Studying the feasibility of serverless actors

Álvaro Ruiz Ollobarren
Graduate Researcher at University Rovira I Virgili
alvaro.ruiz@urv.cat

Motivation

- Actors are a very popular way of simplifying complex applications.
- Based on stateful single threaded entities
- Popular frameworks:

Used in production in[1]:

Used in production in[2]:

[1] https://www.lightbend.com/case-studies#filter:akka

[2] http://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html

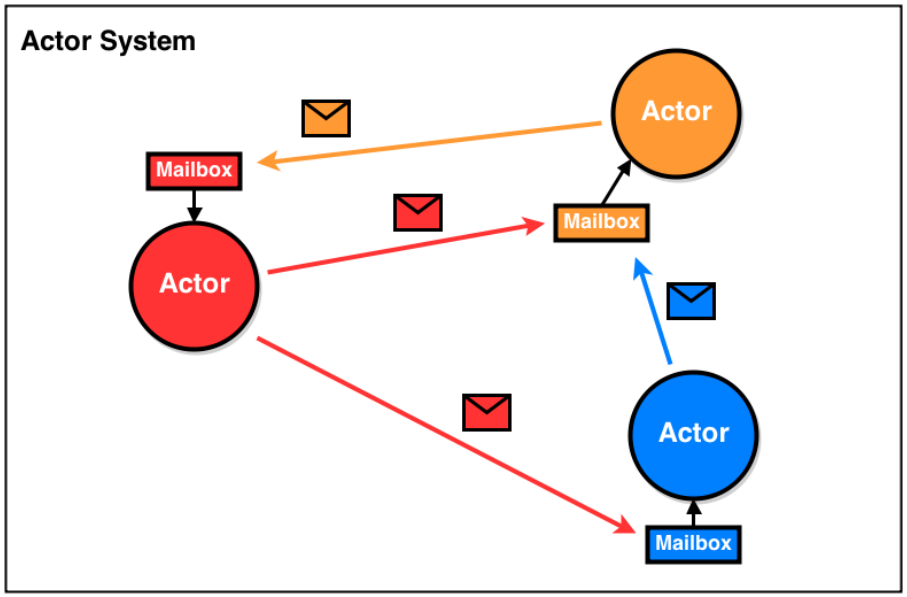
Background

- Not all applications have a straightforward migration to the serverless environment
- Actor model could benefit from 2 main aspects:
  - Billing
  - Scalability
- Simplest use case: Counter

Challenges

- **Addressing:**
  - Actors need to receive and send messages to other actors.
  - Currently FaaS only support invocation requests.

    ⟶ usage of external communication services is required.



Challenges

- **Atomicity:**
  - To maintain a consistent state, there cannot be more than one instance of the same actor executing at the same time.
  - Serverless functions scale automatically by spawning concurrent containers

     We need to limit function concurrency

Challenges

- **State:**
  - Actors are stateful
  - FaaS are stateless: consequent calls to the same function may not maintain previous state

     External storage services must be used

Challenges

- **Passivation:**
  - What to do when no messages arrive?
  - Fully event-driven approach: each actor invocation would imply a cloud function request to an external storage
  - Keeping the actor running approach: extremely expensive

