

Monografia - Linguagens Funcionais: Erlang e Scala

Bernardo Amorim - Num. USP: 9869659 e Fernando Buzato - Num. USP: 10405302

MAC 5742-0219 Introdução à Programação Concorrente, Paralela e Distribuída

1. Introdução

Nesta monografia iremos abordar os principais aspectos das linguagens de programação funcional e suas aplicações em computação concorrente, paralela e distribuída.

Para isso, vamos focar na implementação de duas linguagens: Erlang e Scala. Vamos explorar a história, as principais características e a aplicação destas linguagens no contexto da computação concorrente, paralela e distribuída.

2. Programação Funcional

A ideia de programação funcional é mais antiga do que os computadores eletrônicos. A sua base foi fundamentada no cálculo lambda de Alonzo Church, desenvolvido em 1930.

A primeira linguagem de programação funcional que surgiu foi a linguagem Lisp na década de 50. Outras linguagens tornaram-se popular durante o tempo, como por exemplo: Scheme, SML, Erlang, Haskell, OCaml e F#. ¹

Durante muito tempo as linguagens de programação funcional foram amplamente utilizadas no meio acadêmico e pouco utilizadas no meio comercial. Entretanto, recentemente houve um aumento no interesse sobre estas linguagens pelo meio comercial e pelos desenvolvedores em geral. ²

Atualmente os novos computadores e dispositivos móveis vêm utilizando cada vez mais processadores com multicore para aumentar a capacidade de processamento. Para que os sistemas e programas tirem maior proveito deste aumento da capacidade de processamento, é necessário que estes programas estejam preparados para trabalharem com paralelismo e concorrência.

As programação funcional ajuda principalmente nestas questões pois o paradigma permite trabalhar melhor com dados imutáveis e isto ajuda na

¹ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. Programming in Scala. 3 ed. Mountain View: Artima, 2016

²PAYNE, Alex; WAMPLER, Dean. Programming Scala. 2 ed. O'Reilly Media, Inc, 2014

hora de implementar um programa ou um sistema que tenha que executar uma tarefa paralela ou concorrente.

2.1. Principais Aspectos

Os principais aspectos da programação funcional são a pureza e a imutabilidade:

Pureza Semelhante a uma função matemática, um código funcional tem a sua saída condicionada exclusivamente à sua entrada, de modo que uma entrada garanta sempre a mesma saída após a execução do mesmo método, eliminando os chamados side-effects, que são as mudanças de estado ocasionadas por outros fatores que não dependam dos parâmetros do método. Quando a função possui este comportamento, podemos chamar esta função de **pura**, ou seja, ela opera apenas com os parâmetros de entrada da função.

Imutabilidade A palavra "Variável" em programação funcional tem um novo significado. Variáveis em programação funcional são **imutáveis** diferente das variáveis em programação imperativa que são mutáveis. Isto é outra consequência da orientação matemática.

Quando temos uma expressão $y = \sin(x)$, assim que associamos o resultado de $\sin(x)$ a y , y torna-se uma constante e não é mais alterado.³

2.2. Aplicação em computação concorrente e paralela

Os aspectos citados acima facilitam a implementação de programas e sistemas concorrentes e paralelos. A combinação de funções puras e variáveis imutáveis fazem com que a programação funcional seja utilizada como uma melhor forma para se escrever softwares concorrentes.

A imutabilidade elimina um dos problemas mais difíceis quando falamos em concorrência: compartilhamento de memória e mutação de estados. Quase toda a dificuldade de se trabalhar com multithreads em um programa reside na sincronização de acesso a memória compartilhada e na mudança de estados. Se a mutabilidade é removida, este tipo de problema tende a desaparecer do programa.

Esta combinação também proporciona uma drástica diminuição de bugs nos softwares. Estados mutáveis são as fontes de bugs mais fatais que os softwares podem ter. São os bugs mais difíceis de serem detectados antes do deploy em produção e são os mais difíceis de se consertar.

