**PROGETTO DI LINGUAGGI DI PROGRAMMAZIONE AVANZATI**

**Simulazione di un aeroporto in Scala**

*Il problema*

Si vuole creare una rete di aeroporti. Gli aerei partono da un aeroporto e atterrano in un altro (diverso da quello di partenza).

Gli aerei hanno:

1. un nome
2. aeroporto di partenza
3. aeroporto di arrivo

Ogni aeroporto ha una sola pista. La pista dev'essere usata dagli aerei sia per decollare che per atterrare. Per far si che non avvengano scontri, ogni aeroporto deve un sistema che gestisca:

1. le code degli aerei in partenza e degli aerei in arrivo
2. l'occupazione della pista così che non si verifichino scontri

Dev'essere sempre possibile che allo stesso tempo avvenga che un aereo si accodi in partenza, un secondo si accodi in arrivo e un terzo occupi la pista.

L'ordine con cui gli aerei occupano la pista è dato da una tabella oraria.

Il sistema deve anche essere in grado di gestire eventuali ritardi degli aerei, senza che questi blocchino la prosecuzione del normale traffico aereo.

*La soluzione*

Scala è un linguaggio molto flessibile e consente di implementare due diversi tipi di soluzione: una che segue la programmazione imperativa e la gestione della concorrenza con i Thread java, un'altra che invece segue la programmazione funzionale e la gestione della concorrenza utilizzando gli akka Actors.

*Soluzione Imperativa*

Un linguaggio imperativo descrivono la computazione in termini di statement che cambiano lo stato del programma. Esiste quindi un concetto di stato, dal quale consegue l'uso di strutture dati modificabili (1) e di assegnamenti (2). Da ciò consegue che occorre prestare attenzione al controllo della memoria e che tali tipi di linguaggi consentono l'introduzione di alias e alle funzioni di fare side effect. Altra caratteristica importante è l'uso di loops, come for, do while, while, e di conditional branches (if-else), piuttosto che di ricorsione e l'uso di funzioni first-order.

Per meglio adattare Scala a questo tipo di programmazione di è utilizzato:

1. strutture dati mutable (1) e var per permettere gli assegnamenti (2)
2. non si è utilizzata la ricorsione, ma solo loop quali while e for, oltre a strutture condizionali come if/else
3. i metodi implementati non prendono funzioni come argomenti, ne le ritornano come risultato della computazione
4. uso di eager evaluation in ogni circostanza
5. concesso l'uso di aliasing e alle funzioni di fare side effect

Per gestire la concorrenza si sono utilizzati i Thread java. L'utilizzo del modello ha Thread ha comportato il dover affrontare il problema della sincronizzazione sulla memoria condivisa. In ogni punto del programma si deve quindi ragionare su

* quali dati si stanno modificare
* se i dati a cui si sta accedendo possono essere modificati concorrentemente da altri thread
* quali lock si possiedono
* di quali lock si ha bisogno
* si è causato deadlock, data race o priority inversion

La memoria condivisa è più efficiente quando c'è un controllo centralizzato.

Infine, un thread è pesante, in quanto è mappato su un thread del s.o. che consuma memoria e i context switch tra thread sono costosi.

Tenendo conto di queste caratteristiche si è modellato il problema seguendo un approccio centralizzato come segue. Ogni aeroporto è stato caratterizzato dal possedere tre aree di memoria condivisa (le code delle partenze e arrivi e la pista) e un controllore che, a partire da una tabella oraria, riserva la pista per una partenza o un arrivo.

La tabella oraria è stata implementata come un array di “A” e “P”, dove “P” indica che è prevista una partenza (quindi la pista va riservata per il primo aereo tra quelli in coda che attendono di partire), “A” indica che p previsto un arrivo (quindi la pista va riservata per il primo aereo tra quelli in coda che attendono di atterrare)

1. Controllore:
2. int index = 0
3. while (index < tabella\_oraria.size)
4. if (tabella\_oraria[index] == “P”) // è una partenza
5. aereo = null
6. synchronize(coda\_partenze)
7. if(! coda\_partenze.isEmpty)
8. aereo = coda\_partenze.pop()
9. if(aereo != null)
10. syncronize(pista)
11. print “aereo partito”
12. synchronize(coda\_arrivi)
13. coda\_arrivi.push(aereo)
14. coda\_arrivi.notifyAll()
15. else
16. new Waiting(coda\_partenze, decolla).start() //aereo è in ritardo
17. else // è un arrivo
18. aereo = null
19. synchronize(coda\_arrivi)
20. if(! coda\_arrivi.isEmpty)
21. aereo = coda\_arrivi.pop()
22. if(aereo != null)
23. syncronize(pista)
24. print “aereo arrivato”
25. else
26. new Waiting(coda\_arrivi, atterraggio).start() //aereo è in ritardo

Quando un aereo è in ritardo (cioè quando la coda è vuota), si crea un nuovo thread che si occupa di aspettare che l'aereo arrivi e poi lo fa atterrare/decollare non appena la pista è libera.

