

# The Erlang Rationale

By

**Robert Virding**

This is a description of some of the basic properties and features of Erlang and an attempt to describe the rationale behind them. Erlang grew as we better understood the original problem we were trying to solve, telephony, and as we evolved the basic concepts for solving the problem.

One major point I hope to show here is that most of the features of Erlang, both the language and the system, are not isolated properties or were developed in isolation. They were designed to all interact with each other. For example: processes, process communication, distribution and error handling are all based on common principles which allow them to interact more or less seamlessly with each other; pattern matching, which is ubiquitous, is always the same irrespective of where it is used and is the only way to bind variables.

## ***Terminology***

I try to avoid using standard terms here in a non-standard context. So “objects” refers to objects in the standard OO manner, “processes” refers to Erlang and OS processes as opposed to threads, which we never have to deal with in Erlang.

## ***First principles***

This is no history of Erlang, read Joe’s HOPL paper for that (I am still incapable of making small changes to anything), but knowing the initial problem and our solution to it will help to understand Erlang. The problem was telephony, i.e. large switches, and the properties of a language/system to program telephony we felt should contain:

Lightweight concurrency – This is critical, the system should be able to handle large number of processes, and process creation, context switching and inter-process communication must be cheap and fast.

Error handling – This is critical, the system must be able to detect and handle errors.

Continuous evolution of the system – We want to upgrade the system while it is running and with no loss of service.

High level language to get real benefits.

Asynchronous communication – The problem domain used asynchronous communication.

Process isolation – We don’t want what is happening in one process to affect any other process.

Some other properties we thought were important:

The language should be simple – Simple in the sense that there should be a small number of basic principles, if these are right then the language will be powerful but easy to comprehend and use. Small is good.

We should provide tools for building systems not solutions – We would provide the basic operations needed for building communication protocols and error handling.

All these properties were so basic and important that we felt that they had to be built in from the beginning and supported by the language. Adding them afterwards would just not cut it.

From the beginning we saw Erlang as a language to control hardware.

## ***Erlang “Things”***

It was early decided that there would only be two different basic types of “things” in Erlang, the normal immutable data structures and processes. All things in Erlang were meant to be either a data structure or a process. We also added mutable things (like the process dictionary) but we did not overly publicize the fact, and discouraged use of the process dictionary. Note that while the process dictionary itself is mutable, it is really just an implicit dictionary, the data stored in it is not.

## **Immutable data structures**

These are the normal Erlang terms. Having them as immutable made everything much simpler, both conceptually and in the implementation.

Personal thought: Immutable data suits a high-level language, having mutable data gets you in to all sorts of trouble and difficulties, just read descriptions of other languages which have it and the difficulties in describing what gets changed and when, for example Python’s copy and deep\_copy. Mutable data, however, is much easier to comprehend in a low-level language like C, K&R C, where you directly see which data is passed by value and which is passed by reference, i.e. mutable.

## **Processes**

A process is something which obeys process semantics:

All communication is through asynchronous message passing.

Links/monitors for error detection/handling.

Obey/transmit exit signals.

Parallel independent execution.

There were to be no “back doors” in how processes communicate and all messages sent were complete, no partial messages. This is in fact essential for building robust systems, you **know** that either the message has been sent, or it hasn't, there is no uncertainty. . Also, you don't know if the message has been received.

**N.B. Nothing is said here about how processes are implemented, or where, or in what language etc, only how the rest of Erlang perceives them. If it obeys process semantics then it is a process.**

### ***Process communication***

All communication between processes was meant to be asynchronous, this means messages, exit signals and control signals (link, unlink, etc). It means that all BIFs dealing with processes are also meant to be asynchronous, they can really only check their arguments and not the result of sending off their message/control signal. This is why BIFs like link/1 used to return errors of non-existent processes by exit signals not by return values.

Having all communication asynchronous also allowed us to avoid a number of problems with built-in synchronous communication:

The level of security in the message protocol.

How to decide when an error has occurred.

Many protocols are asynchronous.

This was really only broken in one way and that was when sending a message to a registered process where it is checked whether a process is actually connected to the name.

