

---

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA  
DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA  
E DELL'INFORMAZIONE "GUGLIELMO MARCONI"

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Uso di eBPF per connettività di rete in ambienti  
Cloud Native

Elaborato in  
Comunicazioni Digitali e Internet

Relatore  
Prof. Walter Cerroni

Presentato da  
Agostino Fiumana

I Sessione Luglio  
Anno Accademico 2023/2024



# Indice

<b>1. Introduzione</b>	<b>5</b>
<b>2. eBPF</b>	<b>7</b>
2.1. Workflow di eBPF . . . . .	8
2.1.1. Processo di caricamento . . . . .	8
2.1.2. Network Hooks . . . . .	9
2.2. Programmi eBPF . . . . .	11
2.2.1. Mappe . . . . .	11
2.2.2. Helpers Functions e Function Calls . . . . .	12
2.2.3. Tail Calls . . . . .	13
2.2.4. Struttura di un programma . . . . .	14
<b>3. Ambiente Cloud Native</b>	<b>19</b>
3.1. Introduzione agli ambienti Cloud Native . . . . .	19
3.2. Docker . . . . .	20
3.3. Kubernetes . . . . .	21
3.4. Kind . . . . .	23
3.5. Helm . . . . .	23
<b>4. Cluster Networking</b>	<b>25</b>
4.1. Gestione del traffico in Kubernetes . . . . .	25
4.1.1. Container Network Architectures . . . . .	26
4.1.2. Service mesh . . . . .	28
4.2. Esempi di Soluzioni di Networking per Kubernetes . . . . .	29
4.2.1. Legacy Routing . . . . .	29
4.2.2. Istio . . . . .	30
4.2.3. Cilium CNI . . . . .	32
<b>5. Benchmark</b>	<b>37</b>
5.1. Prometheus . . . . .	37
5.2. Grafana . . . . .	38
5.3. Risultati ottenuti . . . . .	39

*Indice*

<b>6. Conclusioni</b>	<b>43</b>
<b>7. Ringraziamenti</b>	<b>45</b>
<b>Bibliografia</b>	<b>48</b>
<b>A. Appendice</b>	<b>49</b>

# 1. Introduzione

La trasmissione dei dati in rete ha subito trasformazioni significative negli ultimi decenni, evolvendo da semplici connessioni punto a punto a complessi ecosistemi di reti distribuite e *cloud*. Con il progresso delle tecnologie di rete, la velocità di trasmissione è aumentata notevolmente, portando l'attenzione sulla difficoltà dei calcolatori nel sostenere flussi di dati sempre più corposi. La necessità attuale di ricevere le informazioni di rete a livello di *kernel* e doverle copiare in *user space* per poter effettuare elaborazioni, comporta un grave calo nelle prestazioni e aumenta le operazioni necessarie per gestire il traffico di rete.

Una nuova frontiera in esplorazione è la possibilità di sondare i pacchetti di rete a livello di kernel, eliminando completamente l'elaborazione in user space. In questo contesto, *extended Berkeley Packet Filter* (eBPF) si distingue come una tecnologia innovativa che consente l'elaborazione dei dati direttamente a livello di kernel con un livello di sicurezza e flessibilità simile a quello offerto dallo user space. Permette di eseguire *bytecode* in un ambiente protetto all'interno del kernel, consentendo una rapida elaborazione dei pacchetti di rete e altre operazioni di sistema.

I vantaggi offerti da questa tecnologia in rapida espansione sono maggiormente apprezzabili in contesti ad alta velocità di trasmissione dove i flussi di dati sono dell'ordine di decine di gigabits al secondo. A livello industriale, queste trasmissioni di dati sono delegate a *data center*, infrastrutture che ospitano grandi quantità di server interconnessi tramite reti ad altissima capacità. Nei servizi moderni, i server fisici sono stati sostituiti in larga misura da applicazioni virtualizzate, progettate per offrire scalabilità, flessibilità e resilienza superiori rispetto alle architetture tradizionali. Questi data center si servono quindi di applicazioni virtualizzate che devono essere in grado di elaborare flussi di dati ad elevate prestazioni.

In virtù di questa transizione, eBPF può essere integrato all'interno di ambienti *cloud native*, sfruttando l'operatività a livello di kernel per applicazioni virtualizzate a traffico elevato. Cilium è una tecnologia di Isovalent che rappresenta un'applicazione di eBPF nel campo delle reti, particolarmente adatta agli ambienti cloud. Nel 2021 Gartner ha definito Isovalent *cool vendor* in quanto, grazie ai suoi software "migliora drasticamente le prestazioni, la visibilità, la sicurezza e la scalabilità della rete Kubernetes iniettando funzionalità di sicurezza e di logging direttamente in

## 1. Introduzione

livelli molto bassi dello stack Kubernetes” [1]. La dilagante popolarità di Cilium come soluzione per la gestione di reti virtualizzate è attribuibile all’ingente aumento delle prestazioni garantito da eBPF. In questa tesi, verranno confrontate diverse soluzioni di gestione delle reti, tra cui Cilium, mettendo in evidenza i vantaggi offerti da eBPF nell’elaborazione a livello di kernel rispetto ad altre tecnologie. Si esploreranno i benefici specifici per gli ambienti cloud e microservizi, dimostrando come eBPF possa rappresentare una soluzione ottimale per migliorare le prestazioni e la sicurezza della rete in questi contesti.

Nel corpo dell’elaborato è contenuta un’introduzione teorica a eBPF, presentandone il flusso di lavoro e la struttura di un programma, con particolare enfasi sul ruolo che svolge nell’implementazione di funzionalità di rete. Successivamente sono esposti i fondamenti software di ambienti *cloud native* basati su *Kubernetes*, seguiti da un’illustrazione del funzionamento delle connessioni di rete in cloud e dalla presentazione di possibili soluzioni per amministrare l’interconnessione all’interno di ambienti virtualizzati. Infine sono mostrati risultati sperimentali ottenuti confrontando, tramite misurazioni su latenza, throughput e consumo di CPU, tre diverse soluzioni di networking in ambienti cloud, con lo scopo di sondare l’impatto di eBPF in servizi virtualizzati. In Appendice è riportata una guida pratica per la riproduzione delle misure effettuate.

## 2. eBPF

Aggiornare e manipolare le funzionalità del kernel è notoriamente una sfida ostica, l'*extended Berkley Packet Filter* (eBPF) è una moderna tecnologia che permette di scrivere codice e caricarlo dinamicamente all'interno del kernel, proponendo un nuovo approccio a questa sfida e aprendo gli orizzonti per una gamma di funzionalità in piena esplorazione. Per affrontare la trattazione di eBPF e presentare le caratteristiche distintive di questa tecnologia è stata seguita la struttura proposta nel testo *Learning eBPF* [2].

Ciò che in tempi recenti prende il nome di eBPF è in realtà la deriva di una tecnologia nata per la prima volta nel 1993, nota allora come *BSD Packet Filter*, che aveva come applicazione principale il *networking* e permetteva di utilizzare filtri per pacchetti di dati. Durante la sua evoluzione fu introdotta con il nome di *Berkeley Packet Filter* in Linux nel 1997, come supporto al pacchetto *tcpdump*, e successivamente nel 2012 i programmi BPF in Linux hanno ottenuto la capacità di accettare o rifiutare le chiamate di sistema provenienti dallo spazio utente, andando oltre le funzionalità di packet filter diventando così un vero e proprio strumento per interagire efficientemente con il kernel. Al giorno d'oggi il nome eBPF porta ancora in sé la denominazione di packet filter ma le sue applicazioni sono innumerevoli e diffuse grazie alla capacità dei programmi eBPF di essere iniettati nel kernel di Linux e di estrarre da esso informazioni di ogni tipo. I principali impieghi di eBPF sono:

- tracciabilità delle prestazioni, rispetto a diversi aspetti del sistema;
- networking ad alte prestazioni, garantendo efficienza e visibilità;
- applicazioni di sicurezza, grazie alla visibilità e tracciabilità sulle operazioni di sistema.

Grazie alla sua versatilità questa tecnologia si è resa pioniere di un nuovo modo di interpretare il kernel di Linux come riprogrammabile e accessibile.

## 2.1. Workflow di eBPF

Il modo in cui i programmi eBPF sono caricati nel kernel può essere descritto come *event-based*, infatti ogni programma eBPF è associato ad un evento, e agisce in maniera passiva, ossia in risposta all'evento, solo quando esso si verifica. Un pregio di eBPF che ne dimostra la potenza e flessibilità è la possibilità di poter caricare programmi senza richiedere un riavvio del sistema e quindi garantire operatività immediata. Caricare un programma all'interno del kernel prevede però un insieme di restrizioni che ne impattano il codice e le applicazioni.

### 2.1.1. Processo di caricamento

Un programma eBPF è generalmente realizzato in linguaggi di alto livello, principalmente C, utilizzando una porzione ridotta di librerie, escludendo quindi diverse funzioni note e diffuse come *printf()* e *scanf()*. Il compilatore utilizzato per i programmi eBPF è *Clang*, del progetto *LLVM* che permette di scrivere e compilare codice in file oggetto, più precisamente file ELF, costituito da codice binario relativo ai valori di spazio di indirizzamento dei registri contenenti le diverse istruzioni eBPF. Il compilatore Clang introduce diverse restrizioni pensate per la sicurezza del kernel, in particolare [3]:

- non è permesso definire variabili globali non statiche;
- ogni ciclo deve essere limitato e avere dimensione fissa;
- lo spazio dello *stack* è limitato a 512 bytes, per cui non è possibile compilare programmi di grande dimensione.

Questo sottoinsieme di istruzioni C introduce quindi diverse restrizioni, ma anche diverse funzioni *helper* che permettono di interagire con elementi nativi di eBPF.

Una volta compilato in formato ELF, il programma è ora soggetto ad un controllo da parte del *verifier* che agisce in due tempi:

- traduce le istruzioni del programma in un *Directed Acrylic Graph* (DAG), formato ottimale per calcolare il tempo di esecuzione nel caso peggiore possibile e per controllare che tutti i loop abbiano una fine;
- controlla ogni possibile percorso dalla prima istruzione, creando un modello di macchina a stati finiti così da poter riutilizzare gli stati già sondati in precedenza e ridurre il carico di lavoro, inoltre limita anche la lunghezza massima di ciascun percorso.



Il *verifier* svolge anche due controlli fondamentali: un controllo sulle licenze, in quanto alcune funzioni eBPF necessitano una corretta licenza per essere chiamate, e si assicura che ogni accesso a variabili sia seguito da un controllo di limite, così da non accedere a porzioni di memoria esterne al programma e garantire la sicurezza delle informazioni contenute all'interno del kernel.

Il codice prodotto dal compiler a questo punto è effettivamente codice binario ma non è ancora nel formato di codice macchina richiesto dal kernel, per questo eBPF si avvale di un compilatore *Just-in-Time*(JIT), che legge il file ELF e restituisce una serie di istruzioni macchina leggibili dalla CPU e quindi processabili dal kernel. L'intero flusso di lavoro di eBPF è sintetizzato in figura 2.1.

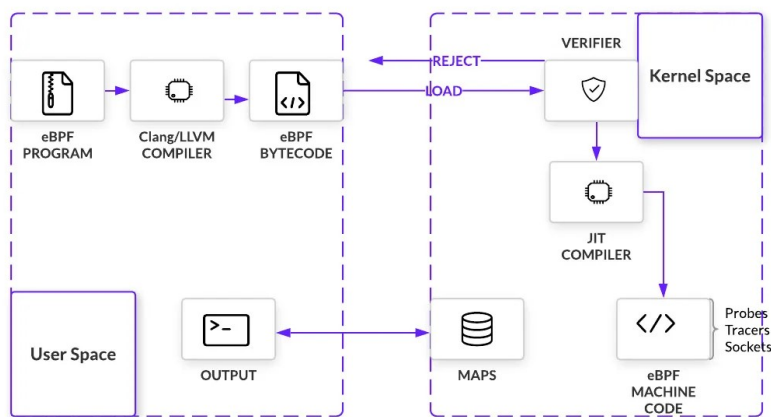


Figura 2.1.: *Workflow* di eBPF [4].

### 2.1.2. Network Hooks

I *network hooks* [3] sono elementi di grande importanza nell'ambito del networking e dell'elaborazione di pacchetti. Questi ganci sono ubicati all'interno della *network stack* del kernel, mostrata in Figura 2.2, e permettono la cattura di pacchetti di dati prima di elaborazioni ad alto livello. Un programma eBPF può fare uso di questi ganci per posizionarsi in un determinato punto di ricezione dei dati. I due hooks fondamentali per l'elaborazione e l'instradamento di pacchetti sono *eXpress Data Path* (XDP) e *Traffic Control* (TC), che permettono la manipolazione di dati prima che raggiungano lo user space e svolgono un ruolo determinante nell'efficienza di eBPF.

