ffProbe: una cache NetFlow implementata utilizzando una pipeline multithreaded

Daniele De Sensi

A.A. 2009/2010

1. Introduzione e scopo del progetto
2. Compilazione e parametri di avvio
3. Struttura
4. Test
5. Conclusioni
6. Riferimenti

1. Introduzione e scopo del progetto

Lo scopo del progetto è quello di creare una cache Netflow [1] utilizzando una libreria a skeleton [2] e analizzare le performance al variare del numero di thread utilizzati. La libreria a skeleton in questione è FastFlow [3]. Fastflow è un framework per la programmazione parallela su piattaforme multicore, particolarmente adatto per applicazioni che lavorano su stream di dati, motivo per cui è stato scelto l'utilizzo in questa implementazione di una cache Netflow. Questo framework è strutturato a livelli che astraggono man mano l'architettura a memoria condivisa sottostante, ed offre una serie di meccanismi di comunicazione a bassa latenza sfruttando delle code *lock-free*. Lo svantaggio sta però nel fatto che questi thread devono effettuare attesa attiva proprio per la mancanza di meccanismi di sincronizzazione. Nello sviluppo della cache è stato innanzitutto utilizzato un costrutto tipico della programmazione a skeleton e messo a disposizione da Fastflow: la *pipeline.* Inoltre a causa dell'overhead dovuto alle frequenti allocazioni/deallocazioni dinamiche, oltre all'allocatore standard, è stato utilizzato anche l'allocatore messo a disposizione dalla libreria. Questo allocatore permette di allocare e deallocare dinamicamente la memoria senza utilizzare alcun meccanismo di sincronizzazione fra thread, a patto che sia solo un thread ad allocare la memoria (mentre non vi è nessun limite sul numero di thread che possono effettuare deallocazioni). Ovviamente nei casi in cui vi è la necessità da parte di più thread di effettuare allocazioni dinamiche è stato utilizzato l'allocatore standard (ad esempio nella gestione delle tabelle hash sono molti thread a richiedere l'allocazione di memoria dinamica).

2. Compilazine e parametri di avvio

Il programma per essere compilato necessita della libreria *libpcap* [4] installata sul sistema. Per quanto riguarda fastflow la libreria non necessita di alcuna istallazione, tutte le informazioni necessarie al funzionamento del programma sono contenute nei file .hpp nella directory ./fastflow.

Dopo aver installato libpcap per compilare il programma basta eseguire il comando *make all.* Eventualmente si può effettuare l'installazione con il comando *make install* (e disinstallazione con *make uninstall*).
Per eseguire ffProbe basta digitare il comando *./ffProbe -i interface.*
E' possibile inoltre specificare una serie di parametri opzionali:

