Primeiro Trabalho Prático de Programação Concorrente

Pedro Eduardo Nogueira da Mota, up201805248

Resumo

Para este trabalho, foi-nos proposto a implementação de um servidor, correspondente a uma base de dados para guardar requisições de livros de uma biblioteca, e de uns clientes que poderão enviar um determinado conjunto de mensagens ao servidor. Toda a implementação deveria ser feita através da linguagem Erlang. Começamos, então, por fazer uma leve introdução à linguagem, com ênfase na sua parte concorrente. Por fim, voltamos a atenção à tarefa, a qual designamos por “Livraria”, apresentando a arquitetura da implementação que foi feita, explicando algumas das decisões tidas em conta, terminando com a apresentação de alguns dos resultados obtidos.

Prefácio

Ao longo do trabalho, utilizou-se como material de suporte à linguagem *Erlang* a documentação oficial da linguagem [1], disponível *online*, e o livro “Programming Erlang: Software for a Concurrent World“, escrito por Joe Armstrong, o próprio criador da linguagem [2]. O livro está bem estruturado e é rico em exemplos práticos, o que é uma mais-valia, para quem está a ter o primeiro contacto com a linguagem.

Para a criação do diagrama relacional da base de dados, que irá surgir posteriormente, foi utilizada a ferramenta *dbdia*, do professor Eduardo R. B. Marques, da Faculdade de Ciências da Universidade do Porto. [3]

Erlang

Motivação

Quando vamos aprender uma linguagem de programação nova, ou algo em geral, é importante identificar algumas das razões pelas quais o vamos fazer, desde a sua utilidade ou o que apresenta de revolucionário, face ao que já conhecemos.

Nas próximas secções, respondemos a estas perguntas, indiretamente, indicando em que contexto o Erlang surge, seguido de alguns conceitos básicos da linguagem, tentando convencer o leitor de que o Erlang é uma linguagem de programação interessante de se conhecer, principalmente, se o leitor está mais familiarizado com linguagens sequenciais, como C e Java. Pode ser, ainda, além do primeiro encontro com a programação concorrente, o seu primeiro encontro com a programação funcional.

Concorrência

O universo é concorrente. Como tal, seria interessante se, quando o modelássemos, pudéssemos manter essa mesma concorrência.

Nesse sentido, surgem as linguagens de programação concorrente.

Uma linguagem de programação concorrente é uma linguagem de programação com construtores explícitos que permitem construir programas concorrentes.

Programas concorrentes são programas compostos por um conjunto de processos que se comunicam entre eles.
Neste contexto, surge o *Erlang*: uma linguagem de programação funcional, com tipagem forte e dinâmica, sendo, também, concorrente.
A comunicação entre processos é feita de forma assíncrona, através da troca de mensagens.
Além de possuir construtores de suporte à concorrência, esta última é suportada pela máquina virtual que acompanha o *Erlang*, o que se distingue de outras linguagens de programação, nomeadamente as sequenciais, onde o suporte à concorrência é fornecido pelo sistema operativo ou por bibliotecas externas, o que pode resultar em o mesmo programa ter comportamentos diferentes, em diferentes sistemas operativos.
A concorrência traz, também, várias vantagens, nomeadamente, melhorias de performance, escalabilidade e tolerância a erros.

Concorrência e Paralelismo
Gostaria, também, de aproveitar para salientar a diferença entre concorrência e paralelismo. A primeira diz respeito a *software* e a segunda a *hardware*.
Concorrência é quando duas ou mais tarefas começam, executam ou terminam em tempos sobrepostos. Um exemplo é o de *multitasking* numa máquina *singlecore*.
Já o paralelismo é quando estas tarefas executam, literalmente, ao mesmo tempo, o que pode acontecer numa máquina *multicore* ou em rede, por exemplo.
Um programa concorrente pode ser executado paralelamente, ou não.

