Studying the feasibility of serverless actors

Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, David Arroyo Pinto, Pedro García López
Universitat Rovira i Virgili
Tarragona, Spain
{daniel.barcelona, alvaro.ruiz}@urv.cat, david.arroyop@estudiants.urv.cat, pedro.garcia@urv.cat

Abstract

Serverless is a very promising model with many benefits to simplify the development of cloud applications. However, many applications are not easy to build on a serverless environment due to the lack of built-in key features, such as state and function coordination. In this paper we focus on the actor model. As this popular computational model challenges many aspects of the current serverless tools, the feasibility of building a serverless actor framework is unclear.

Our goal is to study whether the actor model can be successfully deployed on top of a Functions as a Service (FaaS) environment like AWS Lambda. To do that, we design and build a prototype to evaluate the serverless actors requirements and performance. We conclude with a successful prototype implementation and stating the necessary run-time extensions to the serverless core to improve the support for serverless actors.

1 Introduction

The serverless model benefits from several features that make the developer's life easier; such as scalability, minimal deployment process, hardware and Operating System configuration, and sub-second billing. The current main offer of serverless services comes in the form of Functions as a Service, where the user codes small functions that respond to events.

Unfortunately, not all applications have a straightforward migration to the serverless architecture that easily benefits from its advantages. This is the case of the actor model, which is a highly popular computational pattern for building concurrent applications. The model simplifies the job of composing parallel and distributed executions by using a basic unit of computation: the actor.

An actor is an isolated, independent unit of compute and state with single-threaded execution. Many actors can execute simultaneously and independently of each other to build complex applications. Actors can communicate between them by sending messages and act upon them.

The actor model benefits from the serverless computing framework in two main aspects:

- *Billing.* FaaS platforms charge per compute time—at small-grained periods—and the amount of resources consumed by the application, which in this case is more cost-effective than having to purchase or rent servers with much more coarse-grained billing time units. We pay only for actual actor run time.
- *Scalability.* Users do not need to spend their time managing servers and setting up auto-scaling systems, as cloud providers are responsible for seamlessly scaling the capacity on demand. We can have a virtually infinite number of concurrent actors. [1]

A simple, yet convenient use case would be the counter example. In which, there's no need to have a server running an application to handle the counter increments, as the state only changes in reaction to events. This counter example can't be efficiently implemented with stateless functions as they don't guarantee state persistence between invocations, so a remote external storage must be used. In contrast, serverless actors are a better fit as a result of their combination of state persistence and fine-grained billing.

In this paper we discuss the feasibility of the actor migration to a serverless environment and the series of challenges that arise. This study begins defining the main challenges that we have to solve to implement serverless actors. After that, we design and implement a solution on top of the current serverless offering explaining how we solve the previous challenges. Finally, as a result of evaluating the solution, we discuss the necessary run-time extensions to the serverless core to improve the support for serverless actors.

2 Challenges

The base of our work is that a serverless function (FaaS) constitutes an actor. Actors can receive messages and act upon them and their own state. They only process one message at a time.

Serverless functions are not designed to support the actor model. As a consequence, there are certain requirements of implementing actors that are not offered in current FaaS cloud services.

Addressing. The most important element of the actor model is that actors can receive and send messages to other actors. A message income implies the execution of the actor. That actor processes the received message by performing an action.

Despite that most cloud providers offer some kind of invocation endpoints, they are all limited to invocation requests, in other words, once a lambda functions is running, it can't receive external data unless it makes an explicit request. Therefore, the usage of external communication services is required.

In addition, the serverless model works better with events, so the service should hold them until they are processed.

Naming is another issue. Each actor instance needs its unique identifier that other actors use to establish communication and send messages.

Atomicity. The actor model works on the base of atomicity of actors. To maintain a consistent state, and execute correct actions in response to messages, there cannot be more than one instance of the same actor executing at the same time.

Serverless functions scale automatically by spawning concurrent containers. This breaks the atomicity of actors if two concurrent functions consume from the same message channel. Therefore, we need to limit function concurrency.

State. Actors are stateful and maintain a mutable state that takes part in action decision and logic. On the other hand, serverless functions are built stateless, in a way that consequent calls to the same function may not maintain previous state. Cloud providers don't offer built-in state management for their FaaS services, and external storage services must be used.