³PAYNE, Alex; WAMPLER, Dean. Programming Scala. 2 ed. O'Reilly Media, Inc, 2014

3. Linguagem Scala

3.1. História

A linguagem Scala foi desenvolvida em 2001 por Martin Odersky e pelo grupo dele na Escola Politécnica Federal de Lausane, na Suíça. O objetivo de Odersky era criar uma linguagem que combinasse o paradigma de Orientação a Objetos com programação funcional.⁴

Portanto, trata-se de uma linguagem multiparadigma que incorpora recursos da linguagem Orientada a Objetos e recursos da linguagem de programação funcional.

O nome Scala refere-se ao acrônimo de "Scalable Language". Segundo Odersky, a linguagem foi criada para escalar de acordo com as necessidades dos usuário/desenvolvedores, ou seja, a linguagem proporciona mecanismo para que o desenvolvedor crie desde simples scripts até sistemas altamente complexos.⁵

Anteriormente Odersky havia trabalhado na criação do compilador Java e na criação do Generics em Java. A ideia foi criar uma Linguagem totalmente interoperável com Java e portanto Scala roda sobre a JVM padrão do Java.

A linguagem vem recebendo cada vez mais adeptos no mercado. Empresas como LinkedIn, Twitter, Tumblr, Foursquare e Apache já utilizam o Scala em seus principais sistemas. A Apache por exemplo criou o Apache Spark utilizando Scala como principal linguagem.

3.2. Características

Concisão A linguagem Scala, quando comparada com outras linguagens orientadas a objetos padrões de mercado como Java, C++ ou C#, possui uma sintaxe bem mais concisa permitindo ao programador a criar programas com menos linhas de código.

O fato da linguagem possuir uma sintaxe mais concisa não quer dizer que é mais complexa de entendimento. Na linguagem Scala é possível escrever códigos mais concisos e muitas vezes mais claros do que nas linguagens convencionais.

Abaixo segue um exemplo de uma Classe escrita em Java:

⁴PAYNE, Alex; WAMPLER, Dean. Programming Scala. 2 ed. O'Reilly Media, Inc, 2014

⁵ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. Programming in Scala. 3 ed. Mountain View: Artima, 2016

```

1 package seminario;
2
3 public class Pessoa {
4
5     private String nome;
6     private String sobrenome;
7     private Integer idade;
8
9     public Pessoa(String nome, String sobrenome, Integer idade) {
10         super();
11         this.nome = nome;
12         this.sobrenome = sobrenome;
13         this.idade = idade;
14     }
15
16     public String getNome() {
17         return nome;
18     }
19
20     public String getSobrenome() {
21         return sobrenome;
22     }
23
24     public Integer getIdade() {
25         return idade;
26     }
27
28 }
29 }

```

Figura 1: Classe Pessoa.java

É possível verificar na imagem que para criar a classe Pessoa em java, apenas com os métodos getters desta classe foram necessários 29 linhas de código.

Abaixo segue um exemplo da mesma Classe anterior escrita em Scala:

```

1 package seminario
2
3 class Pessoa (val nome: String, val sobrenome: String, val idade: Int)

```

Figura 2: Classe Pessoa.scala

Em Scala escrevemos a mesma classe com apenas 3 linhas de código. O compilador já se encarrega em criar o construtor da classe e os métodos getters automaticamente.

Tipagem Estática e Inferência de Tipos Scala é uma linguagem de tipagem estática, ou seja, os tipos das variáveis são conhecidos em tempo de compilação assim como nas linguagens tradicionais como Java, C e C ++.

Além disso, a linguagem permite a inferência de tipos, ou seja, não é necessário o desenvolvedor declarar explicitamente os tipos de variáveis.

Variáveis Imutáveis Scala permite que o desenvolvedor trabalhe com variáveis Imutáveis. Trabalhar com variáveis imutáveis junto com a aborda-

gem da programação funcional permite que os softwares desenvolvidos em Scala estejam preparados para evitar problemas de concorrência.

