Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Magistrale in Informatica

Tesi di Laurea

DPI per traffico HTTP implementato in FastFlow

Candidato
Francesco Baldini

Relatori
Prof. Marco Danelutto
Prof. Luca Deri

Controrelatore
Prof. Maurizio A. Bonuccelli

Anno Accademico 2011/2012

Indice generale

Capitolo 1

Introduzione

Nelle reti di calcolatori, il traffico di rete che transita in una LAN (Local Area Network) viene solitamente controllato e filtrato da dispositivi noti come firewall.
Questi sono una struttura hardware e/o software di difesa, che collega una rete ad un'altra, solitamente una LAN privata (rete interna) ad Internet (rete esterna).
Il firewall ha come scopo quello di garantire una difesa perimetrale della rete, tramite l'analisi ed il monitoraggio del traffico, consentendo agli amministratori di rete di definire le politiche di sicurezza necessarie per il controllo e la gestione dei flussi di traffico passanti.

I firewall [ROS13] possono essere classificati in varie categorie sulla base delle informazioni prese in considerazione per effettuare l'analisi del traffico.
Si parla di “packet filter” firewall, quando le decisioni sono principalmente adottate andando ad analizzare i valori delle intestazioni protocollari (*header*), pacchetto per pacchetto, l'uno indipendentemente dall'altro.
Si parla di “stateful filter” firewall quando l'analisi è ancora condotta andando a leggere le informazioni presenti nelle intestazioni dei pacchetti, inoltre si associa ai pacchetti anche un concetto di “stato del flusso”.
In questo frangente le decisioni sono prese sia in base ai valori letti dai singoli pacchetti, sia in base allo stato del flusso precedentemente memorizzato.

Per ottenere un più elevato livello di sicurezza, le politiche di filtraggio possono essere basate anche sulle informazioni contenute nei dati applicativi, oltre che sull'analisi delle intestazioni protocollari.
Il Deep Packet Inspection [OU13] è una tecnica di analisi dei pacchetti che esamina i dati trasportati al loro intero (*payload*), alla ricerca di specifici pattern. In questo modo è possibile classificare il traffico che difficilmente sarebbe identificabile con semplici controlli basati solo sui parametri disponibili nelle intestazioni protocollari dei livelli di astrazione della pila TCP/IP, in quanto l’associazione fra “known port” e protocollo applicativo non viene più rispettata negli ultimi anni.
Ciò che deve essere analizzato e filtrato con il DPI necessita di ricerche di pattern ad hoc, protocollo per protocollo, per questo motivo risulta essere particolarmente oneroso in termini di utilizzo di risorse computazionali.
Si parla di “application layer” firewall, quando si integra l'analisi DPI con il concetto di “stateful filter”.

I dispositivi di rete quali router, bridge e layer-3 switch offrono le funzionalità di firewalling che sono generalmente realizzate mediante due approcci alternativi, hardware-based e software-based (talvolta possono essere anche combinate assieme).
Per quanto riguarda le soluzioni hardware-based, queste sono fondamentalmente basate sull'impiego di hardware specializzato per i fini di analisi. Vi sono soluzioni che offrono funzionalità ridotte, che si limitano al packet filtering, quindi ad un'analisi principalmente basata sulle intestazioni protocollari e vi sono soluzioni hardware-based anche più complesse, che sono in grado di effettuare analisi più approfondite, come il DPI a livello applicativo [FSE13, ACF05, WTD06, YL05].

Per quanto riguarda l'approccio software, esso consiste nella realizzazione delle funzionalità di firewalling in uno o più sistemi software, che verranno fatti girare sopra il sistema operativo del router oppure sopra il sistema operativo di macchine con CPU x86, specificatamente dedicate a scopi di filtraggio.
L'approccio basato sul firewalling software ha sicuramente dei vantaggi rispetto all'approccio hardware-based, in particolar modo legati ad un minor costo economico e ad una maggiore versatilità in relazione alla possibile introduzione di nuovi protocolli da dover gestire.
L'alta banda del traffico su rete, richiede un sistema di filtraggio che garantisca un'elevata banda di elaborazione, pertanto l'utilizzo del calcolo parallelo, in linea di principio, permetterebbe di poter aumentare le prestazioni del calcolo puramente sequenziale, raggiungendo prestazioni paragonabili a quelle hardware-based.
Negli ultimi anni la direzione di sviluppo architetturale è passata da macchine a singola CPU a macchine multi-processore/multi-core. Questo fatto ha un enorme impatto sulle applicazioni, in quanto queste dovranno essere sviluppate sfruttando il parallelismo derivante da tali architetture. In relazione a questo trend, si sono recentemente sviluppati vari framework di programmazione parallela e strumenti per lo sviluppo di applicazioni parallele.
La realizzazione di un'applicazione che sfrutta il calcolo parallelo, con lo scopo di ottenere le migliori prestazioni possibili, deve essere impostata con una progettazione ex-novo, cercando la migliore decomposizione del problema in attività concorrenti/parallele, definendo tutte le singole attività nello specifico. Partire da un'applicazione già pensata e progettata in sequenziale, semplicemente innestando la logica applicativa dell'analisi all'interno delle strutture predefinite offerte dai framework di programmazione parallela, non risulta essere l'approccio corretto nel caso in cui si vogliano massimizzare le prestazioni. Ad esempio uno dei motivi più comuni è determinato dal fatto che è possibile dover introdurre dei meccanismi di lock laddove se ne presenti la necessità per garantire la mutua esclusione sulle sezioni critiche emerse. Ciò comporta una sequenzializzazione delle operazioni protette dai meccanismi di lock, con un conseguente freno al parallelismo fisico fra i vari moduli paralleli.
Tuttavia la maggior parte delle soluzioni esistenti per il filtraggio a livello software non

sfrutta il parallelismo derivante dalle architetture multi-core, pertanto l'analisi del traffico viene condotta in modo prettamente sequenziale [NOE13, ALP13, AN13]. C'è da aggiungere che vi sono anche alcune applicazioni che forniscono un metodo di analisi che sfrutta il calcolo parallelo [YX11, WCH09], tuttavia soffrono delle criticità introdotte dagli overhead di comunicazione e di sincronizzazione fra i vari moduli paralleli, nonché di bilanciamento di carico fra le varie unità di esecuzione parallele.

L'obiettivo del lavoro di tesi è stato quello di realizzare un'applicazione di firewalling che sfruttasse il parallelismo offerto dalle architetture multi-core con CPU x86, al fine di realizzare un sistema in grado di poter garantire un'elevata banda di elaborazione. L'applicazione sviluppata effettua un'analisi DPI per il protocollo applicativo HTTP, relativamente al traffico su reti Ethernet. Per la realizzazione del supporto per lo sfruttamento del parallelismo derivante dalle architetture multi-core è stato impiegato FastFlow [FAS13, ADT12].
Questo è un framework di programmazione parallela efficiente, per sistemi multi-core, sviluppato dall'università di Pisa e di Torino. Il framework offre dei meccanismi di comunicazione con bassa latenza, particolarmente adatto a computazioni di grana molto fine, come quelle che vengono eseguite su stream di pacchetti, su reti di calcolatori a bande molto elevate. Dato che FastFlow è stato scritto in C++ [STR08], anche l'implementazione dell'applicazione per il filtraggio, è stata realizzata nel medesimo linguaggio di programmazione.

Al termine dell'attività di tesi è stata realizzata l'applicazione di firewalling con un'implementazione “lockless”, cioè che non utilizza al suo interno nessun meccanismo di lock. Il lavoro svolto ha dimostrato di aver raggiunto pienamente gli obiettivi prefissati, in quanto è risultato essere in grado di gestire le alte bande di traffico al variare del numero di interfacce di rete in gestione. Questo è stato documentato mediante il calcolo di vari indici prestazionali (che verranno presentati nei capitoli successivi).

La presente tesi è strutturata nel seguente modo:

Nel capitolo 2 sono descritti gli skeleton algoritmici, il modello teorico delle prestazioni, nonché gli strumenti e tecnologie utilizzate durante lo svolgimento dell'attività di tesi, quali: il framework di programmazione parallela FastFlow, la libreria PCAP [TL13] utilizzata per l'interfacciamento con le schede di rete ed infine il protocollo HTTP.
Nel capitolo 3 è descritto il progetto architetturale dell'applicazione, con una visione ad alto livello dei moduli paralleli e sequenziali con cui è strutturato l'applicativo, astraendo sui dettagli tecnico/implementativi.
Nel capitolo 4 è descritta la realizzazione del progetto architetturale spiegato nel precedente capitolo, mettendo in evidenza le scelte implementative adottate, i relativi aspetti tecnici ed il modello teorico delle prestazioni della struttura parallela.

Nel capitolo 5 è presentato lo skeleton “multi-emitter farm”, un nuovo pattern parallelo che si presenta come una variante dello skeleton farm. Se ne descrive la struttura, se ne delineano le relative caratteristiche e si costruisce il modello teorico delle prestazioni per il nuovo pattern.
Nel capitolo 6 sono presentati i risultati sperimentali, in cui si forniscono i grafici relativi agli indici di prestazione dell'applicazione e si comparano i risultati attesi in fase preliminare durante la costruzione del modello teorico delle prestazioni, con i dati rilevati dopo la sperimentazione.
Nel capitolo 7 sono descritte le conclusioni sull'attività di tesi, in particolare ciò che è stato appreso al termine, gli obiettivi raggiunti con la progettazione e realizzazione dell'applicazione, le future possibilità di estensione dell'applicativo.

Capitolo 2

Tecnologie utilizzate

In questo capitolo sono spiegate le tecnologie utilizzate durante l'attività di tesi, maggiore spazio è stato dato alla spiegazione della libreria FastFlow, su cui si basa fondamentalmente la parte parallela del software sviluppato, viene successivamente illustrata la libreria PCAP, su cui si basa la parte dell'applicazione che si interfaccia con le schede di rete.

2.1 Skeleton algoritmici

Lo skeleton [COL91] è una struttura di computazione parallela, indipendente dalla funzione che si vuole calcolare, è una forma di parallelismo che calcola una certa funzione. In altre parole, lo skeleton è lo scheletro di un algoritmo parallelo, lo schema generico di una computazione parallela istanziabile con qualunque funzione specifica.

Lo skeleton algoritmico può anche essere definito come una funzione di ordine superiore che definisce un ben preciso modello di utilizzo del parallelismo, la quale ha come parametri funzionali la logica dell'applicazione stessa.

La definizione di skeleton algoritmico si è leggermente evoluta nel tempo. Oggigiorno la una nuova definizione [DAN11] comunemente accettata è: uno skeleton algoritmico è un'astrazione di programmazione, parametrica, riusabile e portabile che modella un pattern parallelo noto, comune ed efficiente.

Gli aspetti cruciali di questa definizione sono i seguenti:

- lo skeleton deve modellare un pattern parallelo ben conosciuto e comune, pertanto non si è interessati in pattern particolari che non sono comunemente impiegati nelle applicazioni. Questo non rappresenta una limitazione sostanziale, in quanto la maggior parte delle computazioni parallele che vogliamo descrivere, possono essere modellate utilizzando un piccolo numero di pattern paralleli;
- la portabilità, per cui deve essere possibile fornire un'implementazione efficiente del pattern sulle varie architetture;
- la riusabilità, significa che tale pattern può essere utilizzato in vari contesti/applicazioni differenti, senza dover modificare il pattern;

- parametricità, lo skeleton è parametrico perché può specializzare il proprio comportamento in funzione dei suoi parametri. I parametri possono essere funzionali, pertanto può prendere altre funzioni come parametri. Inoltre una delle più importanti caratteristiche degli skeleton è che possono essere annidati fra loro. Pertanto uno skeleton può prendere come parametro un altro skeleton, ottenendo una composizione fra questi;
- efficienza, significa che lo skeleton modella quei pattern paralleli che hanno un'implementazione efficiente sulle varie architetture.

La programmazione mediante skeleton è detta “programmazione parallela strutturata”. Scrivere un'applicazione parallela usando un framework specifico, consiste nell'istanziare uno skeleton adatto alla risoluzione del particolare problema da risolvere e passare allo skeleton la funzione che deve eseguire ogni modulo dello stesso.

Gli skeleton offrono dei notevoli vantaggi rispetto ai tipici approcci alla programmazione parallela in cui si fa uso delle note librerie Posix [PTP13] o MPI [MPI13].

L'approccio a skeleton permette all'utente di svincolarsi dai problemi relativi alla creazione e avvio dei moduli paralleli nonché all'orchestrazione della comunicazione fra questi, offrendo una visione più ad alto livello rispetto a quella fornita dalle librerie Posix o MPI. Perciò in fase di debugging, il programmatore deve solo preoccuparsi della correttezza della logica funzionale dell'applicativo.

Inoltre siamo svincolati dalle problematiche relative alla portabilità dell'applicazione parallela, al suo deployment ed esecuzione in quanto sono completamente a carico del framework di programmazione parallela.

Gli skeleton possono essere suddivisi in classi:

- *data parallel*: l'elemento in input che si vuole processare, viene partizionato in vari parti ed ogni computazione relativa ad una sottoparte viene eseguita in parallelo.

  I più comuni skeleton data parallel sono la map e la reduce;
- *stream parallel*: il parallelismo è ottenuto grazie alla computazione in parallelo su differenti elementi presenti nello stream in ingresso.

  I più comuni skeleton stream parallel sono la pipeline e la farm, che fanno parte delle successive spiegazioni data la loro presenza in FastFlow.

Nei vari framework di programmazione parallela gli skeleton sono principalmente offerti al programmatore in due modi alternativi:

- costrutti primitivi: gli skeleton vengono definiti utilizzando i costrutti primitivi del linguaggio, pertanto essi fanno parte delle “key word” del linguaggio stesso. In questo caso gli skeleton sono forniti tramite costrutti primitivi, è quindi

necessario definire un nuovo linguaggio di programmazione, in cui gli skeleton fanno parte del linguaggio stesso;

- libreria: gli skeleton vengono utilizzati tramite invocazioni di interfacce di una libreria che implementa gli skeleton. Pertanto scelto un determinato linguaggio di programmazione, gli skeleton sono implementati sopra tale linguaggio, sfruttando anche le librerie disponibili. Il programmatore può definire ed istanziare gli skeleton tramite le tipiche chiamate di libreria del linguaggio scelto.

2.1.1 Pipeline

La pipeline fa parte degli skeleton che operano su stream, cioè che sfruttano il parallelismo su computazioni relative ad elementi indipendenti che sono presenti nello stream in ingresso.

La pipeline è tipicamente utilizzata per modellare computazioni espresse in stadi (o stage), in cui ognuno di questi verrà eseguito in parallelo.

Consideriamo n funzioni

$$f_1 : D_0 \rightarrow D_1, f_2 : D_1 \rightarrow D_2, \ldots, f_{(n-1)} : D_{(n-2)} \rightarrow D_{(n-1)}, f_{(n)} : D_{(n-1)} \rightarrow D_{(n)} \quad \text{e}$$

siano $x_1, x_2, \ldots, x_m$ gli elementi dello stream di ingresso dove $x_i \in D_0 \; per \; i \in [1, m]$

La pipeline è una funzione $P : D_0 \rightarrow Dn$ definita come:

$$P(x_i) = f_n(f_{(n-1)}(\ldots(f_1(x_i)))) = y_i \quad \text{per} \quad i \in [1, m]$$

$$\text{dove} \quad y_i \in D_n \; per \; i \in [1, m]$$

e lo stream in uscita risulta:

$$y_1, y_2, \ldots, y_n$$

In sostanza la pipeline non è altro che una composizione di funzioni, costituita da n stadi, una funzione per ciascuno stadio. In cui ciascuno stadio prende in input il dato dallo stadio precedente calcola la sua funzione e lo invia allo stadio successivo, come si mostra nella figura che segue.

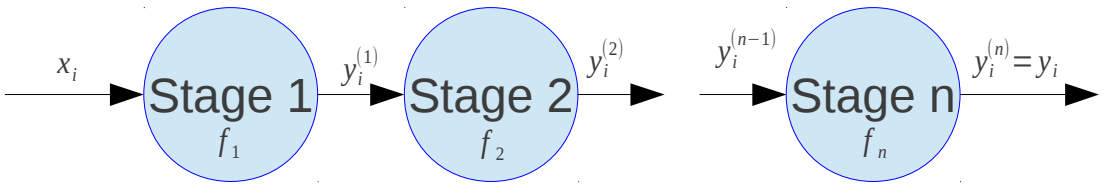


Figura 2.1 Lo skeleton pipeline

La Fig. 2.1 illustra la struttura dello skeleton pipeline con n stage, dove x_i con $i \in [1, m]$ presente sull'ingresso rappresenta uno degli elementi facenti parte dello stream in input $x_{1,} x_{2,} \ldots, x_m$. Dove lo stadio k-esimo della pipeline raffigurato a forma di cerchio applica la relativa funzione f_k con $k \in [1, n]$, precisamente calcolando $f_k(y_i^{(k-1)}) = y_i^{(k)}$ dove $y_i^{(1)} = f_1(x_i)$, ed essendo $y_i^{(k-1)} = f_{(k-1)}(\ldots f_1(x_i))$, $y_i^{(k)}$ può essere riscritto come $y_i^{(k)} = f_k(f_{(k-1)}(\ldots f_1(x_i)))$.

2.1.2 Farm

La farm fa parte degli skeleton che operano su stream, e necessita solo della definizione della funzione f che si vuole applicare agli elementi dello stream in ingresso.

Sia quindi:

$f : D \rightarrow C$ una funzione pura e siano $x_{1,} x_{2,} \ldots, x_m$ gli elementi dello stream di ingresso dove $x_i \in D \; per \; i \in [1, m]$

La farm applica f ad ogni elemento x_i e lo invia sul proprio stream di uscita, in cui ciascun elemento $f(x_i)$ viene calcolato all'interno di ogni worker, in modo indipendente ed in parallelo rispetto agli altri elementi.

$$f(x_i) = y_i \quad \text{per} \quad i \in [1, m]$$

$$\text{dove} \quad y_i \in C \; per \; i \in [1, m]$$

Lo stream in uscita risulta essere quindi così definito:

$$f(x_1), f(x_2), \ldots, f(x_m)$$

Si può infatti notare come a regime questa struttura parallela riesce a computare n $f(x_i)$ negli n worker diversi.

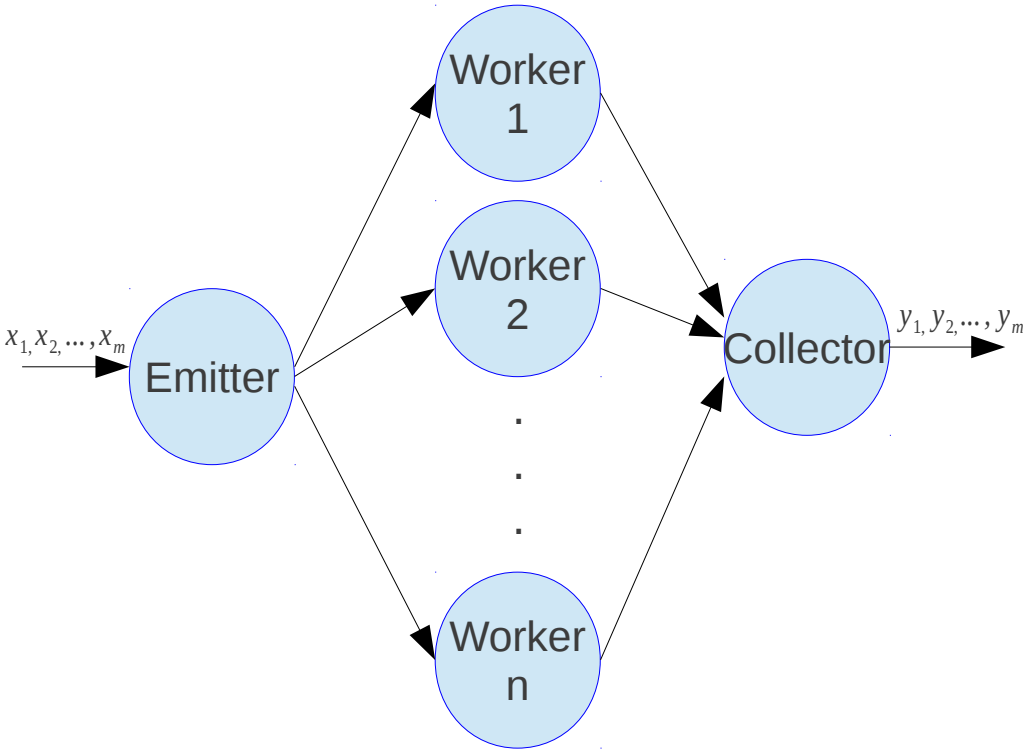


Figura 2.2 Lo skeleton farm

La Fig. 2.2 illustra una delle possibili implementazioni dello skeleton farm, in cui:

- L'emitter ha il compito di leggere in ingresso gli elementi $x_1, x_2, \ldots, x_m$ e di schedularli verso i worker, con la possibilità dell'uso di varie politiche per una corretta gestione del bilanciamento di carico dei vari worker;
- Il generico worker ha il compito di applicare la funzione f all'elemento x_i che gli viene passato dall'emitter, cioè esegue $f(x_i) = y_i$ ed invia y_i al collector;
- Il collector ha il compito di acquisire i risultati calcolati dai vari worker e di inviarli sul proprio stream di uscita.

L'emitter può utilizzare varie strategie per schedulare i task verso i worker, fra cui:

- “on demand”, in cui il worker quando ha terminato la propria comunicazione segnala all'emettitore di essere disponibile ad una nuova computazione, quindi

l'emitter non fa altro che inviare nuovi elementi x_i ai worker liberi che ne fanno richiesta;

- "round robin", in cui l'emitter invia gli elementi x_i in modo ciclico a partire da 1 sino a N_{worker}, più nello specifico, se l'emitter deve inviare l'elemento x_i, esso sarà inviato al worker con ID = $(i \bmod N_{worker})$.

La definizione delle strategie di schedulazione è molto importante in quanto in base a questa viene determinato il bilanciamento di carico.

Quando i worker hanno una bassa varianza del tempo di servizio, una strategia "round robin" è decisamente una buona scelta, in quanto mantiene il bilanciamento di carico.

Quando invece si ha una grande varianza del tempo di servizio dei worker, la strategia "on demand" è preferibile, ovviamente tenendo in considerazione il fatto che la strategia "on demand" aggiunge dei costi di comunicazione ulteriori rispetto a quella "round robin", in quanto oltre al canale di comunicazione unidirezionale fra l'emitter e tutti i worker è necessario disporre di canali di comunicazione anche nell'altra direzione, per comunicare la disponibilità del worker a ricevere un nuovo input.

2.1.3 Modello teorico delle prestazioni

Le prestazioni di un'applicazione parallela, sono sostanzialmente determinate dai seguenti aspetti:

- dallo skeleton algoritmico che stiamo utilizzando;
- dal modo con cui il pattern parallelo è stato realizzato (l'implementazione dello skeleton stesso);
- dall'architettura che stiamo utilizzando per l'esecuzione dell'applicativo.

Saremo quindi interessati alla definizione di un insieme di funzioni $S=\{P_1, P_2, \ldots, P_n\}$, ciascuna che modella un indicatore delle prestazioni diverso, che indicheremo più avanti, dove $\forall P_i \in S$ abbiamo che $P_i: D_1 x D_2 x \ldots x D_m \rightarrow C$, dove D_j sono i domini dei parametri della generica funzione P_i, mentre C è il codominio della funzione P_i, che solitamente corrisponde ai reali.

Una volta definite tali funzioni, possono essere utilizzate fondamentalmente per i seguenti tre scopi:

- predizione: conoscere che tipo di prestazioni possiamo ottenere quando si utilizza un determinato skeleton, ancora prima di scrivere il codice sorgente;
- comparazione: comparare i vari pattern paralleli adatti per risoluzione di un determinato problema, stabilendo quale sia la migliore soluzione, oppure comparare differenti implementazioni dello stesso pattern parallelo;

- valutazione: determinare quanto overhead è aggiunto alla nostra implementazione e quindi quanto la nostra prestazione si discosta da quella teorica.

In generale vi sono due principali e distinti tipi di misure da tenere in considerazione:

- Tempo assoluto relativo all'esecuzione di una certa applicazione (o di una parte di essa);
- Tempo relativo, il *throughput* dell'applicazione, cioè la velocità con cui vengono rilasciati i vari risultati.

2.1.3.1 Metriche di base

Relativamente alla categoria del tempo assoluto dell'esecuzione di una certa applicazione abbiamo:

- latenza (L): Il tempo che intercorre fra il momento che una determinata attività riceve in ingresso un elemento da processare e il momento in cui tale elemento processato viene rilasciato sullo stream d'uscita;
- tempo di completamento (T_C): rappresenta la latenza dell'intera applicazione, su un determinato insieme di elementi da processare (dataset). Non è altro quindi che il tempo che intercorre fra l'inizio dell'applicazione e la sua terminazione.

Per quanto riguarda la categoria del throughput dell'applicazione si ha:

- tempo di servizio (T_S): rappresenta il tempo che intercorre fra due rilasci di due successivi elementi sullo stream di uscita (oppure fra due letture dello stream di ingresso);
- banda (B): è data dall'inverso del tempo di servizio: $\frac{1}{T_S}$.

Solitamente se si è interessati alla latenza ed il tempo di completamento è perché si vuole completare un'attività il prima possibile, mentre se siamo interessati alla banda ed al tempo di servizio, è perché si vuole fornire i risultati con la maggior frequenza possibile, senza necessariamente minimizzare la latenza.

Ad esempio, è infatti possibile che un'applicazione possa avere un basso tempo di servizio con una corrispondente alta banda rispetto alla versione sequenziale, pur avendo una peggiore latenza rispetto all'applicazione sequenziale.

2.1.3.2 Indici delle prestazioni

Una volta definite metriche di base, possono essere definite delle metriche prestazionali derivate [DAN11, VAN12], che ci forniscono delle utili informazioni riguardo la bontà dell'applicazione parallela, quali:

- Speedup: rappresenta la bontà dell'algoritmo parallelo rispetto alla relativa versione sequenziale per la risoluzione del medesimo problema. E' definito come il rapporto fra il miglior tempo di esecuzione sequenziale ed il tempo di esecuzione parallelo sfruttando un grado di parallelismo fissato ad n.

$$Speedup(n) = \frac{T_{seq}}{T_{par}(n)}$$

  - T_{seq} è il miglior tempo di esecuzione sequenziale;
  - $T_{par}(n)$ è tempo di esecuzione parallelo, con n il grado di parallelismo utilizzato.

  Il miglior algoritmo parallelo possibile presenta uno speedup lineare, che è rappresentato dalla funzione identità $Speedup(n) = n$.

  Nelle pratica lo speedup lineare è difficilmente raggiungibile in quanto nella realizzazione parallela oltre al codice funzionale è introdotto del codice relativo all'orchestrazione delle comunicazioni e quindi relativo al passaggio dei dati fra le varie unità di esecuzione parallele. Ciò rappresenta un overhead aggiuntivo rispetto meramente al codice dell'algoritmo sequenziale.