Programar em *Erlang*
Em *Erlang*, processos e concorrência são as ferramentas que utilizamos para implementar uma determinada solução a um problema.
Um primeiro passo, então, na construção de um programa em *Erlang*, consiste em identificar um conjunto de processos que irão resolver o nosso problema - semelhante a identificar os objetos necessários, numa solução do paradigma orientado a objetos. Esta tarefa não é, de todo, fácil, requer alguma experiência.

Posto isto, iremos, agora, introduzir alguns conceitos básicos da linguagem, culminando depois na gestão de processos e como estes comunicam entre si:

- A *shell*
  A *shell* do *Erlang* permite-nos escrever expressões, que são depois avaliadas e é imprimido o seu resultado.
  Cada expressão deve ser finalizada por um ponto final.
  Após a instalação do *Erlang*, a *shell* pode ser inicializada, executando o comando `erl`, no prompt.
  Para executar uma função específica, de um dado módulo, seguimos o seguinte formato: `Module:Function(Arg1,...,ArgN)`.

- Estrutura de um programa em *Erlang*
  Um programa, em *Erlang*, vai consistir em um conjunto de funções, definidas em diferentes módulos.
  Um ficheiro em *Erlang*, de extensão ".erl", constituí um módulo, com o mesmo nome do ficheiro. A nomeação do módulo é feita no início do ficheiro, com o uso do atributo *module*: `-module(file_name).`
  Segue-se, tipicamente, um atributo de exportação e outros de importação. Na exportação, define-se que funções ficam visíveis para o exterior deste módulo. Na importação, define-se as funções que queremos importar, indicando também o módulo a que pertencem, podendo depois usá-las como se fossem locais. A título de exemplo:
  ```
  -import(list, [foreach/2]).
  -export([foo/3, bar/2]).
  ```
  Finalmente, sucede-se código escrito em *Erlang*.
  Para utilizar um módulo, precisamos primeiro de o compilar. Para tal, executamos na shell: `c(module)`.

- Variáveis e o operador =
  O operador = é um operador de *pattern matching*. Permite-nos atribuir um valor a uma variável, uma única vez. Como já foi referido, o *Erlang* é uma linguagem funcional, então quando uma variável toma um valor, esse valor nunca mais pode mudar, tal como em *Haskell*, por exemplo.
  As variáveis começam, sempre, por letra maiúscula.
  É de salientar que o facto da linguagem não possuir estados mutáveis e, consequentemente, memória partilhada, facilita a paralelização dos programas, posteriormente.

- Átomos
  Em *Erlang*, também existem átomos, como, por exemplo: `monday` e `masculine`. Átomos não são variáveis, mas sim constantes simbólicas. Distinguem-se das variáveis, começando por uma letra minúscula, tal como em *Prolog*.

- Operador aritméticos, lógicos e relacionais
  A linguagem *Erlang* define um vasto conjunto de operadores aritméticos, lógicos e relacionais. Introduziremos, apenas, os mais essenciais.
  Relativamente aos operadores aritméticos, temos: +, -, * e /, para a operação normal de soma, subtração, multiplicação e divisão, respetivamente.
  Já no que toca aos operadores lógicos, temos o *and*, *or* e o *not*, que representam o *e*, *ou* e a *negação* lógica, respetivamente.
  Por fim, para operadores relacionais temos: ==, /=, <, <=, > e >=, tal como em *Haskell*.

- Tuplos
  Os tuplos permitem guardar um número finito de itens, numa só entidade. Um tuplo é criado envolvendo os valores que queremos representar, separados por vírgulas, em chavetas, como, por exemplo:

  ```
  {person, 9999999, "Dolores", "WestWorld", 12345679}
  ```

  Os tuplos são criados, automaticamente, como dito anteriormente. Toda a alocação de memória, bem como a sua dealocação, é feita pelo *garbage collector* do *Erlang*.
  Uma vez que os campos não têm nome, para obter os valores dos mesmos, pode-se fazê-lo através de *pattern matching* ou da função *elem*/2.

