ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

Elaborato di progetto per l'esame di
Paradigmi di Programmazione e Sviluppo

Smart-Chat

Giacomo Scaparrotti
giacomo.scaparrotti@studio.unibo.it

Massimiliano Giunchi
massimiliano.giunchi@studio.unibo.it

A.A. 2017/2018

Indice

Capitolo 1

Processo di Sviluppo

1.1 Metodologia di sviluppo

Relativamente al processo di sviluppo, si è deciso di adottare e testare una metodologia *Agile Scrum-like* dove i membri del team risultano allo stesso livello: non esiste né uno *Scrum Leader*, né uno *Scrum Manager*; per ciò che concerne lo scheduling delle attività abbiamo impiegato Trello, mentre per quel che riguarda la cadenza degli sprint, questi sono avvenuti con cadenza settimanale.

All'inizio del processo di sviluppo abbiamo delineato il Product Backlog, contenente gli items, con relativa priorità e stima del tempo; successivamente, è stata creata una modellazione del sistema. Con cadenza settimanale si sono eseguiti degli incontri (sprint review) volti a verificare se i compiti assegnati fossero stati portati a termine, nonché risolvere eventuali problemi emersi durante lo sviluppo della feature e decidere quelle da implementare successivamente, le quali sono state scelte sulla base delle priorità delineate.

La metodologia di sviluppo adottata si basa sul modello *Test Driven Development* il quale prevede che la stesura dei test avvenga prima dell'implementazione e che lo sviluppo sia orientato esclusivamente all'obiettivo superare i test predisposti. In particolare, esso prevede un ciclo di sviluppo di tre fasi: nella prima si scrive un test per la nuova funzione da sviluppare, nella seconda si sviluppa il codice necessario per passare il test ed infine si esegue il refactoring del codice per adeguarlo agli standard di qualità.

Per quel che riguarda la gestione dei repository, è stato utilizzato il repository di uno dei due membri del gruppo (Giacomo) come repository “verità”, contenente quindi la versione considerabile “ufficiale” del software. Di conseguenza, ogni volta che l’altro membro del gruppo (Massimiliano) effettuava delle modifiche al proprio repository, doveva aprire una nuova Pull Request in modo che queste potessero essere integrate in mainline.

1.2 Strumenti adottati

I principali strumenti utilizzati durante l'intero processo di sviluppo sono i seguenti:

- **Git**: quale sistema di versioning distribuito;
- **GitHub**: servizio di hosting per il version control e per il repository;
- **Travis CI**: esso è stato impiegato per la continuous integration, in quanto permette di verificare che ad ogni nuova modifica del repository il progetto continui a compilare e che i test siano eseguiti con successo;
- **Trello**: permette di tracciare le fasi di sviluppo.

Capitolo 2

Requisiti

L'obiettivo del progetto è quello di realizzare un'applicazione di chat; deve essere possibile creare più chat attive contemporaneamente, sia individuali che di gruppo. Gli utenti, una volta registrati, potranno aggiungersi e rimuoversi dalle chat in maniera dinamica, ed inviare e ricevere messaggi di testo o allegati quali immagini; qualora un utente riceva un messaggio in una chat differente rispetto a quella selezionata, comparirà una notifica. Infine, ciascun utente visualizzerà i messaggi nell'ordine in cui questi sono ricevuti dal server.

2.1 Requisiti di business e funzionali

Dall'analisi del problema sono emersi i seguenti requisiti:

- possibilità da parte di un utente di creare chat individuali con più utenti;
- possibilità da parte di un utente di create chat di gruppo;
- possibilità da parte di un utente di aggiungersi ad una chat di gruppo;
- notifica della presenza di un messaggio/allegato non letto in una chat (sia individuale che di gruppo)
- visualizzazione dei messaggi nell'ordine in cui sono ricevuti dal chat-server
- possibilità di inviare immagini come allegati e di mostrarne un'anteprima in linea con il testo

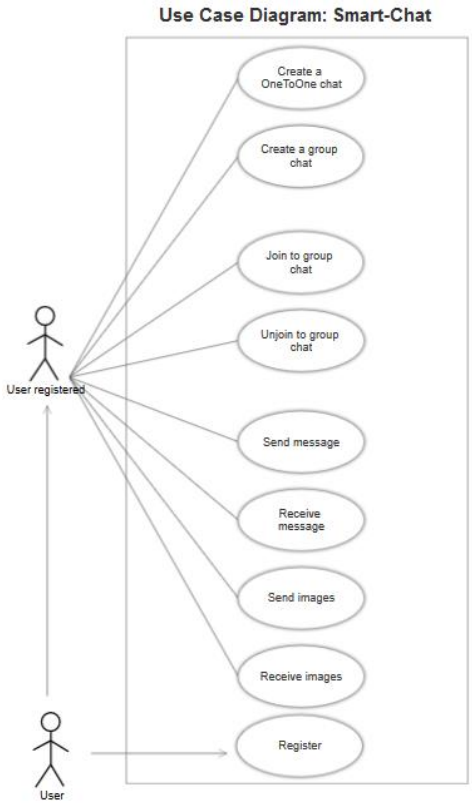


Figura 1: diagramma dei casi d'uso

2.1.1 Registrazione e deregistrazione dell'utente

Per poter accedere al sistema, l'utente deve registrarsi, ossia deve indicare un nome attraverso cui sia univocamente identificabile. Quando un nuovo utente entra a far parte del sistema, tutti gli altri utenti devono venirne a conoscenza, in modo da poter eventualmente richiedere la creazione di una nuova chat con il nuovo utente. Inoltre, quando un utente abbandona il sistema, ad esempio perché il programma di chat viene chiuso, gli altri utenti devono essere informati, in modo da non poter più creare una chat lo user che si è deregistrato.

2.1.2 Creazione di una chat singola

Un utente registrato può creare una chat con un altro utente: una volta creata sarà possibile inviare al destinatario messaggi o allegati, e quest'ultimo potrà naturalmente fare altrettanto.

2.1.3 Creazione di una chat di gruppo

Un utente registrato può creare una chat di gruppo: all'atto della creazione, esso non contiene alcun membro. Tutti gli utenti possono vedere un nuovo gruppo e chiedere di aderirvi.

2.1.4 Join / Unjoin ad una chat di gruppo

Ogni utente può decidere di unirsi o eliminarsi da una chat di gruppo: nel momento in cui egli accede, visualizzerà da lì in poi i messaggi degli utenti partecipanti (e non quelli precedentemente scambiati).

