**Emerging Programming Paradigms**

[ lezioni 23-25 marzo ]

**Accesso ai bit a basso livello**

L = [1, 2, 3] M = [a, b]

[ {X, Y} || X <- L, Y <- M ].

Il costrutto precedente sta ad indicare “l’insieme delle coppie X, Y tali che X assuma ogni valore di L ed Y ogni valore di M”. Nota: l’ordine con cui sono messi i valori dopo il || cambia il risultato della lista.

Con la dicitura: << 6:3, “a”, 2:4 >> si indica “rappresentazione del numero 6 usando 3 bit concatenati alla stringa “a” concatenato al numero 2 usando 4 bit”.

Si può fare anche pattern matching:

<< X:4, Y:4 >> = M.

Ovvero si divide M in due porzioni da 4 bit in cui la prima verrà chiamata X e la seconda Y.

Un classico utilizzo è la rappresentazione di colori in esadecimale e componenti RGB:

D = << 16#FF23A3:24 >>

<< R:8, G:8, B:8 >> = D.

Un altro esempio (non corretto sintatticamente, da guardare meglio sulle slide):

<< X:4 || X <= << 1435535143:24 >> >>.

(list comprehension in cui si richiedono tutte le “porzioni” di 4 bit all’interno del numero)

**Concorrenza in Erlang**

Riprendendo l’esempio di Hello World concorrente in Erlang:

main(N) ->

io:format(“Hello, counter=~p~n”, [N]),

receive

{add, M} -> main(N+M)

end.

Nota: la receive è bloccante, la send è asincrona quindi non bloccante.

La send è rappresentata tramite il punto esclamativo: P ! {add, 4}.

Due scenari in cui un processo non riceve messaggi sono dati da:

- problemi di rete

- l’agente che deve inviarli è morto

Per gestire queste situazioni si può utilizzare la keyword after per gestire un tempo di attesa dopo il quale il processo se non riceve messaggi fa qualcosa.

receive ok -> 0 after 2000 -> ko end.

receive ok -> 0 after infinity -> ko end.

receive ok -> 0 after 0 -> ko end.

after 0 è un modo per ottenere una receive asincrona, se non c’è una corrispondenza in coda si passa subito oltre senza bloccare.

Un classico esempio è la funzione sleep:

sleep = fun (X) -> receive after X -> ok end end.

Test Receive

sleep (N) -> receive after N -> ok end.

g() ->

receive

a -> io:format(“a~n”, []);

b -> io:format(“b~n”, []);

c -> io:format(“c~n”, [])

end,

g().

f() ->

sleep(200),

g().

main() ->

P = spawn(fun f/0),

P ! b,

P ! c,

P ! a,

sleep(5000).

L’output è in base all'ordine di ricezione dei messaggi, non ai pattern di ricezione. Se si ricevono due messaggi inviati da stesso mittente a stesso ricevente, eseguiti sullo stesso nodo, arrivano nello stesso ordine di invio. Tuttavia è sbagliato fare assunzioni di ordine di questo tipo perché se in futuro si fanno delle modifiche le assunzioni crollerebbero.

Se si vuole processare ad esempio a e se a non arriva si processa b, forzando un ordine:

receive

a -> io:format(“a~n”, [])

end,

receive

b -> io:format(“b~n”, [])

end

Implementazione in Erlang del **problema dei filosofi a cena**.

Il problema: si hanno N filosofi attorno a un tavolo rotondo che devono mangiare utilizzando due bacchette, una posta alla loro destra ed una alla loro sinistra. Alternano momenti in pensano a momenti in cui mangiano. Se tutti prendessero la bacchetta destra e poi la sinistra si creerebbe un deadlock, inserendo un filosofo mancino si eviterebbe banalmente il problema.

[Codice/filosofi.erl](https://drive.google.com/drive/u/1/folders/1I2XDnZ5p0jaGmEQOGlpsNv7XnMHUFhCq)

Gli attori possono avere dei nomi pubblici, sotto forma di atomi, tramite la funzione register.

L’operatore -- toglie da una lista gli elementi di un’altra, ad esempio: [1, 2, 3] -- [3, 1] = [2].

Quando un attore muore la sua mail box viene cancellata, i messaggi che erano in transito non andranno quindi a buon fine e gli attori che hanno il suo PID e provano ad inviargli messaggi non riceveranno nessuna risposta. Non vengono lanciati errori, semplicemente i messaggi vengono persi.