  In casi rari è tuttavia possibile ottenere uno speedup superlineare in cui $Speedup(n) > n$. Questo risultato può essere raggiunto ad esempio grazie allo sfruttamento della cache, in cui i dati relativi al calcolo possono essere totalmente contenuti nelle cache dei vari processori, mentre questi non possono essere contenuti nella versione sequenziale. Quindi il vantaggio ottenuto può risultare maggiore dell'overhead necessario ad avviare ed orchestrare le varie attività parallele.

- Scalabilità: rappresenta la capacità dell'applicazione parallela di poter aumentare le proprie prestazioni all'aumentare delle risorse disponibili (ad esempio al raddoppio dei processori a disposizione, l'applicazione parallela dovrebbe essere in grado dimezzare il proprio tempo di completamento).

  E' definita come il rapporto fra il tempo necessario alla terminazione della computazione parallela utilizzando un grado di parallelismo eguale ad uno, ed il tempo tempo necessario alla computazione parallela con grado di parallelismo eguale ad n.

$$Scalability(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

Dove $T_{par}(i)$ rappresenta il tempo di esecuzione dell'applicazione parallela con grado di parallelismo i.

La presenza di uno o vari colli di bottiglia nell'applicazione parallela potrebbero vanificare l'aumento di risorse messe a disposizione all'applicazione, evidenziando appunto un basso indice di scalabilità.

Generalmente uno scarso indice di scalabilità, porta conseguentemente anche ad un cattivo indice di speedup, in quanto se non si ha scalabilità, significa che non vengono sfruttate a pieno le risorse computazionali aggiuntive, e non sfruttando questo parallelismo si perde anche in confronto al tempo di esecuzione sequenziale.

Questi due indici derivati possono essere calcolati a partire dalle misure base, ed hanno senso solo se vengono utilizzate in modo consistente. Infatti sia il tempo di servizio, la latenza che il tempo di completamento possono essere utilizzati nelle precedenti formule, ma sempre purché sia mantenuta la consistenza nell'uso.

- Efficienza: è rappresentata da un valore compreso fra zero ed uno, essa stima la bontà dell'implementazione dell'applicazione parallela, nonché l'abilità di quest'ultima di fare un buon uso delle risorse che le sono state messe a disposizione.

  L'efficienza è definita come il rapporto fra il tempo di esecuzione ideale e quello effettivo.

$$Efficiency(n) = \frac{T_{id}(n)}{T_{par}(n)}$$

  - $T_{id}(n)$ il tempo di esecuzione ideale, con grado di parallelismo n.
  - $T_{par}(n)$ è tempo di esecuzione parallelo, con grado di parallelismo n.

  Dato che il tempo ideale di esecuzione può essere espresso come il tempo dell'esecuzione sequenziale diviso il grado di parallelismo utilizzato dall'applicazione,

$$T_{id}(n) = \frac{T_{seq}}{n}$$

dove T_{seq} è il tempo impiegato dall'esecuzione sequenziale, l'efficienza può essere quindi riscritta in altri termini:

$$Efficiency(n) = \frac{T_{id}(n)}{T_{par}(n)} = \frac{T_{seq}}{n\,T_{par}(n)}$$

e dato che $Speedup(n) = \frac{T_{seq}}{T_{par}(n)}$, può essere ulteriormente riscritta nei seguenti termini:

$$Efficiency(n) = \frac{Speedup(n)}{n}$$

Abbiamo ottenuto che l'efficienza è in funzione dello speedup, pertanto è possibile in primo luogo calcolare lo speedup e successivamente calcolare l'efficienza, senza effettuare due procedimenti di calcolo indipendenti come si poteva pensare nell'espressione iniziale della formula dell'efficienza.

Avendo a disposizione due implementazioni parallele diverse di un determinato problema, è possibile calcolare l'indice di efficienza di entrambe e compararle, al fine di vedere quale delle due è considerata “migliore”. In questo caso “migliore” è un risultato offerto dalla comparazione fra indici di efficienza, ciò non vuol dire che sia migliore in tutte le sue caratteristiche, infatti è possibile che la soluzione con migliore efficienza abbia un tempo di completamento maggiore rispetto all'altra soluzione.

Finora si è parlato di concetti generali, che sono fondamentalmente dei modelli astratti, che rappresentano la massima performance teorica raggiungibile e che dipendono solo dalle caratteristiche del pattern parallelo.

Questi indici sono teorici e non tengono in considerazione dei dettagli implementativi delle applicazioni, né tanto meno delle architetture su cui vengono eseguite.

Esistono anche dei modelli che tengono in considerazione dei differenti meccanismi con cui sono implementate le varie parti che costituiscono l'applicazione parallela, tenendo quindi in considerazione anche altri fattori, come ad esempio:

- Template performance model: il modello tiene in considerazione sia delle caratteristiche dello skeleton, che del modo con cui è realizzato lo skeleton stesso.
- Architecture performance model: il modello tiene in considerazione le caratteristiche dello skeleton, il modo con cui è realizzato lo skeleton stesso, nonché i differenti meccanismi dipendenti dall'architettura, come i vari tipi di comunicazione fra i moduli paralleli.

Ad esempio, tutti gli overhead di comunicazione sono generalmente modellati in secondo la seguente espressione:

$$T_{com}(l) = T_{setup} + \frac{l}{B}$$

dove T_{setup} è il tempo che si paga all'avvio della procedura di invio del dato, l la lunghezza del messaggio e B è la banda del canale di comunicazione.

Consideriamo i medesimi meccanismi, facendo riferimento in questo caso specifico alle architetture multi-core, con comunicazione implementata in memoria. Nel caso di comunicazioni singolo produttore singolo consumatore, abbiamo un thread che scrive i messaggi in memoria ed un altro thread che li legge. Il produttore ed il consumatore si devono sincronizzare in modo tale che i consumatori non leggano da canali "vuoti", ed i produttori non scrivano su canali "pieni". Pertanto $T_{com}(l)$ può essere modellata con la seguente espressione:

$$T_{com}(l) = T_{sync}$$

In generale l'espressione con cui è definita T_{sync}, è determinata da vari fattori, come la lunghezza del messaggio ed il numero di consumatori e produttori coinvolti.

Nei successivi capitoli durante l'analisi e lo sviluppo dei modelli delle performance si farà riferimento ai modelli a "template performance model", facendo emergere anche i costi di comunicazione, senza però scendere nei dettagli architetturali riguardo l'implementazione della comunicazione stessa.

2.1.4 FastFlow

FastFlow [ADT12, ADK11a, ADK12, FAS13] è un framework di programmazione parallela strutturata , per architetture multi-core a memoria condivisa, strutturato con vari livelli di astrazione software. Fornisce al programmatore gli skeleton algoritmici stream parallel, supportando lo sviluppo di applicazioni parallele di grana molto fine. Questo framework è implementato in C++ [STR08] e mantenuto dal Dipartimento di Informatica dell'università di Pisa e di Torino.

Il punto principale di forza di FastFlow è la buonissima efficienza quando siamo di fronte a computazioni di grana molto fine, che deriva dal basso overhead di comunicazione fornito dal livello più basso di FastFlow.

Inoltre FastFlow offre l'opportunità di implementare le computazioni data parallel attraverso gli skeleton algoritmici stream parallel offerti agli alti livelli della libreria.

FastFlow si basa su meccanismi di sincronizzazione lock-free. Per spiegare il termine, si deve prima introdurre il concetto di non-blocking. Un algoritmo è definito non-blocking, se è assicurato che i thread che sono in competizione per una risorsa condivisa, non

posticiperanno indefinitamente la loro esecuzione a causa della mutua esclusione. Un algoritmo non-blocking è definito lock-free, se è garantito il progresso dell'intero sistema.

Il tradizionale approccio alla programmazione multi-threaded, si basa sull'utilizzo di meccanismi lock per sincronizzare gli accessi alle strutture dati condivise, utilizzando ad esempio mutex, semafori, monitor, ecc.. Se un thread tenta di acquisire un lock posseduto già da un altro thread, il thread che prova l'acquisizione viene sospeso finché il lock non sarà rilasciato. Bloccare un thread non è sempre un'azione desiderabile, in quanto mentre un thread è bloccato non può effettuare nessuna operazione, vengono introdotti overhead relativi alle commutazioni di contesto, che risultano pesanti per le computazioni di grana fine, inoltre talune interazioni fra thread potrebbero portare a fenomeni di deadlock.

Ad esempio nella libreria POSIX, le operazioni di sincronizzazione come la lettura e scrittura ad esempio su code condivise singolo produttore singolo consumatore, vengono realizzate tramite l'impiego dei mutex e delle condition, che risultano meccanismi pesanti per le computazioni di grana fine.

FastFlow invece mette a disposizione dei meccanismi di comunicazione molto più leggeri di quelli offerti da POSIX. Ad esempio le scritture e le letture dalle code FIFO singolo produttore singolo consumatore, non richiedono l'utilizzo di alcun meccanismo di lock, in quanto nel caso in cui si cerchi di leggere da una coda vuota, il thread che effettua l'operazione, riottiene immediatamente il flusso di controllo, con l'esisto del fallimento lettura, senza che tale thread venga in alcun modo sospeso.

La libreria è realizzata utilizzando l'approccio basato su template, in cui le attività interne di uno skeleton sono realizzate da un insieme di thread.
Come definito in [DAN11], più formalmente un “process” template è una rete parametrica di attività concorrenti, definita da un grafo $G=(N,A)$, dove gli $n_i \in N$, rappresentano le attività concorrenti, e A è un insieme di coppie $\{(h,k) \in A\}$ che denotano la comunicazione fra il nodo $n_h \in N$ e $n_k \in N$.

FastFlow è strutturato con vari livelli di astrazione software, per rispondere a vari tipi di esigenze dei programmatori:

- Run-time Support: offre ai più alti livelli, i canali di comunicazione singolo produttore singolo consumatore (code SPSC), canali asincroni senza meccanismi di lock e con operazioni di lettura e scrittura non bloccanti, basate sull'algoritmo di Lamport [LAM83].

  Utilizzando FastFlow a questo livello è possibile costruire reti di comunicazione (a basso livello), in modo del tutto simile a quando si programma usando i thread POSIX. A questo livello è possibile far comunicare quindi i vari thread attraverso le code SPSC, anziché utilizzando i meccanismi di sincronizzazione POSIX che hanno delle latenze nettamente superiori a quelle di FastFlow.

- Low-level programming: sfruttando il livello di astrazione più basso sopra delineato, si può notare come questo livello di astrazione fornisca le code per i tipi di comunicazioni uno a molti, molti a uno e molti a molti.
  - SPSC: singolo produttore – singolo consumatore;
  - MPSC: multi produttore – singolo consumatore;
  - SPMC: singolo produttore – multi consumatore;
  - MPMC: multi produttore – multi consumatore.

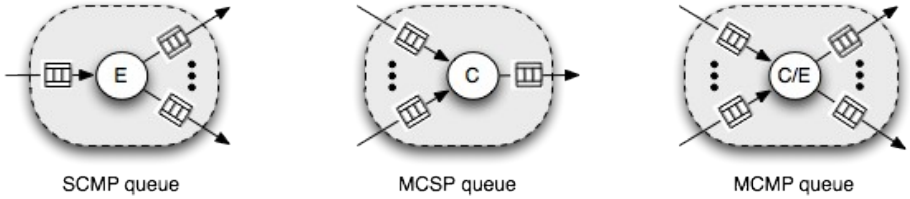


Figura 2.3 Tipi di code FastFlow offerte dal “low-level programming”

In linea generale le comunicazioni 1:N, N:1, N:M possono essere realizzate in più modi, usando ad esempio meccanismi di lock che incapsulano le scritture sulle singole code SPSC (nel caso N:1), e meccanismi di lock che incapsulano le letture sulle singole code SPSC (nel caso 1:N), pagando ovviamente un overhead dovuto alla sincronizzazione diretta per l'accesso alla sezione critica.

Utilizzando invece un approccio lock-free si devono almeno utilizzare le operazioni atomiche, che sono utilizzate per forzare la corretta serializzazione degli aggiornamenti dei produttori e consumatori alla fine della coda.

In FastFlow le comunicazioni 1:N, N:1, N:M sono realizzate tramite un thread arbitro, senza l'impiego di meccanismi di lock. Infatti nel caso del singolo produttore – multi consumatore (SPMC), sarà un unico thread produttore chiamato emitter, che deciderà di inviare i dati ad un ben determinato consumatore, precisamente scrivendo i dati nella relativa coda SPSC dedicata per la comunicazione fra il produttore e tale consumatore

Invece per quanto riguarda le code multi produttore – singolo consumatore (MPSC), sarà il thread consumatore, chiamato collector, che andrà a prelevare i dati dalla relativa coda SPSC del produttore.

Per le code MPMC, FastFlow offre una soluzione che si comporta similarmente a quanto detto prima, in cui il thread arbitro, chiamato collector-emitter (CE) si comporta sia da collettore che da consumatore.

- High-level programming: offre la possibilità di utilizzare gli skeleton algoritmici stream parallel, quali pipeline e farm, sfruttando il low layer di FastFlow.

  E' possibile utilizzare vari versioni di farm e pipeline, come ad esempio farm con canale di feedback, in cui è allocato anche un canale diretto fra il collettore e l'emettitore, farm senza uso del collettore ecc.. E' inoltre possibile la effettuare la composizione di skeleton.

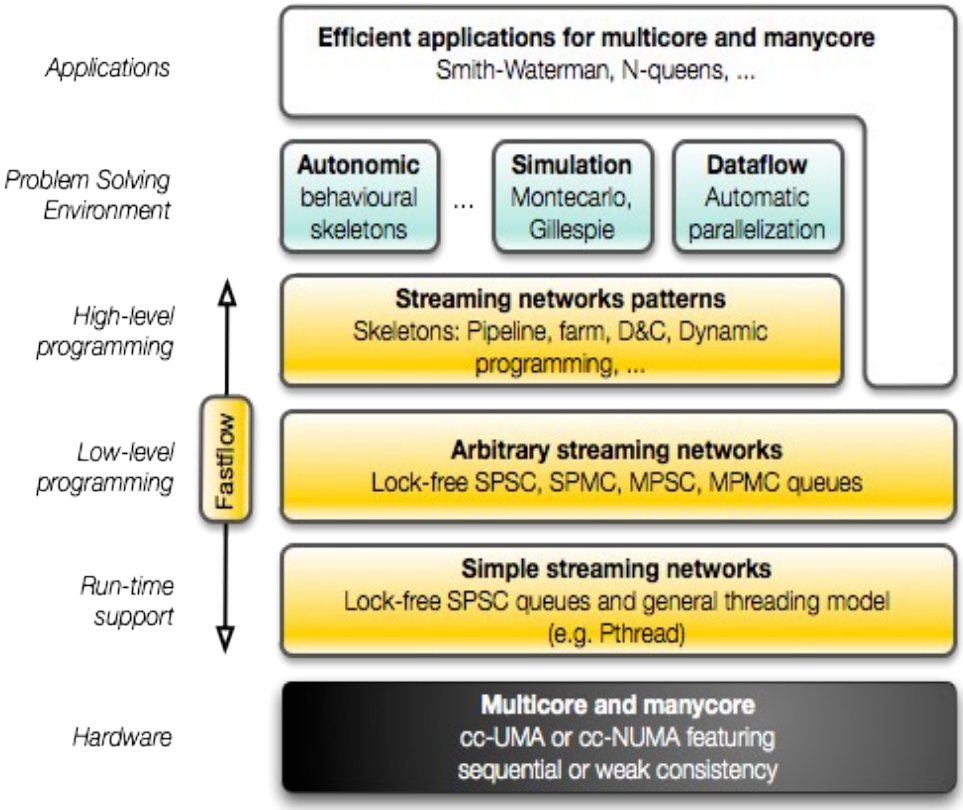


Figura 2.4 Architettura software di FastFlow

La Fig. 2.4 illustra l'architettura software di FastFlow, mettendo in luce i vari livelli di astrazione, che sono forniti all'utilizzatore.

2.2 PCAP (Packet Capture)

L'ispezione e analisi del traffico HTTP necessita l'impiego di una libreria per lettura e

scrittura del traffico dalla scheda di rete.

La libreria PCAP (*Packet Capture*) [TL13] fornisce un'interfaccia per la cattura del traffico di rete, con cui è possibile leggere i dati che arrivano dalla scheda di rete, sia quelli destinati alla macchina stessa che effettua la lettura, sia quelli destinati ad altri host sulla rete.

Tale libreria è stata scritta in C, ma esistono anche altre versioni per altri linguaggi di programmazione che semplicemente effettuano il wrapping della libreria originaria C.

È stata la libreria scelta poiché ovviamente disponibile per il linguaggio C++ (requisito dell'attività di tesi) ed inoltre in quanto offre una visione a basso livello dei frame letti dalla scheda di rete, avendo la possibilità della manipolazione bit a bit degli stessi, cosa decisamente importante per l'efficienza dell'applicativo.

Le principali funzioni di questa libreria permettono con una estrema facilità di cercare ed utilizzare le schede di rete, intese come NIC (Network Interface Card), impostarle in modalità promiscua[1], gestire filtri potenti e flessibili, selezionare grazie a questi il solo traffico che vogliamo ispezionare.

Permette inoltre un'accurata gestione degli errori nonché di debugging ed infine possiede anche degli strumenti per l'estrapolazione di statistiche sulle analisi effettuate.

L'architettura software di cattura dei pacchetti usata da PCAP su sistemi operativi *UNIX* like BSD (Berkeley Software Distribution), è rappresentata da BFP (Berkeley Packet Filter), si veda [MJ92].

1 La modalità promiscua è una modalità di controllo dell'interfaccia di rete con cui si vuole catturare tutto il traffico di rete che viene letto dalla scheda stessa, quindi i pacchetti che sono destinati alla macchina leggente e anche quelli che hanno un indirizzo MAC (Media Access Control - indirizzo a livello collegamento) diverso e quindi destinati ad altre macchine.

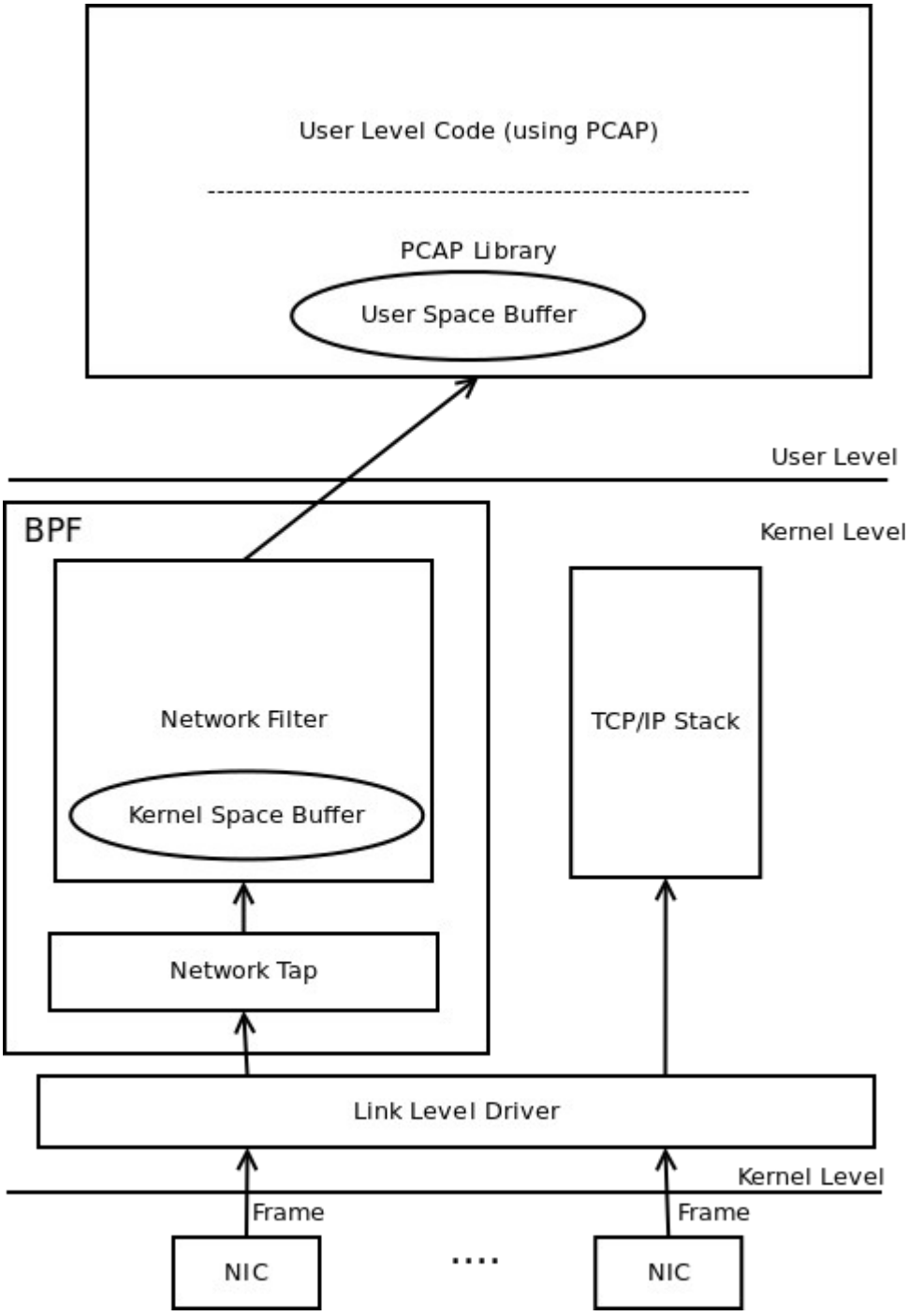


Figura 2.5 Architettura software BFP

La Fig. 2.5 illustra l'architettura software di cattura dei pacchetti usata da PCAP su sistemi operativi *UNIX* like BSD (Berkeley Software Distribution), in cui:

A livello più basso si hanno le NIC ed il driver per la gestione della relativa NIC, a livello del kernel l'architettura software BPF è fondamentalmente composta da due moduli:

- Network Tap: che ha il compito di prendere i pacchetti dalla NIC;
- Network Filter: che ha il compito di filtrare il pacchetto

A livello più alto abbiamo la libreria PCAP, che astrae sulle chiamate di più basso livello, riguardo l'interfacciarsi con i moduli del BPF di più basso livello, offrendo un'interfaccia di programmazione standard per i programmatori.

Da un punto di vista logico senza scendere nei dettagli implementativi, quando un pacchetto giunge alla NIC, il relativo driver della scheda di rete, normalmente lo invia al modulo di sistema che gestisce lo stack protocollare TCP/IP della macchina.

In questo caso invece il pacchetto viene prima inviato al BPF, e solo successivamente quando il BPF ha terminato la propria attività il pacchetto viene passato ai moduli per la gestione dello stack protocollare TCP/IP.

Il modulo Network Tap, prende il pacchetto e chiama il modulo del filtro per l'applicazione dei filtri su tale pacchetto. Il Network Filter, controlla il pacchetto per ciascun filtro presente, se tale pacchetto è accettato dal filtro, questo viene copiato nel relativo buffer di sistema (non necessariamente tutto il pacchetto è possibile che sia stato specificato solo una parte), e poi successivamente passato all'applicazione che ha allocato tale filtro quando l'applicazione ne richiede la lettura.

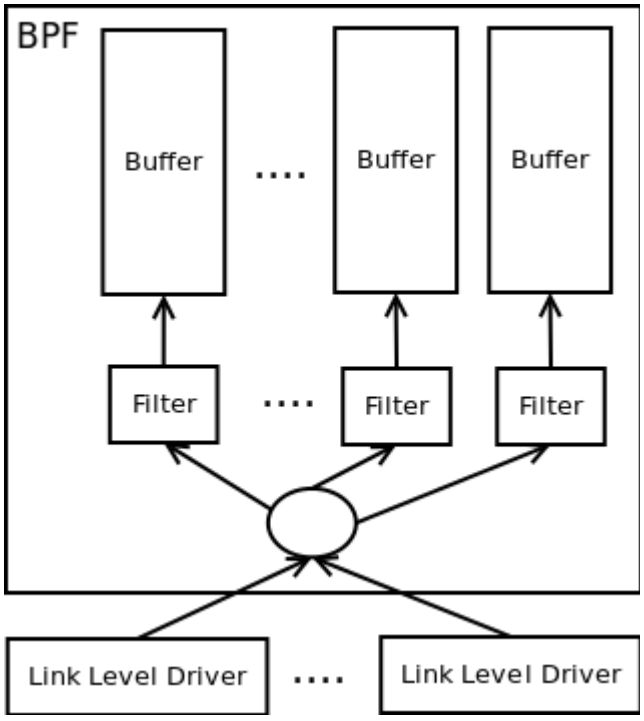


Figura 2.6 interna del BPF nel dettaglio

Per ridurre il numero di copie in memoria nei buffer di sistema, il pacchetto viene filtrato *"in place"* (ad esempio dove il driver della scheda di rete lo scrive in DMA in memoria) invece che essere copiato in qualche buffer di sistema prima del filtraggio. Questo permette al sistema di non effettuare nessuna copia se il pacchetto è scartato, in modo tale da gestire in modo efficiente tutti quei casi in cui si vuole solo una determinata parte del pacchetto.

A livello di codice applicativo dopo aver preparato un buffer per la ricezione dei pacchetti in spazio utente, si effettua una chiamata PCAP di lettura, dopodiché il pacchetto contenuto nel buffer di sistema gestito da BPF viene copiato nel buffer dell'applicazione in spazio utente.

Il filtro non è altro che una funzione che viene applicata ad ogni pacchetto letto e restituisce un valore booleano, per discriminare se il pacchetto può essere accettato o meno.

Nell'architettura software BPF sopra delineata, il filtro è realizzato tramite un processore virtuale [MJ92], dotato di un accumulatore, un registro indice, una memoria ed un program counter implicito, in grado di eseguire istruzioni molto semplici, quali: load, store, salto, confronto ed una serie di operazioni aritmetiche e poche altre ancora.

Un'applicazione che desideri filtrare una serie pacchetti letti dalla scheda di rete, non

deve far altro che specificare un espressione di filtraggio in una stringa in stile C, quindi come un array di caratteri con terminatore, secondo una determinata sintassi, si veda [MJ92], una volta specificata l'espressione, questa con un'ulteriore funzione PCAP viene compilata in un programma di filtraggio, che corrisponde ad un assembler simbolico RISC molto semplice, e questo viene inviato al modulo di filtering del BPF.

Questo metodo di filtraggio offre indubbiamente una serie di vantaggi:

- estendibilità: è facilmente estendibile verso nuovi protocolli di rete, modificando solo le funzioni ad alto livello di compilazione dei filtri, senza dover modificare l'architettura BPF di basso livello che gira in kernel mode;
- genericità: è utilizzabile in nuove situazioni peculiari;
- tempestività: il filtro è applicato il prima possibile e se il pacchetto non viene accettato, questo non è copiato in nessun buffer di sistema del BPF.

PCAP offre anche la possibilità dell'invio dei pacchetti, senza passare attraverso l'allocazione di socket o altre strutture dati che mette a disposizione il sistema operativo.

