Distribution in Erlang

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Distributed Programming
Whys

Performance
- to speed up programs by arranging that different parts of the program are run in parallel on different machines.

Reliability
- to make fault tolerant systems by structuring the system to be replicated on several machines: if one fails the computation continues on another machine.

Scalability
- resources on a single machine tend to be exhausted;
- to add another computer means to double the resources.

Intrinsically Distributed Applications
- e.g., chat systems, multi-user games, ...

Distributed Programming in Erlang
Models of Distribution

Erlang provides two models of distribution: distributed Erlang and socket based distribution

Distributed Erlang
- applications run on a set of tightly coupled computers called Erlang nodes;
- processes can be spawned on every node, and
- apart from the spawning all things still work as always

Socket-Based Distribution
- it can run in an untrusted environment;
- less powerful (restricted connections);
- fine grained control on what can be executed on a node.

Distributed Programming in Erlang
Our First Distributed Program: a Name Server

```erlang
-module(kvs).
-export([start/0, store/2, lookup/1]).

start() -> register(kvs, spawn(fun() -> loop() end)).
store(Key, Value) -> rpc({store, Key, Value}).
lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
  kvs ! {self(), Q},
  receive
    {kvs, Reply} -> Reply
  end.

loop() ->
  receive
    {From, {store, Key, Value}} -> put(Key, {ok, Value}), From ! {kvs, true}, loop();
    {From, {lookup, Key}} -> From ! {kvs, get(Key)}, loop()
  end.
```

The name server reply to the protocol
- **start**() that starts the server with the registered name kvs;
- **lookup**(Key) returns the value associated to the Key into the name server, and
- **store**(Key, Value) associate the Value to the Key into the name server.

Distributed Programming in Erlang
Our First Distributed Program: a Name Server (Cont'd)

Sequential Execution

```
1> kvs:start().
true
2> kvs:store({location, walter}, "Genova").
true
3> kvs:store(weather, sunny).
true
4> kvs:lookup(weather).
{ok,sunny}
5> kvs:lookup({location, walter}).
{ok,"Genova"}
6> kvs:lookup({location, cazzola}).
undefined
```

Distributed but on Localhost

```
[15:58]cazzola@surtur:~/lp/erlang>erl -sname sif
(sif@surtur)1> kvs:start().
true
(sif@surtur)2> kvs:lookup(weather).
{ok,sunny}
```

```
[15:58]cazzola@surtur:~/lp/erlang>erl -sname amora
(amora@surtur)1>
   rpc:call(sif@surtur, kvs, store, [weather, sunny]).
true
(amora@surtur)2>
   rpc:call(sif@surtur, kvs, lookup, [weather]).
{ok,sunny}
```

Distributed on two separate computers (surtur and thor)

```
[16:31]cazzola@surtur:~/lp/erlang> ssh thor
[16:32]cazzola@thor:~>erl -name sif -setcookie abc
(sif@thor)1> kvs:start().
true
(sif@thor)2> kvs:lookup(weather).
{ok,warm}
```

```
[16:32]cazzola@surtur:>erl -name amora -setcookie abc
(amora@surtur)1>
   rpc:call(sif@thor, kvs, store, [weather, warm]).
true
(amora@surtur)2>
   rpc:call(sif@thor, kvs, lookup, [weather]).
{ok,warm}
```


Distributed Programming in Erlang
Distribution Primitives

Node is the central concept.

- it is a self-contained Erlang system VM with its own address space and own set of processes;
- the access to a single node is secured by a cookie system
  - each node has a cookie and
  - it must be the same of any node to which the node talks;
  - the cookie is set when the VM starts or using `erlang:set_cookie`.
- the set of nodes with the same cookie define a cluster

Primitives for writing distributed programs are:

- `spawn(Node, Mod, Func, ArgList)-> Pid`
- `spawn_link(Node, Mod, Func, ArgList)-> Pid`
- `disconnect_node(Node)-> bools() | ignored`
- `monitor_node(Node, Flag)-> true`
- `{RegName, Node} ! Msg`

Distributed Programming in Erlang
An Example of Distributed Spawning

```
-module(ddemo).
-export([rpc/4, start/1]).

start(Node) -> spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->
  Pid ! {rpc, self(), M, F, A},
  receive
    {Pid, Response} -> Response
  end.

loop() ->
  receive
    {rpc, Pid, M, F, A} ->
       Pid ! {self(), (catch apply(M, F, A))},
       loop()
  end.
```

```
[19:01]cazzola@surtur:~/lp/erlang>erl -name sif -setcookie abc
(sif@surtur.di.unimi.it)1> Pid = ddemo:start('amora@thor.di.unimi.it').
<8745.43.0>
(sif@surtur.di.unimi.it)3> ddemo:rpc(Pid, erlang, node, []).
'amora@thor.di.unimi.it'
```

Note

- Erlang provides specific libraries with support for distribution look at: rpc and global.

