Corso di Laurea in Informatica

PROGETTO ESAME CORSO LABORATORIO II

”Il Paroliere” videogioco online scritto in C, multi-threaded, CLI, POSIX, client/server

Professori:
Patrizio Dazi, Luca Ferrucci

Candidato:
Giulio Nisi

ANNO ACCADEMICO 2023/2024

Indice

Note

Si consiglia la lettura del codice in caso di necessitá, in quanto, é stato molto commentato, forse piú del necessario e quindi potrebbe risultare piú chiaro di tale relazione riassuntiva. Il codice é scritto in inglese per essere compreso potenzialmente da un pubblico piú ampio e pubblicato su GitHub. Tuttavia, per ottemperanza alle richieste, le specifiche scritte nel testo del progetto sono tutte rispettate in italiano, ad esempio, il client accetta anche i comandi in italiano (oltre a quelli in inglese), ed entrambe gli eseguibili gestiscono gli argomenti passati da riga di comando italiani (oltre a quelli in inglese). Il codice é stato sviluppato su macOS (Sonoma 14.5) (Darwin Kernel Version 23.5.0) (sfruttando le chiamate POISX supportate) e testato sulla macchina di laboratorio (laboratorio2.di.unipi.it).

> ”Scrivete programmi che facciano una cosa e che la facciano bene. Scrivete programmi che funzionino insieme. Scrivete programmi che gestiscano flussi di testo, perché quella é un’interfaccia universale.”
>
> *Peter H. Salus, A Quarter Century of Unix, Unix philosophy.*

1. Strutture dati ed algoritmi

1.1 Server

1.1.1 Lista giocatori

La struttura dati principale adottata per la gestione dei clients/giocatori, é una lista concatenata di elementi con una struttura e puntatore al prossimo elemento della lista. É stata scelta per:

- Possibilitá di gestire potenzialmente infiniti giocatori, risorse computazionali permettendo.
- Allocazione dinamica on demand che permette di sprecare poca memoria (non preallocandola) quando i giocatori si scollegano (la memoria viene liberata), o sono assenti.
- Tutte le operazioni riguardanti un solo determinato client possono essere svolte dal relativo thread attraverso il puntatore alla struttura rappresentante suddetto giocatore e qui si hanno tutte le sue informazioni che potrebbero servire.

Lo svantaggio sostanziale é che talvolta dobbiamo scorrere interamente la lista sincronizzandoci opportunamente con particolare attenzione, perché, in caso contrario, a differenza di un array per esempio, potrebbero verificarsi problemi molto piú gravi in certe fasi, come nell'aggiunta o rimozione di elementi. Per sincronizzarsi si utilizza (ma non é l'unico) "listmutex".
Tale lista é condivisa tra clients registrati e non.

1.1.2 Matrice di gioco

La matrice di gioco, avendo dimensione predeterminata alla compilazione e costante é allocata staticamente globalmente ("server.h") come matrice di caratteri (char[][]). Il 'Qu'

viene salvato come 'Q', ma gestito opportunamente contando come singolo carattere.

1.1.3 Parole e dizionario

Le parole del file dizionario sono memorizzate in un array "words" di stringhe (char**) allocato dinamicamente, sia l'array, che le stringhe stesse. Le parole sono caricate tutte in memoria, leggendo il file, solo una volta inizialmente, limitando l'impatto delle costose operazioni di I/O con il disco. Vi é poi una copia di "words", "words_valid", allocata in memoria, dinamicamente, peró, NON riallocando le stringhe sprecano inutilmente altra memoria, ma semplicemente copiando i puntatori da "words". Ad ogni inizio nuovo gioco, contestualmente al cambio della matrice di gioco, i puntatori alle stringhe di "words_valid" vengono aggiornati, quelli delle parole presenti nella nuova matrice (trovate con l'algoritmo che segue), rimangono invariati (copiati da "words"), mentre quelli delle parole NON presenti, vengono incrementati fino a raggiungere il terminatore ($\backslash 0$) di stringa. Vi é poi un'altra copia di "words_valid", "words_validated", allocata identicamente, per ogni giocatore REGISTRATO, il cui puntatore si trova nella struttura rappresentate tale giocatore. Quando una parola valida (presente in "words_valid" E in "words_validated") viene sottomessa da un utente registrato, per la PRIMA volta in questo gioco, "words_validated" viene aggiornato, incrementandone il puntatore fino al terminatore, cosícché ad una nuova sua sottomissione della medesima parola, essa possa essere rifiutata senza attribuirne nuovamente il punteggio al player.

