Documentazione del Progetto: "trasporto affidabile multi-percorso"

per Reti di Calcolatori a.a. 2011/2012

Componenti del gruppo

- Mattia D'Ambrosio matricola: 0000589696
- Riccardo Serafini matricola: 0000593281

Finalità e scenario generale del progetto

Si vuole simulare uno scambio di dati tra due programmi, attraverso una rete inaffidabile che scarta e ritarda i pacchetti transitanti.
Questa funzione viene ricoperta da un programma *ritardatore,* attraverso il quale passano tutti i pacchetti.
Il ritardatore comunica con due host diversi, in particolare con due programmi proxy, *proxysender* e *proxyreceiver* che inoltrano il flusso TCP attraverso dei datagram UDP.

Ci siamo occupati di implementare *proxysender, proxyreceiver* ed un protocollo che garantisca l'arrivo ordinato dei pacchetti al *receiver*.
Il progetto è stato realizzato secondo le specifiche fornite dal professore Vittorio Ghini disponibili all'indirizzo:
http://www.cs.unibo.it/~ghini/didattica/reti_lpr/progetti/aa2011-12/SpecificheProgettoLabReti_aa11-12.pdf

Protocollo implementato

Tra *proxysender* e *proxyreceiver* viene implementato un protocollo UDP che permette di trasportare un flusso TCP tra i due host (*sender* e *receiver*) mantenendolo invariato.
Il *ritardatore* scarta i pacchetti e li ritarda in modo casuale, a volte mandando una notifica, a volte senza avvisare il processo mittente. Il protocollo UDP deve rimediare a queste perdite, cioé deve essere in grado di capire se un pacchetto non é arrivato a destinazione ed eventualmente inviarlo di nuovo.
I datagram UDP, inoltre, possono arrivare per loro natura in ordine diverso da quello di invio, quindi il protocollo deve anche prevedere il riordinamento dei pacchetti, in modo da inviarli nella giusta sequenza al *receiver*.

I datagram UDP hanno 5 byte di header in cui sono contenute delle informazioni relative al pacchetto:

- 4 byte contenenti un intero senza segno in endianess di rete che rappresenta **l'ID** univoco del pacchetto
- Un byte contenente un carattere che indica il **tipo** del pacchetto: 'B' rappresenta un pacchetto di dati, 'I' rappresenta un pacchetto ICMP, uno speciale pacchetto generato dal ritardatore che notifica la perdita di un datagram.

Dopo l'header é presente una stringa di dati di dimensione indefinita.

Apertura del protocollo

Non abbiamo implementato una vera e propria apertura del protocollo UDP, semplicemente questo si avvia nel momento in cui viene trasferito il primo datagram valido con ID = 1.

Trasferimento dei dati

Man mano che il *proxysender* legge i dati TCP dal sender, incapsula le stringhe di dimensione fissata che legge nell'header appena descritto. Il primo pacchetto ha id 1, e questo viene incrementato di uno ad ogni pacchetto letto, in modo che ad ogni datagram corrisponda un ID univoco.
Una volta formato, il *proxysender* invia il datagram ad una delle tre porte del *ritardatore* e lo aggiunge in coda alla lista dei pacchetti che devono essere confermati. Per garantire l'invio di tutti i pacchetti nonostante le perdite, infatti, e' necessario che il *proxyreceiver* confermi la ricezione di ogni datagram. La porta a cui il *proxysender* invia il datagram cambia ad ogni ciclo, per distribuire uniformemente il carico sulla rete.

