Towards an Abstraction for Remote Evaluation in Erlang

Adrian Francalanza[1] and Tyron Zerafa[1]

Department of Computer Science, University of Malta
{adrian.francalanza | tzer0001}@um.edu.mt

1 Introduction

Erlang is an industry-standard cross-platform functional programming language and runtime system (ERTS) intended for the development of concurrent and distributed systems[1]. An Erlang system consists of a number of processes [2] (actors) executing concurrently across a number of nodes/ locations. These processes interact with each other (mainly) through asynchronous messaging and are capable of creating (spawning) further processes, either locally or at remote locations.

2 Process Creation

In Erlang, a functional object (variable/ value of type function) is a data type consisting of a set of attributes along a pointer to the function's executable code. An Erlang process, which can only execute functional objects, starts by looking for the referenced executable code inside the underlying node's directories and loads it into the code server. In a distributed setting, a process may request the execution of a functional object on a remote node; however, the executing (remote) location is not allowed to retrieve code from the original (requesting) node. Erlang assumes that during remote process creation the required code would be present on the executing node prior to execution initialization. A process would fail if it attempts to execute a function whose code *is not defined* on its (executing) node; alternatively, the end result would be different if the executing node holds executable code that *differs* from that residing at the functional object declaration location [1, 5].

Erlang's remote process creation resembles the remote evaluation paradigm in which nodes demand the execution of code on remote locations through requests that contain the required code. We propose a solution that facilitates remote evaluation, *i.e.,* transfers the (missing) required code during a remote process creation, while adhering to Erlang's semantics and best practices. In the rest of this document, § 3 explains why existing Erlang support is not adequate to attain code migration while § 4 discusses considerations that arise in the design of our remote evaluation mechanism. § 5 concludes this document.

3 Inadequacies of the Existing Support

Erlang's standard serialisation mechanism encodes data/ values that needs to be transmitted over a network into an intermediate representation known as the External Term Format (ETF). The intermediary representation of Erlang functions is composed of a number of attributes which include a symbolic link to the respective module's binary file (called a BEAM file) containing the function's code. Upon a remote execution request, the respective function ETF (with its symbolic references to the BEAM file) is sent to the remote node, assuming that the referenced BEAM file is present.

In order to overcome this limitation attributed to the serialization mechanism, Erlang provides two mechanisms that facilitate the dynamic linking/ loading of code modules/ binaries inside remote ERTSs. The simplest mechanism broadcasts whole modules onto entire Erlang clusters (a set of connected nodes) resulting in huge bandwidth usage spikes and superfluous memory overheads. On the other hand, the second mechanism transmits portable code resulting in a less-expensive finer-grained control over what's loaded where.

At first these approaches may seem attractive, however, after a deeper analysis it becomes evident that these are far from complete. For starters, they lack any form of *dependency analysis* which has to be handled explicitly by end developers to ensure that all the required code is transferred. Furthermore, these mechanisms do not take into consideration the possibility of different code versions which are so critical in real life development environments. All these problems, coupled with the possibility to remotely execute higher-order functions (functions that accepts other functions as argument) over remote nodes require a huge development effort from the end developers and necessitates a proper framework that manages Erlang code in such a heterogeneous distributed environment.

4 Considerations for Proposed Solutions

A solution to handle code management during remote evaluation can be programmed; however, as described in the previous section increases the responsibility and effort on the part of application developers, who would need to contend with the difficulties discussed. We propose a solution that abstracts over these difficulties and automates the functionality for code-dependency analysis, code correspondence and code migration, in line with other proposed fine-grained code mobility approaches [3, 4]. This automation should aspire to mimic the behaviour of a local process execution in the presence of missing code using the least possible bandwidth and storage overheads.

The solution would need to determine a feasible unit of code migration to adopt. More specifically, whereas the unit of process creation is a functional object, the ERTS standard unit of code loading is the Erlang module. Issues may arise when, in order to remote execute a particular function whose code is not present at the destination node, an arbitrarily large module (containing the required function) would need to be migrated and loaded; the problem could be more acute in the case of transitive function dependencies.

Conventions for how to migrate code would also need to be established. At one extreme, the solution may decide to migrate the missing code *eagerly* in one phase, once the missing dependencies are *statically* determined. Alternatively, code migration may happen incrementally in *lazy* fashion, whereby only the immediately execution functions are sent. The latter approach is in general more complex and may incur more bandwidth overhead. However it is able to use runtime information relating to code dependencies, *e.g.,* code branches taken by the spawned remote actor, so as to minimise the code that is migrated—the function dependencies in branches that are not taken need not be migrated. The proposed solution may even decide to adopt a hybrid model of code migration, that adapts according to the requirements of the nodes and that of the underlying network.

Ideally, the solution should also embrace the realities of distributed computing and adhere to the philosophy of the host language, *i.e.,* Erlang. Failures such as nodes crashing and flaky node connections should not be ruled out by the proposed solution, which should in turn affect the underlying architecture and operations. For instance, in order to withstand a degree of failure, the proposed solution should be as decentralised as possible. Moreover, once the missing code dependencies are determined, the code need not be migrated from the source node; instead it may be obtained from another node having a faster or more reliable connection to the remote node where the processes is to be created.

5 Conclusion

We have argued why that the existing mechanisms for remote evaluations in Erlang is inadequate for a distributed setting with heterogeneous codebases. We then outlined possible requirements to consider for a language extension that addresses these shortcomings. We are currently working on a prototype that takes these suggestions into account.

References

1. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly (2009)
2. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI. pp. 235–245. Morgan Kaufmann (1973)
3. Jul, E., Levy, H., Hutchinson, N., Black, A.: Fine-grained Mobility in the Emerald System. ACM Trans. Comput. Syst. 6(1), 109–133 (Feb 1988)
4. Mascolo, C., Picco, G.P., Roman, G.C.: A fine-grained model for code mobility. SIGSOFT Softw. Eng. Notes 24(6), 39–56 (Oct 1999)
5. Wikström, C.: Distributed Programming in Erlang. In: Symp. on Parallel Symbolic Computation. pp. 412–421 (1994)