0. Erlang

Sequential Programming

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

FIB

Why Erlang?

- We need a language to implement some of the algorithms that we will learn
- Erlang is a general-purpose language and runtime environment well suited for scalable concurrent and distributed programming
  - Developed at Ericsson in late eighties
  - Functional Programming (e.g. Prolog, Lisp)
  - Available as free and open-source software
  - Built-in support for concurrency, distribution, and fault tolerance

Why Erlang?

- Companies use Erlang in their production systems, e.g. Amazon, Yahoo!, Facebook, WhatsApp, T-Mobile, Motorola, Ericsson, …
  - https://en.wikipedia.org/w/index.php?title=Erlang_(programming_language)&oldid=754567048#Companies_using_Erlang
- Popular applications use Erlang, e.g. Chef, Ejabberd, CouchDB, GitHub, RabbitMQ, …
  - https://en.wikipedia.org/w/index.php?title=Erlang_(programming_language)&oldid=754567048#Software_projects_written_in_Erlang

Relevant primitive data types

- atom: literal, constant with name (foo, my_F, ...)
  - To be enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, '_', or '@'
    - 'Monday', 'phone number', ...
  - Atoms true and false used to denote Boolean values
    - operators: not, and, or, xor, andalso, orelse
- number
  - integers: 10, -24, 16#A42B (i.e., base#value), $a ($char, i.e., ASCII code of the character char), ...
  - floats: 17.3, -56.62, 2.3e3, ...
  - ops: +, -, *, /, div, rem, bnot, band, bor, bxor, bsl, bsr

Relevant primitive data types

- reference: globally unique term

```
> make_ref().
#Ref<0.2853579700.305135618.16717>
```

- fun: to create an anonymous function
  - Can be passed as argument to higher-order functions

```
> F = fun (X) -> X+1 end.
#Fun<erl_eval.6.39074546>
> F(2).
3
```

- pid: process identifier (PID)
  - `self()` returns the PID of the calling process: <0.76.0>

Relevant compound data types

- tuple: used to store a **fixed** number of terms
  - {123, def, abc}
  - {person, 'Joe', 'Armstrong'}
  - There exist a number of BIFs to manipulate tuples
- list: used to store a **variable** number of terms
  - [] (empty list)
  - [foo, 12, bar, zot]
  - See lists module for list processing functions
- **"..."** is shorthand for the list of ASCII codes of the enclosed chars within the quotes
  - "abcdefghi" is [97,98,99,100,101,102,103,104,105]

Term comparison

- Operators: =<, <, >=, >, == (equal to), =:= (exactly equal to: same value and type), /= (not equal to), =/= (exactly not equal to)
- The following order is used to compare terms of different data types
  - *number < atom < reference < fun < port < pid < tuple < map < nil < list < bit string*
  - Lists compare element by element
  - Tuples compare by size and then element by element

1==1.0	1=:=1.0	1 > ab	[1,2] < [2,1]	{1,1} < {2}
true	false	false	true	false

Variables

- Used to store values of terms
- The scope for a variable is its function clause
- Variables can only be bound **once**
  - The value of a variable can never be changed
- They start with an upper-case letter (or '_')
  - Abc, A_var, _Foo
  - Variables starting with '_', do not generate warnings when unused
  - The **anonymous variable '_'** can be used when a variable is required but its value can be ignored

Pattern matching

- Assign values using pattern matching
  - A = 10 → Succeeds: binds A to 10
  - B = {z, foo, 4} → Succeeds: binds B to {z, foo, 4}
  - {B, C, D} = {10, foo, bar}
    - Succeeds: binds B to 10, C to foo, D to bar
  - {A, A, B} = {abc, abc, foo}
    - Succeeds: binds A to abc, B to foo
  - {A, A, B} = {abc, def, 123} → Fails
  - [A, B, C] = [1, 2, 3]
    - Succeeds: binds A to 1, B to 2, C to 3
  - [A, B, C, D] = [1, 2, 3] → Fails

