Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria Informatica

Analisi delle performance di Data-Plane Development Kit in dispositivi virtualizzati P4

Relatore:
Chiar.mo Prof.
Marco Prandini

Correlatori:
Dott. Amir Al Sadi
Dott. Andrea Giovine

Presentata da:
Patrick Di Fazio

Sessione 1
Anno Accademico 2021-2022

Indice

Abstract

Data Plane Development Kit è un set di librerie a basso livello, scritto in linguaggio C, che offre elevate prestazioni a livello di Data Plane, una branca del Software Defined Networking che si occupa di forwarding e che offre numerosi vantaggi sfruttando la tecnologia del Kernel Bypassing. Per analizzare le prestazioni di DPDK sono stati usati dei generatori di traffico collegati a degli switch programmati con Programming Protocol-Indipendent Packet Processors (P4), un linguaggio di programmazione utilizzato per configurare regole e azioni dei dispositivi di rete. Si presenta di seguito l'analisi delle prestazioni con utilizzo della tecnologia P4, che prevede l'impiego di uno o più switch interposti tra un ricevente e il generatore di traffico DPDK. Lo studio è condotto tenendo conto della quantità di pacchetti che vengono persi durante l'inoltro, della velocità effettiva di inoltro e della potenza di calcolo richiesta per effettuare la trasmissione. I risultati suggeriscono una elevata capacità di generazione di pacchetti sfruttando DPDK e mettono in risalto la riprogrammabilità degli switch P4. Diverse infrastrutture generano diversi risultati, con prestazioni variabili. L'analisi dei risultati è utile in funzione ai possibili sviluppi futuri. Unendo le prestazioni di DPDK alla versatilità di P4, sarebbe possibile avanzare un nuovo approccio alle SDN migliorandone le performance a livello Data Plane.

Capitolo 1

Introduzione

Le reti moderne distaccandosi dal classico modello di rete chiusa, possono adottare nuovi sviluppi e innovazioni come quelli introdotti dalle SDN. P4 si propone come linguaggio con un grande potenziale. Grazie alla sua programmabilità, permette infatti di rendere le funzioni che una volta erano cablate nel firmware del dispositivo, aperte, programmabili e modificabili a runtime. L'approccio adottato è infatti quello "top-down", dove è il programmatore che definisce le funzionalità che la rete deve avere, senza essere limitato dall'hardware che il vendor produce, tipico scenario dell' approccio "bottom-up" delle reti tradizionali. Questo consente di allontanarsi dal modello in cui la rete è sviluppata, sfruttando la sinergia tra hardware e software, e di centralizzare lo sviluppo del networking sulla base di programmi scritti dallo sviluppatore che ora è in grado di avere una completa panoramica sulle funzionalità dell'ambiente in cui opera.
DPDK fornisce delle migliorie a livello di Data Plane. Grazie alle sue librerie ottimizzate, riesce a velocizzare il forwarding dei pacchetti sfruttando tecnologie come il Kernel Bypassing, che verranno approfondite in seguito. DPDK riesce infatti a portare nello user-space e quindi a livello utente, le interfacce di rete che prima erano legate al Kernel, delegando alla NIC (Network Interface Controller) il completo controllo dell'applicazione.
Questo paradigma permette di avere una visione completamente nuova della rete. Coniugando queste due tecnologie sarebbe possibile infatti controllare l'instradamento di pacchetti accelerandone le prestazioni, senza essere legati all'hardware del dispositivo.

Software Defined Networks

Con le SDN si disaccoppiano hardware e software, rendendo disponibile una ampia programmabilità di rete che permette di sfruttare anche più nodi, caricare regole dinamicamente e su richiesta dell'amministratore di rete senza avere notevoli perdite di pacchetti. Grazie alla virtualizzazione della rete si possono caricare infatti le regole "on the fly", permettendo di cambiare le configurazioni degli switch e dei router mentre sono accesi, con una perdita di pacchetti quasi nulla. Nelle reti Software Defined si disaccoppiano anche il livello di inoltro dei pacchetti, il cosiddetto Forwarding Plane o Data Plane, dal modello di Control Plane, che si occupa di come i pacchetti vengono inoltrati. Si realizza così una netta divisione che permette lo sviluppo di tecnologie separate per livelli separati, rendendo la rete completamente aperta dal punto di vista del programmatore.
Grazie alla loro flessibilità e dinamicità e grazie alla loro predisposizione ad una rete divisa su più nodi, le SDN hanno trovato luogo nel Cloud Computing, dove è necessario gestire una grande quantità di dati. Restano però ancora aperti i numerosi problemi di sicurezza che un approccio così centralizzato può avere. In **Figura 1** è mostrata l'architettura di una SDN, in particolare i due livelli di Data e Control Plane. [25]

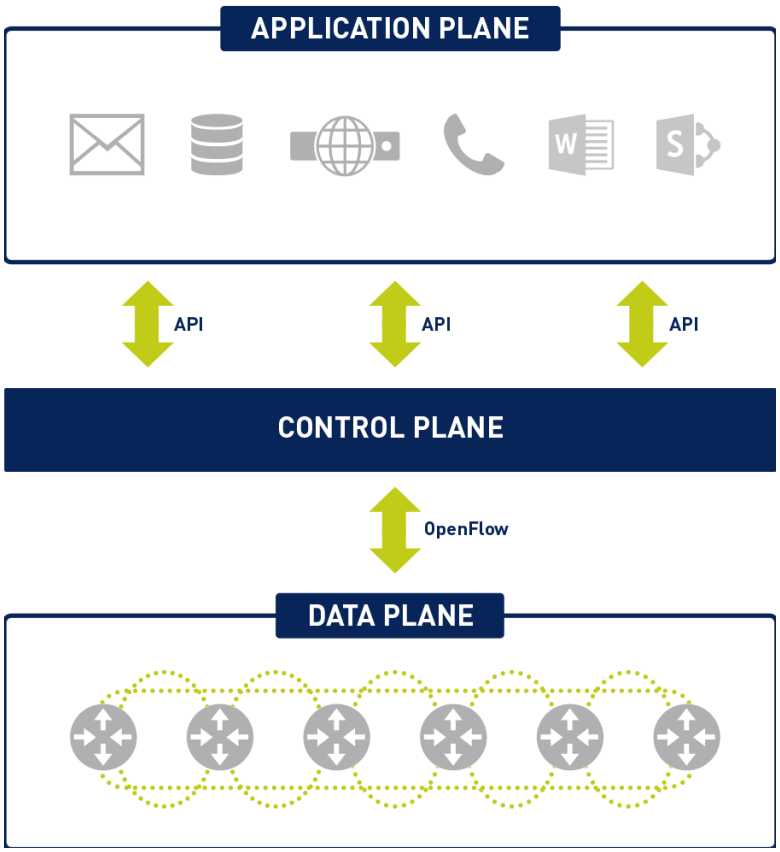


Figura 1: *Architettura delle SDN*

Control Plane

Il Control Plane è il piano di controllo adibito a numerosi ruoli. Nelle SDN che utilizzano OpenFlow [32], il Control Plane si occupa della scelta del percorso del pacchetto in rete, della popolazione della Routing Table e della Forwarding Table. Le nuove reti SDN, invece, puntano alla riprogrammabilità a livello di Data Plane, delegando a questo livello le decisioni sull'instradamento dei pacchetti. P4, ad esempio, riesce a rendere programmabili i dispositivi come gli switch iniettando direttamente le regole per il forwarding e definendo tramite il "codice P4", che verrà compilato in un eseguibile, le azioni che il dispositivo deve svolgere.

Data Plane

Il Data Plane è la parte effettiva che si occupa del forwarding dei pacchetti. È il livello che si occupa del controllo del flusso reale dei dati che passano in rete. Generalmente, dopo aver ricevuto direttiva di inoltro da parte del piano di controllo, si applica il forwarding ai pacchetti. A questo livello è fondamentale per la velocità della rete avere degli algoritmi efficaci tali che usino meno risorse possibili. È in questo piano che poniamo DPDK e P4.

Tecnologie Utilizzate e Casi D'Uso

Di seguito sono elencate alcune delle tecnologie a livello applicativo nelle quali è possibile introdurre le SDN, in particolare P4 e DPDK.

- **Indipendenza dai Protocolli**. Le SDN forniscono una programmabilità versatile che permette di specificare il parsing degli header dei pacchetti di dati (anche a runtime), rendendo i programmi, come quelli su switch P4, adattabili al tipo di protocollo adottato.
- **Monitoraggio di Rete**. Fornendo un Data Plane programmabile, con P4 è possibile monitorare e analizzare il traffico di rete introducendo specifiche regole e azioni da applicare con l'arrivo di determinati header o pacchetti di dati. Un esempio reale è l'introduzione di una DoS protection tramite le SDN [14].
- **Distribuzione della Rete**. Con le SDN in generale, ma nello specifico con P4, è possibile costruire una rete virtuale composta da molti nodi, centralizzata e programmabile "on the fly" che permette di applicare modifiche molto velocemente. Uno caso molto diffuso è quello delle Virtual Private Cloud (VPC) offerte da Amazon Web Services (AWS), ovvero reti virtuali interne che servono per isolare spazi di lavoro [1].
- **Controllo della Rete**. Tramite le SDN è possibile applicare ai dati in arrivo le politiche di Load Balancing, Congestion Control e Logging così da gestire la rete mantenendone il massimo controllo. Queste soluzioni possono essere adottate a livello Enterprise per avere un alto livello di "Governance" sulle proprie reti interne. Un chiaro esempio di Load Balancing basato su SDN è il SLB (Software Load Balancer) di Microsoft Azure [26].
- **Velocità di trasmissione**. Con DPDK è possibile trasmettere dati ad una velocità elevata anche utilizzando hardware di basso costo, rendendo così le SDN adattabili a tecnologie di uso quotidiano. Un caso di utilizzo di DPDK a livello Server è nell'introduzione di questa tecnologia nei Data Center di Lenovo [7].

