TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

BATTAGLIA NAVALE MULTIUTENTE

Tesina per il corso di Sistemi Operativi A.A. 2022/2023

Autori:

Elisa Cacace

Francesco Masci

SPECIFICA DI PROGETTO

Realizzazione di una versione elettronica del famoso gioco "battaglia navale" con un numero di giocatori arbitrario. In questa versione più processi client (residenti in generale su macchine diverse) sono l'interfaccia tra i giocatori e il server (residente in generale su una macchina separata dai client). Un client, una volta abilitato dal server, accetta come input una mossa, la trasmette al server, e riceve la risposta dal server. In questa versione della battaglia navale una mossa consiste oltre alle due coordinate anche nell'identificativo del giocatore contro cui si vuole far fuoco. Il server a sua volta quando riceve una mossa, comunica ai client se qualcuno è stato colpito se uno dei giocatori è il vincitore (o se è stato eliminato), altrimenti abilita il prossimo client a spedire una mossa. La generazione della posizione delle navi per ogni client è lasciata alla discrezione dello studente.

SCELTE E TECNICHE DI PROGETTO

Il progetto è diviso in una applicazione server e una applicazione client.
Per far partire il gioco è necessario avviare un server.

Una volta avviato, il server crea un socket che utilizza il protocollo TCP su cui viene eseguito il **binding** con la porta 6500 e sull'indirizzo speciale 0.0.0.0 (entrambi i parametri possono essere modificati a tempo di compilazione con le opportune macro nel file *config.h* della cartella *shared*). L'indirizzo 0.0.0.0 permette al software (e in particolare al socket che stiamo usando per la connessione con il client) di accettare pacchetti provenienti da qualsiasi scheda di rete presente sulla nostra macchina che funge da server. Affinchè il socket possa però accettare connessioni, a seguito della configurazione dell'indirizzo viene chiamata la funzione **listen**.
A questo punto vengono creati dei thread, indicati come WAITING_THREADS, che si occuperanno di accettare le connessioni al server da parte dei client. L'attività dei thread viene sincronizzata utilizzando un mutex, sbloccando inizialmente il primo thread. Per ogni thread la funzione **_waiting_thread** riceve come parametro l'indice del thread stesso.Il thread tenta di bloccare la propria entrata del mutex per capire se è il suo turno. In caso positivo, viene chiamata la funzione **accept** per accettare una connessione sul socket; dopodiché viene sbloccato il thread successivo, viene creato un thread che si occuperà della gestione di quel client (tramite la funzione **`clientHandler`** che analizzeremo più avanti) e il waiting thread torna ad aspettare il suo turno per accettare una nuova connessione futura.
Questa tecnica ci permette di ridurre al minimo il tempo in cui il server non è in grado di accettare connessioni in quanto occupato a creare il thread di gestione del client.

Analizziamo ora come avviene la connessione dal lato client.
La connessione può essere fatta dal giocatore specificando l'indirizzo IP del server al quale vuole connettersi o l'URL di quest'ultimo. Alternativamente possiamo utilizzare la ricerca in locale per cercare un server disponibile che si trova in una delle reti a cui il nostro client è connesso.
In entrambi i casi viene utilizzato un socket UDP per inviare un pacchetto al server sulla porta 6501. Se l'host a cui tentiamo di connetterci è effettivamente un server di gioco questo ci risponderà a sua volta con un pacchetto UDP sulla porta 6502 (entrambi i pacchetti non hanno alcun contenuto). Altrimenti, se così non fosse, al termine di un timeout di 5 secondi, il client stampa un messaggio per informare l'utente che il collegamento non è avvenuto, dandogli la possibilità di ricollegarsi. Nel caso in cui la ricerca venga fatta automaticamente, il pacchetto UDP anziché essere inviato dal client ad un host specifico sulla rete internet verrà inviato in broadcast sulla rete. In particolare, sarà l'utente a specificare su quale rete alla quale è collegato vuole inviare tale pacchetto.
Se il client riceve il pacchetto di risposta dal server, avvia la connessione con il socket TCP descritto sopra e i due software sono effettivamente connessi.

