`cars`: un semplice sistema di car sharing
Relazione finale del progetto

Nicola Corti - 454413
Corso di Laurea in Informatica (L-31) - Università di pisa

12 Luglio 2011

Sommario

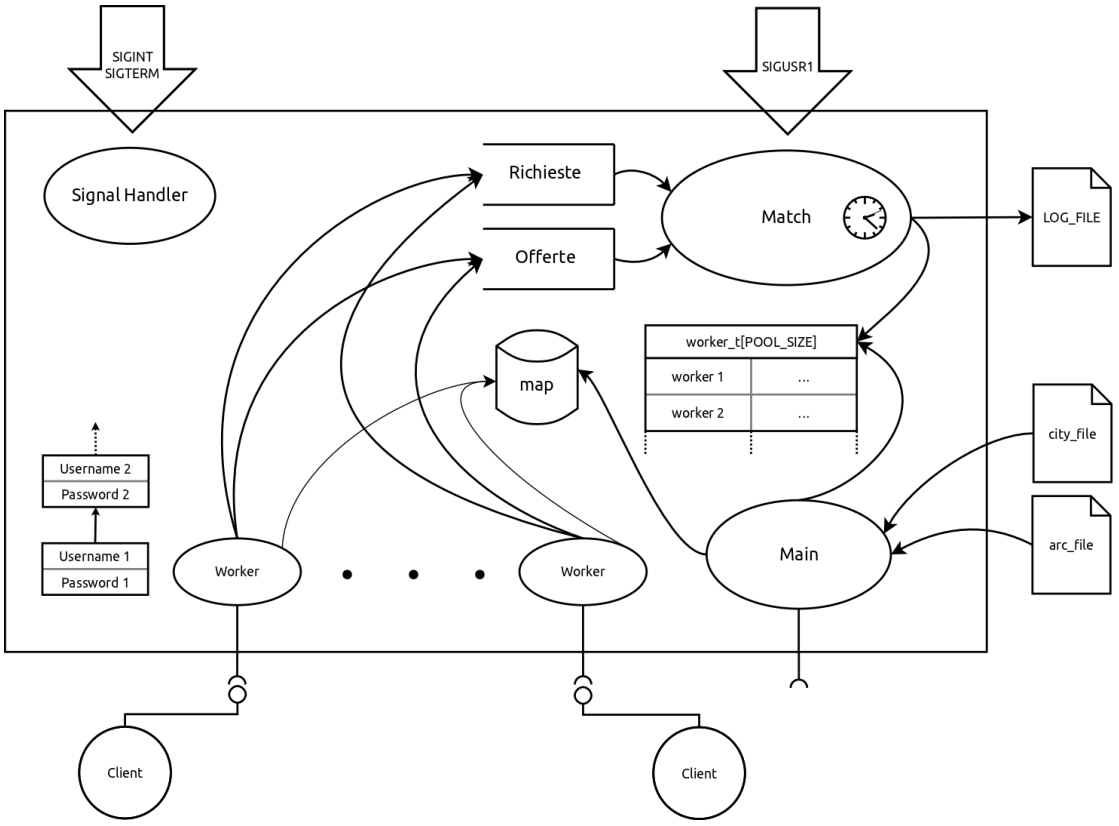
Questa relazione ha lo scopo di illustrare i dettagli implementativi del sistema di car sharing `cars`, nel caso in cui si fosse invece interessati soltanto alla guida per l'uso si può consultare la documentazione dell'utente, in allegato al kit d'installazione

Indice

1 Il Server - mgcars

Il server `mgcars` rappresenta il nucleo di tutto il sistema `cars`.

Una rappresentazione schematica del server può essere espressa mediante il diagramma seguente:



Al suo avvio il server esegue in ordine i seguenti passi:

- Cerca di caricare in memoria un grafo orientato partendo dai file delle città e degli archi che sono dati in input.
- Cerca di aprire il file di log specificato dalla macro LOG_FILE_NAME.
- Prova a far partire i thread worker.
- Prova a far partire il thread gestore dei segnali.
- Prova a far partire il thread match.
- Apre una socket in ascolto all'indirizzo specificato dalla macro SOCKET_PATH.

Una volta che tutti questi passi sono stati eseguiti correttamente il server è pronto per ricevere connessioni in entrata.

Per quanto riguarda i thread si è deciso di provare ad avviarli, e nel caso non vi fossero sufficienti risorse attendere DELAY secondi, e riprovare per MAX_TRY volte. Nel caso comunque che non si riescano ad avviare dei thread per mancanza di risorse, si prenda in considerazione l'idea di cambiare i parametri con cui esegue il server o di cambiare elaboratore.

1.1 Il thread main

Da questo momento in poi il thread main resta in ascolto sulla socket da poco aperta di eventuali connessioni e le schedula ad un thread fra quelli presenti nel pool.