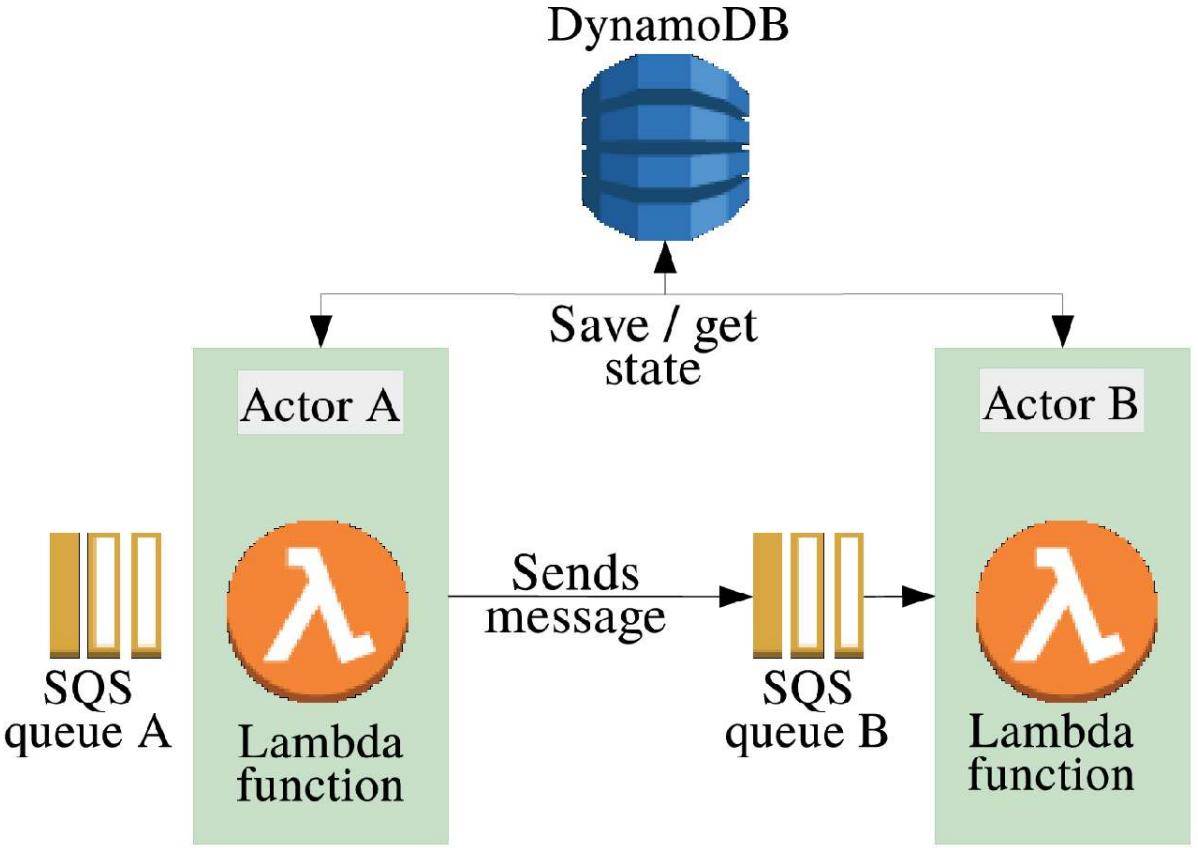
Challenges

- **Performance:**
  - The actor model must be functional, a minimum performance is mandatory.
  - This is a special requirement given the high network latencies of the remote components

Implementation

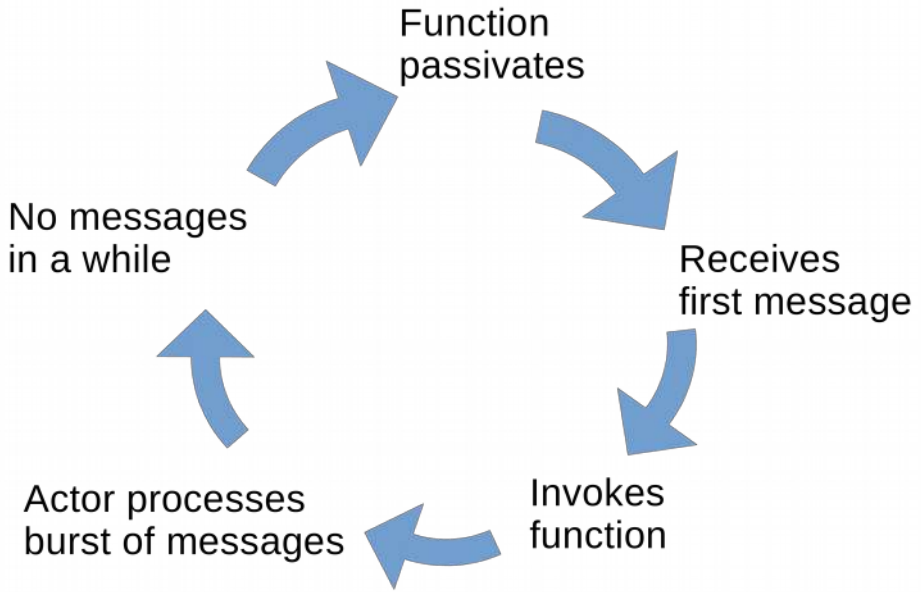
Architecture overview

- Addressing: SQS
- State: DynamoDB
- Atomicity: configuration



Implementation

- **Passivation:**
  - We propose a hybrid solution:
    - Actors process all available messages in a single execution until they don’t receive a message for a while.
    - Then, actors load the state into the DynamoDB and finish the execution.



Implementation

- **Passivation:**
  - 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.

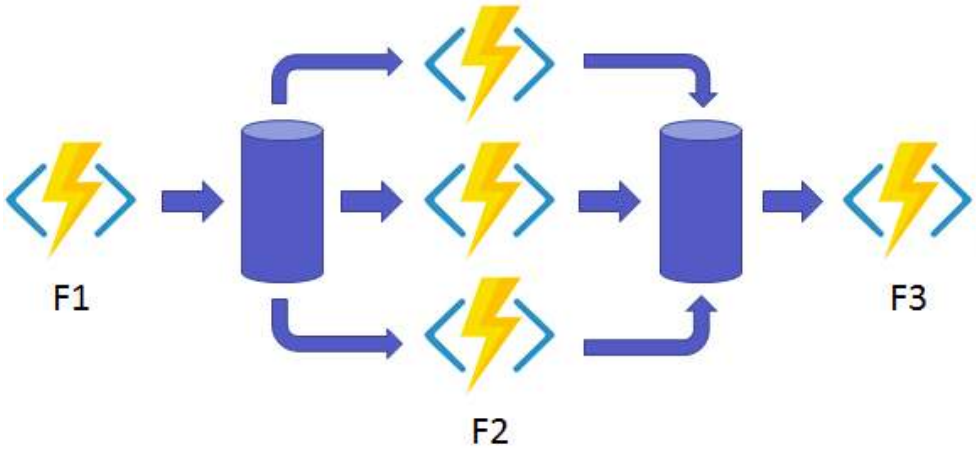
Implementation

- **Passivation:**
  - This behavior requires an external client to:
    - Schedule the execution of actors. This client is notified when an actor passivates.
    - Listen to the passivated actor queue to invoke the actor with the first message received.