Capitolo 2

Tecnologie Utilizzate

Data-Plane Development Kit

Data-Plane Development Kit è un framework sviluppato da Intel, con primo rilascio nel 2010 e ora supportato da The Linux Foundation. DPDK consiste in un set di librerie e di driver che consentono di accelerare il packet processing. Il suo punto di forza consiste nell'eseguire le applicazioni direttamente in user-space, in modo che vengano eseguite direttamente sulla NIC. Uno dei punti di forza di DPDK è la performance elevata su piccoli pacchetti di dati. Un flusso di dati con pacchetti di piccole dimensioni (tipicamente 64 byte) viene gestito male dal Networking Stack di Linux perché genera un numero elevato di interrupt (uno per ogni pacchetto). Grazie ai Poll Mode Driver di cui fa uso DPDK, la gestione diventa a polling in modo da usare una minore quantità di interrupt.
DPDK sfrutta inoltre il Kernel Bypassig, così da non consultare il Networking Stack e avere prestazioni migliori. Le applicazioni in questo modo riescono a dialogare direttamente con la NIC che riceve il traffico di dati, senza interferire con il Kernel. DPDK è compatibile con un vasto numero di CPU, architetture e schede di rete. Gli obiettivi del set di librerie sono i seguenti.

- Ricevere e inviare i pacchetti nel minor numero possibile di cicli della CPU, per massimizzare l'efficienza del Data Plane
- Catturare i pacchetti più velocemente possibile
- Essere compatibile con i Fast Path Stack di terze parti

In **Figura 2** è presente l'esempio di un programma che stampa "Hello World" con DPDK.

```
main(int argc, char **argv)
{
        int ret;
        unsigned lcore_id;

        ret = rte_eal_init(argc, argv);
        if (ret < 0)
                rte_panic("Cannot init EAL\n");
        /* >8 End of initialization of Environment Abstraction Layer */

        /* Launches the function on each lcore. 8< */
        RTE_LCORE_FOREACH_WORKER(lcore_id) {
                /* Simpler equivalent. 8< */
                rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
                /* >8 End of simpler equivalent. */
        }

        /* call it on main lcore too */
        lcore_hello(NULL);
        /* >8 End of launching the function on each lcore. */

        rte_eal_mp_wait_lcore();

        /* clean up the EAL */
        rte_eal_cleanup();

        return 0;
}
```

Figura 2: *Hello World con DPDK*

Kernel Bypassing

DPDK adotta il Kernel Bypassing [3] e grazie a questo riesce a raggiungere prestazioni elevate. Il Kernel Bypassing permette di saltare lo stack di networking che è presente nei sistemi operativi, ovvero i vari livelli che un pacchetto di rete deve percorrere per arrivare alla scheda di rete. La completa eliminazione del passaggio del pacchetto nello stack, è possibile grazie all' astrazione che DPDK introduce portando nello user-space le applicazioni. In questo modo il dialogo tra hardware e software non avviene più a livello del Kernel, ma viene portato sulla scheda di rete. Questo avvantaggia il processo di ricezione e invio del singolo dato perché soggetto ad una pipeline più breve e meno operazioni di copia. In **Figura 3** si mostra il livello di direttezza del Kernel Bypassing.

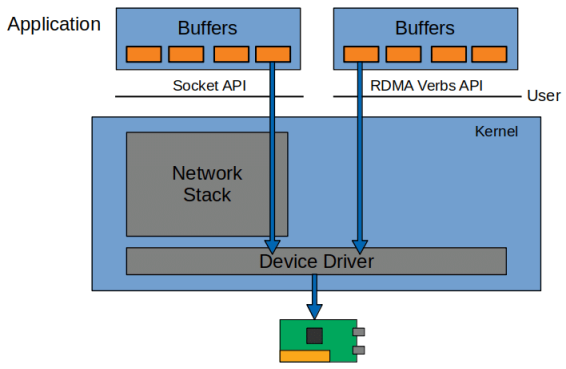


Figura 3: *Accesso diretto alla NIC grazie al Kernel Bypassing* [11]

Vantaggi di DPDK

Polling

DPDK sfrutta i Poll Mode Drivers (PMD) ovvero una modalità a polling che si contrappone al classico modello ad interrupt e che è funzionale nel caso in cui si gestiscano enormi quantità di dati (esempio reti da **1 - 2 Gbit/s**).
Questa modalità prevede che assegnando, ad esempio, i core 1 e 2 (flag -l 0-1) su 4 core di sistema, i primi due core verranno dedicati interamente alla gestione del polling di DPDK. Questo succede perché vengono considerati in modalità non-preemptive e prioritaria: ogni volta che viene richiesto un pacchetto, i due core hanno la priorità di esecuzione sulla gestione di questo. I pacchetti sono gestiti in code (queues) e la CPU li può leggere in base al carico dati che ha in quel momento.
La CPU notifica la NIC nel momento in cui è possibile leggere altre code e così la NIC può inviare nuovamente l'interrupt per i prossimi dati.

Zero Copy

Nel corso dell' arrivo di un pacchetto e del successivo processamento, una nuova copia viene generata all'interno dell'applicazione che lo ha richiesto. Con DPDK, avendo la NIC il diretto accesso ai pacchetti, si dimezza il tempo necessario in quanto le librerie di cui fa uso consentono di impiegare direttamente i dati in arrivo, in modo "raw" (la NIC ne prende totalmente carico), senza doverli copiare.

User Defined Ring Buffer

DPDK delega completamente e in modo trasparente alla NIC la gestione del pacchetto e quindi il Kernel non attiva nessun ring.
In **Figura 4** sono presentate le differenze tra un sistema che usa DPDK e un sistema che non lo usa.

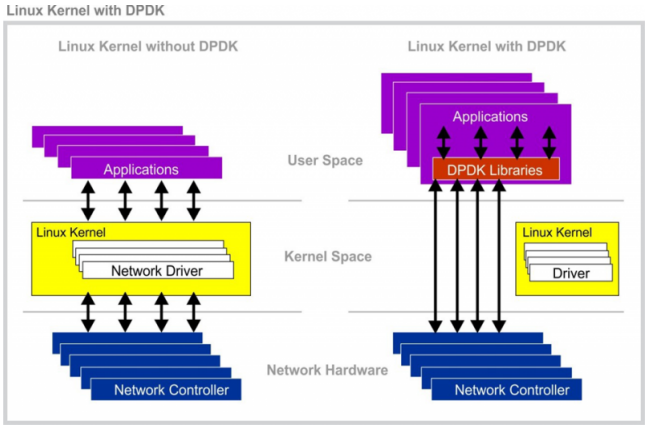


Figura 4: *Vantaggi di DPDK*

DPDK nel dettaglio

Il set di librerie che DPDK offre è diviso in molti punti chiave. Riassumendo, il "core" del framework è suddiviso nei seguenti componenti.

- **Memory Manager** responsabile dell' allocazione di pool di oggetti in memoria
- **Buffer Manager** prealloca la memoria necessaria pronta per l'uso all'esecuzione di DPDK
- **Queue Manager** implementa le code utilizzabili se supportate dalla scheda di rete, sia per la ricezione che per la trasmissione
- **Flow Classification** permette di introdurre i pacchetti più velocemente nel flusso di dati
- **Poll Mode Drivers** evita di usare il meccanismo classico ad interrupt. Dei core della CPU si dedicano completamente al polling rendendo la CPU sempre disponibile alla ricezione o all'invio di pacchetti.

Hugepages

Le Hugepages sono blocchi di memoria di dimensione variabile, composti da singole pagine da 4096 byte. DPDK utilizza le "Persistent Hugepages" allocate in modo dinamico, più stabili delle tipiche "Transparent Hugepages" che sono allocate a runtime in modo automatico.
Usando pagine di grande dimensione è possibile accedere alla memoria più velocemente. Queste sono infatti preallocate per l'esecuzione del programma e permettono di migliorare le performance del sistema, non dovendo allocare memoria per ogni operazione richiesta. Secondo la documentazione di DPDK è consigliato usare Hugepages da 1GB, ovvero preallocare 256.000 pagine di memoria.
Conseguentemente al grande numero di pacchetti che si deve gestire, essendo DPDK pensato per gestire flusso dati su larga scala, si utilizzano le hugepages per evitare di accedere al disco per ogni pacchetto in arrivo. La memoria è preallocata e non ha bisogno di essere riallocata per ogni operazione di rete: si evitano molti problemi e si garantisce velocità di lettura. Usando normali pagine si avrebbero molte più entry da controllare, ciò risulterebbe in un maggiore "tempo di burst" per la CPU.
Le hugepages possono essere da 2M o da 1GB. Nel caso di una Virtual Machine si possono usare 2048K di pagine per far funzionare correttamente la suite DPDK. Per lo studio delle prestazioni in trasmissione e ricezione sono state usate delle pagine da 1GB all' interno di una macchina virtuale.