### ***Error Handling***

#### ***Distribution***

While it was Klacke who first implemented distribution we had planned for distribution before that while we were thinking about process communication and error handling. We decided that truly asynchronous communication was best for distribution..

Distribution is based on the concept of loosely coupled nodes.

Distribution was always meant to be transparent, if so desired. This meant process communication and error handling must work the same for distributed processes as for local processes.

## **Ports**

Ports are the mechanism for communicating with the outside world, i.e. anything “outside” of Erlang. Ports were designed to obey process semantics as this was the best way to make them fit into the standard Erlang execution model. In fact, there was a serious discussion whether there should be a separate port data type or whether they should be Pids. We finally decided to make them a separate data type as we felt that there could be times when it might be necessary to be able to detect if we were communicating with a process or a port. But apart from a type test and the `open_port` function ports behaved as processes, they *were* processes in the Erlang sense. As processes are concurrent things then sending message will never hang the sender which can happen if the interface is based on function calls. It is up to the implementation to ensure this never happens. It also meant that no, for Erlang, new mechanisms were needed for handling ports.

As ports were meant to look like processes their interface was message/link based as are processes. Also ports shouldn't know more than is communicated by messages.

The model for ports was streams from UNIX. Having ports as processes would make it transparent with what was actually being communicated with. You could transparently insert filters between the application and the port for processing data.

This makes the comments in the module `erlang` documentation of how the `port_XXX` functions are cleaner and much more logical than the message interface a bit *strange*. The message protocols were first, the functions came later.

## **Typing**

We all came from a dynamically typed past so having Erlang dynamically typed seemed the most natural thing to do. Although this has been the cause of much discussion and made us a little off in the academic functional language world I still think it was the right decision. Also with loosely coupled distribution where nodes were more or less allowed to come and go as they pleased it would be hard to statically type communication between nodes.

## **Modules, code and code loading**

### **I/O system**

The i/o system is process based, as it should be in Erlang. This allows it to be very versatile, for example starting shells on one node and having its i/o redirected to another node is trivial.

The central part of the i/o system is the i/o-server. It is the process to which i/o requests are sent from the application and which maps these into suitable messages to the i/o device/port. An i/o-server must handle device specific requests as well as the generic i/o requests for example through the module 'io' from the application. It is the i/o-server which handles matching the characters read/written in the applications i/o requests against the actual device which may need buffering to create requests of appropriate size. It is only in very trivial cases like having fixed size records where this is not needed. All necessary buffering of data between requests, both input and output is done by the i/o-server.

Splitting the i/o system in this fashion has two important benefits:

It made it possible to use generic i/o functions. The function for doing formatted output would not need to know to what type of device it was generating output, the i/o-server would handle the specifics of the device. The same applies for input, we only need *one* function to scan for Erlang tokens and the i/o-server provides it with input characters as needed. The alternative would be to have one set of i/o functions for each type of device, which quickly become completely unmanageable.

It meant that an i/o-server for a specific device is generic and can be used for many different types of data requests. It is not necessary to open a device for a certain type of interaction, for example lines or Erlang forms. So it is possible to interlace requests to read Erlang tokens, lines, fixed-length records and anything else through one i/o-server to a device.

The generic part of the i/o-server is truly generic and can handle any read or write requests. The module 'io' together with the module 'io\_lib' implements a basic Erlang oriented i/o interface but an i/o-server can implement any form of interface through the following protocols.

## **Basic message format**

**{io\_request, From, ReplyAs, Request}**

**{io\_reply, ReplyAs, Reply}**

These are the messages for sending a request to an i/o-server and returning the reply. The ReplyAs term is created by the client and sent to the server as the way for it to identify to which request this is a reply to. It can be any term and the i/o-server only uses it in its replies. Doing it this way is both efficient and very versatile. A client can, for example, send many requests to a server and then selectively receive replies to these requests in any order it wishes, which can be useful if the i/o-server has non-blocking requests. It also allows the i/o-server to pass requests on to other processes which can then directly reply to the client.

