Scaling functional multi-agent computations with reactive streams

...

Daniel Krzywicki, Lambda Days 2018
@kierdeja, daniel.krzywicki@fabernovel.com

Motivation

Evolutionary Multi-Agent Systems

EMAS are hybrid metaheuristics (a.k.a optimization algorithms) which combine multi-agent systems with evolutionary algorithms.

They consist in evolving a population of agents without a central selection mechanism.

Selective pressure is designed to be an emergent behavior which results from peer-to-peer agents interactions.

EMAS were shown to be efficient and effective in hard optimization problems.

Evolutionary Multi-Agent Systems

Here are the rules of a simple EMAS:

- Generate an initial population of agents
- Every agent owns a candidate solution to the optimization problem, with some fitness value
- Distribute a finite amount of discrete energy among agents
- Agents interact with each other in pairs:
    - if both have enough energy, they spawn new agents and split their energy among them
    - otherwise they fight by comparing fitness, and the winner takes some energy from the loser
- Agents die if they run out of energy

Selective pressure emerges from such interactions. The size of the population may vary, but the total energy in the system remains constant and ensures stability.

Concurrency in EMAS

Multi-Agent Systems are supposed to model highly concurrent processes, but the first EMAS implementations were mainly sequential, with drawbacks such as:

- not enough parallelism to efficiently use modern many-core hardware (trivial, coarse-grained parallelism like the island model is under optimal for such cases)
- losing the semantics of the underlying model

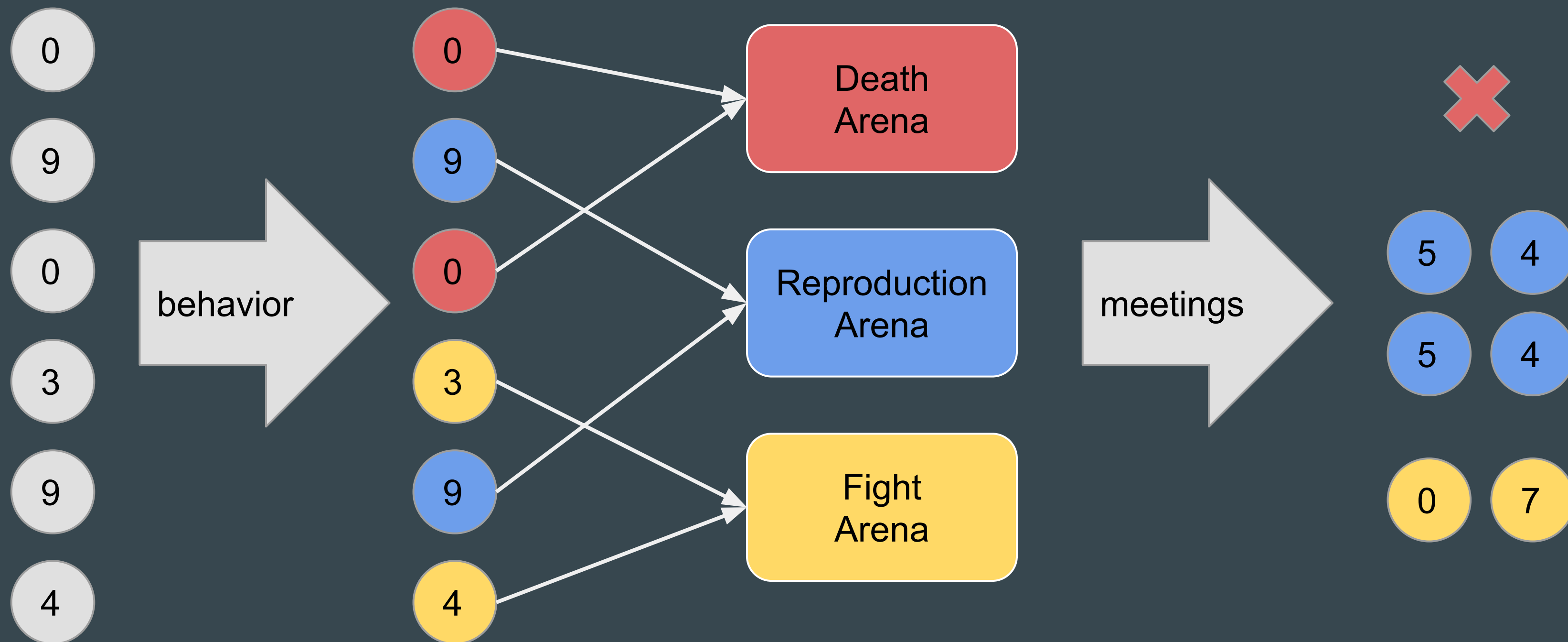
Meeting Arenas

In previous work, we proposed a pattern to decouple the semantics of the computation from the actual implementation of agents interactions, similar to the Map Reduce model.