## 2. eBPF

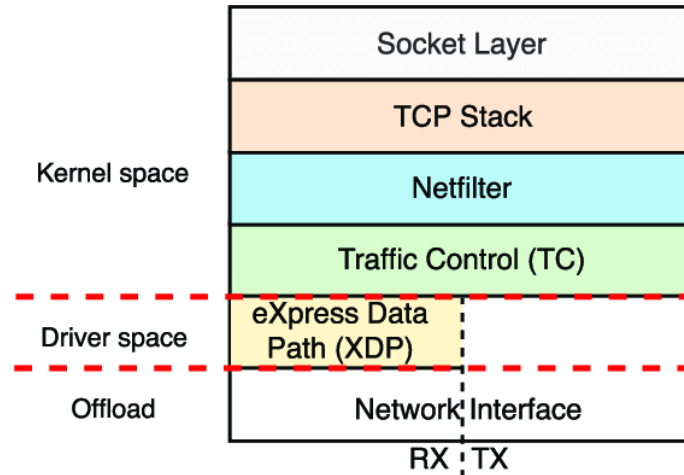


Figura 2.2.: Network stack del kernel [3].

### eXpress Data Path

XDP si trova nello strato più basso della pila ed è il primo hook in cui è possibile operare sui pacchetti. Non è possibile effettuare trasmissione di pacchetti da questo punto, ma solo riceverli, questa limitazione è data dalla profondità a cui si trova XDP, che è in grado di interagire con le informazioni ancora prima che venga allocata memoria da parte del sistema operativo per ospitarle. In questo gancio un programma può prendere decisioni veloci basate sui dati relativi al pacchetto in esame evitando *overhead* eccessivi introdotti dall'analisi a livello più alto. Questa caratteristica di XDP lo rende molto performante e ottimo per applicazioni di tipo *firewall*. Ogni programma XDP restituisce un verdetto riguardo ai dati in esame che ne determina la destinazione. Come mostrato in tabella 2.1, XDP offre cinque

Tabella 2.1.: Descrizione delle azioni XDP.

Value	Action	Description
0	XDP_ABORTED	Segnala errore e scarta il pacchetto.
1	XDP_DROP	Scarta il pacchetto.
2	XDP_PASS	Consente ulteriori elaborazioni da parte dello stack del kernel.
3	XDP_TX	Trasmette all'interfaccia da cui proviene.
4	XDP_REDIRECT	Trasmette il pacchetto verso un'altra interfaccia.

possibile verdetti, implementati con un comando *return* a fine funzione. Il verdetto *XDP\_REDIRECT* è il più complesso, può indirizzare pacchetti su: schede di rete diverse, su CPU diverse o sulla *socket AF\_XDP* per consentire l'elaborazione in user space. Il comando inoltre richiede di specificare l'interfaccia in cui inoltrare

i pacchetti, questa informazione può essere estratta grazie a due funzioni di supporto: `bpf_redirect()` o `bpf_redirect_map()`, la prima più specifica per dispositivi hardware, la seconda invece estrae informazioni da una mappa che contiene anche CPU o socket.

## Traffic Control

Un grande limite di XDP è l'impossibilità di interfacciarsi con porte di trasmissione. Traffic control è l'hook più vicino alla scheda di rete con possibilità di trasmettere pacchetti, nei sistemi Linux è utilizzato infatti per implementare le politiche di rete. Trovandosi più in alto nella pila, i pacchetti presenti su TC sono già stati processati dal kernel, che ha estratto metadati e li rende disponibili ai programmi eBPF eseguibili. Questo rende TC un hook più completo che può implementare politiche più complesse a discapito di prestazioni inferiori. I possibili verdetti in questo gancio sono numerosi, per questo di seguito sono riportati solo i più diffusi.

Tabella 2.2.: Descrizione dei verdetti TC.

Value	Action	Description
-1	TC_ACT_UNSPEC	Usa l'azione TC predefinita.
0	TC_ACT_OK	Porta il pacchetto sulla coda TC.
1	TC_ACT_RECLASSIFY	Inizia nuovamente a classificare i pacchetti .
2	TC_ACT_SHOT	Scarta il pacchetto.
3	TC_ACT_PIPE	Esegue l'azione successiva.

La sinergia di programmi eBPF situati in entrambi gli hooks presentanti sostiene le tecnologie di networking basate su eBPF, garantendo alta efficienza e visibilità.

## 2.2. Programmi eBPF

In quanto insieme di istruzioni, eBPF introduce costrutti e funzioni d'ausilio che, dichiarate all'interno del codice, permettono di interagire con gli elementi nativi di eBPF e offrono maggiore visibilità sull'operato dei programmi.

### 2.2.1. Mappe

La possibilità di creare mappe è una delle caratteristiche fondamentali di eBPF, che dimostra l'accessibilità che questa tecnologia garantisce nelle sue interazioni con il kernel. Le mappe sono delle strutture *key-value*, disponibili per ogni programma

## 2. eBPF

eBPF, definibili dall'utente e quindi accessibili e modificabili dallo user space. Le mappe sono definite come una struttura globale, e permettono l'interazione tra diversi programmi eBPF e tra *user space* e kernel. In figura 2.3 è mostrato il flusso di lavoro di una mappa. Esistono diversi tipi di mappe, listati nella documentazione

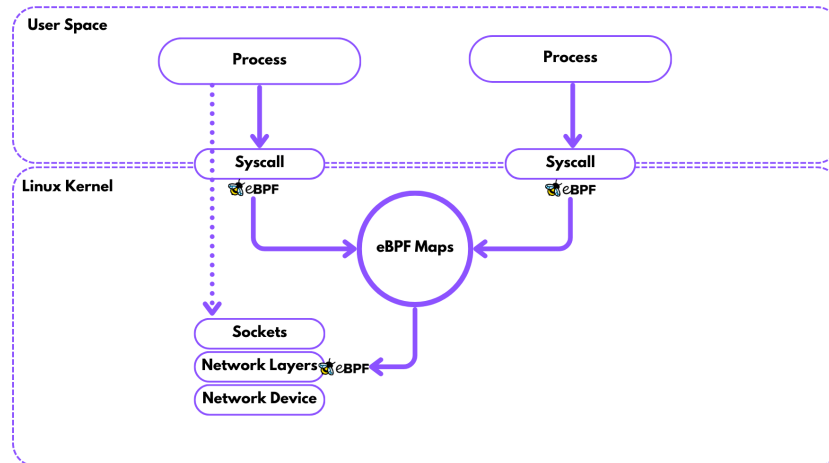


Figura 2.3.: Flusso di lavoro di mappe eBPF.

bpf. Le più utilizzate sono le *array* e *hash maps*, definibili anche con le specifiche *Least Recently Used*(LRU) e *PER\_CPU* che permettono rispettivamente di: avere una semantica LRU per l'eliminazione dei dati e associare una mappa ad un singolo core per ottimizzare l'elaborazione dei dati.

Le mappe possono essere definite con la sintassi mostrata nel listato 2.1, in questo caso è stata definita una mappa di tipo array, con chiavi di tipo intero a 32 byte, la mappa fa riferimento alla struttura *datarec* presente nello user space e ha un massimo di cinque valori in entrata, che in questo caso saranno *array*.

```
1 struct {
2   __uint(type, BPF_MAP_TYPE_ARRAY);
3   __type(key, __u32);
4   __type(value, struct datarec);
5   __uint(max_entries, 5);
6 } xdp_stats_map SEC(".maps");
```

Listing 2.1: Definizione di una mappa

### 2.2.2. Helpers Functions e Function Calls

eBPF fornisce una particolare gamma di funzioni dette *helper functions* atte a facilitare le interazioni all'interno dell'ambiente di lavoro del programma. Queste funzioni

sono specifiche per l'applicazione del programma e sono definite all'interno del codice sorgente del kernel. Per potere utilizzare una di queste funzioni sarà sufficiente includere la libreria in cui è definita senza la necessità della presenza, all'interno del codice, di una controparte in linguaggio C, riducendo quindi notevolmente il carico di lavoro del singolo programma e del compilatore. Alcune delle funzioni di supporto più comuni sono quelle utilizzate per ottenere messaggi di *debug* o per modificare la *endianness* di stringhe di byte.

Se invece è richiesto definire nuove funzioni all'interno del programma eBPF è necessario seguire un formato preciso. Ciascuna funzione nuova deve essere definita tramite il comando [2]:

```
1 static __always_inline void function(void *ctx, int val)
```

Questa definizione è necessaria per evitare che all'interno del programma siano eseguite operazioni *jump*, copiando quindi le istruzioni contenute nel programma nell'area di memoria in cui devono essere lette, questo evita problemi in fase di compilazione e di verifica. In versioni recenti questa restrizione è stata superata ma non tutte le interfacce di compilazione supportano questo aggiornamento per cui nel codice è sempre buona pratica seguire la definizione presentata precedentemente.

### 2.2.3. Tail Calls

Per la creazione di programmi più complessi eBPF include anche la tecnologia *tail calls*. Le chiamate in coda sostituiscono il contesto di esecuzione del programma chiamante con quello del programma chiamato, senza però ritornare al chiamante, cioè permettono la concatenazione di due diversi programmi. Questo approccio è adottato in diversi ambienti di programmazione ed è solitamente atto a non sovrappopolare la pila di istruzioni da eseguire ed evitare problemi di *overflow*. In eBPF questa ottimizzazione diventa fondamentale poiché la pila di un singolo programma è limitata, come menzionato in precedenza. Le chiamate in coda alleggeriscono il singolo programma favorendo l'utilizzo di più programmi legati dinamicamente. Per realizzare una chiamata in coda è necessario utilizzare la funzione di supporto [2]:

```
1 long bpf_tail_call(void *ctx, struct bpf_map *prog_array_map, u32 index
    )
```

Che prende in ingresso i seguenti tre argomenti:

1. *ctx* un puntatore che trasferisce il contesto di esecuzione al programma successivo;
2. *prog\_array\_map* una mappa di programmi definita appositamente per la chiamata, contenente tutti i programmi che saranno chiamati in coda;

## 2. eBPF

3. *index* l'indice del programma nella mappa da chiamare nell'istanza attuale.

Si nota quindi come sia indispensabile definire una mappa di tipo `prog_array` contenente i programmi da concatenare.

### 2.2.4. Struttura di un programma

Per comprendere a fondo la struttura di un programma eBPF nel listato 2.2 è riportato un esempio, scritto in linguaggio C, di cui saranno esaminate le diverse fasi di scrittura, compilazione e caricamento.

```
1 #include <linux/bpf.h>
2 #include <bpf/bpf_helpers.h>
3
4 int num = 0;
5
6 SEC("xdp")
7 int xdp_prog_simple(struct *ctx){
8     bpf_printk("Example %d", num);
9     num++;
10    return XDP_PASS;
11 }
12
13 char _licens[] SEC("license") = "GPL";
```

Listing 2.2: Programma `xdp_pass_kern` in linguaggio C

Nelle prime due righe di codice sono incluse le librerie necessarie per lavorare con gli strumenti fondamentali di eBPF e per chiamare funzioni helper. Successivamente è dichiarata una variabile globale incrementata ad ogni esecuzione del programma. La macro *SEC* indica che il codice entra in una sezione specifica dedicata all'interfaccia XDP. All'interno della funzione *xdp\_simple\_prog* è contenuto il corpo del programma eBPF, che in questo caso stampa la stringa *Example* seguita dalla variabile *num* e restituisce `XDP_PASS`. Questo valore restituito è un verdetto per l'interfaccia XDP che dichiara il libero passaggio del pacchetto in arrivo su di essa. La macro *SEC* inserita come ultima riga di codice è necessaria per la dichiarazione della licenza, che blocca l'utilizzo di funzioni eBPF se non correttamente indicata.

Per ottenere dal programma *xdp\_pass\_kern.c* un file oggetto è necessario adoperare l'apposito compilatore CLANG del progetto LLVM, il comando 2.3 realizza il file oggetto desiderato.

```
1 clang -target bpf -Wall -O2 -c xdp_pass_kern.c -o xdp_pass_kern.o
```

Listing 2.3: Compilazione del file oggetto

Per visualizzare il contenuto di un file oggetto è possibile utilizzare il comando 2.4 che fornisce una visualizzazione di facile interpretazione.

```
1 llvm-objdump -S xdp_pass_kern.o
```

Listing 2.4: Comando object dump di LLVM

Nel listato 2.5 è riportato il contenuto del file *xdp\_pass\_kern.o*.

```
1 xdp_pass_kern.o: file format elf64-bpf
2
3 Disassembly of section xdp:
4
5 0000000000000000 <xdp_prog_simple>:
6 ;   bpf_printk("Example %d", num);
7     0: 18 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r6 = 0 11
8     2: 61 63 00 00 00 00 00 00 r3 = *(u32 *) (r6 + 0)
9     3: 18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0 11
10    5: b7 02 00 00 0b 00 00 00 r2 = 11
11    6: 85 00 00 00 06 00 00 00 call 6
12 ;   num++;
13    7: 61 61 00 00 00 00 00 00 r1 = *(u32 *) (r6 + 0)
14    8: 07 01 00 00 01 00 00 00 r1 += 1
15    9: 63 16 00 00 00 00 00 00 *(u32 *) (r6 + 0) = r1
16 ;   return XDP_PASS;
17   10: b7 00 00 00 02 00 00 00 r0 = 2
18   11: 95 00 00 00 00 00 00 00 exit
```

Listing 2.5: File oggetto .ELF

La struttura generale del file include una prima riga che fornisce informazioni sul formato dell'oggetto, successivamente vengono mostrate le varie sezioni, in questo caso all'interno della sezione *xdp* è contenuta l'interezza del programma che è suddiviso in ogni sua operazione. Per le funzione *bpf\_printk* sono dedicate cinque righe di codice binario, per l'incremento della variabile *num* tre e per *return XDP\_PASS* due.

