Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Real Time Programming

Laboratory work #1

Stream Processing with Actors

Author:
Popa Eugeniu
std. gr. FAF-202

Supervisor:
Alexandru Osadcenco

Chișinău 2023

1 P1W1

Minimal Task Initialize a VCS repository for your project.

I have created a repository on GitHub.

Minimal Task Write an actor that would read SSE streams. The SSE streams for this lab are available on Docker Hub at alexburlacu/rtp-server, courtesy of our beloved FAFer Alex Burlacu.

```
defmodule Reader do
  use GenServer

  def start_link(url) do
    GenServer.start_link(__MODULE__, url)
  end

  def init(url) do
    IO.puts "Connecting to stream"
    HTTPoison.get!(url, [], recv_timeout: :infinity, stream_to: self())
    {:ok, nil}
  end

  def handle_info(%HTTPoison.AsyncChunk{chunk: ""}, url) do
    HTTPoison.get!(url, [], recv_timeout: :infinity, stream_to: self())
    {:noreply, url}
  end

  def handle_info(%HTTPoison.AsyncChunk{chunk: chunk}, state) do
    [_, data] = Regex.run(~r/data: ({.+})\n\n$/, chunk)
    case Poison.decode(data) do
      {:ok, result} -> send(Printer, result)
      {:error, _} -> nil
    end
    {:noreply, state}
  end

  def handle_info(%HTTPoison.AsyncStatus{} = status, state) do
    IO.puts "Connection status: #{inspect status}"
    {:noreply, state}
  end

  def handle_info(%HTTPoison.AsyncHeaders{} = headers, state) do
    IO.puts "Connection headers: #{inspect headers}"
    {:noreply, state}
  end

  def handle_info(_, state) do
    {:noreply, state}
  end
end
```

This code defines an Elixir module named Reader, which is also a GenServer process. The Reader process connects to a streaming API and receives real-time data as JSON messages. It then sends these messages to the Printer process.

The start_link function is used to start the Reader process, taking a url argument. It uses GenServer.start_link to start a new process for Reader with the module and url as arguments.

The init function is called when the Reader process is started and initializes the process. It prints a message to the console indicating that the process is connecting to the streaming API. It then uses HTTPoison.get! to connect to the streaming API and receive data as chunks of JSON.

The recv_timeout option is set to :infinity to ensure that the connection remains open. The stream_to: self() option ensures that the chunks of data are sent as messages to the Reader process. The function returns :ok, nil to indicate that initialization was successful.

The handle_info function is used to handle incoming messages to the Reader process. It has several cases to handle different types of messages. When the message contains an empty chunk of data, the function sends a new HTTP request to the streaming API to continue receiving data. When the message contains a non-empty chunk of data, the function uses a regular expression to extract the JSON data from the chunk. It then uses Poison.decode to parse the JSON data and sends the resulting map to the Printer process using send(Printer, result). If decoding fails, the function returns nil.

The function also has two cases to handle status and headers messages sent by the streaming API. These messages are printed to the console using IO.puts.

Finally, the function has a catch-all case that returns :noreply, state to indicate that it has finished handling the message and that no state changes are needed.

Overall, this module demonstrates how Elixir processes can communicate with each other using message passing and how Elixir can be used to handle real-time streaming data.

Minimal Task Create an actor that would print on the screen the tweets it receives from the SSE Reader. You can only print the text of the tweet to save on screen space.

Main Task Continue your Printer actor. Simulate some load on the actor by sleeping every time a tweet is received. Suggested time of sleep – 5ms to 50ms. Consider using Poisson distribution. Sleep values / distribution parameters need to be parameterizable.

```
defmodule Printer do
  use GenServer

  def start_link(state) do
    GenServer.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(state) do
    {:ok, state}
  end

  def handle_info(data, state) do
    IO.puts("Received tweet: #{data["message"]["tweet"]["text"]} \n")
    min = 5
    max = 50
    lambda = Enum.sum(min..max) / Enum.count(min..max)
    Process.sleep(trunc(Statistics.Distributions.Poisson.rand(lambda)))
    {:noreply, state}
  end
end
```

The Printer module is a GenServer process that is responsible for receiving and printing tweets.

The start_link function is used to start the Printer process, taking a state as an argument. It uses GenServer.start_link to start a new process for Printer with the module, the state, and the name Printer assigned to the process.

