

UNIVERSITÀ DEGLI STUDI DI
CASSINO E DEL LAZIO MERIDIONALE



DIPARTIMENTO DI INGEGNERIA ELETTRICA
E DELL'INFORMAZIONE "MAURIZIO SCARANO"

CORSO DI LAUREA IN INGEGNERIA INFORMATICA
E DELLE TELECOMUNICAZIONI

**Assessment delle performance di Elixir
nell'ambito IOT.**

Relatore:

Prof. Ciro D'Elia

Candidato:

Nico Fiorini

ANNO ACCADEMICO 2022/2023

Questa è una dedica

La perfezione non è il nostro obbiettivo ma la nostra tendenza

Omar Palermo

Abstract

L'industria del software si trova a fronteggiare la necessità di sviluppare software sempre più scalabili e performanti per fronteggiare l'aumento degli utenti e di servizi che ne fanno utilizzo. In questo contesto, Elixir, un linguaggio di programmazione funzionale e concorrente basato su Erlang, emerge come una scelta promettente per la costruzione di sistemi altamente affidabili e reattivi, semplificando di molto lo sviluppo di software concorrentiale.

Questo studio si propone di analizzare le caratteristiche di Elixir e le sue performance attraverso una serie di esperimenti empirici esplorando diversi aspetti delle performance mettendo in rilievo vantaggi e svantaggi nell'adottarlo.

I risultati di questa ricerca forniranno una comprensione approfondita delle capacità di Elixir in termini di prestazioni e affidabilità consentendo agli sviluppatori di fare una scelta pensata alle esigenze dei loro progetti.

Indice

1	Introduzione	1
2	Caratteristiche di Elixir	3
2.1	Introduzione	3
2.2	Software utilizzati	4
2.2.1	Installazione software utilizzati	4
2.3	Il paradigma funzionale	6
2.3.1	Struttura di un progetto Elixir	7
2.3.2	Basi dichiarative	8
2.3.3	Transizione al funzionale	9
2.3.4	Conseguenze sulle prestazioni dell’immutabilità	10
2.4	Concorrenza	11
2.4.1	La concorrenza in Beam	11
2.4.2	Concorrenza basata su attori	12
2.5	OTP Platform	14
2.5.1	GenServer	15
2.5.2	Supervisor	17
2.6	Integrazione con codice esterno	17
2.6.1	Interoperabilità tramite NIF	18
2.6.2	Interoperabilità con Port	19
3	Performance - Test sperimentali	21
3.1	Introduzione	21
3.1.1	Hardware utilizzato	21
4	Conclusioni	22
	Bibliografia	23

Capitolo 1

Introduzione

Elixir è un linguaggio di programmazione dinamico e funzionale sviluppato nel 2012 da José Valim, con l'obiettivo di favorire una maggiore scalabilità e produttività nella macchina virtuale di Erlang, mantenendo al contempo la compatibilità con l'ecosistema di Erlang[3]. Elixir si è affermato come una promettente scelta nell'industria del software, specialmente in contesti dove è richiesta scalabilità, tolleranza agli errori e reattività grazie al suo approccio concorrenziale.

In particolare, Elixir può risultare vantaggioso nel campo dell'IoT per diversi motivi:

1. **Concorrenza:** Nell'ambito dell'IoT, la gestione simultanea di dispositivi è essenziale. Elixir, grazie alla sua capacità di gestire facilmente la concorrenza, consente il monitoraggio e il controllo efficiente di numerosi dispositivi contemporaneamente.
2. **Fault Tolerance:** Data la natura degli ambienti IoT, dove i dispositivi possono guastarsi improvvisamente, Elixir offre strumenti per la supervisione e la gestione degli errori, garantendo la continuità delle operazioni anche in caso di fallimenti.
3. **Sviluppo Rapido e Manutenzione:** Elixir è un linguaggio moderno che offre una sintassi efficiente e snella, oltre a strumenti di sviluppo come Mix per la gestione delle dipendenze e l'ambiente interattivo iex. La presenza di un package manager (Hex)[[Hex63:online](#)] e la possibilità di generare automaticamente la documentazione facilitano il processo di sviluppo e manutenzione del codice.

Il trattato esplora Elixir concentrandosi su due aspetti principali: la semplicità e le performance. Si analizzano i punti di forza di un linguaggio funzionale

e come questi sono sfruttati in Elixir, con un focus sulla concorrenza. Nella scelta di un linguaggio, la semplicità è fondamentale e deve essere accessibile a tutti i programmatori. Tuttavia, l'efficienza è altrettanto importante, quindi vengono condotti test empirici per valutare le performance di Elixir.

In particolare il lavoro effettuato è così ripartito:

- Nel capitolo 2 si discute del linguaggio funzionale, esaminando le astrazioni offerte da Elixir per lo sviluppo di codice affidabile, si tratta la concorrenza e come la Erlang VM si occupa della gestione dei processi.
- Nel capitolo 3 si spiega il lavoro sperimentale svolto e i risultati ottenuti (continuare)

Capitolo 2

Caratteristiche di Elixir

2.1 Introduzione

In questo capitolo, esamineremo le caratteristiche distintive di Elixir, un linguaggio di programmazione funzionale e concorrente che sfrutta appieno la potenza della piattaforma OTP (Open Telecom Platform). Non si vuole coprire ogni dettaglio del linguaggio, ma mettere in evidenza le caratteristiche fondamentali per iniziare a capire come pensare il codice con questo linguaggio e quali sono le caratteristiche che lo rendono un'opzione da considerare in determinati contesti.