In ciascuna riga la prima informazione disponibile è l'offset dalla locazione in memoria della funzione *xdp\_prog\_simple*, come si può notare la prima istruzione in codice binario è lunga 16 bytes per questo l'incremento dell'offset nella seconda istruzione ammonta a 2. L'istruzione contiene inoltre nel primo byte il valore *0x18* relativo all'operazione da eseguire, in questo caso, consultando la documentazione eBPF di IOvisor [5], si tratta di un'istruzione di memoria con pseudocodice *dst=imm*, che porta il valore immediato alla destinazione specificata. La destinazione è riportata nel byte seguente, 06 che indica il registro r6, mentre il valore si trova nel quinto byte ed è 0. La visualizzazione mostra anche una traduzione più

## 2. eBPF

accessibile a destra del codice binario che conferma quanto scritto e specifica che l'istruzione è di lunghezza doppia con la dicitura *ll*.

Per caricare il programma all'interno del kernel è possibile utilizzare il seguente comando 2.6 contenuto nella libreria *bpftool*. Si noti che il comando necessita una specifica directory nella sintassi, questa directory contiene infatti tutti i programmi *pinned*, ossia fissati, all'interno del kernel. Nel caso del pacchetto *bpftool* non è possibile caricare un programma senza fissarlo, infatti se così non fosse il programma verrebbe eliminato subito dopo la prima esecuzione all'interno del kernel, non essendo più chiamato. Tramite l'esplicita dichiarazione di questa directory è quindi possibile caricare il programma in maniera che possa essere *pinned* e facilmente sondato nel futuro. Nel caso un programma non sia fissato, non essendo presente all'interno della directory, si possono ottenere le sue specifiche solo tramite *file descriptor*, ma il processo è più complesso.

```
1 bpftool load xdp_pass_kern.o /sys/fs/bpf/xdp_prog_simple
```

Listing 2.6: Comando load

Per ottenere la lista di programmi eBPF caricati nel kernel è sufficiente usare il seguente comando e cercare il programma appena caricato.

```
1 bpftool prog show
```

Isolando con il comando *grep* le informazioni di interesse dall'output del comando precedente si ottiene 2.7.

```
1 131: xdp name xdp_prog_simple tag 720ff25c2d8e3a86 gpl
2 loaded_at 2024-04-27T15:27:02+0200 uid 0
3 xlated 96B jited 67B memlock 4096B map_ids 12,13
4 btf_id 102
```

Listing 2.7: Informazioni su xdp\_prog\_simple

Sono mostrate diverse informazioni riguardanti il programma in esame:

- *ID*: è un numero assegnato alla singola istanza del programma che lo distingue da ogni altro programma eBPF presente nel kernel. Questo identificatore rimane unico anche in caso di due istanze uguali dello stesso programma;
- *type*: il secondo campo indica il tipo di programma eBPF, in questo caso si tratta di un programma xdp;
- *name*: questo campo indica il nome del programma, si noti come il loader ignori il nome con cui è stato salvato o compilato il programma, ma utilizzi il nome dichiarato nella sezione xdp;



- *tag*: questa stringa di caratteri rappresenta una somma *Secure Hashing Algorithm* (SHA) generata durante la compilazione, relativa alle istruzioni contenute nel programma. Questo implica che due programmi eBPF identici posseggono lo stesso Tag;
- *GPL*: la licenza utilizzata.;
- *loaded\_at*: indicazione temporale di quando è avvenuto il caricamento del programma;
- *UID*: rappresenta l'identificativo dell'utente che ha caricato il programma, in questo caso l'utente 0 è il root;
- *xlated*: indica la dimensione in bytes del programma in eBPF codice binario;
- *jited*: indica la dimensione in bytes del programma compilato dal JIT compiler, quindi in linguaggio macchina;
- *memlock*: mostra la dimensione di memoria massima allocata per il programma;
- *map\_ids*: sono indicati gli identificativi delle mappe di cui fa utilizzo il programma per scambiare informazioni nell'ambiente del kernel e con lo user space;
- *btf\_id*: in questo campo è riportato l'identificativo del *BPF Type Format* (BTF) relativo al programma. Il BTF mostra come vengono manipolate le strutture e i dati contenuti nel codice, per fornire informazioni sulla configurazione del programma.

Grazie all'identificativo è possibile visualizzare il contenuto del programma caricato in formato eBPF codice binario tramite il comando mostrato nel listato seguente.

```
1 bpftool prog dump xlated id 131
```

Nel listato 2.8 è mostrata una traduzione effettuata dal compilatore Clang. Si può notare come molte dichiarazioni siano in tutto e per tutto in formato C, ma le interazioni con variabili, fasi di scrittura e di lettura siano invece eseguite facendo riferimento a registri. Questo è l'aspetto che assume il programma dopo aver soddisfatto i requisiti del verifier.

```
1 int xdp_prog_simple(struct xdp_md * ctx):  
2 ; bpf_printk("Example %d", num);  
3 0: (18) r6 = map[id:12][0]+0
```

## 2. eBPF

```
4   2: (61) r3 = *(u32 *)(r6 +0)
5   3: (18) r1 = map[id:13][0]+0
6   5: (b7) r2 = 11
7   6: (85) call bpf_trace_printk#-108752
8 ; num++;
9   7: (61) r1 = *(u32 *)(r6 +0)
10  8: (07) r1 += 1
11  9: (63) *(u32 *)(r6 +0) = r1
12 ; return XDP_PASS;
13 10: (b7) r0 = 2
14 11: (95) exit
```

Listing 2.8: Traduzione CLANG del file .ELF

Si noti la coerenza con il file .ELF mostrato in precedenza e come il compilatore offra una traduzione ben leggibile, mostrando le operazioni relative ad ogni sezione del programma e i codici operazionali delle istruzioni eseguite.

## 3. Ambiente Cloud Native

Nel mondo moderno il mercato dei servizi cloud è in grande espansione, molte organizzazioni adottano un approccio virtualizzato per erogare i loro servizi, servendosi di orchestratori di sistemi virtualizzati come *Kubernetes* o *Amazon Elastic Container Service*. All'interno di questi sistemi virtualizzati sono presenti server su cui sono eseguiti programmi, e ognuno di essi fa riferimento ad un kernel. In particolare in un ambiente Kubernetes tutte le immagini virtuali condividono lo stesso kernel, quindi un programma eBPF, caricato in un kernel Linux, avrebbe effetto su ogni immagine, offrendo la possibilità di migliorare l'interazione tra le diverse immagini virtuali.

### 3.1. Introduzione agli ambienti Cloud Native

I diversi problemi presentati dai server fisici, come l'impossibilità di dividere il carico di lavoro equamente e di isolare le applicazioni in esecuzione su determinati server, hanno contribuito all'espansione di tecnologie di dispiegamento virtualizzato.

La grande innovazione della virtualizzazione consta nella possibilità di utilizzare diverse *virtual machines* (VM) per eseguire applicazioni su un'unica CPU fisica. Le VM permettono grande visibilità, una migliore distribuzione del carico di lavoro e una maggiore scalabilità, in quanto ognuna di esse è una macchina a sè stante in esecuzione sullo stesso hardware, che può isolare programmi e mostrare informazioni non accessibili con tecnologie precedenti.

L'ultima innovazione nel campo del dispiegamento virtualizzato è la tecnologia a *container*. Simili alle macchine virtuali ma più leggeri, ciascun container infatti lavora sullo stesso sistema operativo ed è totalmente disgiunto dall'infrastruttura sottostante. Questa caratteristica ne determina la grande portabilità e la possibilità di utilizzare gli stessi containers su diversi sistemi operativi senza incorrere in problemi di compatibilità.

La grande praticità e scalabilità dei containers li ha resi un importante passo avanti nello sviluppo di ambienti virtualizzati e ha segnato una svolta nell'approccio di molte organizzazioni alla distribuzione di servizi. Le architetture cloud native sono basate sul concetto di *microservices*, una particolare implementazione strutturale

### 3. Ambiente Cloud Native

che prevede la dissezione di una grande applicazione in parti più piccole e disgiunte, mantenute e aggiornate separatamente. Nel caso di applicazioni virtualizzate è possibile frammentare un'applicazione in diversi container comunicanti. Osservare un ambiente di questo tipo richiede vari software che concorrono a creare e orchestrare un'architettura virtuale detta *cluster*, ossia un insieme di container comunicanti.

## 3.2. Docker

Docker [6] è una piattaforma ad accesso libero che permette di lavorare con container e eseguire applicazioni su di essi. Per garantire queste funzionalità Docker si serve di tre macroambienti accessibili all'utente :

- *Docker Client*: il client è l'interfaccia grazie a cui l'utente può eseguire comandi Docker. Tutte le richieste dell'utente per ottenere immagini e container provengono dal client;
- *Docker Registry*: qualsiasi immagine Docker è conservata all'interno del registry, che funge da archivio per scaricare e aggiornare applicazioni eseguibili su container;
- *Docker Host*: il Docker host è responsabile dell'esecuzione dei container e delle risorse di sistema dedicate ad essi. Inoltre all'interno dell'host è contenuto il *Docker Daemon* che funge da intermediario tra il client e il registry, quando il client cerca di accedere al registry il daemon è in grado di fornire i dati richiesti.

Questa gerarchia è mostrata in figura 3.1. Una nozione di grande importanza per

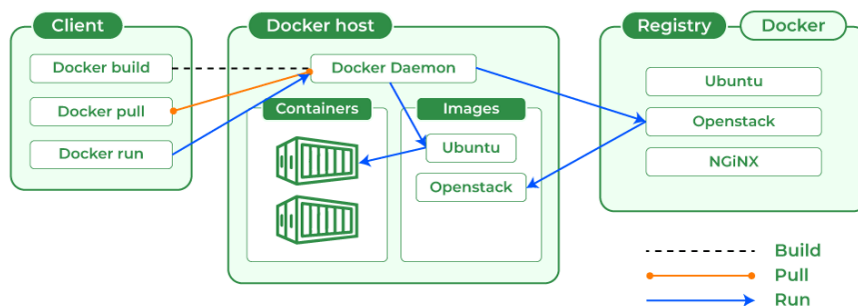


Figura 3.1.: Architettura di Docker [6].

comprendere il funzionamento di Docker è la differenza tra immagini e container.

Un'immagine è un file statico che contiene l'ambiente di esecuzione e i file necessari per eseguire un'applicazione in un container. Le immagini possono anche essere sviluppate aggiungendo istruzioni a immagini preesistenti e l'aggiornamento di un'immagine è un'operazione poco onerosa.

Un container invece è l'istanza manovrabile di un'immagine. Un sistema operativo può ospitare più containers contemporaneamente garantendo grande maneggevolezza. Un singolo container può connettersi alla rete, scambiare informazioni e creare un'immagine derivante dal suo stato attuale.

Grazie a Docker è quindi possibile ottenere un'immagine dal *Docker Hub* e creare un container derivante dall'immagine richiesta, questo permette di caratterizzare il ruolo di ogni container all'interno di un cluster, passaggio fondamentale nella configurazione dell'ambiente di lavoro.

## 3.3. Kubernetes

Kubernetes [7] è un sistema open-source per la gestione di ambienti di lavoro noti come cluster. Un cluster è insieme di macchine lavoranti, dette nodi, che possiedono un ambiente Linux indipendente e contengono applicazioni containerizzate. All'interno di ogni nodo del cluster sono ospitati dei *pod*: raccolte di container che condividono contesto di rete, memoria e CPU. Il cluster è gestito da un pannello di controllo che si occupa di manovrare nodi e pod.

Kubernetes implementa *load balancing* automaticamente all'interno del cluster: infatti quando si presenta una nuova attività, gestisce grazie al *control plane* la richiesta e la reindirizza al pod più adatto per eseguirla. Inoltre semplifica notevolmente l'intera configurazione dell'ambiente in fase di sviluppo in quanto esegue ogni richiesta tradizionalmente gestita da Docker, sia per il *pull* delle immagini destinate ai pod, sia per possibili aggiornamenti del cluster o controlli dello stato di salute, alleggerendo il carico di operazioni da svolgere durante la definizione dell'ambiente. Per applicazioni industriali Kubernetes permette di integrare le specifiche relative ai servizi offerti da *provider* di ambienti cloud in maniera automatica. L'architettura di un cluster è mostrata in figura 3.2.

