Code Management Automation for Erlang Remote Actors

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

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

Abstract

Distributed Erlang provides mechanisms for spawning actors remotely through its remote spawn BIF. However, for remote spawn to function properly, the node hosting the spawned actor must share the same codebase as that of the node launching the actor. This assumption turns out to be too strong for various distributed settings. We propose a higher-level framework for the remote spawn of side-effect free actors, abstracting from and automating codebase migration and management.

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

General Terms Distributed Programming, Actors

Keywords Distributed Erlang, Code Migration, Load-Balancing.

1. Introduction

Full location transparency is often unattainable in distributed settings [9], due to either physical constraints such as communication latencies and partial failures (revealing the underlying distributed structure), or administrative and security concerns (precluding unfettered distributed computation). Instead, various distributed programming technologies choose to include locations as part of the computational model; to facilitate programming, they also provide constructs for describing computation across these locations, automating solutions that aim to abstract (as much as possible) from the differences between local and distributed operations. This is true for actor-based abstractions [15] of distributed systems—a natural fit for the inherent concurrency found in distributed settings. Languages and frameworks such as [10, 14, 16, 27] distribute actors (processes) across a number of locations (nodes): these actors are usually allowed to communicate seamlessly with one another, both locally and remotely, through common interfaces that abstract away from the intricacies of remote communication.

In a distributed setting, actor *locality* affects computation in a variety of ways. For instance, frequently communicating actors are best co-located at the same node, so as to *reduce communication overhead*, whereas replicated actors are best dispersed across independently-failing locations so as to increase *fault tolerance*. Alternatively, actors *accessing an irreplicable resource e.g.,* a database, should preferably be co-located next to the resource, whereas large systems consisting of numerous and/or computationally-intensive actors may need to distribute them evenly across locations for *load-balancing* purposes.

Actor-based distributed technologies usually let the programmer control the locality of an actor, since its optimality may best be determined at runtime (*e.g.,* depending on which location has the least load), and may vary over the course of an actor's execution (*e.g.,* accessing two resources in sequence, located at different nodes). Some languages, *e.g.,* [19, 27], support actor migration permitting an actor to change its locality. Others provide mechanisms for mimicking this behaviour through *remote evaluations* [13]: instead of migrating itself, the actor *remotely spawns* a new actor at the location where it intends to continue executing, *delegating* the execution to the newly spawned actor (communicating remotely with it when needed). This option is attractive for a number of reasons (*e.g.,* it circumvents issues relating to the synchronisation, serialisation and transfer of actor states), and is adopted by numerous industry strength technologies, *e.g.,* [16, 18].

Erlang [1, 10, 20], a cross-platform programming language and runtime system (ERTS) intended for the development of enterprise distributed systems, is one such example. It provides a spawn built-in function (BIF), the basic version of which accepts a module, a function (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 of the newly process to the spawning actor; it also provides a *variant* of this BIF that can be used for remote evaluation purposes: an *additional node name* argument is passed, specifying the host node where the new process is spawned. In order to facilitate programming, the remote BIF variant aims to *emulate* the functionality of Erlang local spawning, by abstracting away from the additional tasks required to perform the remote process launch [10, 28].

Crucially, remote spawn is able to emulate local spawn *only* when an important condition holds: the source node and the service node must *share the same codebase*, *i.e.,* the set of modules and function definitions. When this is not the case, remote spawn execution may differ from its local counterpart. For instance, 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 definition for the spawned function that *differs* from the definition at the spawning node, the eventual outcome may be different still—similar, but more intricate, discrepancies between local and remote spawning arise when the spawned function depends on other function definitions, possibly from other modules.

Occasionally, code homogeneity across nodes can either be infeasible or even undesirable. Reasons range from local code updates (which may be arbitrarily frequent), different node hardware (which may require dedicated software or impose codebase restrictions due to resource limitations such as memory size) and local security constraints (*e.g.,* one node trusts the newest code package version, whereas another prefers an earlier, more stable, version).

Erlang provides lower-level mechanisms for dynamic module loading inside a remote ERTS, which can be used to program so-

lutions for issues associated with heterogeneous codebases. However, this increases the responsibility and effort on the part of the programmer, who needs to contend with lower level implementation concerns such as the possibility of codebase name clashes. There are also efficiency concerns to be considered. For instance, the programmer cannot blindly upload the entire local codebase at a remote node, but instead needs to identify the *least* amount of (missing) code to load remotely so as to allow remote spawn to emulate its local counterpart. Determining this codebase subset would typically entail a dependency analysis of the code to be evaluated remotely. In addition, the developer would also need to establish conventions such as: (a) whether to migrate missing code dependencies *eagerly* in one phase, or else incrementally when needed (*lazily*), since the branching structure of the spawned computation may not require all dependencies at runtime; or (b) whether to use more standard units of migration such as modules, or else finer ones such as function closures. There are also other disadvantages associated with burdening programmers with code migration management. For instance, individual programmers may adopt conflicting decisions *wrt.* the design alternatives discussed above, which further increases the complications associated with remote spawning. Finally, there are also security issues that may, on the one hand, restrict what computation the source node is allowed to delegate, and on the other, require the destination node to observe.

In this paper we propose a solution that abstracts over the difficulties associated with Erlang remote spawn in the presence of heterogeneous codebases, automating the functionality for code-dependency analysis, function definition correspondence and code migration, in line with the fine-grain code mobility approaches proposed in [17, 21]. Attuned to the constraint of distributed computing, our code-management solution is decentralised and able to tolerate degrees of failures while using low bandwidth and storage overheads. Since it is unclear to what extent security issues are to be abstracted away from the application developer, we only provide rudimentary mechanisms for enforcing security aspects in our proposed platform, focussing instead on aspects relating to correctness and efficiency.

In the rest of the paper, we discuss a use-case in Sec. 2 whereas Sec. 3 presents existing Erlang infrastructure and mechanisms confining the design space. Sec. 4 describes out proposed solution and Sec. 5 evaluates this solution *wrt.* the use-case presented in Sec. 2. Related work is discussed in Sec. 6 and Sec. 7 concludes.

