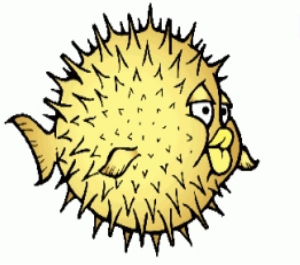
Programmazione di applicazioni Multihoming in ambiente POSIX

Sviluppo di un prototipo in C



```
gcc -o mynet_app mynet_app.c
```

Siamo un gruppo di studenti di Informatica(Bonicolini Matteo ,Gentile Alessandro Dennis Dalla Torre)e come progetto per l'esame di Reti di Calcolatori ci è stato assegnatoun proxy che doveva implementare multihoming mediante lo stack TCP/IP. Poichè il progetto è stato per noi di grande interesse, abbiamo deciso di portarlo oltre il semplice esame universitario.
Siamo intenzionati a mettere il progetto su sourceforge

Scopo del talk

In questo talk si analizza l'implementazione del prototipo realizzato. Gli argomenti trattati sono:

1)Codice del prototipo realizzato(system call, algoritmi,funzioni di libreria ecc...)

2)Sviluppi futuri del prototipo

3)Discussione sul lavoro svolto dal gruppo (sono ben accetti suggerimenti e/o critiche costruttive)

Sarà mantenuto attivo un mini-laboratorio per permettere ai curiosi e agli interessati di smanettare sul prototipo.

Metodo Utilizzato

Per toccare tutti gli argomenti di questo talk si procederà con il seguente approccio:

1) Spolverata di programmazione di rete con il linguaggio C le System Call necessarie per poter sviluppare software di rete in ambiente POSIX

2) Descrizione di alcuni miniprogrammi introduttivi alla programmazione di rete che usano api del prototipo

3) Descrizione vera e propria del codice del prototipo

4) Descrizione degli sviluppi futuri del prototipo e dibattito

File descriptor

1)Il file descriptor è un riferimento astratto per poter manipolare file.

2)Esso è rappresentato da un numero intero (un tipo int del linguaggio C)

3)Ogni processo attivo ha almeno 3 file descriptor aperti(standard input,standard output,standard error).

In ambiente Linux , se il kernel è stato compilato con il filesystem proc, per vedere con i propri occhi quanto qui basta compilare un semplice programma che dorme per n secondi.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int pid=0;

    pid=getpid();
    printf("pid del processo=%d \n",pid);
    fflush(stdout);
    sleep(20);
    return 0;
}

#ls /proc/pid_associato/fd
0 1 2
0=stdin 1=stdout 2=stderr
```

Quando un processo apre un file, il kernel aggiunge un numero(fd) all'elenco dei file descriptor attualmente gestita dal processo corrente.Tale valore numerico è, fino a quando non viene chiuso il file,l' associazione al file stesso quindi un interfaccia di basso livello per le operazioni di I/O su file.

L'altra interfaccia per operare su file è lo stream che è rappresentato da un puntatore a una struttura file(File*).Lo stream è un interfaccia per file ad alto livello.Se si apre un file tramite l'interfaccia degli stream , il kernel associa comunque un fd e lo aggiunge alla lista.
Per gli stream ci sono set di istruzioni che facilitano le operazioni di I/O su file.E' possibile ,nel caso in cui sia necessario eseguire operazioni di basso livello,convertire lo stream nel file descriptor corrispondente(in quanto entrambi indicano il solito file).

```
int main()
{
  int fd=0;
  char buffer[5]="ciao\n";
  int check=0;int pid=0;

  pid=getpid();
  printf("sono il pid %d \n",pid);
  fd=open("./my_file.txt",O_RDWR|O_CREAT,00700);
  sleep(20);
  check=write(fd,buffer,5);
  close(fd);
  printf("ho chiuso il file \n");
  sleep(20);
  return 0;
}
```

Prima di stampare "ho chiuso il file" ,dalla chiamata della open() in poi, il segunte comando restituirà

#ls /proc/pid_number/fd
0 1 2 3

Dopo la chiamata close()

#ls /proc/pid_number/fd
0 1 2

Il programma è semplice(è anche incompleto non avendo controlli) , crea un file(my_file.txt),dorme 20 sec,ci scrive "ciao", chiude il file e dorme altri 20 sec.

#cat my_file.txt
ciao

int open(const char* path,int flags,mode_t mode) :

La funzione open apre o crea un nuovo file.L'argomento path indica il percorso nel filesystem del file, il campo flags indica la modalità tramite il quale si vuole accedere al file(crearne uno nuovo,aprirlo in lettura , aprirlo in scrittura) mentre il campo mode indica i permessi sul file.il ritorno della funzione è il file descriptor corrispondente al file.In caso di errore ritorna -1 e setta opportunamente errno
#man 2 open per delucidazioni

int close(int fd):

La funzione chiude un file descriptor ,ritorna -1 in caso di errore e setta errno in modo opportuno
#man 2 close per delucidazioni

```
#include <stdio.h>
int main()
{
      FILE* myfile=NULL;
      myfile=fopen("./myfile.txt","w+")
      if(myfile==NULL) return 1;