Related work

Azure Durable Functions

- Are created, queried or terminated through HTTP-triggered functions.
- Can orchestrate other functions.
- They also offer:
  - Eternal orchestration functions
  - Singleton functions
  - Async events



Related work

Azure Durable Functions

- Are created, queried or terminated through HTTP-triggered functions. **Require 2 invocations**.
- Can orchestrate other functions.
- They also offer:
  - Eternal orchestration functions.
  - Singleton functions. **Not atomic**
  - Async events. **Lose events**

Evaluation

Serverless actors vs AWS Lambda

- 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.
  - Modify a counter variable in the actor local memory.

- FaaS:
  - Each message implies a new function invocation
  - Make a request to DynamoDB.

Evaluation

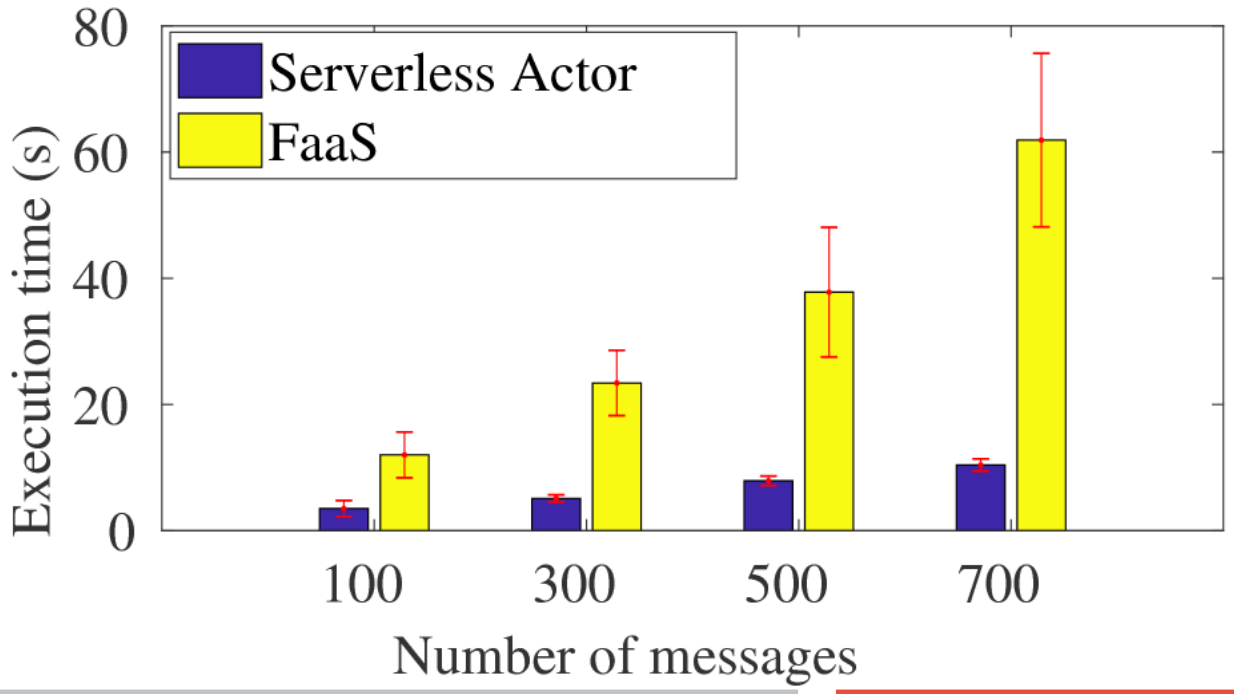
Serverless actors vs AWS Lambda

- Single concurrent lambda
- 3 GB of memory
- Warm containers
- Same invocation process

Evaluation

Serverless actors vs AWS Lambda

- Up to 5.95 X faster
- Smaller deviation



Discussion

- **Addressing**
  - SQS limitations

     Built-in support for lambda intercommunication

- **Passivation**
  - Events limitations

     Run time support for functions capable of awakening when messages arrive to a queue, but able to read all available messages from that queue

Conclusion

- Serverless actors are possible!
- Our prototype processes up to 5.95× more messages than its FaaS counterpart
- However, we also argue that run-time extensions to the serverless core would be necessary:
  - Support for intercommunication
  - Event system capable of processing messages efficiently and triggering new functions when necessary

Code and tests available at: https://github.com/danielBCN/faasactors