Federici Marco165183 Simonetto Luca 166540

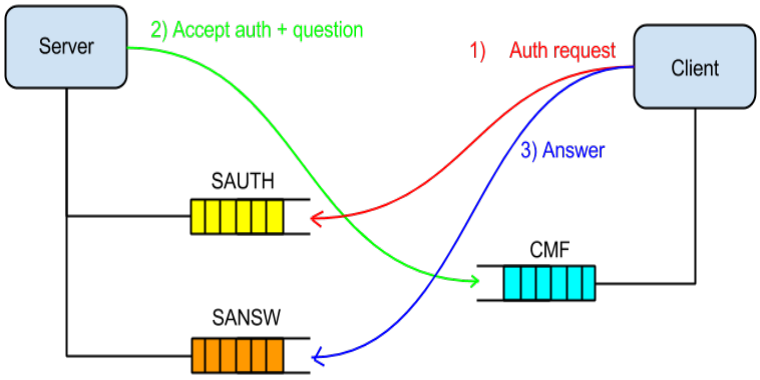
Progetto di sistemi operativi: Multiplayer Game

INTRODUZIONE

Abbiamo deciso di dividere la relazione riguardante il progetto Multiplayer Game in più parti, in modo da poter evidenziare le varie scelte architetturali effettuate nel corso dello sviluppo.
Partendo dalla struttura di base, cercheremo di trattare tutti i punti salienti procedendo per livelli di astrazione fino ad arrivare ai problemi riscontrati e soluzioni adottate nella sezione finale.

SCHEMA DI COMUNICAZIONE

Essendo il programma suddiviso in più processi, uno degli aspetti basilari è stato quello di decidere il metodo di comunicazione tra clients e server.
Il seguente schema rappresenta l'architettura da noi proposta.

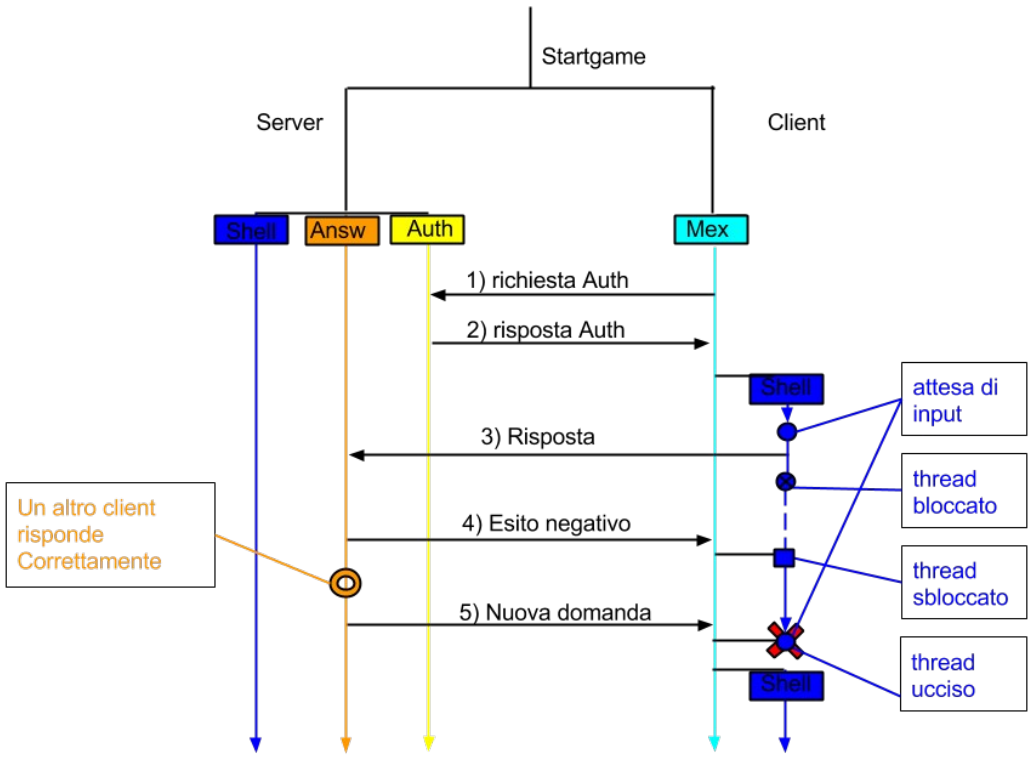


In totale vengono create 2 FIFO per il server e una per ogni successivo client.
La scelta di utilizzare una doppia FIFO per il server è dovuta al fatto che in questo modo è possibile definire una priorità tra diverse tipologie di messaggi.
I messaggi di autentificazione e richiesta di connessione vengono inviati nella FIFO 'SAUTH' e non vengono così messi in coda con le risposte nella FIFO 'SANSW' garantendo un livello di feedback superiore oltre che a una divisione logica dei compiti.