Per comunicare i due software utilizzano le funzioni di invio e ricezione di stringhe e numeri contenute nei rispettivi file *helpers.c*.
Per l'invio/ricezione di numeri, vengono utilizzate le funzioni **`waitNum`** e **`writeNum`.**

In `writeNum`, i numeri inviati sono tutti di 4 bytes e, per far sì che nella comunicazione non ci siano problemi di endianness, viene utilizzata la funzione **`htonl`**, fornita dallo standard di libreria, che permette di convertire la rappresentazione in memoria del numero che vogliamo inviare dall'endianness dell'host (che può essere *big* o *little*) a quello *network* che per convenzione è *big endianness*.
In `waitNum`, una volta ricevuto il numero, l'host lo converte dal byte order della rete alla sua endianness, tramite la funzione **`ntohl`**.
Per quanto riguarda le stringhe, si utilizzano le funzioni **`waitString`** e **`writeString`.**
L'host, prima di inviare una stringa, spedisce un pacchetto contenente la lunghezza della stringa che sta per inviare, utilizzando le funzioni già descritte per l'invio/ricezione di numeri. Questo permette alla macchina che deve ricevere la stringa di allocare un buffer avente dimensione esatta per ricevere la stringa. Per l'invio/ricezione effettivi delle stringhe, le due funzioni si basano sulle sys call **`read`** e **`write`** e tengono conto di possibili interruzioni di quest'ultime, provocate da eventuali errori o dall'arrivo di segnali.
Tutte le funzioni ausiliarie, infatti, sono progettate per garantire l'invio o la ricezione del dato a meno che non ci sia un errore nella system call.

Continuiamo ora ad analizzare il funzionamento del gioco.
Il server mantiene in memoria un array **`players`**, contenente tutti i metadati relativi ai giocatori. Ogni elemento dell'array è di tipo `player_t` e contiene indice del giocatore, il socket per lui aperto, il nickname, un booleano che indica se il giocatore è pronto ad iniziare la partita e un puntatore alla mappa del giocatore, che a sua volta è una struct contenente sia la griglia completa che la lista delle navi. Le funzioni per la gestione dell'array `players` si trovano nel file *player.c*.

Una volta che il client è connesso, la partita viene configurata tramite le funzioni **`clientConnection`** (dal lato client nel file *connection.c*) e **`clientHandler`** (dal lato server nel file *handler.c*).

Il server aggiunge il nuovo giocatore nell'array **`players`** e gli assegna in automatico *'Giocatore i'* come nickname, dove i è l'indice del giocatore nell'array.
Dopodiché il server entra in attesa di un comando da parte del giocatore. I comandi possibili sono:

- **CMD_SET_NICKNAME**
- **CMD_LIST_PLAYERS**
- **CMD_START_GAME**
- **CMD_CLOSE_CONNECTION**
- **CMD_SEND_MAP** (utilizzato nella fase seguente del gioco)

Al client viene quindi data la possibilità di scegliere tra le prime quattro opzioni.

CMD_SET_NICKNAME: permette di cambiare il proprio nickname; il nuovo nickname viene poi inviato al server per aggiornarlo utilizzando la funzione **`setNicknamePlayer`**;

CMD_LIST_PLAYERS: permette di visualizzare la lista di tutti i giocatori attualmente connessi; quando il server riceve questo comando invia al client una stringa contenente tutti i nicknames dei giocatori separati da un ';' così che il client, usando la funzione **`strtok`**, può facilmente separarli e stamparli su stdout;

CMD_START_GAME: indica al server che il giocatore è pronto ad iniziare la partita; una volta che il server ha ricevuto questo comando, imposta il valore *ready* nella struct del player che ha inviato il comando a *true*; se tutti i giocatori hanno segnalato di essere pronti ad iniziare il gioco, il server chiude tutti i waiting threads e invia a tutti i giocatori il comando CMD_START_GAME, che porta la funzione `clientConnection` a terminare, riportando il controllo alla funzione main;

CMD_CLOSE_CONNECTION: indica al server che il giocatore vuole disconnettersi; il server invia lo stesso comando al client (il quale in risposta chiude il proprio socket), lo rimuove dall'array di

giocatori e controlla se tutti i giocatori rimasti nella partita sono pronti a iniziare la partita; se questo è vero si comporta come nel caso in cui abbia ricevuto CMD_START_GAME, altrimenti viene semplicemente fatta terminare la funzione clientHandler per il giocatore che si è disconnesso.

Una volta che il gioco è stato avviato da tutti i giocatori, le applicazioni client entrano nella fase di inizializzazione della mappa, tramite la funzione **mapInitialization**, contenuta nel file *map.c* e servendosi anche delle altre funzioni presenti nello stesso file.

Durante questa fase del gioco, ogni client mantiene in memoria una variabile **map** che contiene una rappresentazione della mappa del giocatore. Abbiamo deciso di realizzare la mappa come un array di `char`, lungo MAP_SIZE * MAP_SIZE.
Ogni carattere di `map` può essere:

- `'0'` → nessuna nave/azione eseguita
- `'1'` → parte di una nave
- `'2'` → nave colpita
- `'3'` → cella senza nave colpita

Inizialmente la mappa viene inizializzata per contenere tutti caratteri `'0'`.
Ulteriormente, il client mantiene anche un array **ships**, il quale mantiene tutte le informazioni relative alle navi, ovvero dimensione, coordinate di riferimento nella mappa e la direzione in cui la nave si sviluppa a partire da quelle.

