Distributed Actors in OCaml

Wenke Du, Ludovic Henrio, Gabriel Radanne

Distributed Actors in OCaml

Wenke Du, Ludovic Henrio, Gabriel Radanne
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France.
Lyon, France

I. Introduction

The Actor Model [1] is a concurrency model where each "actor" is an entity that executes in its own logical thread. Actors each possess their own private state, which can't be accessed externally. And communicate with each other via asynchronous messages, often implemented by promises [2]. Thanks to these characteristics, Actors prevent common concurrency issues such as low-level data races and deadlocks, making them easy to use to develop large-scale parallel or distributed applications [3]. Actors libraries have been successfully developed in numerous languages, notably Scala [4] and Erlang [5], among others [6], [7].

OCaml 5 introduced shared memory parallelism [8] and algebraic effect handler [9]. This provides the opportunity to investigate Actors as an easy-to-use concurrency programming model for parallel and distributed applications in OCaml. Indeed, Actors promise to be a good fit for OCaml: they promote encapsulation of state and the use of structured concurrency primitives such as promises, both of which are already practiced by the OCaml community. Actors also combine well with functional objects, as present in OCaml, to form "active objects", where message passing becomes method calls.

M. Andrieux, L. Henrio, and G. Radanne [10] gave the formal basis for Actors in OCaml using algebraic effect in a concurrent (but not distributed) setting. In this extended abstract, we showcase the `actors-ocaml` library for concurrent and parallel programming and its recent extension to distributed applications.

II. Actor in OCaml

In this section, we demonstrate how to use our library to easily develop and scale an application to its parallel and distributed versions, while ensuring the absence of common issues such as data races. We begin with a simple counter example to introduce the syntax, then present a more evolved example solving the N-queens problem.

A. Define actors by annotation

Listing 1 defines a counter actor with an OCaml object. The counter is initialized to 0 and each call to the `incr` method increments the counter by n. The only difference

```
1 class%actor counter = object
2   val mutable x = 0
3   method get = x
4   method incr n = x <- x + n
5 end
```

Listing 1: Definition of a Counter Actor

```
1 class counter : object
2   method get : int Promise.t
3   method incr : int -> unit Promise.t
4 end
```

Listing 2: Signature of a Counter Actor

between this code and standard OCaml objects is the `%actor` annotation placed at the beginning. This extension transforms the class into an actor class. The signature of the resulting actor is displayed in Listing 2, and shows that each method is now **asynchronous**: its result is wrapped inside a `Promise.t` . Finally, Listing 3 shows some use of a counter: it first creates it, on Line 1, then increment it by synchronous call `c#.incr` on Line 2. We then use an asynchronous call `c#!get` which return a promise. While this call resolves, we can do some other computation (Line 4) before `await` ing the promise and using the result.

An actor is a fully-fledged concurrent object, but where only one statement of the object's methods is executed simultaneously. Since the internal state is only available through publicly exposed methods, this ensures the absence of data-races. Only 'await' can interrupt the execution of a method and start executing a new one. This ensure that, from the programmer's point of view, the inside of a method that has no 'await' statement can be thought of as sequential codes. Each actor is thus equipped with an internal cooperative scheduler to orchestrate the execution of the various methods. We also prevent various mistakes, such as leaking

```
1 let c = new counter in
2 c#.incr 2; (* A Synchronous call *)
3 let p = a#!get in (* An Async call *)
4 let a = f () (* Some code while p is resolved *)
5 Promise.await p + a (* Obtain the value of p *)
```

Listing 3: Spawn Counter Actor on a green thread

internal state via closures. For more details on the semantics and properties of our implementation of Actors, see [10].

B. A concurrent example

Let us now look at a more ambitious program, and how to scale it: an (embarrassingly parallel) n-queen solver, implemented in Listing 4. Our implementation uses a classical brute-force algorithm with persistent arrays that recursively place queens column by column to enumerate all solutions. To parallelize it, an actor is created for the next column (Line 17), and the result is resolved asynchronously (Line 21). Promises containing the solutions are collected at the end (Line 32).

We then exploit a new capability of our Actors: when calling `new` we can specify a **location** where the actor should spawn. We can easily provide a precise location, to spawn on a specific node in the distributed system, or use a `Spawner` which tries to balance the workload, based on a distribution strategy. For instance, `Spawner.GreenThread` would create a new green thread for every actor, yielding a multi-threaded version. Conversely, `Spawner.GlobalGreenThread`, used in Listing 6 on Line 3, creates green threads in distant nodes, yielding a distributed version. We could also provide a more specific location to spawn on a specific node in the distributed system.