Il *proxyreceiver* legge i datagram e li aggiunge ad una lista ordinata in base all'ID dei pacchetti. In testa alla lista c'é il pacchetto con ID minore, in coda alla lista quello con ID maggiore.
Quando viene ricevuto il datagram, inoltre, il *proxyreceiver* invia immediatamente un pacchetto di acknowledgement al *ritardatore*.
L'ACK e' un pacchetto di 6 byte: i primi 4 sono l'ID del pacchetto che deve essere confermato, il quinto byte e' il carattere 'B' e il sesto byte e' il primo carattere della stringa di dati del pacchetto da confermare.
L'ACK viene rispedito indietro dalla stessa porta del *ritardatore* da cui e' arrivato, perché ci sono maggiori possibilitá che questa non sia in BURST.
Il *proxyreceiver* mantiene una variabile con l'ID che sta attendendo per poter inviare nuovi dati al receiver. Se riceve un datagram con quell'ID, scorre la lista dalla testa ed invia tutti i pacchetti che trova con ID consecutivo. Appena trova un pacchetto con ID non successivo del precedente, si interrompe e memorizza tale ID nella variabile.
Se al *proxyreceiver* arrivano pacchetti duplicati o già inviati, questi vengono semplicemente scartati, così come se al *proxysender* arriva un ACK per un pacchetto già confermato.
Quando al *proxysender* arrivano gli ACK dei pacchetti, questi vengono semplicemente rimossi dalla lista. Se un pacchetto non viene confermato entro un tempo TIMEOUT, questo viene inviato nuovamente dal *proxysender* al *ritardatore* e re-inserito in coda alla lista.
Quando arriva un ICMP al *proxysender*, questo preleva il pacchetto corrispondente dalla lista, lo invia di nuovo e lo riaggiunge in fondo alla lista.
Quando arriva un ICMP al *proxyreceiver* viene inviato indietro un nuovo ACK per quel pacchetto.
Questa procedura si ripete in modo iterativo per un tempo indefinito, fino a quando il *sender* chiude la connessione TCP con il *proxysender* e la lista dei pacchetti da confermare nel *proxysender* si svuota completamente.
A questo punto si avvia il protocollo di chiusura UDP.

Protocollo di chiusura
Questo protocollo si avvia quando tutti i pacchetti sono stati trasferiti, e serve per informare il *proxyreceiver* che puo' chiudere la connessione TCP con il *receiver* e terminare l'esecuzione.
Senza questo scambio finale, infatti, il *proxyreceiver* non avrebbe modo di sapere se tutti i pacchetti sono stati inviati correttamente. Il *proxysender* avvia un handshake di chiusura inviando un datagram di 6 byte, con ID=0, tipo 'B' e con un carattere '1' dopo l'header. Il *proxyreceiver* risponde a questo datagram con un ACK, come al solito. Se questi pacchetti vengono scartati dal ritardatore, vengono inviati nuovamente come accade durante l'invio dei dati, finché al *proxysender* non arriva l'ACK.
A questo punto il *proxysender* invia un secondo pacchetto di chiusura, con ID=0, tipo 'B' e con un carattere '2' dopo l'header. Il *proxysender* ora sa che il *proxyreceiver* sa che deve chiudere l'esecuzione, quindi può terminare lui stesso l'esecuzione.
Questo ultimo pacchetto non deve essere confermato. Se il *proxyreceiver* lo riceve si chiude immediatamente, non prima di aver terminato l'invio di tutti i dati al *receiver*, altrimenti attende CLOSETIMEOUT secondi e termina la sua esecuzione, chiudendo la connessione TCP con il *receiver*.

Documentazione tecnica
Il linguaggio di programmazione utilizzato per realizzare questo progetto, è "ANSI C", compilato con i flag "*-ansi -pedantic -Wall -Wunused*".
Abbiamo implementato *proxysender* e *proxyreceiver* in 4 files:

- **"proxysender.c"** contiene il main di esecuzione del proxysender
- **"proxyreceiver.c"** contiene il main di esecuzione del proxyreceiver
- **"utils.c"** contiene le procedure utilizzate da tutti e due i programmi
- **"utils.h"** contiene i prototipi delle funzioni, le strutture e le costanti del progetto

Abbiamo deciso di gestire le connessioni senza usare i thread, ma solo con l'uso di I/O Multiplexing (select()).