## Generic requests to an i/o-server

**{put\_chars, IoList}**

**{put\_chars, Module, Function, Args}**

These requests are for output. The first takes an iolist generated by the application and sends it to the device while the second the i/o-server calls `Module:Function(Args)` which must return an iolist which is then sent to the device. This allows the application to decide *where* the work to generate the output data is to be done. For example in the module 'io' calling `io:fwrite(...)` causes `{put_chars,io_lib,fwrite,[...]}` to be sent to the i/o-server which then evaluates the `fwrite`.

These reply either 'ok' or '{error,Error}'.

**{get\_until, Prompt, Module, Function, ExtraArgs}**

This request is for input. The Prompt is included in the input request. If a prompt is not needed for that device then it is ignored by the i/o-server.

The major problem when doing input is that in most cases you don't know how many characters are needed from the device to complete the input request. For example reading an Erlang form can require anything from a few characters up to tens of thousands (for a truly enormous function). Even a line can be of variable length. For some types of devices, for examples files, it may be feasible to read in the whole input in one go, but in many cases this is impossible, for example requiring users to enter their whole shell input in one go before doing any processing is not realistic.

Two ways of handling this are:

Call the input function with an argument which is a function to be called when more characters are needed.

Make the input function re-entrant so if there are not enough characters then a continuation is returned which can be called with more characters to continue the collecting.

The second alternative was chosen as it completely separates the input function from collecting characters from the device and also allows the i/o-server to retain control during input. For example even during input processing the i/o-server can still easily look for and handle output requests, if the device allows it.

The continuation is a data structure created by the input function which it can use to continue collecting input when called with more characters. If implemented today it would most likely be a fun, but these did not exist then.

You call the input function with some characters and it tries to collect enough characters

```
apply(Module, Function, [Continuation, Characters | ExtraArgs])
```

and it returns

```
{done, Result, RestChars}
{more, Continuation}
```

If the input function could get enough characters then it returns 'done', the result to return to the caller and the remaining characters to keep for the next input request. If there are not enough characters then 'more' is returned and a new continuation. The input function is then called again with the new continuation and more characters and this is continued until enough characters have been collected. The continuation is initially set to []. A generic loop would look something like:

```
get_until(Cont, Chars, Mod, Func, Args) ->
  case apply(Mod, Func, [Cont,Chars|Args]) of
    {done,Result,RestChars} ->                               %Result may be error
      {done,Result,RestChars};
    {more,NewCont} ->
      MoreChars = get_some_chars(),                          %Device specific
      get_until(NewCont, MoreChars, Mod, Func, Args)
  end.
```

For example `io_lib:collect_line/2` obeys this protocol and returns a line, `erl_scan:tokens/3` obeys this protocol and returns a list of tokens up to and including a 'dot' and the lexical analyzer generator 'leex' creates functions `token/3` and `tokens/3` which obey this protocol.

## ***Process groups, Jobs and JCL***

As Erlang evolved there was the idea that Erlang was, in many ways, like an operating system, not just a language running within an OS. One result of this way of thinking was that in one Erlang system you would want to run many separate applications concurrently. Each application would consist of a number of processes working together but sharing the basic Erlang system. As the applications were separate then there would be general "system" information that would be specific for each application but it should be able to be accessed in a common way.

To address this problem process groups were created. They are not the same as the UNIX concept of the same name. It was decided keep the flat, independent process structure which already existed, and still exists, and not impose a tree structure as exists in UNIX. This means that the process group is a much looser entity and in many ways completely transparent.

Each process has a 'group leader' which it inherits from the process which spawned it. A process group consists of all the processes which have the same group leader. The group

leader is a normal process which has no a priori knowledge of the processes in its group. A process can change group simply by setting its 'group leader' to another process. Processes could then communicate with the group leader to get the process group's common data.

There was much system information which could be "specialised" in this way; two obvious ones are default i/o streams and current working directory. Unfortunately this feature was never used to its full extent and is now only used for default i/o, the standard group leader process is an i/o-server process. Two reasons for it not being used is that the idea of running multiple applications on one Erlang system is not the common use (correct me if I am wrong) and the general lack of knowledge of the feature (again, correct me I am wrong).

