Acquisizione del traffico IP da una rete ad elevato throughput utilizzando un sistema Open Source: tecnologie e problematiche

Marco Mezzalama, Gianluca Oglietti[1]
Politecnico di Torino – DAUIN
Corso Duca degli Abruzzi, 24 - 10129 Torino (TO)
marco.mezzalama@polito.it
[1]Politecnico di Torino – AreaIT
Corso Duca degli Abruzzi, 24 - 10129 Torino (TO)
gianluca.oglietti@polito.it

L'attività di acquisizione del traffico IP di una rete di calcolatori, attività di fondamentale importanza durante le normali operazioni di monitoring, di troubleshooting e nell'ambito della Sicurezza Informatica, è divenuta negli ultimi anni sempre più complessa a causa non solo del continuo e costante incremento della quantità dei dati scambiati ma anche e soprattutto a causa dell'elevato throughput raggiunto (all'interno di alcuni data center aziendali infatti il throughput è aumentato addirittura di 100 volte negli ultimi 4 anni passando ad esempio dai 100 Mbit/s del 2007 ai 10 Gbit/s del 2011). Questo costante e continuo aumento della velocità del flusso di dati non è però coinciso con un equivalente aumento del clock delle singole CPU che, al contrario, si sono evolute verso architetture di tipo multi-core. Una tale diversità evolutiva ha fin da subito evidenziato una serie di problemi legati soprattutto all'elevatissimo numero di operazioni di I/O che i sistemi avrebbero dovuto gestire a questi livelli di throughput.

1. Introduzione

Le interfacce di rete (Network Interface Controllers, NIC), nonostante il continuo e costante incremento delle velocità delle connessioni dati, sono state per molto tempo equiparate a delle normali periferiche a cui assegnare compiti relativamente semplici come ad esempio l'acquisizione dei singoli pacchetti IP dal livello fisico (modello ISO/OSI) e la copia del loro contenuto nella memoria centrale. Tutto il carico computazione relativo all'elaborazione dei singoli pacchetti veniva quindi demandato alla sola CPU di sistema. Questo approccio è rimasto praticamente immutato per circa 2 decadi e cioè fino a quando il considerevole aumento del throughput sulle reti informatiche ha iniziato a farne emergere una serie di gravi limitazioni strutturali sia in presenza di sistemi con

architetture mono-core che utilizzando architetture multi-core.

In un primo tempo, quando cioè la quasi totalità dei sistemi di acquisizione utilizzati erano realizzati con architetture mono-core ed il throughput di rete non era superiore ai 100Mbit/s, le migliorie che sono state via via apportate ai NIC avevano come unico obbiettivo quello di alleggerire il carico computazione della CPU di sistema. La principale modifica in questa direzione è stata infatti l'introduzione, direttamente all'interno delle schede di rete, di una piccola unità di calcolo specializzata nell'elaborazione dei pacchetti. Parallelamente sono stati poi modificati alcuni aspetti dello stack TCP/IP per diminuire il numero delle richieste di interrupt inviate dalla scheda di rete alla CPU (Interrupt Coalescing) o per ottimizzare alcuni aspetti della trasmissione, o della ricezione, di particolari famiglie di pacchetti IP come ad esempio quelli aventi una dimensione molto superiore al MTU (Maximum Transmission Unit) imposto dalla rete utilizzata (Large Segment Offload).

Con l'aumentare del throughput e con il superamento della soglia del Gbit/s le ottimizzazioni appena descritte si sono rilevate però insufficienti per consentire ai sistemi di acquisizione di gestire una così elevata mole di dati anche in presenza di architetture computazionalmente molto performanti come i sistemi multi-core di tipo NUMA (Non Uniform Memory Access). Per questa ragione sono state apportate una serie di modifiche che hanno interessato profondamente tutta la catena di acquisizione dei pacchetti: le schede di rete (con l'introduzione dei sistemi multi-coda), i driver dei NIC (con lo sviluppo dei driver TNAPI), l'architettura delle schede madri (con l'implementazione di nuove tecnologie come il DCA) e lo stack di rete a livello del kernel del sistema operativo (con lo sviluppo del socket PF_RING).

2. Background

Nei paragrafi seguenti verranno descritte brevemente le principali migliorie apportate negli ultimi anni a tutta la catena di acquisizione dei pacchetti IP.

2.1 Evoluzione hardware in sistemi mono-core

Le interazioni fra la CPU, la memoria di sistema e la scheda di rete, fin dall'avvento dei primi NIC con throughput massimo di 10Mbit/s, sono sempre state relativamente complesse e hanno richiesto lo studio e l'implementazione di numerosi meccanismi per risolvere sistematicamente tutti i problemi via via riscontrati. Il principale di questi meccanismi è la tecnica del Direct Memory Access (DMA) [1] che permette di escludere l'intervento del processore durante le operazioni di lettura o scrittura effettuate dall'interfaccia di rete sulla memoria di sistema. Le interazioni tipiche presenti in un sistema che utilizza il DMA sono mostrate in Figura 1.

I problemi legati a questo genere di architettura sono dovuti al fatto che l'unico processore di sistema, oltre a doversi occupare dell'esecuzione del codice dei programmi a livello utente e a livello kernel, deve anche eseguire buona parte del codice presente nello stack TCP/IP. Durante l'esecuzione del codice poi, in presenza di un elevato throughput, il processore viene

continuamente interrotto dalle richieste di interrupt della scheda di rete: queste interruzioni non fanno altro che generare dei “context switch” della CPU che comportano una repentina diminuzione delle prestazioni dell'intero sistema (il processore trascorre più tempo a gestire gli interrupt della scheda che ad eseguire codice utente).

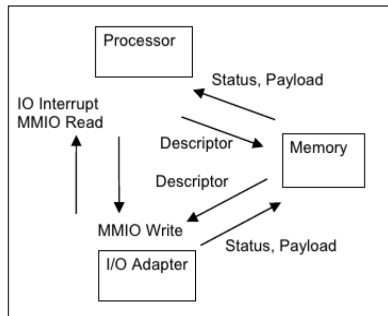

Figura 1 - Iterazioni fra CPU, memoria di sistema e scheda di rete in una architettura mono-core

Per diminuire il carico sul processore di sistema è stata introdotta su tutte le schede di rete una piccola unità di calcolo specializzata sulla quale è stata via via spostata l'esecuzione di buona parte del codice relativo allo stack TCP/IP. Il calcolo del checksum di ogni pacchetto trasmesso e/o ricevuto (Checksum Offload) e la segmentazione dei pacchetti (Large Segment Offload), operazione necessaria per inviare i dati secondo il MTU (Maximum Transmission Unit) della rete, sono due esempi relativi al codice migrato.

Per diminuire il numero di interrupt generati dalla scheda di rete è stato poi introdotto il meccanismo dell'Interrupt Coalescing (o Interrupt Moderation) che permette alla scheda di non contattare il processore ad ogni pacchetto TCP/IP ricevuto ma di contattarlo invece solo quando ne sono stati ricevuti un certo numero (o quando è stato raggiunto un determinato time-out).