Il control plane è il gestore dell'intero ambiente e generalmente è situato all'interno di una macchina che non contiene applicazioni virtualizzate. All'interno del pannello di controllo sono contenute diverse applicazioni che cooperano per la gestione del cluster:

- API: è il vero e proprio frontend del pannello e gestisce le comunicazioni con i diversi nodi;

### 3. Ambiente Cloud Native

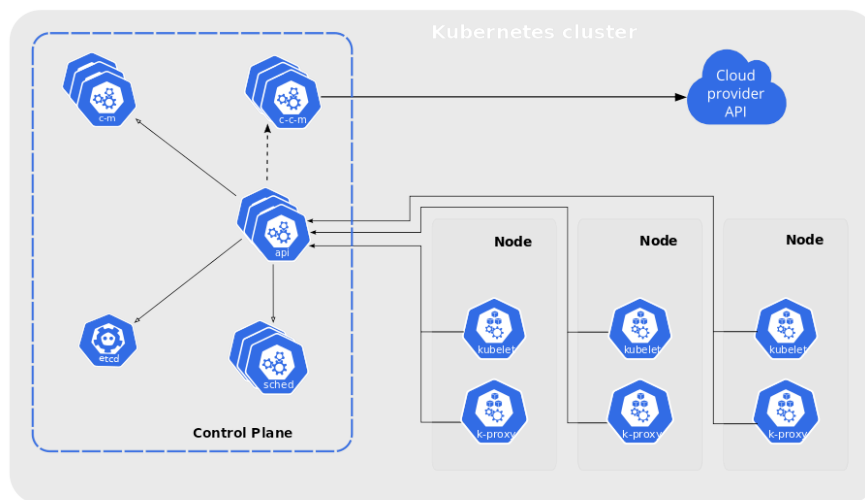


Figura 3.2.: Architettura di un cluster [7].

- *etcd*: indica la memoria contenente i diversi dati presenti, questa memoria è di tipo chiave valore e viene aggiornata in tempo reale per mantenere lo stato desiderato;
- *scheduler*: si occupa di allocare e offrire risorse ai vari pod, sia su richiesta del *developer*, sia per necessità interne all'ambiente;
- *controller manager*: è un grande insieme di controller che svolgono varie funzioni, principalmente si occupa di controlli sullo stato del cluster, come richieste ai nodi e ricezione di *flag* o l'aggiornamento del contesto di esecuzione dei pod;
- *cloud controller manager*: è l'elemento che si occupa di interfacciare il cluster a possibili provider cloud del servizio in utilizzo, questo permette di adeguare l'ambiente alle specifiche del provider.

L'elemento fondante dell'architettura che è regolato dal pannello di controllo è il nodo. Ogni nodo è una singola macchina, fisica o virtuale che ospita diversi pod ed è responsabile per l'allocazione delle risorse necessarie alla loro esecuzione, come memoria e CPU. Inoltre è dotato di un'interfaccia di rete che permette la comunicazione con altri nodi interni al cluster. Ciascun nodo inoltre ospita diversi servizi di supporto di Kubernetes:

- *kubelet*: che amministra i container assicurandosi che siano eseguiti secondo le specifiche richieste;
- *container runtime*: che supervisiona l'esecuzione dei container e gestisce la comunicazione con il sistema operativo sottostante;

- *kube-proxy*: un *proxy* di rete che permette la comunicazione all'interno e all'esterno dell'ambiente, implementando anche politiche di rete.

### 3.4. Kind

Per poter concludere la configurazione dell'ambiente è essenziale poter avviare un cluster. *Kubernetes IN Docker* (Kind) [8] è un tool da riga di comando ideato per garantire maneggevolezza nella creazione di un cluster. Kind è un strumento a libero accesso che permette di avviare cluster Kubernetes sfruttando la tecnologia container di Docker, ottenendo un approccio agevole, ideale per creare ambienti di sperimentazione. Kind si serve della tecnologia Kubernetes, condensando ogni elemento richiesto all'interno di un container Docker, in questo modo l'intero ambiente viene installato su un container grazie ad un'immagine che rispecchia la configurazione desiderata. I singoli elementi sono comunque gestiti da Kubernetes, che però invece di appoggiarsi su macchine reali, o singoli container virtualizzati, inzializza ogni elemento virtuale in un unico grande container di supporto.

Kind esprime solo una modalità possibile per la configurazione di un contesto virtualizzato ed è popolare per via della sua semplicità. Infatti grazie alle funzionalità Docker, sarà Kind a eseguire il pull dell'immagine adatta alle esigenze strutturali del cluster, evitando così un'installazione più complessa e dispersiva. Inoltre la definizione delle caratteristiche dell'architettura è eseguita tramite un file *.yaml* che adotta una sintassi accessibile e garantisce grande versatilità nella personalizzazione dell'ambiente. Un altro aspetto positivo di Kind è la rapidità con cui permette di dispiegare il cluster, questo aspetto agevola fasi di sperimentazione, dove spesso è necessario riconfigurare il sistema virtualizzato.

### 3.5. Helm

La gestione di un ambiente cloud native è spesso ostica poiché sono attuabili disparate configurazioni, è quindi possibile essere oberati dalla mole di informazioni da includere per una corretta esecuzione. Helm [9] è uno strumento per amministrare applicazioni Kubernetes e automatizzare il processo di verifica delle dipendenze dei singoli applicativi all'interno di un cluster.

Per operare, Helm si serve di uno speciale formato di pacchetti Kubernetes, detto *chart*. Un chart Helm è un pacchetto preconfigurato contenente tutte le risorse Kubernetes necessarie per eseguire un'applicazione, come servizi, pod e configurazioni. Ogni chart necessita di un file *.yaml* in cui è definito e di una sezione che ne specifichi

### 3. Ambiente Cloud Native

le dipendenze, in questo modo Helm, quando un chart viene utilizzato all'interno di un cluster Kubernetes, provvede a rendere disponibili le sue dipendenze.

Helm è strutturato in due macroambienti, un *client* e una *library*. All'interno del client è possibile utilizzare Helm per configurare chart o fruire di chart già esistenti, in particolare se si vuole richiedere una versione specifica di un chart è possibile consultare la library di Helm, in cui sono dislocati in diversi repository. Helm risulta particolarmente utile quindi per includere applicativi specifici all'interno dell'ambiente cloud con dichiarazioni più complete e che permettono di inserire esplicitamente impostazioni di configurazione da riga di comando. Anche se non chiamato esplicitamente spesso è richiesta la presenza di Helm per poter implementare applicazioni all'interno di un cluster in quanto diversi comandi *install* di pacchetti e applicativi si appoggiano su di esso implicitamente per assicurarsi che l'installazione sia eseguita il più precisamente possibile.



## 4. Cluster Networking

Durante il *deployment* di un cluster, Kubernetes mette a disposizione le risorse necessarie per comunicare tra le macchine virtualizzate, in particolare gestisce la divisione del traffico e l'assegnamento di *IP addresses*. Lo scheletro della struttura di Kubernetes prevede una compartimentazione in *namespaces* [10], che permettono di isolare le risorse di rete e creare ambienti con un'unica interfaccia di rete e tabelle di indirizzamento ad hoc.

### 4.1. Gestione del traffico in Kubernetes

Per gestire il traffico di rete all'interno di un namespace, Kubernetes utilizza i *services*. Un service è un'astrazione che permette di indirizzare in maniera stabile un gruppo di pods. Un servizio ha un IP assegnato e si occupa di loadbalancing e di *routing* nel cluster, in particolare possiede una tabella di indirizzamento e un proxy che consulta per rispondere alle richieste che provengono dai pod. Utilizzare un servizio per la comunicazione è più stabile in quanto all'interno di una data implementazione i pods hanno spesso brevi cicli di vita e i loro IP possono cambiare dinamicamente, contrariamente un servizio conserva una lista aggiornata in tempo reale di pods collegati ad esso, mantenendo un IP statico.

In un dato namespace la comunicazione è stratificata e può avvenire a più livelli:

- *container-to-container*: dal punto di vista della rete ciascun container all'interno di un pod si comporta come se si trovasse sul medesimo host. Può raggiungere l'host locale in ogni momento per ottenere i dati relativi alle porte di comunicazione, garantendo alte prestazioni e sicurezza, poiché questi dati non sono accessibili all'esterno di un pod;
- *pod-to-pod*: ad ogni pod è associato un indirizzo IP reale, è possibile realizzare la comunicazione inter pods sia ricorrendo a un servizio *Domain Name System* (DNS) interno al cluster, sia servendosi di services per l'indirizzamento del traffico. Utilizzando un servizio destinato al routing è possibile evitare di ricorrere all'allocazione di risorse per la gestione delle richieste dei pod, alleggerendo considerevolmente il carico di rete nel cluster.

## 4. Cluster Networking

- *pod-to-service*: ogni servizio uniforma i pods sotto una politica di accesso comune creando un IP virtuale a cui i client possono accedere e che viene comunicato in maniera trasparente ai pods. Ogni nodo esegue un processo tramite kube-proxy che programma le regole *iptables* per eseguire accessi agli IP del servizio e reindirizzarli ai nodi corretti. In questo modo si ottiene una soluzione di bilanciamento del carico altamente disponibile e con un basso overhead di prestazioni.
- *external-to-internal*: non è presente un modo per amministrare la comunicazione dall'esterno all'interno del cluster garantito da Kubernetes. Questa comunicazione è generalmente pilotata dai provider cloud, che agganciano al confine del namespace un blocco di loadbalancing comunicante con ogni elemento all'interno che risolve le richieste provenienti dall'esterno.

Per poter configurare l'intera architettura necessaria per gestire il traffico sono state proposte due tecnologie: *Container Network Architectures* (CNA) che lavorano allo strato di rete e trasporto definiti dal modello *Open Systems Interconnection* (OSI), e *service mesh* che lavora allo strato di applicazione.

### 4.1.1. Container Network Architectures

Una *Container Network Architectures* [11] è un plugin che si occupa di amministrare l'assegnamento di indirizzi IP e gestire le politiche di indirizzamento all'interno del cluster. Una CNA può essere immaginata come una *Application Program Interface* (API) intermediaria tra il container e la rete. Negli anni sono stati proposti due diversi modelli di CNA: *Container Networking Model* (CNM) e *Container Networking Interface* (CNI).

#### Container Networking Model

Il *Container Networking Model* è la soluzione predefinita di Docker. In particolare docker utilizza un'istanza di CNM detta *Libnetwork* che interfaccia il Docker Daemon con i plugin di rete. Un CNM si serve di tre elementi per realizzare la struttura di rete:

- *sandbox*: il particolare namespace che isola uno o più container rispetto all'host e ad altri container;
- *endpoints*: i punti di collegamento tra la rete e la sandbox;

- *network*: la rete adottata nel caso di Libnetwork è una LAN virtuale, all'interno della quale sono incluse anche tabelle e politiche di indirizzamento e specifiche DNS.

In figura 4.1 è mostrata la struttura adottata da Libnetwork.

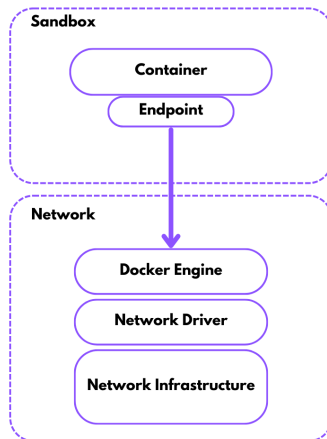


Figura 4.1.: Architettura di *Libnetwork*.

## Container Networking Interface

Il progetto Container Networking Interface proposto da *Cloud Native Computer Foundation* è una specifica di rete che permette di descrivere una configurazione tramite un file *JavaScript Object Notation* (JSON). Durante l'implementazione di un CNI il file JSON viene consultato per realizzare un namespace dedicato alla gestione della rete da parte della *CNI specification*. Successivamente è il *CNI network plugin* a configurare l'intera rete, secondo le informazioni contenute nella notazione JSON, assegnando indirizzi IP e istituendo politiche di rete. All'interno di ogni pod, kubelet esegue la chiamata per consultare la CNI per ottenere specifiche sulla morfologia della rete, i cambiamenti sono effettuati tramite il plugin CNI che può intervenire direttamente sul cluster. I tre elementi fondanti di ogni CNI quindi sono:

- *CNI specification*: l'istanza vera e propria della CNI, utilizzata come API tra kubelet e la sezione di network plugin;
- *CNI network plugin*: il componente della CNI che si occupa di implementare le specifiche di rete richieste;
- *libraries*: contengono le definizioni in linguaggio Go delle specifiche di rete, per la lettura da parte di kubelet.

In figura 4.2 è mostrata la struttura di lavoro di una CNI.

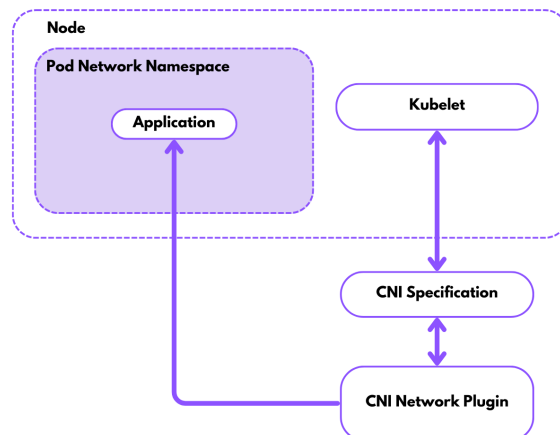


Figura 4.2.: Architettura di una CNI.