It would be good to have some example programs that illustrate this 😊

Using process groups it is very easy for different applications to have different default i/o channels. For example some could be communicating through separate windows, some directly to a file and some over TCP to another system (a browser interface to Erlang).

One problem with running multiple applications concurrently is that if more than one is communicating with the user through the same channel then the i/o will become very jumbled.

The solution to this was to introduce the concept of a 'job' and a user driver which controls which job is currently connected to, and hence communicating with, the user. A job is a process group together with its group leader. The user driver communicates the job through its group leader. Within the user driver you can start new jobs both locally and remotely, kill jobs and change which job is connected to the user. The i/o from jobs which are not connected is buffered (simply by not receiving it) until another job is connected and its i/o sent to the user. Controlling jobs in the user driver is done with JCL.

In the current implementation only new jobs running Erlang shells can be started but this is not an inherent limitation, any function could be spawned instead. In LFE the user\_drv module has a trivial extension which allows this.

```
SHAPE \* MERGEFORMAT
```

## ***Group servers and the default group server, group.erl***

### ***Patterns, Pattern matching and Guards***

Patterns and pattern matching are a Big Win. This was not always obvious to people, especially those coming from more traditional languages where you explicitly have to

construct data structures, not just write them down. One of the things we decided very early on was to make sure that the pattern used to construct data and the pattern used match and pull a part data must be the same. Fortunately Erlang has been able to keep this rule even with new data types which did not exist in the beginning (for example binaries).

Patterns provide a clear way of describing the structure of a term, both for constructing and for matching, but it is difficult to include other types of information, for example relationships or data types, in the pattern itself. Not impossible, just difficult to keep it clear. Guards were originally intended to be simple tests providing a simple extension to pattern matching which constrains the values of the variables in the pattern. Originally when they were just simple tests the difference between guards and normal expressions was greater and more distinct. When guards were extended with full boolean expressions (which in itself is was probably a Good Thing) this difference became much less distinct and guards then became seen as restricted normal expressions, which they are not. This has caused problems.

### ***Variables, scoping and the '=' operator***

Variables in Erlang are just references to values. They can never be rebound; once they have been bound to a value then they have that value throughout the rest of the function body. This property is common to most (all?) declarative languages. While we all came from an updatable back-ground (C, Basic, FORTRAN, Lisp, Smalltalk ...) this feature was never seen as a problem.

One of the things which Erlang inherited from its Prolog beginnings was variable scoping, or rather lack of variable scoping. Once a variable has been defined (bound) in a function body then it exists throughout the rest of the function body and all occurrences of a variable with the same name refers to the *same* variable. The scope of the variable is the whole function body.

These two properties affect many different things, for example pattern matching. If the name of a bound variable occurs in a pattern then it the same variable as the bound one, so it must already have a value and matching against it must mean testing its value. This is different from most functional languages where all variables occurring in a pattern are new unbound (scoped) variables. Changing this would mean that you would either have to add scoping or allow variables to be rebound.

While it is possible to never use variables to store intermediate values this is seldom practical. In Erlang we first used '=' to bind a variable to the value of an expression. This was more consistent than using a 'let' construction as we had no variable scoping. You often want to return multiple values from a function, for example the actual return value and some updated state information. We decided to do this by returning a tuple from the function and then pulling it apart to get at the different values. It was then logical to extend

'=' by allowing a pattern instead of just a variable and using full pattern matching so as to be able to pull the return value apart in one step.

As Erlang is a functional language everything is an expression and all expressions must return a value and we decided that the "Pattern = Expression" expression should return the value of the RHS. This allowed doing "Pattern1 = Pattern2 = Expression" so you could apart you cake and keep it at the same time. This was before aliases were added to patterns.

We did have discussions on whether to use another operator than '=' as we were no longer doing an assignment, but we could never really agree on an alternative and we didn't consider it an important issue so we just dropped it.

## **Macros**

Erlang macros are based on traditional C macros with all their power and limitations. They were originally added to provide named constants. While adding them I thought that I might as well add arguments to macros and conditional compilation. One thing missing is having macros of the same name but with different arities like normal Erlang functions.