I pacchetti inviati sono a tutti gli effetti dei “raw packet” e devono essere “compilati” prima dell'invio per non spedire dati che non hanno senso, in altre parole è necessario preparare tutti le intestazioni protocollari al vari livelli della pila TCP/IP, senza che questi vengano inseriti di default dal sistema operativo come accade con i socket.

2.3 HTTP

In questo paragrafo si vuole presentare HTTP, senza scendere troppo nei dettagli, dando quindi una panoramica generale ed essenziale, puntualizzando solamente gli aspetti che sono importanti per lo scopo dell'analisi DPI.

Come afferma [KR08], il protocollo HTTP (Hypertext Transfer Protocol) è il protocollo di rete di livello di applicazione su cui si basa il servizio Web, tale protocollo è strutturato su un'architettura software client-server.
In altre parole HTTP definisce il modo in cui il client HTTP richiede le informazioni al server HTTP e come quest'ultimo risponde a tali richieste.
HTTP utilizza il protocollo a livello trasporto TCP (Transmission Control Protocol), il quale offre un trasferimento dei dati affidabile.

E' necessario fornire alcune informazioni riguardanti le varie tipologie di connessioni HTTP, in particolare le connessioni persistenti e il pipelining, la cui comprensione sarà necessaria ai fini dei capitoli successivi.

Nella versione 1.0 di HTTP, una transazione HTTP formata da una richiesta e da un

seguente risposta, utilizza un'unica connessione TCP. Questo implica che per ogni richiesta e seguente ricezione dell'oggetto richiesto, viene aperta una nuova connessione TCP, si parla in questo caso di HTTP con connessioni non persistenti.

Con la versione 1.1 di HTTP, la connessione TCP una volta inviata la risposta è lasciata aperta dal server, in questo modo le richieste e le risposte successive fra i due terminali possono essere inviate sulla medesima connessione TCP.
Utilizzando le connessioni permanenti sul lato client si ottengono i seguenti vantaggi:

- riduzione del numero di connessioni TCP aperte;
- diminuzione dell'uso di memoria, in quanto per ogni connessione si deve allocare un buffer e mantenere lo stato sia sul client che sul server;
- diminuzione dell'uso di CPU, in quanto in questa versione, per ogni oggetto inviato non si paga tutte le volte lo startup del canale TCP, con il *3-way-handshake* e la chiusura della connessione.

Tuttavia l'uso delle connessioni permanenti sul lato server comporta un costo addizionale, dato dall'aumento del fabbisogno di memoria.

Le modalità con cui possono essere effettuate le richieste HTTP sono le seguenti:

- *back-to-back*: quando si invia la richiesta si attende la risposta, e solo successivamente si invia una nuova richiesta;
- *pipelining*: quando le richieste di oggetti possono essere effettuate una dopo l'altra senza aspettare le risposte dell'altra parte in questione. In questo caso le risposte devono essere ricevute nello stesso ordine in cui sono state inviate le relative richieste, perché non è definito un modo esplicito di associazione fra la richiesta e la relativa risposta.

Attualmente i *broswer* (client HTTP) impiegano comunemente le connessioni persistenti con *pipelining*. Il formato dei messaggi di richiesta e risposta sono definiti nelle specifiche del protocollo HTTP, e scritti in ASCII, si veda [FIE99].

Per quanto riguarda i messaggio di richiesta, la prima riga è detta di "richiesta" e quelle seguenti di "intestazione".

La riga richiesta è formata da tre campi:

- metodo (GET, PUT, DELETE, ecc.);
- URL (Uniform Resource Locator) della pagina;
- versione HTTP, è autoesplicativa, appunto definisce se stiamo usando HTTP 1.0 o HTTP 1.1 ecc.

La maggior parte dei messaggi riguardanti le richieste HTTP sono effettuate tramite il metodo GET.

Come detto precedentemente, nella richiesta HTTP possono essere inserite anche varie e righe di intestazione, come ad esempio:

- Host: rappresenta il server e il numero di porta su cui risiede l'oggetto richiesto;
- User-agent: specifica il tipo di broswer che sta inviando la richiesta.

e molti altri ancora che non verranno delineati, in quanto ai fini dei capitoli successivi è necessaria soltanto la comprensione delle richieste GET HTTP e dell'intestazione Host.

2.4 Related work

In questo paragrafo si vuole dare una panoramica generale, dei framework di programmazione parallela struttura per il linguaggio C++ alternativi a FastFlow, nonché delle soluzioni di analisi DPI che riguardano lavori simili all'attività di tesi.

2.4.1 Framework per la programmazione parallela strutturata

Si presentano alcuni framework di programmazione parallela struttura, mettendone in evidenza le relative caratteristiche e spiegando la motivazione per cui FastFlow è stato preferito rispetto a loro.

- Muesli [TMS13]: è un framework di programmazione parallela strutturata per il C++, sviluppato dall'Università di Munster. Mette a disposizione del programmatore gli skeleton data e stream parallel. L'implementazione del framework è basata sull'approccio a template. Inizialmente era un framework dedicato allo sviluppo di applicazioni distribuite, la cui implementazione si basava sull'utilizzo di MPI [MPI13]. Successivamente il framework è stato esteso, con la possibilità della generazione di codice OpenMP [OTO13a], per lo sfruttamento del parallelismo derivante anche dalle architetture multi-core. Uno dei meriti maggiormente apprezzati di questo framework, è rappresentato dalla possibilità di esprimere gli skeleton tramite lo stile di programmazione OO tipico del C++, risultando un metodo molto intuitivo per gli utenti.

  Muesli non è stato impiegato come framework di programmazione parallela nell'implementazione dell'applicazione di firewalling, in quanto OpenMP non fornisce delle latenze di comunicazione con le stesse prestazioni di FastFlow. Per questo motivo non si presta particolarmente bene a computazioni di grana molto fine, come quella prevista nel nostro caso.

- SkeTo [SP13]: è un framework di programmazione parallela strutturata per il C++, sviluppato dall'Università di Tokyo. Il framework fornisce principalmente gli skeleton data parallel, anche se in alcune versioni intermedie, ha supportato quelli stream parallel. L'implementazione del framework è basata sull'approccio a template. Il framework è implementato utilizzando MPI [MPI13], quindi è utilizzato per supportate la programmazione parallela e distribuita. Principalmente permette agli skeleton di operare su strutture dati quali array e liste. La principale caratteristica che lo distingue dagli altri framework, consiste nel fatto che gli skeleton sono forniti come chiamate di libreria nel codice sequenziale, differentemente da quello che avviene negli altri framework in cui il programma parallelo è interamente costruito attorno agli skeleton.

  SkeTo non è stato utilizzato come framework di programmazione parallela nell'implementazione dell'applicazione di firewalling, in quanto non offre gli skeleton stream parallel.

- SkePU: è un framework di programmazione parallela C++, che fornisce principalmente skeleton data parallel, per CPU multi-core e sistemi multi-GPU. Il supporto relativo al parallelismo offerto dai sistemi multi-GPU è implementato con CUDA [CPP13] e OpenCL [OTO13b], mentre per quanto riguarda il supporto per il parallelismo offerto dalle CPU multi-core è implementato con OpenMP [OTO13a].

  Dato che il supporto per le architetture multi-core è implementato con OpenMP, questo framework non si adatta particolarmente bene alle computazioni di grana molto fine, per questo motivo non è stato impiegato nella realizzazione dell'applicazione. Inoltre non offre gli skeleton stream parallel (anche se in alcune versioni intermedie erano supportati).

- ASSIST (A software development system based upon integrated skeleton technology) [VAN02, ACC03, CDG01]: è un ambiente di programmazione parallela sviluppato dal Dipartimento d'Informatica dell'Università di Pisa. ASSIST mette a disposizione gli skeleton algoritmici, template-based per il linguaggio C++. ASSIST è un linguaggio di coordinamento (ASSISTcl) che fornisce un ambiente di sviluppo, inoltre fornisce gli strumenti di compilazione per ASSISTcl ed un run-time support portabile. Oltre agli skeleton algoritmici classici, è fornito il costrutto ParMod, uno “skeleton generico”, con cui è possibile definire forme di parallelismo arbitrarie.

  L'implementazione di ASSIST è basata su POSIX TCP/IP, pertanto è impiegato per il calcolo distribuito fra i vari nodi della rete, non fornendo nessun particolare supporto per lo sfruttamento del parallelismo derivante dalle architetture multi-core/multi-processore dei singoli nodi.

  ASSIST fornisce un ambiente di programmazione completo, espressivo e

flessibile. Applicazioni reali ne hanno dimostrato la validità, riportando ottimi valori di performance, si veda [ACD04, CPR04]. Tuttavia non è stato impiegato nell'attività di tesi, in quanto non supporta la programmazione parallela per lo sfruttamento delle architetture multi-core.

2.4.2 Soluzioni d'analisi DPI

Analizziamo alcuni dei molti lavori correlati all'attività di tesi, che riguardano il medesimo ambito di ricerca:

- Ndpi [NOE13]: è una libreria per l'analisi DPI, che ha lo scopo d'estendere le funzionalità di OpenDPI [OPE13], arricchendo tale libreria con ulteriori protocolli applicativi da poter analizzare. Il DPI è effettuato sull'intero pacchetto, quindi l'analisi è fatta tenendo conto sia delle intestazioni protocollari della pila TCP/IP che del payload. Tuttavia la libreria al suo interno non effettua il calcolo in parallelo, sfruttando a pieno le architetture multi-processore, è quindi il programmatore che deve farsi carico di progettare e realizzare la struttura parallela e su questa costruire un sistema di analisi sfruttando nDPI.
- L7-filter [ALP13]: è un classificatore per Netfilter [TNP13] (modulo firewall di Linux) che identifica i protocolli applicativi basandosi su dei pattern matching configurati. La classificazione del traffico è fondamentalmente basata sul pattern matching tramite espressioni regolari, tale analisi è effettuata solo sui primi pacchetti di ciascuna connessione TCP. Ovviamente un'analisi del genere è meno accurata rispetto ad un approccio full-DPI in cui si analizza ogni pacchetto, in ogni suo possibile campo.
- Libprotoident [AN13]: l'analisi effettuata si basa sul metodo LPI (Lightweight Packet Inspection). Questa consiste nella ricerca di determinate “application signature” nei primi byte del payload applicativo. LPI consiste in un'analisi DPI più leggera, la quale prende in esamina solo alcuni byte del payload. Nel caso specifico di Libprotoident, solo i primi quattro byte del payload vengono presi in considerazione, pertanto l'analisi LPI risulta essere meno accurata rispetto ad un approccio full-DPI.

  Ad esempio, nel nostro ambito di applicazione, quindi quello dell'analisi del protocollo HTTP, è possibile classificare con il metodo LPI se i pacchetti HTTP, tuttavia se volessimo leggere e parsare l'header host del pacchetto HTTP, dovremmo condurre un'analisi ben più in profondità.

  La libreria fornisce la possibilità di processare pacchetti sequenzialmente, pertanto qualora si volesse effettuare l'analisi sfruttando il parallelismo, il programmatore deve farsi carico della realizzazione della struttura parallela, ed al

di sopra di questa, effettuare le opportune chiamate di libreria.

- Nella soluzione offerta da [YX11] è presentato un sistema di analisi DPI che sfrutta il parallelismo offerto dalle architetture multi-processore. Tuttavia la distribuzione dei dati fra i vari thread è basata su una struttura dati condivisa. Per garantire la correttezza delle operazioni sulla struttura dati condivisa, sono presenti dei meccanismi di lock.

  L'impiego di tali meccanismi implica una sequenzializzazione delle operazioni che sono protette dai meccanismi di lock, la quale rappresenta un freno al parallelismo fisico fra le varie unità di esecuzione parallele, con una conseguente impossibilità nel raggiungimento dello speedup ideale.

- Nella soluzione offerta da [WCH09], è presentato un sistema di analisi DPI che sfrutta il parallelismo offerto dalle architetture multi-processore. Le operazioni di analisi a partire dal livello collegamento sino a quello applicativo sono parallelizzate con l'impiego del pattern pipeline.

  Inoltre in questo lavoro è presentato un metodo di compilazione per trasformare le applicazioni di rete a quelli sequenziali, in strutture parallele pipelined per consentire a tali applicazioni di sfruttare le architetture multi-core.

  Tuttavia il sistema presenta delle criticità derivanti dal bilanciamento di carico fra le varie unità di esecuzione parallele. Fatto che se non opportunamente gestito, porta ad un aumento dei relativi tempi di servizio dei vari moduli paralleli, con una conseguente degradazione delle prestazioni generali del sistema.

- In letteratura sono presenti molte soluzioni hardware-based. Esse realizzano l'analisi DPI andando ad effettuare del pattern matching sui dati a livello applicazione, alla ricerca di particolari pattern per classificare il protocollo oppure alla ricerca di pattern che potrebbero identificare delle minacce di sicurezza, come viene effettuato nei sistemi IPS/IDS (Intrusion Detection/Prevention Systems), si veda [ROE99, SID12].

  Le implementazioni di queste soluzioni sono fondamentalmente basate sull'utilizzo di FPGA (Field Programmable Gate Arrays) si veda [ACF05, TRI05, THD12], oppure sull'impiego di CAM (Content Addressable Memories), si veda [YRH04, WTD06, YL05, AMK06]. La FPGA, è un circuito integrato digitale la cui funzionalità è programmabile via software, si tratta infatti di dispositivi la cui funzionalità da implementare non viene impostata dal produttore, sono

programmati direttamente dall'utente finale. La CAM è una memoria associativa, si tratta della trasposizione hardware del concetto di vettore associativo. A differenza delle memorie RAM, in cui l'utente fornisce un indirizzo di memoria e il dispositivo restituisce il dato contenuto in esso, nelle CAM l'utente fornisce il dato e la memoria restituisce la lista di indirizzi in cui esso è immagazzinato.

Tuttavia queste soluzioni non associano all'analisi DPI un concetto di stato del flusso, pertanto l'analisi è effettuata indipendentemente pacchetto per pacchetto.

Capitolo 3

Progettazione architetturale

In questo capitolo si vuole presentare l'architettura software dell'applicazione parallela per effettuare firewalling DPI per il protocollo applicativo HTTP, relativamente al traffico su reti Ethernet, utilizzando FastFlow.

In questo capitolo si fornisce una visione ad alto livello, senza quindi scendere in dettagli implementativi. Inoltre si delineano le principali scelte progettuali, le problematiche rilevanti incontrate e le soluzioni adottate per farne fronte.

Progettare un'applicazione parallela significa pensare ad identificare tutte le possibile attività concorrenti che possono essere eseguite in parallelo, definendo quali e quante sono.

Quando si parla di gestione e di coordinazione fra le varie attività concorrenti, si intende:

- la gestione della cooperazione tra le entità di esecuzione concorrente, che prevede uno scambio di dati fra le stesse;
- la gestione delle competizione, che si ha nel momento in cui le entità di esecuzione concorrente richiedono l'uso di risorse comuni che non possono essere utilizzate contemporaneamente.

Si parla di sincronizzazione diretta o esplicita per indicare i vincoli propri di una cooperazione, come ad esempio la comunicazione che si può avere nel caso di un produttore e di un consumatore tramite un buffer, in cui esiste un vincolo di precedenza fra le operazioni delle varie entità concorrenti, che devono pertanto essere sincronizzate.

Si parla invece di sincronizzazione indiretta o implicita per indicare i vincoli imposti sulla competizione, in cui anche in questo caso esiste un vincolo di precedenza tra le operazioni con le quali i processi possono accedere alla risorsa condivisa In questo caso l'ordine di accesso alla risorsa è indifferente, purché le operazioni siano nel tempo mutuamente esclusive.

3.1 Schema logico

Si vuole progettare un'applicazione che effettui del bridging fra diverse schede Ethernet, che effettua firewalling relativo al protocollo applicativo HTTP con analisi DPI.

Andiamo ad analizzare ad alto livello la migliore decomposizione del problema in attività

concorrenti.

I pacchetti vengono letti da una NIC (Network Interface Card), successivamente questi devono passare attraverso un modulo di firewalling, che effettuata un'analisi DPI.

Il firewall dichiara lo status del pacchetto in analisi, esso può essere definito “legale” e conseguentemente verrà scritto in uscita sulla NIC su cui il pacchetto era originariamente diretto, altrimenti il pacchetto viene scartato e non verrà successivamente inviato.

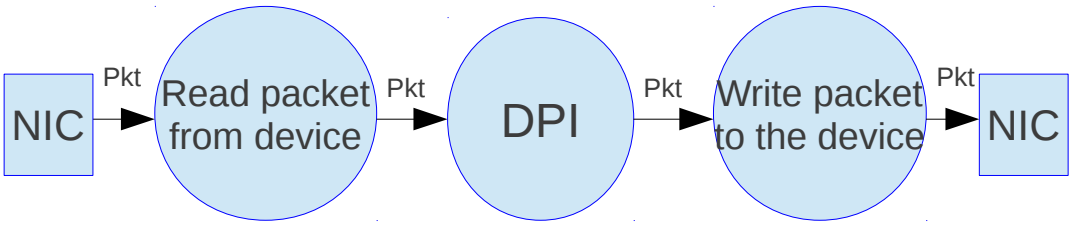


Figura 3.1 Decomposizione del problema in attività concorrenti

Una volta che la computazione è stata progettata a livello logico, possiamo valutare le soluzioni relative agli skeleton algoritmici (oppure i meccanismi di basso livello) che FastFlow ci mette a disposizione. E' necessario scegliere il pattern di programmazione parallela che più si adatta alla computazione che vogliamo effettuare.

In fase di progettazione si è deciso di utilizzare una farm, che ovviamente è il pattern più adeguato a gestire questo problema, poiché siamo di fronte ad uno problema su stream, si devono leggere i dati dalla scheda di rete, parsarli, analizzarli ed infine scriverli sull'interfaccia di uscita. Viene naturale ed intuitivo allocare l'attività di lettura dalla scheda di rete sull'emitter della farm, l'attività di parsing e di firewalling sul worker ed infine l'attività di scrittura sul collector della farm.

Data la natura del problema, si può notare come sia necessario leggere e scrivere i dati da varie schede di rete diverse. Infatti l'applicazione dovrà essere in grado come minimo di gestire il traffico fra due schede di rete (bridging fra due reti).

Tuttavia l'emitter della farm non sarebbe in grado di mettersi in lettura da tutte le schede di rete in gestione, per ovvi motivi legati alle prestazioni. Allo stesso modo un solo collector non sarebbe in grado di gestire la scrittura di pacchetti su tutte le diverse schede di rete, per le stesse motivazioni legate alle prestazioni.

Quanto appena affermato è l'aspetto cruciale del problema ed è fondamentale trovare una soluzione che ci permetta di allocare un emitter ed un collector per ogni scheda di rete.

La soluzione che andremo a delineare può essere vista come una variante dello skeleton farm, in cui vi sono più emitter e più collector, precisamente un emitter ed un collector per ogni scheda di rete in gestione.

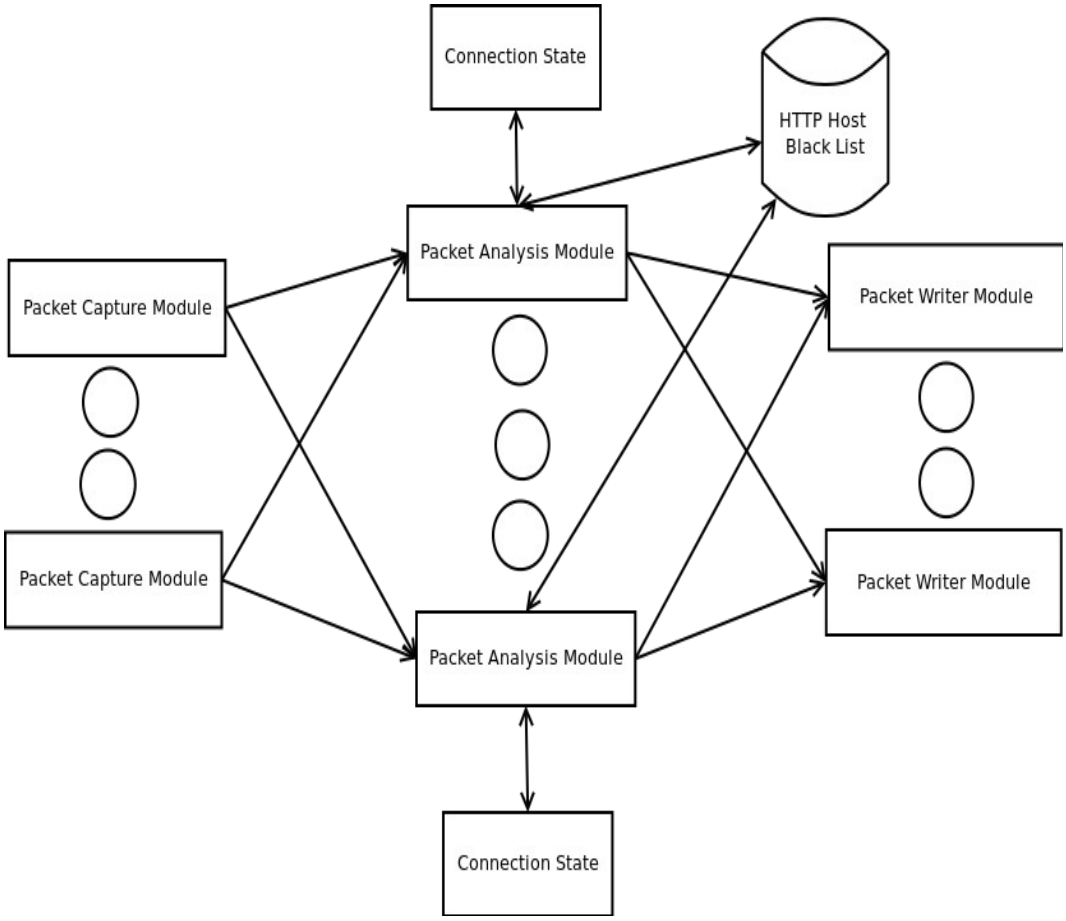


Figura 3.2 Struttura logica ad alto livello dell'applicativo

La Fig. 3.2 mette in evidenza la struttura logica della parte parallela dell'applicazione, nonché le strutture dati d'appoggio per il mantenimento nello stato. Andiamo ad analizzare più dettagliatamente ogni componente:

- Packet Capture Module : rappresenta l'entità di esecuzione parallela delegata a catturare i pacchetti dalla NIC. Nella farm corrisponde al modulo dell'emitter, il quale non fa altro che catturare il pacchetto, ed inviarlo ad un specifico worker per le successive analisi.
- Packet Analysis Module: rappresenta l'entità di esecuzione parallela delegata all'analisi dei pacchetti. Nella farm corrisponde al modulo del worker. Questo modulo parsa il pacchetto e verifica che sia HTTP (se non lo è il pacchetto non viene analizzato). Il pacchetto viene scartato o lasciato passare a seconda delle delle informazioni contenute nel pacchetto stesso e in base allo stato del flusso di

cui il pacchetto fa parte;

- Packet Writer Module: corrisponde all'entità di esecuzione parallela delegata alla scrittura dei pacchetti filtrati sulla NIC. Nella farm corrisponde al ruolo del collector;
- HTTP Host Black List: contiene le informazione relative alla lista nera degli host, cioè tutti quegli host a cui non è consentito collegarsi in HTTP;
- Connection State: rappresenta lo stato del singolo worker. E' una struttura dati privata al worker, in cui si memorizza lo stato dei flussi di sua pertinenza. Questa struttura dati essendo privata al worker, viene acceduta e aggiornata dinamicamente dal solo worker che la possiede.

3.2 Gestione dello stato

Ogni worker possiede un proprio stato privato (a cui gli altri worker non possono accedere), in cui tiene traccia delle varie connessioni TCP su cui viaggiano i messaggi HTTP. Tali connessioni vengono considerate dei flussi di dati bidirezionali che sono univocamente determinate dalle seguenti informazioni.

- indirizzo IP sorgente
- indirizzo IP destinazione
- porta TCP sorgente
- porta TCP destinazione

Un flusso di dati può essere definito in vari modi, si è deciso di utilizzare un approccio per la definizione di flusso in stile NetFlow v5 [MFT13], ma con alcune modifiche:

- l'utilizzo di sottoinsieme degli attributi rispetto al protocollo NetFlow v5, in quanto tutte le informazioni non sarebbero state utili al nostro scopo;
- il flusso è considerato bidirezionale (anziché mono-direzionale con in NetFlow v5).

Sia H_s un terminale che vuole instaurare una connessione usando porta p_s, verso un terminale H_d con porta p_d, ed essendo il flusso bidirezionale, i dati proventi da H_s verso H_d, che da H_d verso H_s, verranno mappati in un unico record di flusso identificato univocamente dalla quadrupla (H_s, H_d, p_s, p_d).

Ogni record di flusso, può trovarsi in uno dei seguenti tre stati:

- Lecito: il flusso è stato valutato in modo positivo, in quanto non sono transitati pacchetti in contrasto con le politiche di firewalling stabilite.

- Bloccato: il flusso è stato valutato negativamente, poiché sono stati letti dei pacchetti in contrasto con le politiche di firewalling stabilite.
- Sconosciuto: l'applicazione non è stata ancora in grado di valutare il flusso. Il flusso è nello stato “sconosciuto” finché non viene letto l'host nel payload HTTP. Una volta letto l'host il flusso viene marcato come “lecito” o “bloccato” a seconda delle politiche di firewalling stabilite.

  Ad esempio nella fase del “three-way handshake” relativa all'instaurazione della connessione, nei primi due messaggi nessun dato applicativo HTTP transita, dunque il relativo modulo di firewalling marca il flusso con stato sconosciuto.

La procedura utilizzata dal protocollo TCP per l'instaurazione di una connessione fra due terminali è chiamata “three-way handshake” indicando appunto la necessità di scambiare tre messaggi tra host mittente e host ricevente.

Per far sì che la connessione sia instaurata correttamente: il lato client di TCP invia al lato server TCP, un particolare segmento[2] con il bit di SYN posto a 1, chiamato appunto segmento di SYN.

Il server risponde a sua volta con un altro segmento con i flag SYN e ACK impostati ad 1. Infine alla ricezione del SYN+ACK, il client TCP risponde con un ACK (quindi con il bit di SIN settato a 0) e la connessione risulta stabilita correttamente. In questo ultimo passaggio il client può già inserire nel segmento del carico utile, cioè dei dati applicativi che vuole inviare al lato server TCP.

I segmenti inviati durante la fase di handshake sono solo puramente intestazioni, ossia non trasportano nessun dato a livello applicativo (essendo questa una fase di sincronizzazione tra i due host e non di scambio di dati), l'applicazione crea quindi un nuovo record di flusso tutte le volte che legge il primo segmento di SYN che ovviamente non corrisponde a nessuna chiave di flusso già precedentemente allocata nella struttura dati privata al worker. Una volta che il record di flusso è stato creato tutta la comunicazione fra i due terminali viene mappata su tale record.

In una comunicazione HTTP, una volta che la connessione TCP è stata instaurata fra i due terminali, il client invia un pacchetto di richiesta HTTP e la parte dell'applicazione di firewalling non fa altro che andare a leggere tale pacchetto e ricercare il campo “Host” (se presente) e prenderne il relativo contenuto.

