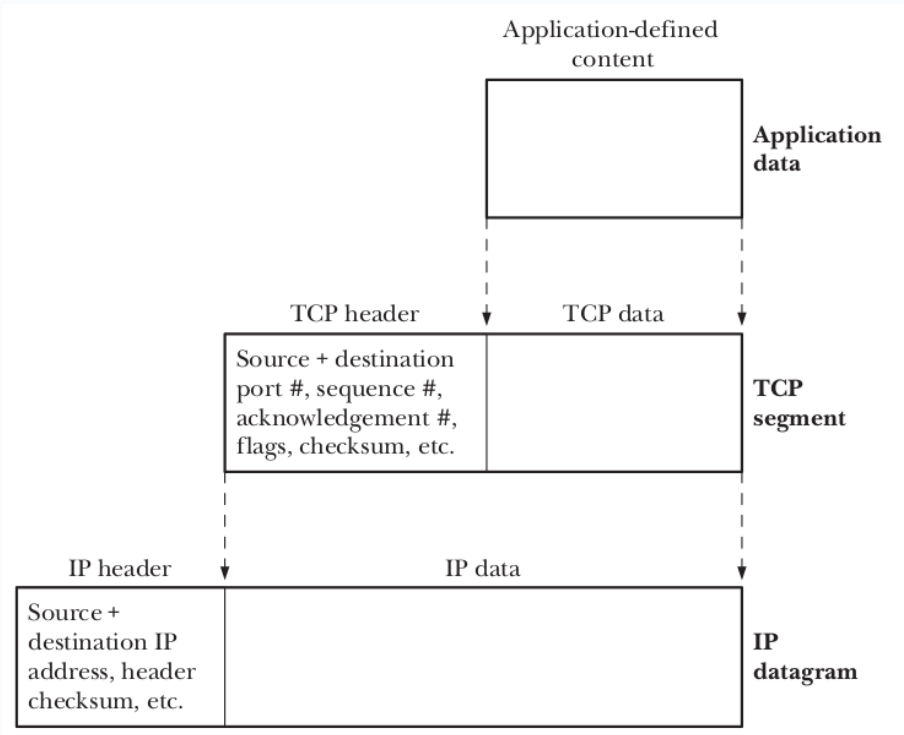
In particolare i livelli sono i seguenti:

- *L1* - Dati binari (funzionalità semplice)
- *L2* - Messaggi (ethernet, wifi, ...)
- *L3* - Identificatori (IP)
- *L4* - Identificativi dei processi, porte
- ...
- *L7* - Applicazioni (Esempio: Server Web)



I protocolli vengono *inscatolati* uno dentro l'altro:

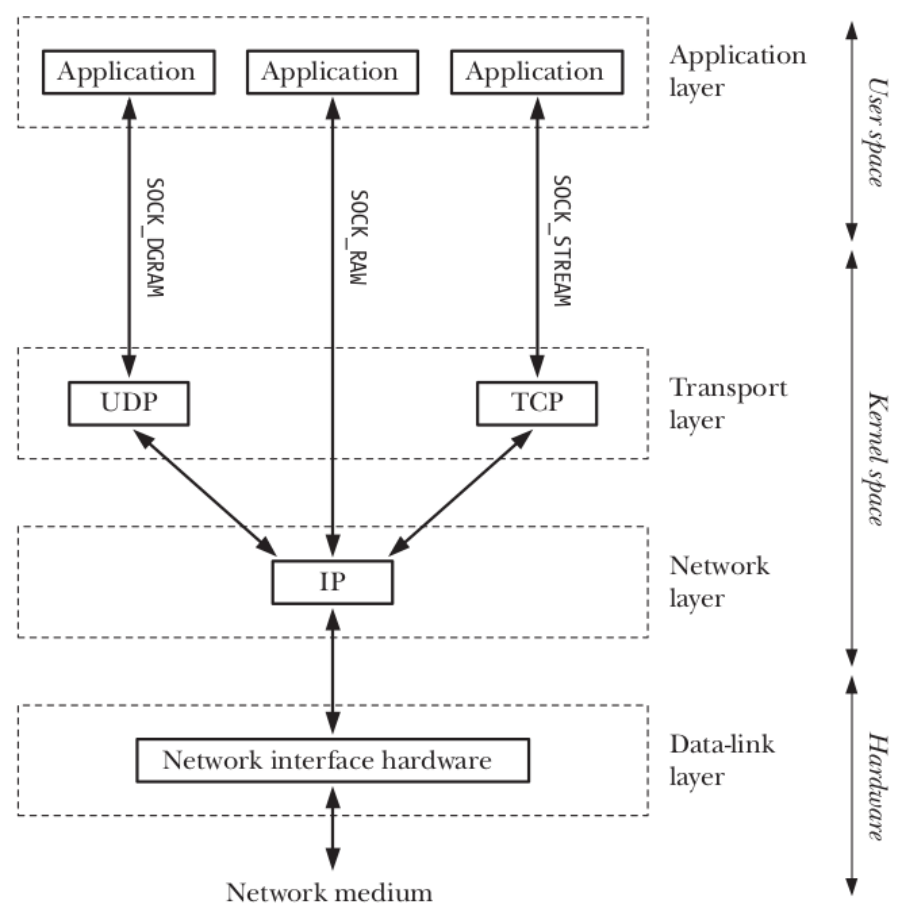
- Un frame **Ethernet** trasporta un pacchetto **IP**
- Un pacchetto **IP** trasporta un segmento **TCP**
- Un segmento **TCP** contiene i dati dell'**applicazione**



Le applicazioni in Linux possono usare i servizi di:

- *TCP* per avviare una comunicazione orientata al flusso (`SOCK_DGRAM`)
- *UDP* per mandare datagrammi (`SOCK_STREAM`)
- *Pacchetti IP generici* (`SOCK_RAW`)
  Non li implementeremo, impareremo ad usare uno dei servizi