Environment Abstraction Layer (EAL)

L' Environment Abstraction Layer è il responsabile dell'accesso alle risorse a basso livello e alla memoria. È un livello generico di astrazione che permette di eseguire DPDK senza che esso conosca l'hardware sottostante. I compiti dell' EAL sono molteplici, come ad esempio avviare DPDK, assegnare le istruzioni agli specifici core e riservare zone di memoria per le interazioni con la scheda di rete.
Per il setup, il programma ha bisogno della funzione **rte_eal_init()** che inizializza l' Enviroment Abstraction Layer in modo da accedere alle risorse di basso livello. In particolare utilizzando le librerie **pthread** possiamo delegare ogni esecuzione ad un thread diverso (ad esempio in una macchina con 4 core che dispone di hyperthreading ad 8 thread, si può lanciare il programma delegando a tutti gli 8 core i vari processi). L'inizializzazione come ogni programma **C** è delegata alla glibc, ma l'inizializzazione effettiva è una chiamata alle librerie phread eseguita tramite rte_eal_init(). Al termine del programma, per la deallocazione delle risorse si ha una chiamata a **rte_eal_cleanup()** In **Figura 5** è mostrata l'inizializzazione delle EAL al lancio del programma con la conseguente creazione dei pthread. È quindi

intuibile da questo tipo di startup che il programma sia stato eseguito su due core e quindi utilizzando gli specifici parametri EAL

`./program -l 2`

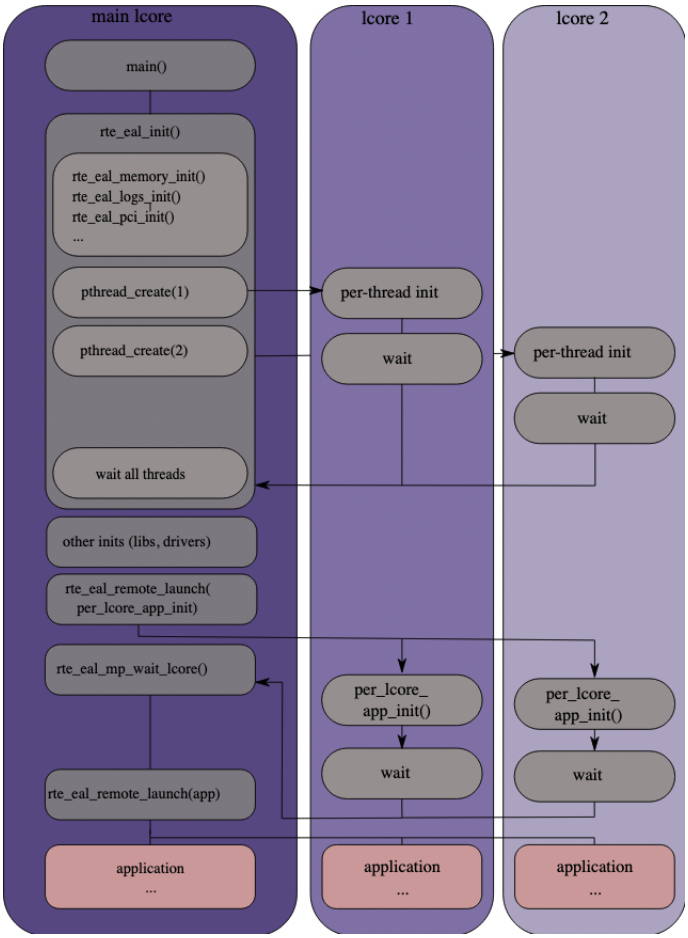


Figura 5: *Inizializzazione dell' EAL*

Poll Mode Drivers (PMD)

I Poll Mode Driver sono un set di API che hanno accesso ai canali di comunicazione in ricezione e trasmissione senza la necessità di utilizzare interrupt. Hanno la funzionalità di configurare i devices e le loro code e funzionano completamente nello user-space. Hanno il compito di processare e inviare i pacchetti nel modo più ottimizzato possibile e in base alle condizioni di carico del sistema al momento e sono configurabili "on the fly".

Programming Protocol-Indipendent Packet Processors: P4

P4 è un linguaggio di programmazione ad alto livello che abbraccia completamente il concetto di SDN, essendo "Protocol Indipendent" [16]. Permette allo sviluppatore di implementare uno Networking Stack personale in hardware di rete come switch o router. La forza di P4 è dare al programmatore tutti gli strumenti necessari per programmare dispositivi di switching in modo da essere facilmente configurabili sulla base di esigenze sempre nuove, come l'introduzione di nuovi header. Grazie alla sua versatiltà può essere introdotto in router, switch o altri dispositivi di rete che processano pacchetti. P4 può essere applicato e compilato per un grande numero di dispositivi che vengono chiamati "Targets". Per ogni dispositivo su cui agisce P4, possono essere predisposte diverse regole.

Architettura

L'architettura PSA (Portable Switch Architecture) [17] descrive le caratteristiche principali che hanno dispositivi di rete come switch, per processare ed inoltrare pacchetti. PSA contiene la libreria di tipi e costrutti disponibili per programmi P4 standard. Il modello PSA ha sei "Programmable Blocks" e due "Fixed Function Blocks". I blocchi sono programmabili tramite P4, mentre i restanti due blocchi dipendono dal target sul quale sono applicati. In **Figura 6** si vede la classica pipeline di una PSA.

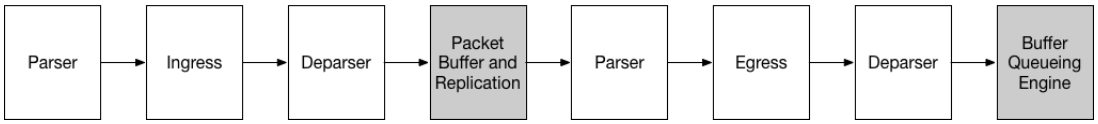


Figura 6: *PSA Pipeline*

Con questa pipeline si validano i pacchetti in arrivo al dispositivo di rete e si applicano le regole che sono stabilite su ingress (match + action). Il primo Function Block permette all'evenienza di duplicare i pacchetti per funzionalità future. Alla fine della pipeline il pacchetto viene serializzato e spedito dal secondo Function Block ai successivi riceventi.

Header

P4 può supportare una gamma di protocolli così vasta grazie alla riconfigurabilità che hanno i suoi header. Gli header di un pacchetto P4 infatti possono essere definiti in modo personalizzato così che si possano adattare all'esigenza della rete, in modo che all'aggiunta in rete di nuovi protocolli, il programma P4 si possa adattare ai nuovi pacchetti. La sintassi degli header è simile a quella delle strutture in C.

Parser e Deparser

Un programma P4 deve essere disposto di un parser e di un deparser [20]. Il parser serve a collegare la rappresentazione a bit del pacchetto alla reale struttura dati che lo rappresenta nel linguaggio P4: i dati vengono inseriti in un pacchetto su cui viene eseguito il parsing. Il deparser ha invece il compito di ricostruire il flusso di bit che verrà ritrasmesso in rete, serializzando l'header e i vari campi che compongono il pacchetto. Dato che nei programmi P4 possono essere introdotti anche nuovi header, è necessario che il parser ed il deparser conoscano la struttura dei pacchetti di rete sui quali vanno ad operare. Il codice sottostante mostra il parsing di header ethernet e IPv4.

```
state start {
return parse_ethernet;
}
state parse_ethernet {
  packet.extract(headers.ethernet);
  return select(headers.ethernet.etherType) {
    0x800 : parse_ipv4;
    default : accept;
  }
}
state parse_ipv4 {
  packet.extract(headers.ipv4);
  return accept;
}
```

BMv2 Target

Uno dei target disponibili su cui lanciare P4 è il behavioral-model 2 che si propone come modello di test, con prestazioni non troppo elevate, ma utile al fine di vedere nella pratica le regole applicate ai dispositivi di rete e utile per la fase di debug. Per incrementare le prestazioni di BMv2, è possibile compilare il target con delle flag specifiche [2].

```
./configure 'CXXFLAGS=-g -O3' 'CFLAGS=-g -O3'
--disable-logging-macros --disable-elogger
```

La suite di BMv2 predispone anche un benchmark per il testing delle prestazioni. Alla fine del tuning si dovrebbero raggiungere le prestazioni di circa 1Gbit/s, molto più elevate rispetto a quelle in assenza di tuning (che si aggirano attorno ai 300 Mbit/s).

Tecnologie di Virtualizzazione

Nelle sezioni successive sono riportate le tecnologie utilizzate durante la fase di progetto inerenti alla virtualizzazione e al setup dell' Hypervisor.