2.1.5 Invio di messaggi di testo e di immagini

Un utente che ha creato una chat individuale o di gruppo, può inviare messaggi o immagini. Di queste ultime deve essere visualizzata un'anteprima in linea con il testo (senza quindi la necessità di aprire l'immagine all'esterno del programma di chat).

2.1.6 Ricezione in ordine dei messaggi

Tutti i messaggi inviati (sia all'interno di chat individuali che all'interno di chat di gruppo) devono essere mostrati a tutti i partecipanti nello stesso ordine, a prescindere quindi dall'ordine con cui i messaggi sono effettivamente ricevuti da ciascun client.

2.2 Requisiti di implementazione

Per lo sviluppo del progetto sono stati definiti i seguenti requisiti implementativi:

- **Scala**: il linguaggio impiegato per lo sviluppo dell'intero progetto è Scala; tale requisito è stato imposto allo scopo di poter sfruttare i vantaggi offerti dal paradigma di programmazione funzionale, il quale ben si sposa con l'altra scelta tecnologica che è stata effettuata, ossia l'utilizzo del framework ad attori Akka, il quale, essendo realizzato in Scala, permette di ottenere i massimi benefici in termini di compattezza e sinteticità, nonché, soprattutto, di espressività.

- **Akka**: tale framework è stato impiegato per gestire il modello ad attori: esso fornisce un livello di astrazione tale per cui lo sviluppatore non deve preoccuparsi di gestire meccanismi di lock o gestione dei thread rendendo di fatto più semplice la scrittura di sistemi concorrenti: tutte le interazioni tra i componenti software avvengono infatti tramite scambio di messaggi. Quest'ultimo aspetto, nel caso di un'applicazione di chat, è utile sia nel realizzare i meccanismi di interazioni tra componenti in esecuzione sulla stessa macchina (sono virtualmente impossibili corse critiche), sia per quanto riguarda l'interazione tra componenti in esecuzione su macchine diverse, o su processi diversi in una stessa macchina (Akka integra un sistema di *remoting*, ossia di invio dei messaggi tramite rete, che solleva il programmatore dall'onere di dover gestire manualmente questo aspetto).

- **JavaFX**: tale libreria è stata impiegata per implementare la GUI del client. Si è scelto di utilizzare questa libreria, in luogo di altre possibili (come ad esempio Swing) per la possibilità di definire semplicemente interfacce grafiche complesse, potendo comporre le varie schermata in modo visuale con degli appositi tools, cosa possibile solo in maniera parziale con altre soluzioni software. In un primo momento è stata anche considerata la possibilità di usare Swing, poi scartato a causa della sua verbosità (boilerplate).

Capitolo 3

Design architetturale

3.1 Pattern architetturali

L'architettura impiegata è del tipo Client-Server. In particolare, all'interno del sistema esistono tre tipi di server. Innanzitutto, all'interno del sistema è sempre presente uno e un solo server che funge da registro (d'ora in poi verrà chiamato RegisterServer), la cui funzione è quella di tener traccia di tutti gli utenti che in ogni momento fanno parte del sistema e di tutte le chat esistenti. Esistono poi i server che si occupano di gestire i messaggi delle varie chat; in particolare, esiste un tipo di server che si occupa di gestire le chat singole (d'ora in poi verrà chiamato ChatServer) e un tipo di server che si occupa di gestire le chat di gruppo (d'ora in poi verrà chiamato GroupServer).

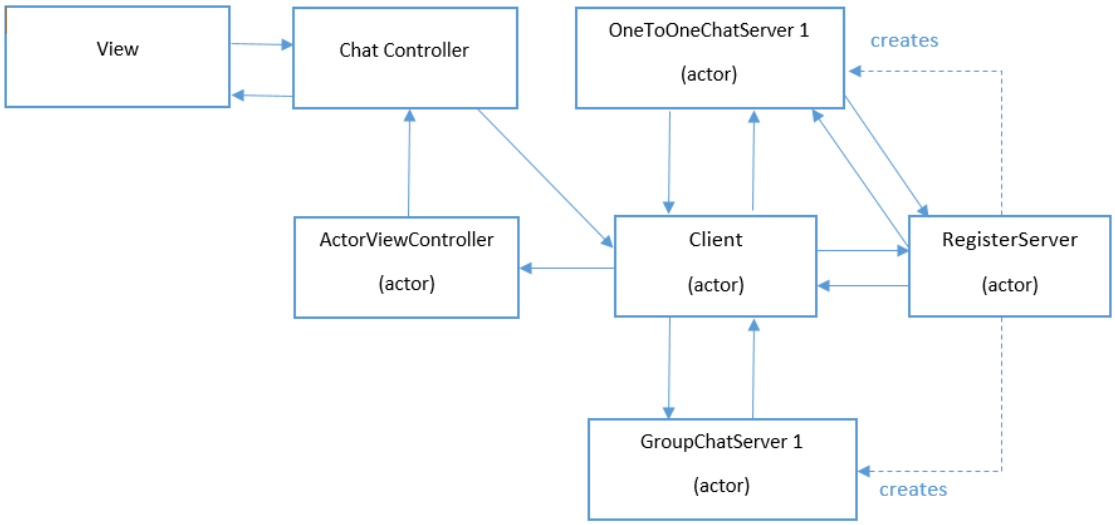
Il sistema è inoltre realizzato secondo il paradigma ad attori: sia i client che i server sono degli attori, che comunicano quindi tramite scambio di messaggi. Come anticipato, questa scelta è stata fatta principalmente per i seguenti motivi: esso fornisce un livello di astrazione tale per cui lo sviluppatore non deve preoccuparsi di gestire meccanismi di lock o gestione dei thread rendendo di fatto più semplice la scrittura di sistemi concorrenti: tutte le interazioni tra i componenti software avvengono infatti tramite scambio di messaggi. Quest'ultimo aspetto, nel caso di un'applicazione di chat, è utile sia nel realizzare i meccanismi di interazioni tra componenti in esecuzione sulla stessa macchina (sono virtualmente impossibili corse critiche), sia per quanto riguarda l'interazione tra componenti in esecuzione su macchine diverse, o su processi diversi in una stessa macchina (Akka integra un sistema di *remoting*, ossia di invio dei messaggi tramite rete, che solleva il programmatore dall'onere di dover gestire manualmente questo aspetto).