Lo stesso discorso non vale per i clients che utilizzano un'unica FIFO in quanto non necessitano di particolari priorità e, in ogni caso, ricevono un numero di messaggi più contenuto.
Le FIFO del server hanno un nome univoco e predefinito per permettere l'immediato riconoscimento da parte dei clients, mentre ogni client genera una fifo il cui nome dipende dal proprio PID in modo da evitare conflitti e lo comunica al server il fase di autenticazione.

STRUTTURA DEI PROCESSI

Il seguente schema rappresenta la divisione in thread dei processi client e server e la loro interazione.



La gestione della shell avviene sia per il client che sul server su thread separati in modo da non bloccare l'intero processo in attesa dell'input dell'utente.
Tale processo può essere eventualmente interrotto e riavviato sul client dall'arrivo di una nuova domanda o da una notifica per evitare di dover aspettare l'input dell'utente prima di mostrare l'aggiornamento.
Nel server abbiamo distinto logicamente il thread di gestione risposte da quello di gestione delle autentificazioni coerentemente con la scelta effettuata per le FIFO.
Le struttura dati condivisa a tutti i thread del server (ovvero quella contenente l'elenco dei giocatori) viene protetta da un mutex per evitare inconsistenza dei dati.

SCHEMA DEI MESSAGGI E TIPOLOGIA

Per la comunicazione tra le varie componenti del programma ci siamo affidati ad una struttura modulare di messaggi con una sintassi comune ed una lunghezza variabile.
Sintassi: *[TipoMessaggio|Contenuto 1|Contenuto 2|...\0]*
Il primo elemento di ogni messaggio rappresenta la sua tipologia, come ad esempio autentificazione, risposta, domanda eccetera, a cui segue il contenuto vero e proprio contenente l'informazione inviata.
I messaggi scambiati sono raccolti in due gruppi: i messaggi verso i client e quelli verso il server.
-Messaggi *client → server*:

1. **Di autentificazione [Q/R]**: contengono il proprio nome utente per permettere al server di associarne un ID univoco. Permettono anche di notificare la propria disconnessione.
2. **Di risposa alle domande [variabile]**: contengono l'id del client (che gli è stato assegnato dal server), l'id della domanda e la risposta.

-Messaggi *server → client*:

1. **Di autentificazione [A]**: permettono o meno l'accesso al gioco al client che ne ha fatto la richiesta
2. **Di domanda [Q/C/W/T]**: contengono una nuova domanda (Q) oppure l'esito delle risposte date dal client. L'esito può essere risposta corretta (C), risposta errata (W), risposta corretta ma in ritardo (T)**.**
3. **Altro [D/K/N/R]**: associati a comandi server e permettono un migliore livello di controllo della partita. Si può avvisare della chiusura server (D), espellere uno specifico giocatore dalla partita (K), notificare alcuni eventi o stampare a schermo (N) ed infine permettere di visualizzare a fine partita la classifica (R).

Si è scelto di utilizzare un numero variabile di caratteri per ogni messaggio in modo da rendere più flessibile la loro creazione e facilitare la aggiunta futura di altri tipi di messaggio. Ne è un esempio il messaggio di classifica, in cui è presente l'elenco completo di partecipanti, che non sarebbe possibile inserire se si usasse una lunghezza fissa ridotta.
Quando viene letto un messaggio viene eseguita una immediata deserializzazione di quest'ultimo in modo da poterlo contenere in una struttura dati appropriata, facile da passare alle varie routines e veloce da leggere. Questo sistema ha migliorato di molto la facilità con cui sono stati creati i vari moduli del progetto.

