Nome in Codice

Luca Bazzocchi
luca.bazzocchi2@studio.unibo.it

Giacomo Casadei
giacomo.casadei12@studio.unibo.it

January 2023

1 Abstract

L'obiettivo del progetto è la realizzazione di una versione distribuita del gioco da tavolo *"Nome in Codice"*.

Nel gioco, i giocatori si dividono in due squadre: rossa e blu con un capo ciascuna. Si posizionano sul tavolo 25 carte, ognuna delle quali rappresenta il nome in codice assegnato a venticinque agenti, in modo da formare un tabellone. Queste parole rappresentano agenti rossi, agenti blu, un letale assassino e innocenti passanti. I capi, i quali sono gli unici a conoscere la posizione dei propri agenti, hanno l'obiettivo di farli individuare alla propria squadra prima degli altri e senza beccare l'assassino. Ogni round viene data una parola come indizio e l'effettivo numero di agenti collegati a tale parola.

Tutte le regole sono consultabili al seguente link: Nome in codice (gioco) - Wikipedia.

2 Goal/Requirements

Il sistema è strutturato ad agenti gestiti tramite framework Akka ed è composto di una architettura client-server, formato da due applicazioni separate e che girano su macchine diverse, con l'obiettivo di essere consistenti e fault-tolerant mentre i server e i client sono online.

I giocatori si connettono al server come client per giocare e questo, in maniera distribuita, gestisce le varie partite rappresentate da stanze di giocatori.

Ogni stanza rappresenta una singola partita ed è formata da un numero minimo di giocatori pari a 4.

Ogni persona, all'apertura dell'applicazione, potrà decidere se:

- creare una nuova stanza, ottenendo un codice univoco;
- connettersi ad una stanza già creata tramite l'inserimento di un codice;

- connettersi ad una stanza in maniera randomica.

Ogni stanza presenta una chat per permettere la comunicazione tra i giocatori durante l'assegnazione dei ruoli e durante la partita; inoltre, una stanza si chiude nel momento in cui, per uscita volontaria o crash della connessione, l'ultimo membro esce da essa.

Lo stato del gioco viene condiviso tra i giocatori attraverso l'id della stanza:

- il server, tramite l'id della stanza a cui si vuole accedere, all'ingresso del client gli invia tutti i dati necessari per fargli visualizzare correttamente lo stato corrente della partita, iniziata o meno;
- per ogni azione di un giocatore, che sia: scegliere il ruolo, selezionare una tessera o scrivere in chat, viene inviato al server un messaggio contenente l'id del giocatore, il tipo di azione e il contenuto di tale azione;
- ad ogni aggiornamento dello stato della partita, tale cambiamento viene propagato a tutti gli altri giocatori di tale stanza.

Le connessioni client-server sono gestite nel seguente modo:

- più server sono attivi contemporaneamente, ognuno dei quali presenta zero o più stanze a cui sono connessi i giocatori;
- ogni server presenta una capacità massima di connessioni;
- ogni client può essere connesso solo a un server per volta;
- gli errori dovuti alle connessioni o ai pacchetti inviati sono gestiti in modo tale da mantenere il sistema consistente finchè il server è online.

La parte grafica viene gestita con la libreria ScalaFX mentre tutte le comunicazioni tra client e server avvengono tramite TCP.

Per migliorare la qualità dell'intero sviluppo, si utilizza sbt come strumento di build automation mentre l'ambiente è sottoposto a CI (attraverso GitLab) per controllare in automatico, ad ogni aggiornamento del progetto, che la build e i test di ScalaTest vadano a buon fine.

Nel repository i vari branch sono suddivisi nel seguente modo:

- master: aggiornato solamente all'uscita di una nuova release al fine di conseguire una continuous delivery.
- develop: rappresenta uno stato inconsistente del progetto dove vengono inserite le feature una volta completate e, raggiunta una certa milestone, questo branch viene portato sul master.
- feature: rappresenta una specifica feature in sviluppo.

2.1 Scenarios

Queste applicazioni client e server hanno lo scopo di intrattenere e divertire il giocatore con uno dei suoi giochi preferiti, senza che egli abbia la necessità di portarsi dietro il gioco fisicamente o di avere abbastanza persone disponibili per iniziare una partita. Infatti, in qualunque momento, che sia in pausa pranzo o a casa o in treno, una persona può collegarsi per giocare con i suoi amici, sconosciuti o anche entrambi.

La figura 1 mostra come l'utente utilizzi l'applicazione quando vuole iniziare o entrare in una partita attualmente in corso. Mentre la figura 2, ciò che è in grado di fare l'utente durante una partita.

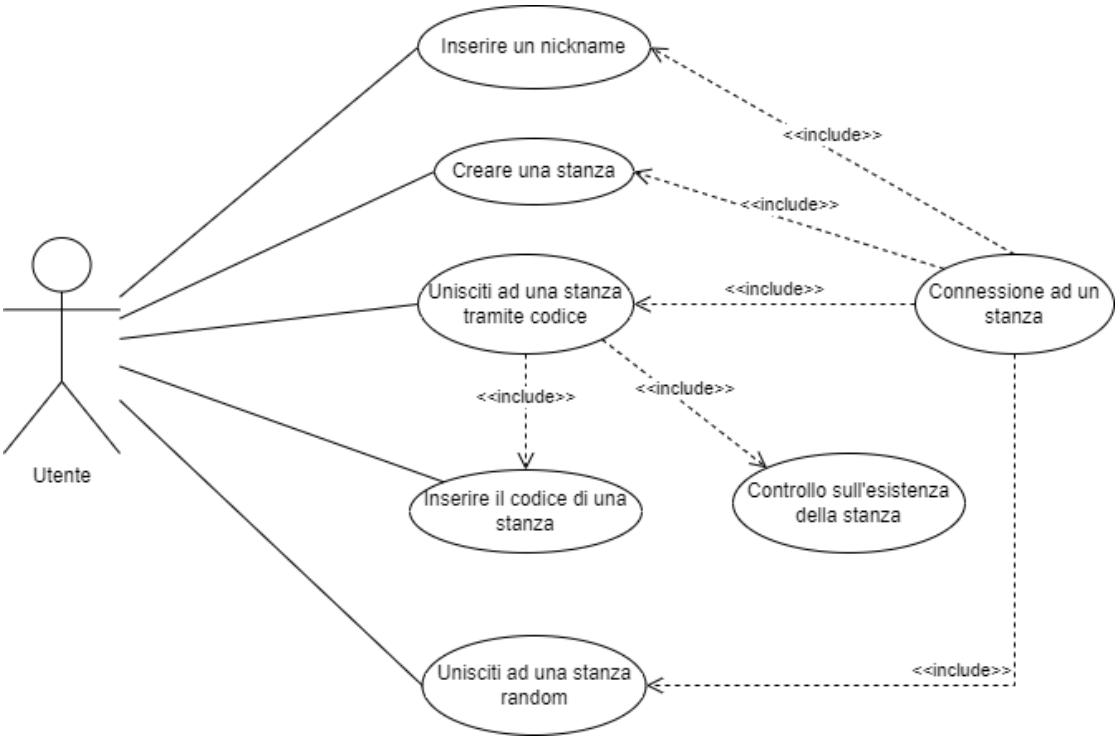

Figure 1: Use case del giocatore che vuole iniziare una partita.

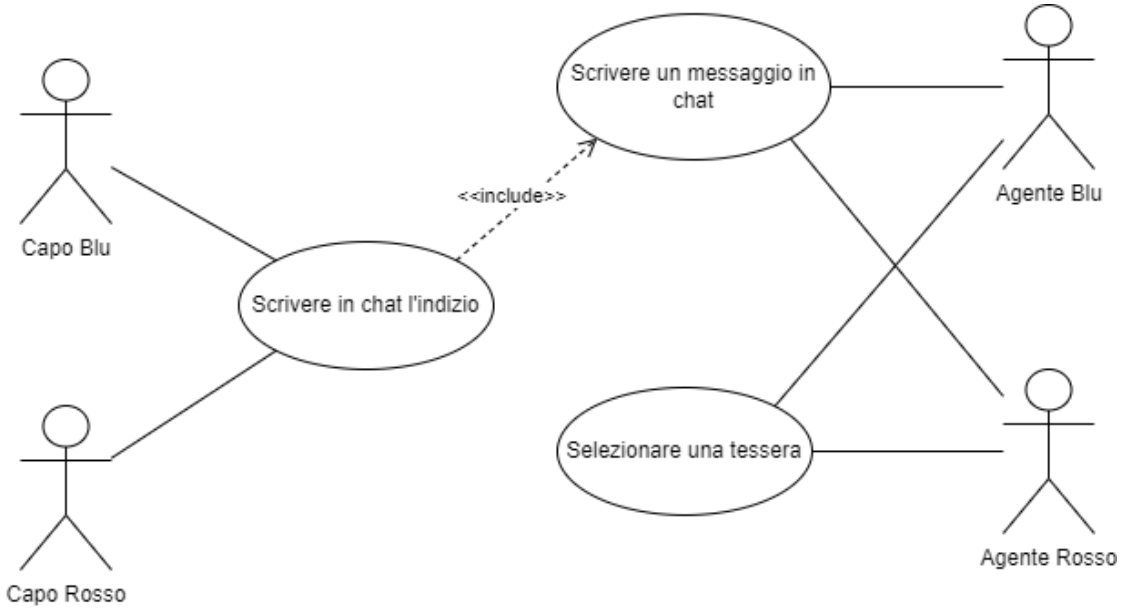

Figure 2: Use case dell'utente mentre gioca.

L'unica azione dei capi è dare l'indizio sulle tessere del proprio colore, mentre gli agenti possono sia scrivere, che selezionare le tessere.

2.2 Self-assessment policy

Essendo il prodotto finale due applicazioni eseguibili, la qualità del software è valutata tramite test che vengono mandati in esecuzione ad ogni release e dall'effettivo raggiungimento della vittoria di uno dei due team mentre sono in corso contemporaneamente più partite in stanze differenti. L'efficacia è data invece dall'arrivo a tale vittoria senza che i giocatori abbiano riscontrato effettivi problemi durante una partita o nell'iniziarne una.

3 Requirements Analysis

Sulla base dei requisiti da implementare e il voler ottenere un software efficace e di qualità, è necessario che, tranne in errori fatali, sia il client che il server siano automatizzati per risolvere le proprie discrepanze senza un intervento diretto dell'utente. Si vuole perciò avere un sistema che risolvi autonomamente gli errori dati da ritardi, perdite di pacchetti o di connessione, senza che il giocatore abbia come risultato il crash dell'applicazione. Verranno utilizzati perciò meccanismi di ack per constatare la corretta ricezione di un messaggio e i logical clock per avere un ordinamento delle azioni di un singolo utente e controllare una possibile duplicazione o mancanza di alcuni pacchetti. Inoltre, nel caso di caduta di connessione, il client proverà prima a riconnettersi sullo

stesso server, per poi cambiarlo con un altro da lui conosciuto, nel caso non ricevesse alcuna risposta.

Per raggiungere la conclusione di una partita, vari dati vengono inviati e ricevuti dai client e server tramite connessioni tcp, è perciò necessario che le applicazioni siano in grado di convertire le azioni degli utenti in oggetti in formato JSON per poterli serializzare all'invio e deserializzarli alla ricezione. Questi messaggi saranno poi controllati per garantirne la corretta gestione nel caso non siano stati corrotti durante il passaggio nella rete.