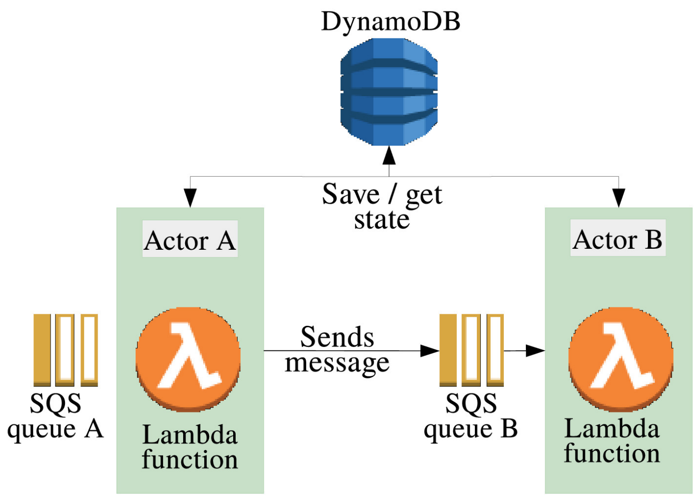

Figure 1: architecture overview.

Passivation A key element for serverless actors is the time with no message income.

In a fully event-driven system, actors should only execute on message reception, as a reaction. This approach in a FaaS environment would be unfeasible, as each actor invocation would imply a cloud function request and accesses to an external storage to load and dump state (severely affecting *performance*).

However, keeping the actor running to avoid these latencies, greatly increases computation cost, as we would be billed for unproductive execution functions.

Performance. The actor model must be functional. Therefore, a minimum *performance* is mandatory for the viability of its adoption. This is a special requirement given the high network latencies of the remote components such as distributed storage and communication.

3 Design and Implementation

In this section we present a solution[1] for serverless actors on top of AWS. Nevertheless, the structure is similar on other platforms like Azure.

Fig. 1 depicts an overview of the solution. We use AWS Lambda as computation power for the serverless actors. Each actor instance is a new function. Then, spawning an actor means deployment and creation of a new Lambda function. Actors' state is persisted in

[1]prototype available at github.com/danielBCN/faasactors

DynamoDB. Finally, SQS queues are used to enable communication between lambdas.

The problems discussed in the previous section are solved in the following ways.

Addressing. Actors should *react* to messages and hold them until they are processed. This could be done with a messaging queue or an event bus. Messaging queues permit to store the messages persistently until the actor processes them.

A queuing service also solves the naming issue, since actors and queues could share identifier. A message `m`, which should be sent to an actor with identifier `aid`, is queued to the queue with name `aid`. This approach allows communication between actors (and from anywhere) by only knowing the actor's identifier.

In our implementation, actors (and their queues) are identified by a unique string. This eases the actor knowing its name at any moment and simplifies addressing. We use Simple Queue Service (SQS) for our queues, as it is integrated with the other cloud systems and offers good performance. Each actor has its queue and waits its messages on it. To communicate with another actor, one only needs to know the other actor's name and send a message to the corresponding queue.

Atomicity. We could solve this issue by limiting function concurrency to one. This way, while there will not be more than one invocation of the same function running at the same time, we can still exploit the FaaS scalability and deploy a virtually infinite number of different functions concurrently.

There are several ways of doing so. AWS Lambda offers a configuration parameter for reserved concurrency [2]. Creating the function with a concurrency reservation of one, multiple invocations are throttled and only one function is executed at a time. However, throttling can suppose significant delays in executions.

State. Our approach uses a disaggregated storage service for persisting state. In this way, the state is retrieved from the store when the function is invoked and stored back before the invocation finishes. This allows persisting the actor for indefinite time at a reduced cost (the storage service's). In contrast, actors aren't always listening for messages, and need to be awaken, which has an extra latency. The approach is closely related to the passivation of actors.

In particular, we use Amazon DynamoDB. The service presents latencies inferior to 10ms for puts and gets of small strings.

Passivation. We are at a crossroads between complete passivation of actors to avoid extra billing and maintaining them running to avoid extra latency.

