

Erlang

Jorge Coelho, Luís Miguel Pinho

Polytechnic Institute of Porto

Portugal

Introduction

Erlang [1,2] is a functional language developed by Ericsson and Ellemtel Computer Science Laboratories on the late 80's directed to the implementation of soft real-time concurrent applications. The original idea was to use a high level language in the telecommunications domain. Since the available languages, namely Prolog and Lisp were not oriented towards this specific domain; a new language was created to accommodate primitives to deal with concurrency and error recovery. Erlang has no types, does not implement backtracking and uses pattern matching as its core. It is based in a virtual machine with garbage collection allowing a high degree of portability.

The natural option for Erlang implementation of concurrency was to use processes without shared state concurrency. Functional programming does not explore the notion of state: variables cannot change after being binded and there is no such thing as global variables. Erlang uses a message-passing mechanism for communication between processes. This approach also removes a lot of problems derived from the use of locks.

Erlang is used in real projects and proved to be a good option for at least some highly fault-tolerant soft real-time applications. As an example, the Erlang community claims that based on the observation of the system performance, Ericsson has managed to achieve a reliability of 99.9999999% using Erlang in the AXD 301 ATM switching system [4]. This corresponds to a down time of 31 milliseconds/year.

Description

In this section we briefly describe the main features of Erlang concerning concurrency and fault tolerance. We start with a simple example to show the language syntax. The next program computes the area of several geometric shapes. It was taken from [3]:

```
-module(geometry).  
-export([area/1]).  
  
area({rectangle, Width, Ht}) -> Width * Ht;  
area({square, X}) -> X * X;  
area({circle, R}) -> 3.14159 * R * R.
```

The first line declares the name of the module, in this case ``geometry''. The second line declares the functions to be exported. This provides a simple form of encapsulation of data. In this program we state that we want to export function named *area* whose arity is 1. The next three lines of code describe the function *area* itself. When we want to call

this function we need to use its name and one argument containing a pair of the form $\{E1,E2\}$ or $\{E1,E2,E3\}$, these are called tuples (start and end with ``{'' and ``}'' respectively). The first element of these tuples is an atom or constant (starts with a lower case letter). The second and third are variables (start with an upper case letter). Whenever some call to the function *area* matches one of the three lines of the definition the result is computing the expression on the right of ``- ''. Let's take a look at this program executing:

```
1> c(geometry).
{ok,geometry}

2> geometry:area({rectangle, 10, 5}).
50

3> geometry:area({circle, 1.4}).
6.15752
```

Here we compiled the file ``geometry.erl'', and no errors were found. Then, we executed *geometry:area({rectangle, 10, 5})*, this matches with the first line of the definition of *area* and results in outputting $10*5$. The next line matches the calculation of the area of a circle. To reinforce the idea that Erlang code is simple and this is one of its main features we can compare the code above with an implementation in Java:

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double ht;
    final double width;
    Rectangle(double width, double height) {
        this.ht = height;
        this.width = width;
    }
    double area() { return width * ht; }
}

class Square extends Shape {
    final double side;
    Square(double side) {
        this.side = side;
    }
    double area() { return side * side; }
}
```

Exceptions

Erlang has an exception mechanism similar to the one present in Java. Exceptions are raised by the system when internal errors are encountered or explicitly in code by calling:

- `throw(Exception)`
- `exit(Exception)`
- `erlang:error(Exception)`

The *throw(Exception)* is used to throw an exception to a caller. The *exit(Exception)* is used when the process should terminate and if not caught is broadcasted to all processes that are connected to the one throwing the exception (connecting processes allows to implement more robust applications). The last exception is for denoting "crashing errors". Erlang implements a *try...catch* as follows:

```
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
```

The *try* evaluates function or expression *FuncOrExpressionSequence* and tries to match the result with one of the patterns (guards can be used to further filter the result). If it matches some of the defined patterns then it evaluates the corresponding expression. If it results in an exception being thrown, the *catch* tries to match the exception with one of the *ExceptionTypes* defined and proceeds evaluating the corresponding *ExExpression* (guards can again be used). The *after* is always executed in the end.

Concurrency

Erlang, processes are managed by the programming language and not the operating system. The programmer can easily create a process by executing, the *spawn* instruction:

```
Pid = spawn(Fun)
```

This creates a new process that evaluates function *Fun* in parallel to the caller. The *Pid* is a value that identifies a process. Erlang also supports the notion of *registered processes* where we can associate a name with a process identifier and use the name whenever we want to refer to the process. To send a message to a process one simply does:

```
Pid ! Message;
```

This process is asynchronous. The sender does not wait; the message is delivered to the receiver process's mailbox which can then process it when needed. Every process has a mailbox and all messages sent to the process are kept on its mailbox until processing. The receiver uses a *receive ... end* primitive to process the data received in its mailbox. Consider the following code:

```
-module(area).
-export([loop/0]).

loop() ->
    receive
        {From,rectangle, W, H} ->
            From ! {result,W*H};
            loop()
    end.
```

This code can be executed in an independent process by calling:

```
1> Pid = spawn(fun area:loop/0).
```

Then, one can send messages to it by calling:

```
2> Pid ! {self(),rectangle,10,5}
{result,50}
```

Where *self()* is a function that returns the process identifier of the calling process in order to receive an answer. We can also add timeouts to a process receive primitive. Consider the following example:

```
-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.
```

Here we use the *after* to execute some code if after *T* milliseconds no message matched the pattern inside the *receive*. As a final note one should state that receives can be guarded in such a way that expressions are evaluated only if the matching happens and some guard condition is true.

Robust programming

Erlang has primitives to deal with errors in concurrent programs. Processes can be linked in such a way that an error in one process is detected by another process which can take some measure to correct the problem. The code presented next, registers a process with a given name and restarts it whenever it exits:

```

keep_alive(Name, Fun) ->
    process_flag(trap_exit, true),
    register(Name, Pid = spawn(Fun)),
    link(Pid),
    receive
        {'EXIT', Pid, _} ->
            keep_alive(Name, Fun)
    end.

```

Also, it is easy to update process code via a message if we implement it in such a way that it is ready to receive and execute a new version. As a small example, consider the following code:

```

loop() ->
    receive
        {become, Something} ->
            Something()
    end.

```

This server waits for a message with the form *{become, Something}* where *Something* is a new definition for the server which is immediately executed. This way code upgrade is straightforward.

Conclusions and Future

Erlang may be suitable for domain specific soft real-time applications, but it lacks considerable features, essential to real-time languages. There is no support to real-time scheduling, and no concurrency and communication mechanisms apart the simple processes and messages.

A recent work addresses the use of Erlang in hard real-time systems [5]. The scheduler implemented into the Erlang virtual machine is a MultiQueue Round Robin scheduler, with three priority levels: normal, low and high. A forth priority level is available but undocumented and intended for internal use. The authors of [5] explore priority level max and use this to encapsulate a hard real-time scheduler as an Erlang service which provides a set of features to define tasks, priorities and deadlines. The work is still in an initial stage and does not address problems like real-time message passing and the effects of the unpredictable action of the garbage collector. These are lines of future work for the Erlang community.

Links and References

- [1] The Erlang open source webpage. <http://www.erlang.org/>.
- [2] Ericsson's Erlang webpage. <http://www.erlang.se/>.
- [3] Joe Armstrong. Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, July 2007.
- [4] Staffan Blau, Jan Rooth, Jörgen Axell, Fiffi Hellstrand, Magnus Buhrgard, Tommy Westin, and Göran Wicklund. Axd 301: a new generation atm switching system. Comput. Netw., 31(6):559-582, 1999.

- [5] Vincenzo Nicosia. Towards hard real-time erlang. In Erlang '07: Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop, pages 29-36, New York, NY, USA, 2007. ACM.