Diagramma di flusso del proxyrsender

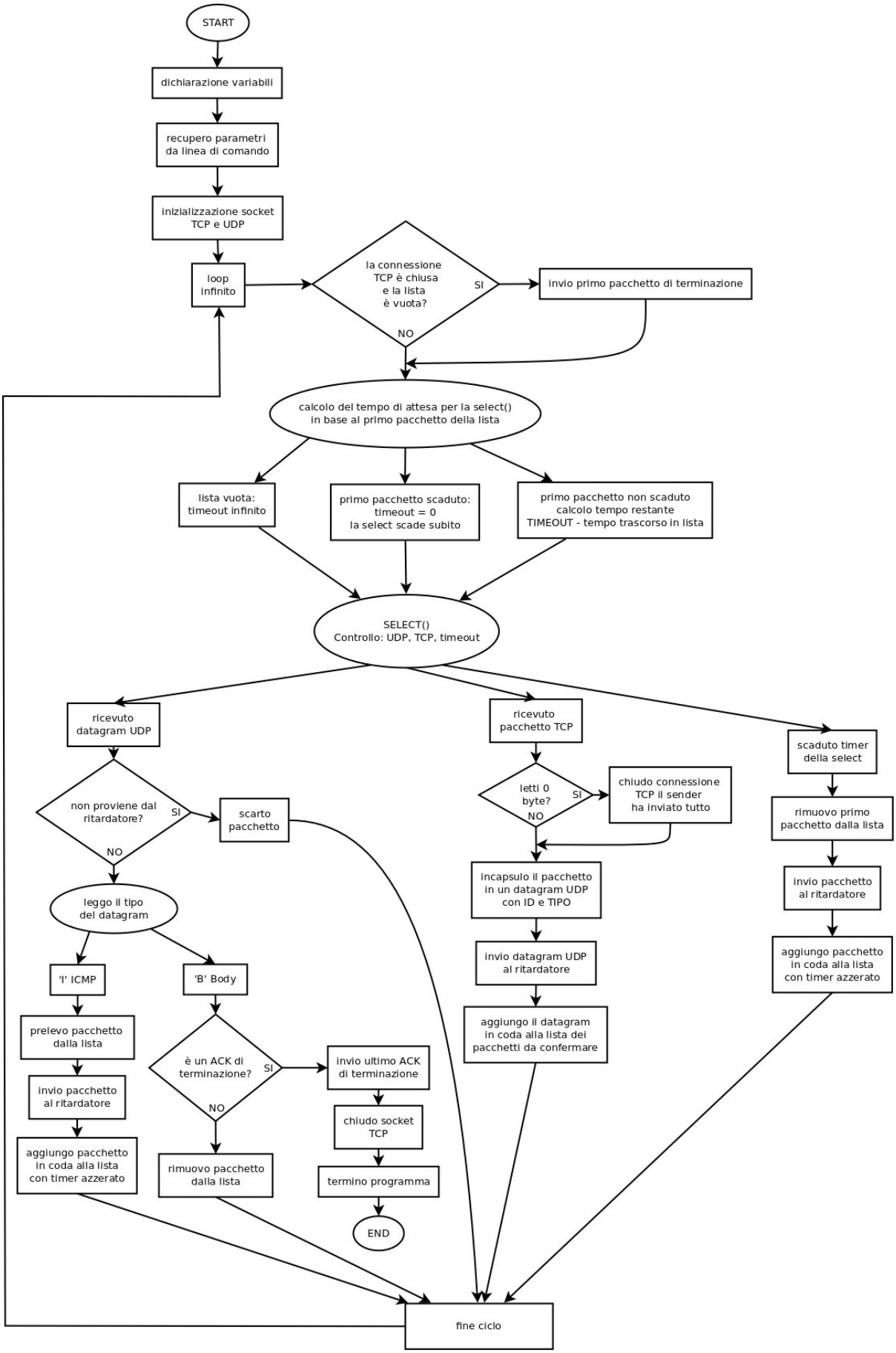
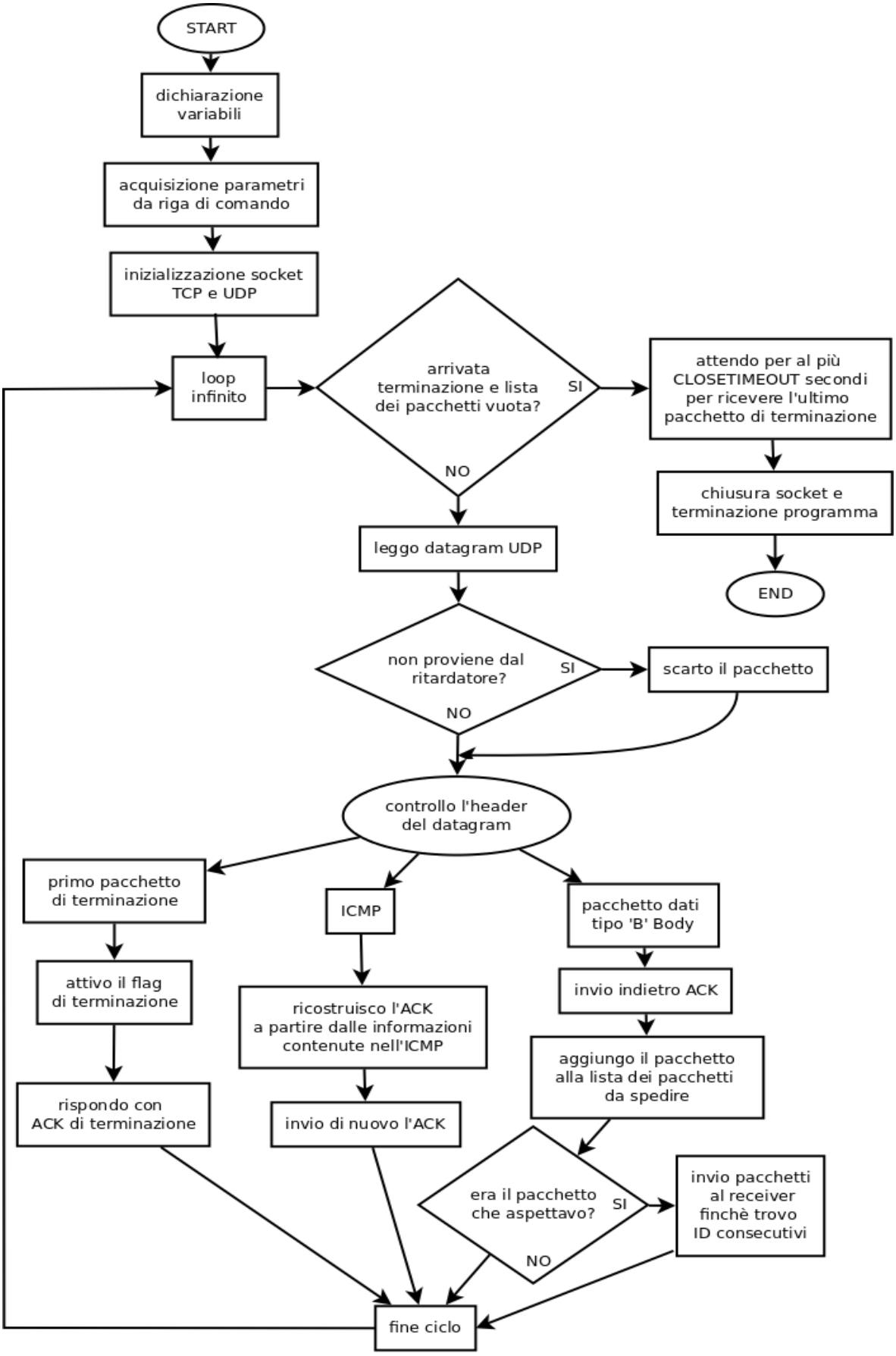