Interfacce TUN/TAP

TUN e TAP sono driver che hanno la funzionalità di creare periferiche virtuali di rete [29]. Queste interfacce virtuali permettono di simulare le connessioni fisiche al sistema, agendo come se fisicamente connesse alla scheda di rete dell'host.
Le interfacce TAP lavorano a livello data-link e quindi a livello 2 dello stack ISO/OSI, mentre le interfacce TUN agiscono da connessione punto a punto e si collocano a livello 3. Sono fisicamente accessibili da /dev/net/tun. Nei setup successivi, il device driver utilizzato sarà MacVTap [30], ovvero una versione semplificata che rimpiazza la combinazione Tun/Tap e supportato da QEMU/KVM [24] [13].

Namespace

I namespace sono spazi di nomi che rinchiudono risorse al loro interno, facendo apparire i processi come separati dal resto del sistema. Sono essenzialmente una astrazione che il sistema operativo fa per far

apparire processi separati come risorse separate. Su Linux un processo di un namespace esistente si trova sotto la cartella $/proc/ < pid > /ns$. Durante l'attività di progetto è stato utile creare namespace per fare dei test tra ambienti separati, come se la comunicazione avvenisse tra calcolatori diversi. Per far comunicare i namespace a ognuno è stato assegnato un IP e sono stati collegati tra loro tramite Virtual Ethernet Device.
I namespaces vengono creati con i seguenti comandi

```
ip netns add net1
ip netns add net2
```

Virtual Ethernet Device (Veth)

Sono dispositivi di rete virtuali che agiscono da tunnel collegando namespaces. Le veth [8] hanno la funzione di bridge o cavo e sono create sempre a coppie. Nella fase di progetto sono state utilizzare per connettere due namespaces su cui risiedevano gli host H1 e H2 collegati agli switch. mentre la coppia di veth viene creata in questo modo

```
ip link add veth1 netns net1 type veth peer name veth2 netns net2
```

Altre Tecnologie

Di seguito sono riportate alcune delle tecnologie legate a DPDK e alle SDN non analizzate in questa tesi.

OpenVSwitch

OpenVSwitch è un' implementazione software di uno switch a più livelli, ovvero offre tutte le funzionalità di uno switch ma può agire, se configurato, anche a livelli OSI superiori. Ha l'obiettivo di rendere programmabile la rete a livello switching. Può essere portato direttamente nello user-space ed è configurabile per sfruttare la tecnologia DPDK a livello Data Plane.

Vector Packet Processing

VPP, Vector Packet Processing, è un Network Stack ad alte prestazioni, scalabile e che esegue direttamente nello user-space. Fa parte del progetto FD.io di The Linux Foundation e usa un framework che può offrire varie funzionalità di rete come switching e routing [33]. VPP può essere utilizzato negli switch e router virtuali, in Firewall e altro ancora. Il vantaggio di VPP è dato dall'abbandono del classico modello di processamento scalare dei pacchetti. Nel modello classico infatti i pacchetti che arrivano al Kernel sono processati uno alla volta in seguito all'arrivo di un interrupt per pacchetto. Con il modello vettoriale, è possibile processare i pacchetti secondo un vettore. Così facendo si ha la notifica dell'arrivo del vettore da un solo interrupt. Il vettore di pacchetti è poi prelevato dalla scheda di rete e processato secondo un set di funzioni. La pipeline di ricezione si riassume nel "Packet Processing Graph" [27] presentato nella **Figura 7** .
VPP alimenta un ring di ricezione di dati fino ad un massimo di 256 pacchetti, il grafo ricostruisce poi tutto l'albero di arrivo dei pacchetti e li processa. Le prestazioni di questa tecnologia sono massimali quando il vettore di ricezione è di dimensione maggiore, dato che con una sola lettura si processano più pacchetti. VPP è in grado di implementare vari plugin, come ad esempio il plugin che sfrutta MEMIF [5] o quello che usa DPDK.

Figura 7: *OVS e OVS with DPDK*

VPP con plugin DPDK

Come si nota in **Figura 7**, è presente un elemento denominato *dpdk-input*. Questo componente è un plugin caricato a runtime da VPP. Per abilitare il supporto a DPDK è inoltre necessario eseguire un "binding" tra la scheda di rete e i driver DPDK. Andando a modificare il file di configurazione di VPP, un esempio chiaro di supporto ai driver DPDK può essere il seguente

```
unix {
    nodaemon
    cli-listen /run/vpp/cli-vpp1.sock
    full-coredump
}
api-trace {
    on
    nitems 500
}
dpdk {
socket-mem 1024,2048

dev 0000:02:00.0 {
```

```
14          num-rx-desc 1024
15          num-rx-queues 2
16      }
17      dev 0000:01:00.0 {
18          num-rx-desc 2048
19          num-rx-queues 4
20      }
21      }
22      plugins { plugin dpdk_plugin.so {enable}}
```

La parte evidenziata mostra come i plugin non siano altro che “shared libraries” caricate a runtime da VPP. Il tag “dev” indica il PCI address della scheda di rete riservata a DPDK. In questa configurazione sono presenti due schede di rete che sfruttano DPDK con due configurazioni differenti per il numero di descriptor nel buffer di ricezione e per il numero di code.

Capitolo 3

Descrizione del progetto

Nella parte progettuale sono state usate e analizzate le tecnologie DPDK e P4 sul piano dell'inoltro dati. Lo scopo del progetto è infatti quello di trovare un modo per velocizzare l'instradamento dei pacchetti su degli switch P4. Per studiare i possibili modi per migliorare l'instradamento, prima sono stati svolti dei test di trasmissione tra due host usando solo DPDK e successivamente tra due host utilizzando solo P4. Alla fine sono stati registrati i risultati e confrontati a livello numerico, per avere un paragone tra le performance delle due tecnologie a livello di forwarding.

La parte di progetto di DPDK ha come scopo quello di analizzare le performance della suite di librerie DPDK, in ambiente virtualizzato e "Bare Metal". Per la generazione e la ricezione di pacchetti è stato usato il tool Pktgen DPDK [34]. La dimensione dei pacchetti è per tutti i test di 1500 byte.

La parte di progetto riguardante P4 ha lo scopo di studiare le prestazioni di ricezione dati degli switch P4, più precisamente sfruttando il target BMv2. I test si riferiscono a un host interno ad una macchina virtuale. Le interfacce sono collegate tramite delle Veth passando per uno o più switch P4. Gli switch P4 hanno un programma che fa accept e forward dei pacchetti, mentre gli host sono collocati su due namespaces differenti.

Sviluppo del progetto: DPDK

Setup

Per avere un supporto ai Poll Mode Drivers, è necessario abilitare l'apposito Kernel driver. VFIO-PCI è un Kernel driver che permette un accesso diretto al dispositivo grazie alle sue API. È consigliato per l'uso con DPDK perché supporta l' IOMMU [9], elemento essenziale per la comunicazione delle CPU con le periferiche, e può sfruttare il PCI Passthrough [23], ovvero può collegarsi direttamente al dispositivo fisico PCI, rendendo le prestazioni ancora migliori. Se si è impossibilitati ad abilitare VFIO, si può comunque usare il modulo standard **uio_pci_generic** incluso nel Kernel Linux, che però presenta qualche limitazione, poiché non supporta la creazione di funzioni virtuali. È inoltre possibile fare il setup delle hugepages e successivamente disattivare la scheda di rete per renderla disponibile all'utilizzo. La NIC viene virtualmente separata dal device driver per essere associata a DPDK.
Il valore **02:00.0** corrisponde al PCI address della scheda di rete [12]. I comandi con prefisso "dpdk" fanno parte della suite fornita con l'applicativo. Per elencare le schede disponibili è possibile utilizzare

```
sudo dpdk-devbind.py -s
```

Oppure manualmente con:

```
sudo lshw -c network -businfo
```

Per velocizzare il setup si possono utilizzare le varie utility fornite dalla suite DPDK per montare le hugepages o fare il binding della scheda di rete. La configurazione sottostante è la stessa per ogni host su cui eseguiranno le istanze di Pktgen.

```
#!/bin/bash
modprobe uio
modprobe uio_pci_generic
dpdk-hugepages.py -p 1G --setup 2G
ifconfig enp2s0 down
dpdk-devbind.py -b uio_pci_generic 02:00.0
```

Pktgen DPDK

DPDK Pktgen è un generatore di traffico che sfrutta DPDK a livello Data Plane che permette di creare pacchetti personalizzati di dimensione, IP/MAC source/destination, arbitrari. Per il corretto funzionamento richiede almeno due core logici. Il core 0 resta in ascolto come "master" e permette il funzionamento della linea di comando di Pktgen, l'altro esegue la ricezione o la trasmissione come "slave". Questo tool può essere eseguito anche nella stessa macchina su più istanze, in modo da avere un metodo di test anche all'interno dello stesso host. Nella interfaccia grafica mostrata in **Figura 9** si osserva come sia possibile manipolare i pacchetti generabili specificandone anche il protocollo o il campo dati, mentre in **Figura 8** si possono notare i diversi pacchetti generati. Un esempio di esecuzione di pktgen è il seguente

```
sudo ./pktgen -l 0-4 -n 3 --proc-type auto -- -T -P -m "1.0, 2.1, 3.2, 4.3"
```

Dove si specificano i logical cores, i memory channels e le opzioni di EAL. In questo caso il core 1 si occupa della porta 0 RX/TX, il core 2 della porta 1 RX,TX, il core 3 della porta 2 RX/TX e il core 4 della porta 3 RX/TX.
Il device driver utilizzato per virtualizzare le schede di rete è MacVTap.

