Università di Pisa

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Ottimizzazione del network stack di FreeBSD per reti ad alta velocità

Relatori
Prof. Luigi Rizzo
Prof. Giuseppe Lettieri

Candidato
Stefano Garzarella

Anno Accademico 2012–2013

Sommario

L'utilizzo di grandi pacchetti rende la comunicazioni di rete molto meno impegnativa per la CPU. Nonostante questo, la retrocompatibilità e la presenza di link lenti, richiede l'utilizzo di pacchetti grandi al più 1500 bytes.

Le moderne schede di rete risolvono questo problema offrendo un meccanismo hardware chiamato TCP Segmentation Offload (TSO) che consente al network stack di generare pacchetti di grandi dimensioni che verranno successivamente segmentati dalla scheda in funzione dell'MTU (Maximum Transmission Unit). Tuttavia, una versione software generica (GSO - Generic Segmentation Offload) fornita dal sistema operativo ha ragione di esistere per quelle situazioni in cui l'hardware non offre supporto, come ad esempio nella comunicazione tra macchine virtuali oppure se il TSO hardware presenta dei malfunzionamenti o semplicemente non è implementato nella scheda di rete.

In questa tesi presentiamo il nostro lavoro per aggiungere il supporto al GSO in FreeBSD. L'implementazione che abbiamo realizzato permette, in funzione della frequenza di clock della CPU, di raggiungere uno speedup fino al 90% rispetto alla segmentazione effettuata nel network stack senza supporto hardware (ad eccezione del calcolo della checksum). Inoltre con frequenze da 2 GHz in su riusciamo a saturare completamente un link 10 Gbit.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il carico della CPU per il trasferimento dei dati nella rete può essere notevolmente ridotto utilizzando grandi pacchetti, dato che la maggior parte del costo computazionale della CPU è dato dal numero dei pacchetti scambiati. Per questo motivo, nelle moderne schede di rete, sono stati inseriti dei meccanismi hardware di "offloading" (TSO, LRO[1]) per incrementare le dimensioni dei pacchetti che attraversano il network stack. Non sempre queste funzionalità sono disponibili, per esempio su schede datate o tra macchine virtuali, e a volte le implementazioni hardware presentano dei bug che ne pregiudicano l'utilità. Di conseguenza alcuni sistemi operativi forniscono uno strato software, da utilizzare in queste situazioni, che effettua la segmentazione (GSO). L'assenza di un meccanismo di segmentazione (sia software che hardware) impone al network stack di generare un maggior numero di pacchetti che devono attraversare tutti i livelli, effettuando operazioni ripetitive e superflue.

Gestire ampi segmenti di dati ha requisiti e problemi differenti sul lato ricevitore, dove sarà necessario effettuare un'aggregazione, e sul lato trasmettitore dove è necessario effettua una segmentazione; FreeBSD offre un supporto software solo per il lato ricevitore, implementando via software l'LRO, ma non per il trasmettitore. Per questo motivo abbiamo deciso di implementare il GSO su FreeBSD per valutare la possibilità di incrementare le performance in assenza di un supporto hardware.

Prima di entrare nel dettaglio delle ottimizzazioni realizzate, descriveremo in modo rapido, fornendo anche alcuni cenni storici, come è strutturato il kernel di FreeBSD e quali sono i meccanismi principali offerti per Inter-process Communication (IPC). Questi argomenti verranno trattati nel Capitolo 2, al termine del quale soffermeremo l'attenzione sull'astrazione fornita dai socket e sulla gestione della memoria nel network stack.

Nel Capitolo 3 entreremo nel dettaglio del sottosistema di networking. In particolare osserveremo tutti i meccanismi che entrano in azione durante la trasmissione dei dati, dato che il lavoro finale ha portato ad ottimizzare questo aspetto della comunicazione. In questo capitolo descriveremo i livelli che compongono il sottosistema di

[1]Large Receive Offload - meccanismo duale del TSO; viene applicato in ricezione per aggregare pacchetti dello stesso flusso nel minor numero di pacchetti prima di passarli al network stack.

networking, partendo dal Socket Layer, che fornisce i meccanismi per un'astrazione completa della rete sottostante, successivamente analizzeremo il Network Stack, che implementa i protocolli necessari per la comunicazione in rete, ed infine analizzeremo le interfacce di rete.

Nel Capitolo 4 discuteremo alcuni meccanismi messi a disposizione dalle moderne schede di rete per ridurre il carico di lavoro della CPU ed aumentare il throughput della comunicazione.

Infine nel Capitolo 5 descriveremo nel dettaglio l'implementazione delle ottimizzazioni da noi proposte in questa tesi e i relativi risultati ottenuti negli esperimenti.

Durante la realizzazione di questa tesi sono stati consultati diversi libri e articoli presenti in letteratura per capire ed analizzare la struttura del sistema operativo FreeBSD[1, 2], in particolare del network stack[6], della gestione della memoria al suo interno[4, 8] e della struttura dei device driver[3]. Inoltre abbiamo consultato opere che descrivono nel dettaglio la tecnologia Internet ed i protocolli utilizzati [5, 9] ed i meccanismi di offload forniti dalle schede di rete [7, 10]. In aggiunta la community di FreeBSD rende disponibili ottimi documenti per la configurazione e lo sviluppo del sistema operativo [12, 13, 14]. Infine il codice FreeBSD è stato analizzando sfruttando un utile servizio web di cross-reference [11].

Capitolo 2

FreeBSD

In questo capitolo introduciamo il sistema operativo FreeBSD facendo dei brevi cenni storici, descrivendo rapidamente la struttura del kernel ed in particolare i meccanismi di Inter-process Communication (IPC). Guarderemo in dettaglio il livello di astrazione fornito dai socket e la gestione della memoria nel network stack.

2.1 Cenni storici

FreeBSD è un sistema operativo Unix-like che discende direttamente da AT&T UNIX attraverso BSD (Berkeley Software Distribution).

La prima versione originale di UNIX venne sviluppata nei laboratori Bell della AT&T agli inizi degli anni '70. Successivamente nel 1977, nell'università di Berkeley in California, venne rilasciata la prima versione del sistema operativo BSD (1BSD). Questa versione fu sviluppata applicando una serie di patch alla sesta release dello UNIX di AT&T, per questo motivo, storicamente, BSD viene considerato come un branch di UNIX in quanto condivide con quest'ultimo il codice base.

Negli anni successivi vennero sviluppate altre versione modificando gran parte del codice iniziale di UNIX, fino ad arrivare alla 4.2BSD, nel 1983, che includeva un'implementazione preliminare del network stack TCP/IP. La versione 4.3BSD, rilasciata nel 1986, oltre a presentare miglioramenti in termini di performance generale, includeva la versione definitiva del TCP/IP che, grazie alla superiorità dimostrata verso la concorrenza, venne scelta come implementazione standard dalla DARPA[1].

Agli inizi degli anni '90 iniziarono a presentarsi i primi problemi legali con l'AT&T che rivendicava la paternità del marchio UNIX ed il copyright su alcune porzioni di codice presenti in BSD. Per questo motivo nacquero diversi progetti, come 386BSD (1992), con lo scopo di epurare il codice BSD da quelle parti coperte da copyright AT&T.

Proprio dal progetto 386BSD nacquero i due sistemi operativi, basati su BSD, attualmente più diffusi: FreeBSD e NetBSD. La prima versione di FreeBSD (1.0) venne rilasciata verso le fine del 1993.

[1] Defense Advanced Research Projects Agency - agenzia del Dipartimento della Difesa degli USA

Nel frattempo l'università di Berkeley, nel 1994, dopo la disputa legale, rilasciò la versione 4.4BSD-Lite totalmente sotto licenza BSD e completamente libera dai vincoli di copyright con l'AT&T. Infine nel 1995 venne rilasciata l'ultima versione del sistema operativo BSD la 4.4BSD-Lite Release 2.

Dopo la release 4.4BSD-Lite, l'intero sistema FreeBSD venne re-ingegnerizzato su di essa e così, nel 1995, venne rilasciato FreeBSD 2.0.

Da allora il sistema operativo FreeBSD è stato portato avanti da un'ampia comunità fino ad arrivare all'ultima versione stabile sulla quale ci siamo basati per questo lavoro di tesi: FreeBSD 9.2.

Nel frattempo è stata rilasciata una nuova versione: FreeBSD 10.0.

2.2 Kernel

Il kernel è la parte centrale di un sistema operativo. Le istruzioni del kernel vengono eseguite in modalità protetta e permettono ai programmi utente di accedere all'hardware sottostante ed ai servizi software fondamentali (filesystem, network stack, etc.).

Il kernel implementa i meccanismi base del sistema operativo e ne fornisce l'accesso ai processi utente attraverso delle funzioni che appaiono come routine di libreria: le system call. Queste vengono implementate attraverso degli interrupt sincroni (trap) che cambiano la modalità di esecuzione della CPU e lo spazio di indirizzamento per eseguire il codice del kernel in modalità protetta. Le applicazioni utente e il kernel operano indipendemente l'uno dall'altro. Ogni processo utente ha il suo spazio di indirizzamento indipendente dove eseguire e differente sia da altri processi che dal kernel.

FreeBSD dispone di un kernel monolitico, ciò significa che tutte le funzionalità principali, compreso gestione del filesystem e network stack, sono realizzate nel kernel ed eseguite in modalità protetta. Questa tipologia si contrappone ai micro-kernel dove alcune funzionalità, come filesystem e network stack, vengono implementate come processi server e vengono eseguiti in modalità utente, fornendo una stabilità maggiore a discapito delle performance, in quanto questa divisione comporta un overhead maggiore. La scelta di utilizzare un kernel monolitico, fatta durante lo sviluppo delle prime versioni di UNIX, fu dettata dalla semplicità di realizzazione e dalle performance che un kernel di questo genere può garantire.

Il kernel di FreeBSD offre quattro meccanismi fondamentali:

- gestione dei processi
- filesystem e gestione dell'I/O
- comunicazione tra processi
- avvio del sistema.

Soffermandoci sull'aspetto della comunicazione, il kernel di FreeBSD, offre le funzionalità tradizionali dei sistemi UNIX come le pipe (byte stream monodirezionale

fra processi) ed i segnali (notifiche di eventi). Inoltre FreeBSD mette a disposizione un vasto sistema di inter-process-communication (IPC) che può essere suddiviso in meccanismi per la comunicazione tra processi esclusivamente locale (su un singolo sistema operativo), come semafori, code di messaggi e memoria condivisa, e meccanismi come i socket che forniscono un'API[2] uniforme per la comunicazione in rete ma anche in locale. Quest'ultimo meccanismo è realizzato sfruttando le funzionalità offerte dal sottosistema di networking che in seguito analizzeremo in dettaglio in quanto rappresenta il punto centrale di questo lavoro di tesi.

2.3 Inter-process Communication

Quando venne deciso di ampliare l'IPC di UNIX, gli sviluppatori BSD si posero diversi obiettivi tra i quali quello di creare un'interfaccia generica per permettere di sviluppare applicazioni network-based indipendemente dal livello sottostante. Un altro obiettivo era quello di favorire la realizzazione di programmi distribuiti, in quanto la *pipe* UNIX richiedeva che i processi comunicati dovessero discendere dallo stesso processo padre, mentre si voleva ottenere la possibilità di mettere in comunicazione processi completamente scorrelati, i quali potevano essere in esecuzione localmente, sulla stessa macchina, o su macchine distinte, collegate mediante una rete.

Furono definiti i concetti centrali alla base di questo nuovo sistema di comunicazione:

- dominio di comunicazione

  definisce i protocolli per la comunicazione e lo standard per il naming degli endpoints. Questo è necessario poiché il sistema deve supportare diverse tipologie di reti ed ognuna definisce protocolli e convenzioni di naming differenti.

- socket

  rappresenta un oggetto astratto attraverso il quale si possono inviare e ricevere messaggi.

- semantica della comunicazione

  definisce quali proprietà deve avere la comunicazione:

  1. consegna ordinata dei dati
  2. consegna senza duplicati dei dati
  3. consegna affidabile dei dati
  4. supporto ad instaurare una connessione (connection-oriented)
  5. lasciare inalterati i confini e la struttura di un messaggio (datagram)
  6. supporto a messaggi "out-of band" (dati urgenti o eccezioni)

[2] Application Programming Interface

2.3.1 Socket

I socket rappresentano gli end-points di una comunicazione. Ogni socket viene creato attraverso la system call:

```
int
socket (int domain, int type, int protocol)
```

domain	dominio di comunicazione
type	tipo del socket
protocol	protocollo specifico da utilizzare (opzionale)

La system call restituisce un file descriptor (fd) che identificherà il socket per tutta la sua vita. Il tipo definisce la semantica, cioè quali proprietà deve avere la comunicazione. Di conseguenza, ad ogni socket viene associato un protocollo di comunicazione che garantisce che lo scambio di informazioni rispetti la semantica associata al tipo del socket e al dominio di comunicazione. L'applicazione utente può richiedere uno specifico protocollo di comunicazione (TCP, UDP, etc.), altrimenti sarà il sistema operativo a scegliere il protocollo in funzione della tipologia di socket specificata. Inoltre ad ogni socket può essere associato un indirizzo, il cui formato e significato dipende dal dominio di comunicazione in uso.

I dati trasmessi attraverso i socket sono dati grezzi, senza tipo, sarà compito delle applicazioni o librerie che sfruttano questi meccanismi IPC affrontare il problema della rappresentazione dei dati trasmessi.

FreeBSD supporta i seguenti tipi di socket:

- datagram - consegna dei dati non affidabile, non ordinata, rispetta i confini e la struttura del messaggio.
- stream - consegna affidabile e ordinata dei dati che vengono interpretati come uno stream di byte.
- sequenced packet - consegna dei dati affidabile, ordinata, rispetta i confini e la struttura del messaggio.
- raw - accesso diretto ai protocolli di comunicazione.

Dopo aver creato un socket, devono essere fatte diverse operazioni per poter comunicare:

- se si è scelto di utilizzare un protocollo connection-oriented, il server ha bisogno di porsi in ascolto di richieste di connessione, per fare questo, bisogna assegnare un'indirizzo al socket attraverso il quale i client possono collegarsi, utilizzando la system call *bind()*, e porre il socket in ascolto attraverso la system call *listen()*. A questo punto può essere invocata la system call *accept()* che restituisce un nuovo socket quando viene instaurata la connessione con il client; su questo socket verranno effettuate le operazioni di I/O. Dal lato client, dopo aver creato il

socket, è necessario instaurare la connessione attraverso la system call *connect()* specificando l'indirizzo del server.

- se si è scelto un protocollo connection-less è necessario effettuare solamente la *bind()* su entrambi gli host per assegnare un indirizzo ai socket.

Dopo queste operazioni preliminare può aver luogo la comunicazione utilizzando diverse system call a seconda del tipo di protocollo utilizzato:

- per i protocolli connection-oriented possono essere utilizzate sia le system call "generiche" *read()* e *write(),* in grado di operare su file descriptor che identificano bytes stream come files o socket, specificando, oltre al file descriptor, il buffer e la lunghezza che contengono i dati da inviare o in grado di contenere i dati da ricevere; sia system call specifiche per i socket come *recv*() e *send*() che consentono anche di impostare dei flags per i messaggi inviati o ricevuti (per esempio per inviare messaggi out-of-band o per fare in modo che la recv attenda tutti i bytes specificati).

- per i protocolli connection-less sono disponibili le system call *sendmsg()* e *sendto()* che richiedono di specificare per ogni messaggio l'indirizzo dell'host a cui sono indirizzati i dati e *recvmsg()* e *recvfrom()* che restituiscono l'indirizzo dell'host che ha inviato i dati. Possono essere utilizzare anche *recv()* e *send()* a condizione che venga chiamata la *connect()* prima di queste operazioni; in questo caso la *connect()* non instaura nessuna connessione, ma il kernel associa al socket l'indirizzo dell'host che verrà utilizzato nella *send()* per inviare i messaggi correttamente e nella *recv()* per accettare solo i messaggi provenienti dall'host specificato.

In aggiunta alle system calls descritte precedentemente, sono presenti altre che permetto di recuperare informazioni sull'indirizzo associato al socket locale *getsockname()* o al socket remoto *getpeername()* e settare/recuperare parametri che controllano le operazioni del socket o dei sottostanti protocolli di rete *setsockopt()*/*getsockopt()*. Quando la comunicazione è terminata, attraverso la system call *close()*, il socket e le strutture allocate per la sua gestione vengono eliminati.

2.4 Gestione della memoria nel network stack (mbuf)

Nel sottosistema di networking la gestione della memoria è sostanzialmente differente dalle altre parti del sistema operativo.