COMANDI SERVER

Il lato server del progetto permette di inserire vari comandi da terminale in modo da accedere alle varie funzionalità.
I vari comandi possono o meno richiedere argomenti ulteriori e particolare attenzione si è prestata a questi ultimi. Infatti, essendo possibile inserire più parole come unico argomento, si è deciso di permettere all'utente di racchiudere frasi tra doppi apici per eseguire poi un parsing ed essere interpretate come un unico parametro.
La lista di comandi è visualizzabile digitando *help* e sono:

1. **kick**: espell euno o più giocatori se è seguito dai loro nomi, o tutti se è seguito dalla parola chiave all. (*es. kick Matteo Luca … / kick all*)**.**
2. **question**: invia una nuova domanda personalizzata con relativa risposta ai client.
3. **list**: stampa la lista di tutti gli utenti attualmente connessi, senza un ordine particolare, affiancati dai relativi punteggi.
4. **clear**: "pulisce" la schermata corrente, in modo da rimuovere ogni scritta di intralcio.
5. **notify**: invia una notifica ai clients in modo da poter stampare sul loro terminale il messaggio specificato. Ha solo uno scopo di visualizzazione e non va a modificare parametri di gioco. Utilizza lo stesso sistema di target di kick (*es. notify "Questa è una notifica" Marco … / notify "Benvenuto" all*)**.**
6. **rank**: stampa la classifica aggiornata dei punteggi affiancando ad ogni client i punti guadagnati, sotto forma di barra dalla lunghezza proporzionale ai punti. Questo tipo di visualizzazione rende

molto più semplice e immediato vedere chi sta vincendo. In modalità color le barre hanno una tonalità che va dal verde al rosso per indicare il grado in classifica.

Particolarmente interessante è l'argomento “all” che si riferisce a tutti i giocatori in partita, utilizzato per rendere più semplice il broadcast di un messaggio o un kick di tutti i giocatori. Opportuni controlli sono stati eseguiti per bloccare nomi utente come all o simili, che viene quindi utilizzato solamente come parola chiave.

Per inserire parametri contenenti spazi è spossibile racchiuderli tra virgolette (*es. “questo è un parametro”*)
Questa sintassi è particolarmente utile per i comandi question e notify.

STRUTTURA PROGETTO/MAKEFILE

Il progetto è organizzato in una gerarchia di cartelle pensate per rendere più ordinato e semplice il tutto.
Gerarchia progetto in seguito alla esecuzione di make test, in modo da avere tutti i possibili files disponibili:

1. *bin/* files compilati e pronti ad essere eseguiti
    1. *startGame* permette l'avvio di server e client, creato per semplificare l'uso
    2. *game/* files strettamente di gioco
    3. *test/* programmi usati per il test e la generazione di assets
2. *src/* files sorgente
    1. *client/* sorgenti del client
    2. *server/* sorgenti del server
    3. *common/* sorgenti delle librerie comuni
    4. *test/* sorgenti dei programmi per il testing
3. *log/* logs, utili per troubleshooting e report dei risulati
    1. *gcc.log* contiene eventuali errori di compilazione dei file. Viene cancellato se vuoto
    2. **.log* altri files utili per risalire allo storico delle azioni eseguite dal makefile
    3. *server/* files di output del server, generati ad ogni avvio del programma, con lo storico dei messaggi
    4. *client/* come sopra, per i clients
4. *assets/* contiene tutti gli assets usati come input per il test
    1. *client/* assets client
    2. *server/* assets server
5. *makefile* file necessario per l'avvio del testing e compilazione dei files sorgente

Il makefile è stato organizzato in modo da essere più modulare possibile per non richiamare più volte le stesse funzioni. E' stato implementato il logging delle istruzioni esegute, e per farlo si è dovuto ricorrere a varie funzioni di wrapping, una per ogni target, in modo da semplificare l'utilizzo da parte dell'utente, che in altri casi avrebbe dovuto usare comandi più sofisticati (ad esempio make bin | tee log/build.log)
I target del makefile vengono visualizzati alla chiamata di 'make'.