1.1.4 Algoritmo di ricerca

Per gestire la ricerca delle parole si é scelto di utilizzare il seguente algoritmo. Ad ogni cambio della matrice di gioco, per ogni parola del file dizionario caricata in memoria, viene visitata tutta la matrice di gioco e per ogni suo carattere (della matrice) invocata la funzione "searchWordInMatrix()", la quale parte dal carattere della matrice specificato (con indici i di riga e j di colonna della corrente iterazione) e ricorsivamente controlla

che i caratteri adiacenti [(i, j+1),(i+1, j),(i,j-1),(i-1,j)] nella matrice siano corrispondenti al prossimo carattere di parola cercato. Se non vi é corrispondenza, o i o j sono fuori dal range di grandezza della matrice, si ritorna un fallimento, altrimenti, se il prossimo carattere di parola voluto é identico al terminatore di stringa, allora significa che precedentemente abbiamo trovato tutte le lettere disposte adiacentemente e quindi ritorniamo un successo. Bisogna prestare attenzione a "marcare" i caratteri giá visitanti per evitare di poter utilizzare piú volte lo stesso carattere della matrice nella composizione di una parola o di perdersi in loop infiniti, nelle chiamate ricorsive successive di funzione. Per quanto riguarda la complessitá? Vediamo un esempio. Per semplicitá assumiamo di avere M, una matrice N x N (utilizzando sempre matrici quadrate), di avere K parole sul file dizionario, e di avere una parola W, da cercare, in input, ossia verificarne la presenza nella lista delle parole E nella matrice corrente di gioco. Con la predetta soluzione, dopo aver letto dal file dizionario le parole ed averle inserite in "words", si dovrá riempire "words_valid" (solo una volta ad ogni inizio gioco), scorreremo quindi "words" in K passi, ad ogni passo iteriamo su M effettuando $N \cdot N$ passi, invocando ad ognuno, "searchWordInMatrix()", la quale per trovare la parola al massimo effettuerá proprio altri $N \cdot N$ passi che rappresentano la parola di lunghezza massima che si puó trovare nella matrice.
Quindi ricapitolando paghiamo $K \cdot N \cdot N \cdot N \cdot N = N^4 K$ la prima volta, a seguire, ogni parola cercata da un giocatore comporterá solamente la ricerca in "words_valid" (ed il controllo in "words_validated", ma questo si fará ad accesso diretto se trovata, dato che gli array sono allineati) con costo K. In conclusione, credo sia una discreta implementazione, che sfrutta la potenza dell'aritmetica dei puntatori, le stringhe sono effettivamente in memoria solo una volta e a fronte di un costo iniziale piú elevato permette di rispondere piú velocemente a seguire ai giocatori.

1.1.5 Coda di fine gioco

Per la struttura del progetto vista, la coda richiesta dalle specifiche, non sarebbe stata necessaria, anzi é risultata un'inutile complicazione. Sarebbe stato sufficiente sbloccare i threads dei clients e segnalarli di inviare il messaggio di fine gioco con la scoreboard al

client gestito. Ogni thread avrebbe potuto inviare il messaggio utilizzando il puntatore alla struttura del giocatore in possesso che fornisce giá tutte le sue informazioni necessarie. Comunque per aderenza alle richieste, la coda é stata utilizzata. Essa é stata implementata come lista concatenata, ogni elemento é una struttura contenente un puntatore al corrispettivo client, un puntatore ad un messaggio (con struttura come richiesta) contenente nel campo ”data”, il nome del client (se registrato, un placeholder altrimenti) ed il punteggio ottenuto nel gioco separati con una virgola, ed infine un puntatore all'elemento successivo. Per la sincronizzazione viene utilizzato (ma non é l'unico) ”queuemutex”.

1.2 Client

Nel client vengono utilizzate due liste allocate dinamicamente. La prima di una struttura contenente stringhe, dove ogni stringa ha una lunghezza massima ”BUFFER_SIZE”. La seconda di una struttura contenente messaggi che rappresentano le risposte ricevute dal server. Per sapere il peculiare motivo di queste fare riferimento alle pagine seguenti. Nessun particolare algoritmo é utilizzato, se non lo scorrimento delle liste.

2. Struttura codice

Il codice si trova tutto nella cartella ”./Src/” (soruces). Suddiviso in tre sottocartelle ”./Server”, ”./Client” e ”./Common”. Le dipendenze sono le seguenti:

”boggle_server” (”paroliere_srv”) eseguibile dipende da ”boggle_server.c”, ”server.c”, ”common.c”.
”boggle_client” (”paroliere_cl”) eseguibile dipende da ”boggle_client.c”, ”client.c”, ”common.c”.

