[Distributed Programming with Asynchronous Message Passing]

Descrizione problema

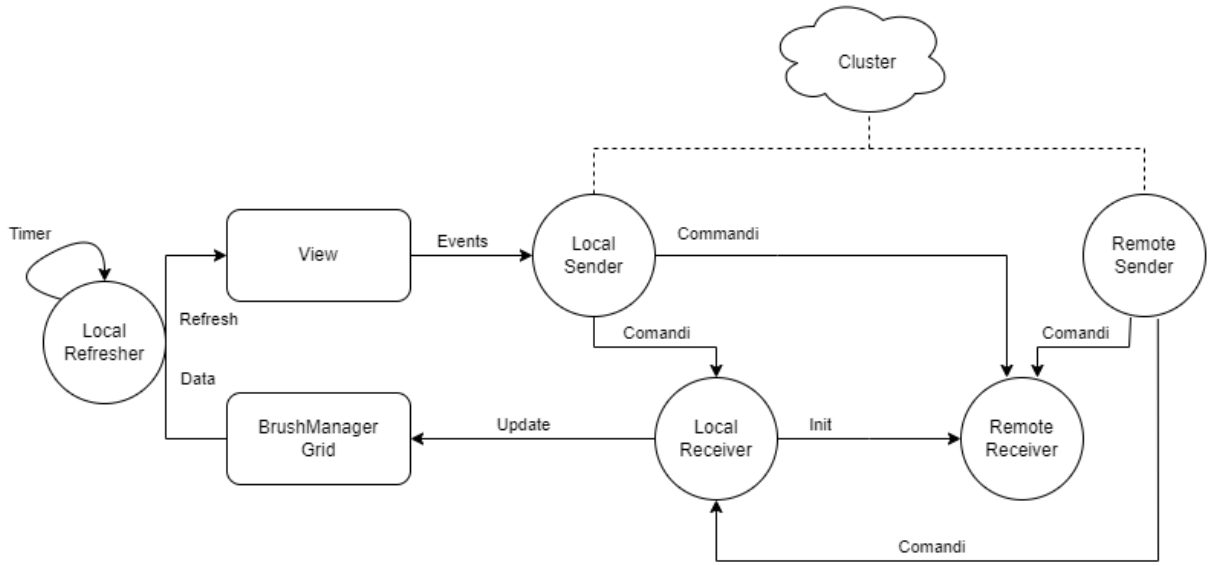
In questo secondo punto si chiedeva di rendere distribuita una pixelArt realizzata per funzionare in locale.
Raccogliendo l'esperienza maturata nello sviluppo del punto uno si sono ancora una volta usati scala e akka. In particolare si è fatto affidamento alla libreria Cluster.

Introduzione

Nella realizzazione del progetto si è cercato di modificare il meno possibile il codice fornito. Gli unici cambiamenti apportati riguardano la gestione dei brush, visto che ora ve ne saranno più di uno. In particolare si è trasformata la lista del brushManager in una mappa<String, Brush>. A parte questo si è deciso di definire un campo booleano isRemote nella classe brush così da poter discriminare il brush locale da tutti gli altri (e poterlo fare apparire più opaco).

Schema

Per la conversione in distribuito basteranno tre attori (di cui solo due davvero necessari). Lo schematico sarà il seguente:



A sinistra è rappresentato il nodo locale con i sender, receiver e refresher locali. A destra si vede, invece, un nodo remoto.
In generale il funzionamento del sistema può essere parzialmente compreso semplicemente guardando lo schema, ma di seguito saranno approfonditi i compiti di ogni singolo attore.

Incapsulamento e gestione risorse

Ogni nodo avrà il proprio brushManager, la propria griglia e la propria view. Non tutti gli attori del nodo, però, potranno accedere alle risorse. In particolare il receiver gestirà griglia e brushManager, mentre il refresher si occuperà del refresh della view che a sua volta avrà il riferimento a griglia, brushManager e Sender. Quest'ultimo non potrà accedere a nessuna risorsa.

Sender

Quando sarà inizializzata la view saranno definiti una serie di event listener in modo tale da far trasmettere al sender messaggi specifici. Questo, a sua volta, in fase di setup, tramite il receptionist, si iscriverà al servizio in cui si registrano tutti i futuri receiver. In questo modo potrà propagare i sopracitati messaggi a tutti i receiver in ascolto (fra cui quello locale, che, almeno dal punto di vista del codice, non godrà di nessun trattamento particolare).
Il sender salverà in uno stato (si applica anche in questo caso l'immutabilità) la lista di receiver attuale. Ogni qual volta il receptionist segnalerà un cambiamento la nuova lista sarà confrontata con quella salvata verificando se la lunghezza è aumentata o diminuita.
In ogni caso sarà mandato un messaggio diverso (per ogni membro eliminato o aggiunto) al receiver locale, cosi che questo possa provvedere a rimuovere il brush associato (nel caso di accorciamento della lista) o a trasmettere il brush e la griglia locale (in modo tale che i nuovi membri del cluster possano allinearsi agli altri).