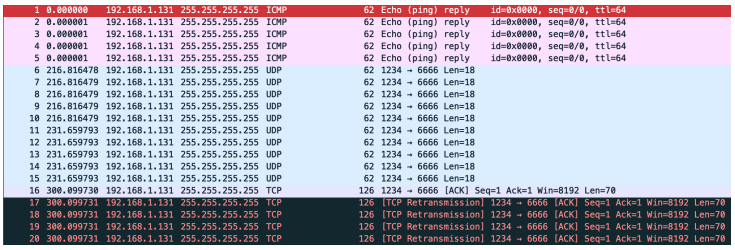


Figura 8: *Manipolazione dei Pacchetti con Pktgen, analisi con Wireshark* [35]

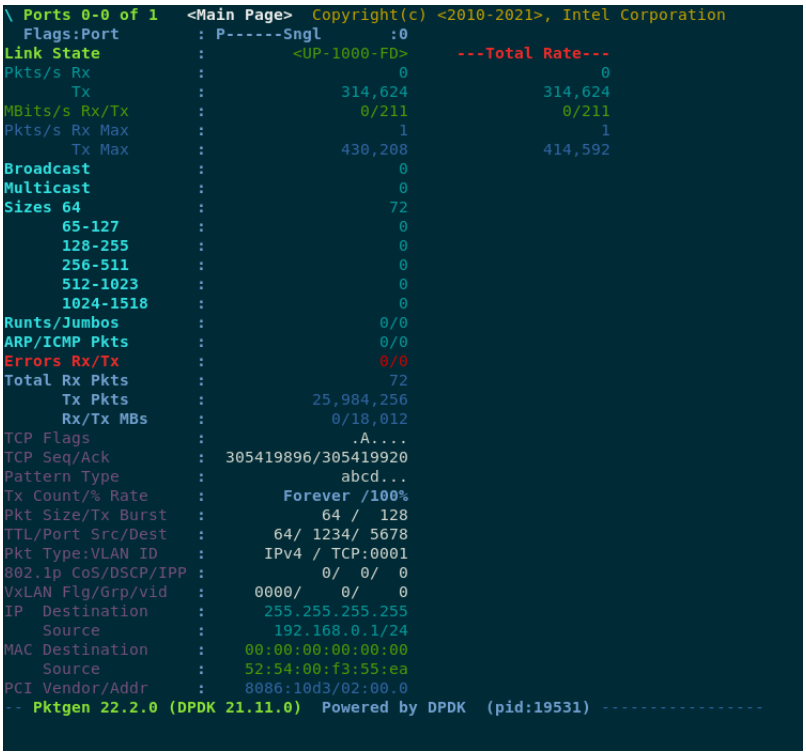


Figura 9: *Interfaccia di Pktgen*

Infrastrutture

Test all' interno della stessa Macchina Virtuale

Nell' infrastruttura in **Figura 10** il test viene eseguito in una macchina virtuale in cui è presente un unico host, collegato ad una scheda di rete paravirtualizzata. Lo stesso host si mette in ascolto e in ricezione. Sono stati utilizzati in totale 2 core ed il master core. Il primo core resta in trasmissione (TX) mentre il secondo in ricezione (RX). I due core sono dedicati alla stessa istanza di Pktgen. I test fanno riferimento alla stessa scheda di rete, l'host dispone di 4GB di RAM.

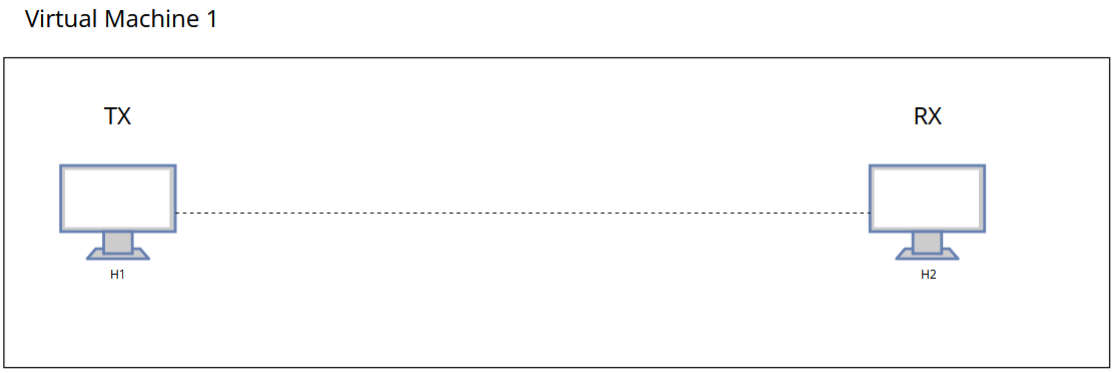


Figura 10: *Test in macchina virtuale con un unico host*

Per configurare le operazioni che deve eseguire il primo core di trasmissione dall'interfaccia di pktgen si può usare la seguente configurazione

```
set 0 src ip 192.168.100.1/24
set 0 dst ip 192.168.100.1/24
set 0 dst mac 00:00:00:00:00:01
set 0 src mac 00:00:00:00:00:01
set 0 size 64
set 0 pattern user
set 0 user pattern test
```

Test tra due Macchine Virtuali

In **Figura 11** il test viene eseguito tra due host in due macchine virtuali collegate in rete interna. I due host sono collegati alla stessa scheda di rete fisica, ma virtualmente a due schede di rete paravirtualizzate con due IP e due indirizzi MAC diversi. H1 è l'host che effettua la trasmissione, mentre H2 effettua la ricezione. Sono stati utilizzati in totale 1 core ed il master core per ogni istanza di Pktgen. Ogni host dispone di 4GB di RAM, per un totale complessivo di 8GB.

Figura 11: *Test tra due NIC di due Host in due macchine virtuali*

In questo caso si specifica l'IP e il MAC del secondo host, in modo da far arrivare i pacchetti a destinazione.

```
set 0 src ip 192.168.100.1/24
set 0 dst ip 192.168.100.2/24
set 0 dst mac 00:00:00:00:00:01
set 0 src mac 00:00:00:00:00:02
set 0 size 64
set 0 pattern user
set 0 user pattern test
```

Test in rete interna in condizioni reali

La **Figura 12** mostra il test effettuato in una rete locale, in cui i due host sono due calcolatori diversi, collegati tra loro da uno switch. In questo caso gli host dispongono di schede di rete fisiche con indirizzi fisici e risiedono all'interno di una rete domestica. H1 è l'host che effettua la trasmissione, mentre H2 effettua la ricezione. Sono stati utilizzati in totale 4 core ed il master core per ogni istanza di Pktgen. Ogni calcolatore dispone di 16GB di RAM.

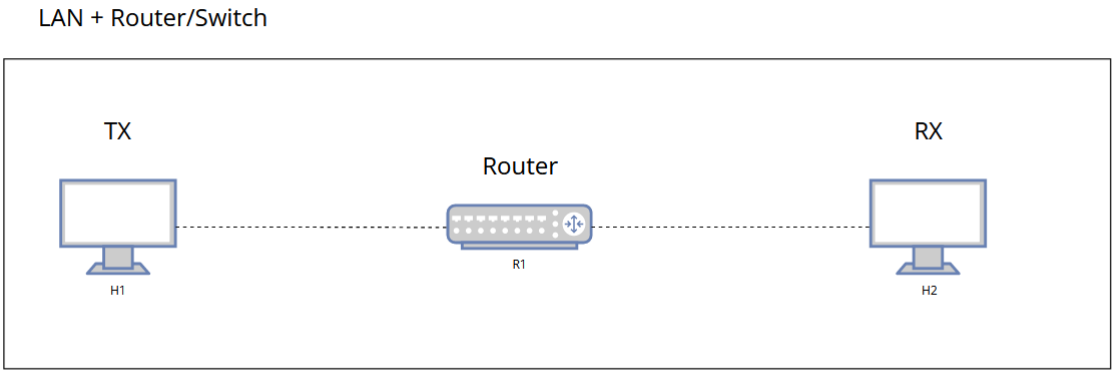


Figura 12: *Test tra due NIC di due Host nella stessa LAN passando per un Router*

```
set 0 src ip 192.168.1.17/24
set 0 dst ip 192.168.1.18/24
set 0 dst mac d8:d3:85:ea:1b:ee
set 0 src mac 00:1b:63:84:45:e6
set 0 size 64
set 0 pattern user
set 0 user pattern test
```

Sviluppo del progetto: P4

Setup

Per creare l'infrastruttura di uno o più switch P4 collegati a due host, si deve per prima cosa disporre di due schede di rete paravirtualizzate. Per avere queste due interfacce di sono stati utilizzati i driver MacVTAP. Una volta create le NIC virtuali, è opportuno creare anche due namespace collegati tramite veth e caricare il programma P4 negli switch. Anche in questa fase di setup ci si è serviti di un software esterno per generare traffico. Questa volta è stato utilizzato IPerf3 [10] su protocollo TCP.

Accept e Forward

Il seguente snippet di codice mostra l' Ingress Processing del semplice programma usato per fare Accept e Forward ed è caricato in tutte le configurazioni degli switch [18]. La versione utilizzata è $P4_{16}$.