Inoltre, visto che ogni server dovrà essere in grado di gestire centinaia di connessioni di utenti provenienti da tutto il mondo, il carico di lavoro sul singolo server dovrà essere il più minimale possibile e lasciare che il client capisca e aggiorni da solo la propria schermata sulla base dei nuovi dati sulla partita inviategli dal server.

4 Design

All'inizio del progetto si sono effettuate delle assunzioni sia sulle applicazioni client che server e queste si sono riflesse nel design e struttura stessa del progetto:

1. ogni server e client può essere in qualunque punto del globo;
2. ogni server ha un indirizzo ip fisso che non cambia;
3. ogni client ha memorizzato internamente tutti i server a cui si può collegare e possiede un codice univoco per ognuno di loro;
4. ogni messaggio inviato deve ricevere un ack come risposta;
5. se un server cade, le stanze di tale server vengono perdute;
6. ogni server presenta una capacità massima di connessioni, se un server è pieno, esso rifiuterà nuovi client;
7. se un client non riesce a stabilire la connessione con un server, proverà autonomamente con un altro di quelli che ha memorizzati.

4.1 Structure

Sulla base delle assunzioni precedenti, si è partiti con l'idea di strutturare sia il client che il server utilizzando il pattern MVC per poi convenire di realizzare il client come interfaccia grafica che non necessita del Model (se non per le strutture dati che memorizzano quanto ricevuto dal server), in quanto la logica interna dell'applicazione viene gestita nel server, e di conseguenza il server senza la parte di View.

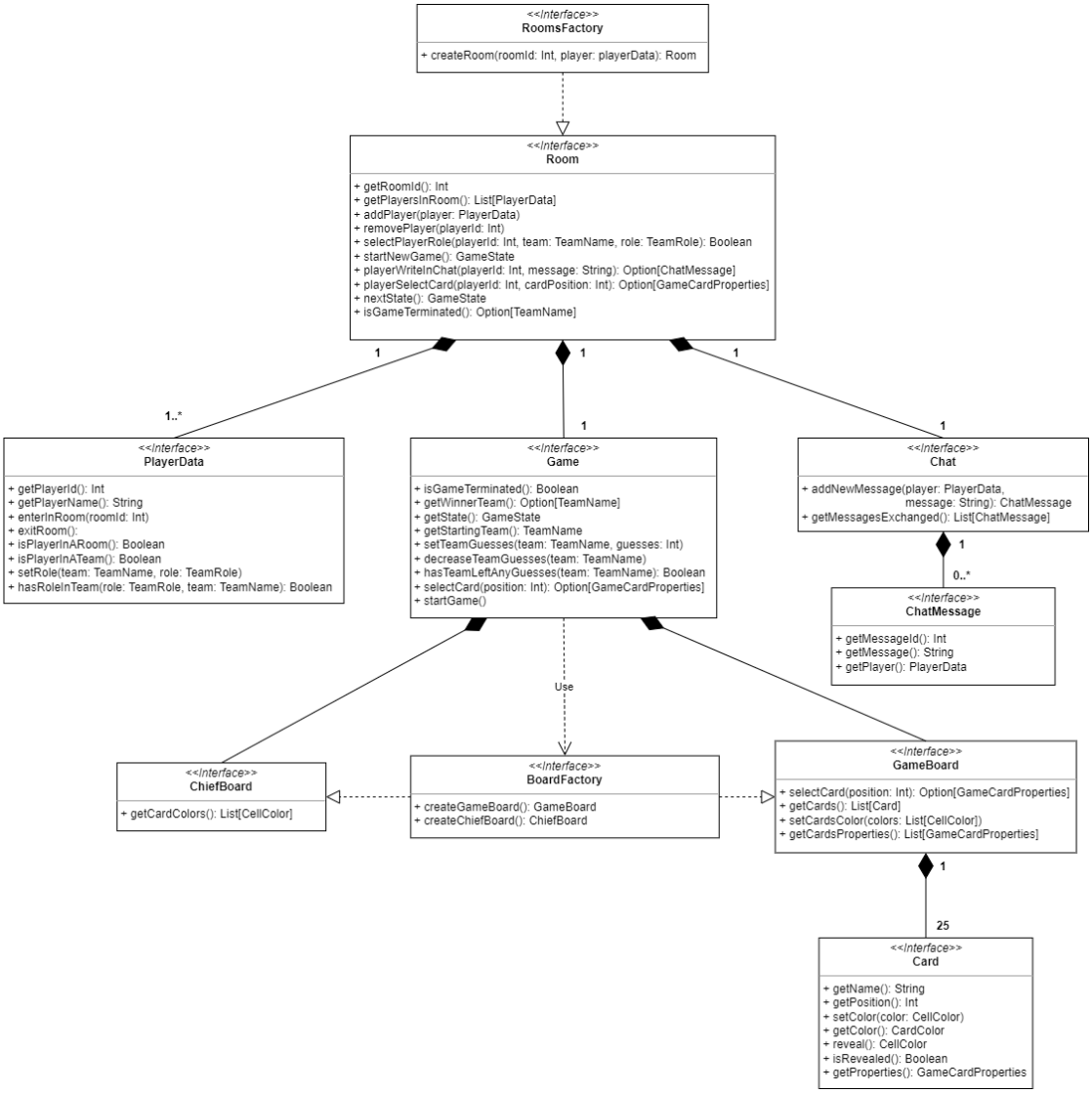

Figure 3: Diagramma delle classi del Model nel Server

La prima struttura a essere stata analizzata è la parte di model del server, in quanto dentro è presente la logica principale che permette il funzionamento del gioco.

Com'è possibile ravvisare in figura 3, all'interno di un server possono essere presenti zero o più stanze, le quali si compongono di:

- *PlayerData*: contiene tutte le informazioni di un giocatore in tale stanza e permette di cambiarne ruolo e team all'interno della stessa stanza. Ogni giocatore è distinto dagli altri tramite l'id assegnato;
- *Game*: mantiene tutte le informazioni sullo stato corrente della partita. Alla

creazione di una stanza la partita non è ancora in corso in quanto è necessario che i giocatori scelgano prima un ruolo; però vengono immediatamente create sia la game board che la chief board per dare ai giocatori una preview di come saranno disposte le tessere in tale partita;

- *Chat*: parte fondamentale insieme alle altre due componenti, in quanto i messaggi dei capi dei team ai propri agenti hanno come effetto il cambio di stato del gioco dal Chief Turn all'Agents Turn. Inoltre, uno scambio di messaggio tra i giocatori è essenziale per una corretta scelta delle tessere da rivelare.

Tramite la stanza e i messaggi ricevuti dai client, tutte queste componenti vengono attivamente modificate finchè almeno un utente è presente all'interno della stanza.

La parte di View del client è invece il componente che permette la visualizzazione dei dati ricevuti dal server.

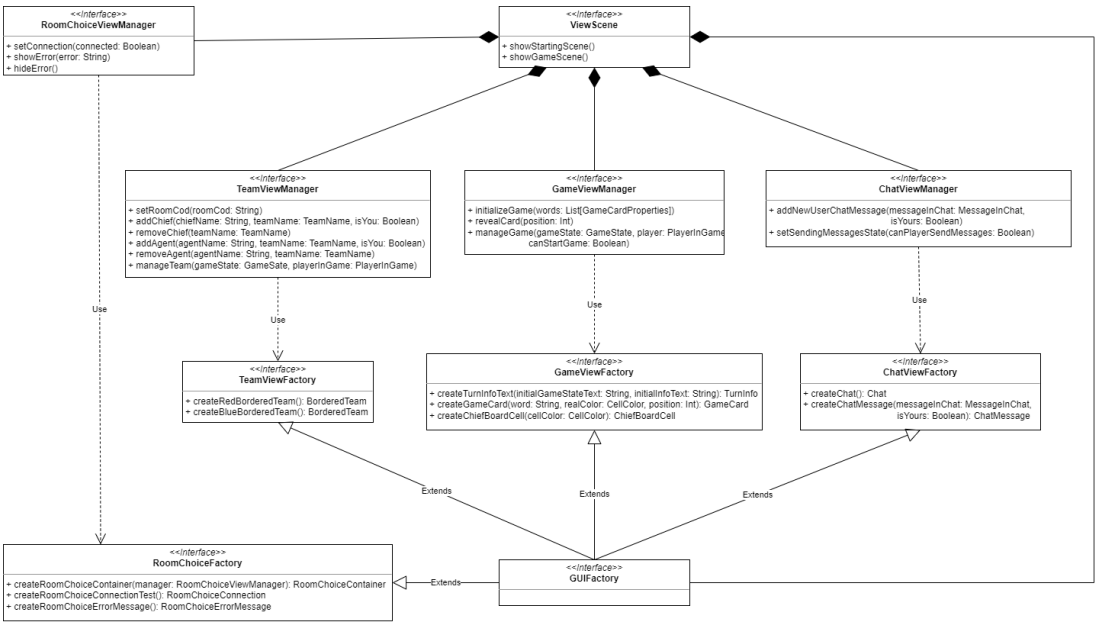

Figure 4: Diagramma delle classi della View nel Client

In figura 4 sono presenti le componenti fondamentali che gestiscono i vari componenti grafici. La view presenta quattro manager, ognuno dei quali gestisce una parte specifica di visualizzazione e utilizza una sottoparte della *GuiFactory* per la creazione dei propri componenti grafici:

- *RoomChoiceViewManager*: manager separato degli altri tre in quanto si occupa direttamente del gioco ma di gestire la scelta di creazione o join in una stanza, con i relativi messaggi di errore;

- *TeamViewManager*: gestisce sia la visualizzazione dei team e i ruoli assegnati agli utenti, sia la possibilità di richiedere o meno il cambio del proprio ruolo sulla base dello stato del gioco;
- *GameViewManager*: gestisce la visualizzazione e interazione con le due board di gioco: la Game Board, visibile a tutti gli utenti ma interagibile solo dagli agenti nel loro turno e la Chief Board, visibile solo ai capi dei rispettivi team.
- *ChatViewManager*: mostra i messaggi scambiati tra giocatori e, sulla base dello stato corrente del gioco, abilita o disabilita la scrittura in chat agli utenti.

Ogni azione dell'utente potrebbe portare a un cambiamento dello stato del gioco e perciò tale azione ha effetto sul model del server. Le varie interazioni vengono portate dalla View al Controller, il quale si occupa di gestire come i messaggi debbano essere convertiti in strutture dati serializzabili per l'invio nella rete, la ricezione di esse e la gestione degli errori.