Il *kernel* implementa i moduli TCP, UDP, IP
Offre delle *System Call* per poterne utilizzare i servizi

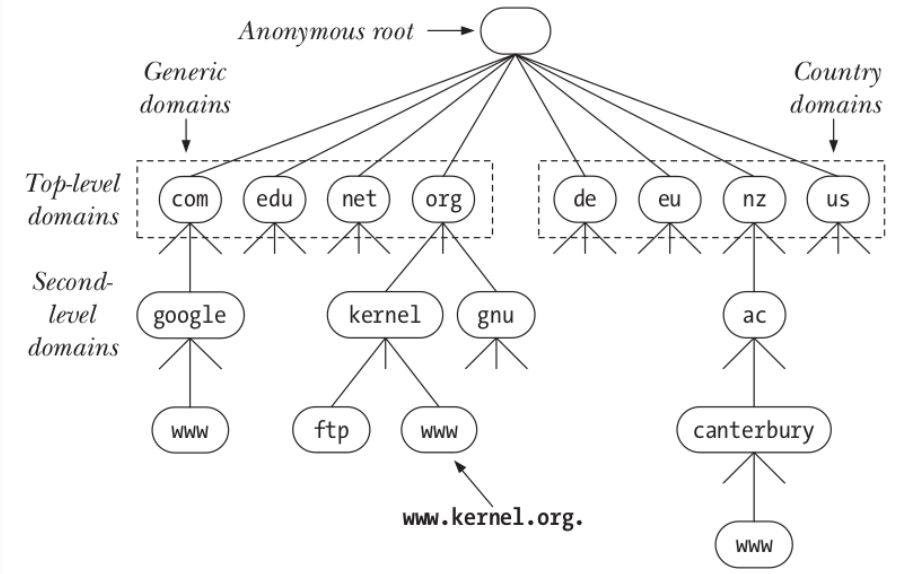


Domain Name System (DNS)

Il *Domain Name System (DNS)* è un sistema di directory distribuito e gerarchico

- Serve per identificare nodi di Internet tramite un *nome di dominio* anziché un indirizzo IP
- Permette la *conversione tra indirizzi IP e nomi di dominio*
  - Una specie di *"elenco telefonico dell'internet"*

Linux offre funzioni per usare il DNS in maniera semplice



I Socket

Definizione di Socket

I *Socket* uno strumento di *Inter-Process Communication* per scambiare dati tra diversi *nodi di rete*

Utilizzo simile alle *pipe* e alle *FIFO*

- Identificati da un *file descriptor*
- Vi si accede con le System Call **read** e **write**

A differenza di *pipe* e alle *FIFO*

- Connettono *nodi diversi*
- Vengono *creati in maniera diversa* con System Call dedicate
- Questo è l'*unico modo* per comunicare tra *sistemi operativi*!

Tipologie di Socket

Esistono quattro tipologie di socket:

- **Stream Socket**: permettono comunicazione tramite *TCP* (*String byte*)
- **Datagram Socket**: permettono comunicazione tramite *UDP* (*Messaggi*)
- **Raw Socket**: permettono comunicazione tramite *pacchetti grezzi IP*
- **UNIX**: permettono comunicazione tra processi *di uno stesso nodo*

Sempre basati su modello **client/server**

Modello Client/Server

Per implementare il *modello client/server* abbiamo i *socket passivi* e *attivi*.

Un **Passive Socket** *aspetta connessioni* in arrivo

- Implementa un *server*

Un **Active Socket** è effettivamente connesso a un altro nodo

- Permette lo *scambio di dati*
- Usato da un *client* per comunicare col server
- Usato anche dal *server*, *dopo* aver accettato una nuova connessione

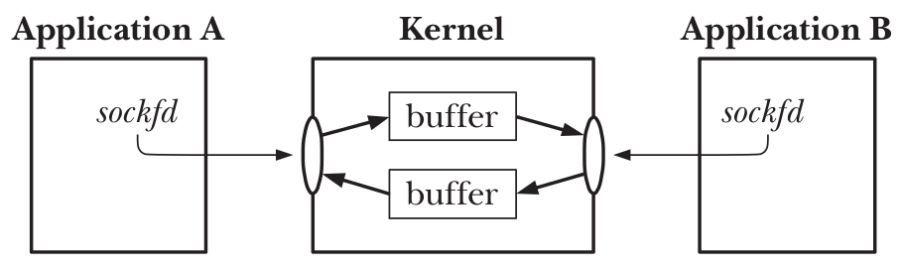
UNIX Socket

Comunicazione tra processi di uno stesso nodo

- Concettualmente *molto simili* a una *pipe* o *FIFO*

Differenza

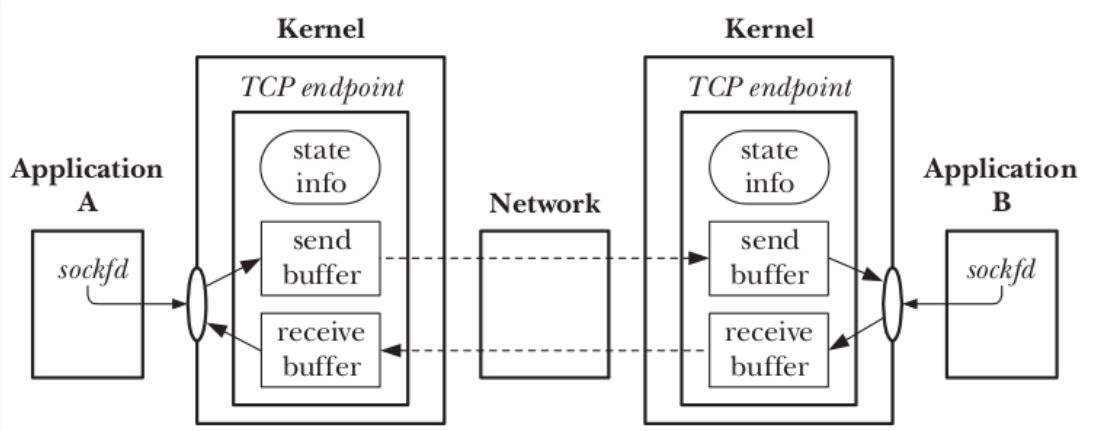
- Usano modello *client/server*
- Un *server* si mette in ascolto
- Un *client* contatta il server e inizia la comunicazione
- Sono *peer-to-peer*