Pattern matching

- Cons cell: **[H | T]**
  - Used for pattern matching on lists
  - The pattern "[H|T] = L" extracts the head into ‘H’ and tail into ‘T’ of the list ‘L’
  - [H|T] = [1,2,3,4]
    - Succeeds: binds H = 1, T = [2,3,4]
  - [A,B|C] = [1,2,3,4,5,6,7]
    - Succeeds: binds A = 1, B = 2, C = [3,4,5,6,7]
  - [H|T] = [abc]
    - Succeeds: binds H = abc, T = []
  - [H|T] = [] → Fails

Pattern matching

- Any element of a tuple or list can be of any type, even another tuple or list
  - {A,_,[B|_],{B}} = {abc,23,[22,x],{22}}
    - Succeeds: binds A = abc, B = 22
- Pattern matching to get any element of a tuple
  - Point = {1,4,5}
  - {_,Y,_} = Point → Succeeds: binds Y = 4
  - {_,Y} = Point → Fails
- Note the use of '_', the anonymous variable, as a wildcard for pattern matching in both examples

Function calls

a) Module:Function(Arg1, …, ArgN)
   - For external functions
     - math2:double(10).
     - lists:keysearch(Name, 1, List).

b) Function(Arg1, …, ArgN)
   - For local functions or auto-imported BIFs
     - times(5, 2).
     - spawn(mod, init, []).

- Arg1 … ArgN can be any Erlang terms
- The module/function names must be atoms
  - Or expressions that evaluate to an atom

Built In Functions (BIFs)

- time()
- max(1, 2)
- length([1,2,3,4,5])
- is_tuple({a,b,c})
- size({a,b,c})
- element(2, {a,b,c})
- self()
- register(foo, Pid)
- link(Pid)
- make_ref()

- The most commonly used BIFs belonging to the erlang module are auto-imported
- They do not need to be prefixed with the module name

Function definition

```
func(Pattern11, Pattern21, ...) [when Guard1] -> ... ;
func(Pattern21, Pattern22, ...) [when Guard2] -> ... ;
...
func(PatternN1, PatternN2, ...) [when GuardN] -> ... .
```

- Clauses are scanned sequentially until one matches its patterns with the given arguments and the guard, if any, is true
- Then, all variables in the heading become bound, the clause body is executed and the value of the last expression is returned
- Variables are local to each clause

Function & module definition

```
-module(demo).
-export([double/1]).
-define(two, 2).
double(X) -> times(X, ?two).
times(X, N) -> X * N.
```

- Functions are defined within Modules
  - Module name is to be same as the file name minus the .erl extension
- -export functions so that they can be called from outside the module: only double/1 is visible
  - double/1: the function 'double' with one argument
- -define macros and '?' to use them: ?two
  - Macros are expanded during compilation
  - There are some predefined macros: e.g., ?MODULE

Function examples

```
-module(mathStuff).
-export([factorial/1, area/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

area({square, Side}) -> Side * Side;
area({circle, Radius}) -> 3.14 * Radius * Radius;
area({triangle, A, B, C}) -> S = (A + B + C)/2,
    math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) -> {invalid_object, Other}.
```

Conditionals: If

```
if
   Guard1 -> Body1;
    … ;
   GuardN -> BodyN
end.

fac(N) ->
   if
      N == 0 -> 1;
      N > 0 -> N*fac(N-1)
   end.
```

- The branches are scanned sequentially until a guard sequence that evaluates to true is found
- If there is none, a run-time error occurs
- The guard expression `true` can be used in the last branch

Conditionals: Case

```
case Expression of
   Pattern1 [when Guard1] -> … ;
    … ;
   PatternN [when GuardN] -> …
end.

sum(L) ->
   case L of
      [] -> 0;
      [H|T] -> H + sum(T)
   end.
```

- Patterns are sequentially matched against the result of the expression
- If there is no matching pattern, a run-time error occurs

Loops

- Functional programming languages usually do not offer looping constructs
- We must use recursion, notably, tail recursion
  - Used when the last expression of a function is a call to the same function
  - The stack frame of the current function is simply replaced with the one of the called function, allowing to implement loops efficiently in Erlang

```
loop(N) ->
    io:format("~w~n", [N]),
    loop(N+1).
```

