Assignment #3

Martina Magnani
martina.magnani8@studio.unibo.it

Nicola Piscaglia
nicola.piscaglia2@studio.unibo.it

Mattia Vandi
mattia.vandi@studio.unibo.it

Indice

Capitolo 1

Esercizio 1 - Actor-based Game of Life

1.1 Analisi del problema

L'obiettivo della consegna è implementare Implementare in Java o in Scala una versione ad attori del gioco *"The Game of Life"*.
Il gioco consiste nel calcolare e visualizzare l'evoluzione della matrice di celle che caratterizza il gioco, come sequenza di fotogrammi (ognuno dei quali rappresenta lo stato del mondo).
Nella matrice, ogni cella può essere in uno dei due stati possibili, *live* o *dead*.
Dato lo stato $s(t)$ della matrice, lo stato $s(t+1)$ si computa con le seguenti regole:

- una cella $m[i,j]$ che nello stato $s(t)$ è *live* e ha zero o al più una cella vicina *live* (e le altre *dead*), nello stato $s(t+1)$ diventa *dead* ("muore di solitudine")
- una cella $m[i,j]$ che nello stato $s(t)$ è *live* e ha quattro o più celle vicine *live*, nello stato $s(t+1)$ diventa *dead* ("muore di sovrappopolamento")
- una cella $m[i,j]$ che nello stato $s(t)$ è *live* e ha due o tre celle vicine *live*, nello stato $s(t+1)$ rimane *live* ("sopravvive")
- una cella $m[i,j]$ che nello stato $s(t)$ è *dead* e ha tre celle vicine *live*, nello stato $s(t+1)$ diventa *live*

Il gioco deve presentare un'interfaccia grafica con pulsanti `start` e `stop` con cui si fa partire e fermare il gioco. Ogni stato del gioco deve essere visualizzato, insieme al numero di celle nello stato *live*.

1.1.1 Requisiti del sistema

- Il programma deve funzionare anche con matrici di dimensioni significative (ad es. 5000x5000);
- Massimizzare il throughput, minimizzando il tempo di calcolo di ciascun fotogramma ed eventualmente anche della sequenza di fotogrammi;
- Massimizzare la reattività della GUI;
- Studiare e implementare meccanismi di coordinazione/sincronizzazione basati sullo scambio di messaggi.

1.2 Descrizione della soluzione proposta

1.2.1 Architettura del sistema

L'architettura del sistema è stata scomposta in tre livelli utilizzando il pattern Model View Presenter, a differenza della prima consegna l'aggiornamento della schermata utente e l'aggiornamento della scacchiera di gioco è completamente demandata ad un insieme di attori, di conseguenza vi sono tre moduli: il dominio applicativo, gli attori e l'interfaccia grafica:

- Nel primo livello vi è una rappresentazione object-oriented delle componenti del problema;
- Nel secondo livello è stata incapsulata la logica di aggiornamento del gioco e dell'interfaccia utente;
- Nel terzo livello viene organizzata l'interfaccia grafica che permette la visualizzazione del gioco e l'interazione con l'utente.

1.2.2 Dinamica del sistema

La dinamica del sistema è stata rappresentata ad alto livello con un diagramma di sequenza, adatto per mostrare in modo efficace lo scambio di messaggi tra le entità del sistema.

