Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

ExoTCP: uno stack di rete minimale e userspace specializzato per il protocollo HTTP

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Gilberto Bertin

Sessione II
Anno Accademico 2013-2014

Only wimps use tape backup:
real men just upload their important stuff on FTP,
and let the rest of the world mirror it.
Linus Torvalds

Introduzione

In questo lavoro di tesi verrà presentata la progettazione e la realizzazione di uno stack di rete TCP/IP minimale e userspace specializzato per il protocollo HTTP.

Il lavoro è motivato dalla curiosità: la curiosità di studiare i protocolli di rete su cui è basato il funzionamento di Internet, la curiosità di mettere in discussione un'architettura (quella di un server Web) che è rimasta pressoché invariata dalla sua prima concezione e la curiosità di capire quale sia il costo che le architetture attuali devono pagare per utilizzare le interfacce generiche per l'accesso alla rete offerte dal sistema operativo.
Oltre allo stack di rete è stato sviluppato un server web che utilizza quest'ultimo, con la finalità di effettuare alcuni test prestazionali.
L'obiettivo è riuscire ad offrire delle performance superiori a quelle di un normale server web, poiché questo proverebbe che l'utilizzo di un'architettura generica avrebbe un costo che porterebbe a non sfruttare appieno l'hardware sottostante.

Indice

Elenco delle figure

Capitolo 1

Motivazioni

L'idea di uno stack TCP/IP specializzato per il protocollo HTTP nasce in seguito a due osservazioni.

1.1 Un Web 2.0 statico

La prima osservazione è che nonostante ci si trovi nel periodo di quello che viene definito Web 2.0, ovvero un Web caratterizzato da un importante utilizzo di contenuti generati dinamicamente, la maggior parte del traffico è dovuta allo scambio di contenuti statici.
I dati scambiati utilizzando il protocollo Hyper Text Transfer Protocol (HTTP [4]) possono essere pagine HTML, fogli di stile CSS e script Javascript, così come contenuti multimediali ai quali si ha accesso con qualsiasi social network o servizio di streaming.

Sono proprio questi ultimi a generare la maggiore quantità di traffico (basti pensare alla quantità di dati che vengono scambiati tra un server web ed un browser sfogliando un album fotografico su Facebook, ascoltando un artista su Spotify o guardando un video su YouTube).
Spesso la gestione di queste risorse viene affidata a sistemi distribuiti specializzati (i Content Delivery Network, o CDN). Il loro compito è quello di servire unicamente contenuti statici in maniera affidabile, utilizzando alcuni accorgimenti ed ottimizzazioni, come ad esempio il caching in RAM delle risposte HTTP o la distribuzione geografica dei datacenter (spesso connessi a multiple dorsali) per permettere l'instradamento delle richieste verso il datacenter geograficamente più vicino.

1.2 Un'architettura mai messa in discussione

La seconda osservazione riguarda la tipica architettura di un server web, che non è mai stata messa in discussione ed è rimasta pressoché invariata da più di vent'anni. Un server web infatti è tipicamente un applicativo in spazio utente che per scambiare dati con la rete si appoggia alle primitive offerte dal sistema operativo. Nel caso di un sistema Unix queste system call prendono il nome di Berkeley (o BSD) socket.

Dal punto di vista dell'applicazione un socket è un file descriptor sul quale invocare le system call per la lettura e scrittura di un file, e questo ha innumerevoli vantaggi: dopo aver deciso il tipo di comunicazione che si vuole utilizzare (come ad esempio una comunicazione affidabile utilizzando il protocollo Transmission Control Protocol, o TCP [9]) l'applicazione può ignorare i dettagli dei protocolli di rete sottostanti, e inviare e ricevere dati semplicemente richiamando le system call *write* e *read* sul file descriptor che rappresenta la connessione.

Tuttavia, nel caso specifico del protocollo HTTP, l'astrazione di flusso di byte affidabile offerta dal sistema operativo ha un costo in termini di prestazioni: il server web deve infatti adattare tutte le entità con cui lavora al concetto generico di stream, siano esse header HTTP, contenuti dinamici o file su disco, in quanto tutto ciò che lo stack di rete si aspetta in input è un flusso di dati.

Invio di un file, implementazione naïve

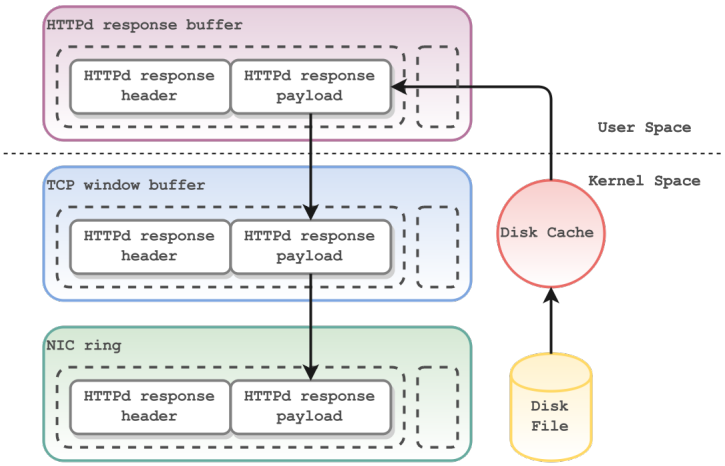

Figura 1.1: Invio di un file, implementazione naïve

La figura 1.1 mostra un'implementazione naïve di un server web e le operazioni di copia che questo deve effettuare per inviare un file.

Come si può notare il contenuto di un generico file viene copiato tre volte (quattro se si considerano eventuali meccanismi di caching del sistema operativo): una prima volta dal disco al buffer in spazio utente che il server web utilizza per creare la risposta, una seconda volta tra il buffer dell'applicazione e quello del socket TCP, poiché TCP mantiene una copia dello stream nel suo buffer utilizzato per gestire il meccanismo dello sliding window, ed una terza volta tra il buffer del socket TCP e quello della scheda di rete fisica.

Invio di un file, versione ottimizzata

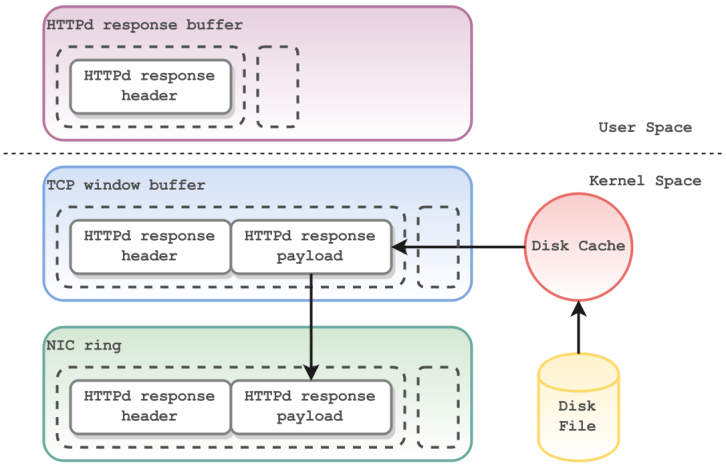

Figura 1.2: Invio di un file, versione ottimizzata

Alcuni sistemi operativi mettono a disposizione delle applicazioni una system call per trasferire dati tra due file descriptor senza dover copiare questi ultimi in spazio utente. Nel caso di sitemi Linux e BSD questa è implementata con il nome di *sendfile*. In questo modo un server web può inviare un file che si trova su disco in maniera più efficiente (figura 1.2), eliminando una copia in spazio utente non necessaria.

Con questa ottimizzazione è possibile incrementare effettivamente il throughput del server web, ma questa ottimizzazione è anche una delle poche attuabili in spazio utente: l'utilizzo della sendfile setta quindi un limite prestazionale in termini di banda oltre il quale non è possibile andare se non intervenendo sui livelli sottostanti.

1.3 Limiti dell'architettura attuale

L'impossibilità di ottimizzare ulteriormente l'architettura nasce da due problemi di fondo.

1.3.1 Uno stack di rete stratificato

Lo stack TCP/IP è progettato a livelli, con ogni strato dello stack che opera in maniera indipendente da quelli sottostanti. Se questo è un vantaggio dal punto di vista implmentativo, in quanto permette di tenere separate chiaramente le funzionalità e rende l'architettura modulare (rendendo possibile cambiare la funzionalità di un certo strato semplicemente garantendo che questo comunichi con lo strato inferiore e superiore aderendo ad un protocollo), ciò ha un impatto negativo sulle performance.

Il passaggio da uno strato all'altro introduce inevitabilmente degli overhead nel tempo di processing dei pacchetti e la scelta di design di tenere le componenti logicamente separate preclude la possibilità di applicare alcune ottimizzazioni, poiché la comunicazione tra gli strati è ridotta. Idealmente ogni strato dovrebbe infatti trattare i dati ricevuti dallo strato superiore come semplice payload opaco del proprio protocollo.

1.3.2 TCP e il concetto di stream

TCP è un protocollo di rete di livello 4 (livello di trasporto) che garantisce una trasmissione affidabile e ordinata dei dati. Utilizzando questo protocollo il sistema operativo può offrire alle applicazioni l'astrazione di flusso affidabile di byte per scambiare dati in rete. Ciò permettette di semplificare l'architettura e lo sviluppo degli applicativi che devono offrire servizi di rete, ma preclude al tempo stesso la possibilità di introdurre qualsiasi ottimizzazione.

L'algoritmo Sliding Window

Per garantire affidabilità e ordinamento dei byte dello stream il protocollo utilizza un algoritmo denominato della finestra scorrevole: la finestra del mittente è un buffer circolare dove l'applicazione scrive i dati che devono essere inviati. TCP considera un segmento come consegnato solo quando riceve dall'altro estremo della comunicazione un pacchetto di conferma dell'avvenuta consegna, ed utilizzando questo buffer è possibile effettuare, se necessario, una eventuale ritrasmissione di un segmento perso. I dati vengono infatti rimossi da questo buffer solo quando viene ricevuta la relativa conferma di

ricezione. In caso contrario, ovvero se non viene ricevuta una conferma entro una certa finestra temporale, il segmento viene ritrasmesso.

Specializzare TCP

Nell'ipotesi che i dati da inviare siano immutabili e persistenti, non ci sarebbe nessuna necessità per TCP di effettuare una copia di questi ultimi nei suoi buffer.

Per inviare un file, sarebbe sufficiente leggerlo da disco e scrivere il suo contenuto direttamente nei ring della scheda di rete, calcolando l'offset corretto rispetto alla dimensione degli header TCP/IP del pacchetto (figura 1.3). Nel caso di segmenti persi durante la trasmissione lo stack di rete dovrebbe solamente identificare l'offset del file corrispondente al pacchetto perso, rileggere i dati da disco (o più probabilmente dalla cache del sistema operativo) e rispedire il pacchetto.

Lo stesso concetto si può applicare ad un buffer in spazio utente, sia esso una copia di un file in RAM, una copia cache di una richiesta o semplicemente l'header di una risposta HTTP. Le uniche ipotesi necessarie sono garantire che queste entità siano immutabili e persistenti.

In questo modo un'applicazione che utilizza un socket TCP non dovrebbe più lavorare con il concetto generico di stream, ma potrebbe ragionare in termini di buffer e file descriptor, poiché lo stack di rete sarebbe in grado di trattare queste entità.

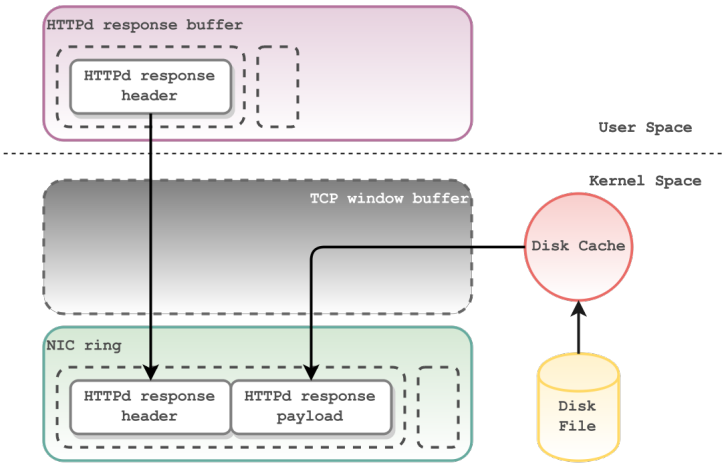

Figura 1.3: Invio di un file, versione zerocopy

1.4 ExoTCP e Eth

Da queste considerazioni nasce l'idea di sviluppare ExoTCP ed Eth. Il primo è uno stack di rete specializzato per il protocollo HTTP mentre il secondo è un server web che utilizza lo stack.

L'obiettivo è creare uno strumento con cui effettuare benchmark per valutare quanto effettivamente le astrazioni che il sistema operativo fornisce nell'ambito dei socket di rete comportino un limite alle prestazioni, o, dualmente, quanto sia possibile migliorarle passando dall'utilizzo di uno stack generico ad uno che sia profondamente integrato con il protocollo HTTP e con l'applicativo del server web.

Va notato che esistono già alcuni progetti basati su concetti simili, come Sandstorm [6] (un server Web con uno stack di rete specializzato), o mTCP [5] (uno stack di rete userspace progettato per scalare linearmente con il numero di core di CPU utilizzati), tuttavia del primo non viene fornito il codice sorgente (e differisce inoltre dal design di Eth per alcuni aspetti), mentre il secondo non è stato pensato appositamente per il protocollo HTTP.

