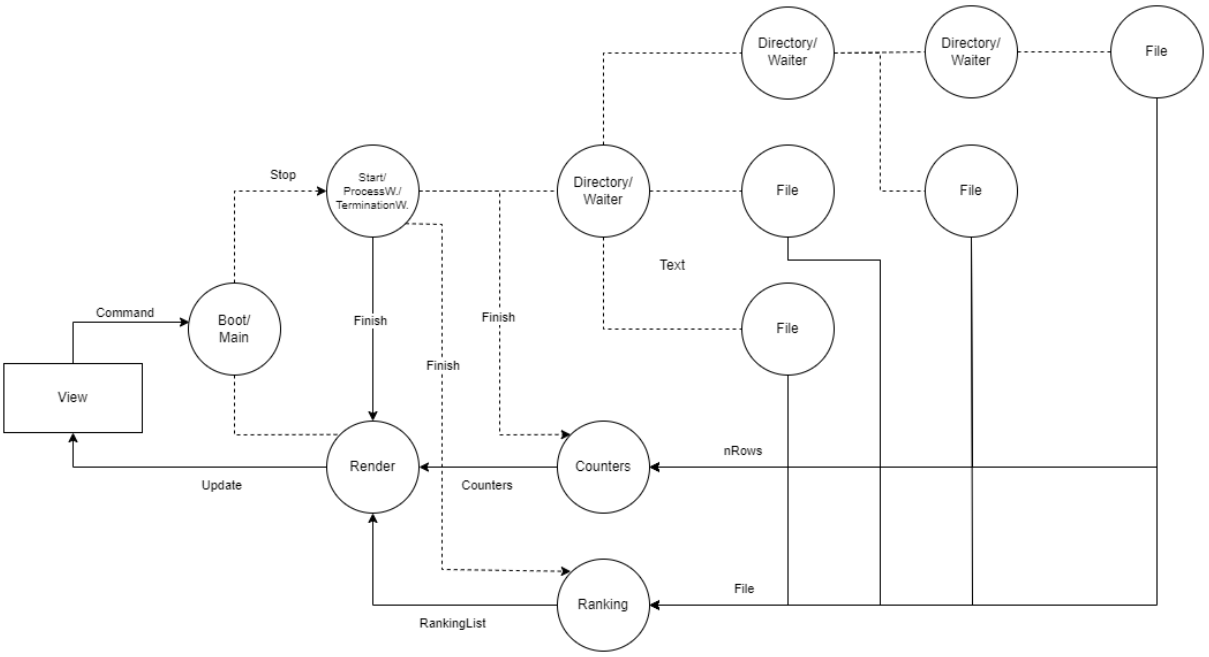
[Actor programming]

Presentazione problema

In questo assignment si chiede di proporre una soluzione con attori del problema affrontato nelle precedenti due consegne: il calcolo della classifica e della distribuzione dei file di una data directory d.
Per fare questo si è utilizzato il toolkit akka e il linguaggio scala.

Introduzione

Nella definizione del progetto di è partiti dall'idea che ad ogni attore si dovessero assegnare dei behaviors il più semplici possibile, anche se questo significava dover creare un grande numero di istanze (dalla documentazione si evince che gli attori sono così leggeri che se ne possono spawnare a milioni). Lo schema che ne è derivato è il seguente:



Ogni freccia indica lo scambio di messaggi verso un attore, mentre le linee tratteggiate definiscono un legame di parentela (a sinistra si hanno i genitori e a destra i figli). Infine una freccia tratteggiata indica un flusso di messaggi verso un proprio figlio (in pratica serve solo a non dover disegnare sia la freccia che la linea tratteggiata).

Strutture ereditate

In questo assignment si ripresentano alcune strutture che hanno caratterizzato i vecchi progetti: la view, la rankingList e l'array di contatori. Ognuna di queste è stata tradotta da

java a scala apportando le modifiche necessarie per sfruttare al meglio le potenzialità offerte dal nuovo linguaggio.
Rimane anche la classe comune “Common” con all’interno l’insieme delle costanti condivise dai vari attori.
Non è più presente, invece, il dataWrapper con il conseguente spostamento dei suoi metodi in più classi (a testimonianza della volontà di decentralizzare il più possibile il codice). In particolare ranking e counters ora avranno un metodo apposito per la generazione del testo di aggiornamento della view, che ricorda un po ' il toString..

Actors

All’avvio della view sarà inizializzato l’actor system, definendo un primo attore che prenderà il nome di MainActor. Questo in un primo momento (fase di setup o boot) chiamerà una spawn per generare il renderActor, e successivamente si metterà in attesa di ricevere comandi da parte della view (che come nei progetti precedenti potranno essere di tre tipi: start, stop, exit).
In caso di start inizierà un processo di generazione a catena, che vedrà inizialmente la generazione di uno startActor che si occuperà di tutto il futuro processo. Il suo lavoro si dividerà in tre fasi:

1. Inizializzare gli attori chiave, ovvero il directoryActor per spazzare la directory d, il countersActor per la definizione degli intervalli e il rankingActor per la generazione della classifica.
2. Si metterà in attesa della fine del processo, curandone poi la segnalazione a renderActor (per la stampa del tempo totale impiegato), rankingActor e countersActor.
3. Aspetterà che rankingActor e countersActor abbiano finito, per terminare lui stesso.