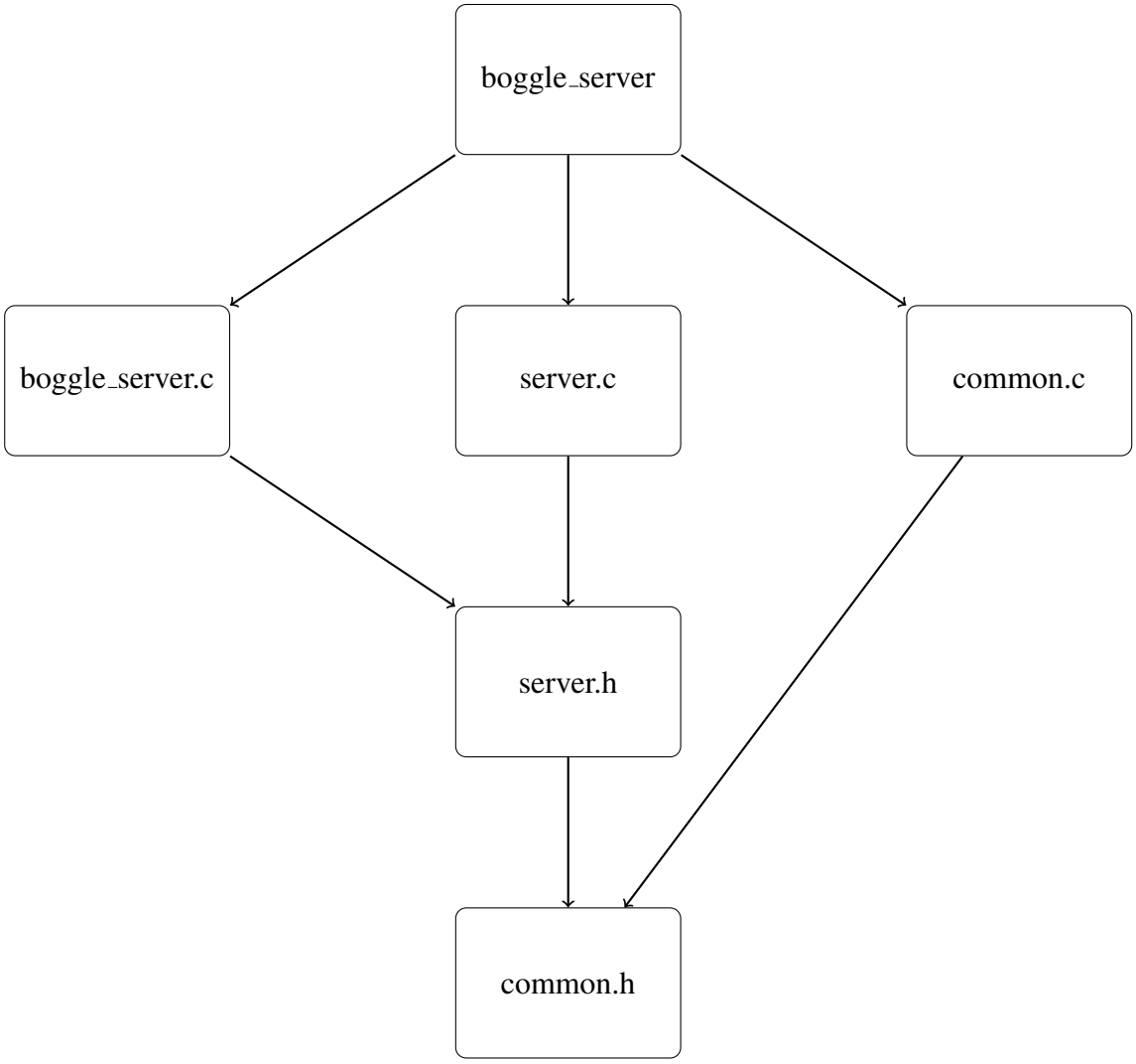
”boggle_server.c”, ”server.c” dipendono da ”server.h”.
”boggle_client.c”, ”client.c” dipendono da ”client.h”.

”server.h”, ”client.h” dipendono da ”common.h”.

”common.c” dipende da ”common.h”.

Si noti che ”common.h” e ”common.c” includono librerie, dichiarazioni ed implementazioni comuni a server e client. ”boggle_server.c” e ”boggle_client.c” contengono solamente i setup e lo scheletro del loro funzionamento. L’effettiva complessa e sostanziale implementazione é demandata rispettivamente ai ”server.c” e ”client.c” files. ”server.h” contiene le dichiarazioni di funzioni e le strutture dati condivise tra ”boggle_server.c” e ”server.c”. Similmente per il client.

A seguire un diagramma (o grafo) delle dipendenze per il server. Quello del client é analogo.