Writing output to terminal

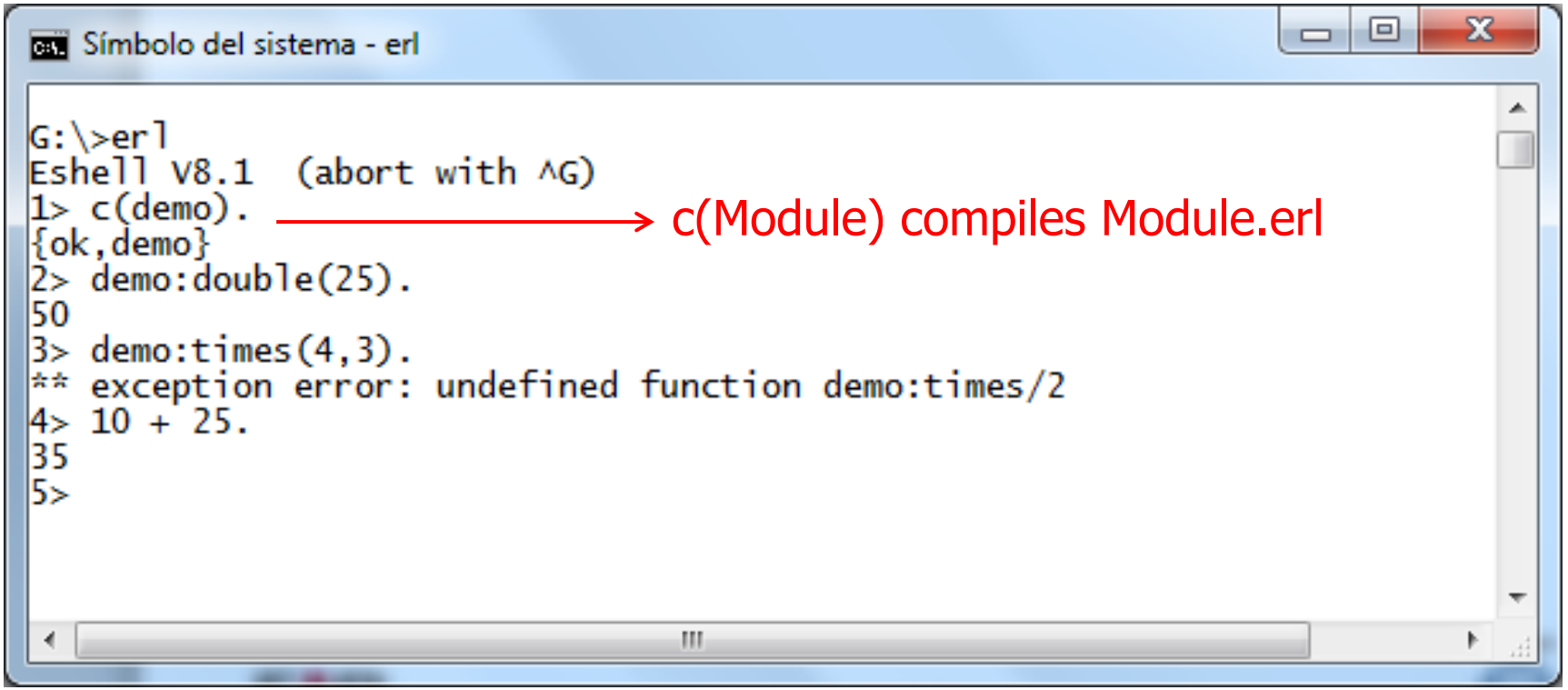
- io:format(Format, [Data]) BIF is used to write formatted data to the standard output
  - Format: string with formatting control sequences
    - ~B: integer, ~f: float (ddd.ddd), ~e: float (d.ddde+-ddd)
    - ~g: float (if >= 0.1 and < 10000.0 as ~f, else as ~e)
    - ~s: string, ~c: ASCII code
    - ~w/~p: any Erlang term (standard / pretty-printing)
    - ~n: (platform-specific) new line sequence

```
io:format("s:~s w:~w p:~p B:~B c:~c g:~g~n",
["Hi", "Hi", "Hi", 72, 72, math:pi()]).
s:Hi w:[72,105] p:"Hi" B:72 c:H g:3.14159
```