Qualsiasi parte del kernel richiede che allocazione e accesso alla memoria siano effettuati in modo efficiente, ma i protocolli di rete necessitano spesso di buffer in grado di variare la dimensione efficientemente in quanto i pacchetti vengono creati in modo incrementale. Inoltre il principio di incapsulamento, che contraddistingue i protocolli di rete, richiedono molto frequentemente di anteporre o rimuovere intestazioni ai dati impacchettati. Oppure capita spesso che, quando i pacchetti vengono inviati, i dati

nel buffer da trasmettere devono essere suddivisi in più pacchetti, mentre quando i pacchetti vengono ricevuti, devono essere aggregati in un unico buffer.

Tutte queste particolari necessità hanno dato vita ad una gestione della memoria differente dal resto del sistema incentrata sulla struttura dati chiamata *mbuf* (memory buffer); questi buffer sono utilizzati per ottimizzare l'attraversamento dei dati nel network stack, fino ad arrivare ai device driver dove i pacchetti vengono trasferiti nelle code delle schede (NIC[3]) e inviati sulla rete.

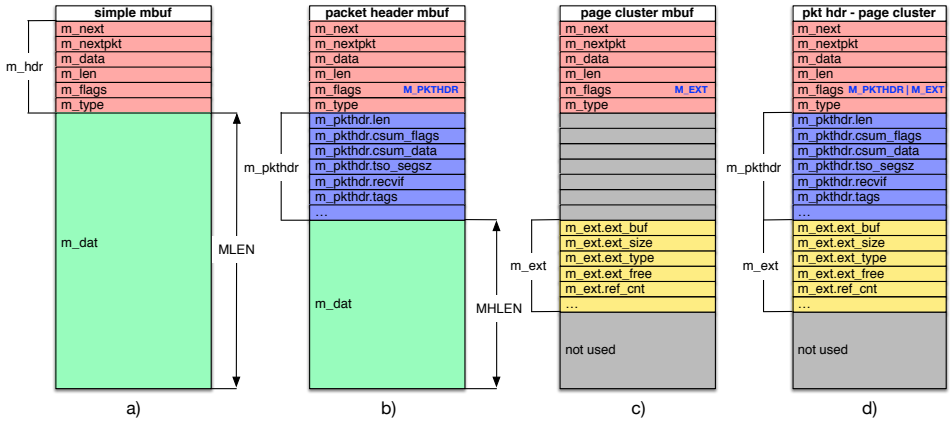

Figura 2.4.1: mbufs

Gli mbufs sono in grado di variare la dimensione in funzione di ciò che devono contenere. Tutti gli mbuf contengono un'intestazione di dimensione fissa, m_hdr [Figura 2.4.1 a], al cui interno vengono memorizzate tutte le informazioni che descrivono l'mbuf in questione:

- m_next: puntatore utilizzato per collegare più mbuf tra loro per formare una catena di mbuf (mbuf chain) in grado di contenere un pacchetto di dimensioni arbitrarie.
- m_nextpkt: puntatore utilizzato per collegare tra loro più catene di mbuf (pacchetti) per formare una coda (mbuf queue).
- m_data: puntatore alla prima locazione di memoria occupata dai dati nel buffer interno o esterno.
- m_len: lunghezza dei dati contenuti nell'mbuf.
- m_flags: flags che descrivono i singoli mbuf e i dati contenuti nel pacchetto.
- m_type: identifica il tipo di mbuf (dati o controllo).

I dati possono essere memorizzati all'interno dell'mbuf stesso oppure esternamente, in questo caso l'mbuf conterrà un riferimento ad un *mbuf cluster* esterno, una zona di memoria virtuale privata, la cui dimensione dipende dall'architettura in uso, ma generalmente è di 2 KB. Solo una delle due modalità può essere utilizzata. In entrambi

[3]Network Interface Controller

i casi, all'interno dell'mbuf, vengono memorizzati un puntatore ai dati (m_data) e la lunghezza della zona di memoria occupata (m_len). Questo permette alle routine di aggiungere o eliminare dati in modo efficiente, evitando di effettuare copie. Per esempio se si desidera eliminare dei dati dalla fine dell'mbuf, è sufficiente decrementare la lunghezza, mentre se si desidera eliminare dati dalla testa, bisogna incrementare il puntatore e decrementare la lunghezza. Le operazioni di cancellazione o inserimento in testa o in coda degli mbuf sono molto frequenti nel network stack e questa organizzazione della memoria le rende molto efficienti.

Quando i dati di un mbuf sono memorizzati in un *mbuf cluster* esterno, è presente un'ulteriore header, m_ext [Figura 2.4.1 c], che descrive quest'area di memoria; la presenza del buffer esterno e di conseguenza dell'header sono segnalate con il flag M_EXT memorizzato nel campo *m_flags* nell'header dell'mbuf; in particolare le informazioni mantenute in m_ext sono: un puntatore al buffer esterno (ext_buf), le dimensioni dell'intera area di memoria disponibile (ext_size), un puntatore alla funzione da utilizzare per rilasciare il buffer (ext_free) e un puntatore al reference counter (ref_cnt). Quest'ultimo è necessario in quanto uno stesso buffer esterno può essere condiviso fra più mbuf, permettendo di condividere gli stessi dati tra più pacchetti senza effettuare copie, ma esclusivamente i puntatori alle pagine esterne che li contengono.

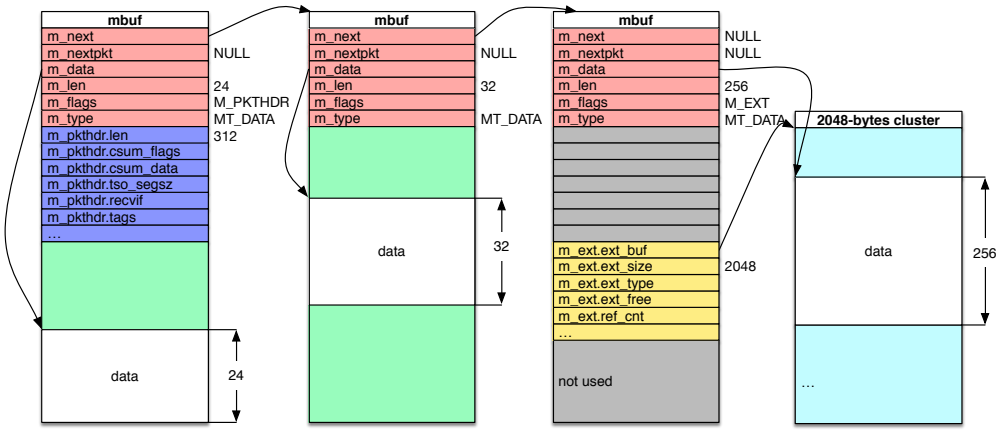

Figura 2.4.2: mbuf chain

Per creare buffer in grado di contenere una quantità arbitraria di dati, gli mbuf possono essere concatenati fra loro per formare una catena (mbuf chain - [Figura 2.4.2]). In questo caso, il primo della catena contiene un header, m_pkthdr [Figura 2.4.1 b], che descrive l'intera catena o pacchetto e il flag M_PKTHDR settato in *m_flags*; all'interno di m_pkthdr sono memorizzati: l'intera lunghezza del pacchetto (len), data dalla somma delle lunghezze dei singoli mbuf collegati fra loro per formare la catena; due campi (csum_flags, csum_data) relativi al calcolo della checksum e a relativi meccanismi di offloading, come il TSO (Tcp Segmentation Offloading), richiesti sul pacchetto; un campo (tso_segsz) che specifica le dimensioni che devono avere i segmenti generati dal TSO; un puntatore (tags) ad una lista arbitraria di tags

associati al pacchetto e un puntatore (recvif) alla struttura che descrive l'interfaccia sulla quale il pacchetto è stato ricevuto.

I flags contenuti in *m_flags* che abbiamo menzionato fino ad ora descrivono la tipologia dei singoli mbuf, ma sono presenti altri flags che descrivono il pacchetto contenuto nella catena di mbuf come M_BCAST e M_MCAST, i quali indicano che il pacchetto è da inviare o è stato ricevuto in modalità broadcast (M_BCAST) o multicast (M_MCAST).

La dimensione di un singolo mbuf è definita dalla macro MSIZE (256 bytes). A seconda se l'mbuf contiene o meno l'm_pkthdr, l'area di memoria interna disponibile varia, ed è definita dalle macro MLEN , se si tratta di un mbuf semplice [Figura 2.4.1 a], o MHLEN, se l'mbuf contiene l'm_pkthdr [Figura 2.4.1 b]. Queste dimensioni dipendono dall'architettura in uso.

2.4.1 Funzioni di utilità per gestire mbuf

Esistono molte routines, funzioni e macro, per allocare, manipolare e rilasciare mbuf. Di seguito descriviamo le principali funzioni utilizzate:

MGET() alloca un mbuf semplice.

MGETHDR() alloca un mbuf e lo inizializza con l'intestazione m_pkthdr.

MCLGET() aggiunge un cluster esterno ad un mbuf.

m_free() rilascia un singolo mbuf.

m_freem() rilascia una catena di mbuf.

m_append() inserisce nuovi dati in coda alla catena di mbuf; se necessario estende la catena allocando nuovi mbuf.

M_PREPEND() in funzione delle dimensioni specificate, alloca un nuovo mbuf e lo pone in testa alla catena. Questa macro ottimizza la funzione m_prepend(), utilizzando, se possibile, spazio disponibile in testa al primo mbuf della catena.

m_copym() restituisce una copia dell'mbuf chain passata come parametro. Si possono specificare l'offset e la lunghezza dei dati da copiare. La copia restituita è read-only perché gli mbuf clusters non vengono copiati, ma viene semplicemente incrementato il loro reference count e copiato il puntatore alla pagine esterna.

m_copydata() copia dati da un mbuf chain in un buffer tradizionale (area di memoria contigua indirizzata da un puntatore C). Si possono specificare l'offset e la lunghezza dei dati da copiare.

m_copyback() copia dati da un buffer tradizionale in un mbuf chain; se necessario estende la catena allocando nuovi mbuf. Si può specificare l'offset all'interno del mbuf chain dove copiare i dati.

m_cat() concatena due mbuf chains.

m_adj() rimuove dati dalla testa o dalla coda dell'mbuf chain. Questa operazione viene semplicemente eseguita modificando la lunghezza e i puntatori degli mbuf.

m_pullup() modifica l'mbuf chain per fare in modo che i primi len bytes, passati come parametro, siano memorizzati in un'area di memoria contigua e quindi accessibili con mtod(). Non alloca nessun mbuf clusters, quindi len deve essere minore di MHLEN.

mtod() converte un puntatore ad un mbuf in un puntatore C tradizionale, il tipo deve essere specificato nei parametri.

Capitolo 3

Sottosistema di networking

Dopo aver introdotto l'astrazione fornita dai socket, scendiamo nel dettaglio analizzando il *network subsystem* e tutti i livelli che lo compongono [Figura 3.0.1].

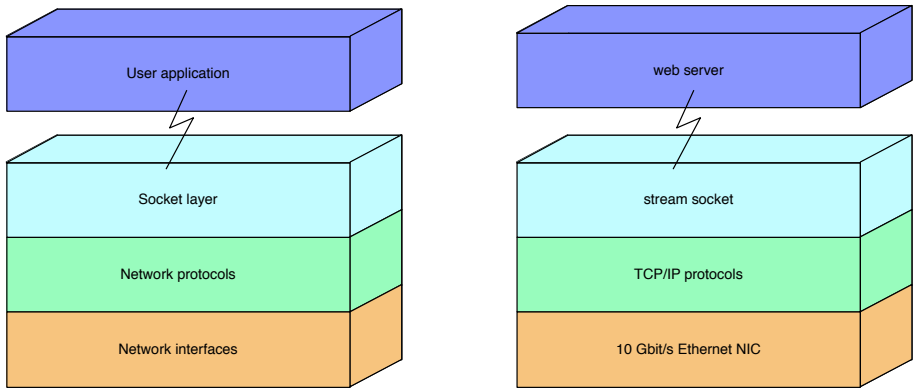

Figura 3.0.1: Sottosistema di networking

3.1 Socket Layer

Il livello socket è il più alto e offre tutti i meccanismi IPC ai programmi utente attraverso le system call descritte nella [Sezione 2.3.1].

Ad ogni system call invocata dai programmi utente sono associate solitamente due livelli di routine. La routine di primo livello rappresenta un'interfaccia tra lo spazio utente e quello kernel, ha il compito di raggruppare i parametri della system call e convertire i dati utente nel formato atteso dalle routine di secondo livello. Tutte le routine di secondo livello hanno il nome con prefisso *so* (*so*create(), *so*bind(), *so*send(), *so*receive(), *so*close(), ...). La maggior parte dell'astrazione fornita dai socket viene implementata in questo livello, il cui compito principale è quello di manipolare le strutture dati associate ad ogni socket e gestire la sincronizzazione tra attività asincrone.

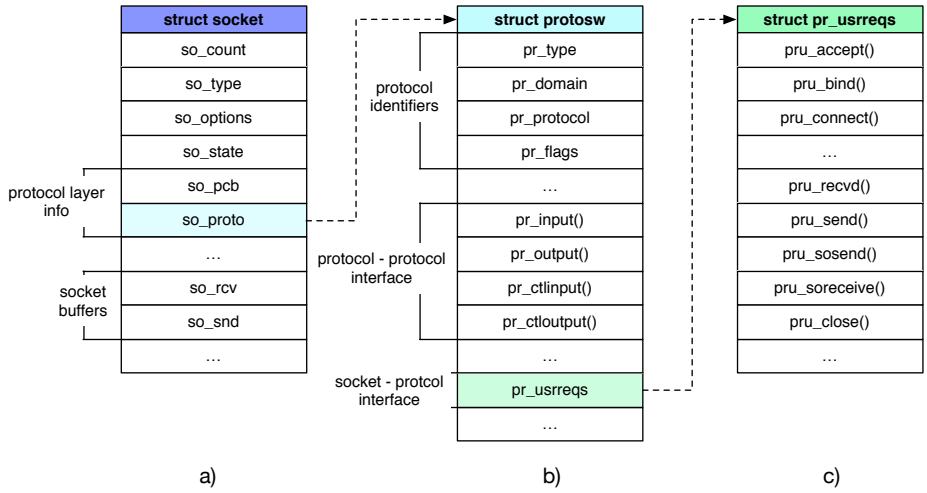

Figura 3.1.1: struct socket

Tutte le informazioni di stato del socket vengono mantenute, in questo livello, utilizzando la struttura dati *struct socket* [Figura 3.1.1 a]. Questa viene allocata dinamicamente con la syscall socket() e associata ad un file descriptor; contiene informazioni sul tipo di socket (so_type), sul protocollo di supporto utilizzato per la comunicazione (so_pcb, so_proto), sullo stato del socket (so_state) e mantiene i riferimenti ai socket buffer di ricezione (so_rcv) e trasmissione (so_snd) dove vengono accodati i pacchetti; sono inoltre presenti altri campi per gestire eventuali connessioni.

L'interfaccia tra il socket layer e il network layer viene fornita dalla *struct protosw* [Figura 3.1.1 b]. Questa struttura rappresenta la *Protocol Switch Table*; quando il socket viene creato, viene selezionato l'array contenente tutte le *Protocol Switch Table associate* al dominio scelto (allocate staticamente per ogni protocollo), a questo punto, viene scelto una *Protocol Switch Table* in funzione del tipo di socket (pr_type) e, se specificato, del protocollo (pr_protocol). L'elemento selezionato sarà puntato dal campo *so_proto* della struttura *socket*.

All'interno della *protocol switch table* troviamo, oltre ai campi che identificano il protocollo, un campo (pr_flags) che descrive alcune peculiarità del protocollo (connessione richiesta, indirizzo specificato in ogni messaggio, messaggio atomico, etc.), dei campi che offrono un'interfaccia sia per i dati che per informazioni di controllo tra i protocolli del network stack (es. *pr_input()* è utilizzata dal protocollo di rete per passare un messaggio al protocollo di trasporto) e un puntatore (pr_usrreqs) che indirizza la tabella delle *user request routines* (*struct pr_usrreqs*) [Figura 3.1.1 c]. In quest'ultima struttura sono contenuti i puntatori alle funzioni che rappresentano l'interfaccia tra il socket layer e il network layer.