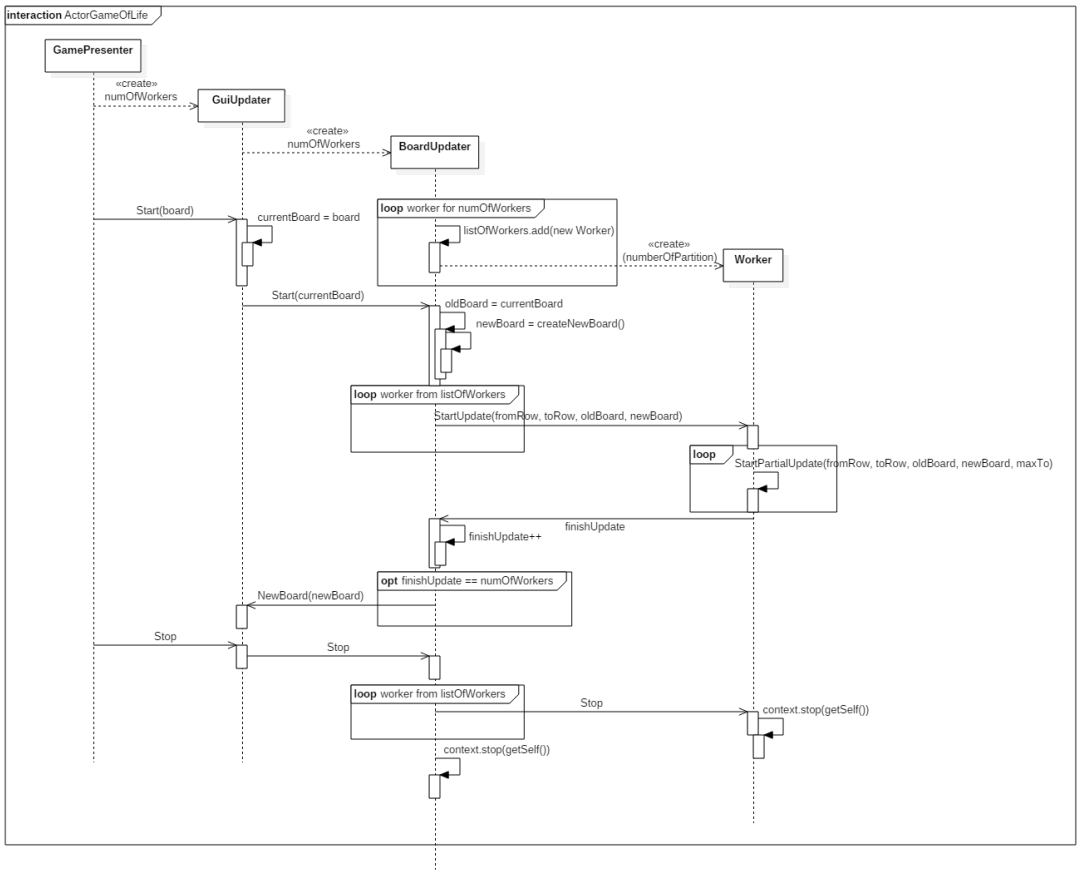

Figura 1.1: Diagramma di sequenza che modella la dinamica della computazione dello stato di gioco

1.2.3 Implementazione

1.2.3.1 Dominio applicativo

Abbiamo implementato il dominio applicativo attraverso due classi: `Cell` e `Board`.
L'enumerazione `Cell` rappresenta lo stato di una cella della scacchiera che può essere *live* o *dead*. La classe `Board` rappresenta la scacchiera di gioco. Questa è caratterizzata da un'*altezza* e una *larghezza* variabili e dalla possibilità di recuperare/impostare lo stato di una cella date le coordinate x e y.

1.2.3.2 Attori

Abbiamo implementato un attore `BoardUpdater` che si occupa dell'aggiornamento della scacchiera di gioco.
Dell'attore `BoardUpdater` è possibile configurare il numero di `Worker` che verranno utilizzati per aggiornare la schermata di gioco (ognuno dei quali verrà tradotto in un nuovo attore).

- Il messaggio `StartUpdate`, inviato dal `BoardUpdater`, informa ogni `Worker` della porzione di scacchiera di gioco che dovrà aggiornare.
- Il messaggio `StartPartialUpdate`, inviato da un `Worker` sé stesso, informa sé stesso sulla porzione di scacchiera di gioco da aggiornare prima di passare alla successiva. Tale strategia consente ad un attore di rimanere reattivo ai messaggi ricevuti da altri attori.
- Il messaggio `FinishedPartialUpdate`, inviato da un `Worker` a sé stesso, informa sé stesso che un aggiornamento parziale è terminato. Se l'aggiornamento parziale completato è l'ultimo allora informa sé stesso che può informare il `BoardUpdater` che l'aggiornamento assegnato al `Worker` è terminato, altrimenti l'attore informa sé stesso sulla prossima porzione di scacchiera da aggiornare.
- Il messagio `FinishedUpdate`, inviato ad un `Worker`, informa il `BoardUpdater` che l'aggiornamento assegnato ad un `Worker` è termianto. Se l'aggiornamento terminato è anche l'ultimo allora notifica il mittente fornendogli la scacchiera di gioco aggiornata.
- Il messagio `NewBoard`, inviato dal `BoardUpdater`, informa il `GuiUpdater` sulla prossima scacchiera di gioco da visualizzare all'utente.