1. Waiting:
2. aereo = null
3. synchronize(coda)
4. while(coda.isEmpty)
5. coda.wait()
6. aereo = coda.pop()
7. if atterraggio
8. synchronize(pista)
9. print “aereo atterra”
10. else
11. synchronize(pista)
12. print “aereo decolla”

Ogni aeroporto si occupa di comunicare alla torre che i suoi aerei sono pronti a partire, inserendoli nella coda\_partenze (e facendo poi notifyAll su di essa).

*La soluzione funzionale*

Un linguaggio funzionale considera la computazione come la valutazione di funzioni matematiche, evitando stati e strutture dati mutabili. In tali linguaggi quindi non ci sono assegnamenti, le strutture dati sono immutabili, persistenti, le funzioni non fanno side effect e sono higher order, la ricorsione è la principale struttura di controllo, è concessa la lazy evaluation.

Scala consente di modellare il problema di partenza anche per poterlo risolvere con un approccio più funzionale.

Si è quindi cercato di usare il più possibile tutte quelle funzionalità che Scala mette a disposizione per una miglior programmazione funzionale possibile.

Tra le caratteristiche più rilevanti di Scala che sono presenti nel progetto si elencano le seguenti:

* uso dei singleton object (contiene il main)
* si è cercato di usare soprattutto dati immutabili (strutture dati immutable e val), limitanto il più possibile l'uso di dati mutabili
* i tipi sono inferiti
* metodi senza side effect per garantire referential transparency
* uso di funzioni anonime e lazy evaluation dove necessarie
* uso di higher order function
* uso della ricorsione
* molto uso del costrutto di pattern matching al posto di conditional branching (anche case class per i messaggi degli attori)
* uso di metodi implicit
* uso del trait mixin per aggiungere funzionalità (modifiche impilabili, creati al volo)
* variabili di classe implementate come getter e setter
* uso di monadic methods come map, filter
* uso della flessibilità sintatica che scala offre (no ;, ogni metodo usabile cpome operatore infisso, nessuna distinzione tra statement e espressione perché tutto è funzione, no parentesi ad eccezione della convezione sui side effect,...)
* uso dei placeholder \_
* uso del for nelle due sue accezioni più particolari: foreach e for...yield
* specializzazione di funzioni
* uso di funzioni parziali

La parte di concorrenza in questo caso, invece, viene implementata utilizzando gli attori; si tratta quindi di modello message passing.

È più scalabile su architetture diverse del modello a Thread, oltre che più leggero, in quanto la computazione dell'attore è un Task (poi c'è uno Scheduler che raccoglie i Task e li assegna a un thread worker). Un attore risulta quindi separato dal thread/processo che lo esegue e adatto a parallelismo massivo.

Le decisioni degli attori sono prese sulla base dell'info locale, minimizzando cosi la dipendenza da uno stato. Data-flow e control-flow sono inseparabili, garantenndo così la necessità di sincronizzazioni e massimizzando la possibilità di concorrenza.

Risulta essere un sistema più adatto alla programmazione distribuita.

Non ci sono lock quindi per evitare side effect ogni attore deve inviare solo messaggi immutabili. Ogni attore (receive) processa un messaggio alla volta, ma non garantisce in che ordine. Nel caso di questo progetto, dove abbiamo usato la mailbox FIFO di default degli akka actors, vale la seguente regola:

for a given pair of actors, messages sent from the first to the second will not be received out-of-order

Nel nostro problema abbiamo quindi utilizzato quattro attori:

1. un gestore della coda degli aerei in partenza
2. un gestore della coda degli aerei in arrivo
3. un gestore degli aerei in ritardo
4. la pista

1 e 2 contengono uno stato locale:

* coda degli aerei
* numero di ritardi

1 e 2 consentono di non leggere e scrivere sulle cose contemporaneamente

3 consente di gestire i ritardi senza interrompere la lettura della tabella oraria

4 consente che non ci siano scontri in pista

la tabella oraria di ogni aeroporto è letta in un future:

per ogni “A” o “P”, il future chiede (con ask) all'attore 1 o 2 di ritornargli un aereo da far par partire o decollare. Se gli viene ritornato un aereo, allora questo è spedito alla Pista con un messaggio, in modo da assegnargliela non appena sarà libera. Se la coda era vuota, allora localmente il gestore della coda si ricorda che c'è un ritardo e dice al future di andare avanti con la lettura. Quando la coda torna a contenere aerei, controlla se c'erano ritardi da gestire. Se non ce n'erano, allora mette gli aerei in coda, altrimenti li manda in un messaggio al gestore dei ritardi, il quale li manda alla Pista, che a sua volta gli consentirà il decollo/atterraggio non appena sarà libera.

Grazie alla comunicazione sincrona tra i gestori delle code e alla regola sopra citata, l'ordine con cui gli aerei occupano la pista rispetta quello della tabella oraria a meno dei ritardi.

I messaggi scambiati tra gli attori sono oggetti mai mutabili (gli aerei) e gli stati in 1 e 2 sono locali (quindi solo 1 e 2 rispettivamente prendono decisioni basandosi su essi).