`cars`: un semplice sistema di car sharing

Progetto del modulo di laboratorio dei corsi di SO A/B 2010/11

Indice

1 Introduzione

Il modulo di Laboratorio di Programmazione di Sistema del corso di Sistemi Operativi e Laboratorio (277AA) prevede lo svolgimento di un progetto individuale suddiviso in tre frammenti. Questo documento descrive la struttura complessivia del progetto e dei vari frammenti che lo compongono.

Il progetto consiste nello sviluppo del software relativo a `cars`: un sistema che permette il car sharing fra un insieme di utenti. Il software viene sviluppato

e documentato utilizzando gli strumenti, le tecniche e le convenzioni presentati durante il corso.

1.1 Materiale in linea

Tutto il materiale relativo al corso può essere reperito sul sito Web:

`http://www.cli.di.unipi.it/doku/doku.php/informatica/sol/laboratorio11/`

Il sito verrà progressivamente aggiornato con le informazioni riguardanti il progetto (es. FAQ, suggerimenti, avvisi), il ricevimento e simili. Il sito è un Wiki e gli studenti possono registrarsi per ricevere automaticamente gli aggiornamenti delle pagine che interessano maggiormente. In particolare consigliamo a tutti di registrarsi alla pagina delle 'FAQ' e degli 'Avvisi Urgenti'.

Eventuali chiarimenti possono essere richiesti consultando i docenti di del corso durante l'orario di ricevimento, le ore in laboratorio e/o per posta elettronica.

1.2 Struttura del progetto e tempi di consegna

Il progetto deve essere sviluppato dallo studente individualmente e può essere consegnato entro il 31 Gennaio 2012.

La consegna del progetto avviene *esclusivamente* inviando per posta elettronica il tar creato dal target `consegna3` del `Makefile` contenuto nel kit di sviluppo del progetto. Il tar deve essere allegato ad un messaggio con soggetto

`lso11: Consegna Terzo Frammento`

ed inviato a `susanna@di.unipi.it`. Le consegne sono seguite da un messaggio di conferma da parte del docente all'indirizzo di mail da cui la consegna è stata effettuata. Se la ricezione non viene confermata entro 3/4 giorni lavorativi, contattare il docente per e-mail.

I progetti che non rispettano il formato o non generati con il target `consegna3` non verranno accettati. Si prega di controllare che tutti i file necessari alla corretta compilazione ed esecuzione del progetto siano presenti nel tar prima di inviarlo.

La data ultima di consegna è il 01/02/2012. Dopo questa data gli studenti dovranno svolgere il nuovo progetto previsto per il corso 2011/12.

Inoltre, il progetto è suddiviso in tre frammenti. Gli studenti che consegnano una versione sufficiente di ogni frammento entro la data di scadenza specificata sul WEB accumulano dei bonus di 2 punti che contribuiscono al voto finale con le modalità illustrate nelle slide introduttive del corso (vedi sito).

1.3 Valutazione del progetto

Al progetto viene assegnata una valutazione da 0 a 30 di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. La valutazione del progetto è effettuata in base ai seguenti criteri:

- motivazioni, originalità ed economicità delle scelte progettuali
- strutturazione del codice (suddivisione in moduli, uso di makefile e librerie etc.)
- efficienza e robustezza del software
- modalità di testing
- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della relazione (vedi Sez. 9.3)

La prova orale tenderà a stabilire se lo studente è realmente l'autore del progetto consegnato e verterà su tutto il programma del corso e si tutto quanto usato nel progetto anche se non fa parte del programma del corso. Il voto dell'orale (da 0 a 30L) fa media con la valutazione del progetto per delineare il voto finale. In particolare, l'orale comprenderà

- una discussione delle scelte implementative del progetto e dei frammenti
- l'impostazione e la scrittura di script bash e makefile
- l'impostazione e la scrittura di programmi C + PosiX non banali (sia sequenziali che concorrenti)
- domande su tutto il programma presentato durante il corso.

Casi particolari Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare i due frammenti e il progetto finale in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30. In questo caso è necessaria la certificazione da consegnare al docente come da regolamento di ateneo.

Gli studenti che svolgono il progetto per abbreviazioni delle nuove lauree specialistiche sono invitati a contattare il docente.

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di Lab. 4 contribuiva al voto di SO) avranno assegnato un voto in trentesimi che verrà combinato con il voto del corso di SO corrispondente secondo modalità da richiedere ai due docenti coinvolti.