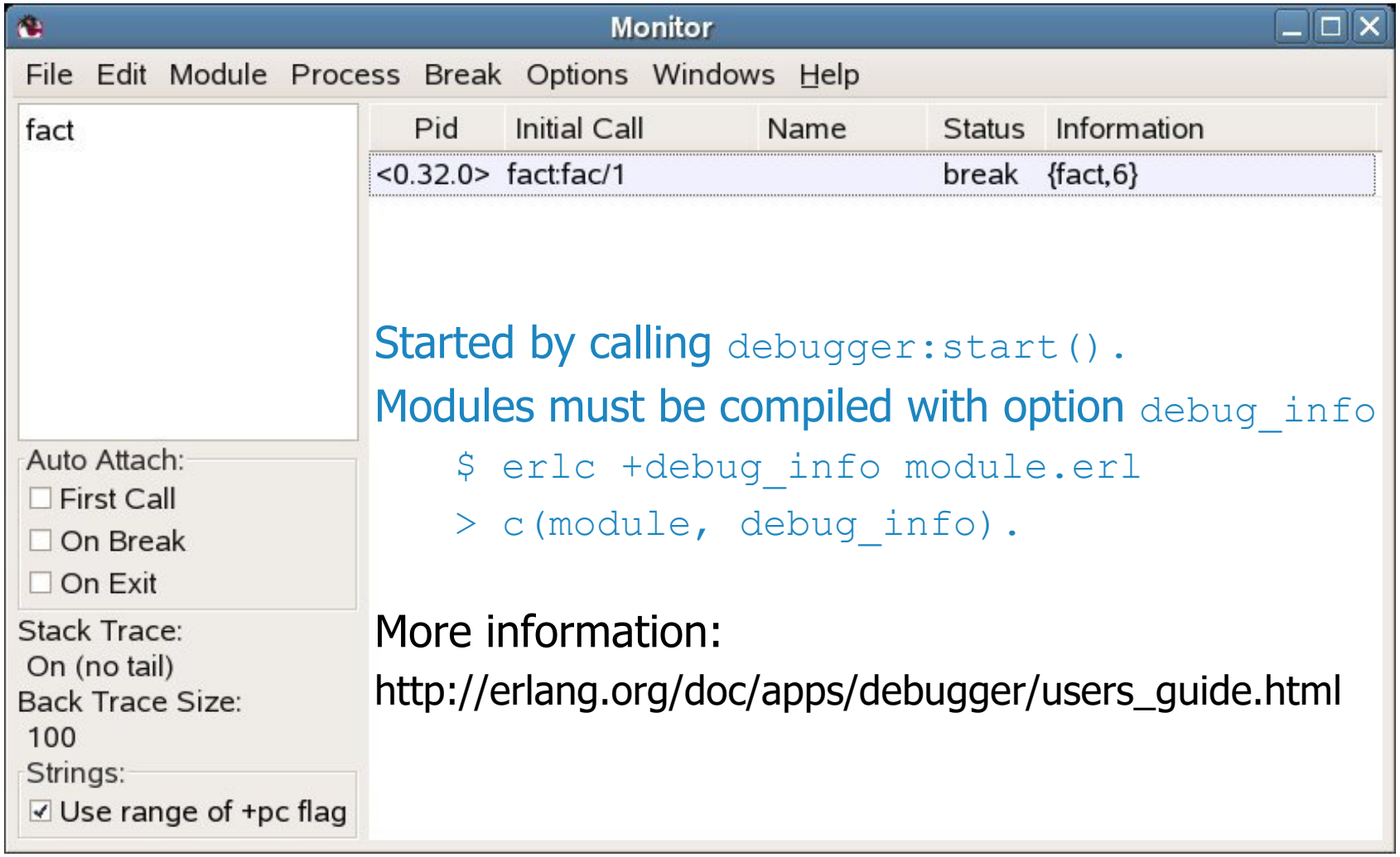
Runtime system

- The Erlang runtime system gives you an interactive shell
- In the shell you can compile and load modules and call functions
- Run it by itself, inside vim, emacs, or in a IDE such as Eclipse

