# Haskell and other Functional Languages: Clojure

Isabella Mendonça, Italo Moraes e Lucas Romão

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

## 1. Introdução

Clojure é uma linguagem de tipagem dinâmica executada na JVM que oferece um vasto número de grandes *features* e características aplicáveis para resolver modernos desafios computacionais.

No presente trabalho, discutiremos, inicialmente, um breve histórico de sua implementação, além de suas principais características. Discutiremos também como o paralelismo foi aplicado na linguagem, apresentaremos alguns exemplos de códigos e uma análise comparativa com outras linguagens, avaliando sua implementação e desempenho.

## 2. Breve histórico

Rich Hickey é o criador do Clojure, um dialeto baseado em Lisp. Antes do Clojure, Rich desenvolveu outras linguagens, como dotLisp (um projeto similar baseado no *framework* .NET), além de tentativas de criar algo entre o Lisp e o Java.

Com esse histórico, Hickey elaborou o Clojure, linguagem capaz de operar em mecanismos do JavaScript, na Common Language Runtime (CLR) e na Java Virtual Machine (JVM). Muito referenciada como uma linguagem de propósito geral, o Clojure foi criado com evidente foco no conceito de linguagens funcionais e própria para concorrência.

Apesar de ser uma linguagem relativamente nova, cujo lançamento ocorreu em 2007, já existem grandes empresas que utilizam Clojure em algum de seus processos, entre elas o Walmart, Nubank, Heroku, Netflix e Spotify.

## 3. Utilização

Clojure segue todos os princípios das linguagens funcionais que são valores imutáveis, funções como cidadãos de primeira classe e possuem *High Order Function*. Cada um desses princípios será explicado nas subseções que seguem junto com exemplos de código para exemplificar sua utilização.

### *3.1. Valores imutáveis*

Em Clojure, qualquer valor declarado é imutável, diferente das linguagens imperativas onde os valores são constatemente alterados ao longo de um programa. Valores imutáveis são o que garantem que funções escritas em linguagens funcionais não produzam efeitos colaterais uma vez que os valores anteriormente definidos não são afetados depois de uma chamada de função.

Valores imutáveis apresentam uma grande utilidade em programas *multi-thread* por evitar problemas de concorrência causados por leitura e escrita de um mesmo valor por diferentes *threads* pois para que uma *thread* escreva neste valor será necessário criar uma cópia local deste ou uma nova referência a partir do original, em ambos os casos o valor original é preservado.

Em Clojure o conceito de imutabilidade se extende também a suas estruturas de dados como listas, vetores, conjuntos e mapas. Entretanto tais estruturas de dados costumam ser utilizadas justamente em casos onde são alteradas conforme a execução do programa.

Neste caso, para adicionar ou remover elementos de uma coleção é necessário criar uma nova a partir da original. Esse processo seria custoso se feito trivialmente, uma vez que seria necessário um consumo de tempo linear para remover ou adicionar um elemento ao invés do tempo constante. Para garantir a imutabilidade e, ainda sim, a criação de novas coleções em tempo constante, as coleções em Clojure possuem a propriedade da persistência. Essa propriedade garante que a versão anterior da coleção ainda fica disponível após a mudança e as operações triviais têm sua performance garantida.

Com a persistência, as versões novas não são feitas a partir duma cópia completa da original, mas são implementadas utilizando estruturas de dados ligadas para que possam compartilhar a estrutura da versão original.

O código abaixo exemplifica a imutabilidae em estruturas.

```
1  (let [my-vector [1 2 3 4]
2        my-map {:fred "ethel"}
3        my-list (list 4 3 2 1)]
4    (list
5      (conj my-vector 5)
6      (assoc my-map :ricky "lucy")
7      (conj my-list 5)
8      ;the originals are intact
9      my-vector
10     my-map
```

```
11      my-list))
12  -> ([1 2 3 4 5] {:ricky "lucy", :fred "ethel"} (5 4 3
    ↪  2 1) [1 2 3 4] {:fred "ethel"} (4 3 2 1))
```

Nas linhas de 1 a 3, declara-se um vetor, um mapa e uma lista respectivamente. Nas linhas 5 a 7 essas coleções são modificadas com a adição de um novo elemento em cada uma, e são colocadas numa lista junto com as originais. Na saída do código vê-se que as referências originais não foram alteradas.

