Distributed Systems / Middleware
Distributed Programming in Erlang

Alessandro Sivieri

Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy

sivieri@elet.polimi.it

http://corsi.dei.polimi.it/distsys

Slides based on previous works by Alessandro Margara

Politecnico di Milano

Outline

- **Introduction**
- Sequential Programming
  - Data structures
  - Single assignment
  - Pattern matching
  - Functional abstractions
  - Dynamic code loading
- Concurrent / Distributed Programming
  - Processes
  - Fault tolerance
  - Distributed Erlang
  - Socket based distribution
- OTP Introduction

Politecnico di Milano

Why Erlang?

The world is parallel.

If you want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.

Use a language that was designed for writing concurrent applications, and development becomes a lot easier.

Erlang programs model how we think and interact.

Joe Armstrong

Politecnico di Milano

Erlang history

- 1982-1986. Programming experiments: how to program a telephone exchange
- 1986. Erlang emerges as a dialect of Prolog. Implementation is a Prolog interpreter
  - 1 developer (Joe Armstrong)
- 1986. Own abstract machine, JAM
  - 3 developers, 10 users
- 1993. Turbo Erlang (BEAM compiler)
- 1993. Distributed Erlang
- 1996. OTP (Open Telecom Platform) formed
- 1996. AXD301 switch announced
  - Over a million lines of Erlang
  - Reliability of nine 9s

Erlang history/2

Politecnico di Milano

- 1998. Erlang banned within Ericsson for other products
- 1998. Erlang "fathers" quit Ericsson
- 1998. Open source Erlang
- 2004. Armstrong re-hired by Ericsson
- 2006. Native symmetric multiprocessing is added to runtime system
- 2011. December. Latest stable release: R15B

Politecnico di Milano

Getting started

- To run Erlang programs we will use the BEAM emulator
- Similar to Java JVM
  - Programs are compiled in BEAM ByteCode …
  - … and then executed inside the emulator
- Similar to Python
  - It offers an interactive shell …
  - ... that we will use to run our examples
- To start the BEAM compiler type the command **erl**

Politecnico
di Milano

Installing Erlang

- Windows
  - Binary installation of the latest version are available at http://www.erlang.org/download.html
- Linux (Debian-based systems)
  - apt-get install erlang
- Linux / Mac OS X
  - Build from sources
  - Download latest available version (R15B) at http://www.erlang.org/download.html
  - Compile and install

Politecnico di Milano

What is Erlang?

- Erlang is a ***functional*** and ***concurrent*** programming language
- Why functional?
  - Computation is performed by means of mathematical function evaluation
    - Often recursive
  - Functions are first-class values
    - Can be used as parameters to define higher order abstractions
- Why concurrent?
  - Asynchronous message passing
    - Message passing = No shared memory
      - No side effects
      - No locks
    - Asynchronous = No synchronous invocations
      - Isolation between processes
      - Fault-tolerance
  - Efficient concurrency management
    - Lightweight processes and efficient communication

Our approach

Politecnico di Milano

- Few slides on the syntax
- Many examples
  - Available online as source code
- Focusing on the following aspects
  - Features and abstractions offered by functional programming languages
  - Concurrent / distributed programming
- We will use only base Erlang
  - We will mention some abstractions built inside existing libraries as examples of functional programming power

Outline

- Introduction
- **Sequential Programming**
  - **Data structures**
  - **Single assignment**
  - **Pattern matching**
  - **Functional abstractions**
  - **Dynamic code loading**
- Concurrent / Distributed Programming
  - Processes
  - Fault tolerance
  - Distributed Erlang
  - Socket based distribution
- OTP Introduction

Politecnico di Milano

C quicksort

```c
void QuickSort(int list[], int beg, int end)
{
    int piv; int tmp;
    int  l,r,p;

    while (beg < end)
    {
        l = beg; p = (beg + end) / 2; r = end;
        piv = list[p];
        while (1)
        {
            while ((l <= r) && ((list[l] - piv) <= 0 )) l++;
            while ((l <= r) && ((list[r] - piv) > 0 )) r--;
            if (l > r) break;
            tmp = list[l]; list[l] = list[r]; list[r] = tmp;
            if (p==r) p=l;
            l++; r--;
        }
        list[p] = list[r]; list[r] = piv;
        r--;
        if ((r - beg) < (end - l))
        {
            QuickSort(list, beg, r);
            beg = l;
        }
        else
        {
            QuickSort(list, l, end);
            end = r;
        }
    }
}
```

Erlang quicksort

Politecnico di Milano

```
qsort([]) -> [];
qsort([Pivot|Rest]) ->
    qsort([ X || X <- Rest, X < Pivot]) ++ [Pivot] ++ qsort([ Y || Y <- Rest, Y >= Pivot]).
```

Termination characters syntax…

- … or: the main error source in your/our first Erlang listings
- Four possible termination characters:
  - '.' is used for single lines in the shell or for the last line of a function
  - ',' is used for each intermediate line in a function
  - ';' is used for terminating a code block inside ***case/if/receive/try/catch*** (more on this later)
  - '' (no termination character) is used for terminating the last code block inside ***case/if/receive/try/catch***
- Thank Prolog for all this mess!

Politecnico di Milano

Variables

- Variables must start with a capital letter
- Variables are untyped
  - A = 123456789.
  - B = “erlang”.
  - C = 123456.12 * 654321.345.
- Variables don’t vary!!!
  - Single assignment

Single assigment

- A variable that has had a value assigned to it is called a ***bound*** variable …
- … otherwise it is called an ***unbound*** variable
- All variables start off unbound
- When Erlang sees a statement such as X = 1234, it binds variable X to the value 1234
- Before getting bound, X could take any value
- Once it gets a value, it holds on to it forever

Politecnico di Milano

Single assignment

- Single assignment is like algebra
  - If you use a variable X on different parts of an equation, it always keeps the same value
  - Not like in imperative programming languages where statements like X = X + 1 are allowed
    - In Erlang X = X + 1 is an error
- Why single assignment is good?
  - Only one possible value inside a given scope
    - Increases readability
      - X will always represent the same value
    - Prevents from modifications in global state
      - Global variables cannot be modified by functions
        - Forces better design choices
      - Isolation of different functions
        - Fault-tolerance
        - Hot-swap

Atoms

Politecnico di Milano

- Atoms are used to represent different non-numerical constant values
- Atoms start with a lower case letter
- Atoms are global, and this is achieved without the use of macro definition or include files
- The value of an atom is just the atom
- If you want to write a calendar application the atoms "monday", "tuesday" etc. will have the same value everywhere
- Sometimes, you may need ' (ticks) for specifying atom names with strange characters
- Remember that also module, function, host names (and many other types) are actually atoms