2 Il progetto: `cars`

Lo scopo del progetto è lo sviluppo di un sistema client server per effettuare car sharing su un insieme di rotte.

Il sistema è costituito dai seguenti processi concorrenti:

- `docars`: il processo client (uno per ogni utente connesso) che si occupa di interagire con il server per inviare offerte e richieste di car sharing per conto di un utente,
- `mgcars`: il processo server che verifica se un utente può accedere al servizio, mantiene traccia delle richieste e delle offerte, stabilisce i gruppi di condivisione ed invia le risposte ai client;

`docars` e `mgcars` sono i due processi multithreaded che devono essere realizzati nel progetto didattico. I processi comunicano via socket AF_UNIX. Per rendere più semplice lo sviluppo ed il testing dell'applicazione sulle macchine del centro di calcolo per la didattica tutte le socket vengono create nella directory locale `./tmp` invece che in `/tmp` come sarebbe logico aspettarsi. Quest'ultima soluzione renderebbe infatti possibili interazioni indesiderate e fastidiose fra progetti sviluppati da utenti diversi sulla stessa macchina.

Inoltre, deve essere realizzato uno script bash (`carstat`) che analizza il file di log scritto dal server e genera alcune statistiche sul servizio di car sharing come specificato in Sez. 6.

3 Il server

`mgcars` viene attivato da shell con il comando

```
$ mgcars file_citta file_strade
```

dove

- `file_citta` è il file di testo che contiene le città della mappa su cui lavora il server secondo il formato analizzato nel primo frammento, ovvero separate da `\n` come ad esempio

  ```
  PISA
  FIRENZE
  LA SPEZIA
  ```

  la lunghezza del nome di una città è limitata dalla macro `LLABEL` in `dgraph.h`. Un nome di citta può contenere solo caratteri alfanumerici e lo spazio bianco.

- `file_strade` è il file di testo che contiene le informazioni sui collegamenti stradali. Il formato è

  ```
  LBLSORGENTE:LBLDESTINAZIONE:DISTANZA_IN_KM
  ```

  dove `LBLSORGENTE` e `LBLDESTINAZIONE` sono città (con i vincoli espressi nel precedente paragrafo), mentre `DISTANZA_IN_KM` è una stringa che rappresenta un numero reale (lunghezza massima fissata dalla macro `LKM` definita in `dgraph.h`). I vari collegamenti sono separati da `\n`, ad esempio:

  ```
  FIRENZE:PRATO:20.4
  LUCCA:LA SPEZIA:90.1
  ```

Il server utilizza anche un file di log `./mgcars.log` con modalità descritte nel seguito.

Il funzionamento del server è il seguente. Se i file passati come argomento esistono vengono aperti in lettura ed utilizzati per creare un grafo diretto che descrive una mappa stradale. Dopo la creazione della mappa viene creato il file di log (se il file già esiste viene troncato) e viene creata una socket AF_UNIX di indirizzo

```
./tmp/cars.sck
```

su cui i client apriranno le connessioni con il server per inviare le richieste e le offerte di car sharing.

Il server è multithreaded in quanto ogni richiesta viene servita da un thread `worker` e viene servita in parallelo con l'accettazione di connessioni da parte di altri utenti e con il calcolo dei gruppi di condivisione.

Ogni worker controlla che l'utente sia abilitato a inviare richieste. L'utente per connettersi invia il proprio nome ed una password[1]. La password inviata la prima volta che l'utente si connette viene ricordata dal server e dovrà poi essere fornita per tutte le connessioni successive dello stesso utente. Le password vengono azzerate ad ogni nuova attivazione del server.

Se l'autenticazione ha successo l'utente può inviare richieste e offerte di sharing in modo interattivo. Le risposte alle richieste fatte verranno inviate connettendosi su una socket AF_UNIX creata dal client nella directory `./tmp`.

Le richieste corrette vengono accodate ed eleborate periodicamente dal server con le modalità descritte nel seguito. Spetta allo studente decidere quali strutture dati vengono utilizzate per accodare le richieste. La scelta deve essere documentata e motivata nella relazione finale.

All'avvio il server crea un thread `match` che periodicamente elabora le richieste pendenti ed invia le risposte agli utenti che le hanno inviate. Tutti gli accoppiamenti trovati e inviati agli utenti vengono registrati nel file di log. Il formato di un accoppiamento è il seguente:

```
user1$user2$partenza$citta1$ ...$cittaN$arrivo
```

[1] Nel progetto non viene trattata in alcun modo la sicurezza delle informazioni inviate, che è oggetto di corsi successivi. In particolare la password viene conservata e trasmessa in chiaro senza alcuna precauzione.

dove `user1` è l'utente che offre lo sharing, `user2` è l'utente che ha richiesto lo sharing, `partenza` è la città di partenza, `citta1$ ...$cittaN` sono le città attraversate dalla rotta e `arrivo` è la città di arrivo.

