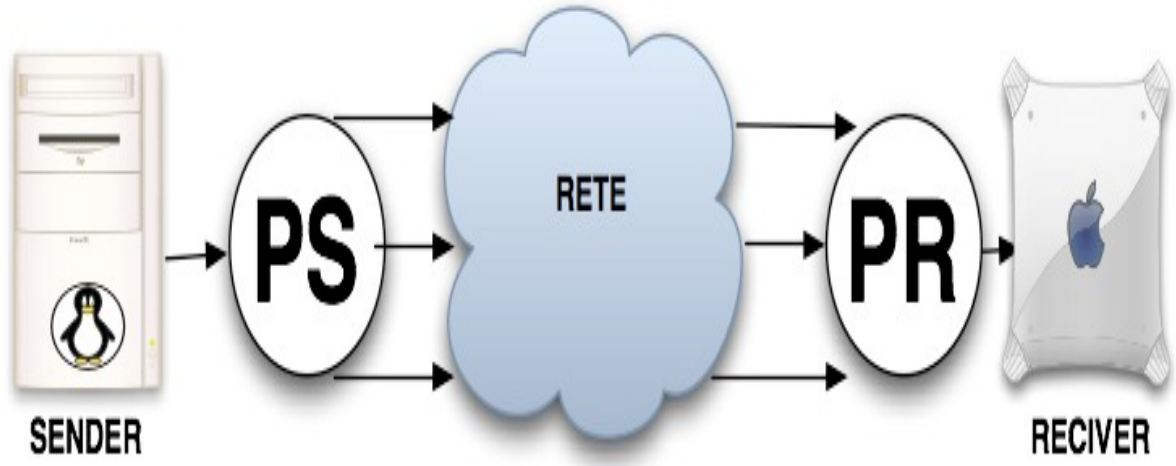
Fornire Interattività a Comunicazione TCP Sfruttando Multi-homing

Relazione di Progetto per l'esame di Laboratorio di Programmazione di Rete
a.a. 2006/2007, Dott. Vittorio Ghini

Studenti:
Matteo Bonicolini
Alessandro Gentile
Dennis Dalla Torre



INDICE

1
Funzionalità Realizzate

1.0 Quali

Il sistema realizzato, composto da due applicativi Proxy Sender (PS) e Proxy Receiver (PR), si occupa di instradare dati da una generica applicazione di tipo Client ad un'altra di tipo Server utilizzando tre diverse connessioni TCP e cercando di ottimizzare l'uso di ciascun canale al fine di permettere a blocchi di dati di circa 2 KB di arrivare a destinazione entro un tempo limite di 500 ms, o comunque nel minor tempo possibile.

In fase di progettazione, sono state evidenziate ed affrontate le seguenti problematiche. In primo luogo, per sfruttare al meglio i tre canali di connessione, è necessario bilanciare correttamente il carico dati, onde evitare il flood di un canale più lento o viceversa lasciare inutilizzate le risorse di un canale in grado di sostenere molto traffico. Questo può ragionevolmente portare a dividere un singolo blocco di dati in diverse parti, che non necessariamente viaggeranno sullo stesso canale.

Così facendo si apre un secondo problema: il TCP, su ciascun canale, garantisce il corretto ordine di arrivo dei dati, ma poiché vengono utilizzati tre canali (presumibilmente diversi fra di loro in prestazioni) non si può garantire ad esempio che le parti A, A', e A'' inviate sul canale x, arrivino prima di quelle B, B' e B'' inviate sul canale y.

Inoltre se una delle connessioni risultasse inutilizzabile durante l'invio di dati, non ci sarebbe modo di verificare se i pacchetti che viaggiavano sul canale al momento del crash siano effettivamente arrivati a destinazione o meno.

L'intero progetto è stato concepito per avere una concreta portabilità su sistemi posix.

1.1 Come?

Per quanto riguarda il bilanciamento del carico dati, è stata abbandonata l'idea iniziale di eseguire test periodici (basati sull'algoritmo packet-pair) che monitorassero le condizioni di ogni canale (in termini di bandwidth e latenza), poiché per permettere una buona interattività del programma, la scansione temporale di questi test sarebbe dovuta essere variabile e non statica. Inoltre questo tipo di test non si è rivelato adatto alla rete di riferimento utilizzata dal proxy (e simulata dal Ritardatore)[1]. Si è quindi deciso di utilizzare un algoritmo adattivo che, in base al RTT di ogni pacchetto, valuta la disponibilità di invio su ogni canale e gestisce il flusso dati di conseguenza. Nel primo round di scrittura, i dati da spedire vengono divisi in blocchi da 1024 byte (pari all'MTU del protocollo). L'algoritmo assegna un pacchetto ad ogni canale, fino all'esaurimento dei dati letti dal Client in quell'istante. Ad ogni round successivo in base alle prestazioni misurate, l'algoritmo modifica l'MTU proprio di ogni canale. In questo modo ogni qual volta il PS ha dei dati da inviare, sa esattamente quanti di questi può inviare su ogni canale senza sovraffollarlo. Nell'ipotesi in cui il calcolo fosse errato, o le condizioni della rete siano cambiate, nel round successivo si misureranno per forza di cose valori differenti di RTT e l'algoritmo modificherà nuovamente in maniera opportuna il carico dati.

Per il problema del riordinamento dei dati, sono state implementate apposite strutture dati (linked list) dove vengono memorizzati temporaneamente i pacchetti ricevuti sui tre canali. Il PR a questo punto controlla la lista ad ogni pacchetto ricevuto e, non appena sarà arrivato il pacchetto successivo a quelli già inviati, lo invierà al Server, mentre gli altri pacchetti non in ordine aspetteranno il loro turno nella lista.

Infine un sistema di ACK fa in modo che eventuali pacchetti persi (per crash di una connessione o per qualsiasi altro motivo) vengano ritrasmessi con priorità rispetto ai dati nuovi, per ridurre al minimo il ritardo. Gli ACK inoltre rappresentano lo strumento fondamentale per la misura del RTT.

2
Progettazione

Nell'approccio al lavoro, pur mantenendo focalizzata l'attenzione sugli obbiettivi del progetto, il gruppo ha cercato di mantenere una certa versatilità. Ci si è quindi orientati verso la realizzazione non tanto di un programma fine a se stesso, ma di un "core" minimale e generico che potesse fornire la base per una gamma di prodotti attinenti al Multi-homing e al bilanciamento del carico dati.

Gran parte di questo risultato si deve alla progettazione di un sistema di compilazione condizionale basata su flag che individuano informazioni sul sistema e sull'architettura utilizzati (portabilità) e permettono all'utente di definire varie strategie di bilanciamento del programma (versatilità). Tutto questo si concretizza nel programma Proxy_Configure.