Diagramma di flusso del proxyreceiver



Utility

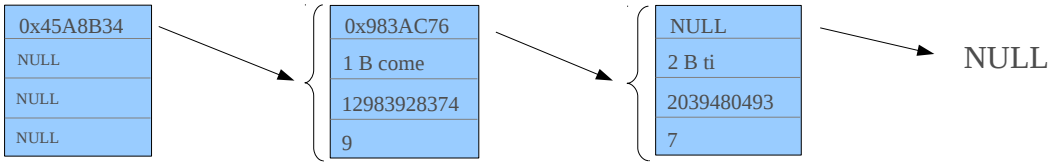
Nei files "**utils.c**" e **"utils.h"** abbiamo racchiuso le dichiarazioni di strutture, costanti e funzioni utilizzate da tutti e due i programmi proxy, in particolare:

Strutture, liste dinamiche

Abbiamo dichiarato la struttura **lista** per implementare delle liste dinamiche con *malloc()* e *free()*.
La struttura è così definita:

```
typedef struct lista{
    struct lista* next;          /* il puntatore all'elemento successivo */
    packet p;                    /* il pacchetto */
    struct timeval sentime;      /* il timestamp abbinato al pacchetto */
    int size;                    /* la dimensione del pacchetto */
} __attribute__((packed)) lista;
```

Si tratta di una lista con puntatore all'elemento successivo non circolare con sentinella, la sentinella è rappresentata da una struttura **lista** di cui si utilizza solo il campo **next**.