Il Checksum Offload, il Large Segment Offload e l'Interrupt Coalescing vengono comunemente raggruppati ed indicati come meccanismi di Traffic Onload Engines (TOE) [2].

Tutti questi meccanismi, assieme all'incremento delle prestazioni delle CPU e delle memorie di sistema, hanno permesso ai normali server di gestire con una discreta efficienza un traffico di rete avente un throughput massimo che, nell'arco dell'ultimo decennio, è passato da 10 Mbit/s a 1 Gbit/s. Questi meccanismi però, anche se utilizzati con le ultimissime CPU messe in commercio, non risultano essere più sufficienti in presenza di throughput ancora più elevati.

2.2 Evoluzione hardware in sistemi multi-core

La necessità di dover incrementare il throughput delle reti informatiche, a causa principalmente dell'avvento di quello che viene definito “Web 2.0” (maggiori contenuti multimediali disponibili in Internet, pagine web più

complesse, VoIP, servizi in real time, P2P, ecc), ha messo in luce tutti quelli che sono i limiti imposti dai meccanismi di tipo TOE. Con l'aumentare del throughput infatti questa soluzione, oltre ad essersi rivelata poco scalabile, poco flessibile e addirittura costosa, si è rivelata essere anche strutturalmente inadatta a sfruttare al meglio le caratteristiche e le potenzialità messe a disposizione dai sistemi multi-core di ultima generazione come ad esempio l'architettura di tipo Non Uniform Memory Access (NUMA) di Intel mostrata in Figura 2.

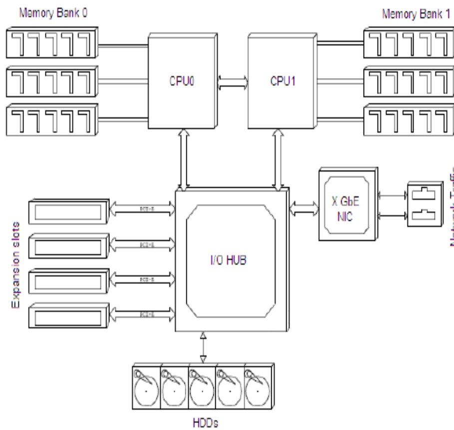

Figura 2 - Architettura di un sistema NUMA di Intel

Le architetture NUMA della famiglia Intel (in questo articolo verranno trattate solo architetture di questo tipo a causa della loro ampia diffusione sul mercato e per alcuni interessanti meccanismi in esse implementati) sono caratterizzate dal fatto che ognuna delle CPU presenti nel sistema contiene al suo interno un controller dedicato per indirizzare una parte della memoria di sistema e dal fatto che per comunicare fra di loro e con il chipset viene utilizzato un bus ad elevate prestazioni chiamato QuickPath Interconnect (QPI).

Se da un lato queste caratteristiche aumentano notevolmente le prestazioni del sistema (diminuendone la latenza) dall'altro la presenza contemporanea di due o più controller per la gestione della memoria di sistema, ne richiede un attento utilizzo. Usando in modo scorretto un'architettura di tipo NUMA si può infatti incorrere ad esempio alla saturazione del bus QPI, a problemi di coerenza della memoria cache dei singoli core e all'incremento sostanziale della latenza durante le operazioni di lettura (o scrittura) in memoria. I nuovi miglioramenti introdotti recentemente ai vari componenti della catena di acquisizione dei pacchetti IP cercano quindi di sfruttare al meglio le caratteristiche messe a disposizione dalle architetture NUMA (e più in generale da tutte le architetture multi-core) per ottenere una maggiore scalabilità e flessibilità rispetto ai meccanismi TOE precedentemente utilizzati.

Per poter sfruttare le caratteristiche messe a disposizione dalle architetture di tipo NUMA sulle nuove schede di rete ad alte prestazioni (con throughput di 1 Gbit/s e 10 Gbit/s) sono state introdotte alcune tecnologie il cui scopo principale

è quello di distribuire in modo ottimizzato il flusso dei dati acquisito su tutti i core presenti sul sistema.

Le principali tecnologie supportate dalle schede di rete di ultima generazione, quindi, sono [3, 4, 5, 6, 7]: Multiple descriptor queues (per gestire più code di pacchetti sulla stessa scheda), Receive Side Scaling o RSS (per suddividere i pacchetti fra le varie code), Virtual Machine Device Queue o VMDq (per assegnare alla medesima coda tutti i flussi di dati diretti verso una stessa macchina virtuale), Extended Message-Signaled Interrupt o MSI-X (per assegnare gli interrupt generati da una coda ad un particolare core), Receive Side Coalescing o RSC (per aggregare più pacchetti provenienti dallo stesso flusso in un pacchetto più grande), Low Latency Interrupts o LLI (per assegnare ad un particolare tipo di traffico IP una maggiore priorità) e Header Splitting and Replication (per ottimizzare la gestione dei pacchetti in memoria).

Le idee che stanno alla base di queste tecnologie sono quelle di:

- creare all'interno della scheda di rete alcune code indipendenti di pacchetti;
- assegnare in modo univoco ogni pacchetto ricevuto alla coda corretta;
- assegnare ad ogni core di sistema (o macchina virtuale) la coda di pacchetti corretta;
- facilitare l'interazione fra il sistema, le code e i vari core.

Queste tecnologie non sono però ancora sufficienti per poter utilizzare al meglio le potenzialità messe a disposizione dai sistemi di calcolo con architettura NUMA. L'elevato numero di interrupt generato dalle code di pacchetti riducono infatti notevolmente le performance del sistema in quanto i vari core devono occupare molti dei loro cicli di clock per effettuare le operazioni di I/O necessarie a spostare i pacchetti dalle code alla memoria di sistema.

Per risolvere questo problema Intel è andata a modificare alcuni componenti dell'architettura NUMA introducendo una tecnologia, o meglio, un'insieme di tecnologie chiamate Input/Output Acceleration Tecnology (I/OAT) [8, 9, 10]. Queste nuove tecnologie contribuiscono ad aumentare le prestazioni di un sistema di acquisizione di pacchetti IP andando ad ottimizzare l'utilizzo della memoria centrale e della memoria chace di ogni processore. Le tecnologie che sono alla base dell'I/OAT sono principalmente due: Intel QuickData Technology e Direct Cache Access (DCA).

L'Intel QuickData Tecnology è stata introdotta in quanto buona parte del tempo necessario ad elaborare un pacchetto appena ricevuto veniva impiegato (inutilmente) dalla CPU per spostare dati all'interno della memoria di sistema. In una situazione normale infatti, quando il processore esegue la copia o lo spostamento dei dati contenuti nella memoria di sistema, deve rimanere di fatto bloccato fino al termine dell'operazione. Usando questa tecnologia invece il processore non deve attendere il completamento dell'operazione di copia in quanto essa viene demandata al motore DMA inserito direttamente nel controller. Come è possibile osservare in Figura 3 la CPU, una volta istruito il controller (AMC) sull'operazione da eseguire, non rimane in stallo ad attendere che il dato venga copiato ma può iniziare ad elaborare l'istruzione successiva

(ad esempio può iniziare ad elaborare il prossimo pacchetto). Sarà poi lo stesso controller ad avvisare il processore quando la copia richiesta in precedenza è stata ultimata.