Capitolo 2

Eth

Eth è l'applicativo che implementa il server web. I tre componenti principali di cui è composto sono Netmap, ExoTCP, ed il modulo che implementa la logica del server Web. La figura 2.1 mostra lo schema dell'architettura, con le strutture dati principali utilizzate.

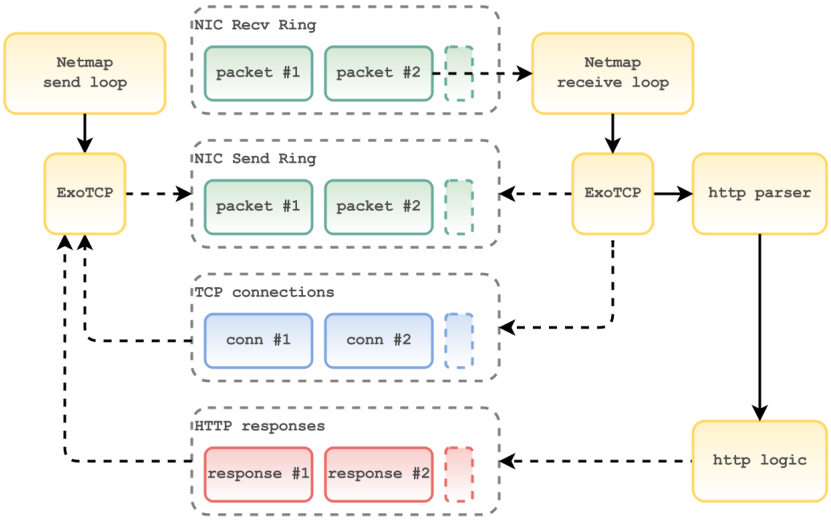

Figura 2.1: Architettura di Eth

Di seguito verrano descritti nel dettaglio i 3 componenti.

2.1 Netmap

Netmap [10] è un framework utilizzato per processare velocemente pacchetti di rete. Implementato come modulo kernel per i sistemi FreeBSD

e Linux, permette di mappare i ring della scheda di rete in spazio utente. Un'applicazione che utilizza questo framework può quindi creare e scrivere i pacchetti direttamente nei ring dell'interfaccia di rete, bypassando il sistema operativo. Netmap inoltre permette di fare questo riducendo il numero delle syscall da utilizzare: è possibile infatti scrivere e leggere un ring completo (che a titolo esemplificativo su una scheda Ethernet con chipset Intel e1000 ha una dimensione di 256 pacchetti) richiamando una sola system call.

2.1.1 Architettura di Netmap

Netmap permette alle applicazioni in spazio utente di accedere velocemente ai pacchetti di rete, utilizzando il sistema operativo per mediare l'accesso all'interfaccia di rete nel caso di operazioni potenzialmente pericolose.
Permette inoltre il raggiungimento di prestazioni elevate utilizzando diverse tecniche: una rappresentazione leggera dei metadati dei pacchetti, la possibilità di inviare e ricevere centinaia di pacchetti con una sola system call e la preallocazione dei buffer dei pacchetti.

Strutture dati

Per ogni interfaccia utilizzata Netmap esporta 3 tipi di strutture dati:

- *packet buffer*: rappresenta un buffer condiviso tra la scheda di rete e le applicazioni in spazio utente.
  Ogni packet buffer può contenere un frame, ha una dimensione di 2Kb ed è preallocato quando l'interfaccia viene inizializzata in modalità Netmap.

- *Netmap ring*: un descrittore (indipendente dal device fisico utilizzato) che rappresenta i ring della scheda di rete. I campi più importanti della struttura sono gli indici *head*, *cur* e *tail*, utilizzati per delimitare la regione dei pacchetti che possono essere utilizzati dall'applicazione e quella dei pacchetti riservati al kernel, ed il vettore degli slot dei pacchetti, dove ogni slot contiene la lunghezza e il puntatore (nello specifico un indice) al buffer del pacchetto

- *Netmap if*: un descrittore dell'interfaccia utilizzata; contiene informazioni come la dimensione dei ring fisici

Tutte le strutture descritte appartengono ad una regione di memoria che è condivisa tra lo spazio utente e lo spazio kernel, tuttavia non possono verificarsi race condition, in quanto in ogni momento ci può essere un solo flusso

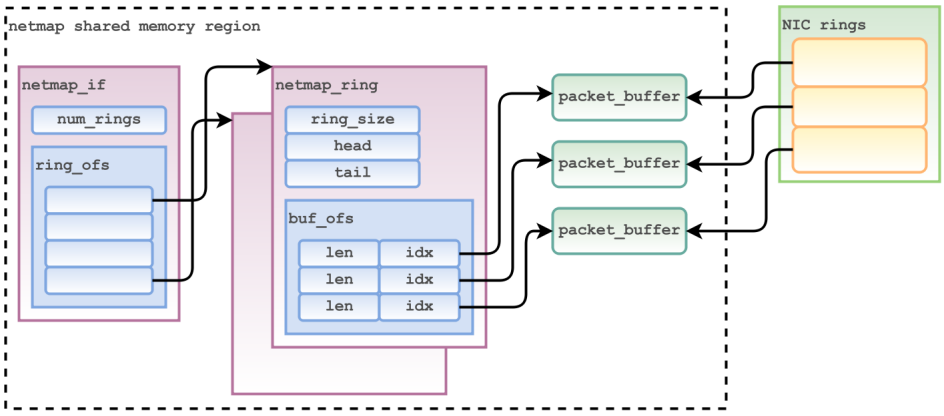

Figura 2.2: Strutture dati di Netmap

di esecuzione che accede dell'area di memoria. Le strutture che descrivono i ring appartengono infatti all'applicazione in spazio utente, tranne quando questa sta eseguendo una system call. I buffer dei pacchetti sono invece divisi in due partizioni: i pacchetti tra gli inidici *head* e *tail*-1 appartengono all'applicazione, mentre i rimanenti appartengono alla scheda di rete. I confini tra le due regioni vengono aggiornati solo all'interno del contesto di una system call.

API

Per utilizzare un'interfaccia in modalità Netmap è necessario richiamare la syscall *ioctl* sul device */dev/netmap* specificando il nome dell'interfaccia. In caso di successo viene ritornata la posizione e la dimensione della regione di memoria condivisa (contenente le strutture dati precedentemente menzionate), ed è sufficiente una chiamata alla system call *mmap* per rendere disponibile tale regione all'applicazione.

È inoltre possibile utilizzare la funzione ausiliaria *nm_open* per inizializzare un device. Questa permette di ignorare i dettagli dell'apertura del descrittore e della mappatura della memoria, e permette di specificare configurazioni avanzate dell'interfaccia.

Ricezione dei pacchetti

È possibile ricevere un pacchetto utilizzando il codice seguente.

```
struct nm_desc *netmap;
struct pollfd nm_fds;
struct netmap_ring *recv_ring;
```

```
unsigned int i, idx, len;
char *buf;

netmap = nm_open("netmap:iface", NULL, 0, 0);

nm_fds.fd     = NETMAP_FD(netmap);
nm_fds.events = POLLIN;

poll(&nm_fds, 1, -1);

recv_ring = NETMAP_RXRING(netmap->nifp, 0);

i   = recv_ring->cur;
idx = recv_ring->slot[i].buf_idx;

buf = NETMAP_BUF(recv_ring, idx);
len = recv_ring->slot[i].len;

/* process_packet(buf, len); */

recv_ring->cur = recv_ring->head = nm_ring_next(recv_ring, i);
```

Listing 2.1: Ricezione di un pacchetto con Netmap

Dopo aver inizializzato l'interfaccia con *nm_open*, viene richiamata la system call *poll*, in questo caso con un comportamento bloccante. Quando questa ritorna un valore, viene utilizzata la macro *NETMAP_RXRING* per accedere al ring per la ricezione dei pacchetti, viene letto l'indice corrente e l'offset del buffer rispetto a quest'ultimo, e con la macro *NETMAP_BUF* viene acceduto il buffer del pacchetto. Dopo aver processato quest'ultimo è possibile avanzare gli indici *head* e *cur*, per segnalare a Netmap (quando verrà richiamata nuovamente la system call *poll*) che il pacchetto è stato letto.

È possibile ottenere un comportamento non bloccante usando un timeout di 0 secondi oppure usando la system call *ioctl* con il parametro *NIOCRX-SYNC* sul file descriptor di Netmap.

Invio dei pacchetti

È possibile inviare un pacchetto utilizzando il codice seguente.

```
struct nm_desc *netmap;
struct pollfd nm_fds;
struct netmap_ring *send_ring;

unsigned int i, idx, len;
char *buf;

netmap = nm_open("netmap:iface", NULL, 0, 0);

nm_fds.fd     = NETMAP_FD(netmap);
```

```
nm_fds.events = POLLOUT;

poll(&nm_fds, 1, -1);

send_ring = NETMAP_TXRING(netmap->nifp, 0);

i   = send_ring->cur;
idx = send_ring->slot[i].buf_idx;

memcpy(NETMAP_BUF(send_ring, idx), packet_buf);
send_ring->slot[i].len = packet_len;

send_ring->cur = send_ring->head = nm_ring_next(send_ring, i);

ioctl(NETMAP_FD(netmap), NIOCTXSYNC);
```

Listing 2.2: Invio di un pacchetto con Netmap

Il codice per inviare un pacchetto è simile a quello per riceverlo. In questo esempio è stata usata la system call *poll* con un comportamento bloccante per richiedere la disponibilità del ring di trasmissione (ovvero per inviare gli eventuali pacchetti già presenti nel ring) e la system call *ioctl* (non blocante) con il parametro *NIOCTXSYNC* per spedire il pacchetto.
Utilizzando la system call *poll* con gli eventi *POLLIN* e *POLLOUT* è inoltre possibile sincronizzare con una sola chiamata a funzione il ring di ricezione e quello di trasmissione.

2.1.2 Inizializzazione

Prima di utilizzare Netmap è necessario caricare due moduli kernel. Il primo è *netmap.ko*, ovvero il modulo che implementa le funzionalità di Netmap e mette a disposizione il device */dev/netmap*. Il secondo è il modulo della scheda di rete ricompilato con le patch di Netmap.

Netmap supporta infatti un numero limitato di schede di rete, e per utilizzarle è necessario caricare i relativi moduli driver ai quali sono state applicate delle apposite patch. Nel caso in cui si tenti di utilizzare una scheda di rete non supportata (o non venga caricato il modulo ricompilato) Netmap funzionerà in una modalità di emulazione.
Dopo il caricamento dei moduli l'applicazione inizializza l'interfaccia di rete con una chiamata alla funzione *nm_ open* specificando semplicemente il nome dell'interfaccia da usare.

2.1.3 Processing dei pacchetti

Il server Eth utilizza tre loop per processare i pacchetti: *nm_ recv_ loop*, *nm_ retx_ loop* e *nm_ send_ loop*. Questi loop vengono eseguiti a loro volta

all'interno di un loop infinito più esterno (listato 2.3).

```
void
nm_loop(void)
{
  while (1) {
    nm_sync_rx_tx_ring();

    nm_recv_loop();
    sort_all_unackd_segments();

    nm_retx_loop();
    nm_send_loop();
    sort_all_unackd_segments();
  }
}
```

Listing 2.3: Loop principale del modulo Netmap

Sincronizzazione dei ring

Prima di richiamare il loop per la ricezione dei pacchetti viene effettuata la sincronizzazione dei ring, ovvero vengono ricevuti ed inviati i pacchetti presenti in questi ultimi (aggiornando gli indici delle strutture dati di Netmap che descrivono i ring). Il codice è mostrato nel listato 2.4.

```
static inline
void
nm_sync_rx_tx_ring(void)
{
  int timeout;
  struct pollfd nm_fds;

  nm_fds.fd     = NETMAP_FD(netmap);
  nm_fds.events = POLLIN;

  if (nm_has_data_to_send) {
    timeout = 0;
  } else if (! list_empty(&per_conn_min_retx_ts)) {
    tcp_per_conn_min_retx_ts_t *min_retx_ts = list_first_entry(&
    per_conn_min_retx_ts, tcp_per_conn_min_retx_ts_t, head);
    timeout = min_retx_ts->retx_ts - cur_ms_ts();
  } else {
    timeout = -1;
  }

  poll(&nm_fds, 1, timeout);
}
```

Listing 2.4: Sincronizzazione dei ring di Netmap

Se non ci sono pacchetti che devono essere trasmessi la sincronizzazione dei ring assume un comportamento bloccante: in questo modo non viene consumata CPU e la *poll* viene risvegliata non appena sono disponibili nuovi

pacchetti nel ring di ricezione.
Se invece sono presenti segmenti da inviare, o segmenti che dovrebbero essere ritrasmetti dopo un certo timeout in caso non vengano ricevuti ACK, la poll viene chiamata rispettivamente con un timeout di 0 secondi (non bloccante) o con un timeout pari al minimo timeout oltre il quale (in assenza di ACK) sarà necessario ritrasmettere un segmento.