Scala também permite ao desenvolvedor trabalhar com variáveis mutáveis. No entanto, esta abordagem não é recomendada pois dependendo do software que está sendo desenvolvido, utilizar variáveis mutáveis pode trazer problemas de concorrência para a aplicação. Portanto, sempre que for possível, é recomendado trabalhar com variáveis imutáveis.

Orientação a Objetos e Programação Funcional A linguagem Scala permite que o desenvolvedor trabalhe com todos os recursos da programação Orientada a Objetos: É possível trabalhar com Classes, Interfaces (Traits), Herança, Polimorfismo, Sobrecarga, etc.

Em Scala tudo é tratado como um objeto. Tipos primitivos, valores e funções são objetos.

A linguagem também permite que o desenvolvedor trabalhe com todos os recursos da programação funcional: É possível trabalhar com Funções de Ordem Superior, Funções Puras, Funções Anônimas (Lambdas), Closures, Currying, etc.

A linguagem permite o desenvolvedor trabalhar com os dois paradigmas ao mesmo tempo. É possível criar sistemas utilizando Orientação a Objetos e Programação Funcional.

3.3. Aplicação em computação concorrente, paralela e distribuída

Quando falamos de concorrência, paralelismo e distribuição em Scala, a linguagem fornece basicamente três abstrações para o desenvolvedor: Parallel Collections Library, Futures e Actors. A linguagem também permite o desenvolvedor trabalhar com as bibliotecas padrões de concorrência e paralelismo do Java devido a compatibilidade do Scala com a linguagem Java, porém, não vamos abordar este assunto na monografia.

Parallel Collections Library Scala como outras linguagens possui bibliotecas para trabalhar com coleções de objetos como Listas, Vetores, Mapas, etc. Porém, a linguagem criou uma abstração sobre estas bibliotecas para permitir o desenvolvedor a trabalhar com operações de forma paralela sobre os elementos destas bibliotecas de forma transparente.

No exemplo abaixo, criamos um vetor de 10 elementos e executamos a impressão de cada um dos elementos na tela. Na primeira execução, utilizamos o `foreach` de forma sequencial para imprimir os elementos do vetor. Na segunda execução, utilizamos o método *par* antes do `foreach`. Este método é uma abstração do Parallel Collection que diz que cada elemento será executado de forma paralela.

```

1 package seminario
2
3 object aplicacao extends App {
4
5   val v = Vector.range(0, 10)
6
7   println("-----Execução Sequencial-----")
8   v.foreach(println)
9   println("-----")
10
11  println("-----Execução Paralela-----")
12  v.par.foreach(println)
13  println("-----")
14 }

```

Figura 3: Exemplo Parallel Collections

Desta forma, na segunda execução, é possível observar que os elementos foram impressos de forma aleatória de acordo com a execução de cada thread independente.

```

|-----Execução Sequencial-----
0
1
2
3
4
5
6
7
8
9
-----
-----Execução Paralela-----
5
0
2
6
3
4
7
8
9
1
-----

```

Figura 4: Resultado da Execução

Basicamente, quando o desenvolvedor utiliza uma parallel collection, a JVM cria um `ExecutionContext`, que é uma espécie de Thread Pool, e distribui o processamento dos elementos neste `ExecutionContext`. Por padrão, um `ExecutionContext` cria uma thread para cada núcleo do processador que está executando a JVM⁶. Por exemplo, se a JVM está rodando em um computador com um processador de 4 CPUs, será criado um `ExecutionContext` com 4 threads.

⁶PROKOPEC, Aleksandar. Learning Concurrent Programming in Scala. PACKT, 2014

Futures Futures é uma abstração da linguagem Scala para trabalhar com concorrência e paralelismo proporcionando uma maneira de executar operações em paralelo de forma assíncrona e não bloqueante.

Futures funcionam como uma espécie de objeto reservado na memória que é criado para um resultado que ainda não existe.⁷

Além das Futures proporcionarem uma maneira de executar operações paralelas de forma assíncrona e não bloqueante, é possível também combinar o resultado de várias Futures em uma única Future através de funções de Map, Reduce e For-Comprehensions.

