CARS System
Relazione

1 - Breve descrizione generale

È stato realizzato un sistema di car sharing secondo il modello client-server, il quale è costituito da un processo server e da uno o più processi client, tutti multithreaded. Ogni client comunica con il server tramite un socket AF_UNIX dedicato. Il sistema prevede che il server raccolga ed elabori delle offerte e delle richieste di sharing, fatte dai suddetti client, e che provveda a recapitarle ai diretti interessati.

2 - Protocollo di comunicazione

Il protocollo di comunicazione è implementato all'interno della libreria *libsock.a* (avente header *comsock.h*), la quale mette a disposizione varie funzioni che consentono:

- al server, in particolare, di creare il file socket per l'accettazione delle connessioni (sul quale si connetteranno i client – ./tmp/cars.sck) e di mettersi in attesa di queste sullo stesso;

- al client, in particolare, di formulare una richiesta di connessione al server;

- ad entrambi, di scambiarsi messaggi prelevando le informazioni dalle strutture aventi tipo *message_t* (previste da specifica) e di impacchettarle come unica stringa secondo il formato:

<< Tipo_mexDim_buffer.ContenutoDelBuffer >>

dove, ad esempio, se il client “pippo”, avente password “alice”, volesse inviare MSG_CONNECT, si avrebbe:

<< C22.pippo\0alice\0pippo.sck >>.

Le stesse funzioni di *libsock.a* sono state ovviamente realizzate utilizzando varie system call per Inter Process Communication presenti nelle librerie standard.

NB: Ogni qualvolta, nel seguito della relazione, si farà riferimento a termini come impacchettamento, invio o ricezioni di messaggi, sarà sottinteso l'uso delle funzioni implementate in questa libreria per eseguire l'operazione.

3 - Processo Client (docars)

Il processo client, una volta lanciato (./docars username), provvede a richiedere da input la password all'utente e successivamente tenta di stabilire una connessione con il socket del server.
Se la connessione avviene con successo, invia il messaggio di login e attende il messaggio di esito (login riuscito oppure login fallito).

In caso di login corretto, vengono attivati i due thread che si occuperanno di due specifici compiti:

- *thread interattivo* (input thread):

provvede a raccogliere i comandi digitati in input, interagendo direttamente con l'utente, ciclando fino ad invocazione del comando *“%EXIT”* o a digitazione di *“CTRL-D” (lettura di EOF)*.
Per ottenere informazioni riguardanti sintassi e semantica di tutti i possibili comandi da digitare, l'utente può invocare, durante la fase interattiva, il comando *“%HELP”* .

Ogni comando digitato che risulterà **scorretto** al parsing, non verrà elaborato e verrà stampato un messaggio di errore in output. In caso, invece, di invocazione di un comando sintatticamente corretto riguardante offerte o richieste di sharing, il thread provvederà ad impacchettare quest'ultimo sotto forma di messaggio come previsto dal protocollo di comunicazione e, ignorandone la semantica, ad inviarlo sul socket dedicato.

Il parsing e l'invio dei messaggi sono realizzati, rispettivamente, tramite due specifiche funzioni presenti all'interno della libreria di supporto *libclient.a* (header *fun_client.h*).

La prima funzione, oltre ad assicurarsi che l'input rispetti la sintassi prevista, classifica il comando associandolo ad un carattere η, con η ∈ { 'p' (richiesta) , 'n' (offerta) , 'h' (aiuto) , 's' (uscita) , 'e' (errore) } , e restituisce la classificazione stessa.
La seconda provvede ad inizializzare, dipendentemente dal carattere di classificazione, la struttura che conterrà il messaggio e dalla quale verrà prelevato per l'impacchettamento.

- *thread ricezione* (receiving thread):

si occupa di ricevere e di stampare in output tutti i messaggi che il server invia sul socket dedicato.
Talvolta certi messaggi possono essere di rifiuto rispetto all'accodamento di una richiesta o di un offerta, anche se questa ha superato il parsing: in questi casi il client si accorge che la semantica della richiesta/offerta inviata era scorretta.