Stream Socket

Comunicazione tramite *TCP*

- Servizio orientato alla *connessione*
- Client e server comunicano tramite un *flusso di byte*
- Molto affidabile! Circa $1/1000$ dei pacchetti vengono persi, che vengono comunque ritrasmessi
  Simile a una *pipe* o *FIFO* tra *nodi diversi*



Datagram Socket

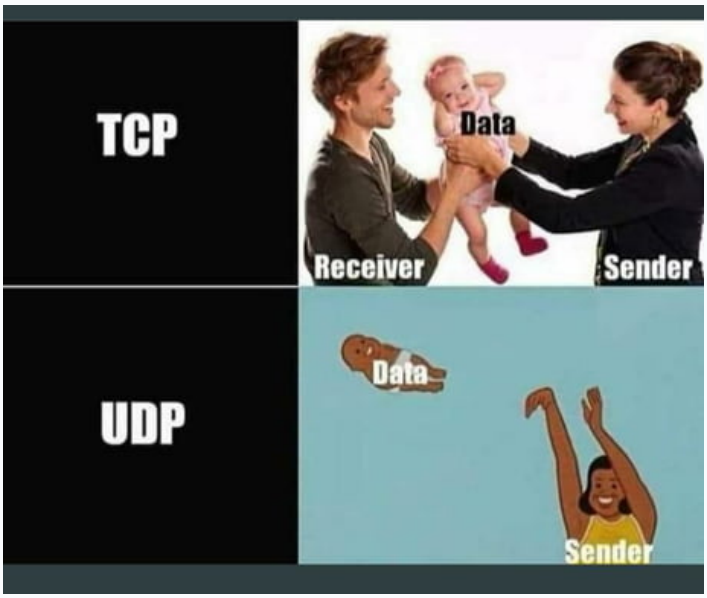
Comunicazione tramite UDP

- Client e server si scambiano *messaggi*
- Servizio non affidabile
  - Possibile perdita di pacchetti che non vengono ritrasmessi

Differenze tra Socket

Differenze:

- Datagram Socket:
  - Orientato ai *messaggi*
  - Non affidabile
- Stream Socket e UNIX socket:
  - Orientato allo *stream*
  - Affidabile



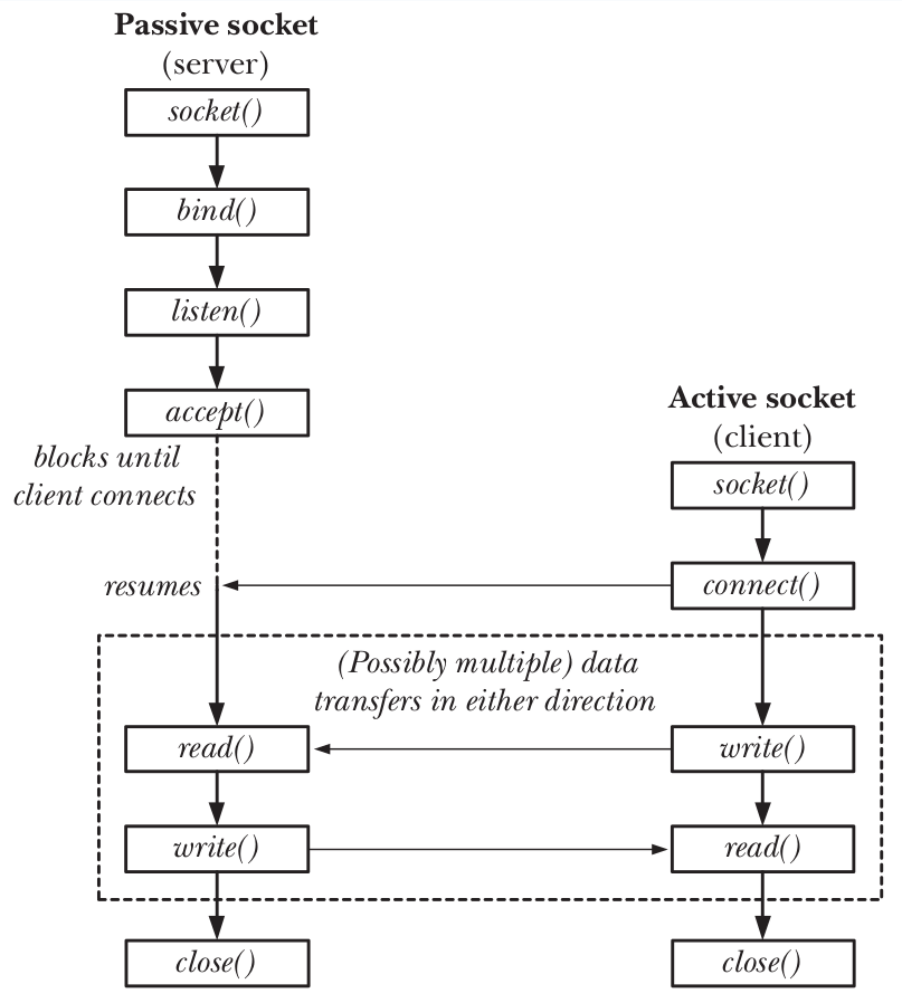
Funzioni e System Call per i Socket

I sistemi Linux/POSIX mettono a disposizione System Call per usare i socket

- Ogni socket è identificato da un File Descriptor
- Similmente ai file aperti o *FIFO* aperti, o *pipe*.
- Si effettua I/O usando le System Call **read** e **write**
  - Tranne che per i *Datagram Socket* (si usano **sendto** e **recvfrom**)
  Per creare un socket, si usano *System Call dedicate*
- Bisogna specificare *indirizzi IP e porte*
- *Attendere* che il kernel stabilisca la connessione