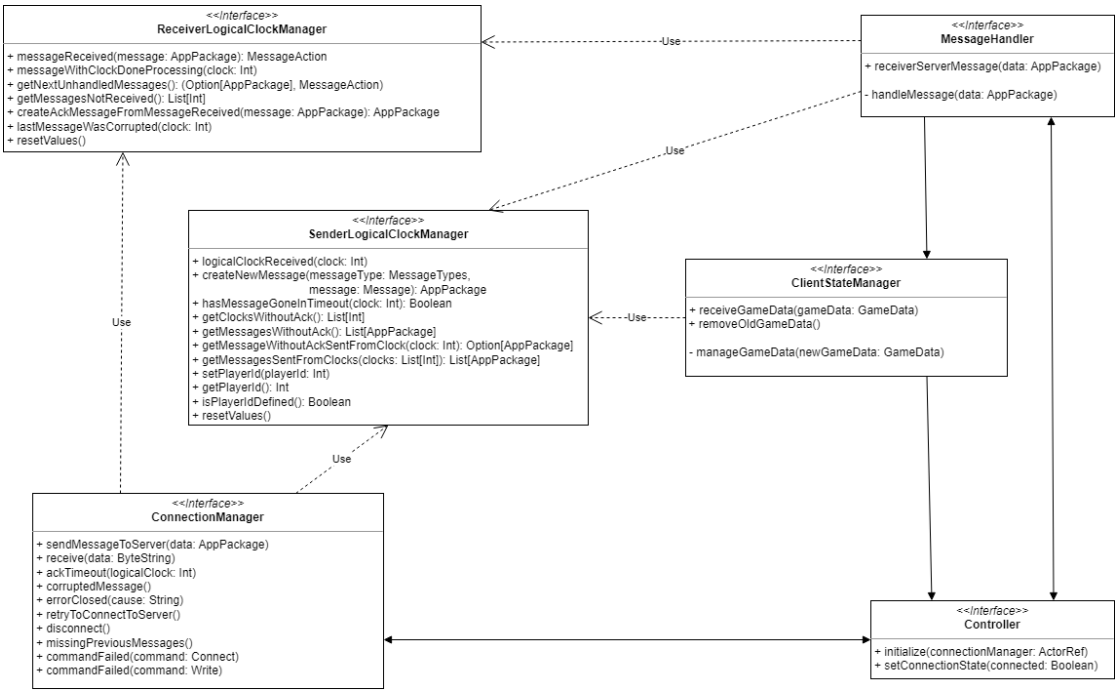

Figure 5: Diagramma delle classi adibite alla gestione dei messaggi nel Client

In figura 5 sono presenti le componenti principali di gestione dei messaggi:

- *ConnectionManager*: attore che gestisce la connessione tcp tra client e server, come anche gli errori che possono essere causati durante lo scambio di pacchetti. Ogni messaggio viene inviato e ricevuto da questo componente;

- *SenderLogicalClockManager*: gestisce il clock logico interno del client e memorizza tutti i messaggi che sono stati inviati nel caso sia chiesto un rinvio o non si è ricevuto un ack come risposta. Alla creazione di un nuovo messaggio, incrementa il clock interno e lo inserisce all'interno di esso, ottenendo così un ordinamento tra i messaggi;

- *ReceiverLogicalClockManager*: gestisce il clock logico del server e memorizza tutti i messaggi ricevuti. Per ognuno di essi, effettua un controllo iniziale, prima ancora che questi siano gestiti e controlla se generare o meno degli errori. Con la ricezione di un messaggio, questo componente decide l'azione da intraprendere, che può essere: Handle, quando il messaggio può essere gestito perchè è arrivato in ordine rispetto a quello gestito precedentemente, Do Nothing, se non può essere gestito al momento perchè è in corso l'handle di un altro messaggio o se è un duplicato, Start Timeout, se mancano dei messaggi e bisogna chiederli allo scadere di un certo lasso di tempo;

- *MessageHandler*: gestisce i messaggi ricevuti la cui azione è "handle" e controlla se ulteriori errori sono presenti all'interno;

- *ClientStateManager*: gestisce lo stato intero del client sulla base dei dati di gioco (GameData) ricevuti dal server. Ogni modifica necessaria alla grafica viene poi passata al controller fino alla view.

- *Controller*: intermediario tra le azioni dell'utente e i cambiamenti grafici dati dai messaggi ricevuti.

All'interno del server il funzionamento riguardante lo scambio dei messaggi avviene in modo similare al client. Com'è possibile vedere in figura 6, i cambiamenti principali sono due:

- il server presenta multiple connessioni, una per client. Perciò sia il ReceiverLogicalManager che il SenderLogicalClockManager tengono conto degli id dei vari giocatori e gestiscono i messaggi per ognuno di loro;

- *RoomManager* al posto del *ClientStateManager*, in quanto ogni messaggio ricevuto dal client contiene l'azione da intraprendere nella stanza in cui il giocatore risiede, come un cambio di ruolo, un messaggio in chat, la selezione di una tessera, etc.

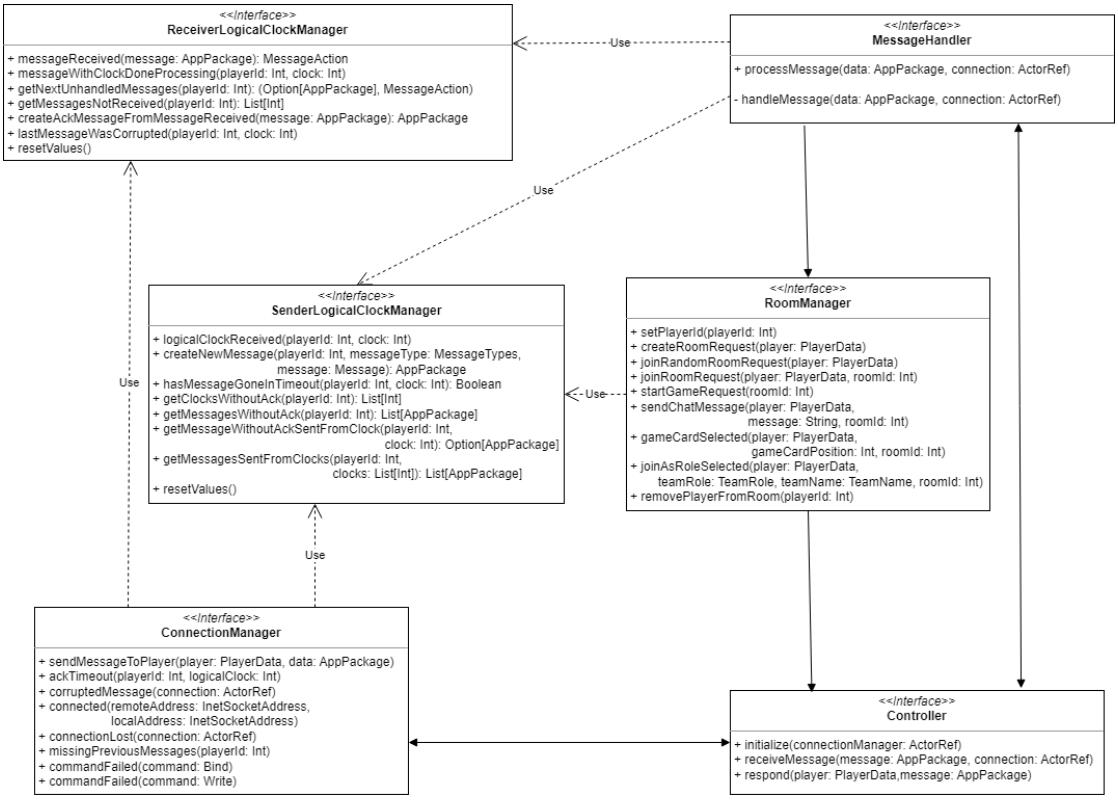

Figure 6: Diagramma delle classi adibite alla gestione dei messaggi nel Server

4.2 Logical Clocks

Per gestire il possibile verificarsi di errori legati ai messaggi quali: il mancato ordinamento, il reinvio e la perdita, è stato concepito un meccanismo che si rifà alla nozione di **Clock Logici**.

Il Logical Clock è un numero progressivo che identifica univocamente ogni messaggio inviato da un'applicazione, che sia client o server.

Al momento della ricezione di un messaggio, viene confrontato il suo clock logico con quello atteso e, in base alla relazione tra essi, si verifica uno dei casi seguenti:

- **in sequenza**: quando il clock logico coincide con quello atteso il messaggio viene gestito e si attende il successivo;
- **fuori sequenza**: si verifica nel caso in cui il clock logico è maggiore di quello atteso ed entro un certo limite massimo prestabilito. Il messaggio viene dunque inserito in una coda in attesa che arrivino i messaggi mancanti. Se ciò non avviene entro un certo limite di tempo, vengono richiesti i messaggi il cui clock logico è compreso tra quello atteso e quello ricevuto;

- **fuori range o duplicato**: se il clock logico è superiore al limite massimo prestabilito o inferiore a quello atteso il messaggio viene scartato, supponendo perciò che si tratti di un messaggio corrotto o una duplicazione.

Alla ricezione viene inviato un messaggio di ACK che possiede lo stesso clock logico del messaggio ricevuto per informare il mittente dell'avvenuta consegna. Così facendo, questi non contano al fine della progressione dei clock logici; in caso non venga ricevuto nessun ACK entro un tempo limite, il messaggio viene reinviato.

Il client memorizza il proprio clock logico e quello del server a cui è attualmente connesso, mentre ogni server mantiene, oltre al proprio, i clock logici di tutti i client a lui connessi.

4.3 Messages between Client and Server

Tutta l'interazione tra client e server è una interazione basata sui messaggi. Perciò come un messaggio sia strutturato, rappresenta una delle parti fondamentali del progetto.

Ogni messaggio è una struttura dati denominata *AppPackage* che contiene al suo interno quattro componenti:

- **Player Id**: identifica chi invia o riceve il messaggio;
- **Message Type**: indica il tipo di messaggio contenuto. Utilizzato sia per capire se il messaggio risulta corretto, sia per identificare il tipo del componente *Message* da estrarre;
- **Message**: il messaggio specifico sotto forma di stringa. Esso viene poi convertito sulla base del type;
- **Logical Clock**: il clock logico a cui si trova il sender al momento della creazione di tale messaggio. Permette un ordinamento tra i messaggi.

Esclusi i messaggi di errore o di configurazione, che presentano comunque la medesima struttura, i messaggi del client sono vari in quanto indicano l'azione intrapresa dall'utente, mentre i messaggi del server, riguardanti l'aggiornamento del gioco, sono di una sola tipologia: **GameData**.

La scelta di avere una sola tipologia è presa al fine di alleggerire il carico di lavoro sul Server. Infatti, sulla base del messaggio ricevuto da un client, il server aggiorna solamente la struttura dati GameData scegliendo unicamente se l'azione è applicabile o meno, senza dover analizzare per ogni client le singole modifiche da applicargli. In questo modo, per ogni azione che ha modificato il GameData, il server invia ad ogni giocatore in quella stanza, il medesimo GameData; è poi il client che, analizzando le differenze tra il corrente GameData e quello appena ricevuto, aggiorna le componenti grafiche opportune. Nel client tutta questa gestione avviene all'interno del *ClientStateManager*.

