SISTEMI OPERATIVI E LABORATORIO
A.A. 2012/13

RELAZIONE PROGETTO bris

Stefano Forti

I must complain the cards are ill shuffled until I have a good hand.
[Jonathan Swift]

I. INTRODUZIONE

Il progetto bris prevede la realizzazione di un server *multithreaded* in grado di gestire il gioco della briscola tra più client e di eseguire semplici elaborazioni statistiche sui file di log registrati dal server dopo ogni partita.

La struttura del progetto consta delle seguenti parti previste dalla specifica:

1. libreria bris per la gestione delle partite e delle carte,
2. libreria users per la gestione degli utenti registrati al gioco e delle loro eventuali richieste,
3. libreria comsock per la gestione delle comunicazioni server-client (e viceversa)
4. script di analisi dei file di log, bristat

Vi sono poi i file contenenti il codice del client e del server che sfruttano, per quanto è stato possibile, le funzioni contenute in 1,2,3.

Per rendere modulare quest'ultima parte e semplificarne la fase di programmazione/debugging sia per il client che per il server si è scelto di gestire la partita vera e propria con funzioni *ad hoc*, manageMatch() per il server e joinMatch() per il client, contenute in file separati rispetto al codice che gestisce la registrazione/cancellazione degli utenti nel sistema e l'accesso in partita. Ciò rende possibile una successiva implementazione di altri giochi con le carte, semplicemente aggiungendo nuove funzioni per la gestione della partita (ad esempio per la scopa manageMatchScopa(), joinMatchScopa()...) e una libreria analoga a bris, lasciando all'utente la possibilità di scegliere a quale gioco prendere parte ogni volta ma mantenendo pressoché invariato il codice da eseguire prima della partita.

II. STRUTTURE DATI

Le strutture dati adottate nel progetto sono state, oltre a quelle standard del C:

a. CARTA (tipo carta_t): si compone di un seme e di un valore e è definita nell'header bris.h (insieme ai tipi seme_t e valore_t cui si fa riferimento), scelta come rappresentazione virtuale di una carta da gioco
b. MAZZO (tipo mazzo_t): definito in bris.h, si compone di un vettore di NCARTE e contiene l'indice della prossima carta da pescare e il seme di briscola per la partita corrente (eliminando quest'ultimo dettaglio sarebbe possibile riutilizzare mazzo_t anche per implementare altri giochi gestiti dal nostro sistema, anche se necessitano di un numero differente di carte)
c. UTENTE (tipo utente_t): definito in users.h e contenente il nome e la password con cui l'utente si registra al sistema

d. ALBERO DEGLI UTENTI(composto da strutture nodo_t): si tratta di un albero binario di ricerca i cui nodi contengono ciascuno un utente_t (con nome e password), lo stato dell'utente e l'intero associato alla socket utile al server per comunicare con l'utente. Nonostante non si prevedano forme di bilanciamento dell'albero(AVL, R/B), la scelta si rivela comunque valida poiché, supponendo aleatoria la scelta del nome utente su un numero di utenti abbastanza grande e, l'altezza media dell'albero binario e la complessità degli algoritmi di ricerca su tale struttura si mantiene circa O(log n).
e. ARRAY PARALLELI: si tratta di quattro vettori paralleli utilizzati nello script bristat per lavorare sui dati dei file di log e tener traccia dei risultati ottenuti. Vi è un vettore USERS[] che contiene i nomi degli utenti su cui si eseguono elaborazioni e mantiene l'associazione utente-indice per indirizzare i vettori WON[](numero delle vittorie conseguite), PLAYED[](numero delle partite giocate), SCOREW[](punteggio nelle partite vinte) e SCOREL[](punteggio nelle partite perse). Tali strutture rendono agevole il calcolo di medie, e punteggi complessivi per ciascun utente una volta reperitolo nel vettore indice USERS[].
f. MESSAGGIO (tipo message_t): come descritto nel documento di specifica e definito in comsock.h
g. SOCKET AF UNIX: scelte come meccanismo di comunicazione tra i processi client e il server in quanto consentono la comunicazione bidirezionale tra processi appartenenti a gerarchie diverse.
h. SIGMASK, SIGSET: strutture per la gestione dei segnali
i. LISTA CONCATENATA(composto da strutture el_t): definita nel file brsserver.c e i cui elementi contengono ciascuno il thread ID dei thread worker per gestire i client e un puntatore al nodo successivo. Tale struttura è utile nel momento in cui, ricevuto un segnale di SIGINT o SIGTERM, il server deve attendere la terminazione dei thread figli prima di poter terminare e chiudere il proprio canale.

