Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Virtual Distributed Container

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Alessio Volpe

Sessione II
Anno Accademico 2017-2018

A Ludovica,
per avermi spinto in questa follia.

Indice

Elenco delle figure

Introduzione

Ad oggi lo scenario della virtualizzazione è dominato dai *container*. Tecnologie come *Docker* sono impiegate nella realizzazione di applicazioni composte da piccoli processi indipendenti e altamente disaccoppiati. Ogni cellula è incaricata di un compito ben specifico e tutte cooperano a formare un organismo più complesso e dinamico.
Questo approccio fa un massiccio uso delle reti come mezzo di comunicazione tra le parti. Le reti in questione sono molto spesso software, o meglio virtuali. Esse fanno le veci delle ormai consolidate reti fisiche che, per definizione, sono statiche e difficilmente modellabili.

Virtual Distributed Ethernet (VDE) è un esempio di rete virtuale in grado di offrire molteplici configurazioni e può essere utilizzato come rete di uso generale e/o privato. Essa implementa meccanismi di commutazione switched e switchless conformemente allo standard LAN e VXLAN. Inoltre, la compatibilità Ethernet di VDE permette già una completa compatibilità tra le più conosciute Virtual Machine.

Il lavoro presentato in questa tesi è intitolato *Virtual Distributed Container* (VDC) a indicare un nuovo modello VDE applicato alla rete Docker. Il progetto mira a offrire tutte le caratteristiche VDE come supporto all'interoperabilità dei *container*. VDC si presenta come un plugin che può essere installato all'interno di Docker e utilizzato esattamente allo stesso modo di un driver di rete nativo.

In questa trattazione saranno analizzate le principali tecnologie fondative di VDC. In particolare si guiderà il lettore attraverso quelli che sono i principi della virtualizzazione, per poi inoltrarsi negli aspetti implementativi e funzionali di Docker e VDE.
Infine saranno presentate le caratteristiche del modello VDC, l'implementazione e le sue applicazioni.
Sono stati realizzati, inoltre, alcuni test per mostrare al lettore la qualità del lavoro svolto in relazione al più diffuso modello di rete Docker: il Bridge Linux.

Capitolo 1

Principi di virtualizzazione basata su Container

Negli ultimi anni si sente spesso discutere di virtualizzazione basata su *container*, ma di cosa si tratta in realtà? Per rispondere a questa domanda è necessario conoscere il significato di virtualizzazione e le sue origini.

La virtualizzazione è una tecnica attraverso la quale è possibile astrarre le risorse di una macchina in rappresentazioni logiche equivalenti. Le risorse logiche, così ottenute, sono finalizzate a formare le Virtual Machine (VM). Interi sistemi (virtuali) in grado di prosperare all'interno della medesima macchina host.

Il concetto di macchina virtuale, appena esposto, assume un diverso significato a seconda del livello in cui si opera. Dal punto di vista di un processo che esegue un programma utente, la macchina è uno spazio d'indirizzamento logico della memoria e un insieme di system calls. Diversamente, per il sistema operativo (SO) le caratteristiche hardware sottostanti ne definiscono la macchina stessa [36]. Dunque è chiaro che esistano diversi tipi di virtualizzazione che prendono il nome di: *Virtualizzazione Hardware-level* e *Virtualizzazione System-level*.

La **virtualizzazione Hardware-level** prevede l'impiego di uno strato software chiamato *Virtual Machine Manager* (VMM) in cima alle risorse hardware o al sistema operativo. Un VMM, o *hypervisor*, ha il compito di virtualizzare l'hardware sottostante o di emularne uno completamente nuovo. Al di sopra di esso si trovano le Virtual Machine, ognuna con il proprio Kernel, inclusi i file binari, le librerie e le applicazioni.

La **virtualizzazione System-level** si basa sul concetto di "contenitore": un ambiente isolato che include tutte le risorse necessarie all'esecuzione di un'applicazione. Tali ambienti, conosciuti come container, sono delle istanze virtuali del sistema host.

La differenza sostanziale tra VM e container è che quest'ultimo non possiede un proprio SO ma condivide il Kernel del sistema host. Pertanto la virtualizzazione System-level è definita come leggera e performante.

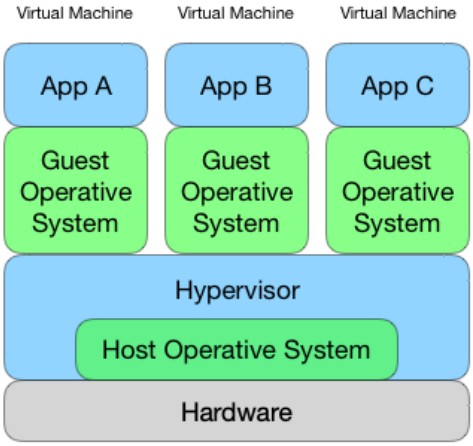

Figura 1.1: Virtual Machine

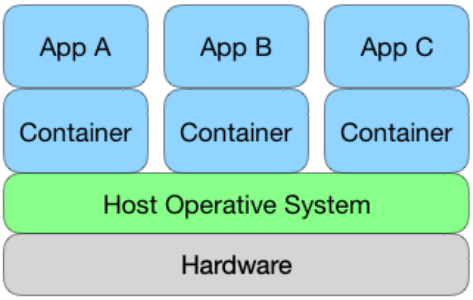

Figura 1.2: Container

Dunque un container racchiude in se stesso un file system completo e tutto ciò che è necessario per eseguire un'applicazione: binari, ambiente runtime, tool di sistema librerie eccetera. Tutto quello che avviene all'interno di un container non influisce in alcun modo sul sistema host e viceversa. Questa astrazione è permessa grazie all'impiego di alcune funzionalità del Kernel Linux che sono sostanzialmente tre: *Namespace*, *Cgroups* e *LXC: Linux Container*.

1.1 Linux Namespace

Introdotti nel Kernel Linux dalla versione 2.4.19, i *Namespace* sono alla base dell'architettura di un container. Essi virtualizzano il Kernel per la realizzazione di diversi ambienti operativi in cui i processi sono parzialmente o completamente isolati dal sistema host [29]. Tale definizione sembrerebbe coincidere con quella di container, e concettualmente sono equivalenti. Tuttavia, la parola container indica un insieme ben più ampio di tecnologie che cooperano per realizzare questo tipo di virtualizzazione.
Esistono diverse tipologie di Namespace ognuna incaricata di un compito ben specifico:

- Cgroup Namespace: virtualizza il contenuto di cgroupfs (sezione 1.2);
- IPC Namespace: virtualizza il meccanismo di Inter Process Communication (IPC), ovvero gli oggetti System V IPC e le code dei messaggi POSIX [27];

- Network Namespace: fornisce un nuovo stack di rete isolato da quello del sistema host;
- Mount Namespace: nasconde i punti di montaggio del file system host e ne offre una gestione isolata. I processi in diversi Mount Namespace possono avere una visione differente del file system;
- PID Namespace: consente di creare una nuova enumerazione dei Process IDs (PIDs) differente dal sistema host, in modo che lo stesso PID possa essere riutilizzato. Il primo processo all'interno del namespace assume il ruolo di init (PID 1) godendo della maggior parte dei suoi privilegi. Si noti che la terminazione di tale processo comporta la terminazione dell'intera gerarchia all'interno del namespace [35];
- User Namespace: lo UserID e il GroupID di un processo possono essere diversi all'interno e all'esterno dello User Namespace. Ciò permette a un processo, che normalmente (all'esterno del namespace) è un processo utente, di acquisire privilegi root;
- UTS Namespace: UNIX Time-sharing System (UTS) Namespace nasconde ai processi i due identificatori del sistema host: l'hostname e il nome di dominio all'interno del Network Information Service (NIS) [27].

I Namespace hanno apportato un enorme contributo nel campo della virtualizzazione rendendola flessibile e performante. Tuttavia, nonostante tale tecnologia sia matura e consolidata è bene prestare prudenza nel suo utilizzo. Infatti, dal momento che tutti meccanismi implementativi dei Namespace risiedono all'interno del Kernel è inevitabile un aumento della vulnerabilità [4]. Ad esempio, se un processo contenuto nello User Namespace fuoriuscisse dalla virtualizzazione, potrebbe di fatto assumere il controllo del sistema host. Per questo motivo, tecnologie come Docker che fanno largo uso dei Namespace attualmente stanno migrando verso un approccio ibrido di virtualizzazione [1].

1.2 Linux Control Groups

Linux Control Groups, conosciuto come *cgroups*, è una funzionalità del Kernel complementare ai Namespaces nella costruzione di container. *Cgroups* permette infatti di distribuire le risorse del sistema -CPU, memoria, rete, eccetera- in modo controllato a gruppi di processi gerarchicamente organizzati [25].

[1] Docker ha introdotto una sorta di *hypervisor*, chiamato gVisor [24], tra i container e il Kernel.

Un cgroup è una raccolta di processi legati da un insieme di limiti o parametri definibili attraverso *cgroupfs*: uno pseudo-filesystem che funge da interfaccia con il Kernel [28]. La struttura gerarchica costituente il gruppo richiama alla gestione dei processi Linux [35]; allo stesso modo i "figli ereditano dal processo genitore" le restrizioni imposte.