Politecnico di Milano

Tuples

- A tuple is a structure composed by a fixed number of unnamed fields
- For example X = {temp, 12}
  - creates a tuple with two fields
  - the first field is the atom “temp”
  - the second field is the integer value 12
  - the tuple is assigned to variable X

Pattern matching

- Pattern matching is a central concept in Erlang
- The pattern matching operator is =
- It evaluates the right side and than matches the result against the pattern on the left side
- We have already seen an example of pattern matching
  - Variable assignment
  - X = 10
  - Erlang says to itself "What can I do to make this statement true?"
  - In this case it binds the value 10 to the variable X, so that the equation is satisfied

Politecnico di Milano

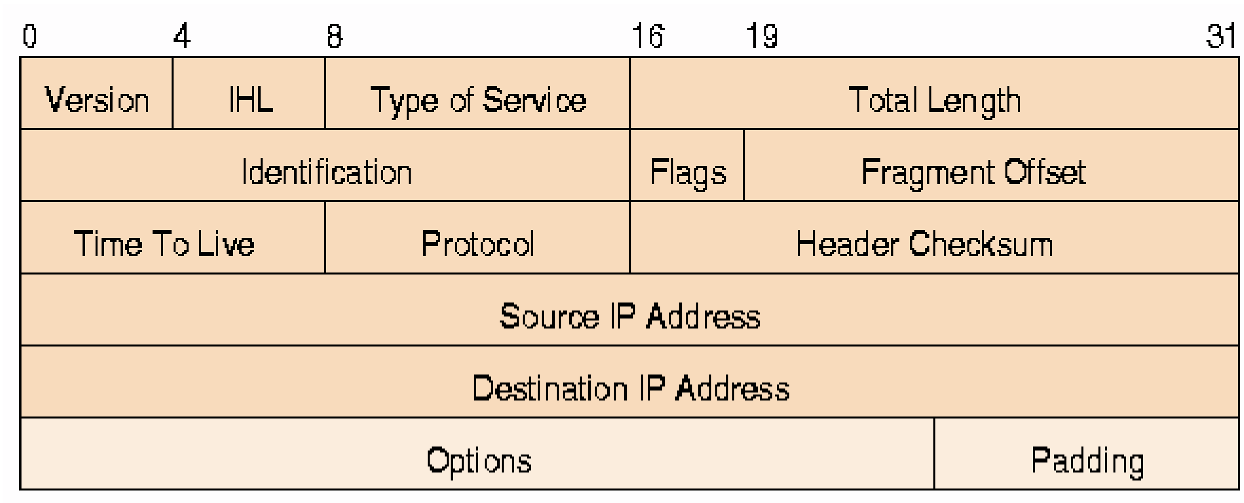
Pattern matching

- Pattern matching works with tuples …
- … enabling programmers to extract values
- Point = {point, 10, 12}
- {point, X, Y} = Point
  - Assigns 10 to variable X and 12 to variable Y
- {point, Z, Z} = Point
  - Returns an error: it is not possible to make the statement true
- { _ , _ , W} = Point
  - Assigns 12 to variable W
  - _ is the anonymous variable: different occurrences don't have to hold the same value

Politecnico di Milano

Bit syntax

- Erlang has a very interesting bit syntax
  - Header = <<IpVersion:4, HLen:4, SrvcType:8, TotLen:16, ID:16, Flgs:3, FragOff:13, TTL:8, Proto:8, HdrChkSum:16, SrcIP:32, DestIP:32, RestDgram/binary>>



Politecnico di Milano

Pattern matching on bits

- You can specify:
  - The type of the variable (integer, float, binary)
  - The signedness
  - The endianess
  - The size (from 1 to 256 bits)
- You can match some pattern in ***case/receive***…
- Remember to add a final field ready to get the rest of the variable, especially if working on network protocols
  - the payload has (probably) no fixed size

Lists

Politecnico di Milano

- Lists are used to store multiple things
  - Es. ToBuy = [{apple, 10}, {pear, 12}, {lemon, 3}]
- Lists can have heterogeneous elements
  - Es. [erlang, 10, {lemon, 3}]
- The first element of a list is called ***Head***
- If you remove the Head from the list what's left is called the ***Tail*** of the list
- If T is a list than also [H|T] is a list
  - | separates the Head from the Tail
- [ ] is the empty list

Lists

- Pattern matching can be applied to lists as well
- Buy = [{apple, 10}, {pear, 12}, {orange, 4}, {lemon, 6}]
- [AppleTuple, PearTuple | Others] = Buy
  - Assigns {apple, 10} to AppleTuple
  - Assigns {pear, 12} to PearTuple
  - Assigns [{orange, 4}, {lemon, 6}] to Others
- NewBuy = [{milk, 2} | Others]
  - Assigns [{milk, 2}, {orange, 4}, {lemon, 6}] to NewBuy

List comprehensions

- Creating lists from existing lists
- Standard construct in functional languages
- [Element | | Element <- List, *conditions*]
- Example:

```
> NewList = [X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].
[4,5,6]
```

Politecnico di Milano

Strings

- Strictly speaking there are no strings in Erlang
- Strings are really just lists of integer
- Strings are enclosed in double quotation marks
  - For example you can write S = "Hello"
  - "Hello" is just a shorthand for the list of integers representing the individual characters in the string
- The shell prints a list of integers as a string if and only if all integers in the list represent a printable character

Politecnico di Milano

Dictionaries

- A dictionary performs exactly as a list of 2-dimension tuples
- [{Key1, Value1}, {Key2, Value2}, {Key3, Value3}]
- Becomes:
  - dict:new()
  - dict:append(NewKey, NewValue, Dict)
  - Value = dict:fetch(Key, Dict)
  - …
- You can move from lists to dictionaries and back easily (dict:from_list, dict:to_list)

Politecnico di Milano

Modules

- Erlang programs are splitted in modules
- Each module is a ".erl" file
- To compile a module m.erl you can either
  - Call erlc m.erl from outside the BEAM emulator
  - Or call c(m) from within the BEAM emulator
- This creates a m.beam file, containing the bytecode of the module

Modules

- Each module consists of a set of functions
  - Used internally
  - Or externally visible
- Each module starts with
  - -module(module-name).
    - module-name must be the name of the file
  - -export([fun1/arity1, fun2/arity2, … fun-n/arity-n])
    - Where fun1, fun2, … fun-n are the names of the functions that have to become visible outside the module
    - And arity1, arity2, … arity-n are the arity (i.e. number of input parameters) required by each function

