Lenguajes de programación

Programación Distribuida en Erlang

Modelo Cliente-Servidor

- **Arquitectura para programación concurrente** donde hay un servidor, que administra ciertos recursos, y un número de clientes, que mandan solicitudes al servidor para tener acceso a sus recursos
- El **cliente** y el **servidor** son procesos separados
- Para su comunicación utilizan mensajes normales de Erlang
- Ambos pueden estar en la misma máquina – **programación concurrente**
- O en máquinas distintas – **programación distribuida**
  - Las máquinas se deben de poder ver entre sí

Cliente y Servidor

- Las palabras *cliente* y *servidor* hacen referencia a los roles que juegan los procesos
- El cliente siempre inicia un cómputo mandándole una **solicitud al servidor**
- El servidor calcula y le manda una **respuesta al cliente**
- Para esto, ambos deben de **conocer o mandar sus PIDs** correspondientes

Ejemplo: Areas

Cliente

Servidor

```
-module(areas).
-export([ciclo/0, rpc/2]).

rpc(Pid, Solicitud) ->
        Pid ! {self(), Solicitud},
        receive
                Respuesta -> Respuesta
        end.

ciclo() ->
        receive
                {De, {rectangulo, Base, Altura}} ->
                        De ! Base * Altura,
                        ciclo();
                {De, {circulo, Radio}} ->
                        De ! 3.14159 * Radio * Radio,
                        ciclo();
                {De, Otra} ->
                        De ! {error, Otra},
                        ciclo()
        end.
```

Ejemplo: Areas

- **Creación del Servidor**

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

- **Solicitudes de Clientes**

```
2> areas:rpc(Pid, {rectangulo,6,8}).
48
3> areas:rpc(Pid, {circulo,6}).
113.09723999999999
4> areas:rpc(Pid, calcetines).
{error,calcetines}
```

Encadenamiento de Procesos

- Utilizarlo cuando un proceso depende de otro
- Se utiliza la función **`link/1`**
- Ambos procesos encadenados se monitorean respectivamente:
  - Si el proceso A muere, se le enviará una **señal de salida** a B
  - Si el proceso B muere, entonces A recibe la señal

Efecto de la Señal de Salida

- Si el receptor no realiza pasos especiales, la señal causa que el receptor también **muera** (salga)
- Si el receptor se convierte en un **proceso del sistema**, este prosigue después de la señal y puede reaccionar a la misma

Ejemplo de Encadenamiento

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

Ejemplo de Encadenamiento

```
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]}]}
```

Programación Distribuida

- Todos los primitivos vistos para programación concurrente en Erlang tienen las mismas propiedades en sistemas distribuidos
- Basada en el concepto de **nodo**
- **Nodo**: sistema Erlang ejecutándose (ejecución de **`erl`**) que puede tomar parte en transacciones distribuidas
- Un sistema distribuido consiste de varios nodos en una o varias computadoras conectadas a una red

Aplicaciones Distribuidas

- Razones para escribirlas:
  - **Velocidad**
    - Ejecución paralela en varios nodos
  - **Confiabilidad y Tolerancia a Fallas**
    - Redundancia y cooperación de varios nodos
  - **Acceso a recursos que residen en otro nodo**
    - Base de datos, periféricos, etc.
  - **Distribución inherente de la aplicación**
    - Sistemas naturalmente distribuidos como para la reservación de vuelos
  - **Extensibilidad**
    - Escalar la capacidad del sistema agregando nuevos nodos

Modelos de Programación

- **Erlang distribuido** (el que veremos): las aplicaciones se ejecutan en un *ambiente de confianza* entre computadoras fuertemente acopladas
  - Cualquier nodo puede realizar cualquier operación en cualquier otro nodo Erlang
  - Típicamente las aplicaciones se ejecutan en clústers sobre la misma *LAN* detrás de un *firewall*
- **Distribución basada en Sockets**: aplicaciones que pueden ejecutarse en *ambientes no confiables*.
  - Menos poderoso, pero más seguro