Cgroups si interpone tra le risorse e i processi assumendo il ruolo di contabile (tiene traccia delle richieste) e di controllore; esso limita l'utilizzo della memoria e ripartisce le priorità di accesso alla CPU e ai dispositivi I/O [35]. Questo meccanismo disciplina i processi nell'uso delle risorse preventivando, inoltre, attacchi di tipo DoS (Denial-of-Service) [39].

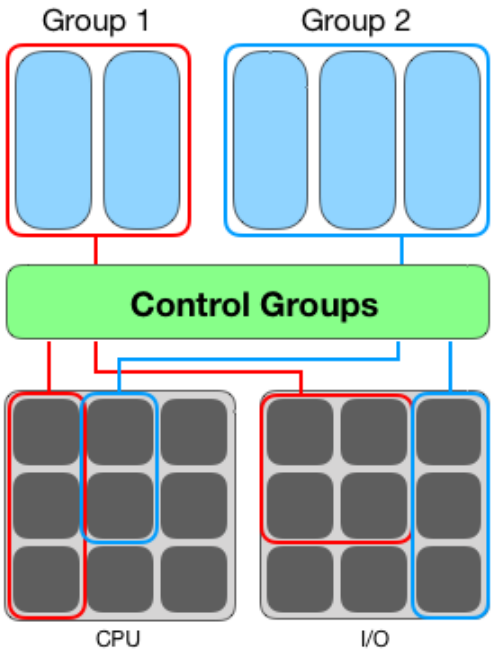

Figura 1.3: Linux Control Groups: *cgroups*

Si noti che nonostante le risorse siano limitate i processi continuano ad avere accesso ai dati dell'host, pertanto *cgroups* è necessario ma non sufficiente a garantire una completa virtualizzazione.

1.3 LXC: Linux Container

Linux Container (LXC) è il risultato ottenuto dalla cooperazione di *Namespace* e *cgroups*. LXC dialoga direttamente con le funzioni di virtualizzazione del Kernel e implementa

una serie di strumenti e librerie finalizzate alla creazione e la gestione dei container. Storicamente è possibile classificare LXC come la prima implementazione di virtualizzazione basata su container.
Linux Container, inoltre, è stato fonte d'ispirazione per la nascita di tecnologie come Docker. Attualmente esso è sostituito dai suoi successori LXD e LXC files ystem (LXCFS) [5] che tuttavia lasciano la struttura interna di un container pressoché invariata.

Capitolo 2

Docker

Per comprendere a pieno in cosa consiste Docker è bene introdurre due problemi che ricorrono sistematicamente durante il ciclo di vita del software. Un'applicazione software è spesso legata a un ambiente di runtime ben preciso contenente le librerie necessarie al suo funzionamento. L'eterogeneità dei sistemi rende difficile la portabilità, in quanto non è garantito che un'applicazione possa trovare tutto ciò di cui necessita in un sistema differente da quello di sviluppo. Inoltre, la crescita incessante di applicazioni sempre più complesse e interconnesse alla rete fa sì che gli ambienti di runtime siano popolati da diversi moving parts. In questo scenario anche un semplice aggiornamento diventa difficoltoso dal momento che bisogna prestare attenzione a non infrangere la convivialità delle parti e a garantire la continuità del servizio; questi due problemi sono noti rispettivamente come il problema delle dipendenze e il problema del deployment. Tali necessità sono state il presupposto alla base sviluppo di Docker.

Docker è un software open-source con il quale *pacchettizzare* e *deployare* applicazioni all'interno di container [1]. Ciò garantisce alle applicazioni di funzionare esattamente allo stesso modo in qualunque ambiente di runtime poiché tutto ciò di cui necessitano è racchiuso nel container. In realtà, lo scenario è ben più ampio: Docker è in grado di orchestrare infrastrutture basate su container finalizzate alla realizzazione di applicazioni distribuite. Sviluppato dalla *dotCloud Inc.* (2013, oggi *Docker Inc.*) sin da subito ha riscosso un enorme successo posizionandosi tra le principali tecnologie di virtualizzazione [26]. In questo capitolo saranno analizzate le componenti organiche di Docker, focalizzando l'attenzione sulle qualità che lo hanno reso così popolare nel campo dello sviluppo software: l'interoperabilità (*Docker Networking*) e l'estensibilità (*Docker Plugin*).

2.1 Docker Engine

Docker è basato su un'architettura client-server con tre attori protagonisti che nell'insieme sono definiti come Docker Engine [17].

Docker API REST: formalizza il protocollo di comunicazione tra client e Docker Daemon.

Docker Client: implementa la Command Line Interface (CLI) che traduce gli input da console in richieste REST compatibili con Docker API. Inoltre Docker Client può comunicare con più Daemon.

Docker Daemon: gestisce gli oggetti Docker come immagini, container, reti e volumi. Un Daemon può anche comunicare con altri Daemon per soddisfare le richieste.

Questa scelta implementativa ha permesso una buona suddivisione delle responsabilità ma di fatto ha abbattuto le barriere della "località", facendo sì che lo stesso Engine possa essere distribuito su host differenti.

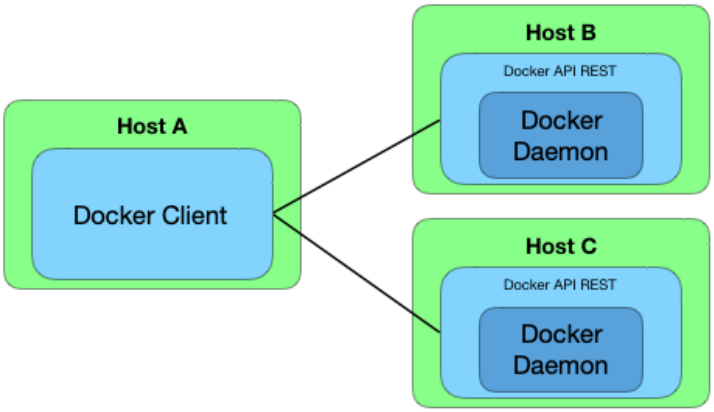

Figura 2.1: Docker Engine distribuito

Quando si parla di un container Docker in realtà si fa riferimento a un insieme di componenti. Il container, come presentato nella sezione 1.3, è esso stesso una componente. Di seguito saranno brevemente introdotti Docker Image e Docker Volume.

2.1.1 Image

Un'immagine Docker è la rappresentazione dello stato interno di un container, in particolare del file system e dei suoi costituenti (librerie, applicazioni, file, ecc.). Grazie all'impiego dell'*Advanced Multi-layered Unification Filesystem* (AUFS) è possibile unificare diverse directory in un singolo punto di mount.
Docker costruisce queste immagini come uno stack di file system in cui ogni livello è l'insieme delle differenze rispetto al livello sottostante [16].
Quando un container viene avviato da un'immagine, il driver di archiviazione aggiunge un ulteriore strato chiamato *container-layer* all'ultimo livello sullo stack. Tutte le modifiche

del file system durante l'esecuzione vengono salvate utilizzando la tecnica Copy-On-Write (CoW) sul *container-layer*. Tuttavia, tale layer sarà cancellato alla terminazione, a meno che non sia espressamente salvato come nuovo livello dell'immagine di partenza. Poiché ogni container ha il proprio *container-layer*, più container possono condividere l'accesso alla stessa immagine continuando ad avere il proprio stato dei dati.
Infine, la struttura a livelli, precedentemente descritta, offre la possibilità di costruire nuove immagini a partire da immagini preesistenti[1].

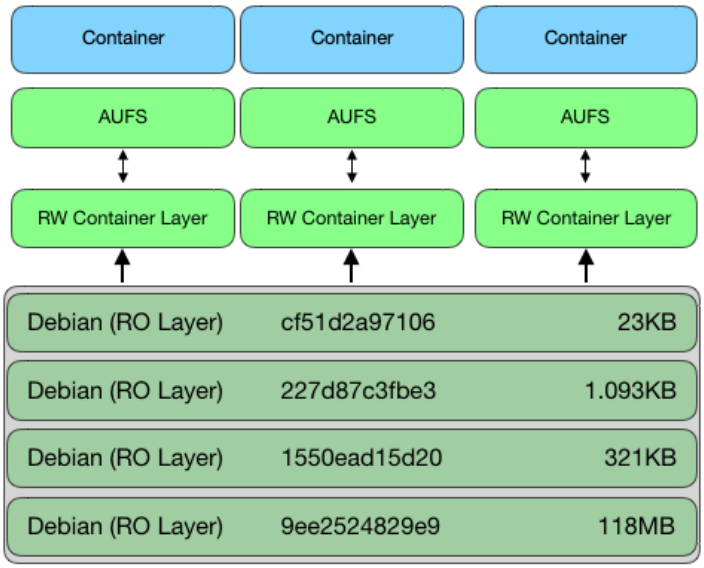

Figura 2.2: Docker Image layer

2.1.2 Volume

Un volume è una directory o un file del file system host montato direttamente in un container; l'utilizzo dei volumi, dunque, è il mezzo più semplice per ottenere memoria condivisa tra container e host. Le operazioni sui volumi sono persistenti e ignorate dal driver di archiviazione.