4.4 Behaviour

Sulla base del gioco da tavolo e sul come vengono svolti i vari turni, sono stati identificati i seguenti stati:

- **Role Selection**: stato iniziale di ogni stanza. In questo stato è possibile scegliere un ruolo, cambiarlo e scrivere in chat. Alla richiesta di start game si passerà poi al turno del capo di uno dei due team.
- **Red Chief Turn**: turno in cui il capo della squadra rossa decide l'indizio che raggruppi più parole possibili, insieme alla loro quantità. Solo egli può scrivere in chat e solamente l'indizio.
- **Red Agents Turn**: turno degli agenti della squadra rossa. Essi possono scrivere in chat e selezionare le tessere finchè non raggiungono la quantità prefissata o sbagliano tessera.
- **Blue Chief Turn**: stessa modalità del turno del capo rosso, ma per il capo blu.
- **Blue Agents Turn**: stessa modalità del turno degli agenti rossi, ma per il team blu.
- **Red Team Victory**: rappresenta la vittoria della squadra rossa, per aver trovato tutte le tessere della propria squadra o per non aver scelto l'assassino.
- **Blue Team Victory**: rappresenta la vittoria della squadra blu, per aver trovato tutte le tessere della propria squadra o per non aver scelto l'assassino.
- **Waiting Players**: stato in cui la partita è ancora in corso ma la disconnessione di alcuni giocatori ha portato le squadre a non avere un numero sufficiente di persone. Appena raggiunto il numero minimo, si torna allo stato precedente.

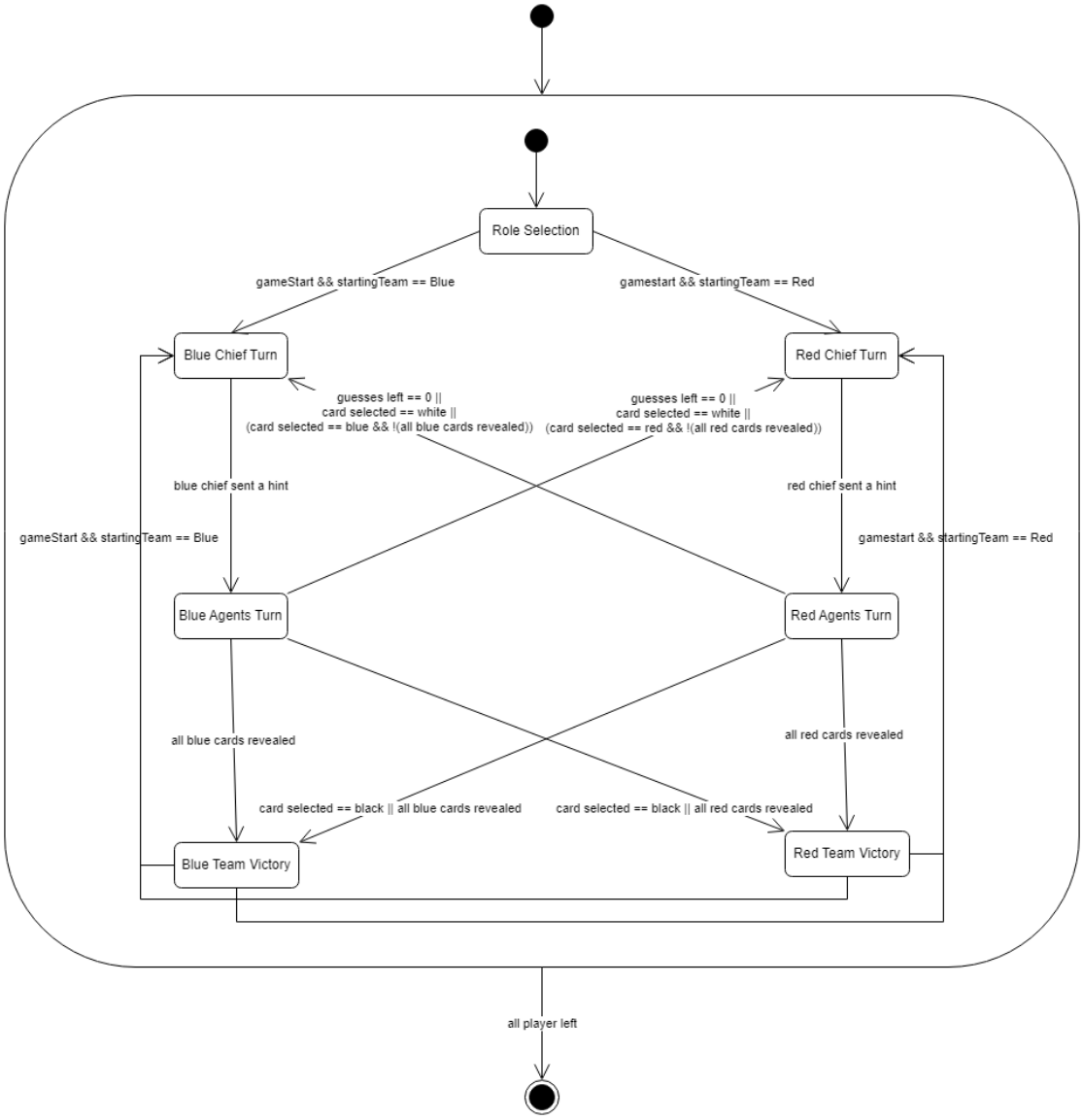

Figure 7: Diagramma degli stati di una partita generica

In figura 7 è possibile osservare come cambi lo stato della partita sulla base delle azioni dei giocatori. Alla creazione di una stanza lo stato iniziale di una partita è sempre in *Role Selection*, per poi continuare fino alla vittoria di uno dei due team. Se a questo punto una nuova partita dovesse iniziare, in modo casuale, verrà nuovamente scelto uno dei capi dei due team. In qualunque momento, se tutti i giocatori abbandonano la stanza, tale stanza verrebbe eliminata e la partita terminerebbe a sua volta.

Per non complicare eccessivamente la comprensione della figura 7, è stato scelto di non rappresentare lo stato *Waiting Players*, in quanto al ritorno di un numero di persone

sufficienti per team, lo stato tornerebbe a quello precedente e ciò porterebbe ad avere un collegamento tutti gli altri stati fatta eccezione per la selezione dei ruoli.

4.5 Interaction

Sulla base di quanto specificato in precedenza, in questa sezione viene mostrato il flusso di alcune interazioni tra client e server.

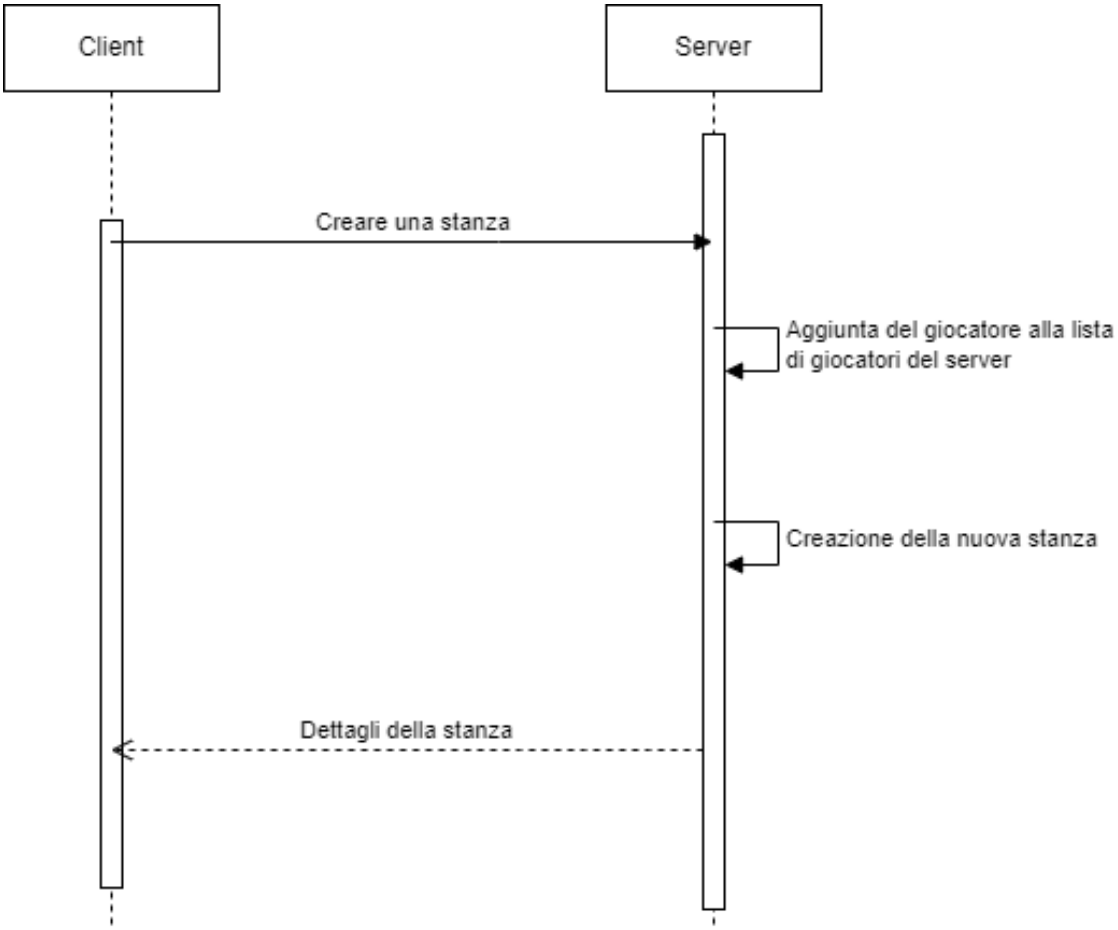

Figure 8: Sequence diagram del giocatore che vuole iniziare una partita creando una nuova stanza.

La figure 8, 9 e10 mostrano le differenze tra le varie richieste nella creazione/join di una stanza.

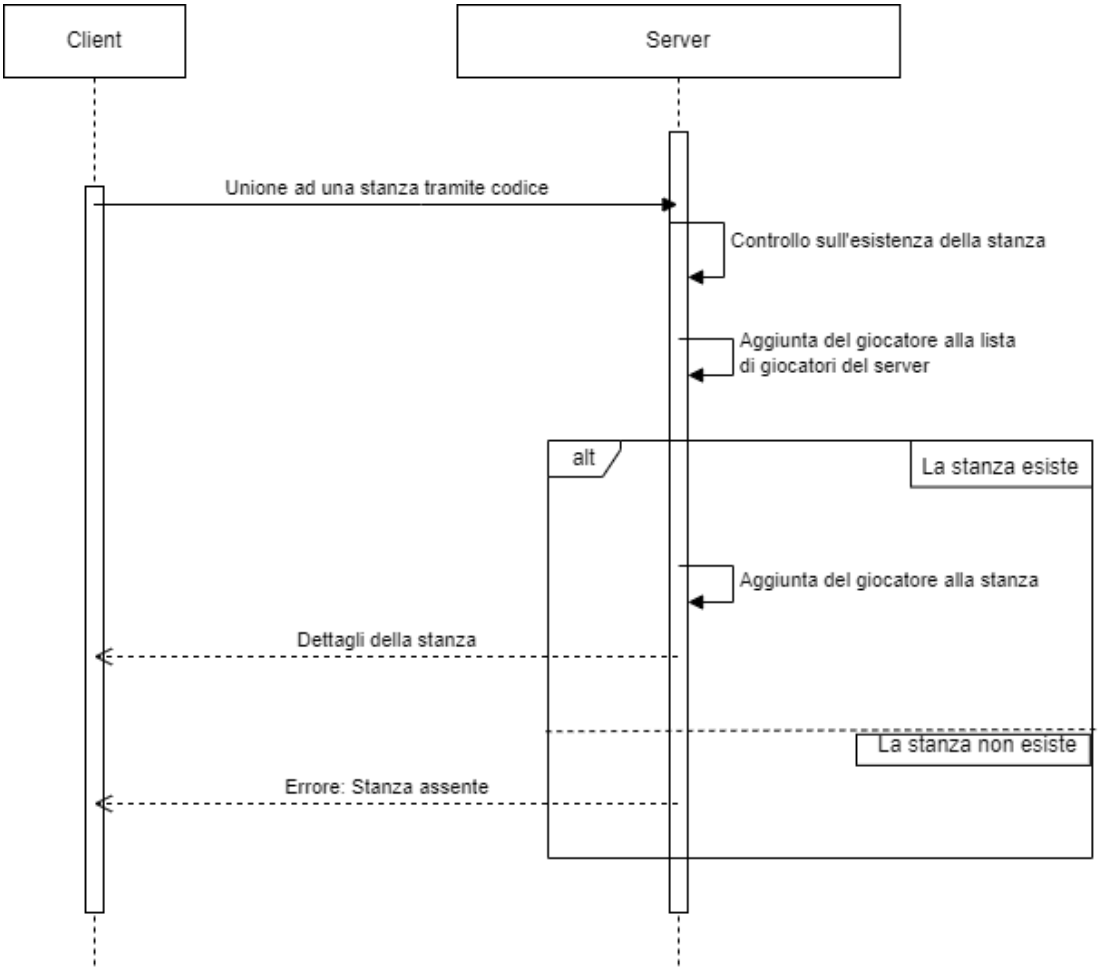

Figure 9: Sequence diagram del giocatore che vuole iniziare una partita unendosi ad una stanza già presente.

