BRIS - Gioco di briscola multiutente
Progetto di Laboratorio di Sistemi Operativi
Anno Accademico 2012/2013

Orlando Leombruni

1 luglio 2013

Indice

1 Introduzione

Il progetto consiste nell'implementazione in C di server e client per il gioco della Briscola e di uno strumento in `bash` per l'analisi dei file di log generati dal server.

Il server è stato strutturato in modo multithreaded e consente a diversi client di collegarsi ed effettuare operazioni in parallelo. I client possono richiedere la registrazione e la cancellazione dal servizio, la disconnessione forzata oppure la ricerca di avversari per la partita. L'amministratore del server può usare lo strumento `bristat` per ottenere informazioni statistiche sulle partite giocate.

2 Macro e gestione dell'errore

In ogni parte del progetto si è cercato di sfruttare lo strumento delle macro per migliorare la leggibilità e la facilità di modifica del codice.

2.1 Stringhe di testo: `commonstrings.h`

Nel file header `commonstrings.h` sono state inserite tutte le stringhe di testo utilizzate nel server e nel client. In questo modo è possibile modificare velocemente elementi come il percorso del socket e le opzioni del client; inoltre è possibile tradurre l'output a schermo (sia del server che del client) in qualsiasi lingua senza dover mettere mano al codice dei due eseguibili.

2.2 Gestione degli errori: `errors.h` ed `errors.c`

La gestione dell'errore è implementata nei file `errors.h` (header) ed `errors.c` (funzioni). L'approccio adottato è quello volto ad ottenere la maggiore leggibilità possibile a livello di codice mantenendo comunque una certa verbosità nell'output delle informazioni di errore a schermo; il tutto è stato scritto a partire dagli esempi presenti nel libro del Rochkind, personalizzandoli e ripulendoli dagli elementi ritenuti non necessari (ad esempio è stata eliminata la pila degli errori in favore di una stampa diretta su `stderr`). I file `errors.h` ed `errors.c` costituiscono una libreria a parte, chiamata `liberr`.

3 Moduli

Le funzionalità necessarie all'implementazione del server e del client sono state divise in tre parti: un modulo di primitive e strutture dati specifiche al gioco della briscola (`bris`), un modulo di primitive e strutture dati per un generico servizio di gestione di un insieme di utenti (`users`) ed un modulo per la realizzazione delle comunicazioni tra server e client usando un socket AF_UNIX (`comsock`).

3.1 Il gioco della briscola: `bris`

Il "fulcro" del modulo `bris` è la struttura dati `carta_t`, composta da un campo *seme* e un campo *valore*, sulla quale vengono incentrate le funzioni di questo modulo. Un'altra struttura dati ivi definita è `mazzo_t`, che rappresenta il mazzo di una partita di briscola (composto da `NCARTE = 40` carte e dall'indicazione del seme di briscola). Le primitive disponibili sono:

- Creazione di una carta a partire da una stringa di testo e viceversa;
- Stampa di una carta (in formato a due caratteri) su uno stream di uscita;
- Confronto tra due carte e decretazione del vincitore (secondo le regole della briscola);
- Conteggio dei punti complessivi di un array di carte (secondo le regole della briscola);
- Allocazione e generazione delle carte di un mazzo (in maniera casuale o fissata), thread-safe o meno;
- Estrazione di una carta dal mazzo;
- Stampa di un intero mazzo su uno stream di uscita.

I file relativi a questo modulo sono `bris.h`, `bris.c` e `newMazzo_r.h`.

3.2 Gestione degli utenti: `users`

Gli utenti registrati al servizio sono inseriti in un albero formato da nodi di tipo `nodo_t`. La struttura dati `nodo_t` mantiene nei suoi campi le informazioni sull'utente (mediante un'ulteriore struttura dati, `user_t`, composta a sua volta da un campo *username* e un campo *password*), sul suo stato (disconnesso, connesso e in partita) e sul canale di comunicazione. L'albero è binario di ricerca, ordinato lessicograficamente nel campo *username*. Le primitive disponibili sono:

- Creazione di una struttura utente a partire da una stringa (del tipo `username:password`) e viceversa;
- Aggiunta e rimozione di un utente all'albero, rispettandone le proprietà di ricerca;
- Recupero e modifica di informazioni (stato, canale, presenza) di un utente dall'albero;

- Autenticazione di un utente;
- Recupero della lista di tutti gli utenti in un certo stato;
- Stampa dell'intero albero su `stdout`;
- Caricamento/salvataggio dell'intero albero da/su file.

I file relativi a questo modulo sono `users.h` e `users.c`.

3.3 Comunicazioni client/server: `comsock`

Le comunicazioni tra client e server avvengono attraverso un socket AF_UNIX e l'utilizzo delle system call `read` e `write` per lo scambio di messaggi. La struttura dati `message_t` è deputata a contenere l'intera informazione, ossia il *tipo*, la *lunghezza* ed il *contenuto* (buffer) del messaggio. Per alcuni messaggi, il contenuto degli stessi è irrilevante in quanto è sufficiente specificare solo il loro tipo.