2.2 Docker Networking

In sistemi distribuiti o di microservizi la rete diventa un tassello fondamentale per la comunicazione e la sincronizzazione tra le parti. Docker implementa la rete di un container con l'impiego del Net Namespace (sezione 1.1) fornendo pertanto a ogni container

[1]Docker Hub e Docker Cloud sono delle piattaforme pubbliche da cui è possibile caricare/scaricare immagini pre-built [17].

un proprio stack di rete isolato dal sistema. Tuttavia, con appropriate regole di routing, è possibile indirizzare pacchetti al dispositivo di rete di un container. Inoltre, Net Namespace supporta solo dispositivi di rete virtuali (es. veth [30]). I "reali", collegati direttamente all'hardware fisico della macchina, sono esclusivamente destinati al root namespace (il sistema host).

2.2.1 Networks

Docker offre una serie di network nativi con la possibilità d'installarne altri successivamente sotto forma di plugin (sezione 2.3).
Di seguito saranno presentate alcuni dei principali driver di rete supportati.

Bridge (Default)

Una rete Bridge sfrutta l'interfaccia virtuale di tipo Linux Bridge che consente la comunicazione tra container collegati allo stesso device. Bridge è la rete di default alla quale Docker assegna tutti i container che non ne specificano un'altra [19].

Host (Default)

Il driver Host fa si che lo stack di rete del container non sia isolato dal sistema. Ad esempio, se si esegue un container sulla porta 80 con la rete host, l'applicazione sarà disponibile sulla porta 80 all'indirizzo IP dell'host [19].

Overlay (Default)

Il driver Overlay crea una rete distribuita tra più Docker Daemon, posizionandosi in cima (overlay) alle reti degli specifici host. Consente dunque ai container che adottano questo driver di comunicare in modo sicuro eliminando, inoltre, la necessità di routing a livello di sistema operativo [19].

Macvlan (Default)

Macvlan assegna un indirizzo MAC al container rendendolo visibile come una macchina fisica sulla rete. In questo caso Docker dirotta il traffico da e verso gli indirizzi MAC [19].

Open vSwitch

Open vSwitch [21] (OVS) è uno switch virtuale nato per interconnettere VM all'interno di un host e tra host differenti attraverso meccanismi di *tunneling* come VXLAN. OVS si presenta come un network plugin ed è in grado d'interconnettere i singoli container, appartenenti ad host differenti.

Weave Net

Weave Net è una rete virtuale in grado di collegare i container su più host. Con Weave Net i container che costituiscono applicazioni basate su micro-servizi possono risiedere ovunque: su un host, su più host o su data center come se fossero tutti collegati allo stesso switch di rete. Dunque, i servizi offerti dalle applicazioni possono essere esposti al mondo esterno, indipendentemente dalla loro posizione [43].

2.2.2 CNM: Container Network Model

Questa sezione descrive come Docker implementa il network dei container. Lo strumento fondamentale è *libnetwork*: un progetto che mira a realizzare piccole componenti modulari e indipendenti.
Libnetwork implementa il *Container Network Model* (CNM) il quale formalizza i passaggi per fornire il supporto di rete ai container. Tempo stesso, il CNM offre l'astrazione per cui un container può utilizzare diversi driver di rete concorrentemente [13, 20].
Il CNM è sostanzialmente formato da tre componenti *Sandbox*, *Endpoint* e *Network*.

Sandbox

Un sandbox può essere un Linux Network Namespace, un Jail FreeBSD o un altro concetto simile contenente lo stack di rete di un container. È incaricata della gestione delle interfacce, della tabella di routing e delle impostazioni del DNS [33, 20]. Inoltre, a ogni container appartiene una solo sandbox che a sua volta può gestire diversi endpoint.

Endpoint

Un endpoint all'interno di un sandbox è l'istanza di un dispositivo di rete virtuale come veth o tun*tap* che il sistema host mette a disposizione per il container.

Network

Un Network è un gruppo di endpoint in grado di comunicare tra loro. L'implementazione di una rete potrebbe essere un Bridge Linux o una VLAN[33].

Libnetwork, dunque, fornisce un API che va a scindere l'implementazione di una rete rispetto alla sua gestione. Ciò ha permesso un enorme passo in avanti per quanto riguarda l'estensibilità e la facilità d'integrazione con tecnologie all'infuori di Docker.

2.3 Docker Plugin

"Batteries included but swappable" [15].

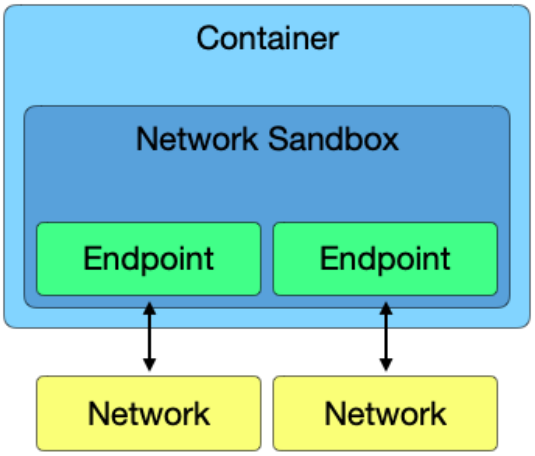

Figura 2.3: Container Network Model

L'ecosistema di Docker cresce in modo esponenziale e sempre più applicazioni necessitano di soluzioni personalizzate. Per il Team Docker la soluzione è stata offrire un'interfaccia col quale gli utenti potessero estendere Docker con strumenti di terze parti, conservando tuttavia la portabilità, la compattezza e la facilità d'uso.

"I plugin Docker sono estensioni out-of-process che aggiungono funzionalità al Docker Engine" [18]. I plugin, dunque, consentono al Docker Engine di comunicare con servizi esterni che vanno a integrare le singole componenti come Volume e Network. Tali componenti, già presenti all'interno di Docker, potrebbero non essere adatte allo scopo di una specifica applicazione.

Docker, inoltre, va incontro alle esigenze degli utenti rilasciando la libreria *go-plugins-helpers* [12] che formalizza il protocollo di comunicazione tra i plugin e il Docker Engine. La libreria è sviluppata nel linguaggio Go, nativo dell'implementazione Docker, e attualmente supporta le seguenti tipologie di estensione:

- Authorization: Estende il meccanismo di autorizzazioni;
- Network: estende la gestione delle reti;
- Volume: estende il persistent storage;
- IPAM: estende IP Address Management;
- Graph: estende le immagini e il file system.

Nel caso del Network Plugin, go-plugins-helpers rispecchia perfettamente le esigenze del CNM e della libnetwork (sezione 2.2.2).

2.3.1 Docker Discovery

Il Discovery è il meccanismo con il quale Docker individua e instaura una connessione con i plugin. Docker si interfaccia con i plugin attraverso socket Unix o connessioni TCP [33]. Un plugin ha il compito di fornire al Discovery gli estremi della comunicazione attraverso file di tipo .sock, .spec o .json nel seguente modo:

I file .sock sono dei socket Unix che devono essere posizionati nella directory /run/docker/plugins.

I file .spec o .json descrivono il plugin, e in particolare contengono l'URL al quale Docker può mandare le richieste. Ad esempio: unix:///[name].sock o tcp://localhost:8080 [18].

In precedenza i plugin sono stati classificati come server out-of-process. Da un altro punto di vista è lecito definire un plugin semplicemente come un'applicazione che può essere, dunque, *pacchettizzata* in un container. Infatti, Docker da la possibilità d'installare estensioni di terze parti sotto forma di Docker Image.

Capitolo 3

VDE: Virtual Distributed Ethernet

"A swiss army knife for emulated networks" [6].

Virtual Distributed Ethernet è un insieme di strumenti, appartenenti alla raccolta di progetti Virtual Square [2], per la costruzione e gestione di reti virtuali. Il nome VDE sta a indicare quelle che sono le principali caratteristiche:

- Virtual: reti puramente software;
- Distributed: i nodi sono distribuiti;
- Ethernet: conforme allo standard LAN.

Il software è organizzato in modo semplice e modulare, secondo la filosofia Unix, offrendo all'utente la massima libertà di espressione nel comporre la propria rete.
Le componenti VDE virtualizzano le loro controparti fisiche -cavi, connettori, switch, eccetera- ottenendo così il vantaggio di essere impiegate nell'interconnessione tra VM. Inoltre, essendo conforme a Ethernet, VDE consente di estendere la rete stessa degli host fisici.

Attualmente, molti dei VMM più utilizzati hanno il supporto nativo di VDE. Questa lista comprende qemu, kvm, virtualbox, user-mode e vuos [38].
Una rete VDE può essere quasi interamente costruita all'interno di uno o più host senza alcun privilegio root. Per questo motivo VDE trova applicazioni nell'insegnamento, nella sperimentazione e nel testing senza esporre il sistema a rischi di sicurezza. Inoltre, la maturità e le prestazioni raggiunte hanno aperto la strada anche per l'applicazione di VDE in ambito professionale.

3.1 Componenti

L'obiettivo principale della progettazione VDE è l'interoperabilità che si sviluppa su più moduli, o componenti, in grado di cooperare tra loro al fine di realizzare una rete interamente virtuale. Questo modello consente una chiara separazione delle responsabilità e la riduzione al minimo dell'accoppiamento tra le parti. La comunicazione avviene attraverso meccanismi di IPC [35] come pipe e socket; le interazioni sono regolate da un API interna. La decentralizzazione dei compiti, così ottenuta, offre un'elevata resilienza ai guasti.
Di seguito saranno introdotte le componenti che costituiscono una rete VDE.

