Documentazione VDE Plugin Docker

Fabio Mirza

Contents

1 Installazione **2**
- 1.1 Dipendenze . . . 2
- 1.2 Installazione come out-of-process Daemon . . . 2

2 Architettura del Plugin **3**
- 2.1 Ciclo di vita . . . 3
- 2.2 Strutture dati . . . 4
  - 2.2.1 Struttura dati per l'Endpoint . . . 4
  - 2.2.2 Struttura dati per il Network . . . 5
  - 2.2.3 Struttura dati per il Driver . . . 5
- 2.3 Metodi . . . 5
  - 2.3.1 Metodi per gli Endpoint . . . 6
  - 2.3.2 Metodi del Driver . . . 6
  - 2.3.3 Metodi del Datastore . . . 8
  - 2.3.4 Metodi C . . . 8

1 Installazione

Come suggerito dalla documentazione Docker, i plugin vanno eseguiti come demoni out-of-process. In queste implementazione, tali demoni sono definiti come systemd Unit, in particolare attraverso una unit `.service` e una unit `.socket`.

1.1 Dipendenze

Il plugin Docker per VDE richiede le seguenti dipendenze nella macchina host:

- golang $\geq$ 1.7.4
- vdeplug4
- Docker

1.2 Installazione come out-of-process Daemon

Clonare la repository

```
$ git clone https://github.com/mapokko/vde_plug_docker_v2.git
$ cd /vde_plug_docker_v2
```

Installare i go packages richiesti dal plugin nella macchina host

```
$ go mod tidy
```

Creare l'eseguibile

```
$ go build .
```

Copiare l'eseguibile, installare il servizio ed eseguirlo

```
$ sudo make install
```

I systemd Unit sono presenti nella cartella `/service`. In particolare, `vde_plug_docker.service` indica la posizione dell'eseguibile mentre `vde_plug_docker.socket` indica la posizione del socket. Quest'ultima e' la posizione suggerita dalla documentazione Docker.

E' inoltre fornita un immagine in questo Link con tutte le dipendenze gia' installate dove e' possibile eseguire i seguenti comandi per sperimentare con `vde_plug_docker`. Bisogna accedere con utente `user` e password `virtualsquare`. Per eseguire la VM e' necessario che nel sistema sia installato `qemu-system-x86`.

Eseguire la VM e accedere all'account

```
$ qemu-system-x86_64 -enable-kvm -smp $(nproc) -m 2G -monitor stdio -cpu host  \
-netdev type=user,id=net,hostfwd=tcp::2222-:22 -device virtio-net-pci,netdev=net \
-drive file=$(echo debian-sid-v2-amd64-daily-20230221-1298.qcow2)
```

Creare la rete indicando `vde` come driver, `vxvde://239.1.2.3` come il socket, `vd` come prefisso per i TAP e `vdenet` come nome del network

```
$ sudo docker network create -d vde -o sock=vxvde://239.1.2.3 -o if=vd \
--subnet 10.10.0.1/24 vdenet
```

Creare un nuovo container collegato alla rete `vdenet` con indirizzo 10.10.0.2

```
$ sudo docker run -it --net vdenet --ip 10.10.0.2 debian &
```

Ora possiamo verificare che la rete e l'endpoint del container siano stati costruiti correttamente guardando nel `datastore`

```
$ cat /etc/docker/vde_plug_docker.json
```

Creiamo ora un interfaccia tap nella macchina host

```
$ sudo ip tuntap add dev tap0 mode tap
$ sudo ip addr add 10.10.0.5/24 dev tap0
$ sudo ip link set tap0 up
```

Usiamo un plug VDE per collegare il TAP creato alla rete dei container

```
$ vde_plug tap://tap0 vxvde://239.1.2.3 &
```

Verifichiamo che la comunicazione sia attiva

```
$ ping -I 10.10.0.5 10.10.0.2
```

2 Architettura del Plugin

Come descritto nella tesi relativa a `vde_plug_docker`, i plugin Docker che vogliono implementare delle funzionalita' di networking per i container devono rispettare l'API del *Container Network Model* (CNM) delineato dalla libreria *libnetwork*. In particolare, il plugin fornisce un Driver che deve esporre una API contenente l'implementazione specifica per la gestione del network di cui si occupa (in questo caso una rete VDE). Il CNM definisce i seguenti oggetti:

- **NetworkController**: si tratta dell'entrypoint per *libnetwork* attraverso cui Docker gestisce i network. *libnetwork* supporta molteplici Driver (tra cui network plugin) e NetworkController permette di associare i Driver ai network che deve gestire.
- **Driver**: consiste nella effettiva implementazione delle funzioni che gestiscono il network. I Driver possono essere built-in (es. Bridge) o remoti (i plugin). Ai Driver possono essere inoltre passati dei parametri. Nel caso di `vde_plug_docker` sono 2: `sock` e `if`, che vedremo meglio successivamente.
- **Network**: consiste nell'oggetto network CNM, ovvero un gruppo di endpoint in grado di comunicare direttamente tra loro. I network sono gestiti dal NetworkController che, come detto in precedenza, espone le API che si appoggiano al Driver associato al network.
- **Endpoint**: rappresenta un endpoint per un servizio esposto da un container. Ogni endpoint appartiene ad un singolo network. Nonostante un container possa avere piu' endpoint, `vde_plug_docker` associa a ogni container un solo endpoint, al quale e' a sua volta associato il thread del `vde_plug` che connette l'interfaccia TAP dell'endpoint alla rete VDE esterna.
- **Sandbox**: A ogni container e' associato una sandbox, che a sua volta racchiude gli endpoint, potenzialmente provenienti da network differenti. Libnetwork si occupa di spostare nella sandbox i dispositivi TAP create nella macchina host dal Driver di `vde_plug_docker` e associati agli endpoint.

2.1 Ciclo di vita

Docker interagisce con i CNM Object per gestire i network dei suoi container attraverso i seguenti passaggi:

1. L'oggetto `NetworkController` viene creato ed eventualmente configurato con i Driver e opzioni per esso.
2. Viene creato un Network a cui viene associato un nome e un networkType. In base al networkType viene individuato il Driver corrispondente. Una volta associato, tutte le operazioni su questo Network saranno gestite dal Driver. Nel caso di Docker, il comando che esegue queste operazioni e' il seguente

   ```
   $ docker network create -d vde -o sock=vxvde://234.0.0.1 -o if=new \
   --subnet=20.0.0.1/24 newvde
   ```

In questo caso sta venendo creato un network con driver `vde` a cui vengono passati le opzioni `sock` impostato a `vxvde://234.0.0.1` e `if` impostato a `new`. La nuova rete si chiamera' newvde e suoi endpoint avranno interfaccie TAP i cui nomi cominceranno con il prefisso `new`. La struttura dati del Driver per il network verra', da questo momento in poi, regolarmente salvato nel `datastore`.

3. Attraverso l'API `controller.CreateEndpoint()` vengono creati gli endpoint relativi al Network attraverso una relativa chiamata di funzione al driver `Driver.CreateEndpoint()`. Notare che in questo momento, l'endpoint non e' ancora associato a nessun container, ma i suoi dati sono salvati nel `datastore`.

4. Una volta creato l'endpoint, esso viene associato ad un container attraverso l'API `endpoint.Join()` che invoca la realtiva funzione nel driver `Driver.Join()`. Nel caso di `vde_plug_docker`, questa chiamata si occupa di creare un dispositivo TAP, associarlo al socket VDE fornito durante la creazione del network attraverso un plug VDE e mantenere il PID di tale plug nel `datastore`, cosi' che esso possa essere terminato quando l'endpoint non e' piu' in utilizzo. Inoltre e' proprio in questa fase che LibNetwork si occupa di spostare il TAP dell'endpoint nella sua relativa Sandbox.

5. Quando un container viene fermato, i suoi endpoint chiamano la API `endpoint.Leave()` che, in modo analogo alle precedenti, chiama `Driver.Leave()`. Nel caso di `vde_plug_docker`, il driver termina il VDE plug che associava il dispositivo TAP alla rete VDE, per poi eliminare il TAP stesso associato all'endpoint. Notiamo che i dati relativi all'endpoint sono ancora presenti nel datastore nel caso il container venisse fatto ripartire successivamente

6. Quando invece l'endpoint stesso viene eliminato, i dati relativi ad esso vengono anche rimossi dal datastore.

7. Infine, viene chiamato `network.Delete()` per eliminare il network stesso. In questo caso quello specifico network verra' rimosso dal `datastore`.