I files eseguibili si trovano nella cartella ”./Bin” e sono compilati qua, alla fine copiati da make nella root del progetto con alias ”./paroliere_srv” e ”./paroliere_cl”.

I files dizionario si trovano in ”./Data/Dicts/”.

I files matrici si trovano in ”./Data/Matrices/”.

Tutto il materiale di tests and debug si trova in ”./Tests/”.

Vi é una cartella ”./Studies/”. Contiene degli snippets di codice per fare prove e tests usati per assicurarsi del funzionamento di alcune meccaniche prima di implementarle nel progetto con il rischio di introdurre tediosi bugs.

3. Struttura programmi

3.1 Server e client in comune

3.1.1 Funzioni comuni

Il server ed il client condividono librerie utilizzate e codice per delle funzionalitá richieste comuni grazie ai files ”common.h” e ”common.c”.
Enunciamo sinteticamente le funzioni condivise. Abbiamo una funzione per fare il parsing dell'IP ottenuto dalla riga di comando, una per trasformare le stringhe in UPPERCASE o lowercase, le fondamentali ”receiveMessage()” e ”sendMessage()” che servono a scambiarsi i messaggi tra server e client, una funzione ”destroyMessage()” per deallocare tali messaggi, una funzione ”createBanner()” che crea delle stringhe carine da stampare per suddividere l'output e facilitarne la lettura. Tali funzioni sono identiche in server e client.

3.1.2 Funzioni comuni ma con differente implementazione

Client e server condividono il possedere un thread dedicato alla gestione dei segnali ricevuti. L'implementazione del thread differisce leggermente tra i due, poiché il server aveva la necessitá di gestire il SIGALRM in modo sostanzioso (perché usato per terminare il gioco), che il client non doveva.

3.1.3 Gestione segnali

Come anticipato, vi é un thread dedicato alla gestione dei segnali. Questo intercetta i segnali ricevuti da gestire, attraverso un loop, con una ”sigwait()”. Inizialmente lo sviluppo é stato fatto con la registrazione dei signals handlers, ma questo esponeva ad innumerevoli difficoltá dovute al fatto che il segnale puó finire ad un thread casuale e sopratutto che nel suo gestore, si é costretti ad utilizzare un numero estremamente ridotto di funzioni che devono avere la caratteristica di essere async-signal-safe. Con l'uso di un thread dedicato

e della ”sigwait()” invece sappiamo giá il thread che riceverá il segnale e possiamo gestirlo liberamente senza preoccuparci di alcuno stato.

3.2 Server

3.2.1 Thread main acceptClient()

Il server, dopo un'iniziale fase di setup, con controllo degli argomenti ricevuti da riga di comando, avvio e configurazione del thread per la gestione dei segnali, registrazione del gestore dell'unico segnale (SIGUSR1) non gestito dal predetto thread, caricamento del file dizionario e matrici (se presente), apre un socket, si mette in ascolto su IP e porta ricevuti da riga di comando, avvia il primo gioco (impostando un timer di fine partita con la chiamata ”alarm()”) e si mette ad accettare clients indefinitamente in un ”while(1)”. All'arrivo di un client (connessione), lo accetta, inizializza un nuovo elemento della lista clients, aggiungendolo a tale lista (con dovute sincronizzazioni) ed infine avvia un thread (”clientHandler()”) dedicato alla gestione del giocatore, passandogli come argomento il puntatore alla propria struttura dati aggiunta alla lista.

3.2.2 Thread clientHandler()

Il neoavviato thread si mette in un loop ”while(1)” ad attendere la ricezione di un messaggio dal client attraverso la funzione ”receiveMessage()”, che al suo interno utilizza la chiamata ”read()”, per leggere le varie componenti del messaggio dal socket utente. All'arrivo di un messaggio COMPLETO tenta di acquisire il proprio mutex. Ogni client, infatti, all'interno della propria struttura dati, ha anche un suo personale mutex ”handlerequest”, che acquisisce durante l'elaborazione di un messaggio ricevuto, per poi rilasciarlo al termine. Tale mutex é posto DOPO la lettura del messaggio, ma PRIMA della sua elaborazione e permette al seguente ”signalsThread()” thread di aspettare che tutte le richieste ricevute siano elaborate prima di terminare il gioco.