Lo scheduling viene effettuato con politica *round robin*, grazie alla funzione findActivateThread, contenente un variabile statica che viene utilizzata per cercare nel pool un thread disponibile a servire un nuovo utente.

Con uno scheduling di questo tipo si è certi che il carico di lavoro viene ripartito in modo equo fra gli worker, e non sono presenti worker che rimangono dormienti per tutto il tempo dell'esecuzione del server.

Una volta trovato un thread che è in attesa lo si attiva, e gli si comunica il socket sul quale può mettersi in attesa di messaggi in entrata.

Nel caso in cui non sia possibile trovare un thread disponibile, si è deciso di chiudere il socket relativo al client appena connesso, e di non comunicare al client che il server ha raggiunto il suo carico massimo; così da tentare di sventare attacchi del tipo DoS (l'utente, non essendo a conoscenza del fatto che il server è sovraccarico, potrebbe imputare il problema ad altri fattori).

1.1.1 Il pool di thread

Si è deciso di implementare gli worker mediante un pool di thread, la cui dimensione viene stabilita dalla macro POOL_SIZE.

È stata operata questa scelta, al fine di conferire maggiore stabilità al server, ed una maggiore adattabilità; dato che l'amministratore del server può definire il numero di thread da utilizzare in funzione dell'infrastruttura a sua disposizione.

Di fatto il pool di thread è un vettore di strutture `worker_t` così definite:

```
typedef struct worker{
    pthread_t tid;
    int in_sck;
    int out_sck;
    int active;
    pthread_cond_t cond;
    int sentreqs;
} worker_t;
```

tid Rappresenta il Thread Identifier dello specifico thread, utilizzato per interagire con il singolo thread (invio di segnali, join, etc...)

in_sck Socket in entrata, viene settata dal thread main all'atto dell'attivazione del thread. Quando il thread non è attivo vale -1.

out_sck Socket in uscita, viene settata dal singolo worker, nel momento in cui ci si riesce a connettere sul socket specificato dall'utente. Quando il thread vale -1, a meno che il server non abbia avviato la procedura di spegnimento e il numero delle richieste inviate sia maggiore di 0. In tal caso il thread match, dopo che avrà calcolato gli ultimi accopiamenti, invierà i dovuti messaggi e chiuderà le socket in uscita.

active Bit che indica se il singolo thread è attivo, oppure se è in attesa di una nuova connessione.

cond Variabile di tipo `pthread_cond_t` su cui il thread resta in attesa, e serve per risvegliarlo e per farlo partire.

sentreqs Numero di richieste inviate (non necessario ai fini del pool di thread, ma viene utilizzato dal thread match per inviare i messaggi di SHARE_END).

Tutto il pool viene gestito in mutua esclusione grazie alla variabile `mux_pool` di tipo `pthread_mutex_t`, dato che tutti i thread accedono al pool per leggere e per modicare i dati.

1.2 Il thread worker

All'avvio ogni thread worker si mette in attesa di essere attivato dal thread main, restando bloccato su `waitForActivation`.

Una volta attivato attende di ricevere il messaggio di login sul socket indicato nella sua struttura; se il messaggio è un messaggio di login valido (come definito dal protocollo di comunicazione) allora ne estrapola i dati necessari per verificare il login.

Se il messaggio non è un messaggio di login valido, il thread torna in attesa (invocando la funzione `goingBackToSleep` che lo riporta in stato di attesa. Anche in questo caso si è deciso di terminare immediatamente la comunicazione col client, percé probabilmente si tratta di un client che non sta funzionando a dovere, o peggio, di un client manomesso, che potrebbe inficiare la corretta esecuzione del server.

1.2.1 Autenticazione dell'utente

Una volta estrapolati i dati dell'utente si passa alla sua autenticazione: I dati degli utenti vengono salvati in chiaro in una lista linkata di questo tipo:

```
typedef struct credential{
    char username[LUSERNAME];
    char password[LUSERNAME];
    int active;
    struct credential *next;
} credential_t;
```

username Nome utente

password Password (salvata in chiaro[1])

active Bit che indica se un utente è già loggato presso un worker in questa sessione del server.

next Puntatore alla prossima struttura

Anche questa struttura viene acceduta in mutua esclusione grazie alla variabile `mux_pool`.
I possibili esiti della procedura di autenticazione dell'utente `isAlreadyUser`, possono essere:

- **Esiti positivi**
  - Utente non presente, allora si provvede ad inserire una nuova occorrenza nella lista degli utenti, a settare l'utente come attivo.
  - Utente non attivo, password corretta, allora si provvede a settare l'utente come attivo.
- **Esiti negativi**
  - Password errata, la password non combacia con quella utilizzata nelle precedenti sessioni, si invia dunque il messaggio di errore.
  - Utente già attivo, l'utente risulta già loggato presso un altro worker quindi si invia il messaggio di errore.
  - Utente o Password invalidi, indica che la stringa del login è malformata (probabilmente è presente un malfunzionamento nel client).