All'interno di brssserver.c la struttura dati albero degli utenti viene dichiarata globale e vi si accede in mutua esclusione completa per mezzo del pthread_mutex semTree. Un'ulteriore variabile globale nPartite, tiene il conto delle partite giocate sul server ed è accessibile in mutua esclusione sul pthread_mutex semTree.
In questo caso l'implementazione avrebbe potuto prevedere un maggiore parallelismo addirittura “a grana fine” su ogni nodo dell'albero ma ciò avrebbe complicato la struttura dati d. e la sua gestione.

III. SERVER
Il server si compone di un main per la creazione dei thread worker che gestiscono le interazioni più un thread apposito per la gestione dei segnali (gest).Si ripercorre adesso il funzionamento del server, soffermandoci nei punti dove si sono incontrate criticità nel corso dello sviluppo.

<u>main</u>
All'avvio del server si inizializza nel processo main un insieme di segnali cui si aggiungono i segnali SIGUSR1, SIGINT e SIGTERM che vengono mascherati nella signal mask relativa al processo: la loro gestione verrà demandata a uno specifico thread handler creato poco dopo. Una volta caricato un primo file di utenti nell'albero (dichiarato globale) si iniziano ad accettare connessioni e a creare thread worker per gestire le richieste ricevute. Ogni thread creato viene aggiunto in testa alla lista concatenata.
Vi è un primo problema di sincronizzazione con il gestore dei segnali: nel momento in cui si ricevono SIGTERM o SIGINT il server deve attendere che si chiudano tutte le connessioni presenti per poter terminare. Non deve dunque accettarne di nuove. Per ovviare al problema si è scelto di utilizzare una variabile globale sigBool, inizializzata a TRUE e messa a FALSE dal gestore dei segnali subito dopo la ricezione di SIGINT o SIGTERM; prima di accettare ogni nuova connessione il server controlla sigBool.

gest
Il thread gest, una volta inizializzato il proprio sigset come nel main, si mette in ricezione dei segnali, gestendoli come da specifica.

worker
Ciascuno dei thread worker creati dal main deve gestire le richieste di un client, per questo motivo si passa come parametro al thread l'indirizzo della socket restituito al server nel momento in cui ha accettato una connessione.
All'inizio dell'esecuzione del worker viene costruita una struttura dati utente_t in cui si memorizzano il nome utente e la password di chi ha richiesto l'accesso al servizio. Il messaggio in ingresso è della forma nomeutente:password. Tutte le funzioni usate in questa parte si trovano in users.

Registrazione/Cancellazione
Se la richiesta da parte dell'utente è di registrazione o cancellazione la sua gestione è relativamente semplice e prevede che si controllino tutti i possibili esiti delle funzioni addUser e removeUser per l'inserimento/cancellazione nell'albero.

Connessione
Più interessante è il caso della richiesta di connessione a seguito della registrazione. Dopo aver controllato i dati di login (o se esiste una precedente connessione dell'utente), si setta il suo stato nell'albero a PLAYING (funzione setUserStatus) per evitare che un altro client acceda con le stesse credenziali e si invia al client la lista degli utenti in stato di attesa (se non è vuota, altrimenti si mette in WAITING l'utente.).
In questo frangente si è previsto che l'utente possa commettere errori nel digitare l'input da tastiera pertanto, se ciò dovesse accadere, si deve inviare una lista di utenti disponibili aggiornata e non la prima inviata. E' allora previsto un ciclo che funziona come segue:
I.si propone una lista (getUserList) inviando un messaggio di tipo MSG_OK
II.se la risposta da parte del client è di tipo MSG_OK si verifica se l'utente proposto è davvero disponibile a giocare (si setta la variabile booleana trovatoAvversario) e in tal caso si segnala con un MSG_OK che l'utente era corretto e si lascia il controllo alla funzione di gestione della partita manageMatch, cui si faceva riferimento nell'Introduzione; altrimenti si invia un MSG_NO e si ripete il ciclo (ovviamente il client sarà in attesa di un messaggio diverso da MSG_NO).
Si inserisce all'interno di questo ciclo anche la gestione del tipo di messaggio MSG_WAIT che l'utente può sempre decidere di inviare e cui il server risponde con un MSG_WAIT.

Attesa
Qualora non ci fossero utenti disponibili a giocare il server il suo stato verrebbe settato a WAITING nell'albero e il client riceverebbe immediatamente un MSG_WAIT.

matchServer
Nel file matchServer.c è contenuto il codice della funzione manageMatch (e potrebbero essere contenuti anche i codici di gestori di partita per altri giochi). Tale funzione ha il compito di gestire le venti mani della partita a briscola e riceve in ingresso i nomi dei giocatori e le relative socket, un booleano per l'eventuale test del sistema e il numero di partita che si sta per giocare (raccolto dal worker in mutua esclusione). Le funzioni utilizzate qui sono contenute in bris.