Netmap receive loop

Questo loop ha lo scopo di ricevere dalla rete i pacchetti, processarli ed eventualmente inviare una risposta se il pacchetto ricevuto la richiede. Il codice è mostrato nel listato 2.5.

```
static inline
void
process_packet(char *buf, size_t len)
{
  packet_t p;
  socket_t s;

  p.buf = buf;
  p.len = len;

  set_cur_pkt(&p);
  set_cur_sock(&s);

  process_eth();
}

static inline
void
recv_packet(struct netmap_ring *recv_ring)
{
  unsigned int i, idx, len;
  char *buf;

  i   = recv_ring->cur;
  idx = recv_ring->slot[i].buf_idx;

  buf = NETMAP_BUF(recv_ring, idx);
  len = recv_ring->slot[i].len;

  process_packet(buf, len);

  recv_ring->head = recv_ring->cur = nm_ring_next(recv_ring, i);

}

static inline
void
nm_recv_loop(void)
{
  struct netmap_ring *recv_ring;

  recv_ring = NETMAP_RXRING(netmap->nifp, 0);
```

```
  while (!nm_ring_empty(recv_ring)) {
    recv_packet(recv_ring);
  }

  ioctl(NETMAP_FD(netmap), NIOCTXSYNC);
}
```

Listing 2.5: Loop di ricezione di Netmap

Per ogni pacchetto ricevuto viene richiamata una funzione dello stack di rete che ha il compito di processarlo. Lo stack può processare richieste Address Resolution Protocol (ARP) o Internet Control Message Protocol (ICMP), richieste di instaurare una nuova connessione TCP o segmenti TCP contenenti dati o acknowledgement. È in questo loop che vengono inviate le risposte a questi pacchetti: risposte ARP o ICMP, segmenti TCP per completare l'istaurazione di una nuova connessione o per segnalare con un acknowledgement la ricezione di un segmento contenente dati.

L'unico tipo di pacchetto che non viene inviato in questo loop è quello contente un segmento TCP all'interno del quale sono presenti dei dati. Per inviare dati viene infatti utilizzato un loop separato.

Al termine del loop viene effettuata la sincronizzazione del ring di trasmissione. Questo potrebbe infatti essere pieno poiché ogni pacchetti ricevuto potrebbe potenzialmente comportare la scrittura di un pacchetto di risposta nel ring di trasmissione della scheda di rete.

Netmap retransmit loop

Questo loop ha lo scopo di ritrasmettere eventuali segmenti TCP per i quali non è stato ricevuto un ACK dopo un certo timeout (pari a 4 volte il round trip time).

```
static inline
void
nm_retx_loop(void)
{
  tcp_per_conn_min_retx_ts_t *min_retx_ts, *tmp;
  bool did_retx = false;

  list_for_each_entry_safe(min_retx_ts, tmp, &per_conn_min_retx_ts, head) {
    tcp_unackd_segment_t *seg, *tmp2;

    if (min_retx_ts->retx_ts >= cur_ms_ts()) {
      break;
    }

    set_cur_conn(min_retx_ts->conn);
    set_cur_sock(cur_conn->sock)

    list_for_each_entry_safe(seg, tmp2, &cur_conn->unackd_segs, head) {
      if (seg->retx_ts >= cur_ms_ts()) {
```

```
        break;
      }

      did_retx = true;
      tcp_retransm_segment(seg);
    }
  }

  if (did_retx) {
    ioctl(NETMAP_FD(netmap), NIOCTXSYNC);
  }
}
```

Listing 2.6: Loop di ritrasmissione del modulo Netmap

La gestione della ritrasmissione è organizzata in due livelli: per ogni connessione viene mantenuta una lista di descrittori di segmenti che ancora non hanno ricevuto un ACK: un descrittore è composto dal numero di sequenza del pacchetto e dal timestamp oltre il quale il pacchetto deve considerarsi come perso (timestamp di ritrasmissione). Questa lista viene mantenuta ordinata per timestamp crescenti.

Viene inoltre mantenuta una seconda lista globale con un elemento per ogni connessione che ha pacchetti in transito non ancora confermati da un ACK: ogni elemento di quest'ultima lista contiene un puntatore alla connessione ed il minimo tra tutti i timestamp di ritrasmissione di quella connessione. Anche questa lista viene mantenuta ordinata per timestamp crescenti.

In questo modo è possibile determinare facilmente se ci sono dei pacchetti che devono essere ritrasmessi.
Il controllo inizia iterando la lista globale: se il timestamp di ritrasmissione dell'elemento preso in considerazione è minore del timestamp corrente la connessione ha uno o più pacchetti che necessitano di essere ritrasmetti. In questo caso si procede quindi a scandire la lista specifica di quella connessione. La lista viene scandita fino a che il timestamp degli elementi è minore del timestamp corrente e per ogni elemento viene invocata la funzione dello stack che ha il compito di ritrasmettere il segmento.
Dopo che una connessione è stata esaminata si può riprendere con la scansione della lista globale, continuando con la connessione successiva.

Netmap send loop

Questo loop ha lo scopo di inviare i segmenti TCP contenenti dati, ovvero le risposte HTTP, composte da header e body del messaggio.

```
static inline
void
```

```
nm_send_loop(void)
{
  struct netmap_ring *send_ring;
  static bool resume_loop = false;

  send_ring = NETMAP_TXRING(netmap->nifp, 0);

  nm_has_data_to_send = true;
  while (!nm_ring_empty(send_ring) && nm_has_data_to_send) {

    if (unlikely(!resume_loop)) {
      nm_send_loop_cur_conn = list_first_entry(nm_tcp_conn_list, tcp_conn_t,
    nm_tcp_conn_list_head);
    }

    resume_loop         = false;
    nm_has_data_to_send = false;

    tcp_conn_t *n;
    list_for_each_entry_safe_from(nm_send_loop_cur_conn, n, nm_tcp_conn_list,
    nm_tcp_conn_list_head) {
      set_cur_conn(nm_send_loop_cur_conn);

      if (unlikely(nm_ring_empty(send_ring))) {
        resume_loop = true;
        break;
      }

      if (tcp_conn_has_data_to_send()) {
        set_cur_sock(cur_conn->sock);

        tcp_conn_send_data();
        nm_has_data_to_send = true;
      }
    }
  }
}
```

Listing 2.7: Loop di invio del modulo Netmap

All'interno del loop viene iterata la lista delle connessioni TCP, e per ognuna di queste viene valutato se sono presenti dati da spedire (e se la finestra effettiva lo permette). In caso affermativo viene inviato un certo numero di segmenti per ogni connessione. Quando il ring della scheda di rete è pieno il loop viene interrotto, e alla prossima esecuzione l'iterazione verrà ripresa dalla connessione su cui si stava iterando all'ultima chiamata.

Ne risulta che i dati vengono inviati con una politica di tipo round robin che garantisce fairness: ad ogni connessione viene concesso di inviare un numero massimo di pacchetti, e dopo che i dati sono stati inviati questa deve attendere che tutte le altre connessioni abbiano potuto tentare di inviare il loro massimo numero di pacchetti concessi.

2.1.4 Funzioni ausiliarie

Sono state implementate alcune funzioni per nascondere il funzionamento interno dei ring e semplificare la fase di invio di frame.
Queste permettono di richiedere un descrittore del primo packet buffer disponibile nel ring di trasmissione della scheda di rete.
Un descrittore è composto da un puntatore ad un buffer sul quale andrà copiato il pacchetto e da un puntatore ad intero, sul quale andrà scritta la dimensione del pacchetto.
La funzione *nm_get_tx_buf* accetta un puntatore ad una struttura di tipo *nm_get_tx_buf* e la inizializza con l'indirizzo del buffer del pacchetto su cui copiare i dati e con un puntatore alla variabile su cui dovrà essere scritta la dimensione del pacchetto. Dopo aver inizializzato il descrittore la funzione aggiorna gli indici delle strutture di Netmap che descrivono i ring.

Vengono inoltre messe a disposizione le funzioni *nm_send_packet* e *nm_send_packet_with_data* che permettono di inviare rispettivamente un pacchetto ed un pacchetto con il relativo payload (i due casi vengono distinti in quanto il buffer del pacchetto e quello del payload sono allocati separatamente).

2.2 Modulo HTTP

Il modulo che implementa la logica HTTP può essere suddiviso in due componenti: il parser che riceve in input una stringa e genera la rappresentazione di una richiesta, e l'applicazione che elabora la richiesta e genera una risposta.

```
void
handle_http_request(void)
{
  http_response_t *response;

  if (!cur_conn->http_response) {
    response               = malloc(sizeof(http_response_t));
    response->parser       = new_eth_parser();
    cur_conn->http_response = response;
  } else {
    response = cur_conn->http_response;
  }

  eth_parser_execute(response->parser, (const char *) cur_conn->data_buffer,
    cur_conn->data_len + 1, 0);

  if (eth_parser_finish(response->parser) == 1) {
    eth_http_response(response);
  }
```

```
  response->parser->parsed = true;
  response->sent           = false;
}
```

Listing 2.8: Gestione di una richiesta HTTP

Il listato 2.8 mostra la funzione *handle_http_request*, richiamata dallo stack di rete quando questo riceve un segmento TCP contenente dei dati.
Quando viene ricevuta una nuova richiesta HTTP viene allocata una nuova struttura di tipo *http_response_t* per la connessione corrente e viene inizializzato un nuovo parser HTTP. Viene quindi eseguito il parser sulla stringa della richiesta HTTP e se il parsing si è concluso con successo viene effettivamente creata la rappresentazione della risposta invocando la funzione *eth_http_response*.
Il parsing può non concludersi per 2 motivi: la stringa contine una richiesta malformata e quindi la richiesta viene scartata, oppure la stringa contiene solo una parte della richiesta HTTP. In tal caso viene mantenuta in un buffer e il parser viene richiamato in seguito alla ricezione di nuovi segmenti.

2.2.1 Parser HTTP

Il parser HTTP è una versione leggermente modificata del parser utilizzato dal server web Puma [8]. È composto da una grammatica Ragel [12] e da una serie di callback che vengono richiamate durante il parsing.

Ragel è uno compilatore di macchine a stati finiti: in input riceve un file che descrive la grammatica, ed in output produce il codice C del parser. I file di input descrivono le grammatiche utilizzando una sintassi basata su espressioni regolari; in questi file è inoltre possibile incorporare direttamente codice C, che verrà richiamato quando il parser esamina un determinato token.

Ragel permette inoltre di effettuare il parsing incrementale delle richieste, esportando una variabile che rappresenta il suo stato interno.
In questo modo è possibile riprendere il parsing nel caso il parser fosse stato invocato passandogli una stringa incompleta (ovvero nel caso non fossero stati ricevuti tutti i segmenti TCP che contengono la richiesta HTTP).

Al termine del parsing della richiesta, viene inizializzata una struttura di tipo *eth_parser_t* che permette di accedere al metodo, all'URI e agli altri campi della richiesta HTTP.

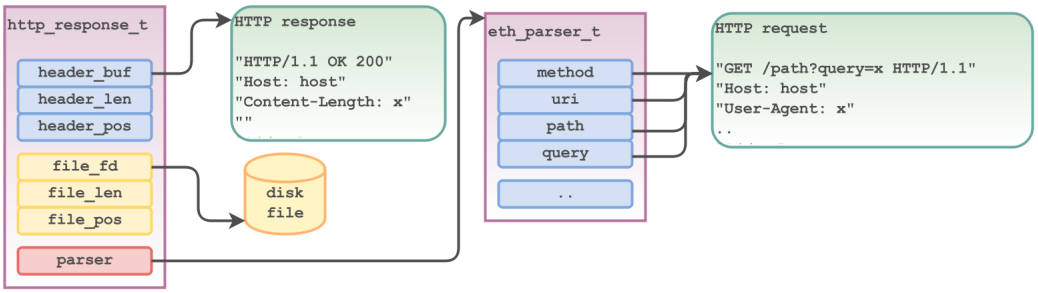

Figura 2.3: Strutture dati HTTP

2.2.2 Applicazione HTTP

Dopo il parsing della richiesta, il compito di produrre la risposta HTTP è affidato al modulo che implementa la logica del server Web.

```
void
eth_http_response(http_response_t *response)
{
  char *path;
  char *wd = getcwd(NULL, 0);
  struct stat stat;

  asprintf(&path, "%s/htdocs/%s", wd, response->parser->path);

  response->file_fd = open(path, O_RDONLY);

  if (response->file_fd == -1) {
    build_404(response);
    return;
  }

  fstat(response->file_fd, &stat);

  asprintf(&response->header_buf,
      "HTTP/1.1 200 OK\r\n"
      "Host: internet\r\n"
      "Content-Length: %d\r\n"
      "\r\n", (int) stat.st_size);

  response->header_len = strlen(response->header_buf);
  response->header_pos = 0;

  response->file_len   = stat.st_size;
  response->file_pos   = 0;
}
```

Listing 2.9: Ricezione di un pacchetto con Netmap

Il suo compito è controllare che il file esista, e nel caso creare una rappresentazione della risposta HTTP. Questa è descritta dalla struttura *http_response_t* (Figura 2.3).

Una risposta HTTP è quindi una struttura contenente la stringa dell'header ed il file descriptor del file da inviare, con le relative dimensioni. All'interno di questa struttura sono inoltre contenuti i campi che consentono al protocollo TCP di mantenere lo stato relativo al progresso della trasmissione della risposta: *header_pos* e *file_len*.

Questa organizzazione della risposta non preclude l'invio di semplici risposte testuali; nel caso è infatti sufficiente settare la stringa *header_buf* con l'header ed il body della risposta e settare la dimensione del file (*file_len*) a 0.
Eth utilizza questa modalità per inviare ad esempio la riposta con codice 404 nel caso il file non fosse presente.