Runtime system



Debugger



Started by calling `debugger:start().`

Modules must be compiled with option `debug_info`

```
$ erlc +debug_info module.erl
> c(module, debug_info).
```

More information:

http://erlang.org/doc/apps/debugger/users_guide.html

0. Erlang

Concurrent Programming

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

FIB

Process creation

- The Erlang BIF spawn is used to create and start the execution of a new process
  a) Pid = spawn(Module, Function, [Arg1, …, ArgN])
  - New process runs Module:Function(Arg1,...,ArgN)
  - Function must be exported from the module

  b) Pid = spawn(fun (Arg1, …, ArgN) -> Body end)
  - New process runs Body
- spawn returns the process identifier of the new process

Message passing: send

- To send a message you need the process identifier of the receiver

  Pid ! Message

  - Message can be any valid Erlang terms
  - The value of Message is also the return value of the expression
  - Sending a message is **asynchronous**
    - No acknowledgement
  - Sending a message to a pid never fails
    - Even if it refers to a non-existing process

Message passing: receive

- All messages sent to a process are stored in its own queue in the order they are received
  - Messages from the same sender are ordered in the same order as they were sent (a.k.a. FIFO)
- Selective receive: process can specify which messages it is willing to handle
- Implicit deferral: messages remain in the queue until explicitly handled

↑ Can decide in what order to handle messages

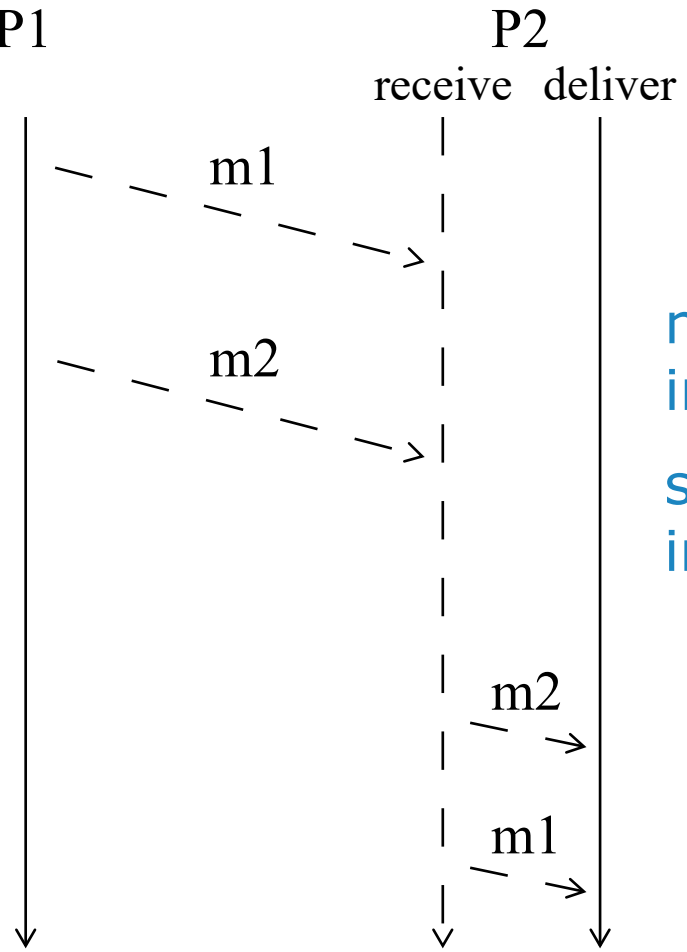
↓ Queue may fill with forgotten messages