Gli accoppiamenti sono separati da `\n`. Ad esempio:

```
pippo$pluto$LIVORNO$PISA$VIAREGGIO
pippo$ciccio$LIVORNO$PISA
minnie$ciccio$PISA$LUCCA$ALTOPASCIO$PISTOIA
```

rappresentano tre accoppiamenti trovati per gli utenti `pippo`, `pluto`, `ciccio` e `minnie`. L'ordine in cui gli accoppiamenti compaiono nel file di log è irrilevante.

Gli accoppiamenti vengono sempre calcolati sul cammino minimo (quello con la minima lunghezza in Km) fra la città di partenza e quella di arrivo. L'algoritmo di accoppiamento fra richieste ed offerte e le strutture date usate per realizzarlo devono essere decisi dallo studente e documentati e motivati nella relazione. Se necessario, le strutture definite in `dgraph.h` possono essere estese aggiungendo campi per contenere informazioni aggiuntive e nuove funzioni possono esere aggiunte alla libreria. L'unico vincolo da rispettare è che i test del primo frammento continuino ad essere superati.

L'elaborazione delle richieste pendenti da parte del thread `match` avviene quando si verifica uno dei seguenti eventi:

- sono passati `SEC_TIMER` secondi, dove `SEC_TIMER` è una macro definita in `comsock.h` e settata di default a 30;
- è stato ricevuto un segnale di `SIGUSR1`;
- è stata richiesta la terminazione del server con un segnale di `SIGINT` o `SIGTERM`, in questo caso si terminano i thread worker, si elaborano tutte le richieste pendenti e infine il server viene terminato.

Prima di terminare, il server ripulisce sempre l'ambiente eliminando la socket `cars.sck` da `./tmp` e completa la scrittura sul file di log di tutti gli accoppiamenti pendenti.

Il protocollo di interazione fra client e server è descritto nella Sezione 5.

4 Il client

Il client è un comando Unix che viene attivato da linea di comando con

```
$ ./docars username
```

dove `username` è il nome dell'utente che desidera connettersi. La lunghezza massima del nome utente è stabilita dalla macro `LUSERNAME` in `comsock.h`. Appena attivato, il client effettua il parsing delle opzioni, e se il parsing è corretto chiede la password all'utente e cerca di collegarsi con il server (su `./tmp/cars.sck`) per un massimo di 3 tentativi a distanza di 1 secondo l'uno dell'altro (vedi macro `NTRIELCONN` in `comsock.h`). La password deve contenere almeno un carattere.

Prima di inviare la richiesta di connessione il client crea una socket AF_UNIX con nome univoco (ad esempio aggiungendo al nome utente il pid del processo `docars`) su cui si connetterà il server per inviare la risposta. A questo punto, il client invia la richiesta di connessione con un messaggio di `CONNECT` contenente nome utente, password e nome della socket (per il formato vedi Sec. 5).

Il client è formato da due thread paralleli:

- un thread che si occupa di leggere le richieste da standard input, di verificarne la correttezza e di inviarle al server
- un thread che ascolta le richieste di connessione e le risposte in arrivo dalla socket creata.

Ogni utente può inviare più richieste al server.

Il formato con cui l'utente effettua richieste sullo standard input è il seguente. Per le offerte di sharing è sufficiente specificare città di partenza e di arrivo e numero di posti separati da `':'` e terminati da da newline `\n` es:

```
PISA:LA SPEZIA:4
```

per i nomi di città valgono le limitazioni specificate precedentemente, il numero dei posti è un intero positivo. La richiesta di posti ha un formato simile preceduto da `%R` ad esempio

```
%R PISA:LA SPEZIA
```

in questo caso il numero dei posti non viene specificato perché gni richiesta è relativa ad un solo posto. Quando la sequenza di richeste è terminata si può utilizzare il comando

```
%EXIT
```

oppure chiudere lo standard input con un EOF (CTRL-D). Questo termina la sessione interattiva ma non il processo client che rimane in attesa delle risposte alle richieste pendenti da parte del server. Inoltre si richiede di dare brevi indicazioni di uso se viene invocato il comando

```
%HELP
```

Il client termina quando riceve un segnale `SIGTERM` o `SIGINT`. Alla terminazione viene rimossa la socket creata per la risposta.