We propose a hybrid solution where actors' state is persisted on a storage system, which allows passivation of actors when they haven't received a message for a while, solving the extra billing problem. But, when invoked, they process all available messages on a single execution (minimizing extra latency). Once the actor is passivated, to process new messages, it would be necessary to invoke it again with a special event, in which case it will recover its state from the remote storage.

This approach requires an event system with two main properties. 1) To trigger a new execution when the actor's underlying function has been passivated. 2) When the function is running, notify it without enqueueing more functions invocations. Unfortunately, cloud providers do not offer this kind of event system as far as we know. Such is the case for AWS Lambda and SQS. A lambda function with the SQS trigger enabled, consumes all the available messages trying to enqueue invocations. As a result, when the function starts running and tries to receive messages, they are no longer available. This behavior, indeed, requires an external client that schedules the execution of actors. This client is notified when an actor passivates. After that, the client listens to the passivated actor queue, so that when the first message arrives, the client invokes the actor with the message in the payload.

3.1 Adapting actor's code to FaaS

Actors usually communicate calling each other's methods. Unfortunately, neither the underlying FaaS nor SQS support remote method calling. Thus, we implement an abstraction layer following the Active Object Pattern [3] to seamlessly handle the needed reflection.

Firstly, we inspect the code dependencies for every actor and zip them into the AWS Lambda deployment package. We find similar approaches in [4]. Then we serialize every remote actor method call and send it over SQS. And finally, we deserialize every SQS message and call the appropriate actor method.

4 Related work

Microsoft's Azure cloud offers features that seem to be the most helpful to build a serverless actor framework. Azure Durable Functions (ADF) [5] is an extension of Azure Functions and Azure WebJobs that lets you write stateful serverless functions. These special Functions as a Service can orchestrate other functions, providing state management, checkpointing, and sync/async function calling and chaining. ADF also includes eternal orchestration functions [6]. This feature intends on maintaining a long execution of the same function so that state is kept along. To avoid FaaS' execution time limit, functions detect when the end time is near and create a new invocation with the state as payload. Additionally, ADF provides singleton orchestrators functions, which ensure that only one invocation of them is ran at a time. When invoked, functions detect if there is already another instance of the same function running; in which case, the function ends immediately. Unfortunately, this singleton orchestrators are not atomic at the moment of writing [7], so they can't be used to build serverless actors.

In [8] we find an example of an actor model implementation using Azure Durable Functions. In this example, each orchestrator function (ADF) is an actor with a specific instance identifier. The author makes use of singleton orchestrators and their capability of waiting for external events to perform actor operations. An actor receives messages from others by waiting for external events. When the event is created, the orchestrator function awakens, processes the message, and calls itself with its state as payload by using the eternal orchestration feature. Orchestrator instances (actors) are created, queried or terminated through HTTP-triggered functions, which, in turn, raise special events specifying the instance identifier and the desired actor operation. However, these orchestrator functions don't guarantee message delivery, events can be lost depending on the function activity [9].

As for singleton functions, extra invocations that do not perform any work imply additional costs. Moreover, the implementation requires extra steps in function invocation (HTTP) that involve additional complexity and latency.

In addition, the state offered is limited for two reasons: it uses eternal functions, so the state is transferred by payload; and it uses an external service for persisting state when the function waits for message events. That is, ADF persists the state between different function activations by an Event Sourcing replay mechanism [5, 6]. Consequently, not only there is an overhead penalty when recreating the state, but also function executions must be deterministic [5].

In summary, ADF seems to offer great tools to build serverless actors. However, they still fall short to provide atomicity, state and invocation performance, and guaranteed event delivery.

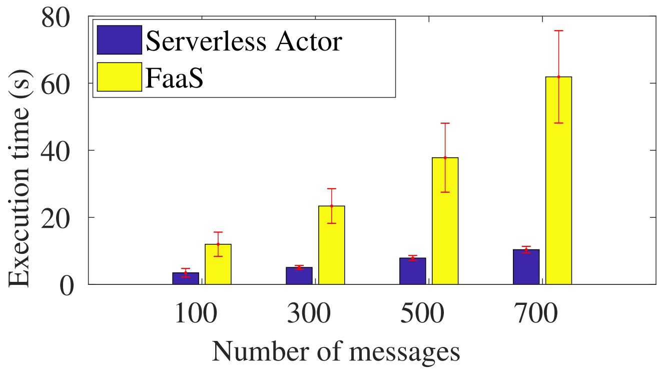

Figure 2: Serverless actor vs FaaS.

