PROGETTO **RETI INFORMATICHE** A.A 23-24 **| CASTELLANI FILIPPO** - 635466

INTRODUZIONE

È stata sviluppata un'applicazione con modello Client-Server che permette di giocare in autonomia o in cooperativa ad una Escape Room. Il servizio sfrutta comunicazione con connessioni TCP (ci serve che il servizio sia affidabile, il ritardo introdotto da tale protocollo non è rilevante ai fini del gioco).

I messaggi vengono scambiati secondo protocollo text, tramite la serializzazione delle seguenti strutture dati:

- *Request (int command, char param1[20], char param2[20], usr[20])*
- *Response (int status, char param[20])*
- *Char buffer[1024]*

Il client manda delle *Request* ed il Server risponde con delle *Response*. Nel caso di messaggi lunghi il server invierà prima una *Response* che comunicherà l'imminente arrivo di un messaggio senza struttura, ricevuto l'ok dal client, il server procederà all'invio. In quest'ultimo contesto si necessitano due risposte per averne effettivamente solo una, aumentando la complessità, ma ci permette di avere uno scambio di dati più leggero negli altri contesti, che coprono la maggioranza delle comunicazioni. Le definizioni dei *command* e degli *status* sono definiti nei file *gameServer,h* e *gameClient.h*.

Durante la partita, ogni volta che il client riceve una *Response* dal server farà partire una richiesta di aggiornamento sul tempo rimanente della partita (non chiederà il tempo alla Response di quest'ultima).

È' stata implementata una sola room all'interno del gioco via codice, ma il servizio è stato strutturato in modo da permettere una futura implementazione con file.

SERVER

Il programma parte sulla porta desiderata dandoci la possibilità di avviare al massimo 10 server di gioco (comando: *start <port>*), ogniuno verrà avviato tramite un processo figlio usando la primitiva *fork()*. La comunicazione tra i processi avviene tramite l'uso della memoria condivisa e dei semafori. In caso di errori con uno dei server figli lo si potrà terminare e riavviare con la stessa o con nuova porta (comando: *restart <id> <port>*). Si possono visionare i server attivi tramite il comando *list*. Infine, con il comando *stop* potremmo de allocare le risorse e terminare tutti i processi, tenendo conto che l'operazione viene bloccata nel caso vi sia almeno una partita in corso.

Sia il server principale che i suoi figli sono basati su sistema I\O multiplexing. Grazie a ciò abbiamo una facile interazione con lo standard input e il numero dei processi viene limitato, di contro abbiamo possibili ritardi nelle risposte (tempi comunque sia trascurabili per il nostro servizio). Per i server di gioco, come detto prima, si è optato per una divisione in processi (ogni partita un processo) ciò ci garantisce un isolamento tra il lato gestionale del server padre e tra le partite stesse, ma richiede l'allocazione di più risorse.

I server di gioco non leggono comandi da tastiera, ma ci tengono aggiornati a video sull'interazione con gli utenti.

Entrambi i server attendo connessioni e richieste da parte dei client tramite la primitiva *select*(). La richiesta viene estratta nella struttura *Request* e poi processata, se non ci sono problemi crea una risposta con la struttura *Response*, la serializza e la invia.

Il server salva via file le credenziali degli utenti nel file *files/users.txt*.

CLIENT

Il programma lato utente presenta due fasi:

- Gestionale: il client si collega al server e chiede all'utente di effettuare il login o di registrarsi. Una volta fatto ciò si ha la possibilità di chiedere la lista dei server disponibili (server attivi e con partite non avviate) con il comando *list* (avvia un una iterazione nella quale si ricevono n+1 *Response* per *n* server disponibili, l'ultima comunica la fine dell'invio della lista, c'è un elevato overhead). Inoltre, tramite il comando *connect <id>* ci si può collegare al server di gioco desiderato. Ci verrà inviata la porta del server di gioco, si chiuderà il precedente socket e ne aprirà uno nuovo con la porta ricevuta, passando alla seconda fase. Durante questa fase si ha una comunicazione iterativa con il server.
- Gioco: il client visualizzerà la lista delle room disponibili dandoci la possibilità di avviarne una con il comando *start <room>*. In seguito, possiamo richiedere informazioni su quello che vediamo con il comando *look [object|location]*. Potremmo raccogliere gli oggetti con il comando *take <object>* (dovremmo rispondere ad una domanda o completare un proverbio nel caso in cui l'oggetto sia bloccato). Se avremo la necessità di controllare il nostro inventario, abbiamo a diposizione il comando *objs* (l'inventario è condiviso tra i player). Gli oggetti si possono usare con il comando *use <object> [object]* (nel caso di doppio parametro gli oggetti potranno esser posizionati in qualunque modo, sarà il server a capire come usarli). In fine avremo la possibilità di terminare prematuramente la partita con end. La partita prevede la possibilità di giocare in solo o in coppia, i giocatori condividono inventario e token. Ogni volta che il client riceve una comunicazione dal server, ne richiede un'altra per poter mostrare a schermo il tempo restante e il numero di token. La partita termina quando si hanno tutti i token con una vittoria, in maniera neutra tramite una *end* e con una sconfitta se finisce il tempo (quest'ultimo viene aggiornato e la partita terminata a causa sua solo se c'è una richiesta da client. In caso di disconnessione prematura del client o per inattività il server potrebbe restare in stallo, in tal caso bisogna intervenire con il comando *restart* dalla console del server). In questa fase il client sfrutta l'I\O multiplexing, per ascoltare standard input e messaggi del server senza limitazioni dovute all'iterazione. Il server deve sempre specificare a cosa sta rispondendo ed il client si deve ricordare cosa ha chiesto.

FUNZIONE AGGIUNTIVA

Il servizio durante la partita permette agli utenti di scambiarsi messaggi con il comando *send <word> [word]*. Per semplicità si possono inviare al massimo due parole da 20 caratteri così da usare la struttura *Request*. Il server invierà a tutti gli altri giocatori un messaggio lungo con il seguente formato: *[mittente] parola parola*

L'invio e la ricezione sono asimmetrici, nel primo c'è bisogno di una sola *send()*, nella seconda il client dovrà fare due *recv()* per il formato lungo.