### 4.1.2. Service mesh

Una service mesh [12] è un plugin dedicato alla comunicazione tra servizi, realizzato tramite una serie di proxy di rete, dispiegati servendosi della tecnologia *sidecar*. Ciascun nodo nel cluster sarà quindi affiancato da un proxy di rete, in grado di supervisionare il traffico e offrire visibilità sui pacchetti in entrata ed uscita dal nodo. In particolare questo container permette di gestire le informazioni relative alle porte di comunicazione verso esterno e interno, realizzando politiche di routing e il deployment può avvenire secondo varie modalità che ne definiscono le caratteristiche di lavoro, concentrando le risorse su politiche di ingresso, uscita o visibilità dei pacchetti. Risulta essenziale quindi che ogni proxy interno ai nodi si interfacci con il sidecar per comprendere le modalità tramite cui avviene la comunicazione. Le definizioni relative al sidecar sono globali rispetto al namespace e dichiarabili in un file `.yaml` consultato al momento dell'istanziamento.

A livello logico una service mesh è divisa in *control plane* e *data plane*. Il control plane dispiega i sidecar richiesti e li amministra aggiornando le politiche di routing relative ad essi, inoltre è in grado di eseguire operazioni telemetriche per collezionare dati inerenti alla rete. Nel data plane sono situati i servizi comunicanti, qui i vari sidecar-proxy si occupano di: effettuare controlli di salute del sistema, bilanciamento del carico, monitoraggio del traffico e implementazione delle politiche di indirizzamento.

Questa intera struttura è mostrata in figura 4.3.

## 4.2. Esempi di Soluzioni di Networking per Kubernetes

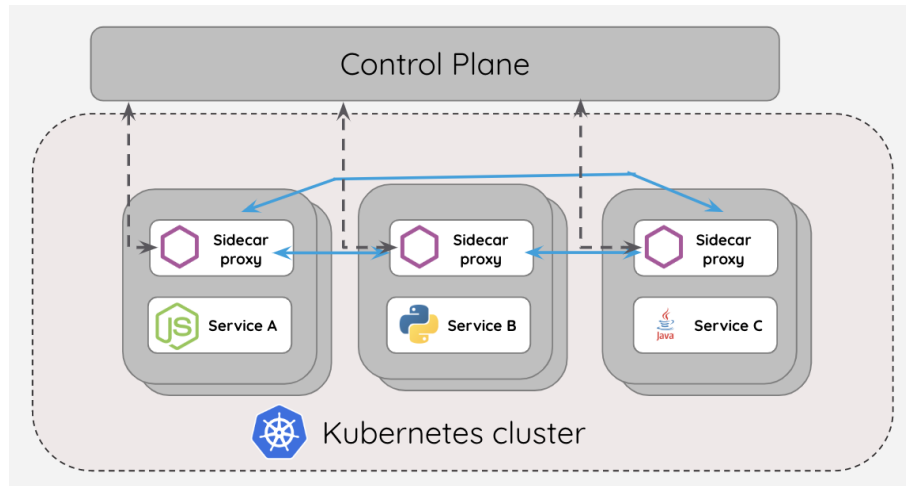


Figura 4.3.: Ambiente di lavoro di una service mesh [13].

## 4.2. Esempi di Soluzioni di Networking per Kubernetes

Di seguito sono presentate le tre soluzioni prese in esame per la gestione del traffico di rete all'interno di un cluster. Si noti che, service mesh e CNI non sono tecnologie mutualmente esclusive, poiché, come menzionato in precedenza, lavorano a strati OSI differenti, ma i loro impieghi possono facilmente sovrapporsi, e il loro utilizzo contemporaneo risulterebbe in un overhead aggiuntivo. Spesso le due tecnologie quindi sono implementate in maniera isolata e non concorrente, rendendo valido un confronto tra le due.

### 4.2.1. Legacy Routing

Per implementare delle regole di indirizzamento in un cluster privo di CNI o service mesh, Kubernetes si serve del *legacy routing* [7], basato su iptables Linux. All'interno di ogni nodo nel cluster è contenuto un componente kube-proxy, responsabile del networking che implementa un meccanismo a indirizzi virtuali destinato alla configurazione dei servizi.

Ogni istanza del proxy consulta il pannello di controllo per conoscere in tempo reale il numero di servizi attivi e gli *endpoints* associati, inoltre comunica con le API adatte per catturare un pacchetto sulla porta corretta e reindirizzarlo all'endpoint desiderato. Durante ogni ciclo di lavoro un loop di controllo verifica che kube-proxy faccia rispettare le politiche di indirizzamento stanziate. In figura 4.4 è mostrato il comportamento interno al nodo.

Per realizzare questo comportamento sono essenziali le API delle iptables fornite dal sistema operativo Linux. Per ogni nuovo service e endpoint creato il blocco kube-proxy installa iptables e le configura secondo le specifiche richieste dal cluster.

## 4. Cluster Networking

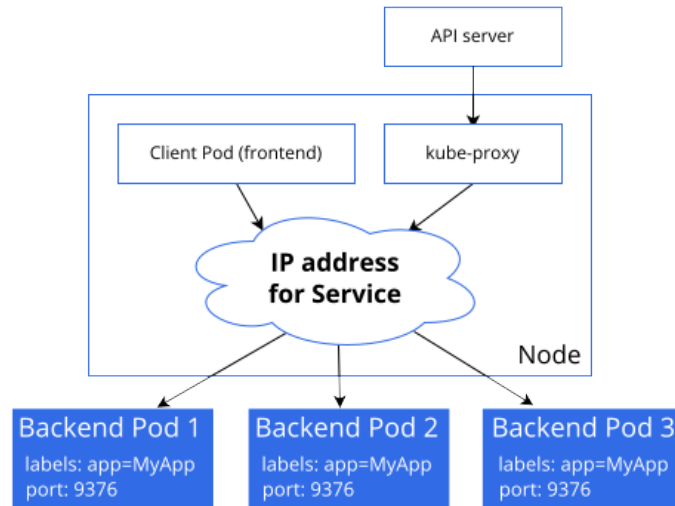


Figura 4.4.: Flusso di lavoro di kube-proxy [7].

Quando un client si interfaccia ad un servizio entra in azione la regola iptable associata che reindirizza i pacchetti sull'endpoint scelto. Le iptables generate emettono verdetti in base all'*header* di un pacchetto, perciò il loro utilizzo richiede un'analisi del contenuto del pacchetto in user space.

### 4.2.2. Istio

Istio [14] è una service mesh ad accesso libero che permette di operare con maggiore libertà nell'interconnessione di servizi virtualizzati. Istio espande le possibilità all'interno di un ambiente virtualizzato grazie ad una varietà di aggiunte:

- comunicazione tra servizi con crittografia *Transport Level Security* (TLS);
- load balancing automatizzato durante comunicazioni con protocolli HTTP e TCP;
- controllo del traffico a grana fine, con la possibilità di implementare politiche di indirizzamento, e configurazione di API per limitazioni del traffico.
- visibilità e esportazione di metriche grazie a dashboard online comunicanti con Istio.

In qualità di service mesh, Istio si serve di un control plane programmabile che aggiorna il cluster in tempo reale per garantire lo stato desiderato. All'interno del data plane invece è possibile utilizzare due modalità di mesh: sidecar, che istanzia un *Envoy* proxy per ciascun pod nel cluster e ambient che utilizza due proxy per nodo agli strati di trasporto e applicazione. L'architettura di Istio sfrutta la divisione

## 4.2. Esempi di Soluzioni di Networking per Kubernetes

classica in control plane e data plane, ma utilizza un particolare tipo di proxy detto *Envoy* e un articolato control plane detto *Istiod*.

Envoy è un proxy attivo nello strato di applicazione, è l'unico componente di Istio che agisce sul traffico, sia in entrata che in uscita, all'interno del data plane. Envoy è un proxy ad alte prestazioni studiato per garantire diversi benefici tra cui: controlli di salute, accessibilità alle metriche, bilanciamento del carico e sistemi di *fault injection* per verificare lo stato del sistema in situazioni di alto carico. Poiché Istio agisce esclusivamente tramite proxy Envoy non è necessario modificare il cluster a runtime ma sarà sufficiente adattare il deployment dei sidecar alle necessità di lavoro correnti.

Istiod è il pannello di controllo della mesh e permettere di convertire politiche di routing scritte ad alto livello in informazioni per riprogrammare i proxy presenti nel cluster. Inoltre si occupa di operazioni di *service discovery* per mantenere aggiornata la comunicazione con ogni servizio dinamicamente, in modo che servizi aggiunti a runtime siano identificati dai proxy, e che le regole di indirizzamento siano aggiornate di conseguenza. Tramite Istiod sono anche implementati i certificati di autenticazione previsti dal protocollo TLS, per la crittografia della comunicazione nello strato applicativo.

L'intera architettura di Istio è mostrata in figura 4.5

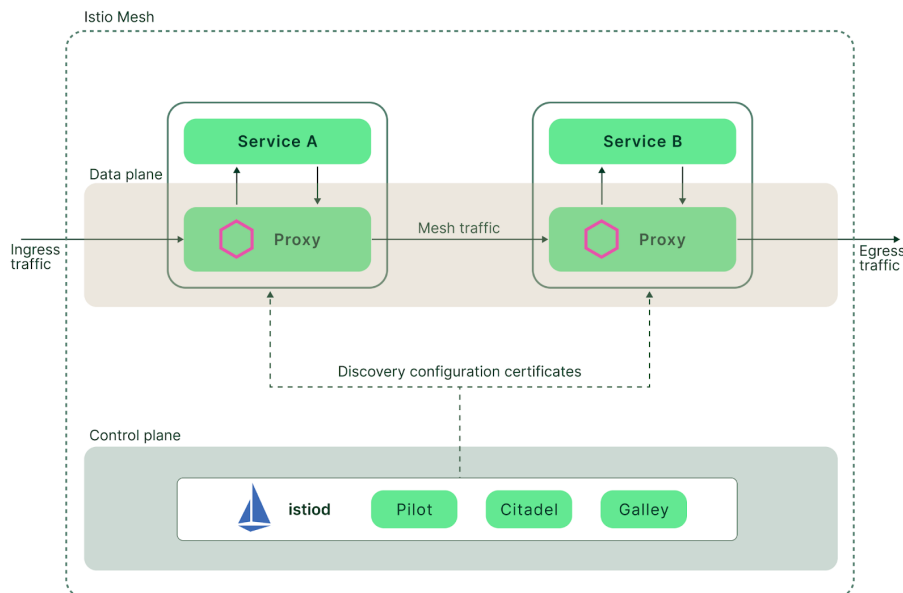


Figura 4.5.: Architettura di Istio [15].

### 4.2.3. Cilium CNI

Cilium [16] è una CNI, che pone le sue basi su eBPF per garantire la comunicazione tra pod all'interno di un cluster. Questa tecnologia innovativa unisce la visibilità e le prestazioni di eBPF alla comunicazione in ambienti virtualizzati, garantendo alta efficienza in termini di traffico e latenza. A livello pratico Cilium una volta installato in un cluster Kubernetes prende le veci di supervisore del traffico sostituendosi al componente kube-proxy. La grande differenza tra la modalità operativa dei due container è la soluzione adottata per il reindirizzamento dei pacchetti. Nel caso predefinito da Kubernetes, il blocco kube-proxy genera delle tabelle di indirizzamento, che vengono consultate durante ogni inoltro di pacchetti all'interno della rete, sia pacchetti provenienti dall'esterno che inter nodo. Cilium invece carica all'interno del kernel della macchina host una serie di programmi eBPF che concorrono all'implementazione di varie funzioni, ideate per sostituire totalmente le tabelle di indirizzamento. Facendo leva su eBPF, Cilium garantisce anche grande scalabilità per i servizi, permettendo l'inserimento di politiche di inoltro in maniera dinamica, e adattando le funzionalità eBPF all'architettura correntemente in utilizzo.

Nella gerarchia del cluster Cilium si inserisce tra l'orchestratore e l'ambiente virtualizzato, rendendo possibile la comunicazione con ambo i membri dell'ambiente. L'architettura di Cilium è mostrata in figura 4.6. L'intera struttura è basata sulla cooperazione di tre operatori: Cilium, Hubble e il Data Store.