Il ciclo di ricezione si arresta in caso di “MSG_SHAREND”, messaggio che, quando inviato dal server, indica l'esaurimento delle richieste pendenti relative al client e quindi l'impossibilità di ricevere ulteriori share. In tal caso, e se il thread interattivo ha terminato la sua esecuzione, il client non ha alcun motivo di rimanere attivo.

L'intero processo client è dedito a corretta *terminazione* in caso di:

- inserimento di una coppia «username, password» differente rispetto a quella inserita in una precedente connessione con il server, condizione valida a patto che il server sia rimasto attivo durante l'arco di tempo trascorso tra le due connessioni;
- ricezione di segnali di tipo SIGINT o SIGTERM;
- perdita della connessione con il server;
- ricezione di ”MSG_SHAREND”;
- al verificarsi di eventuali errori riguardanti le allocazioni in memoria.

Inoltre, ignorando SIGPIPE, è stato possibile gestire la terminazione prematura del processo in caso di disconnessione del server.

Per quanto riguarda la ricezione di segnali di tipo SIGINT e SIGTERM, è stata preferita una gestione a livello di processo, mascherando l'espansione del segnale ad entrambi i thread.

Quindi, alla ricezione di uno dei due segnali di terminazione, il processo passa ad eseguire una handler che si occupa di cancellare entrambi i thread e di rimuovere il socket dedicato utilizzato per la comunicazione. Infine il processo provvede a ripulire l'ambiente dalle strutture dati allocate ed a raccogliere lo stato di terminazione di entrambi i thread.

4 - Processo Server (mgcars)

Il processo server (eseguibile invocando il comando “./mgcars file_mappa1 file_mappa2”) ha il compito di ascoltare le richieste di connessione da parte dei processi client e, per coloro per i quali il login è andato a buon fine, di ricevere ed elaborare richieste ed offerte di sharing, selezionare per ogni richiesta eventuali offerte che la soddisfano (calcolo degli *accoppiamenti*) e di inviare tali accoppiamenti (se ve ne sono) al client che ha effettuato la richiesta. Le connessioni vengono stabilite su un file socket (*cars.sck*) creato dal processo non appena viene lanciata l'esecuzione.

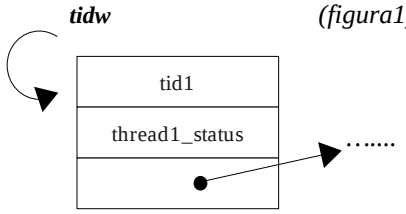
L'elaborazione è basata sulla creazione di una mappa composta da varie città e da collegamenti fra queste, ognuno avente una distanza in km. Viene naturale rappresentare tale mappa tramite un *grafo* orientato pesato, acquisibile tramite i due file che devono passati come argomento al momento dell'invocazione del processo. Tutte le funzioni di manipolazione del grafo, compresa quella di acquisizione e creazione della mappa, sono implementate all'interno della libreria *libcars.a* (header *dgraph.h*) .

Essendo multithreaded, il processo server è composto da:

- un *thread match*, che provvede a calcolare gli accoppiamenti e ad inviarli ai rispettivi client;

- un *thread worker* per ogni client connesso, il quale accetta/rifiuta il login e l'accodamento delle richieste/offerte inviatogli dal singolo client.

Per mantenere in memoria, istante per istante, gli identificatori e lo stato di esecuzione di tutti i thread worker coinvolti nell'esecuzione, il processo mantiene il riferimento ad una lista di elementi di tipo *tid_list*, definito tramite la struttura *threads:*



(figura1)