La join a una stanza è peculiare in quanto, oltre a quello che è mostrato in figura 9, se il client rileva precedentemente che il codice della stanza appartiene ad un altro server, prima di tutto effettua una disconnessione con esso e, solo dopo essersi connesso al nuovo server, gli invierà la richiesta di join. Maggiori informazioni sono presenti nella sezione 5.6.

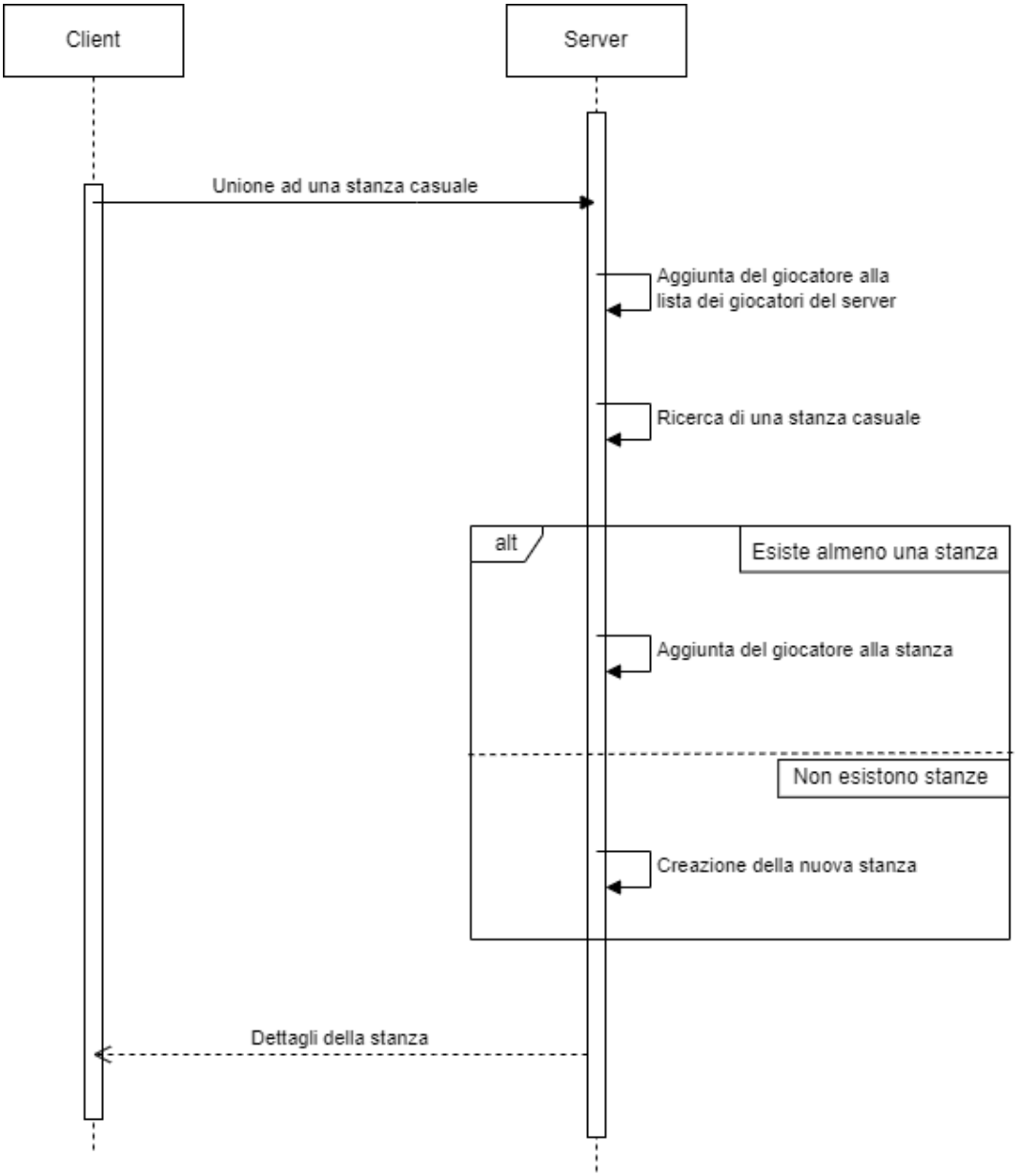

Figure 10: Sequence diagram del giocatore che vuole iniziare una partita unendosi ad una stanza casuale

In figura 11 sono presenti le interazioni tra i client e server durante i due turni consecutivi di capo e agenti. Il capo non terminerà il proprio turno finchè non verrà inserito in modo corretto l'indizio da dare agli agenti, mentre questi, sulla base della tessera scelta potranno: far vincere il proprio team, fa passare il turno o far vincere gli avversari.

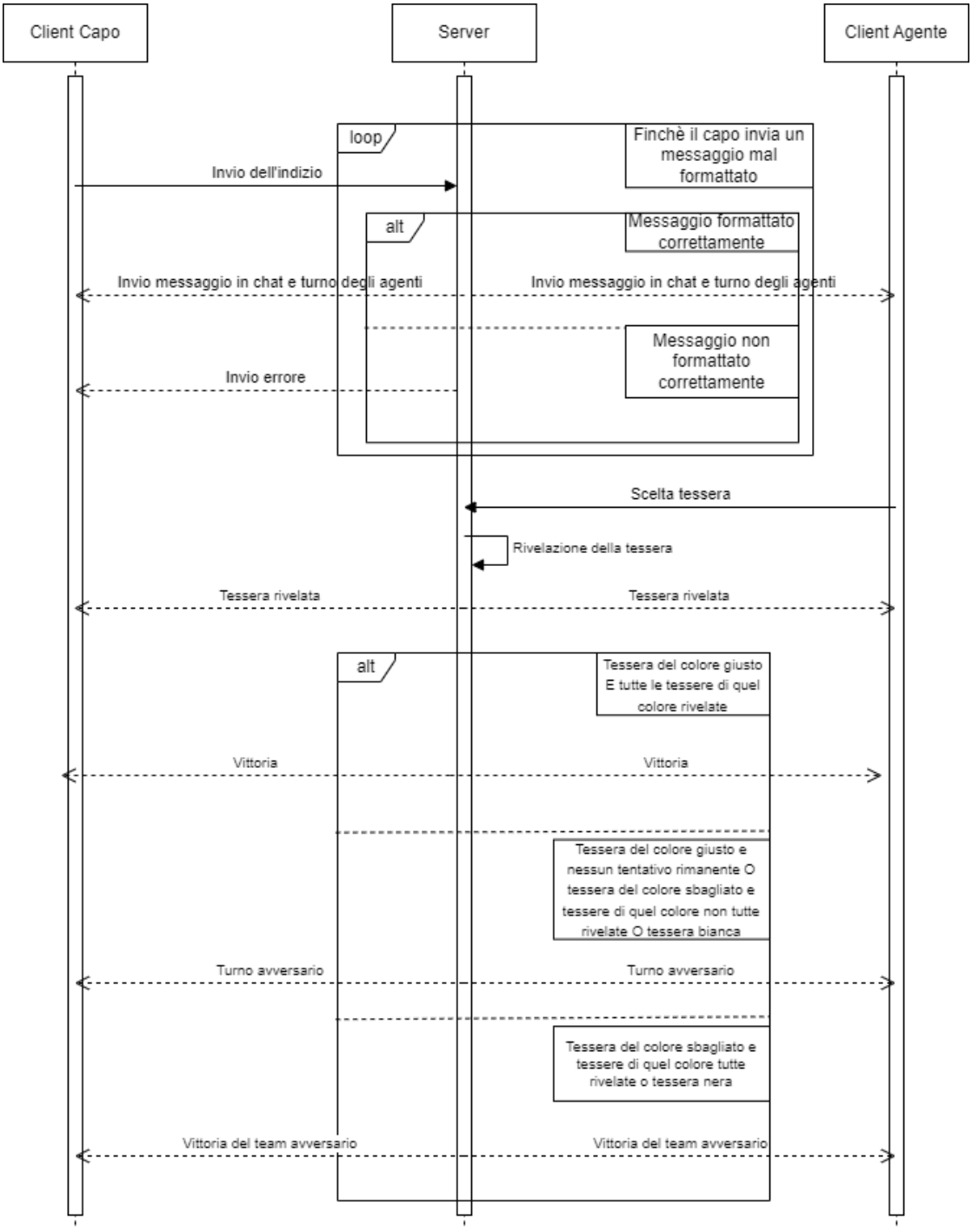

Figure 11: Sequence diagram di una squadra durante la partita

4.6 Fault Tolerance Management

Uno degli obiettivi principali del progetto è avere un sistema che sia tollerante ai guasti dovuti a errori nella rete, in modo tale che i giocatori possano giocare divertendosi.

Come prima cosa sono stati identificati i punti in cui possono verificarsi tali errori durante una normale interazione tra client-server:

1. INVIO DELLA RICHIESTA entra in errore: la rete non è riuscita a recapitare il messaggio.
2. CONSEGNA DELLA RICHIESTA entra in errore: il messaggio arriva correttamente al server, ma questo si arresta in modo anomalo subito dopo aver ricevuto il messaggio.
3. CONVALIDA DELLA RICHIESTA entra in errore: il server decide che il messaggio non è valido.
4. AGGIORNAMENTO DELLO STATO DEL SERVER entra in errore: il server non riesce ad aggiornarsi.
5. INVIO DELLA RISPOSTA entra in errore: il server potrebbe non riuscire a inviare la risposta, indipendentemente dal fatto che stia cercando di rispondere con esito positivo o negativo.
6. CONSEGNA DELLA RISPOSTA entra in errore: il recapito del messaggio al client potrebbe non aver successo, anche se la rete funzionava in una fase precedente.
7. CONVALIDA DELLA RISPOSTA entra in errore: il client decide che il messaggio ricevuto non è valido.
8. AGGIORNAMENTO DELLO STATO DEL CLIENT entra in errore: anche se il messaggio è stato ricevuto, il client riesce ad aggiornare il proprio stato, non riesce a capire il messaggio o non riesce a completare l'azione per qualche altro motivo.