Politecnico di Milano

Functions

- A function is univocally identified by a name and an arity
- Each function consists of an ordered list of clauses
- Each clause has a pattern and a piece of code
- During a function call clauses are evaluated in order
- When the pattern of a clause is matched, then the associated code is evaluated and the function returns
- If no single pattern can be matched, then an error is generated

Functions

Politecnico di Milano

- Example: we define a module geometry with only one (exported) function of arity 1, which computes the area of different figures
- We can use it within the BEAM emulator to compute the area of a square
  - geometry:area({square, 10})

```erlang
-module(geometry).
-export([area/1]).

area({square, X}) ->
    X*X;
area({rectangle, X, Y}) ->
    X*Y;
area({circle, R}) ->
    3.14*R.
```

Functions

- Sometimes it is useful to check constraints on input values
- For this reason Erlang introduces guards
- Introduced after a pattern, using the ***when*** keyword
- The example shows a ***single guard*** (X > Y)
- It is possible to combine single guards using logical ***and*** (,) or logical ***or*** (;)
- Beware that if you want short-circuit expressions, you have to use ***andalso*** or ***orelse***

```
-module(guards).
-export(max/2).

max(X, Y) when X > Y ->
    X;
max(X, Y) -> Y.
```

Case and if expressions

- Pattern matching and guards can be used to define conditional blocks
  - *Case* expressions
  - *If* expressions
- Expressions are evaluated in order
- If no match is found an error is generated
- Beware that in ***If*** expressions you always need the "else" part

```
case Expression of
    Pattern1 [when Guard1] -> Expr_seq1;
    Pattern2 [when Guard2] -> Expr_seq2;
    ...
    Any -> io:format("Unknown sequence: ~p~n", [Any])
end
```

```
if
    Guard1 ->
        Expr_seq1;
    Guard2 ->
        Expr_seq2;
    ...
    true ->
        Default_seq
end
```

Erlang: a functional PL

Politecnico di Milano

- In just a few slides we have already seen all the building blocks of Erlang
  - We can now write every sequential program
  - Without ***while***, ***for*** statements!
- Erlang is a functional programming language
  - Everything is performed through function evaluation
  - Functions are values
    - It is possible to assign functions to variables …
    - … to use functions as parameters for other functions …
    - … and to return function as result of other functions

Erlang: a functional PL

Politecnico di Milano

- There are no loop statements
- Iteration is performed using recursive functions
- Example: we want to sum all the elements of a list of integers
- We recursively sum the head to the rest of the list until we arrive to the empty list

```
-module(listSum).
-export([sum/1]).

sum([]) ->
    0;
sum([H|T]) ->
    H+sum(T).
```

Saving space: tail recursion

Politecnico di Milano

- In the previous code H+sum(T) cannot be evaluated until the function sum(T) returns
  - Every function call requires stack space
  - The function sum(X) evaluates in O(length(X)) space
- We can implement the same function to evaluate in constant space
  - Using an accumulator
  - Using tail recursion (= the last thing a function does is calling itself)
  - Same cost as in imperative programming loops
- Every recursive function can be transformed in a tail recursive function
  - It is good practice to use tail recursion

```erlang
-module(tail).
-export([tailSum/1]).

tailSum(X) ->
   tailSum(X, 0).

tailSum([H|T], Acc) ->
   tailSum(T, Acc+H);
tailSum([], Acc) ->
   Acc.
```

Higher order function

- A common task is the execution of the same transformation on all the elements of a list
- We can write a single function for each possible transformation
- Or we can use the possibility to use functions as values
  - Map executes a “generic” task on all the elements of a list
  - It is said to be a higher order function

```
-module(map).
-export([map/2,double/1]).

double(N) -> N*2.

map([H|T], F) -> [F(H)|map(T, F)];
map([], _) -> [].

>c(map).
>D = fun(X) -> map:double(X) end.
>A = [1,2,3].

>map:map(A, D).
>[2,4,6]
```

Compile

Assign a function to a variable

Use the function as parameter

Functions that return functions

Politecnico di Milano

- Not only can functions be used as arguments to functions …
- … but functions can also **return** functions
  - It is not used that often, at least wrt to the previous mode
- Suppose we have a list of something (es. Fruit)
- Fruit = [apple, pear, orange]
- We can define a function Test that returns a function that checks whether an element is in a list
  - Test = fun(L) -> (fun(X) -> lists:member(X, L) end) end.
  - lists:member is a function that returns true if X is in L
- We can now create a function IsFruit
  - IsFruit = Test(Fruit)
  - IsFruit(apple) will return true
  - IsFruit(cat) will return false

Programming abstractions

- Using higher order functions enables programmers to create different levels of abstractions
- This is conceptually similar to the creation of object hierarchies in Object Oriented Languages like Java or C#
- Object Oriented Languages simplify reuse of code by defining abstract members
- Functional Languages use function parameterization:
  - Functions are values
  - Functions can be used as parameters

Data manipulation

- Functional languages allow you to write extremely compact code when manipulating data
  - map(): applies a given function to all members of a list
  - fold(): applies a given function to all members of a list, passing an accumulator for getting a result of that function
  - filter(): filters a list according to a given function
  - zip(): puts together two lists in a single list of 2-dimension tuples
  - …
- The same holds for dictionaries

Data manipulation

```
List = [1, 2, 3, 4, 5, 6, 7],
lists:map(fun(Element) -> 2 * Element end, List),
Sum = lists:foldl(fun(Element, AccIn) -> Element + AccIn, 0, List),
EvenList = lists:filter(fun(Element) when Element rem 2 == 0 -> true;
                           (Element) -> false end, List),
Days = [monday, tuesday, wednesday, thursday, friday, saturday, sunday],
DayWithNumber = lists:zip(Days, List).
```

Exception handling

- In Erlang exceptions are raised automatically when the system encounters an error
  - Pattern matching errors
  - Function call with incorrectly typed arguments
- It is also possible to throw exceptions explicitly
  - throw(Why) throws an exception that the caller is expected to handle
  - erlang:error(Why) is used to denote "crashing errors"; something that the caller is not supposed to manage
  - exit(Why) explicitly stops a process; if the exception is not managed a message is broadcast to all linked processes (more on this later)

Exception handling

- Exception handling is very similar to a case expression
- ExceptionType is an atom, which defines the kind of exception (throw, exit, error) one wants to catch

```
try FuncOrExpressionSequence of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
catch
    ExceptionType: ExPattern1 [when ExGuard1] -> ExExpressions1;
    ExceptionType: ExPattern2 [when ExGuard2] -> ExExpressions2;
    ...
after
    AfterExpressions
end
```

Result evaluated here if no exception occurred

Exceptions (if any) Evaluated here

Politecnico di Milano

Some examples

- Write a function that returns the maximum element in a list

```
-module(max).
-export([max/1]).

max([Head|Tail]) ->
    max(Tail, Head).

max([], Max) ->
    Max;
max([Head|Tail], Max) when Head > Max ->
    max(Tail, Head);
max([_|Tail], Max) ->
    max(Tail, Max).
```

Works only with non-empty lists!