Capitolo 3

ExoTCP - Architettura e protocolli Ethernet e IP

ExoTCP è uno stack di rete TCP/IP minimale, userspace e zerocopy, specializzato per il protocollo HTTP.
Minimale: essendo progettato per gestire unicamente traffico HTTP è stato possibile implementare un set ridotto e semplificato di funzionalità.
Userspace: questo permette allo stack di interagire con l'applicativo del server web senza avere un calo delle prestazioni dovuto ai context switch tra kernel e user space.
Zerocopy: utilizzando il framework Netmap è possibile leggere e scrivere frame ethernet direttamente dalla scheda di rete, eliminando la necessità di copiare i dati più volte inutilmente.

Contrariamente agli stack TCP/IP tradizionali, ExoTCP non applica una separazione netta degli strati su cui operano i vari protocolli. Questo permette di introdurre alcuni concetti importanti su cui si basa.

3.0.3 Pacchetti preinizializzati

Un pacchetto preinizializzato è una struttura preallocata che rappresenta una generica classe di pacchetti. Può essere quindi considerato un template, con alcuni campi (quelli comuni a tutti i pacchetti che appartengono alla classe del template) già inizializzati.
In questo modo per creare un pacchetto non è necessario costruirlo interamente; è sufficiente utilizzare il template della classe alla quale appartiene il pacchetto che si deve utilizzare, settando solamente i campi specifici diret-

tamente nel template e copiando questo nel buffer del pacchetto che si trova nel ring della scheda di rete.

In figura 3.1 è mostrato un template di pacchetto TCP utilizzato per rispondere alla richiesta di un client di aprire una nuova connessione, con evidenziati i campi preinizializzati.

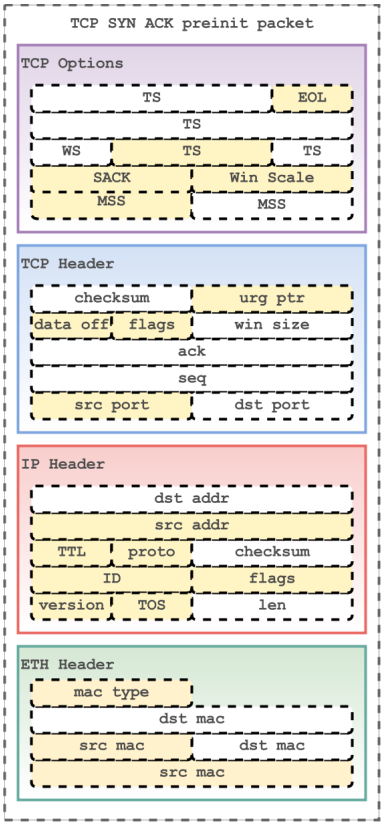

Figura 3.1: Template di un pacchetto TCP del tipo SYN ACK

3.0.4 Consapevolezza del protocollo HTTP

ExoTCP è stato sviluppato assumendo di lavorare esclusivamente con traffico HTTP, e questa assunzione forte permette di semplificare e specializzare lo stack di rete.

3.1 Architettura nel dettaglio

Di seguito viene presentata l'architettura che ExoTCP utilizza. Le funzioni dello stack possono essere divise in tre classi: quelle di inizializzazione, richiamate all'avvio del server, quelle per processare i pacchetti in input e quelle per creare i pacchetti che dovranno essere emessi in output.

3.1.1 Inizializzazione dello stack

La funzione *init_exotcp* ha il compito di inizializzare lo stack.
Dopo aver settato le variabili che contengono gli indirizzi MAC e IP della scheda di rete vengono richiamate le funzioni di inizializzazione dei moduli che implementano i diversi protocolli. Il compito di queste funzioni è inizializzare i pacchetti precostruiti.

Pacchetti preinizializzati utilizzati

Utilizzando i seguenti template generici è possibile descrivere tutte le classi di pacchetti necessarie per gestire il traffico di rete di Eth.
prebuild_arp_packet: utilizzato per rispondere alle richieste ARP. È una struttura composta da un header Ethernet e un header ARP.
prebuild_icmp_packet: utilizzato per rispondere alle richieste ICMP echo. È una struttura composta da un header ethernet e un header ICMP.
syn_ack_tcp_packet: utilizzato per rispondere alla richiesta di apertura di una connessione TCP. È una struttura composta da un header Ethernet, uno IP, ed uno TCP, con le opzioni TCP specifiche da utilizzare nella fase di handshake (ovvero le opzioni *maximum segment size*, *SACK permitted* e *window scale*).
ack_tcp_packet: utilizzato per confermare la ricezione di un segmento TCP.
data_tcp_packet: utilizzato per inviare un segmento TCP contenente dati.
fin_ack_tcp_packet: utilizzato per rispondere alla chiusura una connessione TCP.
rst_tcp_packet: utilizzato per resettare una connessione TCP.

3.1.2 Ricezione e invio di pacchetti

Il punto d'ingresso per processare i pacchetto è la funzione *process_eth*, richiamata per ogni pacchetto ricevuto nel loop di ricezione del modulo net-

map.
Per quanto riguarda l'invio le uniche funzioni esportate agli altri moduli sono *tcp_conn_send_data* e *tcp_retransmit_segment*.
La prima viene utilizzata per iniziare o riprendere l'invio di un determinato numero di segmenti TCP contenenti la risposta HTTP, mentre la seconda viene utilizzata per ritrasmettere un segmento che è presumibilmente andato perso.

3.1.3 Contesto dello stack

ExoTCP utilizza inoltre alcune variabili globali per mantenere il contesto all'interno del quale operare.
cur_packet: un puntatore all'ultimo pacchetto ricevuto, ovvero il pacchetto che lo stack di rete dovrà processare.
cur_socket: un puntatore ad una struttura che contiene alcune informazioni sullo pseudo socket corrente (l'indirizzo mac e ip sorgente e la porta TCP di origine).
cur_conn: un puntatore ad una struttura che descrive la connessione TCP corrente.

Poiché ExoTCP è uno stack di rete single thread, l'utilizzo di queste variabili globali per descrivere il contesto non pone il rischio di race condition: in ogni momento ci potrà essere solo un pacchetto corrente che dovrà essere processato, così come una sola connessione TCP corrente.

3.1.4 Architettura dei moduli

Ogni protocollo è implementato in un file C separato, e tutti i moduli hanno una struttura simile. In particolare ogni modulo è composto da:

- una funzione di inizializzazione, richiamata dalla funzione di inizializzazione globale *init_exotcp*. Questa ha il compito di richiamare le funzioni di inizializzazione specifiche dei diversi pacchetti preinizializzati
- una funzione che ha il compito di processare un pacchetto ricevuto dalla scheda di rete (limitatamente al proprio protocollo)
- eventuali funzioni per il calcolo del checksum dei pacchetti

Inoltre, per ogni pacchetto preinizializzato ogni modulo implementa

- una struttura che descrive il pacchetto

- una funzione per inizializzare i campi che rimangono costanti
- una funzione per effettuare la specializzazione del pacchetto prima dell'invio
- una funzione per l'invio del pacchetto

Per ogni pacchetto ricevuto dalla scheda di rete viene invocata la funzione *process_packet*, che ha il compito di inizializzare il contesto settando le variabili globali *cur_pkt* e *cur_socket* e di richiamare il punto d'ingresso dello stack di rete: *process_eth*.

3.1.5 Protocolli di I e II livello

Ethernet

Il modulo che gestisce il protocollo Ethernet ha solamente il compito di controllare che il frame sia diretto all'indirizzo MAC della scheda di rete. In caso affermativo in base al protocollo specificato nell'header viene richiamata la funzione per gestire un pacchetto ARP o un pacchetto IP.
Il modulo mette a disposizione la funzione *init_eth_packet*, utilizzata per inizializzare l'header Ethernet dei template dei pacchetti, e la funzione *setup_eth_hdr*, utilizzata per specializzare l'header ethernet di un pacchetto che deve essere inviato.

Address Resolution Protocol

Il modulo per la gestione del protocollo ARP supporta solamente un ristretto sottoinsieme del protocollo stesso: è in grado di riconoscere un pacchetto di tipo *ARP request* e rispondere con un pacchetto di tipo *ARP reply*.

Internet Protocol

Il modulo che gestisce il protocollo IP ne implementa solamente una minima parte: durante la ricezione viene effettuato il controllo che il pacchetto contenga un header di tipo IPv4 e che l'indirizzo destinazione sia quello del server web. In caso affermativo viene richiamata la funzione per processare il payload, che può essere un pacchetto ICMP o un segmento TCP. La frammentazione non viene al momento gestita.

Il modulo mette a disposizione la funzione *init_ip_packet*, utilizzata per inizializzare l'header dei template dei pacchetti, e la funzione *setup_ip_hdr*, il cui compito è specializzare l'header di un pacchetto che deve essere inviato.

ICMP

Il modulo gestisce solamente la gestione dei pacchetti di tipo ICMP echo request, rispondendo con un pacchetto di tipo ICMP echo reply.

Capitolo 4

ExoTCP - protocollo TCP

Il modulo TCP è quello che implementa il maggior numero di funzionalità. In questo capitolo verranno descritte le strutture dati utilizzate per organizzare le connessioni, le funzioni esportate agli altri moduli e quelle utilizzate internamente dal modulo stesso.

4.1 Descrittore di connessione

Ogni connessione TCP è descritta da una struttura di tipo *tcp_conn_t*, mostrata nel listato 4.1.

```
typedef struct tcp_conn_s {
  tcp_conn_key_t *key;
  list_head_t nm_tcp_conn_list_head;
  socket_t    *sock;

  uint32_t last_recv_byte;
  uint32_t last_sent_byte;
  uint32_t last_ackd_byte;

  tcp_state_t state;

  uint32_t recv_eff_window;

  list_head_t               unackd_segs;
  tcp_per_conn_min_retx_ts_t min_retx_ts;

  uint32_t last_retx_seg_seq;
  uint32_t last_retx_seg_ts;

  struct {
    uint16_t mss;
    uint8_t  win_scale;
    uint8_t  sack_perm;
    uint32_t ts;
```

```
    uint32_t echo_ts;
  } client_opts;

  uint32_t rtt;

  uint8_t data_buffer[TCP_WINDOW_SIZE];
  size_t  data_len;

  http_response_t *http_response;
  uint32_t         http_response_start_seq;
} tcp_conn_t;
```

Listing 4.1: struttura tcp_conn_t

Tutte le connessioni TCP attive sono organizzate utilizzando due strutture dati: la prima è una tabella hash, la cui chiave è una struttura di tipo *tcp_conn_key_t* composta dai campi porta e indirizzo IP di origine, mentre la seconda è una lista double linked.
Con questa organizzazione è possibile eseguire tutte le operazioni sulle connessioni in modo efficiente: il modulo Netmap può scandire in tempo lineare tutte le connessioni accedendo alla lista mentre è allo stesso tempo possibile conoscere in tempo costante (nel caso medio) se una determinata coppia <indirizzo sorgente, porta sorgente> rappresenta una connesione conosciuta o meno accedendo alla tabella hash.

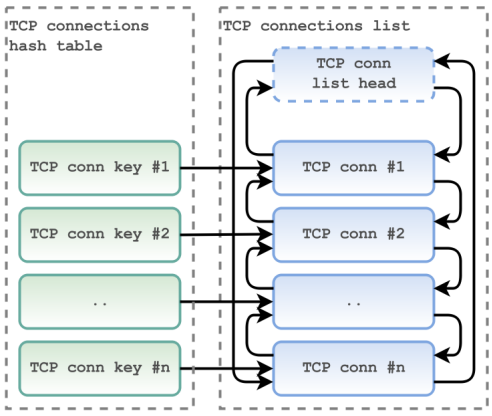

Figura 4.1: Organizzazione delle connessioni TCP

4.2 Processing dei pacchetti TCP

Il punto d'ingresso per processare un pacchetto è la funzione *process_tcp*. Una versione semplificata della funzione è mostrata nel listato 4.2.

```
void
process_tcp_with_conn() {
  switch (cur_conn->state) {
    case ESTABLISHED:
      if (flag_fin(cur_pkt->tcp_hdr)) {
        process_tcp_fin();
      } else {
        process_tcp_segment();
      }
      break;
    case SYN_SENT:
      process_3wh_ack();
      break;
    case FIN_SENT:
      process_closed_ack();
      break;
  }
}

void
process_tcp_without_conn(void) {
  if (flag_syn(cur_pkt->tcp_hdr)) {
    process_tcp_new_conn();
  } else {
    send_tcp_rst_without_conn();
  }
}

void
process_tcp(void)
{
  tcp_conn_t *conn = get_tcp_conn();

  if (conn) {
    process_tcp_with_conn();
  } else {
    process_tcp_without_conn();
  }
}
```

Listing 4.2: Punto d'ingresso per il processing di un pacchetto TCP

Questa funzione controlla come prima cosa se il pacchetto ricevuto appartiene ad una nuova connessione o meno.

ExoTCP utilizza la funzione *get_tcp_conn* per recuperare dalla tabella hash il descrittore di connessione associato all'indirizzo e alla porta sorgente del pacchetto corrente.

Nel caso in cui si tratti di una nuova connessione, se il pacchetto ha il flag SYN attivo viene allocata una nuova struttura *tcp_conn_t*, viene effettuato il parsing delle eventuali opzioni TCP ed inizia la procedura per portare a termine la seconda fase del TCP three way handshake.