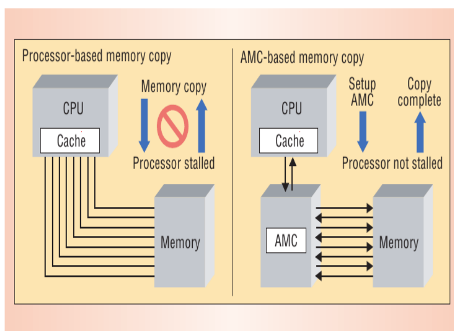

Figura 3 - Intel QuickData Tecnology

Il Direct Cache Access è stato introdotto per migliorare le prestazioni del sistema durante il trasferimento dei dati appena acquisiti dalla scheda di rete verso la memoria cache del processore designato per la loro elaborazione.

In un normale sistema di acquisizione infatti, come è possibile osservare dalla Figura 4, il dato acquisito dalla scheda di rete deve transitare per la memoria di sistema (DMA) prima di poter raggiungere la memoria cache del processore e aggiungendo quindi all'operazione una notevole latenza. Oltre ai problemi di latenza bisogna poi tenere in considerazione un ulteriore overhead dovuto ai meccanismi che mantengono la coerenza dei dati contenuti nella cache. Il DCA risolve questi problemi permettendo alla scheda di rete di scrivere il dato appena acquisito direttamente all'interno della cache del processore corretto senza dover passare dalla memoria di sistema. I principali benefici del DCA sono due: la considerevole riduzione del tempo necessario al dato appena acquisito per raggiungere il cuore dell'unità di calcolo e la riduzione della banda utilizzata dalla memoria di sistema.

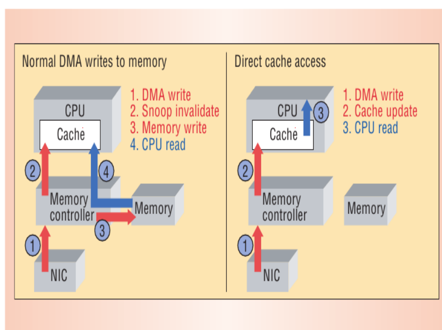

Figura 4 - Direct Cache Access

2.3 Evoluzione Software – il socket PF_RING

Uno dei primi problemi che hanno interessato il sistema operativo Linux in

ambito networking è stato riscontrato nel ramo di sviluppo 2.4 del kernel (in modo particolare nelle versioni precedenti alla 2.4.20) ed era caratterizzato da un'evidente degradazione delle prestazioni del sistema in presenza di un elevato traffico di rete (dell'ordine dei 100 Mbit/s). La causa di questo problema è stata attribuita al fatto che, in queste condizioni, il sistema operativo era impegnato per la quasi totalità del tempo a gestire gli interrupt generati dal NIC all'arrivo dei pacchetti a scapito dei processi in esecuzione a cui era invece riservato il poco tempo restante.

Per risolvere questo problema e per migliorare le prestazioni del sistema, a partire dal ramo di sviluppo 2.5 di Linux, è stata introdotta una nuova interfaccia di comunicazione fra il kernel e i driver dei dispositivi di rete chiamata NAPI (New API) che, fra le varie caratteristiche, introduceva l'utilizzo di una tecnica di mitigazione degli interrupt basata sul metodo del Polling. La caratteristica principale di questa nuova tecnica, infatti, è che i driver del dispositivo non possono inviare un interrupt al sistema in un qualsiasi istante di tempo ma solo dopo essere stati interrogati, ad intervalli più o meno regolari, dal kernel stesso (Interrupt Mitigation).

Alcuni studi sull'acquisizione passiva dei pacchetti IP su reti ad elevato throughput [11] effettuati negli ultimi anni da Luca Deri hanno però evidenziato che molti pacchetti non vengono acquisiti dal sistema in quanto il kernel Linux con interfaccia NAPI impiega un tempo non trascurabile per spostare fisicamente un pacchetto dal device al software in userspace. Per risolvere questo problema Deri ha quindi introdotto un nuovo socket chiamato PF_RING [13] che permette di diminuire il tempo impiegato dal pacchetto ad attraversare le code e le strutture dati presenti all'interno del kernel Linux. Questo socket, come è possibile osservare in Figura 5, è basato su di un buffer circolare che viene allocato in memoria durante l'inizializzazione del socket stesso e viene deallocato solamente alla sua chiusura eliminando quindi la necessità di effettuare operazioni di allocamento della memoria ogni volta che vengono ricevuti dei pacchetti dalla scheda di rete.

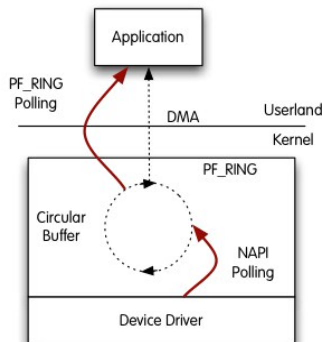

Figura 5 - Il socket PF_RING

Quando un pacchetto viene ricevuto dall'interfaccia di rete (tramite il NAPI Polling) esso, invece di essere gestito dagli strati superiori del kernel, viene

inserito direttamente all'interno del buffer circolare che viene poi esportato a livello utente in modo tale che le applicazioni possano accedervi direttamente per prelevare i pacchetti. Più applicazioni possono accedere allo stesso socket: in questo caso il PF_RING fornirà un buffer circolare ad ogni applicazione evitando quindi l'insorgere di eventuali interferenze fra di esse (la presenza di applicazioni intrinsecamente lente non andranno a rallentare applicazioni più veloci durante la fase di lettura dei pacchetti presenti nei buffer circolari).

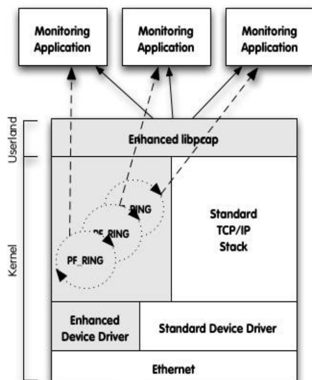

Figura 6 - PF_RING e Clustering

Il codice alla base del socket PF_RING offre poi una serie di funzionalità molto utili durante l'acquisizione passiva del traffico IP di una rete informatica. In particolare infatti il socket permette di filtrare i pacchetti inviati dal driver della scheda prima che vengano inseriti all'interno del buffer circolare, e di incrementare le performance dei software di acquisizione ed elaborazione dei pacchetti grazie ai meccanismi di bilanciamento e di clustering. Come è possibile osservare in Figura 6, questi meccanismi permettono di inviare ad una applicazione solo una porzione dell'intero flusso di pacchetti e di instradare il flusso rimante alle altre applicazioni che fanno parte del cluster. Un cluster di applicazioni può essere configurato attraverso delle semplici regole in base al flusso dei dati o attraverso la tecnica del roud-robin.

