# The Interworking's of Erlang

Yuwei Chen

**Abstract**

Erlang is a general purpose functional concurrent programing language. Erlang is used mainly for telecom services. This research primarily focuses on exploring what defines and separates erlang from other languages as well as it's applications on multithread and concurrent processing. Some of the things that are stated in this paper are some of the environment's erlang code can be run on, the underlying syntax of the language and some of the safety measure and design choices that erlang has to combat errors and downtime.

## Introduction

Erlang first developed in 1986 by Joe Armstrong for the company Ericsson. The language was first designed to make telecom communication easier and more efficient. Since availability always needed to be high, there needed to be a language and system built on concurrency and speed. Erlang provided an easy general-purpose language that specialized in transferring data.

Since most messages sent through telecom does not need to be modified, for this reason erlang was made a functional language. This makes everything much simpler and decreases errors. This research focuses on what makes erlang different and show some of the pros and cons of some of the design choices of the language.

## Erlang Shell/Environments

Erlang can be run on any Unix/ Linux system along with many verities of emulators. If one were to use erlang on a Unix/Linux system, one would have to type in erl -s toolbar. This will launch and instance of the erlang shell. This would allow you to execute erlang commands. A series of text should appear stating that a version of the BEAM emulator has been started. Then you can enter erlang code. To compile a module in erlang. One must using the c function. c(sampleModule.erl) would compile a module called sampleModule. An error would be spit out if any part of the module was not written correctly. This ease of compilation is one of the reasons erlang is so popular.

## Syntax

Erlang syntax is very different from most languages. All lines must end with a period and a white space character. For arithmetic operators, the primitives that erlang supports directly is integers and floats. All operators must come between two numbers. However, integer division is not directly supported by the division symbol. The function div must take the place the division symbol. This is also true for remainders. Remainders is implemented into the language through the function rem. A cool mathematic operator that should be included into other languages is the use of the # sign for the change of base. For example, 2 # 1000. will give you 8. This operator converted the binary sequence 1000 into base 10 which is 8. This also works with Hex. 16 # AE. Will give you 174. This allows for quick conversions between bases.

**Variables**

Functional languages do not allow variables. However, erlang allows non mutable variables. This means that all variables created in erlang can only be bounded once. All variables must start with a capital letter. For example, One = 1. Will bound the value 1 to the variable One. We then can not say One = 2. Erlang will complain about this line. Another interesting thing is that we can assign multiple variables at the same time. Un = Uno = One = 1. will assign the value 1 to all three variables. You can also assign variables to other variables. For example, Un = Uno. would compile because Un and Uno has the same value. Un = Uno + 1. would not compile. The assignment operator uses pattern matching to assign the left side to the right. There is a keyword variable called _. This variable is for testing purposes and will not store any values after it is used in an operation.

**Atoms**

The reason all variables must start with a capital letter is because of atoms. Atoms are literals. Atoms are not mutable and starts with lowercase letters. If one wants to use special characters or uppercase letters in their atoms, they must surround the string with single quotes. For example, atom. will return atom. ‘Atom12@gmail.com’ will return Atom12@gmail. Atoms are made this way so that you will never have an undefined value. This makes atom comparisons really easy as atoms can never change, and you don’t need a specific function to compare two atoms. Although Atoms are very useful, they are not garbage collected. This means

that all declared atoms will stay in memory until the end of the program.

### Tuples

Tuples are used in many languages to organize data for the user. For example, coordinate points. In erlang tuples are surrounded by curly braces. The coordinate tuple {3,4} shows that 3 represents the x coordinate and 4 represents the y coordinate. However, in erlang you can use the underscore as a place holder or a wild card. If you wanted coordinates on the x = 3 coordinate line, one can assign {3, _} equals to coordinate tuple. It will replace the underscore with the value of the right-side tuple. The only restriction is that the number of values I the tuples must match.