Entretanto, apesar das vantagens da imutabilidade, principalmente em programas com computações em paralelo, tem casos onde é necessária realizar mutações em casos de paralelismo e computação concorrente. Para isso Clojure possui o conceito de identidade que são entidades que em dado momento fazem referência a um valor, este por sua vez imutável. Ao modificar uma identidade, é mudado o valor para o qual ela faz referência, mas assim como no caso das coleções, o valor originalmente referenciado ainda pode ser obtido revertendo as operações feitas. Os três tipos de identidade são *atoms, agents* e *refs*. Estes tipos serão melhor detalhados na seção de Paralelismo.

### *3.2. Funções como cidadãos de primeira classe e High Order Functions*

Em linguagens funcionais, funções são cidadãos de primeira classe, o que significa que elas são valores como qualquer outro, podendo ser armazenadas em variáveis, passadas como parâmetros de funções ou ser o valor de retorno das mesmas.

Funções que recebem funções como parâmetro ou que retornam uma função são chamadas de *High Order Function* (Funções de Ordem Superior em português). Por conveniência, nos parágrafos a seguir iremos nos referir a tais funções apenas como **HOF**.

Um exemplo de HOF é a função `map` que recebe uma coleção e uma função que será aplicada a cada um dos valores e retorna o resultado dessa operação.

No exemplo código a seguir, é imprimido uma mensagem `"Hello nome"` onde nome é passado como parâmetro para uma função anônima.

```
(map #(str "Hello " % "!" ) ["Ford" "Arthur"
 ↪  "Tricia"])
;;=> ("Hello Ford!" "Hello Arthur!" "Hello Tricia!")
```

Uma função anônima é declarada da forma

```
#(% faz algo com %)
```

Para realizar o mesmo `map` em uma linguagem imperativa, por exemplo C, é necessário mais linhas de código.

```c
void map(char* (*f)(char *), char *items[], size_t n)
 ↪ {
    for (int i = 0; i < n; i++)
        items[i] = f(items[i]);
}

char * f(char *val) {
    return strcat("Hello ", val);
}

int main() {
    char *list[] = {"Ford", "Arthur", "Tricia"};
    map(f, list, 3);
    return 0;
}
```

É necessário primeiramente definir uma função que fará o `map` e outra função para aplicar a concatenação. Enquanto em Clojure não é necessário criar a função explicitamente, além de já possuir a função `map` já implementada.

No exemplo abaixo, é mostrado a implementação de uma função que retorna outra função.

```clojure
(defn inc-maker
        "Create a custom incrementor"
    [inc-by]
    #(+ % inc-by))

(def inc3 (inc-maker 3))
(inc3 7)
; => 10
```

Neste exemplo, é criada uma função que recebe um valor e retorna uma função anônima que recebe um segundo valor e soma com o primeiro . Em seguida, guarda-se o valor de `inc-maker` recebendo 3 como parâmetro em `inc3`, com isso tem-se em inc3 uma função que recebe um parâmetro e soma 3 a ele. Por último, é `inc3` é chamada com o número 7.

Note que na função mais interna os valores definidos no escopo da função pai ainda estão disponíveis. Tal propriedade é chamada de *closure* onde uma

função é criada junto com todo o escopo definido anterior à sua criação. Importante ressaltar que sem o uso de *closure* não seria possível acessar o valor de `inc-by` pois a função é chamada de fora de seu escopo de criação.

### *3.3. Recursão ao invés de loops*

Outro princípio de linguagens funcionais é a ausência de estruturas de loops como *repeat, for* e *while*, e, ao invés disso, utiliza-se recursão para iterar sobre uma coleção. No exemplo abaixo mostramos uma implementação de uma soma em um vetor em Javascript e em Clojure.

```
var vector = [1, 2, 3, 4, 5];
var sum = 0;

for (i = 0; i < vector.length; i++) {
        sum += vector[i];
}

console.log(sum);

; => 15
```

Já em Clojure seria necessário o uso da recursão para poder fazer a soma.

```
(defn sum [a-seq]
    (if (?empty a-seq)
        0
        (+ (first a-seq) (sum (rest a-seq)))))

(sum [1 2 3 4 5])
; => 15
```