5 Protocollo di interazione

Il server ed il client interagiscono con un socket *socket AF_UNIX*.

I client si connettono al server attraverso la socket `./tmp/cars.sck`, creata all'avvio del server.

5.1 Formato dei messaggi

I messaggi scambiati fra server e client hanno la seguente struttura:

```
typedef struct {
   char type;
   unsigned int length;
   char* buffer;
} message_t;
```

Il campo `type` è un `char` (8 bit) che contiene il tipo del messaggio spedito. `type` può assumere i seguenti valori:

```
#define MSG_CONNECT      'C'
#define MSG_OK           'K'
#define MSG_NO           'N'
#define MSG_REQUEST      'R'
#define MSG_OFFER        'F'
#define MSG_EXIT         'X'
#define MSG_SHARE        'S'
#define MSG_SHAREND     'H'
```

Il campo `length` è un `unsigned int` che indica la dimensione del campo `buffer`. Il campo `buffer` è un puntatore a un array di caratteri. Se buffer contiene una stringa il campo lunghezza ha un valore che comprende anche il terminatore di stringa `\0` finale. Il campo `length` vale `0` nel caso in cui il campo `buffer` non sia significativo.

5.2 Messaggi da Client a Server

Nei messaggi spediti dal client al Server, il campo `type` può assumere i seguenti valori:

`MSG_CONNECT` messaggio di login, spedito dal Client quando desidera connettersi al Server. In questo caso il campo `buffer` contiene il nome dell'utente che si sta connettendo, la password e la socket su cui inviare l'accoppiamento separati da un terminatore di stringa `\0` come in `ciccio\0cicciopwd\0ciccioskt`. Il server risponde con `MSG_OK` la connessione è autorizzata, oppure `MSG_NO` e un eventuale messaggio di spiegazione, ad esempio se la password è errata o l'utente è già connesso.

`MSG_EXIT` Messaggio di disconnessione. Il campo `buffer` non è significativo.

`MSG_OFFER` Messaggio per offrire il car sharing. Il campo `buffer` contiene la richiesta secondo il solito formato

```
PISA:LA SPEZIA:4
```

Il server risponderà con un `MSG_OK` se la richiesta è accettata per essere elaborata, `MSG_NO` se non lo è.

`MSG_REQUEST` Messaggio per richiedere car sharing il formato è lo stesso di `MSG_OFFER` ed il server può rispondere con `MSG_OK` se la richiesta è accettata per essere elaborata, `MSG_NO` se non lo è.

5.3 Messaggi da Server a Client

Nei messaggi spediti dal Server a Client, il campo `type` può assumere i seguenti valori:

`MSG_NO` Messaggio di rifiuto. Questo tipo di messaggio viene spedito quando si rifiuta una richiesta dell'utente. Il campo buffer può contenere una stringa che spiega le motivazioni del rifiuto.

`MSG_OK` Messaggio di risposta positiva. Con questi messaggi il server accetta la connessione o conferma l'accodamento di una richiesta o offerta di sharing. Il campo buffer è generalmente non significativo.

`MSG_SHARE` messaggio che segnala un accoppiamento che risponde ad una richiesta dell'utente. Nel campo buffer è indicato l'utente che fornisce lo share, quello che lo riceve e la rotta seguita con formato `pippo$pluto$LIVORNO$PISA$VIAREGGIO` (vedi Sez. 3)

`MSG_SHAREND` messaggio che segnala che non ci sono più richieste pendenti e quindi share da ricevere.

6 Lo script `carstat`

`carstat` è uno script bash che elabora off-line i file di log generato dal processo `mgcars` e calcola varie statistiche. Lo script può essere invocato con diverse opzioni:

```
$ ./carstat [ -u user ] [ -a citta ] [ -p citta ] log1 ... logN
```

dove `log1...logN` sono file di testo (ASCII) che contengono i log degli accoppiamenti generati dal server secondo il formato specificato in Sez. 3. La semantica delle varie opzioni è spiegata nel seguito.