Il socket PF_RING può essere utilizzato da un qualsiasi device driver ma per alcune periferiche sono stati sviluppati alcuni driver ad hoc (nel seguito indicati come PF_RING-aware) che permettono di copiare i pacchetti appena acquisiti direttamente all'interno del socket stesso senza utilizzare altri meccanismi intermedi messi a disposizione dal sistema operativo. Esistono infine alcuni driver, sempre sviluppati da Deri, che permettono di mappare la memoria della scheda di rete a livello utente (driver PF_RING DNA, Direct NIC Access): in questo modo, come è possibile osservare in Figura 7, i pacchetti vengono copiati all'interno del buffer circolare direttamente dal processore presente sulla scheda di rete senza passare dall'interfaccia NAPI del sistema operativo. Questa ulteriore ottimizzazione però può essere utilizzata solamente da un'applicazione alla volta.

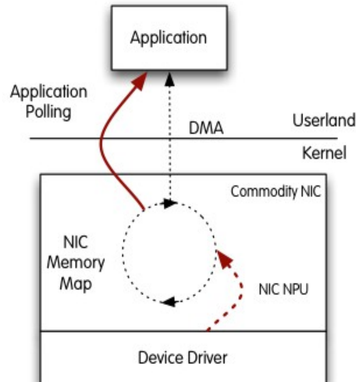

Figura 7 - PF_RING e driver PF_RING DNA

Le applicazioni che già si interfacciano alle schede di rete utilizzando le librerie libpcap, per poter utilizzare il socket PF_RING, non devono essere riscritte o modificate ma devono essere semplicemente ricompilate utilizzando delle particolari librerie chiamate libpcap-PF_RING che permettono di accedere in modo trasparente al nuovo socket PF_RING.

Il socket PF_RING, per garantire la compatibilità con tutti i driver sviluppati in precedenza, può essere avviato in tre diverse modalità di funzionamento configurabili tramite il parametro “transparent_mode”:

- modalità 0: in questa modalità i pacchetti sono inviati al buffer circolare utilizzando i meccanismi standard del kernel. Questa modalità è quella di default ed è utilizzabile da tutti i driver delle schede di rete.
- modalità 1: in questa modalità i pacchetti sono inviati al buffer circolare direttamente dal driver PF_RING-aware della scheda di rete, una copia del pacchetto viene comunque ancora passata al kernel utilizzando i meccanismi standard.
- modalità 2: in questa modalità i pacchetti sono inviati al buffer circolare direttamente dal driver PF_RING-aware della scheda di rete.

La modalità 2 risulta essere la più performante ma è necessario sottolineare che essa comporta l'assenza di connettività per le schede di rete che la utilizzano in quanto i pacchetti ricevuti non vengono inviati al kernel tramite i meccanismi standard.

2.4 Evoluzione Software – i driver TNAPI

Come si è visto nei paragrafi precedenti, le moderne schede di rete ad elevato throughput hanno una struttura interna che permette di suddividere il flusso dei pacchetti acquisiti su più code utilizzando ad esempio alcune funzionalità hardware come il Receive Side Scaling (RSS) di Intel. Queste tecniche però, essendo nate sopratutto per essere utilizzate in ambienti virtualizzati, non sono comunemente sfruttate per il monitoraggio del traffico di

rete anche perché i driver a disposizione non permettono ai programmi a livello utente di sfruttare appieno questo tipo di parallelizzazione. Quello che solitamente accade infatti, come si può vedere in Figura 8, è che i normali driver disponibili per queste periferiche fondono i vari flussi di dati provenienti dalle diverse code della scheda in un unico flusso offrendo ai software (anche se multi-thread) un solo punto di accesso ai dati acquisiti (ad esempio la classica interfaccia eth1). Questo è sicuramente un collo di bottiglia di notevole entità.

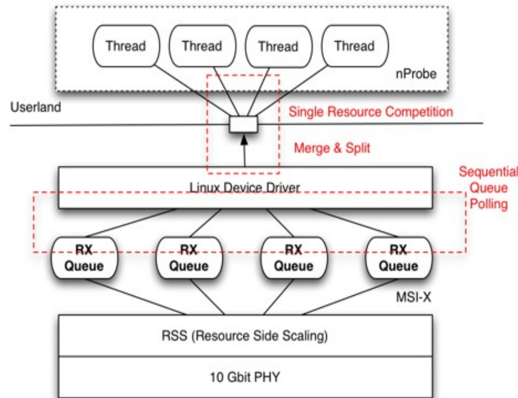

Figura 8 - Driver standard con schede multi-coda

Si pensi ad esempio ad un software multi-thread (ottimizzato cioè per essere eseguito su architetture multi-core) dove i vari thread devono sincronizzarsi fra di loro (di solito utilizzando i semafori) per poter accedere all'unica sorgente di pacchetti disponibile: all'aumentare del numero di thread, contrariamente alle attese, è possibile osservare che le prestazioni del software diminuiscono considerevolmente. Questo accade essenzialmente per due motivi:

- i vari thread trascorrono la maggior parte del tempo a scambiarsi fra loro i segnali di sincronizzazione. Solo un thread alla volta infatti può trovarsi nello stato di “ready” (può cioè accedere all'unica fonte di pacchetti disponibile) mentre tutti gli altri devono obbligatoriamente essere in uno stato di “busy” (sono cioè in uno stato di attesa dove non eseguono alcuna operazione)
- la modalità di accesso alla fonte dei dati, essendo puramente casuale,

non permette di assegnare ad uno stesso core sia l'interrupt generato dalla coda della scheda di rete in cui transita un determinato pacchetto che il thread dell'applicazione che lo elaborerà. Questa casualità strutturale comporta quindi la mancata ottimizzazione della cache dei vari core rendendo necessario un uso intensivo della memoria di sistema (a causa dei numerosi context switch) con una conseguente degradazione delle prestazioni dell'applicazione.

Per risolvere questi problemi Deri ha sviluppato un nuovo driver multi-thread (disponibile solo per alcune periferiche di ultima generazione) chiamato TNAPI [12, 14] (Thread-NAPI) che, come è possibile osservare in Figura 9, associa un

thread di accesso ai dati ad ognuna delle code disponibili sull'interfaccia di rete.

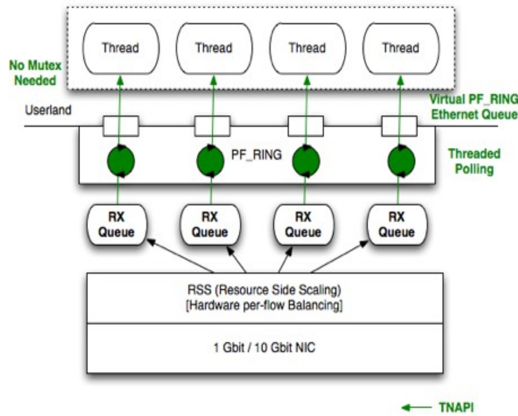

Figura 9 - Driver TNAPI con schede multi-coda

In questo modo è possibile introdurre il concetto di adattatore di rete virtuale (indicato ad esempio con il device linux eth2@4 che sta ad indicare la coda numero 4 dell'interfaccia di rete eth2) fornendo quindi l'accesso alle singole code a livello utente. Le varie applicazioni, in presenza dei driver TNAPI, potranno quindi accedere ai pacchetti acquisiti dal NIC in due modi: utilizzando il device fisico (es. eth1) per accedere ai pacchetti acquisiti da tutte le code o utilizzando i device virtuali (es. eth1@3) per accedere alle singole coda senza la necessità di sincronizzare i vari thread con i semafori.