- Records
  Os *records* são açúcar sintático para tuplos, onde os campos, agora, possuem nomes, o que pode ser bastante útil, caso tenhamos uma estrutura de dados de grande dimensão, pois torna-se complicado relembrar a posição de determinado campo, sendo mais prático o seu acesso através de um nome representativo.
  Para criar um *record*, usa-se o atributo *-record*, da seguinte forma:

  ```
  -record(book, {id, title, authors}).
  ```

  Para obter uma instância do *record* definido:

  ```
  B1 = #book{id=1, title="The little Book of Semaphores",
  authors={"Allen B. Downey"}}.
  ```

  Para acessar a um determinado elemento do *record*: `B1#book.title.`

- Listas
  As listas são semelhantes aos tuplos, mas permitem guardar um número arbitrário de itens e são definidas através de parênteses retos, como, por exemplo:

  ```
  [1, "The little Book of Semaphores", {"Allen B. Downey"}]
  ```

  Podemos, também, definir listas da seguinte forma: sendo XS uma lista, então [X|XS] é, também, uma lista, no caso, a lista XS, com mais um elemento, X, à cabeça. Esta forma de definir listas é bastante útil na definição de funções que operam sobre listas.
  Outra forma útil de definir listas, é através de listas em compreensão, semelhante a como é feito em *Haskell*, como, por exemplo:

  ```
  [X*X || X <- [1,2,3,4], X rem 2 == 0].
  ```

- Funções

  Uma função, em *Erlang*, é definida por várias cláusulas. Cada cláusula é separada por um ponto e vírgula, ao contrário da última, que finaliza com um ponto final. Uma cláusula é constituída por uma cabeça e um corpo, separados por uma seta. A cabeça consiste do nome da função, seguido de zero ou mais padrões. O corpo consiste em sequências de expressões, que são avaliadas, caso os padrões na cabeça combinem com os argumentos de chamada. As cláusulas são testadas conforme a ordem em que aparecem na definição da função.

  Exemplo da implementação do *quicksort*, em *Erlang*:

```
qsort([])-> [];
qsort([P|L])-> qsort([X|| X<-L, X < P] ++ [P] ++ qsort([X|| X<-L, X >= P].
```

  Em *Erlang*, também existem funções anónimas, que são definidas entre as *keywords fun* e *end,* como, por exemplo: `F = fun(X) -> X * X end.`

  As funções podem ser de ordem superior.

- Construtores de concorrência

  Até agora, apenas vimos construtores sequenciais. Iremos apresentar, agora, como podemos escrever um programa concorrente. Relembramos que, em concorrência, a unidade básica é o processo e que os processos comunicam entre si, através da troca de mensagens.

  Precisamos, então, essencialmente, de saber como podemos criar um processo e como enviar e receber mensagens.

  Para criar um processo, usamos a função *spawn*.

```
spawn(Module, Name, Args) -> pid()
  Module = Name = atom()
  Args = [Arg1,...,ArgN]
  ArgI = term()
```

  Esta função irá criar um novo processo, retornando o seu PID.

  O novo processo começa a executar a função Module:Name(Arg1, …, ArgN).

  Para enviar uma mensagem, usamos o operador !, da seguinte forma:

```
Expr1 ! Expr2
```

  O valor de *Expr2* é enviado, como mensagem, ao processo especificado por *Expr1*, que pode ter como valor um PID, por exemplo.

  O valor de *Expr2* é, também, o valor de retorno da expressão.

Para receber uma mensagem, usamos a expressão *receive*:

```
receive
   Pattern1 ->
       Body1;
   ...;
   PatternN ->
       BodyN
end
```

Os padrões *Pattern* são, sequencialmente, combinados com a chegada de mensagens. Se a combinação for bem sucedida, o *body* correspondente é avaliado.
Posto isto, a mensagem é descartada, podendo vir uma próxima mensagem, caso exista. Se não existir, a execução é suspensa indefinidamente, até uma mensagem chegar.
O valor de retorno da expressão *receive* é o mesmo que o da expressão *body* executada.

- *Mnesia*, a base de dados do *Erlang*
  Na distribuição padrão do *Erlang*, existe uma base de dados, designada *Mnesia*, extremamente rápida e configurável.
  A base de dados *Mnesia* permite guardar qualquer estrutura de dados válida em *Erlang*.
  A interação com a base de dados é feita através de código escrito em *Erlang*.