The semantics of the algorithm are defined by two functions:

- a *behavior* function - based on its state, every agents choses some behavior
- a *meeting* function - agents with similar behavior meet in “arenas”, and as a result yield new or modified agents

Meeting Arenas



Concurrent execution models

Decoupling the semantics from the execution model allows to compare different approaches to concurrency and parallelism, and adapt them to the underlying hardware.

In previous work, we explored several such implementations:

- Fine grained actors - highly concurrent, but considerable overhead
- Parallel skeletons in fixed rate streams - highly parallel, but little concurrency

Concurrent execution models

In our recent work, we introduce a variable rate reactive stream implementation which combines the benefits of the previous approaches and generalizes them.

Our approach allows to precisely control the concurrency of the algorithm, while maintaining high performance and parallelism.

Reactive Streams

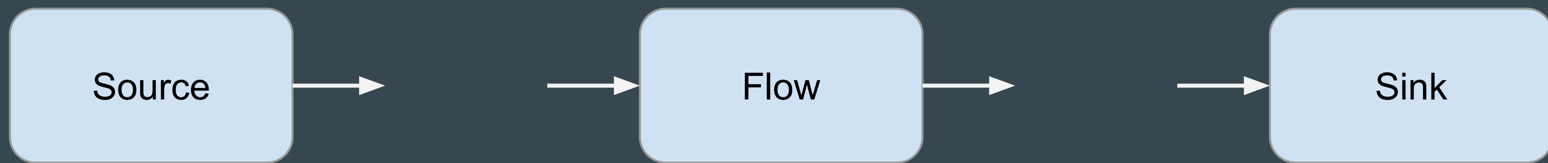
Reactive streams are a standard for asynchronous stream processing with non-blocking back pressure.

They allow to efficiently match the rate of producers and consumers in a stream, while ensuring bounded resources usage.

In our case, these properties will allow us to dynamically change the rate of agents in the stream to allows them to interact.

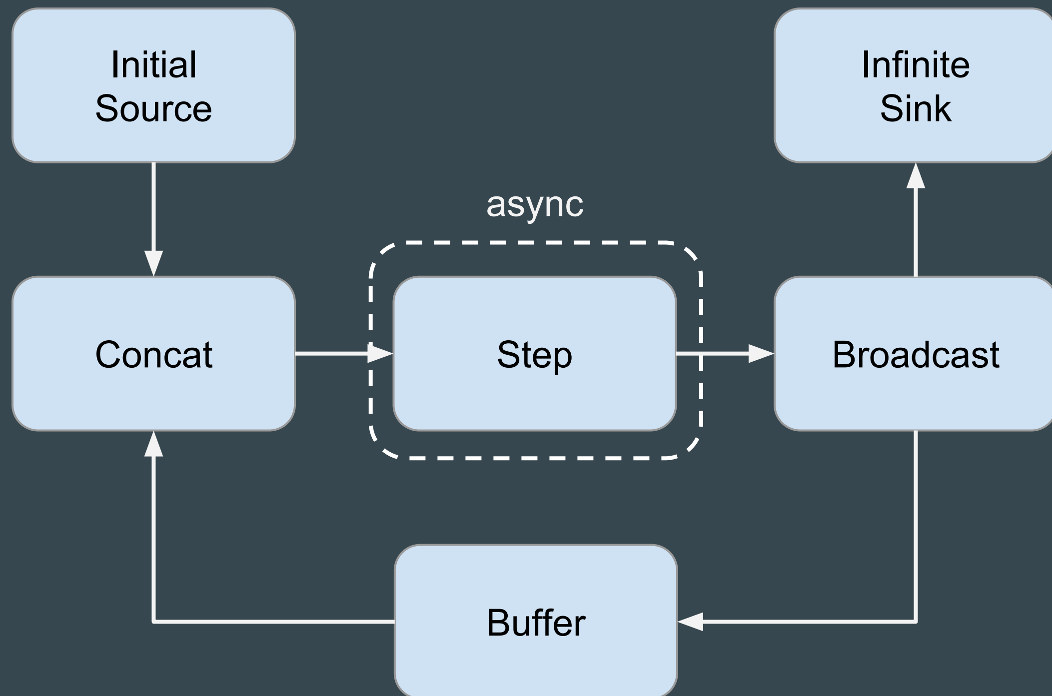
Our work uses the Akka Streams library, an implementation of the Reactive Streams standard.

