Towards an Abstraction for Remote Evaluation in Erlang

Adrian Francalanza
CS, ICT, University of Malta
adrian.francalanza@um.edu.mt

Tyron Zerafa
CS, ICT, University of Malta
tzer0001@um.edu.mt

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms Distributed Programming, Message-Passing Concurrency

Keywords Distributed Erlang, Code Migration.

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 actors [3] (processes) executing concurrently across a number of nodes. These actors interact with one another (mainly) through asynchronous messaging and are also capable of spawning further actors, either locally or at a remote node.

2. Local and Remote Process Spawn

In Erlang, process spawning is programmed using the spawn built-in function (BIF), the basic version of which accepts a module, a function (contained in that module) and a list of values as its arguments, and creates a *new* process at the *local* node executing the function applied to the values in the list; the BIF returns the process identifier (PID) of the new process to the spawning actor. Erlang also provides a *variant* of this BIF, used to program *remote evaluations* [2]; an additional node name argument is passed to the BIF variant, specifying the host node where to spawn the process.

The remote spawn BIF variant aims to *emulate* the functionality of local spawning, by abstracting away from the additional tasks required to perform the remote process launch [1, 6]. However, this emulation relies on an important assumption: the source node and the remote node must *share the same codebase*, *i.e.,* the same set of modules and function definitions. When this is not the case, remote spawn execution may differ: if an actor attempts to spawn a function at a remote node where the function *is not defined*, the remote spawn fails; alternatively, if the remote node holds a (spawned) function definition that *differs* from that at the local node, the computation may be different as well. Similar, but more intricate, discrepancies between local and remote spawning arise when the code for the spawned function depends on other functions, possibly from other modules.

There are a number of circumstances where code homogeneity across nodes is too strong of an assumption. Reasons for this range from induced code updates (which may be arbitrarily frequent), differing node hardware (which may require dedicated software or may otherwise restrict the codebase, due to resource limitations such as memory size) and node-specific security constraints (*e.g.,* one node may trust the newest version of a code package, whereas another would prefer to work with a previous, more stable, version.) Whenever discrepancies exist, codebase homogeneity may be infeasible (or even impossible) to attain. For instance, keeping track of persistent codebase changes is a complicated and expensive task, whereas individual node trust policies may explicitly prohibit such homogeneity.

We argue towards a solution that adheres to the original aims of the remote spawn mechanisms, *i.e.,* emulating the behaviour of local spawn, in a setting with codebase *heterogeneity* across nodes. We also explain why existing Erlang support is not adequate in order to attain this and discuss the main difficulties in attaining the aforementioned aims.

3. Inadequacies of the Existing Support

Erlang provides lower-level mechanisms for dynamic module loading inside a remote ERTS. Dynamic remote code loading can be achieved through mechanisms such as c:nl/1, which broadcasts code-migration and update to all participating nodes. However, this mechanism is too coarse for our needs, since code migration is forced upon nodes that have nothing to do with the remote spawn that needs to be carried out.

A more accurate approach would need to resort to the BIF load_binary/3 (together with some mechanism for remote evaluation), which only loads the necessary missing code at the destination node where the remote spawn is to launch the remote process. This approach is clearly less expensive, since it only involves code migration between two nodes. However, it sits at a lower level of abstraction than that of c:nl/1 and requires the explicit handling of code binaries directly as a data.

The two approaches just discussed may be inadequate still in the general case of remote spawns involving mismatching codebases. For starters, neither of these mechanisms performs any *dependency analysis* on the code to be spawned. More specifically, in order to emulate the behaviour of a local spawn, one cannot simply check whether

Erlang '13, September 28, 2013, Boston, MA, USA.

ACM 978-1-4503-2385-7/13/09.
http://dx.doi.org/10.1145/2505305.2505316

the function to be spawned is present at the destination node, but needs also to check that the functions that the spawned function uses (which could potentially be dispersed across multiple modules and could transitively depend on other functions themselves) are also present at the destination node. Such a dependency code analysis may be explicitly determined through existing Erlang BIFs, such as those found in module xref. However this approach is fairly low level, thus prone to programming errors.

There are further complications associated with mismatching codebases. For instance, if the binaries relating to the function to be remotely spawned are already present at the remote node, one must then ensure that they correspond to the compiled binaries at the local node. Ensuring such code correspondence is non-trivial with the current level of support offered by the Erlang platform. But even if determining whether the respective binaries correspond or not was possible, it would still leave the programmer with a conundrum, in cases where binaries differ: on the one hand, executing the remote spawn with different binaries at the remote node may yield unexpected results; on the other, loading the local version of the binaries at the remote node would overwrite the existing versions, which may in turn corrupt current computation at the remote node.

Since Erlang supports higher-order functions, the difficulties discussed above are not limited to the function being spawned, but also the arguments that are passed to the function when spawned, which may be functions themselves. Erlang's standard serialisation mechanism encodes data into an intermediate representation known as the External Term Format (ETF); in the case of functions, these values are encoded as a data type composed of a number of attributes including a symbolic link to the respective module's binary file (called a BEAM file) containing the function's code. When a remote spawn is executed with function arguments, only the respective function ETF (with its symbolic references to the BEAM file) is sent to the remote node. As before, discrepancies between node codebases may cause this link to be broken at the remote node which again causes the remote spawn to behave differently from it local counterpart. Stated otherwise, similar code-management mechanisms need to be applied for functions passed as parameters as well.

4. Considerations for Proposed Solutions

Although solutions for handling remote spawning in a heterogeneous codebase setting can be programmed, they increases the responsibility and effort on the part of application developer, which would need to contend with the difficulties discussed in § 3, but also low-level implementation concerns such as the possibility of name-structure clashes across codebases. Instead, we advocate for a solution that abstracts over these difficulties and automates the functionality for code-dependency analysis, code correspondence and code migration, in line with the fine-grain code mobility approaches proposed in [4, 5]. Such an automation should aspire to mimic the behaviour of a local spawn 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 spawning is the Erlang function, the ERTS standard unit of code loading is the Erlang module. Issues may arise when, in order to remote spawn a particular function whose code is not present at the destination node, an arbitrarily large module (containing the spawned 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.

The proposed solution should also take into consideration the different requirements of the individual nodes. For instance a remote node may prohibit migrated code of a certain size, originating from certain untrusted nodes, or else lacking certain security certificates. In a setting where multiple codebases are handled, a remote node may also require that certain code dependencies use the local version of the codebase, as opposed to that of the originating node; such restrictions are particularly relevant to dependencies involving standard Erlang code libraries.

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 actor is to be spawned.

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] F. Cesarini and S. Thompson. *Erlang Programming.* O'Reilly, 2009.

[2] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Softw. Eng.*, 24(5):342 –361, 1998.

[3] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245. Morgan Kaufmann, 1973.

[4] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.*, 6(1):109–133, Feb. 1988.

[5] C. Mascolo, G. P. Picco, and G.-C. Roman. A fine-grained model for code mobility. *SIGSOFT Softw. Eng. Notes*, 24(6): 39–56, Oct. 1999.

[6] C. Wikström. Distributed Programming in Erlang. In *Symp. on Parallel Symbolic Computation*, pages 412–421, 1994.