Poiché non tutti i componenti del sistema sono realizzabili unicamente tramite attori, con particolare riferimento alla GUI, i client sono realizzati seguendo il paradigma MVC, in questo caso integrato, appunto, con quello ad attori. Ne consegue quindi che non solo il controller, che è la parte del client che comunica con i server, è un attore, ma anche una parte della view, che si occuperà di comunicare con il client e di aggiornare la GUI con gli appositi metodi messi a disposizione da JavaFX.

3.2 Componenti del sistema

Di seguito sarà eseguita un'analisi dei principali componenti del sistema illustrando per ciascuno le maggiori feature architetturali.

Prima di addentrarci nella descrizione di ogni singolo componente, può essere utile fornire uno schema di insieme che permetta di comprendere le relazioni che sussistono tra di essi:



Dallo schema si possono elencare i seguenti componenti:

- *RegisterServer:* si tratta di un attore che coordina la creazione e la rimozione dei Client, delle chat singole e di gruppo; esso funge anche da registro, ossia contiene i riferimenti di tutti i chat server attivi;
- *Client*: è un attore che interagisce con tutti gli altri componenti; esso è creato ogni qual volta un utente esegue il login attraverso l'interazione con il RegisterServer che deve autorizzare la registrazione. Comunica con la view attraverso ActorViewController, e riceve i messaggi da questa per via del ChatController; interagisce quindi con tutti i chat server cui l'utente desideri dialogare;
- *ActorViewController*: riceve i messaggi dal Client e provvede ad invocare i metodi del ChatController che agiscono sulla view;
- *ChatController:* a seguito dell'interazione dell'utente con la view, sono invocati dei metodi che scaturiscono nell'invio di messaggi verso il client (es. invio di un messaggio, creazione di una chat singola o di gruppo, deregistrazione del Client).
- *OneToOneChatServer*: attore creato dal RegisterServer a seguito della richiesta di un Client di creare una chat singola; esso per ogni messaggio inviato da uno dei due Client partecipanti, provvede a rinviarli ad entrambi (successivamente saranno poi inviati al ActorViewController);
- *GroupChatServer*: attore creato dal RegisterServer a seguito della richiesta di un Client di creare una chat di gruppo; esso per ogni messaggio inviato da uno dei Client partecipanti, provvede a rinviarli a tutti i membri del gruppo (successivamente saranno poi inviati al ActorViewController).

Merita una menzione il modo in cui sono stati integrati i paradigmi MVC e ad attori. Dal momento che, chiaramente, un attore può inviare un messaggio solamente ad un altro attore, è stato realizzato un attore che si occupa esclusivamente di fare da "ponte" tra il client, che è un attore, e la relativa view: ogni volta che il client vuole mostrare qualcosa sulla view lo fa mandando un messaggio a questo apposito attore, che si occupa poi di invocare i metodi appropriati sulla view, che è invece un "normale" oggetto. Al contrario, quando dalla view vengono impartiti dei comandi che devono essere passati al client, non

è necessario passare attraverso l'attore "ponte", in quanto non è necessario che chi manda un messaggio sia un attore, perciò la view può inviare direttamente dei messaggi al client.

Nota: nelle sezioni successive i messaggi sono riportati con i nomi che sono stati loro attribuiti in fase di implementazione, in modo da poter facilmente far riferimento al sorgente durante la lettura.

3.2.1 Client

Il client è la parte del sistema con cui interagiscono gli utenti. Il client si compone sostanzialmente di due controller a cui corrispondono due view (in questo caso, per view si intende una particolare *scene* di JavaFX, a cui viene legato il relativo controller).

Al momento dell'avvio dell'applicazione, il client si occupa di registrarsi presso il RegisterServer, di cui conosce l'indirizzo, mandandogli un apposito messaggio contenente il nome scelto dall'utente tramite la GUI. Questa prima fase è gestita da un'apposita view con il relativo controller (LoginController).

Una volta andata a buon fine la fase di registrazione, si passa ad una nuova view che rappresenta la schermata principale dell'applicazione, da cui è possibile interagire con utenti e gruppi e inviare messaggi.