4.2.1 Invio di un pacchetto SYN+ACK

L'invio della seconda fase del three way handshake avviene specializzando il template generico apposito.
Ad occuparsene è la funzione *send_tcp_syn_ack*, la quale richiama le funzioni messe a disposizione dai protocolli degli strati inferiori per effettuare il setup dei rispettivi header: *setup_eth_header* e *setup_ip_header*.
Successivamente viene invocata la funzione *setup_tcp_hdr*: questa ha il compito di settare i campi dell'header TCP comuni a tutti i template TCP (ed è quindi richiamata non solo per un pacchetto di tipo SYN ACK, ma per qualsiasi pacchetto TCP).
Infine vengono settati i campi specifici del template, ovvero le opzioni TCP, e viene calcolato il checksum.

Per copiare il pacchetto nel buffer del ring della scheda di rete viene invocata la funzione *nm_send_packet*.
Quest'ultima, esportata dal modulo netmap, ha il compito di recuperare il primo packet buffer libero del ring per la trasmissione e copiare il pacchetto (ricevuto come argomento) nel buffer.
Il pacchetto verrà effettivamente trasmesso (ovvero verrà effettuata la sincronizzazione del ring di trasmissione) solo quando il ring sarà pieno oppure al termine del loop di trasmissione del modulo netmap.
Nel listato 4.3 sono mostrate le funzioni utilizzate dai protocolli di secondo e terzo livello.

```
void
init_eth_packet(eth_hdr_t *eth_hdr, uint16_t eth_type)
{
  memcpy(eth_hdr->src_addr, &mac_addr, sizeof(struct ether_addr));
  eth_hdr->mac_type = eth_type;
}

void
setup_eth_hdr(eth_hdr_t *eth_hdr)
{
  memcpy(eth_hdr->dst_addr, cur_sock->src_mac, sizeof(struct ether_addr));
}


void
init_ip_packet(ip_hdr_t *ip_hdr, uint16_t data_len, uint8_t proto)
{
  ip_hdr->version     = 4;
  ip_hdr->hdr_len     = 5;
  ip_hdr->tos         = 0;
  ip_hdr->total_len   = HTONS(sizeof(ip_hdr_t) + data_len);
  ip_hdr->id          = 0;
  ip_hdr->frag_offset = HTONS(0x4000); /* dont fragment */
  ip_hdr->ttl         = 64;
  ip_hdr->proto       = proto;

  memcpy(&ip_hdr->src_addr, &ip_addr, sizeof(struct in_addr));
```

```
}

void
setup_ip_hdr(ip_hdr_t *ip_hdr, uint16_t new_data_len)
{
  ip_hdr->dst_addr = cur_sock->src_ip;

  if (new_data_len) {
        ip_hdr->total_len = HTONS(sizeof(ip_hdr_t) + new_data_len);
  }

  ip_checksum(ip_hdr);
}
```

Listing 4.3: Inizializzazione e setup di un pacchetto SYN ACK. Protocolli Ethernet e IP

Le funzioni *init_eth_packet* e *init_ip_packet* vengono richiamate solamente all'avvio dello stack di rete, ed inizializzano i campi costanti del pacchetto precostruito.
Le funzioni *setup_eth_hdr* e *setup_ip_hdr* vengono invece richiamate per ogni pacchetto che deve essere inviato. Il loro compito è effettuare la specializzazione dei rispettivi header del template (settando ad esempio l'indirizzo MAC ed IP destinazione).

Il listato 4.4 mostra le funzioni *init_tcp_packet_header* e *setup_tcp_hdr*.

```
static inline
void
init_tcp_packet_header(tcp_hdr_t *hdr, uint8_t opts_len, uint8_t flags)
{
  hdr->src_port    = HTONS(listening_port);
  hdr->res         = 0;
  hdr->window      = HTONS(TCP_WINDOW_SIZE);
  hdr->data_offset = (sizeof(tcp_hdr_t) + opts_len) / 4;
  hdr->flags       = flags;
}

void
setup_tcp_hdr(tcp_hdr_t *hdr)
{
  hdr->dst_port = cur_sock->src_port;
  hdr->ack      = htonl(cur_conn->last_recv_byte + 1);
  hdr->seq      = htonl(cur_conn->last_sent_byte + 1);
  hdr->window   = HTONS(TCP_WINDOW_SIZE - cur_conn->data_len);
}
```

Listing 4.4: Inizializzazione e setup di un pacchetto SYN ACK. Protocollo TCP generico

La prima, richiamata solamente all'avvio del server, inizializza l'header di un generico pacchetto TCP precostruito, mentre la seconda viene richiamata per il setup dell'header TCP di ogni pacchetto che deve essere inviato.
Queste sono quindi funzioni generiche, richiamate per ogni template TCP.

Infine ll listato 4.5 mostra le funzioni richiamate per inizializzare e specializzare il template di un pacchetto di tipo *SYN ACK* (ovvero quello per rispondere alla richiesta di apertura di una nuova connessione).

```
void
init_syn_ack_tcp_packet(void)
{
  init_eth_packet(&syn_ack_tcp_packet.eth, ETH_TYPE_IPV4);
  init_ip_packet(&syn_ack_tcp_packet.ip,
    sizeof(tcp_hdr_t) + sizeof(tcp_syn_ack_opts_t), IP_PROTO_TCP);
  init_tcp_packet_header(&syn_ack_tcp_packet.tcp,
    sizeof(tcp_syn_ack_opts_t), TCP_FLAG_SYN | TCP_FLAG_ACK);

  syn_ack_tcp_packet.opts = (tcp_syn_ack_opts_t) {
    .mss       = { .code = TCP_OPT_MSS_CODE,       .len = TCP_OPT_MSS_LEN },
    .sack_perm = { .code = TCP_OPT_SACK_PERM_CODE, .len = TCP_OPT_SACK_PERM_LEN},
    .win_scale = { .code = TCP_OPT_WIN_SCALE_CODE, .len = TCP_OPT_WIN_SCALE_LEN},
    .ts        = { .code = TCP_OPT_TS_CODE,        .len = TCP_OPT_TS_LEN},
    .eol       = TCP_OPT_EOL_CODE
  };
}

static inline
void
send_tcp_syn_ack(void)
{
  setup_eth_hdr(&syn_ack_tcp_packet.eth);
  setup_ip_hdr(&syn_ack_tcp_packet.ip, 0);
  setup_tcp_hdr(&syn_ack_tcp_packet.tcp);

  syn_ack_tcp_packet.opts.mss.size        = HTONS(TCP_MSS);
  syn_ack_tcp_packet.opts.ts.ts           = htonl(cur_ms_ts());
  syn_ack_tcp_packet.opts.ts.echo         = cur_conn->client_opts.ts;
  syn_ack_tcp_packet.opts.win_scale.shift = TCP_WIN_SCALE;

  tcp_syn_ack_checksum();

  nm_send_packet(&syn_ack_tcp_packet, sizeof(syn_ack_tcp_packet));
}
```

Listing 4.5: Inizializzazione e setup di un pacchetto SYN ACK.
Protocollo TCP (opzioni)

Anche in questo caso viene messa a disposizione una funzione di inizializzazione (*init_syn_ack_tcp_packet*) richiamata all'avvio dello stack per inizializzare il template, e una funzione *send_tcp_syn_ack* che ha il compito di specializzare il template generico prima della copia di quest'ultimo nel ring della scheda di rete. È quest'ultima funzione che ha il compito di richiamare le funzioni offerte dai protocolli degli strati inferiori per specializzare i relativi header.

Dopo l'invio del pacchetto la connessione sarà in stato *SYN SENT*, in attesa di un segmento di acknowledge per transitare nello stato *ESTABLISHED*.

4.2.2 Procedura generica per l'invio di un pacchetto

Il modo di operare utilizato per inviare una risposta alla richiesta di apertura di una connessione TCP è comune alla procedura utilizzata per inviare tutti i pacchetti, e può essere generalizzato come segue:

- inizializzazione dei pacchetti precostruiti (operazione effettuata solo una volta all'avvio del server)
- chiamata alle funzioni di setup degli header degli strati inferiori
- chiamata alla funzione di setup generico dell'header TCP
- setup dei campi specifici del pacchetto
- calcolo del checksum
- richiesta di un buffer libero nel ring di trasmissione
- copia del pacchetto nel buffer della scheda di rete
- invio cumulativo di tutti i pacchetti presenti nel ring di trasmissione (quando questo è pieno o quando è terminato uno dei loop del modulo netmap)

4.2.3 Ricezione di segmenti TCP

Dopo l'instaurazione di una nuova connessione il client invierà probabilmente una richiesta HTTP. Poiché lo stack di rete riconosce la connessione come stabilita, viene richiamata la funzione *process_tcp_segment*. Quest'ultima funzione ha il compito di processare un generico segmento TCP di una connessione il cui stato è *ESTABLISHED*.

```
static inline
void
process_tcp_segment(void)
{
  parse_tcp_options(cur_pkt->tcp_hdr);

  if (cmp_seq(ntohl(cur_pkt->tcp_hdr->seq), cur_conn->last_recv_byte) <= 0) {
    send_tcp_ack();
    return;
  }

  if (tcp_payload_len(cur_pkt)) {
    process_tcp_segment_data();
    send_tcp_ack();
  }
```

```
  if (flag_ack(cur_pkt->tcp_hdr)) {
    process_tcp_segment_ack();
  }
}
```

Listing 4.6: Processing di un segmento TCP

Il primo controllo effettuato riguarda il numero di sequenza; se questo è minore dell'ultimo ricevuto significa che il pacchetto è un duplicato (e viene quindi scartato inviando un ACK).

Viene quindi controllato se il pacchetto ha una dimensione del payload diversa da zero. In caso affermativo la funzione *process_tcp_segment_data* ha il compito di estrarre il payload dal pacchetto e inserirlo nel buffer di ricezione.

```
static inline
void
process_tcp_segment_data(void)
{
  char     *payload = tcp_payload_buf(cur_pkt);
  uint16_t len      = tcp_payload_len(cur_pkt);

  memcpy(cur_conn->data_buffer + cur_conn->data_len, payload, len);
  cur_conn->data_len += len;

  cur_conn->last_recv_byte += len;

  if (flag_psh(cur_pkt->tcp_hdr)) {
    handle_http_request();

    if (cur_conn->http_response->parser->parsed) {
      cur_conn->http_response_start_seq = cur_conn->last_sent_byte + 1;
    }
  }
}
```

Listing 4.7: Processing del payload TCP

È importante notare come il buffer di ricezione dello stack e quello dell'applicazione non siano due buffer separati (al contrario di quanto avverrebbe con uno stack di rete convenzionale).
I dati vengono infatti mantenuti in un unico buffer, che viene passato alla funzione *handle_http_request* ogni volta che il client setta il flag PSH nel segmento TCP. Il buffer può essere cancellato solo quando quest'ultima funzione segnala che il parsing della richiesta HTTP è stato effettuato con successo.

In caso di parsing effettuato con successo la struttura che descrive la connessione conterrà un puntatore valido ad un descrittore di una risposta HTTP, che potrà essere usato dal modulo netmap nel loop di invio dei dati.

Nel caso il segmento abbia il flag ACK settato, viene richiamata la funzione *process_tcp_segment_ack*. Questa ha lo scopo di aggiornare la di-

mensione della finestra effettiva di trasmissione (data dalla differenza tra la finestra pubblicizzata dal client e il numero dei byte in transito non ancora confermati).
La funzione ha inoltre il compito di ritrasmettere eventuali segmenti persi: se il client conferma infatti più volte lo stesso ACK significa che ha ricevuto dei pacchetti il cui numero di sequenza è maggiore del numero di sequenza aspettato (ovvero uno o più segmenti non sono stati consegnati correttamente).

4.2.4 Invio della riposta HTTP

Per inviare la risposta HTTP lo stack di rete mette a disposizione la funzione *tcp_conn_send_data*.
Il modulo netmap, durante il loop per l'invio dei dati, scandisce la lista delle connessioni attive e per ognuna di esse richiama *tcp_conn_has_data_to_send*, la quale ritorna un valore vero solamente se la connessione contiene una risposta HTTP che non è ancora stata inviata completamente e la finestra TCP per l'invio dei dati permette l'invio di ulteriori dati.
Se le condizioni sono soddisfatte viene richiamata la funzione *tcp_conn_send_data*, la quale ha il compito di inviare un certo numero di segmenti, a partire da quello successivo all'ultimo inviato.
Poiché il contesto di invio è mantenuto nel descrittore della connessione è sufficiente richiamare quest'ultima funzione più volte per completare l'invio di una risposta.
Per inviare l'header ed il body della risposta la funzione *tcp_conn_send_data* richiama le funzioni *tcp_conn_send_data_http_hdr* e *tcp_conn_send_data_http_file*.