2. Case Study

We consider an illustrating scenario where Erlang nodes offer an *execution-platform-as-a-service* to other nodes for *load-balancing* purposes. Actors at a *client node* are able to delegate computationally expensive tasks to *service nodes* offering this service, by remotely spawning an actor executing the expensive computation at these nodes. Crucially, service nodes cannot be expected to hold homogeneous codebases, *wrt.* the client nodes or other service nodes.

```
1  -module(cpx_math).
2  -export([fac/1,fib/1,...]).
3
4  fac(0) -> 1;
5  fac(X) -> bsc_math:mlt(X,fac(X - 1)).
6
7  fib(0) -> ...
```

Listing 1: Complex Math Module containing a Factorial Function

Consider the module cpx_math (complex math) defined in List. 1, exporting a factorial function, fac/1, amongst others. Its implementation relies on a multiplication function, mlt/2, defined in another module containing basic functions, bsc_math (see List. 2).

```
1  -module(bsc_math).
2  -export([mlt/2,dvd/2]).
3
4  mlt(_,0) -> 0;
5  mlt(X,Y) -> add(X,mlt(X,Y-1)).
6
7  dvd(X,Y) -> if X > Y -> 0;
8                 true  -> 1+dvd(X,sub(Y,X))
9              end.
10
11 add(X,Y) -> ...
12
13 sub(X,Y) -> ...
```

Listing 2: A Basic Maths Module

In Erlang, we may remotely evaluate the factorial of a large value, *e.g.*, 42, on a service node, nodeL, through the code in List. 3: line 4 remotely spawns on nodeL the higher-order function sndRes, with the arguments to execute fac/1 from cpx_math with value 42, sending back the result to the caller process, self().

```
1  sndRes(Md,Fn,Ar,Id) ->
2          Id!{result,(apply(Md,Fn,[Ar]))},
3  ...
4  Id = spawn(nodeL,?MODULE,sndRes,
5                [cpx_math,fac,42,self()]),
6  receive
7     {result,Res} -> Res
8  end.
```

Listing 3: Remote Evalutation of fac/1

We can go a step further, and construct higher-order abstractions for remote evaluations (akin to mobility skeletons [5, 11]) over our execution-platform-as-a-service framework, where service nodes become client nodes themselves. The code in List. 4 implements a distributed version of the lists:foldl/3 function: apart from the usual three parameters representing the function, Fn, the accumulator, Ac, and the list of values over which to perform the folding, Vs, the function foldlR/5 takes a list of nodes, [Nd|Ns], and a continuation function, Cn, *resp.* denoting the service nodes that may be used to delegate the folding computation, and the operation to be performed on the final value of the computation—typically, the continuation sends the result back to the originating actor.

```
1  foldlR(Fn,Cn,Ac,Vs,[Nd|Ns]) ->
2    spawn(Nd,foldlRE,[Fn,Cn,Ac,Vs,Ns++[Nd]]).
3
4  foldlRE(Fn,Cn,Ac,[V|Vs],[Nd|Ns]) ->
5     A = Fn(V,Ac),
6     spawn(Nd,foldlRE,[Fn,Cn,A,Vs,Ns++[Nd]]);
7  foldlRE(_,Cn,Ac,[],_) -> Cn(Ac).
```

Listing 4: Implementation of foldlR/5

The distributed foldl of List. 4 assumes that the folding operation is expensive, and thus distributes the individual folding operations over the service nodes available so as to share the computation burden: *e.g.*, if foldlR/5 is applied for some function $f()$ and initial accumulator a_1 over the list of values [42,53,64] with service nodes [nodeL,nodeK,nodeM] it would execute the distributed code depicted in Fig. 1, distributing the three applications of $f()$ with the accumulating arguments over the three service nodes.

In List. 4, foldlR/5 acts as a wrapper function for foldlRE/5 after launching the initial remote spawn: foldlRE/5 applies the

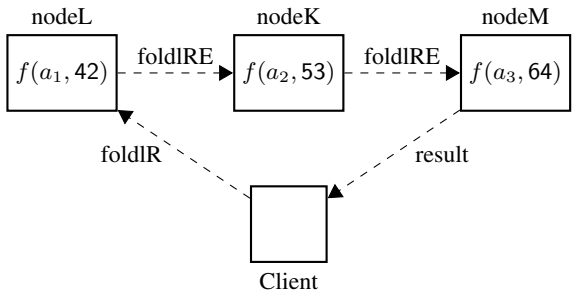

Figure 1: An execution of foldlR applied to $f()$ with a_1 and value list [42,53,64] over service nodes [nodeL,nodeK,nodeM] with a continuation sending the final answer to the client, where $a_2 = f(a_1, 42)$ and $a_3 = f(a_2, 53)$

function, Fn, once on the first value, V, and accumulator, Ac, obtaining a recomputed accumulator, A, (line 5); it then recursively spawns the next iteration with the remaining values, Vs, and the new accumulator, A, on the next service node (line 6) until the value list is exhausted, at which point the continuation function, Cn, is applied to the accumulated value (line 7). The functions foldlR/5 and foldlRE/5 rotate amongst the service nodes when the number of foldl operations exceed the number of service nodes, *i.e.,* they are appended back to the node list as Ns++[Nd] on lines 2 and 6.

```
1  Id = self(),
2  Cn = fun(X) -> Id!{result,X} end,
3
4  Fn = fun(X,Ac) -> cpx_math:fac(X)*Ac end,
5
6  Vs = [42,53,64],
7  Ns = [nodeL,nodeK,nodeM],
8  foldlR(Fn,Cn,1,Vs,Ns),
9  receive
10   {result,Res} -> Res
11 end.
```

Listing 5: Using foldlR/5 with cpx_math:fac/1

We can use foldlR/5 to compute the product of the factorials of the values 42,53 and 64 using the code in List. 5. Lines 1 and 2 define the continuation, Cn, as an *anonymous* function [1, 10]. The folding function that incrementally calculates the product of the factorials, Fn, is also defined as an anonymous function (line 4), using functions fac/1 from module cpx_math; the accumulator is initialised to 1 when foldlR/5 is called (line 8).

2.1 Issues relating to Remote Evaluation

There are a number of correctness and efficiency considerations to take into account when executing the distributed computations discussed in List. 3 and List. 5. For instance, for the remote spawn in List. 3 to run as expected, the ERTS at nodeL must have the code for module cpx_math with function fac/1 from List. 1 loaded,[1] even if one assumes that all service nodes come equipped with the generic wrapper function sndRes/4. Whenever a module is *not* loaded at the respective service node, the developer needs to load it explicitly; moreover if the respective code to be loaded is not present at the node, it needs to be migrated as well.

In the proposed framework, standard Erlang mechanisms for dynamic remote code loading such as c:nl/1 are too coarse, because they broadcast code-migration and update to *all* participating nodes. Instead, a developer would need to resort to using the load_binary/3 BIF (together with some mechanism for remote evaluation), which would require direct handling of code binaries as data. In cases where it is acceptable for the programmer to manage code migration explicitly, mechanisms such as load_binary/3 may still be inadequate. For starters, these mechanisms do not perform any code *dependency analysis*. For instance, in the case of List. 3, the developer would also need to ensure that bsc_math:mlt/2, used in the implementation of cpx_math:fac/1, is also loaded at the ERTS at nodeL; such dependencies need, in turn, to be explicitly determined through BIFs such as those found in module xref (further increasing the burden on the programmer).

