**Assignment-03**

**1)Actor Programming**

**Progettazione degli attori**

In questo elaborato sono stati implementati i seguenti attori, il cui compito è descritto di seguito:

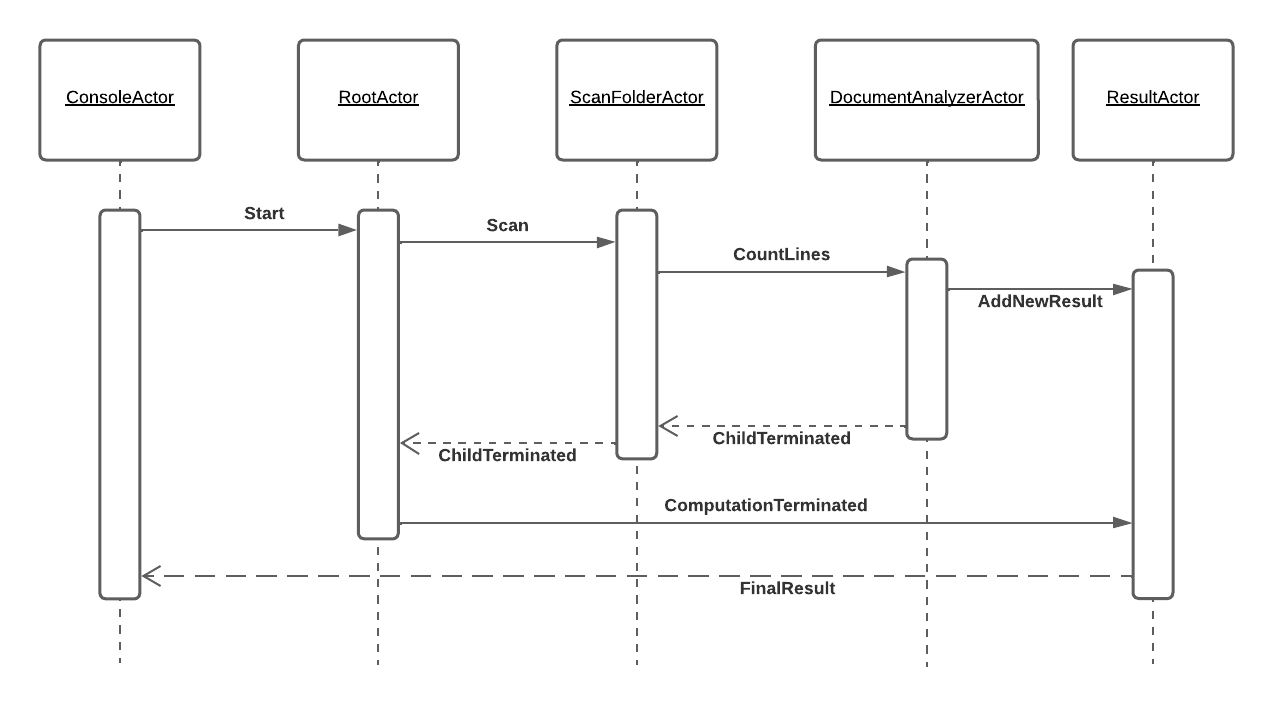
* **RootActor**: Attore che svolge il ruolo di coordinatore e viene creato direttamente dall’ActorSystem, ha il compito di creare a sua volta tre attori principali, il **ConsoleActor/GUIActor**, il **ResultActor** e il primo **ScanFolderActor**.
* **ScanFolderActor**: si occupa dello scanning di una directory, effettuando lo *spawn* di un nuovo **ScanFolderActor** per ogni subfolder e di un nuovo **DocumentAnalyzerActor** per ogni documento.
* **DocumentAnalyzerActor**: ha il compito di contare le righe del documento che gli viene assegnato e di inviare il risultato al **ResultActor**.
* **ResultActor**: riceve i risultati elaborati dal **DocumentAnalyzerActor** e aggiorna periodicamente il ranking e la distribuzione, inviando successivamente il report da mostrare in output al **ConsoleActor/GUIActor**.
* **ConsoleActor**: si occupa di recepire i parametri di input, dare inizio all’esecuzione comunicando con il **RootActor,** ricevere e visualizzare il report finale quando disponibile.
* **GUIActor**: svolge il medesimo compito incapsulato nel **ConsoleActor** ma con la possibilità di fermare l’esecuzione e visualizzare incrementalmente i risultati parziali.

**Scambio di messaggi**

Di seguito viene sintetizzata l’organizzazione dello scambio di messaggi tra gli attori, indicando per ognuno a quali tipologie di comandi reagisce.

* **RootActorCommand**
  + **Start:** comando per avviare l’esecuzione che contiene all’interno le informazioni necessarie per il setup (directory iniziale, numero di files da visualizzare ecc.)
  + **Stop:** Utilizzato per fermare l’esecuzione (viene inviato dal **GUIActor** alla pressione del pulsante Stop).
* **ScanFolderCommand**
  + **Scan:** utilizzato per avviare una scansione, contiene all’interno la directory su cui effettuarla
  + **ChildTerminated:** viene ricevuto ogni volta che un attore figlio termina l’esecuzione e di conseguenza inviato al padre.
* **DocumentAnalyzerCommand**
  + **CountLines:** avvia il conteggio delle righe del file contenuto al suo interno.
* **ResultCommand**
  + **Subscribe:** contiene il riferimento di un nuovo attore che si sottoscrive (utilizzato per avere il riferimento alle view per l’aggiornamento dell’output)
  + **AddNewResult:** ricevuto ogni volta che viene computato un nuovo risultato (che è di conseguenza contenuto al suo interno.
  + **ComputationTerminated:** ricevuto quando l’intera computazione dei files è terminata
* **ViewActorCommand**
  + **IncrementalResult:** contiene al suo interno il risultato incrementale da visualizzare in output.
  + **FinalResult:** contiene il risultato finale da mostrare in output.

