Programación Declarativa Avanzada

Erlang 1

Juan Rodríguez Hortalá y el equipo de erlang.org

Referencias

Francesco Cesarini y Simon Thompson. **Erlang programming**. O'Reilly, c2009.

también disponible como recurso electrónico

Joe Armstrong. **Programming Erlang: software for a concurrent world**. Pragmatic Bookshelf, 2008.

Martin Logan, Eric Merritt, Richard Carlsson. **Erlang and OTP in action**. Manning, 2011.

también disponible como recurso electrónico

Documentación del **API**: http://www.erlang.org/erldoc

Erlang Reference Manual User's Guide v 5.9: http://www.erlang.org/doc/reference_manual/users_guide.html

El lenguaje Erlang

Erlang un lenguaje funcional impaciente, impuro y con tipado dinámico diseñado para construir sistemas soft real-time masivamente escalables y con requerimientos de alta disponibilidad

- Soporte integrado para la programación concurrente, distribuida y tolerante a fallos

- Concurrencia basada en el “actor model”

- Desarrollado por Ericsson y utilizado por empresas como Ericsson, Amazon, Facebook, T-Mobile, Telia, Nortel y proyectos como la Apache Software Foundation y Ubuntu

- Aplicación a telecomunicaciones, banca, comercio electrónico, bases de datos distribuidas, ...

Sintaxis básica

Muy pocos tipos de datos

- Números: integer y floats
- Binaries y Bitstrings
- Átomos
- Tuplas
- Listas
- Identificadores únicos: Pid, Ports, Referencias
- Funciones

String y char los char se representan mediante integer, los strings son listas de integer. Azúcar sintáctico:

```
> "hola" .
"hola"
> [104,111,108,97] .
"hola"
> $h.
104
```

Comentarios empiezan por `%`

Sintaxis básica
Números

Integer

tamaño arbitrario

se pueden escribir en cualquier base de 2 a 36

```
> 42 .
42
> -10 .
-10
> $9 .
57
> 2#101.
5
```

Float

64-bit IEEE 754-1985 representation (double precision)

notación de punto flotante

```
> 1.5 .
1.5
> 6.0e3 == 6000 .
true
```

Operadores típicos +,-,*,/, div, rem, ... y promoción automática de integer a float

```
> 4 + 4.5 .
8.5
```

Sintaxis básica
Binaries y Bitstring

Permiten manipular la memoria directamente

```
> term_to_binary("hola") .
<<131,107,0,4,104,111,108,97>>

> binary_to_term(<<131,107,0,4,104,111,108,97>>) .
"hola"
```

Sintaxis básica
Átomos y tuplas

Átomos como en Prolog son cadenas de caracteres sin espacios o entre comillas simples. No hay booleanos, los átomos ‘true’ y ‘false’ los sustituyen

```
> un_atomo .
un_atomo
> 'atomo con espacios' .
'atomo con espacios'
> true .
true
> 'atomo' == atomo .
true
```

Tuplas secuencia ordenada de términos Erlang de tamaño fijo

```
> {1, two, 3} .
{1,two,3}
> {complex, {nested, "structure", {here}}} .
{complex,{nested,"structure",{here}}}
> {} .
{}
```

Sintaxis básica
Listas

Listas igual que en Prolog, tenemos

- lista vacía `[]`
- cons `[a | []]`
- azúcar sintáctico `[a | [b | [c]]] == [a,b,c]`
- listas impropias `[a,b|c]`
- concactenación `[a,b] ++ [1,2] == [a,b,1,2]`

Otras funciones predefinidas y módulo `lists`

```
> length("hola") .
4
> lists:reverse("hola").
"aloh"
> lists:map(fun(X) -> X + 1 end, [1,2,3]) .
[2,3,4]
> lists:foldr(fun(X, S) -> X * S end, 1, [1,2,3]) .
6
```

Sintaxis básica
Pids, Ports y Referencias

Pids identificadores de proceso

Puertos similar a los Pids, se utilizan para conectar a Erlang con otros lenguajes como C

Referencias identificadores únicos empleados para simular llamadas síncronas entre procesos, para crear cookies, etc

Sintaxis básica
Funciones

Funciones la forma más básica de una función es una función anónima o lambda abstracción

```
> (fun(X) -> {X,X} end)(0) .
{0,0}
```

Orden superior (HO: higher order) las funciones son un tipo de datos, se pueden pasar funciones como argumentos de otras funciones, y se pueden devolver como valor resultado

```
> (fun(F) -> {F(0), F(0)} end)(fun(X) -> X + 1 end).
> (fun(F) -> {F(0), F(0)} end)((fun(Y) -> (fun(X) -> X + Y end) end)(5)).
```

Más adelante veremos como definir funciones con nombre más complejas usando módulos

Como se ve en el ejemplo una función `f` de n argumentos se aplica `f(exp1, ..., expn)` para `exp1, ..., expn` expresiones Erlang cualesquiera

En Erlang, ha diferencia de Haskell, no se soporta la currificación de funciones: todas las funciones se tienen que aplicar completamente

Esto se debe en parte a que, de forma similar a Prolog, se permite definir dos funciones con el mismo nombre pero con un número diferente de argumentos

Sistema de tipos

Tipado dinámico al igual que Prolog

- Como no hay muchos tipos de datos no hay demasiado que comprobar: no hay una jerarquía de clases ni declaraciones de tipos algebraicos estilo Haskell

- Sin embargo sí que se comprueba que los tipos se respetan en operaciones como por ejemplo la suma.

```
> 1 + "hola" .
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as 1 + "hola"
```

- Pero esta comprobación se realiza sólo en tiempo de ejecución: dinámicamente

```
> (fun(X) -> 0 end)(fun() -> 1 + "hola" end) .
0
```

como la suma `1 + "hola"` nunca llega a ejecutarse entonces no se lanza el error de tipos: **los tipos nunca pueden hacer que se rechace un programa que funciona correctamente**