Elixir, scritto in Erlang ed eseguito sulla macchina virtuale Erlang (BEAM), eredita gli obiettivi di Erlang, ma apporta miglioramenti significativi per rendere il linguaggio più appetibile e moderno.

Erlang, nato nel 1986, è stato progettato per semplificare lo sviluppo di software concorrente robusto e scalabile. Elixir si basa su queste fondamenta solide, offrendo un'API più pulita e astrazioni dell'OTP che consentono ai programmatori di ragionare a un livello più elevato, facilitando la scrittura di codice concorrente in modo intuitivo.

Una delle massime principali di Erlang e, di conseguenza, di Elixir, è "Let it crash" (Lascia che si schianti), che riflette l'approccio alla gestione degli errori nei sistemi concorrenti, incoraggiando la gestione degli errori tramite il rilancio e la supervisione anziché il blocco del processo.

Per capire come lavorare con questo linguaggio, bisogna affrontare un po' di questioni e farsi un po' di domande. Bisogna capire come la macchina virtuale Beam affronta la concorrenza, Elixir in particolare è un linguaggio orientato alla concorrenza e le astrazioni che fornisce sono proprio per far sì che si programmi in modo concorrenziale portando ad avere un codice responsivo, e gestendo bene i processi attraverso il meccanismo di Supervision, il software

diventa anche robusto. Un altro punto che si affronta è l'immutabilità dei dati, un concetto chiave in Elixir ed Erlang, ed è proprio questa caratteristica che ci semplifica la programmazione concorrentiale.

2.2 Software utilizzati

I studi sulle performance riportati nel capitolo 3 sono stati effettuati utilizzando il sistema operativo Linux per scelta personale e per la varietà di software disponibile su Linux. Viene riportata l'installazione effettuata su Linux per i vari test e le relative configurazioni, in modo da poter testare gli esempi riportati in questo capitolo, l'installazione su Windows è immediata, infatti è disponibile sia l'installer per Erlang al seguente link: <https://www.erlang.org/downloads.html> e l'installer anche per elixir nel sito ufficiale <https://elixir-lang.org/install.html>, mentre su linux necessita qualche configurazione in più e viene riportata l'installazione tramite asdf.

2.2.1 Installazione software utilizzati

Per avere l'ultima versione di Erlang ed Elixir su Linux bisogna affidarsi al Tool version manager **asdf** in quanto le versioni disponibili nei package manager della distro di riferimento non sono aggiornate all'ultima versione.

Installazione asdf

Asdf è un tool version manager open source che consente di installare, gestire e utilizzare diverse versioni di software senza dover installare o gestire separatamente ciascuna versione. È soprattutto utile quando si valora su progetti che richiedono versioni specifiche del linguaggio. Nel nostro caso ci permette di scegliere l'ultima versione disponibile su qualunque distro linux indipendentemente dalla distro scelta.

Per l'installazione di asdf basta seguire le istruzioni della documentazione ufficiale del tool al seguente URL: <https://asdf-vm.com/guide/getting-started.html>

Le dipendenze necessarie da installare sono **curl** e **git**, una volta installate le dipendenze richieste basta clonare la repository. In una distro debian i comandi risultano i seguenti:

```
> sudo apt install curl git
> cd $HOME
> git clone https://github.com/asdf-vm/asdf.git ~/.asdf --branch v0.14.0
```

Inserire le seguenti righe nel file di configurazione `.bashrc`:

```
. "$HOME/.asdf/asdf.sh"  
. "$HOME/.asdf/completions/asdf.bash"
```

Assicurarsi che l'installazione sia avvenuta con successo:

```
> asdf --version  
  
v0.14.0
```

Installazione Erlang

Una volta installato asdf, installare Erlang con la versione desiderata seguendo le istruzioni della seguente repository: <https://github.com/asdf-vm/asdf-erlang> Per una distro ubuntu si richiedono le seguenti dipendenze da installare con il package manager apt:

```
sudo apt-get -y install build-essential autoconf m4 libncurses5-dev  
libglu1-mesa-dev libpng-dev libssh-dev unixodbc-dev xsltproc  
fop libxml2-utils libncurses-dev openjdk-11-jdk
```

Per evitare warning nell'intellisense di VScode, prima di procedere con i comandi di asdf, bisogna configurare le seguenti variabili d'ambiente per compilare la documentazione di Erlang necessaria al corretto funzionamento dell'intellisense di VSCode:

```
> export KERL_BUILD_DOCS=yes  
> export KERL_INSTALL_HTMLDOCS=yes  
> export KERL_INSTALL_MANPAGES=yes
```

Facendo attenzione ad usare la stessa sessione del terminale in cui sono stati eseguiti i precedenti comandi eseguire:

```
> asdf plugin add erlang https://github.com/asdf-vm/asdf-erlang.git  
> asdf install erlang 26.2  
> asdf global erlang 26.2
```

Si può anche scegliere una versione differente di Erlang, fare attenzione alla versione dell'OTP che viene installata in quanto sarà necessaria per l'installazione di Elixir.

Se tutto è andato a buon fine si può testare l'avvenuta installazione tramite il comando `erl` che avvia l'ambiente interattivo di erlang.