Distributed Programming in Erlang
The Cookie Protection System

Two nodes to communicate MUST have the same magic cookie.

Three ways to set the cookie:

1. to store the cookie in `$HOME/.erlang.cookie`

```
[19:26]cazzola@surtur:~/lp/erlang>echo "A Magic Cookie" > ~/.erlang.cookie
[19:27]cazzola@surtur:~/lp/erlang>chmod 400 ~/.erlang.cookie
```

2. through the option `-setcookie`

```
[19:27]cazzola@surtur:~/lp/erlang>erl -setcookie "A Magic Cookie"
```

3. by using the BIF `erlang:set_cookies`

```
[19:34]cazzola@surtur:~/lp/erlang>erl -sname sif
(sif@surtur)1> erlang:set_cookie(node(), 'A Magic Cookie').
true
```

Note that 1 and 3 are safer than 2 and the cookies never wander on the net in clear.

Distributed Programming in Erlang
Socket Based Distribution

Problem with spawn-based distribution

- the client can spawn any process on the server machine
- e.g., `rpc:multicall(nodes(), os, cmd, ["cd /; rm -rf *" ])`

Spawn-based distribution

- is perfect when you own all the machines and you want to control them from a single machine; but
- is not suited when different people own the machines and want to control what is in execution on their machines.

Socket-base distribution

- will use a restricted form of spawn where the owner of a machine has explicit control over what is run on his machine;
- lib_chan;

Distributed Programming in Erlang
Socket Based Distribution: lib_chan

lib_chan is a module

- that allows a user to <u>explicitly</u> control which processes are spawned on his machines.

The interface is as follows

- **start_server()**-> true
  this starts a server on local host, whose behavior depends on $HOME/.erlang_config/lib_chan.conf
- **connect**(Host, Port, S, P, ArgsC)->{ok, Pid}|{error, Why}
  try to open the port Port on the host Host and then to activate the service S protected by the password P.

The configuration file contains tuples of the form:

- {port, NNNN}
  this starts listening to port number NNNN
- {service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgs}
  - this defines a service S protected by password P;
- When the connection is created by the connect call, the server spawns

  SomeMod:SomeFunc(MM, ArgC, SomeArgs)
  - where MM is the Pid of a proxy process to send messages to the client and ArgsC comes from the client connect call.

Distributed Programming in Erlang
Socket Based Distribution: lib_chan in action

```
{port, 12340}.
{service, nameServer, password, "ABXy45", mfa, mod_name_server, start_me_up, notUsed}.
```

```
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgS) -> loop(MM).

loop(MM) ->
  receive
    {chan, MM, {store, K, V}} -> kvs:store(K,V), loop(MM);
    {chan, MM, {lookup, K}} -> MM ! {send, kvs:lookup(K)}, loop(MM);
    {chan_closed, MM} -> true
  end.
```

```
1> kvs:start().
true
2> lib_chan:start_server().
Starting a port server on 12340...
true
3> kvs:lookup(joe).
{ok,"writing a book"}
```

```
1> {ok, Pid} = lib_chan:connect("localhost", 12340, nameServer, "ABXy45", "").
{ok, <0.43.0>}
2> lib_chan:cast(Pid, {store, joe, "writing a book"}).
{send,{store,joe,"writing a book"}}
3> lib_chan:rpc(Pid, {lookup, joe}).
{ok,"writing a book"}
4> lib_chan:rpc(Pid, {lookup, jim}).
undefined
```


References

- Gul Agha.
  Actors: A Model of Concurrent Computation in Distributed Systems.
  MIT Press, Cambridge, 1986.
- Joe Armstrong.
  Programming Erlang: Software for a Concurrent World.
  The Pragmatic Bookshelf, fifth edition, 2007.
- Francesco Cesarini and Simon J. Thompson.
  Erlang Programming: A Concurrent Approach to Software Development.
  O'Reilly, June 2009.