RELAZIONE ASSIGNMENT 3 PCD

**Semprini Luca - 0000854447**

**Riciputi Jacopo**

La richiesta di questo assignment consisteva nell’implementazione di due esercizi riguardanti il modello ad Attori, rispettivamente:

* **Esercizio 1**: Implementare una versione ad attori di Game Of Life ad attori.
* **Esercizio 2**: Implementare una chat distribuita ad attori.

**ESERCIZIO 1 – Game of Life (Actor-Based)**

Per la realizzazione dell’esercizio 1 è stato costruito un sistema di attori partendo dalla base fornita dalle soluzioni dell’Assignment 1, pur andando a modificare la classe principale MatrixImpl, ma lasciando sostanzialmente inalterata la parte di View.

La parte di modello, in sostituzione a quella basata sui Thread, si suddivide in due tipologie di attori, **Master** e **Worker**.

**-Master-**

Vi è un unico riferimento a questo attore all’interno del sistema; esso si occupa della gestione del calcolo della matrice e della coordinazione tra Worker e View.

Al momento dell’avvio della computazione, crea, se non esiste, una lista di Worker ai quali andrà ad assegnare una parte di celle da computare.

La delega alla computazione da parte dei worker avviene ad uno ad uno così da permettere una maggiore reazione in caso di **Stop** da parte dell’utente.

Al termine computazione ogni Worker manda una notifica al master tramite il messaggio **UpdateGUIMsg**. A questo punto il master in stile *Cyclic Barrier* attende una notifica da parte di tutti i Worker per poi avviare l’aggiornamento della GUI e successivamente ripetere le operazioni.

Dettaglio sui messaggi a cui risponde Master:

* **StartSystemMsg**: Se non ha già dato l'avvio ai Worker li richiama e li avvia. Nel caso in cui sia il primo avvio crea anche i Worker.
* **CallWorkerMsg**: Se non si è in stato di stop e non sono stati lanciati tutti i Worker li richiama e ne avvia uno. Eseguita questa operazione si richiama se devono essere chiamati altri Worker.
* **UpdateGUIMsg**: Lavora ideologicamente come una *Cyclic Barrier*, dopo essere stato contatto tot volte significa che i worker hanno terminato i rispettivi compiti sulla matrice ed è pronta per essere renderizzata.
* **StopMsg**: Notifica che dalla GUI è stato dato input di bloccare le operazioni.

Propaga il messaggio anche a tutti i Worker.

**-Worker-**

L’implementazione prevede la creazione di tanti Worker quanti sono i core della macchina su cui sta girando il programma (con un fattore moltiplicativo che dà la possibilità di avere più Worker per ogni core). Questo attore si “sveglia” quando riceve un messaggio di tipo Start da parte del Master; a questo punto si auto-invia un messaggio di tipo **UpdateMatrix**, che gestisce l’operazione di update della matrice (che consiste sostanzialmente nel computare una generazione), controllando che l’attore non sia stato preventivamente stoppato; il Worker si auto-invia, a questo punto un messaggio di **ComputeUpdateMatrix**, che gestisce l’omonima operazione di computeUpdate della matrice (che consiste nell’aggiornare le celle correnti con quelle calcolate dall’operazione di update), e invia un messaggio al Master di tipo **UpdateGUI**. Worker accetta anche un messaggio di tipo Stop che setta il flag di stop a true.

Dettaglio sui messaggi a cui risponde Worker:

* **StartMsg**: Setta il flag di stop a false ed avvia il cronometro. Invia a sé stesso un messaggio di UpdateMatrix con tutte le informazioni sulla matrice su cui lavorare.
* **UpdateMatrix**: Controlla che non sia stato stoppato. Tramite il messaggio esegue operazione di update sulla matrice e infine invia a sé stesso un messaggio di ComputeUpdateMatrix.
* **ComputeUpdateMatrix**: Controlla che non sia stato stoppato, ferma il cronometro e invia al Master un messaggio per aggiornare la GUI.
* **StopMsg**: Notifica che dalla GUI è stato dato input di bloccare le operazioni, quindi setta il flag di stop a true.