VDE Switch

Fin dall'inizio vde_switch ha rappresentato il nodo portante nelle reti VDE.
Esattamente come un commutatore fisico, lo switch VDE si occupa d'implementare i meccanismi *datalink* del modello ISO/OSI [33]. Esso è in grado, dunque, d'interconnettere più nodi e gestire il *forwarding* di pacchetti. Inoltre, implementa l'algoritmo di Spanning Tree e l'invecchiamento del Port Mapping per consentire la riconfigurazione quando la topologia della rete cambia [33].
Lo switch VDE è di tipo managed e supporta connessioni VLAN secondo la specifica IEEE 802.1Q [3]. È dotato infine di un'opzione per utilizzarlo come hub; utile al debug e per collegare analizzatori di traffico.

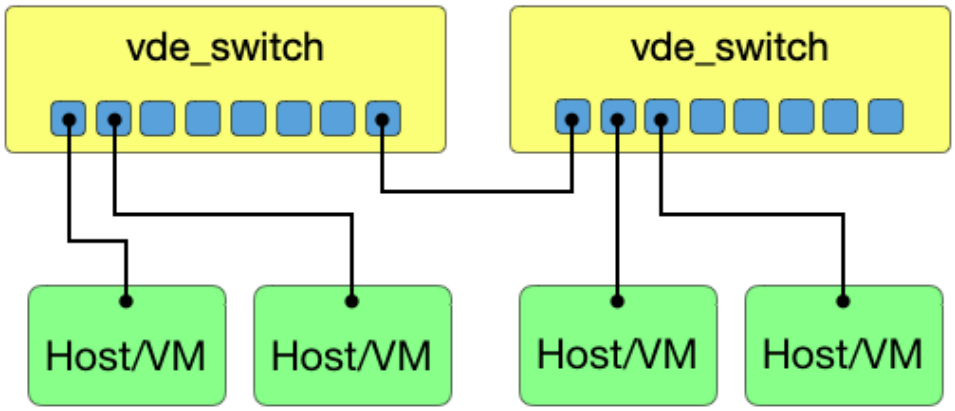

Figura 3.1: vde_switch

È possibile far riferimento a uno switch VDE attraverso socket Unix:

- [switch].ctl: di default situato nella directory /tmp/, è il socket che permette d'instaurare una connessione;
- [switch].mgmt: mette a disposizione l'interfaccia manager dello switch.

VDE Wire

Nel gergo VDE, tutti i meccanismi o strumenti in grado di trasportare dati da un estremo all'altro della rete assumono il nome di Wire. Ad esempio: pipe, dpipe [32], socket Unix, netcat, ssh.

VDE Plug

Analogamente alla sua controparte fisica, un plug VDE rappresenta logicamente un connettore. Un organo di accoppiamento in grado di legarsi a uno specifico nodo e fornirgli l'interfaccia necessaria alla ricezione/invio dei dati da/per la rete VDE.

VDE Cable

La composizione di un Wire e due Plug VDE prende il nome di Cable. Esattamente come un normale cavo Ethernet un Cable VDE è in grado d'interconnettere due estremità della rete. Uno degli strumenti più utilizzati per la creazione di collegamenti è vde_plug. Nella sua versione attuale, vde_plug [10] ingloba al suo interno sia la funzione di plug che quella di wire, rendendolo un ottimo tool per la gestione dei cablaggi.

```
$ vde_plug vde:///tmp/switch1.ctl vde:///tmp/switch2.ctl
$ vde_plug vde:///tmp/switch2.ctl \
cmd://"ssh remote.host vde_plug /tmp/switch3.ctl"
```

Si noti che la possibilità di usare strumenti come ssh apporta diversi vantaggi all'utente: in questo caso è stato possibile interconnettere due nodi geograficamente distanti attraverso una Virtual Private Network (VPN [33]).

VDE Wire Filter

Uno dei vantaggi delle reti virtuali è che ignorano i tipici problemi che possono incorrere nelle connessioni fisiche. Tuttavia, nel campo del testing è utile poter analizzare il comportamento di un'applicazione distribuita la quale inevitabilmente deve fare i conti con gli inconvenienti dei supporti fisici.

Wire filter (wirefilter [41]) è lo strumento col quale è possibile riprodurre errori e limitazioni delle connessioni reali all'interno di una rete VDE. Da un punto di vista pratico agisce come un vero e proprio filtro che si interpone tra due nodi. Wire filter mette a disposizione una serie di parametri e opzioni configurabili a runtime [11]. Di seguito alcuni di essi:

- Larghezza di banda;
- Velocità di trasmissione;
- Millisecondi di ritardo;

- Percentuale dei pacchetti persi;
- Percentuale dei pacchetti duplicati;
- Numero di bit per megabyte danneggiati.

Inoltre, è possibile creare uno scenario più complesso attraverso l'utilizzo di una catena di Markov che emula i diversi stati del collegamento e le loro transizioni [11].

3.2 VDE Plug 4

Le componenti all'interno di VDE2 [37] erano sviluppate come entità singole e fini a se stesse. Estendere il framework a nuove funzionalità richiedeva un massiccio riutilizzo del codice. Infatti, il supporto a differenti tipologie di nodi richiedeva l'implementazione di specifici plug, ad esempio: vde_plug2*tap* e vde_pcapplug [11].

VDEplug4 [10], dunque, nasce dall'esigenza di correggere la struttura troppo rigida del suo predecessore. Esso implementa un protocollo generale che è alla base di tutte le sue componenti. Inoltre, VDEplug4 offre una la libreria *libvdeplug*: un'API con la quale un nodo (VM o applicazione) è in grado d'implementare il plug necessario al collegamento VDE.

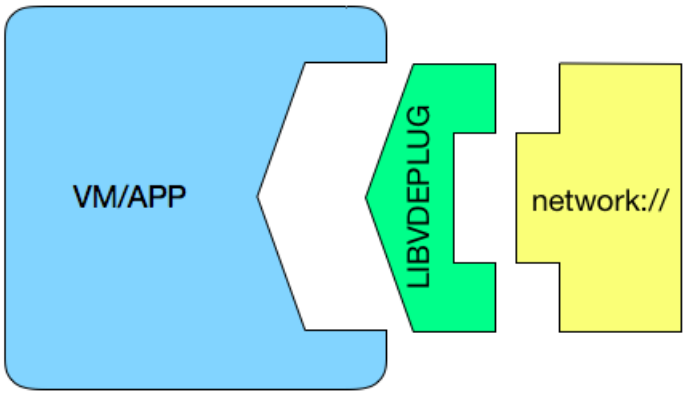

Figura 3.2: VDE Plug 4: libvdeplug [9]

La libvdeplug utilizza degli URL per identificare una rete o un nodo. Ogni indirizzo è costituito dal tipo di plugin e da eventuali settaggi o parametri, ad esempio: `plug://plug_set/arg=value`. Di seguito è riportata la classificazione dei plugin supportati da libvdeplug:

- vde: vde_switch 3.1;
- p2p: connessione peer to peer tra due VM;
- *tap*: utilizza un device tun*tap* come nodo della rete;

- vxlan: interconnette due, o più, nodi alla vxlan.
- udp: udp tunneling.
- vxvde: Virtual Extensible VDE 3.2.1.
- vxvdex: Distributed Private Network Namespaces 3.2.2.

3.2.1 Virtual Extensible VDE: VXVDE

VXVDE è definita come: *"la fusione evolutiva della filosofia VXLAN"* [8].

Virtual Extensible LAN (VXLAN) nasce dall'esigenza di scalabilità dei grandi ecosistemi di Cloud Computing, non più soddisfatto dallo standard VLAN originale (802.1Q). Virtual Extensible VDE (VXVDE) va a implementare questo nuovo protocollo di comunicazione, ma con un valore aggiunto.

VXVDE è una variante del classico VDE visto che non richiede né demoni né switch virtuali. È una rete Ethernet "switchless" basata sul metodo IP multicast [33].
Nel meccanismo di comunicazione ogni nodo della rete diventa attore in un ambiente che supporta fino a 16 milioni di reti logiche per indirizzo IP multicast e per numero di porta [8].
L'unica configurazione richiesta da VXVDE è un indirizzo IP multicast a cui inviare i pacchetti Broadcast, Unknown unicast and Multicast (BUM).
Tutte le informazioni riguardanti la topologia della rete sono contenute all'interno dei nodi. Quando il modulo VXVDE riceve un pacchetto salva l'associazione tra indirizzo MAC e posizione del mittente. Pertanto, non è necessaria alcuna riconfigurazione durante la migrazione di un nodo, poiché l'architettura di VXLAN è completamente distribuita. Se l'indirizzo del destinatario è già noto, VXVDE invia pacchetti UDP non-BUM senza l'header specifico di VXLAN. Dunque, un grande trasferimento di dati tra due estremità della rete avviene quasi interamente come traffico UDP end-to-end. Al massimo, il primo pacchetto deve essere inviato utilizzando la connessione multicast [8].

