marco.canducci@studio.unibo
07/06/2018

Assignment #3

Gruppo di lavoro

Il progetto è stato sviluppato a quattro mani, dal sottoscritto Canducci Marco (matr. 833069) e dal compagno Schiavi Daniele (matr. 843692). Così come nei precedenti assignment, anche in questo caso non c'è stata alcuna separazione dei compiti tra i componenti del gruppo: si è proseguito nello sviluppo ragionando insieme su ogni scelta logica ed implementativa considerata rilevante.

Esercizio 1: Game of Life

Premessa

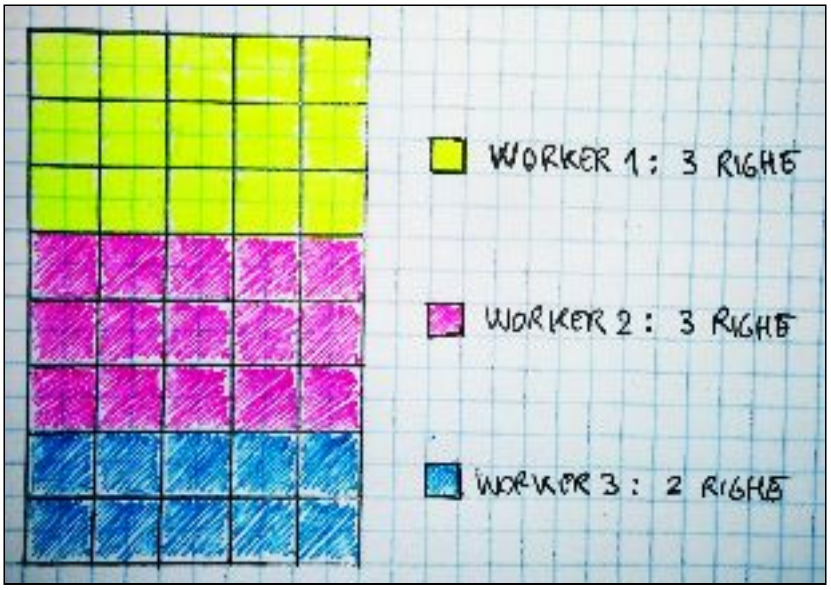
Prima di introdurre il paradigma ad attori si è deciso di ripulire dai dettagli superflui la soluzione (quasi *over-engineered*) proposta nel primo assignment, andando inoltre ad integrare in essa alcune idee presenti nella soluzione fornita dal docente poiché ritenute particolarmente interessanti. Solo dopo aver ottenuto una base solida, minimale e soddisfacente che funzionasse con semplici meccanismi di sincronizzazione a semafori, si è iniziato a pensare a come adattarla allo scambio di messaggi tra attori.

Analisi del problema

- Seppure in questo caso si lavori in locale, si vogliono comunque rispettare le linee guida della programmazione ad attori: non verrà quindi sfruttata memoria condivisa, ed i messaggi scambiati conterranno solo oggetti immutabili.
- Il passaggio da una soluzione gestita con semafori ad una basata sullo scambio di messaggi risulta particolarmente agevole. Molte delle entità viste precedentemente come *Thread* possono ora essere interpretate come *Attori*. Alcune delle *release* che venivano effettuate su semafori condivisi tra più *thread* possono ora essere sostituite dall'invio di opportuni messaggi. Seguendo lo stesso ragionamento (un po' forzato), alle *acquire* possono essere associati concettualmente degli handler in risposta al giusto tipo di messaggio.
- La logica dietro alla soluzione proposta rispecchia quasi totalmente quella illustrata in modo estensivo nella relazione relativa al primo assignment. Ogni volta che viene richiesto un mondo al *master*, questi lo fornisce alla GUI (se pronto) ed inizia il calcolo del mondo successivo demandando il lavoro ai propri *worker*.
- Si ritiene consigliabile utilizzare componenti e librerie grafiche (pensate per l'interazione in puro stile OOP) incapsulando la loro gestione dentro ad uno o più attori. Questi avrà/avranno il compito di fare da traduttore/i tra i messaggi scambiati con altri attori e l'interfaccia grafica.

Suddivisione del lavoro tra i Workers

- Il mondo è diviso per righe.
- Ogni worker è responsabile del calcolo dello stato successivo di una parte di mondo.
- Il resto della divisione *nRows* / *nWorkers* è quindi distribuito sui worker come nell'esempio riportato di seguito.



Dettagli implementativi

- Si è fornita la possibilità di creare un'istanza di *World* immutabile a partire da un *array* di matrici. Ogni matrice rappresenta (in ordine) il sotto-mondo calcolato da ogni *worker*. Ad ogni turno il *master* (GameActor) si occupa di raccogliere i nuovi sotto-mondi e di concatenarli opportunamente per comporre il nuovo mondo da visualizzare.
- La gestione del *framerate cap* (a 60Hz) è stata implementata grazie alla funzionalità *scheduleOnce()* fornita dal dispatcher di Akka.
- Per l'interazione tra il GameActor ed il Controller di JavaFX si è utilizzato un coordinatore denominato VisualizerActor: il suo compito è quello di tradurre eventi generati dalla GUI in messaggi in uscita e, similmente, di aggiornare l'interfaccia grafica in risposta alla ricezione di nuovi messaggi.