Invio dell'header HTTP

tcp_conn_send_data_http_hdr ha il compito di inviare l'header HTTP presente nella struttura di tipo *http_response_t* che descrive la risposta.

```
void
tcp_conn_send_data_http_hdr(tcp_send_data_ctx_t *ctx)
{
#define MAX_SLOT 64

  http_response_t *res;
  nm_tx_desc_t    tx_desc;

  char    *payload_buf;
  uint16_t payload_len = 0;
```

```
  res = cur_conn->http_response;

  while (http_res_has_header_to_send(res) &&
    tcp_conn_has_open_window() && ctx->slot_count < MAX_SLOT) {
    if (unlikely(nm_send_ring_empty())) {
      break;
    }

    nm_get_tx_buff(&tx_desc);

    payload_buf = TCP_DATA_PACKET_PAYLOAD(tx_desc.buf);
    payload_len = MIN(ETH_MTU - sizeof(data_tcp_packet),
      res->header_len - res->header_pos);

    memcpy(payload_buf, res->header_buf + res->header_pos, payload_len);

    *tx_desc.len     = sizeof(data_tcp_packet) + payload_len;
    res->header_pos += payload_len;

    if (payload_len == cur_conn->client_opts.mss - sizeof(tcp_data_opts_t) ||
      res->file_len == 0) {
      send_tcp_data(tx_desc.buf, payload_buf, payload_len);

      cur_conn->last_sent_byte  += payload_len;
      cur_conn->recv_eff_window -= payload_len;

      ctx->slot_count++;
    } else {
      ctx->http_hdr_last_tx_desc.buf = tx_desc.buf;
      ctx->http_hdr_last_tx_desc.len = tx_desc.len;

      ctx->last_http_hdr_pl_len      = payload_len;

      break;
    }
  }
}
```

Listing 4.8: Invio dell'header di una risposta HTTP

Fino a ché la risposta non è stata completamente inviata, la finestra TCP è ancora aperta ed il numero di buffer disponibili non è stato utilizzato completamente, viene recuperato un buffer dal ring di trasmissione della scheda di rete con la funzione *nm_get_tx_buff* e viene copiato in questo all'offset corretto (ovvero dopo gli header dei protocolli Ethernet, IP e TCP) la parte di header HTTP che può essere contenuta nel segmento. Quindi possono darsi due casi.

Il payload riempie completamente il pacchetto: in questo caso può essere invocata la funzione *tcp_send_data*. Questa ha il compito di copiare nel buffer del ring di trasmissione (dove è già stato scritto l'header HTTP) gli header dei protocolli sottostanti, ottenuti ancora una volta specializzando un template generico in parte preinizializzato.

Il payload non riempie completamente il pacchetto: in questo caso non può essere richiamata subito la funzione *tcp_send_data*, in quanto po-

trebbe esserci una parte di body HTTP da copiare nel pacchetto prima di inviarlo (e nella pratica questa è la situazione più comune).
Viene quindi utilizzato un ulteriore contesto per segnalare alla funzione *tcp_conn_send_data_http_file* la presenza di un pacchetto nei ring di trasmissione della scheda di rete con una prima parte di payload già utilizzata. In questo modo quest'ultima funzione può riempire il payload del pacchetto che contiene già la parte finale dell'header HTTP (scrivendo la parte iniziale del body HTTP), invocare la funzione *tcp_send_data* e solo in seguito richiedere un nuovo buffer del ring di trasmissione.

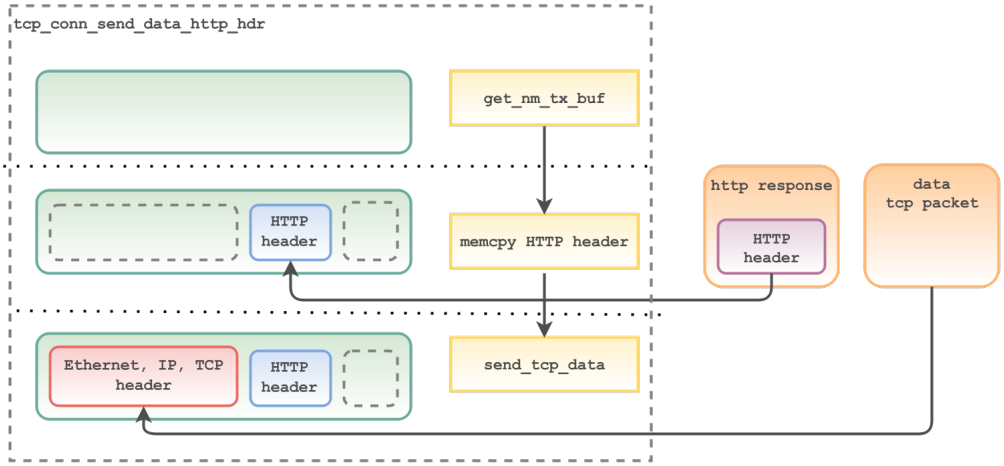

Figura 4.2: Invio dell'header di una risposta HTTP

Invio del body HTTP

La funzione *tcp_conn_send_data_http_file* ha il compito di inviare il body della risposta HTTP, ovvero un file letto da disco.

```
void
tcp_conn_send_data_http_file(tcp_send_data_ctx_t *ctx)
{
  http_response_t *res;
  nm_tx_desc_t    tx_desc[MAX_SLOT];
  struct iovec    iov[MAX_SLOT];
  int             iovcnt;
  size_t          start_pos;
  char            *payload_buf;
  uint16_t        payload_len;
  uint16_t        payload_offset;

  res            = cur_conn->http_response;
  iovcnt         = 0;
```

```
  start_pos      = res->file_pos;
  payload_offset = ctx->last_http_hdr_pl_len;

  while (http_res_has_file_to_send(res) && tcp_conn_has_open_window()
    && ctx->slot_count < MAX_SLOT) {
    if (nm_send_ring_empty()) break;

    if (payload_offset) {
      tx_desc[iovcnt].buf = ctx->http_hdr_last_tx_desc.buf;
      tx_desc[iovcnt].len = ctx->http_hdr_last_tx_desc.len;
    } else {
      nm_get_tx_buff(&tx_desc[iovcnt]);
    }

    payload_buf = TCP_DATA_PACKET_PAYLOAD(tx_desc[iovcnt].buf) + payload_offset;
    payload_len = MIN(ETH_MTU - (sizeof(data_tcp_packet) + payload_offset),
      res->file_len - res->file_pos);

    *tx_desc[iovcnt].len       = sizeof(data_tcp_packet) + payload_offset +
      payload_len;
    cur_conn->recv_eff_window -= payload_offset + payload_len;
    payload_offset             = 0;

    iov[iovcnt].iov_base = payload_buf;
    iov[iovcnt].iov_len  = payload_len;
    res->file_pos       += payload_len;

    ctx->slot_count++;
    iovcnt++;
  }

  if (iovcnt > 0) {
    preadv(res->file_fd, iov, iovcnt, start_pos);

    if (ctx->last_http_hdr_pl_len) {
      iov[0].iov_base = (char *) iov[0].iov_base - ctx->last_http_hdr_pl_len;
      iov[0].iov_len  = iov[0].iov_len + ctx->last_http_hdr_pl_len;
    }

    for (int i = 0; i < iovcnt; i++) {
      send_tcp_data(tx_desc[i].buf, iov[i].iov_base, iov[i].iov_len);
      cur_conn->last_sent_byte += iov[i].iov_len;
    }
  }
}
```

Listing 4.9: Invio del body di una risposta HTTP

La funzione utilizza la system call *preadv*. Questo permette di leggere un file da disco ponendo il contenuto in buffer multipli con una sola chiamata a funzione. *pvread* accetta come argomenti un file descriptor, un vettore di strutture iovec (chiamato *iov*) e la dimensione del vettore (*iovcnt*).
Una struttura di tipo *iovec* contiene due campi: un puntatore chiamato *iov_base* che contiene l'indirizzo di quello specifico buffer e un intero che contiene il numero di byte che quel buffer può contenere (campo *iov_len*).

La system call permette quindi di leggere un numero di buffer pari a *iovcnt*, copiando ognuno di questi direttamente nel buffer puntato dal campo

iov_ base della rispettiva struttura *iovec*.
In questo modo è possibile effettuare la lettura del file in maniera efficiente, copiando le porzioni di file direttamente nei buffer dei pacchetti all'offset corretto (dove l'offset è dato dalla somma delle lunghezze degli header dei protocollo sottostanti ad HTTP).

Per fare questo la funzione lavora in tre passaggi.
Packet buffer: con il primo ciclo la funzione recupera un certo numero di descrittori di buffer di pacchetto dal ring di trasmissione ed inizializza contemporaneamente le strutture di tipo iovec settando l'indirizzo su cui copiare il payload (pari all'indirizzo base del buffer del pacchetto al quale viene sommata la lunghezza degli header).
Payload dei pacchetti: la seconda fase è la chiamata alla funzione preadv: con una sola system call i payload di ogni pacchetto vengono inizializzati con la porzione di file che spetta loro.
Header dei pacchetti: per ogni pacchetto viene invocata la funzione *send_ tcp_ data*, che, come con l'invio degli header HTTP, inizializza gli header dei protocolli sottostanti.

4.3 Trasmissione affidabile

Per rendere affidabile la trasmissione dei dati ExoTCP implementa alcune tecniche per individuare eventuali pacchetti persi e ritrasmetterli.
Il protocollo TCP richiede che per ogni segmento ricevuto venga inviato un segmento di acknowledgment (ovvero con il flag ACK attivo), nel quale il ricevente conferma il numero di sequenza dell'ultimo segmento ricevuto. Il mittente ha due modi di riconoscere un (possibile) pacchetto non ricevuto dal destinatario.
Un primo modo consiste nel tenere traccia dei segmenti inviati e dei relativi ACK ricevuti: se il destinatario non risponde con un ACK entro un tempo pari a 4 volte il round trip time (RTT) il mittente può assumere che il pacchetto sia andato perso, e rieffettuare la trasmissione.
Vi è inoltre un secondo modo per riconoscere un probabile pacchetto non consegnato. Un segmento di acknowledgment non contiene infatti il numero di sequenza dell'ultimo segmento ricevuto, bensì il numero di sequenza dell'ultimo segmento che è stato accettato per ricostruire lo stream in ricezione. Quindi, nel caso il mittente spedisca i segmenti x, y e z, ed il segmento y vada perso, il ricevente confermerà con un primo ACK il segmento x, e alla ricezione di z confermerà nuovamente il segmento x (non potendo ancora accettare il segmento z).

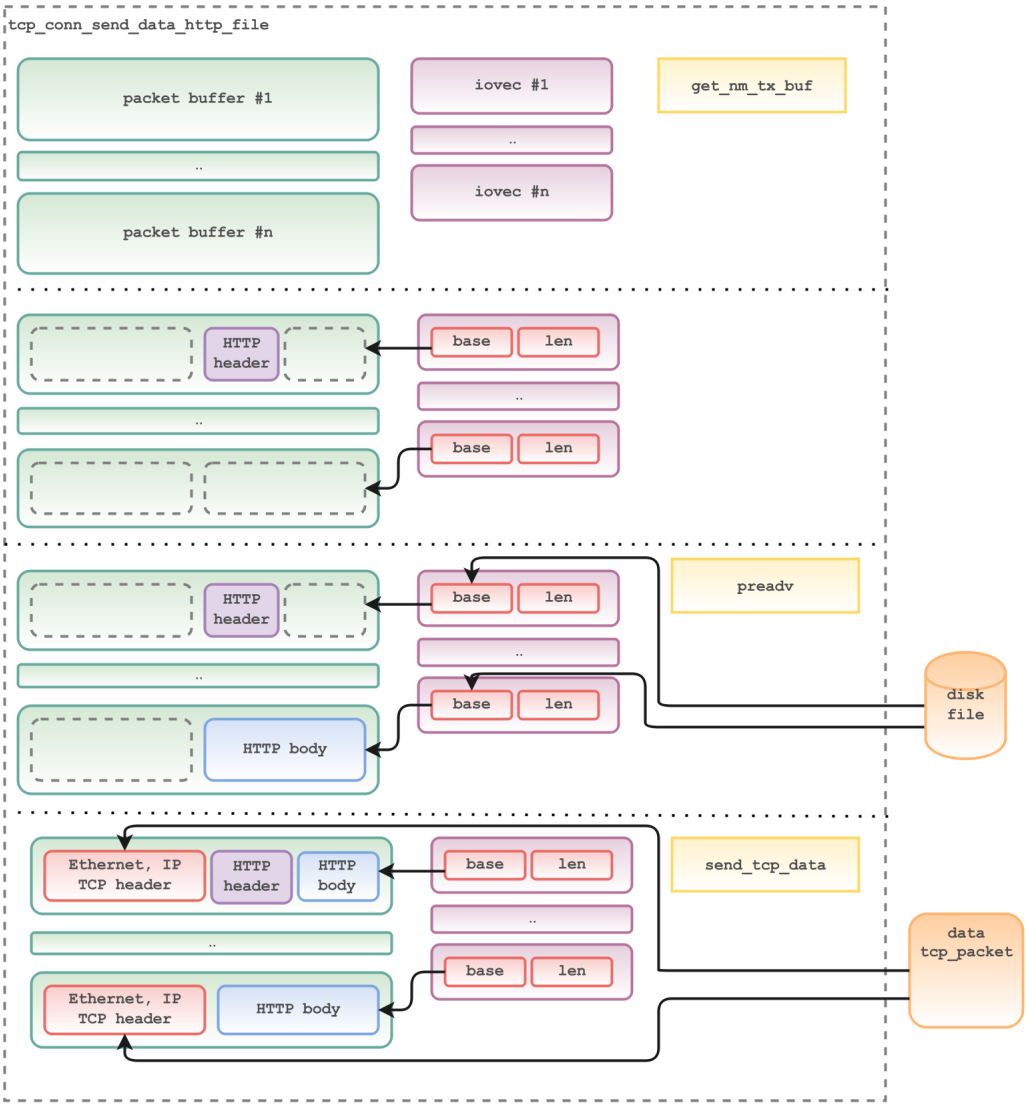

Figura 4.3: Invio del body di una risposta HTTP

In entrambi i casi la perdita di pacchetti è solo presunta, in quanto potrebbe essere andato perso il segmento di acknowledgment (o essere stato consegnato un ACK duplicato). Tuttavia il secondo metodo è più efficiente in quanto permette di ritrasmettere eventuali pacchetti persi (evitando che la finestra di trasmissione si chiuda) senza attendere il timeout di 4 volte il RTT.