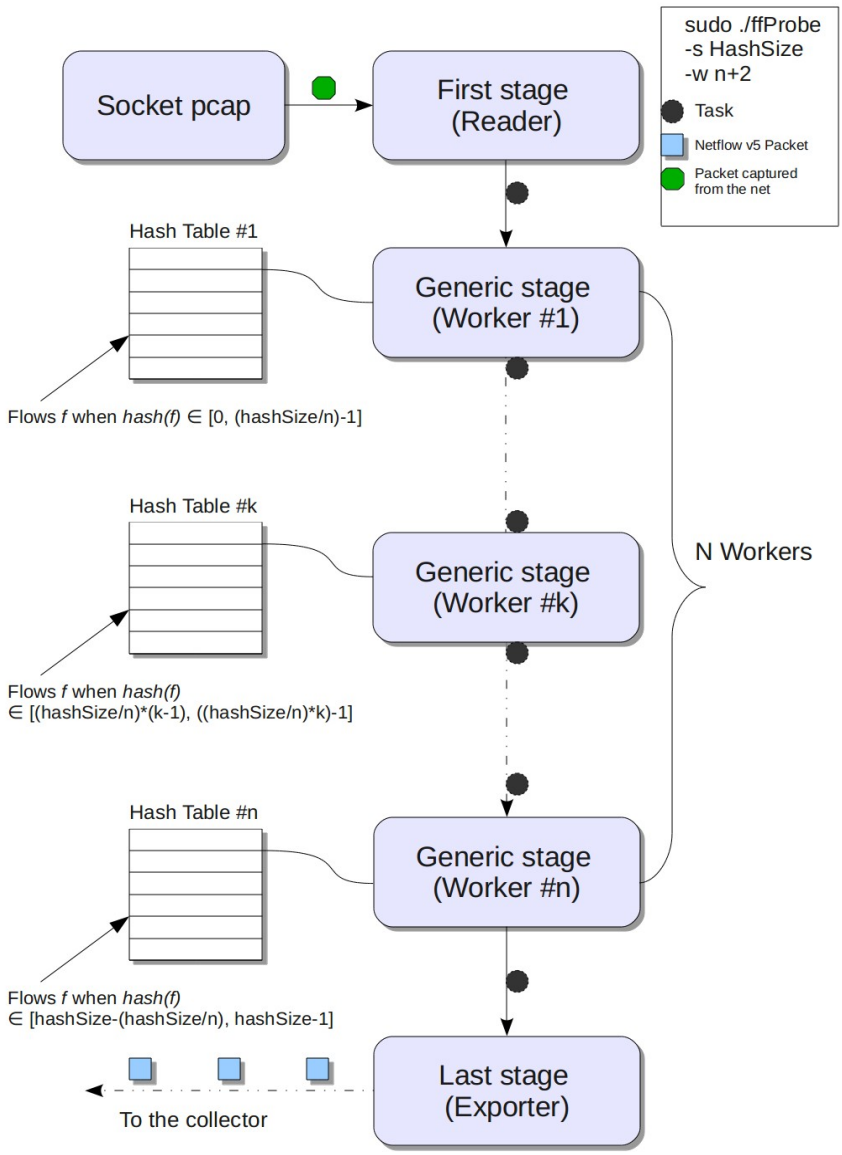
-i <captureInterface|pcap> Nome dell'interfaccia su cui si effettua la cattura dei pacchetti, o nome del file .pcap.
[-b <bpf filter>] Specifica un filtro bpf.
[-d <idleTimeout>] Specifica il tempo massimo (in secondi) di inattività di un flusso [default 30].
[-l <lifetimeTimeout>] Specifica il tempo massimo (in secondi) di vita di un flusso [default 120].
[-q <queueTimeout>] Specifica ogni quanti secondi i flussi scaduti vengono emessi [default 30].
[-t <readTimeout>] Specifica il timeout di lettura (in millisecondi) sul socket pcap [default 30 secondi].
[-w <parDegree>] Specifica quanti thread devono essere attivati [default Esecuzione sequenziale]. HashSize % (parDegree-2) deve essere uguale a 0.
[-s <hashSize>] Specifica la dimensione della tabella hash nella quale i flussi sono memorizzati [default=4096]. HashSize % (parDegree-2) deve essere uguale a 0.
[-m <maxActiveFlows>] Limita il numero di flussi attivi per un worker. E' utile nel caso in cui si vuole limitare la memoria allocata ad ffProbe [default 4294967295].
[-c <cnt>] Il massimo numero di pacchetti da processare prima di ritornare da una lettura. Non è un valore minimo poiché quando si legge da una device, al più vengono letti tutti i pacchetti presenti nel buffer, quindi non si attende l'arrivo di altri pacchetti in modo da arrivare al valore di *cnt*. Se non ci sono pacchetti attende l'arrivo per al più *readTimeout* millisecondi. Con -1 o 0 tutti i pacchetti ricevuti nel buffer sono processati nel caso di cattura da device, mentre nel caso di lettura da un file .pcap, sono processati tutti i pacchetti nel file [default -1].
[-f <outputFile>] Stampa i flussi in formato testuale su un file.
[-a <maxAddCheck>] Numero massimo di flussi da controllare in

seguito all'aggiornamento di un flusso della tabella hash (-1=tutti) [default 1].
[-z <maxNullCheck>] Numero massimo di flussi da controllare nel caso in cui un worker non riceve flussi destinati ad esso (-1=tutti) [default 1] .
[-k <maxReadTOCheck>] Numero massimo di flussi da controllare quando scade un timeout (-1=tutti) [default -1].
[-n <host>] Indirizzo Ip su cui è in esecuzione il collector [default 127.0.0.1].
[-p <port>] Porta su cui è in ascolto il collector [default 2055].
[-y <minFlowSize>] Dimensione minima di un flusso TCP (in byte). Se un flusso TCP è minore della dimensione specificata, il flusso non è emesso. 0=nessun limite [default nessun limite].
[-r] Imposta l'interfaccia in modalità “non promiscua”.
[-h] Stampa l'help.