Poderíamos acrescentar mais informação sobre a linguagem, bem como aprofundar os conceitos que foram apresentados, principalmente, a base de dados *Mnesia*, contudo, de forma a manter o tamanho do relatório dentro do aceitável e uma vez que já abordamos o suficiente para realizar a tarefa proposta, vamos terminar aqui a introdução ao *Erlang*, recomendando, vivamente, a leitura de [1] e [2].

Finalmente, iremos voltar a nossa atenção à tarefa proposta.
Iremos começar por detalhar o que foi pedido. De seguida, iremos apresentar a arquitetura da implementação que foi feita, bem como explicar algumas das decisões tomadas, finalizando com alguns dos resultados obtidos.
Recomenda-se, portanto, que seja feita a leitura das próximas secções, seguindo os dois ficheiros de código (*server.erl* e *client.erl*) que acompanham este relatório.

Livraria

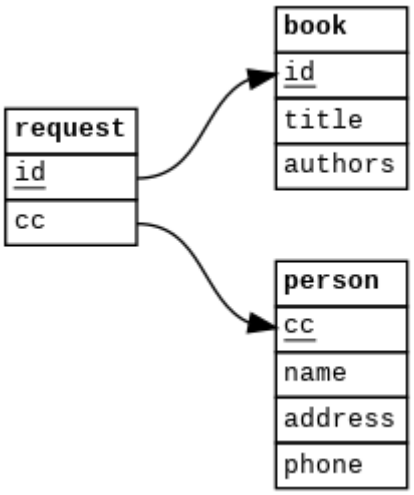
Objetivo

O objetivo passa por implementar, em *Erlang*, um servidor correspondente a uma estrutura de dados/base de dados para guardar requisições de livros numa biblioteca.
As requisições seriam pares de identificadores pessoa-livro, correspondendo à requisição de um livro, por uma determinada pessoa.
Relativamente aos livros, devíamos manter informação do seu código, o seu título e os seus autores, já no que toca às pessoas, devíamos manter o seu número de cartão de cidadão, nome, morada e telefone.
Posto isto, devíamos implementar uns clientes que pudessem enviar umas mensagens de *lookup* (através do número de cartão de cidadão de uma pessoa, obter a lista de livros requisitados por essa pessoa; através de um título de um livro, obter a lista de pessoas que requisitaram esse livro; através de um código de um livro, determinar se o livro está requisitado ou não; através de um título de um livro, obter a lista de códigos de livros com esse título; através do número de cartão de cidadão de uma pessoa, obter o número de livros requisitados por essa pessoa) ou *update* (através dos dados de uma pessoa e um livro, fazer a requisição ou devolução do mesmo).

Arquitetura da implementação

Definida a tarefa, decidimos dividir o programa em dois ficheiros, correspondentes ao servidor (*server.erl)* e ao cliente (*client.erl*).

Para guardar toda a informação que deve ser mantida, optamos por escolher a base de dados *Mnesia*, que já foi atrás apresentada.
O servidor vai ser, então, responsável por inicializar a base de dados, ficando depois a atender eventuais pedidos de clientes, garantido a consistência da base de dados, ao longo do tempo.
Segue-se um diagrama com o esquema relacional da base de dados implementada:



Iremos manter três tabelas, uma relativa aos livros, outra às pessoas e, por fim, uma relativa às requisições.
A correspondência entre os atributos e o que guardam são explicáveis por si só.

No diagrama apresentado, as chaves primárias aparecem a sublinhado.
Os atributos *id* e *cc*, da tabela *request,* são chaves externas para a chave primária das tabelas *book* e *person*, respetivamente.

É importante salientar que na base de dados *Mnesia*, o conceito de relação de integridade não existe, pelo que cabe ao programador verificar todas as transações feitas, de forma a garantir a consistência da base de dados, que foi, precisamente, o que fizemos.

Na tabela *request*, a chave primária é apenas o *id* do livro, uma vez que um determinado livro só pode ser requisitado uma única vez, pelo que torna a sua chave primária suficiente para identificar a entrada da requisição.