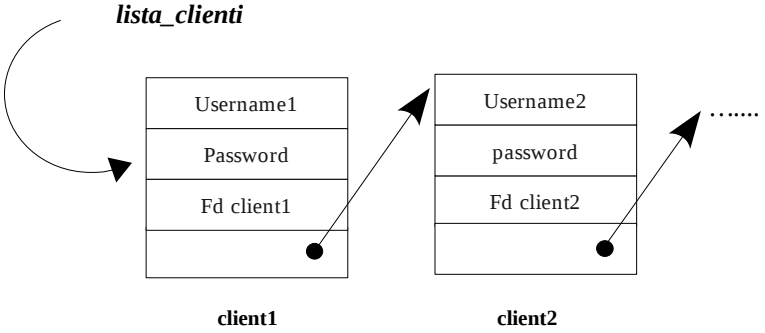
```
typedef struct threads {

 pthread_t tid;       /* tid */
 unsigned int active;  /* stato del thread */
struct threads *next;

}tid_list;
```

Passiamo ora ad esaminare il comportamento dei suddetti thread.

- *Thread Worker:*

Come preannunciato, viene creato un thread worker per ogni client che stabilisce una connessione con il server.
In particolare, tale thread si occupa per prima cosa di ricevere il messaggio di login dal client che ha causato la sua creazione, e in base a se la password coincide con quella usata in precedenti connessioni (se ce ne sono state), di inviargli la conferma (in caso di login effettuato con successo) o il rifiuto (altrimenti).

Le informazioni relative ad ogni client che ha effettuato almeno una connessione con il server, sono mantenute nella *lista dei client connessi*, implementata mediante la struttura *cliente (figura w1).*



(figura w1)

```
typedef struct cliente{

char *username;
char *password;
int fd_c; /* file descriptor socket */

}archive;
```

Dopodiché, se il login ha avuto successo, il thread worker inizia il ciclo di ricezione delle richieste e/o delle offerte di sharing mettendosi in attesa sul socket dedicato (creato dal client stesso prima di effettuare la connessione).
Per ogni messaggio ricevuto, il thread ne analizza la semantica: avrà comportamenti diversi in base alla tipologia.

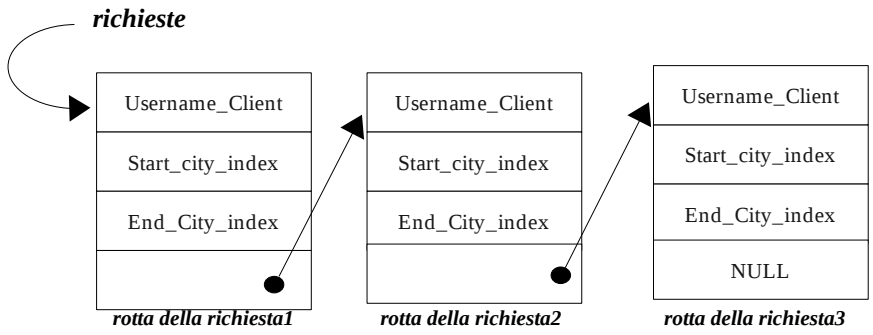
Se ha ricevuto un messaggio di:

- *offerta o richiesta di sharing*, controlla se le città che la compongono appartengono alla mappa: in caso affermativo, l'offerta/richiesta viene accodata e viene inviato un messaggio di conferma dell'accodamento sul socket dedicato; altrimenti viene inviata una risposta di rifiuto dell'accodamento.
  Soltanto in caso di richiesta accodata, viene incrementato un contatore locale che tiene traccia del numero di richieste accodate correttamente relative al singolo client.

- *MSG_EXIT*, indica che il client ha terminato la fase interattiva e che quindi non ci sono più messaggi da ricevere; in tal caso, il thread worker provvede a terminare il proprio ciclo di ricezione.