Notiamo che l'implementazione di queste interazioni e' resa molto piu' semplice dalla SDK fornita da Docker chiamata *go-plugins-helpers*, il quale rispecchia le esigenze del CNM e permette di mettere in contatto il Docker Engine con il Driver, limitandosi a definire le interfaccie per un Network plugin.

2.2 Strutture dati

All'interno del plugin, sono state definite e utilizzate diverse strutture dati per aiutare nella gestione dei network, degli endpoint e delle loro interazioni. Notiamo che tutte le strutture dati descritte sono fondamentalmente raccolte all'interno del Driver stesso, il quale viene salvato in formato JSON proprio nel file `datastore`. Cio' e' fatto in modo tale da recuperare le informazioni sui network e sugli endpoint gia' esistenti nel caso di riavvio del daemon del plugin.

La notazione che segue ogni campo delle strutture e' un meccanismo che permette di associare facilmente una struttura JSON a partire da oggetti Go.

2.2.1 Struttura dati per l'Endpoint

```
type EndpointStat struct {
        Plugger         uintptr `json:"Plugger"`
        IfName          string  `json:"IfName"`
        SandboxKey      string  `json:"SandboxKey"`
        IPv4Address     string  `json:"IPv4Address"`
        IPv6Address     string  `json:"IPv6Address"`
        MacAddress      string  `json:"MacAddress"`
}
```

Come si puo' intuire dal nome, questa struttura dati racchiude le informazioni riguardo gli endpoint. Durante la descrizione dei metodi degli Endpoint verra' spiegato quando questa struttura e' utilizzata, al momento ci si limita a dire:

- **Plugger**: Come anticipato, il Plugger mantene il PID del `plug_vde` che collega il dispositivo TAP dell'endpoint alla rete VDE.

- **IfName**: Il nome dell'interfaccia TAP.
- **SandboxKey**: Il path assoluto dove si trova il sandbox a cui e' associto l'endpoint.
- **IPv4Address**: Indiritto IPv4 dell'endpoint. Obbligatorio.
- **IPv6Address**: Indirizzo IPv6 dell'endpoint. Opzionale.
- **MacAddress**: Indirizzo MAC dell'endpoint. Viene generato nel caso non venga fornito da Docker.

Osserviamo come la struttura degli Endpoint abbia anche dei metodi associati ad esso, in modo similare ai metodi di una classe C++. Tali funzioni verrano esplorate in seguito.

2.2.2 Struttura dati per il Network

```
type NetworkStat struct {
        Sock            string `json:"Sock"`
        IfPrefix        string `json:"IfPrefix"`
        IPv4Pool        string `json:"IPv4Pool"`
        IPv4Gateway     string `json:"IPv4Gateway"`
        IPv6Pool        string `json:"IPv6Pool"`
        IPv6Gateway     string `json:"IPv6Gateway"`
        Endpoints       map[string]*endpoint.EndpointStat `json:"Endpoints"`
}
```

Si tratta della struttura dati associata ai Network

- **Sock**: il socket VDE a cui devono connettersi gli endpoint di questa rete.
- **IfPrefix**: definito alla creazione del network, si tratta del prefisso che devono avere i dispositivi TAP associati agli endpoint di questa rete.
- **IPv4Pool**: Subnet in formatio CIDR che rappresenta un network segment per indirizzi IPv4.
- **IPv4Gateway**: Gateway IPv4 per la rete.
- **IPv6Pool**: Subnet in formato CIDR che rappresenta un network segment per indirizzi IPv6.
- **IPv6Gateway**: Gateway IPv6 per la rete.
- **Endpoints**: Una map, ovvero coppie chiave-valore, che racchiunde gli endpoint associati a questo network. Come chiave si utilizano gli EndpointID forniti dal Docker, mentre il valore e' un instanza della struttura per gli endpoint definita prima.

2.2.3 Struttura dati per il Driver

```
type Driver struct {
        mutex       sync.RWMutex                `json:"-"` // ignore
        Networks    map[string]*NetworkStat      `json:"Networks"`
}
```

- **mutex**: Mutex utilizzato per prevenire race conditions durante la manipolazione della struttura dati. In particolare, poiche' il `datastore` non e' altro che la struttura dati del Driver, ogni volta che esso viene modificato, si utilizza questo mutex.
- **Network**: Analogo a Endpoints della struttura precedente, si tratta di coppie chiave-valore dove la chiave e' un NetworkID fornito da Docker e il valore e' un instanza della struttura dati per i Network.