In the case of List. 5, the problem is even more acute, since dependency analysis would need to be repeated for *every* service node used in the remote evaluation. This introduces further complications: since, ideally, service nodes only load the code that is required for servicing remote evaluations, the client node in List. 5 could not pre-load service nodes with the missing modules, because the number of service nodes used in the remote evaluation is dependent on the computation—in fact, this generally cannot be determined statically. This means that codebase correspondence checks can only be carried out *dynamically*, and the code in List. 4 (describing higher-order remote-evaluation management) would need to be cluttered with functionality for codebase management.

There are other complications. For instance, if the binaries for cpx_math:fac are already loaded, they must (in some sense) *correspond to* those compiled from List. 1. Since we envisage nodeL to be a service node, it is possible that it hosts a *different* version of the module cpx_math whereby, for example, fac/1 returns the list of factors for a number instead.[2] On the one hand, executing the remote spawn of List. 3 with different binaries may not yield the expected results. On the other hand, loading the client's version of the module at nodeL may corrupt existing computation hosted at the service node. Thus service nodes need to handle *multiple* codebase versions (originating from different clients).

```
1  ...
2  foldlR(Fn,Cn,1,[42,53],[nodeL,nodeM]),
3  foldlR(Fn,Cn,1,[64,75],[nodeK,nodeM]).
```

Listing 6: Repeated calls to foldlR/5 with common service nodes

Service nodes would also need to manage these multiple codebase versions efficiently. Consider a slight variation to the remote evaluation call using foldlR/5 in List. 5, described in List. 6, where the chain of remote evaluations is performed twice, using a common service node nodeM in each chain, as depicted in Fig. 2. Ideally, the code for cpx_math:fac/1 and its dependencies should *not* be migrated and stored twice at nodeM, even though they are coming from distinct "client" nodes, *i.e.,* nodeL and nodeK, since both remote evaluations at nodeM refer to the same function definitions.

There are other considerations relating to the *efficiency* and *scalability* of code migration and loading. For instance, when loading client specific binaries, it is important to load the *least* amount of code necessary at the service nodes' ERTSs, since these typically may be servicing a large number of client nodes. In the case of bsc_math:mlt/2 of List. 2 (used by cpx_math:fac/1) loading the entire module would also load the *exported* function dvd/2 together with the *internal* function sub/2, even though these are not required for cpx_math:fac/1 to execute successfully. Similar redundancy issues arises when loading the entire module cpx_math for the sole purpose of executing fac/1: the module can be arbi-

[1] Erlang provides mechanisms to check for loaded code, such as rpc:call(nodeL, code, all_loaded, []).

[2] Since Erlang is dynamically typed, return-value mismatch is detected late.

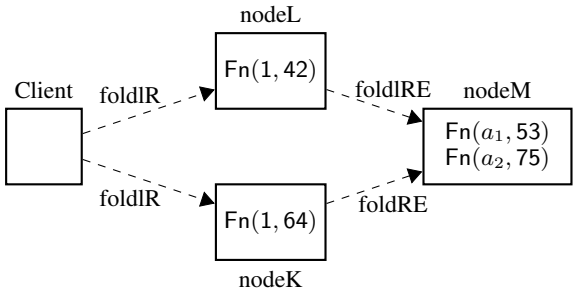

Figure 2: Repeated calls to foldlR from List. 6. with nodeM as a common service node.

trarily large in practice, containing at least redundant code for Fibonacci number generation, fib/1 (and possibly a lot more).

Apart from efficiency considerations relating to what to load, there are also aspects relating to how and when to load this code. In the case of List. 3, the function cpx_math:fac/1 (statically) depends on bsc_math:mlt/2, which in turn depends on the internal function bsc_math:add/2. One possible strategy would be to *eagerly* migrate and load all the static dependencies upfront at the service node before execution starts. There are, however, cases when cpx_math:fac/1 does not use its dependencies at runtime (*e.g.,* if cpx_math:fac/1 is called with argument 0, no dependencies from module bsc_math are used; if the same function is called with argument 1, the execution only uses bsc_math:mlt/2). In general, eager codebase migration/loading—from multiple client nodes using larger codebases with more dependencies and conditional branchings —may lead to a substantial increase in redundant codebase-management computation. An alternative strategy would be to migrate missing dependencies only when needed: although this may introduce an additional runtime cost for the client node's point of view, compared to a more eager approach (a lazy approach is bound to increase the amount of remote communication across nodes), it guarantees better space usage at the service nodes.

Then there are also restrictions imposed by the service nodes, which may attach conditions to their service. For instance a service node may prohibit migrated code of a certain size, originating from certain nodes or else lacking certain security certificates. In a setting were multiple codebases are handled, a service node may also require that certain code dependencies use the local version of the codebase, as opposed to that of the originating client node; such restrictions are particularly relevant to dependencies involving standard Erlang code libraries.

3. Erlang Architecture and Mechanisms

We overview the relevant Erlang characteristic, limitations and constraints which define the design space considered by our solution to the problems outlined in Sec. 2.1.

3.1 Naming Structure and Bindings

Erlang code is structured into (named) *modules*, *e.g.,* cpx_math and bsc_math from List. 1 and List. 2 *resp.* These contain *named functions*, *e.g.,* fac/1, mult/2 and add/2 in List. 1 and List. 2, a subset of which are exported by the module, *e.g.,* fac/1 and mult/2. Named functions may, in turn, contain *anonymous functions*, *e.g.,* the functions defined on lines 2 and 4 in List. 5. This organisation creates a *fixed lexical scoping*: since Erlang modules do not define variable bindings outside of the named function scope, (valid) named functions do not have any free variables; conversely, anonymous functions may contain free variables that are bound outside the scope of the function definition, *e.g.,* Id on line 2 in List. 5. Erlang does not support variable updates and enforces single binding for its variables *e.g.,* Id on line 2 in List. 5 cannot be bound to another value.

3.2 Higher-Order Code and Parameter-Passing

Erlang supports higher-order code, whereby functions are first-class citizens that may be passed as data to other functions. For efficiency reasons, when a function is passed as a parameter, only a reference to its respective implementation is passed (encapsulated within a functional object). An *external object* is created whenever a *named function* is assigned to a variable or passed as an argument to another function. List. 7 presents the external object for cpx_math:fac/1 of List. 1. During its execution, the ERTS loads and executes the most recent version of the referenced function.

```
1 [{module, cpx_math},
2  {name, fac},
3  {arity, 1},
4  {env, []},
5  {type, external}]
```

Listing 7: External Object erlang:fun_info(fun cpx_math:fac/1).