3.2.3 Thread signalsThread()

Quando il tempo di gioco termina, si riceve un segnale SIGALRM che verrá intercettato da questo thread gestore segnali ed inizierá la fase piú complessa del progetto. Sincronizzandosi opportunamente bloccherá tutti i threads ”clientHandler()” acquisendo ciascun loro mutex ”handlerequest” dopo che avranno elaborato i messaggi ricevuti. Se non dovesse riuscire ad acquisirne uno, ritenterá. In questo modo si é scelto di dare la prioritá al thread ”clientHandler()” che sta gestendo un messaggio, questo ci garantisce la piacevole proprietá che tutte le richieste ricevute PRIMA dello scadere del timer siano processate come é giusto che sia. A seguire, il thread ”signalsThread()” abilita la pausa, modificando una variabile globale ”pauseon”, che in seguito istruirá i ”clientHandler()” threads di rispondere alle richieste di conseguenza (siamo in pausa). Ad ognuno di questi threads da parte del ”signalsHandler()” che sta gestendo il SIGALRM verrá inviato il segnale SIGUSR1 per il quale nel main era stato registrato un handler (questo segnale NON é gestito dal ”signalsThread()” thread, ma ogni ”clientHandler()” thread lo riceverá e gestirá). Lo scopo di questa azione é interrompere eventuali ”read()” sulle quali il ”clientHandler()” si dovesse trovare, perché adesso (dopo esser stati sbloccati dal ”signalsHandler()” thread con il rilascio del proprio mutex ”handlerequest”) i ”clientHandler()” threads dovranno riempire la coda con i messaggi contenenti il nome e punteggio del giocatore con le dovute sincronizzazioni (usando il ”queuemutex”). Quando la coda sará stata riempita, il ”signalsThread()” thread avvierá il thread ”scorer()” che stilerá la classifica finale recuperando i messaggi dalla coda ed ordinandoli per punteggio discendente con una chiamata ”qsort()”, creando la CSV scoreboard in formato ”nomeutente,puntiutente”. Al termine del ”scorer()” thread, il ”signalsThread()” che lo avrá aspettato con una join, riblocchera tutti i ”clientHandler()” threads (sempre acquisendo ciascun loro mutex ”handlerequest” scorrendo la lista giocatori), li comunicherá che la scoreboard CSV é pronta e loro (i ”clientHandler()” threads), dopo esser stati tutti liberati, invieranno la scoreboard ai propri (corrispettivi) utenti gestiti con un messaggio ”MSG_PUNTI_FINALI”. A questo punto, ”signalsThread()” non dovrá far altro che avviare il thread ”gamePauseAndNewGame()” (poi tornerá a gestire altri eventuali segnali) che eseguirá una ”sleep()” di durata della pau-

sa. Nel frattempo, ovviamente, tutti i ”clientHandler()” threads saranno liberi di operare rispondendo liberamente alle richieste dei giocatori (tenendo conto che la pausa é stata abilitata). Al termine della ”sleep()”, ”gamePauseAndNewGame()” thread, bloccherá nuovamente tutti i ”clientHandler()” threads acquisendo ”handlerequest” mutex, disabiliterá la pausa, imposterá una nuova matrice di gioco (dal file o causale), aggiornerá il ”words_validated” array cercando tutte le parole del dizionario nella nuova matrice, imposterá di conseguenza il ”words_validated” di ciascun giocatore resettandogli anche il punteggio ed infine avvierá un nuovo gioco (nuova chiamata ad ”alarm()”).

3.2.4 Threads scorer() e gamePauseAndNewGame()

Questi threads sono solo di supporto, la loro spiegazione é nella sottosezione precedente (”signalsThread()”).

3.3 Client

L’idea fondamentale sulla quale si é basato lo sviluppo del client é stata la semplicitá. Ho deciso intenzionalmente di demandare tutta la complessitá al server. Per far sí che il client si limiti solamente ad inviare richieste al server e stamparne le risposte. Tutti i comandi vengono inviati al server, ad eccezione di ”fine”. Lo sviluppo del client é stato sorprendentemente quasi piú arduo di quello del server. Un primo problema affrontato é stato il non sapere anticipatamente la grandezza dell’input dell’utente. Non vi sono infatti limitazioni sulla lunghezza della parola o del nome utente da registrare. Per poter gestire input di arbitraria dimensione si leggono ”BUFFER_SIZE” caratteri dallo STDIN con una ”read()”, essi vengono inseriti in un ”char s[BUFFER_SIZE]” statico, all’interno di una lista concatenata di stringhe. Scegliendo cosí un ”BUFFER_SIZE” adeguato allocheremo sempre la dimensione di memoria che piú si avvicina alla grandezza dell’input dell’utente, evitando di preallocare tanta memoria inutilmente che poi comunque se allocata staticamente potrebbe terminare (in caso di input utente grandi), con una lista di stringhe si ovvia a questo problema.