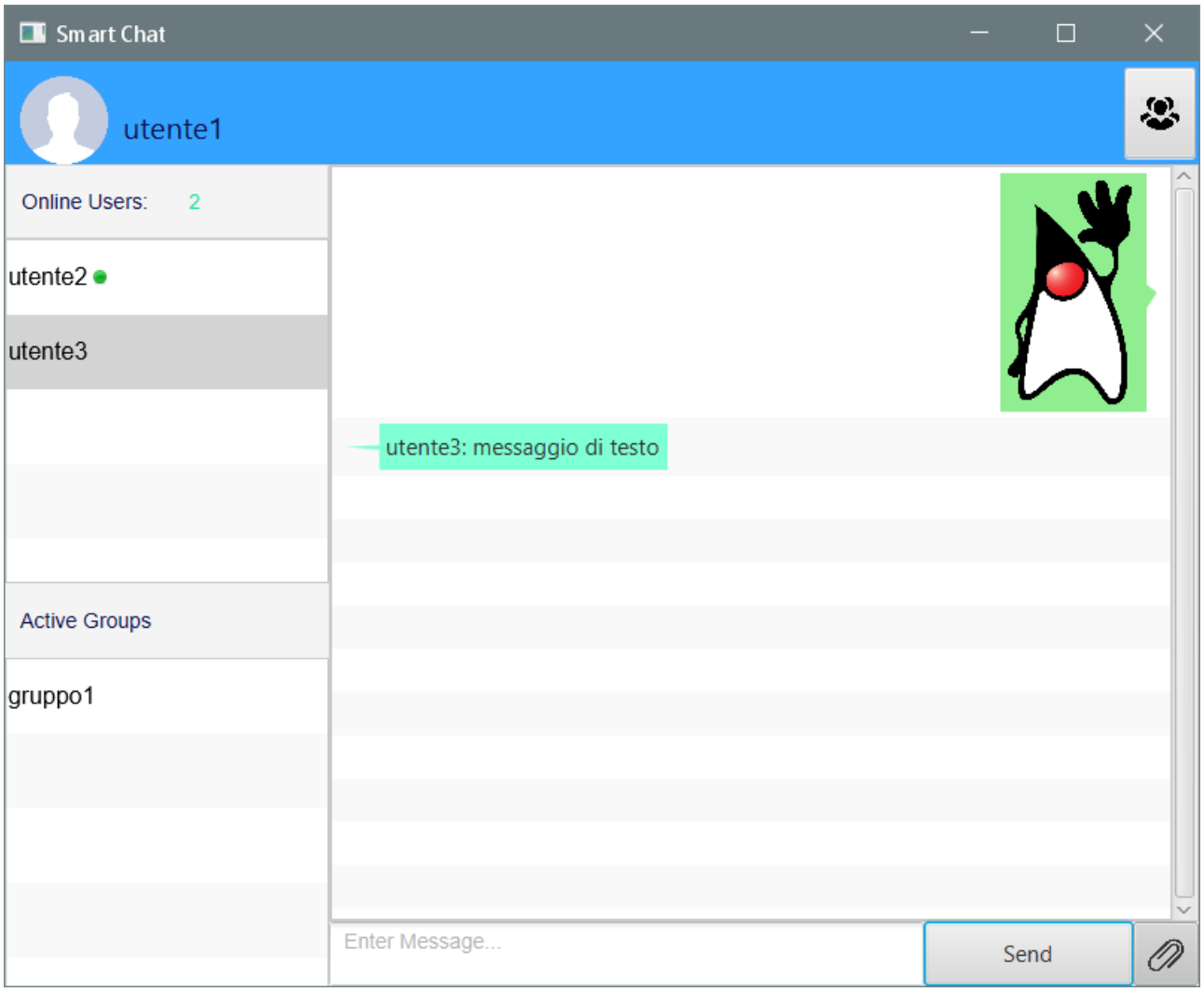


Figura 2 - Schermata principale dell'applicazione. Nella parte destra sono elencati gli utenti attivi e i gruppi esistenti. In particolare, nell'elenco degli utenti si può notare che l'utente con cui si sta chattando al momento è "utente3" e che c'è un messaggio non letto proveniente da "utente2", identificato dall'icona verde di fianco al nome. Si noti inoltre come i messaggi di testo e gli allegati siano mostrati nello stesso modo all'interno della schermata della chat attiva.

Un client deve essere in grado di ricevere i seguenti messaggi:

- Dal RegisterServer:
  - `AcceptRegistrationFromRegister`: esito richiesta di registrazione di un nuovo utente (è negativo qualora ci sia un utente registrato avente il medesimo username inviato);
  - `ResponseForChatCreation`: esito richiesta creazione chat singola o di gruppo;
  - `UserAndGroupActive`: elenco degli utenti e dei gruppi attivi;
  - `ResponseForServerRefRequest`: restituisce il riferimento del server che gestisce la chat (sia singola che di gruppo: risulta negativo qualora l'utente richiedente non sia registrato, indipendentemente dall'esistenza o meno della chat).
- Dal LoginController (l'attore che gestisce l'interazione con la GUI durante il login):

    - `SetActorLogin`: messaggio con cui indicare qual è l'attore che si occupa di gestire la view durante il login;
    - `LoginFromConsole`: messaggio contenente la richiesta di registrarsi presso il RegisterServer con un certo username.
- Dal ChatController (l'attore che gestisce l'interazione con la GUI principale):
    - `SetActorView`: messaggio con cui indicare qual è l'attore che si occupa di gestire la view durante il normale funzionamento dell'applicazione;
    - `RequestForChatCreationFromConsole`: richiesta di creazione di una chat uno-a-uno da inoltrare al RegisterServer;
    - `CreateGroupRequestFromConsole`: richiesta di creazione di una chat di gruppo da inoltrare al registro;
    - `JoinGroupRequestFromConsole`: richiesta di unirsi ad una chat di gruppo da inoltrare al GroupServer che gestisce la particolare chat di gruppo;
    - `UnJoinGroupRequestFromConsole`: richiesta di non far più parte di una chat di gruppo da inoltrare al GroupServer che gestisce la particolare chat di gruppo;
    - `StringMessageFromConsole`: messaggio testuale proveniente dalla view e da inoltrare al server che gestisce la particolare chat;
    - `AttachmentMessageFromConsole`: immagine proveniente dalla view e da inoltrare al server che gestisce la particolare chat;
    - `StopRequest`: richiesta di deregistrazione dell'utente da inoltrare al RegisterServer.
- Da un ChatServer:
    - `StringMessageFromServer`: messaggio di testo proveniente da un ChatServer;
    - `AttachmentMessageFromServer`: immagine proveniente da un ChatServer.
- Da un GroupServer:
    - `StringMessageFromGroupServer`: messaggio di testo proveniente da un GroupServer;
    - `AttachmentMessageFromGroupServer`: immagine proveniente da un GroupServer;
    - `ResponseForJoinGroupRequest`: esito richiesta di poter unirsi ad una chat di gruppo;
    - `ResponseForUnJoinGroupRequest`: esito richiesta di non partecipare ulteriormente ad una chat di gruppo.

Anche senza approfondire la funzione dei singoli messaggi, è evidente che il client deve essere in grado di gestirne un grande numero. Questo è dovuto a diversi fattori: innanzitutto, il client si trova a dover comunicare con tre diversi tipi di server (RegisterServer, ChatServer e GroupServer), e questo significa evidentemente che deve poter ricevere tutti i messaggi di cui questi hanno bisogno; in secondo luogo, il client (inteso come attore) deve poter interagire con la relativa view, da cui vengono impartiti tutti i comandi che andranno poi recapitati ai vari server che si occupano di ciascun compito.

3.2.2 RegisterServer

Il RegisterServer è il server che si occupa di gestire gli utenti registrati e le chat attive, sia singole che di gruppo. Esso deve perciò essere in grado di ricevere i seguenti messaggi:

- Da un client:
  - `JoinRequest`: richiesta da parte di un client di unirsi al RegisterServer
  - `Unjoin`: richiesta da parte di un client di essere rimosso dal RegisterServer
  - `AllUsersAndGroupsRequest`: richiesta dei nomi di tutti gli utenti e di tutti i gruppi registrati
  - `NewOneToOneChatRequest`: richiesta di creare una nuova chat uno-a-uno
  - `NewGroupChatRequest`: richiesta di creare una nuova chat di gruppo
  - `GetServerRef`: richiesta di ottenere il riferimento al ChatServer relativo alla chat uno-a-uno tra il mittente del messaggio e il client indicato
  - `GetGroupServerRef`: richiesta di ottenere il riferimento al ChatServer relativo alla chat di gruppo avente il nome indicato
  - `ContainsMembers`: risposta inviata da un ChatServer o da un GroupServer relativa alla richiesta riguardante la presenza di un particolare insieme di utenti all'interno di uno dei suddetti server

Come è possibile notare, i messaggi che possono essere ricevuti da un RegisterServer sono tutti relativi a richieste che il server dovrà poi evadere. Questo è pienamente in linea con il paradigma di interazione Client-Server, in cui il server ha il compito di rispondere alle richieste provenienti dai client, e non dà autonomamente inizio alla loro interazione.