Inoltre per rendere più leggero il lavoro all'utente finale (i.e. il programmatore che vorrà sviluppare un programma basato su questo core), è stato realizzato un set di API utili a risolvere i vari sottoproblemi incontrati, astraendo dai dettagli del linguaggio. In questo modo si ottengono i seguenti vantaggi: il programmatore può utilizzare le API senza scendere nel dettaglio linguistico (socket, listening, bind, malloc, ecc.), può individuare e correggere facilmente eventuali bug presenti nelle API stesse, ha garanzia della portabilità quanto meno su sistemi posix, può eventualmente ampliare la portabilità facendo modifiche semplici e ben localizzate nel codice.

2.0 API fornite

`Proxy_config.h` : questo header viene scritto dal programma Proxy_Configure (primo step del make) ed incluso in tutti gli altri header. Contiene la definizione di tutte le flag relative alle strategie di bilanciamento, all'architettura e alla conformità posix e ANSI C89 del sistema utilizzato.

`Message.h` : il modulo contiene funzioni per la gestione di messaggi nel programma. Include messaggi di help e di errore (con relativa gerarchia).

`Queue.h` : definisce la struttura dati coda un set di funzioni per la manipolazione di tale struttura. Includendo questo modulo il programmatore potrà utilizzare code (implicitamente) polimorfe.

`L_list.h` : include Queue.h e in più definisce le funzioni per lavorare su linked list (implicitamente) polimorfe.

`Packet.h` : definisce la struttura dati pacchetto del protocollo realizzato e le funzioni per lavorare su di esso.

`Channel.h` : definisce la struttura dati channel, che contiene tutte le informazioni relative ad un singolo canale di connessione (fd del socket, MTU del canale, ecc.) e le funzioni per lavorare su tale struttura.

`Net_Util.h` : contiene il set di funzioni che costituiscono il core del progetto, dalle funzioni che lavorano direttamente sui socket, a quelle che gestiscono code o liste di pacchetti e canali utilizzando gli header sopra elencati.

2.1 Struttura del programma

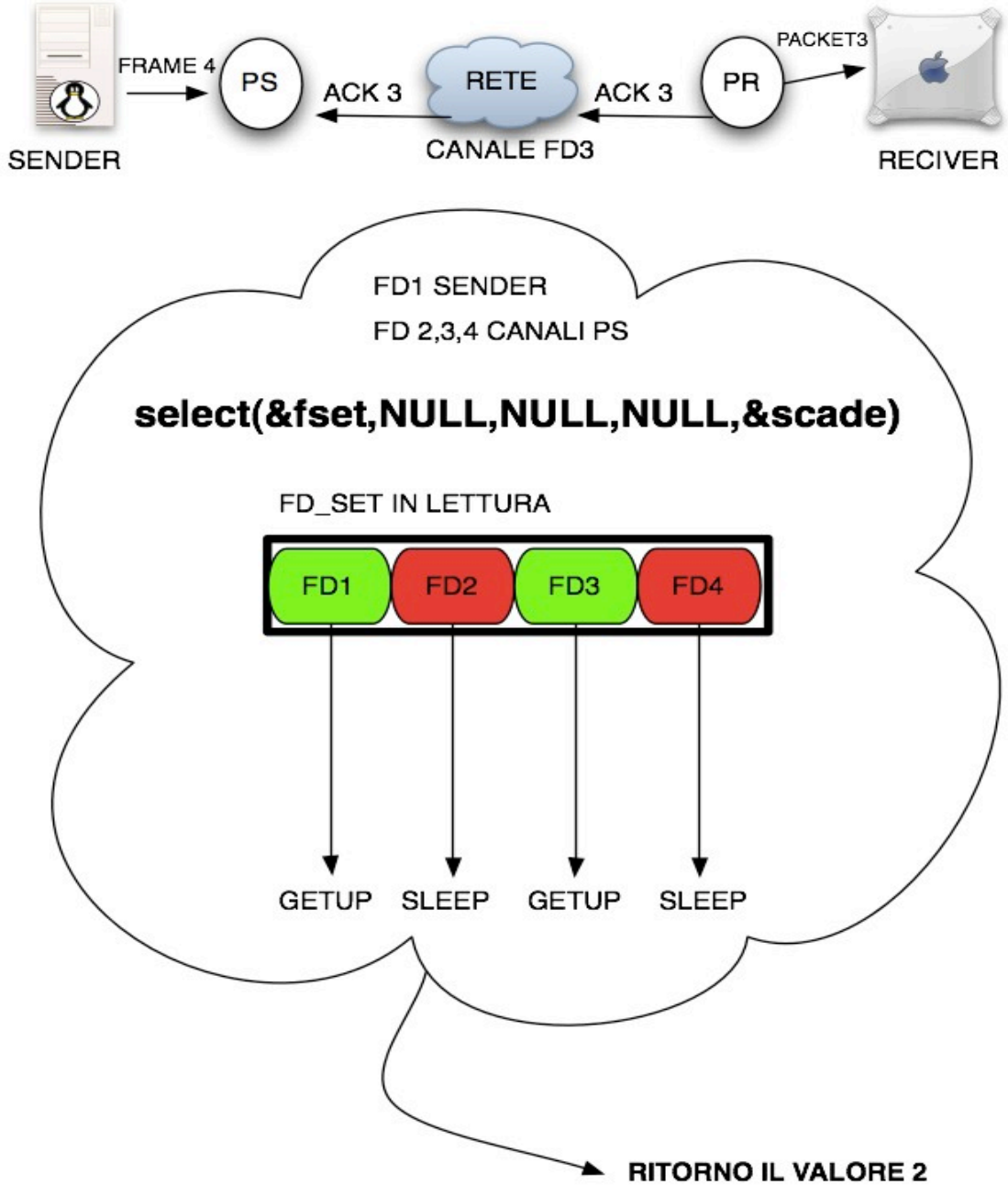
Il sistema Proxy realizzato è composto di due applicativi: Proxy Sender (PS) e Proxy Receiver (PR), idealmente interposti nella normale comunicazione fra Client e Server. Il PS accetta una connessione da un solo Client (da cui poi riceverà i dati) e instaura fino ad un massimo di tre connessioni verso il PR; questo si occupa di connettersi al Server e di consegnargli i dati nel giusto ordine[2].

I due programmi non effettuano alcun tipo di handshake, bensì ognuno dopo l'instaurazione delle connessioni, resta in attesa dei dati.