Devido ao fato de em linguagens funcionais a recursão ser utilizada ao invés de estruturas de loop, chamadas de recursão de cauda são otimizadas para que não consumam epsaço da pilha de recursão o que faz com que as chamadas recursivas consumam tempo constante. Entretanto devido ao fato de Clojure executar sobre a JVM, a otimização das recursões caudais não são otimizadas, então no caso de iterações sobre uma coleção suficientemente grande ocorre o risco de estourar a pilha de recursão. Para fazer essa otimização é necessário usar o comando `recur`. Utilizando `recur, sum` seria definido da seguinte forma.

```
(defn sum [a-seq]
        (let [sum-helper (fn [acc a-seq]
                           (if (? empty a-seq)
                            acc
                            (recur (+ acc (first a-seq))
                            (rest a-seq))))]
    (sum-helper 0 a-seq)))
```

## 4. Paralelismo

Em Clojure as identidades são denominadas tipos de referência e cada um possui um comportamento diferente em relação à concorrência.

Primeiramente, definiremos os tipos de operações concorrentes que são possíveis em Clojure.

- Síncrono: A *thread* que executou a operação aguarda obtenção do resultado requisitado ou algum retorno, não podendo realizar nenhuma operação durante esse tempo.
- Assíncrono: As operações realizadas não bloqueiam a *thread* que a executou.
- Coordenado: Uma operação que depende de outras operações para produzir o resultado esperado.
- Não coordenado: As operações são independentes entre si.

Conforme dito na seção anterior, existe três tipos de referência em Clojure: *refs, agents e atoms.*

### *4.1. Atoms*

*Atoms* são referências cujo resultado pode ser imediatamente visualizado pelas demais *threads* e seu valor é atualizado de maneira atômica. Suas atualizações são feitas de maneira não coordenada e síncrona. Devido à atualização atômica, evitam a utilização de locks explícitos no programa.

Para criar um *atom* utiliza-se a função `atom` e para modificarmos o valor referenciado usamos a função `swap!`.

No exemplo abaixo mostramos a criação de um *atom* e uma atualização em seu valor.

```clojure
(def currently-connected (atom []))

@currently-connected
;; => []
(deref currently-connected)
;; => []
currently-connected
;; => #<Atom@614b6b5d: []>

(swap! currently-connected conj "chatty-joe")
;; => ["chatty-joe"]
currently-connected
;; => #<Atom@614b6b5d: ["chatty-joe"]>
@currently-connected
;; => ["chatty-joe"]
```

O valor original de `currently-connected` pode ser obtido usando a função `reset!`.

### *4.2. Agents*

*Agents* são referências que são atualizadas assíncronamente, isto é, suas atualizações não acontecem imediatamente. São utilizados em casos onde não é exigida uma consistência estrita entre as *threads*.

Para a criação de um *agent* usa-se a função `agent` e são atualizados usando a função `send` ou `send-off`, a diferença de uma função para outra é que a *thread pool* da segunda possui tamanho variável.

No exemplo abaixo, é mostrado a criação e atualização de um *agent*:

```clojure
(def errors-counter (agent 0))
;; => #'user/errors-counter
errors-counter
;; => #<Agent@6a6287b2: 0>
@errors-counter
;; => 0
(deref errors-counter)
;; => 0

@errors-counter
;; => 0
(send errors-counter inc)
;; => #<Agent@6a6287b2: 0>
```

```
@errors-counter
;; => 1

;; 10 is an additional parameter. The + function will
 ↪ be invoked as (+ @errors-counter 10).
(send errors-counter + 10)
;; => #<Agent@6a6287b2: 1>
@errors-counter
;; => 11
```

*4.3. Refs*

São o único tipo de referência que opera de maneira coordenada e são usadas para garantir que múltiplas identidades possam ser modificadas concorrentemente dentro de uma transação. Seu funcionamento numa transação funciona sob três princípios:

1. Ou todas *refs* terminam ou nenhuma termina.
2. Não pode haver condições de corrida entre as *refs*.
3. Não há a possibilidade de impasse entre elas.

O exemplo abaixo mostra a criação de duas *refs* e sua atualização dentro de uma transação.

```
(def account-a (ref 1000))
;; => #'user/account-a
(def account-b (ref 1000))
;; => #'user/account-b

(dosync
  ;; will be executed as (+ @account-a 100)
  (alter account-a + 100)
  ;; will be executed as (- @account-b 100)
  (alter account-b - 100))
;; => 900
@account-a
;; => 1100
@account-b
;; => 900
```

Além desses tipos de referência, Clojure possui estruturas de dados e rotinas que são utilizadas para computação paralela.