Si nota inoltre che il genere di richieste che vengono effettuate al RegisterServer non sono relative alla fase di svolgimento delle chat, ma piuttosto sono relative alla loro gestione (creazione, rimozione, ottenimento delle informazioni sulle chat attive, ecc…).

Come primo diagramma di sequenza mostriamo il processo di registrazione di un utente: in questa fase l'attore che si occupa di intercettare le azioni dell'utente è ActorLoginController, il quale attende la risposta del RegisterServer (sempre per il tramite del Client).

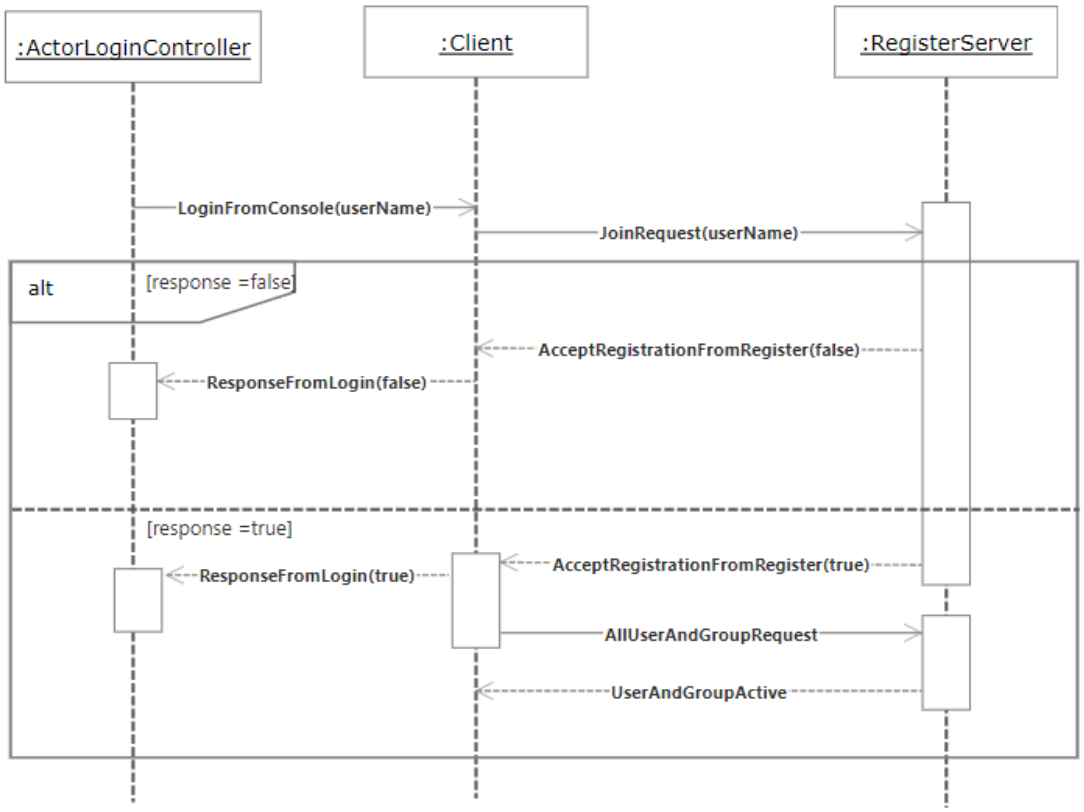


Figura 3 - processo di registrazione di un utente

Il seguente diagramma di sequenza mostra il processo di interazione tra client e RegisterServer al momento della creazione di una nuova chat:

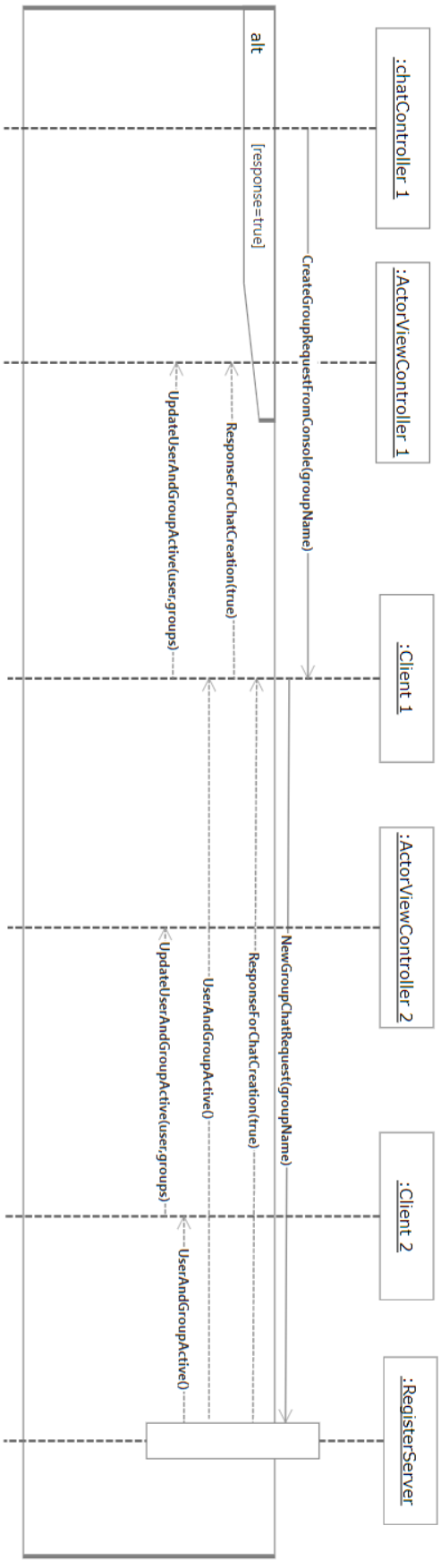


Figura 4 - creazione di una nuova chat uno-a-uno tra due client (è mostrato solamente lo scambio dei messaggi tra gli attori)

3.2.3 OneToOneChatServer

Si tratta di un attore creato dal RegisterServer a seguito dell'accettazione della richiesta del Client di creare una chat con un preciso user da lui selezionato; caratterizzato da un riferimento di cui il RegisterServer tiene traccia, esso di fatto permette di scambiare i messaggi tra il richiedente ed il destinatario. Esso deve essere in grado di ricevere i seguenti messaggi da parte del Client:

- `Message`: messaggio di testo inviato da uno dei due utenti che dovrà poi essere rinviato ad entrambi;
- `Attachment`: allegato inviato da uno dei partecipanti che sarà poi rinviato ai partecipanti.

