Programming languages

Distributed Programming in Erlang

Client-Server Model

- **Architecture for concurrent programming** where there is a server, which manages certain resources, and several clients, which send requests to the server to access its resources.
- **Client** and **server** are separate processes.
- For their communication, they use normal Erlang messages.
- They can be on the same machine – **concurrent programming.**
- Or on different machines – **distributed programming.**
    - The machines must be able to see each other.

Client and server

- The words *client* and *server* refer to the roles that processes play.
- The client always initiates a computation by sending a **request to the server.**
- The server calculates and sends a **response to the client.**
- For this, both must **know or send their corresponding PIDs**

Example

Client interface

Server function

```
-module(areas).
-export([server/0, client/2]).

client(Pid, Request) ->
        Pid !{self(), Request},
        receive
                Answer -> Answer
        end.

server() ->
        receive
                {From, {rectangle, Base, Height}} ->
                        From ! Base * Height,
                        server();
                {From, {circle, Radius}} ->
                        From ! 3.14159 * Radius*Radius,
                        server();
                {From, Other} ->
                        From ! {error, Other},
                        server()
        end.
```

Example

- **Server creation**

```
1> Pid = spawn(fun areas:server/0).
<0.36.0>
```

- **Client requests**

```
2> areas:client(Pid, {rectangle,6,8}).
48
3> areas:client(Pid, {circle,6}).
113.097239999999
4> areas:client(Pid, socks).
{error,socks}
```

Process linking

- Use it when one process depends on another.
- The **`link/1`** function is used.
- Both chained processes are monitored respectively:
  - If process A dies, an **exit signal** will be sent to B.
  - If process B dies, then A receives the signal.

Exit signal effect

- If the receiver does not perform special steps, the signal causes the receiver to also **die** (exit).
- If the receiver becomes a **system process**, it continues after receiving the signal and can react to it.

Linking example

```
on_exit(Pid, Fun) ->
     spawn(fun() ->
          process_flag(trap_exit, true),
          link(Pid),
          receive
               {'EXIT', Pid, Why} ->
               Fun(Why)
          end
     end).
```

Linking example

```
1> F = fun() ->
       receive
              X -> list_to_atom(X)
       end
   end.
2> Pid = spawn(F).
<0.61.0>
3> lib_misc:on_exit(Pid,
       fun(Why) ->
              io:format(" ~p died with:~p~n",[Pid, Why])
       end).
<0.63.0>
4> Pid ! hello.
hello
<0.61.0> died with:{badarg,[{erlang,list_to_atom,[hello]}]}
```

Distributed programming

- All the primitives seen for concurrent programming in Erlang have the same properties in distributed systems.
- Based on the **node** concept.
- **Node**: Erlang system running (**`erl`** execution) that can take part in distributed transactions.
- A distributed system consists of several nodes on one or several computers connected to a network.

Distributed applications

- Reasons to write them:
  - **Speed**
    - Parallel execution on multiple nodes.
  - **Reliability and Fault Tolerance**
    - Redundancy and multi-node cooperation.
  - **Access to resources residing on another node**
    - Database, peripherals, etc.
  - **Application inherent distribution**
    - Naturally distributed systems, such as for flight reservations.
  - **Extensibility**
    - Scale system capacity by adding new nodes.

Programming models

- **Distributed Erlang** (the one we will see): applications run in a *trusted environment* between tightly coupled computers.
  - Any node can perform any operation on any other Erlang node.
  - Applications typically execute in clusters on the same *LAN* behind a *firewall.*

- **Socket-based distribution**: applications that can run in *untrusted environments*.
  - Less powerful, but safer.

Magic cookie

- For 2 distributed Erlang nodes to communicate they must have the same **magic cookie**.

- **Methods**:
  1. Store it in **`$HOME/.erlang.cookie`**
  2. Start Erlang with:
     `erl-setcookie Cookie`
  3. Use the function:
     `erlang:set_cookie(Node,Cookie).`

Predefined functions

- `spawn(Node, Mod, Func, Args)` – spawns a process on a remote node.
- `spawn_link(Node, Mod, Func, Args)` – creates a remote process and link it to the process.
- `monitor_node(Node, Flag)` – if the Flag is true, it monitors the `Node` and if it fails or does not exist, it returns a message `{nodedown, Node}` to the process.
- `node()` – returns the name of the node itself.
- `nodes()` – list of known node names.
- `node(Element)` – returns the name of the `Pid`, reference or port, given as `Element.`
- `disconnect_node(Name)` – disconnects from the `Name` node.

Example: Banking server

- **Server code**

```
-module(bank_server).
-export([start/0, server/1]).

server(Data) ->
  receive
        {From, {deposit, Who, Amount}} ->
                From ! {bank_server, okay},
                server(deposit(Who, Amount, Data));
        {From, {consult, Who}} ->
                From ! {bank_server, search(Who, Data)},
                server(Data);
        {From, {withdraw, Who, Amount}} ->
                case search(Who, Data) of
                        undefined ->
                                From ! {bank_server, no},
                                server(Data);
                        Balance when Balance > Amount ->
                                From ! {bank_server, ok},
                                server(deposit(Who, -Amount, Data));
                        _ ->
                                From ! {bank_server, no},
                                server(Data)
                end
 end.
```

Example: Banking server

- **Server code** (Keep going…)

```
start() ->
  register(bank_server,
           spawn(bank_server, server, [[]])).

search(Who, [{Who, Value}|_]) ->
        Value;
search(Who, [_|T]) ->
        search(Who, T);
search(_, _) ->
        undefined.

deposit(Who, X, [{Who, Balance}|T]) ->
        [{Who, Balance+X}|T];
deposit(Who, X, [H|T]) ->
        [H|deposit(Who, X, T)];
deposit(Who, X, []) ->
        [{Who, X}].
```

Example: Banking client

- **Client code**

```
-module(bank_client).
-export([consult/1, deposit/2, withdraw/2]).

% long server name (name@machine)
bank_node() -> 'server@BAN280'.
% interface functions
consult(Who) ->
        call_bank({consult, Who}).
deposit(Who, Amount) ->
        call_bank({deposit, Who, Amount}).
withdraw(Who, Amount) ->
        call_bank({withdraw, Who, Amount}).
% client
call_bank(Message) ->
        Bank_node = bank_node(),
        monitor_node(Bank_node, true),
        {bank_server, Bank_node} ! {self(), Message},
        receive
                {bank_server, Answer} ->
                        monitor_node(Bank_node, false),
                        Answer;
                {nodedown, Matrix} ->
                        no
        end.
```

The name of the server must be included

Example: Banking transactions

- **Create two nodes on the same machine**
  1. Open two terminals
  2. Run in a terminal:
     `erl -sname server`
     1. Compile the server code:
        `c(bank_server).`
     2. Start the server: **`bank_server:start().`**
  3. Run in the other terminal:
     `erl -sname client`
     1. Compile the client code:
        `c(bank_client).`
     2. Send requests for consultation, deposit and withdrawal by the client:

```
bank_client:consult(Who).
bank_client:deposit(Who, Amount).
bank_client:withdraw(Who, Amount).
```