Anonymous functions can also be assigned to variables producing a *local functional object* that stores the specific version of the module (calculated from the compiled code) in which it is defined; see List. 8 for the anonymous function Cn of List. 5. Local functional objects also store function *referencing environments* describing the free variables bindings (line 10 in List. 8).

```
1  [{pid, <0.31.0>},
2   {module, test},
3   {new_index, 0},
4   {new_uniq, <<101,224,59,0,138,38,206,
5     114,154,210,50,218,246,220,11,224>>},
6   {index, 0},
7   {uniq, 53412312},
8   {name, '-fact_prod/0-fun-0-'},
9   {arity, 1},
10  {env, [<0.31.0>]},
11  {type, local}]
```

Listing 8: Local Functional Object for erlang:fun_info(Cn).

3.3 Architectural Organisation

The ERTS consists of three layers, with the Erlang kernel at the core, layered over by the Erlang Virtual Machine (EVM) and the Open Telecoms Platform (OTP). Our proof-of-concept implementation treats the Kernel layer as a black box and focusses on modifying BIF definitions at the OTP and EVM layer.

3.4 Function dependencies

We are concerned with modifying the underlying mechanisms so as to automate code migration relating to any dependencies of a remote spawn. The dependencies of a (remote) spawn consist of (i) the function that is spawned, (ii) the functions (transitively) called within its body together and (iii) any functions passed as parameters to the spawned function (and the functions they depend on). For instance, in the case of the remote spawn call on lines 5-6 in List. 3, the dependencies would be the function sndRes/4 (whose function body does not call any other function), but also the function cpx_math:fac/1 that is passed as a parameter, which transitively depends on bsc_math:mult/2 and bsc_math:add/2.

3.5 Remote Loading and Migration

In Erlang, modules can be compiled to either native code or BEAM files, executable code that be loaded and executed on any ERTS; as in other VM setups, an ERTS presents a homogeneous view for every Erlang node. Remote spawning requires a serialisation mechanism for communicating data across nodes. Erlang's standard serialisation mechanism encodes data into an intermediate representation known as the External Term Format (ETF): for functions, the respective ETF creates a *symbolic* link to the BEAM file of the module containing the function passed as data; the linked BEAM files need to be loaded at the destination node for the remote spawn to execute properly. Erlang supports the *dynamic loading* of modules inside an ERTS, which allows for subsystems to start executing before their code is fully loaded, an essential feature for long-running, open systems where the full extent of the code to be used cannot be determined prior to execution. In our case, dynamic loading allows us to rectify missing dependencies for remote spawns.

In Erlang, there is a also mismatch between the code unit for (remote) spawning, which is the *function*, as opposed to the code unit for dynamic loading, which is the BEAM file of the corresponding *module*. This discrepancy poses efficiency problems for the dynamic migration and loading of code necessary for a correct functioning of a remote spawn, as discussed earlier in Sec. 2.1.

3.6 Mechanisms for Dependency management

Erlang offers a Cross Reference Tool (`xref`) to statically analyse BEAM files to determine their relation and the call-graph dependencies between their named functions. Unfortunately, `xref` lacks the ability to identify the dependencies of *anonymous functions* which are constructed at run-time within the scope of other functions. Furthermore, `xref` only employs a static code analysis, and is unable to determine all the function dependencies from the modules' BEAM files alone (*e.g.,* calls to functions with either a variable module or a function name that is passed as a parameter). This gives rise to unresolved dependencies.

4. Solution

We present a solution for elevating the abstraction level of the Erlang's remote spawn, addressing both considerations discussed in Sec. 2.1 and architectural constraints outlined in Sec. 3. Apart from the *code-pushing* from the source nodes carrying out the remote evaluations, the underlying architecture also needs to handle *code-pulling* from the destination node (*e.g.,* in the case of lazy migration, which requires a form of *code-on-demand* [13]). Distribution further requires us to provide a solution that is decentralised; this avoids performance bottlenecks and facilitates fault-tolerance.

Our prototype implementation adopts on a number of (simplifying) assumptions which may be relaxed for more complex implementations. We limit ourselves to a pure actor-view of the language, and consider only remote spawning of code that is *side-effect free* (apart from inter-process communication side-effects); this limits bindings to codebase resources, which are stateless, and rules out bindings to stateful resources such as files and databases. From an architectural point, we assume that all nodes run the same version of the ERTS. Nodes are assumed to start off with *one* codebase version which remains fixed through the extent of their computation (*i.e.,* no updates) and that the respective BEAM files are compiled with the `debug_info` flag—this includes the Abstract Syntax Tree (AST) of the compiled source code within the same BEAM file, which can be retrieved later.

4.1 Architecture overview

At each node, our implementation uses a designated registered process, acting as the *service manager* that handles incoming requests

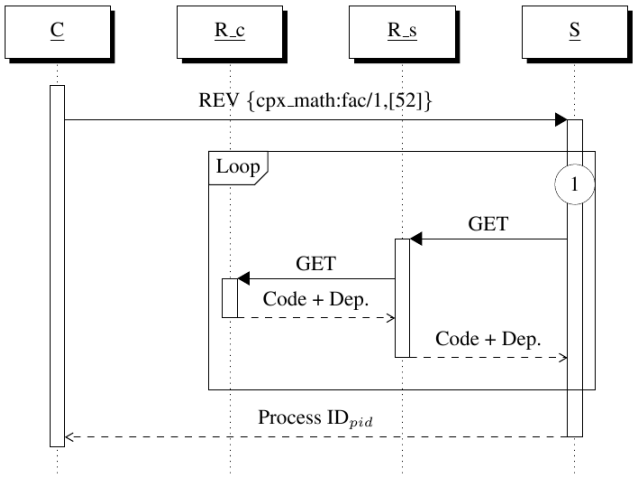

Figure 3: Protocol Overview

for remote spawns. Each node also contains a *policy file*, defining node specific requirements; in this prototype implementation, policy files are limited to specifying the mode of code migration, *i.e.,* whether eager or lazy, and a list of named functions found locally at the service node that clients must use when spawning their code. In addition, every node runs an instance of a *distributed repository service* as presented in [20], used for advertising, locating and fetching code resources.

Fig. 3 describes the top-level protocol of the augmented remote spawn BIF of our implementation, involving the client process at the client node executing the spawn (`C`), the service manager process at the service node (`S`), and the respective processes running the distributed repository service at each node (`R_c` and `R_s`). When a client executes a remote spawn, a `REV` request message is sent to the respective service manager of the target (service) node, detailing the function and arguments to be spawned. At this point, the service manager determines the (immediate) missing code dependencies that are not loaded in the service node ERTS and consults the local policy file to determine whether the missing code can be uploaded by the client (blob 1 in Fig. 3). If so, it signals its local distributed repository service to fetch the missing code. This repository, in turn, consults with the repository services at the client node (and other repositories across the rest of the network) to fetch the necessary missing code.[3] Once the missing code is located and fetched, the repository process at the service node adds the newly obtained code to its own repository, and sends it back to the service manager together with a list of the next level of dependencies. At this point the service manager consults the policy file again: if a lazy code-migration policy is to be followed, the service manager has the necessary code required to spawn the process and return the remote process identifier to the client process executing the remote spawn; alternatively, if an eager policy is to be followed, the service manager repeats the previous procedure for the next level of dependencies until there are no missing dependencies left.