Descrittori di segmenti non riconosciuti

Per implementare il meccaniso di individuamento e ritrasmissione ExoTCP utilizza la struttura *tcp_unackd_segment_t* con la quale tiene traccia dei segmenti che sono stati inviati ma non hanno ancora ricevuto una conferma.
Questa struttura contiene due campi: il numero di sequenza del segmento ed il timestamp oltre al quale il segmento deve considerarsi perso. Ogni connessione mantiene la propria lista (ordinata per timestamp crescenti) di segmenti che attendono una conferma.

L'invio di un segmento con la funzione *tcp_send_data* comporta la chiamata della funzione *track_unackd_segment*, la quale ha il compito di istanziare una struttura di tipo *tcp_unackd_segment_t* e di aggiungerla alla lista locale della connessione.
La ricezione di un segmento con il flag ACK attivo comporta invece la chiamata della funzione *ack_segment*, che ha il compito di scorrere l'intera lista di segmenti non ancora confermati, rimuovendo da questa tutti i descrittori il cui numero di sequenza è minore dell'ultimo ACK ricevuto (il ricevente potrebbe infatti confermare con un solo ACK cumulativo più segmenti).

Oltre alla lista di ogni connessione viene mantenuta una lista globale di elementi di tipo *tcp_per_conn_min_retx_ts_t*. Questa lista contiene un elemento per ogni connessione che ha segmenti non ancora confermati, e contiene il timestamp minimo tra tutti i timestamp di ritrasmissione di quella connessione. Anche questa lista viene inoltre mantenuta ordinata per timestamp crescenti.

L'ordinamento delle liste di ogni connessione e quello della lista globale viene effettuato al termine del loop di ricezione dei pacchetti (in quanto la ricezione di un ACK duplicato comporterebbe la ritrasmissione di un pacchetto e l'aggiornamento del suo timestamp di ritrasmissione) e al termine del loop di invio.

Individuazione e ritrasmissione di segmenti non consegnati

La maggior parte dei pacchetti persi viene riconosciuta nel loop di netmap per la ricezione dei pacchetti. È qui che vengono ricevuti ACK duplicati che indicano la possibile perdita di un pacchetto. Se lo stack riceve un ACK duplicato provvede a richiamare la funzione per la ritrasmissione del segmento successivo all'ultimo confermato.

Poiché è inoltre probabile che, in caso di perdita di pacchetti, lo stack riceva più volte lo stesso ACK duplicato (in quanto il loop per l'invio dei dati invia decine di pacchetti sequenzialmente per ogni connessione), viene tenuta

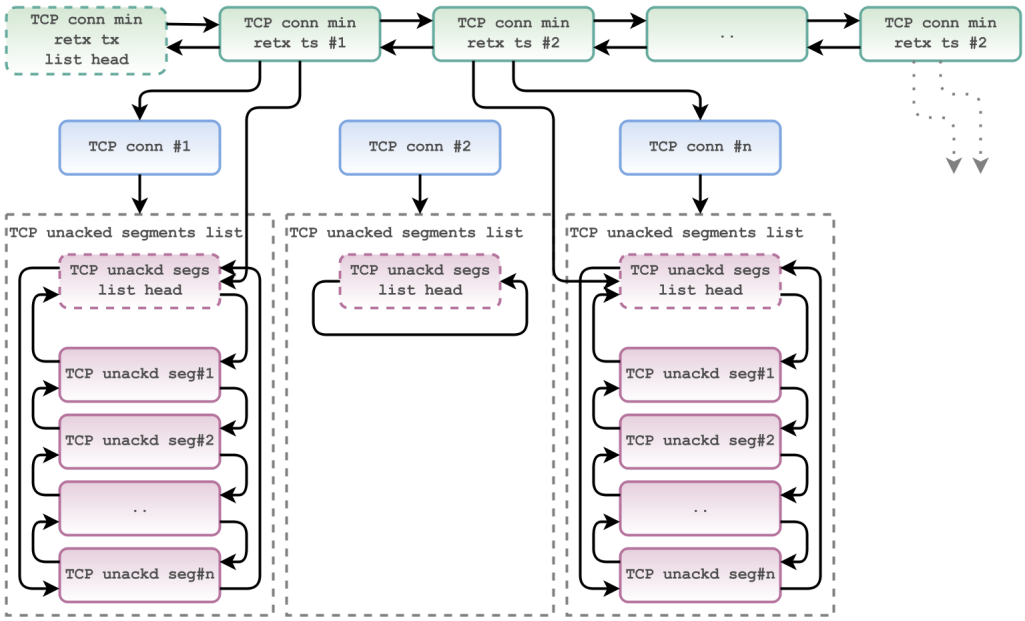

Figura 4.4: Strutture dati per la gestione degli ACK dei segmenti trasmessi

traccia dell'ultimo pacchetto ritrasmetto: se viene ricevuto un ACK duplicato che segnala nuovamente la perdita dell'ultimo pacchetto ritrasmesso, la ritrasmissione non viene effettuata (a meno che non sia scaduto il timestamp di ritrasmissione anche per il segmento ritrasmesso).
Viene inoltre implementato il meccanismo di controllo attivo basato su timeout: nel loop di netmap viene richiamata la funzione *nm_retx_loop* che ha il compito di scandire la lista globale delle connessioni che hanno pacchetti non ancora confermati.
Per ogni elemento della lista, se il valore del timestamp di ritrasmissione è antecedente a quello del timestamp corrente significa che uno o più pacchetti di quella connesione non hanno ricevuto un ACK. In questo caso viene quindi acceduta la lista della connessione contenente tutti i segmenti non ancora confermati, e scandita finché non si incontra il primo segmento che può considerarsi non ancora perso (il cui timestamp di ritrasmissione è quindi maggiore di quello corrente).
Entrambi i metodi appena descritti utilizzano quindi la funzione *tcp_retransm_segment* per ritrasmettere effettivamente il segmento.

```
void
tcp_retransm_segment(tcp_unackd_segment_t *seg)
{
  http_response_t *res;
```

```
  uint32_t start_byte;
  uint32_t header_start;
  uint32_t header_len;
  uint32_t file_start;
  uint32_t file_len;

  nm_tx_desc_t tx_desc;

  char    *payload_buf;
  uint16_t payload_len;
  uint32_t seq;

  header_start = 0;
  file_start   = 0;
  seq          = seg->seq;
  start_byte   = cmp_seq(seq, cur_conn->http_response_start_seq);

  res = cur_conn->http_response;

  if (start_byte < res->header_len) {
    header_start = start_byte;
    header_len   = MIN(cur_conn->client_opts.mss - sizeof(tcp_data_opts_t), res->
    header_len - header_start);
  } else {
    header_len = 0;
  }

  if (ETH_MTU - sizeof(data_tcp_packet) - header_len > 0) {
    file_start = start_byte - res->header_len;
    file_len   = MIN(cur_conn->client_opts.mss - sizeof(tcp_data_opts_t) -
    header_len, res->file_len - file_start);
  } else {
    file_len   = 0;
  }

  nm_get_tx_buff(&tx_desc);

  payload_buf = TCP_DATA_PACKET_PAYLOAD(tx_desc.buf);
  payload_len = header_len + file_len;

  *tx_desc.len               = sizeof(data_tcp_packet) + payload_len;
  cur_conn->recv_eff_window -= payload_len;

  memcpy(payload_buf, res->header_buf + header_start, header_len);
  pread(res->file_fd, payload_buf + header_len, file_len, file_start);

  send_tcp_data_retx(tx_desc.buf, payload_buf, payload_len, seq);

  seg->retx_ts = retx_ts();

  cur_conn->last_retx_seg_seq = seg->seq;
  cur_conn->last_retx_seg_ts  = seg->retx_ts;
}
```

Listing 4.10: Ritrasmissione di un segmento TCP

Poiché con ogni nuova richiesta HTTP viene salvato il numero di sequenza iniziale (ovvero il numero di sequenza del primo segmento contenente la risposta HTTP) l'unico parametro necessario alla funzione *tcp_retransm_segment*

è il numero di sequenza del pacchetto da ritrasmettere.
La differenza tra quest'ultimo e il numero di sequenza iniziale dà infatti l'offset del byte da cui iniziare a ristrasmettere e conoscendo l'offset iniziale, la lunghezza dell'header HTTP e la lunghezza del file è possibile ricostruire facilmente il payload del segmento che deve essere ritrasmesso.

Capitolo 5

Realizzazione e Prestazioni

L'applicazione è stata realizzata utilizzando il linguaggio C (in particolare lo standard GNU 11) senza utilizzare alcuna libreria esterna ed il sistema operativo target è Linux. Durante lo sviluppo ogni componente è stato realizzato e testato dapprima come applicazione separata e solo in seguito integrato con il resto del progetto.

Il primo modulo ad essere stato implementato è stato quello che interagisce con Netmap. L'obiettivo era disporre di un'applicazione che fosse in grado di ricevere e inviare pacchetti semplicemente leggendoli e scrivendoli sui ring della scheda di rete.

Il passo successivo è stato aggiungere al di sopra di quest'ultimo modulo uno stack di rete, e la scelta è ricaduta su PicoTCP [1], uno stack TCP/IP userspace implementato in C. L'obiettivo era ottenere un'applicazione che si comportasse come un echo server TCP (ovvero un servizio TCP che rispedisce al mittente i dati che questo gli invia). La scelta di PicoTCP è dovuta al fatto che inizialmente era stata valutata l'idea di adattare uno stack già esistente al protocollo HTTP invece di progettarne uno nuovo. Tuttavia ben presto l'idea è stata scartata perché adattare uno stack generico non avrebbe reso possibile ottenere il grado di flessibilità necessario; PicoTCP è stato infatti progettato per essere generico e adatto a dispositivi embedded.

In seguito è stato sviluppato il parser HTTP, adattando il codice del server Web Puma, testandolo prima come applicazione separata (utilizzando alcune richieste HTTP di test sottoforma di stringa passata al programma) ed integrandolo infine nel progetto.

L'ultimo modulo ad essere stato realizzato è quello che implementa lo stack di rete ExoTCP. Anche in questo caso lo sviluppo è stato incrementale: per prima cosa è stato implementato un semplice supporto ai protocolli Ethernet e ARP, cosicché fosse possibile testare se il server era in grado di

rispondere al primo pacchetto che tipicamente riceve (ovvero una richiesta ARP).
In seguito è stata quindi implementata la gestione dei protocolli IP e TCP: per quest'ultimo il primo obiettivo è stato completare il three way handshake. Successivamente è stata implementata la ricezione di segmenti con un payload, la risposta con un ACK, e l'invio dei dati. Ognuna di queste fasi ha richiesto di intervenire ulteriormente sul modulo di Netmap. L'ultima funzionalità dello stack ad essere stata implementata è stata la tramissione affidabile.

5.1 QEMU-KVM

Per il testing delle performance è stata utilizzata una macchina virtuale. Nello specifico è stata utilizzata un'installazione di Arch Linux con un kernel 3.14 (la corrente versione Long Term Support) su una macchina virtuale QEMU [2] con attivo il supporto per KVM.

La macchina host è un portatile con una CPU Intel Haswell Core i5 4288U ad una frequenza di 2.60 GHz, 8 GB di RAM ed un disco SSD da 512 GB. Lo script di avvio della macchina virtuale è riportato nel listato 5.1.

```
#!/bin/sh

qemu-system-x86_64 -enable-kvm -m 2048 -cpu Haswell                    \
-net nic,vlan=0 -net tap,vlan=0,ifname=tap0,script=no,downscript=no     \
-net nic,vlan=1 -net tap,vlan=1,ifname=tap1,script=no,downscript=no     \
-net nic,vlan=2 -net user,vlan=2                                        \
-smp cpus=2,threads=2                                                  \
-boot order=c -drive file=arch.img,if=virtio -display none              &
```

Listing 5.1: Script di avvio della macchina virtuale

La scelta della macchina virtuale è stata obbligatoria per alcuni motivi:
kernel LTS: utilizzando una macchina virtuale è possibile mantenere un kernel LTS nell'ambiente di test (meno adatto per essere usato su un sistema desktop). In questo modo si riduce la possibilità che un aggiornamento del sistema possa portare all'insorgenza di problemi di compatibilità con le nuove versioni del kernel
host ring: la comunicazione via rete tra due processi che sono in esecuzione sulla stessa macchina è soggetta ad alcune ottimizzazioni del sistema operativo. I frame di rete non vengono infatti scritti sui ring dell'interfaccia fisica, ma vengono posti su dei ring virtuali gestiti dal sistema operativo (poiché non devono essere trasmessi nella rete). L'utilizzo di questi ring potrebbe falsare la misura delle prestazioni, settando una limitazione superiore presta-

zionale che probabilmente non potrà mai essere raggiunta.
Utilizzando una macchina virtuale ed una scheda di rete emulata i benchmark potranno dare dei risultati prestazionali peggiori o al massimo uguali a quelli di una macchina fisica, ma non potenzialmente migliori (e quindi irraggiungibili nella pratica) come quelli che ci si potrebbe aspettare dall'utilizzo degli host ring.
costo delle schede 10 gigabit: poiché anche un server web tradizionale riesce a saturare la banda di una scheda di rete gigabit, è necessario utilizzare schede Ethernet 10 gigabit per poter apprezzare una eventuale differenza prestazionale, tuttavia il costo di queste schede è elevato.
Su QEMU è possibile emulare una scheda gigabit con chipset e1000, ma l'emulazione non impone il limite della banda di 1 gigabit.
In questo modo è possibile apprezzare la differenza prestazionale tra le soluzioni tradizionali ed Eth, avendo a disposizione una banda effettivamente superiore a quella gigabit di una scheda fisica.