Il Proxy Sender è regolato da una select(), con un timeout di 4 secondi utile a capire se i canali sono down, che si sveglia quando sono richieste una o più delle seguenti operazioni. Sinteticamente gli step sono:

1) Leggere i dati in arrivo dal Client e memorizzarli nella coda di invio;
2) Leggere gli ACK in arrivo dai canali, misurarne il RTT (in base al quale settare la MTU del canale) e togliere i pacchetti confermati dall'apposita lista;
3) Prendere una quantità adeguata di dati dalla coda di invio, creare fino a tre pacchetti di dimensione pari all'MTU di ogni canale (comunque mai superiore a 1024 byte, MTU globale del protocollo) e inviarli sui tre canali.

ESEMPIO DI SELECT NEL PROXY SENDER IN LETTURA DURANTE LA COMUNICAZIONE



Ognuno di questi step è regolato da appositi permessi che vengono settati da un quarto step eseguito alla fine di ogni round che, in base alla specifica strategia di bilanciamento (decisa a tempo di compilazione) e alle misurazioni effettuale (RTT pacchetti) modifica delle flag che influenzano il comportamento del programma prima della select(). In questa fase preliminare alla select() infatti vengono inseriti negli fdset della select() gli fd del Client e dei canali solo a condizione che la flag lo permetta.

Durante la valutazione degli RTT dei pacchetti, viene anche gestito il resend dei medesimi. Se il timeout di un pacchetto scade senza che venga ricevuto l'ACK corrispondente, esso viene inserito in un'apposita lista i cui elementi avranno precedenza sugli altri nello "step 3" del round successivo.

Il Proxy Receiver è anch'esso regolato da una select() che a differenza del PS controlla solo gli fd in lettura e non ha nessun timeout. Il programma è diviso in due fasi: una di lettura e una di scrittura; la prima inizia allo svegliarsi della select(), mentre la seconda si concatena immediatamente dopo.

La fase di lettura esegue i seguenti step:

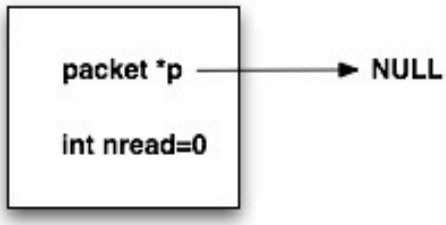
1) Si appoggia ad una struttura dati ausiliaria, contenente la porzione di pacchetto letta fino a quel momento e un intero che indica il numero di byte letti. Se entrambi i campi sono vuoti, viene effettuata la read() dell'header (6 byte). Altrimenti la read() prosegue la lettura fino a quando il numero di byte letti non è equivalente alla lunghezza totale del pacchetto (specificata nell'header).
2) Finito di leggere un pacchetto completo invia il corrispondente ACK verso il PS e inserisce il pacchetto nella lista di invio verso il Server.

La fase di scrittura, invece

1) Prende in considerazione un contatore che indica il numero seriale del prossimo pacchetto da inviare al Server e cerca il pacchetto corrispondente nella apposita lista;
2) se il pacchetto è presente, lo invia al Server e incrementa il contatore, altrimenti torna in select();
3) Ripete questi due step finchè la lista contiene pacchetti consecutivi.

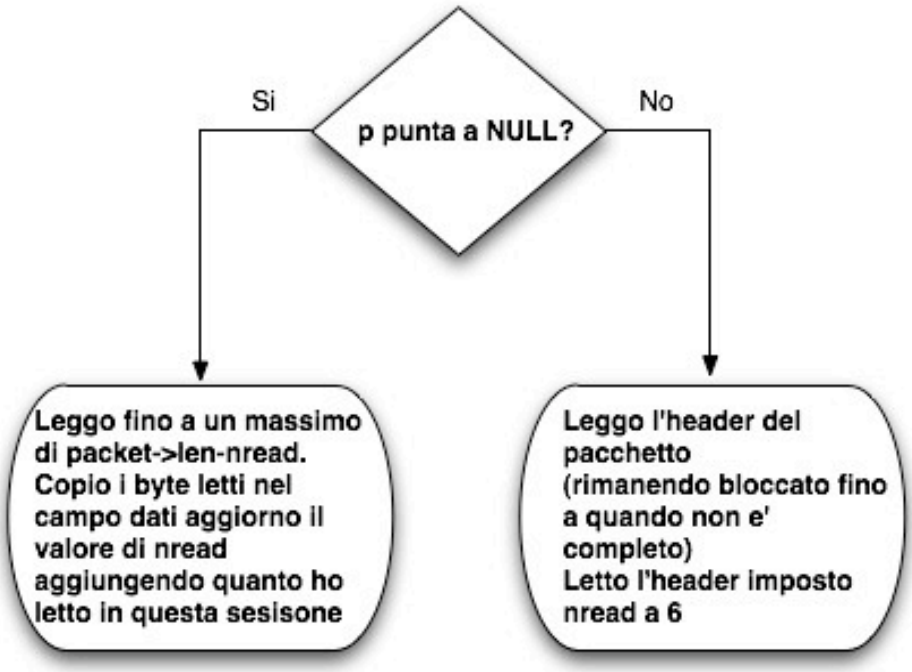
FUNZIONE READ_PACKET


```
struct i_pack
{
    packet *p;
    int nread;
};
```



Rappresentazione struttura i_pack ala sua creazione

La funzione read_packet() viene usata dal Proxy_reciver per leggere i pacchetti inviati dal Proxy_sender.Usa socket non bloccanti e viene quindi utilizzata dopo una chiamata alla funzione select().Da quindi per scontato che quando viene eseguita il file descriptor del canale sia pronto per la lettura.Inoltre la funzione prende pacchetti o totalmente assenti(cioe' bisogna leggere) o incompleti.Successivamente alla chiata della funzione il chiamante controlla se il pacchetto letto e' completo



