Universiteit van Amsterdam

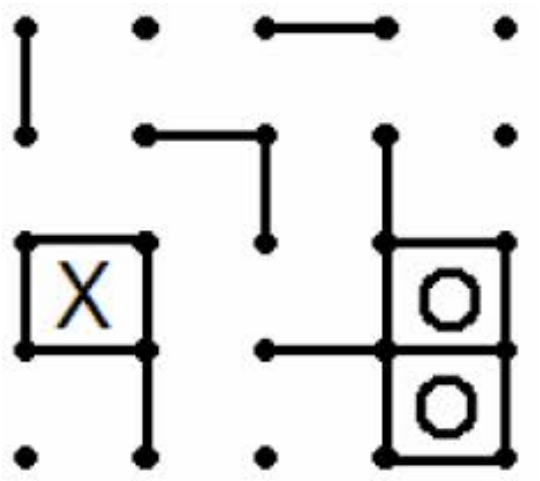
Programmeertalen

Erlang

Taco Walstra - Erik Kooistra - Jordy Perlee - Stephan van Schaik - Robin de Vries - Seyla Wachlin - Damian Frölich

---

Kamertje Verhuur



Deadline: Woensdag 9 maart 23:59

Introductie

Dit tweede blok van het vak Programmeertalen staat in het teken van gedistribueerde talen, waarbij we kijken naar Erlang. Omdat deze taal enige oefening vergt is er gekozen om iedereen even te laten wennen aan typische Erlang constructies in de vorm van een paar kleine opdrachten. Na de kleine opdrachten volgt een ietsje grotere puzzel in de vorm van een *gen_server* applicatie voor boter, kaas en eieren (tictactoe). Gedeeltes van deze applicatie zijn reeds gegeven en je moet deze applicatie alleen verder uitwerken. Vervolgens volgt de echte opgave waar je een simpele multiplayer architectuur gaat maken. Hierbij worden er computer processen gebruikt om de code parallel, m.a.w. naast elkaar, uit te kunnen voeren. Aangezien deze opdracht een vrij aardige uitdaging is, staan de logische stappen om de opdracht tot een goed einde te brengen in dit document beschreven.

Voor het nakijken worden weer tests gebruikt met behulp van het `eunit` framework, welke automatisch geïnstalleerd wordt bij het installeren van Erlang. Je mag ervan uitgaan dat je OTP versie 23 tot je beschikking hebt.

Introductie oefening met Erlang

Er zijn drie kleine opdrachten waarvoor reeds een skeleton file aangeleverd is op Canvas.

Pi

De eerste opdracht bestaat uit het uitrekenen van het getal pi met een beperkt aantal (5) decimalen via de Leibniz formule. Wanneer je de file **pi.erl** opent zie je de aanroep `pi(1,3,-1)` die vervolgens de functie eronder aan zal roepen. Voeg een paar regels code toe die pi recursief uitrekent en als resultaat pi returnt.

Users

Je ziet in de opdracht een lijstje van vier personen (`get_user/0`) met wat kenmerken. Aan jouw de taak om de overige drie functies te implementeren, waarbij de functies ook moeten werken voor andere lijsten van personen.

`get_females` moet een lijst geven van alle vrouwen uit het lijstje. `split_by_age` moet een lijst geven van personen jonger dan 18 jaar. `get_id_name` moet een lijst geven van het id-nummer en de naam van de personen.

shopping

Bij het college is het onderwerp van list comprehensions behandeld (ongeveer slide 50). Een van die slides heet ”list comprehension - your turn”. Welnu, dat is de opdracht.

Mergesort

Dit is een iets moeilijkere opdracht. Kijk eventueel eerst ook eens naar de code van quicksort en voeg eventueel nog extra `io:format` statements toe. Schrijf nu een module voor mergesort. Nog even ter herinnering wat mergesort doet:

```
L=[28,1,3,9,7,2,11].
```

De lijst wordt gesplitst in 2 gedeeltes: `[28,1,3,9]` en `[7,2,11]`. Dit doen we meerdere keren recursief tot er atomaire waardes ontstaan.

Daarna vindt de merge stap plaats: 2 getallen wisselen van plaats wanneer de tweede kleiner is dan de eerste:

1 - 28; 3 - 9; 2 - 7; 11

1 - 3 - 9 - 28; 2 - 7 -11

1 - 2 - 3 - 7 - 9 -11 -28

Gebruik de mergesort skeleton file and maak een uitbreiding op de manier zoals quicksort is aangepakt (gebruik bijvoorbeeld de ++ om twee list delen te combineren).

Mocht je toch nog tijd over hebben: probeer ook eens bubblesort.