VXVDE, dunque, presenta delle notevoli prestazioni e un elevato bandwidth di trasmissione [8]. Tuttavia, esso non implementa alcuna protezione delle reti virtuali; qualsiasi utente, autorizzato a eseguire processi sugli host del cluster, è in grado di allacciarsi a una qualunque rete VXVDE. In parole povere, non esiste alcun controllo di accesso [7].

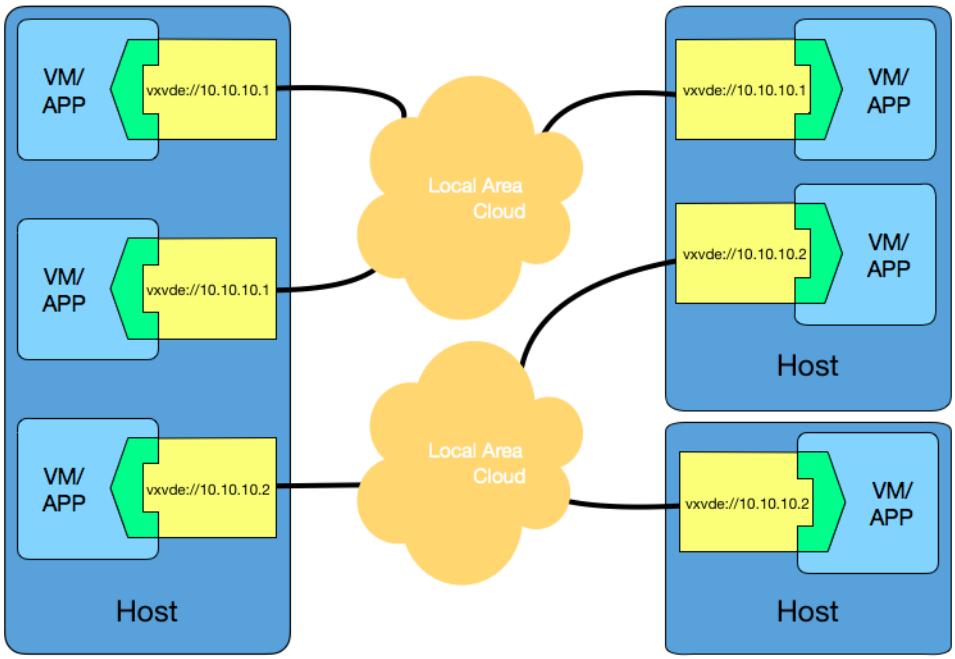

Figura 3.3: Virtual Extensible VDE: VXVDE [9]

3.2.2 Distributed Private Network Namespaces: VXVDEX

I data center in grado di offrire macchine virtuali o servizi Process-as-a-Service (PaaS), presentano spesso una tipologia eterogenea di utenti. In queste infrastrutture convivono utenti ignari della topologia della rete (esclusivamente delegata agli amministratori di sistema) e utenti con i privilegi di lanciare VM, creare Namespace e costruire reti private.
In questo scenario è ragionevole pensare all'impiego di VXVDE come supporto alla rete. Tuttavia, come preannunciato in precedenza, VXVDE non implementa alcun controllo di accesso.

VXVDEX è stata la soluzione proposta dal team VDE per far fronte a questa problematica. Esso è in grado di proteggere la rete condivisa negando l'accesso diretto alle reti virtuali. In un certo senso, svolge lo stesso ruolo del bus di sistema di un computer; nasconde il livello di trasporto dei pacchetti agli utenti in modo tale da applicare un filtraggio del traffico illecito [7]. Il modulo Kernel vxvde.ko fornisce un nuovo protocollo di comunicazione identificato dal flag AF_VXVDEX. Ogni utente è assegnatario di una Virtual Network Instance (VNI) e di un indirizzo IP multicast direttamente associati al suo *GroupID*. Tutti i classici meccanismi di comunicazione vengono filtrati dal modulo verificando che l'*effective GID* del processo sia compatibile con la rete richiesta. Nel caso ci fosse un'incongruenza degli identificatori, i pacchetti verrebbero semplicemente scartati

Ad esempio: un processo con *effective GID* pari a 1000 avrà accesso solamente alle reti identificate dagli indirizzi multicast come 239.0.3.232 o 225.0.3.232 (3 * 256 + 232) in IPv4 e ff05::3e8 o ff03::1234:13e8 in IPv6 [7].

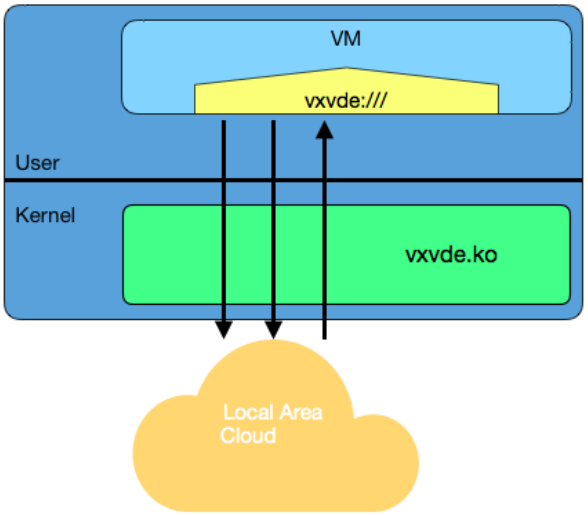

Figura 3.4: VXVDEX [7]

Capitolo 4

Virtual Distributed Container

Le due tecnologie, ampiamente discusse in precedenza, presentano diverse analogie strutturali e filosofiche. Modularità, scalabilità, estensibilità e interoperabilità sono concetti che trovano spazio sia in Docker che in VDE.
Osservando con attenzione le reti supportate da Docker è possibile notare le molte similitudini funzionali che tutte hanno con uno o più componenti VDE. Ad esempio Bridge, Macvlan e VXVDE oppure Open vSwitch e VDE Switch. Reti come Overlay e Weave Net non hanno controparti VDE in scala 1:1 ma possono essere costruite a partire dalle stesse. Le VPN VDE realizzate da strumenti come ssh possono far fronte alle Weave Nets, come la stessa LAN/VXLAN VDE è in grado di costruire un overlay tra i Docker Engine distribuiti.
A fronte di queste osservazioni è nata l'idea di una rete semplice e compatta che potesse unificare le preesistenti senza perdita di performance. Virtual Distributed Container (VDC) è il risultato ottenuto dall'impiego della rete VDE come supporto all'interoperabilità dei container Docker.

4.1 Stato dell'arte

Dalle ricerche preliminari è emerso come già esistesse un supporto VDE per Docker sotto il nome di docker-vde-plugin [34]. Tuttavia, dopo un'attenta analisi sono emerse diverse problematiche. Il plugin offre supporto solo ad alcune funzionalità del vecchio VDE2, in particolare esclusivamente alle reti basate su switch. Per quanto riguarda l'aspetto implementativo, il codice si limita semplicemente ad avviare delle routine che eseguono comandi da shell: istanziano dispositivi tun*tap* e avviano il vecchio plug VDE: vde_plug2*tap*. L'obsolescenza, la scarsa efficienza e il ridotto supporto offerto hanno contribuito all'idea di una reimplementazione radicale del plugin.

4.2 Il plugin

L'attore principale di VDC è vde_plug_docker, un Docker Network Plugin (sezione 2.3) con il compito di fornire il supporto VDE ai container. Vde_plug_docker è un demone out-of-process

che deve essere avviato sul medesimo host del Docker Engine affinché venga individuato dal Discovery (sezione 2.3.1).
L'utente può avviare container connessi attraverso VDE, semplicemente indicando "vde" come driver di rete durante la creazione di un network Docker. Si noti che il plugin ha l'incarico di fornire il punto di accesso alla rete. La costruzione e la gestione di quest'ultima è al di fuori delle sue competenze. Resta dunque all'utente o agli amministratori di sistema il compito di creare l'infrastruttura necessaria.
È possibile, inoltre, edificare reti eterogenee formate da container, VM, IoT e gli stessi host.

4.2.1 La Struttura

La struttura del plugin è formata essenzialmente di due componenti: *Driver* e *Plugger*.

Il Driver non è un oggetto visibile all'utente, ma fornisce l'effettiva implementazione della rete. Infatti, riceve le richieste dal Docker Engine ed è incaricato dell'intera gestione dei sandbox, nonché dell'assegnamento degli endpoint ai container. Nella logica del driver ogni rete è identificata dal NetworkID fornito da Docker e dall'URL VDE (sezione 3.2).

Gli endpoint sono associati alle reti con il medesimo meccanismo e contengono le informazioni del nodo: l'interfaccia di rete, l'indirizzo IPv4/IPv6, l'indirizzo MAC e l'ID del sandbox di appartenenza.

Il driver si avvale di un mutex per gestire le richieste concorrenti che necessitano di un accesso in scrittura ai dati, ad esempio: la creazione di una nuova rete o di un nuovo endpoint. Tutte le altre richieste, di sola lettura, vengono elaborate concorrentemente a patto che non ci sia nessun "writer" con accesso alla struttura dati.

Inoltre, il Driver tiene traccia delle reti attive servendosi di un file .json e offre un'interfaccia per il debug che mostra agli amministratori i log delle richieste ricevute.