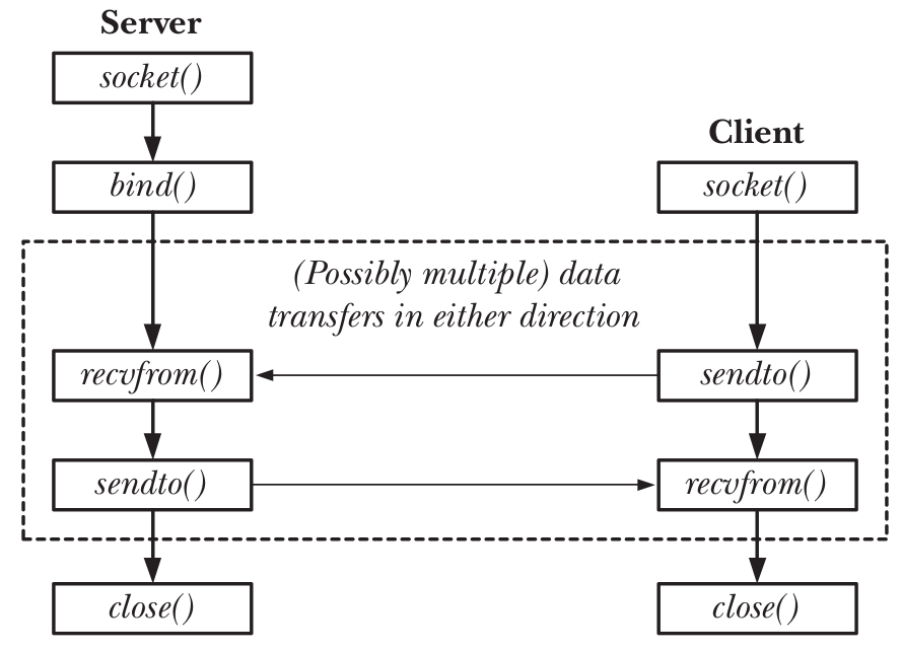
Stream Socket e UNIX Socket

- *Client* usa: **socket** e **connect**
- *Server* usa **socket**, **bind**, **listen** e **accept**
- *Entrambi* usano **read** **write** e **close**



Datagram Socket

- *Client* usa: **socket**
- *Server* usa **socket**, **bind**
- *Entrambi* usano **sendto** e **recvfrom** e **close**



Funzioni e System Call per i Socket

Noi vediamo in dettaglio *solo gli Stream Socket*

- Che utilizzano *TCP*
- Sono *affidabili*
- Orientati alla *connessione*
- Client e server comunicano tramite un stream di byte
- Semantica simile a *pipe*, ma *bidirezionale*
  Nelle prossime slide, sono presentate le System Call, ipotizzando di creare uno *Stream Socket*

Creazione di un Socket

```c
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Crea un socket. Gli argomenti **domain** e **protocol** ne *specificano la natura*.

- Argomento **protocol**, per noi sempre **0**

Ritorna il File Desciptor, se no -1.

Esempi:

- **Stream Socket** `int fd = socket(AF_INET, SOCK_STREAM, 0)`
- **UNIX Socket** `int fd = socket(AF_UNIX, SOCK_STREAM, 0);`
- **Datagram Socket** `int fd = socket(AF_INET, SOCK_DGRAM, 0))`

Struttura `sockaddr`

Prima di vedere le *System Call* nello specifico, vediamo una struttura dati importante per rappresentare *indirizzi IP* e *porte*.

```c
struct sockaddr {
    sa_family_t sa_family;  /* Address family (AF_* constant) */
    char sa_data[14];       /* Socket address (size varies
                               according to socket domain) */
};
```

La **`struct sockaddr`** contiene un indirizzo IP, una porta o entrambi.
Deve essere *generica*: supportare protocolli potenzialmente diversi da suite TCP/IP

Il campo **`sa_data`** deve contenere gli indirizzi e le porte
Quando si usano socket con TCP/IP si utilizza la **`struct sockaddr_in`**

Viene usata in tutte le System Call che richiedono una **`struct sockaddr`**.

- Le System Call *solo generiche*
- Se noi usiamo TCP/IP, usiamo **`struct sockaddr_in`**

Lato Client

```c
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr * addr, socklen_t addrlen );
```

Rende il socket **`sockfd`** *attivo* e si connette a indirizzo IP e porta specificati in **`addr`** e **`addrlen`**
Ritorna 0 in caso di successo, se no -1

La **`struct sockaddr`** contiene un indirizzo IP, una porta o entrambi

- Entrambi in questo caso

La **`connect`** è *bloccante* finché non viene stabilita la connessione (TCP).

Lato Server

1. *Trasformazione in passivo*

```c
#include <sys/socket.h>
int bind(int sockfd , const struct sockaddr * addr , socklen_t addrlen );
```

Rende il socket **sockfd** passivo, ovvero *lo mette in ascolto sulla porta specificata* in **addr** e **addrlen**
Ritorna 0 in caso di successo, se no -1

La **addr** punta a una **struct sockaddr**, che sarà sempre di fatto una **struct sockaddr_in**:

- Contenente solo una porta in questo caso

2. *Attivazione di un socket passivo*