3
Implementazione

3.0 Header

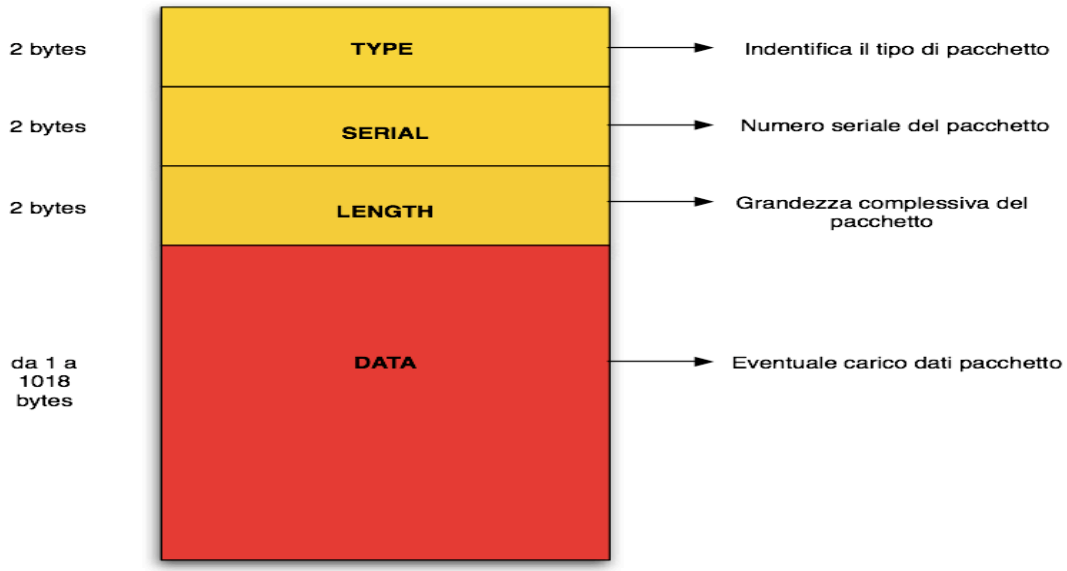
Come accennato all'inizio della documentazione, per sopperire alle carenze del TCP rispetto ai sistemi Multi-homing, l'applicazione necessita di alcune informazioni supplementari al semplice carico dati. La soluzione a questo problema è l'header. Allo stato attuale del protocollo esso è composto da 3 short (6 byte, su tutti i sistemi supportati) che vengono interpretati dagli algoritmi del programma come segue:

1) campo Type: indica il contenuto del pacchetto. Può essere:
   a) DATA. Contiene del carico utile;
   b) ACK. Conferma la ricezione di un pacchetto con carico utile;
   c) QUIT. Segnala al processo ricevente di terminare;
   d) DIE. Segnala la caduta di un canale, specificando nel corpo dati di quale canale si tratta[3];

2) campo Serial: contiene il numero seriale che identifica il pacchetto. E' utile al PR per inviare idoneamente i pacchetti DATA al Server, e al PS per capire a quali pacchetti si riferiscono gli ACK ricevuti. Questo campo è settato a 0 per gli altri tipi di pacchetto. Il numero seriale è un insieme modulo, per cui possiede valori circolari (min, min+1, …, max, max+1 = min);
3) campo Length: indica la dimensione complessiva (header + data) del pacchetto, espressa in byte.

Di seguito verranno esposti nel dettaglio gli step di funzionamento dei due applicativi PS e PR, già acc

STRUTTURA DEL PACCHETTO DEL PROTOCOLLO



Lo schema sopra riportato rappresenta il pacchetto usato dal protocollo. Le sezioni in giallo sono l'Header del pacchetto mentre la sezione rossa rappresenta l'eventuale carico di dati.Da come si deduce dal grafico dunque la dimenzione minima di un pacchetto e' 6 bytes mentre quella massima di 1024 (6 di header + 1018 di carico dati)

Types disponibili

```
#define DATA    1
#define QUIT    2
#define DIE     3
#define ACK     4
```

3.1 Protocollo

Il core del programma, per quanto concerne l'analisi della rete ed il conseguente bilanciamento del carico dati sui tre canali, è implementato nel Proxy Sender. Il Proxy Receiver si limita a svolgere i compiti già sufficientemente illustrati nella sezione relativa alla progettazione (lettura dati dalla rete, risposta con ACK, invio dei dati al Server). Passiamo dunque ad analizzare nello specifico il comportamento del Proxy Sender.

Si può idealmente dividere il programma in due entità: la prima che svolge del lavoro (lettura dal client, lettura degli ACK dalla rete, scrittura verso la rete), la seconda che analizza le condizioni globali di funzionamento (quanti dati sono ancora da smaltire, quanti aspettano ancora ACK e con quale ritardo, ecc.).

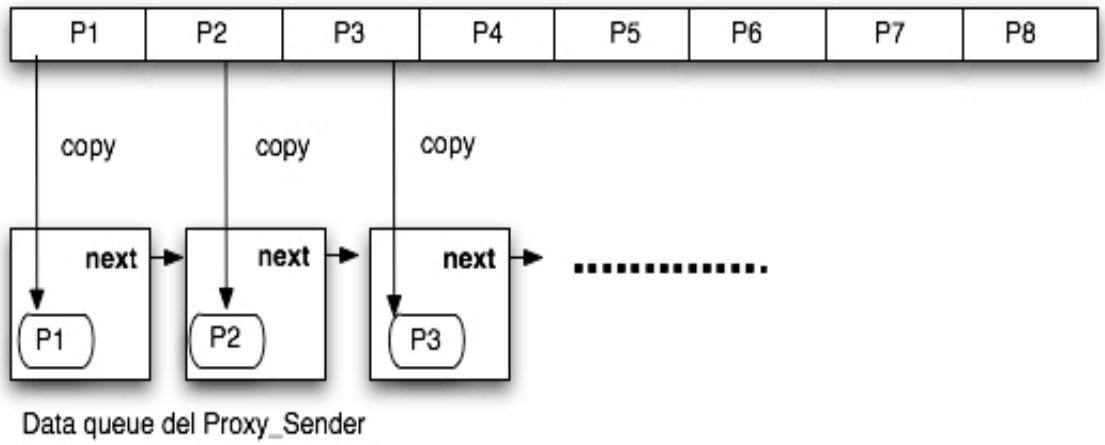
Questa fase di analisi, prepara delle flag booleane che influenzano il comportamento della fase di lavoro (i tre step sotto esplicati) e la scelta dei parametri da passare alla select(). Segue l'analisi di tutti gli step del protocollo.

Proxy Sender->step1 : nel momento in cui la select() si sveglia e se l'fd del Client è attivo, viene eseguita la funzione read_data(). Essa prende due argomenti: un puntatore a struttura channel e un puntatore a struttura queue. Inizialmente legge dal fd del Client fino ad un massimo di 4KB e dispone i byte letti in un vettore; successivamente esegue delle push nella coda a gruppi di massimo 512 byte, fino a quando non esaurisce la quantità letta. Ritorna un puntatore alla coda aggiornata.