Nei casi di esito positivo, si invia il messaggio di conferma e si inizia la sessione di ricezione messaggi con il client. Nei casi di esito negativo il thread torna in fase di attesa.

1.2.2 Sessione di ricezione messaggi

Durante la sessione di ricezione messaggi, il worker sta in attesa sulla socket in entrata di messaggi da parte del client.

La correttezza sintattica dei messaggi viene controllata dal client (in modo da non far fare ulteriori elaborazioni al server), per cui si assume che i messaggi in arrivo dal client siano corretti. Dobbiamo invece verificare che le località inviate siano presenti nella mappa caricata sul server.

Ciò è controllato dalla procedura `checkValidMessage`, che in base al valore ritornato permette di capire se si puó accettare l'offerta/richiesta o meno (nome della città di partenza/arrivo non presente, troppo lungo, problema con il numero dei posti, etc...).

Nel caso in cui l'esito sia positivo si provvede ad accodare la richiesta o l'offerta nella rispettiva coda (e ad incrementare il numero delle richieste ricevute, se necessario).

La sessione termina se cade la connessione, se il client invia il messaggio di exit, oppure se è stato ricevuto il segnale di terminazione; in tali casi si provvede a sloggare l'utente, e a tornare in modalità di attesa.

1.3 Il thread match

Il thread match sta costantemente in attesa su una `sigtimedwait` in attesa di un segnale SIGUSR1. Nel caso in cui il segnale non arrivi entro SEC_TIMER secondi e NSEC_TIMER nanosecondi, il thread match parte ugualmente ed inizia a cercare degli accoppiamenti.

Le richieste e le offerte pervengono al match mediante due code distinte. Si è deciso di utilizzare tale struttura dati per servire le richieste con una politica *FIFO* (le prime richieste che sono pervenute sono le prime che saranno servite).

Le code sono implementate mediante delle liste linkate, dove gli inserimenti vengono effettuati in coda, e le estrazioni in testa, con delle strutture di questo tipo:

[1] Si sarebbe potuto salvare l'hash della password ed un eventuale seme, e controllare mediante questi l'identitá dell'utente (in un modo similare a quanto avviene nei sistemi UNIX/Linux), ma non era questo l'obiettivo del progetto.

```
typedef struct reqoff{
    int type;
    char user[LUSERNAME];
    char depart[LLABEL];
    char arriv[LLABEL];
    int place;
    int share_fd;
    char *path
    struct reqoff *next;
} reqoff_t;
```

type indica se si tratta di un'offerta o di una richiesta.

user utente che ha inviato la richiesta/offerta.

depart località di partenza.

arriv località di arrivo.

place numero dei posti, nel caso delle offerte.

share_fd indice della socket dove si deve inviare la risposta (nel caso delle richieste).

next puntatore al prossimo elemento della coda.

path puntatore ad una stringa che indica il cammino minimo fra la località di partenza e la località di arrivo.

La stringa path rappresenta il cammino minimo fra le due località ed è generata dagli worker, nel momento in cui inseriscono nelle code una nuova richiesta/offerta, cosí da non rendere piú oneroso il carico del worker.

Le due code vengono gestite in mutua esclusione mediante due variabili `pthread_mutex_t`. Si è deciso di gestirle con due variabili distinte, in modo che l'aggiunta di una richiesta e di un'offerta possa essere eseguita in modo concorrente.

1.3.1 La ricerca di accoppiamenti

Una volta avviata la fase di match, il thread acquisisce la mutua esclusione sulle due code, e sul pool di thread[2].

Quindi il thread inizia a cercare degli accoppiamenti per ogni elemento della coda richieste, invocando la funzione `findAssociation`. Tale funzione utilizza a sua volta la funzione `findRec` che indica se esiste o meno un possibile percorso che riesca a soddisfare la richiesta[3].

La funzione `findRec` è una funzione ricorsiva che opera sulle stringhe del path e cerca di costruire un percorso per soddisfare la richiesta. Nel caso in cui riesca a costruirlo lo salva in una lista di tipo assoc_t.

```
typedef struct assoc{
    int city_found;
    int begin_from;
    reqoff_t *found_off;
    struct assoc *next;
} assoc_t;
```

city_found rappresenta il numero di città che sono state trovate che corrispondono in quella offerta.

begin_from rappresenta il numero ci città da cui parte il percorso trovato (vedi esempio)

found_off rappresenta il puntatore all'offerta che corrisponde a questa parte di percorso.

next puntatore al prossimo passo nel percorso.

Esempio 1: Se l'offerta relativa alla prima parte del percorso ha il seguente path: `PISA$LUCCA$ALTOPASCIO$PISTOIA` e i valori delle variabili sono `city_found = 2` e `begin_from = 1` allora il percorso effettuato in questo passo sarà `LUCCA$ALTOPASCIO`.