Un altro concetto chiave su cui mi sono concentrato é stato la pulizia della stampa delle

risposte del server e la successiva stampa (e gestione degli input) del prompt. Senza la giusta attenzione, l'interfaccia grafica risultava disastrosa. Puó sembrare un dettaglio, ma é stato impegnativo far fronte a questa questa esigenza, sulla quale mi sono incaponito nel non voler scendere a compromessi. Nello specifico, il problema é che quando viene ricevuta e stampata una risposta dal server, l'utente potrebbe essere nel mezzo di una digitazione e aver giá inserito dei caratteri nel buffer STDIN. Per vedere il problema si puó guardare il codice nel file "./Studies/C/ClientInputOutputSync/brokeninputoutput.c". Ho individuato due modi per affrontare tale problema:

- Prevenirlo. Sviluppando il client come un singolo thread, che attende che l'utente completi l'input (premendo ENTER, che tramite '\n' interrompe la "read()"), poi esegue l'input ed infine controlla se ci sono risposte del server da stampare e ricomincia il ciclo. Oppure, sviluppare il client come multithread e sincronizzare il thread che gestisce l'input con quello che stampa le risposte del server. Entrambe le soluzioni, hanno lo stesso difetto: la "read()" deve essere obbligatoriamente bloccante, per essere sicuri di leggere l'intero buffer STDIN.

- Risolverlo, lasciando che il buffer STDIN possa "sporcarsi" e pulirlo quando necessario. Sembra semplice, ma non lo é, ho trovato solo metodi non funzionanti su internet.

Dopo innumerevoli tentativi sono riuscito a risolvere il problema in modo semplice, ricorrendo a due librerie, ma senza doverne fare un uso estensivo. Ho utilizzato "termios", vedere "./Studies/C/ClientInputOutputSync/termiostests.c" e "fcntl". Le due librerie citate servono solo a rendere la "getchar()" non bloccante e permettermi di pulire l'STDIN. Per approfondire vedere "./Studies/C/ClientInputOutputSync/inputoutputasynctests.c".

3.3.1 Thread main inputHandler()

Il client, dopo un'iniziale fase di setup, con controllo degli argomenti ricevuti da riga di comando, avvio e configurazione del thread per la gestione dei segnali, si collega al server con IP e porta ricevuti da riga di comando e si occupa della gestione dell'input utente e la stampa delle risposte del server (quest'ultime vengono lette da una lista messaggi).

Interrompo la ”read()” molto spesso e controllo se ci sono risposte del server in lista, in caso affermativo, le stampo, ripulisco lo STDIN con una ”getchar()” NON BLOCCANTE, svuoto la lista e ritorno al prompt pulito, in modo che se l'utente dovesse aver digitato qualcosa di incompleto possa riscriverlo e/o modificarlo (a seguito della risposta ricevuta e visualizzata). Invece, se non ci sono risposte dal server in lista, la ”read()” riprende con lo STDIN inserito dall'utente che non si accorge di nulla. L'unico svantaggio é che le stampe delle risposte del server non avvengono in modo asincrono, si ottiene un effetto simile per l'utente, perché la ”read()” viene interrotta ogni pochi millisecondi per stampare le risposte del server.

3.3.2 Thread responsesHandler()

La ricezione delle risposte e l'inserimento in lista invece é eseguita da questo thread ad-hoc e quindi é totalmente asincrono. Questo thread legge le risposte del server dal socket e le inserisce in lista messaggi. Si sincronizza con il precedente mediante il mutex ”listmutex”. La scoreboard CSV di fine gioco ricevuta dal server, viene elaborata e stampata in un formato visualmente gradevole.

4. Struttura tests

Inizialmente l'idea era di scrivere i tests in C in un file eseguibile separato. Ho intrapreso questa strada, testimone il fatto che nella cartella "./Tests/C/" ho lasciato il file "tests.c", il quale funziona, si puó compilare ed eseguire. Tuttavia, con lo sviluppo mi sono reso conto che per come ho strutturato il progetto era disastroso. Avrei potuto testare poche singole funzioni ma non ad un livello soddisfacente nell'insieme. Causa il fatto di avere i codici divisi in piú file ed di aver utilizzato per molte funzionalitá variabili globali condivise. Allora, ho guardato qualche framework di testing in C, ma non ho trovato nulla che facesse al caso mio. Cosí, messo alle strette, ho pensato di approcciare il problema in modo "particolare". Mi é venuto in mente che avrei potuto utilizzare la semplicitá e la potenza di Python, che conosco un poco, per fare i tests in modo esterno, ma simulando proprio il comportamento di utenti ed interazioni con il server anche su larga scala. Ha funzionato benissimo! Solamente **mi dispiace di esser dovuto andare a scomodare un linguaggio che esula dal corso**. Tuttavia, credo sia stato didatticamente utile perché, seppur in un linguaggio diverso, ho dovuto maneggiare files, PIPES e segnali. In questo modo ho potuto eseguire moltissimi tests che sono stati vitali e mi hanno permesso di correggere innumerevoli bugs che con pochi clients (tstati manualmente) non si verificano, soprattutto sviste con la memoria. Per eseguire questi tests é sufficiente:

1. Avviare il server con i parametri desiderati.
2. Dalla root del progetto eseguire "python3 ./Tests/Python/pythontests.py server_ip porta_server".