2.5 SMP IRQ Affinity

Un'importante fattore di cui bisogna tenere conto quando si vuole acquisire del traffico di rete ad elevato throughput è l'associazione fra gli interrupt generati dai NIC (o dalle code presenti nelle schede) all'arrivo dei pacchetti IP e il core che dovrà elaborarli. Tale associazione prende il nome di IRQ affinity o SMP affinity [15, 16] e può essere effettuata in diversi modi in base a quale dei seguenti obbiettivi si intende raggiungere:

- massimizzare il livello di coerenza della cache di ogni core
- migliorare la distribuzione degli interrupt fra i vari core

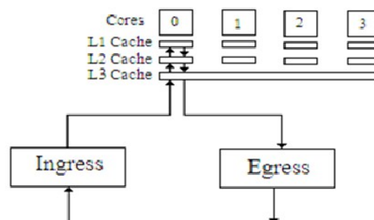

Figura 10 - SMP Affinity per l'ottimizzazione della cache

In Figura 10 è possibile osservare la configurazione di IRQ Affinity da

adottare per massimizzare la coerenza della cache mentre in Figura 11 è rappresenta la configurazione di IRQ Affinity per migliorare la distribuzione degli interrupt fra i vari core.

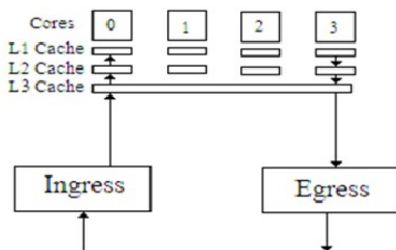

Figura 11 - SMP Affinity per migliorare la distribuzione degli interrupt

Le due configurazioni sono sostanzialmente differenti in quando la prima tende ad assegnare gli interrupt ad uno stesso core (migliorando le prestazioni di un sistema NUMA ma peggiorando le prestazioni in caso di elevato throughput) mentre la seconda a distribuirli fra tutti i core disponibili (migliorando le prestazioni in caso di elevato throughput ma peggiorandole in presenza di un sistema NUMA). La configurazione che deve essere adottata nella pratica dipende da molti fattori come ad esempio il tipo di applicazione in esecuzione sui core, il tipo di traffico da acquisire ed il suo throughput: in alcuni casi le prestazioni migliori si ottengono massimizzando il livello di coerenza delle cache mentre altre volte migliorano distribuendo gli interrupt su più core, altre volte ancora utilizzando una configurazione ibrida.

In Figura 12 è possibile infine osservare la configurazione consigliata che dovrebbe essere adottata in presenza di un sistema Simultaneous Multi Threading (SMT) di Intel dove ogni core fisico, utilizzando la tecnologia HT (Hyper Threading) è affiancato da un core logico (il numero dei core utilizzabili è doppio rispetto al numero di core fisici). In questo caso è consigliabile, quando possibile, associare tutti gli interrupt diretti al core fisico e al relativo core logico in quanto condividono la stessa memoria cache.

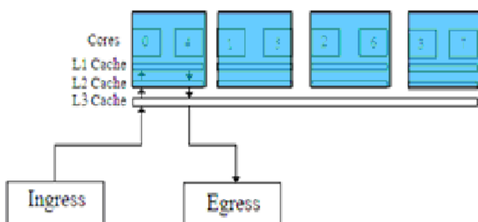

Figura 12 - SMQ Affinity consigliata con sistema SMT

3. Realizzazione di un sistema di acquisizione

Per poter effettuare alcuni test sulle tecnologie analizzate nei paragrafi precedenti si è realizzato un sistema di acquisizione per reti ad elevato throughput (fino a 10Gbit/s).

Nei paragrafi seguenti quindi sono state descritte le caratteristiche principali dei componenti (scheda di rete, server, sistema operativo e driver) che sono stati utilizzati per la realizzazione del sistema di acquisizione del traffico IP.

3.1 Scheda di rete

La scheda di rete utilizzata è il modello “X520-SR2” [17] di Intel.

Questo NIC è composto da due interfacce in fibra da 10 Gbit/s (tramite adattatori GBIC) ed è basato sul chip 82599 che risulta essere uno degli ultimi chip messi in commercio da Intel. Le caratteristiche principali di questa scheda sono:

- Multiple queues: 128 code di pacchetti in ricezione e 128 code in trasmissione (vengono attivate solo una quantità di code pari al numero di core del sistema)
- Receive (and Transmit) Side Scaling per i protocolli IP (v4 e v6), TCP/UDP
- Checksum Offload per i protocolli IP (v4 e v6), SCTP, TCP e UDP
- Header Splits and Replication per il traffico ricevuto
- Advanced Packet Filtering (filtraggio dei pacchetti in hardware)
- Low Latency Interrupts
- Intel Direct Cache Access (DCA)
- Supporto agli interrupt MSI-X
- Interfaccia PCI-Express 2.0 (5.0 GT/s)
- Processore dedicato: Intel 82599

3.2 Server di acquisizione

Il server utilizzato è il modello “SYS-6026T-6RFT+” della casa produttrice SuperMicro.

In un primo tempo si era pensato di utilizzare un server della casa produttrice HP appartenente alla famiglia dei DL-380 ma, purtroppo, analizzando il BIOS del sistema ci si è subito accorti che non supportava alcune tecnologie fondamentali come l'I/OAT di Intel. Per questo motivo è stato scelto il server SuperMicro che, per contro, permette una configurazione molto più puntuale (attraverso il BIOS) di molti parametri dell'hardware del sistema. Le principali caratteristiche di questo server sono:

- CPU: 2 x Intel Xeon X5560 @ 2.80GHz (16 core)
- Architettura NUMA con bus QPI (6.4 GT/s)
- RAM: 12 GB DDR3 @ 1333 MHz ECC
- Chipset: Intel 5520 e ICH10R
- Supporto completo alla tecnologia I/OAT 3 e VMDq

3.3 Sistema operativo, driver e software

Per garantire la massima versatilità e sicurezza possibile, sul server SuperMicro “SYS-6026T-6RFT+”, è stato installato il sistema operativo Open

Source GNU/Linux Debian 6.0.2 (Squeeze, stable) per architetture a 64bit con kernel Linux 2.6.38.8 compilato manualmente per abilitare tutti i moduli necessari al corretto funzionamento del sistema (supporto ad architetture NUMA, modulo ioatdma per la gestione della tecnologia QuickData di Intel, modulo dca per il supporto alla tecnologia Direct Cache Access, …).

Per poter utilizzare tutte le funzionalità messe a disposizione dalla scheda di rete scelta sono stati utilizzati i driver TNAPI (versione 3.3.8) e il socket PF_RING (versione 4.7.0 – revisione 4701) [18].