### *4.4. Delays*

*Delays* são estruturas de dados que tem seu valor computado a primeira vez que é dereferenciada e esse valor ficará em cache para futuras dereferenciações. É possível checar se uma *delay* já possui algum valor usando a função `realized?`.

Na seção de Exemplos, será utilizado *delays* em conjunto com outras rotinas para evitar problemas de concorrência. Portanto iremos nos ater apenas à definição da estrutura de dados.

### *4.5. Future*

*Future* é uma rotina usada para definir uma tarefa a ser realizada em paralelo sem exigir que o resultado seja imediatamente devolvido. Para obter o resultado de uma *future* basta dereferenciá-la.

É possível usar a função `realized?` para verificar se a execução em paralelo já terminou sua execução. Seu uso será demonstrado na seção Exemplos.

### *4.6. Promises*

Assim como *futures*, *promises* são utilizadas para obter valores de maneira assíncrona, porém diferentemente de *future*, não computam alguma tarefa em paralelo e sim recebem um valor quando necessário através do comando `deliver`.

Seu uso será demonstrado na seção Exemplos.

### *4.7. Locks Comuns*

Além das operações definidas aqui, também é possível criar locks de maneira explícita e é possível utilizar outras ferramentas de controle como semáforos.

Por ser executado sobre a JVM, Clojure possui monitores implementados.

O exemplo abaixo mostra o uso de locks para a execução de código em Clojure em conjunto com código Java.

```
(let [l (java.util.ArrayList.)]
  (locking l
    (.add l 10))
  l)
;; => #<ArrayList [10]>
```

## 5. Exemplos

Os exemplos a seguir foram retirados do livro *Clojure for the Brave and True*.

*5.1. Evitando o problema de exclusão mútua*

No primeiro exemplo, é utilizado *delays* como uma forma de evitar problemas de concorrência causados pela ausência da exclusão mútua no acesso a determinada região do programa.

Na situação do problema tem-se uma lista de fotos que se deseja fazer *upload* em um sistema e o dono das fotos deseja ser avisado por e-mail assim que a primeira estiver no sistema, entretanto ele não deseja receber e-mail para as demais fotos. Para fazer o *upload* de forma mais rápida, cada *thread* será responsável por uma das fotos e, a primeira que terminar irá notificar o usuário por email.

```
1  (def gimli-headshots ["serious.jpg" "fun.jpg"
   ↪ "playful.jpg"])
2
3  (defn email-user
4          [email-address]
5          (println "Sending headshot notification to"
           ↪ email-address))
6
7  (defn upload-document
8          "Needs to be implemented"
9          [headshot]
10     true)
11
12 (let [notify (delay (email-user
   ↪ "and-my-axe@gmail.com"))]
13         (doseq [headshot gimli-headshots]
14         (future (upload-document headshot)
15                                 (force notify))))
```

Da linha 3 a 10, são definidas nossas funções que serão responsáveis por notificar o usuário e fazer o *upload* do documento.

Dentro do escopo do `let` atribuímos a `notify` o valor do nosso *delay*. O comando `doseq` nos permite repetir instruções definidas em seu corpo para cada elemento duma coleção passado como parâmetro. Em seguida fazemos para que cada arquivo da lista seja enviado por uma *thread* através do uso de *futures*.

Ao criarmos um *delay*, garantimos que a função chamada dentro do corpo deste só será executada a primeira vez que `notify` for dereferenciado. Para as outras vezes que for dereferenciado, não irá imprimir que está enviando notificação para o email. Note que se definissemos que `email-user` retorna

algum valor, este seria obtido nas outras dereferenciações, porém nenhuma outra computação seria realizada.

*5.2. Evitando problemas de condição de corrida*

No segundo exemplo utilizaremos *promises* como uma forma de evitar problemas de condição de corrida, isto é, a saída de uma sequência de operações é afetada pela ordem conforme as leituras e escritas a um determinado valor são realizados.

Conforme dito na seção sobre paralelismo em Clojure, *promises* são objetos que só podem ser escritos uma vez. Com isso é possível evitar inconsistências e garantir que o resultado final será o esperado.