3.1.1 Trasmissione dati

Tralasciando il meccanismo di connessione, ci soffermiamo sul sistema alla base della trasmissione dei dati [Figura 3.1.2]. Tutte le richieste di invio e ricezione dati, at-

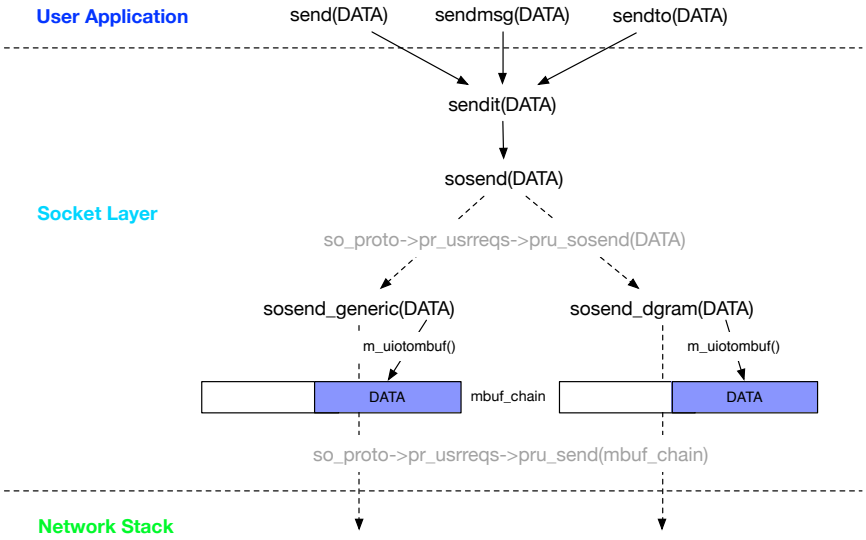

Figura 3.1.2: Data flow in Socket Layer

traverso i socket, effettuate invocando le system call menzionate nella [Sezione 2.3.1] sono convertite in un unico formato e passate alle due routine del livello socket che gestiscono il trasferimento dati: *sendit()* e *recvit()*. Dato che in questo lavoro di tesi abbiamo cercato di ottimizzare la trasmissione dei dati, analizziamo nel dettaglio il percorso che i dati seguono durante l'invio.

La funzione *sendit()* ha il compito di raccogliere tutti i parametri specificati dall'applicazione utente (indirizzo destinatario, eventuali dati di controllo, flags opzionali) e copiarli nello spazio di indirizzamento del kernel, ad eccezione dei dati da trasferire. A questo punto viene invocata la routine *sosend()* che a sua volta richiama *sosend_ generic()* o *sosend_ dgram()* a seconda del tipo di protocollo associato al socket, questo viene fatto sfruttando il puntatore a funzione *pru_ sosend()*, inizializzato alla creazione del socket. Queste due routine, che rappresentano l'ultimo step all'interno del livello socket, hanno il compito di gestire la maggior parte delle opzioni presenti in questo livello, in particole viene controllato lo stato del socket (se è possibile effettuare la trasmissione, se l'eventuale connessione è ancora attiva, se ci sono errori pendenti, etc.), vengono gestiti eventuali messaggi *out-of-band* e viene sospeso il processo chiamante se i dati da trasferire superano lo spazio disponibile nel socket buffer di invio. Infine, se la trasmissione può avere luogo, i dati vengono copiati dallo spazio utente all'interno di mbufs nello spazio kernel (*m_ uiotombuf()*) e poi invocata la funzione *pru_ send()* per inviare il pacchetto al protocollo associato al socket.

3.2 Network Protocols

Il livello socket appena descritto si appoggia direttamente sul *network stack*. In questo livello vengono implementati i protocolli per la comunicazione in rete. Dato che

la tecnologia Internet è la più diffusa, analizzeremo nel dettaglio l'organizzazione e l'implementazione dell'*Internet protocol stack*.

Questi protocolli vennero inizialmente sviluppati grazie a finanziamenti da parte della DARPA per la realizzazione di una rete di calcolatori (Arpanet).

3.2.1 Internet protocol stack

Internet venne concepito per un modello di rete dove gli host erano connessi a reti con caratteristiche diverse e le reti interconnesse dai router.

I protocolli di Internet sono stati progettati per sfruttare reti "packet-switching" che permettono la trasmissione di pacchetti attraverso link che non forniscono alcuna indicazione sull'effettiva consegna dei dati, come Ethernet. Per questo motivo tutti i protocolli della suite assumono che la rete sottostante non sia affidabile.

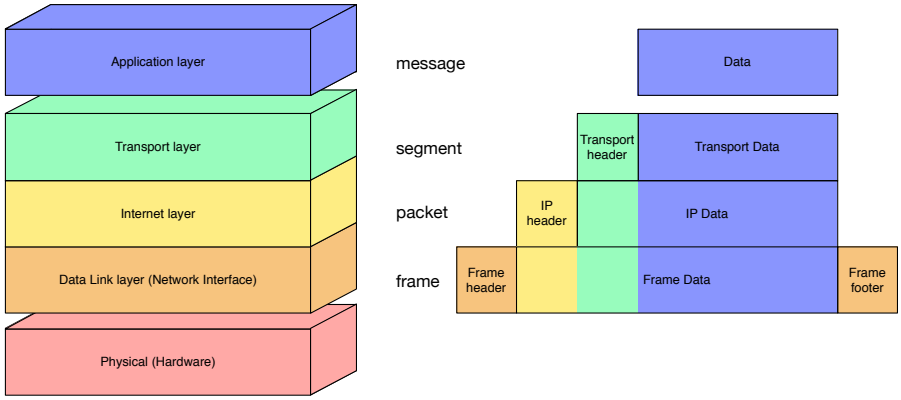

Figura 3.2.1: Internet protocol stack

Lo stack protocollare globalmente può essere organizzato in 4 livelli più il livello hardware [Figura 3.2.1]:

data-link è il livello più basso ed è strettamente correlato con l'hardware; definisce il protocollo da utilizzare per comunicare con l'host più vicino (next-hop) all'interno della stessa rete.

internet (IP - Internet Protocol) permette ad un pacchetto inviato da un host di raggiungere l'host destinatario, attraversando reti diverse interconnesse da router. Questo livello produce un'astrazione della rete sottostante, nascondendo la topologia reale; inoltre fornisce i servizi di indirizzamento degli host, di routing e, se la rete sottostante non è in grado di trasmettere pacchetti oltre una determinata dimensione, offre un meccanismo di frammentazione e riassemblaggio. Tutti gli altri protocolli sfruttano i servizi offerti da IP.

trasporto consente ad un messaggio di raggiungere, attraverso i socket, il processo applicativo destinatario. Di conseguenza ad ogni socket viene assegnato un id univoco all'interno dell'host (porta). In questo modo attraverso l'indirizzo

dell'host e la porta, si può identificare univocamente un socket nella rete. Internet fornisce due protocolli di trasporto: TCP (Transmission Control Protocol) e UDP (User Datagram Protocol). TCP fornice un servizio di comunicazione connection-oriented, affidabile, senza duplicati e presenta un meccanismo di controllo di flusso e controllo di congestione. UDP, invece, oltre ai servizi di IP, offre esclusivamente un meccanismo per verificare l'integrità del messaggio (checksum); per questo motivo il servizio fornito è non affidabile, connection-less e inoltre sono possibili messaggi duplicati e non ordinati. TCP viene utilizzato per supportare i socket di tipo *stream* nel dominio Internet, mentre UDP è a supporto dei socket di tipo *datagram*.

applicazione il livello più alto dello stack è costituito da tutti quei protocolli utilizzati per scambiare informazioni tra programmi in esecuzione sullo stesso host o su host distinti. Esistono molti protocolli del livello applicazioni come HTTP, SMTP, FTP, DNS, etc.

Lo stack di internet utilizza la tecnica di *encapsulation/decapsulation* [Figura 3.2.1]. Quando un messaggio deve essere inviato esso parte dal livello applicazione e percorre tutto lo stack, nella fase di *encapsulation*, in un ordine preciso, dato che ogni livello può comunicare solo con il livello sottostante. Ogni livello dello stack incapsula i dati del livello precedente inserendo un'apposita intestazione (header) necessaria per la fase complementare di *decapsulation*. Il livello di trasporto crea un *segmento* a partire dal *messaggio* generato dal livello applicazione, il segmento viene incapsulato in un *pacchetto* dal livello internet e il livello data-link produce un *frame,* inserendo un header e un footer, che viene poi trasmesso fisicamente dalla scheda di rete. Durante la fase di *decapsulation* il frame parte dal livello più basso e, grazie alle informazioni contenute negli headers, risale lo stack fino a giungere al livello applicazione.

3.2.2 Protocol Control Blocks

Ogni volta che viene creato un socket UDP o TCP, è necessario provvedere ad allocare una struttura dati che contenga tutte le informazioni relative al protocollo.

Questa struttura è la *Internet protocol control block* (struct inpcb) [Figura 3.2.2 a]; al suo interno sono memorizzati gli indirizzi e le porte locali e remote, informazioni di routing, informazioni sulla tipologia del flusso dei dati, eventuali opzioni da utilizzare in IP, dei campi per effettuare caching di informazioni e puntatori ad eventuali strutture dati ausiliarie. Nel caso di socket TCP, dato che è necessario instaurare una connessione tra i peer e quindi mantenere informazioni di stato, viene allocata un'ulteriore struttura dati, *TCP control block* (struct tcpcb) [Figura 3.2.2 b], al cui interno vengono mantenute tutte le informazioni necessarie per l'implementazione.

3.2.3 User Datagram Protocol

UDP è un protocollo di trasporto molto semplice, di conseguenza anche l'header del pacchetto è di dimensioni ridotte (8 bytes) [Figura 3.2.3]. Dato che UDP non

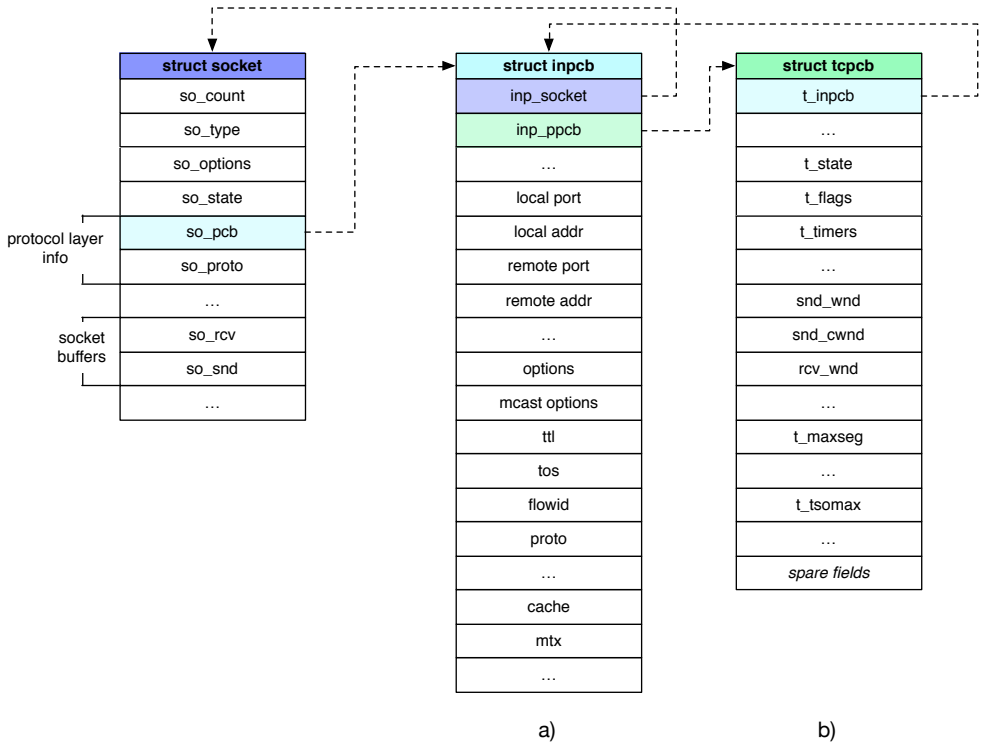

Figura 3.2.2: Protocol Control Block

offre nessun meccanismo per gestire il trasferimento affidabile dei dati, l'ordine dei messaggi, il controllo di flusso e congestione, l'overhead introdotto dal processamento di questi pacchetti è molto basso. Per questi motivi, UDP è largamente utilizzato per supportare applicazioni time-sensitive, dove la perdita di pacchetti è preferibile rispetto a pacchetti con elevato ritardo.

offsets	byte	0								1								2								3							
byte	bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Total Length (header + data)																Checksum															
8+	64+	UDP DATA																															

Figura 3.2.3: UDP header

I servizi che vengono offerti, come abbiamo già illustrato in precedenza, sono quelli basilari che un protocollo di trasporto offre:

- controllo di integrità dei pacchetti, attraverso una checksum.
- multiplexing delle comunicazioni, attraverso il meccanismo delle porte.

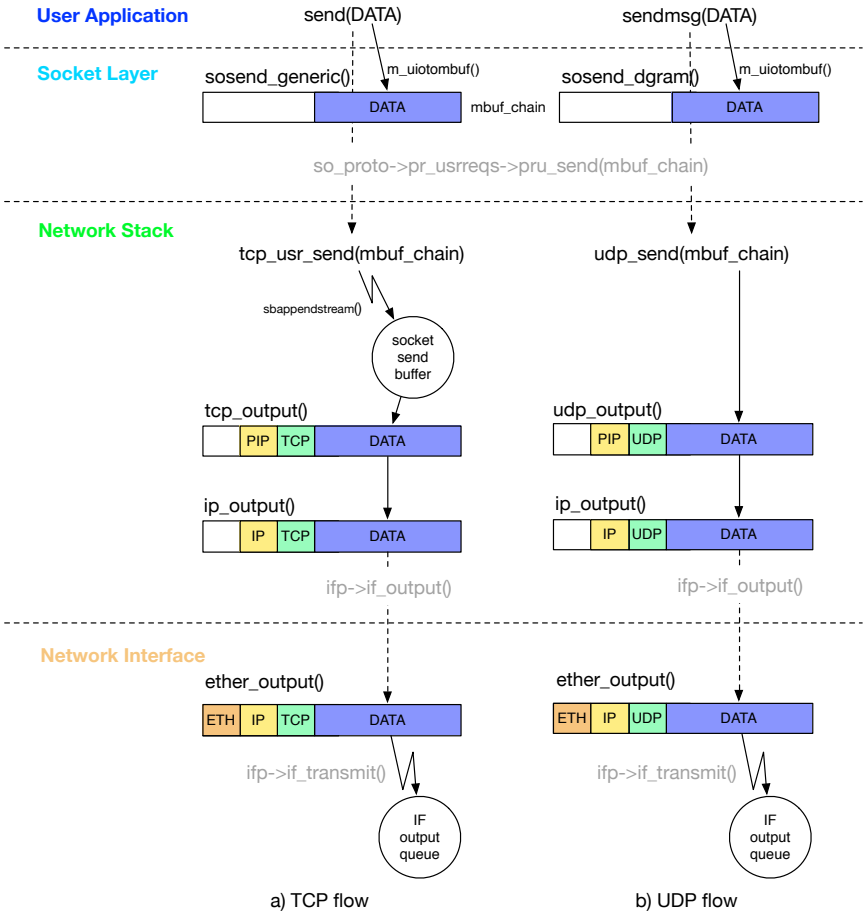

Figura 3.2.4: Data flow in Network Stack

3.2.3.1 Trasmissione dati

Nella [Sezione 3.1.1] abbiamo osservato che con la chiamata pru_send() il messaggio, già copiato in una catena di mbuf, viene passato al network stack. Nel caso di socket UDP, il puntatore pru_send() riferisce alla funzione *udp_ send()*. Questa funzione, a sua volta, invoca la routine di output del protocollo UDP [Figura 3.2.4 b]:

```
static int
udp_output(struct inpcb *inp, struct mbuf *m,
        struct sockaddr *addr, struct mbuf *control,
        struct thread *td)
```

inp	puntatore alla struttura *Internet protocol control block* associata al socket
m	catena di mbuf che contiene il messaggio da inviare
addr	mbuf opzionale che contiene l'indirizzo di destinazione
control	mbuf opzionale che contiene eventuali dati di controllo (indirizzo sorgente o opzioni)

td struttura dati che identifica il thread che ha inviato il pacchetto

Questa funzione ha il compito di recuperare l'indirizzo e la porta sorgente attraverso *inp* (Internet protocol control block) o, se specificato nei dati di controllo (*control*); successivamente vengono identificati l'indirizzo e la porta del destinatario, se il socket è stato "connesso" queste informazioni vengono recuperate attraverso *inp*, altrimenti devono essere specificate con l'argomento *addr*.