Per evitare problemi derivanti dalla lettura/scrittura asincrona sul socket, tali operazioni devono risultare atomiche; è dunque necessario esprimere l'informazione contenuta in una struttura dati `message_t` mediante un tipo più gestibile la cui dimensione è nota a tempo d'invio messaggio. Nel modulo è stato usato un array di caratteri (ossia un array di locazioni di 1 byte ciascuna). La primitiva d'invio messaggi crea un array di caratteri in questo modo:

Tipo (1 byte)	Lunghezza N (4 byte)	Buffer (N byte) ...

Se `N+5 < 4 kB` allora la `write` è garantita atomica. Nella sezione "lunghezza" dell'array di caratteri vengono scritti direttamente i bit che rappresentano l'intero `N`. In questo modo è garantita la stessa rappresentazione su ogni architettura.

La primitiva di ricezione messaggi ritrasforma l'array di caratteri siffatto in una struttura `message_t`. La natura bloccante della system call `read` ivi utilizzata permette di realizzare in modo molto semplice attese indefinite da parte di chi riceve (ad es. un client che aspetta comunicazioni). La funzione di ricezione non fa assunzioni sulla lunghezza del messaggio ricevuto, in quanto esso potrebbe essere stato inviato con funzioni diverse da quelle implementate in questo modulo. Le primitive rese disponibili dal modulo `comsock` sono:

- Creazione e chiusura di un socket lato server;
- Apertura di un socket lato client;
- Connessione ad un socket;
- Accettazione di connessioni ad un socket;
- Creazione di messaggi arbitrari (di tipo `message_t`);
- Invio e ricezione di messaggi mediante le strutture viste sopra.

I file relativi a questo modulo sono `comsock.h` e `comsock.c`.

4 Organizzazione in librerie

Si è scelto di organizzare i moduli visti sopra in maniera leggermente diversa da quanto suggerito dai docenti. Le librerie usate nel progetto sono infatti:

`libbris` Contiene il necessario per implementare il gioco della briscola. E' formata da `bris.h`, `bris.c`, `users.h`, `users.c` e `newMazzo_r.h`.

`libcomm` Contiene le specifiche di comunicazione tra client e server. E' formata da `comsock.h` e `comsock.c`.

`liberr` Contiene i metodi per la gestione dell'errore. E' formata da `errors.h` e `errors.c`.

5 Struttura del server

Come già detto in precedenza, il server è stato implementato in modo multithreaded per permettere la connessione contemporanea di più client; tutti i thread sono implementati nel file `brsserver.c`. Il server si avvia mediante l'eseguibile `brsserver`.

5.1 Thread `main`

Come prima cosa, il `main` blocca i segnali di tipo SIGINT, SIGTERM, SIGUSR1 e SIGPIPE (quest'ultimo per evitare che l'intero server venga ucciso da un segnale di questo tipo, generato da una scrittura fallita sul socket).

Il `main` si occupa poi della creazione del socket, dell'apertura del file degli utenti, del caricamento di questi ultimi nell'albero e dell'avvio dei thread `signaler` e `dispatcher`, mettendosi in attesa della loro terminazione. Quando essa avviene, il `main` aggiorna il file degli utenti, chiude la socket e termina.

5.2 Thread `signaler`

Il thread `signaler` è deputato alla gestione dei segnali e alla loro corretta gestione.

In particolare, se il server riceve un segnale SIGUSR1, il thread si occuperà della scrittura dell'attuale albero degli utenti in un file di checkpoint; in presenza invece di un segnale SIGINT o SIGTERM provvederà a terminare il `dispatcher` e sé stesso, facendo continuare l'esecuzione del `main`.

5.3 Thread `dispatcher`

Il thread `dispatcher` si occupa di accettare le connessioni in arrivo dai client e di assegnare ogni client ad un nuovo thread `worker`.

Il thread mantiene una lista degli ID dei `worker`, dal più vecchio al più recente. Quando viene terminato, il `dispatcher` chiama una funzione di attesa della terminazione dei `worker`, che possono dunque finire le loro operazioni.

5.4 Thread `worker`

Il thread `worker` si occupa di gestire ed eseguire le operazioni richieste dai vari client, e in particolare inizia e supervisiona le partite registrandone l'andamento nei file di log.

Nell'implementare questo thread si è scelto l'approccio più diretto possibile, facendo gestire da funzioni ausiliarie le richieste dei client (registrazione, cancellazione, disconnessione forzata e inizio connessione) e cercando di minimizzare il numero di messaggi inviati e ricevuti. Se un utente sceglie di giocare una partita, il thread ottiene dall'alberp il canale dell'avversario scelto e provvede a "risvegliarlo" mediante l'invio di un messaggio di tipo MSG_STARTGAME. La partita vera e propria viene poi gestita da un'apposita funzione.