5.1.1 Networking

La macchina virtuale è collegata alla macchina fisica utilizzando due interfacce TAP, ognuna delle quali è mappata sulla relativa interfaccia interna della macchina virtuale.
La prima viene utilizzata per accedere al sistema utilizzando SSH, mentre la seconda viene utilizzata dal server web e da Netmap.

5.2 Prestazioni

Nel seguito verranno presentati i risultati dei test prestazionali effettuati. Le prestazioni vengono comparate con quelle del server web Nginx [11]. La configurazione utilizzata per quest'ultimo è quella di default.

5.2.1 Trasferimento di un file, singola connessione

Il primo test è stato effettuato sul trasferimento di un singolo file del peso di 500MB.

Con Nginx la velocità media di trasferimento del file è di circa **160 MB/s**. Utilizzando Eth la velocità di trasferimento media è di circa **460 MB/s**. Con una singola connessione Eth riesce quindi a sfruttare maggiormente la banda (in questo caso virtuale) disponibile dimostrandosi quasi tre volte

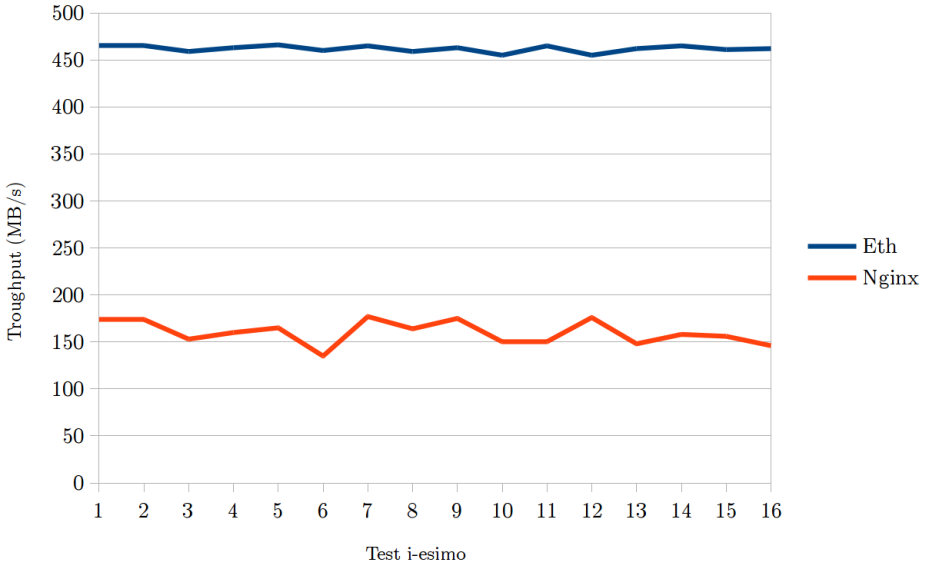

Figura 5.1: Prestazioni: QEMU-KVM, 4 Core 2.6GHz, 2GB RAM, singola connessione

più veloce. Nella figura 5.1 sono riportati nell'asse delle ascisse e delle ordinate rispettivamente la richiesta HTTP i-esima e la relativa velocità di trasferimento.

5.2.2 Trasferimendo di un file, multiple connessioni concorrenti

Testando Eth con più richieste concorrenti si è registrato un calo prestazionale in termini di banda, e allo stesso tempo con Nginx si è registrato invece un incremento della banda effettiva sviluppata.
Con 10 connessioni concorrenti Eth riesce a servire ogni richiesta con una banda media di circa 28 MB/s, sviluppando quindi una banda complessiva di **280 MB/s**.
Nginx riesce invece a servire ogni connessione con una banda media di circa 36 MB/s, sviluppando quindi una banda complessiva pari a **360MB/s**.
Il grafico è mostrato in figura 5.2, dove sono riportati il numero di connessioni simultanee utilizzate nel test e la relativa banda complessiva sviluppata.

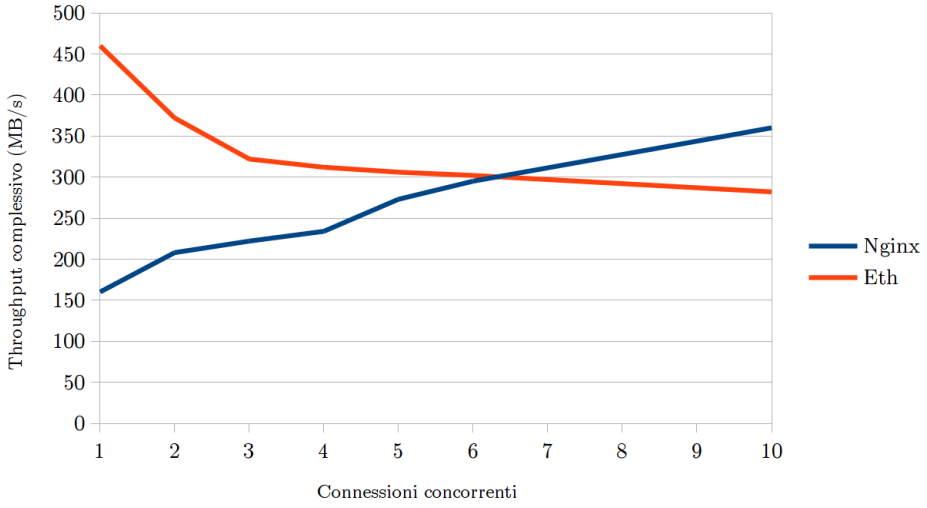

Figura 5.2: Prestazioni: QEMU-KVM, 4 Core 2.6GHz, 2GB RAM, connessioni multiple

Il test tuttavia non può essere considerato completamente significativo, in quanto i server vengono eseguiti su una macchina virtuale, la quale a sua volta è eseguita sulla stessa macchina fisica che si comporta da client (e quindi non è da escludere che la CPU sia satura per il carico di lavoro).
Inoltre Eth al momento utilizza un'architettura single thread e quindi è in grado di utilizzare solo un core mentre Nginx può lanciare un processo per ogni core disponibile, e può quindi sfruttare la potenza di calcolo di quattro core anziché uno.

Capitolo 6

Conclusioni e sviluppi futuri

Lo sviluppo di questo server web ha permessi di provare che un'architettura altamente specializzata è effettivamente in grado di offrire performance superiori a quelle di un'architettura tradizionale che si appoggia completamente al sistema operativo.
Se il costo dei servizi che il sistema operativo offre è generalmente trascurabile in una macchina che si comporta da client come un notebook o uno smartphone, questo non è altrettanto vero se si considera un sistema di tipo server. In quest'ultimo caso un aumento delle prestazioni si traduce infatti nella possibilità di ridurre il numero di macchine utilizzate e in un minor consumo di queste ultime, ovvero in una riduzione dei costi per l'acquisto ed il mantenimento dei sistemi.
Per quanto riguarda i test effettuati, in quelli su singola connessione Eth riesce ad essere quasi 3 volte più veloce rispetto a Nginx, e probabilmente è possibile migliorare ulteriormente le prestazioni utilizzando una scheda di rete fisica e le funzioni che questa offre (come il *TCP Segment Offloading*, che permette di delegare alla scheda di rete alcuni oneri come il calcolo del checksum dei pacchetti, alleggerendo il carico della CPU).
I test con connessioni multiple invece, se da un lato decretano al momento la vittoria delle soluzioni tradizionali, offrono allo stesso tempo uno spunto sulla direzione verso la quale deve procedere lo sviluppo di Eth, ovvero il supporto multicore.

Il server web Eth può inoltre essere inserito all'interno di un contesto denominato "Internet of Threads" [3].
Con questo termine si fa riferimento al concetto di considerare un processo come un nodo attivo della rete, dotato di un proprio indirizzo.
Il processo del server in esecuzione rappresenta proprio questo: un'entità dotata di un proprio indirizzo IP, che bypassa il sistema operativo. Ne segue

che l'indirizzo IP del server non identifica più la macchina, ma il processo stesso: il sistema operativo non è infatti a conoscenza dell'attivita di rete del server web, delle sue connessioni o del suo indirizzo.

6.1 Codice Sorgente e Sviluppi Futuri

Il progetto è open source ed il codice sorgente è rilasciato sotto licenza GNU General Public License v2.0.
È possibile accedervi all'indirizzo *http://github.com/jibi/eth*.
Eth è innanzitutto un progetto personale, e per questo il suo sviluppo verrà portati avanti. Alcune delle modifiche in programma sono l'introduzione del supporto a interfacce di rete multiple, lo studio di strutture dati più avanzate, l'introduzione del supporto al protocollo HTTPS ed Secure Socket Layer (SSL), il supporto ad una modalità che consenta di usarlo come proxy per effettuare caching di risposte HTTP ed il supporto al sistema operativo FreeBSD.

Appendice A

Strutture Dati

Per non ricorrere all'utilizzo di librerie esterne sono stati implementati due tipi di strutture dati, che verranno brevemente descritti in questa appendice.

A.1 Double Linked List

Una double linked list è una struttura che permette di descrivere una sequenza di elementi. Ogni elemento contiene un puntatore all'elemento precedente e uno all'elemento successivo (da cui il nome lista doppiamente linkata). L'implementazione è basata ampiamente su quella che utilizza il kernel Linux.

A.1.1 API

Le funzioni per manipolare le liste sono le seguenti:
list_init e **list_new**: utilizzate rispettivamente per allocare una nuova lista vuota o inizializzarne una già esistente
list_add e **list_add_tail**: utilizzate rispettivamente per aggiungere un elemento in testa o in coda ad una lista
list_del: utilizzata per rimuovere un elemento dalla lista alla quale appartiene
list_head_attached: utilizzata per verificare se un elemento è inserito in una lista
list_empty: utilizzata per verificare se una lista contiene elementi
list_for_each_entry: macro utilizzata per scandire tutti gli elementi di una lista; viene espansa come ciclo *for*

A.2 Hash Table

Una tabella hash è una struttura dati che permette di memorizzare un insieme di chiavi e di valori, mettendo in relazione ogni chiave con il rispettivo valore. La tabella hash permette inoltre di accedere al valore associato ad una chiave in tempo O(1) nel caso medio. Per questo è stata la scelta ideale per memorizzare i TCP Control Block (come valori), acceduti tramite la relativa chiame (una coppia indirizzo sorgente e porta sorgente).

In questo caso è stata implementata una tabella hash basata su liste di trabocco.

A.2.1 API

Le funzioni per manipolare le tabelle hash le seguenti:
hash_table_init: alloca una struttura di tipo *hash_table_t* e la inizializza. I parametri sono due puntatori a funzione: la prima funzione viene richiamata quando deve essere calcolato l'hash di una chiave, la seconda quando deve essere effettuato il confronto di due chiavi
hash_table_insert: crea una nuova associazione chiave-valore nella tabella hash. Se la chiave è già presente il valore viene sovrascritto con quello nuovo. I parametri sono la tabella hash, la chiave ed il valore
hash_table_lookup: effettua la ricerca della chiave e restituisce il valore associato. I parametri sono la tabella hash e la chiave
hash_table_remove: rimuove, se presente, il valore specificato dalla chiave. I parametri sono la tabella hash e la chiave

A.2.2 Funzione Hash

Per il calcolo dell'hash è stata utilizzata la funzione Murmur.
Murmur è una funzione hash non crittografica (ovvero non è difficile invertirla) che si comporta bene con un insieme di chiavi distribuite casualmente ed è facile da calcolare. È stata scelta perché è stata adottata da alcuni importanti progetti come la libreria standard C++, Perl, Nginx e Ruby.

Bibliografia

[1] Tass Belgium. Picotcp. `http://www.picotcp.com/`.

[2] Fabrice Bellard. Qemu. `http://wiki.qemu.org/Main_Page`.

[3] Renzo Davoli. Internet of threads: Processes as internet nodes. In *International Journal in Advances in Internet Technology*, volume 7 n. 12, pages 17–28. IARIA, 2014.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[5] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[6] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 9:1–9:7, New York, NY, USA, 2013. ACM.

[7] L.L. Peterson, B.S.D. Larry Peterson, and B.S. Davie. *Reti di calcolatori. Idee & strumenti*. Apogeo, 2008.

[8] Evan Phoenix. Puma web server. `http://puma.io`.

[9] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[10] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, Bellevue, WA, August 2012. USENIX Association.

[11] Igor Sysoev. Nginx. `http://nginx.org`.

[12] Adrian Thurston. Ragel state machine compiler. `http://www.colm.net/open-source/ragel/`.