```c
#include <sys/socket.h>
int listen(int sockfd , int backlog);
```

Dopo che un socket **sockfd** è stato specificato come passivo (con **bind**), la **listen** lo *mette effettivamente in ascolto sulla porta* specificata.
Il parametro **backlog** determina *quante connessioni* in attesa *possono accodarsi* prima di essere servite. Di solito è un numero piccolo, tipo 5
Ritorna 0 in caso di successo, se no -1

3. *Accettazione di connessioni*

```c
#include <sys/socket.h>
int accept(int sockfd , struct sockaddr * addr , socklen_t * addrlen);
```

Attende che una connessione arrivi al socket passivo **sockfd**

- *Bloccante* finchè non arriva una connessione

Nel momento in cui arriva una nuova connessione:

- La funzione ritorna
- Il valore di ritorno è un *nuovo socket attivo* (ovvero l'indirizzo IP)
- In **addr** (e **addrlen**) è specificato l'indirizzo del client

Tipicamente messo in un *ciclo while infinito*.

Lettura e Scrittura su Socket

Un socket attivo viene creato:

- Direttamente da un client dopo che si è connettersi
- In un server, ogni volta che la `accept` ritorna, e permette la comunicazione con un client

Un socket è *bidirezionale*. In caso di *Stream Socket*:

- Si effettua I/O con `read` e `write`, o volendo con le funzioni specifiche per i socket `send` e `recv`
- Un socket viene chiuso tramite la `close` (col pacchetto *FIN*, *SYN* nel caso di connessione stabilita)

Conversione di Indirizzi IP

Necessarie funzioni per convertire indirizzi IP in stringa e in formato binario su $4B = 32bit$

```c
char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
```

IP in formato stringa specificato come `char *`
IP in formato binario specificato come `struct in_addr`

- Tipicamente si usa:

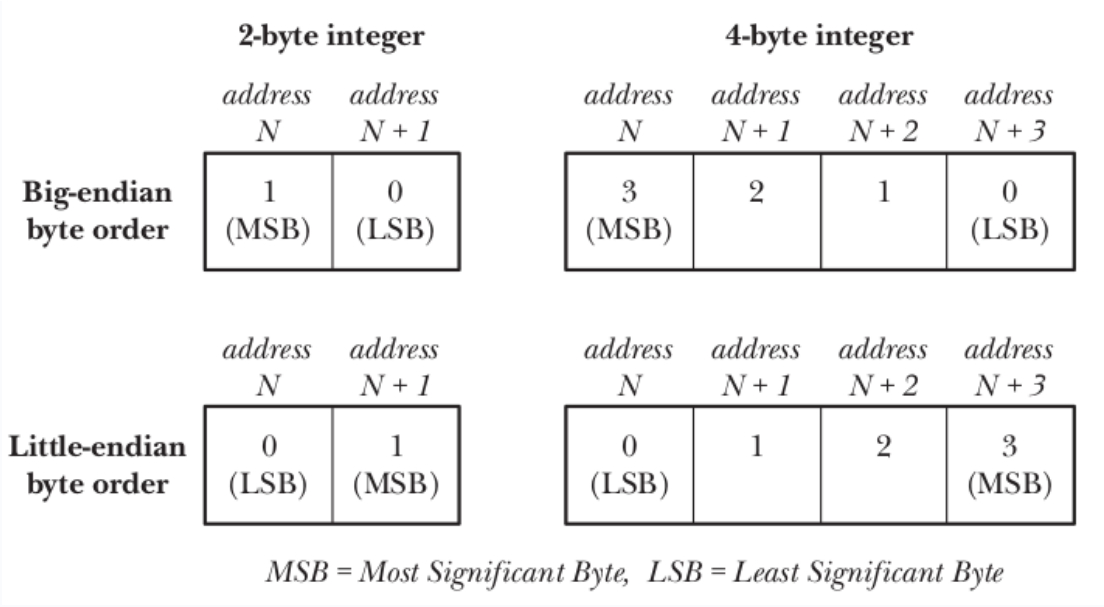
  ```c
  struct sockaddr_in s;
  inet_aton("1.2.3.4", &s.in_addr);
  ```

  Le varianti `inet_ntop` e `inet_pton` sono equivalenti, ma più moderne

Network Byte Order

Indirizzi IP e porte sono numeri interi su 32 e 16 bit.
Diverse architetture usano *convenzioni diverse* per l'ordine delle cifre
Necessario mettersi d'accordo quando si trasmettono via rete!

In rete si usa *Big Endian*, anche detto *Network Byte Order*: ovvero mettiamo *prima* le cifre più significative, poi alla fine le *cifre meno significative*.



Diverse architetture usano convenzioni diverse. Per ovviare a questi problemi, abbiamo le seguenti funzioni per convertire i numeri *Little-Endian* in *Big-endian* (o *Big-Endian a Big-Endian*)

```c
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Convertono da formato dell'architettura corrente (`h`) a Network Byte Order (`n`), numeri su $32bit$ `l` e su $16bit$ (`s`), e viceversa

Esempio:

```c
uint16_t port_h = 12345;
uint16_t port_n = htons(port_h);
```

Modificare le Opzioni di un Socket

```c
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname,
               void *restrict optval, socklen_t *restrict optlen);
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

Manipolano le opzioni per il socket **sockfd**.
Modificano comportamenti di default:

- Forzare la bind a una certa porta: **SO_REUSEADDR**
- Parametri di funzionamento di TCP
- Molte altre

Flusso Tipico per Socket Scream

LATO CLIENT.

```c
// Creazione
int fd = socket(AF_INET, SOCK_STREAM, 0);

/* Connessione: specifica indirizzo IP
   e porta del server */
connect(fd,
        (struct sockaddr*)&address,
        sizeof(address)));

// Input/Output
write(fd, buffer, n);
read(fd, buffer, SIZE);

// Chiusura
close(fd);
```

LATO SERVER.