Lo script deve controllare la validità dei suoi argomenti, scorrere i file di log e valutare quanto richiesto dalle opzioni. Se invocato come

```
$ ./carstat log1 ... logN
```

lo script conta quante richieste ed offerte di car sharing sono state realizzate per ogni utente e lo stampa sullo stadard output con il formato

```
utente:n_offerte_accettate:n_richieste_accettate
```

ad esempio

```
minnie:9:80
pippo:5:7
pluto:10:3
```

Se invocato con opzione `-u` come in

```
$ ./carstat -u user log1 .. logN
```

viene rilevata solo l'informazione relativa all'utente `user`

```
$ ./carstat -u minnie logfile
minnie:9:80
```

Le opzioni `-a` e `-p` servono a modificare il comportamento descritto in modo da considerare solamente gli accoppiamenti che partono (`-p`) o arrivano (`-a`) in una determinata città. Ad esempio,

```
$ ./carstat -u minnie -a PISA logfile
minnie:1:8
```

considera solo gli sharing con città di arrivo PISA, mentre

```
$ ./carstat -u minnie -p "LA SPEZIA" -a PISA logfile
minnie:0:0
```

considera solo gli sharing con città di partenza LA SPEZIA e città di arrivo PISA. La sintassi delle città deve corrispondere a quella utilizzata nei file che descrivono la mappa usata dal server per generare gli accoppiamenti ed i file di log.

7 Istruzioni

7.1 Materiale fornito dai docenti

Nei kit del progetto vengono forniti

- funzioni di test e verifica
- `Makefile` per test e consegna
- file di intestazione (.h) con definizione dei prototipi e delle strutture dati
- vari `README` di istruzioni

7.2 Cosa devono fare gli studenti

Gli studenti devono:

- leggere *attentamente* i README dei vari frammenti e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle
- verificare le funzioni con i test forniti dai docenti (attenzione: questi test vanno eseguiti su codice già corretto e funzionante altrimenti possono dare risultati fourvianti o di difficile interpretazione)
- preparare la *documentazione*: vedi la Sez. 9.
- sottomettere i tre frammenti del progetto esclusivamente utilizzando il makefile fornito e seguendo le istruzioni nel README.

8 Parti Opzionali

Possono essere realizzate funzionalità ed opzioni in più rispetto a quelle richieste (ad esempio altri comandi del client o altre statistiche nello script).

Le parti opzionali devono essere spiegate nella relazione e corredate di test appropriati.

9 Codice e documentazione

In questa sezione, vengono riportati alcuni requisiti del codice sviluppato e della relativa documentazione.

9.1 Vincoli sul codice

Makefile e codice devono compilare ed eseguire CORRETTAMENTE su (un sottinsieme non vuoto del) le macchine del CLI. Il README (o la relazione) deve specificare su quali macchine è possibile far girare correttamente il codice. Inoltre, se si usano software e librerie non presenti al CLI: (1) devono essere presenti nel tar TUTTI i file necessari per l'istallazione in locale del/i tool e (2) devono essere presenti nel makefile degli opportuni target per effettuare automaticamente l'installazione in locale. Se questa condizione non è verificata il progetto non viene accettato per la correzione.

La stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo delle regole appropriate nella parte iniziale del makefile contenuto nel kit;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- NON devono essere utilizzate funzioni di temporizzazioni quali le `sleep()` o le `alarm()` per risolvere problemi di race condition o deadlock fra i processi/thread. Le soluzioni implementate devono necessariamente funzionare qualsiasi sia lo scheduling dei processi/thread coinvolti
- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)

9.2 Formato del codice

Il codice sorgente deve adottare una convenzione di indentazione e commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file che contiene: il nome ed il cognome dell'autore, la matricola, il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore; firma dell'autore.
- un commento all'inizio di ogni funzione che specifichi l'uso della funzione (in modo sintetico), l'algoritmo utilizzato (se significativo), il significato delle variabili passate come parametri, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per puntatore etc.
- un breve commento che spieghi il significato delle strutture dati e delle variabili globali (se esistono);
- un breve commento per i punti critici o che potrebbero risultare poco chiari alla lettura
- un breve commento all'atto della dichiarazione delle variabili locali, che spieghi l'uso che si intende farne

9.3 Relazione

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 10 pagine) il cui scopo è quello di descrivere la struttura complessiva del lavoro svolto. La relazione *deve rendere comprensibile il lavoro svolto ad un estraneo, senza bisogno di leggere il codice se non per chiarire dettagli implementativi. In particolaer NON devono essere ripetute le specifiche contenute in questo documento.* In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati principali, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file, librerie etc.)
- la struttura dei programmi sviluppati
- la struttura dei programmi di test (se ce ne sono)
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- README di istruzioni su come compilare/eseguire ed utilizzare il codice

La relazione deve essere in formato PDF.