### Cilium

I seguenti elementi sono riferiti all'implementazione di Cilium pura:

- *agent*: è presente in ogni nodo e sostituisce la modalità di lavoro di kube-proxy, comunica con le APIs soprastanti per ottenere informazioni sullo stato desiderato dell'ambiente, inoltre effettua operazioni di service discovery per conoscere lo stato del cluster in tempo reale e gestisce e implementa i programmi eBPF caricati;
- *client*: il client è l'interfaccia utilizzabile da riga di comando per sfruttare le diverse funzionalità offerte da Cilium, permette quindi di interagire con i nodi gestiti da cilium nel cluster;
- *operator*: si occupa di gestire compiti a livello dell'intero cluster, invece di propagarli nodo per nodo, per esempio rende più efficiente il processo di *IP address management* (IPAM) e effettua controlli di salute sui nodi, la sua presenza non è essenziale al funzionamento del cluster ma la sua assenza può portare a gravi malfunzionamenti;



## 4.2. Esempi di Soluzioni di Networking per Kubernetes

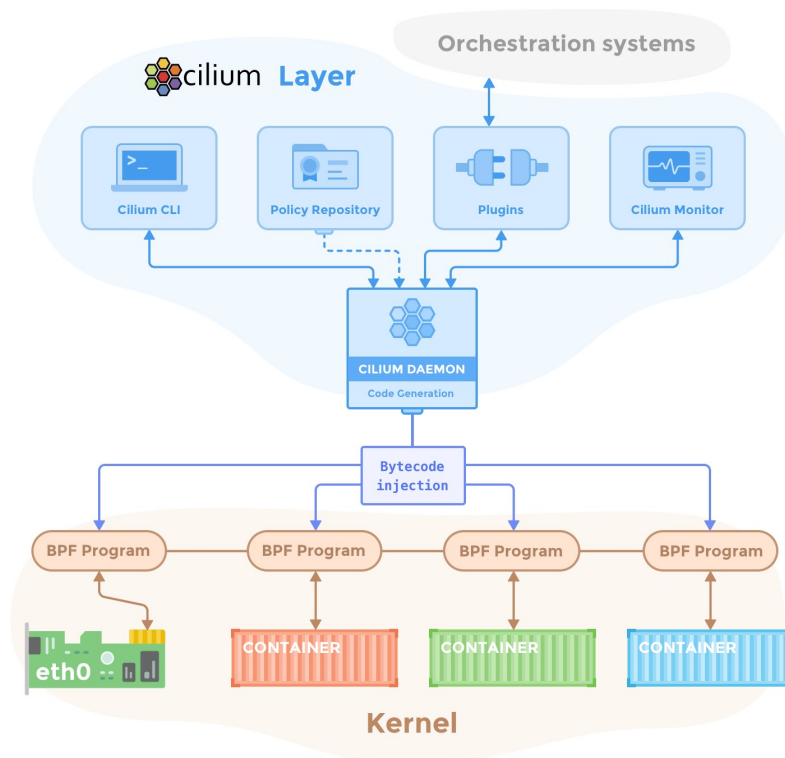


Figura 4.6.: Architettura di Cilium [16]

- *CNI Plugin*: il plugin si occupa di configurare gli elementi Cilium nel momento in cui sono istanziati o terminati, è il componente kublet all'interno di ogni nodo a invocarlo quando si verifica uno di questi eventi, e contiene tutte le informazioni necessarie a fornire un ambiente in grado di dialogare a livello di rete.

### Hubble

Hubble è una piattaforma dedicata all'osservabilità, integrata con Cilium, che sfrutta la tecnologia eBPF per accedere alle metriche relative alla comunicazione in ambiente in tempo reale. Per servirsi di Hubble è necessario introdurre i seguenti elementi all'interno dell'ambiente di lavoro:

- *server*: integrato all'interno di ogni agente Cilium, se è in utilizzo il server recupera i dati relativi al traffico e alle metriche in tempo reale ;
- *relay*: conosce ogni istanza server presente nel cluster e le connette alle rispettive APIs grazie al protocollo gRPC;
- *client*: anche hubble possiede un client utilizzabile da riga di comando per interfacciarsi con il blocco relay e ottenere informazioni sul flusso di pacchetti;

#### 4. Cluster Networking

- *graphical user interface*: un elemento ad alto livello che permette di utilizzare una interfaccia browser per visualizzare il traffico all'interno di un dato namespace;

#### Data Store

Per poter comunicare lo stato a tutti gli agenti Cilium è necessario utilizzare un archivio dati, accessibile in tempo reale, situato all'interno del cluster. È possibile implementare un *data store* in due diverse modalità: *Kubernetes CRDs* e *key-value*. Il caso *custom resource definitions* (CRDs) rappresenta l'implementazione predefinita per Cilium, questa memoria contiene dati nella forma di configurazioni e stati rappresentati tramite componenti Kubernetes. Un archivio di tipo key-value è invece il caso classico di archivio impiegato da Kubernetes, in particolare il blocco etcd presente in un cluster predefinito, in generale è una soluzione più complessa ma scalabile più efficientemente.

#### eBPF in Cilium

Il meccanismo con cui Cilium carica e fa uso di programmi eBPF è particolarmente complesso. All'interno del codice sorgente [17] di Cilium sono distinguibili due directory che regolano il contenuto dei programmi eBPF e la loro implementazione: *cilium/bpf* e *cilium/pkg*.

Nella directory *bpf* si trova il codice utilizzato dalla CNI per realizzare una varietà di programmi eBPF dedicati alla dichiarazione di regole di indirizzamento e raccolta di dati per la visibilità. In questo caso il codice è prevalentemente scritto in linguaggio C e definisce diversi *header* files utili per la scrittura di programmi eBPF più complessi.

Nella directory *pkg* è contenuta la maggior parte del codice che regola il funzionamento stesso di Cilium, in questo caso nelle subdirectory *bpf*, *ebpf* e *maps* si trova il contenuto inerente a eBPF. In particolare tutto il codice qui presente è scritto in linguaggio Go e si serve dell'interfaccia del codice C menzionato precedentemente per ottenere i programmi e le mappe eBPF necessarie al corretto funzionamento della CNI. Nella subdirectory *ebpf* si trovano le modalità di creazione e caricamento preventivo delle mappe, qui è definita la struttura delle mappe che saranno utilizzate, mentre in *maps* è presente il codice per compilare i campi di ciascuna mappa in base allo stato desiderato.

Successivamente i programmi e le mappe richieste saranno caricati grazie al contenuto della subdirectory *bpf* in cui è situato il programma *collection.go* tramite cui sono create quelle che vengono dette *collection*: una modalità unica in cui Cilium

## 4.2. Esempi di Soluzioni di Networking per Kubernetes

definisce un insieme di programmi e mappe. In particolare una collection raggruppa i programmi sotto un campo del tipo  $x/y$  dove  $x$  indica l'id del programma o della mappa e  $y$  indica un preciso slot, ossia un determinato indice all'interno della mappa di tipo array che si occupa di tail calls, in cui il programma dovrà essere salvato. Queste collection sono definite in formato ELF ma non vengono caricate all'interno del kernel con questa sintassi. Successivamente infatti le collection sono tradotte in programmi con una specifica section che corrisponde al nome della funzione C di riferimento e vengono caricati mantenendo coerenza con i dati contenuti nel campo  $x/y$ . Così facendo si ottengono una serie di mappe e programmi eBPF raggruppati sotto la stessa section, con id e slot unici. Questa modalità di lavoro di Cilium è stata confermata tramite l'utilizzo del pacchetto bpftools, confrontando varie istanze differenti dell'ambiente di lavoro e notando come i parametri sopra citati rimangano costanti durante installazioni diversificate della CNI.



## 5. Benchmark

Di seguito sono esposti i risultati ottenuti eseguendo dei raffronti tra le tre soluzioni di networking: *legacy routing*, Istio e Cilium. I tre parametri di interesse sondati per condurre l'analisi sono: *throughput* di dati, latenza di rete e utilizzo di CPU. Tutte le misurazioni sono state svolte su un sistema operativo Ubuntu 22.04.4 installato su una macchina virtuale con 8GB di Ram dedicati e 4 core in utilizzo. La macchina su cui è stata eseguita la VM è un laptop HP con processore AMD Ryzen 7 5700U, 1801 Mhz, 8 core, 16 processori logici.

Per ottenere i dati sono stati utilizzati due servizi di *scraping* e visualizzazione di metriche: *Prometheus* e *Grafana*.

### 5.1. Prometheus

Prometheus [18] è uno strumento open-source per collezionare serie temporali di dati. Una serie temporale è una sequenza di dati fissati nel tempo, in Prometheus è rappresentata da una metrica e un insieme di etichette che identificano univocamente la serie. Per collezionare queste serie è adottato un modello pull in cui un elemento server associato a Prometheus effettua richieste HTTP alla sorgente dati configurata ad intervalli regolari. Per operare in questo modo è necessario che il cluster esponga un endpoint HTTP dedicato allo scraping di metriche da parte di Prometheus. Una volta ottenute le metriche Prometheus deve tradurle in un formato leggibile, per cui si serve di *exporters*: librerie atte alla conversione dei dati ottenuti. Al fine di aumentare la fruibilità delle *user interfaces* è utilizzato PromQL per rappresentare i dati: un linguaggio *query* che permette di personalizzare l'UI per la rappresentazione di metriche.

La configurazione di Prometheus si appoggia su un file `.yaml` in cui dichiarare l'intervallo temporale di scraping e l'obiettivo da raggiungere per ottenere i dati. Se per esempio il servizio richiesto fosse `localhost:8080` con tempo di scraping di 15 secondi, Prometheus invierà una richiesta HTTP GET a `http://localhost:8080/metrics` ogni 15 secondi per raccogliere i dati esposti dall'applicazione. Grazie a questa implementazione tramite un manifest in formato `.yaml` Prometheus è facilmente installabile all'interno di un cluster sotto forma di servizio. In questo modo potrà

## 5. Benchmark

eeguire operazioni di service discovery e comunicare con ogni nodo incluso nell'obiettivo. Tutti i dati ottenuti sono immagazzinati nel database Prometheus e sono accessibili tramite l'UI dedicata o applicazioni di terze parti.

Le query utilizzate per accedere ai dati sono mostrate nel listato 5.1, le prime due sono state condensate per ottenere un unico risultato di bytes ricevuti e trasmessi al secondo, osservando così il *throughput* totale all'interno del cluster. La terza query indica l'utilizzo di CPU per unità di tempo riferito all'intero cluster, il dato è ottenuto in relazione al numero totale di core allocati per la VM, ossia 4, nonostante sia possibile che il sistema operativo limiti l'utilizzo di CPU da parte di processi concorrenti per periodi di tempo prefissati, ciascuna implementazione è coerente sotto questo aspetto quindi confrontabile. Per ottenere la latenza è stato utilizzato il tempo necessario per il pull di una metrica tramite la quarta query, in questo modo si osserva il tempo necessario per elaborare e rispondere ad una richiesta HTTP.

```
1 sum (rate (container_network_receive_bytes_total{kubernetes_io_hostname=~"~$Node$"}[1m]))
2 sum (rate (container_network_transmit_bytes_total{kubernetes_io_hostname=~"~$Node$"}[1m]))
3 sum (rate (container_cpu_usage_seconds_total{id="/" ,kubernetes_io_hostname=~"~$Node$"}[1m]))
4 scrape_duration_seconds{instance="localhost:9090", job="Nodo di interesse"}[1m]
```

Listing 5.1: Query PromQL utilizzate.

## 5.2. Grafana

Grafana [19] è una piattaforma open-source per la visualizzazione e l'analisi dei dati, utilizzata principalmente per monitorare e gestire i dati di serie temporali. Permette l'integrazione di metriche ricavate da Prometheus in dashboards personalizzabili. In particolare possiede diverse dashboards dedicate all'osservazione di ambienti cloud native e offre una vasta gamma di personalizzazione. Implementare Grafana è stato essenziale per accedere ai dati in maniera semplice e precisa. Tutte le misurazioni realizzate sono state sondate tramite dashboards e i risultati sono basati su osservazioni della durata media di un'ora di lavoro del cluster. Per integrare Grafana e Prometheus all'interno di un cluster sono necessari dei file .yaml dedicati alla configurazione dei due operatori. Nel caso *legacy routing* sono stati utilizzati i manifest contenuti nel repository *techiescamp/kubernetes-prometheus* mentre nei casi di Cilium e Istio sono forniti manifests dedicati che includono dashboards personalizzate.

## 5.3. Risultati ottenuti

Tutte le misurazioni sono state effettuate su un cluster Kubernetes creato con Kind. Al fine di popolare il cluster e osservare un'architettura a microservizi è stato utilizzato il contenuto del repository *Microservices-Demo* [20]. L'applicazione simulata dal repository, è *web-based* e pensata per dimostrare la possibilità di dividere un servizio cloud in diversi microservizi, scritti in linguaggi informatici differenti, ma coesistenti nello stesso ambiente e comunicanti.

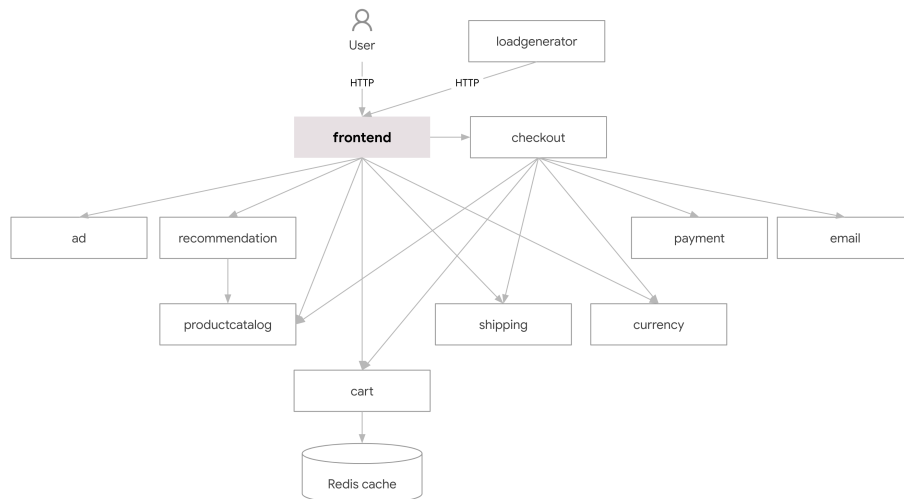


Figura 5.1.: Architettura di *Online Botique*.