They are token based and have very little knowledge of Erlang syntax. This allows you to do wonderful things with them but can also make using them confusing. The closest things to proper syntax based macros like in Lisp are parse transforms. As these directly manipulate the Erlang AST they are not trivial to use.

As they are a compile time construct in one sense they don't really exist.

It was probably a mistake to have token level macros. We should have also provided a string level macro facility as well. Note, we know that macros destroy many of the nice properties of a module, but they also allow the language to be used for real applications. There is a delicate balance here between purity (=no macros) and power (=macros). We chose to add a little extra power at the expense of purity. We have always taken the *pragmatic* view of language construction and allowed impure features provided that they are not too *filthy* and that the benefit is large.

## **Records**

Records were added to solve a problem for our first customer. They were good users and we really wanted to help them. What they wanted was:

Named fields in tuples, preferably with default values.

Not slower than doing it explicitly with element/setelement, if it was slower they would not use it.

These requirements basically meant that everything had to be done at compile time, finding fields at runtime would just not cut it. This forced records into being a compile-time construction which went against the grain of rest of Erlang. Together with macros they are still the only compile-time constructions in the language.

One basic choice which had to be made was whether the names of fields were to be unique for each type of record or co-exist in a global field name space. Both choices had pros and cons and there would be a trade off in making a choice:

A record unique field name space would allow clearer names and remove problems with name clashes between records (except for the record name). However it would mean that each use of records would need to include the record name as all variables are untyped.

A global field name space would mean that the record name would not be always needed but would allow for clashes between field names. The best solution for field names would then to be to prefix them explicitly with the record name as is needed with module names. Many do not like this feature of a flat global module name space.

Record unique name spaces were chosen.

It was also decided to use a special syntax for record operations. Designing this syntax was not trivial as the constructor/access patterns would have to be consistent with matching patterns and interact with the '=' operator which does not assign but matches and returns the value. This means that writing something like:

```
X#person.name = "Robert"
```

while it looks wonderful, *must* mean something completely different to changing the 'name' field of the 'person' record stored in X. Unfortunately. Many have complained but so far none have come up with a better, consistent suggestion.

In retrospect it may have been better to use a function call like syntax as is done in Common Lisp structures and CLOS, and as is done in LFE. This however would have looked very funny in patterns, at least from an Erlang point of view.

### ***Non-terminating discussions***

These are a number of discussions we had which did not result in changes to Erlang but which were interesting in their own right (I think).

### **RPC-like function API vs message protocol**

One discussion we had over many years was whether it was better to specify an interface

through a function API or through a message protocol. An RPC interface is often easier to use but much less versatile, while exposing the message protocol gives the user full control over the connection.

### **Variable scoping and 'let'**

Should we have scoped variables as in other functional languages or should we have the no scoping as in Prolog. Scoping is clearer while no scoping makes it easy to do multiple returns from if/case/receive which can be practical. In the end inertia and backward compatibility prevailed and we did nothing. Although I still feel that is wrong!

### **The problem with 'if'**

Sigh. If was added as an almost on the spur of the moment hack. We knew how to compile guards and thought it might be useful in the case where you don't have a value to match against. We never used it much ourselves and therefore didn't really worry much about it. There were on and off discussions about it which never led anywhere.

I have nothing against type checkers like dialyzer (which does other useful checks as well) but I still see no need to include them in the *language*.

We must use apply here as we need to build the argument list.

Job Control Language.

Lisp Flavoured Erlang, a new front end to Erlang with Lisp syntax.

Even in Lisp, which is powerful language personified, you have to construct data structures. This makes keeping the sameness between constructing and matching difficult. Yes, I know about backquote but it is still not trivial.

I have seen examples of where this has been done and I do not think the result was very legible.

We could have added multiple returns as in lisp but we felt that using tuples was easier to implement and safer. And probably more efficient when you didn't need them.

Yes I have seen suggestions which remove the need for the record name in some cases, but they do not attack the field changing syntax which so many love.

Group leader

Group leader

■■■■■■■

Process group

Process group

”user”

User driver

Job 1

Job 2