4.2 Portable Functions

When named functions are passed as parameters in a remote spawn call, our implementation uses *portable functions* [23], instead of the standard functional objects discussed in 3.2 (which rely on sym-

[3] Distributed repositories may hold overlapping code resources originating from other nodes, and consulting multiple repositories makes that service faster and more fault-tolerant.

bolic links to external BEAM files for the respective function implementations). In essence, portable functions encapsulate the concrete implementation of the function within the functional object that is passed around and serialised upon distribution (using ETF); once loaded within the ERTS of the service node, they preserve the same semantics of the standard functional object implementation with external links to BEAM files.

Portable functions carry a number of advantages over the standard function encodings:

1. they are self contained and do not rely on external links, which makes them easier to serialise and transport. By contrast, communicating functional objects across ERTSs potentially changes their semantics (whenever the codebases differ);
2. they allow a finer-grained management and transport of code, since they disentangle the function implementation definitions from their container modules: portable functions only hold the implementation of the function, as opposed to the implementation of the entire module, as is the case of the BEAM files;
3. they keep function implementations in a format—namely an AST—that is more amenable to the processing required by the protocol of Sec. 4.1: as opposed to BEAM file bytecode representations, ASTs facilitate code introspection, required when performing checks against policy files, and simplify code changes through aspect-oriented tools.

4.3 Global Naming Convention

Our implementation also devises a new *naming convention* to uniquely identify portable functions. This enables it to load multiple versions of the function f from module m, originating from different nodes, thereby solving the name clashes problem discussed in Sec. 2.1. More specifically, the module name containing a migrated function is modified by appending it with the *node name* from where it originates; since node names are guaranteed to be *unique*, this would in turn create a unique module name across the entire network of nodes.

Our naming convention needs to go a step further. For efficiency reasons, the code included in a portable function object is limited to that of the function itself (as opposed to that of the entire module containing it). From the point of view of service node loading the functions individually, this acts as a form of *module partitioning* for every function contained in it. Using the same module name for each module partition causes problems since the unit of code loading is the module, and the same module cannot be reloaded without overriding the previously loaded code. Thus, our naming convention also adds the function name and arity to the module (partition) when renaming, so as to also uniquely identify each module partition across the entire network: the uniqueness of a function and its arity within a module guarantees *global* uniqueness.

Portable functions and global naming uniqueness also help to address the problem of storage efficiency and complications associated with establishing code correspondence, discussed in 2.1 and depicted in Fig. 2. In that case, the second remote spawn is able to determine that the respective code is already present at node nodeM, since the underlying name of portable function would allow it to determine the code origin, *i.e.*, the client node (as opposed to the immediate provenance of the code *i.e.*, nodes nodeL and nodeK).

List. 9 outlines the information contained in the portable function (stated on line 22) for cpx_math:fac/1. The module name is changed from cpx_math to cpx_math-fac-1-@NodeA, appended by both the function name and arity, but also by the unique name of the originating node (line 1). It contains the respective code of the function, encoded as an AST, as part of the new code attribute (lines 5-17).

```
1  [{module,'cpx_math-fac-1-@NodeA'},
2   {name,fac},
3   {arity,1},
4   {env,[]},
5   {code,[
6   [{attribute,1,module,'cpx_math-fac-1-
        @NodeA'},
7    {attribute,3,export,[{fac,1}]},
8    {function,5,fac,1,
9     [{clause,5,[{integer,5,0}],[],[{integer
          ,5,1}]},
10     {clause,6,
11      [{var,6,'X'}],
12      [],
13      [{call,6,
14        {remote,6,{atom,6,'bsc_math-mult@A'}
              ,{atom,6,mult}},
15        [{var,6,'X'},
16         {call,6,{remote,6,{atom,6,'cpx_math
               -fac@A'},{atom,6,fac}},[{op,6,'
               -',{var,6,'X'},{integer,6,1}}]
              ]}]}]}]
17  ]},
18  {calls,[{'bsc_math-mult@A',{mult,2},
19        [{'bsc_math-add@A',{add,2},[]}]
20     }]
21  },
22  {type,portable}]
```

Listing 9: Portable Functional Object erlang:fun_info(fun cpx_math:fac/1)

4.4 Code Dependencies

Portable functions come also equipped with a calls attribute that describes the (static) dependency call graph of the function; this is used to determine code dependencies in the remote spawn protocol presented earlier in Fig. 3. Our implementation relies on Erlang's xref tool to build this dependency call graph of a function. For instance, in the case of cpx_math:fac/1 of List. 1, we obtain the call dependencies described in List. 9 (lines 18-21).

In order to keep code transfer as efficient as possible, we do not include functions that belong to the Erlang/OTP libraries as part of the dependency call graph listed in the portable function calls attribute, since it is assumed that all computational environments engaged in code mobility contain the same version of the ERTS. Thus, for example, if cpx_math:fac/1 of List. 1 had to be altered to the code in List. 10, calling the standard OTP function io:format/2, we would still obtain the same call dependencies of List. 9.

```
1  -module(cpx_math).
2  -export([fac/1,fib/1,...]).
3
4  fac(0) -> 1;
5  fac(X) -> io:format("Factorial: ~p",[X]),
6            bsc_math:mult(X,fac(X - 1)).
7  ...
```

Listing 10: Complex Math Module containing a Factorial Function

Code dependencies may also be *dynamic*, as in the case of the function test_module:variable_mod_fac/2 in List. 11. This function allows the caller to specify which module to use when calling

the factorial function[4] but since this module is only known at runtime, it cannot be determined statically.

```
1 -module(test_module).
2 -export([variable_mod_fac/2]).
3
4 variable_mod_fac(Mod,N) -> Mod:fac(N).
```

Listing 11: Unresolved Call

Our prototype implementation handles dynamic code dependencies by *modifying* the code of the migrated function. More precisely, after applying xref we end up with a list of unresolved calls: for each of these calls we inject a statement within the function's AST that explicitly checks and requests possibly missing call dependencies *at runtime*, by which time the parametrisable call information would be instantiated. List. 12 presents the function definition of the modified AST that will be encoded in the respective portable function of test_module:variable_mod_fac/2 in List. 11; the dynamic check is performed at line 5 through the command mcode:demand_load_code(Mod,fac,2), immediately preceding the (previously unresolved) call.

```
1 -module(test_module).
2 -export([variable_mod_fac/2]).
3
4 variable_mod_fac(Mod,N) ->
5   mcode:demand_load_code(Mod,fac,2),
6   Mod:fac(N).
```