3. Struttura

La struttura complessiva del programma è quella di una pipeline in cui vi sono tre tipi diversi di stadi. Il primo stadio (*Reader*) si occupa di leggere i pacchetti dal socket pcap e di estrarre da essi le informazioni utili alla generazione dei flussi. Sul socket pcap oltre ai filtri definiti dall'utente è stato inserito un filtro in modo da accettare soltanto il traffico ipv4 (infatti NetFlow v5 non supporta altri protocolli di livello di rete). Il reader può leggere dal socket uno o più pacchetti a seconda del valore specificato tramite l'opzione *[-c <cnt>]*. Inoltre il reader per ogni lettura dal socket genera un oggetto di tipo *task* che invierà tramite un canale di comunicazione al secondo stadio (*Worker*). Un task contiene una lista di flussi scaduti e n liste di pacchetti, dove n è il numero dei worker (cioè (parDegree-2) se parDegree>2, 1 altrimenti) e dove la lista i-esima contiene i pacchetti destinati al worker i-esimo. Il reader decide in quale lista inserire i pacchetti catturati a seconda del valore della funzione hash calcolata su quest'ultimi. Ogni worker si occupa di gestire una porzione della tabella totale, e il reader inserirà nella lista del worker i-esimo i flussi che hanno valore della funzione hash compreso tra *(hashSize/n)*(i-1)* e *((hashSize/n)*i)-1*, dove hashSize è specificato con l'opzione *[-s <hashSize>]*. In realtà è stato scelto di dare ad ogni

worker una propria tabella hash anziché avere un'unica tabella hash condivisa fra tutti i worker. In questo modo i thread non condividono alcun dato fra di loro (tranne ovviamente le code *lock-free* per la comunicazione fra i vari stadi della pipeline e l'allocatore di fastflow). Il worker quindi una volta ricevuto il task, controlla il numero *k* di flussi destinati alla sua tabella hash; se k è uguale a 0 allora controllerà la scadenza di al più *maxNullCheck* flussi. Se k è diverso da 0 il worker aggiornerà i flussi della tabella hash e controllerà la scadenza di al più *maxAddCheck*k* flussi. In entrambi i casi i flussi scaduti vengono aggiunti alla lista nel task.



Infine il task viene inviato sul canale di comunicazione verso lo stadio successivo (che può essere un altro worker oppure l'ultimo

stadio della pipeline). Una volta che il task arriva all'ultimo stadio (*Exporter*), i flussi scaduti vengono aggiunti ad una coda privata. Quando la coda contiene 30 flussi o quando scade il timer specificato in *[-q <queueTimeout>]* i flussi nella coda vengono inseriti in un record NetFlow v5 e spediti al collector remoto (eventualmente vengono anche salvati su un file in formato testuale). Se la coda è vuota allo scadere del timeout non viene inviato nulla. L'allocatore fornito da Fastflow viene utilizzato dal thread reader per allocare alcune strutture dati. I task e le liste di pacchetti da aggiungere alle tabelle hash vengono allocati con l'allocatore di Fastflow poiché una volta create vengono modificate dagli altri thread soltanto per rimuovere elementi.
La lista di flussi scaduti invece viene allocata con l'allocatore standard poiché ogni worker può aggiungere a questa lista nuovi elementi. Sono state utilizzate delle liste personalizzate piuttosto che le liste standard fornite dalla libreria C++ proprio per avere la possibilità di utilizzare l'allocatore di Fastflow. Infine è stata aggiunta la possibilità di aggregare gli stadi della pipeline, vale a dire che si può avere uno stadio (eseguito da un unico thread) che faccia da reader+worker oppure da worker+exporter. In effetti se si esegue il programma con grado di parallelismo pari a due vengono creati due stadi, il primo è un reader mentre il secondo è un worker+exporter. Con grado di parallelismo 1 (caso di default), non viene attivato nessun thread e il programma viene eseguito in modo sequenziale. Con gradi di parallelismo maggiori di due si utilizza invece la struttura classica.
E' da notare inoltre la relazione tra *maxReadTOCheck* e *readTimeout*. Infatti, nel caso in cui non si ricevano pacchetti per più di readTimeout millisecondi, tutti i worker non avranno ricevuto dati per questo intervallo di tempo e quindi non avranno controllato la scadenza di nessun flusso. Nel momento in cui riceveranno la lista di pacchetti vuota, controlleranno al più la scadenza di maxReadTOCheck flussi. E' da sottolineare anche l'importanza di *maxNullCheck* e *maxAddCheck* poiché in caso di una rete con poco traffico, valori piccoli per questi due parametri potrebbero portare a rilevare la scadenza dei flussi troppo tardi. La scelta di tutti questi parametri quindi andrebbe tarata in base alle condizioni di traffico medie della rete.