Galleta Mágica

- Para que 2 nodos Erlang distribuidos se comuniquen deben tener la misma galleta mágica (magic cookie)
- **Métodos**:
  1. Almacenarla en `$HOME/.erlang.cookie`
  2. Iniciar Erlang con: `erl -setcookie Cookie`
  3. Utilizar la función: `erlang:set_cookie(Nodo,Cookie).`

Funciones Predefinidas

- `spawn(Nodo, Mod, Func, Args)` – crea un proceso en un nodo remoto
- `spawn_link(Nodo, Mod, Func, Args)` – crea un proceso remoto y lo liga al proceso
- `monitor_node(Nodo, Bandera)` – si la Bandera es true monitorea al `Nodo` y en caso de que este falle o no exista regresa un mensaje `{nodedown, Nodo}` al proceso.
- `node()` – regresa el nombre del propio nodo
- `nodes()` – lista de nombres de nodos conocidos
- `node(Elemento)` – regresa el nombre del `Pid`, referencia o puerto dado como `Elemento`
- `disconnect_node(Nombre)` – se desconecta del nodo `Nombre`

Ejemplo: Banca

- **Código del Servidor**

```
-module(servidor_banco).
-export([inicio/0, servidor/1]).

servidor(Datos) ->
   receive
        {De, {deposita, Quien, Cantidad}} ->
                De ! {servidor_banco, ok},
                servidor(deposita(Quien, Cantidad, Datos));
        {De, {consulta, Quien}} ->
                De ! {servidor_banco, busca(Quien, Datos)},
                servidor(Datos);
        {De, {retira, Quien, Cantidad}} ->
                case busca(Quien, Datos) of
                        indefinido ->
                                De ! {servidor_banco, no},
                                servidor(Datos);
                        Saldo when Saldo > Cantidad ->
                                De ! {servidor_banco, ok},
                                servidor(deposita(Quien, -Cantidad, Datos));
                        _ ->
                                De ! {servidor_banco, no},
                                servidor(Datos)
                end
   end.
```

Ejemplo: Banca

- **Código del Servidor** (Continua…)

```
inicio() ->
   register(servidor_banco,
            spawn(servidor_banco, servidor, [[]])).

busca(Quien, [{Quien, Valor}|_]) ->
        Valor;
busca(Quien, [_|T]) ->
        busca(Quien, T);
busca(_, _) ->
        indefinido.

deposita(Quien, X, [{Quien, Saldo}|T]) ->
        [{Quien, Saldo+X}|T];
deposita(Quien, X, [H|T]) ->
        [H|deposita(Quien, X, T)];
deposita(Quien, X, []) ->
        [{Quien, X}].
```

Ejemplo: Banca

- **Código del Cliente**

```
-module(cliente_banco).
-export([consulta/1, deposita/2, retira/2]).

% nombre largo del servidor (nombre@máquina)
matriz() -> 'servidor@BAN280'.
% funciones de interfase
consulta(Quien) ->
        llama_banco({consulta, Quien}).
deposita(Quien, Cantidad) ->
        llama_banco({deposita, Quien, Cantidad}).
retira(Quien, Cantidad) ->
        llama_banco({retira, Quien, Cantidad}).
% cliente
llama_banco(Mensaje) ->
        Matriz = matriz(),
        monitor_node(Matriz, true),
        {servidor_banco, Matriz} ! {self(), Mensaje},
        receive
                {servidor_banco, Respuesta} ->
                        monitor_node(Matriz, false),
                        Respuesta;
                {nodedown, Matriz} ->
                        no
        end.
```

Se debe incluir el nombre del servidor

Ejemplo: Transacciones

- **Crear dos nodos en la misma máquina**
  1. Abrir dos terminales
  2. Ejecutar en una terminal:
     `erl -sname servidor`
     1. Compilar el código del servidor:
        `c(servidor_banco).`
     2. Iniciar el servidor: **`servidor_banco:inicio().`**
  3. Ejecutar en la otra terminal:
     `erl -sname cliente`
     1. Compilar el código del cliente:
        `c(cliente_banco).`
     2. Mandar solicitudes de consulta, depósito y retiro por parte del cliente:
        `cliente_banco:consulta(Quien).`
        `cliente_banco:deposita(Quien,Cantidad).`
        `cliente_banco:retira(Quien,Cantidad).`