```c
// Creazione
int fd = socket(AF_INET, SOCK_STREAM, 0);

// Bind: specifica porta
bind(fd, (struct sockaddr*)&address, sizeof(address));

// Listen: specifica lunghezza della coda in attesa
listen(fd, 3);

// Servizio ai client
while (1){

   /* Attesa di un client: ottiene indirizzo IP
      e porta del client */
   int active_fd  = accept(fd,
                           (struct sockaddr*)&address,
                           (socklen_t*)&addrlen)
                        );
   // Input/Output
   write(active_fd, buffer, n);
   read(active_fd, buffer, SIZE);

   // Chiusura
   close(active_fd);
}

// Chiusura
close(fd);
```

Risoluzione DNS

Esistono funzioni di libreria per effettuare risoluzioni DNS:

```c
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

Effettua una risoluzione DNS per il dominio **name**.
Ritorna una **struct hostent**, una struttura molto complessa che contiene i risultati della risoluzione

E' deprecata, ora si usa la simile `getaddrinfo`

Non vediamo in dettaglio

Esercizio sui Socket

Esercizio.
Il server $45.79.112.203$ alla porta TCP 4242 offre un servizio di `echo`.
Se un client vi si connette e manda un messaggio, il server risponde con lo stesso messaggio.
Si crei un programma che si connette al suddetto endpoint, manda un messaggio e stampa la risposta un messaggio.

SOLUZIONE.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#define SIZE 1024
#define MESSAGGIO "Ciao Mondo!\n"

int main(int argc, char *argv[]){

    int fd, n;
    char buffer[SIZE];
    struct sockaddr_in address;

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_port = htons(4242);
    if (inet_aton("45.79.112.203", &address.sin_addr) <0){
        perror("convert server ip failed");
        exit(EXIT_FAILURE);
    }

    if ((connect(fd, (struct sockaddr*)&address,sizeof(address)))< 0){
        perror("connect failed");
        exit(EXIT_FAILURE);
    }

    write(fd, MESSAGGIO, sizeof(MESSAGGIO));
    printf("Tramesso: %s\n", MESSAGGIO);

    n = read(fd, buffer, SIZE);
    buffer[n] = 0;
    printf("Ricevuto: %s\n", buffer);
    close(fd);

}
```

Networking in Linux

Interfacce di Rete

La gestione della rete cambia a seconda di distribuzione Linux/POSIX, ma ci sono dei concetti generali.

Ogni *interfaccia di rete* è identificata da un nome.

- Scheda Ethernet: **`eth0`** o **`eno1`**
- Scheda WiFi: **`wifi0`**
- Interfaccia di loopback: **`lo`**

Informazioni sulla Rete

1. **`ifconfig`** è il comando storico per avere informazioni.
   In realtà è obsoleto, ora si usa il comando **`ip addr`**

Esempio:

SHELL

```
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel
state UP group default qlen 1000
    link/ether 2c:f0:5d:c3:7b:b5 brd ff:ff:ff:ff:ff:ff
    altname enp0s31f6
    inet 140.105.50.104/24 brd 140.105.50.255 scope global dynamic
noprefixroute eno1
       valid_lft 101209sec preferred_lft 101209sec
    inet6 fe80::bf0b:ea7e:b8a9:d363/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

2. **`ip route`**: Quando viene generato un pacchetto, il sistema usa la *routing table* per decidere *su quale interfaccia trasmetterlo*

```shell
$ ip route
default via 140.105.50.254 dev eno1 proto dhcp metric 100
140.105.50.0/24 dev eno1 proto kernel scope link src 140.105.50.104 metric 100
```

La routing table viene creata in automatico quando si configurano le interfacce di rete, inserendo indirizzo IP, netmask e default gateway.

Configurazione della Rete

Storicamente, rete configurata tramite file di configurazione.

- **`/etc/network/interfaces`**: indirizzo IP, subnet mask e default gateway
- **`/etc/resolv.conf`**: resolver DNS

Ora si usa il demone *Netplan*, che ha file di configurazione in **`/etc/netplan/...`**

```yaml
network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses: [172.16.86.5/24]
      gateway4: 172.16.86.1
      nameservers:
          addresses: [8.8.8.8, 8.8.4.4 ]
```

Si applica la configurazione col comando:

```
netplan apply
```

I sistemi desktop hanno meccanismi di più alto livello per queste configuazioni

- Ubuntu Desktop ha *Network Manager* per configurare la rete tramite interfaccia grafica
- *Network Manager* scrive i file di configurazione per noi
- Attenzione a cambiare i file manualmente, rischio conflitto

Comandi Misti

Risoluzioni DNS:
`host <dominio>` o `dig <dominio>`

Troubleshooting:
`ping <destinazione>` e `traceroute <destinazione>`

Richieste HTTP:
`curl <URL>` o `wget <URL>` per scaricare pagine Web

Listare tutti i socket nel sistema:
Si usa il comando `netstat`, che ha molte opzioni:

- `-l`: Stampare solo socket passivi
- `-t`: Solo TCP
- `-p`: Stampare il PID e il nome del processo associato al socket

Utile per sapere se un programma server è attivo:

```shell
$ netstat -nplt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1411/sshd
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      950293/nginx:
maste
tcp        0      0 0.0.0.0:443             0.0.0.0:*               LISTEN      950293/nginx:
maste
tcp        0      0 0.0.0.0:5000            0.0.0.0:*               LISTEN      4014584/docker-
prox
```

Creazione di Socket da Comando

Il comando `nc` permette di creare e usare in maniera semplice un socket da riga di comando
Client:

```shell
nc <indirizzo> <porta>
```

Server:

```shell
nc -l <porta>
```

Quando il socket è connesso, si può scrivere e leggere nel socket usando il terminale