Quando si parla di accodamento di richieste e offerte, si intende l'archiviazione di queste in opportune liste, realizzate mediante le strutture dati che realizzano i tipi *req_queue* e *offer_queue (figure w2 e w3).*

(figura w2: lista delle richieste pendenti)

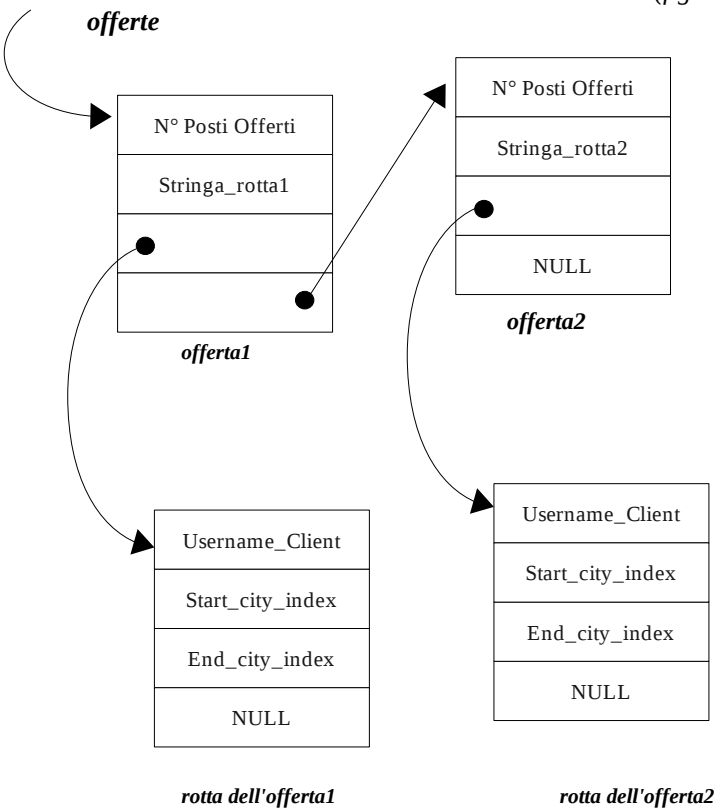


*typedef struct **città{***

char usernm[LLABEL];
unsigned int start;
unsigned int end;
*struct citta *next;*

}req_queue;

(figura w3: lista delle offerte pendenti)



*typedef struct **offer{***

*req_queue rotta**;* */* puntatore a struct in fig.w3 */*
unsigned int posti; / n° di posti offerti */*
char rotta_str; / rotta dell'offerta */*
struct offer next; / successivo */*

***}offer_queue**;*

Visto che l'utente ha esplicitamente richiesto di terminare la fase di accodamento delle richieste e/o delle offerte, il thread non ha motivo di rimanere attivo e quindi setta il proprio stato come “terminato correttamente” all'interno della lista dei tid .
Infine, se non risultano richieste accodate, il thread invia un messaggio di tipo “MSG_SHAREND” al rispettivo client indicandogli la possibilità di disconnessione (vista l'impossibilità di ricevere share) e termina; se, invece, risulta accodata almeno una richiesta, il thread si limita a terminare.

Le strutture dati e le funzioni utilizzate, sono presenti all'interno della libreria di supporto *libserver.a* (header *fun_server.h*).

- *Thread Match:*

elabora ogni 30 secondi (macro SEC_TIMER prevista da specifica) o immediatamente (elaborazione forzata in caso di ricezione di segnali di tipo SIGUSR1), le richieste pendenti presenti nella rispettiva lista.
L'elaborazione consiste nel calcolo degli accoppiamenti, basato sulla ricerca dei cammini minimi tramite l'algoritmo di *Dijkstra* e il calcolo delle rotte, funzionalità implementate nella libreria *libcars.a* (header *shortestpath.h*).

Quindi, una volta trascorsi i 30 secondi o ricevuto il segnale, il thread scorre l'intera lista delle richieste e per ognuna di queste verifica se esistono una o più offerte che la soddisfano:

- se la richiesta è completamente soddisfatta, ovvero dalla città di partenza si riesce ad arrivare alla città di destinazione grazie ad una o più offerte, ogni corrispondenza trovata viene scritta su un file di log (creato dal server a inizio esecuzione e sul quale opererà lo *script bash* **carstat** per calcolare varie statistiche) e inviata - come "MSG_SHARE" - all'utente che ha effettuato la richiesta. Ovviamente, in questo caso, la richiesta viene rimossa dalla rispettiva lista e alle offerte coinvolte viene decrementato il numero di posti offerti (se tale numero diventasse 0, verrebbe rimossa dalla lista anche l'offerta). Inoltre se, rimuovendo una richiesta relativa ad un determinato client, ci si accorgesse che questo ha terminato la sua fase interattiva e che nella rispettiva coda non vi sono altre richieste che gli riguardano, il thread provvederebbe ad inviargli un messaggio - di tipo "MSG_SHAREND" - vista l'impossibilità di ricevere ulteriori share .