Il giocatore può scegliere se posizionare le navi nella mappa in modo automatico oppure manualmente.
Di default ci sono quattro diversi tipi di navi da posizionare, di lunghezza da 2 fino a 5. Il numero effettivo di navi di ogni tipo da posizionare è fissato ma può essere facilmente modificato se si vuole.
Se si sceglie di posizionare le navi manualmente si entra nel `map_init_loop`. Al giocatore viene chiesto di selezionare un comando: posiziona nave, elimina nave oppure invia mappa.
Ogni comando è associato ad una funzione ausiliaria che lo gestisce.

_place_ship
Questa funzione viene attivata quando il giocatore decide di posizionare una nave. Se ci sono ancora navi da posizionare, viene mostrata una lista che permette di scegliere tra di esse. Una volta scelta la nave, viene chiesto di scegliere la cella (immettendo le coordinate [es. A 0]) e la direzione in cui posizionarla. A questo punto si esegue un controllo per assicurarsi che nessuna nave è già presente in qualche cella che verrebbe occupata dalla nuova nave. Se la posizione scelta non va bene, se ne sceglie una nuova; altrimenti la nave viene posizionata correttamente (si impostano ad `'1'` tutte le celle coinvolte e si aggiunge la nave all'array `ships`).

_delete_ship
Questa funzione permette di eliminare una nave già posizionata. Se esiste almeno una nave nella mappa, allora viene stampata una lista delle navi disponibili. Una volta scelta la nave, le celle della mappa a essa corrispondenti vengono riportate a `'0'` e questa viene eliminata dall'array `ships`.

_sendMap
Questa funzione si occupa di inviare la mappa completata al server. L'invio è reso possibile solo se tutte le navi sono state posizionate. In questo caso, viene inviato al server il comando CMD_SEND_MAP. Dopodiché viene inviata una stringa contenente le informazioni sulla mappa. Per utilizzare meno memoria, abbiamo scelto di non inviare direttamente tutta la stringa `map` ma di inviare, invece, una codifica dei dati presenti nell'array `ships`, lasciando che sia il server poi a ricostruire la mappa a partire dalla stringa pervenuta, tramite la funzione **makeMap** che si trova nel file *map.c*.

Se invece si è scelto di posizionare le navi in modo automatico, viene chiamata la funzione **_place_ships** .

Innanzitutto si azzera qualunque eventuale contenuto della mappa, settando ogni cella a `'0'`. Successivamente, per ogni nave da posizionare, viene generata posizione e direzione in modo casuale; dopodiché si esegue lo stesso tipo di controllo eseguito nella funzione **`_place_ship`** per verificare che sia possibile posizionare quella nave in quel punto. In caso positivo, si aggiorna sia la mappa che l'array `ships` e si passa alla nave successiva, fino a che non restano più navi da posizionare. Una volta terminato il posizionamento delle navi, la mappa viene mostrata al giocatore, che può scegliere se generare una nuova disposizione (ovvero viene chiamata nuovamente la funzione **`_place_ships`**), se visualizzare la lista delle navi con le rispettive posizioni oppure se inviare la mappa al server (in qual caso verrà chiamata semplicemente la funzione **`_sendMap`**).
Una volta inviata la mappa, il client non mantiene più in memoria né la propria mappa né l'array delle sue navi.

Quando il server ha ricevuto le mappe da parte di tutti i giocatori (ci si assicura che questo avvenga attraverso l'utilizzo di un semaforo), si procede con l'inizializzazione del gioco.
Questa serve principalmente ad inviare ai client i dati relativi alla partita. Il server invia ad ogni giocatore il numero totale di giocatori collegati, il suo indice nell'array `players` e una stringa contenente i nicknames di tutti i giocatori, separati l'uno dall'altro da un `';'`.