Il Plugger funge da interprete tra il container e la rete. Il suo compito è di gestire il flusso di dati tra i due estremi: i pacchetti provenienti dal dispositivo di rete del container vengono immessi sulla rete VDE, viceversa i pacchetti provenienti dalla rete vengono inviati al container.

Un plugger è un thread associato esclusivamente a un endpoint. Esso opera in modo del tutto indipendente dal Driver genitore ma con un legame indissolubile con la vita del container. Quando quest'ultimo verrà deallocato il Driver comunicherà al plugger la fine del suo operato e ne attenderà la terminazione.

4.2.2 Esempi di applicazioni

In questa sezione saranno illustrati tre esempi d'infrastrutture VDC. In primo luogo si vedrà come realizzare una rete omogenea di container, successivamente si tenterà di estendere la rete anche alle VM e in ultimo verrà mostrata una rete distribuita su più host non locali. L'unico

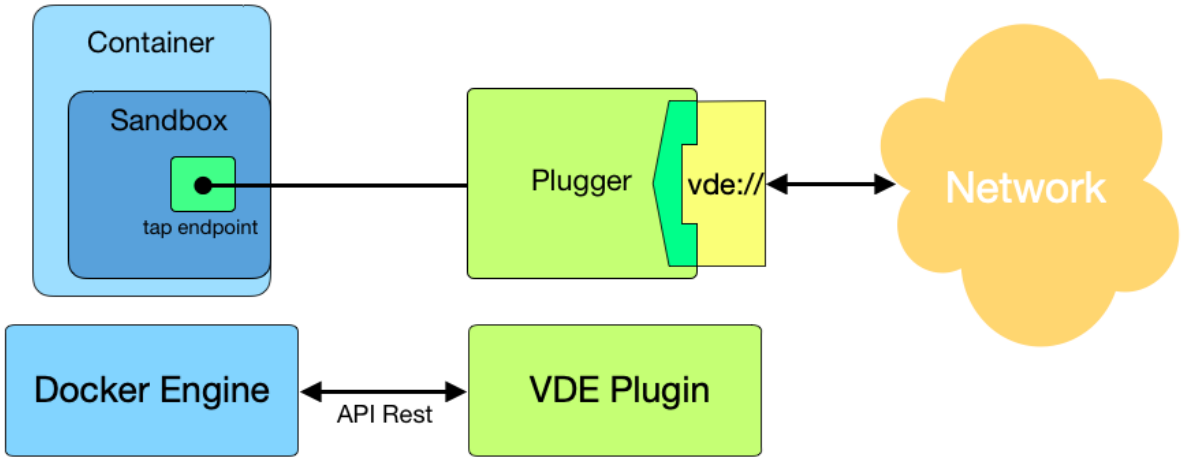

Figura 4.1: vde_plug_docker

prerequisito richiesto è la presenza della libvdeplug sulla macchina host, oltre ovviamente a Docker e vde_plug_docker [42].

Esempio 1: rete omogenea di container

Creazione di una rete VXVDE con IP multicast 239.1.2.3:

```
# docker network create --driver vde --subnet 10.10.0.1/24 \
-o sock=vxvde://239.1.2.3 vxvdenet
```

Avvio di due container sulla rete *vxvdenet*.

```
Container 1:
# docker run -it --net vxvdenet debian bash
# ip addr
2: vde0: .. inet 10.10.0.2/24 brd 10.10.0.255 scope global vde0 ...

Container 2:
# docker run -it --net vxvdenet debian bash
# ip addr
2: vde0: .. inet 10.10.0.3/24 brd 10.10.0.255 scope global vde0 ...

# ping 10.10.0.2
PING 10.10.0.2 (10.10.0.2) 56(84) bytes of data.
64 bytes from 10.10.0.2: icmp_seq=1 ttl=64 time=0.573 ms
64 bytes from 10.10.0.2: icmp_seq=2 ttl=64 time=0.373 ms
```

```
64 bytes from 10.10.0.2: icmp_seq=3 ttl=64 time=0.293 ms
64 bytes from 10.10.0.2: icmp_seq=4 ttl=64 time=0.365 ms
64 bytes from 10.10.0.2: icmp_seq=5 ttl=64 time=0.302 ms
```

Gli indirizzi IP 10.10.0.2/3 vengono assegnati ai container in automatico dal driver IPAM di Docker (sezione 2.3), sulla base del pool di indirizzi deciso al momento della creazione della rete (10.10.0.1/24). È possibile assegnare uno specifico indirizzo usando il parametro `--ip *.*.*.*` .
Si noti inoltre, la coerenza di quanto detto in precedenza sulla rete VXVDE. Il primo pacchetto inviato dal container 2 presenta un tempo nettamente maggiore (0.573 ms) rispetto ai successivi. I container non conoscono a priori i corrispettivi indirizzi, dunque il primo pacchetto è inviato come pacchetto BUM e solo successivamente si instaura una connessione punto punto.

Esempio 2: estendere la rete alle VM

Tenendo valido l'esempio precedente si aggiungerà un nodo alla rete VXVDE rappresentato da una macchina kvm.

```
$ kvm ... -net nic,macaddr=52:54:00:11:22:11 -net vde,sock=vxvde://239.1.2.3
# ip link set eth0 up
# ip addr add 10.10.0.42/24 dev eth0
# ping 10.10.0.2 [container 1]
PING 10.10.0.2 (10.10.0.2) 56(84) bytes of data.
64 bytes from 10.10.0.2: icmp_seq=1 ttl=64 time=0.674 ms
64 bytes from 10.10.0.2: icmp_seq=2 ttl=64 time=0.252 ms
64 bytes from 10.10.0.2: icmp_seq=3 ttl=64 time=0.332 ms
64 bytes from 10.10.0.2: icmp_seq=4 ttl=64 time=0.472 ms
```

Esempio 3: rete distribuita su più host

Si considerino due host geograficamente distribuiti. Per il primo terremo valida la configurazione degli esempi precedenti nel secondo si avvierà un nuovo container collegato alla medesima rete.

Host 1: overlay VXVDE:

```
vde_plug vxvde://239.1.2.3 cmd://"ssh remote.host2 vde_plug vxvde://239.1.2.3"
```

Host 2: avvio del container

```
# docker network create --driver vde --subnet 10.10.0.1/24 \
-o sock=vxvde://239.1.2.3 vxvdenet
# docker run -it --net vxvdenet --ip 10.10.0.6 debian bash
# ip addr
2: vde0: .. inet 10.10.0.6/24 brd 10.10.0.255 scope global vde0 ...
```

```
# ping 10.10.0.2 [container 1]
PING 10.10.0.2 (10.10.0.2) 56(84) bytes of data.
64 bytes from 10.10.0.2: icmp_seq=1 ttl=64 time=110 ms
64 bytes from 10.10.0.2: icmp_seq=2 ttl=64 time=3.21 ms
64 bytes from 10.10.0.2: icmp_seq=3 ttl=64 time=2.16 ms
64 bytes from 10.10.0.2: icmp_seq=4 ttl=64 time=4.74 ms
# ping 10.10.0.42 [kvm]
PING 10.10.0.42 (10.10.0.42) 56(84) bytes of data.
64 bytes from 10.10.0.42: icmp_seq=1 ttl=64 time=22.74 ms
64 bytes from 10.10.0.42: icmp_seq=2 ttl=64 time=6.53 ms
64 bytes from 10.10.0.42: icmp_seq=3 ttl=64 time=9.42 ms
64 bytes from 10.10.0.42: icmp_seq=4 ttl=64 time=8.28 ms
```

Gli esempi presentati mostrano solo una delle possibili configurazione della rete. Inoltre, si fa riferimento esclusivamente a una rete VXVDE, ma ciò non toglie la possibilità di costruirne una basata su switch o la combinazione di entrambe.

4.3 Implementazione

In questa sezione si analizzeranno gli aspetti implementativi di vde_plug_docker. Si ricorda che il plugin è un demone multithread out-of-process, ovvero viene avviato indipendentemente da Docker, o quasi (appendice A).
L'installazione del plugin può avvenire in due modi: attraverso una pura installazione sul computer host come system-daemon o all'interno di un particolare container con privilegi di CAP_NET_ADMIN[1].
Il plugin si serve di alcune librerie: netlink per la gestione dei dispositivi di rete, go-plugins-helpers per implementare il protocollo CNM e infine di libvdeplug per la connessione alla rete VDE. Quest'ultima, in caso di un'installazione system-daemon, deve essere presente sul computer host.
Il plugin sfrutta un socket Unix per la comunicazione con il Docker Engine. Le richieste sono di tipo REST e vengono accolte dalla libreria go-plugins-helpers per poi essere soddisfatte all'interno del Driver.

4.3.1 Il linguaggio

I file sorgenti posso essere di due tipi: *.go o *.c visto che il plugin è sviluppato in parte in linguaggio GO [23] e in parte in linguaggio C. Tale implementazione è dettata dal fatto che la libvdeplug è disponibile esclusivamente in C. Inoltre, nonostante GO implementi i comandi CGO [22] per integrare codice C nei suoi file sorgenti, la stesura si è rilevata artificiosa.

[1] Un'immagine pre-built del plugin è disponibile su Docker Hub: `docker plugin install phocs/vde`.

