Assessed Coursework Report
Advanced Operating Systems

Willian de Oliveira Barreiros Junior
Matric Number: 2105514
guns945@gmail.com

March 12, 2014

Introduction

This report regards the evaluation of the features provided by Erlang programming language by implementing a small multiprocess temperature system.

This report is divided into four parts, a brief introduction on Erlang's background and basic features, the proposed case study (the temperature multiprocess program), the findings on Erlang's qualities and shortcomings given the experience on developing the temperature program, and finally, some final remarks concluding this work.

The source code from this coursework can also be found at `https://github.com/WillianJunior/AdvancedOSExercice`

The Erlang Programming Language

First and foremost, Erlang was chosen given that it already had the message passing system built in its core and support a programming paradigm that uses *share nothing* processes with message passing. Also, it was rather helpful that Erlang is a functional programming language, which, in my experience, results in shorter and more concise code.

Erlang is, as stated before, a functional programming language focused on concurrency and reliability. For being a functional language, parallelism can be easily and safely achieved.

Erlang uses a simple message passing system as the only interprocess communication tool. This can be very useful since it's easier to reason large an complex systems as a set of actors communicating between each other using only messages instead of any kind of shared memory emulation, semaphores and locks.

Erlang proposes a concurrent oriented programming paradigm. This enables scalability and safety given that the processes are independent from each other. This also makes easy to update the system while it's still running by implementing hot code loaders.

One controversial feature is that Erlang is dynamically typed. When comparing with a statically typed language (e.g. Haskell), it's possible to see the shortcomings. It's much harder to have runtime errors with statically typed languages. This, however, is not a problem given the "let it crash" philosophy of Erlang. This approach is taken given that its impossible to predict and treat every possible error (e.g. hardware errors) and that one process should not interfere with another.

Case Study Development

As a case study, a simple multiprocess system was developed. This system had as its main goals extensibility and error tolerance.

The system itself consists on a clock process that notify a couple o sensors processes to display their readings. This was done by requesting the conversion to the converter process through a message. The converter in turn sends a response message to the requester with the result of the conversion. Finally, the sensors send the converted temperature to the display.

Extensibility was achieved by enabling load of new conversion functions and addition of new sensors on runtime. This was easily done with the aid of *atoms*.

Since the processes have a*share nothing* aspect, there isn't much that need to be done in order to achieve fault tolerance. The only extension done was to ensure that if a process fails for any reason, except

normal end of execution or a *shutdown* signal, it will be restarted. This is performed by wrapping each process with a supervisor process and using *atoms* to reference the processes.

Case Study Results

The first positive remark when starting to work with Erlang is that the installation process (on Linux at least) is extremely trivial (just as with Haskell), with the installation package coming with everything you need.

When it comes to the learning, the closest programming language I've worked before was Haskell. Consequently the functional portion of the language was trivial to learn, taking only 15 to 30 minutes to review the concepts. The first barrier encountered on the learning process was the syntax which was a bit awkward when comparing with the most mainstream languages. After that, the next step would be using the message passing system and the multiprocess system. Those were also trivial, resulting in a first draft of the case study system done within the hour, including all message passing protocol and processes.

Given that the first draft had less than 50 lines of code, other improvements were thought in order to carry on with the learning process. After some reading, a message passing pattern arose: encapsulating the message passing into a function of the receiving module. This would abstract the actual message protocol from the sender, resulting in a more reusable code.

The next improvement regards error tolerance. One of the most expressive characteristics of Erlang is the *let it fail* philosophy. This only works given the ease to implement supervisors. A simple supervisor was implemented for each process, restarting it whenever it fails. The implementation was trivial, but also incomplete. A better approach would be to extract the supervisor code into a module and make a tree of supervisors, implementing a protocol of contingencies for different fails on different levels. This was not implemented because it would increase (even more) the complexity of the case study. An important aspect to highlight is the use of *atoms* to make the processes management transparent to the message passing system. E.g. let's say a process die and it's restarted. This process now have a different *Pid*. This is a problem given that in order to send a message you need the receiver's *Pid*. However, instead of having to implement a system to keep track of all *Pid*, *atoms* were used as a kind of self-updating constant (in terms of other processes) *Pid*. The resulting code sized around 90 lines.

The final improvement was to enable loading of conversion functions and creation of new sensors on runtime. For this amend it was necessary knowledge on how to hold state on a process, high-order functions and even more *atoms*. For the loading of functions, the converter process could also receive a *new_function* message, on which it would add the new function to its current state. The function would come along a identifier (for lookup purposes) in the form of an *atom*. The usage of *atoms* proved to be very useful since it doesn't generate a large overhead on the messages (since its implemented as a number) and also makes the code easy to interpret. The dynamic creation of new sensors functionality was also not that difficult given the use of *atoms* as the references to the processes. The final code is around 200 lines long, being almost half of it comments and the replicated supervisor.

Conclusion

Erlang proved itself to be simple enough to learn the basics. The learning speed is somewhat quick given what you can accomplish comparing what you would need to learn on other more conventional languages (e.g. java or C). The size and conciseness of the code are definitely better when comparing to the mainstream languages. The full development process, including studying and testing, took no longer that 8 hours, which is relatively a very small amount of time to actually learn a new programming language.

Some advantages found were the facility to use the message passing built-in functions and to program conforming the paradigm of *share nothing* processes. Also the fact that Erlang is functional helped a lot. One of the most interesting aspects found was *atoms*, which proved to be very powerful.

Although Erlang has its advantages the fact that it is dynamically typed can be a source of several bugs for newbie programmers. One other aspect that may seem unimportant compared to everything else are the anonymous functions. They seem quite unnatural when compared to Haskell. The syntax also is rather unusual and can take some time to get used to it.

Given the abovementioned reasons, Erlang is a viable choice for any kind of programmer that need to build a scalable and safe distributed system.