- se, invece, la richiesta è parzialmente soddisfatta (esistono offerte che soddisfano soltanto una porzione della richiesta) o completamente insoddisfatta (non esistono offerte che la soddisfano), il calcolo degli accoppiamenti relativi alla richiesta in questione viene rimandato fino a prossima riattivazione del calcolo, lasciando le liste invariate e passando ad esaminare la richiesta successiva.

In ogni caso, il calcolo utilizza degli opportuni algoritmi definiti all'interno della libreria *libserver.a* (header *fun_server.h*). Considerata la non banalità di tali algoritmi, descrivo in seguito una generica iterazione eseguita dal thread match su una generica richiesta *req*.

Prima di ogni cosa, viene calcolata la rotta *Rreq*, relativa a *req*, secondo il formato previsto da specifica:

Rreq *città_partenza$città_1$.....$città_n$città_arrivo*

Dopodiché viene lanciato l'algoritmo principale, il quale permette di verificare se la richiesta *req* è soddisfatta e, se lo è, da quali offerte.
L'algoritmo è ovviamente realizzato in modo ricorsivo, in quanto esistono casi nei quali bisogna considerare sotto-richieste di *req*, e consiste nei seguenti passi:

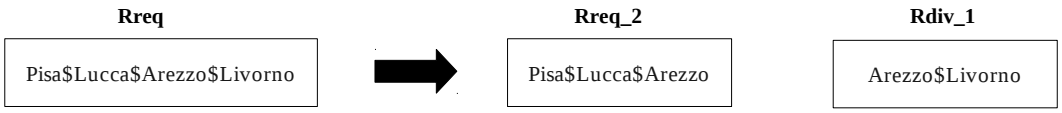
1. ricevuta in input la rotta *Rreq*, verifica se esiste almeno un offerta *off* all'interno della lista delle offerte pendenti, avente rotta *Roff*, che risulta identica a *Rreq* o che la contiene (*Rreq* ne è sotto-rotta).

2. Se *non ne esiste alcuna*, provvede :
    - a modificare la rotta *Rreq* dividendola dall'ultima città (ricavando la sotto-rotta *Rreq_2, figura m3*);
    - a marcare come "da soddisfare" la porzione di *Rreq* ottenuta dalla divisione (*Rdiv_1*), la quale è quindi diventata una sotto-richiesta di *req*, e sulla quale si dovrà ricorrere soltanto in caso di soddisfacibilità di *Rreq_2*. Marcare come "da soddisfare" *Rdiv_1*, vuole significare che verrà mantenuto un riferimento a, quest'ultima in un opportuna struttura dati *(struttura m1)*.
    - a ricorrere su *Rreq_2* (ritorna al passo 1);