**Code Hot Swapping in Erlang**

Prendiamo come esempio un server semplice:

loop(N, M) ->

receive

{add1, X} -> loop(N+X, M);

{add2, X} -> loop(N, M+X)

end.

main() ->

spawn(?MODULE, loop, [0, 0]).

Ora supponiamo si vogliano modificare le strutture dati passate al server.

Si aggiunge un messaggio al server, chiamiamolo “update” in cui quando viene ricevuto si invoca una funzione del nuovo codice.

loop(N, M) ->

receive

{add1, X} -> loop(N+X, M);

{add2, X} -> loop(N, M+X);

update -> hot\_swap:new\_loop(N, M) % eseguito nuovo codice

end.

new\_loop(N, M) ->

loop(N, M).

main() ->

spawn(?MODULE, loop, [0, 0]).

Quando si chiama una funzione senza che sia preceduta dal nome del modulo si va ad eseguire la funzione della “vecchia” versione del codice, se invece si specifica il modulo davanti si esegue la funzione della “nuova” versione del codice. La BEAM mantiene solo due versioni del codice: vecchia e nuova.

% modifico il codice, il precedente rimane disponibile sulla BEAM

loop(R = {N, M}) ->

receive

{add1, X} -> loop({N+X, M});

{add2, X} -> loop({N, M+X});

{add, 1, X} -> loop({N+X, M});

{add, 2, X} -> loop({N, M+X});

update -> hot\_swap:new\_loop({N, M}) end.

% lasciamo due argomenti separati perché

% si possono ricevere ancora in questo formato

new\_loop(N, M) ->

loop({N, M}).

main() ->

spawn(?MODULE, loop, [0, 0]).

Codice completo

[Codice/hot\_swap.erl](https://drive.google.com/drive/u/1/folders/1I2XDnZ5p0jaGmEQOGlpsNv7XnMHUFhCq)

**Attori collegati**

È possibile linkare più attori tra loro in modo che la caduta o il blocco (ad esempio in seguito ad una throw) di uno porti alla terminazione o al blocco degli altri. Questo collegamento avviene tramite la funzione built-in **link**(PID).

[Codice/link.erl](https://drive.google.com/drive/u/1/folders/1I2XDnZ5p0jaGmEQOGlpsNv7XnMHUFhCq)

Utilizzando exit(normal) l’attore che lo invoca viene bloccato e invia il medesimo segnale agli altri attori collegati i quali ignoreranno il segnale e continueranno l’esecuzione.

Utilizzando exit(kill) permette ad un attore di autoterminarsi e inoltrando il segnale anche agli attori collegati fa terminare anche loro.

Infine con exit(PID, foo) si uccide un altro attore collegato mostrando anche un messaggio. Questa è solitamente una pessima idea ma in Erlang è totalmente contemplato in quanto anche tutti i processi a lui collegati terminerebbero (non creando problemi a livello logico).

La BEAM per implementare il meccanismo di linking utilizza il **signal** passing. Ovvero un attore prima di terminare invia un segnale speciale a tutti i processi a lui collegati. Questi processi ricevuto questo segnale termineranno a loro volta con la stessa eccezione (o ignorano il segnale se la terminazione avviene normalmente).

Un processo ucciso tramite la exit() da parte di un altro attore in realtà rappresenta un invito a terminarsi, può anche rifiutarsi se sta computando cose importanti. Un processo però *non può rifiutarsi* di terminare se riceve una kill dentro ad exit().

Un attore muore in caso di: *exit* (suicidio), *throw* (suicidio), *error* (suicidio), *exit* di un’altro attore con PID (omicidio).

La funzione **monitor**() non è bidirezionale, quindi se un processo monitorato muore, l’altro viene avvisato tramite un **messaggio** (non segnale) ma non terminato. Il monitor utilizza un reference counter, quindi se si monitora cinque volte un attore e si “de-monitora” una volta, si sta ancora “monitorando” l’attore quattro volte. Inoltre quando si applica demonitor() è possibile specificare quale istanza di monitoring (se se ne hanno più di una) eliminare. La link() invece è idempotente (bidirezionale, se muore uno dei due muore anche l’altro)

Per evitare *race condition* è possibile utilizzare funzioni combinate: spawn\_link e spawn\_monitor che creano un attore e lo linkano o monitorano subito, se si facessero le due operazioni separate il processo potrebbe cadere dopo la creazione e prima del link/monitor.