1.2.3.3 Interfaccia utente

L'interfaccia utente è stata realizzata con JavaFX ed è costituita da due schermate: nella prima si permette all'utente di settare le dimensioni della scacchiera di gioco (altezza e larghezza) ed il numero di `Worker`. Nella seconda si visualizza la scacchiera e sono presenti due pulsanti *start* e *stop*. Lo *start* permette l'avvio e la pausa del gioco, mentre lo *stop* la terminazione.
Utilizzando il pattern Factory sono stati incapsulate in una apposita classe le funzioni per la costruzione delle finestre grafiche. Le callback di queste finestre sono definite da opportuni `Presenter` che incapsulano la logica di controllo dei componenti (form, pulsanti, ...) di cui sono costituite.
La computazione relativa alla coordinazione dell'aggiornamento matriciale e grafico della scacchiera è effettuata dall'attore `GuiUpdater` e questo permette di mantenere l'interfaccia grafica (gestita dal JavaFX Application Thread) reattiva ed in grado di intercettare l'input dell'utente.

- Il messaggio `Start`, inviato dal `GuiUpdater` al `BoardUpdater`, si occupa di avviare l'aggiornamento della schermata di gioco passata in input.
- Il messaggio `Pause`, inviato dal `GamePresenter` a fronte della pressione del pulsante associato, informa l'attore `GuiUpdater` della messa in pausa del gioco da parte dell'utente.
- Il messaggio `Resume`, inviato dal `GamePresenter` a fronte della pressione del pulsante associato, informa l'attore `GuiUpdater` della ripresa del gioco.
- Il messaggio`Stop`, inviato dal `GamePresenter` a fronte della pressione del pulsante associato, informa l'attore `GuiUpdater`, che informerà gli altri attori sulla volontà dell'utente di voler termianre la simulazione.

Il Presenter addetto alla gestione degli eventi di gioco (`GamePresenter`) utilizza un servizio di rendering offerto dalla classe `RenderingService` nella quale sono state incapsulate le funzioni relative al disegno della scacchiera di gioco.
Quest'ultima è stata realizzata utilizzando un componente grafico `Canvas` sul quale vengono disegnate le celle attraverso la classe `PixelWriter` di JavaFX: una classe ottimizzata per il disegno dei singoli pixel.

1.3 Analisi delle prestazioni

La tabella sottostante mostra gli speedup S (misura quantitativa delle performance) che il sistema raggiunge a seconda del numero di worker utilizzati.
Lo speedup viene misurato calcolando il rapporto tra il tempo di esecuzione del sistema utilizzando un singolo worker e il tempo di esecuzione del sistema utilizzando un algoritmo parallelo.

$$S = \frac{T_1}{T_N} \tag{1.1}$$

Dove:

- $N \leftarrow$ numero di worker utilizzati
- $T_1 \leftarrow$ tempo di esecuzione del sistema utilizzando un algoritmo sequenziale
- $T_N \leftarrow$ tempo di esecuzione del sistema utilizzando un algoritmo parallelo con N worker

	2	3	4	5	6	7	8	9	10
Min.	1.85	2.45	2.49	2.71	2.74	2.81	2.72	2.72	2.78
Max.	1.73	2.14	1.91	2.18	1.58	2.52	2.39	2.40	2.08
Avg.	1.83	2.35	2.36	2.51	2.55	2.62	2.60	2.59	2.56

Tabella 1.1: Lo speedup è stato calcolato su 25 iterazioni considerando una schacchiera di gioco 5000x5000. Sulle colonne sono specificati il numero di worker utilizzati per la parallelizzazione dell'algoritmo, sulle righe i valori statistici (minimi, massimi e medi) degli speedup calcolati. Gli speedup sono stati calcolati su un Apple Macbook Pro (Retina, 15", metà 2015) dotato di un processore Intel Core i7 quad-core a 2.2GHz. È possibile eseguire i benchmark lanciando la classe `pcd.ass03.ex1.Benchmark` e visualizzare sia le velocità di computazione di uno stato della scacchiera, sia i relativi speedup ottenuti.