Una volta raccolte queste informazioni necessarie per la costruzione dell'header, viene riservato dello spazio per contenere gli headers UDP, IP e del livello link (es. Ethernet) in testa alla catena di mbuf attraverso la routine M_PREPEND [Sezione 2.4.1].

A questo punto vengono creati l'header UDP e lo pseudo-header [Figura 3.2.5], necessario per il calcolo della checksum del protocollo di trasporto; di seguito, dopo aver impostato eventuali flags (DONTFRAG, BROADCAST, etc.), se la checksum è richiesta, viene calcolata parzialmente solo sullo pseudo-header e memorizzata nel campo checksum dell'header UDP. Questo calcolo parziale viene eseguito in quanto il calcolo totale della checksum per il protocollo di trasporto, descritto nella [Sezione 3.2.4], può essere completato via hardware, dalle moderne schede di rete (le quali richiedono che il campo checksum del protocollo di trasporto contenga la checksum parziale sullo pseudo-header) o via software nel livello IP, in quanto l'algoritmo è lo stesso per tutti i protocolli di trasporto.

Dopo queste operazioni, il pacchetto viene inviato al livello Internet sottostante mediante la routine di output del protocollo IP: *ip_ output()* [Figura 3.2.4 b].

3.2.4 Internet Checksum

La checksum nel livello di trasporto è utilizzata per effettuare error-detection su dati, quindi per scartare eventuali pacchetti che sono stati corrotti durante la trasmissione.

offsets	byte	0								1								2								3							
byte	bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source IP address																															
4	32	Destination IP address																															
8	64	Reserved								Protocol								Transport total length (transp. header + transp. data)															

Figura 3.2.5: Pseudo-header

I dati sui quali viene effettua la checksum includono lo pseudo-header e tutto il segmento generato dal livello di trasporto (header e dati) [Figura 3.2.6]. Il campo checksum dell'header del protocollo di trasporto deve essere impostato a 0 prima del calcolo della checksum.

Questi dati vengono interpretati come una sequenza di numeri interi a 16 bit (word) rappresentati in complemento a 1. La sequenza viene sommata utilizzando l'aritmetica in complemento a 1 su 16 bit. Il risultato, dopo essere stato negato

(complemento a 1) viene inserito nel campo appropriato all'interno dell'header del protocollo di trasporto. La negazione del risultato viene effettuata in modo che, per verificare che i dati siano integri, è sufficiente ripetere la stessa operazione (includendo anche il campo che contiene la checksum). Se i dati non sono stati corrotti, il risultato ottenuto durante la verifica deve essere 0.

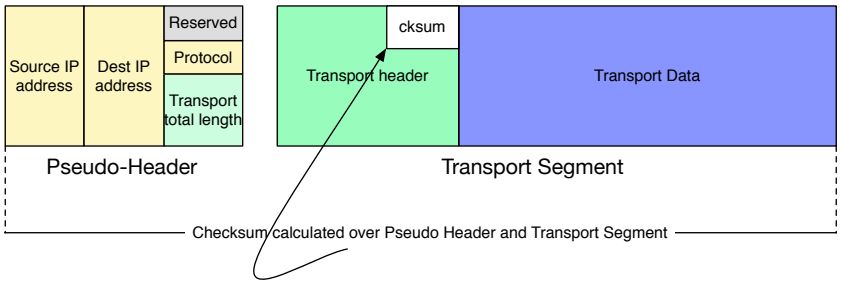

Figura 3.2.6: Transport checksum

Basandosi sull'addizione, questo algoritmo, eredita le proprietà commutativa ed associativa. Questo consente di effettuare la checksum parziale sullo pseudo header e successivamente completarla via hardware o via software.

3.2.5 Transmission Control Protocol

Il TCP è il protocollo di trasporto dell'Internet Network Stack più utilizzato; offre una comunicazione affidabile, stream-based, connection-oriented sulla quale si basano molti protocolli del livello applicazione. Per fornire tutte queste funzionalità, l'implementazione del TCP è molto più complicata rispetto all'UDP e l'header TCP risulta più esteso. Tutto questo implica un maggiore overhead introdotto nella comunicazione, costo che bisogna pagare se si necessita di una comunicazione affidabile.

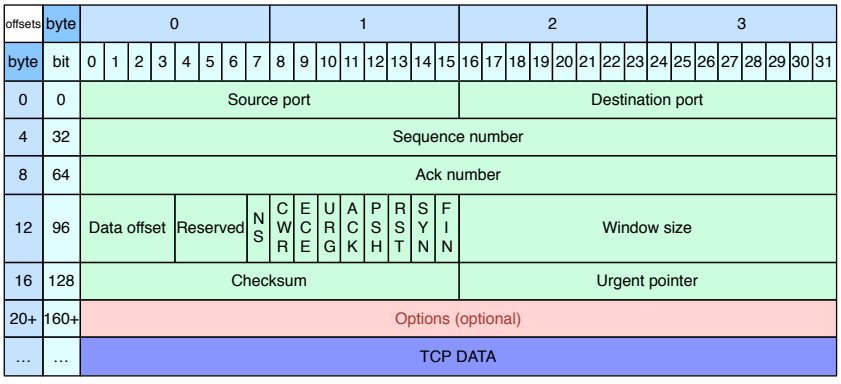

Figura 3.2.7: TCP header

Riassumendo, le principali funzionalità offerte dal TCP sono:

- instaurazione e terminazione della connessione

- consegna dei dati affidabile, ordinata e senza duplicati
- controllo di flusso
- controllo di congestione
- segnalazione di dati "urgenti" out-of-band

Dato che in questo lavoro di tesi ci siamo concentrati sull'ottimizzazione della trasmissione, tralasciamo i meccanismi di connessione (Three-way handshake) e disconnessione del TCP e diamo una breve descrizione dei campi presenti nell'header TCP [Figura 3.2.7] e dei meccanismi di controllo di flusso e controllo congestione.

I campi *Sequence number* e *Ack number* sono utilizzati per fornire un servizio di consegna dei dati affidabile, ordinato e senza duplicati. Il *Sequence number,* che rappresenta il primo byte dei dati che il segmento trasporta, serve per identificare e posizionare in maniera ordinata i dati contenuti nel segmento all'interno del flusso (stream). Il ricevitore, per riscontrare i dati ricevuti, invia un *Ack number* che rappresenta l'ultimo byte (+1) ricevuto correttamente; per questo motivo il trasmettitore mantiene nel socket send buffer i dati finché non vengono riscontrati e un timer per ritrasmettere i dati nel caso in cui non vengano ricevuti ACK.

Il controllo di flusso è quel meccanismo necessario per evitare di provocare overflow nel buffer del ricevitore. Per questo motivo nell'header TCP viene specificato, attraverso il campo *Window size*, lo spazio disponibile nel buffer di ricezione (socket receiver buffer). In questo modo il trasmettitore evita di inviare segmenti se lo spazio disponibile nel buffer del ricevitore non è sufficiente.

Il controllo di congestione permette di limitare il più possibile i fenomeni di congestione all'interno della rete dovuti all'eccessivo traffico che provoca overflow nei buffer dei router, causando perdita di pacchetti. Questo controllo, che viene applicato esclusivamente agli estremi della connessione, cerca di stimare lo stato di congestione della rete osservando eventuali fenomeni di perdita di pacchetti; quando questi si verificano, viene diminuito il rate di trasmissione limitando la quantità di dati che si possono inviare. Gli eventi che indicano al trasmettitore che si è verificato un fenomeno di perdita sono la mancata ricezione di ACK allo scadere del timer (time-out) e la ricezione di ACK duplicati. Esistono vari algoritmi utilizzati per gestire la congestione; l'implementazione di FreeBSD utilizza NewReno di default, ma altri possono essere scelti (Vegas, CUBIC, H-TCP, Hamilton Delay, CAIA-Hamilton Delay, CAIA-Delay Gradient).

3.2.5.1 Trasmissione dati

Nel caso di socket TCP, l'interfaccia tra il livello socket e il network stack per la trasmissione dei dati, *pru_send()* [Sezione 3.1.1], riferisce alla funzione *tcp_usr_send()*. Questa funzione controlla inizialmente lo stato del protocollo TCP, se la connessione non è stabilita, allora procede ad instaurarla e successivamente pone i dati da inviare, precedentemente trasferiti in una catena di mbuf, nel socket buffer di invio attraverso

la funzione *sbappendstream()*. A questo punto viene invocata la routine di output del protocollo TCP [Figura 3.2.4 a]:

```
static int
tcp_output(struct tcpcb *tp)
```

Se il controllo di flusso lo permette, tcp_output() invia immediatamente i dati dopo averli incapsulati in un segmento TCP.

Le operazioni per creare ed inviare un segmento sono simili a quelle eseguite in UDP. La differenza principale è che in UDP il segmento contenente l'intero messaggio proveniente dal livello applicazione viene trasmesso immediatamente, mentre in TCP, entrano in azione i meccanismi di controllo di flusso e controllo di congestione che possono ritardare la trasmissione. Inoltre TCP è un protocollo stream-based, quindi i dati vengono trattati come un flusso ordinato di byte, per questo motivo il messaggio proveniente dal livello applicazione può essere suddiviso in più segmenti TCP o aggregato per formare un unico segmento. All'interno di tcp_output(), in funzione dei meccanismi di controllo di flusso e controllo di congestione, viene decisa la dimensione che il segmento può avere. Se la dimensione dei dati da inviare supera *mss* (maximum segment size) verranno generati e inviati più segmenti all'interno della stessa chiamata di tcp_output.

La routine tcp_output() alloca un mbuf destinato a contenere gli headers TCP, IP e del livello link (es. Ethernet). Se la dimensione dei dati da inviare non è superiore allo spazio disponibile nell'mbuf che contiene gli headers, allora vengono copiati dal socket send buffer attraverso la routine m_copydata() [Sezione 2.4.1], altrimenti viene usata la funzione m_copym() [Sezione 2.4.1] per collegare i dati all'mbuf contenete gli headers formando così una catena di mbuf. La funzione m_copym() non effettua una copia dei dati ma semplicemente alloca degli mbuf che contengono un riferimento alle pagine cluster esterne che contengono i dati, incrementando il reference counter. A questo punto, attraverso la funzione *tcpip_fillheaders()*, vengono creati gli headers IP e TCP attraverso un template e completati con le informazioni mantenute nella inpcb (indirizzo IP e porte sorgente e destinatario, TOS, TTL). Successivamente l'header TCP viene completato impostando il numero di sequenza, i flags (es. se si stratta di dati out-of-band viene settato il flag URG e il campo *Urgent pointer* dell'header TCP conterrà l'offset dell'ultimo byte dei dati "urgenti", quando si richiede una connessione viene settato il flag SYN, quando si richiede di terminare una connessione si setta il flag FIN, se ACK è settato indica che il campo Ack number contiene un ack valido), infine *Window size* viene impostato per informare l'altro peer dello spazio disponibile nel socket recv buffer. A questo punto l'header TCP è pronto e l'ultimo step, prima di passare il segmento al livello IP, è effettuare la checksum parziale sullo pseudo-header come in UDP [Sezione 3.2.6]. La checksum totale verrà completata via hardware, dalla scheda di rete, o via software dal livello IP. Come abbiamo già visto in UDP, il segmento viene passato al livello inferiore attraverso la routine di output del protocollo IP: *ip_output()* [Figura 3.2.4 a].

3.2.6 Internet Protocol

Dopo aver visto i protocolli di trasporto, analizziamo il livello sottostante. L'Internet Protocol (IPv4) è il livello responsabile dell'indirizzamento e routing host-to-host, forwarding dei pacchetti e frammentazione e riassemblaggio dei pacchetti. A differenza dei protocolli di trasporto, che sono in esecuzione solo negli host e sono collegati direttamente ai socket, il protocollo IP viene eseguito anche nei router e può effettuare il forwarding dei pacchetti, o generare dei messaggi di errore attraverso il protocollo ICMP (Internet Control Message Protocol) quando si verificano determinate situazioni.

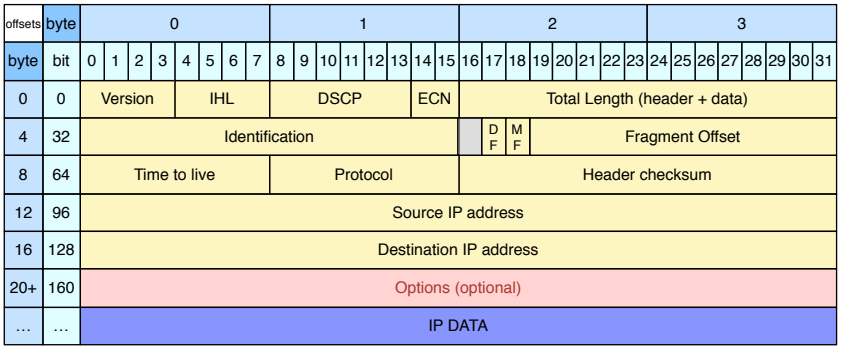

Figura 3.2.8: IP header

All'interno dell'header IP [Figura 3.2.8] troviamo i campi che contengono gli indirizzi IP del mittente e del destinatario (*Source IP Address*, *Destination IP Address*), un campo che indica la versione del protocollo IP (*Version*), i campi che indicano la lunghezza dell'header (*IHL*) e dell'intero pacchetto (*Total length*). I campi *ID*, *Frag. Flags* (DF - Don't Fragment; MF - More Fragments) e *Fragment offset* vengono utilizzati quando si rende necessario frammentare il pacchetto IP per poi essere riassamblato nell'host destinatario. Il campo *TTL* viene utilizzato per evitare che il pacchetto rimanga in circolo per un tempo indefinito nella rete, questo campo viene decrementato ad ogni hop dai router, quando è 0 viene scartato e un messaggio di errore ICMP viene inviato al mittente. Il campo *Protocol* contiene l'identificativo del protocollo a livello superiore e viene usato per il processo di *decapsulation*. Infine è presente il campo *header checksum* destinato a contenere la checksum calcolata esclusivamente sull'header IP con lo stesso metodo illustrato nella [Sezione 3.2.6]. L'header IP inoltre può contenere dei campi opzionali, la loro presenza è deducibile dal campo IHL che indica una lunghezza dell'header maggiore di 20 bytes.

3.2.6.1 Trasmissione dati

I segmenti generati dal livello di trasporto vengono passati al livello internet [Figura 3.2.4] attraverso la seguente funzione:

```
int
ip_output(struct mbuf *m, struct mbuf *opt,
```

```
struct route *ro, int flags,
struct ip_moptions *imo, struct inpcb *inp)
```

m	catena di mbuf che contiene il segmento da inviare (l'header IP è già presente e parzialmente completato)
opt	mbuf opzionale che contiene opzioni IP da inserire dopo l'header
ro	route cache
flags	indicano alcune opzioni (pacchetto broadcast, bypass routing table, etc.)
imo	opzioni per trasmissioni multicast
inp	puntatore alla *Internet protocol control block* associato al socket

Quando la routine ip_output() riceve un segmento dal livello di trasporto, come prima operazione inserisce eventuali opzioni IP, specificate con il parametro *opt*, dopo l'header IP. A questo punto vengono completati i campi restanti dell'header IP (version, IHL, IP). Successivamente, se non viene specificata la route cache *ro*, o la cache non è più valida, viene calcolata la rotta che il pacchetto deve avere (interfaccia di uscita e next-hop). Se la destinazione è un indirizzo multicast, vengono gestiti eventuali opzioni specificate con il parametro *imo* e viene controllata che l'interfaccia di uscita supporti trasmissioni multicast; se la destinazione è un indirizzo broadcast viene verificato che l'interfaccia di uscita sia abilitata a gestire questo tipo di traffico. Dopo aver applicato eventuali regole di filtraggio, viene completata la checksum del protocollo di trasporto [Sezione 3.2.4] con la funzione *in_delayed_cksum()* se l'interfaccia di uscita non è in grado di eseguire questa operazione via hardware. Infine se la dimensione del pacchetto è inferiore all'MTU (Maximum Transmission Unit), il pacchetto viene inviato al livello sottostante (data-link) attraverso la funzione *ifp->if_output()* (ifp è il puntatore alla struttura dati che rappresenta l'interfaccia di uscita individuata durante il calcolo della rotta [Figura 3.3.1]). Altrimenti se la dimensione è maggiore dell'MTU si rende necessario eseguire il meccanismo di frammentazione, messo a disposizione dal protocollo IP, attraverso la funzione *ip_fragment()*.