      fprintf(myfile,"ciao \n");
      printf("pid del processo %d \n",getpid());
      printf("file descriptor associato =%d\n",fileno(myfile));
      sleep(20);
      printf("chiudo il file \n");
      fclose(myfile);
      sleep(20);
      myfile=NULL;
      return 0;
}
```

File* fopen(const char* path,const char* mode) :

La funzione fopen apre o crea un nuovo file.L'argomento path indica il percorso nel filesystem del file.
L'argomento mode le modalità con cui aprire il file.La funzione ritorna un puntatore a File(lo stream).In caso di errore la funzione ritorna NULL e setta errno in modo opportuno.

#man 3 fopen per delucidazioni

int fclose(File* stream):

La funzione chiude uno stream ,ritorna -1 in caso di errore e setta errno in modo opportuno.Se va tutto bene ritorna 0

#man 3 fclose per delucidazioni

Il programma esegue le medesime operazioni di quello che operava su file descriptor ma usa le funzioni per gli stream.Se si visiona l'elenco di fd prima di fclose()

#ls /proc/pid_processo/fd
0 1 2 3

Dopo fclose()

#ls /proc/pid_processo/fd
0 1 2

Nel programma viene anche utilizzata la funzione fileno() che prende come input uno stream e ritorna il file descriptor associato.

Ma non era un talk sul multihoming?Perchè si parla di file descriptor?

Nei sistemi POSIX anche i socket sono file descriptor.

La conseguenza è la seguente: è possibile interagire con i socket utilizzando le System Call relative all' I/O dei fd.

Cos'è un socket?

Un socket è un astrazione software(rappresentato per le operazioni di I/O da un fd) con cui il processo trasmette e riceve dati attraverso una rete(anche di processi locali (UNIX domain)).

Informazioni sui socket?

Tramite la funzione open() è possibile aprire un file o crearne uno nuovo.
La medesima funzione per i socket è la socket().

Esistono numerose System Call apposite per fd che descrivono socket(molte sono originarie dei sistemi BSD).Per avere un idea:

#less /usr/include/sys/socket.h
#man delle funzioni presenti nell'header socket.h
#less /usr/src/linux/net/socket.c(o del file equivalente sul proprio kernel)

Caratteristiche salienti dei socket che operano sullo stack TCP/IP

Poichè i socket sono un interfaccia software, le proprietà possedute, sono intrinsecamente connesse con quelle del protocollo che rappresentano.

Nel caso in cui il socket rappresenti una comunicazione avvenuta tramite lo stack TCP/IP(SOCK_STREAM) il kernel identifica il socket come segue:

1)Associa un fd tramite il quale il processo comunica le operazioni di I/O (mediante la funzione socket())

2)Associa una coppia <IP(4 bytes),porta(2 bytes)> (mediante la bind() e la struct sock_addr_in)

Comunicazione con i socket che operano sullo stack TCP/IP

Come ben noto una comunicazione mediante stack TCP/IP richiede almeno due entità: una client e l'altra server.
Da questa impostazione si evincono gli stati possibili in cui si può trovare un socket di questo tipo:

1)listening: il socket è in attesa di una connessione da parte di un client(mediante la system call listen())

2)established: è partita la sessione TCP, client e server possono comunicare sul canale(mediante connect() e accept())

GRAFICO DI "ALTO LIVELLO" DI UNA COMUNICAZIONE TCP

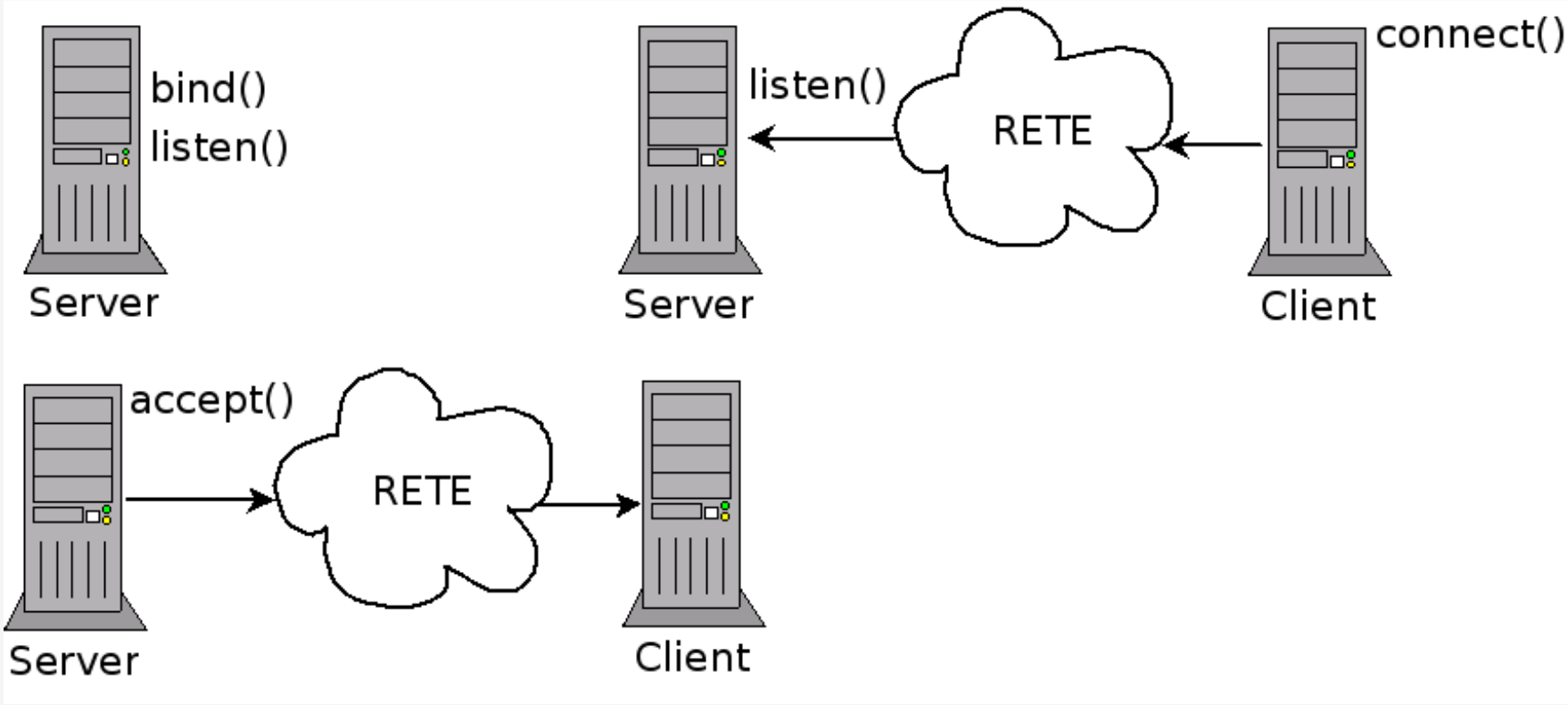


GRAFICO DI BASSO LIVELLO DI UNA COMUNICAZIONE TCP

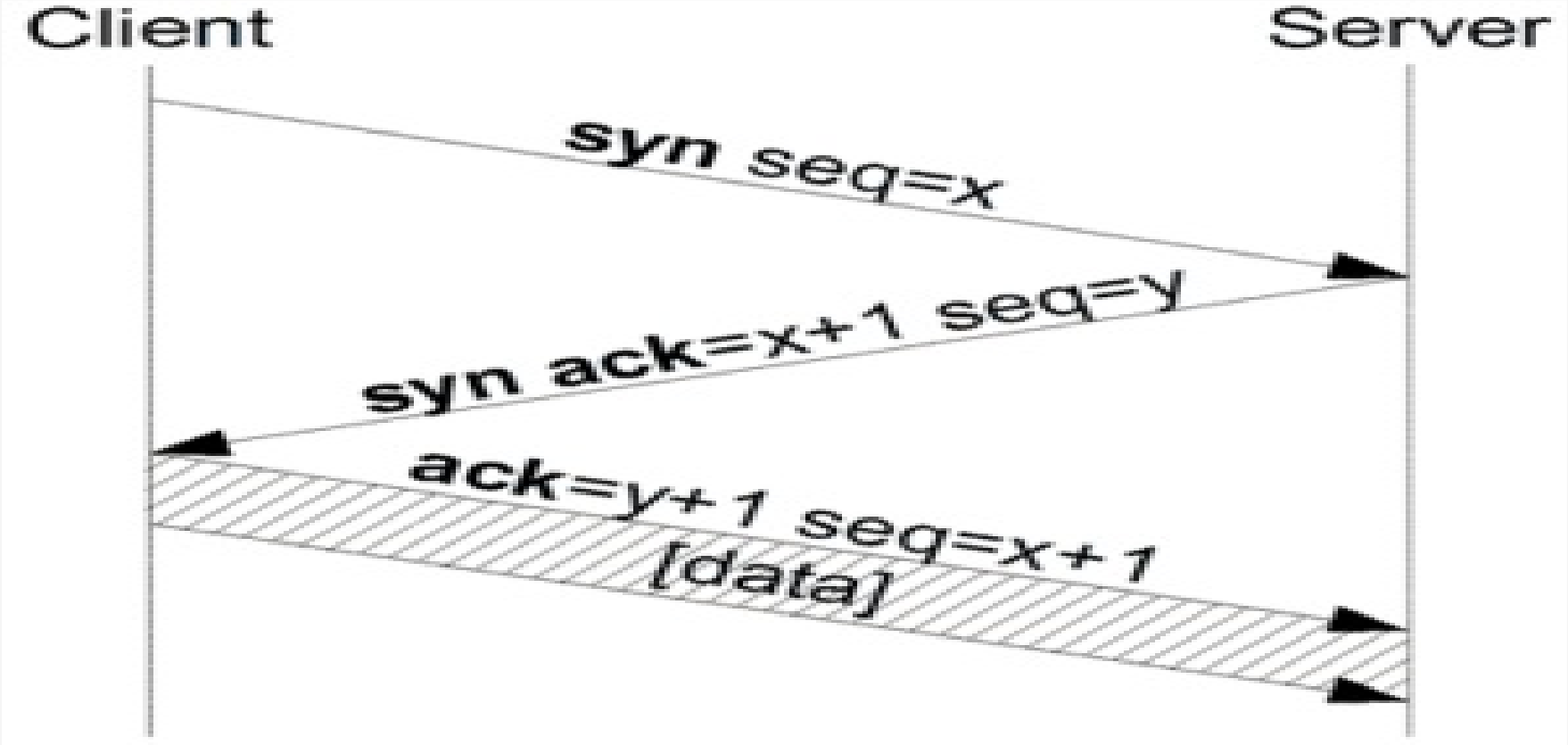


IMMAGINE PRESA DA WIKIPEDIA

L'introduzione alla programmazione di rete finisce qui:

1)L'argomento è talmente ampio che necessita un talk apposito

2)Siamo solo studenti non ci riteniamo capaci di insegnare

3)Si può trovare una quantità di documentazione tendente a infinito

4)Riteniamo quanto detto l'essenziale per avere una percezione funzionale delle API che verranno mostrate nel talk

5)Le API del prototipo potrebbero risolvere alcuni dubbi in quanto le funzioni qui viste vengono utilizzate

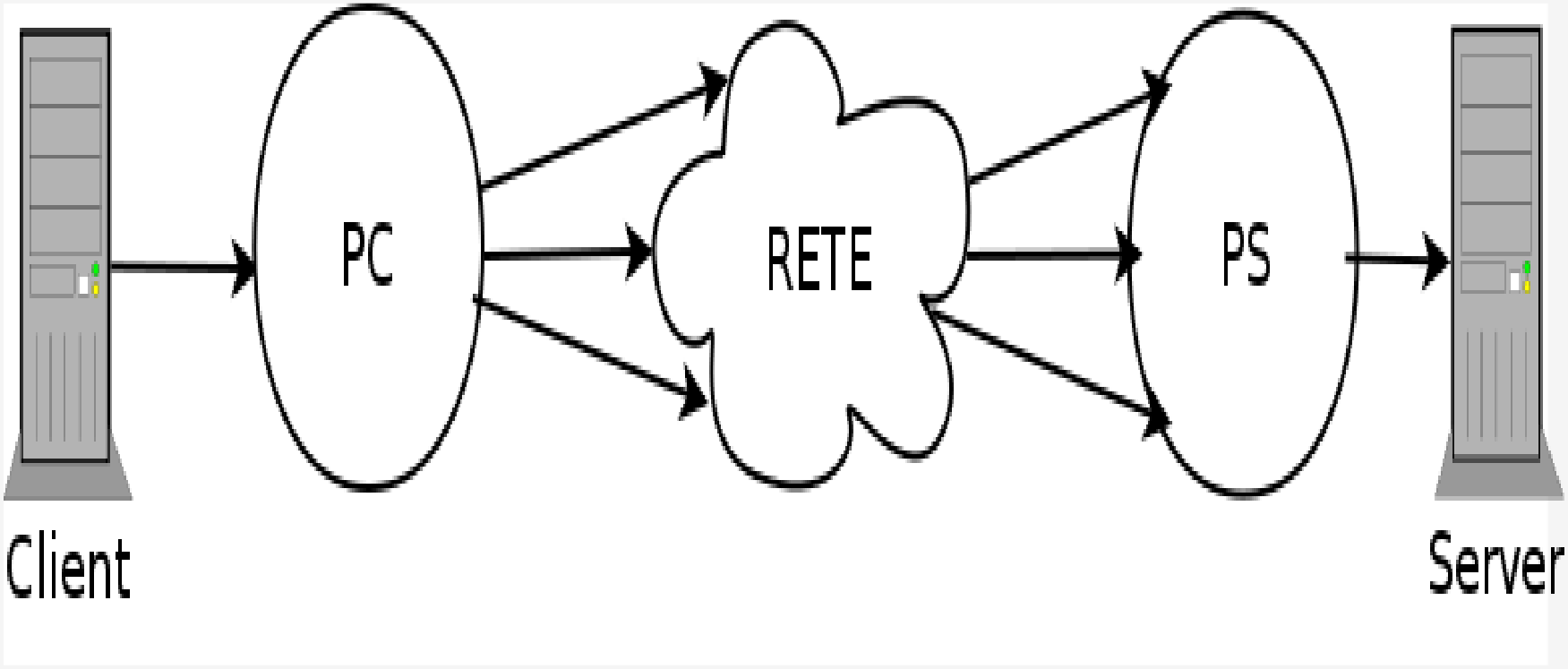
Che tipo di software è stato implementato?

Il gruppo ha implementato un sistema Proxy (quindi un protocollo a livello di applicazione) che implemeta Multihoming mediante tre connessioni TCP
Il protocollo è composto da due applicativi (Proxy_client e Proxy_server)

Lo scopo?

Lo scopo dell'applicativo è quello di usufruire delle 3 connessioni al fine di migliorare le prestazioni generali della comunicazione tra Proxy_client e Proxy_server

Progetto di Proxy



Che tipo di problematiche pone il progetto?

Le problematiche del progetto sono molteplici e il gruppo ha cercato di analizzarle da più punti di vista.

1)Punto di vista dell'utente

2)Punto di vista del progettista/programmatore

3)Soddisfare le specifiche del progetto

Tale metodo porta a una successiva divisione del problema in sottoproblemi

Problemi dell'utente

Si può supporre che l'utente nutra le seguenti ambizioni:

1)Usare il programma con applicativi differenti

2)Usare il programma nel proprio sistema operativo

3)Se non ha competenze , configurazione general purpuse

4)Se ha competenze,configurazione ad hoc del programma

5)Prestazioni di rete massimizzate

6)Consumo ridotto di memoria e CPU

Problemi del progettista/programmatore

I problemi del progettista/programmatore sono un numero maggiore di quelli dell'utente:

1)Soddisfare al meglio tutte le ambizioni dell'utente

2)Progettare un programma flessibile,facilmente aggiornabile e comprensibile anche dal programmatore che non ha dimestichezza con gli strumenti di basso livello per le reti

3)Sfruttare al meglio TCP e superare i limiti del protocollo

4)Testare in modo dettagliato il programma

Approccio risolutivo di alcune problematiche

A questo punto è evidente che le problematiche sollevate non sono poche.

Risolvere ,come prima operazione, ogni singolo problema in modo dissociato non è ,secondo il nostro modestissimo parere di studenti, l'approccio migliore.

Riteniamo invece vincente,in un primo momento, mantenere una visione generale ,e allo stesso tempo snellire e semplificare la gestione dei problemi

Come fare tutto questo?

Proxy_configure.c

Come noto , se si compila un programma a manovella , spesso si ha la fortuna di imbattersi nello GNU autoconf.

Esso ha il compito di rilevare l'ambiente su cui si intende compilare il programma e rileva le flag con cui compilare.

Ma se lo GNU autoconf non c'è?

Questo è un ulteriore piccolo problema

La soluzione potrebbe essere quella di realizzare un autoconf scritto interamente in ANSI C89.

Il Proxy_configure:

1)Rileva l'endianess della macchina utilizzata

2)Rileva se la CPU è a 32 o 64 bit.Verifica l'interfaccia di sistema per tale CPU(ILP32 ;LP32;ILP64;LP64)

3)Verifica la conformità del compilatore all' ANSI C89

4)Legge un insieme di flag che permottono di compilare il programma in modo differente

5)Scrive un file Proxy_config.h dove annota tramite keyword(flag) gli esiti della computazione

Il punto 4 è il più importante: esso rappresenta il cardine della progettazione in astratto dell'intero sistema Proxy

Che vantaggi porta avere un sistema di flag?

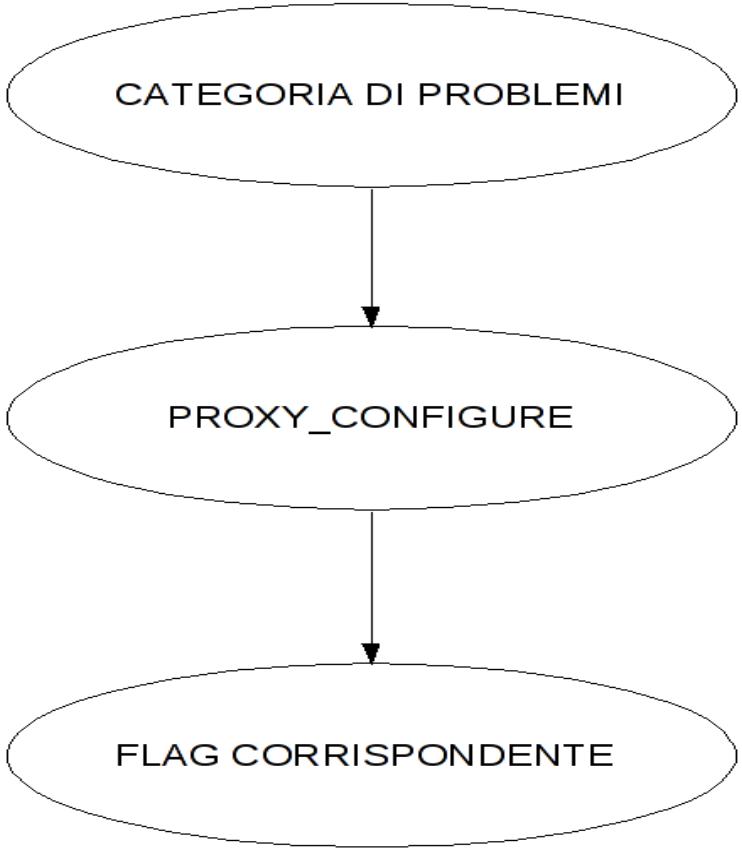
1)Tramite il sistema di flag è stata aggiunta scalarità e design all'applicazione.

2)Tramite piccole keyword è possibile organizzare il codice in modo ordinato

3)Grossolanamente ogni keyword può essere associata a una classe di problemi

4)Tramite keyword il prodotto diventa altamente configurabile

5)Quando ci sarà da affrontare un problema nel dettaglio del codice, basterà risolvere il problema nello scope della keyword


CATEGORIA DI PROBLEMI
PROXY_CONFIGURE
FLAG CORRISPONDENTE

In sintesi:

Il programma Proxy_Configure fornisce un metodo per organizzare e catalogare problemi tramite un sistema di flag.Inoltre individua l'ambiente su cui viene compilato il programma

Offre una soluzione generica all'organizzazione del codice

Scrive nel file Proxy_config.h le Keyword associate al problema

Tutti i sorgenti del Proxy includeranno tale header.

#less Proxy_Configure.c (o l'editor preferito)

#./Proxy_configure LINUX SENDE && less Proxy_config.h

Nello sviluppo di un progetto di rete è importante curare la messaggistica del programma.

1)Mediante la messaggistica il programma comunica con l'usufruitore

2)E' possibile stabilire una gerarchia di tipo tra i messaggi dell'applicazione(e_fatal e_warning,msg_help....)

3)Alcuni comportamenti(e_fatal()) forniscono un meotdo per informare l'usufruitore di un errore fatale o comunque compromettente per il programma.

4)Messaggi appropriati in combinazione a flag relative al debug sono per lo sviluppatore uno strumento importante nel test e nello sviluppo del programma

Message.h and Message.c

Message.h è un interfaccia per usufruire della messaggistica del programma

e_fatal() and e_warning(): Prendono come argomenti una srtinga contenente il messaggio(o NULL) e il valore della variabile errno in caso di errore(0 se non è speicificato niente in errno)

msg_ok() and msg_help() and msg_tutorial(): Prendono come argomento una stringa contenete il messaggio.Stampano il messaggio con colori e indeltazione differente

#less Message.c per informazioni sull'implementazione

```
#define NORMAL          "\033[0m"
#define RED             "\033[01;39;31m"
#define BLUE            "\033[01;39;34m"
#define GREEN           "\033[01;39;32m"
#define YELLOW          "\033[01;39;33m"
#define MAGENTA         "\033[01;39;35m"
#define CYAN            "\033[01;39s;36m"
#define CLEAR_SCR       "\033[2J"