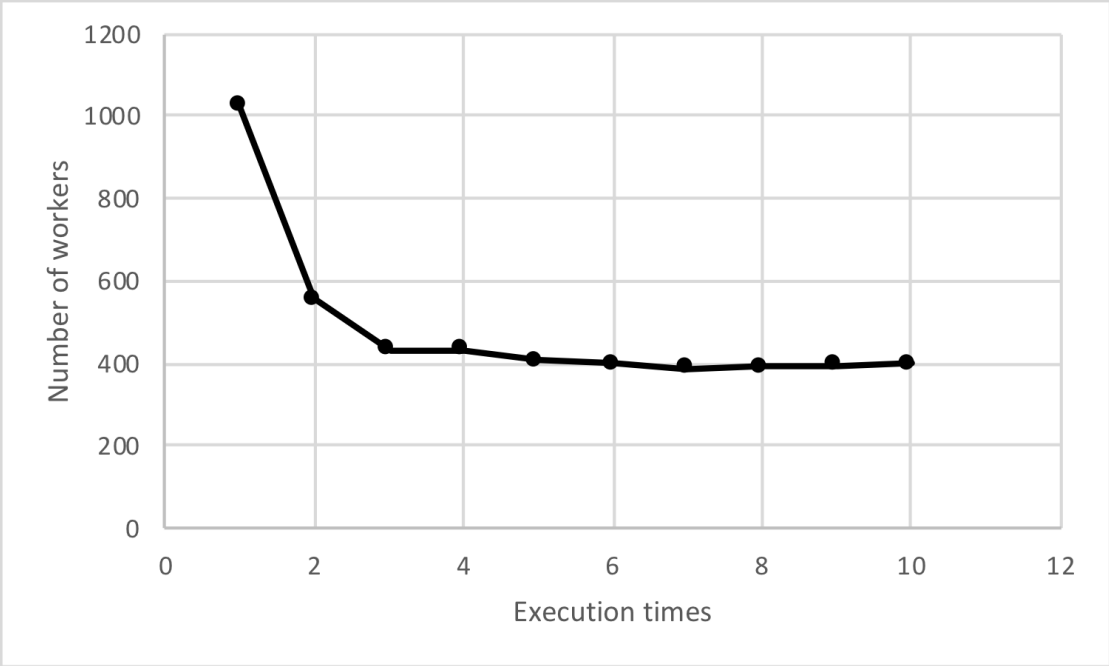

Figura 1.2: Grafico che mostra l'evoluzione dei tempi di esecuzione (in ms) al crescere del numero di worker. I tempi di esecuzione sono stati calcolati come media su 25 iterazioni considerando una schacchiera di gioco 5000x5000. I tempi di esecuzione sono stati calcolati su un Apple Macbook Pro (Retina, 15", metà 2015) dotato di un processore Intel Core i7 quad-core a 2.2GHz.

Capitolo 2

Esercizio 2 - Actor-based distributed chat

2.1 Analisi del problema

Implementare una chat distribuita ad attori, con l'insieme dei partecipanti dinamico:

- Un utente può aggiungersi e rimuoversi dinamicamente.
- Ogni messaggio inviato da un utente deve essere visualizzato da tutti gli altri utenti.
- Il sistema deve essere completamente decentralizzato.
  - A parte, eventualmente, la presenza di un (attore) registro con indirizzo/nome noto che tenga traccia dei partecipanti.
- I messaggi inviati nella chat devono essere visualizzati da tutti i partecipanti nel medesimo ordine.