3.2.7 Internet Protocol version 6

Il motivo principale che ha portato a sviluppare un nuovo protocollo Internet (IPv6) è stato lo spazio di indirizzamento insufficiente fornito da IPv4. Come si può vedere dall'header del nuovo IPv6 [Figura 3.2.9], gli indirizzi passano da 32 bit in IPv4 a 128bit in IPv6.

Oltre a questa caratteristica, IPv6 apporta anche altre migliorie:

- header di dimensione fissa (40 bytes)

  L'elaborazione di un header di dimensione fissa risulta molto più rapida. Eventuali opzioni sono rese disponibili con la tecnica degli *extension header*; in questo caso vengono inseriti ulteriori header subito dopo l'header IPv6, ognuno di essi

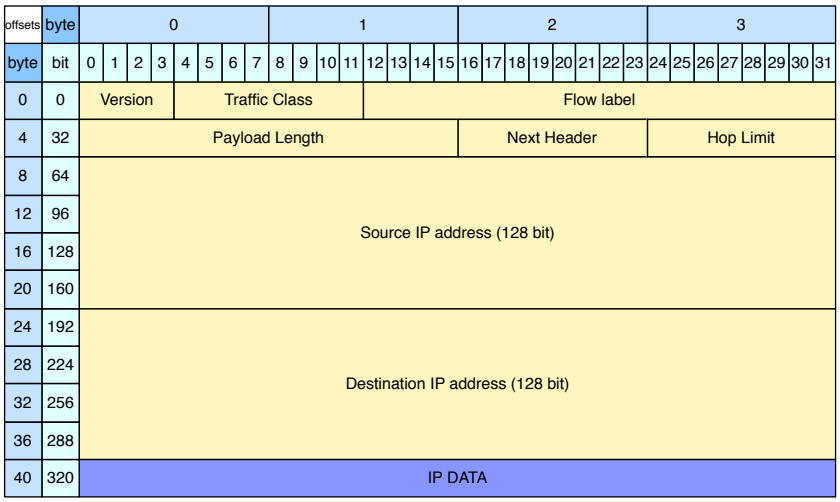

Figura 3.2.9: IPv6 header

ha un campo *Next Header* che permette di concatenare più headers fra loro fino al livello di trasporto (TCP/UDP). Uno dei possibili extension header è il *Fragment Header* utilizzato per frammentare il pacchetto.

- frammentazione non eseguita nei router

  La frammentazione viene effettuata solo dalla sorgente del pacchetto che stima precedentemente l'MTU dell'intero path.

- eliminazione checksum dell'header IP

  Questa checksum risulta ridondante in quanto il livello di trasporto effettua la checksum su tutto il segmento generato (header + dati) e il livello data-link effettua il CRC su tutto il pacchetto da inviare, per questi motivi è stato deciso di eliminare il campo checksum nell'header IPv6.

Tutte queste migliorie sono state fatte per diminuire il carico di lavoro sui router della rete, aumentando così il throughput del core di Internet.

3.3 Network Interfaces

L'ultimo livello del sottosistema di networking [Figura 3.0.1] è rappresentato dalle interfacce di rete. Questo livello ha la responsabilità di effettuare *encapsulation* dei pacchetti in frame (*decapsulation* in ricezione) e, attraverso il device driver, pilotare la scheda di rete per inviare fisicamente il frame sulla rete. Per le interfacce di rete dello stesso tipo (es. Ethernet) il protocollo data-link è implementato separatamente e condiviso tra i device drivers. Ogni interfaccia di rete è rappresentata, all'interno del sistema, dalla struttura dati *struct ifnet* [Figura 3.3.1].

All'interno della struct ifnet vengono mantenute diverse informazioni inizializzate dal driver della scheda quando essa viene riconosciuta dal sistema:

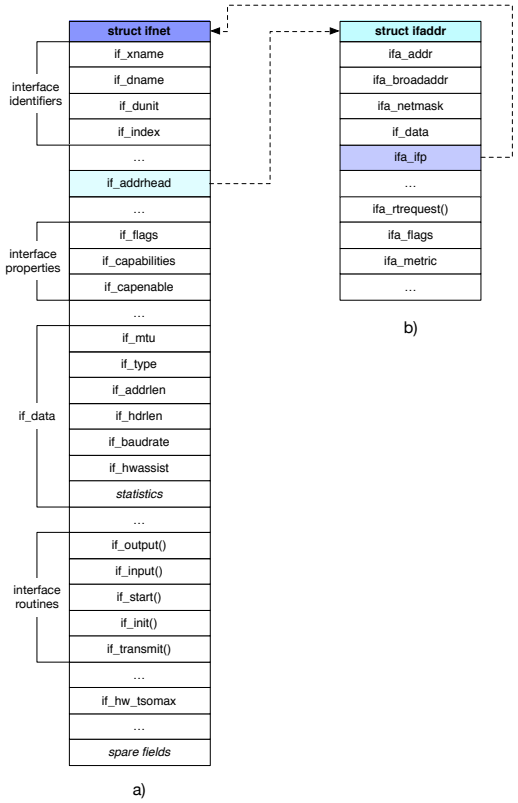

Figura 3.3.1: struct ifnet

- informazioni che identificano l'interfaccia

  Per esempio *if_xname* contiene il nome dell'interfaccia più il numero dell'unità, *if_dname* contiene il nome del driver, *if_index* identifica univocamente l'interfaccia nel sistema.

- indirizzi associati all'interfaccia

  if_addrhead contiene la lista di tutti gli indirizzi associati all'interfaccia. Ogni indirizzo è rappresentato dalla struttura dati *struct ifaddr* che contiene l'indirizzo *ifa_addr*, la maschera di sottorete *ifa_netmask*, un puntatore *ifa_ifp* alla struct ifnet a cui è associato e altre informazioni.

- proprietà dell'interfaccia

  if_flags contiene dei flags che identificano sia lo stato dell'interfaccia (up/down) sia delle funzionalità disponibili (es. broadcast e/o multicast supportati).

  if_capabilities contiene informazioni sulle funzionalità offerte dalla scheda di rete (es. meccanismi di offloading)

  if_capenable identifica le funzionalità abilitate descritte in if_capabilities

- caratteristiche dell'interfaccia

if_ data è una struttura dati che contiene tutte le caratteristiche dell'interfaccia come l'MTU (*if_ mtu*), la tecnologia (*if_ type* - es. Ethernet) e delle statistiche sui frame trasmessi e ricevuti. Inoltre, la struttura if_data, fornisce il campo *if_ hwassist* che contiene le funzionalità di offloading abilitate nella scheda, già presenti in if_capenable ma espressi in modo compatibile con il campo m_pkthdr.csum_flags presente negli mbuf [Figura 2.4.1 b] (es. CSUM_TSO, CSUM_IP_TCP, etc).

- puntatori a routines per gestire l'interfaccia

  if_ init() è utilizzata per inizializzare la scheda

  if_ ioctl() è utilizzata per gestire *ioctl()* effettuate sulla scheda. Attraverso la system call *ioctl()* si possono modificare diversi parametri associati alla scheda e mantenuti nella struct ifnet. Per esempio la *ioctl()* viene utilizzata per aggiungere un indirizzo alla scheda o per abilitare/disabilitare funzionalità di offloading o per cambiare MTU.

  if_ output() è utilizzata dal livello superiore (es. IP) per passare il pacchetto al livello data-link. Per i dispositivi Ethernet, if_output() punta alla funzione *ether_ output()* che descriveremo nella sezione seguente.

  if_ transmit() è utilizzata per passare il frame al device driver che provvederà alla trasmissione.

3.3.1 Trasmissione dati

Un pacchetto generato dal livello IP viene inviato all'ultimo livello data-link attraverso la routine indirizzata dal campo if_output() della struct ifnet associata all'interfaccia di rete attraverso la quale il pacchetto deve essere trasmesso [Figura 3.2.4]. Nel caso di dispositivi Ethernet, questo campo punta alla funzione:

```
int
ether_output(struct ifnet *ifp, struct mbuf *m,
        struct sockaddr *dst, struct route *ro)
```

ifp	interfaccia di rete da utilizzare per la trasmissione
m	catena di mbuf che contiene il pacchetto da inviare
dst	indirizzo del next-hop (gateway) determinato nel livello IP
ro	route cache

Il compito principale di questa funzione è di creare l'header Ethernet [Figura 3.3.2]. Per fare questo è necessario conoscere il MAC address del netx-hop a cui il pacchetto è indirizzato. Di conseguenza come prima operazione, la funzione ether_output() controlla se il MAC è presente nella route cache. Se l'indirizzo non è in cache allora

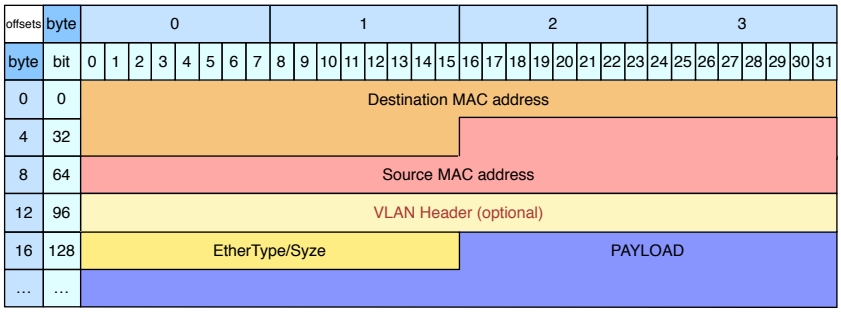

Figura 3.3.2: Ethernet header

è necessario scoprire il MAC address, attraverso l'indirizzo IP del next-hop contenuto in *dst* e specificato dal livello IP, utilizzando il protocollo ARP (Address Resolution Protocol). A questo punto l'header Ethernet viene costruito e inserito in testa al pacchetto, contenuto nella catena di mbuf *m*. Dopo aver applicato eventuali regole di filtraggio, il frame viene passato al device driver attraverso la funzione *ifp->if_ transmit()*.

Capitolo 4

Meccanismi hardware di offload

Le schede di rete negli ultimi anni hanno subito un notevole sviluppo per incrementare il rate di trasmissione. Soprattutto con le più recenti interfacce, che raggiungono un rate teorico di 10 Gbit/s, il collo di bottiglia è rappresentato dal network stack del sistema operativo, non in grado di saturare l'intera banda a disposizione. Per questo motivo sono stati inseriti dei meccanismi "offload", realizzati via hardware all'interno delle schede, per diminuire il carico di lavoro della CPU e quindi aumentare il throughput.

I meccanismi principali che andremo ad analizzare sono:

1. Checksum offload
2. Large Segmentation Offload
3. Large Receive Offload

FreeBSD mette a disposizione un programma di utilità (*ifconfig*) che consente di assegnare un indirizzo all'interfaccia di rete e di configurare i parametri della scheda. Tutti i comandi offerti da *ifconfig* vengono implementati con ioctl() per comunicare con il device driver.

L'utility *ifconfig* offre un'interfaccia CLI (Command Line Interface), quindi può essere utilizzato attraverso una shell o all'interno di script di configurazione del sistema:

```
ifconfig interface address_family [address] [parameters]
```

4.1 Checksum offload

Il meccanismo di "checksum offload" viene utilizzato sia per la checksum del livello di trasporto (TCP - UDP) che per l'header IP. Attraverso questo sistema, la scheda di rete è in grado di effettuare la checksum del livello di trasporto sfruttando un circuito hardware. In questo modo vengono risparmiati notevoli cicli di clock della CPU, che può evitare di calcolare la checksum via software, incrementando così il throughput.

Il calcolo della checksum via hardware inizialmente era stato implementato solo per pacchetti IPv4, ma le schede più moderne prevedono il supporto anche per IPv6.

In FreeBSD questo meccanismo può essere abilitato o disabilitato attraverso ifconfig; i parametri che possono essere utilizzati sono i seguenti:

- txcsum, txcsum6, rxcsum, rxcsum6

  Questi parametri permettono di abilitare "checksum offloading" in trasmissione (txcsum) e in ricezione (rxcsum). txcsum6 e rxcsum6 sono analoghi ma per pacchetti IPv6.

- -txcsum, -txcsum6, -rxcsum, -rxcsum6

  Questi parametri permettono di disabilitare "checksum offloading" in trasmissione (-txcsum) e in ricezione (-rxcsum). -txcsum6 e -rxcsum6 sono analoghi ma per pacchetti IPv6.

Esempio. `ifconfig ix0 -txcsum`

Come abbiamo già illustrato, l'impostazione dei parametri di una scheda di rete, attraverso l'utility ifconfig, si traduce in una ioctl() che permette di comunicare con il device driver. Il driver a questo punto modifica i campi relativi all'interno della *struct ifnet* [Figura 3.3.1] associata all'interfaccia, in funzione dei parametri specificati. In particolare vengono settati o resettati i flag in:

- ifp->if_capenable
  - IFCAP_RXCSUM, IFCAP_RXCSUM_IPV6

    se questi flags sono settati indicano che "checksum offload" in ricezione è abilitato rispettivamente per pacchetti IPv4 e IPv6
  - IFCAP_TXCSUM, IFCAP_TXCSUM_IPV6

    se questi flags sono settati indicano che "checksum offload" in trasmissione è abilitato rispettivamente per pacchetti IPv4 e IPv6

- ifp->if_hwassist
  - CSUM_IP

    se questo flag è settato indica che "checksum offload" per l'header IPv4 in trasmissione è abilitato
  - CSUM_IP_TCP, CSUM_IP6_TCP

    se questi flags sono settati indicano che "checksum offload" per segmenti TCP (IPv4, IPv6) in trasmissione è abilitato
  - CSUM_IP_UDP, CSUM_IP6_UDP

    se questi flags sono settati indicano che "checksum offload" per segmenti UDP (IPv4, IPv6) in trasmissione è abilitato

4.2 Large Segmentation Offload

Il Large Segmentation Offload (LSO) o Large Send Offload è un'altra tecnica utilizzata nelle moderne schede di rete per incrementare il throughput di reti ad alta velocità, diminuendo l'utilizzo della CPU. Questo meccanismo consente al network stack di generare pacchetti indipendemente dall'MTU. Sarà poi compito della scheda di rete suddividerlo in più pacchetti in modo tale che questi possano essere trasmessi. Questo meccanismo è noto sopratutto come TCP Segmentation Offload (TSO), poiché viene applicato a pacchetti TCP, dato che questi possono essere segmentati in modo molto efficiente, in quanto, in TCP, i dati sono trattati come un flusso (stream) ordinato di byte.

L'uso di pacchetti di grandi dimensioni nelle comunicazioni di rete diminuisce drasticamente il carico della CPU. Però, la retrocompatibilità e link lenti, impongono l'uso di MTU pari a 1500 byte. Per questo motivo il TCP è costretto a generare segmenti che rispettino queste dimensioni (IP_hdr + TCP_hdr + TCP_data $\leq$ MTU), ma grazie al TSO i segmenti generati dal TCP possono arrivare fino a 64KB (limite imposto dal campo Length dell'header IP).