FUNZIONE READ_DATA()

La funzione read_data() viene usata dal Proxy_sender per leggere dati dal client. La funzione legge dal client con una read() fino a un massino di 4096 byte e dispone i byte letti in una coda aggiungendo segmenti grandi al massimo 512 byte.Ritorna il puntatore alla coda con le nuove allocazioni

Vettore temporaneo della funzione read_data() in questa rappresentazione ogni casella sono 512 byte quindi 512 caselle di array



Data queue del Proxy_Sender

Nel caso in cui il canale cada la struttura canale viene aggiornata con la flag OFF, se ci sono errori invece la funzione termina con un e_fatal().La funzione considera scontato il fatto che sta facendo operazioni con socket bloccante , quindi non gestisce in alcun modo EAGAIN

Proxy Sender->step2 : quando la select() si sveglia verifica ad uno ad uno gli fd dei canali up e, per ogni fd attivo, viene eseguita la funzione read_ack(). Essa prende come argomenti tre puntatori: alla struttura canale

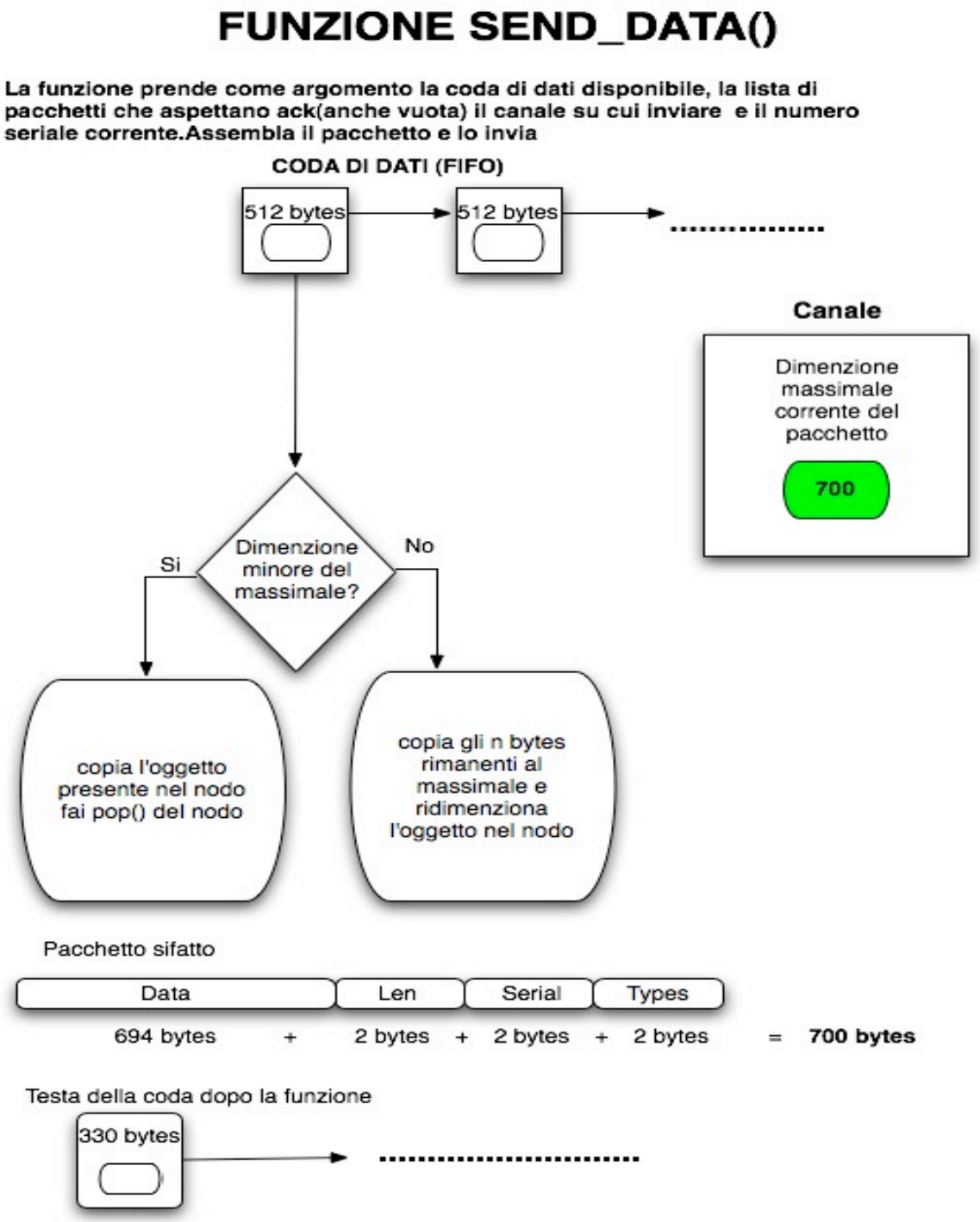
corrente, alla lista dei pacchetti che attendono ACK e un puntatore ad intero che rappresenta una flag. Tale flag consente o meno alla funzione di modificare l'MTU del canale. Tale funzione legge categoricamente 6 byte (socket bloccante), verifica che tali dati siano un pacchetto ACK e li analizza. Il seriale dell'ACK viene ricercato nella lista passata come argomento. Se il seriale non viene trovato, ritorna il puntatore alla lista ed esce. Altrimenti, se la flag passata lo permette, calcola il RTT (differenza fra l'istante di invio e quello corrente). A questo punto in base al RTT misurato la MTU del canale viene aumentata, decrementata o lasciata invariata. Se il RTT è maggiore di 1 secondo, la flag "ack_lock" del canale viene settata ad OFF. L'entità della modifica della MTU varia a seconda della strategia di bilanciamento adottata. Successivamente il pacchetto che è stato confermato viene cancellato dalla lista di attesa e la funzione ritorna il puntatore a tale lista aggiornata.

FUNZIONE READ_ACK()

La funzione read_ack() prende come argomento la lista dei pacchetti gia' inviati e un canale.E' incaricata di leggere gli ack inviati dal Proxy_Receiver e ,valutando il tempo di RTT, modifica la grandezza massima di un pacchetto per quel canale

Proxy Sender->step3 : questo step viene saltato se la flag globale "mywrite" è settata ad OFF. In caso

contrario se la coda di invio contiene dei dati, per ogni canale up (se l'apposita flag locale "success" permette la scrittura) viene eseguita la funzione send_data(). Essa prende come argomenti cinque puntatori: alla coda di invio, alla lista di pacchetti che attendono ack, alla struttura canale corrente, al seriale corrente e ad un intero per memorizzare eventuali errori. La funzione, per ogni canale, esegue delle pop dalla coda di invio fino a raggiungere una quantità di byte minore o uguale all'MTU del canale. Se la quantità è inferiore all'MTU è perché evidentemente la coda dati è stata esaurita. Fatto questo, i dati poppati vengono assemblati in un pacchetto con un header appropriato e inviati sul canale. La funzione ritorna il puntatore alla coda di invio aggiornata.