Supporto per una modalità "sezione critica":

- Un partecipante può chiedere di entrare in sezione critica inserendo un comando predefinito (es: ":enter-cs").
- Quando un partecipante entra in sezione critica, possono essere visualizzati solo i suoi messaggi, senza intervallarli a quelli degli altri utenti.
- Un solo partecipante alla volta può essere in sezione critica.
- Per uscire dalla sezione critica si può prevedere un altro comando predefinito (es: ":exit-cs").
- Un utente può rimanere in sezione critica per un certo tempo massimo Tmax, dopodiché l'uscita è forzata.
- Durante una sezione critica i messaggi inviati dagli altri partecipanti devono essere rigettati.

2.2 Descrizione della soluzione proposta

2.2.1 Architettura del sistema

Il sistema è stato organizzato utilizzando il modello ad attori. In particolare, gli attori presenti nella nostra soluzione sono due:

- `User`: attore che si unisce alla stanza per partecipare alla chat distribuita.
- `Room`: attore centrale che funge da registro degli utenti presenti e inoltra i messaggi a tutti i partecipanti della chat. La `Room` gestisce anche la modalità 'sezione critica' della stanza.

2.2.2 Dinamica del sistema

La dinamica del sistema è stata rappresentata ad alto livello con un diagramma di sequenza, adatto per mostrare in modo efficace lo scambio di messaggi tra le entità del sistema.
Di seguito vengono mostrati i tre scenari d'interazione.

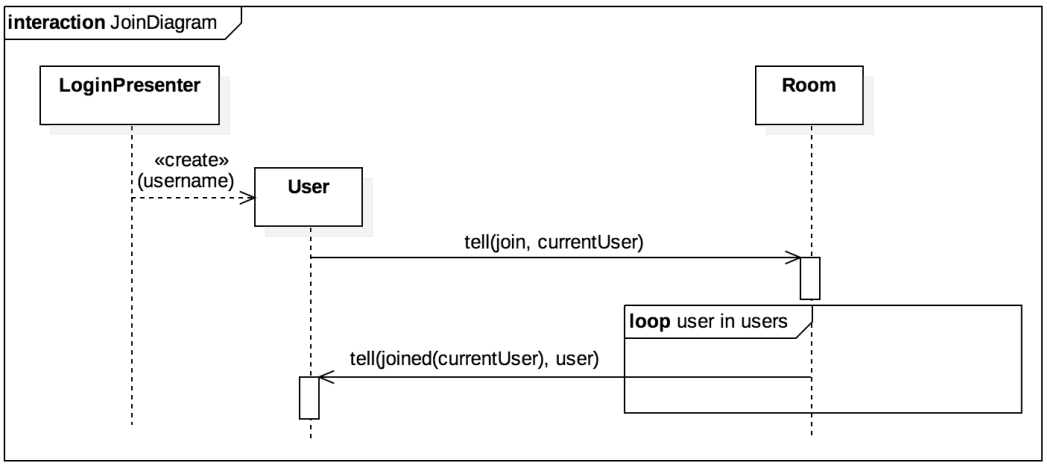

Figura 2.1: Diagramma di sequenza che mostra l'unione di un utente alla stanza.

Nell'interazione descritta in figura 2.1 la sequenza di messaggi scambiati è la sequente: (1) viene creato il nuovo utente; (2) l'utente creato segnala alla stanza l'intenzione di volersi unire (join); (3) la room segnala a tutti gli utenti presenti nella stanza l'aggiunta di un nuovo utente.