No exemplo abaixo, criamos três Futures para realizar a multiplicação por 2 de números que são passados como parâmetros para a função *multiplicaPorDois*. Estas 3 Futures posteriormente são combinadas em uma Future Resultado que realiza a soma do resultado das 3 Futures de multiplicação. A combinação das 3 Futures é realizada através da função *For*.

```
7@object TesteFuture extends App{
8
9@  def delta = System.currentTimeMillis - startTime
10@ def time = System.currentTimeMillis
11@ def sleep(time: Long): Unit = Thread.sleep(time)
12
13@ def multiplicaPorDois(x: Int, delay: Int, name: String): Future[Int] = Future {
14   println(s"Future $name")
15   sleep(delay)
16   x * 2
17 }
18
19
20 val startTime = time
21
22
23 // Future #1
24 val f1 = multiplicaPorDois(x=1, delay=10000, name="f1")
25
26 // Future #2
27 val f2 = multiplicaPorDois(x=2, delay=10000, name="f2")
28
29 // Future #3
30 val f3 = multiplicaPorDois(x=5, delay=10000, name="f3")
31
32
33@ val resultado = for {
34   r1 <- f1
35   r2 <- f2
36   r3 <- f3
37 | } yield (r1 + r2 + r3)
38
39 resultado.onComplete {
40   case Success(x) => {
41     println(s"\nResultado = $x (tempo total(ms): = $delta)")
42   }
43   case Failure(e) => e.printStackTrace
44 }
45
46 val sync = Await.result(resultado, Duration.Inf)
47
48 }
```

Figura 5: Exemplo Futures

Assim como o Parallel Collections, Futures trabalha com ExecutionContext. Portanto, por padrão, é criado um ThreadPool com o mesmo número

⁷THE SCALA PROGRAMMING LANGUAGE. em: <https://www.scala-lang.org/>. Acesso em 18 de jun. 2017

de Threads que o número de CPUs que a JVM está sendo executada. A computação das Futures são distribuídas nestas Threads.

Actors Actors é uma abstração da linguagem Scala para trabalhar com concorrência, paralelismo e distribuição. Esta abstração foi criada baseada no Actor-Model (Modelo matemático para concorrência criado em 73 por Carl Hewitt)⁸.

Este modelo basicamente trata um Actor como um objeto que recebe uma mensagem e toma uma ação sobre aquela mensagem. A ação pode ser processar uma mensagem internamente, encaminhar uma mensagem para outro Actor, enviar uma nova mensagem, ou criar novos Actors para processar a mensagem recebida. Um Actor trabalha com uma mensagem de cada vez e sem troca de contexto.

Até a versão 2.10 da linguagem Scala, havia uma implementação para trabalhar com Actors dentro da biblioteca padrão da linguagem. Apartir da versão 2.11 a implementação de Actors saiu da biblioteca padrão da linguagem e apartir de então, Scala começou a adotar o framework Akka⁹ como padrão para a implementação de Actors.

O framework Akka foi criado pela mesma empresa que mantém o Scala e pode ser utilizado tanto em Scala quanto em Java.

O Akka é um toolkit para a construção de aplicações que trabalham com alta concorrência, distribuição e tolerância a falhas.

O Akka possui vários módulos, entre os principais:

Actor(Core): Biblioteca principal do Akka que é o core para a criação dos Atores e do sistema para controle destes atores. Esta biblioteca é utilizada em todos os módulos do Akka.

Remoting: Módulo do Akka que permite que Atores que estão sendo executados remotamente em diferentes máquinas e JVMs consigam trocar mensagens entre eles.

Cluster: Enquanto o Remoting permite os Atores remotos se comunicarem, o módulo Cluster organiza estes atores em um "Meta-Sistema", permitindo uma organização centralizada dos Atores.