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

Le regole di forward saranno aggiunte nei singoli switch in successiva fase di runtime.

Test di trasmissione tra due host con uno switch P4

Nell'infrastruttura mostrata in **Figura 13** si hanno due host collegati tra di loro passanti per uno switch P4.

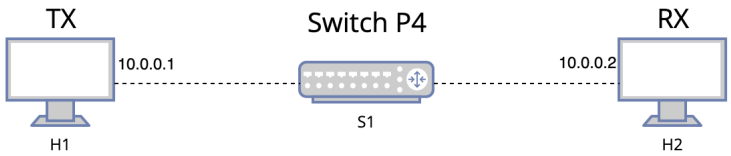


Figura 13: *Test tra due host passando per uno switch P4*

Per avviare il singolo switch si può usare

```
sudo ./simple_switch -i 1@s1-eth0 -i 2@s1-eth1 basic.json
--thrift-port 9091 --notifications-addr ipc:///tmp/bmv2-1-notifications.ipc
```

E poi a runtime per caricare le regole

```
table_clear MyIngress.ipv4_lpm
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.1/32 => 00:00:00:00:00:01 1
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.2/32 => 00:00:00:00:00:01 2
```

Test di trasmissione tra due host con due switch P4

Nello scenario in **Figura 14** sono invece presenti due host collegati tra di loro e passanti per due switch P4. Le configurazioni si duplicano rispetto al caso precedente.

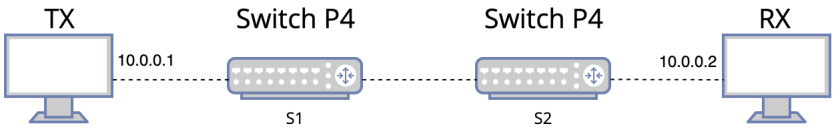


Figura 14: *Test tra due host passando per due switch P4*

Per caricare le regole sui due switch utilizziamo

```
sudo ./simple_switch -i 1@s1-eth0 -i 2@s1-eth1 basic.json
--thrift-port 9091 --notifications-addr ipc:///tmp/bmv2-1-notifications.ipc
sudo ./simple_switch -i 1@s2-eth0 -i 2@s2-eth1 basic.json
--thrift-port 9092 --notifications-addr ipc:///tmp/bmv2-2-notifications.ipc
```

La configurazione a runtime diventa la seguente

Switch S1

```
table_clear MyIngress.ipv4_lpm
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.1/32 => 00:00:00:00:00:01 1
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.2/32 => 00:00:00:00:00:02 2
```

Switch S2

```
table_clear MyIngress.ipv4_lpm
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.2/32 => 00:00:00:00:00:03 1
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.0.1/32 => 00:00:00:00:00:04 2
```

Capitolo 4

Risultati

DPDK

I Test con DPDK hanno riportato i seguenti risultati. I risultati di ricezione e trasmissione virtuale riguardano l'effettivo throughput generabile da DPDK, ovvero la capacità limite di generazione e ricezione di pacchetti che si ha per la specifica infrastruttura. I risultati di ricezione reale invece indicano la quantità di dati effettivamente ricevuti dalla scheda di rete e registrati con il programma **bwm-ng** [31] sul computer host. Appare chiaro che disponendo di una scheda di rete Gigabit è impossibile ricevere un flusso di dati superiore a questo valore. Parte dei test in macchina virtuale sono stati eseguiti anche attivando il PCI Passthrough, ma i risultati riportati si discostano di pochi Mbit/s dai test con le interfacce MacVTap e quindi non sono riportati. Questo distaccamento minimo è possibile grazie alla grande efficienza dei driver MacVTap in combinazione con il Kernel Bypassing effettuato da DPDK.

Infrastruttura	Trasmissione virtuali	Ricezione virtuali	Trasmissione effettivi	Ricezione effettivi
Figura 10	800Mbit/s	800Mbit/s	800Mbit/s	800Mbit/s
Figura 11	3Gbit/s	3Gbit/s	1Gbit/s	1Gbit/s
Figura 12	10Gbit/s	1Gbit/s	1Gbit/s	1Gbit/s

Considerazioni

Nei test mostrati in **Figura 10** la velocità di trasmissione è ridotta rispetto ai test successivi perché DPDK è CPU intensive. In questo caso infatti sullo stesso host ci sono due istanze running di DPDK, una per l'invio e una per la ricezione dei pacchetti che, secondo la semantica dei Poll Mode Drivers, chiedono in modalità polling alla CPU la presenza di pacchetti da processare.

In **Figura 11** invece i test presentano un risultato migliore in termini di performance a livello Data Plane: in questo caso gli interrupt generati dai Poll Mode Drivers sono su due processori diversi in due macchine virtuali diverse risultando in una gestione generalmente migliore. In questo caso, inoltre, ogni macchina virtuale ha due core dedicati, in contrapposizione al precedente caso di singola macchina virtuale con due core totali.

Nei test in **Figura 12** siamo in condizioni reali, ovvero in una rete domestica con un router che si occupa di amministrare i pacchetti spediti dal dispositivo con DPDK. In questo caso, disponendo di una grande capacità di calcolo, ovvero con 4 core dedicati alla generazione dei pacchetti, è possibile riuscire a generare fino a 10 Gbit/s. Non è possibile far passare tale quantità di dati al secondo attraverso il router a causa del limite fisico della scheda di rete. L'invio dei pacchetti infatti sarà limitato alla capacità massima di invio di 1Gbit/s tipico delle NIC standard. Anche se la scheda di rete fosse capace di generare un tale bitrate, il router genererebbe "bottleneck", disponendo anch'esso di porte Gigabit Ethernet. La ricezione virtuale e fisica in questo caso, quindi, coincideranno.

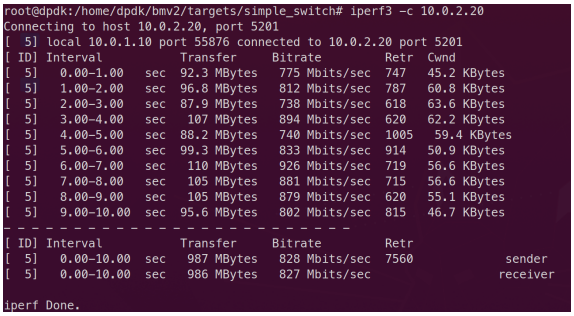
Problematiche

Durante il testing di DPDK, uno dei problemi maggiori è stato monitorare l'invio dei pacchetti senza il tool Pktgen. Sfruttando il Kernel Bypass, DPDK si stacca completamente dal Kernel e si astrae nello user-space, quindi è praticamente impossibile riuscire a leggere i pacchetti in ricezione e in trasmissione se non con dei programmi che sfruttano le librerie apposite di DPDK in user-space. Nei test in macchina virtuale è stato però comunque possibile riuscire a leggere il traffico che DPDK generava, leggendo il buffer delle interfacce MacVTap, potendo monitorare l'effettivo throughput anche da una fonte esterna a Pktgen.

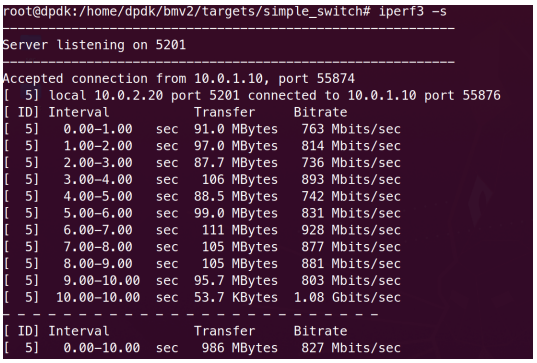
P4

I test con P4 hanno riportato i seguenti risultati. In questo tipo di setup i test sono stati eseguiti solo in una rete interna tra macchine virtuali, quindi non è presente la distinzione tra throughput effettivo e virtuale. Ogni switch riceve i pacchetti, li accetta e li inoltra se l'header fa match con l'header ethernet o l'header IPv4.

Infrastruttura	Trasmissione	Ricezione
Figura 13	830Mbit/s	830Mbit/s
Figura 14	425Mbit/s	425Mbit/s

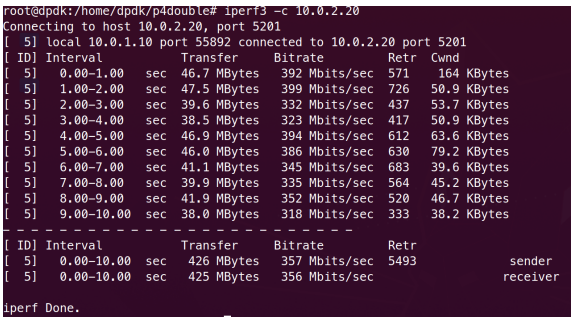


(a) H1

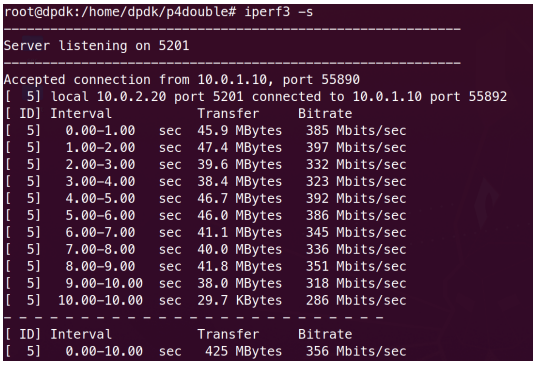


(b) H2

Figura 15: P4 single switch



(a) H1



(b) H2

Figura 16: P4 double switch

Considerazioni

I test in **Figura 15** mostrano le prestazioni di un semplice forwarding. Le prestazioni sono poco sotto il Gigabit in TX (**Figura 15.a**) e in RX (**Figura 15.b**) perché in condizioni ideali su un host reale non virtualizzato, BMv2 ha un throughput medio di 1047 Mbit/s [2]. È quindi ordinario aver un basso bitrate a causa del target su cui si esegue il programma P4, che è infatti solo a scopo di test.