Per ogni azione sul client che comporta un cambiamento di stato del gioco, è necessario gestire questi errori.

Di seguito sono elencati i tipi di errore che vengono generati nel client e come essi siano gestiti:

- **Connection Failed**, non ci si è riusciti a collegare con il server: dopo un piccolo delay, si ritenta la connessione o si cambia server.
- **Connection Lost**, il server si è disconnesso in modo anomalo: se si era all'interno di una partita, si torna alla stanza principale e poi si eseguono gli stessi passaggi di *Connection Failed*.
- **Write Failed**, non si è riusciti ad inviare il messaggio (buffer pieno ad esempio): dopo un piccolo delay, si rinvia il messaggio.

- **Ack Timeout**, il messaggio non è arrivato al server o è arrivato troppo tardi o ha generato un errore: se la connessione è ancora attiva, si rinvia il messaggio allo scadere del timeout.
- **Corrupted Message**, il messaggio arrivato è corrotto: si richiede il messaggio.
- **Missing Previous Messages**, i messaggi arrivati fanno capire che ci sono dei messaggi mancanti da quel server: richiedo il rinvio di tutti quei messaggi attraverso i logical clock.

Rispetto agli errori precedenti, sono elencati quelli che nel server hanno una gestione differente:

- **Connection Failed**, non ci si è riusciti a collegare con il client: nessuna azione.
- **Connection Lost**, il client si è disconnesso in modo anomalo: il client viene rimosso dalla stanza a cui era assegnato.
- **Player Does Not Exist**, arriva un messaggio da un player che non conosco: nessuna azione.

5 Implementation Details

5.1 Establishing Connection

Tutta la parte di interfacciamento alla rete Internet viene gestito tramite l'utilizzo del modulo **Akka.TCP** che astrae dalla gestione a basso livello del canale.

Partendo dalla configurazione interna ad ogni applicazione, vengono impostati gli indirizzi e le porte dei server, che si mettono in ascolto sui loro indirizzi tramite *Bind*.

La connessione di un client ad un server avviene tramite l'invio di un messaggio *Connect* all'indirizzo remoto specificato; all'accettazione della connessione, il server istanzia un attore adibito alla gestione della connessione e invia un messaggio Connected al client. In caso di errore di connessione, viene ricevuto il messaggio *CommandFailed* contenente una descrizione dell'errore.

5.2 Message Arrival

Ogni messaggio che arriva in un client deve essere controllato al fine di prendere una decisione su di esso. Per ogni messaggio, esclusi gli Ack, una delle seguenti azione viene condotta:

- **Handle**: il messaggio può essere gestito;
- **Do Nothing**: il messaggio non deve essere gestito o verrà gestito in futuro;
- **Start Timeout**: mancano dei messaggi e, allo scadere di un certo lasso di tempo, verranno richiesti se non ricevuti.

Tale scelta viene eseguita tramite l'utilizzo dei logical clock; ogni messaggio creato dispone di un identificativo temporale incrementale che parte da 0 rispetto ai messaggi creati in precedenza. Tramite la memorizzazione interna di tutti i messaggi ricevuti e avendo a disposizione un ordinamento per i messaggi provenienti dal server, attraverso il loro logical clock, è possibile individuare sia i messaggi mancanti rispetto l'ultimo ricevuto, sia quelli che possono essere gestiti attualmente.

Su questa base all'arrivo di un messaggio si fanno due controlli iniziali:

1. che il logical clock sia direttamente successivo rispetto a quello arrivato in precedenza o che sia il primo ricevuto;
2. che il messaggio non sia un duplicato.

Se ciò risultasse vero, si memorizza il clock del nuovo messaggio e si procede a controllare che attualmente non si stia gestendo alcun messaggio. In tal caso il messaggio può essere gestito e l'azione sarà *Handle*, altrimenti si mette il messaggio in attesa e si torna *Do Nothing*.

Se però uno due controlli iniziali risultasse falso, seguono altre due condizioni:

1. che il logical clock sia maggiore rispetto a quello che si aspetta ma entro un limite massimo di accettazione;
2. che il messaggio non sia un duplicato.

Questo sta a indicare che il messaggio è non successivo a quello gestito in precedenza e perciò mancano dei messaggi. A questo punto se il sistema nota che non sta venendo gestito alcun messaggio, si andrà in *Start Timeout*, in tutti gli altri casi in *Do Nothing*.

Lo stesso procedimento avviene nel server ma vengono gestiti contemporaneamente diversi messaggi provenienti da più client, ognuno dei quali presenta un logical clock differente e indipendente dagli altri.

5.3 Message Serialization and Deserialization

Il passaggio in rete di ogni messaggio, richiede una serializzazione e, successivamente, una deserializzazione di esso. Per semplificare il procedimento di analisi di un messaggio arrivato, le informazioni che si vogliono inviare vengono racchiuse all'interno di un oggetto di tipo **AppPackage**. Questa case class presenta quattro campi:

- *message*: torna, sotto forma di stringa, le informazioni che sono state serializzate ed inserite alla creazione del messaggio;
- *messageType*: rappresenta il tipo dei dati racchiusi all'interno del messaggio per poterli deserializzare nel modo appropriato;
- *playerId*: l'id del giocatore a cui deve essere inviato il messaggio, nel caso parta dal server, o che lo ha spedito, nel caso del client;

- *logicalClock*: il numero del clock logico attuale del mittente.

Tutto ciò che può essere presente nel campo *message* è una serializzazione di una qualunque case class che estende il trait **Message**. La serializzazione di esso avviene utilizzando il metodo *write* della libreria **Lift-JSON** che converte le classi in stringhe JSON. Successivamente, l'AppPackage viene a sua volta convertito in stringa JSON e poi in *ByteString* tramite il metodo *fromString* di **Akka.util**, per poter essere inviato correttamente in rete.

Al momento della ricezione, il messaggio viene riconvertito in JSON tramite il metodo *parse* di Lift-JSON e si tenta di istanziarlo all'interno di un AppPackage utilizzando il metodo *extract[T]* della stessa libreria, dove T può essere una qualunque classe. In seguito, si esegue una ulteriore estrazione del campo message per riottenere il tipo di messaggio suggerito dal campo messageType. Per aumentarne la riusabilità, è stato implementato il metodo *extractJson[T]* per eseguire questo procedimento utilizzando i generici. Questo può risultare in un esito negativo nel caso il messaggio che si tenta di convertire non sia un Json o non sia del tipo richiesto, in tal caso assumiamo che il messaggio sia corrotto e viene notificato.

5.4 Next Message Management

Al completamento della gestione di un messaggio, si controlla se per un logical clock troppo alto sono presenti dei messaggi in coda, pronti per essere gestiti.

Sulla base dei messaggi memorizzati all'arrivo, perchè il loro clock logico era troppo elevato, si controlla se è presente un messaggio con un clock immediatamente successivo a quello gestito per ultimo. In caso di esito positivo, viene effettuata una *Handle*, altrimenti abbiamo due casi:

1. non sono presenti messaggi in attesa di essere gestiti: *Do Nothing*;
2. sono presenti altri messaggi ma nessuno è gestibile al momento: *Start Timeout*.

Come per l'arrivo di un messaggio, tale comportamento avviene anche nel server che però la gestisce più client.

5.5 Lost or Failed Connection

Come detto in precedenza, più server sono operativi contemporaneamente e, vista l'aspettativa di un alto numero di utenti, la maggior parte della computazione viene effettuata all'interno del client per non rallentare i server.

Nel caso di perdita o fallimento nella connessione, è il client che autonomamente decide a chi collegarsi seguendo i seguenti passi:

1. conoscendo l'ultimo server a cui ha tentato la connessione, la ritenta fino a due volte, con un piccolo delay tra ogni tentativo;

2. se tale passaggio fallisce, va a guardare nella lista di server e prova a connettersi con quello il cui codice è immediatamente successivo.

Questi due passaggi si ripetono finchè non riesce a connettersi, notificando in tal modo l'utente attraverso la scritta *Connected* nella schermata principale.

5.6 Changing Connection on Room Joining

Le stanze sono identificate tramite un id numerico univoco all'interno dello stesso server, ma non tra server differenti. Ciò che diversifica le varie stanze è l'associazione del loro id con il codice del server. Perciò, se un utente vuole accedere alla stanza 104 del server "a", dovrà inserire il codice "a104". Può quindi accadere ad un certo punto che un utente sia connesso al server "b" ma vuole unirsi a una stanza nel server "a". Per gestire tale cambiamento sono necessari i seguenti passaggi:

1. il client capisce che la stanza a cui l'utente vuole unirsi non è presente nel server a cui è connesso;
2. viene memorizzato internamente il messaggio per richiedere il join nella stanza richiesta;
3. viene effettuata una disconnessione con il server corrente;
4. si procede a richiedere la connessione all'altro server:
   a) la connessione ha successo: si aspetta perciò la ricezione di un nuovo identificativo per tale server e infine si invia la richiesta di join;
   b) la connessione non ha successo: il server perciò è offline o pieno; la richiesta di join viene scartata e si procede a connettersi con un altro server.

5.7 Division of Tasks

Essendo il gruppo composto solamente da due persone, la maggior parte del lavoro è stato analizzato, discusso e svolto insieme. Tutto quello che è stato svolto individualmente, è specificato di seguito.

LUCA BAZZOCCHI

Le principali parti che ho svolto individualmente in questo progetto sono state: la realizzazione della componente grafica del client, il GameData e la CI.

Ho cercato di immedesimarmi nel ruolo di giocatore, distaccandomi da quello del programmatore, per cercare di avere una grafica semplice ma efficace, che riuscisse a migliorare l'esperienza dell'utente durante le varie partite.

Sin dall'inizio del progetto ho cercato di automatizzare l'ambiente di sviluppo per semplificare l'implementazione del progetto. La Continuous Integration è stata predisposta nei branch *main* e *develop* e suddivisa in quattro stage:

1. *Client-Build*: esegue una 'sbt compile' nel progetto client per controllare che la build abbia successo;
2. *Client-Test*: esegue un 'sbt test' nel progetto client;
3. *Server-Build*: esegue una 'sbt compile' nel progetto server;
4. *Client-Test*: esegue un 'sbt test' nel progetto server.

Al fine di far funzionare correttamente la CI, viene utilizzata una docker image contenente scala e sbt. Inoltre, visto che la parte di test del client richiede scalafx per inizializzare i componenti grafici, è stato necessario configurare l'ambiente in modo tale da far partire i test in headless mode.

Il GameData è l'oggetto che contiene tutte le informazioni di una partita e viene inviato ad ogni update dal server. Ho fatto in modo che tale classe contenesse quanti più metodi possibili al fine di semplificare la gestione delle differenze tra il GameData corrente nel client e quello nuovo appena ricevuto. Tali differenze indicano quali componenti grafici verranno modificati.

