Principles of Programming Languages: Concurrent Erlang

Alessandro Sivieri, PhD

A.Y. 2013/2014

Outline

The actor model

Other Erlang characteristics

The actor model

- Originated in 1973
- Everything is an actor: an independent unit of computation
- Actors are inherently concurrent
- Actors can only communicate through messages (async communication)
- Actors can be created dynamically
- No requirement on the order of received messages
- ...

Concurrency oriented programming language

- Writing concurrent programs is easy and efficient
- Concurrency can be taken into account at early stages of development
- Processes are represented using different actors communicating only through messages
- Each actor is a *lightweight* process, handled by the VM: it is not mapped directly to a thread or a system process, and the VM schedules its execution
- The VM handles multiple cores and the distribution of actors in a network
- Creating a process is *fast*, and highly concurrent applications can be faster than the equivalent in other programming languages[1]

[1] You can find several statistics online, some can be controversial, I encourage you to check them out (and not necessarily believe only to the language authors).

Concurrrent programming

Three main primitives:

- `spawn` creates a new (concurrent) process executing the specified function, returning an identifier
- `send (!)` sends a message to a process (through its identifier); the content of the message is simply a variable. The operation is asynchronous
- `receive...end` is the code block that extracts the first message from a process's mailbox, and tries to match it against a series of patterns; this is blocking if no message is in the mailbox. The mailbox is persistent (until the process quits)

Concurrrent programming

```
start() ->
    InitialState = 0,
    spawn(fun() -> loop(InitialState) end).

loop(State) ->
    receive
        {Sender, get} ->
            Sender ! State,
            loop(State);
        {Sender, incr} ->
            loop(State + 1);
        {Sender, incrget} ->
            Sender ! State + 1,
            loop(State + 1)
    end.
```

Concurrrent programming

- We can associate a unique name to a PID (using the `register` function), and then address the process by this name
- We can specify a timeout for the `receive` block, to instruct what to do after a certain amount of time

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

stop() ->
    clock ! stop.

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

Fault tolerance

- Erlang supports fault tolerant programs directly at design time
- Processes can be *linked*, and the link monitors their state and communicates when the linked process exits
- The linker will then quit in a cascade-like effect, or it can receive the communication as a regular message and react to it in any way
- This behavior is then enhanced with the concept of *supervisor* (that we will not see here), that allows the design of complex supervision trees of processes with different reacting behavior to unexpected termination of children processes

Distributed programming

- Actors can be distributed in a network of Erlang *nodes*
- Each node is identified by a name and the hostname of the machine it is running in (e.g., `alpha@mycomputer.mydomain`)
- A process can be spawn on a different node (the PID implicitly takes into account the node identifier)
- A message can be sent to a process on a different node (known by PID or by name)

Outline

The actor model

Other Erlang characteristics

Erlang goodies

- OTP (Open Telecom Platform): (big) set of libraries and set of design principles
- *Behaviors*: ready-to-use design patterns (Server, Supervisor, Event manager. . .), only the functional part of the design has to be implemented (callback functions)
- Applications structure, with supervision: support to *code hot-swap*
- Application code can be loaded at runtime, and code can be upgraded: the processes running the previous version continue to execute, while any new invocation will execute the new code

References

- Joe Armstrong, *Programming Erlang: Software for a Concurrent World*, 2nd edition
- Fred Hebert, *Learn you some Erlang for Great Good!*
- Carl Hewitt, Peter Bishop, Richard Steiger, *A universal modular ACTOR formalism for artificial intelligence*, 3rd International Joint Conference on Artificial Intelligence, 1973