extern void e_warning(const char*msg,int myerror);
extern void e_fatal(const char*msg,int myerror);
extern void msg_ok(const char* msg);
extern void msg_help(const char* msg);
extern void msg_tutorial(const char*msg);
```

Proxy_client e Proxy_server cosa fanno?

Ora che ci sono strumenti per organizzare e districare problemi si analizza il funzionamento generale dell'intero sistena proxy.

1) Il Proxy_client legge dati dal Client e li invia al Proxy_server

2) Il Proxy_server legge i dati inviati dal Proxy_client e li invia al Server

Si procede analizzando la comunicazione dal punto di vista dell'applicativo stesso

Sono il Proxy_client e...

1)Leggo i dati dal Client

2)Organizzo i pacchetti da inviare sulla rete

3)Decido come e quanto inviare sui canali

4)Invio i dati

5)Continuo a fare quello di sopra e controllo che i dati che ho inviato arrivino a destinazione

Sono il Proxy_server e...

1)Leggo i pacchetti dalla Rete

2)Verifico l'idoneità dei pacchetti

3)Organizzo i pacchetti letti

4)Invio i pacchetti letti al Server

Si evince a questo punto che è necessario fornire il programma di opportune strutture dati atte a contenere pacchetti e dati.Tali strutture sono necessarie sia a Proxy_client che Proxy_server

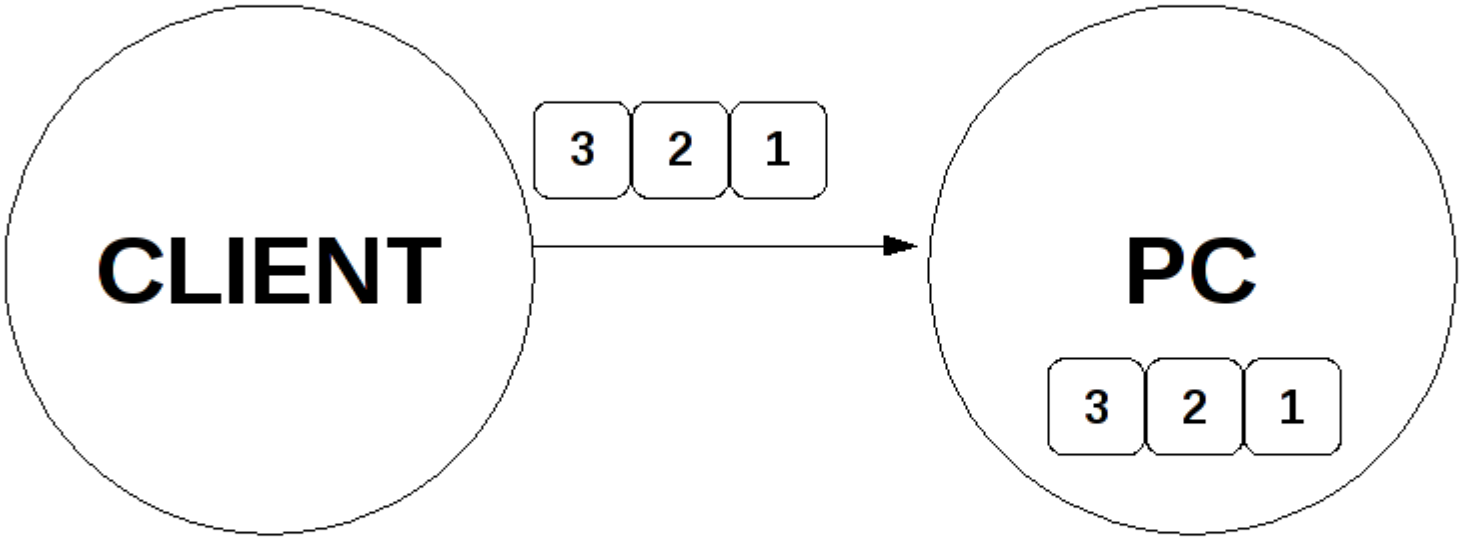
Ordine di dati e pacchetti

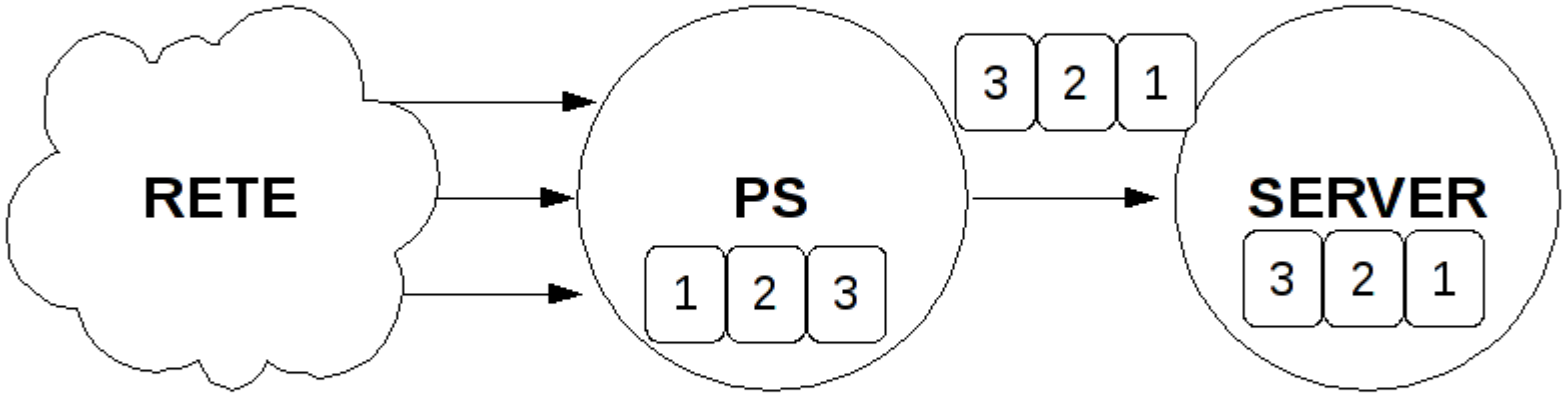
1)Entrambi gli applicativi effettuano letture di dati di cui è necessario conoscere il giusto ordine

2)Le letture sono comunque differenti.Il Proxy_client legge da un solo Client.

4)L'ordine dei dati da suppore giusto è semplicemente quello di lettura

3)Il Proxy_server non è detto che riceva dati nello stesso ordine in cui sono stati inviati dal Client ; deve dunque essere predisposto a organizzare i pacchetti per poterli poi inviare nell'ordine corretto al Server


CLIENT
3
2
1
PC
3
2
1


RETE
PS
1
2
3
3
2
1
SERVER
3
2
1

Strutture dati per pacchetti e dati

1)La coda può essere una buona struttura dati per il PC dove l'ordine dei dati segue una politica FIFO

2)Il PS deve avere l'opportunità di ordinare i pacchetti e quindi la possibilità implicità di poter prelevare liberamente i dati immagazzinati

3)Una struttura semplice che può svolgere questo compito (anche se non con una computabilità efficente)è la linked list

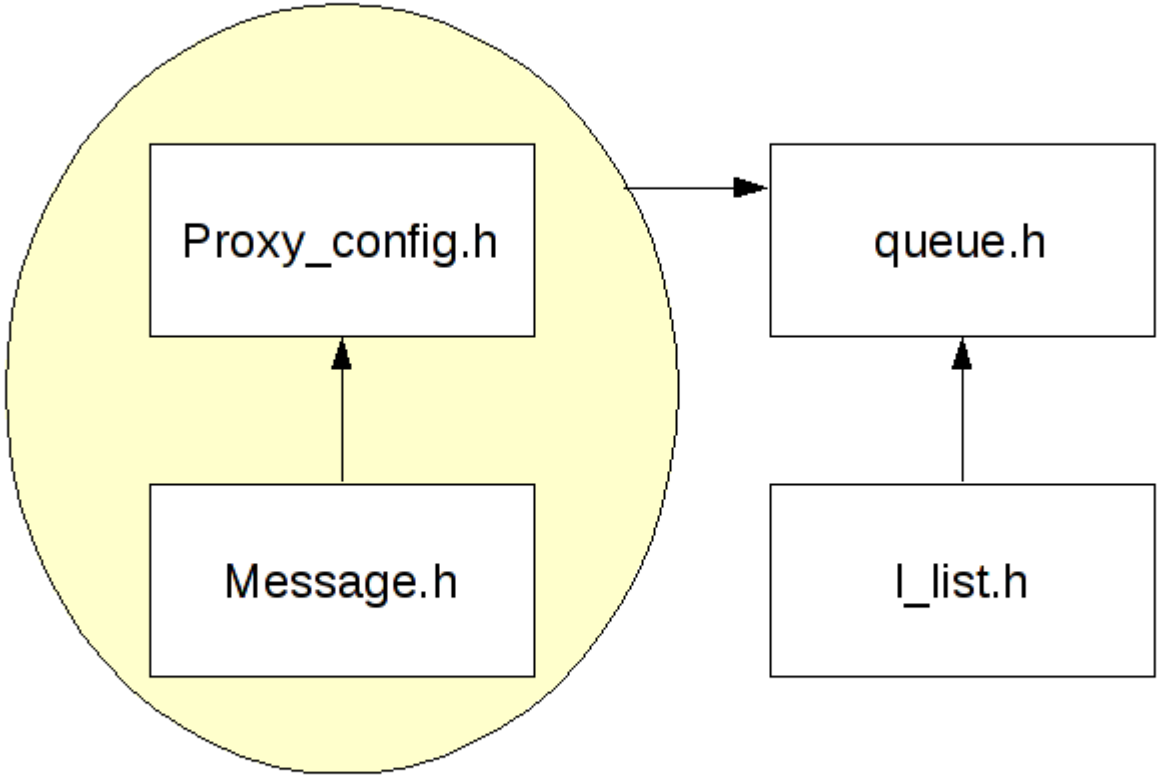
API su code e liste del sistema Proxy

1)Il sistema Proxy mette a disposizione del programmatore due moduli per gestire generiche code e liste.

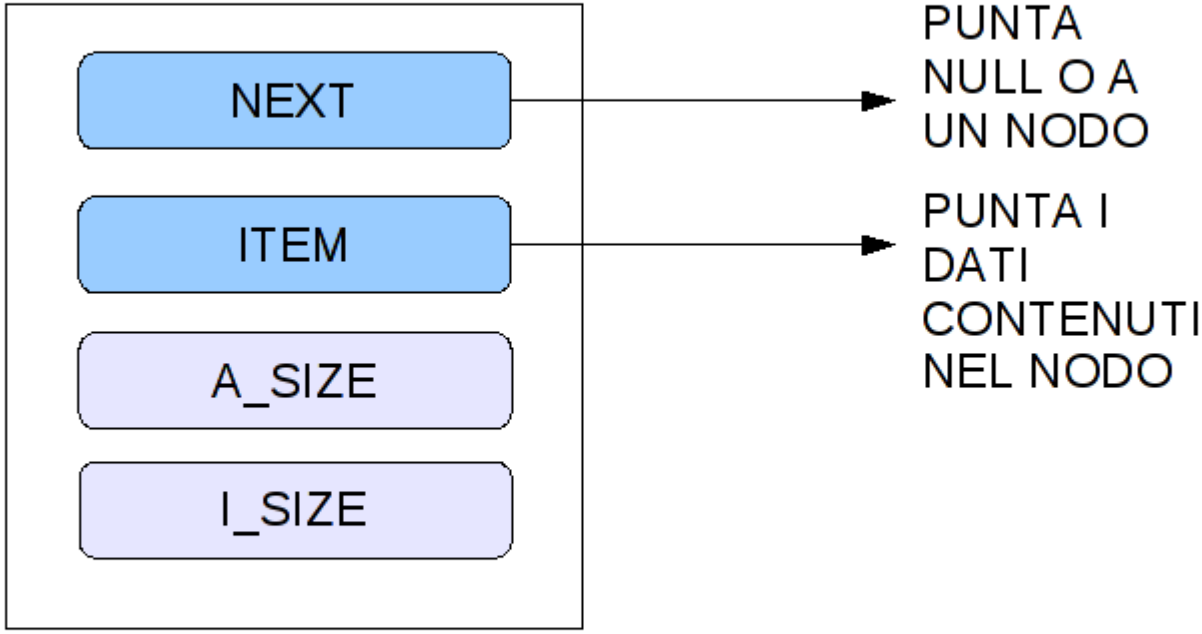
2)Le code e le liste dell'API possono contenere qualsiasi tipo di dato

3)Offrono un set di funzioni ampio e modalità differenziate per eseguire alcune operazioni comuni

4)Data la similarità del nodo della coda e di quello della lista ,l'implementazione è stata immaginata in modo tale da "emulare" un comportamento di ereditarietà dove la classe lista è sottoclasse della classe coda


Proxy_config.h
queue.h
Message.h
l_list.h

STRUTTURA DI UN NODO DI CODA E LISTA



1)next è un puntatore al prossimo nodo

2)item è un puntatore a void

3)a_size indica i byte allocati per contenere i dati puntati da item.E' sempre un valore multiplo di 2

4)i_size indica la grandezza dell'oggetto allocato.Se è dispari è sicuramente minore di a_size

5)Se si vuole usare il nodo come riferimento a una coda si usa il tipo queue altrimenti l_list.

6)queue e l_list sono 2 nomi differenti per indicare un puntatore a nodo

1)Poichè i due moduli sono stati immaginati come due oggetti eistono, in modo similare a C++, una funzione per allocare e creare un oggetto di tipo lista o coda(new_queue() new_linked()) e una corrispettiva funzione delete()(delete_linked(),delete_queue())

2)Poichè lista e coda usano lo stesso nodo è possibile utilizzare le funzioni sulla lista nella coda e viceversa

3)Esiste il concetto di lista o coda vuota che è rappresentato da un unico nodo avente i puntatori a null e a_size e i_size a 0

4)E' possibile effettuare le operazioni comuni su lista e coda sia in modo tradizionale che con funzioni del modulo atte a lavorare con puntatori

1)Data la ricorsività implicita di queste strutture dati molte funzioni che operano su esse sono implementate mediante ricorsioni di tipo tail

2)Le API con liste e code sono compatibili con ILP32 LP32 ILP64 LP64

3)Per continuare a garantire la compatibilità dei due moduli con più ABI basta compilare opportunamente le lo scope delle keyword ILP32 LP32 LP64 ILP64

4)Mediante il meccanismo di a_size e i_size l'applicazione potrebbe sopperire a un kernel che non supporta la paginazione della memoria.La memoria sarebbe dunque da immaginare come un insieme di caselle di 2 bytes

PUSH e POP

Le operazioni di push e pop sono la base della struttura dati coda

cpy_pop() push() eseguono le operazioni in modo tradizionale.cpy_pop scarta la testa della coda e ritorna un puntatore a coda copiando l'elemento nel puntatore passato.

Esiste una variante di queste funzioni che usa solo puntatori.Non viene dunque copiato nessun dato ma vengono spostati puntatori.Questo set di funzioni per quanto più efficente è più soggetto a errori da parte del programmatore

```
#include "queue.h"

int main()
{
        int number=0;
        int *numb=NULL;
        queue *q=NULL;
        int check=0;

        q=new_queue(q);
        if(q==NULL) e_fatal("Non posso allocare la coda\n",0);

        number=5;
        push(&number,q,4);
        number=9;
        push(&number,q,4);
        print_queue(q);
        number=0;
        q=cpy_pop(&number,q,&check,4);
        if(q==NULL) e_fatal("errore",0);
        numb=(int*)q->item;
        q=pop_s_item(q);
        if(q==NULL) e_fatal("errore",0);
        printf(GREEN"\n stampo la coda dopo 2 pop\n"NORMAL);
        print_queue(q);
        printf("gli elemnti salvati sono: \n");
        printf("number=%d num=%d \n",number,*numb);
        q=delete_q(q);
        free(numb);
        return 0;
}
```

```
struct q_node
{
        /*pointer to next queue-node*/
        struct q_node *next;

        /*pointer to item contained in queue-node*/
        void *item;
        unsigned short int i_size;
        unsigned short int a_size;
};
typedef struct q_node queue;

extern queue* new_queue(queue*q);
extern int is_empty_q(queue* q);
extern queue* cpy_pop(void* item,queue* q,int *check,int size);
extern int push(void *elemento, queue *q,int size);
extern int point_push(void *elemento, queue *q);
extern int q_length(queue *q);
extern void*  delete_q(queue* q);
extern void print_node(queue* q,const int flag);
extern void print_queue(queue*q);
extern int get_isize(queue* q);
extern int get_asize(queue*q);
extern queue* pop_s_item(queue*q);
extern void* point_item(void *p, queue* q,int *myerror);
extern queue* add_to_head(queue* q, void* item,const int size);
extern int add_to_item(queue* q, void *data,int size);
extern queue* delete_item(int size,queue* q);
void*  delete_q(queue* q);
```

l_list

1)Le funzioni cardine del modulo l_list sono add_to() ; get_linked();delete_linked_node();

2)Il loro compito è quello di prelevare, aggiungere e rimuovere liberamente dati e nodi sulla lista

3)E' possibile usare sui dati di tipo l_list tutte le funzioni del modulo queue(poichè è un nome differente per la stessa struttura dati)

4)Purtroppo la struttura non è veloce negli accessi.Per prelevare un elemento in posizione n si ha una complessità rispetto al tempo di O(n)

```
#include "l_list.h"

int main()
{
        int number=0;
        int *numb=NULL;
        l_list *l=NULL;
        int i=0;
        l_list *tmp=NULL;

        l=new_linked(l);
        if(l==NULL)   e_fatal("non posso allocare la lista",0);

        number=3;
        if(!(add_item(l,&number,sizeof(int))))
                e_fatal("non posso aggiungere l'elemento alla lista",0);
        for(i=0;i<2;i++)
        {
                number=9;
                if(!(add_item(l,&number,sizeof(int))))
                        e_fatal("non posso aggiungere l'elemento alla lista",0);
        }
        tmp=get_linked(1,l);
        numb=(int*)tmp->item;
        printf("numb is %d \n",*numb);
        tmp=NULL;
        numb=NULL;
        number=5;
        print_linked(l);
        numb=(int*)l->item;
        l=pop_s_item(l);
        l=delete_linked_node(l,0);
        delete_linked(l);
        free(numb);
        numb=NULL;
        return 0;
}
```

```
#ifndef L_LIST_H
#define L_LIST_H
#endif

#include "queue.h"

#ifdef PROXY_C89
typedef queue l_list;
#else
#error compiler is not ANSI C89
#endif

#if defined(PROXY_C89) && defined(QUEUE_H)

extern l_list* new_linked(l_list *ls);
extern int add_item(l_list *ls,void* data,int size);
extern l_list* add_to(l_list *ls,void* data,int size,int index);
extern l_list* delete_linked(l_list *head);
extern l_list* get_linked(int index,l_list *ls);
extern l_list * delete_linked_node(l_list *head,int index);
extern int is_empty_linked(l_list *ls);
extern l_list* delete_to(int start,int end,l_list* head);
extern int add_node(l_list *head, l_list *node);
extern l_list* add_node_index(l_list *head,l_list *node,int index);
extern void print_linked(l_list *head);
extern int add_data_linked(l_list *head,int index,void* data,int size);
extern l_list* go_next(l_list *ls);