Na situação do problema desejamos fazer com que o nosso fiel papagaio tenha a voz de James Earl Jones. Para isso, descobrimos que é necessário utilizar uma manteiga com certo grau de suavidade. Além disso temos um preço máximo ao qual estamos dispostos a pagar pelo produto.

Para verificarmos nas lojas online, usamos uma API para fazer essa verificação.

```
(def yak-butter-international
  {:store "Yak Butter International"
    :price 90
    :smoothness 90})
(def butter-than-nothing
  {:store "Butter Than Nothing"
   :price 150
   :smoothness 83})
;; This is the butter that meets our requirements
(def baby-got-yak
  {:store "Baby Got Yak"
   :price 94
   :smoothness 99})

(defn mock-api-call
  [result]
  (Thread/sleep 1000)
  result)

(defn satisfactory?
  "If the butter meets our criteria, return the
   ↪ butter, else return false"
  [butter]
```

```
23    (and (<= (:price butter) 100)
24         (>= (:smoothness butter) 97)
25         butter))
```

Nesse trecho de código definimos os tipos de manteiga disponíveis, nossa chamada de API que no caso simplesmente fará a *thread* esperar durante um segundo e uma função que diz se uma dada manteiga satisfaz nossos requisitos.

```
1  (time (some (comp satisfactory? mock-api-call)
2              [yak-butter-international
               ↪ butter-than-nothing baby-got-yak]))
3  ; => "Elapsed time: 3002.132 msecs"
4  ; => {:store "Baby Got Yak", :smoothness 99, :price
   ↪ 94}
5
6  (time
7   (let [butter-promise (promise)]
8     (doseq [butter [yak-butter-international
       ↪ butter-than-nothing baby-got-yak]]
9       (future (if-let [satisfactory-butter
         ↪ (satisfactory? (mock-api-call butter))]
10                  (deliver butter-promise
                     ↪ satisfactory-butter))))
11    (println "And the winner is:" @butter-promise)))
12 ; => "Elapsed time: 1002.652 msecs"
13 ; => And the winner is: {:store Baby Got Yak,
   ↪ :smoothness 99, :price 94}
```

No trecho acima, realizamos a procura pela manteiga desejada da lista. Na primeira implementação, fazemos isso sequencialmente portanto não existe nenhum problema de concorrência, em contrapartida, demora três segundos para verificar todas.

Na segunda implementação, fazemos com que cada *thread* seja responsável por verificar se uma determinada manteiga satisfaz nossos requisitos. Caso uma satisfaça, seu valor é guardado em `butter-promise`. Importante notar que caso uma *thread* chegue na linha onde o valor é impresso e a *promise* ainda não possuir um valor, ela ficará bloqueada até que um valor seja atribuído a ela.

Num cenário onde nenhuma das manteigas disponíveis atende ao nosso requisito, todas as *threads* ficariam bloqueadas. Para evitar esse problema,

pode se dereferenciar uma *promise* usando `deref` onde nele não só é possível passar um tempo limite ao qual irá esperar por uma resposta, mas também uma ação que deve ser feita caso esse tempo seja ultrapassado.

No trecho abaixo, imprimimos `timed out` caso 100 milisegundos se passem sem resposta.

```
(let [p (promise)]
  (deref p 100 "timed out"))
```

## 6. Comparativo

Para avaliar o desempenho de algumas linguagens, foi realizado um teste cuja ideia foi considerar, além de comparativos de benchmark, outros fatores, como a facilidade de desenvolvimento.

Um *websocket* com duas mensagens básicas codificadas em JSON (echo e broadcast) foi desenvolvido em Clojure, C++, Elixir, Go, NodeJS e Ruby. Um echo é retornado ao cliente que enviou a solicitação e o broadcast é enviado a todos os clientes conectados.