⁸PROKOPEC, Aleksandar. Learning Concurrent Programming in Scala. PACKT, 2014

⁹AKKA. em <http://akka.io/docs/>. Acesso em 18 de jun. 2017

Distributed Data: Módulo que permite o compartilhamento de dados entre nós e atores distribuídos em clusters.

Streams: Módulo que permite os atores processarem Streams de informação.

Abaixo temos um exemplo em que foram criados três atores para realizar a multiplicação de um determinado número por 2. Estes atores, após multiplicar o número por 2, enviam uma mensagem para um ator sumarizador que é responsável por somar o resultado de cada ator multiplicador.

```
1 import akka.actor.Actor
2
3 object TesteActor2 extends App {
4   def delta = System.currentTimeMillis - startTime
5   def time = System.currentTimeMillis
6
7   case class MultiplicaPorDois (n:Int)
8   case class Sumariza (n:Int)
9   class MultiplicadorActor (sumarizadorActor: ActorRef) extends Actor {
10     def receive = {
11       case MultiplicaPorDois (n) =>
12         println(self)
13         println(n)
14         Thread.sleep(10000)
15         sumarizadorActor ! Sumariza (n * 2)
16     }
17   }
18   class SumarizadorActor extends Actor {
19     var sum = 0
20     def receive = {
21       case Sumariza (n) =>
22         println(self)
23         sum = sum + n
24         println(s"Valor Total Somado: $sum")
25         println(s"Tempo Total de Execução: $delta")
26     }
27   }
28
29   val system = ActorSystem("MultiplicaESomaSystem")
30   val sumarizadorActor: ActorRef = system.actorOf(Props(new SumarizadorActor), "sumarizadorActor")
31   val actorMultiplica1: ActorRef = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica1")
32   val actorMultiplica2: ActorRef = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica2")
33   val actorMultiplica3: ActorRef = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica3")
34
35   val startTime = time
36
37   actorMultiplica1 ! MultiplicaPorDois(1)
38   actorMultiplica2 ! MultiplicaPorDois(2)
39   actorMultiplica3 ! MultiplicaPorDois(5)
40 }
41 }
```

Figura 6: Exemplo Actors

No exemplo, é possível identificar três elementos principais do framework: ActorSystem, ActorRef e o símbolo ! (que representa o envio das mensagens).

O ActorSystem é responsável por controlar a criação de todos os atores. Também é responsável por toda a configuração do ambiente onde os atores serão criados.

O ActorRef é uma referência para um ator. Um ator nunca se comunica com outro diretamente. Sempre é necessário utilizar o ActorRef para esta comunicação.

O símbolo ! como dito anteriormente é o símbolo que representa o envio da mensagem por um ator. Este símbolo em específico significa que o ator

envia uma mensagem e não fica aguardando uma resposta (fire and forget).

Por baixo dos panos, quando um ActorSystem é iniciado, o ActorSystem cria um Dispatcher, que funciona encima de uma estrutura de um ExecutionContext que por sua vez representa um ThreadPool. Desta forma, quando uma mensagem é enviada através de um ActorRef, esta mensagem é enviada para o Dispatcher que é responsável por criar o ator que vai processar a mensagem no ThreadPool e enviar a mensagem para aquele ator que foi criado. O Dispatcher do Akka pode criar mais de um ator por thread dentro de um ThreadPool.

Cada ator que é criado possui uma estrutura chamada de Mailbox que funciona como se fosse uma fila. Portanto, cada mensagem que o Dispatcher envia para o ator é incluída no Mailbox do ator. Desta forma, o ator processa uma mensagem de cada vez, retirando as mensagens da fila a cada processamento realizado.

4. Linguagem Erlang

4.1. História

A linguagem Erlang foi criada por Joe Armstrong e outros engenheiros da *Ericsson* para ser utilizada em sistemas de telecomunicação. Sendo tais sistemas altamente distribuídos por natureza, os projetistas procuraram fazer uma linguagem que possuísse excelentes implementações e interfaces para programação concorrente.

O que era para ser uma linguagem de nicho tornou-se uma referência em programação de sistemas distribuídos, com sua implementação original em *Prolog* sendo substituída por uma open-source em *C*.