Per le varie prove infine sono stati utilizzati applicativi messi a disposizione dalla distribuzione Debian e applicativi presenti all'interno del pacchetto del socket PF_RING distribuito da Deri.

Va evidenziato quindi il fatto che il sistema di acquisizione è stato realizzato interamente utilizzando solo software Open Source.

4 Problematiche riscontrate

La scelta di utilizzare solamente codice Open Source all'interno di un progetto di questo tipo (utilizzo di nuove tecnologie ad elevate prestazioni) è dettata dal fatto che questa tipologia di software porta con se notevoli benefici. Si pensi, ad esempio, all'opportunità di poter accedere al codice sorgente del software per poterlo migliorare correggendo, praticamente in “real time”, gli eventuali errori di programmazione individuati durante le varie fasi di test o aggiungendo via via ad esso tutte le funzionalità di cui si ha bisogno: operazioni che, nella stragrande maggioranza dei casi, risultano essere assolutamente impossibili da realizzare utilizzando software a sorgenti chiusi.

Durante i test effettuati sul sistema di acquisizione appena descritto sono stati però evidenziati alcuni problemi in qualche modo legati alla tipologia di software utilizzato. Tali problemi verranno descritti nei seguenti paragrafi.

4.1 Evoluzione del software Open Source

Una delle caratteristiche peculiari di un software Open Source è la sua modalità di evoluzione. Nella maggior parte dei casi infatti un software di questo tipo è caratterizzato da un'elevatissima velocità di evoluzione soprattutto durante le prime fasi del suo sviluppo. Una così estrema rapidità evolutiva risulta essere però un problema considerevole soprattutto quando il software Open Source viene impiegato in ambienti in cui è necessario utilizzare nuove tecnologie (si pensi ad esempio all'I/OAT di Intel o al socket PF_RING) o dispositivi hardware relativamente recenti (si pensi alle schede di rete 10Gbit/s che implementano la tecnologia multi-coda).

Uno dei problemi riscontrati con maggior frequenza, durante i numerosi test effettuati sul sistema di acquisizione di pacchetti IP implementato, è stato quello di dover continuamente aggiornare una buona parte del software utilizzato a causa delle numerose e frequenti modifiche apportate dagli sviluppatori al codice del socket PF_RING o al codice del driver TNAPI. Il continuo aggiornamento di questi software si è rivelato assolutamente necessario in quanto le modifiche apportate erano focalizzate principalmente a correggere

eventuali errori di programmazione, a ottimizzare alcune parti del codice sorgente o ad aggiungere delle nuove funzionalità.

Saltuariamente, a causa di un semplice aggiornamento del codice del socket PF_RING, era possibile ad esempio riscontrare problemi di compatibilità con i driver TNAPI. Aggiornando quindi i driver TNAPI, per allinearli alla versione del socket PF_RING installata, a volte venivano però riscontrati problemi di compatibilità con il kernel in uso. Aggiornando infine il kernel del sistema operativo era ancora possibile riscontrare alcuni problemi con altre parti del sistema che risultavano essere compatibili, ad esempio , solo con una versione predente del kernel Linux.

Problematiche di questo tipo sono del tutto assenti quando si utilizza codice a sorgente chiuso dove, il più delle volte, gli aggiornamenti del software o dei driver utilizzati vengono rilasciati ad intervalli di tempo più ampi e dove il kernel del sistema operativo non viene in pratica mai interessato.