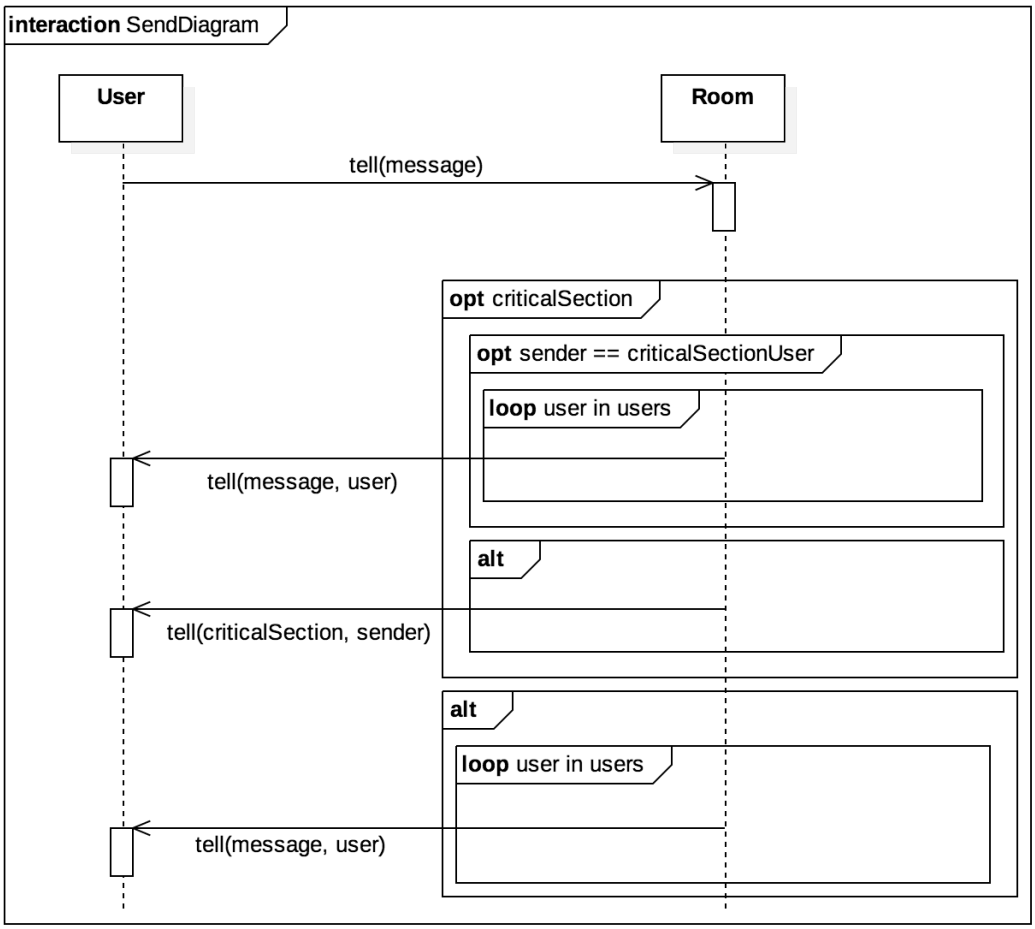

Figura 2.2: Diagramma di sequenza che mostra l'invio di un messaggio da parte di un utente al resto degli utenti presenti nella stanza.

Nell'interazione descritta in figura 2.2 la sequenza di messaggi scambiati è la sequente:

(1) L'utente manda un messaggio alla room.

(2) Nel caso in cui ci sia una sezione critica:

   (a) se il mittente è l'utente in sezione critica il messaggio viene mandato a tutti gli altri;

   (b) se il mittente non è l'utente in sezione critica, il messaggio viene rifiutato e viene segnalata al mittente l'identità dell'utente in sezione critica.

(3) Nel caso in cui non ci sia una sezione critica il messaggio viene inviato a tutti gli altri utenti.