GIACOMO CASADEI

Principalmente mi sono concentrato sulla gestione delle parole sulle carte, sulla parte di comunicazione tra server e client e sulla parte di model che gestisce le singole partite.

Per le parole ho cercato di avere un file contenente un significativo numero di parole che abbiano senso (molti dizionari includevano parole come 'aaaba'). All'effettivo, sono più di 60000 termini e, per evitare ulteriori complicazioni, al momento dell'ottenimento di parole casuali si controlla che abbiano una lunghezza minore di 11.

I vari messaggi generati dalle applicazioni vengono convertiti in formato JSON utilizzando il framework lift JSON (scelto rispetto ad altre opzioni per la sua semplicità di utilizzo) e poi inviati ai destinatari utilizzando il modulo TCP del framework Akka.

L'organizzazione del Model è stata già discussa precedentemente.

Tutte le parti che non sono state nominate in questo capitolo sono state studiate e sviluppate cooperativamente.

6 Self-assessment / Validation

Attraverso il framework ScalaTest ci è stato possibile valutare il codice implementato effettuando test sulle varie situazioni in cui si può trovare il software prodotto.

Sono stati creati due ambienti di esecuzione per i test: uno lato server, l'altro lato client. Nel server ci si è concentrati a valutare la parte di Model, per un totale di 28 test, mentre nel client la parte di View, con invece 37 test.

Uno dei punti focali è stato il testing riguardante i problemi che possono essere causati nell'invio e ricezione dei vari logical clocks dei client e dei server. Questi test riguardano le classi *"ReceiverLogicalClocksManager"* e *"SenderLogicalClocksManager"* che rappresentano il fulcro nella localizzazione degli errori nell'interazione client-server e riguardano messaggi duplicati, non ordinati o non ricevuti. Il receiver per ogni messaggio ricevuto capisce l'azione da intraprendere per tale messaggio (handle, nothing o start timeout), mentre il sender permette un ordinamento temporale dei vari messaggi attraverso il meccanismo di logical clock e memorizza tutti i messaggi inviati in caso sia richiesto un rinvio.

Oltre ai test fatti tramite software, essendo il progetto una trasposizione sul pc di un gioco da tavolo, sono stati anche effettuati test pratici con persone che hanno provato la parte client da casa propria mentre erano attivi due server. Gli utenti sono stati in grado di raggiungere tutte le volte la fine di una partita e, anche se un utente ha avuto un iniziale problema di connessione, è riuscito a ricollegarsi alla stanza e proseguire la partita senza problemi. Si è perciò riusciti a simulare con efficacia un vero e proprio sistema distribuito che è riuscito a soddisfare gli obiettivi imposti.

7 Deployment Instructions

Il progetto consiste di due applicazioni: una client per il giocatore, e una server per la gestione del gioco e delle varie stanze a cui l'utente può connettersi.

Una volta aver fatto la clone del progetto è possibile scegliere tra due modalità per il deploy:

- locale: server e client all'interno dello stesso pc;
- distribuita: server e client presenti in macchine e reti differenti.

In entrambi i casi il procedimento è il medesimo, ciò che cambia è la configurazione da applicare ai client e server per assicurarsi che essi possano comunicare tra loro.

CLIENT:
La configurazione del client si trova all'interno della cartella *ds-codenames-client/ resources*, sotto il nome di *application.conf*.

```
app {
   // The client first will try to connect with the server with this code
   firstConnectionServerCode = "a"

   // List of servers that the client know
   // You can change 'localhost' with an external ip if the server is not in
       the same network
   // Or you can add more servers if you need them
   servers = [
     { "a": "localhost:63222"},
     { "b": "localhost:63223"},
     { "c": "localhost:63224"}
   ]

   // Example of servers with external ip
   //servers = [ { "a": "151.61.27.16:63222"}, { "b": "151.61.27.16:63223"}, {
       "c": "151.61.27.16:63224"}]
}
```

Da qui è possibile modificare la lista di servers a cui il client può connettersi e il codice del server a cui tenterà la connessione inizialmente.

SERVER:
La configurazione del server si trova invece all'interno della cartella *ds-codenames-server/resources*, sotto lo stesso nome di *application.conf*.

```
app {
   // You can change 'localhost' with the internal ip if the server is not in
       the same network as the client
   address = "localhost"

   port = 63222

   // Use ipconfig to find your ip on Windows
   // Then to enable the connection for the clients, you need to open the port
       in your router
   // Example of server with internal ip
   //address = "192.168.1.102"
}
```

L'address del server può essere modificato solo tramite il file di configurazione, la porta invece può essere decisa qui o durante l'esecuzione dell'applicazione dando il numero della porta come argomento. Tale esempio sarà visto successivamente.

7.1 Local Configuration

Per il deployment locale è necessario controllare che la configurazione per i client e i server presenti localhost come address.

Per il client è perciò necessario che la lista di server, presenti per ogni istanza:

"codicedelserver": "localhost:numeroporta"

Dove:

- **codicedelserver** è una stringa univoca qualsiasi Ha rilevanza solo nel client;
- **numeroporta** è un intero univoco per ogni server.

Per il server è invece necessario che l'address sia localhost. La porta può essere lasciata con il valore di default.
Terminata la configurazione, per il deployment e l'esecuzione, si passa alla sezione 7.3.

7.2 Distributed Configuration

Per il deployment distribuito si fanno le seguenti assunzioni:

1. ogni server e client può essere in qualunque punto del globo;
2. ogni server ha un indirizzo ip fisso che non cambia;
3. ogni client ha memorizzato internamente tutti i server a cui si può collegare e possiede un codice univoco per ognuno di loro;

In questa tipologia di configurazione è necessario che i client conoscano gli ip esterni e le porte assegnate ai server. Per il client è perciò necessario che la lista di server, presenti per ogni istanza:

"codicedelserver": "ipesterno:numeroporta"

Dove:

- **codicedelserver** è una stringa univoca qualsiasi. Ha rilevanza solo nel client;
- **ipesterno** è l'indirizzo ip esterno del server;
- **numeroporta** è un intero univoco per ogni server.

Nella configurazione dei server deve invece essere settato l'indirizzo ip interno e la porta. Se si è in un sistema operativo Windows, l'ip interno è ottenibile eseguendo il comando ipconfig.

Avere però una configurazione differente per ogni server sta a significare che poi dovranno essere generati jar differenti per ognuno. Al fine di rimuovere e semplificare tale procedimento, è possibile lasciare la configurazione inalterata e specificare l'indirizzo ip interno e porta come argomenti all'esecuzione del server. Questo procedimento è spiegato nella sezione 7.3 sul deployment ed esecuzione.

7.3 Deployment and Execution

Finito il setting dei file di configurazione, è possibile generare gli applicativi per avviare sia i client che i server. Tornando alla directory principale del progetto, per il client è necessario eseguire i seguenti comandi:

cd ds-codenames-client
sbt assembly

Il jar contenente l'eseguibile del client si troverà dentro la cartella *target/scala-2.13* con il nome *ds-codenames-client-assembly-1.0.jar*. L'esecuzione di un client avviene tramite il comando:

java -jar target/scala-2.13/ds-codenames-client-assembly-1.0.jar

Per il server si esegue in modo similare lo stesso comando:

cd ds-codenames-server
sbt assembly

L'esecuzione di un server può avvenire in due modi:

- senza argomenti aggiuntivi: in questo caso prenderà quella che era presente nel file di configurazione prima di aver eseguito il comando *sbt assembly*. Attenzione perchè l'esecuzione di un secondo server con lo stesso address, porterà ad un errore visto che la porta è la stessa;
- specificando l'address e numero di porta: in modo semplice, senza dover creare jar diversi per ogni server in postazioni differenti, in quanto gli argomenti hanno priorità sulla configurazione.

Il server è perciò possibile avviarlo attraverso il seguente comando:

java -jar target/scala-2.13/ds-codenames-server-assembly-1.0.jar *address* *porta*

Dove **address** e **porta** sono argomenti aggiuntivi che possono o meno essere presenti. Sia nel deployment in locale che distribuito, se tali argomenti vengono utilizzati, essi dovranno rispecchiare la configurazione dei server nell'applicazione client.

È importante ricordarsi che l'address da inserire per un server **non** è il suo address esterno (quello nella configurazione del client), ma l'indirizzo interno; ottenuto ad esempio attraverso il comando ipconfig in Windows.

Esempio in locale:
java -jar *percorsojar* localhost 63222
Esempio nel distribuito:
java -jar *percorsojar* 192.168.1.102 63222

Infine, se il deployment è distribuito, è sempre importante ricordarsi di tenere **aperte** le porte del server per permetterne la comunicazione con il client (se si ha ad esempio un router, bisogna fare un port forwarding da quella porta all'indirizzo ip interno del server in questione).

8 Game Guide

All'apertura dell'applicazione client, l'utente si trova davanti alla schermata iniziale del gioco 12.

Una volta aver inserito il proprio nome utente, è possibile:

- creare una nuova stanza;
- entrare in una stanza scelta a caso;
- entrare in una stanza tramite il suo codice.

Come mostrato in figura, l'applicazione informa l'utente dello stato della connessione con il server e eventuali messaggi di errori. Tra questi sono presenti:

- campi non riempiti;
- la richiesta di entrare in una stanza non esistente;
- connessione persa con il server.