4.2. Características

4.2.1. Visão geral

Uma boa visão panorâmica foi dada pelo próprio criador da linguagem, Joe Armstrong, em sua tese de doutorado¹⁰. Resumidamente, em Erlang:

- Tudo é um processo.
- Processos são fortemente isolados.
- A criação e destruição de processos é uma operação leve.
- Troca de mensagens é a única maneira pela qual processos interagem.

¹⁰Esta visão foi retirada do verbete sobre Erlang da Wikipedia em inglês. Link: [https://en.wikipedia.org/wiki/Erlang_\(programming_language\)](https://en.wikipedia.org/wiki/Erlang_(programming_language))

- Processos possuem nomes únicos
- Se você sabe o nome de um processo, pode enviar-lhe uma mensagem.
- Processos não compartilham recursos.
- O gerenciamento de erros é não-local.
- Processos fazem o que devem fazer ou falham.

4.2.2. *Tipos*

Erlang possui tipagem forte e dinamicamente checada. São dez os tipos de dados primitivos, nomeados aqui segundo a terminologia oficial da linguagem, em inglês:

- *Atoms*: Átomos são utilizados para especificar valores distintos através de uma string, que deve ser iniciada com uma letra minúscula.
- *Binaries*: Criado como uma maneira de armazenamento e manipulação eficiente de dados, este é um título importante em Erlang, dada sua aplicação original em sistemas de telecomunicação.
- *Floats*: Representam números reais, segundo a especificação IEEE 754 para 64 bits.
- *Funs*: Funções, sendo estas cidadãs de primeira classe na linguagem.
- *Integers*: Representam inteiros cujos tamanhos são restritos apenas pela memória do sistema.
- *Lists*: Coleção de tamanho variável de elementos de quaisquer tipos. Como em qualquer linguagem funcional, são muito utilizadas pela naturalidade com que interagem com recursões.
- *Pids*: Símbolos utilizados para identificar os diferentes processos criados pela máquina virtual de Erlang (não possuem correspondência com quaisquer identificadores de processos do sistema operacional).
- *Ports*: Portas são utilizadas como meio de comunicação com agentes externos, sendo esta comunicação feita através da troca de mensagens conformantes ao "port protocol".
- *References*: Criadas pela função nativa *make_ref*, são símbolos utilizados como referência, graças à unicidade em sua criação (a expressão *make_ref() == make_ref()* sempre é avaliada como falsa).
- *Tuples*: Coleção imutável de elementos de quaisquer tipos.

4.3. Aplicação em computação concorrente, paralela e distribuída

4.3.1. BEAM, a máquina virtual de Erlang

Uma linguagem com suporte a sistemas distribuídos verdadeiramente escaláveis e tolerantes a falhas deve possuir processos o mais leve possíveis: para que eventuais corrupções em agentes do sistema não se propaguem ao sistema inteiro, estes agentes devem ser eliminados rapidamente. Adicionalmente, não se pode permitir que processos ocupem qualquer memória além daquela estritamente necessária.

Dado que processos simples criados pela maioria dos sistemas operacionais podem ocupar até alguns MBs, o projetista de linguagens concorrentes não pode recorrer ao sistema operacional para a criação e gerenciamento de novos processos. Para contornar esta problemática, Armstrong e seus colegas recorreram à criação da BEAM, a máquina virtual que hospeda os processos de Erlang.

Possuindo *task scheduler* próprio, a BEAM cria e gerencia todos os processos de Erlang de uma determinada aplicação, ocupando apenas um processo do sistema operacional, ao mesmo tempo em que utiliza todos os *cores* do ambiente no qual a aplicação é executada. Um processo criado pela BEAM pode chegar a ocupar na memória apenas 300 palavras da CPU (o que equivale a 1200 bytes, numa arquitetura de 32 bits).