Dunque, vdeplug.c definisce la routine del thread "plugger" richiamando le funzioni di libreria VDE. I file sorgenti *.go definiscono il server centrale e la comunicazione con il Docker Engine.

GO

Go è un linguaggio open-source proposto da Google nel 2009 pensato per realizzare applicativi e software multipiattaforma.
Scendendo nel dettaglio, Go è un linguaggio compilato, a tipizzazione statica, memory-safe e con meccanismo di garbage-collection [31].
Per la gestione degli oggetti Go utilizza le interfacce, implementando un meccanismo che ricorda le classi astratte del C++. Le interfacce sono un punto fondamentale nella realizzazione di un plugin Docker. Infatti, go-plugins-helpers definisce un'interfaccia per ogni tipologia di plugin, ed è richiesto allo sviluppatore d'implementarne i metodi.

4.3.2 Ciclo di vita del plugin

L'istanza di un Container Network Model segue un ciclo di vita ben preciso suddiviso in step [14]. Dal punto di vista del plugin ogni step non è altro che una richiesta evasa a suo carico dal Docker Engine.
Con un'attenta osservazione è facile notare un'ulteriore suddivisione dell'iter: la vita di un network è l'anello più esteso del ciclo e contiene al suo interno tutte le subroutine associate ai singoli container.
Di seguito saranno analizzati i singoli step che il plugin esegue per la costruzione, la messa in opera, e la distruzione di un network.
Si ricorda che il server può adempiere anche a richieste concorrenti purché non sia necessario alterare lo stato dei dati. In tal caso entra in gioco il meccanismo di mutua esclusione.

1. Create Network

Il comando `docker network create -d vde -o sock=vxvde:// vdenetwork` avvia la richiesta per la creazione dell'istanza *vdenetwork* all'interno del driver. Essa è accompagnata dai seguenti parametri:

- NetworkID: fornito dal Docker Engine identifica la rete all'interno della struttura dati;
- IPv4Data/IPv6Data (opzionale): contiene il pool d'indirizzi della rete. IPv4Data può essere indicato nel flag `--subnet *.*.*.*/*` altrimenti Docker provvederà a fornirne uno. IPv6 (disattivato di default) è supportato dal plugin;
- -o sock (obbligatorio): indica la rete VDE di riferimento (sezione 3.2);
- -o if (opzionale): può essere usato per indicare il nome dell'interfaccia all'interno del container; vde0 default.

Vde_plug_docker si serve di un dizionario (key: NetworkID, value: NetworkStat) in cui viene salvato lo stato della rete. NetworkStat, oltre ai parametri sopraelencati, contiene la raccolta degli endpoint (key: EndpointID, value: EndpointStat).

2. Create Endpoint

La creazione di un container implica la richiesta di un nuovo endpoint. In questa fase vengono salvate le informazioni dell'interfaccia di rete:

- NetworkID: rete di appartenenza.
- EndpointID: identificatore all'interno della rete.
- IPv4Address/IPv6Address (opzionale): se non indicato (`--ip *.*.*.*`) il driver IPAM fornisce il primo IP libero appartenente al pool d'indirizzi della rete.
- MacAddress (opzionale): se non indicato (`--mac-address 52:54:*:*:*:*`) il plugin genera un indirizzo MAC casuale [40].
- Plugger (privato): threadID del plugger.
- IfName (privato): il nome dell'interfaccia di rete da istanziare sulla macchina host.

3. Join

Join è l'ultimo step prima dell'avvio del container. Il plugin crea un'interfaccia di tipo *tap* nello stack di rete host avente gli attributi (IP, MAC, ecc.) indicati nell'EndpointStat. Successivamente, avvia un nuovo thread (plugger) che inizializza la connessione:

- Esegue una open dell'interfaccia *tap* ottenendone il File Descriptor (fd).
- Avvia la connessione VDE utilizzando il parametro *sock*.
- Comunica al processo genitore se tutto è andato a buon fine.
- Si mette in ascolto (ppoll) sulla rete VDE e sul fd del dispositivo *tap*.

Da questo momento in poi, tutti i pacchetti provenienti dalla rete VDE saranno inviati alla periferica *tap*, viceversa i dati provenienti dal dispositivo verranno immessi nella rete VDE utilizzando le chiamate di libreria libvdeplug.
Il plugin durante questa fase resta in attesa di conferma prima di inviare a Docker gli estremi dell'interfaccia *tap*. A quel punto Docker trasferirà il *tap* all'interno del sandbox del container rendendolo inaccessibile dall'esterno.
Si noti che il thread resterà attivo nel root namespace ma continuerà ad avere visibilità del dispositivo attraverso l'fd.

4. Leave

Quando il container passa dalla fase di running a quella di stop Docker distrugge il sandbox, riposiziona il dispositivo *tap* nel root namespace ed effettua la richiesta Leave. Il plugin, a sua volta, manda un segnale di stop al plugger che provvede: a chiudere la connessione VDE, a chiudere il file descriptor dell'interfaccia *tap* e a terminare l'esecuzione. A questo punto il driver di rete distrugge il dispositivo *tap*.

Successivamente, il container potrebbe tornare in fase di running richiamando la procedura del punto 3.

5. Delete Endpoint

Cancella i dati relativi all'EndpointID ricevuto come parametro. Questa fase segna la terminazione permanente del container.

6. Delete Network

Se non ci sono endpoint attivi il network viene deallocato cancellando tutti i dati che lo rappresentavano.

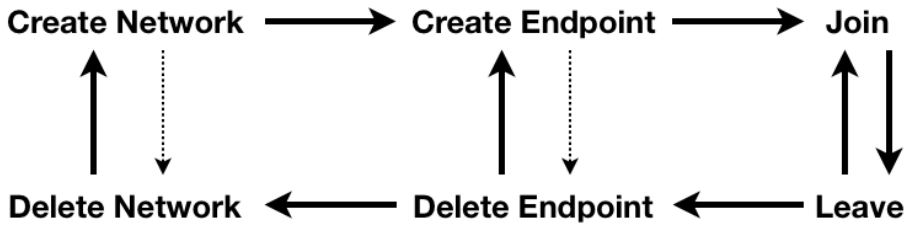

Figura 4.2: CNM Lifecycle [14]

4.4 Analisi delle performance

In molti casi la semplicità d'uso viene realizzata a discapito delle prestazioni, ma questo non è il caso di VDE.
Nell'articolo [8] sono stati esposti alcuni risultati riguardanti le performance di VXVDE a confronto con l'implementazione Kernel del bridge Linux. I risultati, seppur preliminari, mostrano VXVDE con un rendimento di circa il 68% in più.
Considerando che Docker fa largo uso del bridge si è cercato di riprodurre lo stesso test nel contesto presentato in questa tesi. Per motivi logistici è stata utilizzata un'unica macchina host con Linux 4.9.0 e processore Intel i7. Il test prevede l'impiego di 4 container di cui i primi due utilizzano la rete bridge e i restanti sono connessi, sfruttando il plugin, alla rete VXVDE.

Client Setup: `nc -l -p 9999 > /dev/null`

Server: `dd if=/dev/urandom bs=8192 count=1 | nc 10.0.0.2 9999`
Bridge: `8192 bytes (8.2 kB, 8.0 KiB) copied, 0.000216265 s, 37.9 MB/s`
VXVDE: `8192 bytes (8.2 kB, 8.0 KiB) copied, 0.000226726 s, 36.1 MB/s`

Server: `dd if=/dev/urandom bs=8192 count=1024 | nc 10.0.0.2 9999`
Bridge: `8388608 bytes (8.4 MB, 8.0 MiB) copied, 0.0368393 s, 228 MB/s`
VXVDE: `8388608 bytes (8.4 MB, 8.0 MiB) copied, 0.0372533 s, 225 MB/s`

Server: `dd if=/dev/urandom bs=8192 count=102400 | nc 10.0.0.2 9999`
Bridge: `838860800 bytes (839 MB, 800 MiB) copied, 3.34666 s, 251 MB/s`
VXVDE: `838860800 bytes (839 MB, 800 MiB) copied, 3.49121 s, 240 MB/s`

Come si può notare il bridge Linux conserva sempre un piccolo vantaggio, in media circa il 3.2%, rispetto alla rete VXVDE.
Nella trasmissione dei dati sulla rete VXVDE bisogna tener conto dell'overhead introdotto dai due thread plugger, rispettivamente associati ai due container. Oltre al loro oneroso lavoro si aggiungono i tempi di attesa per la schedulazione, che potrebbero divenire rilevanti nel caso in cui la macchina host risultasse affollata di processi.
Per questo motivo, il risultato ottenuto, seppur distante dalle previsioni, resta un ottimo traguardo.

