APPUNTI DISCUSSIONE LSO

COMUNICAZIONE

Nel codice lato server del progetto è stata adottata la comunicazione tramite socket TCP, come si nota dalla riga: socket(AF_INET, SOCK_STREAM, 0);
Questa istruzione crea una socket IPv4 basata su TCP, cioè un protocollo orientato alla connessione che garantisce affidabilità, ordine e integrità dei dati trasmessi.

La scelta di TCP non è casuale, ma risponde perfettamente alle esigenze di un'applicazione come questa, che gestisce un gioco a turni (Tris) tra due giocatori. In un contesto del genere, ogni messaggio — che sia una mossa, una richiesta di rematch o un aggiornamento sullo stato della partita — deve essere ricevuto correttamente e nell'ordine giusto. Non è accettabile, ad esempio, che una mossa venga persa o che i dati arrivino in modo disordinato, perché questo comprometterebbe completamente la logica del gioco.

TCP gestisce tutto questo in modo trasparente. Implementa meccanismi di:
- ritrasmissione automatica dei pacchetti persi,
- ordinamento dei dati ricevuti,
- controllo di congestione e controllo di flusso.

In pratica, l'uso di TCP permette di scrivere un codice server più semplice, perché non è necessario gestire direttamente errori di trasmissione, duplicati o l'ordine dei messaggi. Tutto ciò viene già garantito dal protocollo.

Al contrario, se si fosse scelto UDP (un protocollo connectionless, quindi senza connessione stabile), sarebbe stato necessario costruire a mano tutti questi meccanismi sopra UDP, con un significativo aumento della complessità. UDP può essere utile in contesti dove la velocità è cruciale e qualche perdita di pacchetti è accettabile (come nelle videochiamate o nei giochi in tempo reale), ma non è adatto per applicazioni in cui l'affidabilità è più importante della velocità, come appunto nei giochi a turni.

Per tutti questi motivi, la scelta di TCP risulta tecnicamente solida, coerente con gli obiettivi dell'applicazione e vantaggiosa anche dal punto di vista dello sviluppo, perché riduce il rischio di errori difficili da gestire e semplifica la logica lato server.

CONCORRENZA

Nel server del gioco è stata adottata una struttura in cui ogni client viene gestito da un thread separato, creato dinamicamente al momento della connessione. Questo significa che, quando un nuovo utente si collega, il server principale gli dedica un thread esclusivo, che sarà responsabile per tutta la durata della sessione di quel giocatore: dalla registrazione alla partita, fino all'eventuale richiesta di rivincita o disconnessione.

La motivazione principale è la semplicità. Utilizzare i thread in questo modo consente di scrivere codice che rispecchia molto chiaramente la logica del flusso di gioco. Non è necessario costruire strutture complesse per gestire più utenti contemporaneamente: ogni giocatore ha il proprio flusso indipendente e il codice per ogni sessione può "bloccarsi" tranquillamente in una recv() o in una send() senza preoccupazioni, perché altri utenti stanno lavorando nei loro thread separati.

Inoltre, per un'applicazione come questa, un gioco a turni con un numero limitato di utenti connessi contemporaneamente, questa soluzione è più che sufficiente. Non richiede ottimizzazioni sofisticate o modelli più complessi come select, poll o epoll, che sono utili solo quando si devono gestire centinaia o migliaia di connessioni contemporanee.
Ma qui entra in gioco un punto cruciale: anche se ogni giocatore ha il suo thread separato, ci sono comunque delle risorse che tutti i thread devono condividere.

Ad esempio:

- la lista dei giocatori connessi (player_head)
- la lista delle partite attive (game_head)
- lo stato delle partite in corso

Queste strutture dati sono globali e condivise tra i thread. Questo significa che due giocatori potrebbero, nello stesso momento, voler modificare la stessa struttura (per esempio, uno si disconnette e viene rimosso dalla lista, mentre un altro si connette e viene aggiunto).

SINCRONIZZAZIONE

MUTEX

Per evitare che più thread accedano contemporaneamente e in modo non coordinato a queste strutture, si utilizzano i mutex, ovvero dei meccanismi di blocco. Un mutex funziona come un "lucchetto": un thread lo acquisisce quando vuole accedere a una risorsa, e nessun altro può accedere finché quel lucchetto non viene rilasciato. Un esempio lo possiamo notare nella riga:

```
pthread_mutex_lock(&player_mutex);
// accesso alla lista dei giocatori
pthread_mutex_unlock(&player_mutex);
```

In questo modo, si garantisce che l'accesso sia sicuro e che le strutture condivise non vengano corrotte da modifiche simultanee.