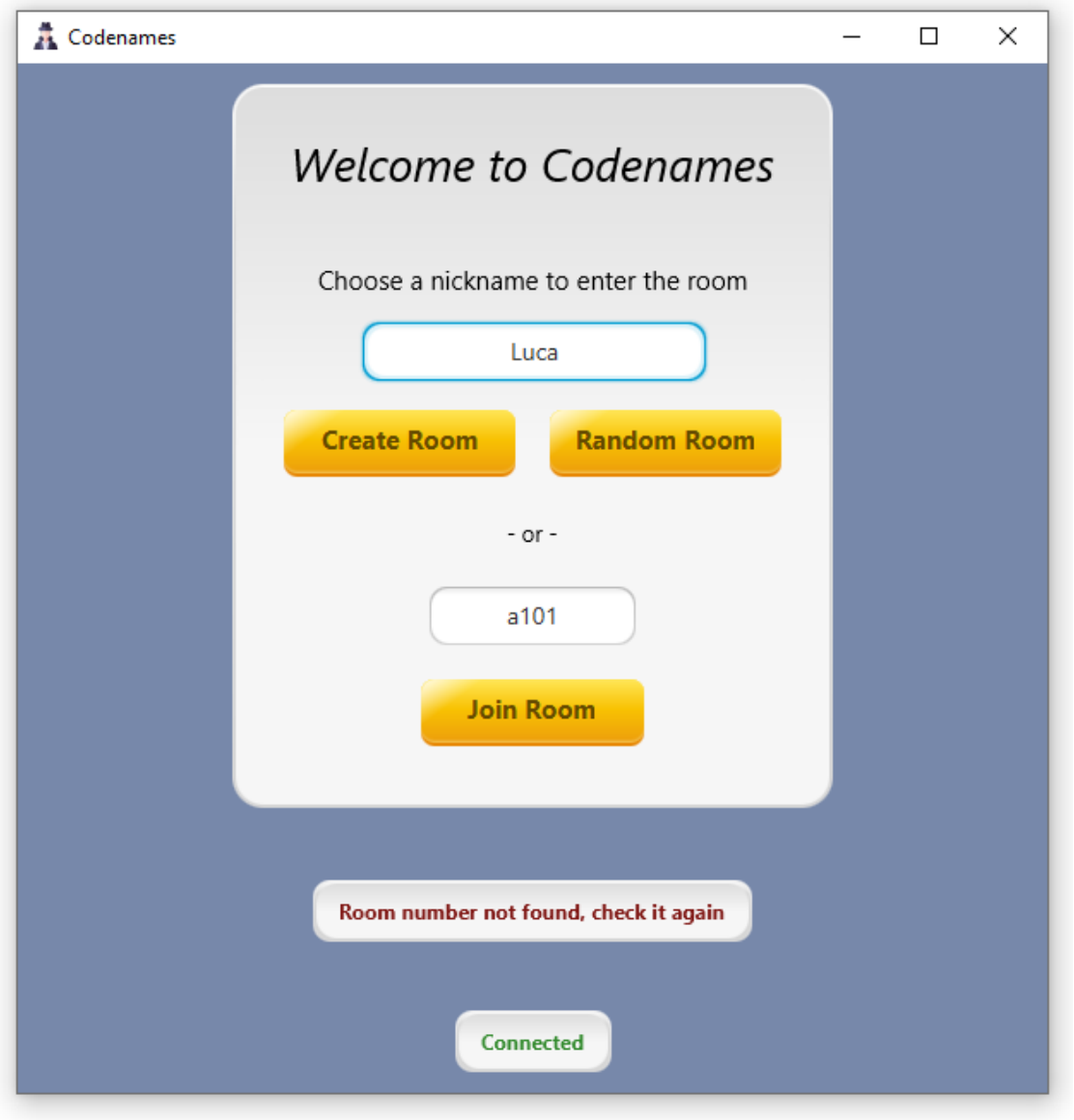

Figure 12: Homepage

È possibile entrare in una stanza in qualsiasi momento della partita. Ogni squadra è formata da un capo e un massimo di tre agenti; se almeno uno di questi ruoli è ancora disponibile, dalla sezione più a sinistra 13, è possibile scegliere il proprio ruolo all'interno della squadra. Una volta iniziata una partita, non sarà poi possibile cambiare il proprio ruolo fino al termine di essa.

Quando le due squadre avranno presenti un capo e almeno un agente, i capi possono decidere di iniziare una partita tramite l'apposito pulsante *"Start Game"*

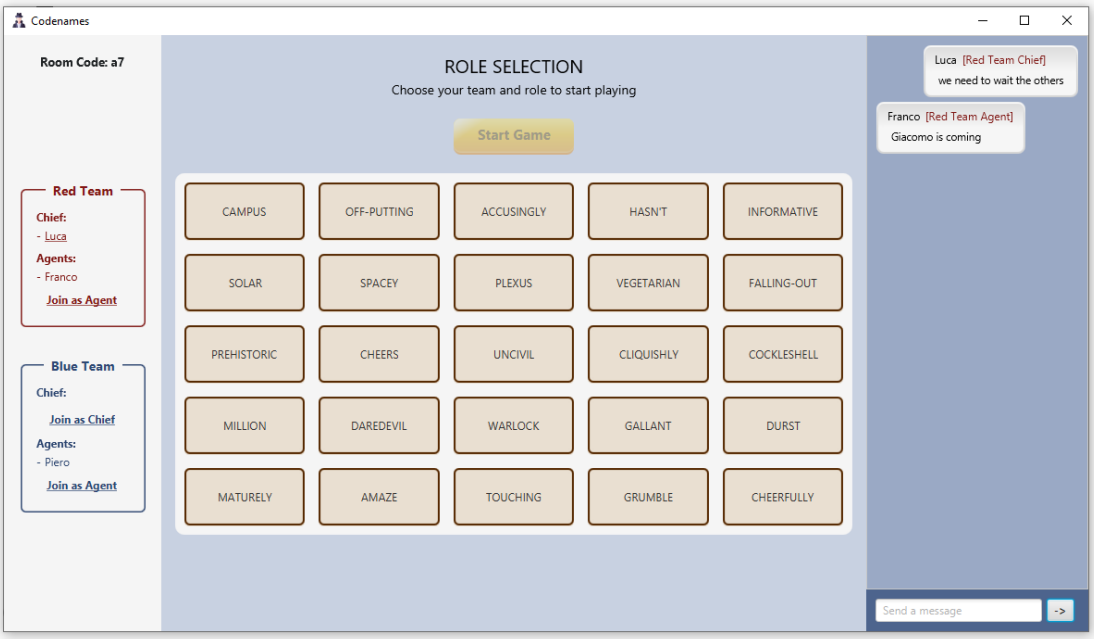

Figure 13: Selezione dei ruoli durante un game

Lo scopo del gioco è far indovinare alla propria squadra le tessere del proprio colore rivelandole e raccogliendo informazioni attraverso la chat. Com'è possibile vedere in figura 14, in grafica sono disposte due griglie:

- gameboard: visualizzabile da tutti e interagibile solo dagli agenti durante il loro turno. Presenta un totale di 25 tessere contenenti delle parole che rappresentano i nomi in codice di agenti delle due squadre. Quelle blu indicano gli agenti blu, quelle rosse gli agenti rossi, quelle bianche dei passanti e la nera un letale assassino;
- chiefboard: visualizzabile solo dai capi; indica la disposizione degli agenti delle due squadre, in bianco i passanti e in nero l'assassino. I bordi colorati indicano il colore della squadra di partenza, in questo caso la rossa.

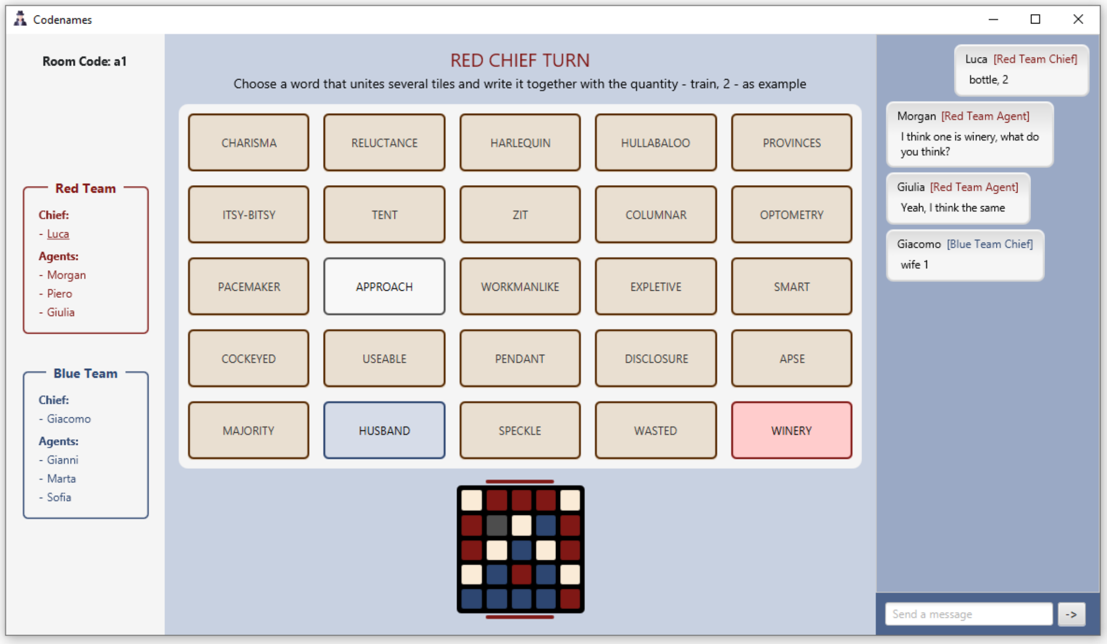

Figure 14: Partita in corso di Codenames

I turni di gioco si svolgono in ordine partendo dal capo della squadra estratta per prima, per poi passare ai suoi agenti. Successivamente è il turno del capo dell'altra squadra e infine dei suoi agenti. Questo ciclo si ripete fino alla conclusione del gioco.

I turni possibili sono perciò i seguenti:

- **Red Chief Turn**: il capo della squadra rossa, guardando la propria griglia colorata in basso, cercherà di far indovinare ai propri agenti le tessere rosse. Egli darà un solo indizio attraverso la chat, che raggruppi più parole possibili, seguito del numero di tessere che fanno parte di questo gruppo (bottle, 2 ad esempio);
- **Red Agents Turn**: gli agenti della squadra rossa, sulla base dell'indizio, cercheranno di indovinare quante più parole possibili, selezionandole. Se viene rivelata una tessera della squadra avversaria o una tessera bianca, il turno passa direttamente agli altri, altrimenti, se la tessera è nera, gli avversari vincono la partita automaticamente;
- **Blue Chief Turn**: come il turno del capo rosso;
- **Red Agents Turn**: come il turno degli agenti rossi.

Per avere un gioco equo, ogni giocatore può scrivere solamente durante il proprio turno. Inoltre, nel caso non si sappia cosa fare durante il proprio turno, in alto è sempre presente una info sullo stato attuale e sulle azioni possibili.

Durante la partita è possibile che alcuni giocatori si disconnettano, rendendo i team troppo piccoli per proseguire. In tal caso si passa ad uno stato chiamato "*Waiting Players*" in cui si attende l'ingresso di ulteriori utenti prima di riprendere la partita dal momento in cui la si era lasciata.

9 Conclusions

Il progetto ci ha permesso di affrontare in maniera diretta i principali aspetti di gestione, progettazione e sviluppo di un sistema software non banale e di addentrarsi in quello che è il mondo della programmazione distribuita.

Oltre alle metodologie utilizzate, abbiamo avuto modo di applicarci utilizzando il linguaggio Scala, il framework Akka e la gestione degli errori in un sistema distribuito, oltre alle tecnologie già citate in precedenza.

In generale ci riteniamo soddisfatti del risultato finale del progetto.

9.1 Future Works

Una possibile espansione di questo progetto consisterebbe nello scambio di messaggi tra i vari server con lo scopo di condividere tra loro le informazioni relative al loro stato e le stanze che stanno gestendo. In questo modo, sarebbe possibile effettuare il recovery di un server arrestatosi in modo anomalo spostando le sue stanze su un altro server, cosa che al momento non è stata reputata critica per questo progetto.

Un'ulteriore espansione sarebbe composta dall'aggiunta di un'ulteriore entità posta tra i client e i server, al fine di sostituire la configurazione statica dei client riguardante l'indirizzo dei server. Questa entità comunica con i vari server per mantenere una lista di server attivi e ad una richiesta di connessione da parte d un client, restituirà il server più opportuno.

9.2 What did we learned

Per la prima volta abbiamo avuto la possibilità di lavorare su un sistema distribuito, e sia da quest'esperienza che dai problemi riscontrati, abbiamo imparato a gestire gli errori emersi durante uno scambio di messaggi in rete tra più entità e come questi messaggi debbano essere formattati per poter essere trasmessi tramite TCP.

References

[1] J. Boner. Akka documentation, 2011.

[2] Jacob Gabrielson. Challenges with distributed systems, 2019.

[3] Lift. Lift-json, 2011.

[4] Sven Reimers Stephen Chin. Scalafx, 2014.

[5] Wikipedia. Codenames (board game), 2015.