In the init function, it returns :ok, state to indicate that the process has been started successfully and its initial state is the state provided as an argument.

The handle_info function is responsible for handling incoming data. It takes two arguments, the data to be handled and the current state of the process. It prints the tweet's text by accessing the message and tweet keys of the data map. It then generates a random delay using a Poisson distribution with a lambda value calculated from the average of the integers between min and max.

Finally, it returns :noreply, state to indicate that the process should continue running with its current state.

Main Task Create a second Reader actor that will consume the second stream provided by the Docker image. Send the tweets to the same Printer actor.

```
defmodule MainSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_args) do
    children = [
      {Printer, :start_link},
      %{
        id: :reader1,
        start: {Reader, :start_link, ["localhost:4000/tweets/1"]}
      },
      %{
        id: :reader2,
        start: {Reader, :start_link, ["localhost:4000/tweets/2"]}
      }
    ]
    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

The MainSupervisor module is a Supervisor that manages the lifecycle of two Reader processes and a Printer process.

The start_link function is used to start the MainSupervisor process, taking no arguments. It uses Supervisor.start_link to start a new process for MainSupervisor with the module and an empty list as arguments and assigns the name MainSupervisor to the process.

The init function is called when the MainSupervisor process is started and initializes the process. It defines the child processes that the supervisor will manage as a list of tuples and maps.

The first child process is an instance of the Printer module, started with the :start_link function. The second and third child processes are instances of the Reader module, with unique IDs of :reader1 and :reader2 respectively. They are started with the :start_link function and a URL parameter that specifies the location of the streaming API that they will connect to.

Finally, the Supervisor.init function is called with the list of child processes and a strategy of :one_for_one. This strategy means that if one child process crashes, only that process will be restarted and not the entire supervisor.

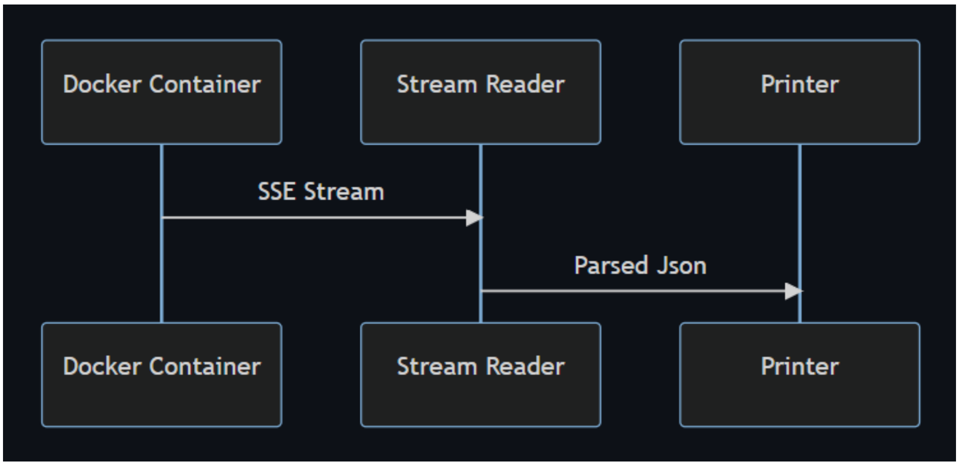

Figure 1: Message flow diagram

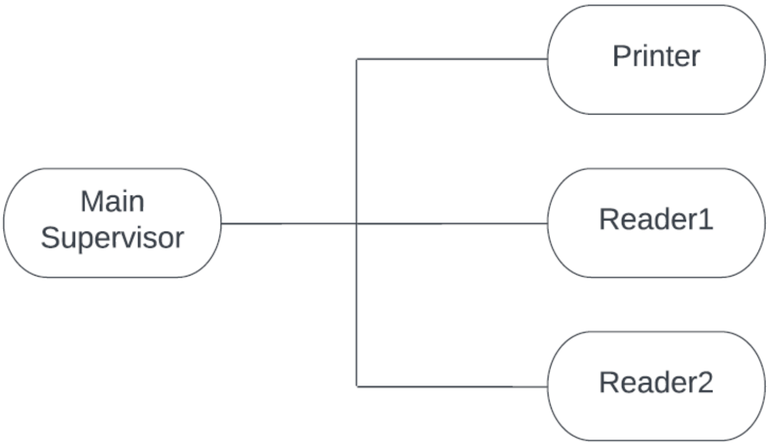

Figure 2: Supervision tree diagram

2 P1W2

Minimal Task Create a Worker Pool to substitute the Printer actor from previous week. The pool will contain 3 copies of the Printer actor which will be supervised by a Pool Supervisor. Use the one-for-one restart policy.