Operadores de comparación

Operator	Description
==	Equal to
/=	Not equal to
=:=	Exactly equal to
=/=	Exactly not equal to
=<	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than

```
> "abc" =< "bcd" .
true
> abc =< bcd .
true
> {bb, abc} < {bb, bc} .
true
> 10 < hola .
true
> 1.0 == 1 .
true
> 1.0 =:= 1 .
false
> 1 =/= 1.0 .
true
> 1 =/= 1 .
false
> 1 /= 1.0 .
false
> 1 /= 1 .
false
```

Comparación lexicográfica

Al comparar expresiones de distintos tipos se utiliza el orden

number < atom < reference < fun < port < pid < tuple < list < binary

Diferencia entre la igualdad/desigualdad exacta (=:=, =/=) y aritmética (==, /=)

Pattern matching

Variables como en Prolog son identificadores que empiezan por mayúscula o con el caracter _ (variables anónmas)

Asignación única: mediante el operador =. En realidad no debe entenderse como una asignación, sino como la definición de un identificador, que es la variable

```
> X = 0 .
0
> X .
0
```

- durante el resto del cómputo la variable X tendrá el valor que le hemos asociado
- mediante el operador , (la coma) se pueden encadenar evaluaciones de expresiones. De esta manera podemos utilizar el valor de una variable definida anteriormente así `X = E, X` es una versión impaciente de `let X = E in X` de Haskell

```
> Y = 0, Z = Y + 1, {ok, Y, Z} .
{ok,0,1}
```

- llamamos variables ligadas a las que ya han sido definidas mediante el uso del operador = o durante el paso de parámetros al evaluar una llamada a una función, y variables libres a aquellas cuyo valor aún no ha sido definido

Pattern matching

Matching acabamos de ver una forma restringida del operador =, cuya forma general es

```
Pattern = Expression
```

- **Pattern** (patrón) es una expresión Erlang construida empleando variables libres o ligadas, literales (números, caracteres o átomos), listas y tuplas
- **Expression** es cualquier expresión Erlang

Para evaluar **P = E** se comienza evaluando **E** y si esta evaluación tiene éxito se intenta encontrar una asignación de las variables libres de **P** que al aplicarse a **P** la haga igual a **E**. Si esta asignación existe entonces se dice que el proceso de ***pattern matching*** (encaje de patrones) tiene éxito, y en otro caso que se produce un fallo

```
> {X, [1, Z]} = (fun(U) -> {U, [U, 2]} end)(1), {X, Z} .
{1,2}
```

También se pueden usar patrones en las cabezas de las funciones anónimas:

```
> (fun({X, Y}) -> Y end)({ok,42}) .
42
```

equivale a

```
> (fun(Z) -> {U, V} = Z, V end)({ok, 42}) .
42
```

Pattern matching y evaluación impaciente

El pattern matching se utiliza para

definir los valores de las variables

extraer valores de datos compuestos `fun({X, Y}) -> Y end`

controlar el flujo de la ejecución de los programas

Evaluación impaciente

ya hemos visto que durante el matching siempre se evalúa la expresión del lado derecho aunque no sea necesaria: **evaluación impaciente**

```
> X = 0 + a, 0 .
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as 0 + a
> X = 20, 0 .
0
```

Erlang también realiza evaluación impaciente para las aplicaciones de función ~ **paso por valor**

```
> (fun(X) -> 0 end)(0 + a) .
** exception error: bad argument in an arithmetic expression
     in operator  +/2
```

¿Qué es un valor?

Valor una expresión cuya evaluación ya ha terminado

- un literal (número, carácter, o átomo) es un valor
- un binary o bitstring es un valor
- un identificador (pid, port o referencia) es un valor
- una tupla de valores es un valor
- una lista que sólo contiene valores es un valor
- una función sin aplicar es un valor

La evaluación de las funciones se detiene hasta que se indican los argumentos de la función, aunque el cuerpo de la función se pueda evaluar

```
> (fun(X) -> 0 end)(fun() -> 1 + "hola" end) .
0
```

- hay muchas funciones cuyo cuerpo no queremos que se evalúe inmediatamente

```
> F = fun() -> io:fwrite("hola~n") end .
> case io:get_line("escribo?(si/no)>") of "si\n" -> F(); _ -> ok end .
```

- las funciones sin argumentos están esperando a que se les pasen los argumentos vacíos ()

```
> (fun() -> 1 + "hola" end)() .
** exception error: bad argument in an arithmetic expression ...
```

Funciones

Usando pattern matching podemos definir funciones con varias reglas (cláusulas) que se prueban en orden: una definición de función es una secuencia de cláusulas separadas por `;` y terminada en `.`

```
fact(0) -> 1;
fact(N) -> N * fact(N-1) .
```

La recursión es el recurso expresivo básico para definir funciones complejas, como es habitual en los lenguajes funcionales

El flujo de ejecución del programa se basa en el pattern matching

```
area({square, Side}) ->
  Side * Side ;
area({circle, Radius}) ->
  math:pi() * Radius * Radius;
area({triangle, A, B, C}) ->
  S = (A + B + C)/2,
  math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) ->
  {error, invalid_object}.
```

Es muy común usar tuplas de la forma `{Descriptor, Valor}` como en `{square, Side}` para definir estructuras compuestas, a falta de otros recursos como las clases de POO o los tipos algebraicos de Haskell y ML

Funciones

Guardas condiciones adicionales para la aplicación de una clásula

```
fact(0) -> 1;
fact(N) when N > 0 -> N * fact(N-1) .
```

Repertorio limitado de expresiones permitidas como guardas http://www.erlang.org/doc/reference_manual/expressions.html#id79005. Algunos ejemplos:

```
number(X)  % X is a number
integer(X) % X is an integer
float(X)   % X is a float
atom(X)    % X is an atom
tuple(X)   % X is a tuple
list(X)    % X is a list

length(X) == 3 % X is a list of length 3
size(X) == 2   % X is a tuple of size 2.

X > Y + Z  % X is > Y + Z
X == Y     % X is equal to Y
X =:= Y    % X is exactly equal to Y
```

Todas las variables que aparecen en las guardas deben estar ligadas

Otras estructuras de control

No aumentan la expresividad del lenguaje pero son cómodas

Case: distinción de casos

```
area(Shape) ->
    case Shape of
        {circle, Radius} -> Radius * Radius * math:pi();
        {square, Side} -> Side * Side;
        {rectangle, Height, Width} -> Height * Width
    end.
```