Akka Streams

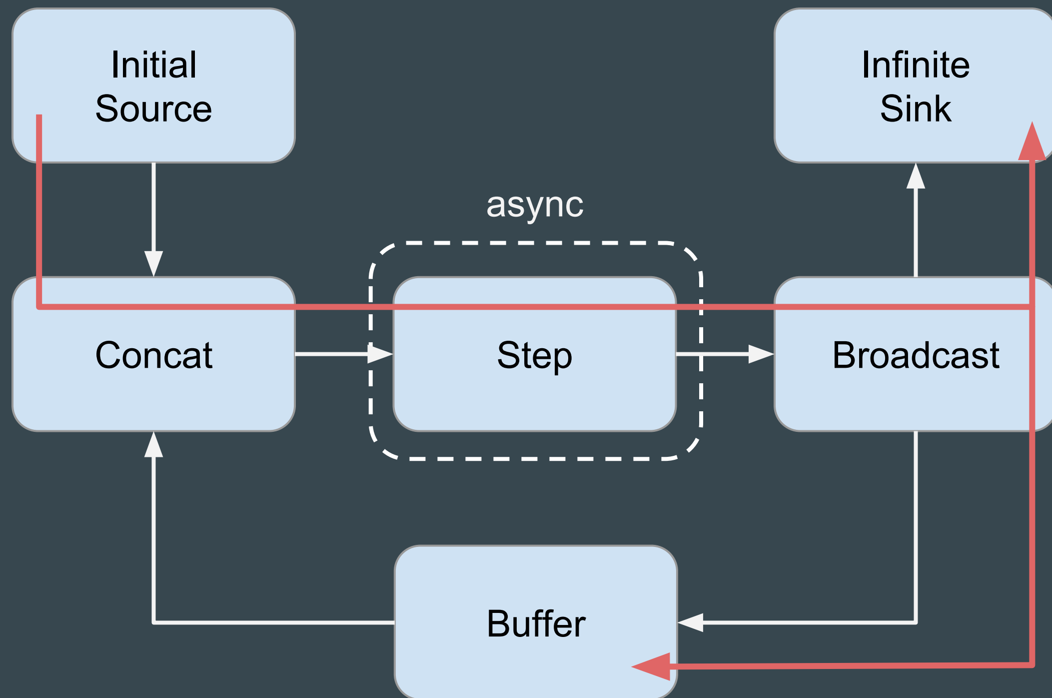


Modelling an iterative algorithm as a reactive stream

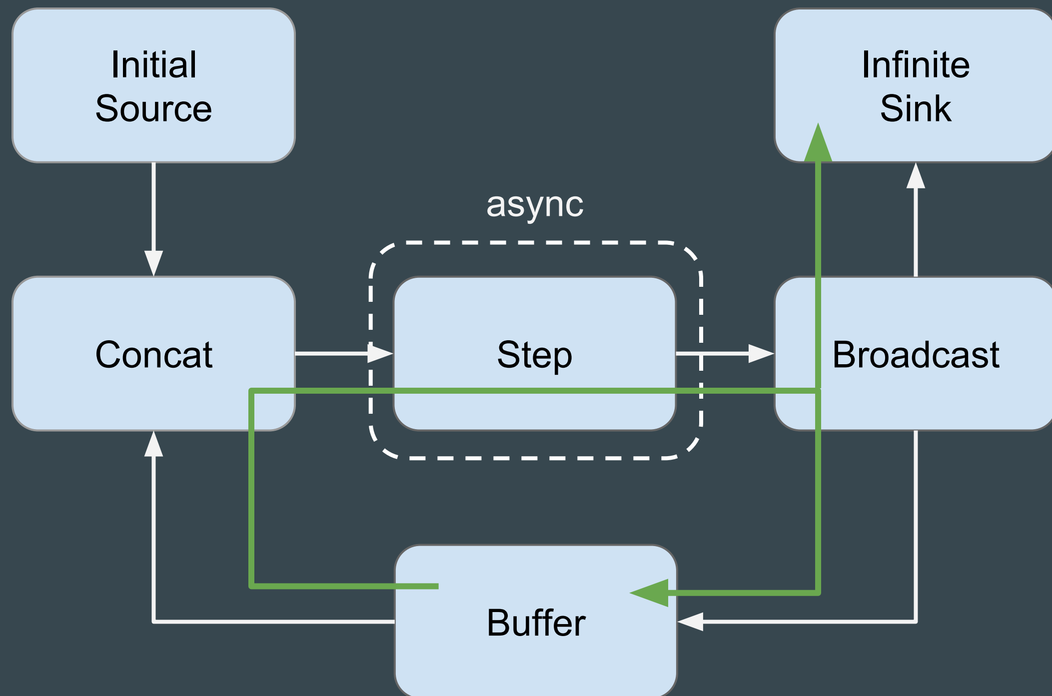
Looping Graph



Looping Graph



Looping Graph



Looping Graph

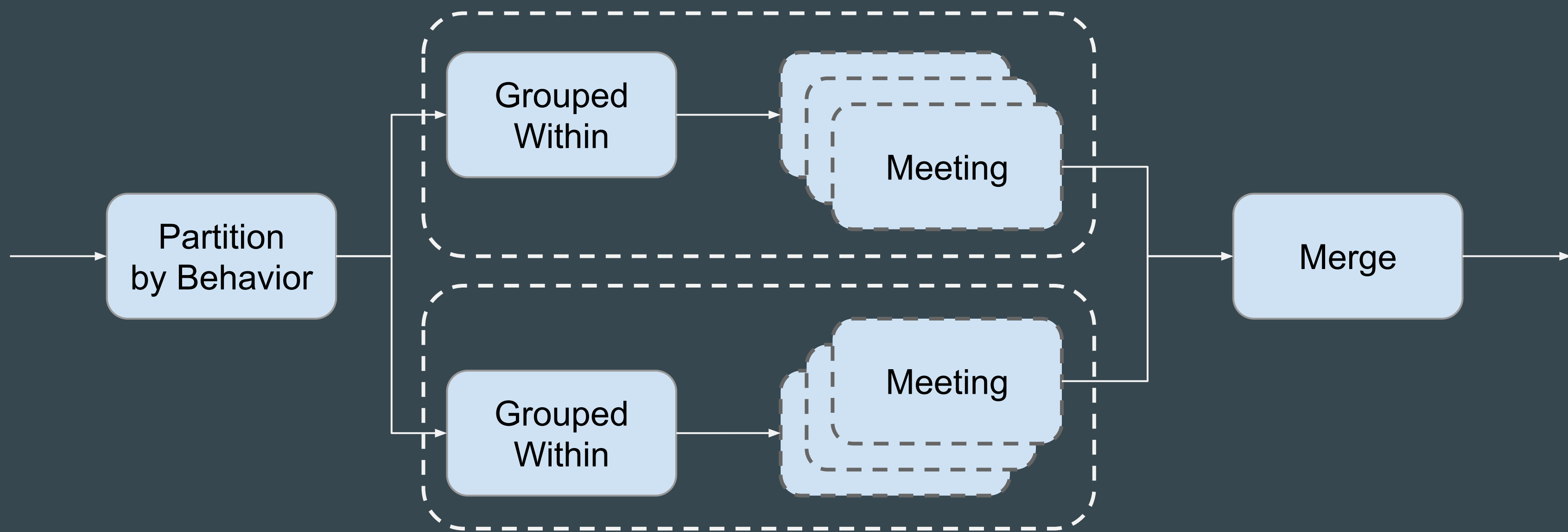
In order to ensure liveliness in the feedback loop, the buffer stage must never backpressure the broadcast stage.

In order to do so while ensuring boundedness, either:

- there must be an upper bound on the number of elements in the loop
- the buffer must be able to conflate incoming elements sufficiently fast

In our use case the first condition is satisfied, as the finite energy in the computation bounds the possible number of agents. The buffer size is chosen accordingly.

Arenas Flow



How to shuffle the elements of a stream?

So far, the algorithm is mostly deterministic. Agents meet in a FIFO pattern: the same agents will keep meeting with the same "neighbours" in the stream.

In general, stronger stochastic properties are necessary for metaheuristics to remain efficient.