O módulo *server* exporta, apenas, uma função: *start/0*, que inicializa a base de dados e deixa o servidor à escuta de pedidos dos clientes.

De forma a garantir a consistência da base de dados, para cada mensagem que chegue de um cliente, todos os dados são validados, ie., é verificado se os dados que constam na mensagem do cliente são, de facto, válidos e se o pedido pode ser realizado. Caso sejam, o servidor responde ao cliente, com o conteúdo pedido, caso contrário, a mensagem de resposta do servidor é a sinalização do erro, em particular, que causou a falha no pedido do cliente.

Já relativamente ao módulo *client*, temos, simplesmente, um conjunto de funções que lhe vai permitir enviar as tais mensagens de *lookup* e *update*, imprimindo depois, num formato amigável, a resposta do servidor.

O módulo *client* exporta sete funções: *person_requests/1, book_requests/1, book_is_requested/1, book_ids/1, person_num_requests/1, request_book/2* e *return_book/2*.

O seguinte quadro apresenta, de forma resumida, o propósito de cada função:

Tipo	Função	Descrição
Lookup	*person_requests/1*	Dado o CC de uma pessoa, retorna os IDs dos livros que essa pessoa requisitou.
	book_requests/1	Dado um título de um livro, retorna os CCs das pessoas que requisitaram o livro.
	book_is_requested/1	Dado o ID de um livro, verifica se o livro está ou não requisitado.
	book_ids/1	Dado um título de um livro, retorna os IDs dos livros com esse título.
	person_num_requests/1	Dado o CC de uma pessoa, retorna o número de livros que essa pessoa requisitou.
Update	*request_book/2*	Dado um título de um livro e o CC de uma pessoa, faz uma requisição desse livro, em nome dessa pessoa.
	return_book/2	Dado um título de um livro e o CC da pessoa que o tem, faz a sua devolução.

Na próxima secção, mostraremos alguns exemplos de como usar as funções.

Resultados

Iremos, então, dedicar esta secção para mostrar alguns dos exemplos de uso.
Vamos utilizar os dados dos livros e pessoas que se encontram na base de dados, logo após o servidor iniciar.
Após inicializar a *shell* do *Erlang* e compilar os dois módulos, chamamos a seguinte função para inicializar o servidor: `server:start().`
Posto isto, podemos, então, enviar mensagens do lado do cliente.
Uma vez que ainda não existem quaisquer requisições, pois o servidor acabou de ser inicializado, iremos começar por ilustrar as mensagens de *update*.

Executemos, então, na *shell*:

```
client:request_book(1, 1000).
```

Irá ser imprimido a mensagem “The book was requested with sucess.”, a informar que o livro foi requisitado com sucesso.

Façamos outras requisições válidas:

```
client:request_book(10, 1000).
client:request_book(3, 2000).
client:request_book(4, 2000).
client:request_book(6, 2000).
client:request_book(7, 6000).
client:request_book(13, 3000).
```

Tentemos agora:
`client:request_book(1, 1003).` ou `client:request_book(20, 1000).`

No primeiro caso, será imprimido a mensagem "There's no person in the system with that CC number.", a indicar que não existe uma pessoa registada no servidor com esse número de cartão de cidadão. Já, no segundo caso, será imprimido "There's no book in this library with that ID.", ou seja, semelhante ao caso anterior, mas relativo ao ID do livro.

Tentemos, agora, algumas devoluções: `client:return_book(3, 1000).`
Surge a mensagem "You don't possess the book to return it.", a indicar que a pessoa com o número de cartão de cidadão 1000, não possui o livro com ID 3, pelo que não o pode devolver!
Já `client:return_book(13, 3000).`, imprime a mensagem "The book was returned with sucess.", indicando que o livro foi devolvido com sucesso.

Iremos deixar de ilustrar as sinalizações de erros, uma vez que, em todas as funções, os cenários são semelhantes: caso o pedido do cliente seja inválido, por alguma razão, o servidor deteta o erro e sinaliza-o, sem qualquer falha.