5.4.1 Partita

Dopo il setup iniziale (creazione del mazzo e pesca di 3 carte per giocatore) la funzione entra in un ciclo regolato da un booleano `finished`, inizialmente settato a `FALSE`. Il ciclo effettua le seguenti operazioni:

- Ricezione della carta giocata dal primo giocatore, verifica della sua validità e comunicazione al secondo giocatore;
- Ricezione della carta giocata dal secondo giocatore, verifica della sua validità e comunicazione al primo giocatore;
- Confronto delle due carte e decretazione del vincitore della mano;
- Controllo delle mani dei giocatori (se entrambe vuote, `finished = TRUE`);
- Se la partita non è terminata: determinazione dell'ordine del prossimo turno, pesca di una carta per giocatore dal mazzo, comunicazione ai giocatori.

Terminata la partita vera e propria, la funzione calcola i punti di ogni giocatore e decreta il vincitore, dandone comunicazione ad entrambi.

6 Struttura del client

Il client è stato implementato in maniera sequenziale, in quanto non si vengono ad avere situazioni in cui è necessaria la concorrenza. Il client apre un nuovo canale di comunicazione sul socket del server e richiede un'operazione sui propri dati (registrazione, cancellazione, disconnessione) o la connessione. Nel caso in cui il client si mette in attesa (volontariamente o forzatamente), tale attesa è realizzata mediante la funzione di ricezione come descritto sopra. Analogamente a quanto accade nel server, anche nel client la partita è gestita da una funzione separata.

6.1 Partita

La funzione che regola la partita è ovviamente speculare a quella deputata allo stesso ruolo nel server. Dopo una fase iniziale di setup, si ha un ciclo di operazioni con la seguente struttura:

- Se nel turno corrente si è primi: invio della carta giocata al server e ricezione della carta giocata dall'avversario;
- Se nel turno corrente si è secondi: ricezione della carta giocata dall'avversario e invio della propria carta giocata al server;
- Ricezione del messaggio di fine turno o di fine partita;
- Se si ha il messaggio di fine turno, sostituzione della carta giocata con la nuova carta ricevuta e assegnamento del nuovo ordine.

7 Analisi delle statistiche: `bristat`

Lo script `bristat` permette di ottenere le statistiche di gioco degli utenti a partire dai file di log generati dal server. Lo script non usa gli array associativi per la memorizzazione temporanea delle statistiche, in quanto essi non sono supportati dalle vecchie versioni di `bash`. Il parsing delle opzioni da riga di comando è fatto senza usare *builtin*, in quanto si richiedeva la possibilità di inserire tali opzioni in qualsiasi ordine.

8 Aggiunte

E' stata aggiunta al client la possibilità di disconnettere forzatamente un utente (impostando sul relativo nodo dell'albero lo stato DISCONNECTED e il canale -1) mediante l'opzione `-d`. Ciò permette agli utenti di potersi ricollegare al servizio dopo la terminazione improvvisa del client o del server (si veda, a riguardo, la sezione Bug noti).

9 Documentazione

E' disponibile una documentazione più ricca sulle funzioni e strutture dati utilizzate mediante il tool Doxygen. Per creare la documentazione in HTML grazie a questo tool, basta eseguire il comando

```
make docu
```

10 Guida per l'utente

10.1 Compilazione

E' possibile compilare in modo rapido gli eseguibili `brsserver` e `brsclient` con il comando

```
make execs
```

10.2 Esecuzione

E' possibile eseguire il server con il comando

```
brsserver users_file [-t]
```

dove l'opzione `-t` indica l'esecuzione in modalità test (mazzi generati in maniera predicibile).

Il client si esegue invece con il comando

```
brsclient username password [-r | -c | -d]
```

dove le opzioni `-r`, `-c` e `-d` indicano, rispettivamente, la richiesta di registrazione, cancellazione e disconnessione forzata.

Infine, è possibile chiamare lo script di analisi con

```
bristat [-p] [-m] [-u user] log1 ...logN
```

dove le opzioni `-p`, `-m` e `-u user` indicano, rispettivamente, la richiesta di statistiche sulle partite perse (anziché vinte), la richiesta di informazioni sulla media punti e la richiesta di informazioni sul singolo utente `user`.

11 Bug noti

Se un utente viene messo in attesa (forzatamente o per propria scelta) e poi termina il client (ad es. con il segnale CTRL+C), la sua disconnessione non viene segnalata sull'albero e dunque lo stesso utente non potrà più collegarsi al servizio. Per ovviare a questo, è stata resa disponibile al client l'opzione `-d` per la disconnessione forzata.

I test previsti dai docenti sono stati passati con successo sulla macchina Olivia.