Politecnico di Milano

Some examples

- Write a function that reverses the order of a list

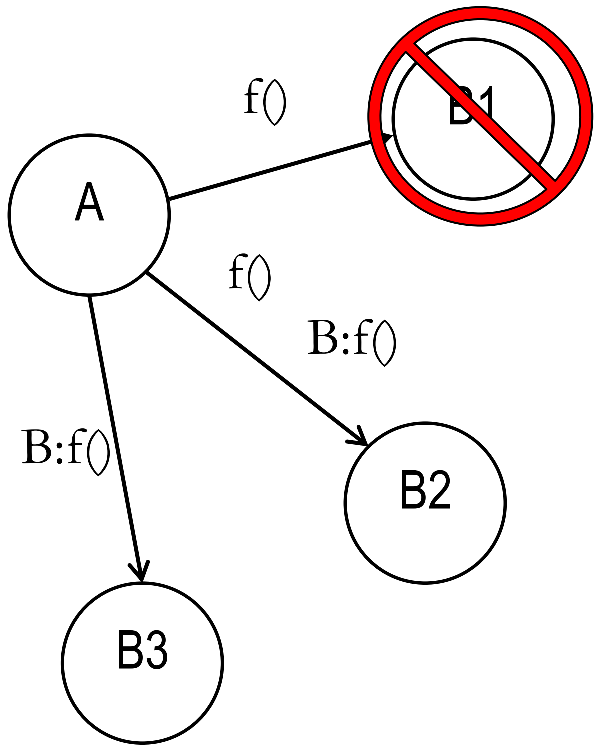
```
-module(reverse).
-export([reverse/1]).

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

reverse([Head | Rest], ReversedList) ->
    reverse(Rest, [Head|ReversedList]);
reverse([], ReversedList) ->
    ReversedList.
```

Dynamic code loading

- Dynamic code loading is a feature directly built inside Erlang
- A module can access a function of another modules in two ways:
  - Importing modules
    - -import(module_name)
  - Using fully qualified names
    - module_name:function_name
- The former continues to adopt the previously loaded version of a module
- The latter ensures that the latest version of the module is used
  - Even if the module has been recompiled

Dynamic code loading

- Two possible versions of a module can exist at the same time
- No more than two versions are allowed
- If a third version is created:
  - B1 is removed
    - Existing computation aborted
  - B2 continues to exist
  - B3 is the new current version



Dynamic code loading

- Dynamic code loading is a low level feature
- It enables programmers to change system code at runtime
  - To fix bugs
  - To include new functionalities
  - To improve performance
- Higher level abstractions have been designed on top of it
  - OTP (Open Telecom Platform) offers
    - Implementation of design patterns that simplify error-free code loading
    - Tools to automatize installation of new software versione involving multiple modules upgrades

Dynamic code loading: example

```
-module(dynCode1).
-export([start/0]).

start() ->
    spawn(fun loop/0).

loop() ->
    Val = dynCode2:val(),
    io:fwrite("Val = ~p~n", [Val]),
    sleep(2000),
    loop().

sleep(Time) ->
    receive
        after Time -> ok
    end.
```

```
-module(dynCode2).
-export([val/0]).

val() ->
    1.
```

If we change value and compile, dynCode1 will print the new value

Prints the value computed by dynCode2 every 2 seconds

Outline

Politecnico di Milano

- Introduction
- Sequential Programming
  - Data structures
  - Single assignment
  - Pattern matching
  - Functional abstractions
  - Dynamic code loading
- **Concurrent / Distributed Programming**
  - **Processes**
  - **Fault tolerance**
  - **Distributed Erlang**
  - **Socket based distribution**
- OTP Introduction

Politecnico di Milano

Concurrent programming

- Erlang is sometimes described as a ***concurrency oriented*** programming language …
- This does not only mean that writing concurrent programs is
  - Possible
  - Easy
  - Efficient
- This refers to the possibility to take into account concurrency when designing complex systems
- Erlang forces programmers to think about processes as independent actors …
- … communicating only through message passing

Politecnico di Milano

Concurrent programming

- Cuncurrent programming requires just three new primitives:
  - ***Pid = spawn(Fun)***. Creates a new concurrent process that evaluates Fun. The new process runs in parallel with the caller. Spawn returns a ***Pid*** (process identifier) which can be used to send messages to the process
  - ***Pid ! Message.*** Sends ***Message*** to the process with identifier ***Pid.*** Message sending is asyncronous: the sender has not to wait, but can continue its own task
  - ***Receive … end.*** Used to receive messages: messages are evaluated using pattern matching. Messages are stored in a sort of mailbox (persistent!) until the received function is called

Simple Server

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

loop() -> receive
        {Sender, Fun, Num} ->
            Sender ! Fun(Num),
            loop()
    end.

start() -> spawn(fun loop/0).
```

The server waits for messages containing the sender Pid, a function and a number

Sends back Fun(Num) and waits again

```
>Dup = fun(X) -> 2*X end.
>c(server).
>Pid = server:start().
>Pid ! {self(), Dup, 256}.
>receive A -> A end.
512
```

In the shell we start the server

We send a request

We receive the response

Clock

- In this example we define a process that executes a function ***Fun*** periodically (period = ***Time***)
- We introduce two new keywords
  - ***After:*** defines what to do if no matching message is received after ***Time*** elapsed
  - ***Register:*** associate the Pid of a process to an atom

```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) ->
    register(clock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> clock ! stop.

tick(Time, Fun) ->
    receive
        stop ->
            void
    after Time ->
            Fun(),
            tick(Time, Fun)
    end.
```

Variable

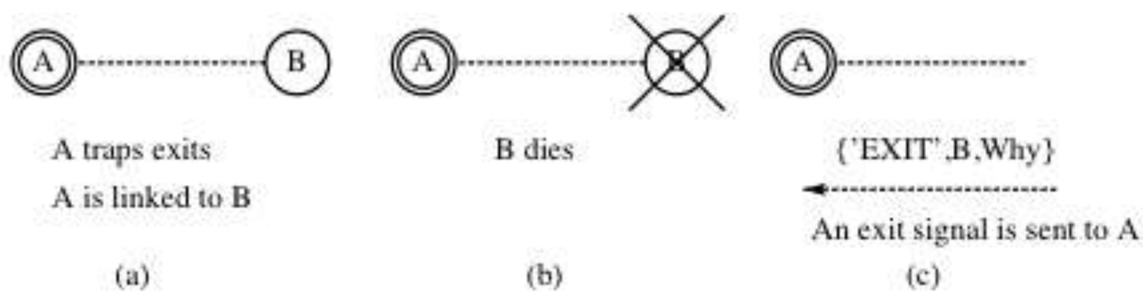
- Write a program that simulates a simple integer variable (allowing init, add, sub and get operations)