Se la stringa relativa al campo host, fa parte degli host che sono presenti nella lista nera che è caricata all'avvio dell'applicazione, ovviamente il pacchetto viene scartato (e non arriverà mai al server), ed il flusso viene marcato con uno stato di “flusso bloccato”.

Le operazioni eseguite all'interno del modulo di analisi dei pacchetti (worker) sono:

2 Il segmento è il termine utilizzato a livello trasporto della pila TCP/IP per indicare i messaggi che si scambiano il client ed il server a tale livello di astrazione. Corrisponde ad un pacchetto visto a livello applicazione.

- Arrivo di un pacchetto HTTP non appartenente a nessun record flusso precedente: il pacchetto viene processato, viene creata una nuova entry nella struttura dati privata relativa al nuovo record di flusso. Tale entry identifica il flusso univocamente e tutta la comunicazione del flusso sarà mappata su tale record. Se non è presente nessun payload applicativo HTTP contenente l'header host il flusso viene marcato con stato sconosciuto. In questa categoria ricadono tutti quei pacchetti che sono inviati nel "three-way handshake. Altrimenti se vi è del payload HTTP viene eseguita l'analisi del pacchetto per controllare che l'host a cui si tenta di accedere non sia presente nella black list degli host.
- Arrivo di pacchetti HTTP appartenenti a record di flusso già esistenti: il pacchetto viene processato, viene controllato se il flusso a cui appartiene è già bloccato od è permesso, ed il pacchetto viene scartato o fatto passare di conseguenza.

  Se invece lo stato del flusso è ancora sconosciuto, si controlla se il pacchetto ha al suo interno una richiesta HTTP. Nel caso della richiesta HTTP, si analizza l'host specificato fra i vari header della richiesta stessa. Se l'host a cui si tenta di accedere non è presente nella black list degli host il pacchetto viene lasciato passare e lo stato del flusso viene settato a "lecito", altrimenti il pacchetto viene scartato e il flusso viene settato a "bloccato". Invece se il pacchetto non contiene nessuna richiesta HTTP il pacchetto viene lasciato passare, e lo stato del flusso rimane invariato.
- Arrivo di un pacchetto non avente come protocollo HTTP: il pacchetto non viene analizzato e viene lasciato passare;

La struttura dati privata ha bisogno di essere periodicamente gestita e "ripulita "in modo tale da eliminare le informazioni che non servono più, che non sono più utili. Queste possono corrispondere a:

- Connessione terminata: una connessione TCP non è considerata una singola connessione bidirezionale, ma piuttosto come due connessioni monodirezionali. Per cui ogni host deve terminare la sua connessione.

  Possono anche coesistere connessioni aperte in un solo senso, in cui solo uno dei due terminali ha chiuso la connessione e non può più trasmettere, ma può ricevere i dati dall'altro terminale. Pertanto la chiusura della connessione si può effettuare in due modi:

    - con l'handshake a tre vie: si ha quando le due parti chiudono contemporaneamente le rispettive connessioni. Il comportamento è lo stesso di quanto fatto per l'apertura della connessione, la differenza consiste nel fatto che questa volta il flag utilizzato è quello di FIN anziché quello del SYN. Abbiamo quindi che un terminale invia un pacchetto con la richiesta FIN, l'altro risponde con un FIN + ACK, ed

infine il primo manda l'ultimo ACK e l'intera connessione viene terminata.

    - con l'handshake a quattro vie: questo avviene quando le due connessioni vengono chiuse in tempi diversi. In questo frangente uno dei due terminali invia la richiesta di FIN e attende l'ACK. L'altro terminale farà poi la stessa cosa successivamente.

- Inattività del flusso: nessun pacchetto relativo a un flusso è stato catturato in un determinato intervallo di tempo.

Capitolo 4

Implementazione

In questo capitolo si presenta e si descrive l'implementazione dell'applicazione progettata. Verrà data una visione del sistema nel suo globale, fornendo una descrizione più dettagliata solo per le parti più rilevanti. Inoltre verranno messi in evidenza gli aspetti realizzativi e tecnici.

4.1 Analisi del pattern parallelo impiegato

Utilizzando il framework di programmazione parallela strutturata FastFlow, è stata svolta un'attività per cercare lo skeleton che più si adattasse alla risoluzione del problema in questione.

Come spiegato nel capito della progettazione, è stato spiegato come lo skeleton farm fosse il candidato ideale per comporre il building block dell'applicativo.

Considerando l'alta banda del traffico in ingresso dalla singola scheda di rete, il solo emitter della farm non sarebbe in grado di mettersi in lettura da tutte le schede di rete in gestione, per motivi prettamente legati alle prestazioni. Quanto appena affermato è l'aspetto cruciale del problema ed è fondamentale trovare una soluzione che ci permetta di allocare un emitter per ogni scheda di rete. In questo modo è possibile allocare un'entità parallela volta a leggere in modo dedicato il traffico da una determinata scheda di rete. Pertanto l'impiego di un insieme di entità parallele in lettura del traffico di rete, precisamente una per ogni scheda in gestione, garantisce la possibilità di “immettere” nel sistema di filtraggio una grande quantità di traffico, che dovrà essere puntualmente processata da altre entità parallele dedite ai compiti di filtraggio.

Allo stesso modo un solo collector non sarebbe in grado di gestire la scrittura dei pacchetti su tutte le diverse schede di rete, sempre per motivazioni legate alle prestazioni. Perciò anche in questo caso si necessita dell'impiego di un insieme di entità parallele (una per ogni scheda di rete in gestione) al fine di scrivere i pacchetti in uscita sulla scheda di rete, per garantire la possibilità al sistema di filtraggio di “smaltire” grandi quantità di traffico già analizzate.

In sintesi per poter ottenere le migliori prestazioni in termini di banda e scalabilità, sia sul numero di interfacce di rete in gestione, si ha la necessità di realizzare una variante della farm, una soluzione avente più emitter e più collector, precisamente uno per ogni scheda di rete in gestione.

E' importante sottolineare come le prestazioni rappresentino l'aspetto cruciale, per cui

esse ci obbligano quindi ad adottare una soluzione alternativa al pattern farm.

Tuttavia FastFlow mette a disposizione una farm in cui è possibile allocare al più un emitter ed un collector (come nella definizione tipica della farm). Per far fronte a questo problema, ed implementare la soluzione con più emitter e collector, abbiamo dovuto analizzare minuziosamente tutte le soluzioni offerte da FastFlow ai suoi vari livelli di astrazione software.

4.1.1 Valutazione delle soluzioni ad alto livello

Ad alto livello di FastFlow sarebbe stato possibile effettuare una composizione di skeleton algoritmici, allocando una pipeline con tre stage, in cui ogni stage è composto da una farm:

1. primo stadio della pipeline: una farm, i cui worker sono delle entità di esecuzione che leggono i dati dalla relativa scheda di rete di pertinenza;
2. secondo stadio della pipeline: una farm, i cui worker sono delle entità di esecuzione che parsano ed effettuano un'analisi DPI sui dati giunti;
3. terzo stadio della pipeline: una farm, i cui worker sono delle entità di esecuzione che scrivono i dati sulla relativa scheda di rete di pertinenza.

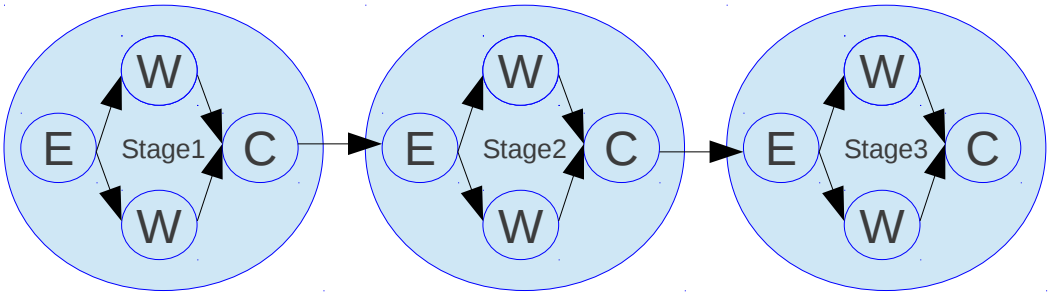


Figura 4.1 Implementazione dell'applicazione tramite composizione di skeleton

In Fig. 4.1 si può notare come ciascuno stadio della pipeline, sia collegato tramite un canale di comunicazione fra il collector dello stadio i-esimo e l'emitter dello stadio i+1-esimo.

Possiamo affermare che questa soluzione non offra una buona scalabilità all'aumentare della schede di rete, in quanto il throughput del primo stadio, è legato al throughput del

solo collector all'interno del primo stadio.

Questo fatto fa sì che aumentando i worker allocati per la lettura dalla scheda di rete si possa non ottenere un aumento del throughput del primo stadio.

Per dimostrare quanto detto, consideriamo il tempo di servizio T_s del primo stadio della pipeline.

Esso è dato dal $Max(T_{emitter}, T_{worker}, T_{collector})$,

dove:

- $T_{emitter}=0$ in quanto non effettua nessuna operazione;
- $T_{worker}=\frac{T_{seq}}{N_{worker}}$.

quindi otteniamo che $T_S=Max(T_{emitter}, T_{worker}, T_{collector})=Max(0,\frac{T_{seq}}{N_{worker}}, T_{collector})$.

Nel caso in cui il numero di schede di rete da cui bisogna leggere i dati aumentino, dobbiamo necessariamente allocare una maggiore quantità di worker in lettura. Un worker per ogni scheda di rete.

Pertanto in termini asintotici abbiamo:

$$T_s=\lim_{N_{worker}\to+\infty} Max(0,\frac{T_{seq}}{N_{worker}}, T_{collector})$$

$$T_s=\lim_{N_{worker}\to+\infty} Max(0,\frac{T_{seq}}{N_{worker}}, T_{collector})=Max(0,\lim_{N_{worker}\to+\infty}\frac{T_{seq}}{N_{worker}}, T_{collector})$$

$$T_s=Max(0,\lim_{N_{worker}\to+\infty}\frac{T_{seq}}{N_{worker}}, T_{collector})=Max(0,\frac{T_{seq}}{+\infty}, T_{collector})=T_{collector}$$

Quindi possiamo dedurre che T_s è limitato inferiormente da $T_{collector}$, che è legato quindi alla frequenza della singola unità di esecuzione (core nel nostro caso) su cui è allocato il collector.

Abbiamo bisogno di trovare un'implementazione della farm che ci permetta una maggiore scalabilità al variare del numero di schede in lettura da dover gestire, in quanto come si è appena visto, in questa soluzione T_s è legato a $T_{collector}$.

4.1.2 Valutazione delle soluzioni a basso livello

Scendere al cosiddetto low-level di FastFlow, significa passare ad un livello di astrazione

tale in cui è possibile utilizzare direttamente i canali di comunicazione fra le entità di esecuzione parallele (come i vari tipi di code già spiegate nel capito delle tecnologie).

A questo livello è possibile utilizzare direttamente i vari tipi di canali di comunicazione, senza l'intermediazione dei pattern paralleli ad alto livello, dovendo gestire anche la creazione, la sincronizzazione e la terminazione dei vari thread, che rappresentano appunto le varie entità di esecuzione parallele.

In sostanza a questo livello non è più possibile avere la visione relativa agli skeleton algoritmici, astraendo sui vari problemi di allocazione delle unità di esecuzione, gestione delle comunicazioni e sincronizzazione fra le entità di esecuzione. Abbiamo però la possibilità di definire in maniera del tutto arbitraria un qualsiasi pattern parallelo, pagando il tutto con un maggiore sforzo a livello di programmazione per la gestione dell'intera orchestrazione.

FastFlow mette a disposizione i seguenti canali di comunicazione:

- le code SPSC bounded e unbounded;
- le code MPSC bounded e unbounded;
- le code SPMC bounded e unbounded;
- le code MPMC bounded e unbounded.

Una coda unbounded può contenere un numero teoricamente infinito di messaggi (ovviamente nella pratica esso è legato alle dimensioni massime della memoria disponibile), mentre la bounded ne può contenere fino a un certo limite predefinito, e se tale limite viene raggiunto, le successive operazioni di inserimento falliscono.

A prima vista le code unbounded sono più allettanti, in quanto non impongono limiti di bufferizzazione da dover gestire, svincolandoci dalla gestione dei fallimenti di scrittura su code piene, tuttavia possono portare ad eventi molto pericolosi.

Nel nostro caso è relativamente semplice la scelta fra una soluzione bounded ed una unbounded, in quanto nelle condizioni in cui il T_s dell'emitter fosse inferiore al T_s del worker ed in caso di massimo carico di traffico di rete da dover gestire, per arco di tempo prolungato, se la coda fosse unbounded si verificherebbe una saturazione della memoria (cosa da evitare ovviamente).

Da questo si evince che data la natura del problema, è necessario optare per una soluzione bounded. In questo caso dunque è necessario definire un'opportuna politica per la gestione della situazione in cui la coda sia piena. Pertanto si applica la tipica politica utilizzata dai router store-and-forward [TAN03], che prevede di scartare l'ultimo dato che non può essere memorizzato per mancanza di capacità in spazio. Questo aspetto verrà ulteriormente puntualizzato nel capitolo delle sperimentazioni.

4.1.2.1 Code MPMC

Usando la struttura dati MPMC [VYU12], un insieme di thread emitter inseriscono nella coda i puntatori ai dati ed un altro insieme di thread worker leggono dalla struttura dati tali puntatori, senza l'utilizzo di chiamate per l'allocazione e liberazione di memoria dinamica.

La struttura dati non è protetta con dei meccanismi di lock, tuttavia l'implementazione di tale struttura, utilizza al proprio interno operazioni atomiche RMW (read-modify-write), primitive fornite direttamente dal processore, che nascondono in realtà un utilizzo effettivo di meccanismi di lock.

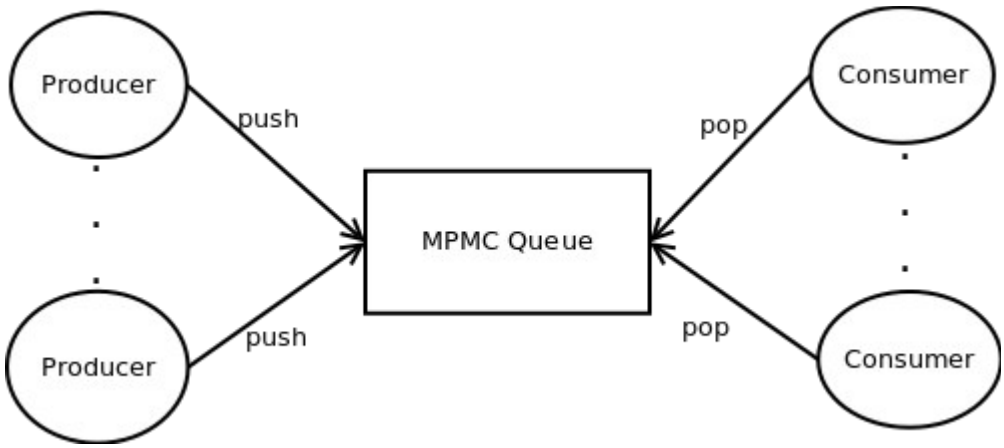


Figura 4.2 Struttura Dati MPMC queue

La Fig. 4.2 illustra la coda MPMC in cui i produttori con delle operazioni di push inseriscono i dati nella coda, mentre i consumatori con delle operazioni di pop estraggono i dati da tale coda.

E' importante sottolineare come la coda MPMC sia una struttura dati, che i vari thread produttori e consumatori possono utilizzare per scambiarsi messaggi. Questa struttura dati è concettualmente diversa dalla coda MPMC delineata nella Fig. 2.3, in quanto in quel caso il comportamento di una coda MPMC è modellato da thread arbitro, chiamato collector-emitter (CE) che si comporta sia da collettore che da consumatore. Il CE preleva i dati dalle singole code SPSC dei produttori e li scrive nelle code SPSC in ingresso dei consumatori. In quest'ultima soluzione non vi è nessuna struttura dati passiva, ma la coda MPMC è rappresentata da un'ulteriore thread, chiamato appunto CE.

Entrambe le due soluzioni non si adattano particolarmente bene al problema in quanto la struttura dati MPMC queue, utilizza al suo interno atomic RMW, che se possibile vorremmo evitare, mentre per quanto riguarda la coda MPMC modellata da un thread CE, sarebbe soggetta a fenomeni di scarsa scalabilità al variare delle schede di rete in gestione, dimostrando il tutto con un ragionamento del tutto simile a quanto affermato in 4.1.

4.1.2.2 Code SPMC/MPSC

Sfruttare le code SPMC significa creare una struttura parallela, in cui ogni thread emitter legge i dati dalla relativa scheda di rete di competenza, ogni thread worker ha allocato una propria coda SPSC in ingresso privata, da cui solo tale worker può leggere. Sarà quindi compito del thread emitter quello di scrivere il puntatore ai dati letti, nella coda del thread worker adeguato.

Riguardo le code MPSC abbiamo che i thread worker, hanno una loro coda SPSC privata in uscita, in cui il worker stesso immette i puntatori ai dati alla fine di ogni sua computazione. Sarà compito di un thread collector quello di leggere dalla coda SPSC di ogni singolo worker per prelevare i dati.

Come si può notare da quanto delineato precedentemente si ha un emitter che ha la possibilità di scrivere nelle code in ingresso dei singoli worker, ed un thread collector che ha la possibilità di leggere da una insieme di code di uscita dei singoli worker.

Tuttavia si ha la necessità di allocare più emitter e più collector, quindi una delle possibili implementazioni sarebbe quella di proteggere le singole code SPSC in ingresso di ogni worker con dei meccanismi di spin-lock, per garantire la mutua esclusione dei produttori sulla scrittura sulla singola coda SPSC di ingresso di ogni worker. Inoltre si deve impiegare anche un altro spin-lock per garantire la mutua esclusione dei consumatori sulla lettura sulla singola coda SPSC di uscita di ogni worker.

Gli spin-lock, si veda [ABC08], sono dei metodi di risoluzione a problemi di mutua esclusione (sincronizzazione diretta), che sono caratterizzati da condizioni di attesa attiva dei thread, che non possono entrare nella sezione critica richiesta.

Questi thread che non possono entrare nella sezione critica ripetono la sequenza di istruzioni con la quale richiedono l'accesso alla sezione mantenendo l'esecuzione sul core su cui sono in esecuzione, quindi con un conseguente spreco di risorsa.

Gli spin-lock hanno il vantaggio di non richiedere il cambio di contesto nel caso in cui il thread sia in attesa dell'accesso alla sezione critica. Tali meccanismi sono utilizzabili solo in sistemi multi-core e dovrebbero essere utilizzati quando l'attesa media è inferiore al tempo di due cambi di contesto. Pertanto dovrebbero essere utilizzati solo per sezioni critiche molto brevi.

Anche la soluzione basata sulle code SPMC/MPSC non è del tutto appropriata, in quanto l'utilizzo di spin-lock o in generale di operazioni di lock come mutex o monitor. Questi meccanismi introdurrebbero una sequenzializzazione nelle operazioni di scrittura degli emitter e nelle operazioni di lettura dei collector, degradando conseguentemente in maniera considerevole le prestazioni dell'architettura parallela dell'applicazione. E'

necessario pertanto trovare una soluzione che non usi tali meccanismi.

4.1.2.3 Code SPSC

Le code SPSC [TOR10], sono delle strutture dati in cui il produttore inserisce nella coda gli elementi ed il consumatore li preleva dalla coda, con una politica FIFO (Fist In First Out).

Le code SPSC di FastFlow sono ottimizzate per le architetture multi-core con cache condivisa. Utilizzano l'algoritmo “Lamport’s circular buffer”, per ulteriori approfondimenti consultare [LAM83].

Utilizzando tale struttura dati è possibile creare una struttura parallela, in cui ogni thread worker ha allocato un insieme di proprie code SPSC in input private, una coda per ciascun emitter esistente. Ciascun thread emitter ha quindi una coda a lui dedicata per ogni worker esistente, può quindi scrivere il puntatore ai dati letti nella coda di sua pertinenza di un generico worker, senza dover introdurre nessun meccanismo di lock.

Una cosa del tutto simile vale per i thread collector, che ha allocato un insieme di proprie code SPSC in input private, una coda per ciascun worker.

Il thread worker è quindi delegato alla scrittura del puntatore dei dati nell'appropriata coda di un determinato thread collector. Il thread worker sceglie il collector in base all'interfaccia di uscita su cui il pacchetto deve essere indirizzato.

Come si può notare da quanto delineato precedentemente, in questa struttura parallela si ha un insieme di emitter che hanno la possibilità di comunicare con ogni worker ed ogni worker può comunicare direttamente con ogni collector.

Ogni worker offre una coda dedicata per ogni emitter, pertanto si evince che tutte le comunicazioni sono punto-a-punto, create sfruttando solamente le strutture dati SPSC. Inoltre questa soluzione è totalmente lockless, infatti non si fa uso di nessun meccanismo di lock. Del tutto speculare il rapporto fra i worker ed i collector.

In questa ultima versione architetturale, i thread worker e collector hanno in ingresso un insieme di code SPSC da cui consumare i messaggi. Essi adottano una politica “round-robin” riguardo lettura sull'insieme di code SPSC in ingresso (quindi una lettura circolare). Le letture dalle singole code SPSC non sono bloccanti, infatti se la coda è vuota, il thread che effettua la lettura riottiene immediatamente il flusso di controllo con un risultato di fallimento in lettura, di conseguenza non rimane bloccato in attesa che il produttore scriva un dato. Per questa ragione, nel caso di lettura da una coda vuota, si può passare immediatamente alla lettura della coda SPSC successiva.

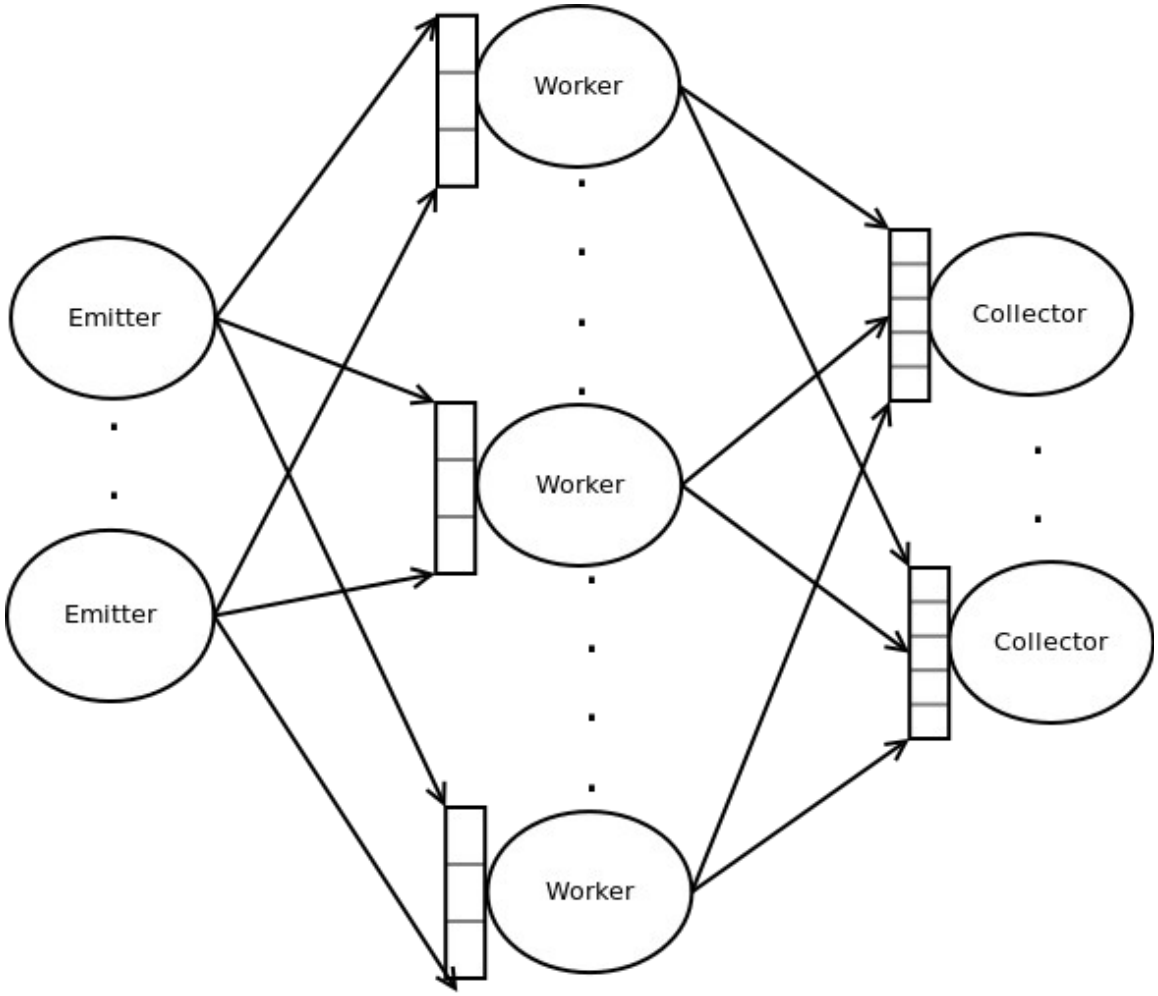


Figura 4.3 Architettura software a livello implementativo

4.2 Gestione dello stato del flusso

Ogni thread worker deve possedere le informazioni relative alla connessioni fra terminali per prendere le decisioni sul traffico di rete.

Questa struttura dati che le contiene, deve essere privata ad ogni thread worker, perché se così non fosse, sorgerebbero una serie di problemi relativamente all'accesso ad una ipotetica struttura dati condivisa, con l'inevitabile presenza di sezioni critiche e con un conseguente utilizzo di meccanismi di lock che sono assolutamente da evitare.

Si pensi ad esempio se tutti i worker per controllare lo stato di un flusso dovessero accedere ad una struttura dati condivisa, in cui si utilizzano dei meccanismi di lock per evitare l'accesso simultaneo alla risorsa. Fondamentalmente la consultazione della struttura dati è l'attività principale del worker, qualora vi fosse un lock sulla struttura, si

verificherebbe una sequenzializzazione degli accessi a tale struttura, che implica una sequenzializzazione dei worker, con una conseguente riduzione del parallelismo fisico fra i vari worker.

E' quindi fondamentale che ogni worker abbia al suo interno uno stato privato, in modo tale che solo lui possa accedervi, così che si possa evitare l'uso di meccanismi di lock.

Sia IP_s l'indirizzo IP di un terminale che vuole instaurare una connessione usando porta p_s, verso un terminale con indirizzo IP IP_d con porta p_d, ed essendo il flusso bidirezionale, i dati proventi da IP_s verso IP_d, che da verso IP_d IP_s, verranno mappati in un unico record di flusso identificato univocamente dalla quadrupla (IP_s, IP_d, p_s, p_d).

Chiamiamo $f \in Flow$ la quadrupla corrispondente a (IP_s, IP_d, p_s, p_d).