Esercizio: usare **nc** per scambiare messaggi tra due PC

Domande

Un server, per compiere pienamente le sue funzioni, usa:
• Socket Passivi **• Socket Attivi** **• Socket Passivi e Attivi**
Risposta: *Socket Passivi e Attivi*

Un client, per compiere pienamente le sue funzioni, usa:
• Socket Passivi **• Socket Attivi** **• Socket Passivi e Attivi**
Risposta: *Socket attivi*

Un Socket Stream è:
• Monodirezionale **• Bidirezionale**
Risposta: *Bidirezionale*

E' possibile usare anche le funzioni **read** e **write** per effettuare I/O su Socket Stream?
• Si **• No**
Risposta: *Sì*

A cosa serve il comando **ifconfig**?
• Configurare il comportamento di un socket
• Configurare le interfacce di rete
• Inviare pacchetti di configurazione
Risposta: *Configurare le interfacce di rete*

u8-s3-package

Sistemi Operativi

Unità 8: Altri Argomenti

Gestione dei Pacchetti Software

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

1. Perchè sono necessari
2. Package Manager
3. Pacchetti `deb` e Package Manager `apt`
4. Package Manager `snap`

Perchè sono necessari: Utenti e Programmi

Appena installato, un SO contiene *solo i programmi di default*

- Utility per gestione del SO: `ls`, `ps`, `free`

Un utente vuole far girare le applicazioni che preferisce. Ha due opzioni:

1. *Scrive un programma, lo compila e lo esegue*
2. Usa un programma *scritto da qualcun altro*

L'opzione 2 è di gran lunga la più usata

Per usare un programma, ci sono diverse opzioni:

- Scaricare il *file binario del programma* ed eseguirlo
- Oppure un *installer*
  - In Windows: Scarico `installer.exe` e installo
- *Scaricare e compilare* il codice sorgente
- Usare un *Package Manager*
  - Come *AppStore* su iOS o Google Play di *Android*

Package Manager

Definizione di Package Manager

Un *Package Manager* è un software che si occupa di organizzare i software in uso in un sistema.
Ha l'obbiettivo di:

- Permettere l'*installazione/rimozione* di pacchetti software
- *Verificare* che il software non sia corrotto e arrivi da fonti sicure
- *Gestire* eventuali *conflitti e dipendenze* tra pacchetti
- *Controllare gli aggiornamenti* dei software installati

Un Package Manager scarica i pacchetti da un *Repository* pubblico

Tipologie di Package Manager

Ci sono più tipologie di package manager. Ne elenchiamo tre.

Monolitici: l'applicazione e le tutte dipendenze *sono nello stesso pacchetto*

- Come MacOS o Docker
- **Vantaggi:** ogni pacchetto si porta dietro tutto ciò che gli serve
- **Svantaggi:** troppo spazio sprecato

A Pacchetti specifici: ogni pacchetto contiene *un singolo software/libreria*. Quando si scarica l'applicazione, il Package Manager *controlla e scarica eventuali dipendenze*.

- Usato tipicamente in Linux: `apt`, `yum`
- **Vantaggi:** installazione veloce, no spreco di spazio su disco
- **Svantaggi:** gestire le dipendenze aumenta di *molto* la complessità!

Source-Based: il Package Manager *scarica e compila il codice sorgente* di ogni pacchetto

- Come `brew` usato su MacOS
- **Vantaggi:** programmi portabili su diverse architetture
- **Svantaggi:** molto lento, dato che la compilazione richiede tanto tempo

Package Manager Principali

Elenchiamo alcuni *package manager* specifici, molti di cui sono noti

1. **Preinstallati nei SO**
   Fatti per l'utilizzo da parte di utenti non esperti
- Windows ⇒ Microsoft Store (precedentemente Windows Store)
- MacOS, iOS ⇒ AppStore
- Android ⇒ Google Play (precedentemente Android Market)

Caratteristiche:

- *Closed-source*: spesso il codice viene *offuscato* per evitare che il codice sorgente venga letto
- *Commerciali*: offrono applicazioni a pagamento

2. **Per la programmazione**
   Ambienti specifici hanno un *Package Manager* dedicato
- Python ⇒ `pip`, `conda`
- Java ⇒ `maven`
- JavaScript⇒ `npm`
- Go ⇒ `go get`

3. **In Linux**
   Esistono due *formati* di *Package Binari*, ovvero che contegono software compilato:
- Pacchetti *Deb*: usati in Debian, Ubuntu
- Pacchetti *RPM*: usati in Red Hat, CentOS
  I *Package Manager* installano *Package Binari* da repository pubblici:
- In Debian, Ubuntu: `apt`
- In Red Hat, CentOS: `yum`, ora rimpiazzato da `dnf`
  Vengono usati tipicamente da riga di comando

4. **In MacOS**
   I software sono tipicamente in immagini *DMG*
- Formato per immagini di disco
- *Contengono tutte le dipendenze* (paradigma monolitico)
  Si possono installare *Package Manager* aggiuntivi:
- `port` o MacPort
- `brew`
  Entrambi scaricano i sorgenti e li compilano (source-based)

Operazioni con Package Manager

Ogni *Package Manager* ha comandi diversi.

- Tipicamente si usano da riga di comando.
- Ma esistono interfacce grafiche per semplificare l'uso

Azioni comuni:

- `install`
- `remove`
- `update`
- `view dependencies`

Pacchetti `deb` e Package Manager `apt`

Approfondiremo il discorso, per quanto riguarda i *package Manager* su *Linux* (in particolare nei sistemi basati su *Debian* e *Ubuntu*)

Nei sistemi basati su Debian e Ubuntu si usa il formato *Deb*. Sono:

- **Package atomici:** ognuno contiene *un singolo software*
- **Binari compilati:** si scaricano programmi già compilati *per la propria architettura*

Un pacchetto *Deb* è un archivio compresso contenente:

- I *file binari*
- *Metadati*: nome, versione
- Lista delle *dipendenze*

- Opzionalmente:
  - File di configurazione
  - Script da eseguire per *installazione* o *disinstallazione*
  - Firma digitale GPG (per evitare eventuali fake da parte di malfattori)

Installazione Manuale dei Pacchetti `deb`

Se scarico pacchetti `.deb` per fatti miei, posso gestirli *manualmente*.