Installazione Elixir

Per Elixir il procedimento con asdf è simile, bisogna solo fare attenzione alla versione OTP di Erlang installata, in quanto Elixir viene compilato con diverse versioni dell'OTP, fare riferimento al seguente link per il procedimento di installazione: <https://github.com/asdf-vm/asdf-elixir>

Installare le dipendenze richieste:

```
> sudo apt-get install unzip
```

Eeguire i seguenti comandi per procedere con l'installazione:

```
> asdf plugin-add elixir https://github.com/asdf-vm/asdf-elixir.git
> asdf install elixir 1.16.0-otp-26
> asdf global elixir 26.2
```

Se tutto è andato a buon fine si può testare elixir avviando l'ambiente interattivo di elixir **iex**.

Configurazione VsCode

VsCode supporta numerosi linguaggi di programmazione, è gratuito e open source, è un editor leggero che consente un'esperienza di sviluppo piacevole e produttiva con una vasta gamma di estensioni installabili. Per quanto riguarda Elixir, è possibile installare l'estensione **ElixirLS** disponibile al seguente link: <https://marketplace.visualstudio.com/items?itemName=JakeBecker.elixir-ls>

2.3 Il paradigma funzionale

Come già accennato Elixir è un linguaggio funzionale, dove il concetto di funzione ricopre il ruolo di protagonista, i dati sono immutabili e il codice è dichiarativo.

Questo modo di vedere le cose deriva dal Lambda calcolo o λ -calcolo [9] un sistema formale definito da Alonzo Church nel 1936, sviluppato per definire formalmente le funzioni e il loro calcolo.

In un paradigma basato su stati come la programmazione ad oggetti spesso si hanno variabili condivise mutabili, di conseguenza più parti del codice possono riferirsi alla stessa variabile, complicando la programmazione multithreading dovendosi preoccupare di meccanismi come il blocco sincronizzato o il locking per evitare le race condition tra più parti del codice, così non è immediato scrivere del codice concorrentiale sicuro riscontrando possibili comportamenti indeterminati se non gestiti bene. In un paradigma funzionale si prediligono le variabili immutabili che aggirano questo problema riducendo il rischio di scrivere codice concorrentiale non sicuro.

Cambiare paradigma non è immediato, un paradigma si può dire che definisce il modo di pensare al problema, nella programmazione ad oggetti per esempio si definiscono le cosiddette classi, pensando al problema come oggetti che hanno un determinato comportamento, e cambiano stato nel

tempo. Questo può essere un modo di sviluppare intuitivo ma non sempre ottimale per la risoluzione di problemi. Nella programmazione funzionale si cambia prospettiva, ovvero si ha un input, si passa l'input alla funzione e si ottiene la trasformazione dell'input ottenendo l'output, quindi si trasformano dati, non si mutano.

In poche parole un linguaggio funzionale assume che scrivere un software complesso sia più facile nel momento in cui il codice ha queste proprietà:

- I dati sono immutabili
- Le funzioni sono pure, ovvero, il risultato di una funzione dipende soltanto dai suoi parametri in input.
- Le funzioni non generano effetti oltre il suo valore restituito.

Con queste proprietà si ha più controllo del flusso del programma, anche se non sempre possono essere soddisfatte anche in un linguaggio funzionale.

2.3.1 Struttura di un progetto Elixir

Elixir è un linguaggio moderno, e come ogni linguaggio moderno che si rispetti fornisce un tool per la creazione e configurazione di progetti, questo tool si chiama **Mix**.

Il tool Mix

È possibile creare un progetto con il comando:

```
mix new <nome-progetto>
```

Verrà creata una struttura per il progetto come nell'esempio 2.1, il codice sarà organizzato nella cartella **lib**, viene creata una cartella **test**, e il file per la configurazione del progetto **mix.exs**. Da notare che anche la configurazione del progetto avviene tramite funzioni.

```
.
>build
>deps
-->...
>lib
-->example.ex
mix.exs
README.md
>test
-->example.exs
```

Esempio 2.1: Struttura progetto

Come sappiamo Elixir fornisce anche un ambiente interattivo (**iex**) per testare il nostro codice, ed è consentito avviare questo ambiente nel contesto della nostra applicazione con il comando:

```
iex -S mix
```

Con Mix possiamo includere e scaricare facilmente anche librerie esterne attraverso il package manager definendo la dipendenza nel file di configurazione `mix.exs`.^[7]

Moduli

Elixir organizza il codice in Moduli, permettendo di definire le funzioni dentro dei namespace, così da separare le responsabilità delle funzioni.

Ci sono varie cose che si possono definire dentro un modulo, si possono definire delle **struct** ma cosa più importante si possono definire i cosiddetti **Behaviour**, un modo per definire un interfaccia Api, Elixir fornisce delle astrazioni proprio attraverso questi, come il `GenServer` ed il `Supervisor`. Ciò che si vuole evidenziare ora è che il progetto è definito in moduli, il modo che Elixir fornisce per organizzare il codice separando in namespace e poter dare delle particolari caratteristiche alle funzioni.

2.3.2 Basi dichiarative

Come già accennato, Elixir adotta un approccio dichiarativo nella definizione delle funzioni. Questo si contrappone all'approccio imperativo, che si concentra su "come posso risolvere questo problema?", mentre quello dichiarativo si pone la domanda "come posso definire un problema?".