### Lists and Strings

Lists and Strings cause a large problem in erlang. The reason being that they have the same syntax. Both strings and lists are surrounded by brackets. This can cause many problems when interpreting the code. Lists in erlang supports multiple types in the same list. This can cause many problems but is very powerful. For example, you can have a mix a list of numbers with atoms and floats and doubles. This increases the amount of applications a list can have. However, this can lead to many problems. Take this list for example [97,98,99].. This will return abc. The reason being the ascii numbers of abc is 97,98, and 99. Erlang will only recognize a list as a list if at least one element in the list cannot be converted to a character.

Some cool features of lists in erlang is the ++ and -- operator. Using this operator on two lists will append lists and finds the difference between two lists. A example would be the [1,2,3] ++ [5,6,7]. This expression would return [1,2,3,4,5,6,7]. You can also take the difference of two lists. [1,2,3] -- [1,2] would return [3].

Some other things one can do with lists in erlang is the use of hd and tl. This is the same as car and cdr in lisp. It takes the head and the tail of a list. One can also construct a new list using the pipe symbol. [Head | Tail] = [1,2,3,4,5]. Would assign the variable Head the value 1 and Tail the list [2,3,4,5]. You can extend this to construct any variation of a list. [1, | [ 2, 3 | []]] constructions a list by appending the separate lists of [1] [2,3] and a empty list. A empty list indicates that there can be more elements in the list.

### List Comprehensions

There are many things one can do with lists in functional languages. Erlang has

many of the same list comprehensions that other function languages has. For example the expression
[X || X <- [1,2,3,4,5,6,7,8,9,10],
X rem 2 =:= 0]. This gets all even elements on said list. In a longer sense the statement says assign X the list [1,2,3,4,5,6,7,8,9,10] then get each element in each individually. If the X modulus 2 is 0 then return that number. The list of atoms
Weather = [{toronto, rain},{montreal, storms}, {london, fog}, {paris, sun}, {boston, fog}, {vancouver, snow}]. can be used in very useful ways. With the statement [X || {X, rain} <- Weather]. One can get all cities that are raining. Such a simple statement are very powerful in functional languages.

## Modules

Modules are like classes in java. They hold several functions with a number of attributes. Modules allow developers to organize and write custom functions into one space. This makes importing new functions much easier. Here is an example a module.

```
-module(useless).
-export([add/2, hello/0, greet_and_add_two/1]).

add(A,B) ->
    A + B.

%% Shows greetings.
%% io:format/1 is the standard function used to output text.
hello() ->
    io:format("Hello, world!~n").

greet_and_add_two(X) ->
    hello(),
    add(X,2).
```

The module is called useless and has three functions. Each function export statement must have the name of the function and number of parameters for said function.

## Control Statements

Controls statements in erlang are similar to languages in that time period. All if statements must be followed by a then statement. There are also optional statements of else if and else statements. These keywords are the same as in Java. However, all if statements must end with the end keyword.

```
function greet(Gender,Name)
    if Gender == male then
        print("Hello, Mr. %s!", Name)
    else if Gender == female then
        print("Hello, Mrs. %s!", Name)
    else
        print("Hello, %s!", Name)
end
```

Another control statement in Erlang is guards. Here is an example of guards.

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
```

It is a series of Boolean conditionals. In this case if you are between the ages of 0 and 15 you are not old enough. As one can see, the underscore symbol is used again for a wild card. It is very important that it is at the end of the list of statements. This is because it the wild card can be any value. A better way of doing this is in the example below.

```
old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.
```

By using the when keyword, we can get rid of the redundancy of all the guard statements.

Switch case is another control structure that erlang implements. In the example below we are trying to see if we should go to the beach.