Receiver

Il receiver ha sia il compito di aggiornare brush, brushManager e griglia che quello di trasmettere le informazioni ai nuovi arrivati. I comandi che può ricevere sono fondamentalmente di due tipi:

1. La prima categoria riguarda i messaggi relativi ad uno dei tre eventi sulla view: cambio di colore del brush, movimento del mouse e click su una casella. Il receiver non sa se un evento è stato scatenato nella view locale o in una remota, ma si limita a usare i dati fornitigli (id del brush, coordinate, colore) per applicare i dovuti cambiamenti.
2. La seconda categoria è legata ai messaggi di cluster, che possono essere:
    a. sendInit: questo tipo di messaggio può arrivare solo dal sender locale, e contiene l'indirizzo del receiver a cui mandare un messaggio di init.
    b. Init: contiene griglia, brush e id di un certo nodo. Serve al receiver per aggiornare la mappa del brush manager e la griglia (colorando tutte le caselle che in quella locale sono bianche, mentre in quella remota no).
    c. removeBrush: il nome è autoesplicativo.

In fase di inizializzazione il receiver instanzierà il brush locale, che sarà inserito nella mappa del brushManager come qualunque altro brush.

Refresher

Come dice il nome si occupa del semplice refresh della view. Ha un behavior with timer, che ogni 10 millesimi di secondo porta l'attore ad autoinviarsi un messaggio. Alla ricezione di questo, sarà semplicemente chiamato il metodo refresh sulla view e si reinizializzerà il behavior.

Connessione

Come era richiesto dalla consegna il cluster userà una connessione di tipo peer to peer. In generale basterebbe conoscere l'indirizzo di un qualsiasi attore del cluster per accedervi, ma in pratica per facilitare le cose si applica qualcosa di più centralizzato.
All'avvio del main si proverà a instanziare un actorSystem ad un indirizzo fisso: se non è presente nessun altro actor system si creerà il cluster, se no nella gestione dell'eccezione che sarà lanciata si sceglierà randomicamente un nuovo indirizzo (anche più volte se è necessario) e ci si collegherà al cluster esistente.

Consistenza

Una volta che un nuovo nodo viene accettato, gli altri membri del cluster trasmetteranno il proprio id, la propria griglia e il proprio brush locale così che il nuovo arrivato possa mettersi in pari.
In particolare la griglia sarà aggiornata più volte (una per ogni altro membro) colorando ogni volta tutte le caselle che in locale sono bianche e in remoto no (se, invece, quella locale è colorata, anche se il colore di quella remota è differente non sarà aggiornata).
Questa è l'unica vera forma di sincronismo che è stata sviluppata (togliendo la propagazione degli eventi via messaggi). Se per qualche motivo uno o più nodi dovessero disallinearsi rispetto agli altri, non si attiverebbe nessuna forma di protezione.
Per evitare di far esplodere la complessità del problema, infatti, si parte dal presupposto che tutti i messaggi siano sempre ricevuti e che l'ordine di ricevimento sia più o meno lo stesso per tutti. Alla fine, infatti, bisogna considerare che l'unico caso che davvero potrebbe creare problemi è quello in cui messaggi relativi a eventi di click sulla stessa cella della griglia siano ricevuti in ordine diverso (cosa che comunque dovrebbe essere abbastanza rara).

Id

Nella gestione dei vari brush riveste un ruolo fondamentale l'id. Questo è ottenuto prendendo la parte numerica dell'actoref, che essendo relativo all'indirizzo di cluster è per forza unico (considerando che tutti i receiver avranno lo stesso nome e l'actoref è composto da nome#numero).

Messaggi

Per quanto riguarda i messaggi si è fatta una scelta discutibile: invece di realizzare un messaggio ad hoc per ogni tipo di richiesta, magari imparentandolo con una classe root, si è

creata una classe unica con una serie di campi opzionali che saranno riempiti o no in base al tipo di comando (che sarà esplicitato nell'unico campo non opzionale).
In questo modo si evita di creare decine di classi diverse, anche se la lettura dei singoli messaggi ne risente.

Classe Methods

Ogni attore è organizzato con solo due metodi (uno nel caso del refresher): l'apply per la definizione della fase di setup e un metodo specifico per configurare il comportamento al ricevimento dei messaggi. Tutti gli altri metodi, come quello per la connessione, l'inizializzazione della view, l'aggiornamento della griglia e l'estrazione dell'id dall'actorRef sono spostati in una classe specifica che prende il nome di Methods.
Lo scopo è ridurre al minimo il codice all'interno delle classi attori.