In figura 5.1 è mostrata la topologia dei servizi. Il blocco *loadgenerator* è di particolare importanza in quanto si occupa, tramite un programma python, di generare costantemente traffico nella rete simulando possibili richieste *HTTP* e *TCP* indirizzate dagli utenti al servizio. Il blocco *frontend*, accessibile agli utenti, riceve la totalità delle richieste e si occupa successivamente di portarle al servizio di riferimento. Dispiegando l'applicazione all'interno di un cluster Kubernetes otterremo quindi undici pod rappresentanti gli undici blocchi della topologia che comunicano vicendevolmente. Successivamente sono state eseguite le misure nei tre casi precedentemente citati, installando quindi i componenti Cilium e Istio singolarmente quando necessario. Grafana e Prometheus sono stati inseriti in un namespace dedicato al monitoring per la raccolta e visualizzazione dei dati. Le misurazioni sono state ottenute osservando le dashboard di Grafana e utilizzando i cursori per ottenere valori precisi di latenza, traffico e utilizzo delle risorse core.

## 5. Benchmark

### Throughput

Il throughput rappresenta la quantità di dati che un sistema può trasmettere in un determinato intervallo di tempo e può essere indicativo delle capacità di scaling e delle prestazioni complessive di un'architettura di microservizi. Un throughput elevato è essenziale per applicazioni che richiedono un'alta velocità di trasferimento dati, come servizi di streaming, e-commerce ad alta frequenza di transazioni e sistemi di analisi in tempo reale. Cilium è in grado di ottenere un throughput più elevato rispetto alle soluzioni tradizionali grazie all'uso di eBPF, che permette di gestire il traffico di rete in modo più efficiente direttamente nel kernel. Questo elimina la necessità di passare i pacchetti tra il kernel e lo spazio utente, riducendo l'*overhead* e migliorando le prestazioni di rete. Istio in qualità di service mesh gestisce il traffico in maniera più intelligente rispetto a kube-proxy introducendo operazioni di load balancing che aumentano notevolmente le prestazioni.

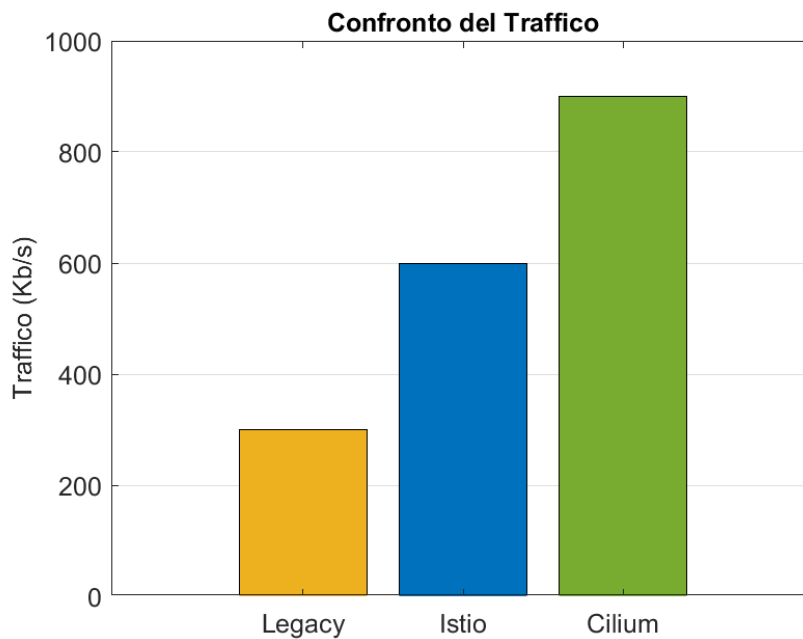


Figura 5.2.: Confronto sul *Throughput* totale.

### Latenza

La latenza misura il tempo che intercorre tra l'invio di una richiesta e la ricezione della risposta. È un parametro cruciale per valutare la reattività e l'efficienza operativa dei microservizi. Confrontare la latenza tra i tre servizi aiuta a determinare quale soluzione offre tempi di risposta più rapidi, un aspetto critico per applicazioni sensibili al ritardo come giochi online, applicazioni finanziarie e sistemi di comunica-



zione in tempo reale. Cilium riduce significativamente la latenza rispetto a Legacy e Istio grazie all'elaborazione delle politiche di rete e sicurezza direttamente nel kernel tramite eBPF. Effettuando l'elaborazione nel kernel per ogni pacchetto vengono minimizzati i tempi di risposta, senza quindi causare latenze aggiuntive date da trasferimenti in user space. Istio introducendo un sidecar proxy aggiunge un ulteriore ritardo all'elaborazione del pacchetto. Nonostante si serva di operazioni di *prefetching* e *caching* dei percorsi di rete per migliorare la latenza la soluzione risulta inferiore a Cilium. L'utilizzo di iptables quindi aumenta considerevolmente il tempo di risposta della rete e nel caso di kube-proxy non sono utilizzati meccanismi efficienti di load balancing risultando in latenze elevate.

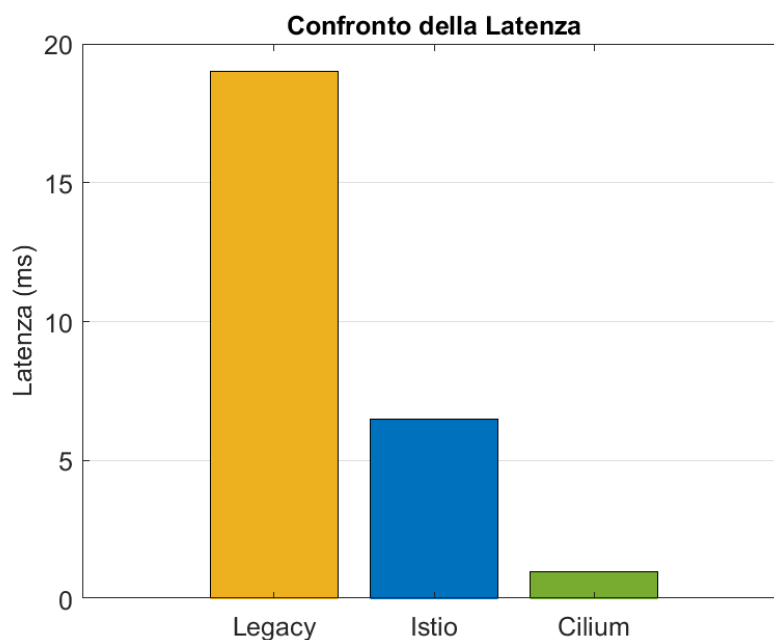


Figura 5.3.: Confronto sulla latenza.

### Consumo di CPU

Il consumo di CPU è un indicatore chiave dell'efficienza delle risorse di un sistema. Misurare e confrontare il consumo di CPU dei tre servizi permette di valutare quale soluzione utilizza le risorse del processore in modo più efficiente. Questo parametro è particolarmente utile per gestire i costi operativi e per pianificare la scalabilità dell'infrastruttura. Un basso consumo di CPU può ridurre i costi energetici e aumentare la durata dei componenti hardware, rendendo la soluzione più sostenibile e conveniente nel lungo termine. Cilium ottimizza il consumo di CPU sfruttando le capacità di eBPF per eseguire operazioni di rete e sicurezza direttamente nel kernel,

## 5. Benchmark

riducendo il carico di lavoro sullo spazio utente. Questo approccio minimizza il numero di *context switch* e le interruzioni del kernel, riducendo l'utilizzo complessivo della CPU. Istio per manovrare il cluster introduce necessariamente un container proxy per ogni pod presente. Questa soluzione aumenta sostanzialmente il carico sulla CPU che deve gestire un numero di container aggiuntivi pari al numero di pod presenti in un cluster. La soluzione è quindi inferiore dal punto di vista della scalabilità rispetto a Cilium e gravosa sulle risorse di CPU nonostante di conseguenza offra un'ottima visibilità.

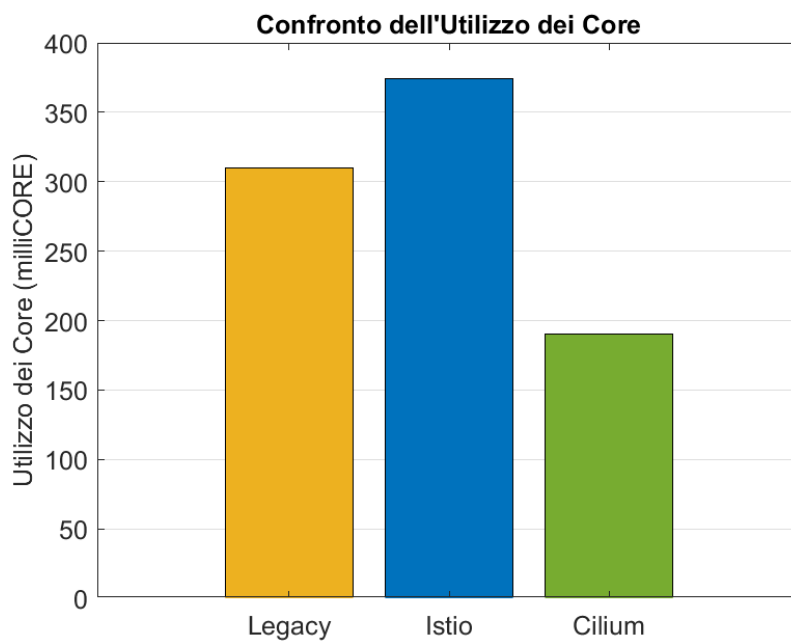


Figura 5.4.: Confronto sull'utilizzo di CPU.

## 6. Conclusioni

Esplorare le possibilità offerte da eBPF, approfondendo le sue applicazioni in ambiente cloud native, ne ha dimostrato, tramite una serie di misurazioni, i vantaggi in termini di latenza, throughput e utilizzo delle risorse CPU rispetto a soluzioni alternative, correntemente in utilizzo in servizi moderni. Grazie alla sua capacità di eseguire codice personalizzato direttamente nel kernel, permette un'elaborazione dei pacchetti più efficiente e veloce, eliminando molti dei *bottlenecks* tipici delle soluzioni che operano esclusivamente in user space. L'adozione di Cilium ha mostrato come eBPF possa migliorare notevolmente le performance delle applicazioni cloud native, rendendolo una scelta ideale per ambienti ad alta scalabilità e per l'implementazione di microservizi. Questa tecnologia non solo ottimizza l'utilizzo delle risorse, ma aumenta anche la flessibilità e la sicurezza delle applicazioni.

Per raggiungere un'analisi più completa delle possibilità operative di Cilium si potrebbero effettuare misurazioni in ambienti più performanti e popolati. La raccolta di metriche in ambienti più vicini allo stato dell'arte, utilizzando infrastrutture più potenti e scalate permetterebbe di valutare le performance di eBPF e Cilium in scenari più realistici e complessi, fornendo dati ancora più rappresentativi dell'efficacia di questa tecnologia. Inoltre tutti i dati raccolti e presentati sono relativi a traffico interno ad un cluster, eseguire esperimenti che coinvolgano traffico proveniente dall'esterno del cluster consentirebbe di analizzare come eBPF e Cilium gestiscono le interazioni con reti esterne e di valutarne la capacità di mantenere alte performance scenari di questo tipo. Grazie alla sua natura dinamica e versatile eBPF è particolarmente adatto ad applicazioni dedicate alla sicurezza, ad esempio attraverso l'esplorazione del progetto Tetragon, integrabile in Cilium, si osserverebbe l'efficacia di eBPF non solo come strumento per migliorare le performance di rete, ma anche come tecnologia di sicurezza avanzata, capace di rilevare e mitigare minacce in tempo reale.

In conclusione, i risultati ottenuti confermano che eBPF e Cilium rappresentano una soluzione valida per la gestione delle reti in ambienti cloud native. Gli sviluppi futuri e le ulteriori ricerche indagheranno le potenzialità di questa tecnologia, rendendola sempre più indispensabile per le architetture ICT moderne.



## 7. Ringraziamenti

Al termine di questo percorso desidero ringraziare chi mi è sempre stato vicino con affetto e chi ha creduto in me. In particolare ringrazio i miei genitori per il loro supporto e per avermi affiancato nella mia crescita accademica. Ringrazio anche il professore Walter Cerroni per avermi guidato attraverso la realizzazione di questo elaborato, dedicandomi il suo tempo e la sua conoscenza.