Message passing: receive

- receive suspends a process waiting for a message that matches one of the patterns
  - receive matches the first message in the queue sequentially against the patterns
  - If no match is found, the procedure is repeated for the second message, and so on

```
receive
    Pattern1 [when Guard1] -> Actions1 ;
     … ;
    PatternN [when GuardN] -> ActionsN
end.
```

Message passing



asynchronous send,
no acknowledgment

P2 ! m1,
P2 ! m2.

messages received in FIFO order

selective receive, implicit deferral

```
receive
   m2 -> true
end,
receive
   m1 -> true
end.
```

Message passing

SENDER

```
P = spawn(wait, hello, []).
P ! "hello".
```

RECEIVER

```
-module(wait).
-export([hello/0]).
hello() ->
   receive
      X -> io:format("message received: ~s~n", [X])
   end.
```

Message passing

- Pids can be included in messages just like any data structure
  - Useful if we expect the receiver to reply

SENDER

```
Pid ! {ping, self()}.
```

RECEIVER

```
receive
   {ping, FromId} -> FromId ! pong
end.
```

Message passing

- receive can be augmented with a timeout
  - If no matching message has arrived within ExprT milliseconds, then ActionsT is evaluated instead

```
receive
   Pattern1 [when Guard1] -> Actions1 ;
    … ;
   PatternN [when GuardN] -> ActionsN
after
   ExprT -> ActionsT
end.
```

Registered processes

- register BIF allows giving names to processes

  register(Name, Pid)

  – Names can be used just as pids to send messages

  P = spawn(wait, hello, []).

  register(foo, P).

  foo ! "hello".

  – Name is automatically unregistered if the process terminates (use unregister/1 to do it manually)

  – Sending a message to a not registered name will cause a run-time error

Links

- Two processes can be linked to each other
  a) link(Pid2)
  b) Pid2 = spawn_link(Module, Func, Args)
- Links are bidirectional and there can only be one link between two processes
- If one of the participants of a link terminates, it will send an exit signal to the other participant
  - Typically, if a process dies from an unexpected throw, error, or exit, the linked process also dies

Monitors

- A process Pid1 can create a monitor for Pid2. The function returns a reference Ref
  - Ref = erlang:monitor(process, Pid2)
  - {Pid2, Ref} = spawn_monitor(Module, Func, Args)
- Monitors are unidirectional and repeated calls create several independent monitors
- If Pid2 terminates with exit reason Reason, a 'DOWN' message is sent to Pid1:
  - {'DOWN', Ref, process, Pid2, Reason}
- erlang:demonitor(Ref) BIF removes a monitor

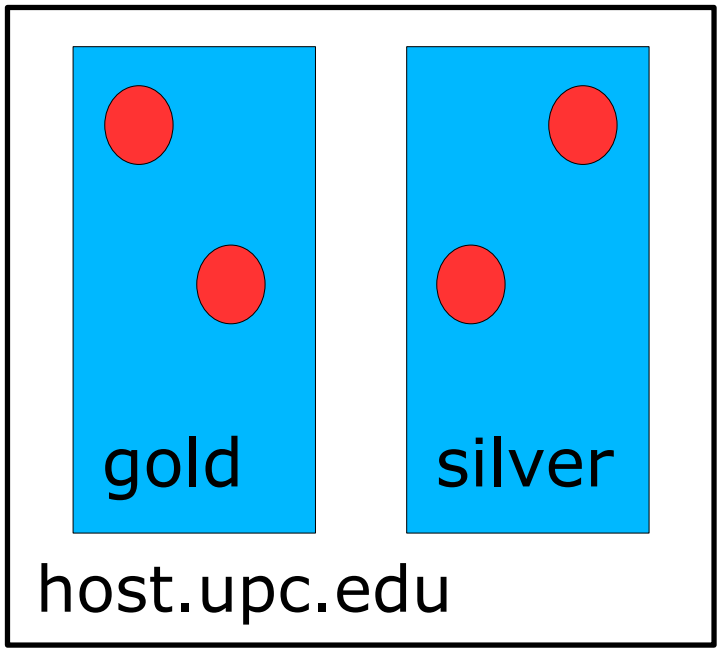
0. Erlang

Distributed Programming

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

FIB

Distributed programming



- Each Erlang instance is a different node in the distributed system

  Node 1: 'gold@host.upc.edu'