Avendo a disposizione il record del flusso di cui il pacchetto HTTP letto fa parte, abbiamo bisogno di mantenere un'associazione fra tale record e lo stato del di tale flusso. Questo può essere fatto utilizzando una struttura dati in algoritmica nota come dizionario.

Questa memorizza un insieme di elementi, e ne fornisce le operazioni di ricerca, inserimento e cancellazione.

Ciascun elemento che per i nostri fini è un record di flusso, $f = (IP_s, IP_d, p_s, p_d) \in Flow$ rappresenta una chiave di ricerca, mentre lo stato del flusso rappresenta l'informazione che vogliamo associare alla chiave.

$$s = (status, time, fin\ counter, ack\ of\ fin\ counter) \in State$$

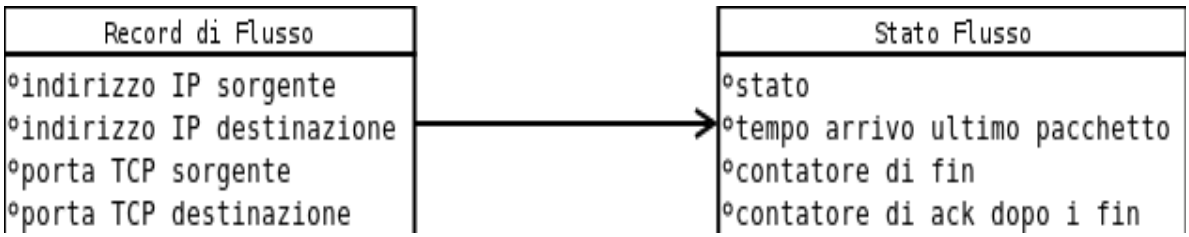


Figura 4.4 Funzione di mappatura

La Fig. 4.4 illustra i campi utilizzati per la realizzazione della funzione di mappatura $M : Flow \rightarrow State$. Quindi per ogni pacchetto p, è possibile immediatamente calcolare la relativa chiave di flusso $f \in Flow$, accedendo ai campi (IP_s, IP_d, p_s, p_d).

Una volta che abbiamo $f \in Flow$, si deve reperire lo stato del flusso, questo viene effettuato tramite $M(f) = s, s \in State$.

Le informazioni presenti nel record di flusso e nello stato del flusso sono autoesplicative,

tuttavia è necessario delineare lo scopo di utilizzo dei campi dello stato del flusso:

- tempo di arrivo ultimo pacchetto: necessario per le politiche di pulizia della struttura dati contenenti i record di flusso;
- contatore di fin: tiene il conteggio dei TCP fin flag, necessario per andare a controllare le terminazioni delle connessioni TCP;
- contatore di ack dopo i fin: tiene il conteggio dei TCP ack flag che sono inviati dopo la lettura dei primo fin flag, necessario per gestire le eventuali ritrasmissioni dopo che gli host intendono chiudere la connessione.

La struttura dati utilizzata per la realizzazione del dizionario è una map [CTS13] (la map della libreria stardard del linguaggio C++).

4.2.1 Gestione struttura dati

Come spiegato nel capitolo del livello di progettazione architetturale, la struttura dati privata ha bisogno di una gestione, nel senso che deve essere periodicamente “ripulita”, in modo tale da eliminare le informazioni “vecchie”, che non hanno più ragione di essere ancora memorizzate.

Gli eventi che fanno sì che un record contenga un'informazione non più utile ai fini delle computazioni, sono i seguenti:

- Connessione terminata: i due terminali hanno chiuso il canale di comunicazione TCP, per cui non c'è più nessun motivo, di mantenere il record di flusso relativo a tale connessione allocato all'interno della struttura dati.
- Inattività del flusso: i due terminali non comunicano più da molto tempo, nel progetto architetturale tale tempo è fissato a due minuti, ed anche in questo caso non c'è più nessun motivo, di mantenere il record di flusso relativo a tale connessione allocato all'interno della struttura dati.

Relativamente alla condizione di terminazione della connessione TCP, la realizzazione consiste appunto nel controllare il numero di pacchetti TCP con il bit di fin settato ad uno. Per considerare una connessione terminata, si controlla che entrambi i terminali lo abbiano inviato, controllando poi successivamente anche l'avvenuta lettura degli ack relativi ai fin.

Tuttavia non è possibile eliminare il record di flusso immediatamente al verificarsi della situazione sopra delineata, in quanto si potrebbero verificare alcune ritrasmissioni nella fase di chiusura della connessione od a connessione già terminata.

Nel caso di una ritrasmissione, se il relativo record di flusso fosse già stato eliminato precedentemente alla ritrasmissione stessa, si avrebbe un effetto negativo, in quanto, il flusso verrebbe ricreato, reinserito nella struttura, per poi non essere più utilizzato in

futuro e solo successivamente eliminato per inattività (il flusso è già terminato, il pacchetto letto è solo un ack successivo i segmenti di fin) con un evidente spreco di risorse.

Per questo motivo il record di flusso non viene eliminato immediatamente al verificarsi della terminazione di connessione. Il record verrà eliminato in una fase successiva, dalla procedura che gestisce anche il controllo dell'inattività dei flussi.

Eliminare un flusso per inattività, significa andare a controllare il tempo in cui è stato letto l'ultimo pacchetto relativo al flusso stesso, controllando che la differenza fra tale tempo ed il tempo in cui stiamo effettuando l'analisi, sia superiore al valore fissato per la definizione di inattività, che come detto precedentemente è rappresentato da 2 minuti.

$$isInactive(flow) = time(now) - time(flow.lastPacket) > val$$

Come possiamo vedere, è necessario mantenere all'interno del flusso il tempo dell'ultimo pacchetto ricevuto. Questo implica che per ogni pacchetto dobbiamo calcolare il tempo in cui esso arriva, ciò sarebbe alquanto oneroso, in quanto dovremo far uso di una chiamata di sistema per il calcolo del tempo di arrivo.

Tuttavia le schede Ethernet sono dotate di contatori hardware, in cui vengono registrati gli istanti di ricezione e di trasmissione, in questo modo non è più necessario pagare il costo della chiamata di sistema, in quanto l'informazione è già presente all'interno del pacchetto letto.

Leggendo i tempi dei pacchetti in arrivo sulla propria coda in ingresso il worker può calcolare direttamente il momento in cui esso deve eseguire la procedura di pulizia della struttura dati, senza l'utilizzo delle chiamate di sistema o segnali periodici preimpostati.

Fondamentalmente vi sono due metodi per la gestione della pulizia della struttura dati:

- cleaning periodico: in cui periodicamente, si controllano alcune parti della struttura dati eliminando le informazioni inutili;
- cleaning istantaneo: in cui per ogni buffer di pacchetti letto, si analizzano i flussi di dati “vicini” a quello/quelli presi in considerazione dall'analisi.

In questo tipo di applicazioni in-line, effettuare un cleaning periodico su grandi parti della struttura dati, rappresenta una criticità, in quanto può portare a momenti in cui il cleaning viene effettuato in momenti di massimo carico di traffico, con una possibile perdita di pacchetti dovuta ad overflow dei buffer di memorizzazione interni.

Per questo motivo si è adottata ed implementata una strategia basata su un “cleaning istantaneo”, dove per ogni buffer di pacchetti letto, si vanno a controllare e ripulire solo pochi record di flusso vicini ai pacchetti analizzati, pagando una piccola latenza aggiuntiva rispetto all'analisi in sé.

Quando si parla di record di flusso “vicini”, significa record presenti nella solita lista di

collisione del pacchetto precedentemente analizzato, in un'implementazione della mappa a liste di collisione, oppure alla visita di una piccola parte di un sotto-albero nel caso in cui la mappa sia realizzata ad albero binario di ricerca bilanciato.

Ovviamente l'attività di cleaning viene svolta dai singoli worker anche nei momenti in cui non hanno lavoro da effettuare, cioè quando le letture dal loro stream di ingresso non portano a nuovi buffer di pacchetti da dover analizzare.

E' importante sottolineare che la pulizia della struttura dati non può essere delegata ad un altro thread, in quanto emergerebbero dei problemi relativi alla sincronizzazione diretta fra il worker e il thread delegato al cleaning della struttura, con un inevitabile utilizzo di meccanismi di lock per la mutua esclusione all'accesso a sezioni critiche.

4.3 Bilanciamento di carico

Come precedentemente affermato abbiamo detto che ogni worker ha un proprio stato privato, a cui solo lui può accedere, è necessario quindi far sì che tutti i pacchetti di un determinato flusso di dati che vogliamo analizzare, siano sempre inviati allo stesso thread worker delegato a gestire quel flusso.

In caso contrario, se pacchetti relativi ad un determinato flusso fossero inviati a diversi worker, essi avrebbero solo delle informazioni parziali sul flusso, compromettendo la correttezza generale. E' quindi necessario definire una funzione hash, che mappi tutti i pacchetti appartenenti ad un determinato flusso sempre sul medesimo thread worker.

La funzione hash viene quindi utilizzata per calcolare l'identificatore (ID) del worker, che sarà incaricato ad analizzare tutta la comunicazione per uno determinato flusso di dati. Si vuole che tale funzione non dipenda solo dall'IP sorgente o solamente dall'IP destinazione del flusso, in quanto in questo modo tutto il traffico verso i server più noti e maggiormente contattati (come ad esempio tutto il traffico verso Google) verrebbe mappato solo su un determinato worker, con conseguenti problemi di bilanciamento di carico.

Per questa ragione si utilizza la funzione xor, che permette di fondere entrambe le informazioni di IP sorgente ed IP destinazione del flusso in un'unica informazione, in modo tale che anche i flussi verso i server più contattati possano essere mappati su worker diversi. Per ulteriori approfondimenti sulla bontà del metodo si veda [KNU98].

Sia p il pacchetto che viene letto, l'identificatore del thread worker, a cui verrà inviato il pacchetto, per la successiva analisi, è definito nel seguente modo:

$$\forall\, p \in IP,\ HashCode(p) = (p.IP_s \ xor\ p.IP_d) \bmod N_{worker}$$

Utilizzando soltanto $p.IP_s \ xor\ p.IP_d$ come definizione della funzione $HashCode(p)$, ed essendo un indirizzo IP implementato con un intero senza segno a 32 bit, potremmo generare un ID fra $[0, 2^{32}-1]$, ovviamente bisogna tenere in considerazione del numero

di worker (N_{worker}) a disposizione e quindi per una corretta implementazione, deve essere applicato l'operatore di modulo per essere sicuri di generare un ID compreso fra $[0, N_{worker}-1]$.

A basso livello le due operazioni possono essere eseguite velocemente in quanto sono realizzabili con uno XOR e una successiva divisione (non necessaria se N_{worker} fosse sempre una potenza di 2).

Come abbiamo spiegato nel capitolo della progettazione, il traffico non HTTP, non viene analizzato dall'applicazione e viene lasciato passare. Nonostante ciò tali pacchetti comunque attraverseranno la nostra applicazione (senza essere analizzati), per cui bisogna anche distribuire nel miglior modo tale tipo traffico fra i vari worker.

Questo traffico in generale potrebbe anche non trasportare traffico IP (si pensi ai pacchetti ARP), quindi non disponendo delle informazioni sugli indirizzi IP, non è possibile ovviamente ricorrere al metodo sopra spiegato per determinare l'ID del worker. Per questo motivo si presenta il problema di dover definire un'opportuna strategia alternativa con cui distribuire questo tipo di traffico.

L'unica informazione certa che abbiamo a disposizione è l'intestazione del frame Ethernet, in quando stiamo lavorando su reti con livello collegamento Ethernet.

Pertanto si è deciso di basare la definizione dell'ID, sugli indirizzi MAC di livello collegamento:

$$\forall\, p \notin IP,\ HashCode(p) = (MAC_s \ xor \ MAC_d) \bmod N_{worker}$$

Dove MAC_s rappresentano i 3 byte meno significati dell'indirizzo MAC sorgente, e MAC_d rappresentano i 3 byte meno significati dell'indirizzo MAC destinazione.

Sono stati scelti appunto i 3 byte meno significati, in quanto essi rappresentano un identificatore univoco all'interno del produttore, mentre i 3 byte più significativi identificano l'organizzazione che ha emesso l'identificatore.

Ovviamente basandosi sull'identificatore univoco per ogni produttore, avrebbe portato ad un cattivo bilanciamento qualora si avesse a che fare con schede di rete di un solo produttore.

In conclusione in condizioni normali di traffico, in cui quest'ultimo è generato da varie macchine, il metodo il metodo spiegato in questo paragrafo porta ad un buon bilanciamento di carico. Tuttavia c'è da tenere in considerazione anche l'analisi al caso pessimo, in cui in un caso estremamente improbabile, qualora tutto il traffico di rete sia generato da comunicazioni HTTP fra due soli terminali o più in generale da terminali che vengono mappati sullo stesso thread worker, si ha ovviamente un cattivo bilanciamento.

4.4 Buffering

Nel pattern parallelo descritto, i thread comunicano grazie a delle code SPSC su cui vengono effettuate di operazioni di push e pop. Queste operazioni rappresentano un overhead, cioè un latenza aggiuntiva che deve essere pagata per realizzare la comunicazione, ma che in realtà non rappresenta il codice funzionale dell'applicazione.

4.4.1 Latenza di comunicazione vs latenza funzionale: modello teorico

Consideriamo l'attività sequenziale del singolo worker, il suo tempo di servizio può essere espresso tenendo in considerazione anche la latenza di comunicazione fra le varie unità di esecuzione parallele:

$$T_w = L_{(read\ data)} + L_{(seq\ worker)} + L_{(write\ data)}$$

dove:

- $L_{(read\ data)}$ è la latenza per la lettura dei dati dalla coda dello stream di ingresso;
- $L_{(seq\ worker)}$ è la latenza relativa al codice funzionale che vogliamo eseguire;
- $L_{(write\ data)}$ è la latenza relativa alla scrittura dei dati sulla coda dello stream di uscita.

$L_{(read\ data)}$ e $L_{(write\ data)}$ rappresentano un overhead necessario per il supporto della comunicazione fra le varie entità computazionali ed è quindi un aspetto che bisogna tener in considerazione soprattutto quando siamo di fronte a computazioni di grana molto fine come quella in questione.

Infatti qualora il tempo di overhead di comunicazione fra le varie unità computazionali sia comparabile o simile alla latenza $L_{(seq\ worker)}$ del business code del worker possono sorgere alcuni seri problemi di speedup.

Infatti andiamo ad analizzare il comportamento worker facente parte di una farm con $N_{(worker)}$ worker.

Si ha che il tempo di servizio risulta essere: $T_s = Max(T_{emitter}, T_{worker}, T_{collector})$.

Per ipotesi supponiamo $T_{emitter} < T_{worker}$ e $T_{collector} < T_{worker}$ in quanto vogliamo analizzare il tutto in funzione della latenza del worker.

Quindi otteniamo che:

$$T_s = Max(T_{emitter}, T_{worker}, T_{collector}) = T_{worker} = \frac{L_{(read\ data)} + L_{(seq\ worker)} + L_{(write\ data)}}{N_{worker}}$$

Andiamo ora ad analizzare il relativo speedup:

$$Speedup(N_{worker}) = \frac{T_{seq}}{T_{par}(n)} = \frac{T_{seq}}{\left(\frac{L_{(read\,data)} + L_{(seq\,worker)} + L_{(write\,data)}}{N_{worker}}\right)} = T_{seq}\frac{N_{worker}}{L_{(read\,data)} + L_{(seq\,worker)} + L_{(write\,data)}}$$

$$Speedup(N_{worker}) = \frac{T_{seq}}{L_{(read\,data)} + L_{(seq\,worker)} + L_{(write\,data)}} N_{worker}$$

Dove l'asintoto di Speedup è dato da $Speedup(n) = n$.

Ovviamente quando ragioniamo di moduli sequenziali come il codice eseguito dal singolo worker, si ha che la latenza ed il tempo di servizio sono eguali, anche se esprimono concettualmente due cose diverse, quindi $T_{(seq\,worker)} = L_{(seq\,worker)}$.

Inoltre $T_{(seq)}$ è sempre minore di $T_{(seq\,worker)}$ per ipotesi, essendo $T_{(seq)}$ il miglior tempo di esecuzione sequenziale. Supponiamo inoltre per facilità di lettura che le operazioni di lettura e scrittura abbiamo la stessa latenza quindi $L_{(read\,data)} = L_{(write\,data)}$, che chiamiamo $L_{(com)}$, tale ipotesi non invalida il ragionamento in quanto anche se $L_{(read\,data)}$ fosse diverso da $L_{(write\,data)}$ ci condurrebbe alle medesime conclusioni.

Incapsuliamo $T_{(seq)}$ all'interno del codice sequenziale worker facendo sì che $T_{(seq)} \approx T_{(seq\,worker)}$.

Costruiamo la funzione $S(x)$ che ci dà l'idea di quanto il rapporto fra latenza di comunicazione e la latenza del codice funzionale sia importante per il raggiungimento di un buon speedup.

$$S(x) = \frac{x}{x + 2L_{com}} N_{worker}$$

Andiamo a studiare la funzione $S(x)$, dove $L_{(com)}$ è una costante maggiore di zero:

- Abbiamo che per $x \to 0$ (lato destro), nella pratica possiamo immaginare che x sia più piccolo di $L_{(com)}$, abbiamo che il fattore $\frac{x}{x + 2L_{com}} \to 0$, per cui tutta la funzione $S(x) = \frac{x}{x + 2L_{com}} N_{worker} \to 0$.

  Con una conseguente degradazione dello speedup.

- Abbiamo che per $x \to +\infty$,

il fattore $\frac{x}{x+2L_{com}} \to 1$ e quindi $S(x) = \frac{x}{x+2L_{com}} N_{worker} \to N_{worker}$.

Nel caso pratico si ha quando $x \gg L_{com}$, cioè quando la latenza del codice funzionale è nettamente preponderante rispetto alla latenza di comunicazione, allora lo speedup può raggiunge il valore asintotico ideale. Infatti maggiormente $T_{(seq)} \approx T_{(seq\,worker)}$, maggiormente lo speedup sarà vicino al suo valore ideale.

Qualora l'indice di speedup non fosse buono, bisogna necessariamente aumentare la latenza del codice funzionale del worker. Bisogna dunque aumentare la grana della logica del worker, in modo tale da far diventare preponderante il peso di $L_{(seq\,worker)}$ rispetto a $L_{(com)}$, cioè $L_{(seq\,worker)} \gg L_{(read\,data)}, L_{(seq\,worker)} \gg L_{(write\,data)}$.

Ottenendo:

$$Speedup(N_{worker}) = \frac{T_{seq}}{T_{par}(n)} = T_{seq} \frac{N_{worker}}{L_{(read\,data)} + L_{(seq\,worker)} + L_{(write\,data)}} = \frac{T_{seq}}{T_{(seq\,worker)}} N_{worker}$$

In questo caso il fattore $\frac{T_{seq}}{T_{(seq\,worker)}}$, è sempre minore di uno per ipotesi, ma migliorando l'implementazione parallela, è possibile far sì che $T_{(seq\,worker)} \approx T_{(seq)}$, e possibilmente raggiungere lo speedup ideale (N_{worker} in questo caso).

Si è quindi svolto un lavoro sperimentale volto alla ricerca del limite inferiore del tempo $T_{(seq\,worker)}$, per cui l'applicazione parallela avesse un speedup e scalabilità ragionevoli. Ovviamente tali test dipendono specificatamente dalle macchine su cui sono stati eseguiti, in quanto al variare della macchina utilizzata cambiano tutti i parametri dei tempi di servizio e di latenza utilizzati nei precedenti ragionamenti.

Una volta trovato il limite inferiore per lo speedup dell'applicativo, questo valore deve essere utilizzato nello sviluppo del lavoro oggetto della tesi, appunto come il tempo minimo di lavoro, inteso come attività computazionale, che il worker deve eseguire affinché l'applicazione possieda appunto uno speedup ragionevole.

Per problemi su stream una delle possibilità per ovviare a tali problematiche e quindi aumentare la grana del codice del worker, è quello di analizzare un blocco di dati anziché un singolo dato alla volta. Dalla coda in ingresso vengono in qualunque caso letti dei puntatori, è quindi possibile che vengano passati puntatori ad array di pacchetti (anziché puntatori ai singoli pacchetti).

In questo modo l'emitter legge i dati da rete, li memorizza in un buffer (uno per ogni worker) e una volta che il buffer è pieno verrà inviato al relativo worker. Si può notare come nelle condizioni di massimo carico, ciò incrementa la latenza totale del singolo

pacchetto all'interno dell'applicazione, tuttavia migliora il tempo di servizio del pattern parallelo e conseguentemente il tempo di completamento su un determinato dataset sperimentale.

Ovviamente in caso di basso carico, i pacchetti non aspettano indefinitamente che il buffer diventi pieno, vi sono appunto dei timer sulle letture da scheda che passato un determinato numero di microsecondi, ritornano dalla lettura con codice di errore e l'emitter invia il buffer al worker, anche se parzialmente riempito.

Si consideri l'applicazione di firewall come un sistema stand alone. Questo sistema visto dall'esterno non garantisce la realizzazione del metodo FIFO, pertanto non è vero che il primo pacchetto ad entrare è il primo pacchetto ad uscire. Infatti anche se tutte le code di comunicazione sono FIFO il pattern parallelo non garantisce la medesima proprietà. Tuttavia il sistema sempre da una prospettiva esterna, garantisce la proprietà FIFO per ogni flusso di dati, quindi per ogni connessione dati il sistema garantisce che l'host destinazione riceva i pacchetti nel solito ordine con cui sono stati spediti.

L'aspetto relativo al buffering verrà ulteriormente portato avanti nel capito dei risultati sperimentali.

4.5 Thread pinning

In un sistema multi-processore/multi-core, il sistema operativo distribuisce i diversi thread sulle varie CPU/core disponibili.

Quando il sistema operativo sospende l'esecuzione di un thread, l'algoritmo di scheduling in generale ha la possibilità di allocare il thread su uno dei core a disposizione del sistema, indipendentemente da dove si trovava in esecuzione precedentemente.

Tuttavia esistono una serie di motivi, nella maggior parte legati alle prestazioni per i quali vorremmo evitare questo fenomeno.

In primo luogo questo fenomeno comporta che tutte le volte che il thread viene schedulato su un core diverso dal core precedente su cui era in esecuzione, la cache deve essere di nuovo portata a regime, pagando come prezzo le latenze dei cache miss. Se invece il thread fosse schedulato sul core su cui era precedente in esecuzione, c'è una buona possibilità che parte della cache, mantenga ancora i dati relativi alla computazione di tale thread. In quest'ultimo caso si avrebbe un numero di cache miss inferiore al caso precedente, di conseguenza a si migliorerebbero in modo significativo le prestazioni.

In secondo luogo è possibile essere nella situazione in cui un thread è delegato ad un lavoro fondamentale nel sistema. Per questa ragione si vorrebbe poter allocare tale thread su un core dedicato alla sua esecuzione e si vorrebbe impedire ad ogni altro thread di poter essere schedulato su tale core.

Il *Thread Pinning* consiste nel mappare un determinato thread su un determinato core o in più in generale su un determinato insieme di core, modificando il comportamento originario dell'algoritmo di scheduling utilizzato sulla macchina in questione.

Il kernel Linux fornisce un insieme di funzioni per consentire il thread pinning tramite la definizione di un “affinity set” per ogni thread. Lo scheduler allocherà quindi il thread solo su uno dei core appartenenti all'affinity set.

Nel caso dell'applicazione in questione, si è deciso che ogni thread venga mappato su un determinato core, con un rapporto 1:1, in modo tale che ogni thread possa sfruttare totalmente il core su cui è allocato.

Nel caso in cui i core fisici siano inferiori al numero di thread che si vogliono allocare all'avvio del programma, più thread verranno mappati su un singolo core, con un conseguente interleaving, quindi parallelismo virtuale, fra i thread mappati sul medesimo core.

4.6 Lettura dei dati e parsing

L'applicazione è stata realizzata in modo tale che il codice che si interfaccia con la scheda di rete per la lettura e scrittura, sia stato scritto in un modulo a parte, in modo tale da svincolare la logica applicativa dalla tecnologia con cui ci si interfaccia con la scheda di rete. In questo modo se si volesse cambiare la libreria con cui interfacciarsi con la scheda di rete non è necessario effettuare nessuna modifica sugli altri moduli software.

Come spiegato precedentemente la libreria scelta per la cattura è stata PCAP. La libreria offre le funzioni per la lettura dei pacchetti a basso livello, che restituiscono: i puntatori ai dati letti, il numero dei byte letti ed alcune informazioni di supporto.

Il parsing del pacchetto in sé è un'attività poco costosa, in quanto si deve gestire solo il protocollo HTTP. Il parsing è effettuato per una piccola parte dall'emitter, mentre la parte di calcolo più pesante in termini di latenza è effettuata dal worker.

L'emitter necessita dell'identificatore del worker a cui deve essere inviato il pacchetto, per cui deve conoscere gli indirizzi IP di livello rete se il frame porta al suo interno un datagramma IP, altrimenti deve accedere agli indirizzi MAC di livello collegamento.

Queste due operazioni sono molto semplici e poco costose in quanto è possibile saltare nel pacchetto ad offset già predefiniti a tempo di compilazione, per andare a leggere questi determinati campi.

La parte più pesante del parsing è invece a carico del worker, che deve controllare se il pacchetto in analisi è HTTP, questo comporta le seguenti operazioni:

1. controllare che il frame porti al suo interno un datagramma IP;
2. saltare nel datagramma IP, accedere al campo “upper layer protocol”,

controllando che trasporti al suo interno un segmento TCP. Infine è necessario calcolare la lunghezza dell'intestazione del datagramma IP. L'intestazione IP può essere variabile a seconda se i flag opzionali sono usati. La lunghezza in byte dell'intestazione IP è necessaria in quanto bisogna successivamente calcolare correttamente la base del segmento TCP.

3. saltare nel segmento TCP, controllando che la porta sorgente o la porta destinazione identifichino una comunicazione HTTP. Infine è necessario calcolare la lunghezza dell'intestazione del segmento TCP, anch'essa può essere variabile a seconda se i flag opzionali sono utilizzati. Tale calcolo serve per calcolare correttamente la base del campo dati portato all'interno del segmento TCP.
4. saltare nel payload del segmento TCP, leggendo quindi i dati a livello applicativo. Se non è presente nessuna richiesta HTTP, il parsing del pacchetto è terminato in quanto altre informazioni non sono necessarie.

   Altrimenti qualora nel payload vi fosse una richiesta HTTP è necessario andare a controllare l'header *Host*, che appunto rappresenta il nome di dominio del server ed il numero di porta (che è opzionale). L'header Host è obbligatorio a partire da HTTP/1.1.

   Esempi di header Host nelle richieste HTTP, sono: “Host: en.wikipedia.org:80” oppure “Host: en.wikipedia.org”.

Una volta che il worker ha completato il parsing del pacchetto, questo può essere analizzato dal worker stesso per la successiva marcatura.