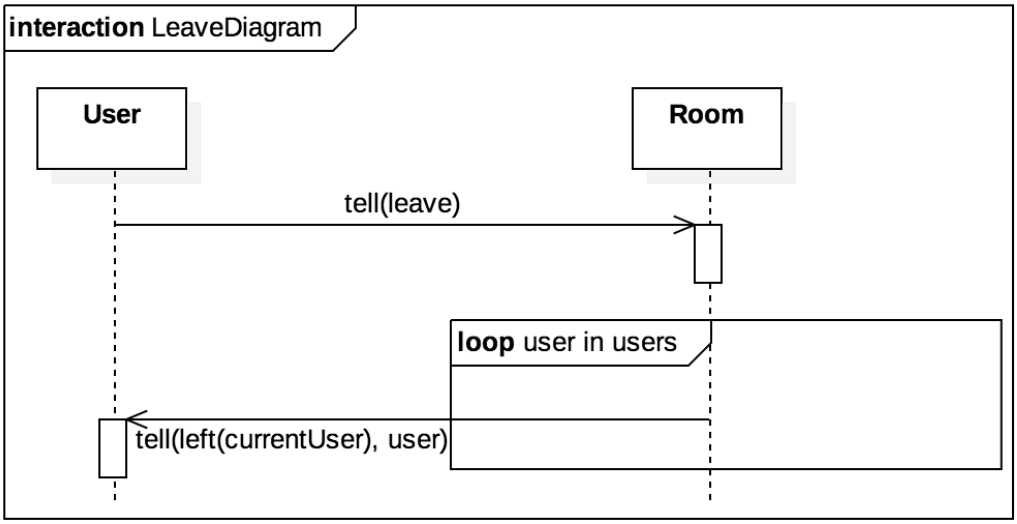

Figura 2.3: Diagramma di sequenza che mostra l'uscita di un utente dalla stanza.

Nell'interazione descritta in figura 2.3 la sequenza di messaggi scambiati è la sequente: (1) l'utente segnala alla room che sta lasciando la stanza; (2) la stanza segnala a tutti gli altri utenti l'identità dall'utente che ha lasciato la stanza.

2.2.3 Algoritmo di ordinamento

Il problema dell'ordinamento dei messaggi può essere scomposto in due sottoproblemi tra loro indipendenti:

- **causal message ordering**: assicura che tutti i messaggi vengano elaborati secondo il loro ordine di invio;
- **total message ordering**: assicura che tutti i processi elaborino i messaggi nello stesso ordine.

Il problema dell'ordinamento dei messaggi è stato risolto utilizzando due algoritmi che descriviamo di seguito. Questo permette ad ogni utente di avere una visione coerente e ordinata dei messaggi.

2.2.4 Causal Message Ordering

Ogni messaggio è costituito dal testo e dal numero di sequenza associatogli dal mittente. Ogni utente mantiene per ogni processo il numero di messaggi inviati e viene aggiornato ogni volta che l'utente è abilitato a processare il messaggio ricevuto.
Un utente è abilitato ad elaborare il messaggio quando il numero di sequenza all'interno del messaggio è successivo al numero di sequenza dell'ultimo messaggio elaborato per quel mittente. Se così non fosse, significa che ci sono dei messaggi precedenti del mittente che non sono ancora stati ricevuti ed è quindi necessario attendere il loro arrivo prima di potere elaborare il messaggio appena ricevuto.

1. Un processo che si unisce al sistema riceve da tutti gli altri processi il loro numero di sequenza corrente.

2. Un processo che invia un messaggio incrementa il proprio numero di sequenza e lo associa al messaggio.

3. Un processo che riceve un messaggio controlla che il numero di sequenza contenuto nel messaggio sia successivo all'ultimo numero di sequenza che ha ricevuto dal mittente.
   Se tale condizione è soddisfatta:

   (a) il messaggio viene elaborato

   (b) se ci sono altri messaggi che possono essere elaborati si ritorna al punto precedente.

   Se tale condizione non è soddisfatta il messaggio viene aggiunto alla coda dei messaggi in attesa.

4. Un processo che esce dal sistema viene rimosso dalle liste di tutti gli altri processi.

2.2.5 Total Message Ordering

I processi inviano i messaggi al *Sequencer*: entità che assegna un numero di sequenza globale ad ogni messaggio. Ogni processo mantiene un contatore che rappresenta il numero di sequenza globale dell'ultimo messaggio elaborato.
Quando un utente invia un messaggio:

1. Il *Sequencer* assegna al messaggio un numero di sequenza.

2. Un processo che riceve un messaggio controlla che il numero di sequenza del messaggio sia successivo all'ultimo numero di sequenza salvato.
   Se tale condizione è soddisfatta:

   (a) il messaggio viene elaborato

   (b) se ci sono altri messaggi che possono essere elaborati si ritorna al punto precedente.

   Se tale condizione non è soddisfatta il messaggio viene aggiunto alla coda dei messaggi in attesa.

2.2.6 Implementazione

La soluzione è stata implementata in linguaggio Scala utilizzando la libreria Akka. È stato scelto di utilizzare Scala e in particolare la libreria Akka perché permette di implementare gli attori in modo più agile e coinciso, evitando di scrivere *boilerplate code*.
Le entità presenti nel sistema sono gli utenti della chat (rappresentati dall'attore `User`) e un sistema che funge da registro e da coordinatore (rappresentato dall'attore `Room`).