However, we can no longer just capture the entire population and shuffle it.

Reservoir Sampling

Reservoir sampling is a technique for choosing a random sample from a stream of unknown (and possibly infinite) size.

It consists in maintaining a pool of elements of finite size - the reservoir. Whenever we observe a new element in the stream, we replace an existing element in the pool with some probability.

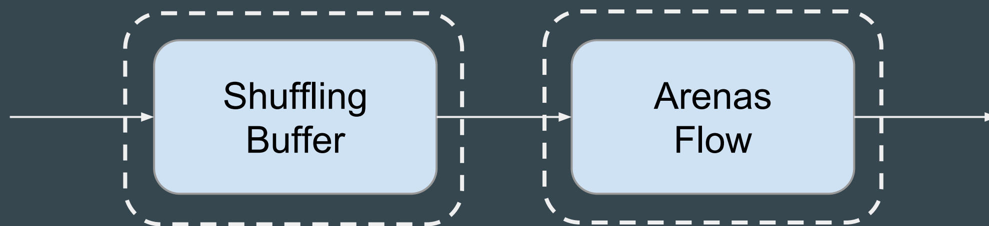
The probability changes with each subsequent element in such a way that when we stop the sampling, every element we have observed so far has an equal chance of ending in the final reservoir.

Shuffling buffer

We use a similar technique to shuffle the agents in the stream. We maintain an internal buffer and fill it with incoming elements.

When an output element is request by downstream, we choose an element from the buffer according to some policy. As observed from outside, the order of elements in the stream appears to change.

Different policies allow to simulate different execution models.



Random Shuffling Buffer

Policy: chose the output element at random from within a buffer of fixed size

- not every permutation is possible (we would have to consume the whole stream into the buffer)
- the bigger the size of the buffer, the more “forward” we can look upstream and the more shuffled the downstream seams
- every element has a finite probability of staying arbitrarily long in the buffer, so it can “jump” into the future.

Best Fitness Shuffling Buffer

Policy: chose the agent with the best fitness from within the buffer as the next output element

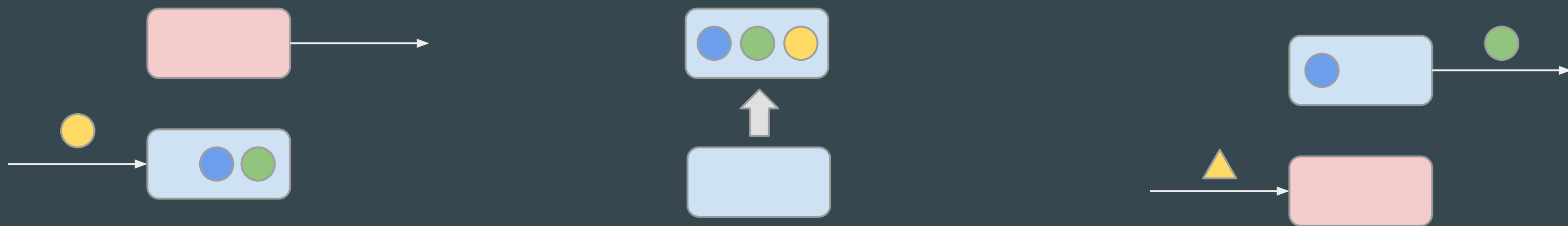
Annealed Shuffling Buffer

Policy: behave like a random buffer with probability ***p***, and like a best fitness buffer with probability ***1 - p***. The probability ***p*** decays exponentially with time.

Barrier Shuffling Buffer

Policy: Bi-state double buffering:

- first, don't emit output elements but buffer incoming ones until the total energy of the system is observed (we know all agents are in there)
- then, switch buffers and shuffle the content
- emit all elements before pulling input from the next generation



This policy allows to simulate a sequential, discrete step execution model.

Experimental Results

We applied the algorithm to two classical benchmark problems: the Rastrigin and the Ackley functions.

The detailed results are described in our next paper. The main conclusions they allow us to draw:

- the barrier buffer had similar characteristics as an iterative variant of the algorithm and gave the worst results
- the random buffer had similar characteristics as the previous actor-based implementation
- an annealed mix of the random and max buffers gave the best results

Future work

- Exploring other shuffling policies, including ones with more auto-adaptation
- Extending the model with support for multiple population and migrations (distributing multiple loops in a cluster)
- Testing the algorithm on real world optimization problems

Questions?