If-then-else: comportamiento peculiar

```
is_greater_than(X, Y) ->
    if
        X>Y ->
            true;
        true -> % works as an 'else' branch
            false
    end
```

Excepciones

Try: forma general

```
try Exprs
catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
end
```

Class - Origen

- error - Run-time error, por ejemplo `1+a`
- exit - El proceso llamó a `exit/1`
- throw - El proceso llamó a `throw/1`

```
try
    some_unsafe_function()
catch
    oops -> got_throw_oops;
    throw:Other -> {got_throw, Other};
    exit:Reason -> {got_exit, Reason};
    error:Reason -> {got_error, Reason}
end
```

También se pueden lanzar excepciones usando `throw`:

```
> try throw(hola) catch C:R -> {C, R} end .
{throw,hola}
```

BIFs

BIFs = Built-In Function primitivas adicionales del lenguaje implementadas en C

- La mayoría están auto importadas: módulo **`erlang`**
- Permiten hacer cosas dificiles o imposibles de implementar usando el resto del lenguaje: operadores aritméticos, impurezas (I/O), algo de metaprogramación, optimizaciones (pej operaciones sobre listas implementadas en C) ...

```
> tuple_size({a,b,c}).
3
> atom_to_list('Erlang').
"Erlang"
> apply(lists, reverse, [[a, b, c]]).
[c,b,a]
> io:fwrite("hola ~w~n",[pepe]) .
hola pepe
ok
> self() .
<0.38.0>
```

Sistema de módulos

```
%% This is a simple Erlang module
-module(my_module).
-export([pi/0]).

pi() -> 3.14.
```

Se guardan en archivos con extensión **`.erl`** y el mismo nombre que módulo: ej **`my_module.erl`**

Directivas empiezan por **`-`** y la directiva **`-module`** siempre es obligatoria

- **`-export`** especifica la lista de funciones exportadas por el módulo
  - . las funciones se identifican por su nombre y su aridad (número de argumentos)
  - . para usar funciones exportadas por otros módulos se tiene que cualificar
  - . las funciones locales al módulo no hace falta cualificarlas

```
-module(mod).
-export([inc_list/1]).

% llamada cualificada a la función lists del módulo map
% nótese la sintaxis para pasar funciones definidas como argumentos de funciones HO
inc_list(Xs) -> lists:map(fun inc/1, Xs) .

% función local
inc(X) -> X + 1 .

> mod:inc_list([0,1,2]) .
   % paso de función cualificada como argumento HO
> apply(fun lists:map/2, [fun(X) -> X + 1 end, [0,1]]) .
```

Sistema de módulos

`-compile(export_all)` permite exportar todas las funciones del módulo

`-import(Module, [Function/Arity,...])` permite importar funciones de otros módulos y llamarlas localmente

```
-import(math, [sqrt/1]).
area({triangle, A, B, C})  ->
  S = (A + B + C)/2,
  sqrt(S*(S-A)*(S-B)*(S-C));
```

Se permiten funciones con el mismo nombre y diferente número de argumentos

```
-module(my_list) .
-export([reverse/1]).

reverse(Xs) -> reverse(Xs, []) .

reverse([], Acc) -> Acc ;
reverse([X|Xs], Acc) -> reverse(Xs, [X|Acc]) .
```

Recursión de cola: la llamada recursiva aparece en la última instrucción --> permite optimizaciones del compilador, uso de funciones auxiliares con acumulador

Procesos Erlang

Procesos ligeros

- no del SO sino de la máquina virtual de Erlang
- crear un proceso Erlang tiene un coste similar a crear un objeto en Java