Questo componente deve essere anche in grado di rispondere al RegisterServer qualora riceva il messaggio `DoesContainsMembers`: esso deve indicare se la chat avviene tra due precisi utenti, in quanto questa informazione risulta utile nei seguenti casi:

- quando un utente decide di creare una nuova chat (se la chat è già attiva, non deve essere creato un altro chat server);
- quando un utente richiede il riferimento del chat server che gestisce una chat con un preciso user;
- quando un utente decide di deregistrarsi, in quanto tutte le chat singole in cui partecipa devono essere chiuse, mentre quelle di gruppo in cui è partecipe devono provvedere ad eliminarlo.

Di seguito si riporta un diagramma di sequenza che mostra il processo di interazione tra 2 Client e relativo OneToOneChatServer dove il "Client 1" invia un messaggio di testo al "Client 2".

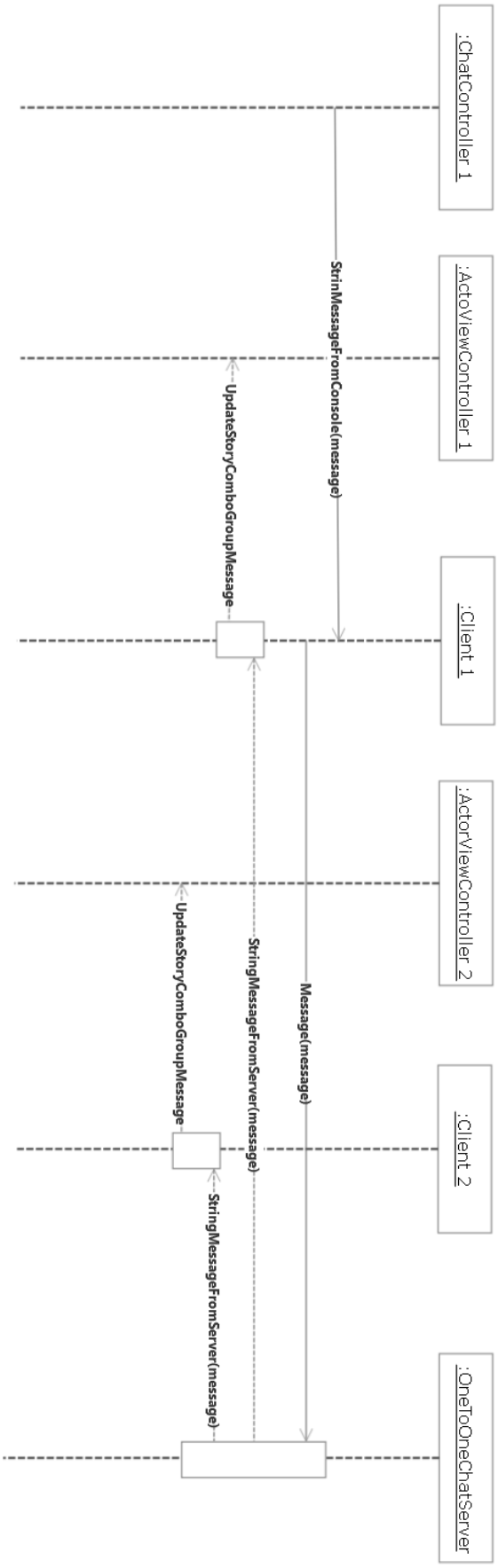


Figura 5 - invio di un messaggio di testo nell'ambito di una chat uno-a-uno (è mostrato solamente lo scambio dei messaggi tra gli attori)

Come è possibile notare la logica adottata è la seguente: il mittente (view del Client 1) invia un messaggio al server, il quale, una volta ricevuto, lo ri-inoltra ad entrambi i client i quali lo comunicano ad i relativi attori che si interfacciano con le view.

3.2.4 GroupChatServer

In maniera analoga al OneToOneChatServer, tale componente viene creato dal RegisterServer nel momento in cui la richiesta di un Client di creare una chat di gruppo, è accolta positivamente. Deve poter ricevere i seguenti messaggi:

- `AddMember`: richiesta di aggiungere un utente alla chat di gruppo;
- `RemoveMember`: richiesta di eliminare un utente;
- `GroupMessage`: messaggio testuale inviato da uno dei partecipanti, che sarà poi inoltrato a tutti i membri del gruppo;
- `GroupAttachment`*:* allegato inviato da un utente del gruppo, rinviato poi agli utenti iscritti al gruppo.

A differenza del server che gestisce chat singole, questo componente riceve messaggi esclusivamente dai client che partecipano al gruppo o che richiedono di partecipare; questa soluzione di bypassare il RegisterServer risulta conveniente qualora vi siano numerose richieste di aggiungersi o eliminarsi dalla chat.

Il seguente diagramma è volto a mostrare l'insieme di messaggi scambiati qualora il "Client1" voglia unirsi alla chat di gruppo "groupName" (caso di risposta positiva) e la sequenza di messaggi scambiati quando il "Client2" richiede di eliminarsi da tale chat (caso di risposta positiva).