Strategie di bilanciamento (Porxy Sender->step4) : l'utente può definire, a tempo di compilazione, la

strategia utilizzata dal programma per bilanciare il carico dati, scegliendo fra sei diverse varianti. Infatti dopo i primi tre step (e parzialmente nella funzione read_ack()) il comportamento del programma viene influenzato da queste diverse strategie nelle modalità in cui modifica alcune flag di permessi (relative alla select()) e memorizza il valore di congestione. Per comodità ci riferiremo a queste diverse strategie con i nomi "A", "B", "C", "D", "E" ed "F".

Una volta terminato il lavoro da svolgere (read e write) il PS analizza le condizioni generali della rete, basandosi sui rilevamenti fatti fino a questo punto. Ecco in sintesi i passi di questa operazione e le differenze fra le varie strategie.
In primo luogo il PS controlla se ci sono pacchetti che attendono ACK e, in caso positivo, prende il primo pacchetto e verifica che il tempo trascorso dal suo invio sia inferiore a 0.9 secondi:

1) Tempo trascorso <= 0.9 secondi

    a) La flag "myread" (lettura verso il Client) verrà settata ad ON. Nel caso siano adottate le strategie B, C o D, viene settata ad ON anche la flag "ack_lock" (permesso di modificare la MTU) relativa al canale su cui era stato inviato il pacchetto preso in considerazione. Ricordiamo che tale flag veniva (indipendentemente dalla strategia) settata ad OFF nello step 2, se il RTT di un pacchetto è maggiore o uguale ad 1 secondo.
    b) L'intero "congestioned" (che rappresenta il valore di congestione globale) viene calcolato in questo punto del programma, sommando la dimensione dei pacchetti in circolo sulla rete e degli ipotetici rispettivi ACK. Le strategie E ed F aggiornano il valore di congestione solo se la flag "ack_lock" del canale è ad ON. Questo perché tali strategie considerano il canale probabilmente già congestionato, poiché già un pacchetto è arrivato con un RTT maggiore ad 1 secondo ("ack_lock" = OFF).
    c) La flag "success" (permesso di scrittura sul canale preso in considerazione) viene settata ad ON (indipendentemente dalla strategia adottata).

2) Tempo trascorso > 0.9 secondi
    a) In tutte le strategie, eccezion fatta per A, la flag "ack_lock" viene settata ad ON e quella "success" del canale ad OFF.
    b) Indipendentemente dalla strategia, se il tempo trascorso è maggiore di 9 secondi, il canale viene chiuso perché considerato in crash[4].

Nel caso in cui non ci siano pacchetti che attendono ACK, le flag "success" e quelle "ack_lock" di ogni canale vengono settate ad ON, eccezion fatta per la strategia A, in cui le "ack_lock" non vengono modificate.

Apriamo una parentesi. Come già accennato in precedenza, nello step 2 (funzione read_ack()), a seconda della strategia utilizzata varia l'entità delle modifiche sulle MTU dei canali. Il comportamento generico prevede infatti che se il RTT è inferiore a 0.8 secondi, la MTU del canale viene incrementata di 64 byte (fino al massimale di protocollo, 1024 B). In caso contrario (RTT > 0.8 secondi) viene decrementato di 64 byte (fino al minimale di 64 B), fatta eccezione per la strategia E che decrementa di 128 byte.
Il comportamento della strategia D in questo frangente è del tutto differente. Nel caso in cui il RTT sia inferiore a 0.8 secondi e la MTU è inferiore a 600 B, la MTU viene raddoppiata. Se invece è maggiore o uguale a 600 B viene incrementata di soli 64 byte. Nel caso in cui invece il RTT sia maggiore di 0.8 secondi, la MTU del canale viene ridotta di 64 byte moltiplicati per i secondi di ritardo (eventualmente 0).

Una volta terminati i controlli di cui sopra, non resta che considerare la quantità totale di dati ancora da smaltire. A questo punto si confronta il valore "congestioned" con il totale di byte presenti nella coda di invio e nella coda dei pacchetti ancora non confermati da ACK. Se quest'ultimo valore è maggiore del valore di congestione, la flag "myread" (permesso di lettura dal Client) viene settata ad OFF. Se tutte le flag "success" dei canali sono OFF, "mywrite" sarà anch'essa OFF.

In questo frangente, la strategia B esegue del lavoro addizionale: per ogni canale con la flag "success" ad OFF, divide il valore "congestioned" per 3 (se tutti i canali fossero OFF, dividerebbe per 27). Questo influenzerà il round successivo nel caso in cui la coda dei pacchetti in attesa di ACK fosse vuota, ovvero nel caso in cui il valore "congestioned" non venisse ricalcolato.

A questo punto il cerchio si chiude. Grazie a queste valutazioni, le flag “myread” e “mywrite”, influenzeranno il successivo ciclo decidendo se far aggiungere o meno rispettivamente gli fd di lettura e quelli di scrittura alla select(). I singoli canali saranno invece influenzati dalle flag “ack_lock” e “success” che permettono (o negano) rispettivamente la modifica del MTU e la scrittura.

3.3 Portabilità, debug e gestione errori

Come richiesto dalle specifiche, nella realizzazione del progetto è stato dato un occhio di riguardo alla portabilità. Allo stato attuale del lavoro, si suppone che il codice compili e venga eseguito con successo su tutti i sistemi conformi agli standard POSIX 2001 e ANSI C89. Come già accennato in precedenza, il programma responsabile della portabilità è il Proxy_Configure (PC) che, dopo aver rilevato le informazioni salienti riguardo all’architettura, alla conformità allo standard posix e al sistema operativo utilizzato, scrive il file Proxy_config.h contenente la definizione di costanti utili alla compilazione condizionale del resto del codice.

Per prima cosa PC rileva il modo in cui il sistema operativo in uso interpreta l’architettura utilizzata (32 o 64 bit, lp32 o ilp32, lp64 o ilp64). Successivamente viene verificato che la CPU in uso sia BigEndian; la rete utilizza infatti questo tipo di endianess e quindi se il computer in uso avesse la stessa endianess potrebbe evitare di effettuare conversioni in determinati punti del codice. Per terza cosa il PC si accerta (tramite l’header unistd.h) della conformità del compilatore in uso allo standard ANSI C89 e la conformità del sistema operativo allo standard POSIX2001. Come ultimo step, viene effettuato il parse delle flag passate al programma dall’utente che, allo stato attuale del lavoro, specificano il sistema operativo in uso (BSD, LINUX, OSX), la strategia di bilanciamento da utilizzare (A, B, C, D, E o F) e opzionalmente abilitano il debug (anche se la flag DEBUG è già prevista dal gcc). Esiste un insieme di flag addizionali che non sono state inserite nel codice del progetto come passi condizionali della compilazione, ma sono state comunque contemplate nel PC in previsione di possibili sviluppi futuri (WIN, CRYPT, LOG_DEBUG).