Il comando `dpkg` permette di gestire pacchetti *Deb*

- Installazione: `dpkg -i <file.deb>`
- Informazioni su un pacchetto: `dpkg -I <file.deb>`
- Disinstallazione: `dpkg -r <nome-pacchetto>`
- Lista di pacchetti installati: `dpkg -l`
- Lista dei file installati da un pacchetto installato: `dpkg -L <nome-pacchetto>`

`dpkg` è un tool *di basso livello*

- Installa pacchetti da file `deb`
- Non risolve le dipendenze
- Non pratico da usare
  Devo fare tutto a mano... c'è un modo per evitarsi questi problemi? Ma certo!
  Solitamente non si usa `dpkg` direttamente, ma *Advanced package tool* (`apt`): risolve i problemi di cui sopra

Advanced Package Tool `apt`

1. *Repository*
   `apt` scarica package da *repository online* (della società privata *Canonical*):
- Lista ottenuta dal file: `/etc/apt/sources.list` e da tutti i file nella cartella `/etc/apt/sources.list.d/`
  - Repository *pre-definiti* quando si installa il SO
- Si possono aggiungere repository per package non presenti di default:
  - E.g., *Chrome, Dropbox*
- Un repository è identificato da un *URL* e ha dei *tag*
  - Esempio: `deb http://it.archive.ubuntu.com/ubuntu/ focal main restricted`

2. *Comandi*
   Per installare paccheti con `apt` si usa il comando `apt` o `apt-get` (più *vecchio* ma *analogo*)
- Installazione: `apt install <nome-pacchetto>`
- Disinstallazione: `apt remove <nome-pacchetto>`
- Aggiornamento delle *liste* di pacchetti disponibili: `apt update`
  - **NOTA!** Questo comando *non*
- Ricerca di pacchetti nei repository: `apt-cache search`

3. *Dipendenze*
   Ogni volta che si installa un pacchetto, **apt** *risolve le dipendenze* (o almeno ci prova)
   - Installa in automatico le librerie i software da cui dipende
   - Problema complesso: generato un *grafo delle dipendenze*

Possono nascere *conflitti*, per problemi di versione: ad esempio voglio scaricare **pacchetto1** che richiede **pacchetto2** versione 2.0, ma ho l'ho già installata in versione 1.0 c'è un problema

> The following packages have unmet dependencies:
> package1 : Depends: package2 (⩾ 1.8) but 1.7.5-1ubuntu1 is to be installed

Tipicamente i pacchetti nei repository di sistema *non hanno questi problemi*

4. *Risoluzione dei problemi*
   In caso di *dipendenze non risolte o altri problemi*, si può dire ad **apt** di fare pulizia
   - **apt autoclean**: elimina i pacchetti **.deb** scaricati relativi a versioni vecchie
     - **Nota:** rimuove *l'archivio Deb*, che è inutile dopo installazione, ma viene tenuto in *cache*. Non rimuove l'*installazione*
   - **apt clean**: elimina tutti i pacchetti **.deb** in *cache*
   - **apt autoremove**: *disinstalla* i *pacchetti orfani*, ovvero dipendenze installate per l'installazione di un'applicazione che poi *rimuovete*, così non sono più necessarie

Da apt a snap

apt funziona molto bene ed è usato con successo nella maggior parte dei sistemi Linux

- *Economizza lo spazio*: i pacchetti hanno *dipendenze*
- Le dipendenze sono *installate e condivise* da tutto il sistema (come visto, ciò è complesso da fare)

Nei sistemi Ubuntu, ora a fianco di **apt** si usa anche *Snap* (monolitico)

- Installato di default su Ubuntu, installabile anche su altre distribuzioni

Package Manager snap

Snap installa pacchetti *self-contained*

- Contengono il *programma* e *tutte le dipendenze*: librerie, altro software
- Di fatto contengono un *File System in formato SquashFS* (ovvero il software con le sue dipendenze)
- Le applicazioni girano in una *SandBox*, con limitato accesso al sistema
- *Concettualmente* simile a un *container*!
  - Simile a Docker, ma pensato anche per utenti non esperti

    - Evita casini di accesso, di vulnerabilità, eccetera...

Vantaggi:

- *Risolve* problemi di *dipendenze*
- *Maggiore sicurezza* grazie a SandBox

Svantaggi:

- Si usa *più spazio su disco*
- Pacchetti sono *più grandi da scaricare* dalla rete
- *Più pesante per il sistema*:
    - Il File System di un pacchetto viene *montato* ad ogni avvio

Per riassumere: ho molti vantaggi, ma pagando un prezzo salato...

Domande

A cosa serve un Package Manager?
• A instradare i pacchetti di rete
• A installare i pacchetti software da repository pubblici
• A installare i programmi creati dall'utente
Riposta: *A installare i pacchetti software da repository pubblici*

Un pacchetto Deb contiene le tutte sue dipendenze:
• Si **• No**
Risposta: *No*

Un pacchetto Deb contiene i file sorgenti:
• Si **• No**
Risposta: *No*

Il Package Manager **apt** installa le dipendenze:
• Automaticamente **• Mai** **• Su richeista**
Risposta: *Automaticamente*

Un pacchetto Snap contiene le tutte sue dipendenze:
• Si **• No**
Risposta: *Sì*