Nell'esempio 2.2 è presentato un approccio imperativo al problema "somma dei primi n elementi" nel linguaggio C, mentre nell'esempio 2.3 è presentato l'approccio dichiarativo con Elixir.

```
1 int sum_first_n(n){
2     int sum=0;
3     for(int i=1;i++;i<=n){
4         sum+=i;
5     }
6     return sum;
7 }
```

Esempio 2.2: Somma N elementi

```
1 defmodule Sum do
2     def sum_recursive(0), do: 0
3     def sum_recursive(n), do: n + sum_recursive(n - 1)
4 end
```

Esempio 2.3: Somma N elementi

Pattern matching

L'approccio dichiarativo è rafforzato tramite il meccanismo del **Pattern Matching**, infatti in Elixir l'operatore `=`, non è un operatore di assegnazione come nei più comuni linguaggi, ma è comparabile all'equivalente algebrico. Quest'operatore ci permette di scrivere dell'equazioni che condizionano il flusso del codice, il controllo del flusso in questo linguaggio è pesantemente influenzato da questo meccanismo, infatti si vedono i costrutti *if - else* molto raramente. Questo è molto utile per l'approccio dichiarativo, infatti possiamo definire più corpi della stessa funzione, ed Elixir capisce quale funzione invocare in base al valore dei suoi parametri attraverso questo meccanismo, è già stato utilizzato implicitamente nell'esempio 2.3, dove viene invocata la funzione `sum_recursive(0)` quando il valore dell'argomento vale 0, altrimenti chiamerà quello con l'argomento `"n"` assegnando il valore dell'argomento alla variabile `n`.

Elixir cerca di assegnare il valore a destra dell'equazione al valore di sinistra cercando di risolvere l'equazione, tenta di fare un assegnazione dove possibile per risolvere l'equazione. Questo ci permette di usare quest'operatore per poter scomporre un dato, viene mostrato nell'esempio?? tramite l'ambiente interattivo *iex*

```
1 # Lists
2 iex> list = [1, 2, 3]
3 [1, 2, 3]
4 iex> [1, 2, 3] = list
5 [1, 2, 3]
6 iex> [] = list
7 ** (MatchError) no match of right hand side value: [1, 2, 3]
8 iex> [1 | tail] = list
9 [1,2,3]
10 iex> tail
11 [2,3]
```

Esempio 2.4: Pattern Matching

2.3.3 Transizione al funzionale

Cambiare paradigma può essere difficoltoso, bisogna cambiare prospettiva, ma con le giuste intuizioni può risultare semplice. Per fare una transizione al funzionale bisogna capire soprattutto qual'è la differenza rispetto ad un linguaggio basato a stati.

Elixir non ha oggetti, il linguaggio ha un forte focus sull'immutabilità. In Elixir trasformiamo dati piuttosto che mutarli, infatti da questo punto di vista probabilmente è più naturale il paradigma funzionale piuttosto che un procedurale, ad esempio in un linguaggio procedurale possiamo scrivere:

```
1 my_array = [1,2,3]
2 do_something_with_array(my_array)
3 print(my_array)
```

Ci potremmo aspettare che la stampa dell'array sia [1,2,3] quando in realtà l'output dipende da cosa fa la funzione. In Elixir non è implicito, ma l'assegnazione deve essere sempre esplicita come segue:

```
1 my_list = [1,2,3]
2 my_list = do_something_with_array(my_list)
3 IO.inspect(my_list)
```

Se *my_list* non viene riassegnata, *my_list* non cambia.

Il problema arriva nel momento in cui ci serve uno stato da condividere, una semplice variabile globale non esiste in Elixir, e per mantenere uno stato dobbiamo affidarci ad un altro processo che lo mantiene per noi, ed Elixir almeno per questo ci dà delle astrazioni a cui fare affidamento come il Modulo **Agent** o il **GenServer** che verranno trattati in seguito.

2.3.4 Conseguenze sulle prestazioni dell'immutabilità

L'immutabilità ci consente di avere un controllo maggiore sul flusso del codice, si può pensare che può essere inefficiente e dovendo sempre copiare dati, inoltre si può pensare che si creano molti dati non più utilizzati da liberare in memoria con il Garbage collector portando anche il garbage collector ad essere inefficiente. Questo è vero solo in parte, infatti Elixir sfrutta la proprietà dell'immutabilità per limitare l'inefficienza.

Elixir sa che i dati sono immutabili, può riusarli in parte o per intero quando si creano nuove strutture. Il seguente esempio mostra come Elixir ottimizza la creazione di nuovi dati:

```
1 iex> lista1 = [ 3, 2, 1 ]
2 [3, 2, 1]
3 iex> lista2 = [ 4 | lista1 ]
4 [4, 3, 2, 1]
```

La *lista2* infatti è creata usando la *lista1* sapendo che non cambierà mai durante il flusso di esecuzione, quindi mette semplicemente in coda la *lista1*, non dovendo creare una nuova copia della *lista1*.

Garbage Collection

Un altro aspetto che può creare dubbi sul copiare dati, è che si lasciano spesso vecchi dati non più utilizzati da dover pulire con il Garbage Collector.