La lista è implementata come una coda, con inserimento in coda (push) ed eliminazione in testa (pop), ma è possibile anche inserire un elemento in ordine a seconda dell'ID e rimuovere un elemento con un determinato ID.

Le procedure realizzate sono le seguenti:

```
void aggiungi( lista* sentinella, packet p, int size );
void aggiungi_in_ordine( lista* sentinella, packet p, int size );
lista pop( lista* sentinella);
lista rimuovi( lista* sentinella, uint32_t id);
```

Queste funzioni aggiornano la lista allocando e deallcoando memoria con *malloc(),* e *free()* e aggiornando i puntatori nel modo corretto.

Abbiamo inoltre dichiarato altre due strutture: **packet** e **ICMP**, due contenitori per ingabbiare i buffer in lettura e scrittura ed estrarre le informazioni dell'header.
Sono dichiarate nel seguente modo:

```
typedef struct packet{
      /* l'ID del paccheto, intero senza segno a 32 bit */
      uint32_t id;
      /* il carattere che identifica il tipo di pacchetto */
      char tipo;
      /* una stringa per il corpo del pacchetto */
      char body[BODYSIZE];
} __attribute__((packed)) packet;

typedef struct ICMP{
      /* intero senza segno a 32 bit che rappresenta l'id dell'ICMP */
      uint32_t id;
      /* un carattere che rappresenta il tipo 'I' dell'ICMP */
      char tipo;
      /* l'ID del pacchetto scartato */
      uint32_t idpck;
} __attribute__((packed)) ICMP;
```

Costanti
Nel file "**utils.h**" abbiamo racchiuso le costanti del programma. I valori di questi costanti sono stati definiti dopo numerevoli tentativi, per cercare di individurare i valori ottimali in base a come si modificavano le prestazioni. Abbiamo definito queste costanti:

```
/* la dimensione massima dei pacchetti */
#define MAXSIZE 65000

/* la dimensione in byte dell'header dei datagram UDP */
#define HEADERSIZE 5

/* la dimensione massima del body dei pacchetti */
#define BODYSIZE (MAXSIZE) - (HEADERSIZE)

/* il timeout della select del proxysender */
#define TIMEOUT 0   /* secondi */
#define MSTIMEOUT 800000 /* microsecondi */

/* timeout di attesa della chiusura UDP da parte del proxyreceiver */
#define CLOSETIMEOUT 5
```

Funzioni per i socket e I/O di rete
Abbiamo definito in "**utils.c**" delle funzioni per creare socket TCP e UDP, fare bind, mettersi in ascolto, collegarsi e trasferire dati attraverso la rete.
Ad ogni chiamata di systemcall viene controllato il valore di ritorno ed in caso di errore il programma viene terminato stampando a video il tipo di errore.

Per inviare i dati, in particolare, abbiamo implementato la funzione **writen**, che scrive su un socket tutti i dati che si vogliono trasmettere, ripetendo la **send** se questa viene interrotta dal sistema operativo, con errore "EINTR".

Prestazioni del progetto
Per testare la velocità di esecuzione del progetto abbiamo effettuato diverse prove, su tre macchine del laboratorio, lanciando il Ritardatore con lo scarto di pacchetti di default del 15%.

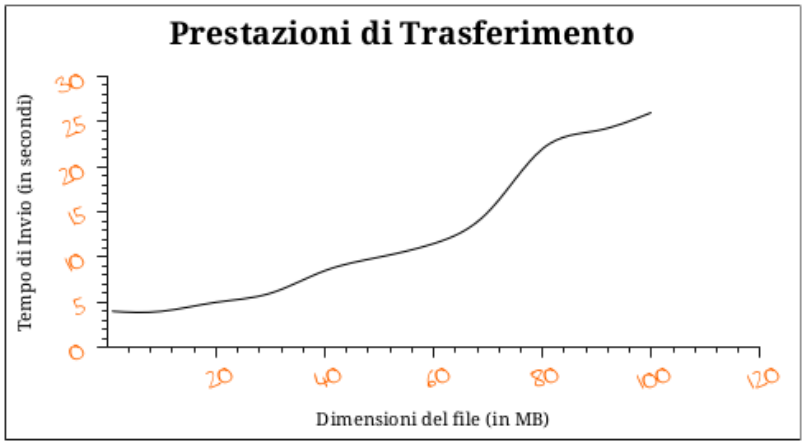
Affidabilitá
Il flusso di dati TCP inviato dal sender, deve essere identico a quello ricevuto dal receiver, i pacchetti devono arrivare tutti e nello stesso ordine.
Per verificare questo requisito abbiamo trasferito dei file e confrontato il fingerprint di questi con le utility *md5sum*, *sha1sum*, *diff* e *cmp*.
In tutte le prove effettuate, il file trasferito si é dimostrato identico a quello inviato.

