Programmazione ad attori

Erlang è considerato il linguaggio che ha ridefinito la programmazione ad attori, e quando oggi giorno si parla di programmazione ad attori, si pensa semplicemente alla programmazione ad attori così come Erlang l'ha implementata.
L'altro linguaggio di programmazione che è interamente all'interno del paradigma di programmazione ad attori è **Elixir**.
Elixir ed Erlang sono praticamente lo stesso linguaggio di programmazione, dove Elixir cambia la sintassi con una sintassi più simile a quelle dei linguaggi più mainstream a cui siamo abituati e aggiunge delle idee, dei costrutti in più rispetto a Erlang. Il cuore del linguaggio e la programmazione ad attori è esattamente quella di Erlang, perché, come vedremo, Erlang ed Elixir condividono lo stesso runtime.

La programmazione ad attori è una tecnica per scrivere **programmi distribuiti**, ovvero programmi che eseguono concorrentemente con thread che possono essere su macchine sparse su una rete. Per fare programmazione distribuita la programmazione ad attori rimane il paradigma più efficace che sia stato mai sviluppato.

Sono presenti anche molte librerie per fare programmazione ad attori con molti linguaggi di programmazione mainstream, come ad esemprio librerie per Java e Scala (libreria *Akka*).
Questo approccio però non è sempre consigliato. Quando viene preso un linguaggio di programmazione in cui si cerca di mettere costrutti da due paradigmi, l'idea naïve è che sia una soluzione migliore rispetto ai due paradigmi di partenza, perché è possibile fare una cosa e l'altra, in verità combinare costruti da due paradigmi in generale è problematico.

Questo perchè quando viene scritto un frammento di codice, vengono garantite delle proprietà dell'output a partire dalle garanzie presenti sulle proprietà dell'input. Avere a disposizione un linguaggio multiparadigma in cui è possibile unire costruti semplifica la parte di produrre un output e quindi in qualche modo dovrebbe semplificare la parte di produzione di un output con certe garanzie. Il problema è che si va ad indebolire le garanzie in input.
Ad esempio, se viene scritto codice in un linguaggio funzionale puro, dove non c'è mutabilità delle strutture dati, si ha la garanzia che il dato non possa essere mutato da qualcun altro. Questo diventa molto importante perché sapere che certe parti del dato non vengono mutate è la chiave che permette di ottenere l'algoritmo con una certa complessità computazionale sintotica nei linguaggi funzionali. Se si va a mischiare con dei costrutti, ad esempio, imperativi che possono mutare il dato, si va a perdere questa garanzia.

Tipicamente un linguaggio di programmazione, un paradigma, viene caratterizzato sia da quello che vi permette di fare, ma anche e spesso soprattutto da quello che *non vi permette di fare*, quindi dalle salvaguardie che vi mette a disposizione.
Mischiare paradigmi indebolisce le garanzie e quindi non sempre è la cosa ottimale da fare.

Erlang è un linguaggio di programmazione **funzionale**. All'interno della classe di linguaggi di programmazione funzionale, Erlang si distingue perchè è **non tipato**, cosa estremamente rara, dato che la maggior parte dei linguaggi di programmazione funzionali oggi giorno sono tipati.

Inoltre, Erlang è un po' più di basso livello rispetto agli altri linguaggi simili. In altre parole, bisogna pensare ad Erlang come l'**assembly** dei linguaggi di programmazione funzionali.

Erlang non nasce come linguaggio di programmazione ad attori, ma la riscopre successivamente dopo la sua nascita nel 1976. Oggi, quando si parla di programmazione ad attori, si parla di Erlang.

Al centro della programmazione ad attori c'è la nozione stessa di **attore**. Un attore è caratterizzato da tre ingredienti principali:

- **PID (Process Identifier)**: identificatore univoco dell'attore. E' un *nome logico*. I nomi logici dicono come raggiungere l'attore. Quindi l'unico modo per contattare un attore, se non si sa dov'è perché il linguaggio lo nasconde di natura, è conoscere il nome logico. Quindi solo chi conosce il nome logico di un attore è in grado di comunicare con quell'attore.
- **Mailbox**: coda per la ricezione dei messaggi. L'idea è che un attore riceva dei messaggi e li metta in coda.
- **Behaviour**: mappa che prende i messaggi e li trasforma in una lista di azioni e un nuovo behavior. Ogni volta che viene ricevuto un messaggio, il behavior può cambiare completamente dal precedente. Essendo una mappa, ad ogni messaggio ricevuto corrisponde una lista di azioni ed un nuovo comportamento da rispettare. Per azione si possono intendere anche computazioni interne.

L'attore può inviare in maniera asincrona messaggi verso altri attori di cui conosce il PID. L'invio è solamente **asincrono**, quindi non c'è sincronizzazione. Viene spedito un messaggio e non so se verrà ricevuto o quando verrà ricevuto. Questa è l'unica forma di comunicazione in Erlang. Inoltre, un attore può dare vita ad altri attori.