Per agevolare la fase di coding del progetto, come già accennato, il modulo Message.h fornisce anche la definizione di funzioni per la messaggistica degli errori. La gerarchia di errore gestita comprende due livelli:

1) Warning: stampa il messaggio di warning passato dal programmatore e, se viene passato un valore di errno, stampa anche la stringa corrispondente a quel valore. Tale funzione si limita a notificare l’errore, ma non causa l’uscita forzata dal programma.
2) Fatal: come warning, ma causa l’uscita forzata dal programma tramite l’utilizzo della funzione exit().

4
Test

In fase di sviluppo i test sono stati effettuati prevalentemente su:

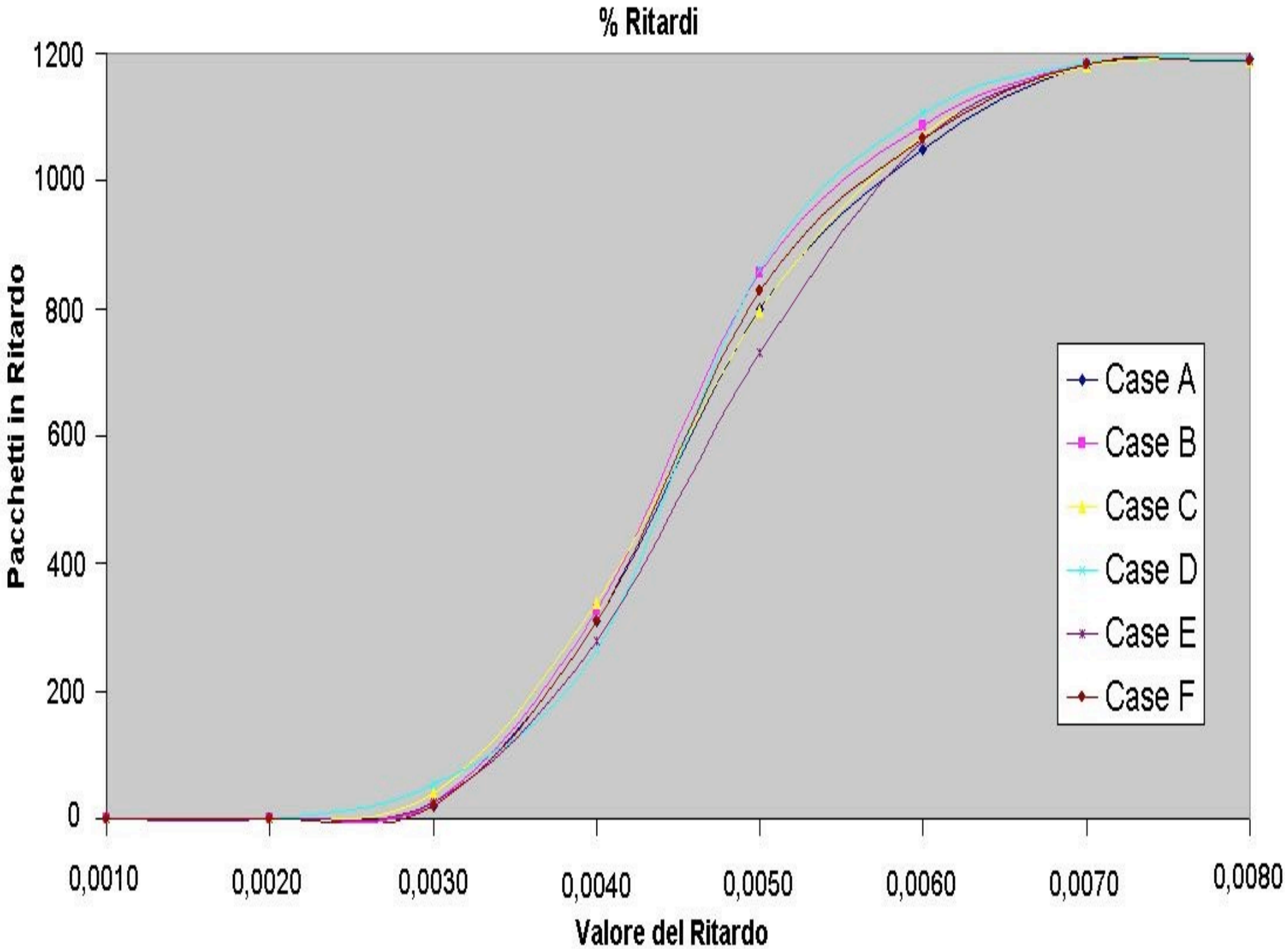
- Intel Pentium III @ 1000 Mhz 128MB RAM, OS linux Debian “lenny“ (testing), kernel 2.6.21.
- PPC G4 @ 1,42 Ghz 1GB RAM, OS Gentoo Linux, kernel 2.6.21, MAC OSX 10.4 e 10.5.