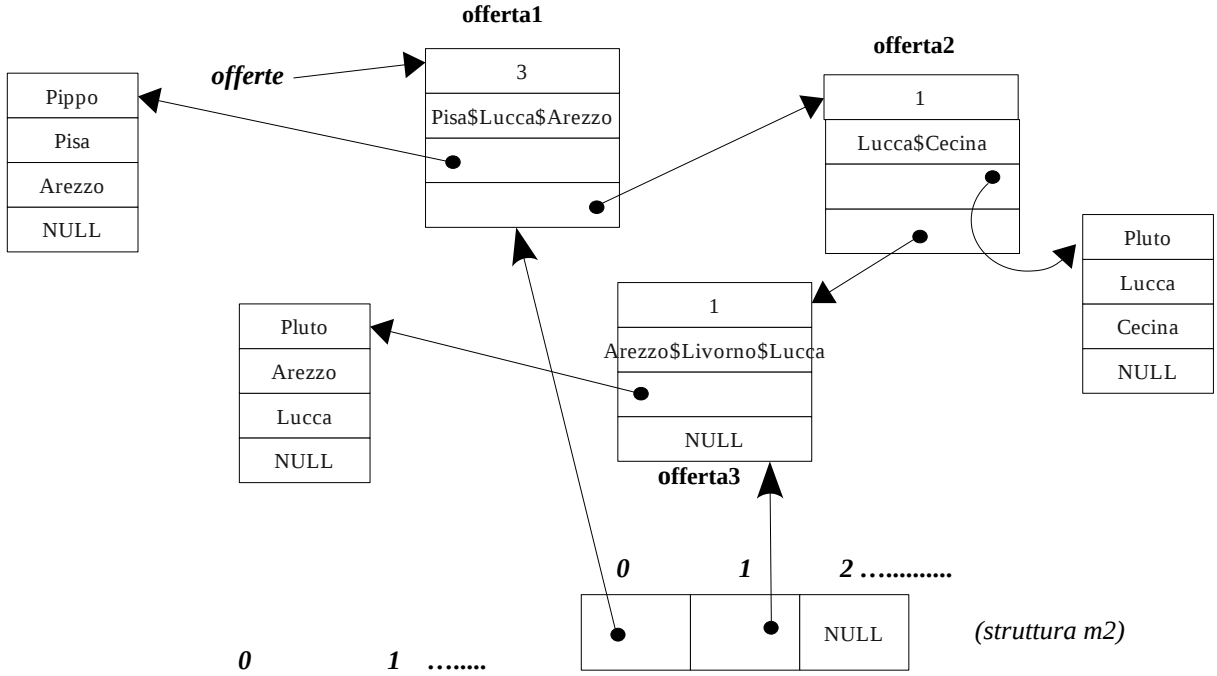
3. Se, invece, tale *offerta esiste*:
    - mantiene un riferimento ad *off* all'interno di un opportuna struttura dati la quale conterrà, al termine dell'esecuzione, tutti i riferimenti alle offerte utilizzate per soddisfare *req (struttura m2*) ;
    - se non sono state create sotto-richieste nelle chiamate precedenti, termina indicando la soddisfacibilità di *req;* altrimenti ricorre su queste.

Ovviamente l'algoritmo si arresta quando, ricorrendo più volte (ad esempio su *Rreq, Rreq_2, ….. , Rreq_i*), la sotto-rotta *Rreq_i* risulta contenere soltanto la città di partenza: l'eliminazione di quest'ultima porterebbe, ricorrendo un ulteriore volta, alla ricerca della rotta vuota all'interno della lista delle offerte. Si ricadrebbe, quindi, in un caso di insoddisfacibilità.

(figura m3): *Divisione della rotta dall'ultima città*



ES. Nella figura sottostante, è raffigurato lo stato dell'algoritmo a fronte di una richiesta da Pisa a Cecina, avente rotta «Pisa$Lucca$Arezzo$Livorno$Lucca$Cecina», dopo aver effettuato due sotto-chiamate: la prima su Pisa$Lucca$Arezzo e la seconda su Arezzo$Livorno$Lucca e subito prima dell'ultima sotto-chiamata (a Lucca$Cecina).



Tornando a ragionare a livello dell'intero processo, all'interno del server e delle funzioni da esso utilizzate sono state individuate varie regioni critiche che possono dar luogo a problemi di *race condition*. In particolare, queste possono essere considerate di 4 “tipologie” indipendenti tra loro:

1. regioni che operano sulle liste dei clienti connessi (mostrata in figura *w1*);
2. regioni che operano sulle liste delle offerte e delle richieste (mostrate nelle figure *w2* e *w3*);
3. regioni che operano sulle variabili di sincronizzazione utilizzate dal thread match per sincronizzare l'attesa/forzatura del calcolo degli accoppiamenti;
4. regioni che operano sulla lista dei tid (mostrata in *figura1).*

Ognuna di queste 4 tipologie ha associato un *mutex differente*, utilizzato proprio per forzare un esecuzione mutuamente esclusiva ma che può parallelizzarsi solo in caso di regioni critiche di tipologia differente.
In particolare per non complicare ulteriormente funzioni già di per se non banali, è stata preferita la soluzione basata sul non rilascio della mutua esclusione sulle liste delle offerte e delle richieste per l'intera durata dell'elaborazione eseguita dal thread match.