La procedura fa uso delle funzioni definite nell'header `match.h` e implementate nel sorgente `match.c`

Se la ricerca del percorso ha avuto esito positivo, si provvede a scrivere sul file di log gli accoppiamenti, ad inviare i messaggi di SHARE, e ad aggiornare la lista delle offerte.

[2] In questo modo gli worker e il main restano bloccati in attesa della terminazione della fase di match; se infatti si accettassero ulteriori richieste/offerte il thread match potrebbe trovari a lavorare in uno stato inconsistente.

[3] Tale esito è rappresentato dai valori TRUE o FALSE, che sono di fatto delle macro con valori rispettivamente 1 e 0, settate al fine di rendere più leggibile il codice

Implementazione alternativa: il vettore dei precedenti Si sarebbe potuto implementare la ricerca utilizzando il vettore dei precedenti restituito dall'algoritmo di *Dijkstra*, ma si è deciso di utilizzare la ricerca su stringhe, malgrado la maggiore difficoltà, perché i percorsi generati tendono ad avere la prima tappa più lunga possibile.

Una ricerca sul vettore dei precedenti avrebbe generato percorsi con l'ultima tappa più corta, essendo una ricerca effettuata all'indietro.

Generando percorsi con la prime tappe più lunghe si cerca di "aiutare" gli utenti facendoli avvicinare il più possibile alla loro destinazione, in modo che se gli offerenti che avrebbero dovuto accompagnarli a destinazione dovessero venire meno all'impegno stabilito, la distanza per raggiungere la destinazione sia la minore possibile (favorendo l'uso di mezzi di trasporto pubblico).

1.3.2 L'aggiornamento delle offerte

Se la macro UPDATE_OFFER risulta definita, il match scomporrà ogni richiesta che compone una parte del percorso in modo da poter sfruttare a pieno tutti i posti disponibili di ogni offerta. Un esempio chiarirà meglio il concetto:

Esempio 2: se l'offerta relativa avesse avuto 4 posti disponibili, sarebbe stata scomposta in altre 3 offerte, una con path `PISA$LUCCA$ALTOPASCIO$PISTOIA` e 3 posti, e altre 2 offerte da 1 posto, relative a `PISA$LUCCA` e `ALTOPASCIO$PISTOIA`.

1.3.3 Fase finale

Dopo aver elaborato tutte le richieste, il thread match invia il messaggio di SHARE_END a tutti i client che hanno inviato almeno una richiesta (tale informazione la può ricavare dai dati nella struttura del pool di thread).

La lista delle richieste pendenti viene completamente liberata.

Se risulta settata la macro FREE_REQ_LIST la lista delle offerte viene liberata, altrimenti le richieste vengono mantenute[4].

1.4 Il thread signal_handler e la terminazione del server

Il thread gestore dei segnali resta costantemente in attesa (mediante una `sigwait`) di un segnale in arrivo. A questo thread vengono indirizzati tutti i messaggi in arrivo al processo, tranne il segnale SIGUSR1 che viene recapitato al thread match.

Per i segnali di errore di sistema (SIGSEGV, SIGILL, etc...) è stato installato un gestore che gestisce la terminazione immediate e visualizza un messaggio di errore critico su *standard error*[5].

Il segnale di SIGPIPE viene ignorato per non far terminare il processo.

I segnali di SIGINT e SIGTERM provocano invece l'inizio della fase di terminazione del processo, il thread signal_handler esegue allora in ordine i seguenti passi:

1. Setta ad 1 la variabile globale `received_signal` che indica che sta iniziando la procedura di spegnimento del server.
2. Risveglia tutti gli worker e gli invia un segnale SIGUSR2
3. Setta a 2 la variabile globale `received_signal` ad indicare che gli worker sono terminati e il match può eseguire per l'ultima volta la procedura di ricerca delle associazioni
4. Invia il segnale di SIGUSR1 al thread match per sbloccarlo.
5. Attende la terminazione del thread match.
6. Chiude i socket che eventualmente sono rimasti aperti dato che il thread match li ha utilizzati per inviare gli ultimi messaggio di SHARE e di SHARE_END.
7. Invia un segnale di SIGUSR2 al thread main, per sbloccarlo nel caso fosse rimasto bloccato su una accept.
8. Il thread signal_handler termina.

Nel frattempo il thread main si è messo in attesa della terminazione del thread signal_handler, appena è terminato esegue le ultime operazioni di pulizia del sistema:

- Libera la memoria allocata dalla lista degli utenti.
- Chiude il *file descriptor* relativo al socket sul quale il main accettava le connessioni entranti.
- Rimuove il file che rappresenta la socket da disco.

[4] ATTENZIONE: se si decide di scompattare l'offerta e di non liberare la lista delle offerte si potrebbero avere dei rallentamenti sul server nel lungo termine, dato che le offerte risultano parecchio frammentate; ciò potrebbe rendere più lungo il processo di ricerca di un percorso (rimarrebbero infatti tanti piccole offerte da un posto con path molto brevi).