#else

#error l_list.h not defined or compiler is not ANSI C89

#endif
```

Ricapitolando

1)E' presente uno strumento che gestisce le flag con cui compilare il programma.Mediante tale strumento è possibile catalogare problemi e organizzare il codice

2)E' presente un modulo per la messaggistica del programma

3)Ci sono strutture dati atte a contenere dati e pacchetti sia mediante una politica ben definita(FIFO) che liberamente(linked_list)

Osservazioni

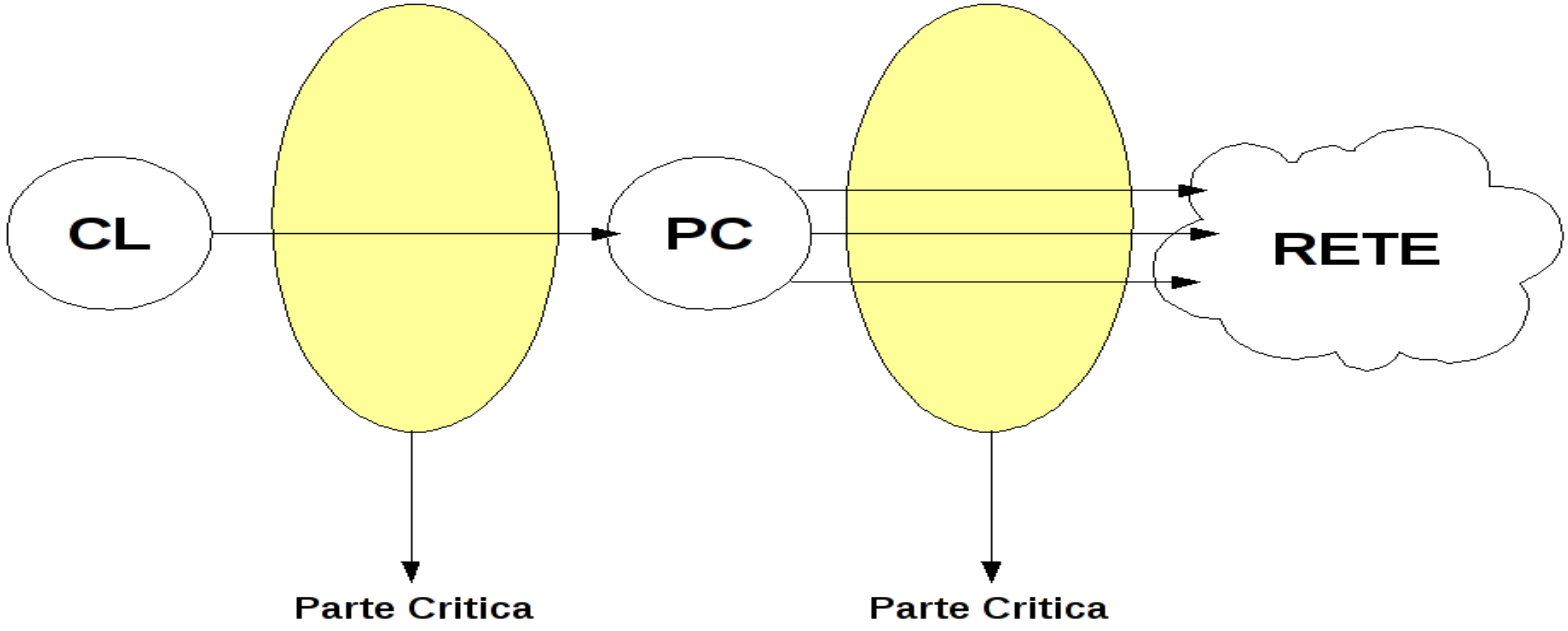
Dato lo scopo del programma, è evidente che gran parte del lavoro e delle performance del sistema Proxy deriveranno dal Proxy_Client

PC ha il compito di sfruttare a pieno i link di connessione con il Proxy_Server fornendo possibilemnte una base per un servizio di QoS

Il PC legge per primo i dati dal Client e lo deve fare in maniera performante

A questo livello il PS può essere immaginato come una semplice entità che legge dati

RAPPRESENTAZIONE DEI PUNTI CRITICI PER IL CORRETTO FUNZIONAMENTO DEL SISTEMA PROXY



Problemi di progettazione relativi alla comunicazione Client---------->Proxy_Client

1)Scegliere un'adeguata politica di lettura

2)Immagazzinare i dati nel corretto ordine

Mediante una politica FIFO è possibile riuscire nell'intento del punto 2.

Il punto 1 è più spinoso poichè deve rispondere a:

Quanti bytes devo leggere?

Con quali modalità?

Problemi di progettazione relativi alla comunicazione
Proxy_Client---------->Rete-------->Proxy_Server

1)Quanto devono essere grossi i pacchetti che mando sulla rete?

2)Il TCP non supporta multihoming,bisogna sopperire in qualche modo

3)Come faccio a scegliere i link di connessione?

4)Quale può essere una giusta politica di invio?

5)Quali sono i problemi tipici di TCP?Come posso risolverli?

6)Come faccio a mantenere una certa interattività?

Osservazioni

1)Il TCP e gli altri protocolli di trasporto, compresi molti protocolli di livello applicazione utilizzano un header per comunicare con l'altra estremità

2)Le reti dei differenti link possono differire in caratteristiche(banda e latenza) quindi l'eventuale strategia di invio deve essere dinamica

3)E' bene immaginare la rete come un entità estremamente inaffidabile,dove gli imprevisti sono all'ordine del giorno

4)Si può immaginare il PC come un entità che dorme.Nel momento in cui percepisce attività inizia il suo lavoro

Approccio risolutivo

1)Finchè si legge da un solo client basta leggere al massimo una certa quantità di dati,ogni qual volta ci sia qualcosa da leggere

2)In casi isolati può essere opportuno bloccare la lettura dal client

3)I pacchetti data l'imprevedibilità della rete sono di grandezza variabile e possibilmente tale valore, dovrebbe essere stabilito nel momento in cui si decide di inviare dati nello specifico link

Packet Pair

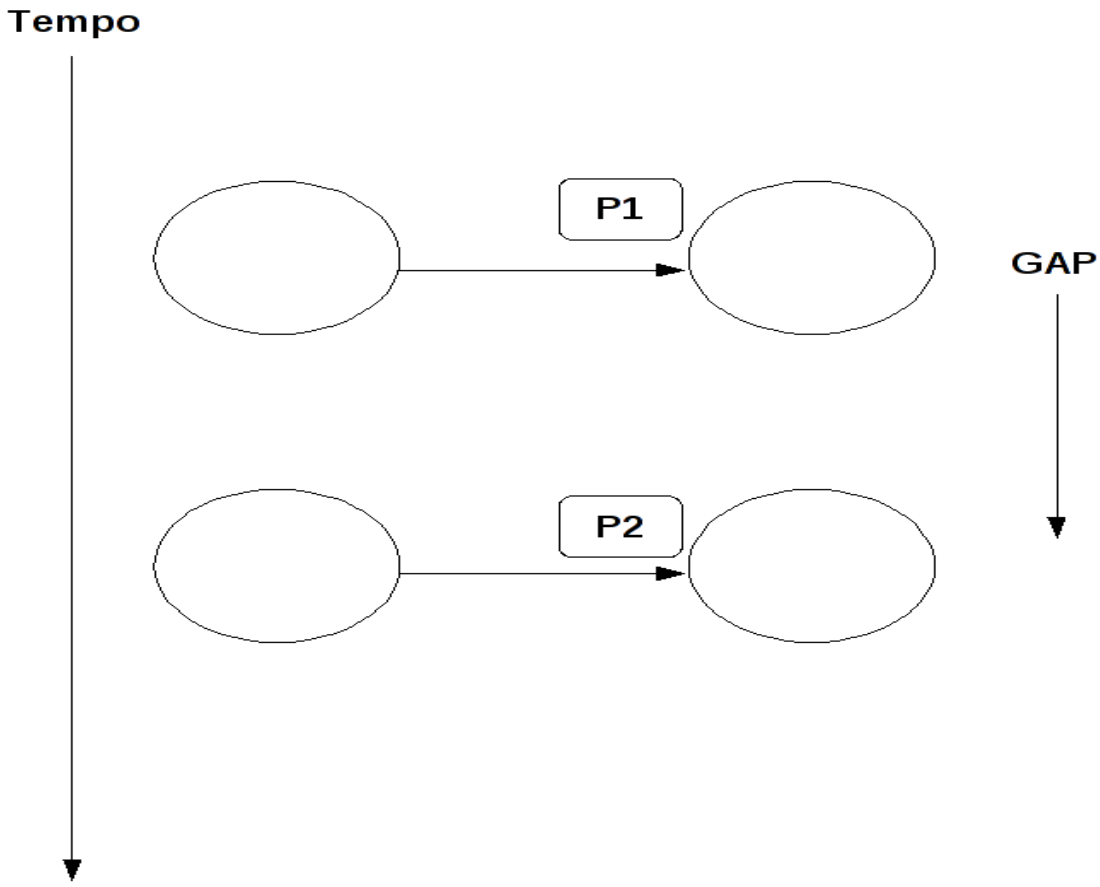
Il packet pair è un algoritmo per individuare la banda di un ipotetico link di connessione.

La sua meccanica è semplice: si invia un pacchetto P1 si aspetta un periodo di tempo prestabilito si invia P2==P1.

La banda misurata sarà data dalla formula:

$$banda = P1/(Tp2 - Tp1)$$

$$banda = P1/gap$$


Tempo
P1
GAP
P2

Svantaggi del Packet Pair

1)Il packet pair punta la sua valutazione sul valore del gap tra i due pacchetti.

2)Il gap è un valore stabile solo in reti punto punto

3)L'accorpamento di dati nei router altera il gap

4)Reti con caratteristica di latenza instabile come ad esempio le reti wireless , provocherebbero alterazioni continue del gap

5)Un algoritmo di invio che si affida a test necessita di test periodici che possono sovraccaricare la rete

Algoritmi Adattivi

Sono algoritmi che cercano mediante statistiche di adeguare le proprie scelte durante l'esecuzione del programma

La chiave per realizzare un buon algoritmo è far eseguire valutazioni oggettive a intervalli di tempo brevi

Con tale politica ,con buona approsismazione,nell'arco di tempo in cui si esegue l'applicazione l'algoritmo si sarà comportato dignitosamente

Un algoritmo adattivo offre anche una maggiore reattività al programma

Scelte dell'agoritmo adattivo del Proxy

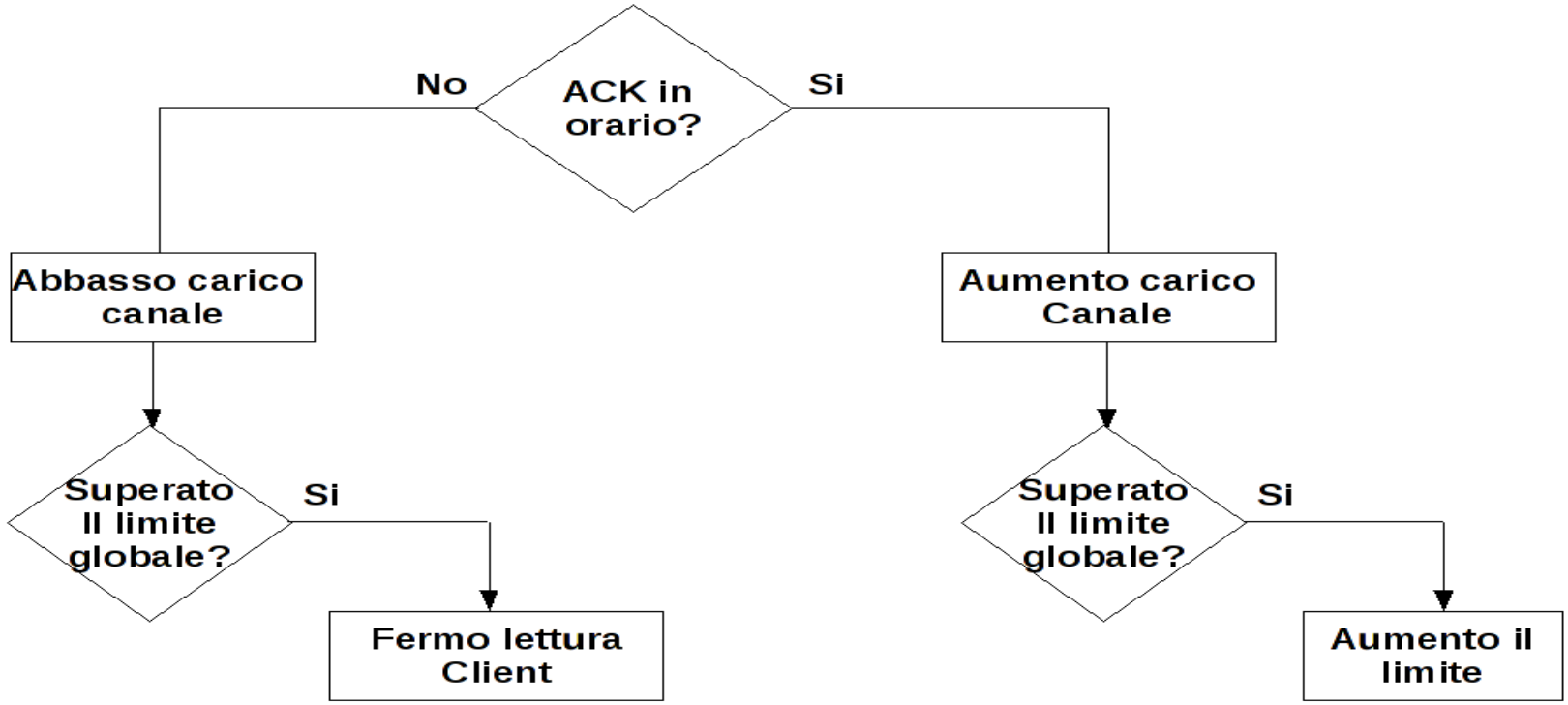
1)Determinare un limite di latenza massimo

2)Se il limite di latenza è maggiore della latenza attuale aumento la grandezza del pacchetto

3)Se il limite di latenza è inferiore rispetto alla latenza attuale diminuisco la grandezza del pacchetto

4)Registro i valori globali del carico di lavoro e segno il valore soglia prima di un fallimento

5)Se ho superato il valore soglia , fermo la lettura verso il client fino a quando non termina il sovraccarico


ACK in orario?
No
Si
Abbasso carico canale
Aumento carico Canale
Superato Il limite globale?
Si
Fermo lettura Client
Superato Il limite globale?
Si
Aumento il limite

Implementazione Proxy Client

Per rendere l'implementazione possibile ,in conformità con quanto detto qui, il gruppo di lavoro ha realizzato altre API per la gestione del protocollo

1)E' stata realizzata la struttura dati pacchetto con relativa API

2)E' stata realizzata la struttura dati canale che rappresenta i link di connessione

3)E' stata realizzato il modulo Net_Util che fornisce un API sia per le system call di sistema sulla rete che per le funzioni del protocollo

Implementazione modulo Packet

1)Il pacchetto del protocollo ha un header di 6 bytes

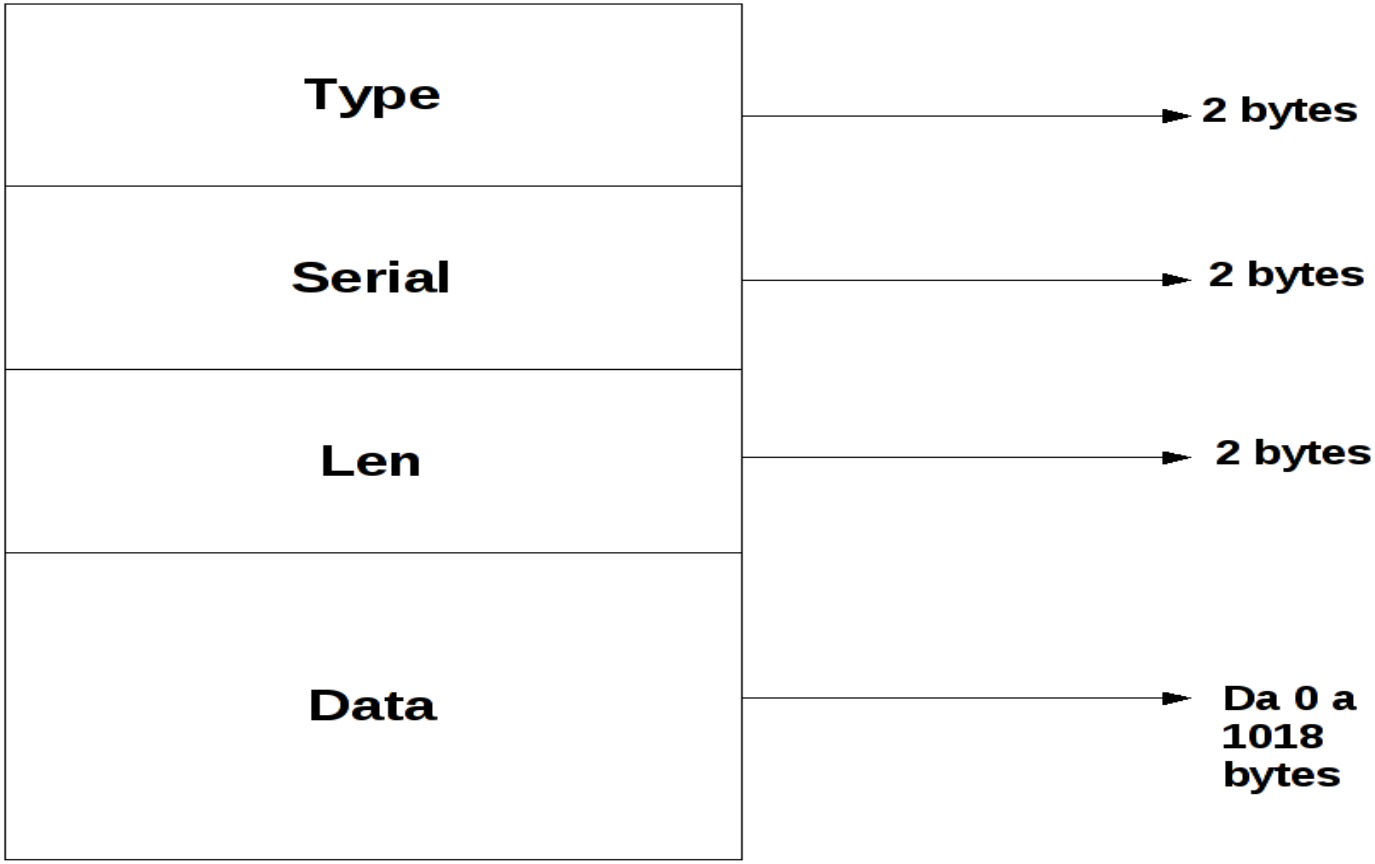
2)Contiene il campo type che indica il tipo di pacchetto(ad esempio data o ack)

3)Un campo serial contenete il seriale del pacchetto

4)Un campo len cotenente un valore che rappresenta la grandezza del pacchetto

5)Un campo data che contiene il carico dati del pacchetto

6)La grandezza header+data è al massimo 1024 bytes


Type
2 bytes
Serial
2 bytes
Len
2 bytes
Data
Da 0 a 1018 bytes

Funzioni del modulo packet

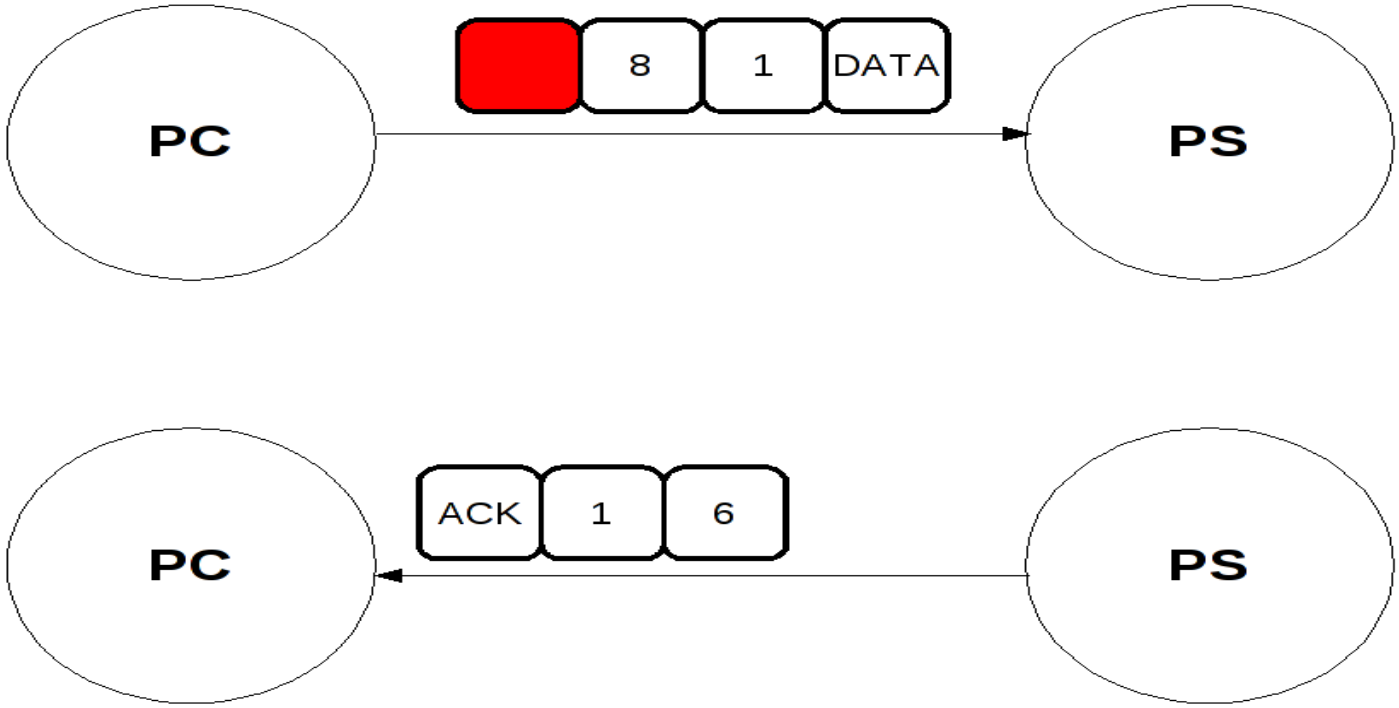
Le funzioni del modulo sono new_packet() make_packet() e delete_packet() che rispettivamente creano e cancellano pacchetti

packet* make_packet(short type,short id,short len,void* data)

Ritorna il puntatore al pacchetto con le caratteristiche passate alla funzione