CONDITION VARIABLE

Tuttavia, ci sono casi in cui un thread deve aspettare che qualcosa accada: per esempio, un giocatore ha creato una partita e deve attendere che un avversario si unisca.
In questi casi non basta un mutex, serve qualcosa che permetta a un thread di mettersi in pausa in modo efficiente, finché non viene "svegliato" da un altro thread: ed è qui che entra in gioco la condition variable.
Una condition variable (pthread_cond_t) è legata a un mutex e permette a un thread di sospendere l'esecuzione finché un certo evento non si verifica. Quando un altro thread segnala che quell'evento è accaduto (pthread_cond_signal()), il thread in attesa si risveglia e continua l'esecuzione.

Un esempio nel nostro codice è dato dalle righe:

```
pthread_mutex_lock(&game->mutex_state);
while (game->status != RUNNING)
    pthread_cond_wait(&game->cv_state, &game->mutex_state);
pthread_mutex_unlock(&game->mutex_state);
```

Questo è usato quando un giocatore (il proprietario) crea una partita e attende che qualcuno si unisca. Finché la partita non è "RUNNING", il thread rimane in attesa, senza consumare CPU.
Quando un avversario accetta la partita, il server lo segnala con:
pthread_cond_signal(&game->cv_state);

L'unione di queste tre componenti permette di avere:
- thread indipendenti per ogni giocatore (flussi separati)
- protezione dai conflitti quando i thread accedono a dati condivisi (mutex)
- sincronizzazione intelligente degli eventi (condition variable)

GESTIONE DINAMICA

Nel progetto, le entità principali – cioè i giocatori e le partite – sono rappresentate tramite strutture dati dinamiche (struct PlayerNode e struct GameNode) collegate tra loro a formare delle liste concatenate (linked list).
La scelta delle liste collegate nasce dalla necessità di gestire dinamicamente un numero variabile di giocatori e partite, senza conoscere in anticipo quanti ce ne saranno. In un contesto client-server come questo, infatti, gli utenti si connettono e disconnettono liberamente, e le partite vengono create e chiuse in continuazione.

Utilizzare strutture dati statiche (come array con dimensione fissa) avrebbe posto dei limiti:

- impossibilità di gestire più utenti del previsto senza riallocare memoria
- spreco di spazio se ci sono pochi utenti ma array molto grandi

Con le liste collegate, invece:
- ogni nuovo giocatore o partita viene allocato dinamicamente con malloc
- l'inserimento è semplice ed efficiente, avviene sempre in testa alla lista (O(1))
- la memoria viene liberata appena non serve più (free()), evitando sprechi

Le liste collegate sono molto flessibili, ma non sono prive di svantaggi:
- Ricerca lenta: per trovare un giocatore o una partita bisogna scorrere la lista dal primo all'ultimo nodo (O(n)) e questo va bene per un numero piccolo di elementi, ma non scala bene se il numero cresce.

- Gestione della concorrenza: le liste sono modificate da thread multipli, bisogna proteggerle con mutex per evitare che due thread leggano o scrivano contemporaneamente.

- Debugging più complesso: gli errori come puntatori non inizializzati o doppi free() sono più difficili da identificare rispetto a un array.

NOTIFICA TRA THREAD

Nel progetto del server per il gioco, un aspetto particolarmente interessante riguarda la sincronizzazione tra i vari thread che gestiscono i giocatori. Oltre all'uso classico di mutex e condition variable, che servono a proteggere l'accesso a risorse condivise o a sospendere un thread in attesa che qualcosa accada, è stato fatto un uso molto funzionale dei segnali Unix per comunicare eventi importanti tra i thread.

In particolare, ogni thread rappresenta un giocatore, e mentre un thread può essere impegnato in un'azione come l'attesa nella lobby, un altro thread – ad esempio quello del giocatore che crea una nuova partita – ha il compito di notificare a tutti gli altri che la lista delle partite è cambiata. A questo scopo viene utilizzata una funzione chiamata show_game_changement, che attraverso il comando pthread_kill invia un segnale (SIGUSR1) a tutti gli altri thread che si trovano nella lobby.

Questo approccio ha un vantaggio fondamentale: permette di "svegliare" in modo immediato e mirato i thread interessati, evitando che restino bloccati o in attesa indefinita. Un thread che riceve il segnale può quindi rileggere la lista delle partite e mostrare al giocatore nuove opportunità di gioco. È una forma di comunicazione diretta e leggera che ben si adatta a un contesto in cui i thread sono pochi e ogni utente ha una sessione separata.

Dal punto di vista tecnico, pthread_kill non ha nulla a che vedere con l'uccidere un thread: il nome è fuorviante. In realtà serve solo a inviare un segnale specifico a uno dei thread attivi, identificato dal suo pthread_t. Il segnale stesso viene poi intercettato e gestito da una funzione registrata tramite signal() o sigaction(). Questo sistema è usato anche per altri scopi, come nel caso di SIGUSR2 per notificare l'eliminazione di una partita, oppure SIGALRM per gestire i timeout.