Passemos agora a *person_num_requests/1.*
Executando `client:person_num_requests(2000)`, irá imprimir a mensagem "The person with CC number 2000 has requested 3 books.", o que está correto, uma vez que essa pessoa requisitou os livros com ID 3, 4 e 6.
Já `client:person_num_requests(3000)`, imprime "The person with CC number 3000 has requested 0 books.", o que também se verifica, pois, apesar de a mesma ter requisitado o livro com ID 13, entretanto, já o devolveu!

Relativamente a *book_ids/1,* o comportamento também é o esperado.
Executando `client:book_ids("Deep Learning")`, obtemos "Available IDs of the book "Deep Learning": 13.", o que se verifica, pois, apenas existe um livro na base de dados com esse título, sendo, de facto, o livro com ID 13.
Já `client:book_ids("Convex Optimization")`, é imprimido "Available IDs of the book "Convex Optimization": 6, 11, 12.", o que também se verifica, pois, existem três livros na base de dados com esse título, que são os livros com ID 6, 11 e 12.

Seguindo para *book_is_requested/1.*
Executando `client:book_is_requested(13)`, é imprimido "The book isn't requested.", o que se verifica, uma vez que o livro foi devolvido pela pessoa com número de cartão de cidadão 3000 e não foi feita mais nenhuma requisição.
Já `client:book_is_requested(1)`, imprime "The book is requested.", o que também se verifica, uma vez que se encontra na posse da pessoa com cartão de cidadão 1000.

Testemos agora a função *book_requests/1.*
Executando `client:book_requests("Deep Learning")`, é imprimido "No one requested the book "Deep Learning".", o que se verifica, uma vez que apenas existe um livro na base de dados com tal título, que é o livro com ID 13 e o mesmo não se encontra requisitado, como vimos anteriormente.
Já se executarmos `client:book_requests("Programming Erlang: software for a concurrent world")`, é imprimido "CC numbers of the people who requested the book "Programming Erlang: software for a concurrent world": 1000, 2000.", o que, também, se verifica, visto existirem dois livros na base de dados com esse título, que são os livros com ID 1 e 3 e os mesmos encontram-se requisitados pelas pessoas de cartão de cidadão 1000 e 2000, respetivamente.

Por fim, a função *person_requests/1.*
Executando `client:person_requests(3000)`, é imprimido "The person with CC number 3000 has no requests made.", o que se verifica, uma vez que a pessoa com o cartão de cidadão número 3000, fez apenas uma requisição e já fez a devolução da mesma.
Já, executando `client:person_requests(2000)`, é imprimido "IDs of the books requested by the person with CC number 2000: 3, 4, 6.", o que, também, se verifica, pelas requisições feitas anteriormente.

Podíamos ilustrar mais casos de uso das funções, mas já deu para exemplificar como a interação servidor-cliente é feita, sendo que o leitor pode testar algum caso em que esteja interessado, uma vez que tem acesso a todo o código envolvido.

Podíamos, também, executar o sistema em múltiplas máquinas virtuais do Erlang. Suponhamos que tínhamos três máquinas virtuais do Erlang, designadas *s*, *c1* e *c2,* ou seja, correspondentes ao servidor e a dois clientes, respetivamente.
Executemos `server:start()`, no nó *s*, de forma a inicializar o servidor.
Agora, através do nó *c1* ou *c2*, para enviar uma mensagem ao servidor, que se encontra em outro nó, basta executar `rpc:call('s@HOSTNAME', client, request_book, [1, 1000])`, no caso, para requisitar o livro de ID 1, em nome da pessoa com o número de cartão de cidadão 1000, por exemplo.
Para outras mensagens, o processo é semelhante.

Referências

[1] Documentação oficial do Erlang - https://www.erlang.org/docs

[2] Armstrong, J. (2013). *Programming Erlang: Software for a Concurrent World*. 2ª edição, Susannah Davidson Pfalzer.

[3] R. B. Marques, E. (2000). *dbdia* - https://github.com/edrdo/dbdia.