Il primo directoryActor nello spazzare la directory genererà un nuovo fileActor/directoryActor per ogni file (.java)/cartella (non vuota) individuata. Successivamente i directoryActor generati faranno lo stesso creando una sorta di albero, che ricalca quella di file stessi.
Ogni fileActor si limiterà a contare le righe del file assegnatoli e a mandare le informazioni (con due messaggi diversi) a rankingActor e counterActor. Questi ultimi, a loro volta aggiornano le proprie strutture dati e, se è passato un certo intervallo di tempo, invieranno un messaggio con i dati per aggiornare la View al renderActor.
Dopo l’invio dei due messaggi i fileActor termineranno mandando un segnale al proprio padre (che sarà un directoryActor).
I directoryActor dopo aver finito di spazzare la propria directory andranno in uno stato di waiting, in cui attenderanno che tutti i figli avranno terminato. Quando ciò accadrà termineranno a loro volta.
Si innesca così una reazione a catena che terminerà quando il directoryActor originale (quello a cui era stata passata la directory d) assumerà un Behaviors.stopped segnalando la fine del processo allo startActor. Questo come si è detto prima avviserà rankingActor e countersActor che invieranno al renderActor le ultime informazioni e termineranno a loro volta. In questa fase ci si affida all’idea che avendo già tutti i fileActors finito, ranking e counters actor dovrebbero aver ricevuto tutti i messaggi con i dati maturati durante il processo. Quindi all’arrivo del messaggio dello startActor, ci si aspetta che il conseguente aggiornamento tenga conto di tutti i file (anche se in generale l’ordinamento temporale dei messaggi ricevuti non dovrebbe essere garantito per attori diversi).

In caso di stop o di chiusura del programma si sfrutta la funzionalità per cui stoppando un attore padre, si fermano anche tutti i figli: se si preme il tasto stop, sarà mandato un messaggio al mainActor, che a sua volta avviserà lo startActor di fermarsi (bloccando di fatto il processo). Quando si preme il tasto exit, invece, a terminare sarà proprio il mainActor.

Immutabilità e concorrenza

Nello sviluppo del progetto si è deciso di adottare scala, e con questo la filosofia dell'immutabilità. Ne behaviors, ne strutture dati aggiornano il proprio stato, ma piuttosto inizializzano nuovi oggetti.
Dal punto di vista della concorrenza ciò permette di trasmettere al renderActor la rankingList e l'array di counters senza doversi preoccupare di corse critiche (non esistono side effects). Di conseguenza non esistono né monitors né altre forme di protezione thread-safe.

Limitazioni

A differenza dei precedenti assignment non viene considerato il caso in cui non vengano trovati file .java nella directory d. L'implementazione di questa funzionalità avrebbe comportato delle complicazioni evitabili, legate alla condivisione della risorsa rankingList. Il mainActor, invece, garantisce il controllo della validità e della non vuotezza della directory d prima del lancio del processo.

Receptionist

Anche se poteva essere usato per non dover propagare gli actorRef di rankingActor e countersActor, si è preferito evitare visto che dovrebbe essere più adatto in quei casi in cui non si ha la sicurezza di dover mandare un certo messaggio o a chi mandarlo (perchè non si conosce chi svolge un dato compito). In questo caso i messaggi, salvo errori, saranno sempre trasmessi, e si sa sempre a chi devono essere inviati.

Messaggi e nomi attori

Non sono stati definiti tipi di messaggi specifici perché si usano sempre tipi base (soprattutto stringhe). L'unico caso degno di nota è quello di rankingActor e counterActor che usano opzionali per distinguere i messaggi contenenti i dati elaborati dai fileActor da quelli di terminazione trasmessi dallo startActor.
Per quanto riguarda i nomi dati agli attori, invece, dato il grande numero di entità create ad ogni run, si è preferito usare spawnAnonimus, fatta eccezione per il BootActor e il RenderActor che rimarranno unici per tutta la durata dell'esecuzione.

Performance

Purtroppo i recenti aggiornamenti windows 11 hanno rallentato notevolmente le operazioni di I/O e di conseguenza si sono dovute calcolare nuovamente le prestazioni di ogni tecnologia per poter fare i dovuti confronti. I risultati sono stati i seguenti:

1. Assignment 1: circa 1200ms per operazione;
2. Task (fork-join executor): circa 1350ms per operazione;
3. Akka: circa 1500ms per operazione;
4. Virtual Thread: circa 1550ms per operazione;
5. Vertx: circa 1700ms;
6. RxJava: circa 4600ms;
7. Sequenziale: circa 5700ms.

I test sono stati fatti nella solita directory da 21mila file. Come si nota i tempi generali sono aumentati notevolmente, più che raddoppiati in generale. L'unica eccezione è l'assignment 1 che ha risentito meno di tutti del rallentamento delle memorie. Per quanto riguarda Akka si posiziona a pari merito (circa) dei virtual thread,risultato più che soddisfacente e che dimostra la bontà della tecnologia.