Come per gli Endpoint, anche la struttura del Driver ha dei metodi associati. Essi sono proprio i metodi richiesti da *go-plugin-helper* per realizzare la comunicazione tra Docker e il plugin.

2.3 Metodi

Si descrivono ora i metodi che permettono di manipolare gli Endpoint e le reti, con particolare enfasi sugli specifici metodi richiesti per implementare un Network plugin.

2.3.1 Metodi per gli Endpoint

```
func NewEndpointStat(r *network.CreateEndpointRequest, IfPrefix string) *EndpointStat
```

Questa funzione non e' un metodo della struttura dati degli Endpoint, ma svolge l'importante funzione di instanziare tale struttura. In particolare, accetta 2 parametri

- **r**: la richiesta di creazione dell'endpoint da parte di Docker. Contiene i dati relativi agli indirizzi IP e MAC.
- **ifPrefix**: il prefisso per gli endpoint della rete a cui questo specifico endpoint si unira'.

Per il momento, `Plugger` dell'endpoint viene lasciato al valore di default 0 poiche' non e' stato ancora instanziato il thread relativo al plug VDE. Inoltre anche `SandboxKey` e' lasciato a default poiche' in questo momento dell'esecuzione l'endpoint non e' ancora collegato ad alcun container.

```
func (this *EndpointStat) LinkAdd() error
```

Questo metodo e' associato ad ogni istanza della struttura `EndpointStat` e si occupa di creare, utilizzando il package netlink, un dispositivo TAP utilizzando `IfName` come nome e associandogli gli indirizzi MAC, IPv4 e IPv6.

```
func (this *EndpointStat) LinkDel() error
```

Duale del precedente, questo metodo si preoccupa dell'eliminazione del dispozitivo TAP relativo a all'endpoint.

```
func (this *EndpointStat) LinkPlugTo(sock string) error
```

Questo metodo si occupa di collegare effettivamente il dispositvo TAP dell'endpoint con la rete VDE. Infatti esso si appoggia alla funzione C `C.vdeplug_join()`, il quale prende il nome del TAP (presente nel campo `IfName`) e il VNL della rete VDE (passato come parametro), e si occupa di instanziare un thread che esegue il plug VDE. Il PID di tale thread viene finalmente salvato nel campo `Plugger` dell'endpoint.

```
func (this *EndpointStat) LinkPlugStop()
```

Come il metodo precedente, anche questo metodo di appoggia ad una funzione C `C.vdeplug_leave()`, che prende come parametro il PID del thread e si occupa di terminarlo e deallocare le sue risorse.

2.3.2 Metodi del Driver

Questi metodi sono quelli descritti nel lifecycle, e vengono eseguiti da Docker ogniqualvolta si interagisce con la rete gestita dal Driver.

```
func NewDriver(storepath string, clean bool) Driver
```

Non si tratta di un metodo della struttura dati Driver, ma utilizzata per crearne uno. il `datastore` viene in gioco durante la sua esecuzione, poiche' esso viene utilizzato per verificare quali endpoint e quali network sono presenti nel sistema, e aggiorna di conseguenza il Driver corrente. Il primo parametro e' il path del file `datastore`, mentre il secondo parametro informa la funzione se svuotare il `datastore` e restituire un Driver vuoto. Questa funzione viene chiamata solamente allo startup del servizio e il Driver restituito dalla funzione viene registrato dal `main()` come handler per le chiamate API relative ai network VDE.

```
func (this *Driver) GetCapabilities() (*network.CapabilitiesResponse, error)
```

Questo metodo del Driver si occupa solamente di restituire se il network ha scope locale o globale. VDE Plug Docker ritorna ogni network come scope locale.

```
func (this *Driver) CreateNetwork(r *network.CreateNetworkRequest) error
```

Metodo che si preoccupa si creare il network richiesto. Il parametro `r` ha la seguente struttura

```
type CreateNetworkRequest struct {
    NetworkID string
    Options   map[string]interface{}
    IPv4Data  []*IPAMData
    IPv6Data  []*IPAMData
}
```

dove `NetworkID` e' l'ID assegnato da Docker alla rete da creare, `Options` contiene le opzioni specifiche per il Driver VDE, ovvero