4.7 Gestione della memoria

Uno degli aspetti cruciali da dover gestire durante la fase di implementazione, è stato senza ombra di dubbio, la gestione dell'allocazione della memoria per la memorizzazione dei dati che i vari thread si scambiano, quali: i buffer di pacchetti, i pacchetti, con le varie informazioni di supporto e l'area dati dei pacchetti stessi.

L'utilizzo dei meccanismi new/malloc presenti in C++ per la richiesta di memoria sullo heap rappresenterebbe il modo più intuitivo per la risoluzione del problema, in cui il thread emitter alloca le aree per i relativi buffer, i pacchetti stessi, ecc. ed il collector effettua la liberazione delle aree precedentemente allocate dall'emitter, con una continua allocazione e deallocazione di zone sulla memoria heap durante l'esecuzione dell'applicativo.

L'allocazione dinamica della memoria ha un ruolo fondamentale in termini di prestazioni, infatti l'utilizzo della funzione malloc, o dell'operatore new, nasconde al suo interno dei meccanismi di lock, utilizzati per l'accesso alla struttura della gestione dello heap. Pertanto reintrodurremmo implicitamente delle sezioni critiche all'interno del codice dei

vari thread, cosa da evitare assolutamente come già spiegato precedentemente.

FastFlow mette a disposizione vari tipi di allocator, specialmente ottimizzati per l'allocazione di piccole strutture dati, appunto con lo scopo di permettere l'allocazione dinamica della memoria sullo heap. Gli allocator offerti da FastFlow in alcuni casi hanno delle prestazioni migliori rispetto a quelli forniti dalla libc-6 standard. Per approfondimenti sulla comparazione fra gli allocator FastFlow e quelli libc-6 standard si veda [ADK11b].

FastFlow mette a disposizione:

- ff_allocator: basato sull'idea di Slab Allocator, per ulteriori dettagli si veda [BON94], in cui solo un thread può effettuare delle operazioni di malloc mentre tutti i thread possono effettuare le operazioni di free;
- FFAllocator: basato su ff_allocator, permette ad un qualsiasi thread di effettuare sia operazioni di malloc che di free.

Tuttavia con un'accurata analisi della struttura di comunicazione fra i vari thread, si può notare come ogni emitter che legge da una determinata scheda di rete, possa inviare i suoi buffer di pacchetti indirettamente soltanto verso un ben specifico collector (ovviamente passando il buffer prima ad un worker, il quale lo invierà al collector specifico definito all'interno del buffer stesso) ed infine il collector potrebbe restituire buffer all'emitter che lo aveva inviato. Questa cosa si profila come un sistema chiuso, in cui non è più necessario allocare e liberare continuamente memoria dalla zona heap, ma bensì riutilizzare la memoria in modo ciclico.

In questo modo è possibile allocare un ulteriore canale di comunicazione, più nello specifico dato il rapporto di comunicazione 1:1, una coda SPSC che collega il collector con l'emitter (similmente come avviene in FastFlow con la farm con feedback).

E' quindi possibile allocare prima dell'avvio della struttura parallela dell'applicazione, un'area di memoria heap sufficientemente grande per preparare i buffer di pacchetti, i pacchetti stessi e le aree dati dei pacchetti, queste ultime fissate alla massima MTU Ethernet più l'intestazione stessa del protocollo a livello collegamento .

Il collector tutte le volte che processa un buffer di pacchetti, lo inserisce nella coda SPSC che lo collega all'emitter, il quale aveva precedentemente inviato il pacchetto.

Per quanto riguarda l'emitter tutte le volte che necessita di un nuovo buffer di pacchetti (anche i byte necessari per i pacchetti e l'area dati del pacchetto sono già contenute nel buffer) non fa altro che estrarre un puntatore al buffer di pacchetti dalla coda SPSC che lo collega al relativo collector.

In conclusione, adottando questa strategia non si devono più allocare e liberare aree di memoria heap durante l'esecuzione, dovendo pagare il costo di tali operazioni che intrinsecamente reintroducono il problema dei lock, che era stato evitato grazie alla

progettazione della parte parallela in modo oculato.

Utilizzando quindi i canali aggiuntivi di feedback come si può notare in Fig 4.5, si paga soltanto un'operazione di inserimento (nel collector) e di estrazione (nell'emitter) da code SPSC, che rappresentano un costo nettamente inferiore rispetto agli operatori new/delete, alle funzioni malloc/free ed ai FastFlow allocator.

4.8 Realizzazione del modello teorico delle prestazioni

Andiamo ora ad analizzare il tempo di servizio T_s e successivamente la latenza L del pattern parallelo scelto per la realizzazione dell'applicazione.

Riferendoci al modello teorico delle prestazioni, possiamo vedere tale pattern come una pipeline di tre stadi. Sapendo che la pipeline con in generale con n stadi in cui ciascun stadio ha un tempo di servizio T_i , abbiamo che il tempo di servizio della pipeline è dato dalla seguente formula:

$$T_s = Max(T_{1,} T_{2,..}, T_n)$$

Pertanto nel nostro caso T_s è dato dal $Max(T_{emitter}, T_{worker}, T_{collector})$.

dove:

$$T_{emitter} = \frac{T_{(seq\ emitter)}}{N_{emitter}} \quad T_{worker} = \frac{T_{(seq\ worker)}}{N_{worker}} \quad T_{collector} = \frac{T_{(seq\ collector)}}{N_{collector}}$$

In questa analisi $N_{emitter}$ e $N_{collector}$ sono da considerasi:

- fissati, in quanto il numero di schede di rete è un dato fissato all'avvio dell'applicazione;
- eguali, in quanto viene allocato un thread emitter in lettura su ogni diversa scheda di rete ed un thread collector in scrittura su ogni diversa scheda di rete.

Quindi possono essere riscritti, per maggiore facilità di lettura come:

$$T_{emitter} = \frac{T_{(seq\ emitter)}}{N_{emitter}}$$

$$T_{collector} = \frac{T_{(seq\ collector)}}{N_{collector}}$$

$$T_S = Max(T_{emitter}, T_{worker}, T_{collector}) = Max(T_{emitter}, \frac{T_{(seq\,worker)}}{N_{worker}}, T_{collector})$$

Ogni emitter come spiegato precedentemente effettua una lettura dei dati dalla scheda di rete, calcola l'hash del pacchetto per stabilire a quale worker esso debba essere inoltrato ed inserisce il pacchetto nel buffer. Infine quando il buffer è pieno, invia buffer di pacchetti sul proprio stream di uscita (che lo collega al worker).

Sia k la lunghezza del buffer di pacchetti, abbiamo quindi:

$$T_{(seq\,emitter)} = k\,L_{read} + k\,L_{hash} + k\,L_{(buffering\ handling)} + L_{(output\ ch\ push)} + L_{(get\ new\ buf)}$$

dove:

- L_{read} : latenza della lettura da scheda di rete del singolo pacchetto;
- L_{hash} : latenza calcolo della funzione hash per singolo pacchetto per stabilire l'ID del worker;
- $L_{(buffering\ handling)}$: latenza per l'inserimento di un pacchetto nel buffer di un determinato worker. Inoltre;
- $L_{(output\ ch\ push)}$: latenza di comunicazione, data da una push su una coda SPSC, per inserire il puntatore del buffer nel corretto input stream del worker;
- $L_{(get\ new\ buf)}$: la latenza relativa all'attività di estrazione di un nuovo buffer dalla coda dei buffer disponibili, quest'ultima cosa costa un'operazione di push su una coda SPSC.

Ogni worker legge dal proprio input stream un buffer di pacchetti, li parsa uno ad uno e per ogni pacchetto HTTP accede alla mappa che contiene i record di flusso privati al worker per leggere lo stato (eventualmente modificandolo), marca il pacchetto a seconda a seconda se è un pacchetto che può passare o meno, ed infine invia il buffer di pacchetti marcati al collector.

Sia k la lunghezza del buffer di pacchetti ed $h \leq k$ il numero di pacchetti HTTP presenti nel buffer, si ha quindi:

$$T_{(seq\,worker)} = L_{(input\ ch\ pop)} + k\,L_{parsing} + h\,L_{(map\ get)} + k\,L_{marking} + L_{(output\ ch\ push)}$$

dove:

- $L_{(input\ ch\ pop)}$: latenza di comunicazione, data da una pop su una coda SPSC, per leggere il puntatore del buffer nel dall'input stream del worker;
- $L_{parsing}$: latenza calcolo per il parsing di ogni pacchetto contenuto nel buffer;

- $L_{(map\ get)}$: latenza per l'accesso alla mappa per l'ottenimento dello stato del flusso ed eventuale modifica di ogni pacchetto contenuto nel buffer;
- $L_{(marking)}$: latenza per la marcatura di ogni pacchetto contenuto nel buffer;
- $L_{(output\ ch\ push)}$: latenza di comunicazione, data da una push su una coda SPSC, per inserire il puntatore del buffer nel corretto input stream del collector.

Ogni collector legge dal proprio input stream un buffer di pacchetti, ed a seconda di come sono stati marcati li scrive o meno sulla scheda di rete gestita.

Sia k la lunghezza del buffer di pacchetti ed $j \leq k$ il numero di pacchetti marcati positivamente dal generico worker, si ha che:

$$T_{(seq\ collector)} = L_{(input\ ch\ pop)} + j\ L_{write} + L_{(buffering\ handling)}$$

dove:

- $L_{(input\ ch\ pop)}$: latenza di comunicazione, data da una pop su una coda SPSC, per leggere il puntatore del buffer nel dall'input stream del collector;
- L_{write} : latenza della scrittura su scheda di rete di un singolo pacchetto contenuto nel buffer e positivamente marcato;
- $L_{(buffering\ handling)}$: latenza per la restituzione del puntatore del buffer di pacchetti all'emitter che li aveva precedentemente letti, data da una push su una coda SPSC.

Una volta stabiliti, i vari tempi di servizio e latenze di ogni modulo interno, vogliamo ora trovare, il numero ottimo worker N_{worker} , da allocare.

Sappiamo che T_s è dato dal

$$Max(T_{emitter}, T_{worker}, T_{collector}) = Max(T_{emitter}, \frac{T_{(seq\ worker)}}{N_{worker}}, T_{collector})$$

ed il solo termine che da cui dipende tale espressione è N_{worker} , quindi .

Pertanto avremo il minimo tempo di servizio quando $\frac{T_{(seq\ worker)}}{N_{worker}}$ non sarà il massimo, cioè non sarà un collo di bottiglia.

Chiamiamo ora $T_{ref} = Max(T_{emitter}, T_{collector})$ quindi il massimo fra il tempo di servizio degli emitter e dei collector, vorremmo trovare quel numero N_{worker} tale che:

$T_{ref} = \frac{T_{(seq\ worker)}}{N_{worker}}$ sostituendo e risolvendo in N_{worker} , abbiamo:

$$Max(T_{emitter}, T_{collector}) = \frac{T_{(seq\,worker)}}{N_{worker}}$$

$$N_{worker} = \frac{T_{(seq\,worker)}}{Max(T_{emitter}, T_{collector})}$$

Tale risultato arrotondato per eccesso all'intero successivo ci fornisce il numero di worker da utilizzare per far sì che il tempo di servizio della parte di computazione effettuata dai worker, non rappresenti un collo di bottiglia per il tempo di servizio del pattern nel suo globale.

Come sopra spiegato abbiamo detto che il tempo di servizio T_S è rappresentato da:

$$Max(T_{emitter}, \frac{T_{(seq\,worker)}}{N_{worker}}, T_{collector})$$

Questo risultato è valido, tuttavia solo nelle condizioni di bilanciamento di carico ideale fra i vari worker.

Come ben delineato precedentemente un determinato flusso può essere gestito da uno e uno solo ben preciso worker, cioè quello competente per l'analisi di tale record di flusso. In un caso estremamente improbabile dove tutto il traffico gestito dall'applicazione sia diretto ad un solo worker (quindi tutti gli altri worker non effettuano computazioni), la struttura parallela dell'applicazione non è altro che una pipeline, con un tempo di servizio stimabile in:

$$T_S = Max(T_{emitter}, T_{seq\,worker}, T_{collector})$$

con una conseguente degradazione delle prestazioni relative al tempo di servizio, senza però variare il alcun modo la latenza del pacchetto all'interno della struttura parallela.

Inoltre si vuole puntualizzare che un utilizzo di un'unica struttura dati per il mantenimento dello stato, a cui tutti i worker possono accedere non sia una buona soluzione. Questo è dettato dal fatto che saremmo costretti ad introdurre dei lock per garantire l'accesso mutualmente esclusivo alle sezioni critiche con una intrinseca sequenzializzazione degli accessi dei vari worker alla struttura dati condivisa, che porterebbe ad un ovvio peggioramento della latenza di accesso alla map e conseguentemente un peggioramento dello speedup.

Potendo vedere il pattern parallelo utilizzato come una pipeline con tre stadi come detto

precedentemente, e sapendo che la pipeline con in generale con n stadi in cui ciascun stadio ha una latenza L_i , abbiamo che la latenza totale è data dalla seguente formula:

$$L = \sum_{i=1}^{n} L_i$$

Pertanto la latenza di un pacchetto all'interno dell'applicativo può essere definita secondo la seguente formula:

$$L = L_{emitter} + L_{worker} + L_{collector} \text{ dove:}$$

$$L_{emitter} = T_{(seq\,emitter)} = k\,L_{read} + k\,L_{hash} + k\,L_{(buffering\,handling)} + L_{(output\,ch\,push)} + L_{(get\,new\,buf)}$$

$$L_{worker} = T_{(seq\,worker)} = L_{(input\,ch\,pop)} + k\,L_{parsing} + h\,L_{(map\,get)} + k\,L_{marking} + L_{(output\,ch\,push)}$$

$$L_{collector} = T_{(seq\,collector)} = L_{(input\,ch\,pop)} + j\,L_{write} + L_{(buffering\,handling)}$$

Nelle reti di calcolatori le prestazioni sono influenzate dai ritardi esistenti, più nello specifico si ha:

- il ritardo di elaborazione: il tempo richiesto per esaminare l'intestazione del pacchetto e per determinare dove inoltrarlo, solitamente questo ritardo è dell'ordine dei microsecondi ed è trascurabile rispetto agli altri tipi di ritardo, a meno che il router non stia effettuando cifratura dei datagrammi come per le VPN;
- il ritardo di accodamento: il tempo che il pacchetto attende la trasmissione sul collegamento, questo tipo di ritardo può essere dell'ordine dei microsecondi o dei millisecondi a seconda dell'intensità del traffico; il jitter, cioè la variazione statistica nel ritardo dei pacchetti, causata dalle code interne ai router congestionati, è prevalentemente determinata da questo tipo di ritardo;
- il ritardo di trasmissione : il tempo richiesto per trasmettere tutti i bit del pacchetto sul canale di collegamento, solitamente questo tipo di ritardo è trascurabile quando la banda è superiore ai 10 Mbit al secondo, diventa un ritardo predominante per collegamenti dial-up a 56 Kbit al secondo;
- il ritardo di propagazione: il tempo richiesto per propagare i bit da una estremità all'altra del collegamento, che dipende dal mezzo fisico (fibra ottica, doppino in rame, ecc.). Nelle reti più estese sono dell'ordine dei millisecondi.

E' possibile considerare la latenza totale di un pacchetto nell'applicazione, come ulteriore un ritardo di elaborazione nelle reti di calcolatori.

Capitolo 5

“Multi-Emitter Farm” come nuovo pattern parallelo

Come si è visto nei capitoli precedenti, il pattern parallelo realizzato nell'applicazione che è stato chiamato “multi-emitter farm”, si configura come una nuova versione di farm, in cui vi sono più emitter, più worker e più collector.

Abbiamo visto come la replicazione degli emettitori sia un aspetto cruciale da tenere in considerazione per il nostro problema. Tuttavia questa struttura parallela non è una soluzione ad hoc solo per il nostro problema, anzi risulta essere un pattern molto utile in vari ambiti d'applicazione.

Ad esempio tutte le volte che l'emettitore rappresenta un collo di bottiglia per il pattern parallelo che stiamo utilizzando, vengono impiegate delle strategie per rimuovere tale collo di bottiglia, come ad esempio:

- parallelizzazione dell'emitter tramite pipeline: l'emitter viene parallelizzato tramite l'utilizzo di una pipeline, in cui l'emitter non è più rappresentato da una singola entità di esecuzione parallela, ma è composto da un insieme di entità di esecuzione parallela, una per ogni stadio della pipeline. Supponiamo che l'emitter che vogliamo parallelizzare abbia un tempo di servizio pari a T, ed essendo la parallelizzazione legata al numero n di stadi con cui definiamo la pipeline, il tempo di servizio ottenuto con questa parallelizzazione è dato dal $Max(T_i), 1 \leq i \leq n$.

  Questo risultato non risulta essere del tutto buono vari motivi. In generale non è detto che sia possibile suddividere il codice dell'emitter in vari moduli paralleli. In secondo luogo qualora riuscissimo a suddividere il codice fra i vari moduli paralleli, è necessario bilanciare la pipeline per ottenere una significativa riduzione del tempo di servizio, cosa in generale non del tutto banale. Nel caso ottimo quindi difficilmente realizzabile, quindi qualora riuscissimo a suddividere il codice dell'emitter in n stadi di un pipeline, e qualora riuscissimo a bilanciare perfettamente la pipeline, per cui $T_i = T_j, 1 \leq i \leq n, 1 \leq j \leq n$ avremo un tempo di servizio pari a $\frac{T}{n}$.

- parallelizzazione dell'emitter tramite farm: l'emitter viene parallelizzato tramite l'utilizzo di una farm, in cui l'emitter non è più formato da una singola entità di esecuzione parallela, ma è modellato da una farm, in cui l'emitter della farm non dovrà far nessun calcolo, i worker della farm, incapsuleranno il codice sequenziale dell'emitter che volgiamo replicare ed il collector non deve far altro

che prendere i risultati generati dai worker della farm ed inviarli sul suo stream di uscita secondo una determinata politica implementata dallo stesso. Supponiamo che l'emitter che vogliamo parallelizzare abbia un tempo di servizio pari a T, ed essendo la parallelizzazione legata al numero n di worker con cui definiamo la farm, il tempo di servizio ottenuto con questa parallelizzazione è dato:

$$Max(0, \frac{T}{n}, T_{collector})$$

All'aumentare di n, in termini asintotici abbiamo che:

$$\lim_{n \to +\infty} Max(0, \frac{T_{seq}}{n}, T_{collector})$$

$$\lim_{n \to +\infty} Max(0, \frac{T}{n}, T_{collector}) = Max(0, \lim_{n \to +\infty} \frac{T}{n}, T_{collector})$$

$$Max(0, \lim_{n \to +\infty} \frac{T}{n}, T_{collector}) = Max(0, \frac{T}{+\infty}, T_{collector}) = T_{collector}$$

Quindi possiamo dedurre che il tempo di servizio è limitato inferiormente da $T_{collector}$.

Ricapitolando questa soluzione garantisce un tempo di servizio pari a $\frac{T}{n}$, supponendo ovviamente che $T_{collector}$ non sia un collo di bottiglia.

Tuttavia questa soluzione soffre di scarsa scalabilità all'aumentare del numero di replicazioni degli emitter (quindi all'aumentare dei worker della farm), come abbiamo visto il tempo di servizio è inferiormente limitato da $T_{collector}$. Inoltre c'è da considerare che per realizzare questa soluzione abbiamo bisogno di impiegare due unità di esecuzione aggiuntive rispetto al puro codice funzionale, quali l'emitter (anche se non effettua calcoli deve essere comunque allocato) ed il collector (che invia i risultati ricevuti sul proprio stream di uscita).

Introduciamo ora un paradigma che ci permette di ovviare a tutte queste limitazioni che abbiamo finora riscontrato.

5.1 Paradigma

Anche questa variante di farm, fa parte degli skeleton che operano su stream, e necessita solo della definizione della funzione f che si vuole applicare agli elementi dello stream in ingresso.

Sia quindi:

$f : D \rightarrow C$ una funzione pura e siano $x_{1,} x_{2,} \ldots, x_m$ gli elementi dello stream di ingresso

dove $x_i \in D \; per \; i \in [1, m]$

La farm applica f ad ogni elemento x_i e lo invia sul proprio stream di uscita, in cui ciascun elemento $f(x_i)$ viene calcolato all'interno di ogni worker, in modo indipendente ed in parallelo rispetto agli altri elementi.

$$f(x_i) = y_i \quad \text{per} \quad i \in [1, m]$$

dove $y_i \in C \; per \; i \in [1, m]$

Lo stream in uscita risulta essere quindi così definito:

$$f(x_1), f(x_2), \ldots, f(x_m)$$

Si può infatti notare come a regime questa struttura parallela riesce a computare n $f(x_i)$ negli n worker diversi.

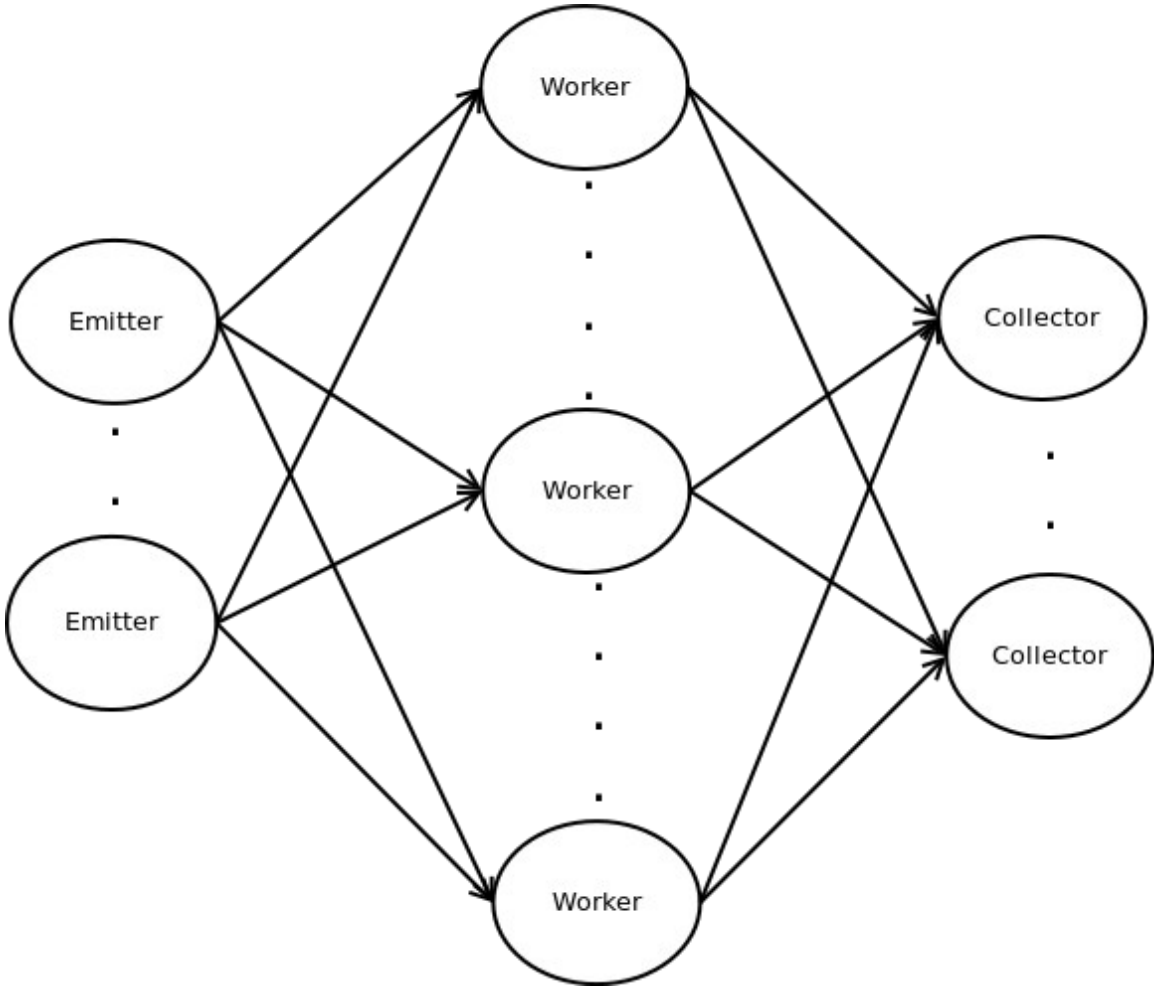


Figura 5.1 lo skeleton “multi-emitter farm”

La Fig. 5.1 mostra la struttura dello skeleton “multi-emitter farm”, in cui:

- Ogni $emitter_k$ ha il compito di leggere in ingresso gli elementi $x_{1,}x_2 \ldots, x_{(n_k)}$ e di schedularli verso i worker, con la possibilità dell'uso di varie politiche per una corretta gestione del bilanciamento di carico dei vari worker, dove $\sum_{k=1}^{N_{emitter}} n_k = m$;
- Il generico worker ha il compito di applicare la funzione f al generico x_i che gli viene passato da uno qualunque degli emitter, calcolando $f(x_i) = y_i$ ed invia y_i ad uno dei collector, secondo una ben definita politica;
- Ogni collector ha il compito di acquisire i risultati calcolati dai vari worker e di inviarli sul proprio stream di uscita.

5.2 Modello delle prestazioni

Andiamo ora ad analizzare il tempo di servizio e la latenza del pattern parallelo in questione.
Sia:

- T_E il tempo di servizio del singolo i-esimo emitter.
  Dato da $T_E = L_E + L_{com}(E_i, W_k)$, dove:
  - L_E è la latenza dell'attività dell'emitter;
  - $L_{com}(E_i, W_k)$ è il costo di comunicazione inviare l'elemento x_i al worker k-esimo.

- T_W Il tempo di servizio del generico worker.
  Dato da $T_{(W)} = L_{com}(E_i, W_k) + L_W + L_{com}(W_k, C_h)$, dove:
  - L_W è la latenza della del calcolo $f(x_i) = y_i$;
  - $L_{com}(E_i, W_k)$ è il costo di comunicazione per leggere l'elemento x_i ;
  - $L_{com}(W_k, C_h)$ è il costo di comunicazione per inviare l'elemento y_i al collector h-esimo;

- T_C Il tempo di servizio del generico collector.
  Dato da $T_C = L_C + L_{com}(W_k, C_h)$, dove:
  - L_C è la latenza dell'attività del collector;
  - $L_{com}(W_k, C_h)$ è il costo di comunicazione per leggere l'elemento y_i dal worker k-esimo;

E' possibile notare come la “multi-emitter farm” possa essere vista come una pipeline con tre stadi. Nel caso in cui si abbia una distribuzione uniforme degli elementi in ingresso per alimentare i worker, il tempo di servizio può essere così definito:

$$T_S = Max\left(\frac{T_E}{N_E}, \frac{T_W}{N_W}, \frac{T_C}{N_C}\right)$$

Supponendo che le latenze di comunicazione siano tutte eguali a L_{com} , può essere riscritto come:

$$T_S = Max\left(\frac{(L_E + L_{com})}{N_E}, \frac{(L_W + 2\,L_{com})}{N_W}, \frac{(L_{com} + L_C)}{N_C}\right)$$

La latenza della multi-emitter farm può essere espressa come:

$$L = L_E + 2L_{com} + L_W + 2L_{com} + L_C$$

che risulta essere la stessa della Farm tradizionale, che risulta essere maggiore della versione sequenziale.