[5] In modo da evitare la visualizzazione di messaggi sgradevoli su terminale (quali ad esempio Segmentation Fault, etc...)

- Libera la memoria allocata dal grafo orientato.
- Chiude il *file descriptor* relativo al file di log

Se tutti questi passi vengono eseguiti correttamente, anche il thread main termina, e il processo ritorna con 0 come valore d'uscita[6]. In tutti gli altri casi il processo ritorna con uno stato diverso da 0.

2 Il Client - docars

Il client `docars` rappresenta l'interfaccia nei confronti dell'utente del sistema `cars`.

Il client consta di due thread, il thread `main` a cui è demandata la relazione diretta con l'utente, l'accettazione di messaggi, la loro verifica e l'invio al server, ed il thread `listener` che si occupa di riceve i messaggi in ingresso da parte del server.

Il client viene avviato avendo come unico parametro l'username dell'utente che desidera collegarsi al sistema. Il client provvede quindi a chiedere all'utente la sua password e a generare un nome univoco per la sua socket; si è deciso di generare il nome della socket concatenando il nome utente al PID del processo, così che il nome della socket sia univoco. [7]

Il client cerca dunque di collegarsi al server, e gli invia i dati relativi all'utente. Successivamente attende una connessione in ingresso sul socket il cui nome è stato generato precedentemente. Cerca quindi di leggere l'esito della connesione al server: se l'esito è positivo l'utente è riuscito ad autenticarsi correttamente, inizia la sessione interattiva (gestita dal thread `main`) e la ricezione dei messaggi dal server (gestita dal thread `listener`) , altrimenti l'esecuzione del client termina visualizzando il messaggio di errore inviato dal server.

2.1 La sessione interattiva

Durante la sessione interattiva il thread `main` presenta il prompt (definito dalla macro PROMPT) e si mette in attesa di dati su *standard input* mediante la funzione di libreria `fgets`[8]; provvede dunque a verificarne la sua correttezza mediante la funzione `messageParser` e a preparare eventualmente un messaggio da inviare al server.

Si è deciso di verificare la correttezza sintattica delle richieste/offerte nel client, come già detto in precedenza, per non sovraccaricare inutilmente di lavoro il server.

Se si tratta di un messaggio valido di Richiesta o di Offerta, si provvede ad inviarlo al server, e ci si mette in attesa dell'esito dal server.

Se si tratta di un messaggio di HELP o di un messaggio sintatticamente scorretto, si provvede a visualizzare dei messaggi informativi a schermo, e si torna in attesa di un input dall'utente.

Se si tratta di un messaggio di EXIT oppure l'utente ha inviato un EOF, si invia un messaggio di EXIT al server e si conclude la sessione interattiva. Dopo di che, se sono state inviate richieste di recente, si attendono eventuali messaggi di SHARE_END da parte del server, altrimenti il client termina la sua esecuzione.

2.2 Il thread listener

Il thread `listener` ha come compito primario quello di restare in attesa di messaggi in arrivo dal server. I messaggi che possono arrivare dal server sono di 2 tipi:

- Messaggi di OK e di NO, che vengono ricevuti in seguito a messaggi di richieste od offerte inviate dal client stesso. Questi messaggi vengono considerati di tipo *sincrono*, dato che possono essere ricevuti soltanto in seguito a determinati eventi determinati a priori. Inoltre senza aver ricevuto un messaggio di questo genere, il thread `main` non può inviare ulteriori messaggi.
- Messaggi di SHARE e di SHARE_END, che possono essere ricevuti in un qualsiasi momento, purché in un passato prossimo sia stata inviata una richiesta dal client e tale richiesta sia stato seguita da un messaggio di OK. Tali messaggi vengono considerati *asincroni* dato che possono essere ricevuti in ogni momenti.

[6] In questo modo il programma risulta conforme alle convenzioni bash (dove 0 indica uno stato d'uscita corretto) e può essere utilizzato all'interno di script bash senza problemi.

[7] Tale assunzione non è totalmente vera, potrebbero infatti crearsi interferenze se rimanesse sul file system un file relativo ad una socket di un client (a causa per esempio di un suo arresto improvviso), e lo stesso utente riavviase il client ottenendo lo stesso PID precedente. Questo evento ha comunque una possibilità comunque molto ridotta di verifirsi, tale da poterci permettere di definire questo nome di socket così generato, univoco.

[8] Nel caso in cui il messaggio si più lungo di BUFF_SIZE si segnala un errore e si provvede a rimuovere i caratteri in eccesso da *standard input*.

2.2.1 Mutua esclusione fra i due thread

Per gestire una corretta interazione fra client e server sono state definite alcuni flag globali e alcune variabili per gestire la mutua esclusione:

`message_sem` Se vale 0, vuol dire che il `main` può inviare un messaggio al server. Una volta che il messaggio è stato inviato, tale flag viene settato a 1. Verrà successivamente risettato a 0 dal `listener` nel momento in cui si riceve un messaggio di OK o di NO. Per gestire la mutua esclusione di questa variabile vengono utilizzate la variabili `mux` e `cond`, rispettivamente di tipo `pthread_mutex_t` e `pthread_cond_t`; il thread `main` si mette infatti in attesa su `cond` che il flag diventi 1 per potere inviare un nuovo messaggio.