No hay (casi) memoria compartida entre los procesos

Paso de mensajes asíncrono

Bien adaptado a arquitectura multicore

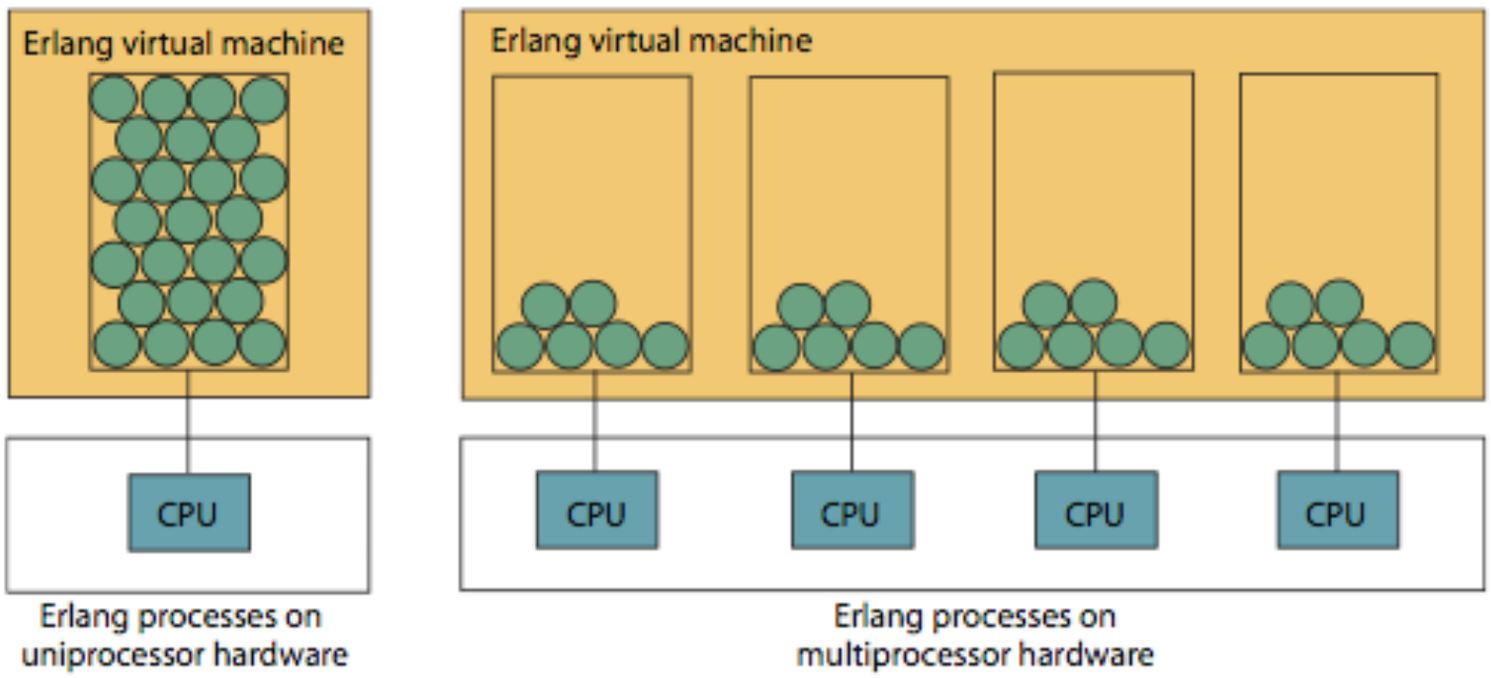


Figure 1.1 Erlang processes running on uniprocessor and on multiprocessor hardware, respectively. The runtime system automatically distributes the workload over the available CPU resources.

Procesos Erlang

Creación de procesos

```
Pid = spawn(io, format, ["erlang!~n"])
```

Paso de mensajes asíncrono como los procesos no comparten memoria entonces se comunican mandándose mensajes

- no pueden hacer asignaciones a variables compartidas porque no se comparte memoria: más adelante veremos como simular memoria compartida
- cualquier término Erlang se puede pasar como mensaje `Pid ! mensaje`
- se hace una copia del mensaje antes de mandarlo: no queremos compartir memoria!

Asíncrono la máquina virtual de Erlang no bloquea al proceso emisor del mensaje hasta que el receptor confirma la recepción del mismo. El paso de mensaje síncrono se puede implementar manualmente

Procesos Erlang

El buzón o mailbox

- cada proceso tiene un buzón asociado donde se van almacenando los mensajes
- el buzón es una estructura FIFO: una cola
- para leer los mensajes de su buzón los procesos utilizan la instrucción receive

```
r2() ->
     receive
          {inc,X} -> io:format("r2: ~w~n", [X+1]);
     end .

> Pid = spawn(buzon, r2, []) .
<0.33.0>
> Pid ! {inc, 41} .
r2: 42
{inc,41}
```

Forma (casi) general de receive similar a un case

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

Demo (buzon.erl)

- **Process** - A concurrent activity. A complete virtual machine. The system may have many concurrent processes executing at the same time.
- **Message** - A method of communication between processes.
- **Timeout** - Mechanism for waiting for a given time period.
- **Registered Process** - Process which has been registered under a name.
- **Client/Server Model** - Standard model used in building concurrent systems.

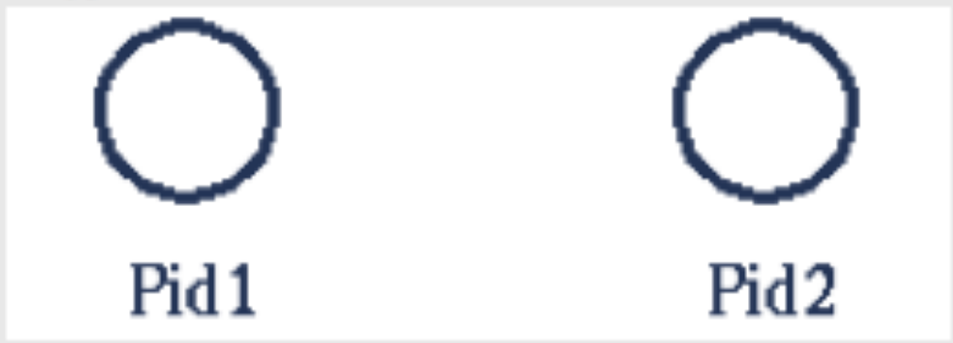
Creating a New Process

Before:

Code in Pid1

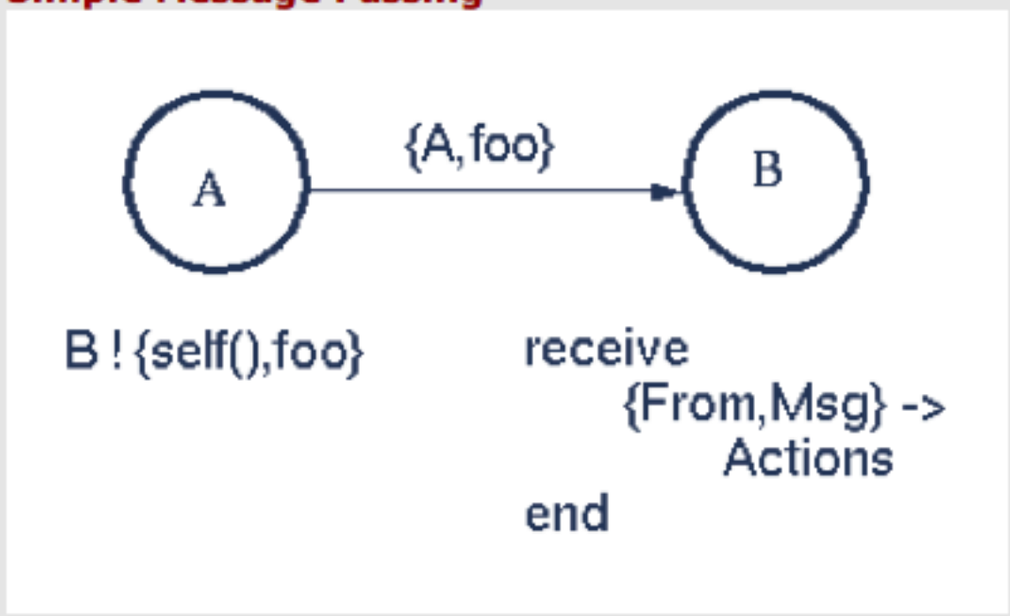
Pid2 = spawn(Mod, Func, Args)