In sintesi, la scelta di utilizzare i segnali per sincronizzare i thread si rivela particolarmente adatta a un ambiente come questo: un server multithread che deve garantire una reazione rapida a eventi asincroni, come la creazione di una nuova partita o la disconnessione improvvisa di un giocatore. È un sistema semplice ma efficace, che completa bene l'uso delle altre tecniche di sincronizzazione già presenti nel progetto.

SIGUSR1
Notifica che è stata creata/modificata una partita
SIGUSR2
Notifica la chiusura o rimozione di una partita
SIGALRM
Gestisce timeout o attese temporizzate
SIGTERM
Termina la sessione o il thread in modo controllato

GESTIONE DEGLI ERRORI

Quando si sviluppa un server che deve rimanere attivo a lungo e gestire più utenti contemporaneamente, uno degli aspetti più critici è prevedere e gestire gli imprevisti. Ogni chiamata di sistema, ogni comunicazione con il client, ogni operazione sulla memoria può potenzialmente fallire. E quando succede, il server non può permettersi di crashare o bloccare tutti gli altri utenti: deve reagire in modo controllato.

Nel codice del server, questa filosofia si riflette in modo molto chiaro: ovunque ci sia una chiamata a una funzione delicata come recv, send, pthread_create, malloc o perfino l'accesso alle strutture condivise, viene immediatamente controllato il valore di ritorno. Se qualcosa va storto, si richiama una funzione centrale: err_handler().

Uno dei punti di forza dell'implementazione è proprio questa scelta: centralizzare la gestione degli errori critici in un'unica funzione. Questo consente di evitare codice duplicato e soprattutto garantisce una reazione coerente in tutti i casi in cui si verifica un errore grave. Quando viene chiamata err_handler(), il server:
- individua il thread (giocatore) coinvolto
- chiude in modo ordinato la sua sessione
- se necessario, rimuove la partita a cui partecipava
- aggiorna la lista dei giocatori e notifica gli altri con segnali (es. SIGUSR2)

Questo approccio fa sì che l'errore di un singolo utente non comprometta il funzionamento dell'intero sistema. In altre parole, l'architettura è resiliente, ovvero capace di isolare e contenere gli effetti di un errore locale.
Un'altra distinzione importante è tra:
- errori recuperabili, come ad esempio un messaggio ricevuto incompleto o una risposta del client assente
- errori irreversibili, come una malloc fallita, o un client disconnesso in modo anomalo

Nel primo caso, il codice spesso riprova o ignora l'evento temporaneamente. Nel secondo, si attiva subito la sequenza di cleanup tramite err_handler.
In alcune situazioni, vengono anche usati timeout (pthread_cond_timedwait, alarm) per evitare di restare bloccati indefinitamente in attesa di eventi che potrebbero non arrivare mai. Questo evita situazioni di deadlock o attese infinite.

PULIZIA

Quando un server è progettato per restare in esecuzione per ore, giorni o addirittura settimane, deve essere in grado di gestire continuamente l'ingresso e l'uscita di giocatori, la creazione e l'eliminazione di partite, e più in generale l'uso dinamico della memoria e delle risorse di sistema.

Questo significa che ogni volta che una struttura dati non serve più, va rimossa correttamente: deve essere liberata dalla memoria, scollegata dalle liste, e tutto ciò che era a essa associato (mutex, thread, socket) deve essere chiuso in modo ordinato.
Nel progetto, questa attenzione alla pulizia emerge chiaramente in più punti, ci sono diversi momenti in cui il server interviene per ripulire risorse:

- Quando un giocatore si disconnette o termina la sessione
- Quando una partita viene conclusa o annullata
- Quando si verifica un errore grave e viene attivato l'err_handler
- Alla fine della partita, quando viene deciso di non fare una rivincita

In tutti questi casi, viene eseguita una rimozione ordinata del nodo (sia esso un giocatore o una partita) dalla relativa lista collegata, seguita dalla liberazione della memoria tramite free().

Il codice segue una logica chiara:
1) Cerca il nodo da rimuovere (giocatore o partita) scorrendo la lista
2) Scollega il nodo aggiornando il puntatore del nodo precedente
3) Chiude eventuali socket aperti (close(sd))
4) Distrugge le eventuali strutture sincronizzative (pthread_mutex_destroy, ecc.)
5) Libera la memoria con free()

Questa procedura viene eseguita anche in condizioni particolari: ad esempio, se il nodo da rimuovere è in testa alla lista, viene trattato in modo diverso rispetto a uno in posizione intermedia o finale. Questo dimostra cura e precisione nell'evitare bug come segmentation fault o memory leak.