2.2.6.1 Dettaglio d'interazione

2.2.6.1.1 Join L'utente che si unisce alla stanza invia a quest'ultima un messaggio di `Join`; tale messaggio viene inviato durante l'esecuzione del metodo `preStart()` del ciclo di vita di un attore Akka. Alla ricezione di tale messaggio la stanza informa gli utenti dell'arrivo del nuovo utente (`Joined(user)`) e risponde a quest'ultimo con il numero di sequenza globale della stanza (che d'ora in poi chiameremo s_g). Gli altri utenti, informati dalla stanza, inviano al nuovo utente

il loro personale numero di sequenza (relativo al numero di messaggi da loro inviati, che d'ora in poi chiameremo s_{u_i}) affinché possa aggiungerlo ad una lista. Questo permette al nuovo utente di sincronizzarsi con gli altri utenti, così da poter elaborare d'ora in avanti i messaggi scambiati. Infine la stanza aggiunge il nuovo utente alla lista degli utenti presenti nel sistema.

2.2.6.1.2 Messages Quando un utente i intende inviare un messaggio prima chiede alla stanza se è in sezione critica. Se la stanza non è in sezione critica incrementa s_{u_i} e poi lo allega al messaggio (`Message(content, user, userCounter)`). Altrimenti nel caso l'utente detenga la sezione critica, allora può inviare il messaggio come descritto nel punto precedente. Nel caso in cui sia detenuta da qualcun altro non può inviare il messaggio. La stanza, alla ricezione del messaggio, incrementa s_g ed inoltra a tutti gli utenti del sistema il messaggio allegandogli s_g.

Ogni utente, alla ricezione del messaggio inviatogli dalla stanza, controlla che s_g contenuto nel messaggio sia successivo all's_g dell'ultimo messaggio elaborato. Se così fosse, è possibile passare al controllo dell'ordine causale che consiste nel verificare che il numero di sequenza s_{u_i} contenuto nel messaggio sia successivo al s_{u_i} dell'ultimo messaggio eleborato che è stato inviato dall'utente i.

2.2.6.1.3 EnterCriticalSection & ExitCriticalSection Per entrare in sezione critica, un utente invia il messaggio `EnterCriticalSection` alla stanza. La sezione critica può essere acquisita solo se non è detenuta da qualche altro utente, in tal caso il richiedente acquisisce la sezione critica e solo lui potrà inviare messaggi.
Per uscire dalla sezione critica, l'utente che la detiene dovrà inviare il messaggio `ExitCriticalSection`. A questo punto la stanza informa tutti gli utenti che la sezione critica è stata rilasciata.

2.2.6.1.4 Leave Quando un utente esce dal sistema, viene chiamato il metodo del ciclo di vita di un attore Akka `postStop()`, il quale invia un messaggio `Leave` alla stanza. Quest'ultima informa tutti gli altri utenti dell'uscita dell'utente, così sia la stanza sia gli utenti possono rimuoverlo dalla lista.

2.2.6.2 Interfaccia grafica

L'interfaccia grafica è stata realizzata utilizzando ScalaFX: una libreria che wrappa le funzionalità di JavaFX fornendole in stile funzionale. All'interno del sistema sono presenti due schermate principali: una per il login e una per la chat.
Per ogni vista è presente un'apposita classe con suffisso `View` che ne definisce i componenti grafici e una classe con suffisso `Presenter` che gestisce gli eventi ad essa associati. La `ChatView` al momento della sua inizializzazione si preoccupa di mettere in esecuzione il sistema degli Attori, mentre il relativo `ChatPresenter` mantiene il riferimento all'attore che modella l'utente per gestire le interazioni nel sistema a seguito degli eventi scatenati nell'interfaccia e viceversa.

.1 Comandi

I comandi inviabili alla stanza sono:

- *:enter-cs*: per richiedere l'entrata in sezione critica da parte di un utente
- *:exit-cs*: per richiedere l'uscita dalla sezione critica da parte di un utente
- *:help*: per richiedere la lista dei comandi disponibili