Scrivendo codice Elixir però ci rendiamo subito conto che ci porta a sviluppare codice con migliaia di processi leggeri, ed ogni processo ha il

proprio heap. I dati nell'applicazione quindi sono suddivisi nei processi, ogni heap è molto piccolo rendendo il garbage collector più veloce. Se un processo termina prima che il suo heap diventi pieno, tutti i suoi dati vengono eliminati, senza necessità di liberare la memoria rendendo il garbage collector efficiente. [10]

2.4 Concorrenza

La concorrenza è un concetto fondamentale nell'ambito dello sviluppo software, si riferisce alla capacità di eseguire più attività contemporaneamente. Questo è particolarmente importante in applicazione che devono gestire molteplici operazioni in parallelo come ad esempio un server web, applicazioni di elaborazione dati.

La concorrenza consente a un programma di sfruttare appieno le risorse disponibili, aumentando l'efficienza e migliorando le prestazioni complessive. Inoltre, permette di creare sistemi più reattivi e scalabili, in grado di gestire un numero crescente di richieste senza compromettere le prestazioni.

In generale, la gestione della concorrenza può essere complessa e soggetta a errori o comportamenti indeterminati, infatti, più processi o thread possono interagire tra loro in modo imprevedibile, essendo così costretti a usare tecniche di locking in concomitanza di memoria condivisa tra più thread.

Nel contesto di Elixir, la concorrenza è un concetto centrale e viene gestita attraverso il modello di programmazione basato su processi leggeri, tutti isolati tra loro con il proprio stack ed il proprio heap.

2.4.1 La concorrenza in Beam

La concorrenza gioca un ruolo chiave per un software che vuole essere altamente responsivo. La concorrenza fa uso dei cosiddetti processi leggeri nella piattaforma Erlang, non sono processi del sistema operativo, ma processi della VM Beam, chiamato processo e non thread in quanto non condividono memoria e sono completamente isolati tra di loro.

Un server tipico deve gestire migliaia di richieste, e gestirle concorrentialmente è essenziale per non far rimanere in attesa il client. Quello che si vuole è gestirli parallelamente il più possibile sfruttando più risorse della Cpu disponibile. Quello che fa la macchina virtuale per noi è permetterci la scalabilità, più richieste allora più risorse da allocare.

Inoltre, siccome il processo è isolato, un errore in una richiesta può essere localizzato senza avere impatto sul resto del sistema, così creando anche un sistema robusto agli errori.

Un processo appena creato occupa in memoria 326 words [5], quindi in una macchina a 64 bit occupa 2608 byte. Lo si può vedere in Elixir facilmente nell'esempio 2.5

```
1 defmodule Examples.Memory do
2   def mypid do
3     receive do
4       :stop -> :exit
5       _ -> mypid()
6     end
7   end
8
9   def benchmark do
10    pid = spawn(fn -> mypid() end)
11    {_,byte_used} = :erlang.process_info(pid,:memory)
12    IO.puts("La memoria usata dal processo e': #{byte_used} byte")
13    send pid, :stop
14  end
15 end
```

Esempio 2.5: Memoria in un processo

Infatti l'output ottenuto dall'esempio 2.5

```
iex> Examples.Memory.benchmark
La memoria usata dal processo e': 2640 byte
:stop
```

Si nota dall'output dell'esempio che la memoria utilizzata è leggermente superiore della memoria minima dichiarata, infatti il processo non fa nulla di particolare oltre ad aspettare il messaggio di ":stop". È da notare che è proprio questa leggerezza nei processi che permette al linguaggio di essere orientato alla concorrenza, e poter usare i processi con più leggerezza rispetto ad altri meccanismi di altri linguaggi che usano i thread. Un altro punto da notare è che si può migliorare significativamente la reattività del programma ma non l'efficienza totale del sistema, infatti non tutti i processi sono eseguiti in parallelo, quindi in una macchina con quattro processori non si possono eseguire più di 4 processi per volta ed in un normale software Elixir è normale avere migliaia di processi che lavorano.

Facendo un esempio con una Cpu dual-core, i processi vengono eseguiti concorrentialmente e gestiti dagli Scheduler della VM, in figura 2.1 è mostrato come la VM gestisce i processi di default.

2.4.2 Concorrenza basata su attori

Elixir usa un modello di concorrenza basato su attori, gli attori sono entità di elaborazione indipendenti che eseguono operazioni in modo asincrono, questi attori non sono altro che processi che vengono identificati attraverso un **PID** univoco. Come già detto sono isolati l'uno dall'altro e comunicano

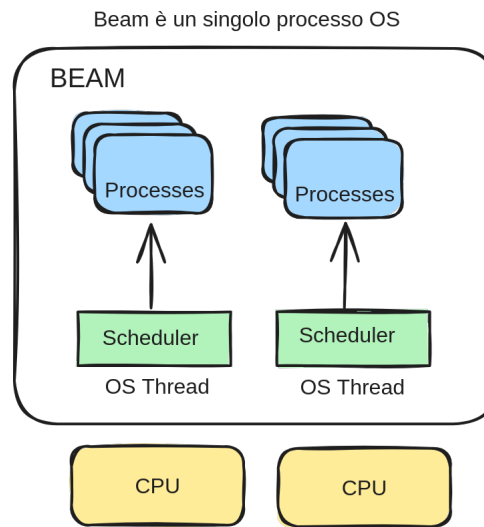


Figura 2.1: Concorrenza nella VM Beam [8]

solo attraverso lo scambio di messaggi, questo scambio avviene attraverso dei canali di comunicazione detti "**mailbox**". Ogni processo ha una propria mailbox dove avviene la ricezione del messaggio da parte di altri processi.