**Scenario di esecuzione**



Nello schema di interazione, viene considerato uno scenario di esecuzione in cui (per semplicità di visualizzazione) si considera un progetto con all’interno una directory e un solo file, eseguito in modalità Console.

Sono rappresentati tutti gli attori coinvolti che si scambiano i messaggi indicati nel paragrafo precedente. In un progetto più complicato, il primo **ScanFolderActor** si occupa della creazione di un altro **ScanFolderActor** per ogni subfolder e un **DocumentAnalyzerActor** per ogni documento. Inoltre se consideriamo la modalità GUI, l’unica differenza è l’invio di un messaggio di tipo **IncrementalResult** da parte del **ResultActor** verso il **GUIActor** ogni volta che è presente un nuovo risultato.

**2)Distributed Programming with Asynchronous Message Passing**

In questo progetto sono presenti più host distribuiti che eseguono un’istanza locale dell’applicazione “Pixel Art”. Gli host comunicano tra di loro sfruttando il paradigma a scambio di messaggi asincrono basato su **middleware**, nello specifico utilizzando **Rabbitmq**.

A livello di logica locale, è stato mantenuto l’approccio riportato dal docente aggiungendo qualche piccola modifica.

**Logica locale**

Ogni host possiede le seguenti entità:

* **PixelGrid**: rappresenta la griglia di pixel a livello logico
* **PixelGridView**: incapsula la parte grafica della griglia di pixel
* **VisualiserPanel**: contiene la view della griglia e si occupa di effettuare il re-paint dei pixel
* **BrushManager**: permette di gestire molteplici brush, aggiungendoli, rimuovendoli ecc.

Per quanto riguarda la connessione tra i vari host tramite Rabbitmq necessaria per implementare la distribuzione, è stata creata un’entità (di cui ogni host possiede un’istanza) chiamata **ConnectionHandler**.

**ConnectionHandler**

Il **ConnectionHandler** di un host rappresenta l’entità centrale per la realizzazione della comunicazione. In particolare, si occupa delle seguenti procedure:

* Creazione di una nuova connessione con Rabbitmq
* Creazione del canale di comunicazione dell’host, che fa utilizzo di **exchange** per poter gestire molteplici code
* Registrazione di una coda per ogni possibile messaggio
* Registrazione delle callback per ogni possibile messaggio

Si noti che il ConnectionHanlder di un host ha quindi lo scopo sia di inviare messaggi ad altri host, sia di reagire ai messaggi ricevuti.

Inoltre, per quanto riguarda le code create, alcune hanno lo stesso nome per tutti gli host connessi, in quanto ricevono messaggi che devono necessariamente essere inviati in “broadcast” (come l’aggiornamento della posizione di un brush, o il cambio di un colore). Mentre altre code sono proprietarie dell’host e utilizzano il suo ID per essere identificate univocamente. (Utilizzate per comunicare con uno specifico host, ad esempio per mandare le informazioni di setting iniziali ad un host appena connesso)

**Messaggi**

Tutti i messaggi implementati ereditano da una classe base **Message** che ha come parametro l’ID del mittente del messaggio. Di seguito si riportano tutte le tipologie di messaggio realizzate:

* **JoinMessage**: viene inviato da un nuovo host quando si connette a Rabbitmq e contiene il colore iniziale del brush corrispondente
* **InitMessage**: messaggio di configurazione iniziale che viene inviato ad un host appena connesso da tutti gli host già presenti e contiene la posizione e il colore del brush corrispondente
* **GridMessage**: viene inviato ad un host appena connesso e contiene lo stato attuale della griglia
* **MovedMessage**: viene inviato a tutti gli host connessi quando un host muove il proprio brush e contiene la nuova posizione
* **ColorChangedMessage**: viene inviato a tutti gli host connessi quando un host cambia il colore del proprio brush e contiene il colore aggiornato
* **PaintMessage**: viene inviato a tutti gli host connessi quando un host colora un pixel della griglia e contiene le coordinate del pixel cliccato

**Comportamento dettagliato del sistema**

Quindi a partire da questa struttura si procede a descrivere nel dettaglio il comportamento del sistema. Ogni host possiede la propria **PixelGrid**, il proprio **ConnectionHandler** e il proprio **BrushManager** (che inizialmente contiene solamente il brush locale). Appena un host si connette a Rabbitmq, tutti gli host già connessi ricevono il corrispondente **JoinMessage** e tutti i **BrushManager** locali si occupano di renderizzare il nuovo brush. L’host appena connesso invece riceve un **InitMessage** da ogni host già connesso e ciò gli permette di renderizzare tutti i brush già presenti. Solamente al primo host che fornisce il primo **InitMessage** viene inviato un messaggio per richiedere la griglia e di conseguenza si ottiene in risposta un **GridMessage** che permette di renderizzarne lo stato attuale. Una volta terminata la fase di inizializzazione, utilizzando i listener locali vengono inviati coerentemente gli aggiornamenti tutte le volte che viene mosso il mouse, cliccato un pixel o cambiato colore, in questo modo i **BrushManager** di tutti gli host che ricevono gli aggiornamenti li renderizzano localmente (il funzionamento rimane quindi invariato). Quando un host si disconnette viene inviato un messaggio, alla cui ricezione tutti gli host connessi reagiscono rimuovendo il brush corrispondente.