4.4. A Interface

Uma implementação de processos enxuta não é o suficiente para garantir o sucesso duma linguagem concorrente - também é crucial a escolha da interface pela qual fornecer-se-á a funcionalidade concorrente. Para os criadores de Erlang, ficou claro que qualquer compartilhamento de memória reduziria tanto a eficiência computacional quanto a tolerância a falhas dos processos da linguagem - locks e outras formas de mutexes não eram uma possibilidade. Optaram, por fim, por agentes com memória exclusiva que interagem unicamente por meio de mensagens: cada processo possui uma caixa de correio, na qual podem ser deixadas mensagens por outros processos. Esta escolha revelou-se fortuita - possuindo apenas 3 funções, a interface de concorrência de Erlang tornou-se um exemplo a ser seguido (é a arquitetura básica por trás do paradigma web de microserviços, por exemplo). Estas funções são:

- **spawn(Mod, Func, Args)** : cria um processo no qual será executada a função *Func(Args[1], Args[2], ..., Args[n])* do módulo *Mod*, sendo *Args* uma lista de tamanho *n*
- **Pid ! Msg** envia a mensagem *Msg* ao processo de pid *Pid*, podendo esta ser qualquer valor de Erlang. A função retorna imediatamente, não bloqueando o processo remetente.

- **receive**

$pattern_1 -> action_1$

$pattern_2 -> action_1$

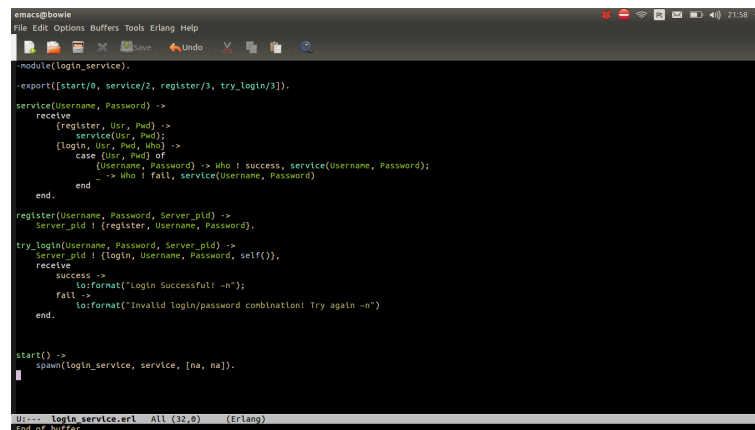
...

$pattern_n -> action_n$

end: Checa a caixa de mensagens do processo no qual o comando é executado, executando a ação dentre $\{action_1, action_2, \dots, action_n\}$ correspondente ao primeiro padrão $\{pattern_1, pattern_2, \dots, pattern_n\}$ que seja condizente com alguma das mensagens da fila, sendo dada prioridade a mensagens que chegaram antes. Caso não haja mensagem compatível disponível, o processo é bloqueado, aguardando novas mensagens.

Além de possibilitar a tolerância a falhas, a troca de mensagens tem como mérito tornar mais explícitos eventuais erros no código do sistema, não exigindo o treinamento geralmente necessário ao programador para identificar eventuais falhas lógicas no sistema concorrente que está desenvolvendo.