Un attore corrisponde ad un **unico thread di computazione**, quindi all'interno di un attore non possiamo avere più thread.
La programmazione ad attori è un caso particolare di programmazione **reattiva** o **event-driven**. Immaginiamo un sistema che abbia decine di migliaia di attori, centinaia di migliaia di attori, milioni di attori. WhatsApp fu implementato inizialmente tramite Erlang. Ogni singola app di WhatsApp installata corrisponde ad un attore. Quindi c'è un attore in esecuzione per ognuno di noi. Erano quindi presenti migliaia di migliaia di attori nel nostro sistema di attori. Gli attori sono normalmente crescenti.

Non c'è però l'idea che gli attori stiano facendo calcoli. Quando un attore riceve un messaggio, fa qualcosa per poco tempo, scatenando magari altri messaggi, per poi tornare a fare quelle scelte. Quindi la programmazione ad attori predilige configurazioni in cui ci siano un numero elevatissimo di attori di cui però pochi siano contemporaneamente facenti qualcosa. È il duale della programmazione multithreading dove il numero dei thread deve rimanere basso, piccolo, perché ogni thread sarà costoso e lanciare più thread rallenta significativamente il sistema.

Inoltre, gli attori non condividono nulla, non condividono ne stato ne memoria, anche se

sono due attori sulla stessa macchina, che magari stanno collaborando fra di loro attraverso scambio di messaggi, non condividono risorse di nessun genere.

Un **sistema di attori** è un sistema complesso composto da più attori in esecuzione. L'esecuzione è indipendente dalla locazione fisica degli attori e la topologia della rete è variabile. Gli attori possono essere eseguiti in maniera totalmente trasparente sullo stesso nodo, sulla stessa Virtual Machine, sullo stesso Core, sulla stessa CPU. Questo vuol dire che in Erlang non c'è di fatto distinzione tra programmazione concorrente su una singola macchina, programmazione distribuita o programmazione parallela.
Uno dei punti di forza della programmazione ad attori di Erlang è aver dato una soluzione drastica ma perfettamente funzionante alla gestione dei guasti.

Esempio pratico in Erlang

Erlang condivide la sua sintassi con il linguaggio di programmazione logica Prolog.
Un esempio di attore per la gestione di un conto corrente è il seguente:

```
-module(hello).
-export([cc/1]).

cc(Bal) ->
    receive
        print -> io:format("Il balance è ~p ~n", [Bal]), cc(Bal);
        {put, N} -> cc(Bal + N);
        {get, PID} -> PID ! Bal, cc(Bal);
        exit -> ok
    end.
```

Il primo step è la dichiarazione del **module**, dove il suo nome deve essere lo stesso del nome del file. In Erlang, quando una riga inizia con il segno **-** è una direttiva, quindi non fa parte del codice ma aiuta il compilatore a fare qualcosa.

Un secondo aspetto è che all'interno di un codice Erlang vengono definite delle funzioni, le quali si può decidere se esportare o meno tramite **export**. In questo modo è possibile scegliere quali funzioni esportare e quali non esportare. Tutto ciò che non viene esportato è nascosto. La direttiva export prende una lista di funzioni da esportare, dove va dichiarato il nome della funzione / il numero di parametri che richiede in input la determinata funzione. Questo perchè ci possono essere funzioni con lo stesso nome ma differenziate dal numero di parametri.

In Erlang, tutte le istruzioni, i comandi, le definizioni, terminano con il **punto**.

Quando definiamo un attore, bisogna definire un behavior. Il behavior deve dire: se ricevo certi messaggi, faccio qualcosa.
Il mio conto corrente deve avere un *balance*, che per l'appunto mi dice quanti soldi ho. Una

volta ricevuto un messaggio, devo acquisire un nuovo behavior.
L'idea di Erlang è che il behavior viene definito da una funzione, tipicamente ricorsiva. Quindi, la mia funzione ricorsiva prenderà in input lo stato, in questo caso prenderà in input il balance attuale, poi descriverà come reagire ai messaggi, e tipicamente, dopo aver ricevuto un messaggio, prenderà un altro behavior. In questo caso, il behavior è sempre quello. Io voglio che il mio conto corrente continui a ricevere messaggi.

Erlang non ammette mutazioni di nessun tipo, tranne per un paio di costruttori particolari. Le variabili non possono quindi essere variate, sono praticamente delle costanti.