Conclusioni

Nei capitoli iniziali si è visto quanto Docker e VDE si somigliassero nella loro filosofia. Tuttavia, c'è una profonda discrepanza dell'idea di pubblico a cui queste tecnologie si rivolgono.
Docker è uno strumento tanto potente quanto potenzialmente dannoso. Esso porta con se il problema dell'escalation poiché un utente senza alcun privilegio, ma con l'abilitazione a utilizzare Docker, sarebbe in grado di assumere i privilegi root del sistema. Basterebbe avviare un qualunque container che monti il file system della macchina host come volume[2]. Per questo motivo Docker si rivolge esclusivamente agli amministratori e si tiene alla larga dal mondo degli utenti sconsigliando fortemente di assegnarvi i premessi.
Viceversa VDE è un progetto nato dallo spirito di sperimentazione in campo accademico. È rivolto principalmente agli utenti e non richiede alcun privilegio, o quasi, per costruire reti virtuali distribuite. VDE offre dunque, la possibilità di testare e amministrare in tutta sicurezza un network funzionante che normalmente avrebbe bisogno di privilegi elevati.
Questa differenza di fondo non è passata inosservata durante lo sviluppo del plugin. Vde_plug_docker attualmente opera esclusivamente in modalità root per sottostare ad alcune regole che Docker vi impone: il path nel quale deve essere aperto il socket Unix è protetto e le interfacce di rete destinate ai container devono essere istanziate nel root namespace. La questione è ancora aperta e al momento non ci sono dei piani alternativi.

Per quanto riguarda le performance si potrebbe pensare allo sviluppo di una nuova interfaccia di rete. Tale dispositivo farebbe le veci del dell'interfaccia *tap* e del thread a esso associato, ottenendo così una discreta riduzione dell'overhead. In tal caso l'invio e la ricezione dei pacchetti sarebbero destinati esclusivamente al nuovo device.

Infine, non manca il desiderio di riuscire a integrare la rete VDE nello stesso Docker Engine. Un primo step potrebbe essere il rilascio della libvdeplug nel linguaggio GO il che, quasi sicuramente, renderebbe molto più immediata l'integrazione con l'ecosistema Docker.

[2] `docker run -it -v /etc:/etc debian`

Appendice A

Appendice vde_plug_docker

vde_plug_docker implements the concept of Virtual Distributed Container. It is a Docker Network Plug-in that allows communication between container on the VDE network. The plugin is responsible for providing the access point to the VDE network. The construction and management of the latter it is outside the plugin's competence.

A.1 Dependencies

- go >= 1.7.4
- libvdeplug: https://github.com/rd235/vdeplug4

A.2 DockerHub install

You can install the pre-built image of the plugin from the Docker Hub. It is required:

- Host network access.
- Host /tmp mounted.
- Device access /dev/net/tun.
- Capability: CAP_NET_ADMIN .

```
# docker plugin install phocs/vde
```

A.3 Git Install

In this way the plugin is installed as a system-daemon according to the [Docker Docs].

```
$ cd vde_plug_docker/
$ make
$ sudo make install
```

A.4 Debug

You can also manually run the plugin for debugging and development.

```
$ cd vde_plug_docker/
$ make
$ sudo ./vde_plug_docker --debug
```

A.5 Examples

Connect 2 containers to the VXVDE network

Creating the network:

```
# docker network create -d vde \
  -o sock=vxvde://239.1.2.3 \
  --subnet 10.10.0.1/24 vdenet
```

Running the containers:

```
# docker run -it --net vdenet --ip 10.10.0.2 debian
# docker run -it --net vdenet --ip 10.10.0.3 debian
# ping 10.10.0.2
PING 10.10.0.2 (10.10.0.2) 56(84) bytes of data.
   64 bytes from 10.10.0.2: icmp_seq=1 ttl=64 time=0.573 ms
   64 bytes from 10.10.0.2: icmp_seq=2 ttl=64 time=0.373 ms
   64 bytes from 10.10.0.2: icmp_seq=3 ttl=64 time=0.293 ms
   64 bytes from 10.10.0.2: icmp_seq=4 ttl=64 time=0.365 ms
```

Add a VM to the network

```
$ kvm ... -net nic,macaddr=52:54:00:11:22:11 \
  -net vde,sock=vxvde://239.1.2.3
# ip link set eth0 up
# ip addr add 10.10.0.42/24 dev eth0
# ping 10.10.0.2          [container 1]
PING 10.10.0.2 (10.10.0.2) 56(84) bytes of data.
   64 bytes from 10.10.0.2: icmp_seq=1 ttl=64 time=0.674 ms
   64 bytes from 10.10.0.2: icmp_seq=2 ttl=64 time=0.252 ms
   64 bytes from 10.10.0.2: icmp_seq=3 ttl=64 time=0.332 ms
   64 bytes from 10.10.0.2: icmp_seq=4 ttl=64 time=0.472 ms
```

Bibliografia

[1] Docker. `https://www.docker.com`.

[2] Virtual square. `https://github.com/virtualsquare`.

[3] Ieee standards for local and metropolitan area networks: Virtual bridged local area networks. *IEEE Std 802.1Q, 2003 Edition (Incorporates IEEE Std 802.1Q-1998, IEEE Std 802.1u-2001, IEEE Std 802.1v-2001, and IEEE Std 802.1s-2002)*, pages 1–322, May 2003.

[4] D. Alam, M. Zaman, T. Farah, R. Rahman, and M. S. Hosain. Study of the dirty copy on write, a linux kernel memory allocation vulnerability. In *2017 International Conference on Consumer Electronics and Devices (ICCED)*, pages 40–45, July 2017.

[5] Linux Containers. Infrastructure for container projects. `https://linuxcontainers.org`.

[6] R. Davoli. Vde: virtual distributed ethernet. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220, Feb 2005.

[7] R. Davoli. Vxvdex: Internet of threads and networks of namespaces. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2017.

[8] R. Davoli and M. Goldweber. Vxvde: A switch-free vxlan replacement. In *2015 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, Dec 2015.

[9] Renzo Davoli. Nfv a' la vde way. `https://archive.fosdem.org/2018/schedule/event/vde/attachments/slides/2090/export/events/attachments/vde/slides/2090/nfvvde_slides.pdf`.

[10] Renzo Davoli. vdeplug4. `https://github.com/rd235/vdeplug4`.

[11] Renzo Davoli and Michael Goldweber. Virtual square (v2) in computer science education. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation*

and Technology in Computer Science Education, ITiCSE '05, pages 301–305, New York, NY, USA, 2005. ACM.

[12] Docker. Go plugins helpers. `https://github.com/docker/go-plugins-helpers`.

[13] Docker. libnetwork. `https://github.com/docker/libnetwork`.

[14] Docker. libnetwork: Cnm lifecycle. `https://github.com/docker/libnetwork/blob/master/docs/design.md`.

[15] Docker. Docker delivers native multi-host networking to advance distributed application portability. `https://www.docker.com/docker-news-and-press/docker-delivers-native-multi-host-networking-advance-distributed-application`, Jun 2015.

[16] Docker Docs. AUFS. `https://docs.docker.com/storage/storagedriver/aufs-driver/`.

[17] Docker Docs. Docker Engine. `https://docs.docker.com/engine/docker-overview/#docker-engines`.

[18] Docker Docs. Docker Plugin API. `https://docs.docker.com/engine/extend/plugin_api/#what-plugins-are`.

[19] Docker Docs. Network drivers. `https://docs.docker.com/network/`.

[20] Rajdeep Dua, Vaibhav Kohli, and Santosh Kumar Konduri. *Learning Docker Networking*. Packt Publishing Ltd, 2016.

[21] Linux Foundation. Open vswitch. `http://www.openvswitch.org/`.

[22] Google. Cgo. `https://golang.org/cmd/cgo/`.

[23] Google. The go programming language. `https://golang.org`.

[24] Google. gVisor. `https://github.com/google/gvisor`, 2018.

[25] Tejun Heo. Control group v2. `https://www.kernel.org/doc/Documentation/cgroup-v2.txt`, 2015.

[26] Datadog Inc. 8 surprising facts about real docker adoption. `https://www.datadoghq.com/docker-adoption/`, Jun 2018.

[27] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. `https://lwn.net/articles/531114/`, 2018.

[28] Linux manual page. cgroups. `http://man7.org/linux/man-pages/man7/cgroups.7.html`, 2018.

[29] Linux manual page. Namespaces. `http://man7.org/linux/man-pages/man7/namespaces.7.html`, 2018.

[30] Linux manual page. Veth. `http://man7.org/linux/man-pages/man4/veth.4.html`, 2018.

[31] S. Martini and M. Gabbrielli. *Linguaggi di programmazione: principi e paradigmi*. McGraw-Hill Italia, 2006.

[32] FreeBSD Manual Pages. dpipe. `https://www.freebsd.org/cgi/man.cgi?query=dpipe&manpath=1`, 2018.

[33] Larry L Peterson and Bruce S Davie. *Reti di calcolatori*. Apogeo Editore, 2008.

[34] Will Rouesnel. docker-vde-plugin. `https://github.com/wrouesnel/docker-vde-plugin.git`.

[35] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[36] J. E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[37] Virtual Square. Vde2. `https://github.com/virtualsquare/vde-2`.

[38] Virtual Square. vuos. `https://github.com/virtualsquare/vuos`.

[39] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. *Computer security: principles and practice*. Pearson Education, 2012.

[40] Linus Torvalds. Linux kernel source tree: eth_random_addr. `https://github.com/torvalds/linux/blob/master/include/linux/etherdevice.h`.

[41] The UNIX and Linux Forums. wirefilter. `https://www.unix.com/man-page/debian/1/wirefilter/`, 2018.

[42] Alessio Volpe. vde_plug_docker. `https://github.com/phocs/vde_plug_docker.git`.

[43] Weave Works. Introducing weave net. `https://www.weave.works/docs/net/latest/overview/`.