Inoltre è possibile definire il grado ottimo di parallelismo, cioè il numero di worker per cui la farm ha un tempo di servizio pari a T_A, il tempo di arrivo di input per ogni emitter.

$$n_{w\,opt} = \frac{L_W + 2L_{com}}{Max\left(\frac{L_E + L_{com}}{N_E}, \frac{L_C + L_{com}}{N_C}, \frac{T_A}{N_E}\right)}$$

Consideriamo la prima parte del sistema e modelliamo la farm multi-emitter come un insieme di client (emitter) che inviano dei risultati verso i server (worker).
Consideriamo il tempo di arrivo nello stream di input del generico emitter dato da T_A^E.
Essendo sequenziale il suo tempo di invio in uscita T_P^E è dato da
$T_P^E = Max(T_A^E, L_E + L_{com})$.

Supponendo che ogni emitter utilizzi una politica di distribuzione dei risultati “round-robin” verso i worker, la probabilità che un risultato sia diretto verso un generico worker è data da $p_i = \frac{1}{n_W}$.
Andiamo ora a calcolare T_A^W.

$$T_A^W = \frac{T_P^E}{\sum_{k=1}^{n_E} \frac{1}{n_W}} = \frac{T_P^E}{n_E \frac{1}{n_W}} = \frac{n_W}{n_E} T_P^E = \frac{n_W}{n_E} Max(T_A^E, L_E + L_{com})$$

Andiamo ora a calcolare l'“utilization factor” della coda in ingresso del generico worker,

$$\rho = \frac{T_W}{T_A^W} = \frac{T_W}{\frac{n_W}{n_E} T_P^E}$$

e $T_W^P = Max(T_A^W, T_W) = Max(T_A^W, L_W + 2L_{com})$

Per far sì che T_W^P sia T_W, dobbiamo avere che $\rho<1$, quindi:

$$\rho<1 \Leftrightarrow \frac{T_W}{\frac{n_W}{n_E}T_P^E}<1 \Leftrightarrow T_W<\frac{n_w}{n_E}T_P^E$$

$$n_w>\frac{T_W n_E}{T_P^E} \Leftrightarrow n_w>\frac{T_W n_E}{Max(T_A^E, L_E+L_{com})} \Leftrightarrow n_w>\frac{T_W}{\frac{Max(T_A^E, L_E+L_{com})}{n_E}} \Leftrightarrow n_w>\frac{T_W}{Max(\frac{T_A^E}{n_E}, \frac{L_E+L_{com}}{n_E})}$$

che nel caso in cui gli emitter non siano un collo di bottiglia $n_w>\frac{T_W}{\frac{T_A^E}{n_E}} \Leftrightarrow n_w>\frac{T_W}{T_A^E}n_E$

quindi per ogni singolo emitter si ha che $n_w>\frac{T_W}{T_A^E}$, che infatti corrisponde al tipico risultato relativo al numero ottimo di worker in una farm.

Capitolo 6

Risultati sperimentali

In questo capitolo si spiega l'attività di test che è stata condotta per il calcolo effettivo delle performance raggiunte dall'applicazione sviluppata.

Sono stati utilizzati dei file PCAP, meramente al fine dei testing, file in cui è possibile memorizzare il traffico di rete precedentemente letto, per testare l'applicazione in condizioni di massimo carico.

E' stato modificato il modulo di lettura dei dati, in modo che quelli contenuti nel file pcap vengano caricati in memoria direttamente dall'applicazione stessa nel main, prima della creazione e dell'avvio della struttura parallela.

In questo modo l'emitter, anziché leggere il traffico direttamente dalla schede di rete (come nella versione originale), lo legge direttamente dalla memoria.

Come già detto questo metodo ci permette di testare il tutto, in una condizione di massimo carico, svincolandoci da problemi che potrebbero essere introdotti dalle librerie di lettura (anche diverse da PCAP) e dalla generazione di traffico per la lettura dalle schede di rete.

In questo caso la lettura di un pacchetto corrisponde alla lettura di un puntatore da un array di pacchetti precedentemente caricati in memoria ed ad un successivo accesso al pacchetto tramite deferenziazione del puntatore. Il costo di lettura è sicuramente minore del costo di una lettura vera e propria da scheda di rete, per questa ragione la banda in ingresso del test risulta essere maggiore di quella del caso reale.

Data quindi la maggiore banda di arrivo di pacchetti in input all'applicazione rispetto alla banda di arrivo nel caso reale, con il conseguente maggior stress di carico, possiamo considerare il test come un'analisi al caso pessimo.

Per completezza c'è da dire che in condizioni di test, la latenza dell'attività di lettura da scheda L_{read}^{TEST}, è minore di L_{read}^{REAL}, in quanto un accesso al pacchetto tramite deferenziazione del puntatore è sicuramente minore della latenza di una lettura vera e propria da scheda di rete, che comporta una chiama read e deferenziazione del pacchetto letto. Ciò comporta una sensibile diminuzione della latenza $L_{emitter}^{TEST}$, che comunque data la piccola differenza con $L_{emitter}^{REAL}$ non impatta con la validità del test in sé.

Il Dataset utilizzato per il test è stato creato durante l'attività di tesi mediante la cattura del traffico dal un portatile collegato in una rete LAN Ethernet. Il file PCAP è di 530

Mbyte, composto da 660.155 pacchetti, tutti trasportano traffico HTTP, di cui nello specifico 27.949 di richieste HTTP, corrispondenti a 1012 flussi diversi.

Un Dataset di questo genere rappresenta un'analisi “worst case” per quanto riguarda gli indici di speedup e scalabilità, in quanto in media ogni flusso possiede 660 pacchetti, quindi avremo in media una scrittura nella mappa dei flussi ogni 660 pacchetti. Questo non è del tutto vero a causa della pulizia dei flussi, che quindi comporta ulteriori operazioni.

Tutto questo implica che il lavoro del worker sia prevalentemente quello di accesso e reperimento del record di flusso dalla struttura dati che li contiene, con la minima latenza possibile per il worker. Dall'altro canto una maggiore dinamicità nella scrittura e lettura dei flussi, ed un'alta percentuale di richieste HTTP implicherebbe un innalzamento della latenza media del singolo worker, con un conseguente incremento del numero ottimo di core da utilizzare (ovviamente a parità di tempo di servizio degli emettitori), e quindi la possibilità di sfruttare più worker fissato il numero di emettitori.

6.1 Ambiente di lavoro

I test che sono stati condotti e che verranno puntualmente descritti in questo capitolo sono stati effettuati su “Titanic”, una macchina del Dipartimento d'Informatica avente le seguenti caratteristiche:

- 2 CPU AMD Opteron(tm) Processor 6176, con frequenza a 2.3 Ghz, in cui ogni CPU possiede 12 core (quindi 24 core in totale), con 12x 512 KB di cache L2, e 2x 6 MB di cache L3;
- 32 Gb di memoria RAM.

6.2 Latenza e tempo di servizio

Analizziamo la latenza media complessiva che un pacchetto trascorre all'interno dei moduli dell'applicativo. Essa è data:

$$L = L_{emitter} + L_{worker} + L_{collector}$$

dove:

$$L_{emitter} = L_{read} + L_{hash} + L_{(buffering\ handling)} + L_{(output\ ch\ push)}$$

$$L_{worker} = L_{(input\ ch\ pop)} + L_{parsing} + L_{(map\ get)} + L_{marking} + L_{(output\ ch\ push)}$$

$$L_{collector} = L_{(input\ ch\ pop)} + L_{write} + L_{(buffering\ handling)}$$

$L_{emitter}$	$O(350\,ns)$
L_{worker}	$O(900\,ns)$
$L_{collector}$	$O(100\,ns)$

La “multi-emitter farm” può essere vista come una pipeline con tre stadi, e nel caso in cui si abbia una distribuzione uniforme degli elementi in ingresso per alimentare i worker, il tempo di servizio quindi può essere così definito:

$$Max(T_{emitter}, T_{worker}, T_{collector})$$

dove:

$$T_{emitter}=\frac{T_{(seq\,emitter)}}{N_{emitter}} \quad T_{worker}=\frac{T_{(seq\,worker)}}{N_{worker}} \quad T_{collector}=\frac{T_{(seq\,collector)}}{N_{collector}}$$

Supponiamo inoltre che $T_A=T_{seq\,emitter}$ per ogni emitter.
Abbiamo che il numero ottimo di worker da utilizzare è dato da:

$$N_{worker}=\frac{T_{(seq\,worker)}}{Max(T_{emitter}, T_{collector})}$$

Per semplificare il ragionamento, supponiamo di usare un solo emitter ed un solo collector, perché è semplice estendere il ragionamento al caso con più emitter e più collector.

Allora abbiamo che il numero ottimo di worker da utilizzare in questo caso è dato da:

$$N_{worker}=\frac{T_{(seq\,worker)}}{Max(T_{(seq\,emitter)}, T_{(seq\,collector)})}$$

cioè $N_{worker}=\frac{T_{(seq\,worker)}}{T_{(seq\,emitter)}}=\frac{900\,ns}{350\,ns}=2.57$

Per questo motivo possiamo affermare che per ogni emitter devo utilizzare tre core, per massimizzare la banda o due core se l'intento è quello di massimizzare l'efficienza.
Un ulteriore aumento del numero di core allocati ai worker migliorerebbe il tempo di servizio della multi-farm, in quanto $T_{seq\,emitter}$ rappresenterebbe un collo di bottiglia.

$T_s=Max(T_{emitter}, T_{worker}, T_{collector})=Max\left(T_{seq\,emitter}, \frac{T_{seq\,worker}}{N_{worker}}, T_{seq\,collector}\right)$, sostituendo:

$$T_s=Max\left(350\,ns, \frac{900\,ns}{N_{worker}}, 100\,ns\right)$$

E' possibile estendere il concetto alla versione multi-emitter e multi-collector, affermando che appunto utilizzando n emitter ed n collector, posso utilizzare al più $2n$ o $3n$

worker seconda che voglia massimizzare l'efficienza o la banda rispettivamente.
Un un ulteriore aumento del numero di core allocati ai worker non migliora il tempo di servizio della multi-farm, in quanto $T_{emitter}$ rappresenterebbe un collo di bottiglia.

Se volessimo effettuare il ragionamento direttamente in termini di multi-emitter e multi-collector, possiamo affermare che:

$$T_s = Max(T_{emitter}, T_{worker}, T_{collector})$$

dove:

$$T_{emitter} = \frac{T_{(seq\ emitter)}}{N_{emitter}} \quad T_{worker} = \frac{T_{(seq\ worker)}}{N_{worker}} \quad T_{collector} = \frac{T_{(seq\ collector)}}{N_{collector}}$$

Supponiamo inoltre che $T_A = T_{seq\ emitter}$ per ogni emitter.
Abbiamo che il numero ottimo di worker da utilizzare è dato da:

$$N_{worker} = \frac{T_{(seq\ worker)}}{Max(T_{emitter}, T_{collector})} = \frac{T_{(seq\ worker)}}{Max(\frac{T_{(seq\ emitter)}}{N_{emitter}}, \frac{T_{(seq\ collector)}}{N_{collector}})}$$

e dato che $N_{emitter} = N_{collector}$ abbiamo:

$$N_{worker} = \frac{T_{(seq\ worker)}}{\frac{T_{(seq\ emitter)}}{N_{emitter}}} = \frac{T_{(seq\ worker)}\, N_{emitter}}{T_{(seq\ emitter)}} = 2{,}57\, N_{emitter}$$

come volevasi dimostrare, si ottiene il medesimo risultato.

Supponiamo ora che per ogni emitter $T_A < T_{seq\ emitter}$, allora $T_{emitter}$ rappresenta un collo di bottiglia per la multi-emitter farm, allora è possibile adottare la seguente strategia e poi ricondurci al caso sopra delineato:
Se $T_A < T_{seq\ emitter}$ significa che per ogni emitter il tempo di arrivo dei pacchetti è inferiore al tempo di servizio del singolo emitter, è però possibile effettuare una virtualizzazione della scheda di rete in cui essa viene suddivisa in varie schede di rete virtuali, ed ognuna di esse viene affidata ad un diverso emitter (replicazione degli emettitori), in questo modo è possibile ricondurci al caso in cui $T_A \geq T_{emitter}$ anche se $T_A < T_{seq\ emitter}$.

Inoltre nel caso in cui si ha che $T_A > T_{emitter}$ si ha che $N_{worker} = \frac{T_{(seq\ worker)}}{T_A}$.

Sull'architettura su cui sono stati effettuati i test, le operazioni che vengono eseguite dal worker sul singolo pacchetto sono operazioni di grana fine infatti il tempo di servizio risulta essere $O(900\,ns)$. Questo fatto come spiegato nei capitoli precedenti frena il raggiungimento dello speedup ideale.

Per ovviare a questo problema, nei test condotti in questo capitolo utilizziamo buffer di k pacchetti. In questo modo si paga la latenza di comunicazione L_{com} appunto ogni k pacchetti anziché ogni pacchetto. Siamo appunto andati a trovare sperimentalmente un k tale da minimizza il T_c sul dataset di riferimento. E' risultato che un k $O(100)$ o $O(1000)$, sia la migliore soluzione possibile. I seguenti test sono stati effettuati con bufferizzazione utilizzando precisamente $1000\,pkt/buf$.

Inoltre dato che T_s dell'emitter è inferiore al T_s del worker, e dato che il test è svolto in condizioni di massimo carico, si raggiungerà la situazione in cui la coda bounded in ingresso al worker si satura. Pertanto per garantire la correttezza del testing è stata disabilità la funzionalità di scarto del pacchetto in condizione di code interne sature, come impiegata nel caso reale. Pertanto qualora l'emitter trovi la coda di input del worker piena, non scarta il pacchetto, ma attende che si liberi spazio per la successiva scrittura nelle medesima coda.

6.3 Scalabilità

Richiamiamo il concetto di scalabilità, essa è definita come il rapporto fra il tempo necessario alla terminazione della computazione parallela utilizzando un grado di parallelismo eguale ad uno, ed il tempo tempo necessario alla computazione parallela con grado di parallelismo eguale ad n. $Scalability(n) = \frac{T_{par}(1)}{T_{par}(n)}$

Dove $T_{par}(i)$ rappresenta il tempo di esecuzione dell'applicazione parallela con grado di parallelismo i.

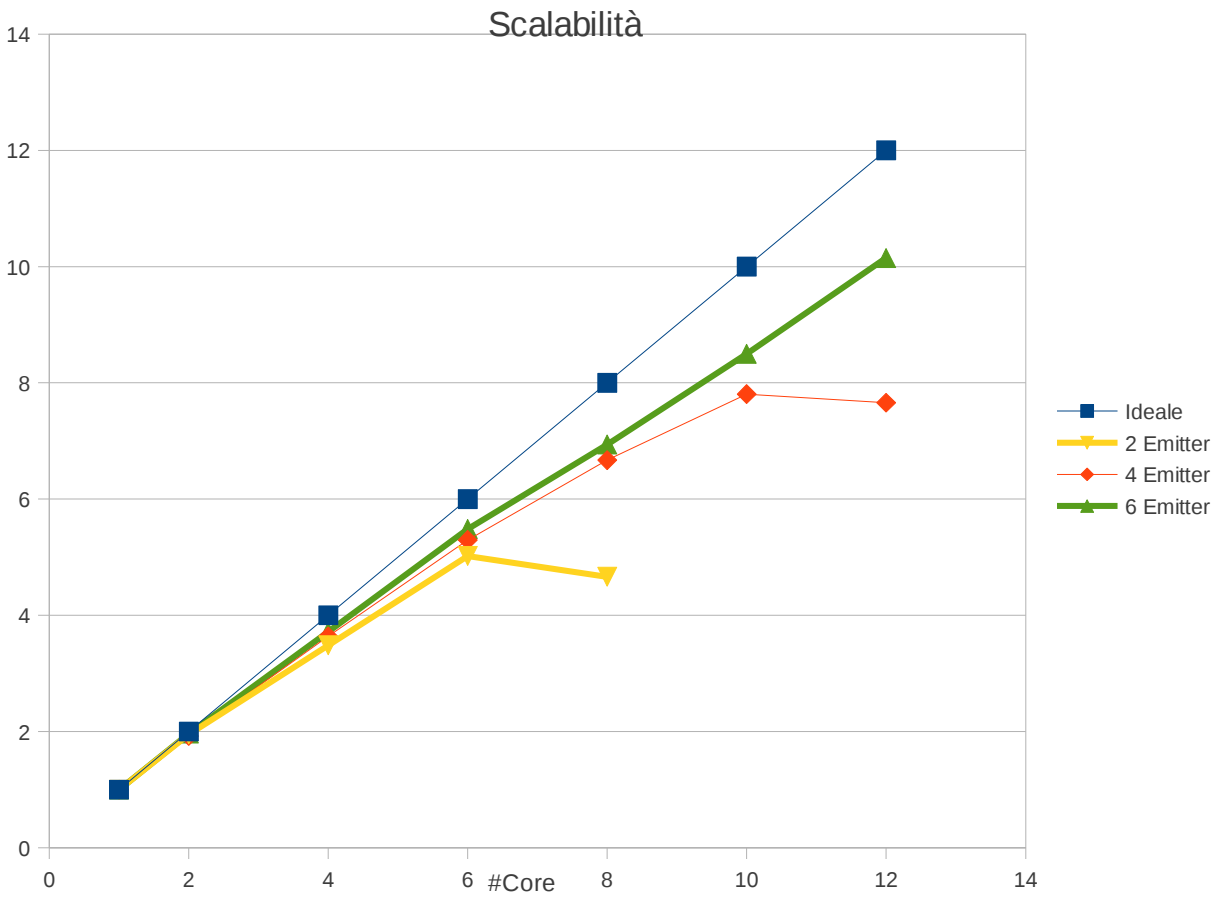


Grafico 6.1 Scalabilità

Il Grafico 6.1 mostra sull'asse delle ascisse n il numero di core e sull'asse delle ordinate la corrispettiva $Scalability(n)$, appunto secondo la formula sopra definita.

Nel grafico è tracciata la scalabilità ideale, data appunto dalla funzione identità $Scalability(n)=n$, mentre le altre curve presenti rappresentano la scalabilità fissato il numero di emettitori allocati in partenza.

Quindi ricapitolando fissato il numero di emettitori n_E utilizzati, è tracciata la curva che identifica la relativa scalabilità $Scalability(n)_{n_E}$.

Fissato n_E, è possibile notare come $Scalability(n)_{n_E}$ mantenga un andamento pressoché lineare sino per un valore di n pari a $2n_E$, poi per $n>2n_E$ si nota come $Scalability(n)_{n_E}$ non abbia più un andamento lineare, cioè l'ulteriore incremento del numero di core allocati ai worker è meno redditizio che precedentemente, sino

addirittura notare un piccolo peggioramento nella $Scalability(n)_{n_E}$ all'aumentare ulteriormente di n.

L'analisi appena fatta è la riconferma della bontà del ragionamento emerso nella fase di calcolo delle latenze e dei vari tempi di servizio di ogni modulo, il quale affermava appunto che per ogni emitter, N_{worker} fosse pari a $2{,}57$, ciò è esattamente in linea con quanto verificatosi nell'analisi dell'indice di scalabilità.

6.4 Speedup

Richiamiamo il concetto di Speedup, che rappresenta la bontà dell'algoritmo parallelo rispetto alla relativa versione sequenziale per la risoluzione del medesimo problema.

E' definito come il rapporto fra il miglior tempo di esecuzione sequenziale ed il tempo di esecuzione parallelo sfruttando un grado di parallelismo fissato ad n.

$$Speedup(n) = \frac{T_{seq}}{T_{par}(n)}$$

- T_{seq} è il miglior tempo di esecuzione sequenziale;
- $T_{par}(n)$ è tempo di esecuzione parallelo, con n il grado di parallelismo utilizzato.

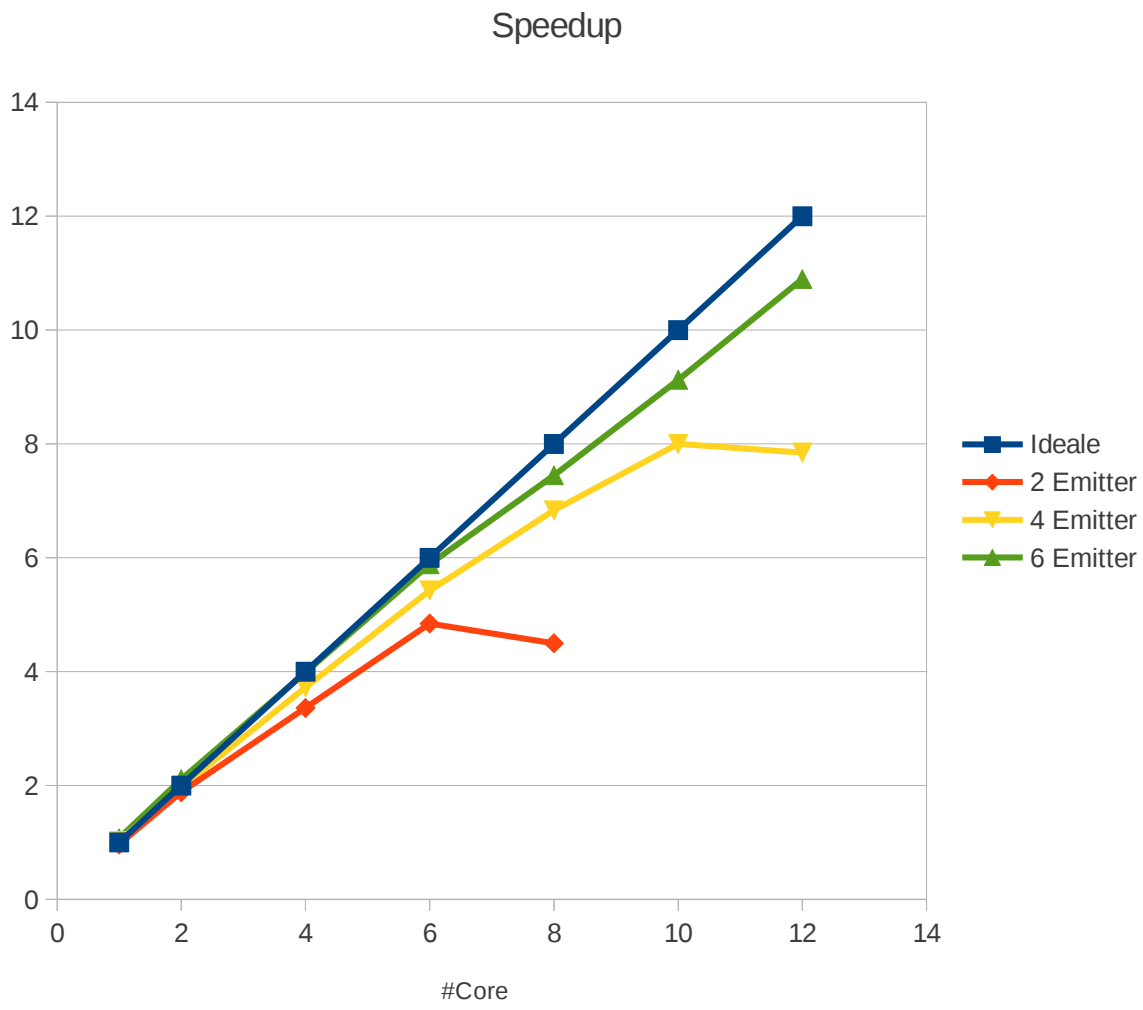


Grafico 6.2 Speedup

Nel grafico 6.2 abbiamo è disegnato lo *Speedup* ideale, dato dalla funzione identità $Speedup(n)=n$. Le altre curve presenti rappresentano lo speedup fissato il numero di emettitori allocati in partenza, corrispondenti al numero di schede di rete da dover gestire.

Quindi ricapitolando fissato il numero di emettitori n_E utilizzati, è tracciata la curva che identifica il relativo $Speedup(n)_{n_E}$.

Come per quanto visto con la scalabilità, anche in questo caso fissato n_E, è possibile notare come $Speedup(n)_{n_E}$ abbia un andamento pressoché lineare sino per un valore di

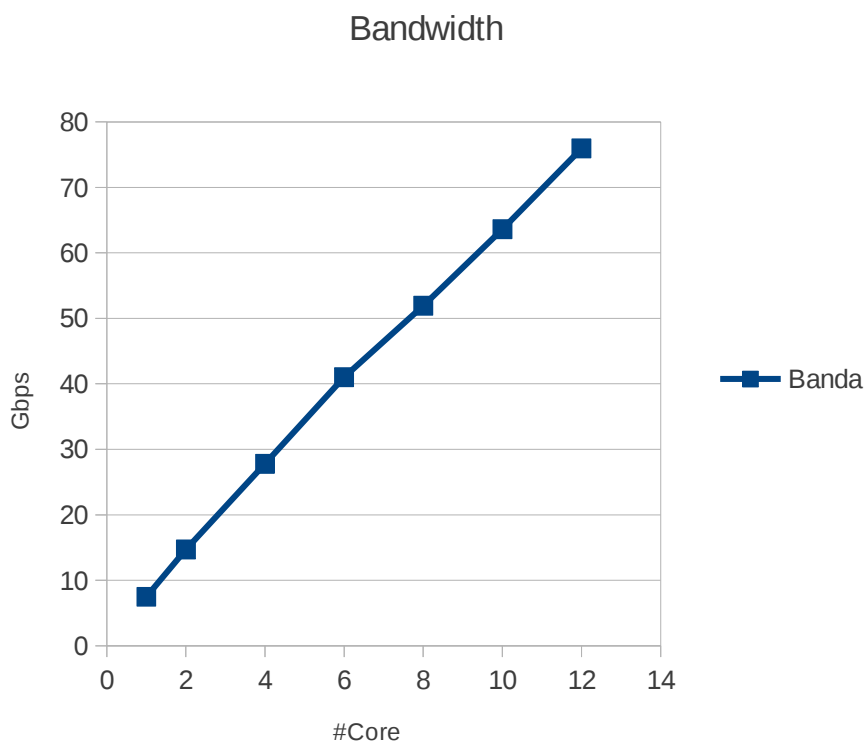
n pari a $2n_E$, poi per $n>2n_E$ si nota come $Speedup(n)_{n_E}$ non mantenga più un andamento lineare, con un comportamento del tutto analogo a quanto analizzato per la $Scalability(n)_{n_E}$.

6.5 Bandwidth

Andiamo ora ad analizzare la banda dell'applicativo espressa in Gbps (Gigabit per secondo). Abbiamo sull'asse delle ascisse il numero di core allocati agli worker, e sull'asse delle ordinate la relativa banda di dati che l'applicazione riesce a gestire, espressa in Gbps.