Latenza
La latenza di base presente tra due macchine del laboratorio, misurata con il comando *ping*, ha un valore medio di 0.475 ms (andata e ritorno dell'ICMP). Considerato che i pacchetti inviati dal sender al receiver effettuano 4 hop, di cui due attraverso Ethernet e due tra processi della stessa macchina, possiamo approssimare il limite inferiore della latenza del progetto a 0.5 ms.
Attraverso l´utility *netcat*, abbiamo utilizzato il progetto come una chat monodirezionale, inviando piccole stringhe di testo. Dal momento in cui si preme ENTER al momento in cui la stringa compare sul terminale del receiver, puó passare un tempo variabile da meno di un secondo a 2 secondi.
I pacchetti che non vengono scartati dal Ritardatore, infatti, arrivano quasi immediatamente, mentre quelli che vengono scartati, vengono ritrasmessi dopo 0.8 secondi ed arrivano in media dopo un paio di secondi.

Uso di CPU
Per misurare l'utilizzo della CPU durante l'esecuzione di un trasferimento, abbiamo utilizzato il programma *htop*, che visualizza le statistiche di tutti i processi del sistema.
Durante il trasferimento di un file di 100 MB, il carico della CPU, misurato nelle macchine del laboratorio è circa del 6% per il proxysender e del 2% per il proxyreceiver.
L' utilizzo della CPU naturalmente dipende dalla quantitá di pacchetti in circolazione, un dato che si può empiricamente dedurre dal numero di pacchetti presenti nelle liste dinamiche del proxysender e del proxyreceiver.

Occupazione di Memoria
Come per l'utilizzo di CPU, abbiamo utilizzato il programma *htop* per verificare l'utilizzo di memoria. Anche questo dipende dalla quantità di pacchetti presenti nelle liste.
Questo valore può essere calcolato semplicemente moltiplicando la dimensione dei pacchetti trasferiti per il numero di pacchetti memorizzati nella lista.
Nel caso del trasferimento di file i pacchetti trasferiti hanno tutti una dimensione simile.
Il proxysender legge N byte alla volta dal sender, e i pacchetti vengono quindi dimensionati secondo questa costante.
Dopo diverse prove, abbiamo impostato questo valore a 65000 byte. Anche se é molto maggiore all' MTU della connessione Ethernet (1500 byte) e quindi si genera frammentazione, abbiamo riscontrato prestazioni decisamente migliori utilizzando questo valore.
In questo caso quindi 100 pacchetti in lista occuperanno circa 6 MB.

Velocitá di trasferimento
Abbiamo provato ad inviare files di diverse dimensioni, misurando il tempo impiegato per trasferirli con il programma *time* e con le statistiche stampate dal *proxysender*.
Come ci aspettavamo, il tempo di trasferimento cresce abbastanza linearmente con la dimensione del file inviato.
Per files piccoli peró, la differenza si nota meno perché il protocollo di chiusura UDP impiega circa sempre lo stesso tempo, indifferentemente dalla dimensione del file trasferito.
Abbiamo rappresentato l'andamento della velocità di trasferimento nel grafico sottostante:



asse X	asse Y
Dimensioni del file (in MB)	Tempo di Invio (secondi)
100	26
90	24
80	22
70	15
60	11.5
50	10
40	8.5
30	6
20	5
10	4
1	4

Conclusioni
Date le considerazioni espresse in questa documentazione, riteniamo di aver implementato con successo il progetto ***trasporto affidabile multi-percorso*** rispettando a pieno le specifiche dettate dal professore.