Conoscendo il PID di un processo può avvenire la comunicazione attraverso la primitiva fornita dal linguaggio **send/2** che permette di inviare un messaggio ad un processo come avviene nell'esempio 2.5, dove il processo principale crea un processo che rimane in ascolto tramite il blocco **receive/1**, fin quando non riceve il messaggio di **:stop** e il processo viene terminato.

Se non ci sono messaggi nella mailbox, il processo aspetta fino a quando non arriva un messaggio, in particolare, nell'esempio tutti i messaggi che non siano **:stop**, verranno ignorati continuando ad ascoltare altri messaggi.

Process Linking

I processi sono isolati, ma si può legare un processo al processo chiamante tramite un'operazione di Process linking, questo per sapere se il processo va in errore oppure termina per un comportamento imprevisto, legando i processi si può propagare l'errore nel caso uno dei due processi non ha senso di esistere preso da solo.

Spesso i processi vengono legati ad un Supervisor, che lascerà andare in errore i processi figli, e si occuperà semplicemente di riavviarli seguendo una strategia scelta. Questa è la filosofia "Let it Crash" di Erlang, che si contrappone alla gestione delle eccezioni di altri linguaggi. Si veda l'esempio 2.6

```
1 iex>spawn(fn -> raise "oops" end)
2 #PID<0.58.0>
3
4 [error] Process #PID<0.58.00> raised an exception
5 ** (RuntimeError) oops
6     (stdlib) erl_eval.erl:668: :erl_eval.do_apply/6
7
8 iex(2)spawn_link(fn -> raise "oops" end)
9
10 17:37:50.425 [error] Process #PID<0.113.0> raised an exception
11 ** (RuntimeError) oops
12 ** (EXIT from #PID<0.110.0>) shell process
13 ** exited with reason: {%RuntimeError{message: "oops"}, []}
14
15 iex(2)
```

Esempio 2.6: Process linking

Notiamo nell'esempio 2.6 che con il comando la funzione `spawn_link/1` il processo va in errore e propaga l'errore all'ambiente interattivo `iex`, riavviando anche l'ambiente `iex`.

Si nota in conclusione al paragrafo che è possibile gestire migliaia di processi molto facilmente senza incrementare in modo considerevole le risorse in memoria grazie ai processi leggeri, permettendo di sviluppare un software orientato alla concorrenza, tutto è facilitato dall'isolamento di ogni processo che permette di non preoccuparsi di meccanismi di locking e sincronizzazione. Elixir permette di non preoccuparci di come scalare le risorse hardware, potendo dare responsabilità ad altri processi creando più flussi di esecuzione.

2.5 OTP Platform

La concorrenza sembra gestita quasi come un gioco in Elixir, ma ciò in genere non basta per semplificare lo sviluppo di un software, per questo Elixir supporta l'insieme di librerie OTP sviluppate per Erlang, con un Api più pulita e moderna rispetto al suo predecessore. L'OTP (Open Telecom Platform) è un insieme di librerie, strumenti e linee guida per sviluppare dei sistemi scalabili e resilienti in Erlang ed Elixir, basti pensare che per via dell'immutabilità dei dati e della natura funzionale del linguaggio, non possiamo avere variabili globali che mantengono uno stato, e per fare ciò, Elixir consente di dare la responsabilità nel mantenimento di uno stato ad un processo. Astrazioni come l'**Agent** o il **GenServer** ci consentono di mantenere uno stato senza dover reinventare la ruota nello scrivere un modulo soltanto per mantenere un semplice stato. L'approccio nella programmazione è totalmente differente e piuttosto singolare rispetto ai più comuni linguaggi di programmazione, ma è proprio questa singolarità che può portare dei vantaggi in alcune nel mondo concorrenziale.

2.5.1 GenServer

Si può volere più controllo rispetto alle primitive utilizzate per gestire la concorrenza, un OTP server è un modulo con il "Behaviour" GenServer. Il "Behaviour" è un meccanismo che consente di definire uno schema comune per un tipo specifico di processo.

Ad esempio il GenServer Behaviour definisce le funzioni e le interfacce necessarie per creare un processo server in grado di gestire le richieste in modo asincrono. Utilizzando il GenServer Behaviour, è possibile definire i comportamenti di base del server e personalizzarli secondo le esigenze specifiche dell'applicazione. Questo fornisce un alto livello di astrazione per la gestione dei processi e semplifica lo sviluppo di sistemi concorrenti e distribuiti in Elixir.

Il vantaggio di utilizzare un GenServer è che ha un'insieme di interfacce standard e include funzionalità di tracciamento e segnalazione degli errori. Si può anche mettere dentro un albero di supervisione.

Questo Behaviour astrae l'interazione Client-server, come si può vedere in figura 2.2 [6].

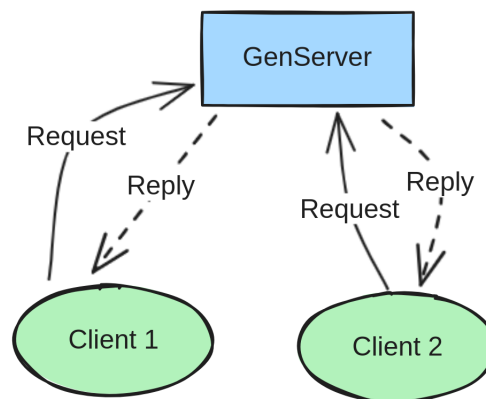


Figura 2.2: Interazione Client-Server

Per implementare il behaviour GenServer, bisogna affidarsi alla documentazione di GenServer, in particolare vanno ridefinite delle callback, ed ogni funzione può restituire un determinato insieme di strutture dati.