Prove di performance

# Workers	Tempo (ms)	Speedup	Efficienza
1	319	-	-
2	167	1.91	0.96
3	138	2.31	0.77
4	109	2.93	0.73
5	93	3.43	0.69
6	77	4.14	0.69
7	70	4.56	0.65
8	67	4.76	0.60

Risultati ottenuti nel il calcolo di una matrice di dimensione 5000 x 5000 utilizzando un processore Intel mobile, dotato di 4 core fisici ed 8 logici.

Possibili miglioramenti

- Piuttosto che far inviare dal *master* tutto il mondo corrente ai suoi *worker*, si potrebbe spedire la più piccola porzione possibile di mondo utile per il calcolo, che corrisponde al sotto-mondo d'interesse più la riga precedente e quella successiva. Quindi, nel caso ad esempio un *worker* dovesse calcolare il mondo successivo dalla riga 10 alla riga 15, sarebbe sufficiente inviargli ad ogni turno il sotto-mondo dalla riga 9 alla riga 16 o, addirittura, le sole righe 9 e 16, poiché tutte quelle centrali potrebbe essere ricordate dal turno precedente.
- Si sarebbe potuta realizzare un'ulteriore versione del progetto per il caso sequenziale (N_WORKERS = 1), in modo da evitare sia la creazione dell'attore *worker*, sia tutto lo scambio di messaggi tra *master* e *worker*. Avendo però letto sulla documentazione di Akka di come venga ottimizzato lo scambio di messaggio in locale (evitando ogni volta di serializzare e deserializzare i messaggi) si è ritenuto ragionevole non aggiungere altro codice e considerare quindi il caso sequenziale come un sotto-caso particolare di quello parallelo.

Esercizio 2: Distributed Chat

Analisi del problema

- È necessario realizzare una *chat room* dove sia rispettato il concetto di *Atomic Broadcast* (detto anche *Total Order Broadcast*).
- Essendo il problema di natura Broadcast, nel caso si riuscissero a garantire:
  1) Ordinamento FIFO sui messaggi inviati dallo stesso mittente
  2) Ordinamento totale totale dei messaggi scambiati tra i vari partecipanti
  Allora si garantirebbe automaticamente anche un ordinamento causale tra tutti gli attori.
- È stato osservato in letteratura come **il problema dell'*Atomic Broadcast* sia equivalente (e quindi riconducibile) al problema del *Consensus*.**
- **Essendo esplicitamente richiesta la totale decentralizzazione dello scambio di messaggi, si sono escluse a priori soluzioni che prevedessero l'elezione di un *leader*.**
- L'unico componente diverso dai partecipanti della chat (chiamati *chatters*) è un attore "Register" noto a priori, il cui ruolo è semplicemente quello di aggiornare e fornire su richiesta l'elenco completo dei *chatters*. Il Register non deve essere coinvolto durante lo scambio dei messaggi: il suo unico utilizzo sarà in fase di login e di logout.
- Per sincronizzarsi con tutti gli altri *chatters* sul prossimo messaggio da visualizzare senza scomodare un *leader*, si introduce allora il **concetto di votazione** (è così che ci si riconduce quindi ad un problema di *Consensus*): ad ogni "round" di votazione tutti i *chatter* inviano a tutti gli altri partecipanti il proprio voto. Per "voto" si intende il prossimo messaggio che a loro parere deve essere visualizzato (il quale coincide con il primo che hanno ricevuto dopo la votazione del round precedente).
- Dopo aver votato, sarà necessario che ogni *chatter* cambi il proprio comportamento interno, mettendosi in attesa dei voti degli altri partecipanti.
- Quando il numero di voti per un determinato messaggio dovesse superare la metà dei votanti, si potrà già eleggere il messaggio votato come quello da visualizzare (e di conseguenza scartare tutti i voti che arriveranno relativi alla votazione già conclusa).

- In un turno T di votazioni è possibile che:
  1) Vengano ricevuti voti obsoleti, relativi a messaggi già eletti **-> Da scartare**
  2) Vengano ricevuti voti relativi al turno T+1 **-> Da *stashare*** (ad esempio nel caso qualcuno abbia già terminato la votazione T ed abbia già inviato il suo voto relativo al turno successivo!!)
- È quindi palese la necessità di allegare ad ogni messaggio di voto un contatore (denominato ***votePulse***) che indichi a quale round il voto faccia riferimento.