Boter, Kaas en eieren

Voor deze opdracht is een skeleton file aangeleverd in de tictactoe map genaamd **tictactoe.erl**. Dit maakt gebruik van het concept `gen_server` in Erlang. Voor meer informatie zie `http://erlang.org/doc/man/gen_server.html` Het `gen_server` model implementeert een basis server in het client-servermodel. Daarnaast biedt `gen_server` automatisch mogelijkheden om fault tolerant code te schrijven, om bijvoorbeeld automatische failovers en restarts mogelijk te maken als de server, waar jouw programma op draait, crasht. De basis van jouw programma zal zitten in de funtie `handle_call/3`, deze wordt elke keer aangeroepen als jouw *process* een bericht krijgt.

Generic server

Omdat Erlang van nature een gedistribueerde taal is moet je op een gecontroleerde manier je state bijhouden. Binnen de `gen_server` is dat als volgende geïmplementeerd. Elke aanroep aan `handle_call/3` krijgt 3 waardes mee, het bericht, wie het gestuurd heeft en wat de huidige state is van de server. Wanneer je het bericht afgehandeld hebt en iets wilt terugsturen moet je een tuple teruggeven met de volgende waardes `{reply,Mesg,State}`, waar `Mesg` de waarde is die de caller terug krijgt en `State` de nieuwe interne staat van de server.

De implementatie

Het doel is om het spel boter kaas en eieren te spelen zodat je tegen een computer het spel kan spelen, en het bord kan weergeven. De bot tegen wie je speelt hoeft niet intelligent te zijn en mag ook telkens een random positie kiezen. Hiervoor moeten jullie de volgende calls en functies implementeren.

1. `get_board/0`, dit geeft het huidige board terug, deze moet je kunnen omzetten naar een format string met `show_board/1`.
2. `show_board/1`, dit zet een board om naar een format string representatie, welke je met `fwrite/1` kan printen. **Let op**: `show_board/1` moet precies terug geven wat wij verwachten, kijk daarom in de tests voor wat voorbeelden.
3. `restart/0` en `restart/1`, dit reset het spel zodat je weer een nieuw potje kan spelen. Waarbij `restart/1` het mogelijk maakt om met een bepaalde configuratie van het bord te spelen.
4. `move/2`, dit neemt een x en y coördinaat als argument en plaatst daar een rondje (hoofdletter o) als speler, mits de speler niet heeft gewonnen plaatst dit ook voor de computer (hoofdletter x) een vakje. Als een speler of computer een move maakt in een vakje dat al bezet is moet je de atom `not_open` terug geven. Als een speler of computer een move maakt in een ongeldige locatie, bijvoorbeeld (-10, -10), dan moet je de atom `not_valid` terug geven. Wanneer het spel is gewonnen moet er de volgende tuple terug gegeven worden `{won,1}` als jij gewonnen hebt en `{won,2}` als de computer gewonnen heeft. In alle andere gevallen kan je `ok` teruggeven.
5. `is_finished/0`, dit geeft aan of het spel afgelopen is.

Bij deze functies moet de volgende interne representatie gebruikt worden voor het bord en de zetten. Het bord is een lijst met daarin tuples van de volgende vorm: `{X-position,Y-position,PlayerID}`, waar de `PlayerID` 1 is voor jouw en 2 voor de computer. Stel, we hebben een bord met daarin een zet van jouw op plek $(0, 2)$, dus de x positie is 0 en de y positie is 2. Ook hebben we een zet van de computer op plek $(0, 1)$. Dan hebben we het volgende bord: `[{0,2,1},{0,1,2}]`, waarbij de eerste zet (die van ons) de eerste tuple in de lijst is. Die `[{0,2,1},{0,1,2}]` is dus wat de functie `get_board/0` moet terug geven.

Dit bord kan dan geprint worden met `print_board/1`, welke het volgende resultaat moet geven:

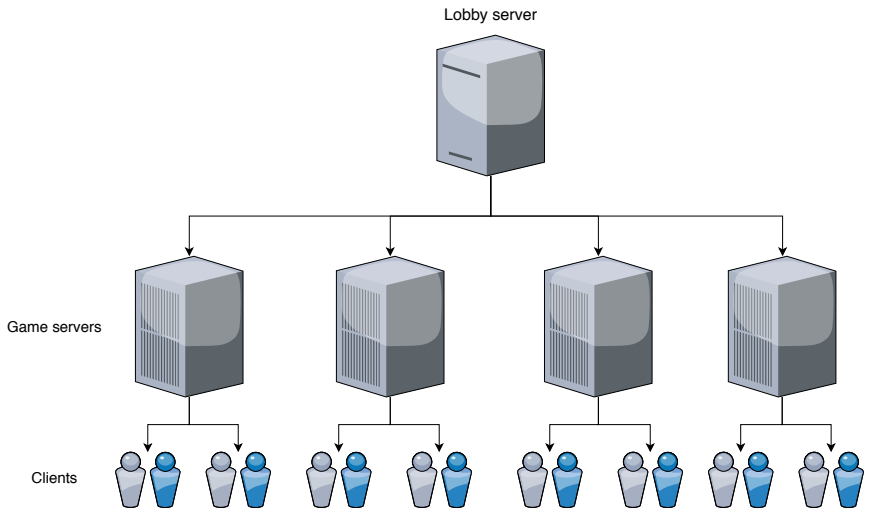
Shell snippet

```
1> c(tictactoe).
{ok,tictactoe}
2> tictactoe:print_board([{0,2,1},{0,1,2}]).
 | |
------
X| |
------
O| |
ok
3>
```

Vervolgens kan je het spel spelen in de erl shell door de `tictactoe` server aan te roepen met de berichten die je hier boven geïmplementeerd hebt.

Kamertje verhuur

Erlang wordt naast chat applicaties ook gebruikt voor multiplayer games `http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf`. In deze opgave gaan we kijken naar een kleine multiplayer architectuur, zie Figuur 1, waar we het spel kamertje verhuur kunnen spelen. Het doel van deze opdracht is jullie kennis te laten maken met onder andere: concurrency, isolation, fault tolerance en hoe Erlang daarbij kan helpen.



Figuur 1: De architectuur die we in deze opgave gaan maken.

Om het spel te kunnen spelen heb je een raster nodig bestaande uit m bij n punten. De beide spelers bouwen om de beurt een muur, die voorgesteld wordt als een lijn tussen twee naburige punten, met als doel een vierkant te vormen, m.a.w. een kamer te bouwen. Iedere keer dat er een kamer vervolledigd wordt, krijgt de speler die als laatste een lijn gezet heeft een punt. Als uiteindelijk alle kamers verhuurd zijn, dan wint de speler met de meeste punten.

De spelregels zijn dus als volgt:

- Het spel wordt gespeeld op een raster van zes bij zes groot.
- Iedere speler zet om de beurt een lijn tussen twee naburige punten.
- Als een hokje van één bij één gevormd is, dan levert dit een punt op voor de speler die als laatste een lijn heeft gezet.
- Als een speler met één lijn, twee kamers kan vervolledigen, dan levert dit twee punten op.
- Als een speler een punt heeft behaald, dan mag deze nog een zet doen.
- Als alle kamers verhuurd zijn, dan wint de speler met de meeste kamers.

Voor meer informatie, zie `http://nl.wikipedia.org/wiki/Kamertje_verhuren_(spel)`

De spelers

Om de twee spelers te simuleren wordt er gebruik gemaakt van computer processen. Deze processen moeten zich als volgt gedragen:

- Als er binnen een beurt een kamer vervolledigd kan worden, dan zet de speler hier zijn lijntje neer om een punt te behalen.
- Als er geen kamer vervolledigd kan worden, dan zet de computer een lijn op een willekeurige onbezette plek in het speelveld.

Opdracht 1A: het speelveld (ontwikkeld)

We beginnen met het implementeren van het speelveld in het bestand **grid.erl**. Het raster kunnen we voorstellen als een tuple met de breedte van het veld, de hoogte van het veld, en een lijst van gezette muren. Implementeer de `new(Width,Height)` functie die een leeg raster aanmaakt.

Shell snippet

```
1> c(grid).
{ok,grid}
2> grid:new(6, 6).
{6,6,[]}
3> grid:new(10, 6).
{10,6,[]}
```

De gespeelde muren worden gerepresenteerd door de coördinaten van de cellen waar de muur tussen ligt op te slaan in een tuple. bv. de tuple `{{4,3},{5,3}}` stelt de muur voor ten oosten van de cel op coördinaat $(4, 3)$ of ten westen van de cel op coördinaat $(5, 3)$. Implementeer de functie `get_wall(X,Y,Dir)` waarbij `Dir` een van de volgende atoms is: `north`, `east`, `south`, `west`. Deze functie geeft een tuple terug als volgt:

Shell snippet

```
1> c(grid).
{ok,grid}
2> grid:get_wall(1, 3, north).
{{1,2},{1,3}}
3> grid:get_wall(4, 3, east).
{{4,3},{5,3}}
4> grid:get_wall(0, 0, north).
{{0,-1},{0,0}}
```

Implementeer ook de functie `has_wall(Wall,Grid)`, die `true` geeft als een bepaalde muur gebouwd is, en de functie `add_wall(Wall,Grid)`, die een nieuw raster teruggeeft met de gebouwde muur.