`sharend_sem` Il suo valore di default è 0, viene settato a 1 da `main` se si è inviata un'offerta, e successivamente viene settato a 2 da `listener` se la risposta a tale richiesta è stata OK. Serve per capire se si deve attendere un messaggio di SHARE_END o meno da parte del server; una volta ricevuto il messaggio di SHARE_END tale flag viene resettato a 0. Il tipo di questa variabile è `volatile sig_atomic_t` in modo da poter trascurare l'utilizzo di variabili ausiliarie (`mutex`) per la gestione della mutua esclusione.

`end_prompt` Il suo valore di defautl è 0, vale 1 se è terminata la sessione interattiva del `main`. Ciò serve a segnalare al thread `listener`, che, se non vi sono più SHARE_END da ricevere può terminare la sua esecuzione, altrimenti si mette in attesa dell'ultimo SHARE_END, e successivamente termina la sua esecuzione. Anche questo flag risulta del tipo `volatile sig_atomic_t` per il motivo sopra indicato.

2.3 Terminazione del client

Per quanto riguarda la terminazione del client, si è utilizzato il meccanismo dell'*interrupted system call*. Per i segnali

Per i segnali di errore di sistema (SIGSEGV, SIGILL, etc...), proprio come è stato fatto nel server, è stato installato un gestore che gestisce la terminazione immediate e visualizza un messaggio di errore critico su *standard error*.

Il segnale di SIGPIPE viene ignorato per non far terminare il processo.

Per i segnali di SIGINT e SIGTERM è stato installato un gestore che setta ad 1 la variabile `received_signal`, e invia il segnale SIGUSR1 al processo. I segnali SIGINT e SIGTERM possono essere ricevuti soltanto dal thread `main`, che nel caso in cui fosse bloccato su una *system call* bloccante risulta sbloccato; lo stesso discorso vale per il segnale SIGUSR1 rispettivamente al thread `listner`.

Il thread `main` si mette dunque in attesa di raccogliere il valore di ritorno del thread `listener` e, successivamente, termina l'esecuzione del client.

3 Parametri d'esecuzione

Mediante il file `settings.h` è possibile impostare i parametri d'esecuzione del sistema[9].

I commenti al file indicano il significato di ogni parametro, i valori consigliati e il possibile impatto sulle prestazioni che può avere l'incremento o il decremento di un determinato parametro.

Se ne consiglia un'attenta lettura prima di installare il server/client.

Fra i valori che è possibile personalizzare vi è la possibilità di cambiare il nome della socket del server, il numero di thread, il tempo di attesa del thread match; vi è la possibilità di impostare o meno l'aggiornamento delle offerte (come descritto nel paragrafo relativo al thread match).

Fra le varie cose e' inoltre possibile settare la macro VERBOSE, che stampa a video ulteriori messaggi informativi, oltre a quelli già visualizzati, che possono essere d'aiuto nel comprendere cosa sta facendo il server e/o il client.

Tali valori vengono visualizzati all'avvio del server, così da elencare con quali opzioni è stata compilata una determinata versione del server.

Inoltre sono disponibili due file: `settings_low.h` e `settings_high.h` che contengono alcuni parametri predefiniti per l'esecuzione rispettivamente su una macchina di bassa potenza, e su una macchina di potenza elevata.

Si ricorda che non è sufficiente modificare i valori e riavviare il server/client, ma è necessario ricompilarli (**NOTA BENE:** vanno ricompilati entrambi, altrimenti si può incorrere in malfunzionamenti del sistema).

4 Lo script bash - carstat

Lo script `carstat` è uno script bash che elabora le informazioni del file di log emesso dal server.

Il *parsing* delle opzioni viene effettuato con l'utility `getopts`.

[9] Si noti che il file contiene la direttiva `#include <stdlib.h>` dato che altrimenti il compilatore `gcc` genera un warning, non riuscendo a trovare del codice sorgente C

Per quanto riguarda la sua implementazione si è deciso di utilizzare un approccio di tipo iterativo, utilizzando tre array, che contengano rispettivamente i nomi degli utenti, il numero delle offerte effettuate e il numero delle richieste effettuate.

I commenti presenti nel codice risultato sufficienti a comprendere la logica applicativa dello script.

4.1 I valori di uscita

Sono stati definiti i seguenti valori di uscita, in modo da comprendere se l'elaborazione è andata o meno a buon fine, ed eventualmente la causa dell'errore:

0	Esecuzione terminata correttamente.
2 - (ENOENT)	Il file non esiste o non è un file regolare.
7 - (E2BIG)	Un parametro è stato inserito troppe volte.
13 - (EACCESS)	Non si hanno i diritti di lettura di un file.
22 - (EINVAL)	I parametri inseriti non sono corretti.