Un behavior viene descritto all'interno di un costrutto **receive-end**. Il behavior deve dirmi: se ricevo un certo messaggio, faccio una certa cosa. Vado a scrivere questo tramite una serie di pattern che verranno confrontati con i messaggi nella mailbox in un certo ordine e se nella mailbox viene trovato un messaggio che fa match con un certo pattern, allora viene eseguito un certo codice.
Un pattern è una descrizione, una possibile forma dell'input. Quindi, ogni possibile descrizione di un pattern può matchare o no qualcosa che era mailbox, se con la stessa forma. E non solo, nel caso in cui il pattern matchi qualcosa che era mailbox, legherà dei nomi di variabile al contenuto del messaggio.

Come sono fatti questi pattern? Il primo e l'ultimo pattern, *print* ed *exit*, corrispondono semplicemente ad avere ricevuto come input print o exit.
Anche in questo caso sono scritti con le lettere minuscole, come *cc*, ma non hanno una tonda. Con la lettera minuscola e parentesi tonde, è una funzione. Senza tonda non è una funzione, ma un **atomo**.
Un atomo è un dato che non ha niente di interessante se non il fatto di essere uguale a se stesso e diverso da tutti gli altri atomi. Lo si può utilizzare per indicare qualcosa, ad esempio l'atomo print è diverso dall'atomo exit. Quindi, in questo caso, lo si usa come messaggio per distinguere.

Passiamo invece al *put*, dove il messaggio deve contenere anche quanti soldi si vogliono versare. Il messaggio, in questo caso, è una coppia di valori dove il primo valore è l'atomo put ed il secondo la variabile con lettera maiuscola. Questo permetterà di fare **pattern match**. Si noti come non c'è nessuna modifica del dato, non viene imperativamente cambiato il balance, semplicemente viene fatta una chiamata ricorsiva passando un nuovo valore.

All'interno della *print*, è possibile osservare la separazione tra due istruzioni. La separazione in Erlang viene fatta con l'utilizzo delle **virgole**, mentre la separazione dei casi con il **punto e virgola**.

Per quanto riguarda *exit*, voglio terminare l'esecuzione del mio attore. Viene quindi restituito un valore di uscita, che può essere qualsiasi cosa.

Andando invece alla *get*. Per ottenere le informazioni riguardanti il proprio conto corrente, bisogna necessariamente conoscere il proprio PID. In questo modo, l'attore saprà a quale PID inviare il risultato della get. La sintassi successiva permette di inviare al PID il valore

del balance.

Passiamo ora all'esecuzione del codice.
Erlang è un pò come Java, in cui il codice viene compilato per poi essere eseguito da una Virtual Machine (chiamata **Beam**). In linea di massima su ogni macchina fisica è in esecuzione una virtual machine, chiamata **nodo**. Nulla però vieta di avere più virtual machine su una macchina, aumentando il numero di nodi.

Quando viene lanciato il nodo della mia macchina virtuale viene anche lanciata una **shell**, la quale permette di dare comandi e vedere risposte. Anche la shell è un attore.
Ogni volta che viene lanciato un attore viene lanciata anche una shell. Ad esempio, in una shell io posso lanciare l'attore conto bancario e poi comunicarci.
L'idea alla base è che tutto è un attore in Erlang.

Come detto in precedenza, la prima cosa da fare è compilare il codice. E' presente una funzione chiamata `c(nome_file)`, dove è possibile passare il nome del file da compilare. La compilazione va a buon fine se non ci sono errori.
Una volta compilato, la funzione potrà essere richiamata automaticamente, venendo caricato il modulo dalla virtual machine in maniera autonoma.

Per creare un attore che abbia quel behavior, si può usare il comando `spawn(nome_modulo, nome_funzione, lista_argomenti)` (NB: la spawn ha diverse sintassi, ne stiamo usando soltanto una in questo caso).
Questo comando restituisce il PID dell'attore appena creato. Questo PID non necessariamente è lo stesso su tutti i nodi, ma il runtime riesce a matchare sempre l'attore inerente a quel PID.

Per eseguire i comandi basterà eseguire nella shell il comando, ad esempio, `PID ! print.`, il quale restituisce il balance del conto.
Allo stesso modo, `PID ! {put, 2}.` aggiornerà il valore del balance del conto corrente.

Per la get la situazione è leggermente differente. All'apparenza non riceviamo una risposta alla richiesta effettuata. Questo succede perchè la shell è un attore un pò strano. La shell è un attore leggermente diverso dagli altri attori, perché gli attori normalmente hanno un behavior che risponde ai messaggi che ricevono. Nel caso della shell c'è una sorta di secondo behavior, dove invece di guardare i messaggi ricevuti si va a guardare quello che viene digitato sullo standard input. Il messaggio ricevuto viene inserito nella mailbox della shell, ma nessuno lo sta processando.
Per tirarlo fuori dalla mailbox bisogna dare un behavior alla shell stessa, ad esempio con il comando `receive MSG -> MSG end.`

Questo metodo di programmazione rispecchia la logica della programmazione ad attori del 1976, prima delle diverse aggiunte di Erlang.