```
defmodule WorkerPool do
  use Supervisor

  def start_link(state) do
    Supervisor.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(_args) do
    children = [
      %{
        id: :Printer1,
        start: {Printer, :start_link, [:Printer1]},
        restart: :permanent
      },
      %{
        id: :Printer2,
```

```
        start: {Printer, :start_link, [:Printer2]},
        restart: :permanent
      },
      %{
        id: :Printer3,
        start: {Printer, :start_link, [:Printer3]},
        restart: :permanent
      },
    ]
    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

This is an implementation of a worker pool using the Supervisor behavior in Elixir. The WorkerPool module defines a supervisor with three child processes of type Printer. The start_link function is used to start the supervisor with a given state.

In the init function, a list of child processes is defined using the % syntax, with each child process having an id and a start option. The id is used to uniquely identify the child process within the supervisor, and the start option specifies how the child process is started. In this case, the start option specifies that a Printer process should be started using the start_link function with a unique name (:Printer1, :Printer2, or :Printer3) as an argument.

The restart option specifies how the child process should be restarted if it fails. In this case, it is set to :permanent, which means that the child process should always be restarted if it fails.

The Supervisor.init function is then used to initialize the supervisor with the list of child processes and a restart strategy of :one_for_one, which means that if a child process fails, only that process will be restarted.

Minimal Task Create an actor that would mediate the tasks being sent to the Worker Pool. Any tweet that this actor receives will be sent to the Worker Pool in a Round Robin fashion. Direct the Reader actor to sent it's tweets to this actor.

```
defmodule TaskMediator do
  use GenServer

  def start_link(state) do
    IO.puts("Starting task mediator")
    GenServer.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(state) do
    {:ok, state}
  end

  def handle_info(data, state) do
    printer = :"Printer#{state}"
    send(printer, data)
    state = state + 1
    if state >= 4 do
      {:noreply, 1}
    else
      {:noreply, state}
    end
  end
end
```

The TaskMediator module is a GenServer that acts as a mediator between a task-generating process and a pool of worker processes. When the TaskMediator process is started, it prints a message to the console indicating that it is starting up. The init function initializes the process by returning :ok, state.

The handle_info function is called when a message is received by the TaskMediator process. It takes the data received as an argument and sends it to one of three Printer processes, identified by their names Printer1, Printer2, and Printer3, which are dynamically generated based on the state variable. The state variable keeps track of which Printer process to send the data to next.

After sending the data, the state variable is incremented by 1. If the state variable is greater than or equal to 4, the function returns :noreply, 1, which signals to the caller that the TaskMediator process should terminate. Otherwise, the function returns :noreply, state, which tells the TaskMediator process to keep running and to send the next message to the Printer process identified by the new state value.

Main Task Continue your Worker actor. Occasionally, the SSE events will contain a "kill message". Change the actor to crash when such a message is received. Of course, this should trigger the supervisor to restart the crashed actor.

```
def handle_info(:kill, name) do
  IO.puts("Killing #{name}")
  exit(:kill)
  {:noreply, name}
end
```

This is a handle_info callback function for a GenServer. It receives two arguments: :kill, which is an atom, and name, which is any value. The purpose of this function is to handle a message sent to the GenServer with the name of a child process that needs to be terminated.

When the function is called with the :kill atom and the child process name, it prints a message to the console indicating that the process is being terminated. It then calls the exit/1 function with the :kill atom as an argument to terminate the child process with that name. Finally, it returns a tuple :noreply, name to indicate that the GenServer should continue running normally after handling the message.