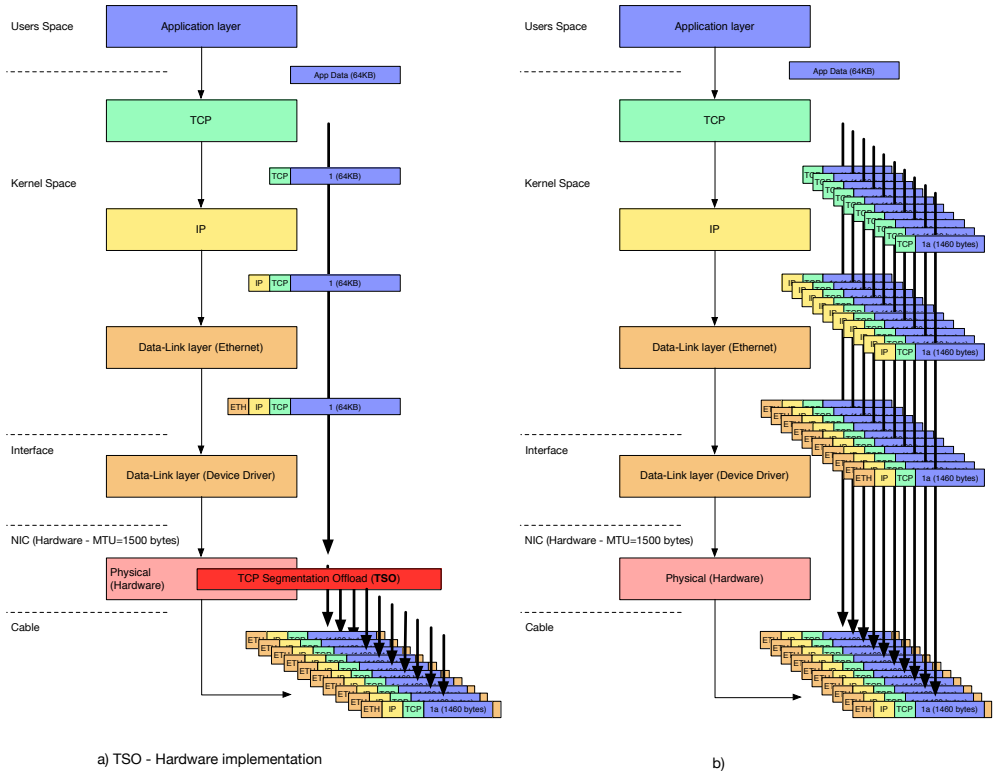

Figura 4.2.1: TCP Segmentation offload

Esempio. Nel caso in cui il TSO non sia disponibile e l'MTU sia pari a 1500 bytes [Figura 4.2.1 b], un messaggio da 64KB dovrebbe essere suddiviso in 45 segmenti TCP, ognuno contenente 1460 bytes di dati (+20 bytes header IP, +20 bytes header TCP) per essere trasmessi dalla scheda di rete. Per ognuno dei 45 segmenti bisognerà

costruire l'header TCP [Sezione 3.2.5.1], poi invocare ip_output() [Sezione 3.2.6.1] che provvederà a costruire l'header IP e successivamente ether_output() [Sezione 3.3.1] che costruirà l'header Ethernet e passerà il controllo al device driver per la trasmissione. Nel caso in cui il TSO sia disponibile ed abilitato [Figura 4.2.1 a], l'attraversamento del network stack sarebbe fatto un'unica volta con un pacchetto che trasporta l'intero messaggio, riducendo notevolmente il carico della CPU; sarà poi compito della scheda di rete effettuare la segmentazione in funzione dell'MTU.

Dato che il guadagno maggiore è nel ridurre il numero di volte in cui si attraversa il network stack, si potrebbe ottenere lo stesso risultato senza supporto hardware, ma posticipando la segmentazione il più tardi possibile, prima che il pacchetto debba essere passato al device driver. Ed è proprio questo che abbiamo realizzato in questo lavoro di tesi e che descriveremo nel capitolo seguente.

In FreeBSD il TSO può essere gestito con ifconfig, per impostarlo sulle singole schede, o attraverso il meccanismo delle sysctl[1]:

- ifconfig
  - i parametri che possono essere utilizzati con ifconfig sono:
    - tso, tso6
      Questi parametri permettono di abilitare il TSO su IPv4 (tso) e IPv6 (tso6).
    - -tso, -tso6
      Questi parametri permettono di disabilitare il TSO su IPv4 (-tso) e IPv6 (-tso6).

    Esempio. `ifconfig ix0 -tso`
  - i flags che vengono settati o resettati dal device driver in *struct ifnet* [Figura 3.3.1] sono:
    - ifp->if_capenable
      - IFCAP_TSO4, IFCAP_TSO6
        se questi flags sono settati indicano che il TSO è abilitato rispettivamente per pacchetti IPv4 e IPv6
    - ifp->if_hwassist
      - CSUM_IP_TSO, CSUM_IP6_TSO
        se questi flags sono settati indicano che il TSO è abilitato rispettivamente per pacchetti IPv4 e IPv6

- sysctl
  - sysctl net.inet.tcp.tso=x
    in questo modo è possibile disabilitare/abilitare (x=0/x=1) il TSO nel network stack (su tutte le connessioni), indipendemente dalle singole interfacce di rete.

[1] utility che permette di visualizzare e impostare dei parametri del kernel a runtime

4.3 Large Receive Offload

L'ultimo meccanismo che andiamo ad analizzare è il Large Receive Offload (LRO) o Receive Side Coalescing (RSC). Questo sistema rappresenta l'operazione inversa del TSO e viene eseguita dal ricevitore. Le motivazioni dietro questa tecnologia sono analoghe al TSO: i pacchetti ricevuti dalla scheda di rete vengono aggregati nel minor numero di pacchetti possibili prima di essere passati al network stack riducendo così il numero di attraversamenti e di conseguenza l'utilizzo della CPU.

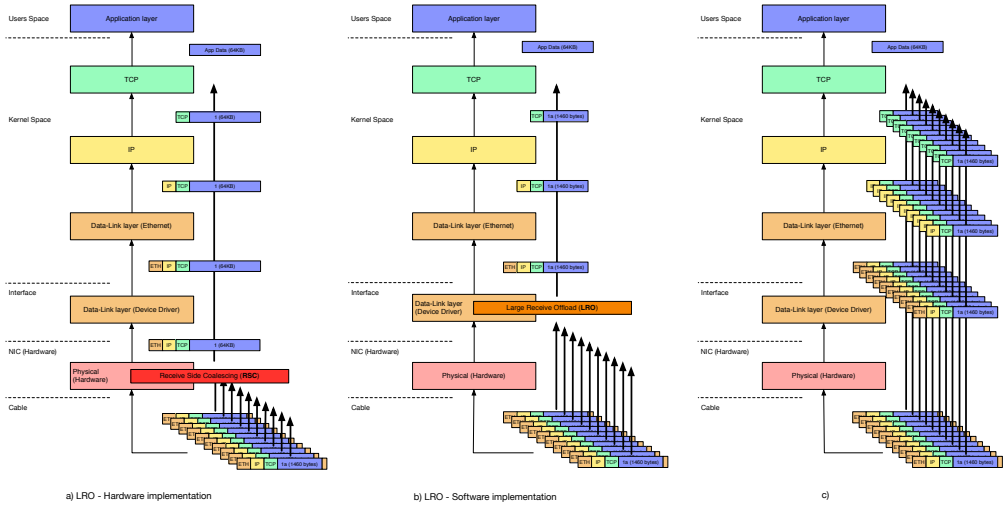

Figura 4.3.1: Large Receive Offload

Questo meccanismo è implementato sia via hardware (Intel lo identifica con il termine RSC [7]) [Figura 4.3.1 a], dalle più recenti interfacce, ma anche via software in FreeBSD (LRO) [Figura 4.3.1 b]. L'implementazione software spesso viene preferita poiché quella hardware presenta notevoli limitazioni dovute soprattutto alla mancanza di risorse in quanto si necessitano di diversi buffer per effettuare l'aggregazione dei pacchetti. Per esempio nel device driver delle interfacce della famiglia Intel(R) 10Gb Ethernet (ixgbe), l'RSC (implementazione hardware dell'LRO) viene disabilitata poiché supporta esclusivamente pacchetti IPv4 ed è in grado di gestire solamente 32 connessioni TCP distinte contemporaneamente; mentre viene implementato nel driver il supporto all'LRO software presente in FreeBSD che è in grado di gestire anche pacchetti IPv6.

Come per gli altri meccanismi, anche questo viene gestito attraverso l'utility ifconfig, i parametri che possono essere utilizzati sono:

- lro

  Questo parametro permette di abilitare l'LRO.

- -lro

  Questo parametro permette di disabilitare l'LRO.

  Esempio. `ifconfig ix0 -lro`

Capitolo 5

Generic Segmentation Offload

Dopo aver illustrato il network stack di FreeBSD e i meccanismi offload, in questo capitolo descriviamo in dettaglio le ottimizzazioni che abbiamo apportato, che rappresentano il lavoro principale di questa tesi.

Come abbiamo descritto nella [Sezione 4.2], l'utilizzo di pacchetti di grandi dimensioni riduce notevolmente il carico della CPU. Per questo motivo le schede di rete più moderne prevedono l'implementazione del TSO in hardware. Nonostante questo, si rende necessaria un'implementazione software (Generic Software Offload - GSO) [Figura 5.1.1] nel caso in cui l'hardware non è di supporto, per esempio nella comunicazione tra macchine virtuali oppure con schede di rete datate o aventi bug nell'implementazione HW del TSO. Inoltre la nostra implementazione supporta anche la frammentazione IP per quei protocolli di trasporto (es. UDP) che non permettono la segmentazione dei dati da trasmettere [Figura 5.1.2].

Il Generic Segmentation Offload (GSO) rappresenta quindi l'implementazione software del TSO. Le motivazioni che sono dietro questa scelta sono le stesse: ridurre al minimo l'attraversamento del network stack.

5.1 Implementazione GSO

Per ridurre al minimo l'utilizzo della CPU, la soluzione ideale sarebbe quella di segmentare il pacchetto nel livello più basso possibile, idealmente all'interno del device driver appena prima di inviarlo alla scheda. Purtroppo questa soluzione comporterebbe la modifica di ogni singolo driver, di conseguenza, per ridurre al minimo le modifiche da apportare al sistema operativo FreeBSD, abbiamo deciso di effettuare la segmentazione appena prima che il pacchetto debba essere passato al driver [Figura 5.1.1].

5.1.1 Gestione GSO

Il meccanismo del GSO può essere gestito attraverso *sysctl* per abilitarlo/disabilitarlo sia sulle singole schede di rete, sia direttamente nel network stack per tutte le

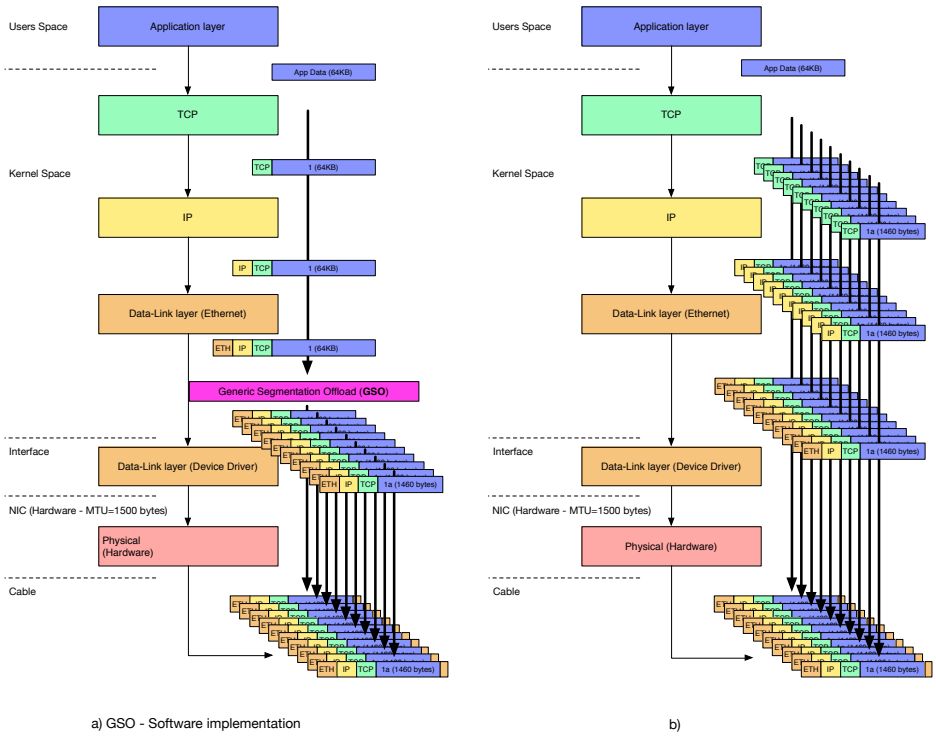

Figura 5.1.1: Generic Segmentation Offload (TCP)

comunicazioni TCP o UDP.

Le sysctl messe a disposizione sono:

- net.inet.tcp.gso

  permette di abilitare/disabilitare il GSO nel network stack (su tutte le connessioni TCP), indipendemente dalle singole interfacce di rete.

- net.inet.udp.gso

  permette di abilitare/disabilitare il GSO nel network stack (su tutte i pacchetti UDP), indipendemente dalle singole interfacce di rete.

- net.gso.dev.XXX.enable_gso

  permette di abilitare/disabilitare il GSO per l'interfaccia di rete XXX

  Esempio. `sysctl net.gso.dev.ix0.enable_gso=1`

- net.gso.dev.XXX.max_burst

  permette di impostare la dimensione massima del burst da segmentare generato dal network stack per l'interfaccia di rete XXX

  Esempio. `sysctl net.gso.dev.em0.max_burst=65536`

5.1.2 Modifiche al network stack

Di seguito illustriamo le modifiche apportate nel network stack di FreeBSD per identificare i pacchetti da segmentare con il GSO.

5.1.2.1 struct ifnet

Per gestire il GSO su ogni singola interfaccia, abbiamo aggiunto, all'interno della *struct ifnet* [Figura 3.3.1] provvisoriamente nei campi "spare", un puntatore alla struttura:

```
struct if_gso {
        struct sysctl_ctx_list clist;    /* sysctl ctx for
            this interface */

        /* GSO parameters for each interface */
        u_int max_burst;          /* GSO burst length limit */
        u_int enable;             /* GSO enable (!=0)*/
};
```

Quando il driver di un'interfaccia viene caricato, vengono allocate le risorse (es. struct ifnet) e inizializzate. Per inizializzare i dispositivi Ethernet appropriatamente, è presente la funzione ether_ifattach(struct ifnet *ifp) che in questo caso viene invocata dal driver dopo aver allocato la struct ifnet. All'interno di ether_ifattach() vengono impostati diversi campi della struct ifnet. Di conseguenza abbiamo inserito in questo punto una chiamata alla funzione:

```
void
gso_ifattach(struct ifnet *ifp)
```

La funzione gso_ifattach() ha il compito di allocare la struct if_gso per questa interfaccia e inserire il puntatore all'interno della struct if_net. Inoltre vengono create le sysctl per gestire le impostazioni dell'interfaccia e collegate ai parametri presenti nella struttura appena allocata.

Quando un driver viene scaricato viene invocata la seguente funzione che ha il compito di liberare la memoria allocata per la struct if_gso e eliminare le sysctl relative all'interfaccia:

```
void
gso_ifdetach(struct ifnet *ifp)
```

5.1.2.2 TCP

Quando viene generato un segmento TCP in tcp_output() [Sezione 3.2.5.1], se la dimensione supera mss (maximum segment size) ma il GSO è abilitato sia nel network stack che nell'interfaccia di uscita, allora viene generato un solo segmento e nel m_pkthdr [Figura 2.4.1] viene impostato il flag CSUM_GSO all'interno del campo *csum_flags*.

Questo flag indicherà ai livelli sottostanti che il pacchetto dovrà essere segmentato prima di essere passato al device driver.

5.1.2.3 UDP e IP

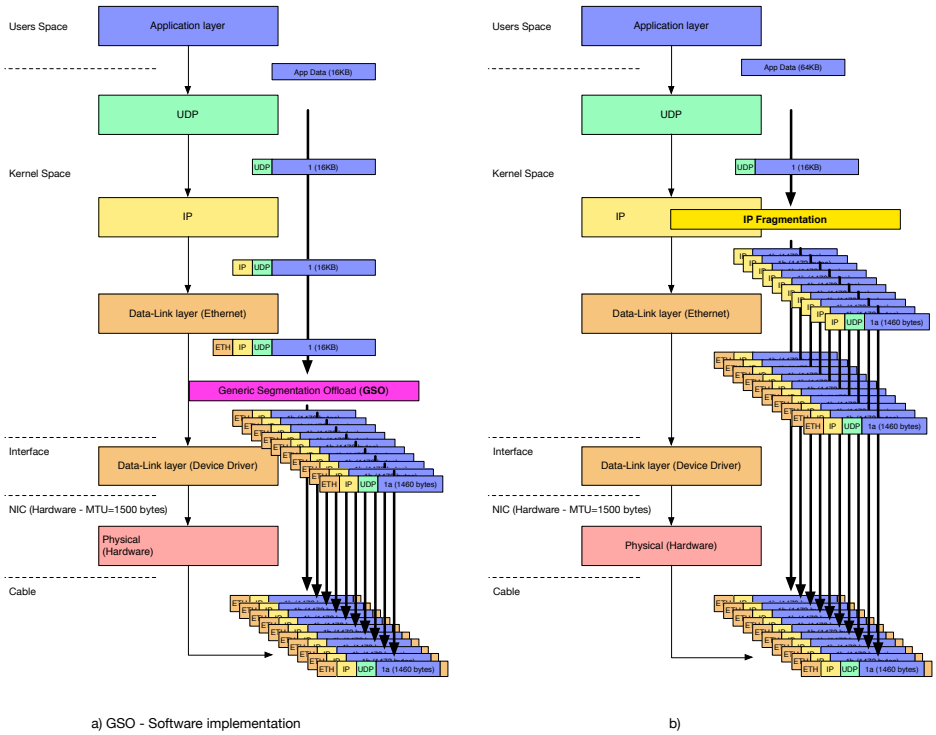

Figura 5.1.2: Generic Segmentation Offload (UDP)

Nel caso in cui venga creato un datagram UDP in udp_output() [Sezione 3.2.3.1], se non è attiva l'opzione "Don't fragment" e il GSO è abilitato nel network stack, allora nel pacchetto viene impostato il flag CSUM_GSO all'interno del campo m_pkthdr. *csum_flags* [Figura 2.4.1]. A questo punto nel livello IP, dopo aver stabilito l'interfaccia di uscita, viene verificato che il GSO è abilitato sull'interfaccia e viene inviato al livello sottostante o viene applicata la frammentazione IP se il GSO non è disponibile.