Ci sono alcuni parametri all'inizio del file che possono essere modificati se voluto. Il file crea tantissimi processi client e per ognuno di essi svolge un numero di azioni che possono essere esattamente tutte quelle che un vero utente potrebbe eseguire. Inviare comandi non validi, registrarsi, richiedere la matrice, sottomettere una parola, di quelle valide e non valide, uscire e persino disconnettersi forzatamente. Tutto viene svolto casualmente con un intensivo uso di random (ma comunque replicabile tramite seed). Tra ogni azione si

aspetta un numero casuale di millisecondi per aumentare l'entropia. I comandi vengono sottomessi da Python, creando il processo del client collegandosi tramite PIPE al suo STDIN. L'input inviato (comandi) e l'output ricevuto, ossia le risposte che il client ha ricevuto dal server, vengono salvati in dei file di logs in "./Tests/Python/Logs/".
I tests dovrebbero funzionare con una recente (al 2024) versione di Python 3, con le librerie installate di default, senza richiedere alcuna risorsa esterna.

5. Compilazione ed esecuzione

Il codice puó essere compilato dalla root del progetto.

Valgono i seguenti targets main:

- make = Compila tutto: client e server.
- make execs = Compila ed esegue il server, con gli argomenti di default (si veda il Makefile).
- make execc = Compila ed esegue il client, con gli argomenti di default (si veda il Makefile).
- make clean = Rimuove tutti i files oggetto, eseguibili, files di logs dei tests Python e termina forzatamente tutti gli eventuali processi in esecuzione di server e client.

Ovviamente oltre a questi nel Makefile é scritto tutto l'albero delle dipendenze.
L'utilizzo degli eseguibili si puó svolgere facendo riferimento al seguente formato:
Server Usage: paroliere_srv server_ip server_port [–matrices matrices_filepath] [–duration game_duration_in_minutes] [–seed rnd_seed] [–dic dictionary_filepath].
Client Usage: paroliere_cl server_ip server_port.

ATTENZIONE: L'utilizzo di alcuni percorsi assoluti hard-coded (ad esempio il percorso del file dizionario di default quando non specificato da CLI) assume che la **WORKING DIRECTORY** sia la **ROOT** del progetto. Stessa cosa per lo script "./Tests/Python/pythontests.py".

6. Miscellanea

Le funzionalitá di cui sono piú orgoglioso sono:

- Il server puó far giocare potenzialmente infiniti giocatori.
- La matrice di gioco puó essere N x N di dimensione arbitraria.
- Le parole del gioco possono avere lunghezza arbitraria.
- L'input dell'utente nel client puó essere di lunghezza arbitraria (parole sottomesse e lunghezza nickname di registrazione).
- Le stampe nel client sono ben strutturate, la GUI non mostra mai stampe incoerenti.
- I tests eseguiti mostrano una buona resa e stabilitá di server e clients.
- Il client é semplice poiché il grosso del lavoro lo esegue il server.

Tutta la sincronizzazione é stata volutamente fatta solamente tramite mutex, dove talvolta le variabili di condizione o i semafori si sarebbero potuti utilizzare, per cercare di rendere il codice piú semplice ed intuitivo possibile, seppur abbia ancora le sue complessitá.
Le parole, indipendentemente da come arrivino dal client e da come siano scritte nei file dizionario vengono convertite in UPPERCASE dal server. Anche i caratteri componenti la matrice di gioco sono gestiti esclusivamente in UPPERCASE, sia nella generazione casuale, sia nella lettura da file (se scritte in lowercase in quest'ultimo vengono convertite in UPPERCASE). Gli unici caratteri ammessi per le parole, le matrici e i nicknames degli utenti sono: "abcdefghijklmnopqrstuvwxyz", contenuto nella definizione "ALPHABET" in "server.h". L'input del client, invece, viene tutto convertito a lowercase prima di essere elaborato ed inviato, quindi il comando inserito "MaTrIcE" sará valido come "matrice".
A tutte le informazioni che possono essere espresse solamente con numeri interi positivi é stato assegnato il tipo "unsigned long int" abbreviato con il "typedef" ad "uli". Per le variabili booleane (o a carattere discreto) é stato utilizzato il tipo "char" poiché occupa solo un byte.