4.2 Opzioni aggiuntive

Inoltre sono state aggiunte delle opzioni aggiuntive, oltre a quelle richieste nelle specifiche del progetto:

-d Visualizza le informazioni separate per ogni file che viene passato in input.

-h Visualizza un breve messaggio di aiuto sui possibili parametri.

-v Visualizza il numero di versione dello script

Le ultime due opzioni sono state inserite al fine di rendere lo script più omogeneo con gli applicativi tipici del mondo UNIX/Linux.

5 La libreria `dgraph`

La libreria `dgraph` è stata sviluppata inizialmente con un approccio errato, è stata sviluppata utilizzando interamente programma strutturata in senso stresso.

Tale approccio ha reso molto più complesso il codice di tale libreria, e si distaccava molto da quelli che sono gli stili di programmazione tipici del linguaggio C, rendendone così difficile la comprensione. Si è dunque deciso di cercare si "destrutturare" il codice sorgente, per quanto riguarda la parte della verifica dei parametri.

Risulta invece ancora da correggere tutta la parte relativa all'elaborazione.

5.1 Scelte implementative

La libreria risulta interamente *thread safe*, visto che sono state utilizzate funzioni solamente *thread safe*. In particolar modo si è utilizzata la funzione `strtok_r` che è una versione rientrante e *thread safe* della funzione `strtok`.

Per quanto riguarda l'algoritmo di *Dijkstra*, si è deciso di implementare la coda di priorità tramite un *heap* (ad esso sono dedicati i file `heap.c` e `heap.h`). Tale struttura offre infatti la possibilità di eseguire operazioni di estrazione e di inserzione in tempo logaritmico, favorendo dunque la velocità di ricerca.

5.2 Utilizzo delle espressioni regolari

La verifica della correttezza delle stringhe viene effettuata tramite le funzioni definite nell'header file `stringparser.h`

Per la verifica della correttezza di un possibile arco è stato implementato un metodo alternativo che permette di utilizzare le *espressioni regolari* invece che le operazioni su stringhe.

Per attivare questa modalità è sufficiente definire la macro REG_EXPR_MODE nel suddetto header.

Tale modalità è stata comunque introdotta solamente a scopo didattico, dato che rende molto più onerosa l'elaborazione, e se ne sconsiglia l'utilizzo. In particolar modo introduce un notevole ritardo nell'avvio del server[10].

Per generare la libreria `libcars` utilizzando questo metodo è possibile utilizzare il target `lib1-regexpr`

[10] Precisamente nel momento in cui viene verificata la correttezza del file `arc_file`

6 Stili di programmazione

Si è cercato di utilizzare uno stile di programmazione che si avvicinasse il più possibile a quello che è lo stile dei software di sistema UNIX.

Le operazioni di acquisizione e di rilascio della mutua esclusione sono state incapsulate all'interno di specifiche funzioni (con nomi spesso autoesplicativi), in modo da evitare errori durante la scrittura del codice, e rendere più agevole la comprensione per un estraneo.

Per quanto riguarda la gestione degli errori, è stato definito un header file comune che contiene le macro condivise da entrambi i sorgenti, il file `commons.h`.

Tali macro sono molto simili a quelle presentate in classe dal docente.

7 Requisiti Minimi

Per l'installazione del sistema è necessario disporre dei seguenti strumenti:

- Un compilatore C (consigliato `gcc`).
- La libreria standard C (consigliato `lic6-dev`).
- Lo strumento di guida alla compilazione `make`.

Il sistema è stato testato su alcune macchine con su installato Ubuntu 11.04 (Natty Narwal) con l'ultima versione del kernel Linux (2.6.38) ed è risultato funzionante, non si sono riscontrati problemi di alcun genere.

Il sistema è stato inoltre testato sulla macchina `olivia.cli.di.unipi.it` ed è risultato funzionante.

Attenzione: l'esito di `test31` `test32` e `test33` può variare in base alla dimensione del pool di thread (macro POOL_SIZE).

8 Installazione del sistema

Per installare il sistema è possibile utilizzare alcuni target del makefile, creati appositamente per questo scopo:

install-client Ripulisce l'ambiente, ricompila le librerie, e installa il client.

install-server Ripulisce l'ambiente, ricompila le librerie, e installa il server.

install Ripulisce l'ambiente, ricompila le librerie, e installa sia il server che il client.

Tutti questi target sono corredati da stampe a video che indicano all'utente i passi dell'installazione.

9 `stress` - uno script per valutare le prestazioni

È stato scritto un breve script `bash` che permette di portare il pool di thread al massimo del suo carico, così da poter valutare le condizioni del sistema in una fase di carico elevato (così da potersi poi regolare per impostare la dimensione del pool).