Intuitivamente e come previsto da specifica, il server deve sopravvivere in caso non ci fosse alcun client propenso a tentare una connessione. Per questo motivo è necessario ignorare eventuali ricezioni di segnali di tipo SIGPIPE (che causerebbero l'immediata terminazione del processo) e far sì che il server attenda fino a future connessioni.

È stato installato, inoltre, un gestore per segnali di tipo SIGUSR1, i quali possono essere inviati dall'utente in qualsiasi momento per forzare il calcolo degli accoppiamenti e far riprendere l'esecuzione in modo analogo a prima della

ricezione del suddetto segnale. Tale gestore provvede semplicemente a settare un opportuna variabile di sincronizzazione, la quale farà scattare l'immediato inizio dell'elaborazione.

Per quanto riguarda, invece, la corretta terminazione, il processo server in caso di ricezione di segnali di tipo SIGINT o SIGTERM si comporta in questo modo:

1. avendo installato un gestore per i due segnali sopra citati, provvede ad eseguirlo e quindi per prima cosa a cancellare tutti i thread worker ancora attivi coinvolti nell'esecuzione ed a raccoglierne lo stato di terminazione;

2. viene forzato il calcolo degli accoppiamenti per l'ultima volta, settando opportunamente il flag di sincronizzazione utilizzato dal thread match; al termine del suddetto calcolo viene raccolto lo stato di terminazione di quest'ultimo;

3. terminati correttamente tutti i thread, vengono chiusi i file aperti ad inizio esecuzione (files per la creazione della mappa, di log, socket) e viene ripulito l'ambiente dal socket creato per l'accettazione e dalle varie strutture allocate sullo heap. Infine il processo termina.

Per garantire un comportamento di questo genere, tutti i thread devono impostare la propria signal mask in modo da bloccare la ricezione dei due segnali di terminazione e dei segnali di tipo SIGUSR1, e terminare soltanto in caso di errore (es. invio messaggi, memoria, ecc) o di cancellazione da parte del processo.
Quindi, è stata preferita una gestione dei segnali “esterna” ai thread, i quali appunto si limitano ad essere cancellati in punti consistenti della loro esecuzione.

Inoltre per sincronizzare, in caso di ricezione di un segnale di terminazione, l'esecuzione del gestore a livello di processo con l'ultimo calcolo degli accoppiamenti da parte del thread match prima della terminazione, è stata introdotta *una condition variable associata al mutex riguardante i flag di sincronizzazione del match (di tipologia 3,* volendo utilizzare la notazione precedentemente introdotta*).*

Operando in questo modo, il processo attende sul mutex finché il calcolo degli accoppiamenti non è stato portato a termine, e quindi fino a segnalazione del thread match stesso. Ricevuta la segnalazione, il gestore cancella il thread.

5 - Calcolo di statistiche (script carstat)

Il logfile scritto con i vari accoppiamenti calcolati durante l'esecuzione del server, può essere passato come argomento dello ***script bash carstat*** (./carstat [opzioni] logfile), il quale dipendentemente dalle opzioni inserite calcola un diverso tipo di statistica riguardante il numero di richieste ed offerte presenti nel logfile.
Si può scegliere di calcolare le statistiche relative agli accoppiamenti che interessano:

- una determinata città di partenza (opzione -p CITTÀ);
- una determinata città d'arrivo (opzione -a CITTÀ);
- un certo utente (opzione -u Username);
- la combinazione delle tre opzioni suddette.