```
#include "Message.h"
#ifdef PROXY_C89
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#else
        #error compiler does not support C89
#endif
#ifndef PACKET_H
#define PACKET_H
#endif

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

struct pack
{
        /*-----------------header-------------------------*/
        short int type;
        short int serial;
        short int len;/*header+data*/
        /*----------------end header--------------------*/
        void *data;
};
typedef struct pack packet;

extern packet* new_packet(packet*p);
extern packet* make_packet(short int type,short int id,short int len,void* data);
extern void* delete_packet(packet *p);
extern packet* parse_packet(void *buffer ,int n_read);
```

```
#include "Packet.h"
int main()
{
    packet* p=NULL;
    char buffer[100];

    p=make_packet(DATA,1,106,buffer);
    if(p==NULL) e_fatal("i can not allocate packet",0);

    printf("len pacchetto=%d seriale=%d\n",
        p->len,p>serial);

    p=delete_packet(p);
    return 0;
}
```


PC
8
1
DATA
PS
PC
ACK
1
6
PS

Il modulo Channel

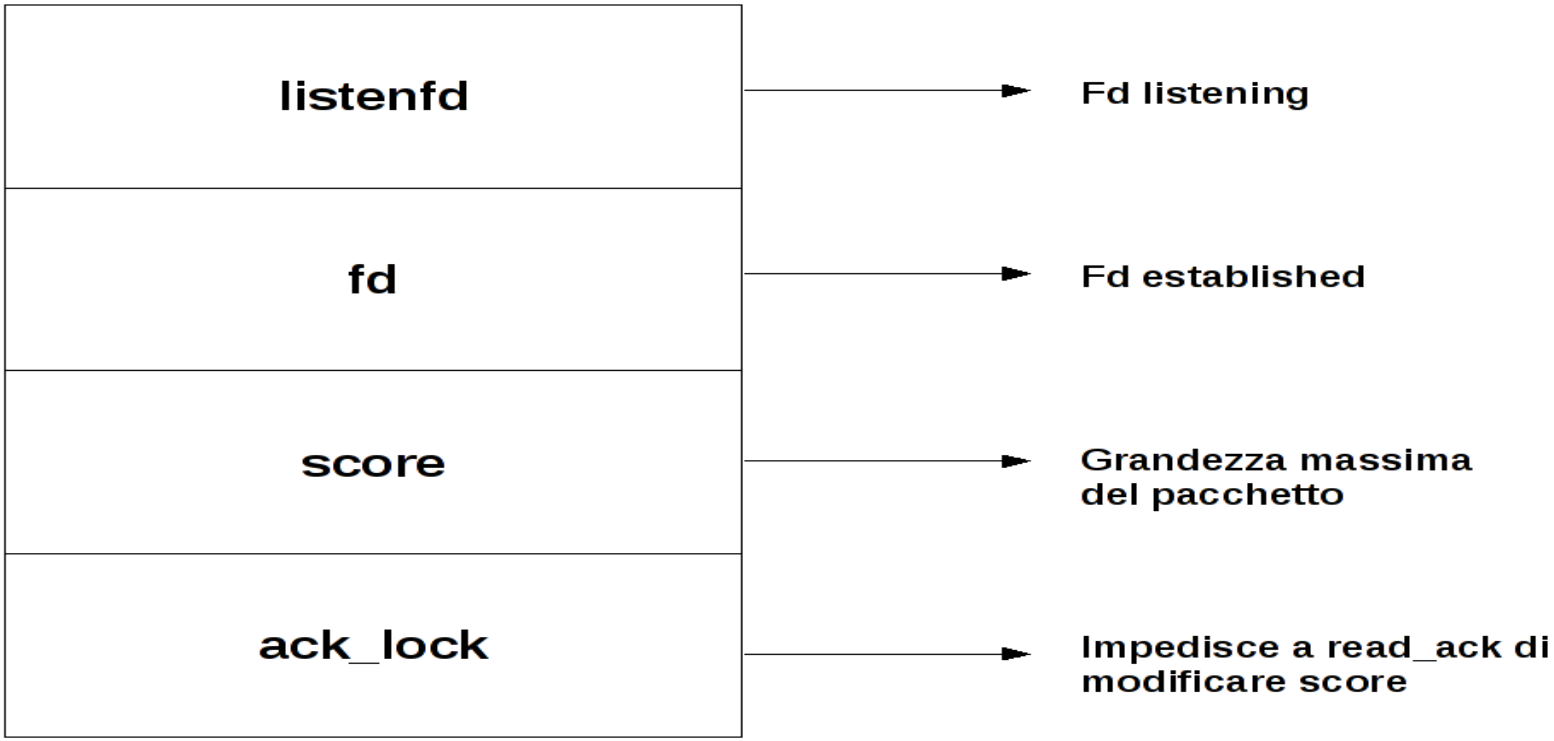
I link delle connessioni sono rappresentate ,dal SO, come un file descriptor.

Queta rappresentazione ci permette di scrivere e leggere dati sulla rete ma non permette ulteriori valutazioni

E' stato ritenuto necessario creare un apposita struttura contenente il file descriptor del canale più altre informazioni come ad esempio la grandezza massima del pacchetto da spedire

La struttura contiene anche altri campi, molti dei quali sono obsoleti e spariranno nelle successive versioni

Elementi essenziali della struttura channel



Il modulo Channel

Il modulo contiene anche altri campi (come ad esempio il numero di porta e l'ip) e API per prelevare ognuno dei suoi elementi.

Questo modulo è datato e probabilmente subirà una forte revisione nello sviluppo.Verranno eliminati molti elementi e molte funzioni al fine di lasciare la struttura nella sua forma essenziale

Gli sviluppi di tale modulo saranno quindi argomento di dibattito a fine talk.
Per il modulo non ci saranno codici di esempio a parte il prototipo e le api che usano channel.
#less Channel.h/.c per info

Il modulo Net_Util

E' uno dei moduli cardine del sistema Proxy.

Il modulo contiene funzioni attienenti alla programmazione di rete ,offre quindi un insieme di API per lavorare con le system call per le reti

Il programmatore non di sistema dovrebbe essere agevolato nell'utilizzare tali funzioni che sono un astrazione delle system call

L'ausilio di tali funzioni garantisce la portabilità nei sistemi supportati poichè molte di esse sono composte da keyword

API di rete di Net_Util

Le API di programmazione di rete del modulo Net_Utili sono astrazioni delle system call bind() listening() connect() setsockopt()

proxy_tpc_opt() usa setsockopt() con la paritcolarità che funziona solo su socket che rappresentano TCP.

La funzione ritorna 1 in caso di successo 0 in caso di errore. La funzione ha la seguente sinopsi:

```
int proxy_tcp_opt(int socket,int sock_opt,int value,
                  int* myerror)
```

su myerror viene memorizzato il valore di errno

proxy_sock_opt() ha un funzionamento analogo a proxy_tcp_opt() ma funziona su socket non TCP.

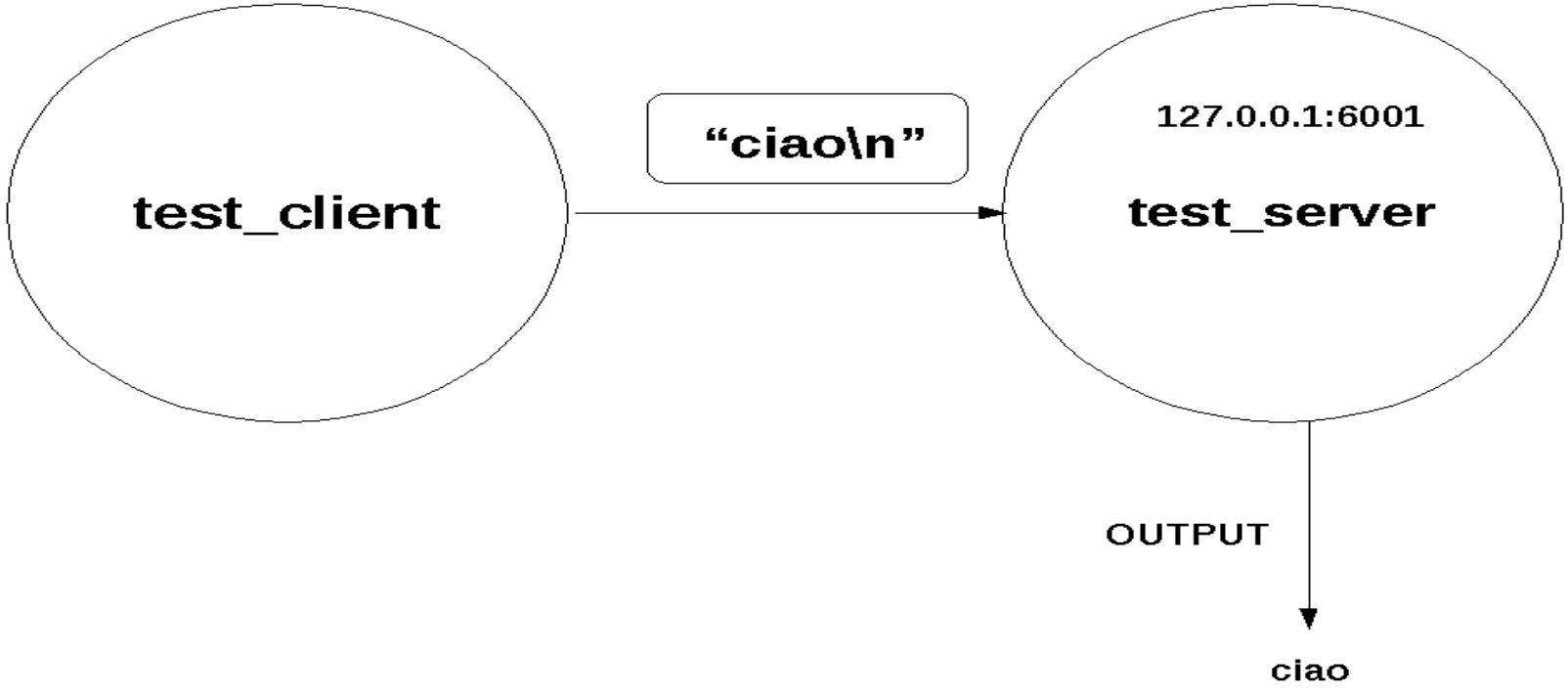
Le funzioni astrazione di bind() listening e connect() sono proxy_listening() e proxy_connect().Le due funzioni esistono anche nella versione advanced che permette di settare la grandezza dei buffer del socket.

Nelle versioni non advanced i buffer sono di 8000 bytes

La funzione proxy_listening() esegue anche una bind()

Gli argomenti passati alle due funzioni sono l'ip sotto forma di stringa e la porta sotto forma di numero intero

Semplice comunicazione Client Server



```
#include "Net_Util.h"
int main()
{
    int listenfd=0;
    int acceptfd=0;
    int len=0;
    SO_IN local;
    byte buffer[10];

    listenfd=proxy_listening("127.0.0.1",6001);
    if(listenfd<0) e_fatal("listening failed",0);

    do
    {
        memset (&local, 0x00, sizeof (local));
        len = sizeof (local);
        acceptfd = accept (listenfd, (struct sockaddr *) &local, &len);
    }while ( acceptfd<0 && (errno==EINTR));

    memset(buffer,0x00,10);
    read(acceptfd,buffer,10);
    printf("%s",buffer);
    fflush(stdout);
    close(acceptfd);
    close(listenfd);
    return 0;
}
```

```
#include "Net_Util.h"

int main()
{
   int connectfd=0;
   byte buffer[10]="ciao\n";

   connectfd=proxy_connect("127.0.0.1",6001);
   if(connectfd<=0) e_fatal("connect() failed",0);