Erlang

Il linguaggio Erlang ha alla base il concetto di programmi concorrenti che vengono eseguiti per lunghissimi periodi di tempo, con concorrenza massiva (>= 100000 attori).
Permette anche di scrivere codice per sistemi distribuiti, con possibilità di effettuare upgrade del codice a runtime senza dover fermare l'esecuzione del programma.
Presenta inoltre diversi meccanismi per la gestione dei guasti, i quali rendono Erlang un linguaggio perfetto se si ha bisogno di runtime "infiniti" (nel caso degli switch Ericsson con software scritto in Erlang, con uptime del 99.999999%).

Le principali scelte di design di Erlang sono:

- **Nessuna memoria condivisa**, lock o mutex, perchè porterebbe ad una gestione troppo complessa e non funzionerebbe bene in scenari distribuiti. Inoltre, il meccanismo della memoria condivisa è pessimo dal punto di vista della fault tolerance, dato che in casi di fallimenti con thread che hanno acquisito un lock si andrebbe a creare un deadlock.

- **Message passing asincrono**, con ricezione out-of-order. L'ordine di ricezione dei messaggi in uno scenario distribuito è casuale. Se un messaggio viene estratto dalla mailbox al momento errato deve essere processato esplicitamente più tardi.

- **Fault tolerance via "Let it fail!"**. La gestione degli errori è molto complessa in scenari distribuiti, a causa di possibili deadlock causati da messaggi persi o processi remoti irraggiungibili. Il meccanismo del *Let it fail* afferma che in presenza di errori, si uccide un processo e tutti coloro strettamente a lui connessi. Una volta terminati, un supervisore farà ripartire i processi terminati.
  L'idea è quella di avere una struttura gerarchica della computazione. Invece di risolvere un problema vengono generati una serie di figli, gli attori figli. Gli attori figli collaboreranno tra di loro per portare al termine la risoluzione del problema.
  L'attore padre si mette in uno stato di inattività ad osservare i figli e attende che i figli vengono terminati. Se uno dei figli fallisce per un motivo, si trascinerà dietro la morte di tutti i suoi fratelli ed eventuali figli nipoti e così via. Il padre che ha generato questi figli osserverà la loro terminazione e potrà far ripartire la computazione. Ovviamente se dopo un po' di tentativi di restart dei figli non cambia niente, il padre terminerà se stesso. Terminandosi si porta dietro tutti gli zii, tutti i suoi fratelli e il nonno verrà informato e farà la stessa cosa.
  Ovviamente quest'idea funziona molto bene se il padre è in grado di ricreare i figli con lo stato che avevano in quel momento. Se il linguaggio di programmazione è funzionale, privo di mutazione, allora nel momento in cui il padre crea i figli ha il suo stato e quello stato non verrà alterato dai figli, perché in un linguaggio funzionale la memoria non viene mai modificata, si crea sempre memoria nuova.
  Quindi nel momento in cui i figli vengono sterminati, il padre può ricrearli in tempo O(1), non deve fare backtracking, non deve ricreare situazioni, semplicemente il padre ha esattamente lo stato in cui era quando ha creato i figli la prima volta e quindi può velocemente crearlo.

La Virtual Machine di Erlang è chiamata **Beam**, la quale gira come unico kernel process multi-threaded, con un thread per core.

Per ogni attore invece è presente un **Language Thread**. Il sistema operativo non conosce nulla degli attori, vengono schedulati sui kernel thread con un load balancing automatico.

Inoltre, il Context Switch per i language thread è estremamente leggero. Erlang si basa sul context switch collaborativo, dove la virtual machine si occupa di effettuare i vari cambi di contesto. Si parla in questo caso di **green thread**, che utilizzano poche risorse.

Anche il memory footprint dei language thread è estremamente leggero, con un nuovo attore che nasce con circa 300 parole di memoria occupata. Questo permette di avere centinaia di migliaia di attori. Gli attori Erlang partono con pochissima memoria utilizzata, senza uno stack preallocato.
Lo stack viene fatto crescere dinamicamente all'occorrenza, andando a creare uno stack di dimensione 2N e copiandoci all'interno il contenuto dello stack precedente. Questo ha un costo computazionale ammortizzato di **O(n) lineare**.
Questo costo è relativo solo al momento della creazione del nuovo stack, quindi negli altri casi il costo è ammortizzato è di *O(1)*. Questo permette di avere milioni di language thread contemporaneamente.

Un'altra caratteristica fondamentale è l'**hot code swap**, ovvero la possibilità di caricare a runtime una nuova versione del codice, senza buttare giù il programma.

La libreria standard si chiama **OTP (Open Telecom Platform)**, nel quale possiamo trovare tutto quello che serve per implementare software distribuiti.