```erlang
-module(var).
-export([init/0, add/1, sub/1, get/0]).

init() ->
    Pid = spawn(fun() -> loop(0) end),
    register(var, Pid).

loop(N) ->
    receive
        {add, X} -> loop(N+X);
        {sub, X} -> loop(N-X);
        {Pid, get} -> Pid ! N,
                      loop(N)
    end.
```

```erlang
add(X) -> var ! {add, X}.

sub(X) -> var ! {sub, X}.

get() -> var ! {self(), get},
    receive Result -> Result end.
```

Fault tolerance

- A key feature of Erlang is its ability to simplify the design of fault tolerant programs
- This is achieved through process linking
  - A process P can link to process Q by calling the ***link***(Q) function
  - When one process dies, an ***Exit*** signal is sent to every linked process



Fault tolerance

- What happens when a process receives an exit signal?
  - If the receiver hasn't declare itself as ***system process,*** the message will cause it too to exit
  - Otherwise the message will be processed as a normal one
  - process_flag(trap_exit, true) turns a process into a system process

Function to process errors

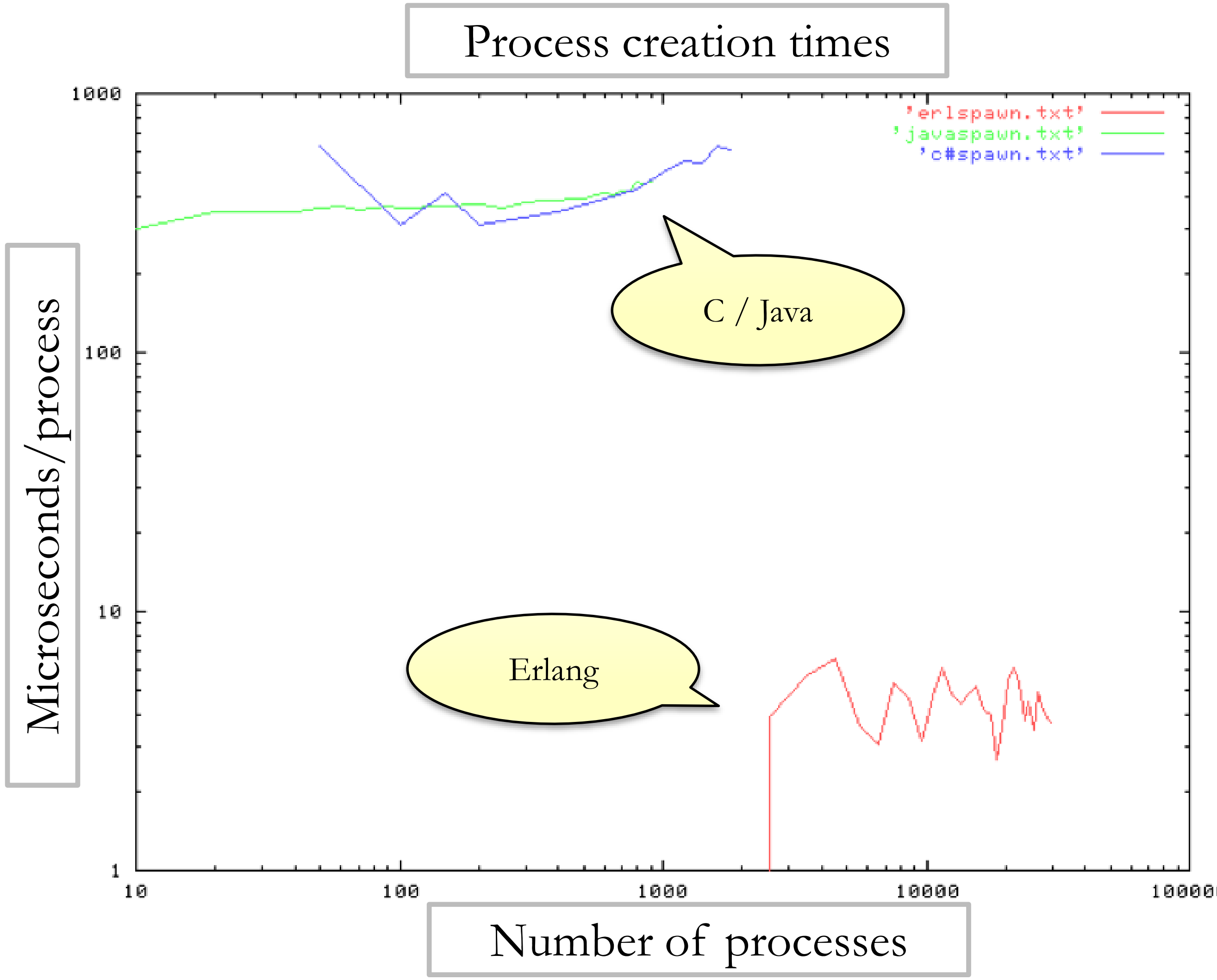
Uses an ad-hoc process

Creates a link to the given process

Becomes a system process

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

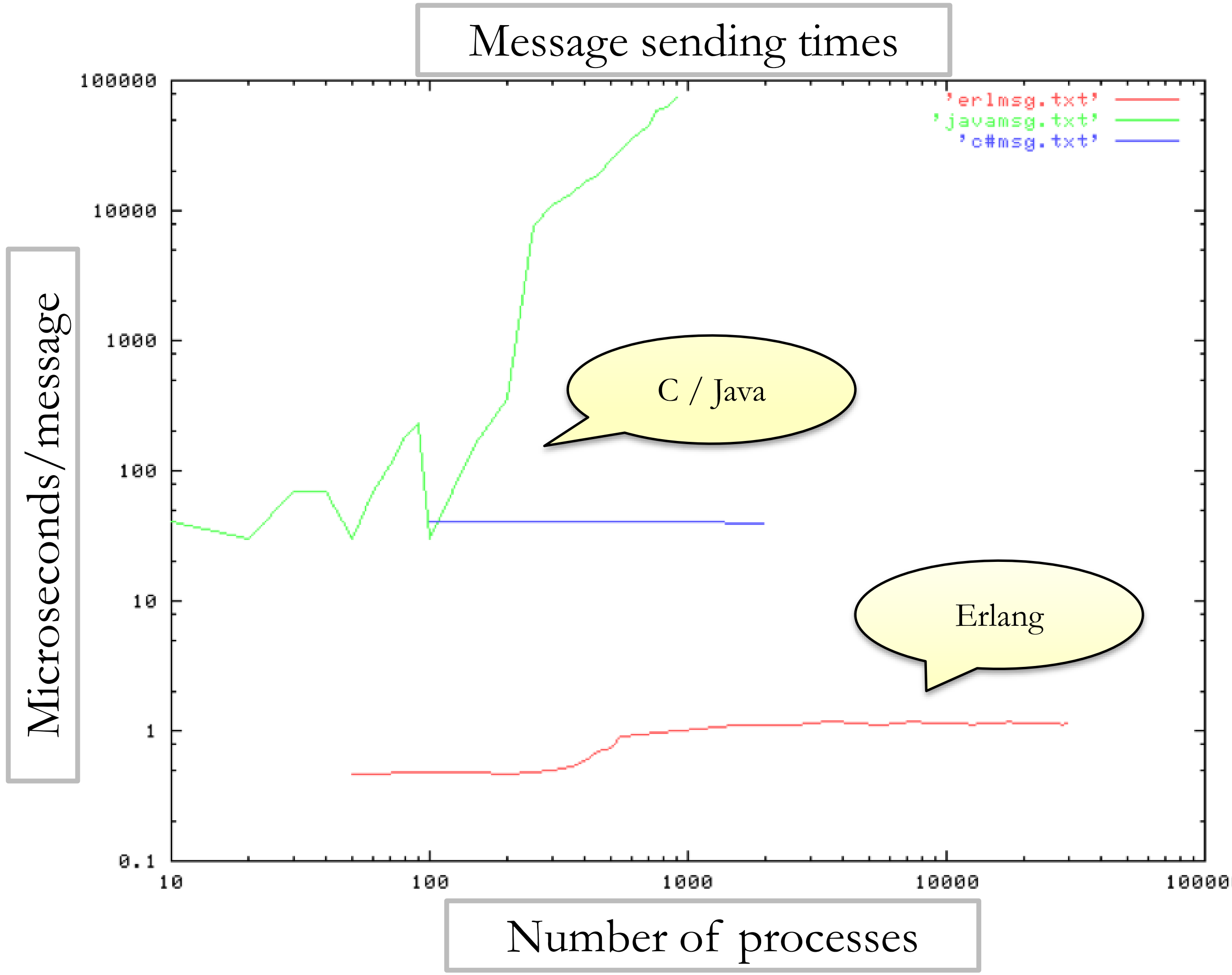
Concurrency: performance



Source: J.Armstrong "Concurrency oriented programming in Erlang"

Concurrency: performance

Politecnico di Milano



Source: J.Armstrong "Concurrency oriented programming in Erlang"

Distributed programming

- Erlang enables the programmers to distribute concurrent processes on different machines
- We will talk about two main distribution models:
  - ***Distributed Erlang***: provides a method for programming applications that run on a single administrative domain (trusted environment, like a LAN)
    - Processes run in Erlang ***nodes***
    - All mechanisms for message passing, error hadling etc. works as in the single node scenario
  - ***Socket-based distribution:*** uses TCP/IP sockets to send messages in an untrusted environment
    - Less powerful but more secure model
    - Used to easily interact with programs written in other languages

Distributed Erlang

- Distributed Erlang enables programmers to spread processes on different ***nodes.***
- A node ***a*** can communicate with a node ***b*** if
  - It knows ***b***'s name
  - ***a*** and ***b*** share the same ***cookie***
- To start a node with a given name and cookie run
  - erl –sname name –setcookie cookie (same host)
  - erl –name name@host –setcookie cookie (across hosts)
- We will see how distribution works with two examples:
  - Sending messages to remote nodes
    - Implementing the Remote Procedure Call (RPC) pattern
  - Spawning processes on remote nodes

Distributed Erlang

Politecnico di Milano

- Set up your environment
  - Start Erlang with the -name option
  - Ensure that both nodes have the same cookie
    - Start the node with –setcookie cookie
  - Make sure that the hostnames are resolvable
    - By DNS or
    - Adding an entry to /etc/hosts
  - Make sure that all systems have the same version of the code
    - Manually copy the bytecode or
    - Use the shell command nl(Mod)
      - It loads the module Mod on all connected nodes
      - Useful for dynamic code loading
  - Test if everything is working using the ping function
    - net_adm:ping(node_name)

Fault tolerant server

Politecnico di Milano

- Two processes
  - Server
  - Error handler
- If the server crashes, the error handler
  - Traps the error message
  - Starts a new server
  - Terminates
- Every time a new server starts it creates a new error handler