   write(connectfd,buffer,5);
   close(connectfd);
   return 0;
}
```

API di protocollo in Net_Util

Le API di protocollo eseguono funzionalità del sistema proxy

La suddivisione delle funzioni dell'API è legata al Proxy quindi c'è una funzione per:

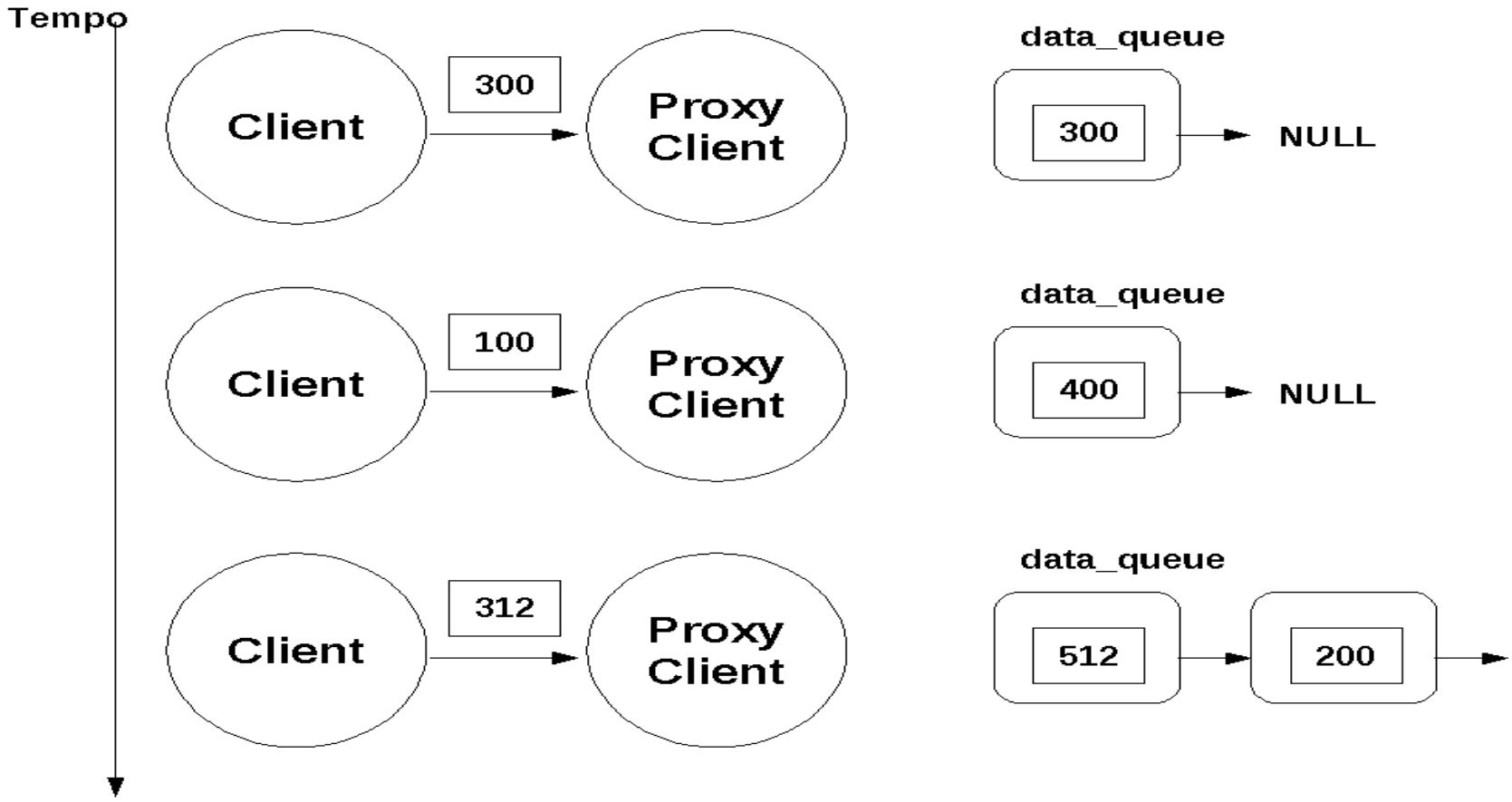
1)Leggere dati dal client e metterli in coda

2)Inviare dati su un canale

3)Leggere ACK

4)Gestire la lista di pacchetti che aspettano ACK

Esempio di read_data()



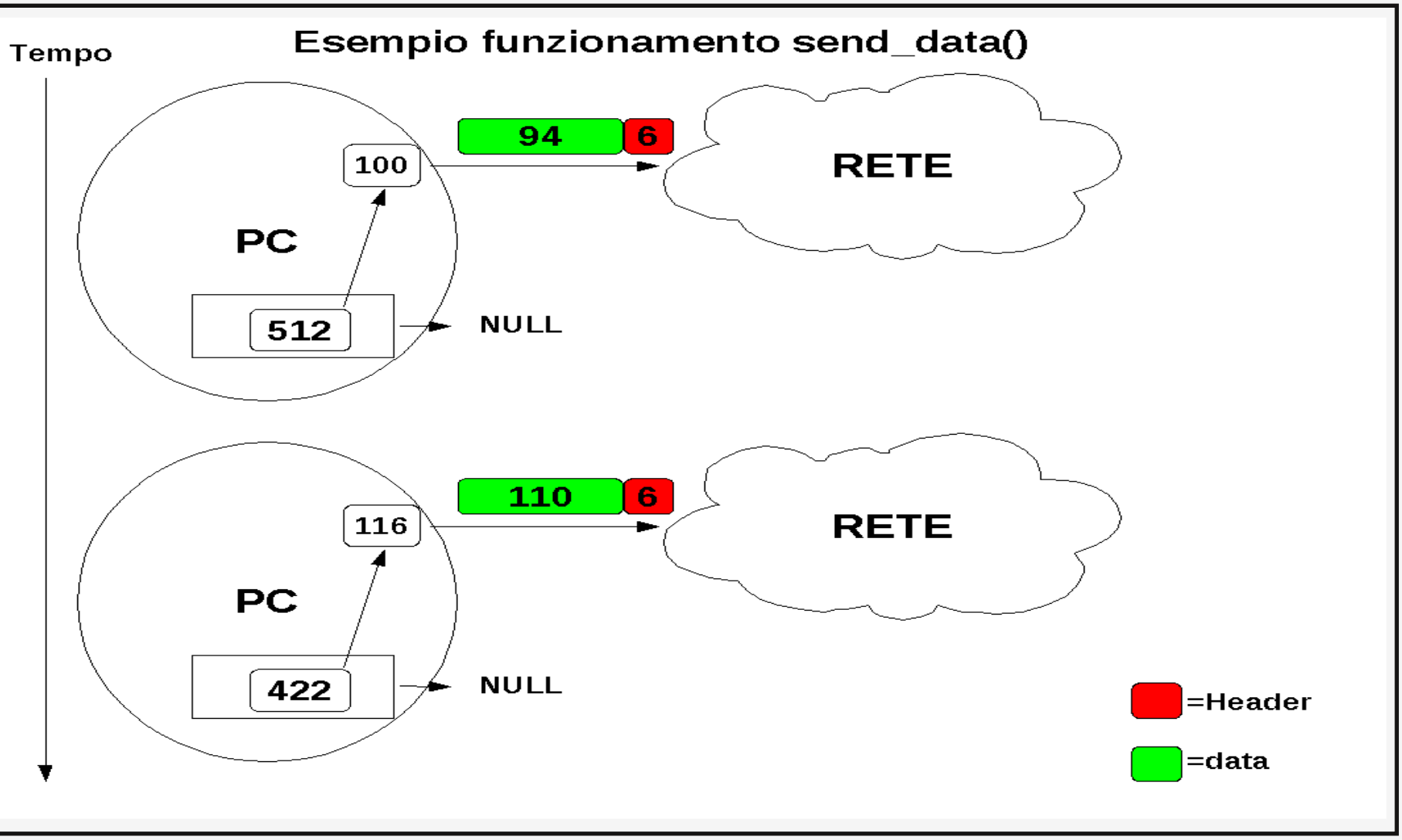
queue* read_data(channel * canale,queue* data_queue)

1)Legge dati da un generico canale TCP e li mette in coda

2)Legge con socket bloccante

3)In caso di errore la funzione termina con un e_fatal() riportando l'errore appropriato

4)Alloca nodi con contenuto dati che al massimo è 512 bytes.Questa operazione è automatica.Il chiamante non può quindi modificare in alcun modo questa scelta


Esempio funzionamento send_data()
Tempo
94
6
100
PC
RETE
512
NULL
110
6
116
PC
RETE
422
NULL
=Header
=data

queue* send_data(queue* q,l_list *ack_list,channel *c,int *serial,int *res)

1)Preleva i dati dalla coda di lettura Client--->PC

2)Assembla un pacchetto grande come lo score del canale

3)Invia il pacchetto appena assemblato al PS

4)Aggiunge il pacchetto alla lista di pacchetti che attendono ACK

5)In caso di errore memorizza l'errore in res o termina con un e_fatal().

Lettura ACK

Il protocollo adotta un sistema di ACK multiplo, vale a dire che per ogni pacchetto inviato il PC si aspetta un ACK.

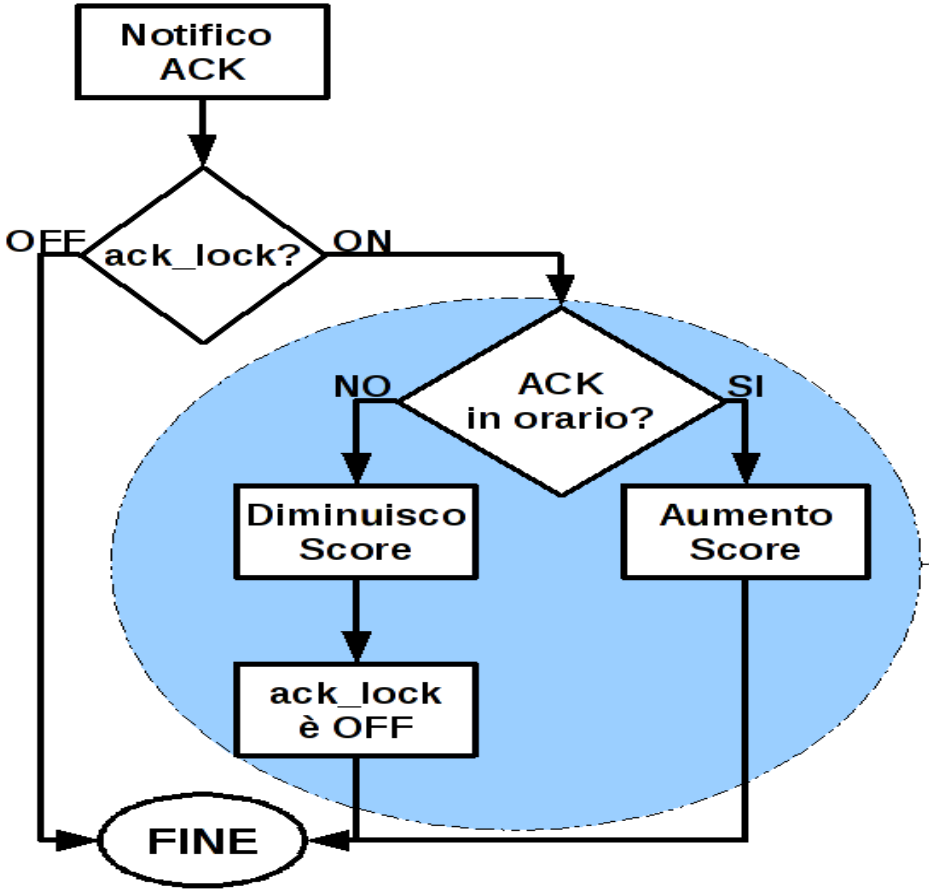
1)Tramite tale modalità si ha il vantaggio di poter monitorare il RTT di ogni singolo pacchetto

2)Si può verificare se pacchetto è giunto a destinazione

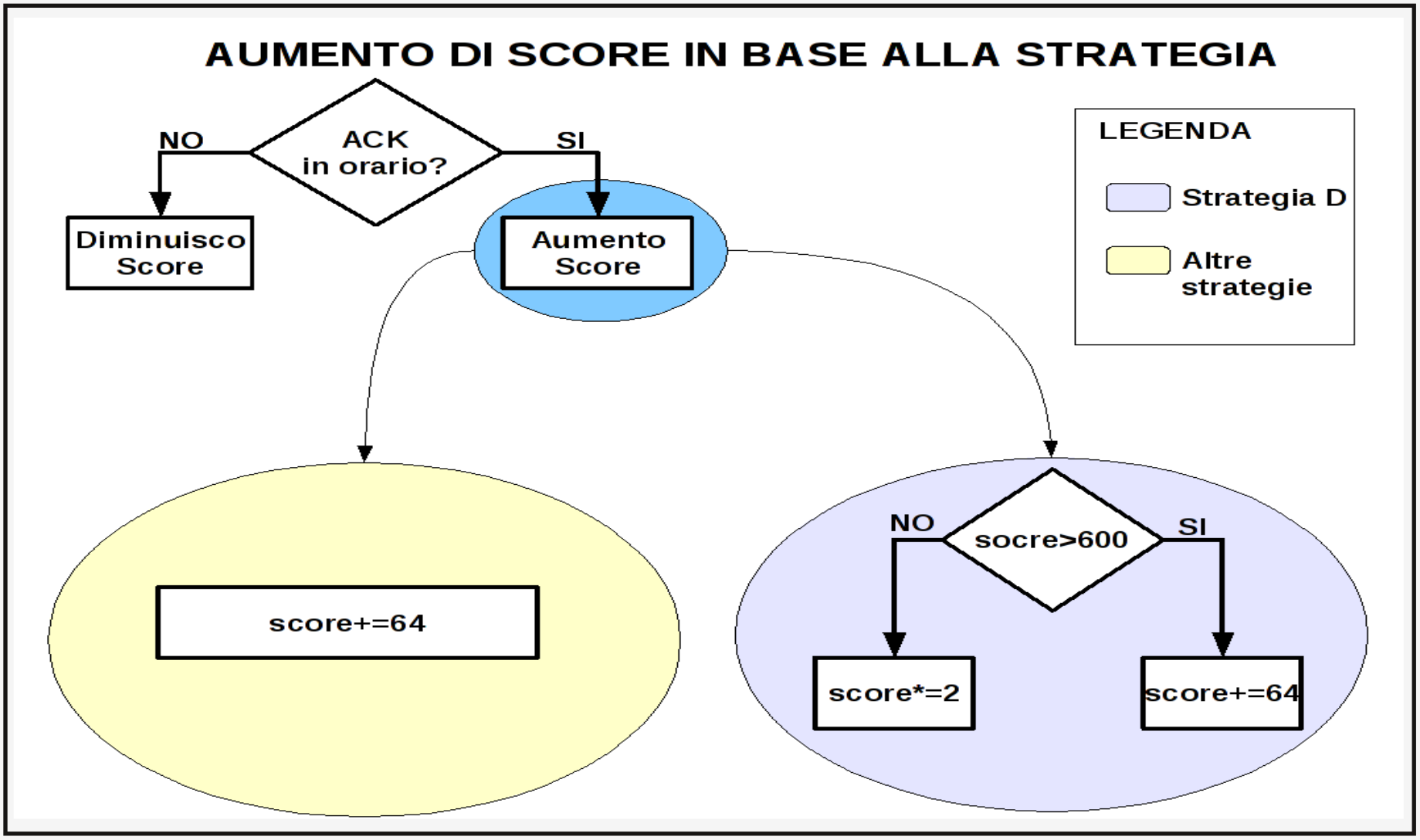
3)Si ha lo svantaggio di dover riservare più banda agli ACK

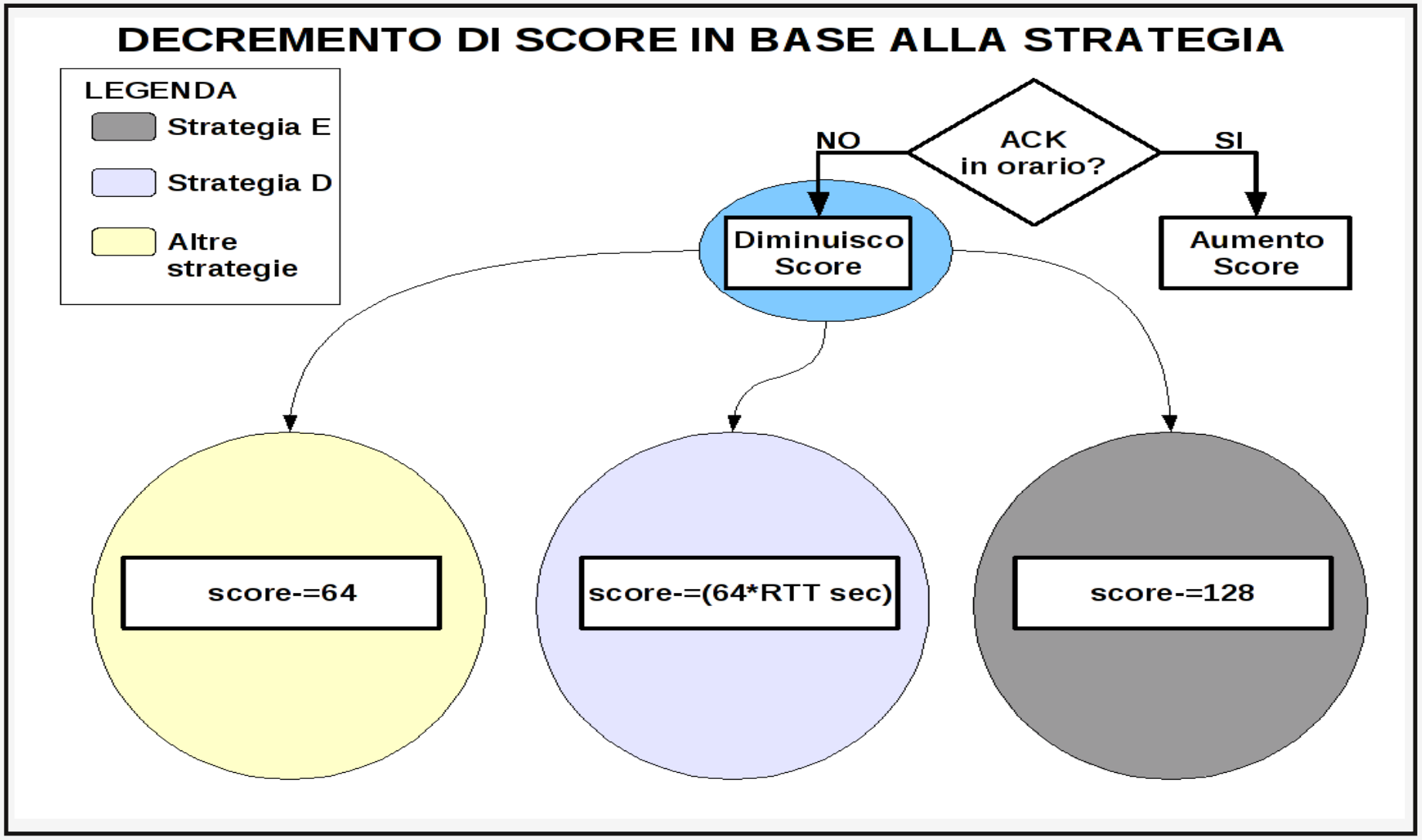
Il punto 1 è molto interessante perchè permette di poter effettuare una prima analisi per l'algoritmo adattivo del sistema Proxy

Diagramma di read_ack()



Questa parte varia leggermente per ogni algoritmo adattivo.In particolare la drasticità con cui si abbassa o si aumenta il valore score


AUMENTO DI SCORE IN BASE ALLA STRATEGIA
ACK in orario?
NO
SI
Diminuisco Score
Aumento Score
LEGENDA
Strategia D
Altre strategie
score+=64
socre>600
NO
SI
score*=2
score+=64


DECREMENTO DI SCORE IN BASE ALLA STRATEGIA
LEGENDA
Strategia E
Strategia D
Altre strategie
ACK in orario?
NO
SI
Diminuisco Score
Aumento Score
score-=64
score-=(64*RTT sec)
score-=128

l_list* read_ack(channel *canale,l_list *ack_list, int *ack_lock)

1)La funzione legge ACK dal canale passato

2)La funzione ritrona un puntatore alla lista di pacchetti che attendono ACK, rimuovendo il pacchetto corrispondente dalla lista

3)La funzione modifica il valore score del canale passato

Ritorna e_fatal() o e_warning() o NULL in caso di errore

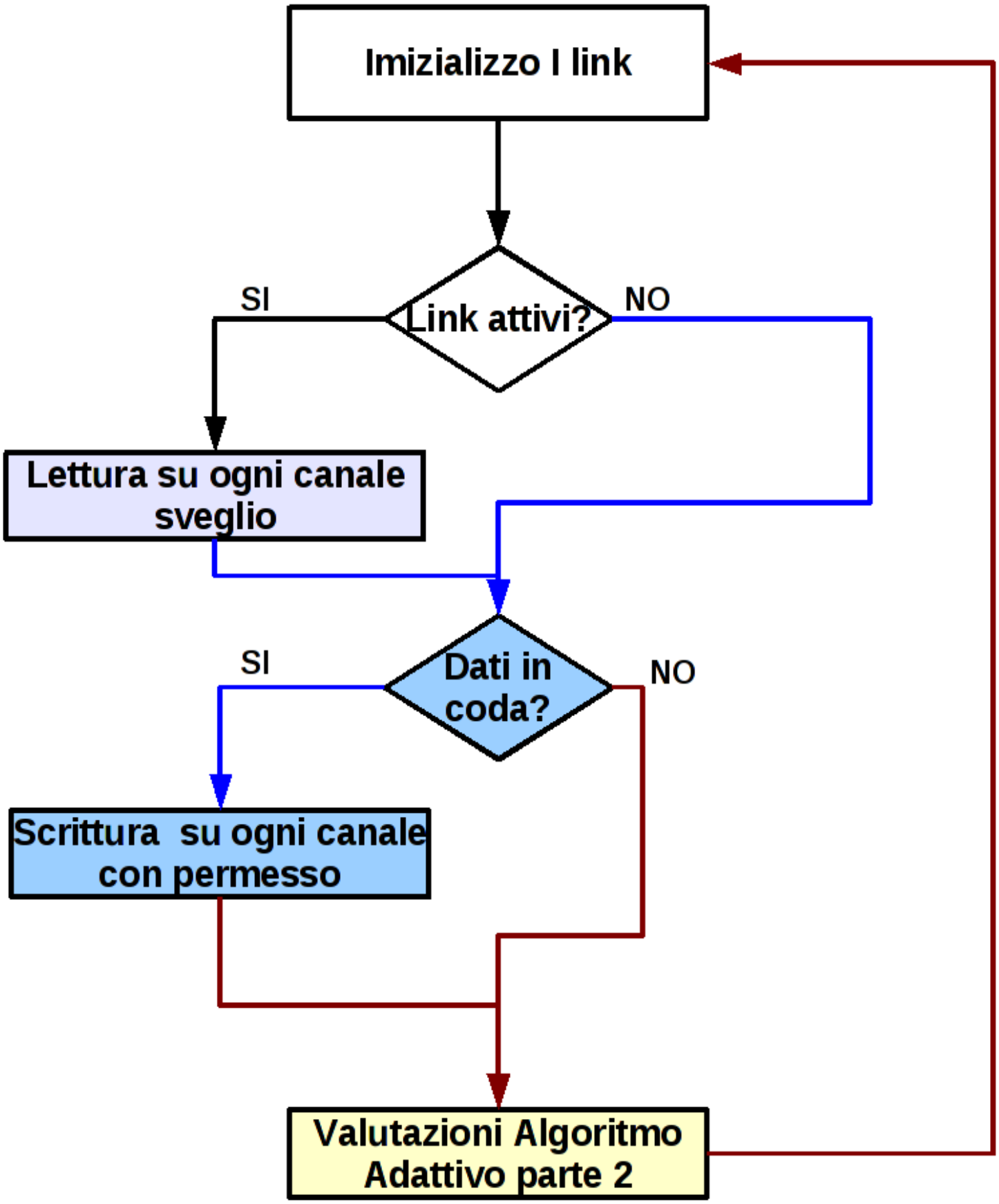
Implementazione del MultiHoming

1)L'API fornita da Net_Util fornisce un insieme di funzioni che operano su un singolo link di connessione

2)Il programma proxy client applica le funzioni del modulo Net_Util a 3 link di connessione implmentando così il multihoming

3)Il programma Proxy_client estende l'algoritmo adattivo introdotto da read_ack() eseguendo valutazioni globali su tutti i link di connessione

DIAGRAMMA SINTETICO DEL PROXY_CLIENT



Gestione simultanea di più link

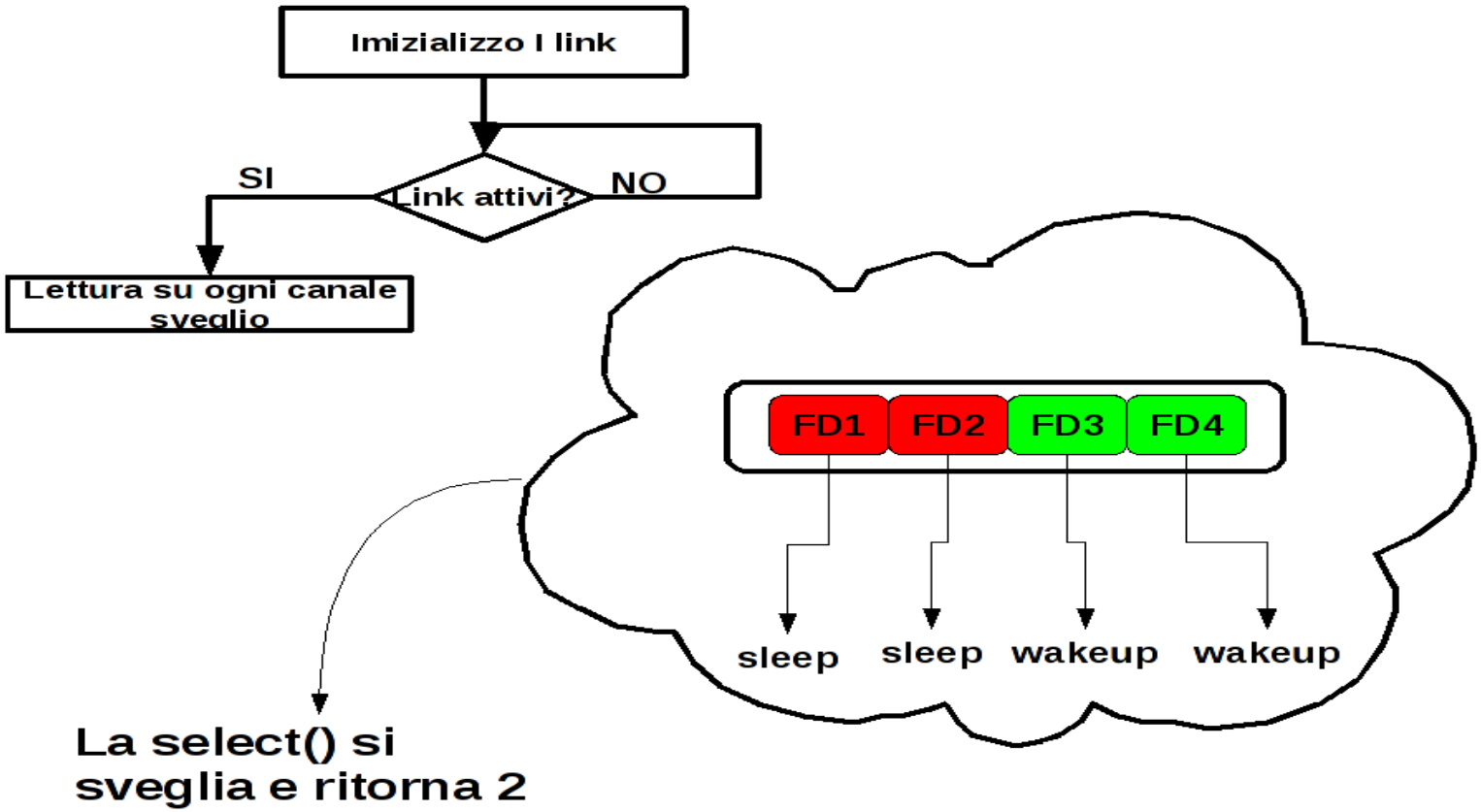
1)Il programma viene inizializzato con una lista contenente tutti i canali dei link.In ogni nodo è locato 1 canale

2)Si passano gli fd dei canali alla funzione select()

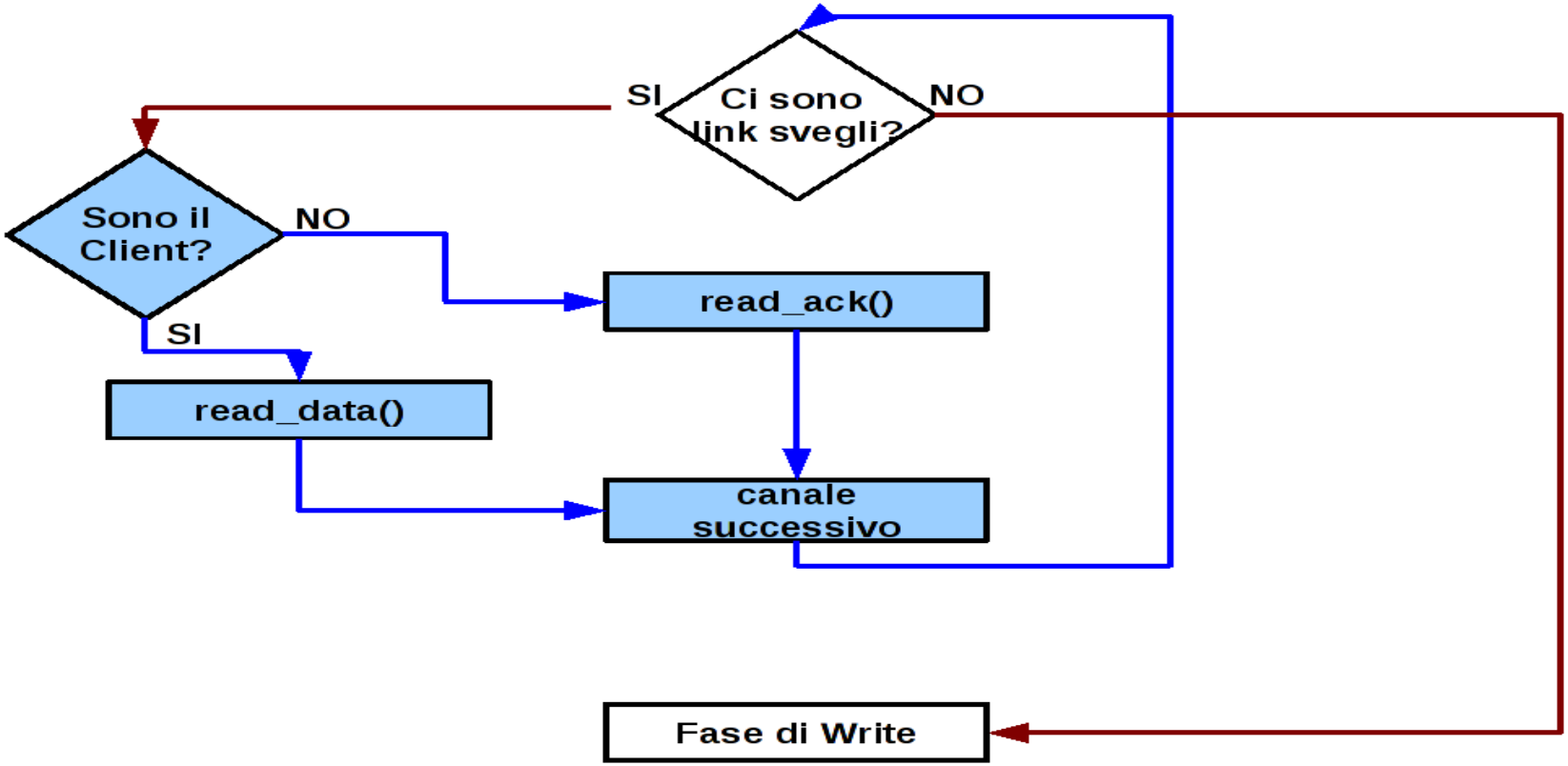
3)Mediante tale modalità è possibile gestire più canali simultaneamente

4)Quando ci saranno operazioni da svolgere sulla rete la funzione select() si sveglierà

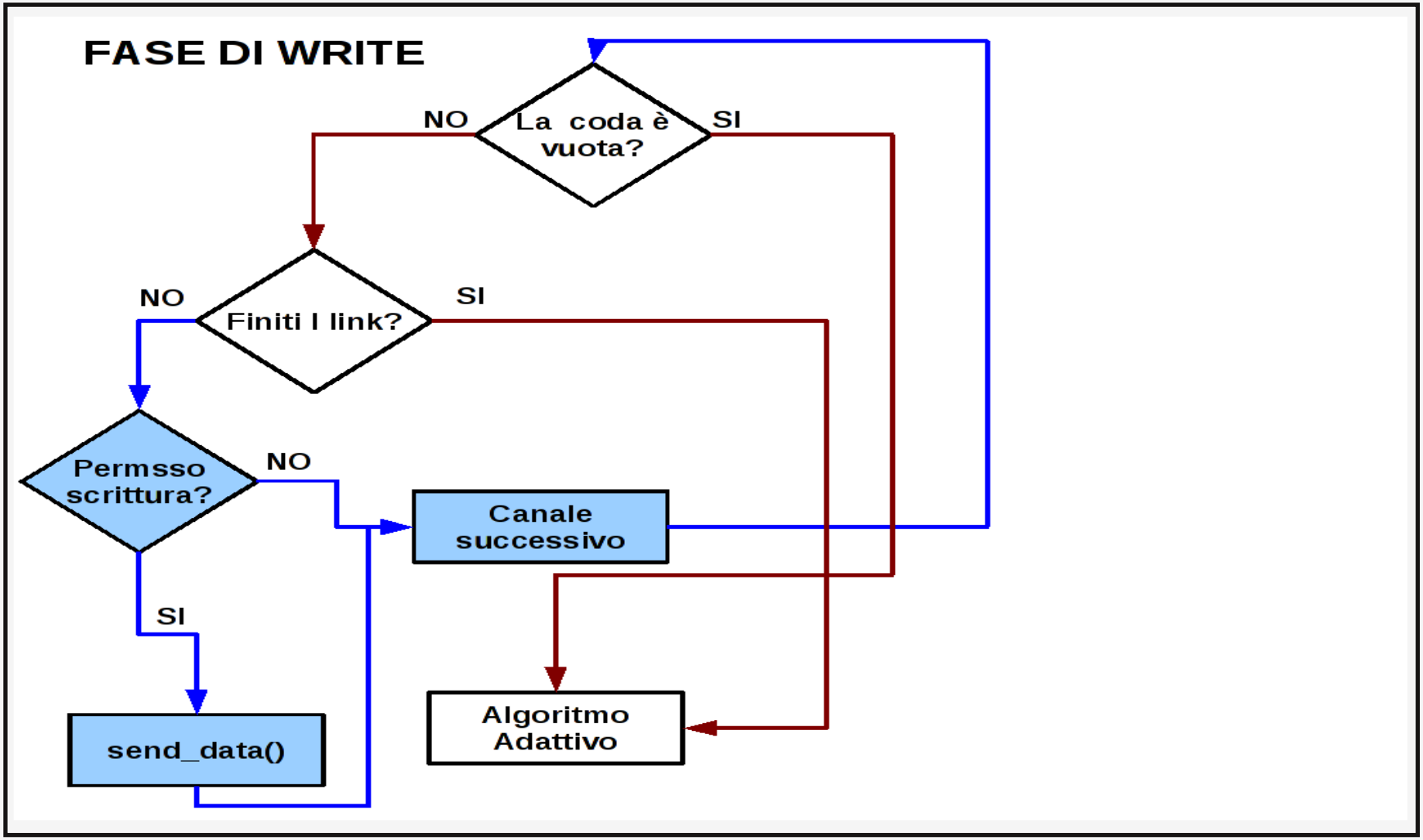
5)Quando si sveglierà la funzione select verranno controllati gli fd dei canali sulla lista


Imizializzo I link
Link attivi?
SI
NO
Lettura su ogni canale sveglio
FD1
FD2
FD3
FD4
sleep
sleep
wakeup
wakeup
La select() si sveglia e ritorna 2

FASE DI READ


Ci sono link svegli?
SI
NO
Sono il Client?
NO
SI
read_ack()
read_data()
canale successivo
Fase di Write