Per quanto riguarda la parte relativa a NetFlow non è presente nulla di diverso da una classica cache NetFlow. I flussi vengono creati in base alla chiave *<Source Address, Destination Address, L4 Protocol, L4 Source Port, L4 Destination Port, Ip TOS>* e inseriti nella cache come spiegato in precedenza. La funzione hash calcolata sulla chiave è molto semplice e restituisce la somma dei sei valori della chiave modulo la dimensione totale della tabella hash. Non è stata effettuata nessuna analisi riguardo al numero di collisioni generate e quindi questa funzione hash potrebbe non essere la più adatta.
Oltre all'utilizzo dell'allocatore di Fastflow sono stati adottati altri accorgimenti per migliorare le prestazioni complessive:

- Alle funzioni i parametri sono stati passati per copia di puntatori anziché per copia dei valori puntati. Questo permette di ridurre il costo dell'inserimento dei parametri sullo stack.
- E' stato fatto un massiccio uso di funzioni *inline* in modo da evitare l'inserimento di parametri sullo stack durante l'invocazione di alcune funzioni.
- La conversione dei valori da *network byte order* ad *host byte order* è effettuata soltanto nel momento in cui i flussi vengono (eventualmente) salvati su file.
- Come accennato in precedenza non c'è nessun dato condiviso fra i thread e non c'è nessun meccanismo di *lock* (tranne quello previsto dall'allocatore standard di g++).

4. Test

Per verificare la funzionalità del programma sono stati eseguiti diversi test su alcuni file .pcap e sono stati confrontati i risultati con quelli ottenuti facendo processare gli stessi file da *nProbe* [5]. I risultati coincidono perfettamente se si elimina il campo IP TOS dalla chiave dei flussi di ffProbe (sembra infatti che in nProbe nella chiave dei flussi non sia presente il TOS IP). Il programma inoltre è stato controllato con il tool *Valgrind* [6] per verificare l'assenza di leak di memoria.
Infine è stato utilizzato *Wireshark* [7] per controllare la correttezza del formato dei pacchetti esportati verso il collector.
Per quanto riguarda i test sulla performance è stato utilizzato un cavo crossover per collegare due macchine fra di loro. La prima

macchina è stata utilizzata per generare traffico (Intel Core 2 Duo T7300 @ 2.0GHz) tramite *MGEN* [8], mentre l'altra macchina eseguiva ffProbe (Intel Core 2 Duo P8700 @ 2.53GHz).
I risultati ottenuti generando 13 flussi UDP contemporanei (stessa porta sorgente, diversa porta destinazione) sono i seguenti:

Packet size (Bytes)	Kpps	Mbps	%Loss (Sequential execution)	%Loss (2-thread execution)	%Loss (4-thread execution)
64	131	96	0	0	0
64	Random with avg=132	102	0	0	0
128	130,5	167	0	0	0
256	130,5	300	0	0	0
512	130	566	0	0	0
1024	106,8	902	0	0	0
1024	Random with avg=112	950	0	0	0

Inoltre sono stati effettuati alcuni test con *Iperf* [9] ma in questo caso con un solo flusso UDP generato.

Packet size (Bytes)	Kpps	Mbps	%Loss (Sequential execution)	%Loss (2-thread execution)	%Loss (4-thread execution)
64	237,77	122	0	0	0,99
128	251,24	257	0	0,99	0,99
256	176,58	362	0	0	0
512	132,22	542	0	0	0
1024	85,64	702	0	0	0

In quest'ultimo caso si possono notare delle piccole perdite nelle versioni multithreaded dovute ai cambi di contesto fra i thread di ffProbe e i thread che gestiscono il server Iperf. Le perdite sono invece assenti nella versione sequenziale poiché molto

probabilmente i thread di ffProbe e i thread di Iperf sono in esecuzione su due core separati.

5. Conclusioni

Sarebbe stato interessante studiare il comportamento della versione parallela nei casi in cui la versione sequenziale subisce delle perdite. Purtroppo però a causa dell'hardware/software a disposizione non è stato possibile generare una quantità di traffico superiore ai 950Mbps per causare perdite di pacchetti nella versione single-threaded.
Allo stato attuale ffProbe sono assenti alcune funzionalità tra cui:

- Informazioni di routing (Src As, Src Mask, NextHop, ifIndexes, ecc..)
- Supporto ad un numero maggiore di protocolli di livello datalink (per il momento funziona solo su ethernet). Per aggiungere il supporto ad altri protocolli di livello datalink basta aggiungere una clausola nello *switch-case* nel costruttore del primo stage. Nella clausola va inserita la lunghezza dell'header del protocollo datalink che si vuole aggiungere.
- Trattamento della frammentazione IP
- Possibilità di catturare pacchetti provenienti da più interfacce
- Possibilità di effettuare sampling
- Supporto ad altre versioni di NetFlow (v9, IPFIX)

6. Riferimenti

[1] http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/prod_white_paper0900aecd80406232.html
[2] http://en.wikipedia.org/wiki/Algorithmic_skeleton
[3] http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about
[4] http://www.tcpdump.org/
[5] http://www.ntop.org/nProbe.html
[6] http://valgrind.org/
[7] http://www.wireshark.org/
[8] http://cs.itd.nrl.navy.mil/work/mgen/index.php
[9] http://sourceforge.net/projects/iperf/