5 Evaluation

We evaluate our prototype and compare it to AWS Lambda to prove the usefulness of serverless actors. Many actor use cases require communication between actors, which are not possible to implement using AWS Lambda or any other FaaS available. Thus, we pick the counter example. A simple use case which can be implemented in both serverless actors and AWS Lambda.

The experiment consists of measuring the amount of time that takes to process different loads of messages. The implementations used in the experiment are as follows:

- Serverless actors: each actor's message will be sent through SQS. Then the message will be read by an already running actor, or a new actor invocation will handle the new message and the upcoming burst. Each message will modify a counter variable in the actor local memory.

- FaaS: each message implies a new function invocation, which in turn will make a read and update request to a remote DynamoDB.

In order to make a fair comparison, both implementations use a single concurrent lambda, 3 GB of memory, warm containers, and the same invocation process. The experiment was repeated 10 times.

Fig. 2 shows the average and standard deviation for the processing time length for both Serverless Actor and FaaS implementation. We see that our serverless actor prototype is up to 5.95× faster than the AWS

Lambda implementation. This is due to the high latency of the invocation and remote storage which happens for every request. We also observe a significant and increasing execution time deviation as a result of the Lambda throtling process, which seems to postpone lambda invocations if it receives frequent invocations.

6 Discussion

During the development of the design and implementation of serverless actors, we have found several restrictions that deserve a discussion. We believe this is a consequence of services' lack of built-in features needed for the application.

One important aspect is addressing. Current FaaS do not offer any kind of direct communication between functions once they are invoked. Therefore, remote services such as SQS must be used. While performant enough for some cases, the distributed nature of this service implies a suboptimal performance due to the latency. A serverless system with complete support for function's intercommunication would get rid of this performance penalty.

Passivation and event processing are the most important elements in a serverless actor system. Current offering allows two major approaches. The one with complete passivation can be thoroughly implemented without external support but has a great penalty in performance; whilst the hybrid solution for passivation that processes messages directly from a queue is performant, it still implies an external controller to wake up passivated actors.

The solution here is, once more, in the cloud service itself. We need run time support for functions capable of awakening when messages arrive to a queue, but able to read all available messages from that queue in a single execution, without requiring two different sources of events. This would suppose an internal manager that knows the function's state. This process would send messages to a running function or create a new invocation otherwise.

7 Conclusions

This paper studied the feasibility of building actors on top of the current serverless offering. We presented the main challenges that we must overcome and how we implemented a prototype successfully solving them. The evaluation compared our prototype to a serverless function and showed that our implementation processes up to 5.95× more messages than its FaaS counterpart.

We have observed that, indeed, serverless actors are possible. However, we also argue that run-time extensions to the serverless core would be necessary. In particular, we claim that serverless functions would need support for intercommunication and an event system capable of processing messages efficiently and triggering new functions when necessary.

References

[1] S. Tasharofi, P. Dinges, and R. E. Johnson, "Why do scala developers mix the actor model with other concurrency models?" in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 302–326.

[2] "AWS Lambda - Managing Concurrency," https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html, 2018.

[3] R. G. Lavender and D. C. Schmidt, "Active object – an object behavioral pattern for concurrent programming," 1995.

[4] J. Spillner, "Transformation of python applications into function-as-a-service deployments," *arXiv preprint arXiv:1705.08169*, 2017.

[5] "Durable Functions overview," https://docs.microsoft.com/azure/azure-functions/durable-functions-overview, 2018.

[6] "Eternal orchestrations in Durable Functions (Azure Functions)," https://docs.microsoft.com/azure/azure-functions/durable-functions-eternal-orchestrations, 2017.

[7] "Add singleton support for functions to ensure only one function running at a time," https://github.com/Azure/azure-functions-host/issues/912, 2018.

[8] "Actors in Serverless using Azure Functions," http://khaledhikmat.github.io/posts/2017-12-27-durable-functions, 2017.

[9] "External event message loss due to async activity in orchestration," https://github.com/Azure/azure-functions-durable-extension/issues/515, 2018.