Una volta terminata l'inizializzazione del gioco, ogni client entra in attesa del proprio turno di gioco.
Partendo dal primo giocatore, con indice 0, il server invia il comando CMD_TURN per notificare il client dell'inizio del proprio turno. Dopodiché il server entra in attesa di ricevere un comando da quel giocatore specifico. I comandi che possono essere inviati al server sono:

- **CMD_GET_MAPS**
- **CMD_GET_MAP**
- **CMD_MOVE**

Tutte le funzioni utilizzate per la gestione di questi comandi si trovano nei rispettivi file *game.c.*

CMD_GET_MAPS
Richiede di visualizzare le mappe dei giocatori.
Il comando viene gestito dalla funzione **`sendMaps`** nel server; questa si occuperà di inviare al giocatore che ha inviato il comando una stringa contenente le mappe di tutti i giocatori.
La stringa inviata sarà strutturata nel seguente modo: un prefisso che indica l'indice del giocatore, seguito dalla sua mappa, per ogni giocatore.
Una volta ricevuta la stringa, al giocatore vengono mostrate tutte le mappe. Questo avviene usufruendo della funzione **`printMap`**. Questa funzione riceve come parametro oltre alla stringa codificata, anche un valore booleano. Questo determina due modalità di stampa diverse. Nel caso la mappa da stampare sia quella del giocatore, la funzione verrà chiamata con `show_all` impostato a *true*, il che significa che verranno mostrate anche tutte le navi non ancora colpite. Nel caso la mappa sia quella di un qualsiasi altro giocatore, allora `show_all` sarà impostato a *false* e sulla mappa verranno mostrati solamente i colpi sparati contro quel giocatore, sia in caso siano andati a buon fine che se l'abbiano mancato.

CMD_GET_MAP
Richiede di visualizzare la mappa di un giocatore a scelta.
Il comando viene gestito dalla funzione **`sendMap`** nel server.
Nel client invece, una volta inviato questo comando al server, il giocatore deve scegliere quale mappa visualizzare. Questo avviene tramite la funzione **`choosePlayer.`** Come prima cosa viene mostrata al giocatore una lista dei giocatori tra i quali può scegliere. La lista può includere o meno anche il giocatore chiamante. Questo viene determinato dal valore booleano che la funzione accetta come parametro. In questo caso, verrà impostato a *true* (ovvero il chiamante sarà inserito nella lista, in quanto il giocatore può scegliere di visualizzare anche la propria mappa). A questo punto, il giocatore chiamante seleziona un giocatore dalla lista. Il client invia l'indice del giocatore scelto, valore ritornato dalla funzione `choosePlayer`, al server, il quale a sua volta invierà al giocatore la

mappa richiesta. Una volta ricevuta la stringa, il client stamperà la mappa chiamando sempre la funzione `printMap`.

CMD_MOVE
Permette di fare la propria mossa contro uno dei giocatori, attraverso la funzione **`makeMove`** nel client e **`getMove`** nel server.
Anche in questo caso, dopo aver inviato il comando al server, al giocatore viene chiesto di selezionare uno degli altri giocatori (tramite la funzione `choosePlayer` chiamata questa volta con il parametro `_insert_me` impostato a *false,* in quanto il giocatore non può fare una mossa contro se stesso).
Una volta scelto il giocatore che si vuole colpire, si chiede di inserire la cella contro la quale si vuole fare fuoco. Dopodiché viene inviata al server la stringa contenente l'indice e le coordinate e il client torna in attesa del proprio turno.

Dopo ogni turno il server si occupa di inviare aggiornamenti sulla partita ai client.
Il server invia il comando CMD_STATUS ad ognuno dei client, i quali entreranno in attesa di ricevere informazioni dal server.
Quando il server ha ricevuto la stringa, controlla innanzitutto se il giocatore contro cui si è fatto fuoco è stato colpito o mancato o se si è colpita una parte di nave già colpita in precedenza. Verrà generato un messaggio, tramite la funzione **`_make_message`**, che contiene questa informazione e sarà poi inviato a tutti i client.
In seguito viene controllato se qualcuna delle navi del giocatore colpito è stata affondata, generando anche in questo caso un messaggio contenente l'informazione che verrà inviato a tutti i client.
Infine, il server controlla se il giocatore colpito è stato eliminato. In tal caso, la funzione `getMove` restituisce l'indice di quel giocatore ed il server lo invia a tutti i giocatori tranne quello eliminato. A questo punto, i client aggiornano la lista dei nicknames ed il valore del numero di giocatori in partita.
Se si rileva che è rimasto un solo giocatore in partita, sia il server che il client rimasto attivo termineranno, annunciando la vittoria del giocatore. Altrimenti il gioco continuerà, ed il client tornerà in attesa del proprio turno di gioco.
Il giocatore eliminato riceverà, invece, il numero *n_players + 1* e il suo gioco terminerà.
Se invece non è avvenuta nessuna eliminazione, il server invia a tutti i client il numero *n_players + 2* e tutti i client tornano in attesa del loro turno.
Una volta eseguiti tutti i controlli, se la partita non è terminata, il server invia il comando CMD_TURN al prossimo giocatore ed il gioco continua fino alla vittoria di uno dei giocatori.