Tuttavia è doveroso mettere in evidenza alcune caratteristiche del dataset utilizzato. Ogni flusso possiede in media 660 pacchetti, pertanto avremo in media una scrittura nella mappa dei flussi ogni 660 pacchetti. Questo significa che la latenza dell'attività del worker sia prevalentemente determinata dall'accesso e reperimento del record di flusso dalla struttura dati che li contiene. Questa latenza rappresenta la minima latenza possibile per l'attività del worker. Pertanto il tempo di servizio medio del worker con questo dataset risulta essere molto basso, conseguentemente si ottiene un'alta banda dato che è definita come l'inverso del tempo di servizio.

Grafico 6.3 Bandwidth



6.6 Confronto con altre soluzioni di analisi DPI

Questo paragrafo è dedicato al confronto delle prestazioni dell'applicativo realizzato con le prestazioni delle soluzioni presentate nel capitolo delle tecnologie, inerenti alle analisi DPI che riguardano lavori analoghi.

Per quanto riguarda le soluzioni che effettuano un'analisi sequenziale del traffico in [AN13] sono stati condotti alcuni test riguardo le performance di Libprotoident ed l7filter. Il primo è in grado di raggiungere una banda di 7,75 Gbps, mentre il secondo è in grado di raggiungere una banda di 2,42 Gbps. Ovviamente Libprotoident utilizza un approccio Lightweight Packet Inspection (LPI), analizzando solo i primi 4 byte del payload applicativo. Per questo motivo risulta anche essere meno accurato di l7filter. Quest'ultima soluzione utilizza la tecnica del pattern matching per la classificazione dei pacchetti, sfruttando le espressioni regolari, pertanto risulta essere un'attività computazionalmente più pesante della lettura di un numero prefissato di byte.

Per queste considerazioni era prevedibile che Libprotoident avesse una banda superiore rispetto a l7filter.

Rispetto alla nostra applicazione, è possibile notare come già con un worker si è in grado di gestire una banda superiore alle soluzioni sopra citate, pertanto già con un grado parallelismo eguale ad 1 si è in grado di effettuare un'analisi full-DPI con una banda superiore. Tuttavia c'è da precisare che la nostra applicazione effettua DPI solo sul protocollo HTTP, mentre le altre offrono un'analisi per una vasta gamma di protocolli applicativi.

Andiamo ora a confrontare l'applicativo sviluppato con le soluzioni che sfruttano il calcolo parallelo per l'analisi. In [YX11, WCH09] non sono forniti i dati relativi alla banda dell'applicativo, per questo motivo ci basiamo sull'indice di speedup che è invece fornito.

In [YX11] utilizzando un grado di parallelismo fissato a 16, si ha uno speedup eguale a 10. Per quanto riguarda la nostra soluzione sviluppata, è possibile notare che quando il traffico è generato almeno da 6 emitter, si ottiene uno speedup eguale a 10, con un numero di worker pari a 12. In aggiunta ai 12 worker utilizzati nel test, c'è ovviamente anche da tenere in considerazione che nella nostra applicazione utilizziamo 6 emitter per generare traffico e 6 collector per la scrittura del traffico. Pertanto in toto abbiamo allocato 24 unità parallele di esecuzione. Possiamo quindi evincere che l'efficienza della nostra applicazione è migliore rispetto all'altra delineata in [YX11] se si tengono in considerazione solo i worker. Si può ulteriormente evincere che a parità di numero di worker impiegati, la nostra applicazione garantisce una banda superiore rispetto a [YX11].

Riguardo la soluzione spiegata in [WCH09], utilizzando un grado di parallelismo fissato a 14, si ottiene uno speedup eguale a 6.25. La nostra applicazione, ha uno speedup eguale a 6, con un numero di worker pari a 6. Possiamo quindi evincere che l'efficienza

della nostra applicazione è migliore rispetto all'altra soluzione.

Inoltre si può affermare che a parità di numero di worker impiegati la nostra applicazione garantisce una banda superiore rispetto all'altra applicazione.

Come si è visto le soluzioni che sfruttano il parallelismo derivante dalle architetture multi-core, non riescono a raggiungere lo speedup ideale in relazione al grado di parallelismo utilizzato. Come visto nei capitoli precedenti una delle motivi per cui [YX11] non riesce a raggiungere lo speedup ideale, è determinata dal fatto che la distribuzione dei dati fra i vari thread è basata su una struttura dati condivisa. Per garantire la correttezza delle operazioni sulla struttura dati condivisa, sono presenti dei meccanismi di lock. L'impiego di tali meccanismi implica una sequenzializzazione delle operazioni che sono protette dai meccanismi di lock, la quale rappresenta un freno al parallelismo fisico fra le varie unità di esecuzione parallele, con una conseguente impossibilità nel raggiungimento dello speedup ideale.

Per quanto riguarda [WCH09] una delle motivi per cui non riesce a raggiungere lo speedup ideale, è determinata dalle criticità derivanti dal bilanciamento di carico fra le varie unità di esecuzione parallele. Fatto che se non opportunamente gestito, porta infatti ad un aumento dei relativi tempi di servizio dei vari moduli paralleli, con una conseguente degradazione delle prestazioni generali del sistema.

Si passa ora a confrontare le soluzione hardware-based con l'applicativo sviluppato. Sostanzialmente realizzano l'analisi DPI andando ad effettuare del pattern matching sui dati a livello applicazione, alla ricerca di particolari pattern per classificare il protocollo oppure alla ricerca di pattern.

Come è stato visto, le implementazioni di queste soluzioni sono fondamentalmente basate sull'utilizzo di FPGA (Field Programmable Gate Arrays) si veda [ACF05, TRI05, THD12], oppure sull'impiego di CAM (Content Addressable Memories), si veda [YRH04, WTD06, YL05, AMK06].

Queste soluzioni per l'individuazione del protocollo applicativo effettuano il pattern matching. Queste hanno la necessità di controllare tutto il payload prima di processare il pacchetto successivo. Inoltre non associano all'analisi DPI un concetto di stato del flusso, pertanto l'analisi è effettuata indipendentemente pacchetto per pacchetto.

Alcune di queste soluzioni [AMK06] possono raggiungere una banda di 20 Gbps nel migliore dei casi. La nostra applicazione già con 4 worker riesce a gestire una banda superiore ai 20 Gbit/s. Questo può essere considerato un risultato positivo, in quanto l'applicazione è in grado di gestire bande di traffico comparabili alle soluzioni basate su hardware. Inoltre c'è da tenere in considerazione che solitamente l'approccio software ha una maggiore economicità ed una maggiore versatilità in relazione alla possibile introduzione di nuovi protocolli da dover gestire.

Per completezza è doveroso aggiungere che la nostra applicazione effettua DPI sul solo protocollo HTTP, mentre queste soluzioni hardware-based effettuano il pattern matching per la classificazioni di una larga quantità di protocolli. Pertanto in linea generale le analisi delle soluzioni hardware-based risultano essere più complesse, conseguentemente aventi latenze di calcolo maggiori rispetto a quelle svolte dalla nostra applicazione. Tuttavia rispetto alla nostra soluzione non hanno un concetto di stato del flusso, per questo motivo non pagano nessun costo di accesso ad una qualsiasi struttura dati che mantiene gli stati dei flussi.

Infine è necessario sottolineare come sia difficile comparare accuratamente le varie soluzioni con la nostra per le seguenti motivazioni:

- il dataset su cui è stato effettuato il test della nostra applicazione è diverso da quelli utilizzati per le sperimentazioni della altre soluzioni;
- la nostra applicazione effettua la sola analisi DPI per il protocollo HTTP, mentre le altre soluzioni forniscono analisi per una vasta gamma protocollare a livello applicativo, pertanto in linea generale, è possibile affermare che effettuano calcoli più complessi da un punto di visto di pesantezza computazionale rispetto alla nostra applicazione;
- l'architettura su cui è stato effettuato il test della nostra applicazione è diversa da quelle utilizzate per le sperimentazioni delle altre soluzioni;

Capitolo 7

Conclusioni e sviluppi

L'obiettivo del lavoro di tesi è stato quello realizzare un'applicazione che sfruttasse il parallelismo offerto dalle architetture multi-core con CPU x86, al fine di effettuare firewalling DPI per il protocollo applicativo HTTP, relativamente al traffico su reti Ethernet. L'applicazione deve essere in grado di poter gestire alte bande di traffico al variare del numero di schede di rete in gestione. Per la realizzazione del supporto per lo sfruttamento del parallelismo delle architetture multi-core è stato impiegato FastFlow.
Questo è un framework di programmazione parallela efficiente, per sistemi multi-core, sviluppato dall'università di Pisa e di Torino. Il framework offre dei meccanismi di comunicazione con bassa latenza, particolarmente adatto a computazioni di grana molto fine, come quelle che vengono eseguite su stream di pacchetti, su reti di calcolatori a bande molto elevate.

E' stato analizzato il problema nel dettaglio e sono state individuate le varie attività sequenziali con cui decomporre il problema. In seguito è stata valutata l'adozione di uno dei pattern paralleli offerti da FastFlow al suo più alto livello di astrazione. A questo livello, l'adozione di una pipeline i cui suoi tre stage fossero composti da una farm, è stata valutata come la soluzione che più si adattasse a ciò di cui si aveva bisogno.
Tuttavia tale pattern non garantiva una buona scalabilità al variare del numero di schede di rete in gestione, in quanto il collector della farm sarebbe potuto diventare un ipotetico collo di bottiglia. Inoltre il pattern necessitava dell'allocazione aggiuntiva di ulteriori core per orchestrare la comunicazione fra le varie farm, pertanto sarebbe stato considerato come uno spreco di risorse computazionali, in quanto non sarebbe risultato essere espressamente funzionale.
Per risolvere i motivi sopra citati è stato ideato un nuovo pattern parallelo, che è stato chiamato “multi-emitter farm”, implementato utilizzando il “low level” di FastFlow sfruttando le code SPSC. Questo pattern si configura come una nuova versione di farm, in cui vi sono più emitter, più collector ed ovviamente più worker.
La “multi-emitter farm” viene utilizzata come building-block dell'applicazione, essa rappresenta una struttura parallela molto utile in vari altri ambiti, per questo motivo sarebbe stata meritevole di studio e realizzabile in sé per sé, senza il fine di utilizzo nell'ambito del firewalling.
Inoltre è stato creato il modello teorico delle prestazioni per la definizione delle metriche di base, quali la latenza ed il tempo di servizio, necessari per il calcolo degli indici prestazionali.

Sfruttando il nuovo pattern parallelo, è stata quindi progettata e implementata l'applicazione parallela che incapsula tutte le singole attività sequenziali definite. L'applicazione è “lockless”, cioè non utilizza al suo interno nessun meccanismo di lock, che sono solitamente considerati una fra le più comuni cause che frenano il parallelismo fisico fra i vari thread, con una conseguente degradazione dello speedup.

Una volta realizzata l'applicazione, è stata effettuata un'attività di verifica e validazione.
In una prima fase è stata effettuata la verifica della correttezza funzionale dell'applicazione, effettuando del firewalling in un caso di utilizzo reale, grazie al quale è stato appurato che l'applicazione è stata correttamente realizzata.
Dopo di che è stata effettuata l'attività di validazione, in cui grazie ai modelli teorici delle prestazioni precedentemente costruiti per il pattern parallelo, sono state rilevate le metriche di base di latenza e tempo di servizio. Successivamente sono stati calcolati gli indici prestazionali per controllare, che si stesse realizzando l'applicazione in modo corretto. In altre parole è stato controllato che gli obiettivi di tesi prefissati fossero stati raggiunti.
Grazie agli indici prestazionali è stato possibile verificare come l'applicazione abbia pienamente soddisfatto gli obiettivi di partenza, offrendo una buona scalabilità e speedup, sia al variare del numero di core utilizzati per i worker, sia al variare del numero di schede di rete.
Per scalabilità al variare del numero di core utilizzati per i worker, si intende fissare il numero di emettitori che generano il traffico e calcolare la scalabilità in funzione del numero di core allocati ai worker; ad esempio ad un raddoppio del numero di core allocati ai worker, ci si aspetta un dimezzamento del tempo di completamento.
Per scalabilità al variare del numero di schede di rete, si intende che variando il numero di schede di rete, conseguentemente variando il numero di emitter (uno per ogni scheda), si ottenga una scalabilità sul numero numero di core allocati ai worker (il concetto espresso sopra).

Tutto ciò ha dimostrato quindi la capacità dell'applicazione di essere in grado di poter gestire l'alta banda di traffico, al variare delle schede in gestione come richiesto in partenza.

Nell'analisi è stato inoltre messo in evidenza il “worst case”. Infatti nel caso in cui vi sia poca dinamicità nei flussi gestiti, si otterrebbe una bassa latenza media dell'attività del worker. Fissando a n_E, il numero di schede di rete in gestione, è stato analizzato come il numero ottimo di worker da poter utilizzare sia rappresentato da $3n_E$ se si vuole massimizzare la banda, e da $2n_E$ se si vuole invece massimizzare l'efficienza. Un ulteriore aumento del numero di worker non sarebbe utile, in quanto gli emettitori non sarebbero in grado con il loro tempo di servizio di alimentare sufficientemente i worker.

Il lavoro svolto ha consentito di raggiungere gli obiettivi prefissati, realizzando un'applicazione: scalabile sul numero di interfacce in gestione, in grado di poter gestire alte bande di traffico e che possiede una scalabilità e speedup, quasi lineare, quindi molto simile a quella ideale.

Il lavoro è comunque estendibile sotto vari aspetti, che non sono stati approfonditi per i limiti imposti dall'attività della tesi stessa ed anche perché avrebbero distolto l'attenzione dagli obiettivi assegnati in direzione di altri ambiti di ricerca.

Sono riportati di seguito i vari aspetti emersi nel corso del lavoro, che si ritengono meritevoli di ulteriore sviluppo o integrazione:

- Integrazione di “multi-emitter farm” in FastFlow: il nuovo pattern realizzato è stato costruito sfruttando il “low layer” di FastFlow, sfruttando quindi le code SPSC. Sarebbe interessante integrare il nuovo pattern parallelo “multi-emitter farm” in FastFlow offrendolo direttamente come una variante della farm, dando la possibilità di utilizzarlo come uno skeleton, visibile al livello di astrazione più alto di FastFlow.
- Datalink abstraction layer: l'applicazione attualmente effettua il firewalling HTTP su reti Ethernet. Vi è quindi la possibilità di estendere l'applicazione in varie direzioni, da un lato la possibilità di offrire il firewalling per più protocolli applicativi, dall'altra la possibilità di realizzare un “datalink abstraction layer”, grazie al quale sarebbe possibile effettuare il firewalling su qualsiasi tipo di rete a livello collegamento.
- Estensione del firewalling: l'applicazione effettua il firewalling HTTP, sarebbe possibile estendere l'applicazione incrementando il numero di protocolli applicativi da analizzare, realizzando analisi DPI su una vasta gamma di protocolli applicativi.

Bibliografia

[AN13] Alcock S., Nelson R., *Libprotoident: Traffic Classification Using Lightweight Packet Inspection*, <http://pearson.org.nz/sites/default/files/lpi.pdf>, University of Waikato, March 2013.

[ACC03] Aldinucci M., Campa S., Ciullo P., Coppola M., Danelutto M,, Pesciullesi P., Ravazzolo R., Torquati M., Vanneschi M., Zoccolo C., *ASSIST demo: a high level, high performance, portable, structured parallel programming environment at work.* In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, Proc. of 9th Intl. Euro-Par 2003 Parallel Processing, volume 2790 of LNCS, pages 1295–1300, Klagenfurt, Austria, August 2003.

[ADK11a] Aldinucci M., Danelutto M., Kilpatrick P., Meneghin M., and TorquatiM. *Accelerating code on multi-cores with FastFlow*, in: Proc. of Euro-Par 2011, Bordeaux, September 2011.

[ADK11b] Aldinucci M., Danelutto M., Kilpatrick P., Torquati M. *FastFlow: high-level and efficient streaming on multi-core. (A FastFlow short tutorial), in: Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing*, Wiley, July 8 2011.

[ADK12] Aldinucci M., Danelutto M., Kilpatrick P., and Torquati M. *FastFlow: high-level and efficient streaming on multi-core. (A FastFlow short tutorial), in: Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing*, Wiley, 2012.

[ADT12] Aldinucci M., Danelutto M., Torquati M. *FastFlow tutorial, Technical Report TR-12-04*, Computer Science Departement, University of Pisa, March 2012.

[ACD04] Ammirati P., Clementis A., D'Agostino D., Giannuzzi V., *Using a structured programing environment for parallel remote virtualization*, in Euro Par 2004 Parallel Processing, Springer-Verlag, 2004.

[ABC08] Ancilotti P., Boari M., Ciampolini A., Lipari G., *Sistemi operativi*, seconda edizione, McGraw-Hill, 2008.

[ACF05] Aldwairi M., Conte T., Franzon P., *Configurable string matching hardware for speeding up intrusion detection*, SIGARCH Computer Architecture News, 33(1):99–107, March 2005.

[ALP13] *Application Layer Packet Classifier for Linux* <http://l7-filter.sourceforge.net/>, March 2013.

[AMK06] Alicherry M., Muthuprasanna M., Kumar V., *High speed pattern matching for*

network IDS/IPS, In Proceedings of the 2006 IEEE International Conference on Network Protocols, IEEE Computer Society, 2006.

[BON94] Bonwick J., *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, Boston USENIX Proceedings, 1994.

[CDG01] Ciullo P., Danelutto M., Vaglini L., Vanneschi M., Guerri D., and Lettere M., *Ambiente ASSIST: modello di programmazione e linguaggio di coordinamento ASSIST-CL* (versione 1.0), Progetto ASI-PQE2000, 2001.

[CTS13] C*lass template – std::Map*, <http://www.cplusplus.com/reference/map/map/>, 22 Gennaio 2013.

[COL91] Cole M., *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, Cambridge, MA USA, 1991.

[CPRZ04] Coppola M., Pesciullesi P., Ravazzolo R., Zoccolo C., A Parallel Knowledge Discovery System for Customer Profiling, in Euro Par 2004 Parallel Processing, Springer-Verlag, 2004.

[CPP13] *CUDA Parallel progragramming platform*, <http://www.nvidia.com/object/cuda_home_new.html>, March 2013.

[DAN11] Danelutto M., *Distributed System: Paradigms and Models, Teaching material* – Laurea magistrale in Computer Science and Networking, Pisa, March 2011.

[FAS13] FastFlow (v2.0) <http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about>, March 2013.

[FIE99] Fielding R., e altri, RFC 2616 *Hypertext Transfer Protocol – HTTP/1.1* , June 1999.

[FSE13] *Firewall security extended to Layer 8*, <http://www.cyberoam.com/firewall.html>, March 2013.

[KNU98] *Knuth D., Volume 3, Sorting and Searching*, Second Edition,Reading, Massachusetts: Addison-Wesley, 1998.

[KR08] Kurose J., Ross K., *Reti di Calcolatori e Internet Un appoccio top-dpwn*, quarta edizione, AddisonWesley , 2008.

[LAM83] Lamport L., *Specifying concurrent program modules*, ACM Transactions on Programming Languages and Systems, 5(2):190–222, April 1983.

[MFT13] *Migrating from Traditional to Flexible NetFloW*, <http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/ps6965/white_paper_c11-545581.html>, March 2013.

[MJ92] McCanne S., Jacobson V., *The BSD Packet Filter: A New Architecture for User-level Packet Capture*, December, 1992.

[MPI13] *Message Passing Interface (MPI) standard*,Course Notes High Performance Computing Marco Vanneschi © - Department of Computer Science, University of Pisa <http://www.mcs.anl.gov/research/projects/mpi/>, March 2013.

[NOE13] *nDPI Open and Extensible GPLv3 Deep Packet Inspection Library,* <http://www.ntop.org/products/ndpi/>, March 2013.

[OPE13] O*pendpi*, <http://www.opendpi.org/>, March 2013.

[OTO13a] *OpenMP - The OpenMP® API specification for parallel programming*, <http://openmp.org/wp/>, March 2013.

[OTO13b] *OpenCL - The open standard for parallel programming of heterogeneous systems*, <http://www.khronos.org/opencl/>, March 2013.

[OU13] Ou G., Understanding Deep Packet Inspection (DPI) Technology, <http://www.digitalsociety.org/files/gou/DPI-Final-10-23-09.pdf>, Digital Society, March 2013.

[PTP13] *POSIX Threads Programming* <https://computing.llnl.gov/tutorials/pthreads/>, 15 March 2013.

[TAN03] Tanenbaum A., Reti di calcolatori, 2003.

[TGP13] *The GNU Profiler*, <http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html>, 3 Dicembre 2012.

[TL13] *TcpDump & LibPcap* <http://www.tcpdump.org/>, 15 Gennaio 2013 .

[TMS13] *The Muenster Skeleton Library Muesli - A Comprehensive Overview,* <http://www.ercis.org/de/node/230>, March 2013.

[TNP13] *The netfilter.org project*, http://www.netfilter.org/, March 2013.

[TOR10] Torquati M, T*echnical Report: TR-10-20 Single-Producer/ Single-Consumer Queues on Shared Cache Multi-Core Systems*, Computer Science Department, November 2010.

[THD12] Thinh T. N., Hieu T. T., Dung V. Q., Kittitornkun S., *A FPGA-based deep packet inspection engine for Network Intrusion Detection System,* in 9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2012.

[TRI05] Tripp G., A finite-state-machine based string matching system for intrusion detection on high-speed networks. In EICAR 2005 Conference Proceedings, 2005.

[ROE99] Roesch M., *Snort - lightweight intrusion detection for networks*, in Proceedings of the 13th USENIX conference on System administration, USENIX Association, 1999.

[ROS13] Ross K.W., *Network Security*, <http://cis.poly.edu/~ross/>, March 2013.

[SP13] *SkeTo Project*, <*http://sketo.ipl-lab.org/*>, *15* March 2013.

[SID12] Snort - Intrusion Detection and Prevention system, <http://www.snort.org/>, 2012.

[STR08] Stroustrup B., *The C++ Programming Language* , Addison-Wesley, December 2008.

[VAN02] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing, 28(12):1709–1732, December 2002.

[VAN12] Vanneschi M., *Course Notes - High Performance Computing* , *Part 1 Structuring and Design Methodology for Parallel Computations* , Department of Computer Science, University of Pisa, 2012.

[VYU12] Vyukov D., *Bounded MPMC queue* <https://sites.google.com/site/1024cores/home/lock-free-algorithms/queues/bounded-mpmc-queue>, 3 Dicembre 2012.

[WCH09] Wang J., Cheng H., Hua B., Tang X., *Practice of parallelizing network applications on multi-core architectures,* in Proceedings of the 23rd international conference on Supercomputing. ACM, 2009.

[WTD06] Weinsberg Y., Tzur-david S., Dolev D., *High performance string matching algorithm for a network intrusion prevention system (nips),* in High Performance Switching and Routing, 2006.

[YX11] Yunchun L., Xinxin Q., *A parallel packet processing method on multi-core systems,* In Proceedings of the 2011 10th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, IEEE Computer Society, 2011.

[YL05] Yusuf S., Luk W., *Bitwise optimised CAM for Network Intrusion Detection Systems,* In Proceedings of the 2005, International Conference on Field Programmable Logic and Applications, IEEE, 2005.

[YRH04] Yu F., Randy, Katz H., and T. V. Lakshman, *Gigabit rate packet pattern-matching using TCAM,* in Proceedings of the 12th IEEE International Conference on Network Protocols. IEEE Computer Society, 2004.

Appendice A: sperimentazioni preliminari

Lo sviluppo dell'applicazione, già spiegata nei capitoli precedenti, è stata preceduta da un'attività sperimentale nelle fasi iniziali del lavoro di tesi, tesa ad ottenere informazioni riguardo il comportamento della libreria FastFlow su computazioni di grana molto fine nei problemi su reti.

L'attività ha riguardato l'implementazione di RIP, un noto algoritmo di routing di tipo “distance vector”, in una versione semplificata, realizzando un applicazione che smista i pacchetti ricevuti sullo stream di input verso verso gli stream di output che danno sulle interfacce di rete di uscita.

La versione è semplificata in quanto il routing prende in esame solo pacchetti IPv4 proventi da un livello collegamento Ethernet e le decisioni riguardo l'inoltro dei pacchetti sono prese solo tramite una corrispondenza perfetta fra indirizzo IP del pacchetto in esame e gli indirizzi IP presenti nella tabella di inoltro.

L'applicazione è sostanzialmente composta da uno skeleton farm, in cui l'emitter genera frame Ethernet contenenti datagrammi IPv4, opportunamente costruiti per poi essere analizzati dai thread worker.

Ogni thread worker può accedere ad una tabella d'inoltro costruita tramite RIP, per decidere su quale interfaccia di uscita deve essere inviato il datagrammma,

La tabella d'inoltro costituisce una struttura dati condivisa in lettura ed in scrittura da tutti i singoli worker, in quanto se il datagramma letto è un datagramma IPv4 questo viene instradato normalmente con una conseguente lettura dalla struttura dati per l'ottenimento dell'interfaccia di uscita su cui inoltrare il datagramma.

Quando invece il datagramma porta nel proprio payload il protocollo RIP, il worker deve eseguire le procedure per l'aggiornamento della tabella d'inoltro con eventuali modifiche della stessa.

Per questa ragione la struttura è protetta con dei meccanismi di lock.

La tabella d'inoltro è stata implementata tramite una hash map, a liste di trabocco. Tuttavia se il lock fosse utilizzato per l'accesso all'intera struttura (la hash map) un solo worker alla volta potrebbe accedere alla struttura, e dato che la parte rilevante del codice del worker effettua solo tale computazione, sarebbe stato come rendere sostanzialmente sequenziale l'intera applicazione parallela.

Per questa ragione è stato deciso di utilizzare un lock per ogni lista di trabocco, quindi tutte le volte che un thread worker vuole accedere ad una chiave-valore appartenente ad una certa lista deve prima acquisirne il lock, poi può eseguire le proprie operazioni ed infine rilasciare tale lock.

Con questa soluzione è garantito un certo livello di parallelismo fra i worker, in quanto solo nel caso sfortunato che tutti i worker vogliano accedere ad una determinata lista di trabocco si verificherebbe una sequenzializzazione del calcolo, che risulta essere poco probabile considerando il basso numero di worker e l'alto numero di liste di trabocco presenti nella hash map.

Una volta implementato il tutto, è stato condotto un lavoro volto a trovare il limite inferiore della scalabilità e dello speedup. E' stato anche impiegato lo strumento GNU gprof (GNU profiler), si veda [TGP13], che permette di analizzare e determinare il tempo impiegato da ciascuna procedura e sotto procedura.

In questo modo è stato possibile analizzare i tempi impiegati per l'esecuzione del codice funzionale ($T_{(seq\,worker)}$) e l'overhead di comunicazione della libreria FastFlow, di lettura e scrittura dalla code per la comunicazione fra le varie unità ($T_{(read\,data)}$ e $T_{(write\,data)}$ rispettivamente).

Il codice è stato esteso con un'ulteriore parte che effettua un ciclo con calcoli in virgola mobile, il cui numero di volte per cui il ciclo deve essere eseguito è reso parametrico e modificabile da argomento del programma.

E' stato così possibile aumentare la grana del thread worker in modo arbitrario e con il metodo della bisezione si è trovato il minimo tempo di servizio che permettesse al codice sequenziale del worker di scalare.