```
beach(Temperature) ->
    case Temperature of
        {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
        {kelvin, N} when N >= 293, N =< 318 ->
            'scientifically favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable in the US';
        _ ->
            'avoid beach'
    end.
```

The reason the erlang implementation of switch cases is more powerful then average switch case is that we can implement functions and complex list comprehension statements within the case statements. In this case we took case of Celsius, kelvin and Fahrenheit all in one switch case statement. Since we have the ability to use complex pattern matching statements in switch statements, we can implement rather complex situations in a fairly simple way.

## Typing

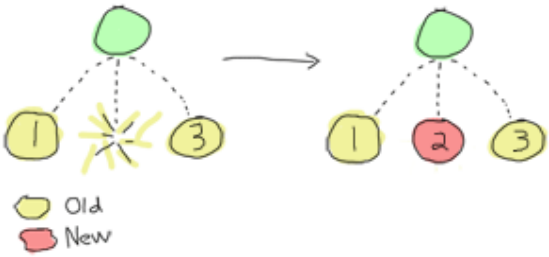
Erlang enforces dynamically type binding. This means that on compile time the compiler does not check that all your variables are initiated, or all your comparisons are correctly written. This does pose a series of problems for most languages that are dynamically typed. However, erlang is the exception to this. Erlang was designed in a way that enforces that no variable can be declared without initializing it. This combined with the independent based functions in erlang. The combination of these and several other design choices ensures that an erlang program will not crash even if sections of a erlang program runs into errors.

Erlang is also a strongly typed language. This means that the language enforces predefined types for all variables in the language. This means does limit the power that the developer has, but it does cut back on errors and variability immensely. Since availability is essential to the language, a large amount of variability is a bad thing.

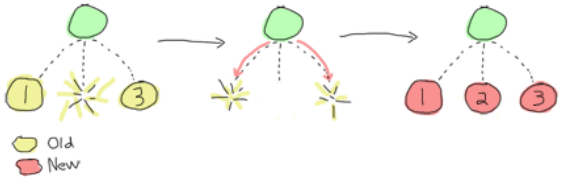
## Processes

A process in erlang can be created through the spawn command. Each process is given a type, a set of instructions and a PID. PID stands for process identifier. It is a unique ID that is assigned to all active processes. There are two types of processes. They are worker and supervisor. Worker processes are processes that do most of the actual coding work. These processes are children of the second type of process. The supervisor process manages a set of other processes including other supervisor processes. Similar to Java's inheritance hierarchy, all processes can only have one parent while supervisor processes can have multiple children. This means that worker processes can not have children.

There are many ways a supervisor process can manage its children processes. For example, one of the techniques is one for one. This means whenever a child process fails, it is restarted. This case is when said process is independent from it's sibling processes. This way said child process can lose its state and not impact its sibling processes.




Another technique that supervisor processes have is all for one. In this situation. All children process heavily rely on each other. An example of this is processes for bank transactions. Withdrawing from a bank account requires the correct amount of money to be subtracted from the bank balance while spitting out the correct amount of money to the customers. Many aspects of the transaction rely on the others to function correctly. Another example of this would be interest rates. Interest should be processed last as the outcome is different depending on the order of the translations. In the supervisor's process's perspective, one terminates all children processes and restarts all of them if one of the child processes has a problem. This is ensuring the correctness of the functions.



Supervisor processes can be configured using the keywords startFunc to start a child processes, restart to restart a existing child process, and shutdown to terminate a single or series of existing children processes. The shutdown command has three types. They are permanent, temporary, and transient. Each child processes have a series of attributes associated with it. Two important ones are childID and type. A childID is a way for a supervisor process to identify which child they are looking at. The type attribute tells the supervisor process what type of process its child is. It can be a worker or another supervisor process.

**Types of Supervision**

Erlang implements two types of supervision. Static and Dynamic. In the above paragraphs, all stated functions are in the context of static supervision. Dynamic supervision is used much more as most processes are always active and in motion. In addition to all the functions that static supervision has, dynamic supervision has the ability to view the specs of the children processes. This is very useful for debugging purposes as the children processes self-logs what went wrong and the supervisor's processes can see this in real time. This makes the system much more resilient to downtime as the system as a whole is self-managing.