Nei test in **Figura 16** si sottolinea come le prestazioni calano significativamente aggiungendo in cascata un ulteriore switch. In **Figura 16.a** sono presenti i test di trasmissione, in **Figura 16.b** sono presenti i test in ricezione.

Il target BMv2 può avere scarse prestazioni per diversi motivi, alcuni dei quali possono essere i seguenti

- Performance dell'hardware sottostante
- La complessità del programma P4 caricato: all'aumento della complessità del programma segue l'aumento della latenza
- Il compilatore utilizzato per generare il JSON
- Utilizzo in macchina virtuale

Problematiche

Durante il setup dell'ambiente, dopo aver testato il ping tra i namespace H1 e H2, si può procedere testando le prestazioni con il software IPerf3 [10]. Durante questa fase è stato difficile capire perché i dispositivi non riuscissero a stabilire una connessione sfruttando questo tool. Uno dei motivi possibili per spiegare questo evento è che durante il tragitto attraverso gli switch, il parser e il deparser di BMv2 andassero a modificare il checksum degli header IPv4. Per risolvere il problema è possibile disabilitare il **TCP checksum offloading**, che serve per controllare la validità del checksum degli header. È possibile disabilitare tale funzionalità su entrambi gli host attraverso il seguente comando, utilizzando ethtool [4]. In questo modo è possibile sfruttare IPerf3 con il protocollo TCP [6].

```
ethtool -K ethX rx off tx off
```

Punti aperti e sviluppi futuri

P4Pi

In futuro sarebbe interessante studiare tecnologie e altri target P4. Un progetto di nome P4Pi [22] porta la compatibilità di P4 sui Raspberry Pi, dei Single Board Computer economici. Così facendo sarebbe possibile agganciare in modo "Plug and Play" a delle reti questi dispositivi così da poterli utilizzare come switch o router.

OVS-P4

Un altro progetto che estende le funzionalità di P4 è OVS-P4 [15]. Lo scopo di questa estensione di P4 è quella di incrementare le prestazioni degli switch P4 tramite OpenVSwtich sfruttando tecnologie come uBPF [28] e DPDK. Questo è possibile perché la modifica a livello di Data Plane appare trasparente a OVS che continua a comportarsi come switch ma con prestazioni di inoltro maggiori. OVS, come nella sua versione standard, può implementare DPDK, beneficiando dei vantaggi di forwarding che questo framework offre. Sfruttando DPDK non è più necessario separare user-space e Kernel, ma è possibile implementare accanto allo switch virtuale i metodi di Kernel Bypassing. In questo modo si completano in user-space Data Plane e Control Plane, così da avere un bypass completo del Kernel.
OVS sfrutta così DPDK ma mantiene anche la versatilità offerta dai programmi P4 applicati ai virtual switch di cui dispone. In **Figura 17** si notano le differenze tra l'utilizzo di OVS con e senza DPDK.

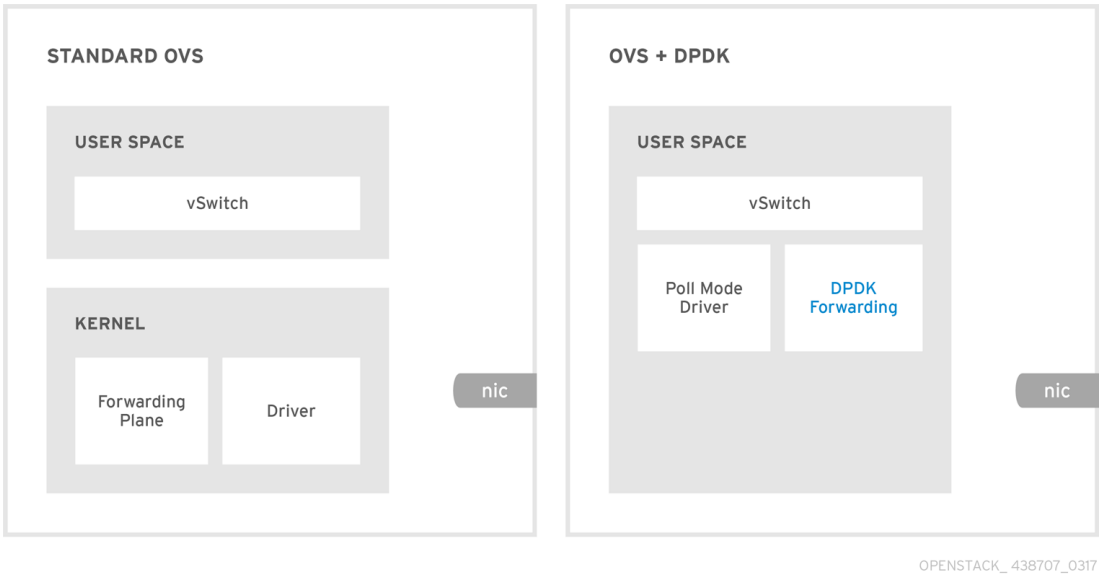


Figura 17: *OVS and OVS with DPDK*

T4P4S

T4P4S (Translator for P4 Switches) è un progetto che ha lo scopo di generare codice P4 ad alte prestazioni [21]. Il compilatore P4 genera un file JSON, T4P4S esegue il parsing e genera codice C che viene poi collegato alle API di DPDK. Questo compilatore sfruttando le API generiche di C riesce a garantire prestazioni elevate, arrivando a superare le prestazioni di OVS+DPDK nei benchmark di forwarding, quindi a puro livello di Data Plane. In **Figura 18** si nota la pipeline di compilazione di un programma P4 in codice C per lo specifico target.

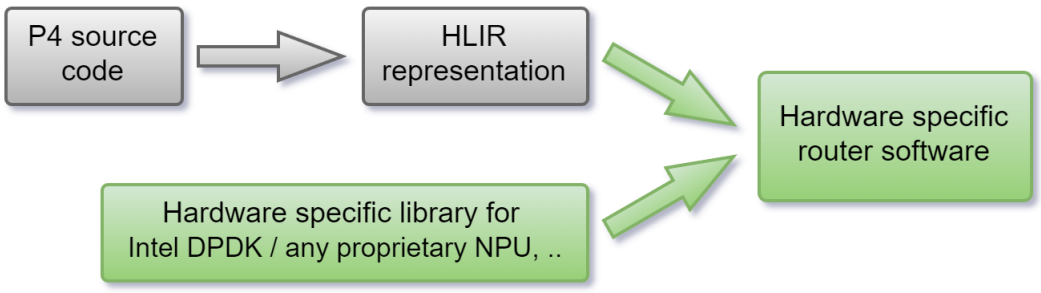


Figura 18: *T4P4S Compiler*

La compilazione segue un modello a pipeline. Si inizia da una rappresentazione intermedia (Intermediate Rapresentation) che sarà poi compilata in codice C grazie alle API del Network Hardware Abstraction Layer. È il Core Compiler che si occuperà di generare il codice per lo specifico target, sfruttando le NetHAL API calls. Il Core di T4P4S implementa infatti il "Packet Parsing" e le "Actions" di P4 e traduce il tutto in funzioni C.

Ambiente di Test

Computer Host

La macchina host su cui sono stati eseguiti i test e su cui sono state lanciate le macchine virtuali dispone dei seguenti requisiti hardware

- CPU: I5 6600K 4 Core 4 Thread 64 Bit
- RAM: 16 GB 2900MHz
- S.O: Linux Ubuntu 20.04 LTS
- NIC: Intel Ethernet 1219-V full duplex 1GBit/s

Virtual Machine

Le Virtual Machine sono copie di una singola macchina virtuale con le seguenti specifiche

- CPU: 2 Cores
- RAM: 4GB
- S.O: Linux Ubuntu 20.04 LTS
- HDD: 50GB

Virtualizzazione

- Software di Virtualizzazione: QEMU/KVM [24]
- Paravirtualized drivers: VirtIO
- Device Drivers: MacVTap

Capitolo 5

Conclusioni

In un futuro in cui è sempre più cruciale la velocità di connessione tra dispositivi e sono più presenti l' Internet of Things e il Cloud Computing, le SDN si collocano come potenziale punto di unione tra le necessità delle nuove tecnologie e la forza di avere un networking volto all' open source. Con il paradigma della rete aperta e programmabile si possono introdurre innovazioni a livello Data Plane e Control Plane.