A implementação em Clojure utilizou o HTTP Kit (um *webserver*) e a conexão com o cliente foi armazenada em atoms, o que garante concorrência para que a linguagem lide com as requisições em paralelo.

```
1 (defonce channels (atom #{}))
2
3 (defn connect! [channel]
4   (log/info "channel open")
5   (swap! channels conj channel))
6
7 (defn disconnect! [channel status]
8   (log/info "channel closed:" status)
9   (swap! channels disj channel))
```

O código implementado para enviar um *broadcast* é muito simples. Nele, uma mensagem é basicamente enviada para todos os canais e o resultado é enviado ao remetente.

```
1 (defn broadcast [ch payload]
2   (doseq [channel @channels]
3     (send! channel (json/encode {:type "broadcast"
      ↪ :payload payload})))
4   (send! ch (json/encode {:type "broadcastResult"
    ↪ :payload payload})))
```

Os outros websockets foram implementados de maneira similiar à que utilizou Clojure, levando sempre em consideração a particularidades da linguagem.

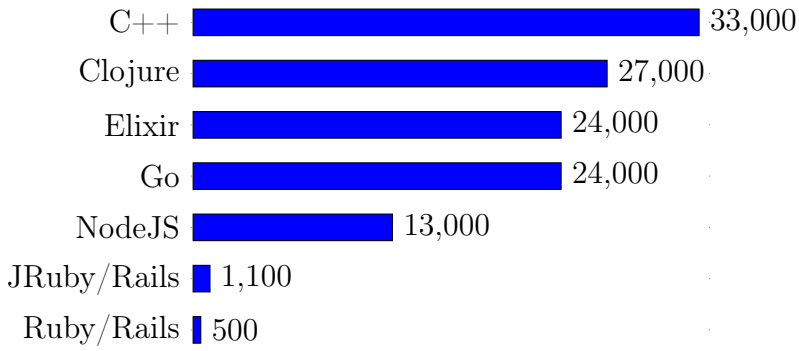
Como exemplo, vemos abaixo a implementação da mesma função de *broadcast* apresentada acima desenvolvida em C++.

```
1  void server::broadcast(websocketpp::connection_hdl
   ↪ src_hdl, const Json::Value &src_msg) {
2    Json::Value dst_msg;
3    dst_msg["type"] = "broadcast";
4    dst_msg["payload"] = src_msg["payload"];
5    auto dst_msg_str = json_to_string(dst_msg);
6
7    boost::shared_lock_guard<boost::shared_mutex>
     ↪ lock(conns_mutex);
8
9    for (auto hdl : conns) {
10     wspp_server.send(hdl, dst_msg_str,
       ↪ websocketpp::frame::opcode::text);
11   }
12
13   Json::Value result_msg;
14   result_msg["type"] = "broadcastResult";
15   result_msg["payload"] = src_msg["payload"];
16   result_msg["listenCount"] = int(conns.size());
17   wspp_server.send(src_hdl,
     ↪ json_to_string(result_msg),
     ↪ websocketpp::frame::opcode::text);
18 }
```

Para os testes de benchmark, foram realizadas sobrecargas do servidor para ver quantas conexões não-ociosas conseguiam lidar mantendo certa performance.

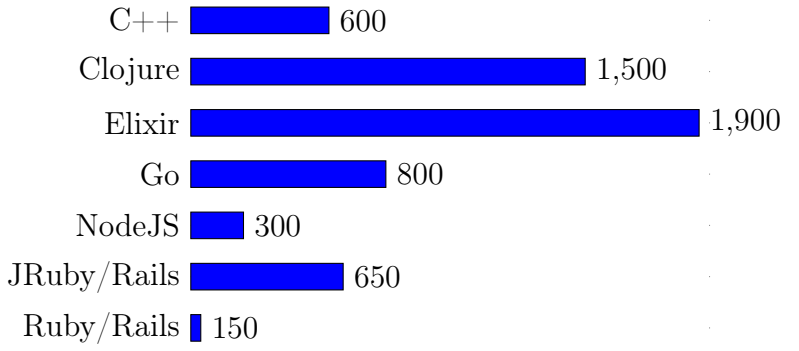
Os resultados foram obtidos ao executar o servidor em uma máquina e a ferramenta de *benchmark* em outra de mesma configuração (4GHz, i7, 4790Ks com 16GB de memória RAM com o Ubuntu 16.04 conectados via Ethernet). Os testes foram executados várias vezes e os melhores resultados foram armazenados.

Número de clientes não-ociosos conectados




O uso da memória também foi um fator considerado para a comparação dos ambientes.