Login e Logout dinamici

Per effettuare il login ed il logout dinamicamente senza introdurre situazioni di errore o di incongruenza tra le liste locali di ogni *chatter*, si può sfruttare il punto di sincronizzazione del voto: ogni *chatter* sarà ufficialmente connesso (*Logged in*) solamente quando il suo messaggio di richiesta sarà stato votato dalla maggioranza dei *chatter* già connessi. E' inoltre fondamentale che nel momento in cui un nuovo partecipante entra in chat, gli venga fornito il numero di votazione in corso (*votePulse*), in modo tale che egli possa intervenire correttamente in tutti i round di votazioni a seguire.

Facoltativo: Sezione critica

Anche per quanto riguarda la sezione critica conviene sfruttare il meccanismo di voto già implementato, in modo da sincronizzare per tutti i processi l'entrata in sezione critica nello stesso "momento logico". Essendo impossibile in un sistema distribuito garantire l'ingresso in sezione critica in contemporanea considerando il tempo fisico, almeno si garantisce che tutti i processi vi entrino dopo aver visualizzato lo stesso messaggio. Il messaggio di entrata (ed uscita) dalla sezione critica viene quindi trattato dal sistema come un qualunque altro messaggio; sarà compito del ricevente capirne la natura e prepararsi ad un nuovo tipo di comportamento (nel quale saranno accettati tutti e solo i messaggi di colui in sezione critica).

Mentre qualcun altro risulta in sezione critica, vengono ovviamente scartati tutti i messaggi che l'utente cercherà di inviare interagendo con la GUI. Si decide però (come scelta di design) di *stashare* tutti i messaggi ricevuti dagli altri *chatters* (a parte quello dentro sezione critica, i cui messaggi verranno direttamente visualizzati) poiché tali messaggi erano sicuramente stati da inviati prima di entrare in sezione critica. Tali messaggi verranno tutti *un-stashati* in seguito, in uscita dalla sezione critica. Per tutti i dettagli implementativi si faccia riferimento al comportamento descritto da *inCriticalSection* dentro alla classe Chatter.java.

Considerazioni tecniche

- Nello sviluppo con Akka, è risultata particolarmente utile la garanzia sull'ordine di arrivo dei messaggi per ogni coppia mittente/destinatario (***message ordering per sender–receiver pair***). Questo ha permesso di evitare controlli aggiuntivi (relativi al clock logico) sull'ordine di ricezione dei messaggi da parte dello stesso mittente (FIFO).

- Altra cosa ritenuta particolarmente utile ed utilizzata in modo intensivo è stata la possibilità di *stashare* ed *unstashare* a propria discrezione i messaggi in arrivo. Al crescere della complessità/dimensione del codice sorgente ho però trovato sempre più difficile decidere quando e dove fosse davvero necessaria una unstashAll().

- Infine, lavorando con Akka si è apprezzata anche la semplicità con la quale fosse possibile cambiare il comportamento interno di ogni attore. Inoltre, si è notato come, al crescere della complessità, una serie di *become* mirate sia più semplice da gestire rispetto all'utilizzo dell'accoppiata *become*/*unbecome*: separando e nominando in modo opportuno i vari stati (comportamenti) in cui un attore può trovarsi, è possibile comprendere molto meglio quello che si sta facendo, anche trovandosi in un paradigma di programmazione non del tutto familiare. Si vedano ad esempio i diversi comportamenti definiti nel costruttore di *Chatter.java*:
  - > *loggedOutBehavior*
  - > defaultBehavior
  - > waitForVotesThenVisualize
  - > inCriticalSection

  Così facendo, è stato possibile interpretare un attore come una sorta di **macchina a stati**, in grado di gestire determinati messaggi (anche in modo diverso) a seconda del suo stato.

Testing

- Sono risultate particolarmente complesse le fasi di debugging e di testing.
- Per testare l'ordinamento totale nella ricezione dei messaggi da parte di tutti i partecipanti della chat si è simulato l'invio di una grande quantità di messaggi da parte di diversi *chatters*. Per ognuno di essi si sono quindi riportati su file di log tutti i messaggi visualizzati sulla GUI (nello stesso identico ordine), dopodiché ci si è assicurati che tutti i file di log fossero identici tramite comando *diff*.
- Per testare login e logout dinamici si è ripetuto un test simile a quello precedente andando però anche a connettere e disconnettere manualmente alcuni partecipanti alla conversazione durante lo scambio intensivo di messaggi.
- Il test riguardante la parte facoltativa sulla sezione critica è stato svolto invece interamente a mano, verificando anche l'uscita dalla CS tramite *timeout*.

Possibili miglioramenti

- Si potrebbe cercare di rendere il sistema più resistente ai guasti, prevedendo ad esempio più round per ogni singola votazione: in particolare si potrebbero prevedere **f+1** rounds per ogni votazione. In ogni round si potrebbe quindi attendere la ricezione dei voti degli altri partecipanti per un tempo massimo tMax (**f** rappresenta il massimo numero di fallimenti ammessi).