Perché nel server le ”read()” nella funzione ”receiveMessage()”, chiamata dai ”clientHandler()” threads, viene interrotta tramite l'invio (da parte del ”signalsThread()” thread) del segnale SIGUSR1, mentre nel client la ”read()” dello STDIN viene interrotta autonomamente grazie alla libreria ”fcntl”? Semplicemente perché non sapevo si potesse fare con questa libreria quando ho sviluppato il server, che ho sviluppato per prima, quando successivamente, sviluppando il client l'ho scoperto, sono rimasto deluso da tutta l'inutile complessitá che ho dovuto gestire nel server, ma questo funziona ugualmente e mostra piú attinenza agli argomenti trattati nel corso, quindi l'ho felicemente lasciato cosí.
Durante lo sviluppo del client, per tentare di risolvere il problema spiegato sulla sincronizzazione input/output e buffer STDIN, sono finito per sperimentare con una nota libreria chiamata ”ncurses”. Questa permette di gestire il terminale nel dettaglio, ma era inapropiata al mio use case. Risolveva il problema, ma generava una complessitá in tutto il resto insostenibile per le semplici stampe che dovevo fare. Questo mi ha insegnato che anche fare delle ”presumibilmente semplici” interfacce grafiche testuali non é sempre cosí scontato come credessi.
All'interno di ”./Studies/C/SchedPolicyPriorityPThreads/” ci sono dei tests realizzati per un problema serio che ha avuto il progetto. Nel testarlo, é emerso che, all'aumentare del numero di giocatori/clienti connessi, il server diventava sempre piú lento nel gestire le fasi di gioco/pausa, nell'accettare nuove connessioni socket e nel rispondere alle richieste dei giocatori, finché il gioco diventava ingiocabile e il server si fermava per tanto tempo (senza mai stallare peró). Mi sono reso conto che, con l'aumentare del numero di threads, era sempre piú probabile che venisse schedulato un thread ”clientHandler()” (gestore di un giocatore) piuttosto che i thread main (mian ”acceptClient()” che accetta le nuove connessioni) e ”signalsThread()” (gestore delle fasi di gioco) (ed anche i suoi threads di supporto ”scorer()” e ”gamePauseAndNewGame()”), che sono quelli critici per il corretto funzionamento dell'intero progetto. Per il modo in cui é strutturato il progetto, questo era un aspetto critico. Infatti, i threads ”clientHandler()”, per garantire una risposta ottimale ai clients, non si sospendono mai, ma semplicemente si bloccano sui mutexes. Lo stesso thread ”signalsThread()” cerca di acquisire i mutexes utilizzati dai threads ”clientHandler()”. Il risultato, esemplificando con un esempio, é che, se abbiamo un mutex M, 100 threads

”clientHandler()” C, e 1 thread ”signalsThread()” S (lo stesso ragionamento si applica al singolo thread main (”acceptClient()”)) che cercano tutti di acquisire M, é molto piú probabile che venga schedulato un thread C, che acquisisca M, faccia qualcosa, lo rilasci e vada in loop, ora arriva un segnale di fine gioco, il thread S interviene per gestirlo, ma si blocca su M, dato che ci sono 100 thread C, é molto probabile che siano schedulati una, piú volte o potenzialmente infinite, e il thread S attende sul mutex M indefinitivamente, non c’é nessun DEADLOCK, ma il server si ferma per molto tempo prima di accettare nuove connessioni o di terminare il gioco, proprio perché deve ”sperare” che il thread ”giusto” sia S, quello che acquisisce M. Un’intera ristrutturazione del progetto avrebbe risolto il problema, ma non volendo perdere il tempo investito, ho cercato di non demoralizzarmi e cercare una soluzione. La prima idea che ho avuto é stata quella di provare ad aumentare la prioritá di schedulazione dei threads ”acceptClient()” e ”signalsThread()”. Nei files di questa cartella vedrete questo tentativo che, sebbene riuscito, non ha risolto il problema, perché anche impostando la prioritá dei thread ”clientHander()” al minimo possibile, questi hanno continuato ad acquisire i mutexes a spese dei threads s”ignalsThread()” e ”acceptClient()”. Alla fine ho notato che la soluzione potesse essere molto piú semplice e indolore, é bastato aggiungere un mutex di ”alta prioritá” aggiuntivo e questa soluzione si trova nel file ”./Studies/C/SchedPolicyPriorityPThreads/prioritypthreadssolution.c”. Questo ha risolto il problema!