Lo scopo della tesi è stato quello di analizzare le prestazioni grezze e le potenzialità di DPDK, cercando di unire la velocità del forwarding di quest'ultimo alla versatilità, quindi programmabilità, del piano di controllo gestito da uno switch P4. L'analisi delle prestazioni di DPDK fa risaltare la potenza della tecnologia, che sfruttando il Kernel Bypassing, è capace di generare un traffico molto elevato anche disponendo di hardware a basse prestazioni. I test sono stati eseguiti trasmettendo pacchetti da 1500 byte, così da ottenere una quantità maggiore di dati trasmessi.

I risultati in condizioni reali riportano un throughput medio di 10 Gbit/s che per le reti odierne è sicuramente un risultato interessante, avendo la capacità di gestire in media qualche giga di dati. Le librerie di DPDK sono generalmente pensate per agire su di un flusso di dati suddiviso in pacchetti di piccola dimensione (64 byte). Negli usi applicativi moderni, come ad esempio scenari di streaming o flussi dati costanti, è difficile riuscire a segmentare il traffico in pacchetti di così piccola taglia. Un ulteriore limite di DPDK può essere trovato nell' usabilità di questo framework. Nella fase di test è stato utilizzato un tool in grado di generare una enorme quantità di pacchetti per valutare le performance di instradamento. Dal punto di vista dell' utilizzo, sarebbe utile usufruire di questa tecnologia in modo trasparente, ovvero poter delegare alla scheda di rete le operazioni quotidiane senza accorgersi del livello sottostante e rendere così compatibile DPDK con il livello applicativo, come ad esempio la navigazione su Internet o l'invio di file ad altri dispositivi.

P4 ha dimostrato grande potenzialità grazie alla sua riprogrammabilità e DPDK si è confermato un'ottima tecnologia per velocizzare il forwarding di pacchetti. Coniugando P4 e DPDK sarebbe possibile unire la versatilità che ha uno switch P4 con il suo approccio "top-down", alla velocità di instradamento dei dati che offre DPDK anche con un hardware mediocre. Un approccio molto interessante si è rivelato quello del progretto P4-OVS. In questa estensione di OVS, è possibile ottenere prestazioni elevate, in modo da rimpiazzare lo standard Kernel Datapath di cui dispone, con un Datapath completamente basato su DPDK [19]. Nella pratica l'uso di OpenVSwitch resta lo stesso a livello di networking, ma eredita i benefici del packet processing che porta P4 e del forwarding interno di pacchetti che è accelerato grazie al driver DPDK ad alte prestazioni.

Ringraziamenti

Ringrazio i relatori ed il team di ULISSe per avermi dato l'opportunità di lavorare su questo tema di ricerca.

Vorrei inoltre ringraziare e dedicare questa tesi alla mia famiglia e alle mie persone speciali.

Bibliografia

[1] *An Introduction to AWS Networking — Virtual Private Cloud*. URL: https://odsc.medium.com/an-introduction-to-aws-networking-virtual-private-cloud-1639a91c67c1 (visitato il 01/07/2022).

[2] *Behavioural Model (BMv2)*. original-date: 2015-01-26T21:43:23Z. 1 Lug. 2022. URL: https://github.com/p4lang/behavioral-model/blob/f16d0de3486aa7fb2e1fe554aac7d237cc1adc33/docs/performance.md (visitato il 01/07/2022).

[3] Pekka Enberg. *On Kernel-Bypass Networking and Programmable Packet Processing*. Medium. 13 Dic. 2018. URL: https://medium.com/@penberg/on-kernel-bypass-networking-and-programmable-packet-processing-799609b06898 (visitato il 01/07/2022).

[4] *ethtool(8) - Linux man page*. URL: https://linux.die.net/man/8/ethtool (visitato il 02/07/2022).

[5] *FD.io VPP: Shared Memory Packet Interface (memif) Library*. URL: https://docs.fd.io/vpp/17.10/libmemif_doc.html (visitato il 01/07/2022).

[6] *Github Issue*. URL: https://github.com/p4lang/behavioral-model/issues/986 (visitato il 02/07/2022).

[7] *Introducing the Data Plane Development Kit (DPDK) on Lenovo Servers*. URL: https://lenovopress.lenovo.com/lp0749-introducing-data-plane-development-kit-dpdk (visitato il 01/07/2022).

[8] *Introduction to Linux interfaces for virtual networking*. URL: https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking (visitato il 01/07/2022).

[9] *IOMMU (Input-Output Memory Management Unit)*. URL: https://www.peerspot.com/articles/iommu-input-output-memory-management-unit (visitato il 01/07/2022).

[10] *iPerf - Download iPerf3 and original iPerf pre-compiled binaries*. URL: https://iperf.fr/iperf-download.php (visitato il 02/07/2022).

[11] *Kernel Bypassing*. URL: https://www.researchgate.net/publication/339048803/figure/fig6/AS:855212877164546@1580910047429/Kernel-networking-vs-Kernel-bypassing.ppm (visitato il 01/07/2022).

[12] *Linux Drivers — Data Plane Development Kit 22.07.0-rc2 documentation*. URL: https://doc.dpdk.org/guides/linux_gsg/linux_drivers.html (visitato il 01/07/2022).

[13] Prasad Mukhedkar. *Mastering KVM Virtualization*. Packt, 2020.

[14] Wajahat Navid e Muhammad Nasir Mumtaz Bhutta. «Detection and mitigation of Denial of Service (DoS) attacks using performance aware Software Defined Networking (SDN)». In: *2017 International Conference on Information and Communication Technologies (ICICT)*. 2017 International Conference on Information and Communication Technologies (ICICT). Dic. 2017, pp. 47–57. DOI: 10.1109/ICICT.2017.8320164.

[15] Tomasz Osiński. *P4-OvS - Bringing the power of P4 to OvS!* original-date: 2020-05-05T17:15:38Z. 1 Giu. 2022. URL: https://github.com/osinstom/P4-OvS (visitato il 03/07/2022).

[16] *P4 – Language Consortium*. URL: https://p4.org/ (visitato il 01/07/2022).

[17] *P4 programming language specifications*. original-date: 2015-08-14T08:45:53Z. 27 Giu. 2022. URL: https://github.com/p4lang/p4-spec/blob/61b5860f07a90b810709843a0e5f5b7f81709eb1/p4-16/psa/psa.p4 (visitato il 01/07/2022).

[18] *P4 Tutorial*. original-date: 2015-09-30T19:17:28Z. 2 Lug. 2022. URL: https://github.com/p4lang/tutorials/blob/9b15344f0b90bd06eeb7b5bc7b4d520cc546a9ee/exercises/basic/README.md (visitato il 02/07/2022).

[19] *P4-OVS*. URL: https://www.dpdk.org/wp-content/uploads/sites/35/2016/10/Day02-Session12-CianFerriter-Userspace2016.pdf (visitato il 01/07/2022).

[20] *P4: Types and Parsers*. URL: https://cornell-pl.github.io/cs6114/lecture05.html (visitato il 01/07/2022).

[21] *P4@ELTE - T4P4S - a retargetable compiler for P4*. URL: http://p4.elte.hu/ (visitato il 01/07/2022).

[22] *P4PI*. original-date: 2021-06-28T20:39:39Z. 29 Giu. 2022. URL: https://github.com/p4lang/p4pi (visitato il 03/07/2022).

[23] *PCI passthrough via OVMF*. URL: https://wiki.archlinux.org/title/PCI_passthrough_via_OVMF (visitato il 01/07/2022).

[24] *QEMU*. URL: https://www.qemu.org/ (visitato il 02/07/2022).

[25] *Software Defined Networking*. URL: https://www.paessler.com/software-defined-networking (visitato il 01/07/2022).

[26] *Software Defined Networking (SDN) in Azure Stack HCI e Windows Server*. URL: https://docs.microsoft.com/it-it/azure-stack/hci/concepts/software-defined-networking (visitato il 01/07/2022).

[27] *The Packet Processing Graph — The Vector Packet Processor v22.06-0-g0d352a97c documentation*. URL: https://s3-docs.fd.io/vpp/22.06/aboutvpp/extensible.html# (visitato il 01/07/2022).

[28] *uBPF*. URL: https://github.com/iovisor/ubpf (visitato il 01/07/2022).

[29] *Universal TUN/TAP device driver — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/v5.8/networking/tuntap.html (visitato il 02/07/2022).

[30] *Using the MacVTap driver - IBM Documentation*. URL: https://www.ibm.com/docs/en/linux-on-systems?topic=choices-using-macvtap-driver (visitato il 02/07/2022).

[31] *vgropp/bwm-ng*. URL: https://github.com/vgropp/bwm-ng (visitato il 01/07/2022).

[32] *What Is OpenFlow?* URL: https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/what-is-openflow/ (visitato il 01/07/2022).

[33] *What is the Vector Packet Processor (VPP) — The Vector Packet Processor v22.06-0-g0d352a97c documentation*. URL: https://s3-docs.fd.io/vpp/22.06/ (visitato il 01/07/2022).

[34] Keith Wiles. *Pktgen - Traffic Generator powered by DPDK*. original-date: 2019-10-02T16:11:34Z. 10 Giu. 2022. URL: https://github.com/pktgen/Pktgen-DPDK (visitato il 01/07/2022).

[35] *Wireshark · Go Deep.* URL: https://www.wireshark.org/ (visitato il 02/07/2022).

Elenco delle figure