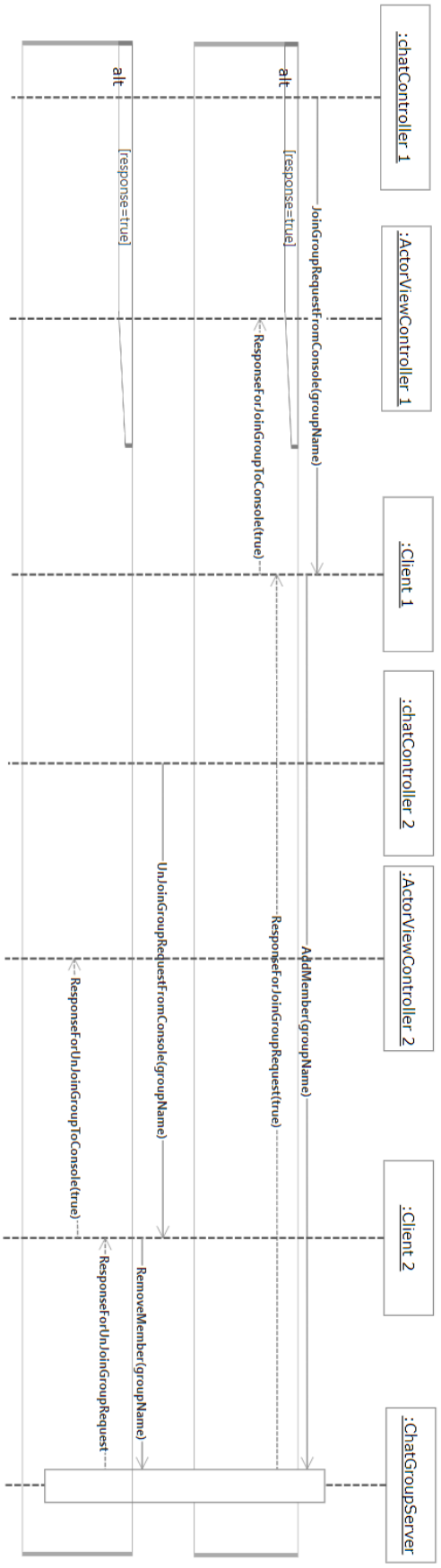


Figura 6 - aggiunta di un client ad una chat di gruppo ed eliminazione di un altro client (è mostrato solamente lo scambio dei messaggi tra gli attori)

3.2.5 ChatController e ActorViewController

Si tratta del controllore che si interpone tra la view ed il Client; come già anticipato, esso invia messaggi al Client a seguito dell'interazione dell'utente con la view, mentre riceve i messaggi dal Client per mezzo del ActorViewController. I suoi principali compiti sono quelli di:

- aggiornare l'elenco degli utenti attivi e delle chat di gruppo;
- visualizzare i messaggi/immagini scambiate con gli altri utenti;
- inviare messaggi/allegati.

ActorViewController è un attore che comunica con la view invocando i suoi metodi pubblici; esso deve essere in grado di ricevere i seguenti messaggi provenienti dal Client:

- `ResponseForChatCreation:` risposta dal server per la richiesta di creazione di una chat singola;
- `ResponseForChatGroupCreation:` risposta del server per la richiesta di creazione di una chat di gruppo;
- `UpdateUserAndGroupActive`: elenco degli utenti attivi e delle chat di gruppo create;
- `UpdateStoryComboMessage`: elenco dei messaggi (sia testuali che di immagini) per le chat singole seguite;
- `UpdateStoryComboGroupMessage`: elenco dei messaggi (sia testuali che di immagini) per le chat di gruppo seguite;
- `ResponseForJoinGroupToConsole:` risposta dal server per la richiesta di aggiungersi ad una chat di gruppo;
- `ResponseForUnJoinGroupToConsole`: risposta del server per la richiesta di non far più parte di una chat di gruppo.

Capitolo 4

Design di Dettaglio

In questo capitolo sono analizzate con un maggiore livello di dettaglio alcune scelte adottate per lo sviluppo dei principali componenti.

4.1 Client

Nella realizzazione di questo componente, riportiamo il dettaglio di alcune soluzioni impiegate per le seguenti funzionalità:

- invio messaggi/allegati/join/unjoin: ogni qual volta risulti necessario inviare un messaggio o richiedere di eseguire una join o unjoin ad un gruppo, è indispensabile avere il riferimento dell'attore quale destinatario del messaggio di richiesta. Qualora il riferimento non sia noto, questo viene richiesto al RegisterServer e successivamente, memorizzato in locale dal Client e quindi completata la richiesta iniziale. Questa successione di azioni è stata svolta attraverso un algoritmo che opera nella seguente maniera:
    - viene eseguita una ricerca nella struttura dati locale del riferimento dell'attore destinatario del messaggio; se la ricerca fornisce esito positivo, si invia il messaggio richiesto e ci si ripone in attesa della ricezione di un messaggio; qualora la ricerca fornisca esito negativo si passa al punto seguente;
    - invio messaggio per richiedere riferimento del server (sia chat singola che di gruppo);
    - si cambia il comportamento dall'attore per fare in modo che messaggi che da questo momento ricevuti e non inerenti la risposta del server, siano posti da parte (bufferizzati), mentre quelli relativi alla risposta del server siano intercettati ed il riferimento ottenuto (qualora esista) sia memorizzato in locale;
    - si ripristina il comportamento inziale;
    - se il riferimento è esistente, si invia nuovamente un messaggio al Client con la richiesta iniziale in modo da ripetere il passo 1 dell'algoritmo che questa volta porta ad esito positivo nella ricerca locale del riferimento dell'attore;
- richiesta di creazione di una chat: è stata realizzata con la medesima logica dove se il nome del client con cui si desidera creare una chat non è noto, esso è richiesto al server, e se ottenuto, si invia la richiesta di creazione della chat.

Poiché uno dei requisiti è quello di mostrare i messaggi nello stesso ordine in tutti i client, e dal momento che esistono due tipi di chat (singole e di gruppo), si è scelto di realizzare un unico metodo in grado di ordinare i messaggi per entrambi i tipi di classe, con degli appositi *adapter* in grado, appunto, di rendere il metodo adeguato ad eseguire questa operazione su ciascun tipo di messaggio.

4.2 RegisterServer e RegisterModel

Per la realizzazione del registro si è scelto di separare il suo comportamento di attore dall'organizzazione dei dati che è chiamato a gestire. Pertanto, si è scelto di incapsulare l'intera gestione delle strutture dati in un'apposita classe, denominata RegisterModel, la quale fornisce al RegisterServer un accesso ai dati "filtrato" degli aspetti implementativi, ossia agendo da *façade*.

L'intero metodo `receive` del RegisterServer è stato quindi organizzato in modo da demandare le operazioni sui dati alla classe RegisterModel, in modo da massimizzare la leggibilità del metodo stesso, che contiene solamente le invocazioni di metodo per l'esecuzione delle operazioni.

La classe RegisterModel è a sua volta strutturata in maniera tale da consentire una manipolazione funzionale dei dati in essa contenuti (aspetto descritto nella sezione successiva), in linea con l'utilizzo del linguaggio Scala.

Capitolo 5

Implementazione

In questo capitolo sono analizzati nel dettaglio i contributi apportati da ciascun membro del gruppo. Come mostrato nel capitolo 1, la modellazione delle funzioni del sistema è descritta tramite casi d'uso, di conseguenza la suddivisione dei compiti è avvenuta proprio seguendo un'assegnazione dei casi d'uso da implementare. Un fattore non trascurabile è il tempo che si è dedicato nella realizzazione dei test, i quali, nell'ottica TDD, risultano essere tanto il punto di partenza dello sviluppo della funzionalità, nonché l'obiettivo da raggiungere inteso come volontà di passare tutti i test realizzati. Per la realizzazione dei test sono state utilizzate le librerie `ScalaTest` ed in particolare per il testing di tutto ciò legato al modello ad attori (che rappresenta la parte pervasiva del progetto), si è impiegato il `TestKit` messo a disposizione dal framework Akka.