- **sock**: si tratta del VNL della rete VXVDE a cui si devono connettere i container.
- **if**: definisce il prefisso da dare al nome delle interfacce TAP degli endpoint relativi allo specifico network VDE. E' limitato a 4 caratteri.

e sono passati durante il comando `docker network create`. Infine `IPv4Data` contiene il subnet e il gateway IPv4 per la rete, mentre `IPv4Data` contiene l'analogo per IPv6. Notiamo che la mancanza di dati IPv4 lancia un errore, mentre la presenza dei dati IPv6 e' opzionale. Questi dati vengono strutturati seguendo `EndpointStat` e salvati nel `datastore`. Notiamo come il network nasca senza alcun endpoint e senza instanziare ancora nessuna rete VDE.

```
func (this *Driver) DeleteNetwork(r *network.DeleteNetworkRequest) error
```

Si tratta del metodo che si occupa di eliminare un network. Il parametro `r` e' una struttura che contiene il singolo campo `NetworkID`, ovvero l'ID del network da eliminare. Il metodo, dopo aver verificato l'effettiva esistenza del network e che non ci siano endpoint connessi ad esso, si limita a rimuovere il network dalla struttura dati del Driver (ovvero dal campo `Networks`) e salvare le modifiche nel file `datastore`.

```
func (this *Driver) CreateEndpoint(r *network.CreateEndpointRequest)
(*network.CreateEndpointResponse, error)
```

Il metodo del Driver che viene chiamato quando e' richiesta la creazione di un nuovo endpoint. Dopo aver verificato che il network esista e che l'ID fornito come argomento non sia gia' assegnato ad un endpoint, viene impegata la funzione `NewEndpointStat()` per generare una nuova istanza della struttura `EndpointStat`, fornendo come primo parametro il `CreateEndpointRequest` che contiene i dati relativi agli indirizzi IP e MAC dell'endpoint, e il prefisso che il nome del TAP deve avere come secondo parametro. Viene poi ritornata una risposta `response`, il quale deve avere il campo `Interface` vuoto qualora il campo `Interface` del parametro `r` sia non-nil. Infine le modifiche al Driver vengono salvate nel `datastore`.

```
func (this *Driver) DeleteEndpoint(r *network.DeleteEndpointRequest) error
```

Questo metodo viene chiamato quando viene richiesto al driver di eliminare un endpoint. Riceve un parametro `r` con la seguente struttura

```
type DeleteEndpointRequest struct {
    NetworkID  string
    EndpointID string
}
```

dove `NetworkID` e `EndpointID` vengono prima di tutto utilizzati per verificare che il network e l'endpoint esistano. Verificato cio', si procede ad invocare `LinkDel()` descritto nei metodi per gli Endpoint. Successivamente si rimuove l'endpoint dalla struttura dati del Driver e si aggiorna il `datastore`.

```
func (this *Driver) EndpointInfo(r *network.InfoRequest) (*network.InfoResponse, error)
```

Metodo che viene richiamato quando vengono richieste informazioni riguardo ad un particolare endpoint. Il parametro `r` ha la stessa struttura di `DeleteEndpointRequest`, e viene analogamente utilizzato per verificare l'esistenza dei network e dell'endpoint. Fatto cio', si costruisce un oggetto `InfoResponse` che racchiude l'id dell'endpoint e il nome dell'interfaccia TAP associata ad esso, per poi ritornarlo.

```
func (this *Driver) Join(r *network.JoinRequest) (*network.JoinResponse, error)
```

Questo metodo viene eseguito quando e' richiesto l'aggiunta di un endpoint alla rete VDE. Il parametro `r` ha la seguente struttura

```
type JoinRequest struct {
    NetworkID  string
    EndpointID string
    SandboxKey string
    Options    map[string]interface{}
}
```