Listing 12: Encoding Variable Function Call

4.5 Distributed Repository

A distributed repository holds resources and code replicas at different nodes so as to facilitate their discovery and exchange. Fig. 4 depicts resource discovery in the case of the multi-node remote evaluation in Fig. 1 of Sec. 2. When the client node invokes the first remote spawn on nodeL, the distributed repository at the service node requests the missing code from the distributed repository of client node and registers it locally, indexed by the unique naming convention discussed in Sec. 4.3. Crucially, when nodeL invokes the remote spawn on nodeK, the distributed repository at the latter service node may obtain the missing code from *either* the distributed repository at either nodeL *or* the client node. This replication leads to a *decentralised* and *fault-tolerant* organisation of code repositories. Stated otherwise, should nodeL fail or become unreachable at the point when nodeK is fetching its missing code, it can still be obtained from the client node.

4.6 Policy Files and Code Introspection

In our prototype implementation, policy files specify two different kinds of service side requirement: one relating to efficiency (eager vs lazy) and another relating to security (what code should be used locally (or prohibited from being remotely loaded). Although we do not give a full and proper treatment of service node policies, our aim is more directed towards introducing a mechanism requiring a form of introspection and filtering when migrating and loading code. This allowed us to construct a prototype that can handle such introspection, and can then be extended through minimal structural modifications to handle more complex policies.

List. 13 depicts a sample policy file that may be specified by a node in our implementation. Line 1 states that the destination node

[4] The user may have multiple implementations of this function, placed in different modules.

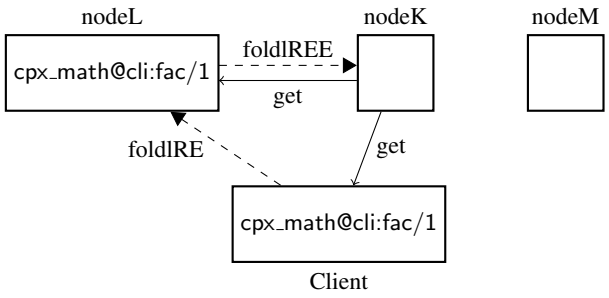

Figure 4: Replication Across Distributed repositories

of the remote spawn prefers code to migrate in lazy fashion. Lines 3 and 4 specify functions that are prohibited from being loaded remotely: for instance line 3 rules out any function named fib with arity 2 (irrespective of the containing module name, whereas line 4 rules out any function contained in modules called danger.

```
1 <lazy>
2
3 <*,fib,2>
4 <danger,*,*>
```

Listing 13: Sample Policy file

4.7 Anonymous Functions

Our implementation also handles remote spawns involving *anonymous functions*, thereby increasing the flexibility and applicability of our solution. For instance, it also allows us to handle mobility skeletons such as the one presented in List. 4 and List. 5.

Anonymous functions pose additional complications to their named counterparts. For starters, when they are encoded as portable functions in our setup, they cannot be extracted directly from their surrounding function and placed in a respective module partitioning. There are two obstacles that prohibit this: (i) they would need to be converted into a named function, since Erlang modules may only *directly* contain named functions (see Sec. 3.1); (ii) they may contain free variables, that are bound in the surrounding function context.

In our implementation, when anonymous functions are encoded as portable functions, they are assigned a name akin to that assigned to them in BEAM files.[5] There, anonymous functions are assigned an indexed name of the form $f - a - \mathsf{fun} - i$, where f is the parent function name, a is its arity and i denotes the *order* in the list of anonymous functions defined inside the parent function. We follow this naming convention and assign the name

$$-m - f - a - \mathsf{fun} - i - a' - @n$$

to the i^{th} anonymous function with arity a' defined inside function f with arity a in module m at node n.

Our implementation addresses the binding of free variables in an anonymous function by *altering* the code included inside the portable function, inserting the respective variable bindings before the function code; this strategy is only possible because of the single assignment property of the language.[6] For instance, in the case

[5] BEAM file bytecode translations represent anonymous functions as named functions.

[6] More precisely, a variable in Erlang may be assigned multiple times, as long as the expressions assigned to it all evaluate to the same value [1, 10]. Our implementation chooses the first binding occurrence.

of the anonymous function on line 2 of List. 5, its portable function would contain the AST representing the anonymous function of List. 14; note how the binding Id = self() is moved to within the body of the function (line 1).

```
1 fun(X) -> Id = self(),
2           Id!{result,X}
3           end
```

Listing 14: Encoding fun(X) − > Id!result,X end of List. 5

Anonymous functions also pose a minor implementation obstacle to our implementation when translating them into portable functions. In particular, xref is not able to analyse the dependencies of anonymous functions and our implementation has to determine dependencies itself in such cases.

```
1  [{module,'test_mod-test_fun-0-fun-0-1@NodeA'},
2   {name,'test_fun-0-fun-0-1'},
3   {arity,1},
4   {code,[
5   [{attribute,1,module,'test_mod-test_fun-0-fun-0-1@NodeA'},
6    {attribute,3,export,[{'test_fun-0-fun-0-1',0}]},
7    {function,4,'test_fun-0-fun-0-1',0,
8      [{clause,4,
9        [{var,4,'X'}],
10       [],
11       [{match,5,{var,5,'Pid'},{pid,5,<0.30.0>}},
12         {op,6,'!',
13         {var,6,'Pid'},
14         {tuple,6,[{atom,6,result},{var,6,'X'}]}}
15       ]
16     }]
17   }]
18  ]},
19  {calls,[]},
20  {type,portable}]
```

Listing 15: Portable Functional Object erlang:fun_info(Cn)

The portable function produced for the anonymous function fun(X) − > Id!result,X end of List. 5 is given in List. 15, assuming that it is defined inside the name function test_fun/0 in module test_mod.

5. Results

We have produced a series of Erlang modules implementing the architecture discussed in Sec. 4. In particular, module mcode, an extended gen_server (a standard OTP behaviour/module [20]), contains the code used by the service manager in Fig. 3 whereas mcode_cb acts as the callback module to mcode. Module resource_service contains the code used by the distributed repositories; it is a gen_server extension as well. Policy files are processed using functions contained in policy_file_parser and reconciliation_service. The logic that constructs our portable function is contained in module closure_rep. Finally mcode_app is the initial module that calls service manager and distributed repository supervisor, by calling the function resource_service found in the mcode_supervisor; we therefore just start up mcode_app at every participant node. The code can be downloaded from [29].