Uso de memória (MB)




O C++ está no topo do gráfico de desempenho por uma margem substancial. É o mais eficiente no uso de memória para o número de conexões. No entanto, o servidor C++ também possui a implementação mais detalhada, verbosa e complexa. A linguagem utiliza diversos paradigmas que incluem acessos de baixo nível à ponteiros, heranças e excessões. Além disso, a implementação em C++ requer *makefiles* e alto tempo e compilação.

Apesar de possuir apenas 82% do desempenho do C++, o Clojure, que possui a segunda melhor performance, passa a ser uma ótima alternativa por possuir código conciso e de fácil desenvolvimento. Além disso, o projeto fica contido exclusivamente em um único arquivo `.jar`, o que torna sua implementação mais simples e que ainda permite a possibilidade de utilizar as bibliotecas da JVM.

Dessa forma, o Clojure mostra que linguagens de alto nível podem ser

muito rápidas e com ótimo desempenho, se apresentando como uma candidata competitiva mesmo ao lidar com linguagens já reconhecidas e estabelecidas no mercado.

## 7. Considerações finais

Clojure apesar de nova se motra uma linguagem muito bem estruturada e definida. Nesta seção vamos citar alguns dos seus pontos positivos e negativos que pudemos notar durante o estudo da linguagem.

Começando pelos pontos positivos podemos dizer que ser funcional por si só já agrega muitos benefícios ao Clojure. Apenas por este fato a linguagem ganha uma sintaxe enxuta, apesar de "pouco"conhecida, e imutabilidade. Isto faz com que a software escritos na linguagem tenham poucas linhas de código, linhas que a princípio aparentam ser mais complicadas, porém basta se iniciar na linguagem e seus programas se tornam muito fáceis de ler. Por ter imutabilidade, a linguagem consegue fornecer bom desempenho por não precisar copiar todas estruturas e ao mesmo tempo boa consistência de dados por fazer cópias e referências seletivas.

A línguagem é "compilada", isto é, é gerado *bytecode* da JVM. Isto fornece à linguagem dois grandes benefícios. O primeiro é que o código obtém um performance melhor do que um código totalmente interpretado, além disso a o código pode se aproveitar de otimizações intrínsecas da JVM como *just in time compiling* e outras *features*. Outro ponto muito positivo de utilizar executar nesta plataforma é que é possível executar código clojure em qualquer lugar que tenha suporte à JVM. Com certeza este foi um fato que facilitou em grande parte a disseminação da linguagem.

Clojure também possui várias estruturas de dados e rotinas *built-in* para auxiliar no desenvolvimento de processamento concorrente. Como quase todas os processadores hoje vêm de fábrica com mais de um núcleo essa *feature* se torna muito relevante, já que elimina grande parte do trabalho do desenvolvedor de criar e gerenciar *locks*, *threads* e contextos. Além disso, Clojure já possui também suporte a algumas funções essenciais para *MapReduce* que facilitam o desenvolvimento de tal arquitetura e, quando utilizadas em conjunto com as *features* de programção concorrente da linguagem, a tornam eficiente para executar tal processo.

Um ponto negativo da linguagem que não pude experienciar mas pude notar um alto índice de reclamações foi a respeito do manageio de erros que a linguagem possui, que aparentemente em casos de execução concorrente é praticamente nulo ou não fornece informações úteis (é sempre muito similar). Em casos de execução não concorrente é extramamente *verbose*, ou seja, apresenta um *stack trace* muito grande o que dificulta o *debugging*. Além

disso parece não ser trivial implementar algo análogo a um *try{} catch{}* o que piora ainda mais a situação.

Por ser mais sucinta, a linguagem transfere uma responsabilidade maior ao desenvolvedor, tendo este que se atentar mais durante o desenvolvimento e normalmente é exigida uma abordagem mais complexa aos mesmos tipos de problemas, porém a complexidade acaba se transferindo para a lógica de programação ao invés da sintaxe. Outro fato observado também nesse quesito de transfência de responsabilidade é que a linguagem ,muito ao contrário do JAVA, é fracamente tipada, portanto também é necessário atenção a este ponto durante o desenvolvimento.

Clojure em geral apresenta mais pontos positivos que negativos, ou melhor, os pontos positivos são mais relevantes e estão fazendo com que esta linguagem seja adotada por diversas empresas grandes. Por ser muito confiável devido à sua arquitetura, executar na JVM e ter potencial para concorrência esta linguagem com certeza vai ocupar um espaço maior ainda no mercado.