```
-module(dist).
-export([start/0, ask/3]).

start() ->
    Pid = spawn(fun loop/0),
    register(server, Pid),
    io:fwrite("Server started\n"),
    on_exit(Pid, fun start/0),
    io:fwrite("Error handler started\n").

loop() -> receive
              {Sender, Fun, Num} ->
                  Sender ! Fun(Num),
                  loop()
          end.

on_exit(Pid, Fun) ->
    spawn(fun() ->
              process_flag(trap_exit, true),
              link(Pid),
              receive
                  {'EXIT', Pid, _} ->
                      io:fwrite("Error trapped\n"),
                      Fun()
              end
end).
```

Fault tolerant server

Politecnico di Milano

- The client can call the dist:ask function to send a message to the server
- If the server does not respond in 100ms, then it returns 'Server crashed'
- Suppose our client wants to execute the function support:dup
  - The module has not been loaded by the server

Sends the message to the process registered as 'server' on the 'Host' machine

```
ask(Host, Fun, Num) ->
    {server, Host} ! {self(), Fun, Num},
    receive
        Reply -> Reply
    after 100 -> 'Server crashed'
    end.
```

```
-module(support).
-export([dup/1]).

dup(X) -> 2*X.
```

Politecnico
di Milano

Fault tolerant server

Client's shell

```
>net_adm:ping(server@serverhost).
pong
>nl(support).
abcast
>dist:ask('server@serverhost', fun support:dup/1, 16).
32
>dist:ask('server@serverhost', fun support:dup/1, aaa).
'Server crashed'
>dist:ask('server@serverhost', fun support:dup/1, 24).
48
```

Check the server

Load the module on the server

Call 2 contains an error.
The server crashes!

The server is up
and working again!

Fault tolerant server

Server's shell

```
>dist:start().
Server started
Error handler started
ok

Error trapped
Server started
Error handler started
```

Starts the server

When request 2 is processed, the wrong message format crashes the server

The error is trapped and the server (together with a new error handler) is started

Distributed Erlang

Politecnico di Milano

- Distributed Erlang also enables programmers to spawn processes on a specific node
- Without modifying our previous code we can start the server from the client shell