```
While(1)
{
    ..........................
    select();

    canale=(channel*) chan_list->item;

    if(FD_ISSET(canale->fd)) read_data();

    tmp_list=tmp_list->next;
    canale=NULL;

    while(tmp_list!=NULL)
    {
        canale=(channel*)tmp_list->item;
        if(FD_ISSET(canale->fd)) read_ack();
        canale=NULL;
        tmp_list=tmp_list->next;
    }

    ..........................................

}
```


FASE DI WRITE
La coda è vuota?
NO
SI
Finiti I link?
NO
SI
Permsso scrittura?
NO
SI
Canale successivo
send_data()
Algoritmo Adattivo

```
While(1)
{
    ..........................
    select();

    /*read session*/
    ..........................
    tmp_list=chan_list->next;

    if(is_empty_linked(ack_list)==0)
    {
        while(tmp_list!=NULL)
        {
            canale=(channel*)tmp_list->item;
            if(canale->success) send_data();
            canale=NULL;
            tmp_list=tmp_list->next;
        }
    }
    ..........................................

}
```

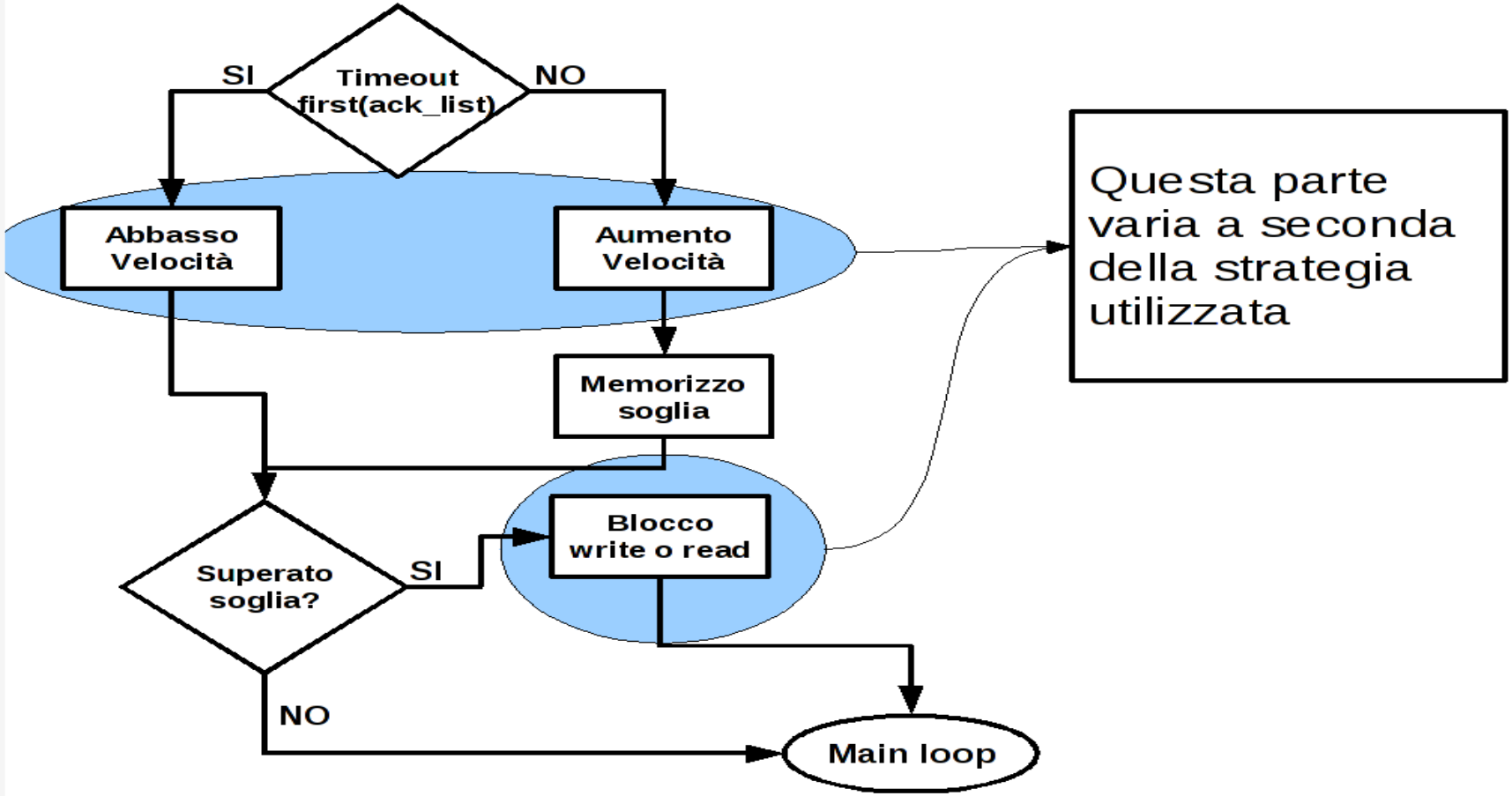
Seconda parte algoritmo adattivo

1)La prima parte dell'algoritmo andattivo è svolta dalla funzione read_ack() che modifica la grandezza massima del pacchetto inviabile mediante send_data()

2)Eseguita la fase di read e la fase di write è possibile fare un bilancio della situazione

3)Un valido strumento di analisi può essere la lista di pacchetti inviati

4)Mediante l' ack_list è possibile stimare il traffico in rete e meorizzare il valore di soglia globale precedente a un ritardo


Timeout
first(ack_list)
SI
NO
Abbasso
Velocità
Aumento
Velocità
Memorizzo
soglia
Blocco
write o read
Superato
soglia?
SI
NO
Main loop
Questa parte
varia a seconda
della strategia
utilizzata