Dopo gli usuali controlli di esistenza, viene invocato `LinkAdd()` descritto tra i metodi degli endpoint. Se la creazione del TAP ha successo, si provede ad invocare `LinkPlugTo()`, che collega il dispositivo appena creato alla rete VDE usando il VNL associato alla rete. Se il collegamento fallisce, si procede ad eliminare il TAP device appena creato e si restituisce un erorre, altrimenti si procede aggiungendo il `SandboxKey` alla struttura dati dell'endpoint. Infine si procede a creare e restituire una struttura `JoinResponse`, che richiede tuttavia i campi `Gateway` e `GatewayIPv6` senza il mask. Si conclude salvando gli aggiornamenti agli endpoint, e quindi alla struttura dati del Driver, nel file `datastore`. Usualmente questo metodo e' chiamato immediatamente dopo la creazione dell'endpoint.

```
func (this *Driver) Leave(r *network.LeaveRequest) error
```

Chiamato quando un endpoint lascia il network, questo metodo si occupa di terminare il plug VDE che collega l'endpoint alla rete VDE e anche di eliminare il dispositivo TAP dell'endpoint. In particolare, dopo gli usuali controlli di esistenza, vengono invocati `LinkPlugStop()` e `LinkDel()` descritti tra i metodi degli endpoint.

2.3.3 Metodi del Datastore

Nei diversi metodi del Driver, si e' parlato di come la sua struttura dati venga salvata e ripristinata dal `datastore`. Vediamo ora quali sono i metodi che permettono di fare cio'.

```
func SetPath(path string)
```

Funzione invocato unicamente da `NewDriver()`, e si occupa di aggiornare il path in cui si deve trovare il datastore. Cio' puo' accadere quando nell'eseguibile `vde_plug_docker` viene impostata l'opzione `--dir-path`. Per eseguire questa modifica bisogna aggiornare `vde_plug_docker.service` aggiungendo l'opzione ad `ExecStart`.

```
func Clean()
```

Come la funzione precedente, anche questa e' invocata nella funzione `NewDriver()` nel caso l'eseguibile `vde_plug_docker` fosse stato eseguito con il flag `--clean`. Questa funzione si occupa di scrivere `nil` nel file `datastore`.

```
func Load(elem interface{}) error
```

Funzione utilizzata in `NewDriver()` per ripristinare la struttura del Driver da un precedente esecuzione del plugin. Si occupa di leggere il file `datastore` indicato dal path e, poiche' ha una notazione JSON, fare un parsing per renderlo una struttura Go adeguata. Cio' che questa funzione ritorna viene impostato come la struttura del Driver.

```
func Store(elem interface{}) error
```

Si tratta della funzione richiamata nei vari metori del Driver per salvare la sua struttura nel file `datastore`. Esso effettua un encoding della struttura del Driver sotto forma di JSON attraverso la chiamata di `json.Marshal()` per poi scriverlo nel file `datastore` indicato dal path.

2.3.4 Metodi C

Il plugin richiede anche la definizione di alcune funzioni C per interagire con la libreria `libvdeplug`. Essi sono principalmente utilizzati nei metodi degli endpoint e sono definiti nel file `vdeplug.c`

```
static int open_tap(char *name)
```

Si occupa di creare un dispositivo TAP con il nome `IfName` indicato nella struttura dati dell'endpoint

```
void *plug2tap(void *arg)
```

Funzione eseguita come thread separato e che richiama `open_tap()` e si occupa di chiamare le funzioni della libreria `libvdeplug` in modo tale da creare il plug VDE, ovvero ricevere e inviare correttamente i frame di dati dalla rete VDE al TAP device dell'endpoint e viceversa.

```
uintptr_t vdeplug_join(char *tap_name, char *vde_url)
```

La funzione che viene effettivamente chiamata dai metodi Go. Si occupa di instanziare il thread a cui viene passato come `start_routine` proprio `plug2tap()`. Infine ritorna il PID del thread che ha creato

```
void vdeplug_leave(uintptr_t th_ptr)
```

Seconda e ultima funzione richiamata dai metodi dei endpoint. Si occupa di terminare il thread creato per far comunicate il TAP dell'endpoint e la rete VDE.

Conclusioni

Il progetto `vde_plug_docker` e' senza dubbio molto interessante e mette a disposizione un meccanismo per integrare i container Docker in una rete VDE. Puo' senz'altro fornire spunti per ulteriori progetti in modo estendere ulteriormente la compatibilita' di VDE in altre applicazioni.

Profondi ringraziamenti vanno all'autore del progetto, Alessio Volpe, per avermi aiutato a chiarire e definire i concetti riportati in questa documentazione.