Julato: Function-as-a-Service (FaaS) platform

Roman Pleshkov

version 1.0

September 26, 2023

Abstract

The paper serves as a specification for the Julato project, focusing on its architecture and providing comprehensive documentation of key components such as event management, function execution, and scheduling, within the context of serverless architecture, particularly FaaS.

1 Introduction

Modern systems should be capable of processing billions of events, scaling effectively, ensuring the ordering of events, and implementing an appropriate consistency model. Julato is a product of distributed systems research over the years, with a strong emphasis on scalability, performance, and consistency. The central concepts that underpin Julato include event-sourcing, serverless functions, and transactional causal consistency.

2 Prerequisites

Julia will serve as the primary language for both development and the code snippets within the paper.

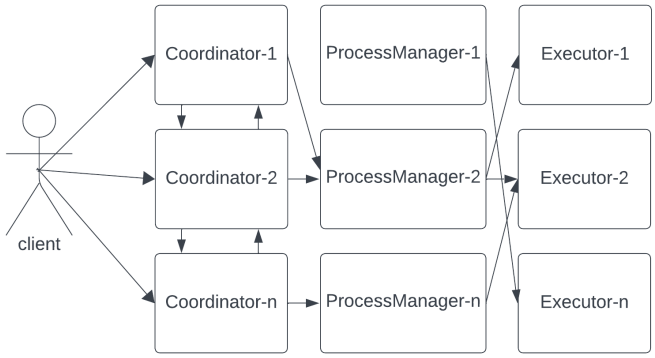

Figure 1: Pitr

3 Architecture

3.1 Main Components

The system consists of the following main components: Coordinator, ProcessManager and Executor.

3.1.1 Coordinator

The coordinator is pivotal within Julato, bearing responsibility for several key functions:

1. **Acting as an Entry Point**: serves as an entry point or gateway for clients through request-response dialog;
2. **Event Routing**: directs events to their respective ProcessManagers;
3. **ProcessManager Management**: oversees ProcessManagers, dynamically allocating new instances as needed to support autoscaling;
4. **Event Storage**: stores events in the **EventStore**. Creates checkpoints or snapshots of the **EventLog** at specific points in time;
5. **Transaction Management:** manages transactions within the system (commit/rollback);

6. **Causal Dependency Graph (CDG) Management**: constructs and maintains a causal graph of events;

3.2 ProcessManager

There are two main concepts that define a process: **behavior** and **state**:

```
mutable struct Process{State}
    state::State
    handler::Any
end
```

Listing 1: Process definition

The reason why **Process** struct is mutable will be explained later.

3.2.1 State

State is a type parameter. It serves as a placeholder for a specific data type that will be determined when an instance of the Process class is instantiated.

3.2.2 Handler

Handler is a function $(Event, State) => State$ that expects two parameters *Event* and *State* and returns a new instance of *State*.

```
strProcess = Process{String}("text", (state, event) -> state)
intProcess = Process{Integer}(1, (state, event) -> state)
```

Listing 2: Process instances

The Handler function doesn't have to be pure and can have side effects; however, it's discouraged to modify the input state. Ideally, the State type should be immutable but is not required.

The *event* parameter can be of any type, requiring a mechanism to determine the function's behavior based on specific types. Julia provides **isa** function to test if an object is of a given type.

```
process = Process{Any}(nothing, (state, event) -> begin
    if isa(event, String)
        println("this is text: $(event)")
    elseif isa(event, Int)
        println("this is integer: $(event)")
```

```
    else
        throw(DomainError(event, "unsupported event type"))
    end
    state
end)

process.handler(nothing, "hello world")
process.handler(nothing, 1)
process.handler(nothing, true)
```

Listing 3: Event matching by type

Output:

```
1 this is text: hello world
2 this is integer: 1
3 ERROR: DomainError with true:
4 unsupported event type
```

An alternative approach involves using Julia's *function-like* object feature to make Process objects callable. We can create a custom macro *@handler* which will simplify the process of adding handlers to a Process. First, let's define a Process struct without a 'handler' property:

```
mutable struct Process{State}
    id::String
    state::State
end
```

Next, we define a **@handler** macro.

```
macro handler(ex::Expr)
    sig = ex.args[1]
    body = ex.args[2]
    eventType = sig.args[1].args[2]
    params = sig.args[2]
    eventParam = nothing
    stateParam= nothing
    if isa(params.args[1], Symbol)
        eventParam = params.args[1]
    else
        eventParam = Symbol(params.args[1].args[1])
    end
```

```
    if isa(params.args[2], Symbol)
        stateParam = params.args[2]
    else
        stateParam = Symbol(params.args[2].args[1])
    end
    quote
        function (self::Process)($eventParam::$eventType, $stateParam)
            $body
        end
    end
end
```

Finally, we have the capability to specify handlers for various event types.

```
# Event
struct Say
    text::String
end

# Handler for event Say
@handler Process{Say}::(event::Say, state::Int) = begin
    println("handle: ", event, ", state: ", state)
    return state + 1
end

p = Process{Integer}("id_0", 0)
println("new state: ", p(Say("text"), p.state))
```

Output:

```
handle: Say("text"), state: 0
new state: 1
```

It is worth noting that parameter types are optional, i.e. Using Julia's *methods* util function we can list methods defined for object p:

```
methods(p)
```

Output:

```
(# 1 method for callable object:
 [1] (var"#5#self"::Process)(event::Say, state))
```

We can use Julia's mechanism of method dispatch to write *adapter* handlers.

```
@handler Process{Say}::(event::Say, state::Int) = begin
    println("handle: ", event, ", state: ", state)
    return state + 1
end

@handler Process{Array{UInt8}}::(event, state) = begin
    return self(Say(String(event)), state)
end

p = Process{Integer}("id_0", 0)
println("new state: ", p(Vector{UInt8}("text"), p.state))

methods(p)
```

Output:

```
handle: Say("text"), state: 0
new state: 1
# 2 methods for callable object:
 [1] (var"#5#self"::Process)(event::Say, state)
 [2] (var"#9#self"::Process)(event::Array{UInt8}, state)
```