# Bibliografia

- [1] Isovalent Named Gartner Cool Vendor. <https://isovalent.com/blog/post/gartner-cool-vendor/#:~:text=I'm%20excited%20to%20share,teams%20to%20dramatically%20improve%20the>, 2024. [Online; accessed 2-July-2024].
- [2] Liz Rice. *Learning eBPF*. " O'Reilly Media, Inc.", 2023.
- [3] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), feb 2020.
- [4] A gentle introduction to eBPF. <https://www.infoq.com/articles/gentle-linux-ebpf-introduction/>, 2024. [Online; accessed 2-July-2024].
- [5] IOvisor bpf-docs. <https://github.com/iovisor/bpf-docs/blob/master/eBPF.md>, 2024. [Online; accessed 2-July-2024].
- [6] Docker Guides 2024. <https://docs.docker.com/guides/docker-overview/>, 2024. [Online; accessed 23-June-2024].
- [7] Kubernetes Documentation v1.30. <https://kubernetes.io/docs/home/>, 2024. [Online; accessed 23-June-2024].
- [8] Devoteam Kind review. <https://www.devoteam.com/expert-view/review-how-is-kind-simplifying-testing-and-local-development-on-kubernetes/>, 2024. [Online; accessed 02-July-2024].
- [9] Helm Docs v3.14.0. <https://helm.sh/docs/>, 2024. [Online; accessed 23-June-2024].
- [10] Victor Marmol, Rohit Jnagal, and Tim Hockin. Networking in containers and container clusters. *Proceedings of netdev 0.1*, pages 14–17, 2015.
- [11] N de Bruijn. ebpf based networking, 2017.

## Bibliografia

- [12] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019.
- [13] What is a service mesh? <https://blogs.mulesoft.com/dev-guides/microservices/what-is-a-service-mesh/>, 2024. [Online; accessed 02-July-2024].
- [14] Istio Documentation v1.22.2. <https://istio.io/latest/docs/>, 2024. [Online; accessed 28-June-2024].
- [15] Istio & Envoy. <https://www.kubecost.com/kubernetes-devops-tools/istio-envoy/>, 2024. [Online; accessed 02-July-2024].
- [16] Cilium Documentation 1.16.0. <https://docs.cilium.io/en/stable/>, 2024. [Online; accessed 30-June-2024].
- [17] Cilium Source Code 1.16.0-rc.1. <https://github.com/cilium/cilium>, 2024. [Online; accessed 30-June-2024].
- [18] Prometheus Documentation. <https://prometheus.io/docs/introduction/overview/>, 2024. [Online; accessed 1-July-2024].
- [19] Grafana Documentation. <https://grafana.com/docs/grafana/latest/introduction/>, 2024. [Online; accessed 1-July-2024].
- [20] Google’s Microservices-Demo. <https://github.com/GoogleCloudPlatform/microservices-demo.git>, 2024. [Online; accessed 24-June-2024].
- [21] Kubernetes Prometheus Manifest. <https://github.com/techiescamp/kubernetes-prometheus>, 2024. [Online; accessed 10-July-2024].



# A. Appendice

Di seguito è fornita una guida per avviare un cluster Kubernetes e riprodurre le sperimentazioni discusse. Per operare è necessario un sistema operativo Ubuntu 22.04.4, o qualsiasi altra distribuzione Linux, per poter utilizzare un kernel Linux. Sono necessari diversi software per poter utilizzare le soluzioni proposte, per prima cosa sarà mostrato come installare ogni singolo software e successivamente i comandi necessari ad avviare un cluster e lavorare al suo interno.

## Installazione dei Software

Per installare Docker è sufficiente accedere al sito ufficiale [6], all'interno del Docker manual, nella sezione *Install*, e seguire i procedimenti consigliati in base alla distribuzione Linux in utilizzo. Poiché Docker necessita di permessi *super user*, come la maggiorparte dei software utilizzati, è buona pratica creare un gruppo *sudo* e aggiungere il proprio utente ad esso, inoltre una volta installato Docker per testarlo basterà lanciare un container *hello-world*, tramite i comandi mostrati nel seguente listato:

```
1 sudo groupadd docker
2 sudo usermod -aG docker $USER
3 docker run hello-world
```

Per installare Kubernetes è consultabile la documentazione Kubernetes [7] che guida all'installazione in maniera esaustiva, per poter utilizzare Cilium è necessario che la versione del client kubectl sia superiore alla 1.14.0. Per l'installazione di Helm la documentazione [9] offre una grande varietà di soluzioni, la più immediata per le distribuzioni Debian/Ubuntu è senza dubbio tramite *apt*. La documentazione Kind non è particolarmente dettagliata, ma offre un'installazione tramite binario o tramite Go, si consiglia di installare il binario di Kind e spostarlo sotto la directory */usr/bin* per evitare problemi di percorso dovuti ad un'installazione tramite Go.

A questo punto si è in possesso di tutto il necessario per avviare un cluster amministrato tramite kube-proxy, di seguito è dettagliata l'installazione di Istio e Cilium. L'installazione di Istio è dettagliata passo a passo all'interno della documentazione [14], in particolare nella pagina *Getting Started*. Durante l'installazione sarà scari-

## A. Appendice

cata tramite comando *curl* una directory contenente il necessario per lavorare con Istio. Il binario di Istio che permette di utilizzare comandi dedicati tramite *istioctl* è contenuto all'interno di questa directory, per poterlo utilizzare è necessario quindi aggiungere all'interno del file *.bashrc* contenuto nella home di sistema una riga di codice per esportare il percorso di Istio. Il contenuto del comando è mostrato nel seguente listing.

```
1 export PATH=$PWD/bin:$PATH
```

Alternativamente si può spostare l'eseguibile all'interno di un percorso noto come */usr/bin* nonostante sia sconsigliato.

Per installare Cilium sarà invece necessario il client Cilium, scaricabile tramite il comando:

```
1 CILIUM_CLI_VERSION=$(curl -s https://raw.githubusercontent.com/cilium/
  cilium-cli/main/stable.txt)
2 CLI_ARCH=amd64
3 if [ "$(uname -m)" = "aarch64" ]; then CLI_ARCH=arm64; fi
4 curl -L --fail --remote-name-all https://github.com/cilium/cilium-cli/
  releases/download/${CILIUM_CLI_VERSION}/cilium-linux-${CLI_ARCH}.
  tar.gz{,.sha256sum}
5 sha256sum --check cilium-linux-${CLI_ARCH}.tar.gz.sha256sum
6 sudo tar xzvfC cilium-linux-${CLI_ARCH}.tar.gz /usr/local/bin
7 rm cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
```

che si occupa in maniera dettagliata di verificare l'installazione dopo averla eseguita. Si noti che per utilizzare Cilium la prima installazione su un cluster sarà comunque manuale e solo successivamente si potranno usare i comandi del client che risultano più immediati, inoltre l'installazione presentata di seguito è relativa all'utilizzo di Kind, in casi diversi è consigliato consultare la documentazione Cilium.

### Installazione di CNI e Service Mesh

La prima installazione di Cilium deve essere eseguita come segue per avere un primo pull dell'immagine di Docker sul cluster, successivamente, dopo aver installato il client, è possibile usare i comandi del client Cilium per installarlo e verificare lo stato del cluster. Per creare un nuovo cluster con kind, è possibile usare il comando seguente:

```
1 sudo kind create cluster --config=kindconfig.yaml
```

All'interno del file *kindconfig.yaml* è contenuta la definizione della struttura del cluster da realizzare, nel caso di questo deployment è fondamentale dichiarare i flag necessari a disabilitare CNI di default e annullare il deployment di kube-proxy, di seguito un manifest esempio:

```

1 kind: Cluster
2 apiVersion: kind.x-k8s.io/v1alpha4
3 nodes:
4 - role: control-plane
5 - role: worker
6 networking:
7   disableDefaultCNI: true
8   kubeProxyMode: none

```

Successivamente è necessario aggiungere il repository Helm di Cilium ed eseguire il pull e caricare l'immagine Docker di Cilium:

```

1 sudo helm repo add cilium https://helm.cilium.io/
2 sudo docker pull quay.io/cilium/cilium:v1.15.4
3 sudo kind load docker-image quay.io/cilium/cilium:v1.15.4

```

A questo punto Cilium entrerà in funzione all'interno del cluster, il processo di deployment del CNI può durare anche minuti, il seguente comando permette di verificare lo stato del deployment in tempo reale:

```

1 sudo cilium status --wait

```

Una volta abilitato Cilium è possibile aggiungere diverse altre applicazioni, in questo caso è mostrato come abilitare Hubble e l'esportazione di metriche per Prometheus e Grafana:

```

1 sudo helm upgrade cilium cilium/cilium --version 1.15.4 \
2   --namespace kube-system \
3   --reuse-values \
4   --set hubble.relay.enabled=true \
5   --set hubble.ui.enabled=true \
6   --set prometheus.enabled=true \
7   --set operator.prometheus.enabled=true \
8   --set hubble.metrics.enableOpenMetrics=true \
9   --set hubble.metrics.enabled="{dns,drop,tcp,flow,port-distribution,
   icmp,httpV2:exemplars=true;labelsContext=source_ip,source_namespace,
   source_workload,destination_ip,destination_namespace,
   destination_workload,traffic_direction}"

```

Successivamente per osservare il traffico tramite Hubble è necessario abilitare il *port forward*, verificare lo status e se desiderato accedere alla user interface:

```

1 cilium hubble port-forward&
2 hubble status
3 cilium hubble ui

```

Se il port forwarding fallisce, è possibile configurare le regole nft dell'host locale tramite questi comandi:

## A. Appendice

```
1 sudo nft list ruleset
2 sudo nft add chain ip filter input { type filter hook input priority 0
   \; }
3 sudo nft add rule ip filter input tcp dport 4245 accept
```

Se l'interfaccia utente non è accessibile, è necessario aprire la porta corrispondente:

```
1 sudo nft add rule ip filter input tcp dport 8080 accept
```

Per applicare i manifest di monitoraggio per Prometheus e Grafana, in particolare per le metriche Cilium è stato utilizzato il seguente comando che applica il contenuto del repository github di Cilium:

```
1 sudo kubectl apply -f https://raw.githubusercontent.com/cilium/cilium
   /1.15.4/examples/kubernetes/addons/prometheus/monitoring-example.
   yaml
```

Successivamente sarà necessario abilitare il portforward anche per Grafana e Prometheus:

```
1 sudo kubectl -n cilium-monitoring port-forward service/grafana --
   address 0.0.0.0 --address :: 3000:3000
2 sudo kubectl -n cilium-monitoring port-forward service/prometheus --
   address 0.0.0.0 --address :: 9090:9090
```

Dopo aver eseguito la prima installazione è possibile inizializzare Cilium tramite il comando *install* dedicato o tramite una dichiarazione con Helm.

Per utilizzare Istio è necessario configurare un cluster e avviarlo tramite Kind come mostrato in precedenza. Successivamente è possibile installare Istio specificando la configurazione desiderata, in questo caso è consigliato utilizzare una configurazione predefinita, ed etichettare il namespace scelto per il deployment per abilitare l'utilizzo di sidecar tramite i seguenti comandi:

```
1 istioctl install --set profile=demo -y
2 kubectl label namespace default istio-injection=enabled
```

L'installazione di Istio è terminata, per includere gli strumenti necessari per il monitoraggio è possibile utilizzare il contenuto della directory installata con Istio:

```
1 kubectl apply -f samples/addons
2 kubectl rollout status deployment/kiali -n istio-system
```

Per accedere alle metriche è possibile effettuare il port-forward di Grafana e Prometheus come mostrato in precedenza. Per eliminare il cluster in utilizzo è possibile utilizzare il seguente comando Kind:

```
1 kind delete cluster --name=nome
```

## Installazione Microservices-Demo

Per installare la struttura a microservizi è sufficiente clonare il repository github Microservices-Demo [20] e utilizzare un comando *apply* per aggiungere ciascun pod al cluster. Si consiglia di utilizzare un namespace dedicato per i microservizi così da facilitare la lettura di metriche tramite dashboards. Utilizzare il comando *apply* non sempre garantisce la corretta operatività dei servizi, si consiglia quindi di utilizzare Skaffold, un software che permette di avviare pods all'interno di un cluster includendo set di dipendenze specifiche. L'applicazione Microservices supporta l'installazione tramite Skaffold che, nonostante sia più onerosa, garantisce una corretta comunicazione tra i servizi.

## Monitoraggio

Nei casi già discussi di Cilium e Istio il monitoraggio è fornito da Prometheus e Grafana grazie a manifest integrati nella documentazione. Nel caso legacy è stato utilizzato il deployment di Prometheus fornito dal repository *techiescamp/kubernetes-prometheus* [21]. Si consiglia di aggiungere Prometheus al cluster tramite comando *apply* in un namespace dedicato e di aggiungere la configurazione Grafana fornita all'interno della documentazione stessa nello stesso namespace. Una volta disponibili i due servizi sarà sufficiente eseguire il port forward per Grafana e da browser aggiungere una dashboard e configurarla tramite le query mostrate nel listato 5.1.