Nell'esempio 2.7 viene implementata una struttura dati per mantenere uno Stack di dati [6].

```
1 defmodule Stack do
2   use GenServer
3
4   # Client
5
```

```

6 def start_link(default) when is_binary(default) do
7   GenServer.start_link(__MODULE__, default)
8 end
9
10 def push(pid, element) do
11   GenServer.cast(pid, {:push, element})
12 end
13
14 def pop(pid) do
15   GenServer.call(pid, :pop)
16 end
17
18 # Server (callbacks)
19
20 @impl true
21 def init(elements) do
22   initial_state = String.split(elements, ",", trim: true)
23   {:ok, initial_state}
24 end
25
26 @impl true
27 def handle_call(:pop, _from, state) do
28   [to_caller | new_state] = state
29   {:reply, to_caller, new_state}
30 end
31
32 @impl true
33 def handle_cast({:push, element}, state) do
34   new_state = [element | state]
35   {:noreply, new_state}
36 end
37 end

```

Esempio 2.7: Implementazione Stack

Nell'esempio possiamo vedere che il modulo stack implementa la funzione **init/1** che inizializza lo stato con gli elementi iniziali quando il server viene avviato, la funzione **handle_call/3** chiamata per le operazioni di `:pop` dello stack, in particolare è una funzione sincrona, quindi viene usata quando ci si aspetta un valore di ritorno, infatti restituisce il valore di testa dello stack. La funzione **handle_cast/2** invece viene usata per le operazioni asincrone, quindi nel caso in esame per l'operazione di push nello Stack.

Quindi il GenServer è un'astrazione che:

- Incapsula un servizio condiviso.
- Mantiene uno stato.
- Permette un'astrazione concorrente ad un servizio condiviso [1].

2.5.2 Supervisor

Un Supervisor è un modulo che implementa il Behaviour Supervisor, sono processi specializzati per un solo scopo: monitorare altri processi che chiameremo d'ora in poi processi figli.

Sono questi Supervisor che ci semplificano lo sviluppo di applicazioni fault-tolerant, supervisionando i processi figli. Bisogna quindi decidere quali sono i processi figli da supervisionare, e una volta avviato il Supervisor con la funzione `start_link/2`, questo ha bisogno di sapere come avviare, fermare o riavviare i suoi figli in caso di errore o uscita imprevista.

Per questo i figli hanno bisogno di avere una funzione `child_spec/1` che definisce il comportamento del supervisore. I moduli che implementano lo GenServer, oppure un Agent automaticamente definiscono questa funzione, quindi non c'è bisogno di modificare il modulo. La funzione `child_spec/1` è una funzione che restituisce una Map per configurare il comportamento in caso di supervisione.

2.6 Integrazione con codice esterno

Elixir compilato sulla Erlang VM (BEAM) offre completa Interoperabilità con il codice erlang, avendo a disposizione tutta la gamma di librerie già sviluppate per Erlang, per questo molte librerie non sono disponibili per Elixir potendo usare quelle già ben testate sulla piattaforma Erlang. Non tutto può essere implementato in modo efficiente su questa piattaforma tramite Elixir o Erlang, basti pensare che l'immutabilità dei dati porta molti vantaggi per lo sviluppo concorrente, ma porta altri svantaggi. Nella seguente lista vengono messi in evidenza alcune lacune che presenta la Erlang VM.

- **Matematica avanzata:** La VM non è stata progettata per operazioni intensive di calcolo numerico. Se l'applicazione dipende dal calcolo di statistiche si possono riscontrare delle limitazioni.
- **Soluzioni algoritmiche tramite matrici:** L'implementazione nativa di matrici in Elixir usa liste di liste, che non è una rappresentazione efficiente delle matrici multidimensionali. Inoltre la mancanza di mutabilità porta a fare una copia di matrici anche grandi per semplici operazioni.
- **Command Line Applications:** La VM impiega circa 0.3 secondi per l'avvio e lo spegnimento anche per l'hardware moderno, dopotutto Erlang è stato progettato per sistemi a lungo termine, ciò non porta ad essere la scelta ottimale per una CLI.

Ci possono essere un sacco di insiemi di problemi in cui Elixir non è la soluzione migliore, ma può risultare utile comunque usare Elixir e risolvere sotto problemi con un codice esterno scritto in un altro linguaggio. In particolare Elixir può comunicare con altri linguaggi in 3 modi diversi:

- NIF (Native implemented functions): L'applicazione può condividere lo stesso spazio degli indirizzi di memoria. Una NIF è una funzione scritta in C/C++, viene compilata come libreria condivisa e caricata all'avvio dell'applicazione.
- Ports: Elixir può invocare processi esterni alla macchina virtuale, è una forma di Interprocess communication (IPC).
- Erlang Distribution Protocol: Si può comunicare con processi esterni anche su macchine differenti attraverso il protocollo di distribuzione fornito per distribuire un'applicazione su più nodi scalando orizzontalmente un sistema.

2.6.1 Interoperabilità tramite NIF

Le funzioni native NIF, permettono di caricare codice esterno nello stesso spazio di memoria della Erlang VM. Questa strategia di interoperabilità porta a dei benefici sulle performance elevati per alcune tipologie di calcolo come quello matematico. Erlang fornisce un'interfaccia [4] per poter scrivere funzioni C che possono essere chiamate direttamente da Elixir/Erlang.