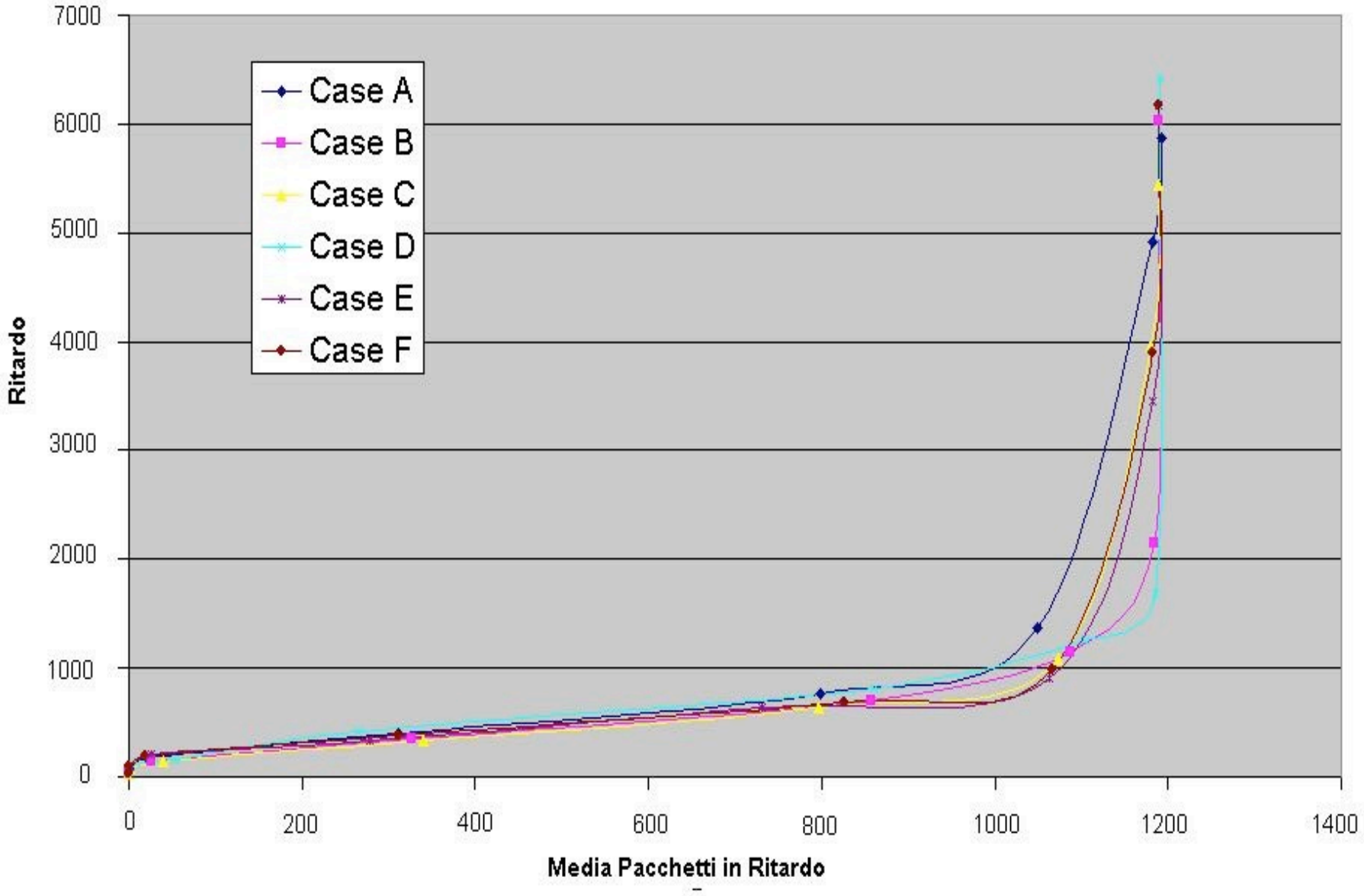
Il programma è stato inoltre compilato con successo tramite emulazione di architettura i386 su Qemu, sui sistemi NetBsd 3.1 e OpenBsd 4.3.

Per il funzionamento sui vari sistemi basati su Bsd (incluso OSX) è stato necessario modificare la dimensione del buffer passato mediante l’opzione SO_SNDBUFFER. Questo perché in tali sistemi, a differenza dello standard, la dimensione che viene passata non viene raddoppiata, ma presa così com’è. Inoltre nei programmi di test (Client e Sender) è necessario ridurre la dimensione dei buffer dei socket creati, poiché tale dimensione (1000000L) non è supportata in questi sistemi.

E’ stato inoltre necessario correggere il file Util.c riguardo la funzione sendn() modificando il nome della flag MSG_NOSIGNAL in MSG_HAVEMORE affinché i programmi di test funzionassero correttamente sui sistemi OSX[5].

Per quanto riguarda le prestazioni del sistema Proxy, leggermente differenti in base alla strategia di bilanciamento utilizzata, si è proceduto eseguendo 10 round di test a scopo statistico, ripetuti per varie percentuali di ritardo del programma di simulazione della rete (Ritardatore). E’ ragionevole presumere, sebbene tutte le strategie presentino risultati molto simili fra di loro, che un programma finale basato su questo core necessiti di statistiche più accurate sulla rete reale su cui dovrebbe funzionare. Ogni strategia di bilanciamento presenta infatti vantaggi e svantaggi in base all’entità del ritardo e soprattutto a quanto tale ritardo varia nel tempo (connessione più o meno stabile). Per un ipotetico utilizzo reale, si consiglia quindi di eseguire test con diversi tipi di simulatori di rete.


% Ritardi
Pacchetti in Ritardo
1200
1000
800
600
400
200
0
0,0010
0,0020
0,0030
0,0040
0,0050
0,0060
0,0070
0,0080
Valore del Ritardo
Case A
Case B
Case C
Case D
Case E
Case F


Case A
Case B
Case C
Case D
Case E
Case F
Ritardo
7000
6000
5000
4000
3000
2000
1000
0
0
200
400
600
800
1000
1200
1400
Media Pacchetti in Ritardo

5
Limiti e possibili migliorie

E' stato supposto, per vincoli di progetto, che la latenza della rete sia al massimo di 500 ms per un funzionamento corretto del programma. Per motivi pratici è stata trascurata l'eventuale funzione "delete" per alcune strutture dati contenenti puntatori (ad esempio i_packet). Per queste strutture, in caso di rimozione, ci siamo accontentati di posizionare il relativo puntatore a NULL, senza effettivamente occuparci di pulire la memoria allocata. Questo fa in modo che non ci siano puntatori pendenti, anche se lascia sezioni di heap sporche e inutilizzate. Sono stati riscontrati alcuni problemi nella gestione della caduta delle connessioni, ancora non risolti.

Alcuni interessanti miglioramenti e ampliamenti del programma potrebbero essere: un sistema di crittografia dei pacchetti, un orologio di applicazione che sincronizzi il tempo sui due differenti host dove il PS e il PR girano (utile per avere calcoli della latenza più precisi), fornire un'interfaccia completa e ampliata per la configurazione, permettere la gestione di più Client, ottimizzare il metodo di lettura del PS rendendolo simile a quello del PR per guadagnare in prestazioni.

NOTE:

1. il tipo di simulatore di rete utilizzato, potrebbe alterare il valore del gap durante il test.
2. Sarebbe possibile gestire più di un client utilizzando il byte di disavanzo del campo type nello header (o aggiungendo un campo supplementare) come identificativo del client da cui proviene il dato.
3. in realtà il type DIE non è ancora stato utilizzato
4. questo timeout di 9 secondi è stato valutato in base al codice del Ritardatore, mentre in un'applicazione reale, a nostro avviso, dovrebbe aggirarsi intorno ai 120 secondi.
5. Si ringrazia a tal proposito la documentazione presente su http://www.developer.apple.com/