Shell snippet

```
1> c(grid).
{ok,grid}
2> Grid = grid:new(6, 6).
{6,6,[]}
3> Grid2 = grid:add_wall(grid:get_wall(4, 3, east), Grid).
{6,6,[{{4,3},{5,3}}]}
4> grid:has_wall(grid:get_wall(5, 3, west), Grid).
false
5> grid:has_wall(grid:get_wall(5, 3, west), Grid2).
true
```

> Bekijk voor het maken van deze opdracht de documentatie van de `lists` (`http://www.erlang.org/doc/man/lists.html`) module.

Opdracht 1B: het speelveld visualiseren (ontwikkeld)

Nu dat we een data structuur hebben gemaakt voor het raster, moeten we het raster kunnen visualiseren. Dit gaan we doen door muren weer te geven met streepjes, waarbij '`--`' gebruikt wordt voor horizontale muren en '`|`' voor verticale muren. De knooppunten doen we met plusjes ('`+`'). De `print(Grid)` functie is al aangeleverd, maar maakt gebruik van twee helper functies: `show_hlines/2` en `show_vlines/2`. Implementeer de functies `show_hlines(Row,Grid)` en

`show_vlines(Row,Grid)` die respectievelijk de horizontale en de verticale lijntjes omzetten naar een format string. De `show_hlines/2` en `show_vlines/2` functies worden getest en daarom moet het gegeven formaat precies gevolgd worden.

Shell snippet

```
1> c(grid).
{ok,grid}
2> Grid = grid:new(3, 3).
{3,3,[]}
3> Grid2 = grid:add_wall(grid:get_wall(1, 0, west), Grid).
{3,3,[{{0,0},{1,0}}]}
4> Grid3 = grid:add_wall(grid:get_wall(0, 1, east), Grid2).
{3,3,[{{0,0},{1,0}},{{0,1},{1,1}}]}
5> Grid4 = grid:add_wall(grid:get_wall(1, 0, south), Grid3).
{3,3,[{{0,0},{1,0}},{{0,1},{1,1}},{{1,0},{1,1}}]}
6> Grid5 = grid:add_wall(grid:get_wall(0, 1, north), Grid4).
{3,3,
 [{{0,0},{0,1}},{{0,0},{1,0}},{{0,1},{1,1}},{{1,0},{1,1}}]}
7> grid:print(Grid5).

Game board:

+  +  +  +
   |
+--+--+  +
   |
+  +  +  +

+  +  +  +

ok
```

> Bekijk voor het maken van deze opdracht de documentatie van de `io` (`http://www.erlang.org/doc/man/io.html`) en `string` (`http://www.erlang.org/doc/man/string.html`) modules.

Opdracht 2A: willekeurige muren vinden (competent)

Het gedrag van de AI bestaat uit twee deelstappen. De eerste deelstap, het plaatsen van een muur naar willekeur, gaan we nu behandelen. Om die deelstap te implementeren gaan we stapsgewijs functies implementeren totdat we een willekeurige muur kunnen bouwen.

We willen eerst kunnen achterhalen welke muren er nog gebouwd kunnen worden. Om dit te doen beginnen we met een eenvoudige hulpfunctie `get_cell_walls(X,Y)` die voor een gegeven cel de lijst van alle mogelijke muren om die cel heen geeft. Met deze hulpfunctie kan dan `get_all_walls(W,H)` geïmplementeerd worden die voor een $(W \times H)$ grid alle mogelijke muren geeft (let op dat je lijst geen dubbele muren bevat). Door de muren die gebouwd zijn van de lijst van alle muren af te trekken kan dan een lijst van alle muren die nog gebouwd moeten worden gevonden worden. Implementeer dit als de functie `get_open_spots(Grid)` die een lijst van alle muren die nog gebouwd moeten worden teruggeeft.

> Bekijk voor het maken van deze opdracht de documentatie van de `random` (`http://www.erlang.org/doc/man/rand.html`) module. Met name de `rand:uniform` functie.

Opdracht 2B: willekeurige muren plaatsen (competent)

In deze sectie werken we in het bestand **client.erl**.

Nu dat we een lijst van alle muren die nog gebouwd moeten worden kunnen krijgen, kunnen we de `choose_random_wall(Grid)` functie implementeren. Deze functie geeft een willekeurige plaats terug waar nog geen muur gebouwd is.