5.1.2.4 Ethernet

Come abbiamo descritto nella [Sezione 3.3.1] il pacchetto, dopo che l'header Ethernet è stato completato all'interno della funzione *ether_output()*, viene inviato al device driver sfruttando il puntatore a funzione *ifp->if_transmit()*.

La modifica che abbiamo apportato in questa parte è stata quella di intercettare il pacchetto appena prima di essere inviato al driver e, se il GSO è richiesto (flag CSUM_GSO settato all'interno del campo m_pkthdr.*csum_flags*), richiamare la seguente funzione per effettuare la segmentazione:

```
int
gso_dispatch(struct ifnet *ifp, struct mbuf *m,
        u_int mac_hlen, uint16_t mac_type)
```

ifp interfaccia di rete da utilizzare per la trasmissione

m catena di mbuf che contiene il pacchetto da segmentare o frammentare

mac_hlen lunghezza dell'header Ethernet

mac_type indica la presenza del VLAN header [Figura 3.3.2]

La funzione *gso_dispatch()* ha il compito di individuare il protocollo internet utilizzato (IPv4 o IPv6) e il protocollo di trasporto (TCP o UDP) e richiamare le funzioni specifiche che effettuano la segmentazione per pacchetti TCP (*gso_ip_tcp()*) [Figura 5.1.1] o la frammentazione IP (*gso_ipv4_frag()* - *gso_ipv6_frag()*) [Figura 5.1.2].

Queste funzioni hanno il compito di segmentare il pacchetto attraverso la funzione *m_seg()* descritta nella [Sezione 5.1.3] e successivamente modificare opportunamente gli headers (copiati dal pacchetto originale) in ogni segmento.

5.1.3 Segmentazione mbuf

Per suddividere il pacchetto generato dal network stack in pacchetti che possono essere trasmessi dalla scheda di rete, abbiamo realizzato la seguente funzione:

```
static struct mbuf *
m_seg(struct mbuf *m0, int hdr_len, int mss, int *nsegs, char
    * hdr2_buf, int hdr2_len)
```

m0 catena di mbuf che contiene il pacchetto da segmentare

hdr_len lunghezza totale degli headers presenti in testa al pacchetto m0 da copiare in ogni segmento

mss massima quantità di dati che ogni segmento può contenere

nsegs restituisce il numero di segmenti creati

hdr2_buf buffer opzionale che contiene un header aggiuntivo da inserire tra gli headers presenti nel pacchetto iniziale m0 e i dati in ogni segmento (usato per la frammentazione IPv6 dove si rende necessario inserire il "Fragment Header" in ogni frammento)

hdr2_len lunghezza dell'header aggiuntivo

La funzione *m_seg()* restituisce una coda di mbuf (mbuf queue) costituita dai segmenti (mbuf chain) che sono stati generati a partire dal pacchetto originale [Figura 5.1.3].

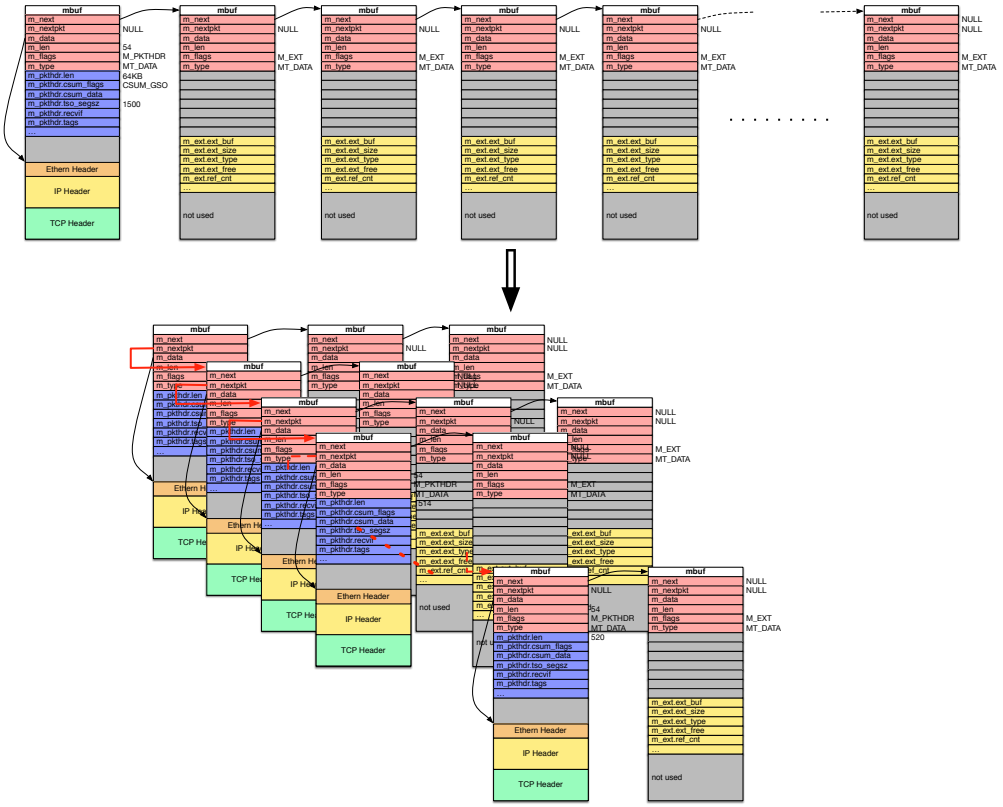

Figura 5.1.3: m_seg()

Per ogni nuovo segmento, la funzione *m_ seg()*, alloca un nuovo mbuf con la routine MGETHDR() [Sezione 2.4.1]. Questa routine restituisce il puntatore ad un mbuf inizializzato con l'intestazione m_pkthdr per poter creare una catena di mbuf. All'interno di questo mbuf vengono copiati i primi *hdr_ len* bytes del pacchetto originale contenuto in m0, a questo punto vengono copiati al più mss bytes dei dati del pacchetto originale a partire dal primo byte non copiato nel segmento precedente (offset) attraverso la funzione *m_ copym()* [Sezione 2.4.1], creando così una catena di mbuf che contiene il nuovo segmento. Se viene specificato il buffer opzionale *hdr2_ buf*, contenete dei dati da aggiungere agli headers già presenti nel pacchetto originale, questi vengono inseriti in ogni segmento tra i primi *hdr_ len* bytes e i dati attraverso la funzione *m_ copyback()* [Sezione 2.4.1].

Tutti i segmenti vengono collegati fra loro per formare una coda di mbuf (mbuf queue) e viene restituito l'indirizzo del primo elemento e il numero di segmenti, attraverso *nsegs*.

5.1.4 Segmentazione TCP

Se il protocollo di trasporto è il TCP, allora *gso_ dispatch()* invocherà la seguente funzione per effettuare la segmentazione:

```
static int
```

```
gso_ip_tcp(struct ifnet *ifp, struct mbuf *m0, int mac_hlen,
    int ip_hlen, int isipv6)
```

ifp interfaccia di rete da utilizzare per la trasmissione

m0 catena di mbuf che contiene il pacchetto da segmentare

mac_hlen lunghezza dell'header Ethernet

ip_len lunghezza dell'header IP

isipv6 flag che indica se il pacchetto è IPv4 o IPv6

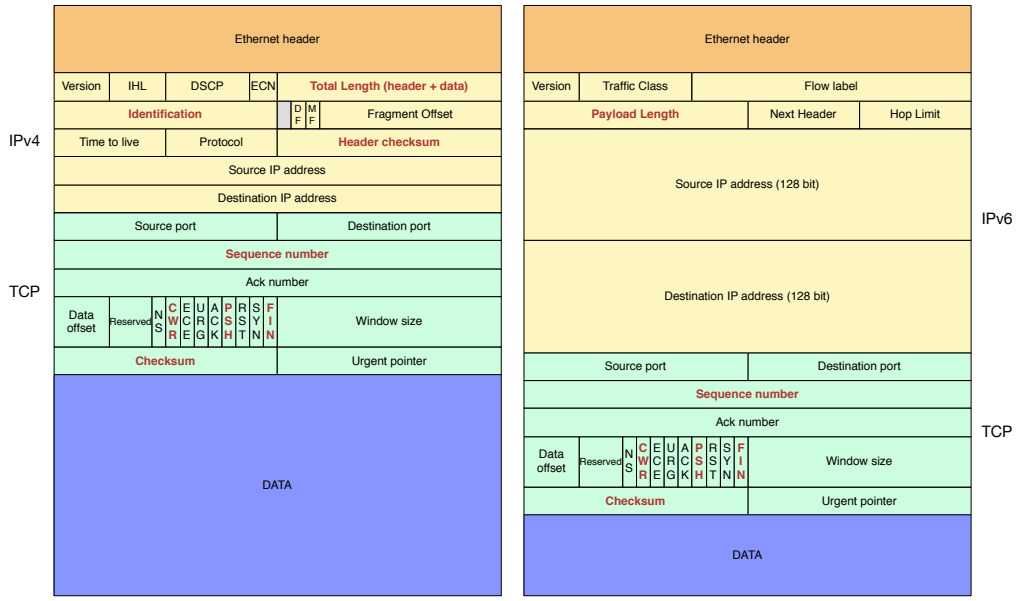

Figura 5.1.4: Segmentazione TCP - Modifiche headers

Questa funzione, dopo aver suddiviso il pacchetto originale attraverso la funzione m_seg(), che ha copiato in ogni segmento gli headers Ethernet, IP e TCP dal pacchetto originale, scorre tutti i segmenti generati per impostare in modo appropriato i campi degli header IP e TCP [Figura 5.1.4]. In particolare per ogni segmento vengono compiute le seguenti azioni:

1. Incrementato il numero di sequenza TCP (*Sequence Number*) in funzione dell'offset dei byte che il segmento contiene
2. Modificati i flags TCP (flags PSH e FIN abilitati solo nell'ultimo segmento, flag CWR abilitato solo nel primo)
3. Impostata la lunghezza del pacchetto nell'header IP (*Total Length* - IPv4 / *Payload Length* - IPv6)
4. Calcolato lo pseudo-header

5. Se l'hardware non è in grado di calcolare la checksum TCP, questa viene calcolata.

6. Se l'hardware non è in grado di calcolare la checksum dell'header IP, questa viene calcolata (solo per IPv4)

7. Il segmento viene inviato al device driver attraverso *ifp->if_ transmit()*

5.1.5 Frammentazione IPv4

Se bisogna segmentare un pacchetto UDP, allora è necessario effettuare la frammentazione IP in quanto il protocollo UDP non prevede questa opzione. Per questo motivo è presente la seguente funzione per effettuare frammentazione IPv4:

```
static int
gso_ipv4_frag(struct ifnet *ifp, struct mbuf *m0, int
    mac_hlen, int ip_hlen)
```

ifp interfaccia di rete da utilizzare per la trasmissione

m0 catena di mbuf che contiene il pacchetto da frammentare

mac_hlen lunghezza dell'header Ethernet

ip_len lunghezza dell'header IP

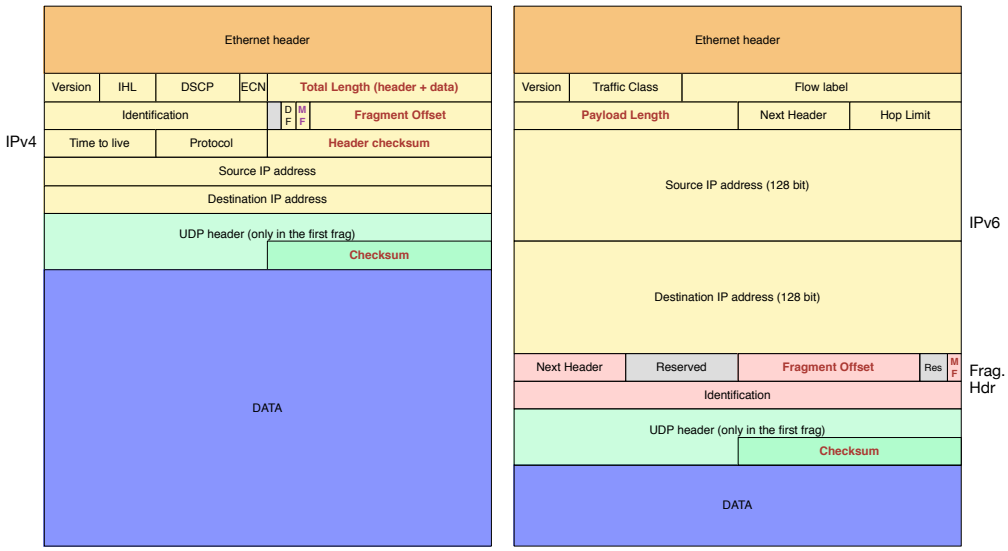

Figura 5.1.5: Frammentazione IPv4 e IPv6 - Modifiche headers

Prima di effettuare la frammentazione è necessario calcolare la checksum del livello di trasporto [Sezione 3.2.4] poiché l'hardware non è in grado di farlo dopo aver diviso il payload in più frammenti IP distinti. A questo punto il pacchetto viene suddiviso, attraverso la funzione m_seg() che copia in ogni segmento gli headers Ethernet e

IP dal pacchetto originale. Successivamente vengono scorsi tutti i frammenti per impostare i campi opportuni all'interno dell'header IP [Figura 5.1.5] attraverso le seguenti azioni:

1. Impostato il campo *Fragment Offset* dell'header IP in funzione dell'offset dei byte che il frammento contiene
2. Impostato il flag *More Fragments* (MF) all'interno dell'header IP ad eccezione dell'ultimo frammento.
3. Impostata la lunghezza del pacchetto nell'header IP (*Total length*)
4. Se l'hardware non è in grado di calcolare la checksum dell'header IP, questa viene calcolata
5. Il frammento viene inviato al device driver attraverso *ifp->if_ transmit()*

5.1.6 Frammentazione IPv6

La funzione che effettua la frammentazione IPv6 è la seguente:

```
static int
gso_ipv6_frag(struct ifnet *ifp, struct mbuf *m0, int
    mac_hlen, int ip_hlen, int nextproto)
```

ifp interfaccia di rete da utilizzare per la trasmissione

m0 catena di mbuf che contiene il pacchetto da frammentare

mac_hlen lunghezza dell'header Ethernet

ip_len lunghezza dell'header IP

nextproto id del protocollo di trasporto

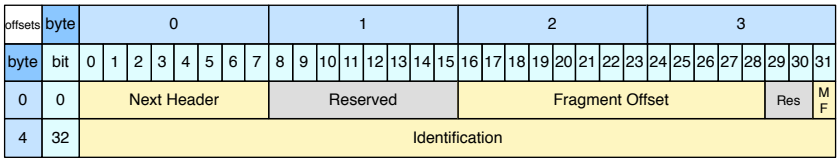

Figura 5.1.6: Fragment header

Anche per IPv6, prima di effettuare la frammentazione è necessario calcolare la checksum del livello di trasporto [Sezione 3.2.4] poiché l'hardware non è in grado di farlo dopo aver diviso il payload in più frammenti IP distinti. Inoltre bisogna creare il *Fragment Header* [Figura 5.1.6] che va inserito in ogni segmento tra l'header IP e i dati. A questo punto il pacchetto viene suddiviso, attraverso la funzione m_seg() che copia in ogni segmento gli headers Ethernet, IP dal pacchetto originale e il Fragment Header precedentemente creato. Successivamente vengono scorsi tutti i frammenti per

impostare i campi opportuni all'interno dell'header IP [Figura 5.1.5] e del Fragment Header attraverso le seguenti azioni:

1. Impostato il campo *Fragment Offset* nel Fragment Header in funzione dell'offset dei byte che il frammento contiene
2. Impostato il flag *More Fragments* (MF) nel Fragment Header ad eccezione dell'ultimo frammento.
3. Impostata la lunghezza del payload del pacchetto nell'header IP (*Payload length*)
4. Il frammento viene inviato al device driver attraverso *ifp->if_ transmit()*

5.2 Risultati

Dopo aver implementato questo nuovo meccanismo all'interno di FreeBSD, abbiamo effettuato diversi esperimenti per osservare le performance.