```
>net_adm:ping(server@serverhost).
pong
>nl(dist).
abcast
>nl(support).
abcast
>spawn(server@serverhost, dist, start, []).
>dist:ask('server@serverhost', fun support:dup/1, 16).
32
```

Code mobility

- The nl instruction loads a module on a remote machine
  - This realizes code mobility
- It is also possible to move locally defined functions, like in the example below
- This works <u>only</u> on the same host, otherwise the two hosts have to share the bytecode!

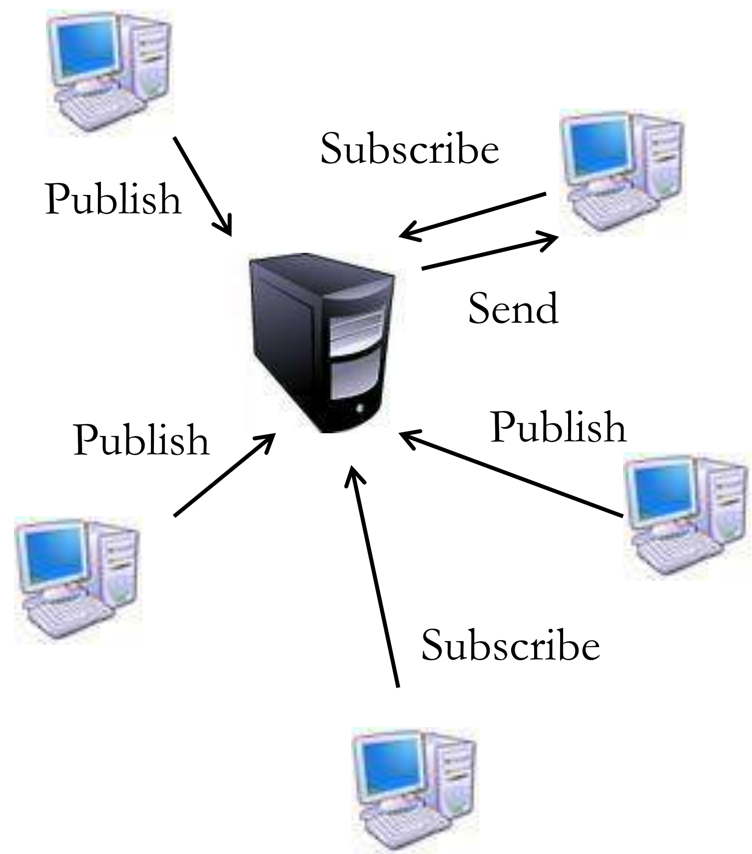
```
>Square = fun(X) -> X*X end.
>dist:ask('server@serverhost', Square, 10).
100
>dist:ask('server@serverhost', Square, aaa).
'Server crashed'
>dist:ask('server@serverhost', Square, 12).
144
```

Defines a local function

Uses the function to call the server

A complete example

- Topic based publish-subscribe
  - The dispatcher receives subscriptions and publications from clients
  - It sends published messages to interested (subscribed) clients
  - Similar to topic based communication in JMS



Exported functions

Politecnico di Milano

- We define a module that exposes the following functions:
  - startDispatcher: starts the dispatcher of messages, that waits for publish/subscribe commands
  - startClient: starts a process on the client that waits for messages
  - publish: publishes a message for a given topic
  - subscribe: used by clients to express their interests

```erlang
-module(pubsub).
-export([startDispatcher/0, startClient/0,
         subscribe/2, publish/3]).

startClient() ->
    Pid = spawn(fun clientLoop/0),
    register(client, Pid).

clientLoop() ->
    receive {Topic, Message} ->
            io:fwrite("Received message ~w for topic ~w~n",
                      [Message, Topic]),
            clientLoop()
    end.

subscribe(Host, Topic) ->
    {dispatcher, Host} ! {subscribe, node(), Topic}.

publish(Host, Topic, Message) ->
    {dispatcher, Host} ! {publish, Topic, Message}.

startDispatcher() ->
    Pid = spawn(fun dispatcherLoop/0),
    register(dispatcher, Pid).
```

Politecnico di Milano

The dispatcher

```erlang
dispatcherLoop() ->
   io:fwrite("Dispatcher started\n"),
   dispatcherLoop([]).
dispatcherLoop(Interests) ->
   receive
      {subscribe, Client, Topic} ->
         dispatcherLoop(addInterest(Interests, Client, Topic));
      {publish, Topic, Message} ->
         Destinations = computeDestinations(Topic, Interests),
         send(Topic, Message, Destinations),
         dispatcherLoop(Interests)
   end.
```

- The dispatcher keeps a list of interests
  - Organized as a list of tuples {ClientID, ClientTopicsList}
  - Modified when a subscription is received
  - Used to compute the destinations of a published message

Politecnico di Milano

The dispatcher

```erlang
computeDestinations(Topic, Interests) ->
   computeDestinations(Topic, Interests, []).
computeDestinations(_, [], Result) -> Result;
computeDestinations(Topic, [{Client, Interests}|T], Result) ->
   Matches = matches(Topic, Interests),
   if Matches == yes ->
         computeDestinations(Topic, T, Result ++ [Client]);
      Matches == no ->
         computeDestinations(Topic, T, Result)
   end.

matches(_, []) -> no;
matches(Topic, [H|_]) when Topic == H -> yes;
matches(Topic, [_|T]) -> matches(Topic, T).

send(_, _, []) -> ok;
send(Topic, Message, [Client|T]) ->
   {client, Client} ! {Topic, Message},
   send(Topic, Message, T).
```

The dispatcher

Recursively analyze interests copying them into "Result"

```
addInterest(Interests, Client, Topic) ->
   addInterest(Interests, Client, Topic, []).
addInterest([], Client, Topic, Result) ->
   Result ++ [{Client, [Topic]}];
addInterest([{SelectedClient, Interests}|T], Client, Topic, Result) ->
   if SelectedClient == Client ->
         NewInterests = Interests ++ [Topic],
         Result ++ [{Client, NewInterests}] ++ T;
     SelectedClient =/= Client ->
         addInterest(T, Client, Topic, Result ++ [{SelectedClient, Interests}])
   end.
```

Until "Interests" becomes empty

Or until the client identifier is found and the new topic is added to the list of interests (First if clause)

The dispatcher: improvements

- Is our implementation of the dispatcher efficient?
- We use a list of {***Client***, ***Interests***} to represent the interest table
- When we process a publish message we need to check the topic in the ***Interests*** list of every client
- We can easily modify our code to store, for each topic, the set of interested clients …
- … using a list of {***Topic***, ***Clients***}
- We only have to slightly modify 3 functions

The dispatcher: improvements