C. Additional Features

On top of a solid concurrent programming style using Actors-based on algebraic effects as described in [10], `actors-ocaml` proposes a convenient library for concurrent, parallel, or distributed applications that allows programmers to reason about their domain logic sequentially and then automatically generates the corresponding actor code. Our implementation supports several key functionalities over the network for distributed operation:

- Node Synchronization: Ensuring nodes within the cluster are aware of each other.
- Actor Communication: Allowing the sending of messages to actors on remote nodes, including sending promises and even actors between nodes.
- Actor programming: Providing a unified API handling actors on local or remote nodes identically.

III. Discussion and Talk

Naturally, a lot is still needed to reach a production library.

1. We would like to integrate with existing OCaml libraries for concurrency. Indeed, the current proliferation of promise libraries makes interoperability delicate, as highlighted by the Picos project. In particular Riot provides runtime primitives that are complementary to the programming model we propose.

```
1  module PArray = CCPersistentArray
2  type row = Free | Occupied of int
3
4  class%actor qsolver spawner board = object
5    (self)
6    val board : row array = board
7
8    method is_valid_position i j =
9    (*Check if a queen can be placed at (i, j)*)
10
11   method with_queen i j =
12     let location = spawner () in
13     let new_board =
14       PArray.set board i (Occupied j)
15     in
16     let new_actor =
17       new qsolver location new_board
18     in
19     Promise.fmap
20       (fun sub -> List.map (fun l -> i::l) sub)
21       (new_actor#!solve (j + 1))
22
23   method solve j =
24     let n = PArray.length states in
25     if n = j then [ [] ] else
26       let solutions = ref [] in
27       for i = 0 to n - 1 do
28         if self#.is_valid_position i j then
29           let new_sols = self#.with_queen i j in
30           solutions := new_sols :: !solutions
31       done;
32       List.concat_map Promise.get !solutions
33 end
```

Listing 4: Parallel n-queen Actor implementation

2. So far, we have leveraged OCaml's support for objects to implement Actor's encapsulation. Another construct provides excellent encapsulation in OCaml: modules and abstract types. We are currently experimenting with a different API based on this idea, hoping that it provides a more idiomatic OCaml API.

In the presentation, we will present additional examples of concurrent and distributed programming using `actors-ocaml`, some of the implementation challenges to provide safe communication between distributed actors, and discuss these various future extensions.

```
1  class queen_solver :
2    location -> row PArray.t -> object
3      method solve : int -> int list promise
4    end
```

Listing 5: Parallel n-queen Actor signature

```
1  let spawner = Spawner.GlobalGreenThread.spawn
2  let initial_board = PArray.init 8 Free
3  let a = new qsolver spawner initial_board
4  let solutions = Promise.await @@ a#!solve 0
```

Listing 6: Distributed n-queen Actor implementation

References

[1] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, in IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[2] F. D. Boer *et al.*, "A Survey of Active Object Languages," *ACM Computing Surveys*, vol. 50, no. 5, pp. 1–39, Sep. 2018, doi: 10.1145/3122848.

[3] J. De Koster, T. Van Cutsem, and W. De Meuter, "43 years of actors: a taxonomy of actor models and their key properties," in *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, Amsterdam Netherlands: ACM, Oct. 2016, pp. 31–40. doi: 10.1145/3001886.3001890.

[4] P. Haller and M. Odersky, "Actors That Unify Threads and Events," in *Coordination Models and Languages*, A. L. Murphy and J. Vitek, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 171–190.

[5] R. Virding, C. Wikström, and M. Williams, *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996.

[6] S. Srinivasan and A. Mycroft, "Kilim: Isolation-Typed Actors for Java: (A Million Actors, Safe Zero-Copy Communication)," in *ECOOP 2008–Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings 22*, 2008, pp. 104–128.

[7] D. Caromel, L. Henrio, and B. P. Serpette, "Asynchronous and deterministic objects," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004, pp. 123–134.

[8] K. C. Sivaramakrishnan *et al.*, "Retrofitting Parallelism onto OCaml," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–30, Aug. 2020, doi: 10.1145/3408995.

[9] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy, "Retrofitting effect handlers onto OCaml," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 206–221.

[10] M. Andrieux, L. Henrio, and G. Radanne, "Active Objects Based on Algebraic Effects," *Active Object Languages: Current Research Trends*, vol. 14360. Springer Nature Switzerland, Cham, pp. 3–36, 2024. doi: 10.1007/978-3-031-51060-1_1.