After



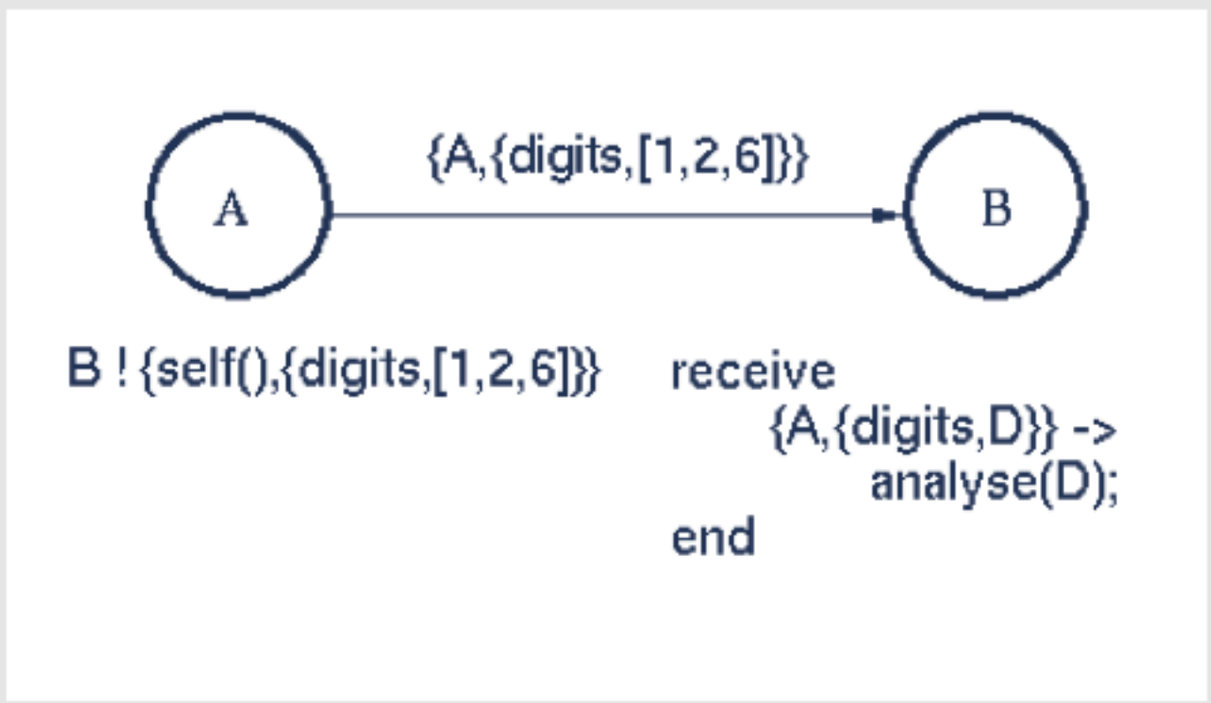
Pid2 is process identifier of the new process - this is known only to process Pid1.

Simple Message Passing



self() - returns the Process Identity (Pid) of the process executing this function.

From and **Msg** become bound when the message is received. Messages can carry data.

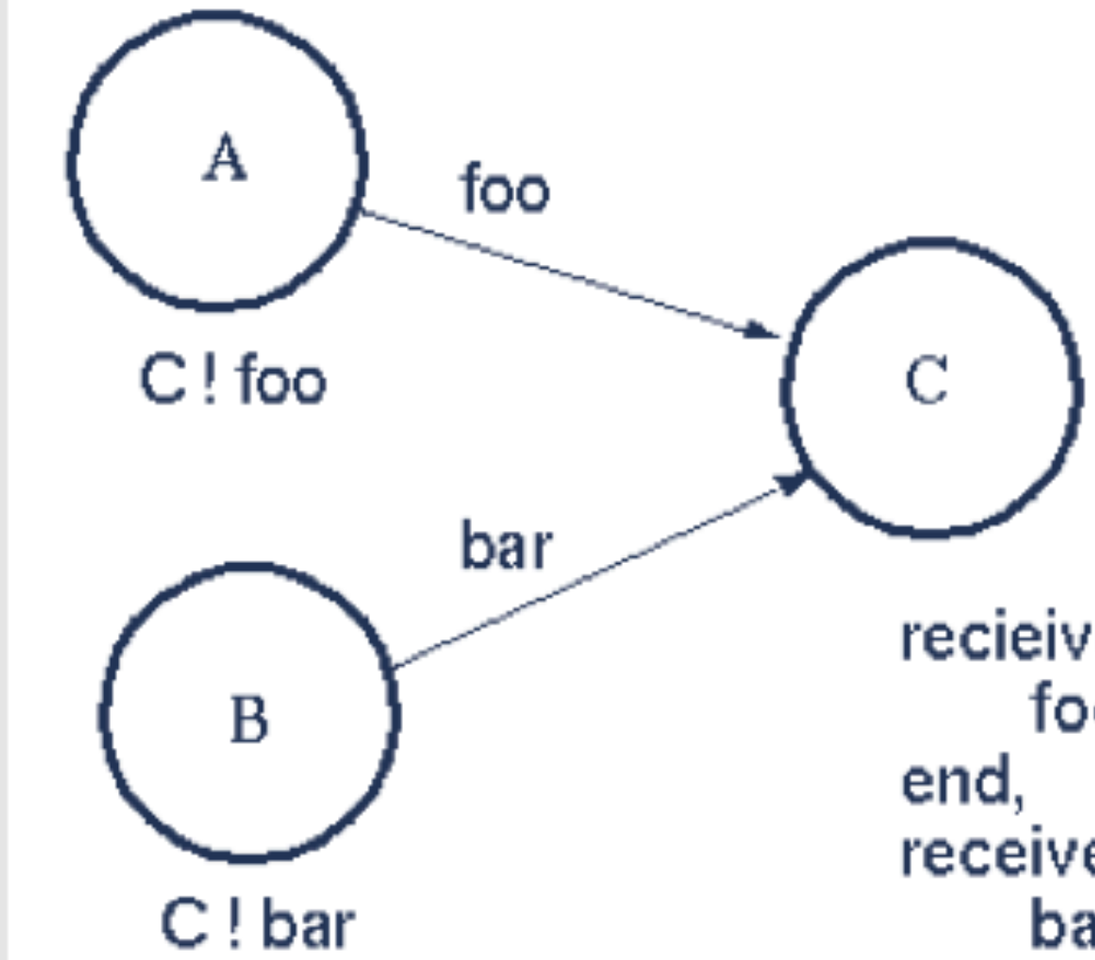


- Messages can carry data and be selectively unpacked.
- The variables **A** and **D** become bound when receiving the message.
- If **A** is bound before receiving a message then only data from this process is accepted.