Lo script si invoca facilmente con il comando

```
./stress.sh <num client> <sec>
```

Lo script provvede dunque ad invocare ¡num client¿ distinti ogni ¡sec¿ secondi.

I client vengono generati con nomi differenti[11], così che i client non vengano rifiutati per password invalida.

10 Nota sull'uso dei segnali

Per inviare il segnale di SIGUSR1 al server è possibile utilizzare l'utility `kill`[12] con la seguente sintassi:

```
kill -USR1 <pid>
```

[11] Viene utilizzato un numero progressivo

[12] Generalmente presente in tutti i sistemi UNIX/Linux

Dove `pid` rappresenta il *process identifier* del processo del server. Può essere ricavato con l'utility `ps`.

Per inviare il segnale di terminazione si può utilizzare la combinazione di tasti `Ctrl-C` se ci si trova nella shell dove è stato avviato il processo oppure usare l'utility sopra citata con parametro -INT o -TERM

Nel caso in cui il sistema risulti bloccato utilizzare l'utility sopra citata con parametro -KILL. Cerca di utilizzare il meno possibile questa soluzione, dato che potrebbe lasciare il sistema in uno stato inconsistente.

ATTENZIONE: non inviare assolutamente i segnali SIGUSR2 al server, e SIGUSR1 al client. Essi sono utilizzati per gestire la procedura di terminazione dell'applicativo. Una loro ricezione inaspettata potrebbe portare il sistema in caso di blocco.

11 Bug Noti

Segue una lista di alcuni bug noti di cui non si è riusciti a trovare una soluzione:

- Il sistema è risultato non funzionante su una vecchia versione di Ubuntu. La compilazione terminava correttamente, ma il client dava un *Segmentation fault* nel momento in cui si invocava la funzione `sendMessage`. Tale problema sembra essere derivato da un scrittura in una locazione di memoria illecita (`comsock.c: 207`). Il problema si è comunque stato risolto con un aggiornamento delle librerie e del compilatore.

  Purtroppo non si è riusciti a comprendere la causa di questo problema, che comunque non si è manifestato su altre macchine.

12 Possibili miglioramenti

Segue una lista dei possibili miglioramenti che potrebbero essere apportati in futuro, al fine di migliorare il software e di estendere le sue capacità.

- Introdurre il supporto per le socket `AF_INET` in modo da poter rendere utilizzabile il software anche sulla rete
- Introdurre la possibilità di loggarsi al server come `root` e di poter sottoporre richieste sullo stato attuale del server (ad esempio: stato del pool di thread, stato delle code delle richieste offerte, lista degli utenti loggati, etc...).
- Introdurre la possibilità di poter modificare il grafo durante l'esecuzione del server, sempre mediante un accesso privilegiato.
- Introdurre una gestione avanzata delle offerte, in modo che si possa comunicare agli utenti che hanno avanzato un'offerta, un elenco delle persone che deve caricare/scaricare ad ogni tappa del percorso.

13 Documentazione allegata

In allegato al codice sorgente è disponibile la documentazione nelle seguenti forme:

- Il seguente documento, di cui sono disponibili il PDF e i sorgenti LaTeX
  - Per compilare il LaTeX di questa relazione e per visualizzarne il PDF eseguire il target `relazione.pdf` (necessari i software `pdflatex` e `evince` per compilazione e visualizzazione).
  - Per compilare il LaTeX di questa relazione e per visualizzarne il PS eseguire il target `relazione.ps` (necessari i software `latex` e `dvips` ed `evince`).
- Le manpages del server, del client e dello script, nella cartella `doc/man/`
  - Per visualizzare le manpages eseguire il target `man-<software>`, ad esempio `man-docars` per visualizzare il man di `docars`
  - Per visualizzare il PS delle manpage eseguire il target `man-<software>-ps` (necessario `evince` per la visualizzazione).
  - Per visualizzare l'HTML delle manpage eseguire il target `man-<software>-html` (necessari i software `man2html` e un browser web per la generazione e per la visualizzazione).
- I README del server, del client e dello script bash.
- I commenti al codice, redatti in formato `doxygen`
  - Per rigenerare la documentazione `doxygen` è possibile utilizzare il target `docu`.

14 Licenza d'uso

Tutto il codice sorgente scritto viene rilasciato sotto licenza Gnu GPL - General Public Licence versione 3, ognuno è libero di modificare e di distribuire il codice sorgente entro i termini di tale licenza. Tale licenza può essere consultata all'indirizzo: http://www.gnu.org/copyleft/gpl.html

Tutta la documentazione è rilasciata sotto licenza Gnu FDL - Free Documentation Licence versione 1.3, ognuno è libero di modificare e di distribuire questo, e gli altri documenti facenti parte della documentazione entro i termini di tale licenza. Tale licenza può essere consultata all'indirizzo: http://www.gnu.org/licenses/fdl.html

Per qualsiasi problema o chiarimento è possibile contattare lo sviluppatore all'indirizzo `cortin [at] cli.di.unipi.it`.