5.1 The Case Study

Our implementation allows us to execute the code discussed Sec. 2 in settings were only the client node contains the codebase relating to the functions spawned. This is done through minimal changes to the code, without having to clutter it with additional low-level mechanisms for code migration, loading and management. List. 16 presents the code required by our implementation, corresponding to the driver code of List. 3 (List. 1 and List. 2 are unaffected). The new code is in fact shorter than that of List. 3: it creates the new remote process as before; upon termination, however, this process (automatically) returns the result to the client's node service manager process, which can be retrieved via an asynchronous call to mcode:get_result/1 using the new process identifier. This approach *avoids* having to use the sendRes wrapper function.

```
1 Id = spawn(nodeL,cpx_math,fac,[42]),
2 Res = mcode:get_result(Id).
```

Listing 16: New Remote Evalutation of fac/1

List. 17 presents the code corresponding to that of List. 5, offloading a series of computations over a set of service node and aggregating their results. This time the new code corresponds more to the original code of Sec. 2, where the only change required regards anonymous functions: they are specified using the fun! construct (lines 2 and 4) which forces them to be compiled differently, giving us access to the bindings of their free variables; this change was necessary because our implementation did not have access to the Erlang kernel internals.

```
1  Id = self(),
2  Cn = fun!(X) -> Id!{result,X} end,
3
4  Fn = fun!(X,Ac) -> cpx_math:fac(X)*Ac end,
5
6  Vs = [42,53,64],
7  Ns = [nodeL,nodeK,nodeM],
8  foldIRE(Fn,Cn,1,Vs,Ns),
9  receive
10   {result,Res} -> Res
11 end.
```

Listing 17: Using foldIRE/5 with Portable Functions

6. Related Work

There are a number of programming languages offering constructs and support that enable computation to be distributed across geographically or logically dispersed computational environments. For instance, parallel and concurrent languages such as [22, 26] facilitate the development of systems that exploit different processors to hasten the completion of a unit of execution. Distributed languages facilitate the communication and management of autonomous software components making up a system over different computational environment to improve its scalability, reliability and fault tolerance; a subclass of these languages support code and computation mobility so as to achieve better flexibility and extensibility [5, 13].

6.1 Code mobility

In their seminal work, Fuggetta *et al.* [13] define code mobility as the ability to dynamically change the binding between the code and their computational environment. Computational environment bindings typically associate code with local resources, *e.g.*, data files and code binaries, and the execution state of the code. *Weak mobility* paradigms migrate only the program code and resources

bindings between different computational environments whereas *strong mobility* abstractions go a step further and facilitate the transfer of bindings relating to the execution state.

6.2 Mobile Coding Extensions

Programming languages supporting mobile code employ different combinations of linking and code loading schemes. One of the first remote evaluation language extensions [24], proposed for the CLU procedural language, statically links all the required executable code during the compilation phase and dynamically loads it onto a service node during remote evaluation. Similar to our distributed repositories, nodes may expose a set of procedures that can be dynamically linked to a client's executable code.

Runtime linking is also employed in *mHaskell* [4, 6], an extension that introduces explicit mobility of Haskell computations over special *channels*. Local code at each node is partitioned into migratable (compiled into bytecode and interpreted using the GHCi compiler) and non-migratable (compiled into native code using the GHC compiler). When code is sent remotely over the channels, only the migratable code is sent, and computation would need to substitute non-migratable code at the source node with corresponding code at the destination node (when available).

Closer to our work is Emerald, an object oriented language that provides constructs that facilitate the mobility of its objects, composed of data, source code (stored in a special object) and execution state [17]. In fact, Emerald goes a step further and expresses strong object mobility: upon object migration, both the data *and* execution state gets transferred and loaded on the remote location, which then has the responsibility to check, retrieve and load missing code.

6.3 Code Mobility in Erlang

Erlang provides lower abstraction mechanisms for code mobility that only allows dynamically linked modules to be loaded at remote ERTS [10, 20]: the construct c:nl/1 loads a module on all connected nodes, whereas code:get_object_code/1, code:load_binary/3 and rpc:multicall/4 provide mechanisms for explicitly obtaining and remotely loading binaries. Erlang portable functions were suggested in an Enhanced Proposal (EEP) [23] to aid in the mobility of functions. To the best of our knowledge, no implementation exists materialising this proposal. Our work provides a prototype implementation of portable functions that are statically linked at run-time and explicitly loaded during execution. In addition to the portable function proposed in [23], our work eliminates the restrictions imposed on code dependencies of portable functions: according to the proposal portable functions could not use plain (non-portable) functions and dynamic dependencies that are determined at runtime (*e.g.,* List. 11).

6.4 Mobility in Actor languages

An actor language supporting remote spawning similar to Erlang is THAL [18]; as in the case of Erlang, the language assumes homogeneous codebases across nodes. There are also a number of distributed languages based on the actor model supporting a *weak* form of actor mobility such as SALSA [27]: again, these technologies assume homogeneous codebases across nodes.

ActorNet [19] is an actor programming platform that supports *weak* actor mobility and code migration. The `migrate` function presented in this language accepts a lambda expression which gets encoded and transferred to the remote location. However, the platform does not provide any form of code dependency management and assumes that all the required code is explicitly passed as a parameter to the `migrate` function. Similarly, in Scala [14], a closure may be sent within an Akka [16] message to a remote actor. This mechanism is nevertheless discouraged by Akka's development team, since Scala closures may contain mutable variables that introduce state sharing.

STAGE [3] is an actor-based distributed language (built on top of Python) supporting *strong* migration, where actors encapsulating both their behaviour and state can be migrated between different nodes. Again, the language assumes homogeneous codebases across nodes. Actor Foundry [2] is an actor framework for Java also supporting strong actor migration. The framework relies on the Java serialisation mechanism to transfer actor state across nodes but does not provide any automated support for codebase management in heterogeneous codebase settings.

6.5 Community Cloud Computing

In community cloud computing models, multiple nodes combine and share their computational and physical resources [7]. The design space for such models incorporates aspects, ranging from remote service execution to digital currency. Our case study outlines the basic infrastructure required for such models, viewing Erlang nodes as executing platforms upon which client nodes can evaluate fine grained units of execution. At a lower abstraction level, our solution includes distributed repositories that store sourced (remote evaluating) programs. We also provide rudimentary (but extendible) mechanisms for managing and restricting access to these resources though our policy files.

The domain specific languages (DSL) called Cloud Haskell, presented in [12], provides Erlang inspired remote evaluation for community cloud computing, allowing new processes to be spawned at remote locations. It also provides mechanisms for handling remote spawn function closures (free variables in functions), as discussed in Sec. 4.7 for anonymous functions. However, similar to Erlang's current setup, this DSL does not support code mobility when it is missing at the destination node.