Gli esperimenti sono stati effettuati sfruttando il software netperf. Questo software consente di stimare il network bandwidth tra due host. Netperf supporta diversi protocolli, tra i quali TCP e UDP, e fornisce dei test predefiniti per osservare le performance.

Le caratteristiche del trasmettitore e del ricevitore utilizzati sono: CPU i7-870 a 2.93 GHz + Turboboost, Intel 10 Gbit NIC e ixgbe driver.

In tutti gli esperimenti nel ricevitore sono stati abilitati tutti i meccanismi offload (Checksum, LRO, GRO[1]), mentre nel trasmettitore è stato disabilitato l'LRO poiché causava diversi problemi che saranno investigati dopo lo svolgimento di questa tesi.

Abbiamo eseguito tre serie di esperimenti per verificare le performance in scenari differenti:

1. Flusso di dati TCP sfruttando il meccanismo di Checksum Offload [Sezione 4.1]

Freq. [GHz]	**TSO**	**GSO**	**none**	**Speedup GSO - none**
2.93	9316.2	9349.0	8937.8	**4.60 %**
2.53	9314.2	9336.4	7298.6	**27.92 %**
2.00	9306.2	9310.2	5880.6	**58.32 %**
1.46	9268.4	8399.6	4375.8	**91.96 %**
1.05	9308.2	6110.4	3161.2	**93.29 %**
0.45	4677.8	2209.0	1338.0	**65.10 %**

Tabella 5.2.1: Esperimenti TCP con Checksum Offload

In questo primo scenario abbiamo confrontato le prestazioni ottenute con un flusso di dati TCP abilitando il TSO hardware, che rappresenta il limite superiore che possiamo raggiungere, successivamente abbiamo disabilitato il TSO e abilitato il GSO ed infine abbiamo disabilitato entrambi. In questi esperimenti

[1] Generic Receive Offload - Implementazione software dell'LRO in Linux

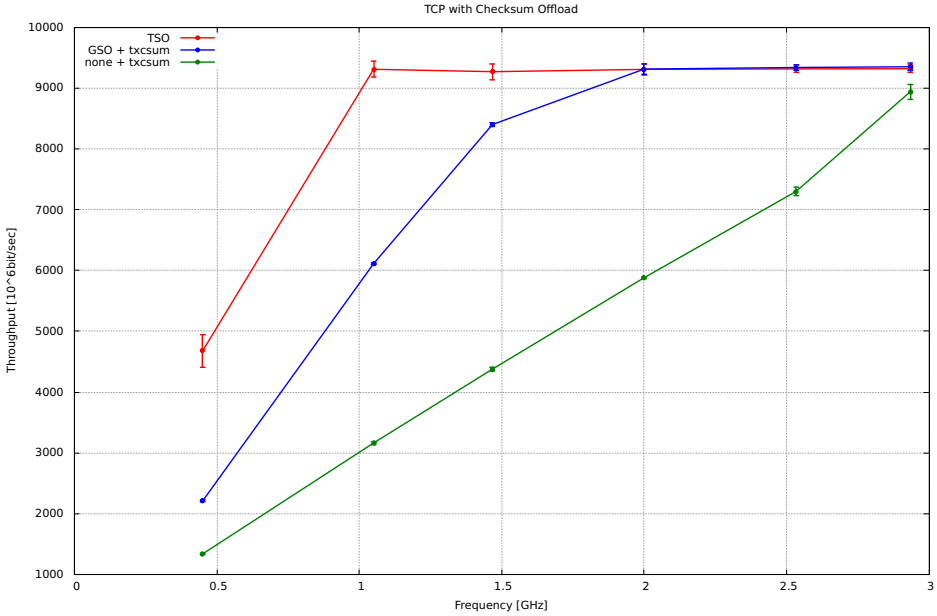

Figura 5.2.1: Esperimenti con flusso TCP (Checksum Offload abilitata)

abbiamo sfruttato il meccanismo di Checksum Offload per calcolare la checksum del pacchetto via hardware nella scheda di rete. (Nel caso in cui il TSO è abilitato la checksum viene sempre calcolata dall'hardware).

Osservando i risultati di questo esperimento [Tabella 5.2.1] si può notare che alla massima frequenza della CPU (2.93 GHz), abilitando il GSO, riusciamo a saturare il link 10 Gbit ottenendo le stesse performance del TSO [Figura 5.2.1]. Di conseguenza, per evidenziare maggiormente lo speedup tra il GSO e un sistema senza meccanismi di offload, abbiamo variato la frequenza di clock della CPU del trasmettitore. In questo modo possiamo osservare che abilitando il GSO riusciamo a saturare il link 10 Gbit fino ad una frequenza di 2 GHz compresa; inoltre notiamo che tra 1.5 e 1 GHz otteniamo lo speedup maggiore (superiore del 90%) che rappresenta effettivamente il risparmio introdotto dal GSO sull'utilizzo della CPU, dato che le basse frequenze della CPU non permettono più di saturare il link 10 Gbit.

2. Flusso di dati TCP senza il meccanismo di Checksum Offload

Freq. [GHz]	**TSO**	**GSO**	**none**	**Speedup GSO - none**
2.93	9316.2	9341.0	7859.0	**18.86 %**
2.53	9314.2	9159.6	6331.2	**44.67 %**
2.00	9306.2	8654.6	5156.2	**67.85 %**
1.46	9268.4	6595.2	3866.8	**70.56 %**
1.05	9308.2	4678.2	2754.2	**69.86 %**
0.45	4677.8	1843.8	1166.8	**58.02 %**

Tabella 5.2.2: Esperimenti TCP senza Checksum Offload

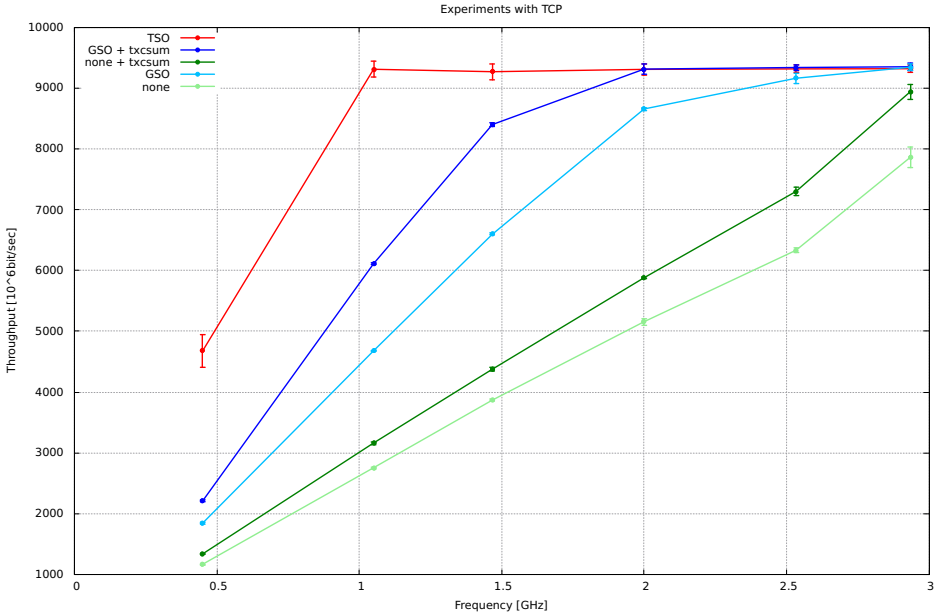

Figura 5.2.2: Esperimenti con flusso TCP

Nella seconda serie di esperimenti abbiamo osservato il comportamento disabilitando il calcolo della checksum via hardware (Checksum Offload), anche in questo caso scalando la frequenza della CPU del trasmettitore per evidenziare lo speedup. Per il TSO questa modifica non influisce in quanto la checksum è sempre calcolata dall'hardware quando viene effettuata la segmentazione.

Come si può vedere dalla [Tabella 5.2.2], il GSO riesce comunque a saturare il link 10Gbit alla frequenza massima, anche senza il supporto della Checksum Offload. Lo speedup massimo che si riesce a raggiungere in questo caso è del 70%.

Freq. [GHz]	**GSO + txcsum**	**GSO**	**Speedup GSO**	**none + txcsum**	**none**	**Speedup none**
2.93	9349.0	9341.0	**0.09%**	8937.8	7859.0	**13.73%**
2.53	9336.4	9159.6	**1.93%**	7298.6	6331.2	**15.28%**
2.00	9310.2	8654.6	**7.58%**	5880.6	5156.2	**14.05%**
1.46	8399.6	6595.2	**27.36%**	4375.8	3866.8	**13.16%**
1.05	6110.4	4678.2	**30.61%**	3161.2	2754.2	**14.78%**
0.45	2209.0	1843.8	**19.81%**	1338.0	1166.8	**14.67%**

Tabella 5.2.3: Confronto esperimenti TCP con e senza Checksum Offload

Dato che in questo caso la checksum viene calcolata via software, consumando cicli di clock della CPU, abbiamo un degrado delle prestazioni rispetto agli esperimenti precedenti [Figura 5.2.2]. Dalla [Tabella 5.2.3] possiamo vedere lo speedup introdotto abilitando il meccanismo di Checksum Offload, sia nel caso in cui il GSO è abilitato (fino al 30%), sia nel caso in cui è disabilitato (tra il 13% e il 15%).

3. Flusso di dati UDP

Freq. [GHz]	GSO	none	Speedup GSO - none
2.93	9266.8	8078.6	**14.71 %**
2.53	7779.0	6621.2	**17.49 %**
2.00	6371.4	5328.8	**19.57 %**
1.46	4855.0	4047.8	**19.94 %**
1.05	3452.0	2867.6	**20.38 %**
0.45	1350.0	1128.0	**19.68 %**

Tabella 5.2.4: Esperimenti UDP

Come ultima serie di esperimenti abbiamo testato le performance del GSO su una comunicazione UDP. In questo caso la checksum deve essere calcolata necessariamente via software, in quanto il pacchetto UDP viene frammentato sfruttando il protocollo IP, di conseguenza il payload sarà suddiviso su più pacchetti IP e la scheda di rete non è in grado di calcolare la checksum del livello di trasporto se il pacchetto ha subito una frammentazione IP.

Analizzando la [Tabella 5.2.4] e la [Figura 5.2.3] si può vedere che anche in questo caso il GSO fornisce uno speedup tra 14 % e 20%.

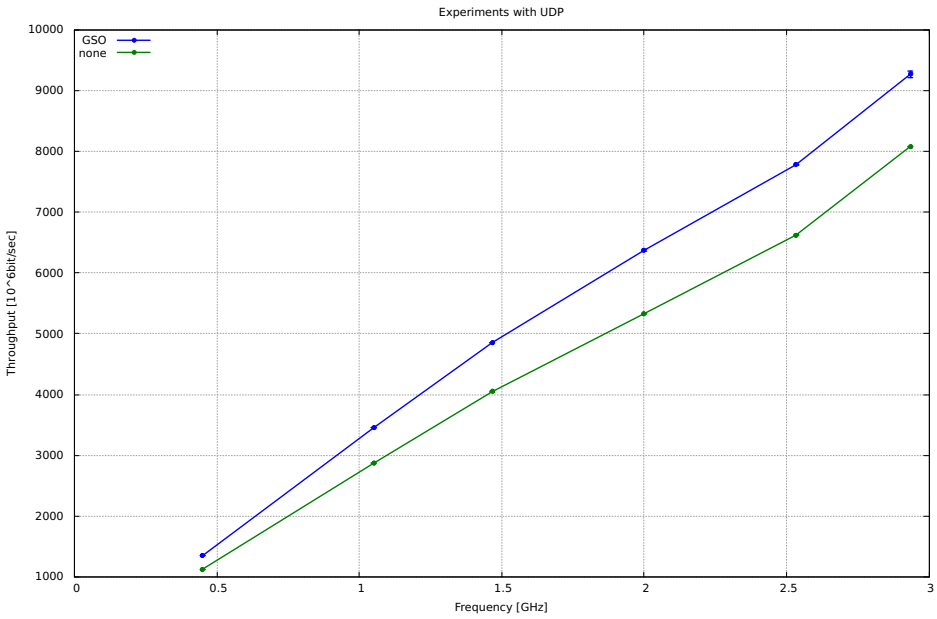

Figura 5.2.3: Esperimenti con flusso UDP

Capitolo 6

Conclusioni

In questa tesi abbiamo ottimizzato il network stack di FreeBSD in trasmissione per reti ad alta velocità; per questo motivo abbiamo concentrato la nostra descrizione sul percorso che i dati seguono nel network stack in trasmissione, tralasciando la ricezione.

Abbiamo preso spunto dal TSO implementato nelle schede di rete più moderne per realizzare un meccanismo software, il GSO, che produca benefici analoghi senza supporto hardware. Infatti il vantaggio maggiore che fornisce il TSO è quello di permettere la generazione di grandi pacchetti TCP, questo provoca una riduzione del numero di volte in cui tutti i livelli del network stack devono essere attraversati, effettuando operazioni ripetitive e non necessarie.

I risultati ottenuti dimostrano che questo vantaggio può essere ottenuto senza l'utilizzo di hardware dedicato (ad eccezione del calcolo della checksum). Il GSO può essere sfruttato li dove l'hardware non offre supporto, per esempio nella comunicazione tra macchine virtuali oppure se il TSO presenta dei malfunzionamenti o semplicemente non è supportato dalla scheda di rete. Inoltre il nostro meccanismo supporta anche traffico UDP che il TSO non è in grado di gestire.

La soluzione software permette di ottenere prestazioni comparabili con il TSO ma con hardware più economico, inoltre consente sia una sicurezza maggiore, in quanto un eventuale bug software può essere corretto molto più facilmente rispetto ad un bug presente nell'implementazione hardware, sia un continuo upgrade, per esempio inizialmente il TSO era disponibile solo per TCP/IPv4 e quindi quelle schede non saranno mai in grado di effettuare segmentazione su comunicazioni TCP/IPv6.

Anche se il GSO non riesce a superare le performance offerte dal TSO hardware sono presenti diversi margini di miglioramento che in futuro possono essere presi in considerazione:

- durante la segmentazione vengono allocati degli mbuf per contenere la copia degli headers del pacchetto originale, per questo motivo si potrebbe pensare di preallocarli per ogni interfaccia, riducendo il costo di questa operazione.

- "software fallback"[1], assumere nel network stack che sia sempre presente un

[1] soluzione software alternativa, di ripiego

meccanismo di segmentazione indipendemente dall'hardware, semplificando così i percorsi del codice e riducendo l'overhead in ogni pacchetto.

Bibliografia

[1] McKusick, Marshall Kirk, and George V. Neville-Neil. *The design and implementation of the FreeBSD operating system.* Addison-Wesley Professional, 2004.

[2] Cevoli, Paul. *Embedded freeBSD cookbook.* Newnes, 2002.

[3] Kong, Joseph. *FreeBSD Device Drivers.* No Starch Press, 2012.

[4] Innocente, Roberto, and Olumide S. Adewale. "Network buffers The BSD, Unix SVR4 and Linux approaches (BSD4. 4, SVR4. 2, Linux2. 6.2)."

[5] Fall, Kevin R., and W. Richard Stevens. *Tcp/ip illustrated.* Vol. 1. Addison-Wesley Professional, 2011.

[6] Herbert, Thomas F. *The Linux TCP/IP Stack: Networking for Embedded Systems.* Charles River Media, 2004.

[7] Intel Corporation. Intel 82599 10 GbE Controller Datasheet, January 2014. Revision 2.9.

[8] Keromytis, Angelos D. "Tagging Data in the Network Stack: mbuf_tags." BSDCon. 2003.

[9] Kurose, James F., Keith W. Ross, and B. Anand. "Computer Networking–A top-Down Approach, 2008."

[10] Binder, James S., et al. "Offload of TCP segmentation to a smart adapter." U.S. Patent No. 5,937,169. 10 Aug. 1999.

[11] FreeBSD and Linux Kernel Cross-Reference. http://fxr.watson.org/

[12] FreeBSD NetworkPerformanceTuning. https://wiki.freebsd.org/ NetworkPerformanceTuning

[13] FreeBSD Developers' Handbook. http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/

[14] FreeBSD Architecture Handbook. http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/index.html