**DOCUMENTAZIONE UML**

Nel progetto sono stati implementati due protocolli di comunicazione distinti, RMI e SOCKET, entrambi i quali permettono di eseguire tutte le funzionalità dell’applicazione senza che l’utente percepisca delle differenze nell’uso.  
Prima di entrare nel dettaglio implementativo partiamo col definire quali sono le funzionalità aggiunte a livello di comunicazione:

- Partite multiple sullo stesso Server, gestite tramite un ID autoincrementale assegnato ad ogni singola partita, resa così univoca;  
- Disconnessioni da parte dei Client, registrate nel momento in cui l’*heartbeat* relativo al singolo game non è stato aggiornato negli ultimi 4 secondi. In tale modo, il server gestore della partita, segnala come disconnesso quel particolare player, che potrà rientrare facendo il login utilizzando lo stesso username precedente. Qualora in una partita dovesse essere rimasto un solo giocatore, parte un countdown di 45 secondi, che termina, nel caso non avvenga nessuna riconnessione, assegnando la vittoria al giocatore;  
- Chat, sia globale che privata per ogni coppia di giocatori; esse sono salvate in una lista di *Chat*, a sua volta una lista di *ChatMessages;* per ottenere l’index della chat tra due giocatori, vi è una corrispondenza univoca tra id giocatori e id chat, gestita tramite la classe *ChatIndexManager.*

In visualizzazione TUI, l’arrivo di una nuovo messaggio viene notificato al giocatore, che potrà tramite menu aprire la chat ed accedere ai messaggi, mentre se è già all’interno della chat, vedrà sia la notifica che il messaggio stesso. Nella GUI, invece, le chat si possono aprire dal menù in alto a sinistra, e sono aggiornate ogni secondo, in modo tale da permettere l’immediata visualizzazione dei nuovi messaggi.

E’ stata inizialmente introdotta una classe *Common\_Server,* questa, oltre a far connettere i server RMI e Socket, contiene tutte le informazioni e metodi comuni ed utili ad entrambi:  
oltre alle varie liste di giocatori e partite, infatti, questa classe si occupa principalmente della fase di Login, che tramite ricerca del nome inserito dal client, controllando tra tutti quelli presenti in tutte le partite, permette di capire se un giocatore si sta riconnettendo o meno.  
  
Questa classe è quindi legata ai due Server RMI e Socket, la cui funzione principale è quella di gestione della fase di *Lobby*, ossia quella fase in cui il Client non ha ancora creato un giocatore associato e deve creare/entrare in una partita.  
Una volta creata una nuova partita si viene anche a creare un nuovo *GameServer*, questo è implementato come un server RMI che si occuperà di far comunicare i giocatori (Client Socket o RMI) di una partita col Controller, permettendo così di modificare il Model.  
Ogni partita avrà quindi un suo GameServer privato che permette di rilassare il carico di lavoro dei thread dei server Lobby.

Nonostante i Client rmi e socket implementino le funzioni dell’interfaccia VirtualView, si basano comunque su due protocolli differenti e affinchè si possa comunicare agevolmente con questo server è stata adottata una logica di desincronizzazione dei Client RMI, questi infatti ad ogni funzione chiamata nel server creano un Thread che non terminerà la sua esecuzione finchè il Server non termina l’esecuzione di tutte le sue funzioni interne, pratica che cozza con quello che è invece il protocollo Socket basato su messaggi immediati.  
Per risolvere la questione è stata quindi introdotta una coda di funzioni, in questo modo quando un client vuole effettuare un’azione tramite chiamata a funzione, questa viene inserita in una coda, il Thread invocato dal Client termina desincronizzandosi da tutti i Thread successivi che verranno creati dalle chiamate Server-Controller-Model.  
Le varie funzioni sono state quindi modellate attraverso uno Strategy Pattern che permetta al server di chiamare il metodo *action()* sugli oggetti che vengono prelevati dalla coda.  
Allo stesso modo il server comunica col Client inviando degli oggetti di tipo *Message* che verranno anch’essi inseriti in una coda di messaggi presente nel Client e contengono qualsiasi tipo di informazione utile.

Per poter rendere più efficiente la visualizzazione del model da parte del Client è stata poi introdotta una classe *Minimodel*, che viene aggiornato dal GameServer ogni qualvolta avvenga un cambiamento nel Model e contiene tutte quelle informazioni che il Client potrebbe richiedere più spesso e che hanno bisogno di essere fornite velocemente, senza doversi appoggiare al GameServer per ogni richiesta di visualizzazione di dati.

Immagine che contiene diagramma, testo, Piano, Disegno tecnico

Descrizione generata automaticamente