6.6 Mobility Skeletons

In List. 4 we have presented an implementation of a mobility skeleton for a weak form of code mobility, expressed at a higher level though remote evaluation. Mobility skeletons are high level programming abstractions for common mobile coding patterns [5]. They have been successfully implemented in Haskell as a set of higher order functions that coordinate the execution of a computation over a static set of predefined locations. Mobility skeletons have also been extended to auto-mobile skeletons which also abstract from the location details: these self-aware computing patterns periodically migrate computations over new locations which are determined at run-time depending on the load of each node within an open network. Auto-mobile skeletons have been implemented in a number of Java extensions such as JavaGO and Java Voyager, as well as in Jocaml (a distributed Caml with join patterns) [11].

6.7 Execution platform safety

Code mobility introduces a number of security vulnerabilities ranging from breaches of confidential data to attacks on the integrity and availability of a system when compared to more traditional distributed systems [25]. These threaten the entire service node and its execution platform requiring safety guarantees about its stability. One of the most prominent Erlang security extensions provide a safe execution environment for mobile coding paradigms [8]. This work introduces a constrained environment, *i.e. sandbox*, that limits the resources available for running mobile code. These resource limit can be set dynamically depending on the authenticated computational environments in question and the over allocated resources. Our solution does not guarantee such safety of the execution environment but merely guarantees that remotely evaluated code does not violate rules specified by service nodes. This mechanism, allows a client and a service node to negotiate the details of

an remote execution as opposed to completely rejecting unresolved requests.

7. Conclusion

We have presented a prototype implementation [29] of an extended Erlang remote spawn that abstracts away from the migration, loading and management of missing code across nodes, giving the illusion that every node shares the same codebase; this ensures that remote spawn emulates the behaviour of a local spawn, its assumed correct behavioural specification. Our implementation attains this while transferring minimal codebase subsets, and coordinates codebase discovery and migration in decentralised fashion, allowing for degrees of fault-tolerance. We also show how our implementation facilitates the use of remote evaluations to implement frameworks offering an execution-platform-as-a-service.

7.1 Future Work

There are a number of possible extensions to our framework that would enhance its applicability. For instance, our enhanced remote spawn implementation could be extended to handle function side-effects such as accesses to files and ETS tables [10]. One possible solution would be to expose such dependencies over our distributed repositories and provide abstractions for shared data across nodes; in special cases where the migrated code has exclusive access to such stateful handles, these resources could even be migrated with the code. Policy files can also be extended to specify whether service nodes access computation with side effects or whether they accept the migration of resources such as files.

Policy files offer a lot of potential for fine-tuning the service offered by remote nodes. Service constraints may be extended with modalities specifying the provenance *i.e.,* source node and migration path, of migrated code as well as the size of the code. They could also used provenance information to specify the actions allowed for certain code *e.g.,* prohibiting them from accessing certain resources: this could be coupled with sandboxed execution environments such as in [8]. Policy files may also specify dynamic requirements, where execution constraints for mobile code could be relaxed or tightened depending on the performance of the code. They could also specify codebase redundancy protocols held at distributed repositories which would improve the fault tolerance capabilities of the framework.

There are also issues relating to performance that our implementation needs to address better. For instance, in the current implementation, any loaded modules as a result of remote spawns are never purged, even though they may never be required again. One possible solution would be to introduce a notion of a session between nodes whereby the closing of a session would indicate when loaded modules can be purged.

Acknowledgments

The authors are grateful to Richard O'Keefe, Steve Vinoski and other contributors on various Erlang fora for fruitful discussions.

References

[1] Joe Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, 2007.

[2] Mark Astley. ActorFoundry: A Java-based Actor Programming Env. `http://osl.cs.uiuc.edu/software/actor-foundry/`. (Aug 2013 last accessed).

[3] John Ayres. *Implementing Stage: the Actor based language*. PhD thesis, Imperial College London, 2007.

[4] André Rauber Du Bois, Hans-Wolfgang Loidl, and Phil Trinder. mhaskell: Mobile computation in a Purely Functional Language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2004.

[5] André Rauber Du Bois, Hans-Wolfgang Loidl, and Phil Trinder. Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288, 2005.

[6] André Rauber Du Bois, Hans-Wolfgang Loidl, and Phil Trinder. Implementing Mobile Haskell. In *Proc. TFP'03*, volume 4, pages 79–94, September 2003.

[7] Gerard Briscoe and Alexandros Marinos. Digital ecosystems in the clouds: Towards community cloud computing. In *DEST*, pages 103–108. IEEE Press, 2009.

[8] Lawrie Brown and Dan Sahlin. Extending Erlang for Safe Mobile Code Execution. In *ICICS*, pages 39–53, London, UK, 1999. Springer-Verlag.

[9] Luca Cardelli. Abstractions for mobile computing. *Secure Internet Programming*, 1603:51–94, 1999.

[10] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly, June 2009.

[11] Xiao Yan Deng, Greg Michaelson, and Phil Trinder. Autonomous Mobility Skeletons. *Parallel Comput.*, 32(7):463–478, 2006.

[12] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the Cloud. *SIGPLAN Not.*, 46(12):118–129, 2011.

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

[14] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *TCS*, 410(23):202 – 220, 2009.

[15] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245. Morgan Kaufmann, 1973.

[16] Typesafe Inc. *Akka Java Documentation*, August 2013.

[17] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

[18] Wooyoung Kim. Thal: An actor system for efficient and scalable concurrent computing. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1997.

[19] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: an actor platform for wireless sensor networks. In *AAMAS*, pages 1297–1300. ACM, 2006.

[20] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang an OTP in Action*. Manning Publications, December 2010.

[21] Cecilia Mascolo, Gian Pietro Picco, and Gruia-Catalin Roman. A fine-grained model for code mobility. *SIGSOFT Softw. Eng. Notes*, 24(6):39–56, October 1999.

[22] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL*, pages 295–308, NY, USA, 1996. ACM.

[23] A. O'Keefe Richard. Erlang Enhanced Proposal (EEP) 15: Portable funs. `http://www.erlang.org/eeps/eep-0015.html`, 2008.

[24] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 12(4):537–564, 1990.

[25] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sept 1997.

[26] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. Gum: a portable parallel implementation of Haskell. In *PLDI*, pages 79–88, NY, USA, 1996. ACM.

[27] Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, December 2001.

[28] Claes Wikström. Distributed Programming in Erlang. In *International Symposium on Parallel Symbolic Computation*, pages 412–421, 1994.

[29] Tyron Zerafa. Erlang's Code Management Extension Source Code. `https://github.com/TyronZerafa/Erlang-Code-Migration`.