4.2 Kernel Panic

Durante alcuni dei test effettuati con il server per l'acquisizione del traffico IP ad elevato throughput è stata riscontrata la presenza di un grave problema a livello del kernel del sistema operativo Linux. In particolare, inviando al server di acquisizione una grande quantità di pacchetti frammentati (generati utilizzando il software hping3), è stato possibile riscontrare saltuariamente alcuni blocchi dell'intero sistema a causa del “kernel panic” qui riportato:

```
[76051.431130] kernel BUG at net/ipv4/inetpeer.c:386!
[76051.488428] invalid opcode: 0000 [#1] SMP
[76051.537630] last sysfs file: /sys/bus/pci/drivers/ixgbe/uevent
[76051.607385] CPU 2
[76051.629288] Modules linked in: pf_ring ixgbe microcode loop usbhid hid uhci_hcd ioatdma igb ehci_hcd usbcore sg psmouse rtc_cmos rtc_core sr_mod
i2c_i801 cdrom i2c_core rtc_lib dca evdev mdio serio_raw [last unloaded: pf_ring]
[76051.874470]
[76051.892326] Pid: 0, comm: kworker/0:1 Tainted: G        W   2.6.38.8-superechelon #1 Supermicro X8DTU-6+/X8DTU-6+
[76052.015641] RIP: 0010:[<ffffffff8127918f>]  [<ffffffff8127918f>] cleanup_once+0x13f/0x210
[76052.113527] RSP: 0018:ffff8800bf4839f0  EFLAGS: 00010287
[76052.177055] RAX: ffff8801b4c4b000 RBX: ffffffff813e74e0 RCX: 0000000000000001
[76052.262381] RDX: ffff8800bf483ad0 RSI: 0000000000000000 RDI: 0000000046d553d5
[76052.347707] RBP: ffffffff813e74e8 R08: 0000000046d74848 R09: 0000000000000012
[76052.433032] R10: ffffffff813e74e0 R11: 0000000000000013 R12: ffff8800bf483a40
[76052.518358] R13: ffffffff8130b080 R14: ffff880171c3a8a8 R15: ffff880171c3a880
[76052.603683] FS:  0000000000000000(0000) GS:ffff8800bf480000(0000) knlGS:0000000000000000
[76052.700427] CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
[76052.769144] CR2: 00007fd7826ad000 CR3: 00000000013b9000 CR4: 00000000000006e0
[76052.854470] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[76052.939797] DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
[76053.025122] Process kworker/0:1 (pid: 0, threadinfo ffff8801b90a8000, task ffff8801b907d780)
[76053.126018] Stack:
[76053.150101]  ffff8800bf483a40 ffff88017d7a9d00 ffffffff813e74e0 ffffffff813e74e8
[76053.239058]  ffff8800bf483a40 ffff8800bf483bc0 ffff8800bf483ad8 ffffffff81279404
[76053.328016]  0000000000000000 00000001383aa170 ffffffff813e74e0 ffff88033818c400
[76053.416975] Call Trace:
[76053.446250]  <IRQ>
[76053.471473]  [<ffffffff81279404>] ? inet_getpeer+0x1a4/0x260
[76053.539153]  [<ffffffff81277d37>] ? ip_route_input_slow+0x307/0x790
[76053.614099]  [<ffffffff8127a1b9>] ? ip4_frag_init+0x89/0xa0
[76053.680742]  [<ffffffff812add87>] ? inet_frag_find+0x97/0x230
[76053.749456]  [<ffffffff8127a316>] ? ip_defrag+0xd6/0xb80
[76053.812984]  [<ffffffff81279d28>] ? ip_local_deliver+0x28/0x1a0
[76053.883779]  [<ffffffff81257639>] ? __netif_receive_skb+0x2f9/0x3d0
[76053.958723]  [<ffffffff8125bfa8>] ? netif_receive_skb+0x78/0x80
[76054.029517]  [<ffffffff8125c52b>] ? napi_gro_receive+0xbb/0xd0
[76054.099272]  [<ffffffff8125c0d8>] ? napi_skb_finish+0x38/0x50
[76054.167993]  [<ffffffffa0155a6c>] ? ixgbe_receive_skb+0x1cc/0x1e0 [ixgbe]
[76054.249164]  [<ffffffff81184641>] ? unmap_single+0x31/0x90
[76054.314769]  [<ffffffffa0159526>] ? ixgbe_poll+0x636/0x14a0 [ixgbe]
[76054.389712]  [<ffffffffa0159e21>] ? ixgbe_poll+0xf31/0x14a0 [ixgbe]
[76054.464657]  [<ffffffff8125c676>] ? net_rx_action+0x86/0x170
[76054.532336]  [<ffffffff812407dc>] ? dma_issue_pending_all+0x7c/0xb0
[76054.607282]  [<ffffffff8103e6f1>] ? __do_softirq+0x91/0x140
[76054.673924]  [<ffffffff8100334c>] ? call_softirq+0x1c/0x30
[76054.739527]  [<ffffffff8100520d>] ? do_softirq+0x4d/0x80
[76054.803053]  [<ffffffff810047ac>] ? do_IRQ+0x5c/0xd0
```

```
[76054.862429]  [<ffffffff812d10d3>] ? ret_from_intr+0x0/0xe
[76054.926993]  <EOI>
[76054.952219]  [<ffffffff8119a524>] ? intel_idle+0xb4/0x110
[76055.016783]  [<ffffffff8119a504>] ? intel_idle+0x94/0x110
[76055.081348]  [<ffffffff8123f101>] ? cpuidle_idle_call+0x81/0xf0
[76055.152140]  [<ffffffff81001640>] ? cpu_idle+0x50/0xa0
[76055.213590] Code: 20 02 0f 95 c1 31 c0 8d 4c 49 01 31 f6 41 8b 7c 07 10 45 8b 44 05 10 44 39 c7 75 c0 ff c6 48 83 c0 04 39 ce 7c e7 4d 39 ef 74 18 <0f>
0b eb fe 48 c7 c7 d0 74 3e 81 e8 b1 7b 05 00 83 c8 ff e9 6e
[76055.446005] RIP  [<ffffffff8127918f>] cleanup_once+0x13f/0x210
[76055.515865]  RSP <ffff8800bf4839f0>
[76055.557910] ---[ end trace 06a70dacbf2d08ad ]---
[76055.613128] Kernel panic - not syncing: Fatal exception in interrupt
[76055.689113] Pid: 0, comm: kworker/0:1 Tainted: G      D W   2.6.38.8-superechelon #1
[76055.781808] Call Trace:
[76055.811081]  <IRQ>  [<ffffffff812ce4d4>] ? panic+0x92/0x18a
[76055.877930]  [<ffffffff810393c1>] ? kmsg_dump+0x41/0x100
[76055.941458]  [<ffffffff8100661b>] ? oops_end+0x9b/0xa0
[76056.002909]  [<ffffffff81003f74>] ? do_invalid_op+0x84/0xa0
[76056.069549]  [<ffffffff8127918f>] ? cleanup_once+0x13f/0x210
[76056.137230]  [<ffffffff810030d5>] ? invalid_op+0x15/0x20
[76056.200757]  [<ffffffff8127918f>] ? cleanup_once+0x13f/0x210
[76056.268436]  [<ffffffff81279404>] ? inet_getpeer+0x1a4/0x260
[76056.336115]  [<ffffffff81277d37>] ? ip_route_input_slow+0x307/0x790
[76056.411059]  [<ffffffff8127a1b9>] ? ip4_frag_init+0x89/0xa0
[76056.477702]  [<ffffffff812add87>] ? inet_frag_find+0x97/0x230
[76056.546420]  [<ffffffff8127a316>] ? ip_defrag+0xd6/0xb80
[76056.609945]  [<ffffffff81279d28>] ? ip_local_deliver+0x28/0x1a0
[76056.680739]  [<ffffffff81257639>] ? __netif_receive_skb+0x2f9/0x3d0
[76056.755686]  [<ffffffff8125bfa8>] ? netif_receive_skb+0x78/0x80
[76056.826478]  [<ffffffff8125c52b>] ? napi_gro_receive+0xbb/0xd0
[76056.896232]  [<ffffffff8125c0d8>] ? napi_skb_finish+0x38/0x50
[76056.964952]  [<ffffffffa0155a6c>] ? ixgbe_receive_skb+0x1cc/0x1e0 [ixgbe]
[76057.046124]  [<ffffffff81184641>] ? unmap_single+0x31/0x90
[76057.111730]  [<ffffffffa0159526>] ? ixgbe_poll+0x636/0x14a0 [ixgbe]
[76057.186674]  [<ffffffffa0159e21>] ? ixgbe_poll+0xf31/0x14a0 [ixgbe]
[76057.261619]  [<ffffffff8125c676>] ? net_rx_action+0x86/0x170
[76057.329298]  [<ffffffff812407dc>] ? dma_issue_pending_all+0x7c/0xb0
[76057.404243]  [<ffffffff8103e6f1>] ? __do_softirq+0x91/0x140
[76057.470884]  [<ffffffff8100334c>] ? call_softirq+0x1c/0x30
[76057.536488]  [<ffffffff8100520d>] ? do_softirq+0x4d/0x80
[76057.600015]  [<ffffffff810047ac>] ? do_IRQ+0x5c/0xd0
[76057.659391]  [<ffffffff812d10d3>] ? ret_from_intr+0x0/0xe
[76057.723954]  <EOI>  [<ffffffff8119a524>] ? intel_idle+0xb4/0x110
[76057.795995]  [<ffffffff8119a504>] ? intel_idle+0x94/0x110
[76057.860561]  [<ffffffff8123f101>] ? cpuidle_idle_call+0x81/0xf0
[76057.931352]  [<ffffffff81001640>] ? cpu_idle+0x50/0xa0
```

Sono state effettuate numerose prove per identificare la causa di questo errore di sistema. In particolare per prima cosa si è deciso di verificare se la causa del problema fosse il codice sviluppato da Deri. Per questa ragione è stato subito disinstallato il socket PF_RING e, di conseguenza, il driver TNAPI è stato sostituito dall'ultima versione del driver standard, per la scheda di rete Intel X520-SR2, disponibile sul sito del produttore. Anche in questo caso, inviando una grande quantità di pacchetti frammentati al server di acquisizione è stato possibile riscontrare l'errore di kernel panic riportato in precedenza con il conseguente blocco del sistema.

Una volta esclusa l'ipotesi che la causa del problema fosse da attribuire al nuovo codice del socket PF_RING o ai driver TNAPI sono state quindi effettuate delle ulteriori prove andando a modificare la IRQ Affinity fra le code della scheda di rete e i core disponibili sui due processori installati sul server. Da questi ultimi test si è ottenuto un risultato abbastanza interessante. Assegnando infatti ad un unico core tutti gli interrupt generati dalle diverse code di acquisizione dei pacchetti non è mai stato riscontrato alcun blocco del sistema. Assegnando invece l'interrupt generato da ogni coda di acquisizione ad un diverso core si è osservato che, sistematicamente dopo alcuni secondi di utilizzo, il server si bloccava con il relativo errore di kernel panic.