Onze AI kan nu een willekeurige muur kiezen maar nog niet communiceren met de game server. Om te communiceren hebben we een client proces nodig die berichten kan ontvangen en versturen. In het bestand **client.erl** staan twee functies `new/0` en `move/0`. De `new/0` functie start een nieuw client proces en is al geïmplementeerd. De `move/0` functie moeten jullie implementeren en deze functie wacht op een bericht van de game server en doet aanleiding daarvan een actie. De volgende berichten moet de client kunnen ontvangen: `{move,ServerPid,Grid}` en `finished`. Het bericht met de `move` atom geeft aan dat het de beurt van de AI is en moet het bericht `{move,Wall}` naar de game server sturen, waarbij `Wall` een muur om te spelen is. Als de client het bericht met de atom `finished` ontvangt is het spel klaar en stopt de client. Vergeet niet om in je client de random number generator seed aan te passen.

Seeding the pseudo random number generator.

```
1 <<S1:32, S2:32, S3:32>> = crypto:strong_rand_bytes(12),
2 rand:seed(exs1024,{S1, S2, S3}),
```

> De game server is een `gen_server`, dus kijk hoe je communiceert met zo een server.

Opdracht 3: game server (gevorderd)

In deze sectie werken we in het bestand **game_server.erl**.

De game server is een implementatie van een `gen_server` en is verantwoordelijk voor 1 instantie van een spel. De volgende state wordt bijgehouden in de game server `{Grid,Players}`. `Grid` is het raster waarop gespeeld wordt en `Players` is een lijst met de spelers van het spel in de vorm van hun PID. De head van de `Players` lijst is de speler die aan zet is.

De game server moet nieuwe zetten aan het speelveld toevoegen en na een zet de juiste speler de beurt geven. Een speler kan een zet doen door `gen_server:call(ServerPid,{move,Wall})` aan te roepen, dus de game server moet de `handle_call` daarvoor implementeren. Deze handle call moet de `Wall` toevoegen aan het raster als het een geldige zet is, de score van de zet bepalen en als laatste bepalen wie de volgende zet mag doen (zie regels). Als response geeft de handle `{ok,Score}` terug, waarbij `Score` de score van de zet is.

Na de handle call wil je de speler die aan zet is informeren of alle spelers als het spel is afgelopen. Hiervoor kan je `continue` gebruiken (zie gen server documentatie). Informeren dat het spel is afgelopen doe je door de atom `finished` te sturen naar alle spelers. Een speler informeren dat die aan zet is doe je door de tuple `{move,ServerPid,Grid}` naar die speler te sturen.

Een ongeldige zet moet je afhandelen met een score van 0 en de beurt geven aan de volgende speler.

> Je eigen PID kan je aanvragen via self().

Opdracht 4: lobby server (expert)

In deze sectie werken we in het bestand **lobby_server.erl**.

We kunnen nu een game spelen, maar het beheren van meerdere games tegelijk is wat lastig. Om dat te versimpelen

gaan we een lobby server maken. De lobby server is de interface naar gebruikers en de manier waarop gebruikers een nieuw spel kunnen starten. Ook houdt de lobby server alle huidige games bij. De state van de lobby server is een lijst van games in de vorm van de PID van de game server. Implementeer de `new_game/3` functie die een nieuw spel start en deze aan de state van lobby server toevoegt. Sinds de lobby server de huidige games bijhoudt, moet je zorgen dat de lobby server geïnformeerd wordt als een game klaar is of als een game crasht[1].

Opdracht 5: smart AI (master)

Implementeer de functies in het bestand **grid.erl**

De huidige AI is redelijk simpel aangezien die gewoon een willekeurige muur pakt. We kunnen de AI iets slimmer maken door een muur te pakken die een kamer vervolledigt, mits zo een muur beschikbaar is. Hiervoor moet er voor iedere kamer bepaald kunnen worden of er nog maar één muur gebouwd hoeft te worden om deze te vervolledigen. Implementeer de functie `get_open_cell_walls(X,Y,Grid)` die voor een gegeven cel de plekken geeft waar nog muren gebouwd kunnen worden.

Nu kunnen we voor iedere kamer kijken of deze vervolledigbaar is door te kijken of er nog één enkele muur gebouwd hoeft te worden. Implementeer de `get_completable_walls(Grid)` functie die een lijst van muren geeft die een kamer vervolledigen als ze gebouwd worden.

Pas de client (**client.erl**) nu aan zodat deze een muur kiest die een kamer vervolledigt. Als zo een muur niet beschikbaar is wordt een willekeurige muur die een geldige zet vormt gekozen.

[1] Erlang maakt dit redelijk simpel, lees goed de mogelijkheden in de gen server documentatie en kijk naar de slides.