L'output è costituito da stringhe del tipo “username:num1:num2”, le quali stanno ad indicare che per l'utente avente *username* sono state utilizzate *num1* offerte e soddisfatte *num2* richieste di sharing.

6 – Conclusioni ed Osservazioni

- Le due librerie dedicate per server e client (rispettivamente *libserver.a* e *libclient.a*) sono state realizzate con l'intento di privilegiare l'astrazione, di rendere più leggibile il corposo codice e di nascondere gli aspetti implementativi.
- All'interno della relazione, è stato descritto in dettaglio soltanto l'algoritmo di calcolo degli accoppiamenti, il quale è l'unico che merita una spiegazione per incentivarne la comprensione. Tutti gli altri algoritmi (aggiunta e rimozione di elementi da liste, parsing, generiche iterazioni dei threads, switch per la decisione del tipo del messaggio da inviare, rimozione di elementi dagli array ecc) sono stati descritti in modo più astratto possibile rispetto all'implementazione, poiché risultano di immediata comprensione alla lettura del codice.
- Per tutti i tipi di segnali non citati nel corso della relazione, la gestione da parte dei processi resta quella di default.

- Tutte le funzioni eseguite dai thread, e quindi tali che possano essere interrotte tramite la cancellazione, presentano al loro interno varie installazioni di cleanup handler, gestite a pila ed utilizzate per evitare eventuali memory leak.

- Allo scopo di classificare in modo preciso i vari stati di terminazione dei thread, sono state definite delle *enum* le quali definiscono dei valori riconosciuti dalle funzioni “classify” del server e dei client.

- Il motivo principale per cui è stato deciso di stampare tutto su stderr è dettato dal voler mantenere, in qualche modo, traccia dell'esecuzione del sistema.

- Le stampe eseguite all'interno dei gestori sono state eseguite tramite chiamate a “fprintf”, con la consapevolezza del dettaglio *non safe*, in quanto sono state bloccate le ricezione dei segnali che possono generare inconsistenze.

7 – Modalità di compilazione ed esecuzione

È possibile compilare ed eseguire i processi tramite il Makefile, il quale provvede ad eseguire tutte le operazioni necessarie per la corretta creazione dei file eseguibili.

In particolare, il Makefile permette di:

- creare le librerie:

  1. *libsock.a*, utilizzata per il protocollo di comunicazione, tramite il comando ***make lib3***;
  2. *libcars.a*, utilizzata per costruire la mappa e calcolare le rotte su questa, tramite ***make lib1***;
  3. *libserver.a*, da supporto al server, tramite ***make lib31***;
  4. *libclient.a*, da supporto al client, tramite ***make lib32***;

- creare gli eseguibili, linkando le opportune librerie, per:

  1. il *server*, tramite ***make mgcars***;
  2. il *client*, tramite ***make docars***;

- lanciare i test, make test**;

Inoltre è stato creato un ulteriore test, il quale tende a sollecitare il calcolo degli accoppiamenti in seguito ad una doppia ricezione di segnali di tipo SIGUSR1 ed una doppia fase di launch di vari client; in particolare si è voluta testare la capacità del server nel continuare correttamente la propria esecuzione in seguito alla ricezione del primo segnale sopracitato, e quindi ad accettare (dopo la ricezione) ulteriori richieste/offerte tenendo conto delle precedenti scrivendo opportunamente sul file di log.

Per far ciò il *Makefile* è stato esteso con il target phony “test34”, quindi il test è eseguibile tramite il comando ***make test34.***

Il test prevede il file di check per testare il logfile (*mgcars.log.3.check*), 5 script client (**34.sh*) e lo script che esegue il test (*testpar2*). Tutti i file sono presenti nella directory “*Altri_test*” all'interno del tar consegnato.

8 – Bug e inconsistenze

Testando in uscita, tramite *Valgrind*, lo stato della memoria allocata, talvolta al termine dei processi si verifica la presenza di 5 blocchi classificati come “still reachable”, dovuti probabilmente alla terminazione dei thread utilizzando la “pthread_exit”.

Giuseppe Miraglia