5.1 Componenti creati congiuntamente

Architettura:

- identificazione componenti e relative funzionalità (aspetti visti precedentemente)
- creazione del diagramma dei casi d'uso in UML;
- creazione dei diagrammi di sequenza in UML.

Componenti:

- ChatGroupServer
- ChatController (controller della view)
- Client, in particolare:
    - Ordinamento dei messaggi in arrivo dalle varie chat

5.2 Giacomo

Lo studente Giacomo Scaparrotti si è occupato dello sviluppo dei seguenti componenti:

- RegisterServer ed in particolare sono state realizzate le seguenti funzionalità:
    - registrazione di un utente e relativa risposta;
    - deregistrazione di un utente;
    - creazione di una chat singola e relativa risposta;
    - creazione di una chat di gruppo e relativa risposta;
    - elenco di tutti gli utenti e dei gruppi attivi;
    - riferimento (inteso come riferimento dell'attore) di un OneToOneChatServer;
    - riferimento di un GroupChatServer;
- RegisterModel
    - Gestione delle strutture dati relative al RegisterServer (es: elenco degli utenti)
- Utils

    - Utilità per la gestione dei dati all'interno dell'applicazione
- OneToOneChatServer

All'interno di questi componenti, si ritiene che siano degni di nota i seguenti aspetti:

- La ricerca della chat singola contenente due particolari attori, necessaria quando viene richiesta da un client, è effettuata interrogando tutti i ChatServer tramite il metodo `ask` messo a disposizione da Akka, che restituisce una Future per ogni messaggio inviato, contenente in questo caso la risposta affermativa o negativa proveniente da un ChatServer. Sarà quindi presente una lista di Future, che viene poi convertita in una singola Future di liste, di cui si potrà poi attendere l'esito per avere la risposta definitiva, contenente una serie di risposte negativa e, eventualmente, un'unica risposta affermativa da parte del ChatServer cercato.
- All'interno del RegisterModel l'elenco degli utenti e dei gruppi è mantenuto sotto forma di mappe aventi come chiavi i nomi degli utenti o dei gruppi e come valori il riferimento al relativo attore. Per poter eseguire delle operazioni a seconda che un utente (o un gruppo) sia presente o meno, è stata definita una classe implicita avente un oggetto generico di tipo `Map[A,B]` come parametro del costruttore, contenente due metodi che, appunto, ricercano una chiave o un valore all'interno della suddetta mappa eseguendo poi una funzione passata come parametro in caso di successo o un'altra funzione in caso di fallimento. Poiché tale classe è definita come implicita, è possibile invocare questi metodi direttamente su una normale mappa senza eseguire alcuna conversione esplicita, semplificandone così l'utilizzo.
- Quando viene effettuata un'operazione su RegisterModel, il metodo che la effettua restituisce un oggetto di tipo `OperationDone[A]`, che offre due metodi (`ifSuccess` e `orElse`, chiamabili in modo *fluent*) a cui è possibile passare delle funzioni che verranno eseguite a seconda che l'operazione richiesta sia andata a buon fine o meno.

5.1 Massimiliano

Lo studente Massimiliano Giunchi si è occupato dello sviluppo dei seguenti componenti:

- Client, in particolare:
    - richiesta di login (registrazione) e ricezione risposta;
    - invio richiesta creazione nuova chat e ricezione risposta;
    - invio richiesta creazione nuova chat di gruppo e ricezione risposta;
    - richiesta di join/unjoin ad una chat di gruppo e ricezione risposta;
    - invio e ricezione di un messaggio di testo ad un chat server;
    - invio e ricezione di un'immagine ad un chat server;
- GUI Login e Client
- LoginController
- ActorViewController

Capitolo 6

Retrospettiva

Nel complesso lo sviluppo si è svolto in maniera efficace e senza particolari intoppi di carattere organizzativo.

Il modello di sviluppo adottato (TDD e Scrum-inspired) si è infatti rivelato vincente, in quanto ha permesso di ottenere rapidamente del codice ben funzionante e, soprattutto, di poterne verificare il corretto funzionamento durante i successivi passi di sviluppo. Più di una volta, infatti, i test precedentemente realizzati hanno portato in evidenza dei bug nel codice appena sviluppato che sarebbe altrimenti stato problematico da riparare in un qualche momento successivo.

Il fatto di aver eseguito degli sprint settimanali, delineando di volta in volta i nuovi compiti di ciascun membro del gruppo, ha consentito di mantenere un ritmo grossomodo costante nell'aggiunta di nuova funzionalità al software. Il delineamento dei diversi compiti è stato facilitato dalla natura del software stesso, in quanto, come visibile nei capitoli precedenti, un membro si è principalmente occupato del lato client, mentre l'altro si è principalmente occupato del lato server.

Trello è stato utile per organizzare il lavoro durante gli sprint, in modo da mettere in evidenza eventuali criticità rilevate dai membri del team anche a distanza.

Figura 7 - utilizzo di Trello durante lo sviluppo del software

Dal punto di vista tecnico sono da evidenziare diversi aspetti positivi e qualche aspetto negativo.

Innanzitutto, è da evidenziare il fatto che tutti i principali requisiti e obiettivi sono stati raggiunti, ottenendo un sistema ben funzionante e testato.

L'integrazione tra il modello MVC e il modello ad attori è risultata ben funzionante, consentendo di cogliere vari aspetti positivi da entrambi: la separazione delle responsabilità tra le varie componenti del software da un lato, la sicurezza e la robustezza dell'interazione tra essi dall'altro.

Al contempo, l'aver strettamente suddiviso le chat in chat singole e chat di gruppo si è rivelato, almeno in parte, un errore, in quanto in realtà questi due tipi di chat sono molto affini, e sarebbe forse stato meglio integrarle in un'unica forma di chat (con un trait in comune), in modo da semplificarne la gestione complessiva.

Nel complesso il team si ritiene soddisfatto del risultato finale, in quanto il sistema realizzato risponde pienamente alle specifiche, e l'esperienza dello sviluppo è stata positiva per le competenze acquisite e per il risultato ottenuto.