An Echo process

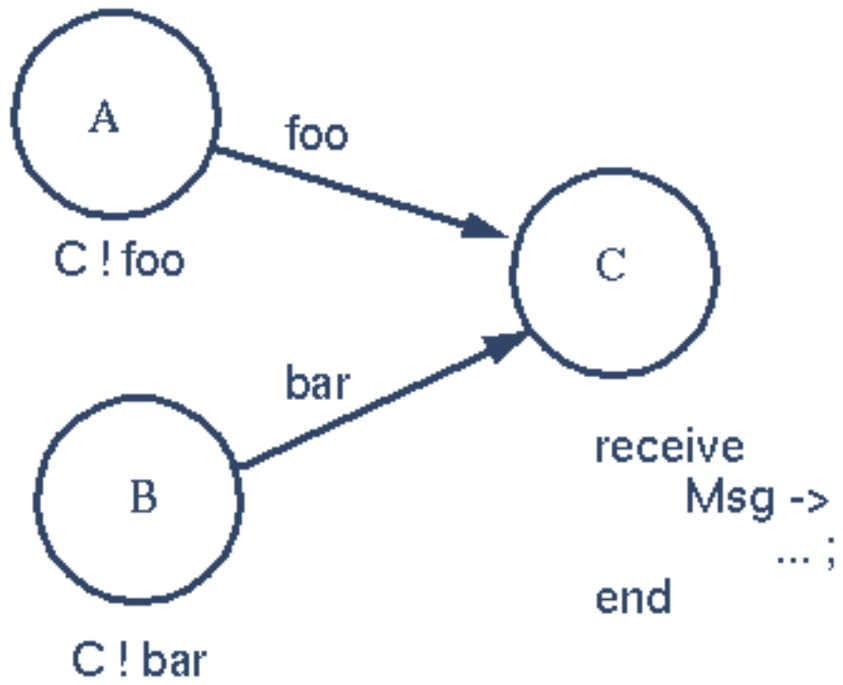
```
-module(echo).
-export([go/0, loop/0]).

go() ->
        Pid2 = spawn(echo, loop, []),
        Pid2 ! {self(), hello},
        receive
                {Pid2, Msg} ->
                        io:format("P1 ~w~n",[Msg])
        end,
        Pid2 ! stop.

loop() ->
        receive
                {From, Msg} ->
                        From ! {self(), Msg},
                        loop();
                stop ->
                        true
        end.
```

Selective Message Reception



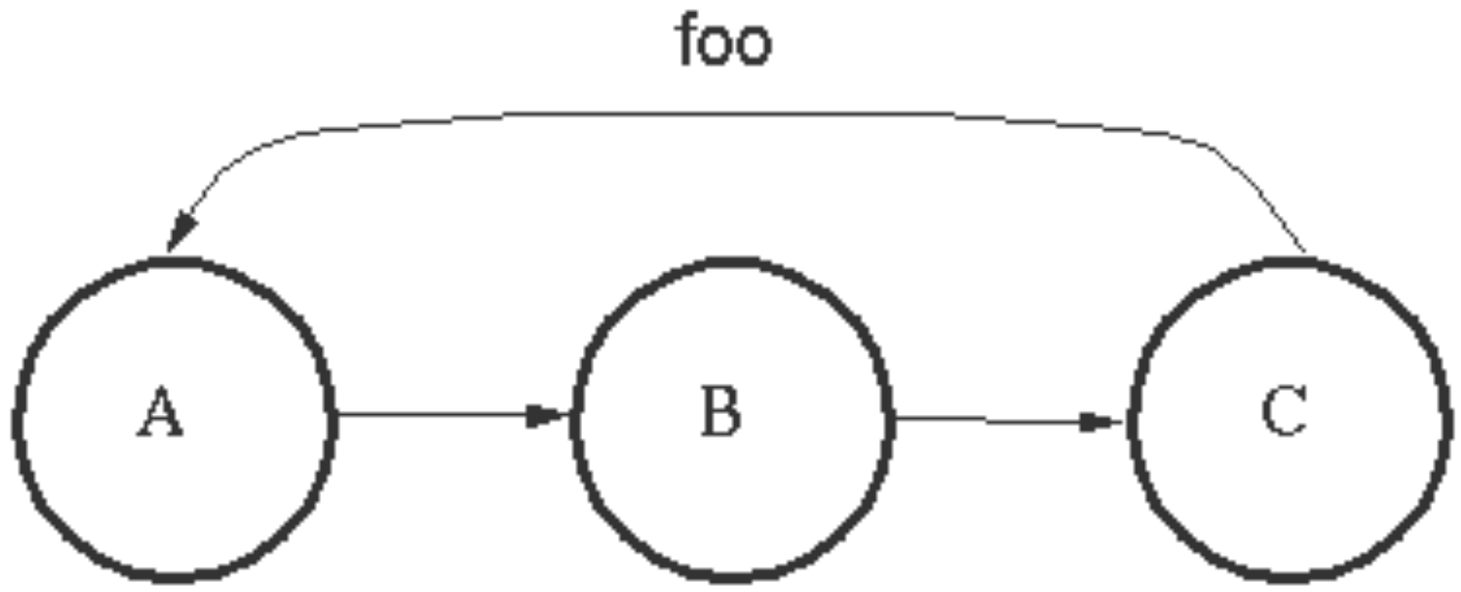
The message **foo** is received - then the message **bar** - irrespective of the order in which they were sent.