  Node 2: 'silver@host.upc.edu'

Distributed programming

- Make Erlang instances network aware
  - Set their name when starting the runtime via a command-line flag. Both long (-name) or short names (-sname) can be used (*)

  erl -name gold@host.upc.edu / erl -name gold
    - node(): gold@host.upc.edu

  erl -sname gold@host / erl -sname gold
    - node(): gold@host

  erl -name gold@127.0.0.1

  (*) A node with a long name cannot communicate with a node with a short name (and vice versa)

Distributed process creation

- Create a process in a remote Erlang node

a) P = spawn('gold@host.upc.edu', M, F, [A])

b) P = spawn('gold@host.upc.edu', fun (A) -> … end)

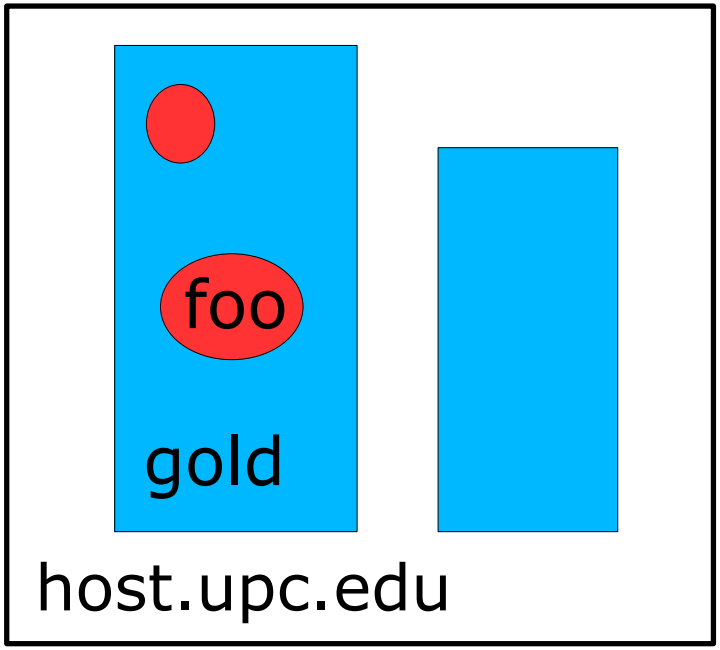
- Connections are established automatically when another Erlang node is referenced
- PIDs returned by spawn (or received in a message) can be used normally
  - Access transparent: local or remote are the same
  - Location transparent: process location is unknown

P ! Message

Registered processes

- Send a message to a process that is locally registered with a name on a remote node

  {Name, Node} ! Message



{foo, 'gold@host.upc.edu'} ! "hello".

  - Sending a message to a pid or name located at another node never fails

Registered processes

- We will use locally registered names, but Erlang offers also globally registered names

```
global:register_name(foo, Pid).
```

- Send a message to the process globally registered as foo

```
global:send(foo, "Hello").
```

- Connections must be established explicitly

```
net_kernel:connect_node(Node).
```

Authentication

- If someone connects to a node, it gets connected to all the other nodes
- Use cookies as a mechanism to differentiate clusters of nodes $\Rightarrow$ Nodes with different cookies are not able to communicate together

  -setcookie mycookie (command-line flag)

  erlang:set_cookie(node(), mycookie) (BIF)

  – Alternatively, you can have a file *.erlang.cookie* with the cookie in your home folder on all nodes

More information

- Erlang official website: https://www.erlang.org/
- 'An Erlang Primer' by Johan Montelius
  - https://people.kth.se/~johanmon/dse/crash.pdf
- 'Learn You Some Erlang for Great Good!'
  - https://learnyousomeerlang.com/
- 'Concurrent Programming in Erlang, Part I'
  - http://erlang.org/download/erlang-book-part1.pdf
- Elixir language: https://elixir-lang.org/
  - Runs on the Erlang runtime