**Hot Code Reloading**

Hot code reloading is the process of swapping out old or failed processes with new replacement processes. Since processes hierarchies are independent from one another, a process failing will not shut down the whole system, and failed processes can be recovered rather quickly. This management system is generally used in practice for other languages. In most cases they are called hot fixes. Although this is a last resort for most languages due to its probability for errors. Erlang's complex fault tolerance system eliminates this issue as the design of the language is built for resilience. This is good thing as the language benefits from the fast debug time while still upholding the level of risk to a manageable level.

**OTP**

Another reason why the reliability of erlang is so high is because of the OTP framework. OTP stands for Open Telecom Platform. It is a set of useful libraries that ensures processes are spawned and initialized safely. The OTP framework allows companies using erlang to set up

servers in a safe and resilient way. There are many models one can implement with OTP. All of them stems form the generic model. There are more specific models like the server client model, but the main design choice for OTP is abstraction. Every process or series of processes should be as generic as possible and strive to do one thing really well. This is very similar to the philosophy of Linux. Chaining small short programming's that do one thing really well to solve more complex problems.

### Asynchronous Message Passing

The way in which different processes talk to each other is through message passing. There are two types of message passing. Asynchronous and Synchronous. Asynchronous message passing does not care if the receiver has processed the message or not. It just confirms that it arrived to its destination location and moves on to the next message to be passed. While Synchronous message passing confirms that the receiver has processed the message before moving on. There are many pros and cons to these two types of message passing. For one asynchronous is much faster but is less reliant as we do not know if the receiver has processed the message or not. Since erlang's other design aspects are to resilient, it chose to use asynchronous message passing over synchronous. This gives it the speed it needs for telecom communications, while still being able to fault check if there are any errors.

### Node Clustering

All processes on a system made from erlang has a series of nodes running the processes. A single node can run hundreds of thousands of processes. However, to optimize this even further, erlang has clustered nodes with processes that are similar or rely on each other. By clustering these nodes and including indexes, the system can pass messages even quicker then non clustered nodes. This process is called Location Transparency. Like processes nodes can be shutdown and restarted again. So, if one node becomes faulty, it is quickly restarted and recovers the last stable state it was in.

### Closing

In this paper I've researched much of the original syntax and much of the design of the language. Although erlang is an amazing language for numbers manipulation, data retrieval, and uptime reliability, it struggles when it comes to string manipulation. This language would not be a good fit if one were to do anything graphically intensive. The

best application would be a bank system or transaction system. Any environment that needs a large amount of processes all being run the same time while ensuring correctness, speed, and availability.

## Future Work

For future work, I would like to dive deeper into the relationship of the server client model. Event handling was also another area I wanted to study more. Although erlang is one of the most resilient language ever made, it does string manipulation pretty poorly. If I can understand the language on a deeper level, maybe I can create a module that makes string manipulations easier and more flexible.

# Annotated Bibliography

Bibliography Banas, Derek. “Erlang Tutorial.” YouTube, 1 Apr. 2017,

www.youtube.com/watch?v=IEhwc2q1zG4

“Youtube tutorial on some of the basic syntax of erlang along with some sample program to practice”

Erlang Documentation Search, Ericsson AB,

https://erlang.org/doc/search/

“Official API documentation for the language erlang developed for Ericsson Telecommunications”

Learn You Some Erlang for Great Good!, Learnyousomeerlang.com. (2019),

https://learnyousomeerlang.com/content

“A in depth guide to learning erlang with really nice examples and explanations.”

“Open Telecom Platform.” Wikipedia, Wikimedia Foundation, 16 May 2019,

https://en.wikipedia.org/wiki/Open_Telecom_Platform.

“Wiki on OTP framework and how it is incorporated in to erlang”

Wireless News, RCR. “What Is Erlang and Why Is It Essential to Telecom? - Coders Episode 32.” YouTube,

18 Nov. 2015, www.youtube.com/watch?v=4elY_RuyBhQ

“Describes the history of erlang and why it was first created for telecom communications”