**-NOTE SULLE PERFORMANCE-**

Sono stati effettuati dei test di performance su varie configurazioni del programma (ad esempio su matrici di differenti dimensioni e con un numero di Worker variabile); questi vengono riportati all’interno di un file “performance.txt”, in allegato alla consegna. In ogni caso, avendo scelto di utilizzare l’implementazione della View fornita a lezione nella soluzione del primo assignment, per ogni generazione effettuata dal programma, viene stampato su un TextField il tempo trascorso in millisecondi per calcolare quella generazione.

**ESERCIZIO 2 – Actor-Based Distributed Chat**

L’implementazione dell’esercizio 2 presentava la difficoltà di mostrare per ogni attore presente nel sistema i messaggi inviati all’interno della chat nel medesimo ordine.

Per fare ciò ci siamo ispirati all’idea presentata dal *Token ring*, solamente chi è in possesso del token ha la possibilità di inviare messaggi o di uscire dalla chat.

Ogni attore Userrappresenta un membro della chat e al momento del suo ingresso viene inserito all’interno dell’attore Register.

Per quanto riguarda, invece, la parte relativa all’interfaccia grafica, è stata realizzata, tramite JavaFX, un’unica finestra che contiene tutte le funzionalità del programma. Quest’ultima è composta da due ListView: una contiene la lista degli attori iscritti alla chat, mentre la seconda viene mostrata al click su un attore della prima lista e mostra la chat relativa a quello specifico attore selezionato. La GUI permette inoltre, una volta selezionato un attore, di inviare messaggi attraverso un bottone “send” e di aggiungere o rimuovere attori, sempre attraverso bottoni; all’aggiunta di un attore comparirà una finestra di dialogo che richiederà all’utente di inserirne il nome, anche fornendone uno di default.

Di seguito vengono presentati ed approfondite le tipologie di attori in cui consiste il programma:

**-User-**

Gli **User** presenti nel sistema si scambiano il token passandolo al successivo elemento fornito dal registro.

Una volta ricevuto il token l’attore va a leggere i messaggi che hanno subito una **stash()** come la richiesta di invio di un messaggio o la volontà di uscire dalla chat e li gestisce.

Inizialmente va ad inviare a tutti gli altri attori presenti tutti i messaggi presenti sul proprio buffer uno ad uno attendendo per ognuno una ricevuta di ritorno da tutti gli attori presenti, solo a questo punto visualizzerà il messaggio sulla propria chat.

Terminata questa operazione verifica se vi è la volontà da parte dell’utente di uscire dalla chat, in caso di positività avvia le operazioni di uscita e solo al termine di esse passa il token all’attore successivo.

Dettaglio sui messaggi che riceve:

* **StartUser**: l'attore lo gestisce solo la prima volta, utilizzato per ottenere il riferimento al Register e al GUIActor.
* **SendMsg**: Riceve l'intenzione da parte dell'attore di inviare un messaggio e lo appoggia nel buffer. Se non si possiede il token verrà chiamata la stash().

Verrà gestito alla ricezione del token.

* **RemActorButtonPressedMsg**: riceve che l'utente è intenzionato ad uscire e viene salvato.

Se non si possiede il token verrà chiamata la stash(). Verrà poi gestita l'uscita solo quando si otterrà il token e tutti i messaggi nel buffer saranno inviati.

* **TakeToken**: riceve il token ed avvia tutte le operazioni dopo aver lanciato la unstashAll(), in ordine:

1. Invio dei messaggi presenti nel Buffer in modalità FIFO.
2. Avvio del procedimento di rilascio del token.
3. Controllo della volontà di uscire, se SI viene gestita.
4. rilascio effettivo del token.

* **AllActors**: viene ricevuto questo messaggio dal Register dopo avergli chiesto gli altri attori registrati. Una volta ottenuti avvia la procedura di invio dei messaggi.

* **SendBroadcastMsg**: verifica che il messaggio da inviare esista,
* se SI:

1. salva il numero di attori da cui dovrà ricevere l'acknowledge, compreso sé stesso, così che funzioni anche se esiste un solo attore.
2. manda a tutti gli altri attori uno ShowMsg con il messaggio da inviare.
3. notifica a sé stesso che i messaggi sono stati inviati con un AcknowledgeMsg.