Selection of any message



The first message to arrive at the process **C** will be processed - the variable **Msg** in the process **C** will be bound to one of the atoms **foo** or **bar** depending on which arrives first.

Pids can be sent in messages




```
B ! {transfer,self()}

                                receive
                                    {transfer, A} ->
                                        A ! foo
                                end

receive
    {transfer, A} ->
        C ! {transfer,A}
end
```

- **A** sends a message to **B** containing the Pid of **A**.
- **B** sends a transfer message to **C**.
- **C** replies directly to **A**.

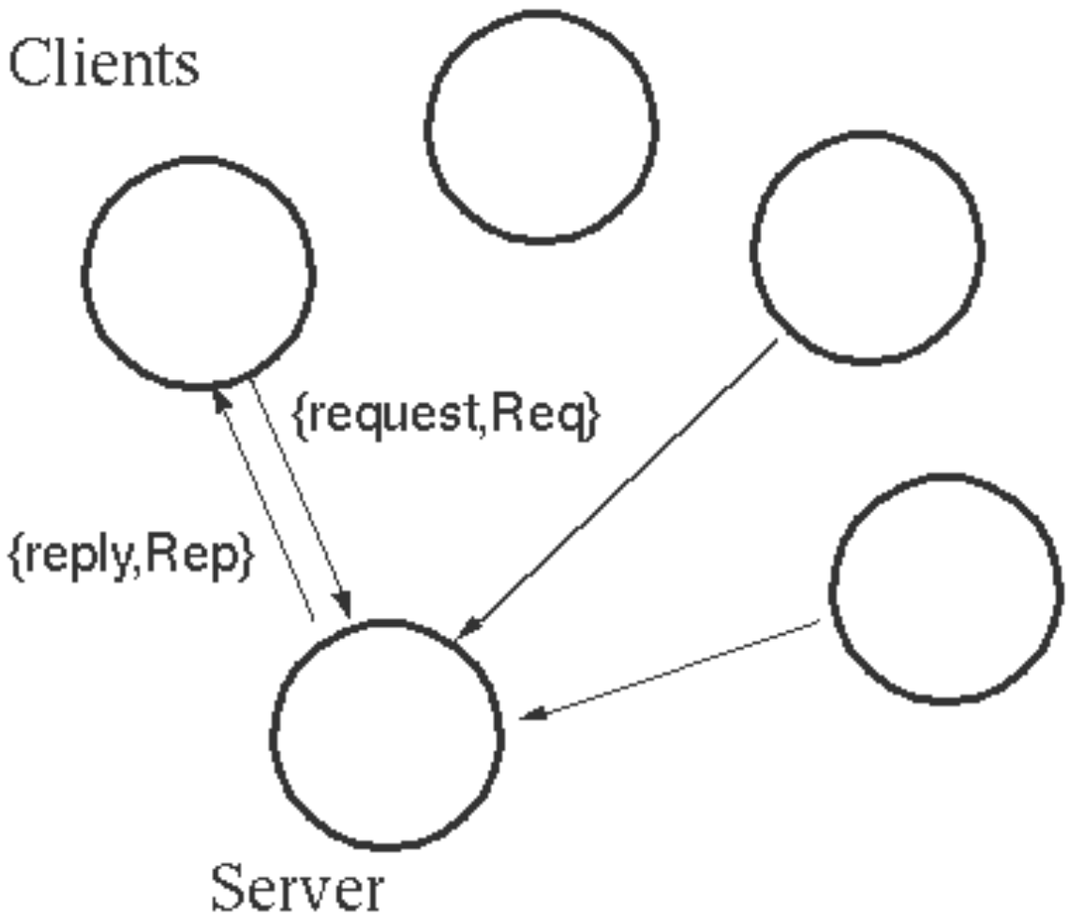
Registered Processes

register(Alias, Pid) Registers the process **Pid** with the name **Alias**.

```
start() ->
        Pid = spawn(num_anal, server, [])
        register(analyser, Pid).

analyse(Seq) ->
        analyser ! {self(),{analyse,Seq}},
        receive
                {analysis_result,R} ->
                        R
        end.
```

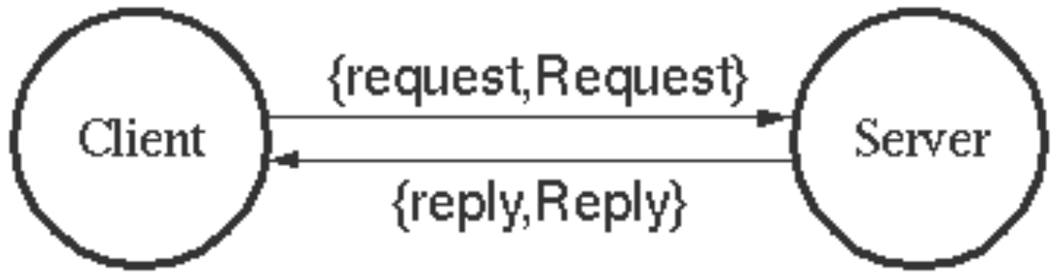
Any process can send a message to a registered process.