O exemplo abaixo ilustra a programação concorrente em Erlang, implementando um (extremamente) simples serviço de login.



```

emaci@bowie
file Edit Options Buffers Tools Erlang Help
- module(login_service).
- export([start/0, service/2, register/3, try_login/3]).

service(Username, Password) ->
  receive
    (register, User, Pwd) ->
      service(User, Pwd);
    (login, User, Pwd, Who) ->
      case (User, Pwd) of
        (Username, Password) -> Who ! success, service(Username, Password);
        _ -> Who ! fail, service(Username, Password)
      end
  end.

register(Username, Password, Server_pid) ->
  Server_pid ! {register, Username, Password}.

try_login(Username, Password, Server_pid) ->
  Server_pid ! {login, Username, Password, self()}.
  receive
    success ->
      io:format("Login Successful! ~n");
    fail ->
      io:format("Invalid login/password combination! Try again ~n")
  end.

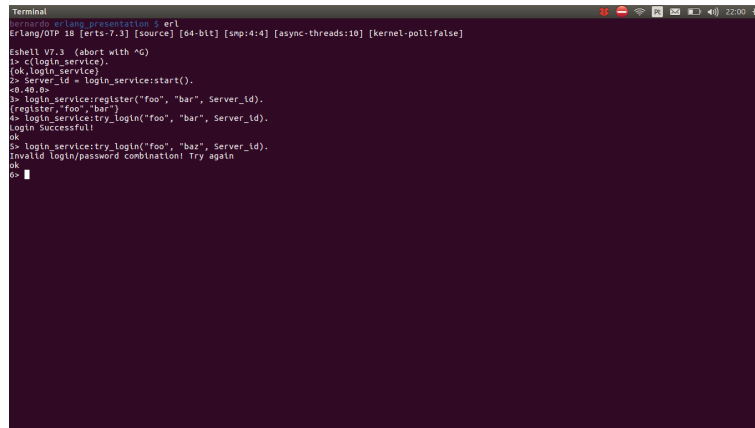
start() ->
  spawn(login_service, service, [na, na]).

U:--- login_service.erl All (32,0) (Erlang)
End of Buffer

```

Figura 7: Serviço de Login em Erlang

A captura do terminal abaixo demonstra o funcionamento do serviço.

A terminal window with a dark purple background and white text. The text shows the execution of Erlang commands to start and use a login service. The commands include starting the service, registering a user, and attempting logins with correct and incorrect credentials. The output shows the service starting successfully, the user being registered, and the login attempt with 'foo' and 'bar' being successful, while the attempt with 'foo' and 'baz' fails.

```
Terminal
bernardo@erlang_presentation:~$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [snmp:4:4] [async-threads:10] [kernel-poll:false]
Eshell V7.3 (abort with ^C)
1> c(login_service).
(ok,login_service)
2> Server_id = login_service:start().
<0.40.0>
3> login_service:register("foo", "bar", Server_id).
(register "foo","bar")
4> login_service:try_login("foo", "bar", Server_id).
login successful!
ok
5> login_service:try_login("foo", "baz", Server_id).
Invalid login/password combination! Try again
ok
6>
```

Figura 8: Utilização do Serviço de Login em Erlang

5. Conclusões

Através dos estudos realizados é possível concluir que o uso das linguagens de programação funcional vem tomando cada vez mais importância no mercado devido aos benefícios que este paradigma proporciona para trabalhar com concorrência e paralelismo. Além disso, cada vez mais os novos processadores vem aumentando o número de cores para aumentar o poder de processamento. Assim, é necessário que os softwares explorem cada vez mais a concorrência e o paralelismo para se beneficiarem ao máximo destes novos processadores.

Em relação a linguagem Scala, é possível observar que é uma linguagem que vem ganhando cada vez mais adeptos pelo fato da linguagem proporcionar uma fácil adaptação ao programador que está migrando da programação imperativa, orientada a objetos, para a programação funcional. Além disso, trata-se de uma linguagem que fornece vários recursos para que o programador possa explorar ao máximo a concorrência, paralelismo e distribuição.

Erlang aliena-se da subjetividade inerente a discussões sobre os méritos de linguagens, especialmente no tocante à abordagem de concorrência - é difícil encontrar neste quesito argumento mais forte e objetivo do que uma diminuição em dezenas de vezes da memória utilizada pelos processos da linguagem. A utilização de mensagens como suporte exclusivo à concorrência, possibilitando assim sistemas tolerantes a falhas e transparência ao usuário quanto à localização exata dos processos com os quais seu sistema interage, soma-se ao mais objetivo primeiro argumento para estabelecer a apropriação de Erlang à programação concorrente como algo consensual entre programadores. À medida que o número de cores de CPUs for crescendo, tornando assim mais lenta a execução de programas sequenciais, é de se esperar que a

linguagem conquiste um espaço compatível com sua beleza e eficiência.