* se NO:

1. manda un messaggio a sé stesso dicendo di avviare le operazioni.
2. prima del passaggio del token.

* **AcknowledgeMsg:** incrementa il contatore delle ricevute, se è maggiore o uguale al numero di attori che si aspettavano un messaggiosignifica che tutti hanno ricevuto il messaggio e visualizzato, perciò dice al GUIActor che anche sé stesso può visualizzarlo e resetta tutte le variabili.
* **GUIAcknowledgeMsg:** ricevuta di ritorno da parte della GUI, indica che il messaggio è stato renderizzato.Esegue a questo punto il controllo se chi ha inviato il messaggio sia lo stesso attore che lo gestisce,
* se SI: Significa che sono l'attore con il token perciò tento di mandare un altro messaggio presente sul buffer.
* se NO: significa che è stato ricevuto un msg da parte di chi ha il token e viene notificato al mittente che il messaggio è stato visualizzato.
* **ShowMsg:** ricevuto quando si richiede di visualizzare un messaggio, viene girato al GUIActor che lo gestirà e ne notificherà l'esito.
* **TerminateUserOperation:** verifica se c'è la volontà di uscire,
* se SI: chiede al registro di eliminarlo dalla lista di attori e aspetta una sua conferma, in CanExit.
* se NO: avvia le procedure di passaggio del token, mandando un messaggio al Register.
* **CanExit:** permesso da parte del Register di uscire dalla chat.

**-Register-**

Attore che funge, appunto, da registro per tenere traccia di tutti gli User iscritti alla chat.

Dettagli sui messaggi che riceve:

* **AddActorButtonPressedMsg**: aggiunge alla lista degli attori registrati il sender.
* **RemActorButtonPressedMsg**: rimuove dalla lista degli attori registrati il sender e gli notificata tramite il messaggio CanExit che può avviare le operazioni di uscita.
* **GetMeActors**: risponde a questo messaggio inviando al sender un messaggio AllActors nel quale fornisce la lista di attori registrati.

**-GUIActor-**

Quest’ultimo attore viene utilizzato, come suggerisce il nome per gestire l’interazione con l’interfaccia grafica. In particolare, riceve messaggi che vengono inviati direttamente dal View Controller alla pressione di pulsanti o alla selezione di attori nella ListView. Questo attore viene creato all’avvio del sistema (in particolare al caricamento della GUI) e vengono passati ad esso i riferimenti alle ObservableList che vengono usate all’interno dell’interfaccia ed a una Label, usata per l’invio dei messaggi.

Funzionalità per ogni messaggio:

* **SendButtonMessageMsg**: notifica all'attore selezionato la volontà di inviare un messaggio con l’invio di un SendMsg. Questo messaggio viene inviato all’attore al momento della pressione del bottone “Send”.
* **AddActorButtonPressedMsg**: creazione di un attore, con nome specificato dall’utente, e inserimento di quest’ultimo all’interno della chat; successivamente si inviano i messaggi di StartUser all’attore appena creato e AddActorButtonPressedMsg al Register. Infine si aggiornano le ObservableList della GUI per aggiornare quest’ultima. Questo messaggio viene inviato al GUIActor al momento della pressione del bottone “Add Actor”.
* **RemActorButtonPressedMsg**: notifica all'attore selezionato la volontà di uscire dalla chat con l’invio di un RemActorButtonPressedMsg. Questo messaggio viene inviato all’attore al momento della pressione del bottone “Remove Actor”.
* **CanExit**: conferma la possibilità di uscire dalla chat da parte di un attore, permettendo di aggiornare le ObservableList della GUI.
* **GUIShowMsg**: richiesta di visualizzazione di un messaggio da parte di un attore ed aggiunta del messaggio inviato alla ObservableList che rappresenta sulla GUI la chat di quell’attore.
* **ActorSelectedMsg**: riassegna la ObservableList che rappresenta la chat corrente corrente con quella dell'attore selezionato presente nella mappa, e modifica la Label con il nome dell'attore. Questo messaggio viene inviato al GUIActor al momento della selezione di un attore all’interno della ListView degli User presenti.

**LINK AL REPOSITORY**: https://github.com/lucasemprini/Assignemnt03PCD