```
computeDestinations(_, []) -> [];
computeDestinations(Topic, [{SelectedTopic, Clients}|T]) ->
  if SelectedTopic == Topic -> Clients;
     SelectedTopic =/= Topic -> computeDestinations(Topic, T)
  end.

send(_, _, []) -> ok;
send(Topic, Message, [Client|T]) ->
  {client, Client} ! {Topic, Message},
  send(Topic, Message, T).

addInterest(Interests, Client, Topic) ->
  addInterest(Interests, Client, Topic, []).
addInterest([], Client, Topic, Result) ->
  Result ++ [{Topic, [Client]}];
addInterest([{SelectedTopic, Clients}|T], Client, Topic, Result) ->
  if SelectedTopic == Topic ->
        NewClients = Clients ++ [Client],
        Result ++ [{Topic, NewClients}] ++ T;
     SelectedTopic =/= Topic ->
        addInterest(T, Client, Topic, Result ++ [{SelectedTopic, Clients}])
  end.
```

The dispatcher: now with stdlib

```
dispatcherLoop(Interests) ->
    receive
        {subscribe, Client, Topic} ->
            dispatcherLoop(addInterest(Interests, Client, Topic));
        {publish, Topic, Message} ->
            Destinations = computeDestinations(Topic, Interests),
            send(Topic, Message, Destinations),
            dispatcherLoop(Interests)
    end.

computeDestinations(Topic, Interests) ->
    dict:fold(fun(Client, Current, AccIn) ->
                    case lists:member(Topic, Current) of
                        true ->
                            [Client|AccIn];
                        false ->
                            AccIn
                    end end, [], Interests).

send(Topic, Message, Destinations) ->
    lists:foreach(fun(Client) -> {client, Client} ! {Topic, Message} end, Destinations).

addInterest(Interests, Client, Topic) ->
    dict:update(Client, fun(Current) -> [Topic|Current] end, [Topic], Interests).
```

Socket based distribution

- Erlang offers facility for socket communications
  - We introduce them using a single example (echo server)
  - This enables interaction with other programming languages

```erlang
-module(echo).
-export([listen/1]).

-define(TCP_OPTIONS,[list, {packet, 0}, {active, false}, {reuseaddr, true}]).

listen(Port) ->
  {ok, LSocket} = gen_tcp:listen(Port, ?TCP_OPTIONS),
  {ok, Socket} = gen_tcp:accept(LSocket),
  do_echo(Socket).

do_echo(Socket) ->
  case gen_tcp:recv(Socket, 0) of
    {ok, Data} ->
      gen_tcp:send(Socket, Data),
      do_echo(Socket);
    {error, closed} ->
      ok
  end.
```

Politecnico di Milano

Socket based distribution

- On the server side:
  - Listen
  - Accept (blocking)
  - Receive
- On the client side:
  - Connect
  - Send
  - Receive

```
> {ok, S} = gen_tcp:connect("localhost", 9000, [{active, false}, {packet, 2}]).
{ok, #Port<0.448>}
> gen_tcp:send(S, "Hello").
ok
> {ok, R} = gen_tcp:rect(S, 0).
{ok, "Hello"}
>R
"Hello"
```

Outline

Politecnico di Milano

- Introduction
- Sequential Programming
  - Data structures
  - Single assignment
  - Pattern matching
  - Functional abstractions
  - Dynamic code loading
- Concurrent / Distributed Programming
  - Processes
  - Fault tolerance
  - Distributed Erlang
  - Socket based distribution
- **OTP Introduction**

Politecnico di Milano

Open Telecom Platform (OTP)

- What is OTP (Open Telecom Platform)?
  - A set of design principles
  - A set of libraries
  - Developed and used by Ericsson to build large-scale, fault-tolerant, distributed applications
  - It also offers different powerful tools:
    - A complete Web Server
    - An FTP Server
    - A CORBA ORB
    - …

OTP Behaviors

- There exist structures/patterns used in a great number of different programs
  - Client / Server
    - Server waits for client commands, execute and return responses
  - Worker / Supervisor
    - Workers are processes that perform the computation
    - Supervisors monitor the behavior of workers
      - React when errors are detected (e.g. by restarting the worker)
    - Hierarchies (trees) of supervisors can be created as well
  - Event Manager / Handlers
    - Similar to Java listeners or to publish-subscribe paradigm
    - The manager detects an event
    - The handlers process the event

Politecnico di Milano

OTP Behaviors

- Let's take, for example, the client server paradigm
- What varies in different applications adopting this design paradigm?
  - Basically, what the server does
    - The ***functional*** part of the problem
  - The structure is fixed
    - The ***non-functional*** part of the problem
- The idea is to use higher order functions abstraction
  - The common non-functional part is implemented in modules called ***behaviors***
  - The functional part has to be implemented in modules that export predefined functions
    - ***Callback functions***
- Do not reinvent the wheel!

Applications

- OTP dictates also a common structure for ***applications*** i.e. pieces of code providing a specific functionality
- Following this structure applications can be:
  - Started, stopped, configured and monitored as a unit
  - Reused to build higher level applications
    - Included applications
- Often applications are defined as distributed
  - Run on different cooperating nodes
  - Realize fault tolerance using distributed worker/supervisor pattern
- This simplifies the design of component based architectures where different functional units can be combined to solve a complex task

Politecnico di Milano

Release handling

- Applications come with a release resource file that defines dependencies between applications
- It is possible to express dependencies involving the versions of considered applications
- Release handling tools
  - Start from release resource files
  - Can generate automatic procedures to update a particular application
    - Automatic resolution of dependencies
  - Based on low level dynamic code loading
  - Work in distributed scenario
  - Try to upgrade without stopping involved applications
  - Not always possible
    - Sometimes it is necessary to restart the application after upgrade
  - Not always easy to configure correctly

Politecnico di Milano

References

- J. Armstrong: "*Programming Erlang: Software for a Concurrent World*". Pragmatic bookshelf, 2007.
- R. Virding, C. Wikstrom, M. Williams: "*Concurrent Programming in Erlang (2nd Ed)*". Prentice Hall Intl, 1996.
- J.Armstrong: "*A History of Erlang*". In Proceedings of the 3rd conference on History of Programming Languages, 2007.
- J. Armstrong: "*Concurrency Oriented Programming in Erlang*". Invited paper in FFG, 2003.
- C. Wikstrom: "*Distributed Programming in Erlang*". In Proceedings of th 1st International Symposium on Parallel Symbolic Computation, 1994.
- J. Larson: "*Erlang for Concurrent Programming*". Queue 2008.
- Documentation available at:
  - http://www.erlang.org/doc.html
- OTP Team "Design Principles"