Analizzando il trace generato in concomitanza al kernel panic è possibile effettuare alcune ipotesi sulle probabili cause del problema. Una delle più attendibili associa l'errore ad una non corretta gestione dei pacchetti frammentati da parte del codice del kernel di Linux che, in ambienti multi-core,

si occupa di ricostruire il pacchetto originario partendo dai frammenti acquisiti dalla scheda di rete. Probabilmente infatti il codice del deframmentatore dei pacchetti IP implementato all'interno del kernel è stato realizzato quando non era pensabile di poter utilizzare schede di rete multi-coda per acquisire pacchetti IP da una rete ad elevato throughput su sistemi multi-core. Andrebbe quindi rivista questa parte di codice del kernel soffermandosi soprattutto sulle tecniche utilizzate per sincronizzare le operazioni di deframmentazione dei pacchetti fra i diversi core in ambienti in cui vengono utilizzate tecnologie particolari (come l'I/OAT di Intel) sviluppate per ottimizzare il funzionamento di dispositivi avanzati come le schede di rete multi-coda introdotte solo di recente.

5 Conclusioni

L'utilizzo di codice Open Source in sistemi tecnologicamente avanzati, come può essere un sistema di acquisizione dei pacchetti IP in reti ad elevato throughput, porta sicuramente a notevoli benefici dovuti principalmente alla filosofia della “condivisione” che sta alla base di questo tipo di software. Permettendo a più persone di poter accedere ai sorgenti del software è infatti possibile ottenere un incremento considerevole della velocità con cui il software stesso si evolve verso una condizione di stabilità, di sicurezza e di completezza. Il più delle volte però, per raggiungere questa condizione, è necessario attraversare fasi intermedie in cui utilizzare software Open Source può risultare quantomeno problematico. Si pensi ad esempio all'incompatibilità che può nascere fra alcuni programmi appena aggiornati e altri software presenti nel sistema. Oppure si pensi al continuo stravolgimento che può subire un software durante la sua primissima fase evolutiva: la nuova versione può stravolgere completamente o in parte la versione utilizzata fino a quel momento. In particolare si sta osservando un comportamento analogo a quello descritto in questo ultimo esempio per il socket PF_RING e i driver TNAPI. Deri, infatti, negli ultimi mesi ha introdotto una nuova versione del socket chiamata PF_RING_DNA e una nuova versione dei driver TNAPI chiamata TNAPIv2; versioni che differiscono in modo considerevole da quelle utilizzate per realizzare il sistema di acquisizione descritto e che, per questo motivo, non sono state trattate in questo articolo.

L'individuazione di un problema a livello del kernel Linux, in grado di bloccare completamente un sistema, è sicuramente una scoperta molto importante ma, per valutarne la gravità, deve essere effettuato uno studio per determinare le cause del problema e le condizioni che si devono verificare per generare il kernel panic. Per quanto riguarda le cause del problema, come si è visto in precedenza, è ragionevole pensare che si possano attribuire quasi certamente al codice del deframmentatore di pacchetti IP implementato all'interno del kernel Linux. Le condizioni che generano il kernel panic invece sono assolutamente certe infatti:

- è necessario aver installato nel sistema una scheda di rete multi-coda di ultima generazione
- è necessario che la IRQ Affinity sia configurata in modo tale da

assegnare un diverso core ad ognuno degli interrupt generati dalle diverse code di acquisizione della scheda di rete

A questo punto per generare il kernel panic è sufficiente inviare un flood di pacchetti frammentati al server di acquisizione utilizzando ad esempio il software hping3.

Da quest'ultima analisi risulta essere chiaro che la probabilità che l'errore individuato a livello del kernel di Linux possa causare un kernel panic su un sistema di uso comune è considerevolmente bassa. Ciò non toglie però che il deframmentatore presente all'interno del kernel di Linux andrebbe controllato o riscritto in modo da poter essere utilizzato correttamente anche in sistemi di acquisizione ad elevato throughput impiegati in ambienti reali dove, peraltro, il verificarsi di un flood di pacchetti IP frammentati analogo a quello generato durante i test è quantomeno improbabile.

Bibliografia

[1] “Direct cache access for high bandwidth network I/O”; Huggahalli R., Iyer R., Tetrick S.; Computer Architecture; 2005.

[2] “TCP onloading for data center servers”; Regnier G., Makineni S., Illikkal I., Iyer R., Minturn D., Huggahalli R., Newell D., Cline L., Foong A.; Computer; November 2004.

[3] “Design Considerations for Efficient Network Applications With Intel® Multi-core Processor-based Systems on Linux*”; Gasparakis J., Waskiewicz P.; Intel White Paper; July 2010.

[4] “Receive Side Coalescing for Accelerating TCP/IP Processing” Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, Jaideep Moses; Lecture Notes in Computer Science; 2006.

[5] “Improving Network Performance in Multi-Core Systems”; Intel White Paper; 2007.

[6] “Intel 82576 Gigabit Ethernet Controller Datasheet (rev 2.61)”; Intel; 2010.

[7] “MSI-X – the right way to spread interrupt load”; A. Sandler; http://www.alexonlinux.com/msi-x-the-right-way-to-spread-interrupt-load; 2009.

[8] “Benefits of I/O Acceleration Technology (I/OAT) in Clusters”; Vaidyanathan K., Panda D.K.; Performance Analysis of Systems & Software; 2007.

[9] “Accelerating High-Speed Networkin with Intel I/O Acceleration Technology”; Intel White Paper; 2006

[10] “Intel I/O Acceleration Technology”; http://www.intel.com/go/ioat/; Intel.

[11] “Improving Passive Packet Capture: Beyond Device Polling”; Luca Deri; 2004.

[12] “Exploiting Commodity Multi-core Systems for Network Traffic Analysis”; Luca Deri, Francesco Fusco; 2010.

[13] “PF_RING”; Luca Deri; http://www.ntop.org/PF_RING.html; 2011.

[14] “Why TNAPI (Threaded NAPI)?”; http://www.ntop.org/TNAPI.html; Luca Deri; 2011.

[15] “SMP affinity and proper interrupt handling in Linux”; A. Sandler; http://www.alexonlinux.com/smp-affinity-and-proper-interrupt-handling-in-linux; 2008.

[16] “Why interrupt affinity with multiple cores is not such a good thing”; A. Sandler; http://www.alexonlinux.com/why-interrupt-affinity-with-multiple-cores-is-not-such-a-good-thing; 2008.

[17] “Intel Ethernet X520 Server Adapters”; Intel; http://www.intel.com/Assets/PDF/prodbrief/322217.pdf; 2008.

[18] “PF_RING User Guide – Linux High Speed Packet Capture”; Luca Deri; https://svn.ntop.org/svn/ntop/trunk/PF_RING/doc/UsersGuide.pdf; 2011