Queste funzioni C devono essere compilate come librerie condivise, shared object (.so) in linux. Viene riportato nell'esempio 2.8 il codice NIF che stampa "Hello from C"[2].

```
1 #include "string.h"
2 #include "erl_nif.h"
3
4 static ERL_NIF_TERM hello(ErlNifEnv* env, int argc,
5                             const ERL_NIF_TERM argv[]) {
6     ErlNifBinary *output_binary;
7     enif_alloc_binary(sizeof "Hello from C", output_binary);
8     strcpy(output_binary->data, "Hello from C");
9     return enif_make_binary(env, output_binary);
10 }
11
12 static ErlNifFunc nif_funcs[] = {
13     {"hello", 0, hello},
14 };
15
16 ERL_NIF_INIT(Elixir.ElixirNif, nif_funcs, NULL, NULL, NULL, NULL)
17
```

Esempio 2.8: Funzione Nif

Si può compilare con il compilatore **gcc** con il seguente comando:

```
gcc -o <destination-directory>/hello.so -shared -fpic \\  
-I $ERL_ROOT/usr/include <file-directory>/hello.c
```

Una volta compilata la libreria si può caricare all'interno del modulo nel quale si vuole usare la funzione come nell'esempio 2.9.

```
1 defmodule ElixirNif do  
2   @on_load :load_nif  
3  
4   def load_nif do  
5     :ok = :erlang.load_nif(String.to_charlist("priv/elixir_nif"), 0)  
6   end  
7  
8   def hello do  
9     "Hello from Elixir"  
10  end  
11 end
```

Esempio 2.9: Caricamento NIF

L'annotazione `@on_load :load_nif` dice alla VM di eseguire `load_nif/0` all'avvio dell'applicazione. La funzione `hello` necessita di essere ridefinita anche in Elixir, così da permettere che la chiamata non fallisca nonostante qualche eventuale errore nel caricamento della NIF.

Nonostante è uno dei metodi più efficienti per eseguire codice C/C++, questo metodo ha degli svantaggi [4]:

- Una funzione nativa che va in crash, farà andare in crash l'intera VM, per questo è sconsigliato usare questo metodo se non strettamente necessario, questo rischio va contro la filosofia di Erlang nella costruzione di sistemi fault-tolerant.
- Una funzione nativa implementata erroneamente può causare un'inconsistenza dello stato interno della VM, che può portare al crash della VM o a comportamenti imprevisti della VM in qualsiasi momento dopo la chiamata alla funzione nativa.
- Una funzione nativa che viene eseguita a lungo degrada la responsività della VM e può portare a comportamenti strani, che possono portare a un utilizzo della memoria eccessivo.

2.6.2 Interoperabilità con Port

Il meccanismo Port è un'alternativa più sicura per integrare codice esterno. Ogni Port comunica con un processo esterno del sistema operativo. In questo caso se il Port termina, il codice Elixir viene notificato con un messaggio e

può agire di conseguenza continuando permettendo di avere un interazione robusta in confronto alle NIF. Per esempio un Segmentation fault in un processo che comunica tramite Port non porta alla terminazione della VM. Per questi motivi prima di integrare codice esterno tramite NIF conviene sempre prendere in considerazione il meccanismo Port prima, per non perdere i benefici dati dalla piattaforma Erlang.

La libreria **ErlPort**

Un caso di particolare interesse può essere la comunicazione tra Elixir e python, per consentire ai programmatori di Elixir di trarre vantaggio dalla vasta libreria offerta dalla grande community degli sviluppatori di Python. Proprio in quest'ambito la libreria **ErlPort** ci consente di avviare istanze python da Elixir, una libreria open source costruita grazie al meccanismo Port.

Performance - Test sperimentali

In questo capitolo si vuole capire quali potenzialità la piattaforma di Erlang/Elixir porta, in modo da poter capire quali siano le applicazioni possibili con questa tecnologia.

3.1.1 Hardware utilizzato

[illegible]

21

Capitolo 4

Conclusioni

Bibliografia

- [1] Bruce Tate Ben Marx José Valim. «Adopting Elixir». In: The Pragmatic Programmers, 2018. Cap. 5, p. 96.
- [2] Bruce Tate Ben Marx José Valim. «Adopting Elixir». In: The Pragmatic Programmers, 2018. Cap. 7, pp. 125–128.
- [3] *Elixir (programming language)* - *Wikipedia*. [https://en.wikipedia.org/wiki/Elixir_\(programming_language\)](https://en.wikipedia.org/wiki/Elixir_(programming_language)).
- [4] *Erlang – erl_nif*. https://www.erlang.org/doc/man/erl_nif.
- [5] *Erlang – Processes*. https://www.erlang.org/doc/efficiency_guide/processes.
- [6] *GenServer — Elixir v1.16.2*. <https://hexdocs.pm/elixir/GenServer.html>.
- [7] *HexDocs*. <https://hexdocs.pm/>.
- [8] Saša Juric. «Elixir in Action». In: 2019. Cap. 5.
- [9] *Lambda calcolo* - *Wikipedia*. https://it.wikipedia.org/wiki/Lambda_calcolo.
- [10] Dave Thomas. «Programming Elixir >= 1.6». In: The Pragmatic Programmers, 2018. Cap. 3, p. 23.