TESTING

Il testing è stato suddiviso in due tipologie che operano a livelli di astrazione differenti.
La modalità di testing più semplice è volta a testare le strutture di comunicazione e verifica che i messaggi non vengano corrotti e il sistema non vada in crash.
La modalità 'intensive test', invece, genera anche un log teorico della sequenza di operazioni svolte dal server che viene, alla fine dell'esecuzione del test, confrontato con quello generato a runtime dal server stesso per verificare la funzionalità e affidabilità delle strutture di gioco.
I file di test sono così strutturati:

Assets client: I file consistono in un alternarsi di operazioni e pause specificate attraverso un tempo in millisecondi. La prima operazione indica un username che dorà essere inserito, le successive operazioni sono interpretabili come le risposte che il client invierà al server.
Il file è terminato da un 'end' che ne indica la fine.

Server question: il file consiste semplicemente in un alternarsi di domande e risposte relative che il server caricherà consecutivamente.
Il file è terminato da un 'end' che ne indica la fine.

Server log: il file simula un log teorico generato insieme agli altri assets che verrà confrontato con quello prodotto a runtime.
Viene generato solamente per l'intensive test'.

La modalità di test semplice, non concentrandosi sull'output prodotto consiste in una sequenza casuale di domande le cui risposte possono essere solamente 1 o 2 e i clients risponderanno a caso solamente questi 2 numeri, simulando un accuratezza del 50%.
La Modalità di test 'intensivo', invece, genera domande casuali con numeri da 1 a 100 e immette le risposte direttamente negli assets del client secondo una percentuale di correttezza specificata a priori (nel nostro caso 75%) aggiungendo eventualmente la parola 'error' per simulare una risposta sbagliata.

PROBLEMI RISCONTRATI

MULTIPLI MESSAGGI IN ENTRATA

Sorto in stadio avanzato, il problema dei messaggi multipli in entrata ha portato alla creazione di una routine apposita per separarne i contenuti: infatti inizialmente quando veniva letto un messaggio il suo parsing si fermava alla fine della stringa, e non si considerava il fatto che possa essere arrivato uno o più messaggi dopo di esso. Allo stato attuale il programma legge i messaggi in arrivo e alloca lo spazio di memoria necessario a contenerli tutti, andando poi ad iterare in tutto l'elenco creatosi.

DEALLOCAZIONE RISORSE

Un particolare problema è sorto alle uscite dei programmi: infatti l'uso di fifo con nomi costanti dal lato server arrecavano il problema del capire se un server era effettivamente presente. La nostra soluzione è stata creare un handler sia a lato server che a lato client che rileva tutti i possibili tipi di uscita dei programmi, consentendo la chiamata ad una routine di deallocazione dei dati e arresto sicuro. Con questo metodo è stato inoltre possibile la notifica di disconnessione da parte dei client al server in quanto viene mandato un apposito messaggio prima di fare l'unlink di tutte le fifo utilizzate. L'handler ha permesso inoltre un facile debugging del programma in casi di arresto anomalo o segmentation fault in quanto intercettava le uscite di vario genere.

PROBLEMI DI SCHEDULING

Durante l'intensive test, l'ordine in cui vengono schedulati i vari clients va a inficiare sul report prodotto dando origine a eventuali sfasamenti tra log teorico e log prodotto.
Per risolvere tale problema ed evitare ulteriori complicazioni nel progetto introducendo un sistema di sincronizzazione volto solo al testing, l'intervallo di tempo tra le interazioni è stato fissato ad un valore sufficientemente grande da garantire l'ordine prestabilito.
Tale soluzione dipende ovviamente dal sistema su cui viene fatto girare l'intesive test e non risolve concettualmente il problema di fondo ma risulta sufficiente per poter effettuare test realistici.
Ovviamente il test semplice non soffre della stessa tipologia di problema.

RISPOSTE A VECCHIE DOMANDE

Un altro problema riscontrato è stato quello delle risposte in contemporanea alla stessa domanda: il server attribuisce un punto al primo client che risponde e passa alla domanda successiva, leggendo solo in seguito il secondo messaggio contenente la risposta data dal secondo client. In questo caso il server toglie un punto a quest'ultimo dato che la risposta è sbagliata se considerata rispetto alla domanda attuale.
Questo problema è stato risolto associando ad ogni domanda un id progressivo ciclico, in modo da poter associare ogni risposta alla rispettiva domanda, riuscendo a capire se il messaggio è arrivato in ritardo o meno (in questo caso viene mandato un messaggio che indica una risposta corretta ma in ritardo).