Client Server Model



Protocol

- Protocol



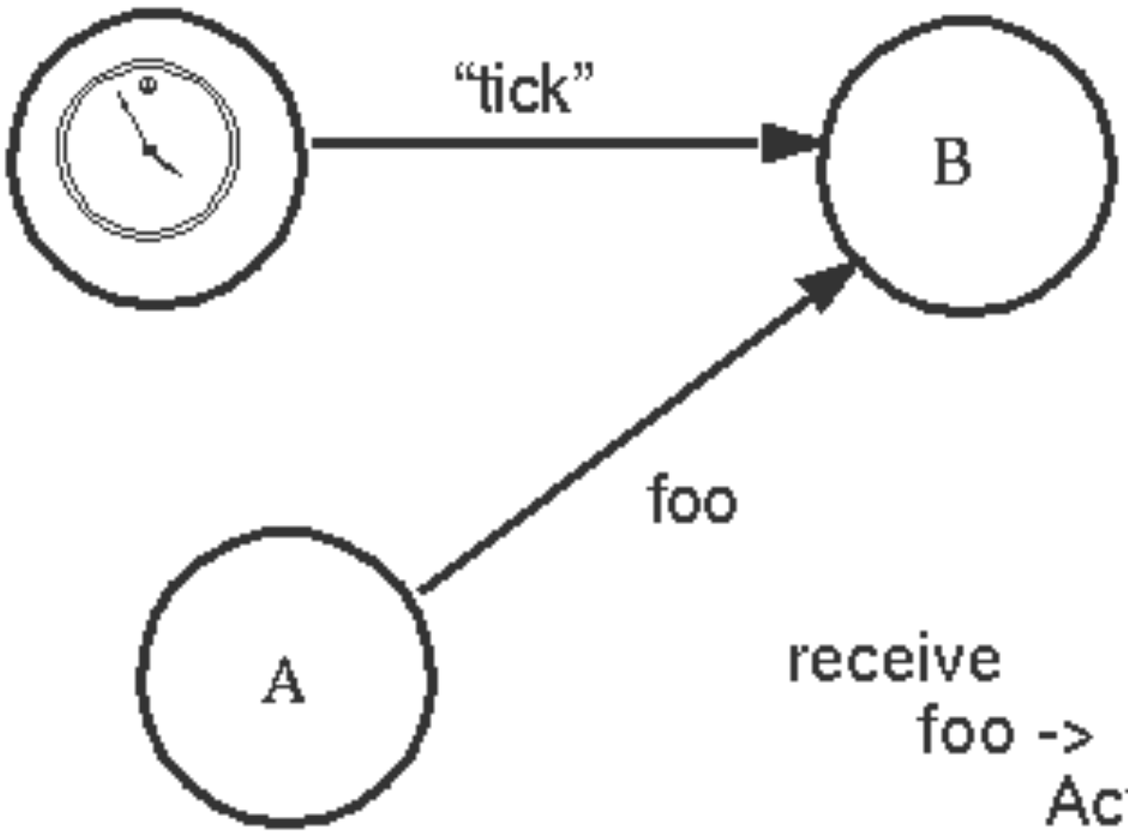
Interface Library

```
-export([request/1]).

request(Req) ->
        myserver ! {self(),{request,Req}},
        receive
                {myserver,{reply,Rep}} ->
                        Rep
        end.
```

Server code

```
-module(myserver).

server(Data) ->
        receive
                {From,{request,X}} ->
                        {R, Data1} = fn(X, Data),
                        From ! {myserver,{reply, R}},
                        server(Data1)
        end.
```

Timeouts




```
receive
      foo ->
            Actions1;
after
      Time ->
            Actions2
end
```

If the message **foo** is received from **A** within the time **Time** perform **Actions1** otherwise perform **Actions2**.

Uses of Timeouts

sleep(T)- process suspends for **T** ms.

```
sleep(T) ->
   receive
   after
      T ->
         true
   end.
```

suspend() - process suspends indefinitely.

```
suspend() ->
   receive
   after
      infinity ->
         true
   end.
```

Uses of Timeouts

alarm(T, What) - The message **What** is sent to the current process iin **T** miliseconds from now

```
set_alarm(T, What) ->
   spawn(timer, set, [self(),T,What]).

set(Pid, T, Alarm) ->
   receive
   after
      T ->
         Pid ! Alarm
   end.
receive
   Msg ->
      ... ;
end
```

flush() - flushes the message buffer

```
flush() ->
   receive
      Any ->
         flush()
   after
      0 ->
         true
   end.
```

A value of 0 in the timeout means check the message buffer first and if it is empty execute the following code.

BIFs interesantes acerca de procesos

`spawn(Fun)`, `spawn(Module, Function, Args)` crea un nuevo proceso que ejecuta la función que se pasa como argumento, y devuelve el pid del nuevo proceso
`self()` devuelve el pid del proceso que la invoca
`processes()` devuelve una lista con los pid de todos los procesos en ejecución
`is_process_alive(Pid)` devuelve `true` si el proceso existe y esta vivo, es decir, no se ha terminado su ejecución, en otro caso devuelve `false`
`register(RegName, Pid)` asocia el átomo `RegName` al identificador de proceso `Pid`, de forma que `RegName` pueda ser usado como una alias de `Pid` en los usos subsiguienes del operador `!`. Falla si `Pid` no corresponde a un proceso en ejecución, `RegName` ya estaba en uso, `Pid` ya estaba registrado o `RegName` es el átomo `undefined`
`registered()` devuelve la lista de átomos que corresponden a los nombres de los procesos registrados actualmente por medio de llamadas a `register/2`
`unregister(RegName)` deshace la asociación entre `RegName` y el pid al que fue asociado mediante `register/2`. Falla si `RegName` no corresponde a ningún pid registrado (no aparece en la lista devuelta por `registered/0`)

BIFs interesantes acerca de procesos

`exit(Pid, Reason)` envía una exit signal con `Reason` como exit reason al proceso correspondiente a `Pid`

- si `Reason` vale `kill` entonces el proceso se termina incondicionalmente con `killed` como exit reason
- si `Reason` vale `normal` entonces el proceso no termina: útil en combinación con error trapping
- si `Reason` toma otro valor entonces el proceso termina con `Reason` como exit reason

`pman:start()` pone en marcha el gestor de procesos gráfico (process manager), que se abre en una ventana nueva. Seleccionar “Hide System Processes” suele ayudar, ver http://www.erlang.org/doc/apps/pman/pman_chapter.html

`toolbar:start()` abre una ventana en la que podemos abrir el gestor de procesos y otras aplicaciones gráficas de monitorización de programas Erlang

Demo (`pman` y `exit/2`)

```
> Pid = spawn(fun() -> receive _ -> io:fwrite("proceso hijo terminado") end end).
> is_process_alive(Pid), exit(Pid, normal), is_process_alive(Pid), exit(Pid, timeout),
is_process_alive(Pid)
```

más información en la documentación del módulo `erlang` http://www.erlang.org/doc/man/erlang.html y la documentación de Erlang http://www.erlang.org/erldoc