Per prima cosa la funzione apre il file di log, in cui registrerà tutte le mani giocate, alloca i “talloni” dei punti per i due giocatori e costruisce i messaggi con le carte da inviare nel MSG_STARTGAME di ciascuno degli utenti.
Ogni partita è organizzata su tre fasi e per gestire la corretta alternanza dei turni si utilizza un booleano turnoA (inizializzato a TRUE) in cui si registra se il turno è o meno di chi ha lanciato la

sfida.

Le prime diciassette mani prevedono che si peschi ogni volta dal mazzo, ovvero che il server invii ogni volta una nuova carta. Un ciclo determinato ripetuto diciassette volte riceve la carta giocata dai due utenti, verifica chi ha vinto il turno chiamando compareCard, assegna il corretto valore a turnoA e aggiunge le carte vinte al "tallone" del vincitore. I messaggi inviati in questa fase dal server sono di tipo MSG_CARD e della forma suggerita nella specifica del progetto (t:CC se si è di turno e si riceve la carta CC, a:CC se non si è di turno).
Successivamente un ciclo determinato, analogo al precedente, gestisce le prime due mani in cui si va ad esaurimento delle carte: il server continua a inviare MSG_CARD ma la carta ha sempre il formato "- -".
Il turno finale è gestito a parte poiché si prevede che il messaggio inviato sia di tipo MSG_ENDGAME. In tale messaggio si inseriscono il nome del vincitore e il computo dei punti come da specifica.

IV. CLIENT
Come il server il client si compone di due file.: il primo, brsclient.c, gestisce le richieste di registrazione, cancellazione e connessione, il secondo, match.c, la partita.

Registrazione/Cancellazione
La gestione di MSG_REG e MSG_CANC non presenta particolari complicazioni.

Connessione
Di nuovo la parte più complicata riguarda la connessione al server poiché si deve prevedere la possibilità di scrivere un nome utente sbagliato cui il server risponda con un MSG_NO.
Prima dell'ingresso per gestire questa situazione si pone forzatamente a MSG_NO il tipo del messaggio in entrata. Questo *escamotage* consente di attendere fino all'arrivo di un MSG_OK all'interno del ciclo (cui segue la chiamata a joinMatch()) sia di un MSG_WAIT che viene gestito fuori dal ciclo.

Attesa
Sia che sia ricevuto all'interno del ciclo precedente (a seguito della richiesta esplicita dell'utente di mettersi in attesa) sia che il server non sia stato in grado di fornire un avversario, l'utente si pone in attesa su receiveMessage. Non appena arriva un MSG_OK dal server si passa il controllo a joinMatch().

joinMatch
La funzione, contenuta in match.c, gestisce la partita lato client. La joinMatch "ignora" il gioco cui si sta giocando e provvede semplicemente a costruire la mano dell'utente e a sincerarsi che la carta giocata di volta in volta sia sempre tra quelle possedute dal client e non corrisponda al simbolo "- -" che indica l'assenza di carta nelle ultime mani.
L'END_GAME è gestito fuori dal ciclo delle mani.

V. OSSERVAZIONI SULLA COMUNICAZIONE CLIENT-SERVER
Il modello client/server implementato segue il protocollo *réndez-vous* con sendMessage() e receiveMessage() bloccanti e atomiche. La sendMessage impacchetta un messaggio nella forma tipo:lunghezza:buffer. Il campo lunghezza è sempre lungo 10 caratteri (come il massimo intero rappresentabile) ed è preceduto da tanti "0" quanti ne servono per completare il riempimento del buffer mentre il buffer del messaggio è allocato su misura. In questo modo la receiveMessage() "conosce" esattamente la posizione nel messaggio del tipo e della lunghezza e può facilmente spacchettarlo con tre read() consecutive.
Ogni volta che si sono usate le funzioni sendMessage() o receiveMessage() si è cercato di gestire

eventuali malfunzionamenti o disconnessioni da parte dei client o del server.

VI.BREVE GUIDA PER L'UTENTE

Per compilare il codice si salvano in un'unica cartella tutti i file, compreso Makefile, che compongono il progetto e vi si accede da terminale. Si eseguono poi:

make brsserver
make brsclient

Automaticamente il codice sarà compilato. A questo punto si prepara un file di testo, diciamo users.txt, con i primi utenti da registrare, secondo il formato:

nomeutente1:password1
nomeutente2:password2
…

e si invoca il server come ./brsserver users.txt. Si possono poi aprire altri terminali in cui lavoreranno i client. Per registrare un client si scriverà:

./brsclient nomeutente password -r

Per accedere al server semplicemente:

./brsclient nomeutente password

Per cancellare la registrazione:

./brsclient nomeutente password -c

Il client termina giocando la partita, oppure a seguito di eventi particolari; il server termina lanciando un segnale SIGINT o SIGTERM, premendo la combinazione CTRL+Z o CTRL+C.

VII.TEST SU MACCHINE DEL CLI

Il test automatico è stato eseguito via ssh e superato sulla macchina Olivia del Centro di Calcolo.