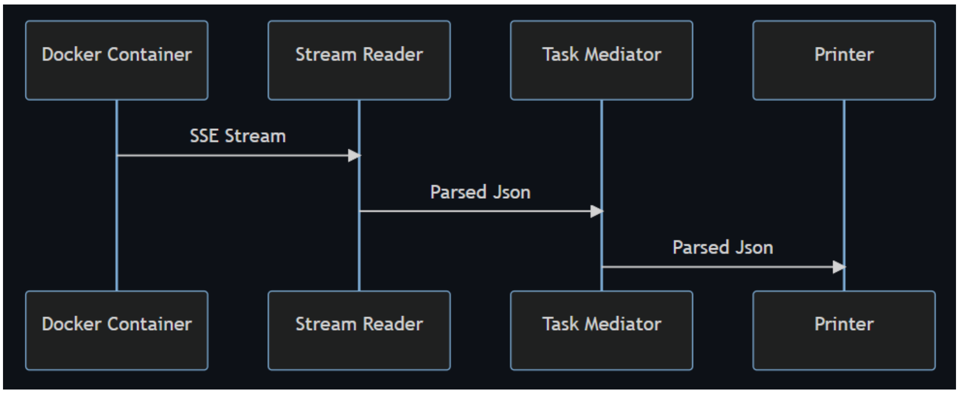

Figure 3: Message flow diagram

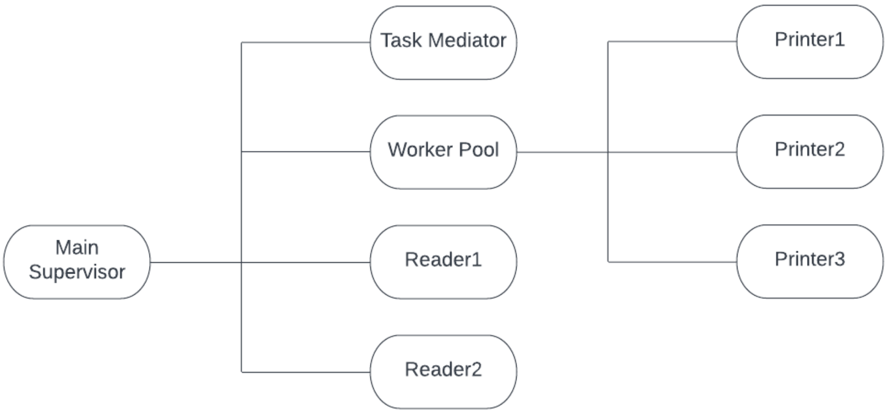

Figure 4: Supervision tree diagram

3 P1W3

Minimal Task Continue your Worker actor. Any bad words that a tweet might contain mustn't be printed. Instead, a set of stars should appear, the number of which corresponds to the bad word's length. Consult the Internet for a list of bad words.

```
defmodule Printer do
  use GenServer

  def start_link(name) do
    IO.puts("Starting #{name}")
    GenServer.start_link(__MODULE__, name, name: name)
  end

  def init(name) do
    {:ok, name}
  end

  def handle_info(:kill, name) do
    IO.puts("Killing #{name}")
    exit(:kill)
    {:noreply, name}
  end

  def handle_info(data, name) do
    {:ok, bad_words_json} = File.read("./lib/bad_words.json")
    {:ok, bad_words_dict} = Poison.decode(bad_words_json)
    bad_words = bad_words_dict["BadWords"]
    tweet_words = String.split(data["message"]["tweet"]["text"], " ", trim: true)
    censored_words = Enum.map(tweet_words, fn word ->
      word_lowercase = String.downcase(word)
      if Enum.member?(bad_words, word_lowercase) do
        String.duplicate("*", String.length(word))
      else
        word
      end
    end)
    censored_tweet = Enum.join(censored_words, " ")
    IO.puts("Received tweet: #{censored_tweet} \n")
    min = 5
    max = 50
```

```
    lambda = Enum.sum(min..max) / Enum.count(min..max)
    Process.sleep(trunc(Statistics.Distributions.Poisson.rand(lambda)))
    {:noreply, name}
  end
end
```

The Printer module defines a GenServer process that receives and processes tweets.

When the process is started, the start_link function is called, which sets up the process and initializes it with a given name. The init function then returns :ok, name to indicate that initialization was successful.

The handle_info function is called whenever the Printer process receives a message. If the message is :kill, the process exits with exit(:kill) and returns :noreply, name. If the message is a tweet (a JSON object), the function reads a list of bad words from a JSON file, censors the tweet by replacing bad words with asterisks, and prints the censored tweet to the console. The function then generates a random sleep time using a Poisson distribution with a mean value based on the range from 5 to 50, and sleeps for that amount of time. Finally, the function returns :noreply, name to indicate that it has processed the message and is ready for the next one.

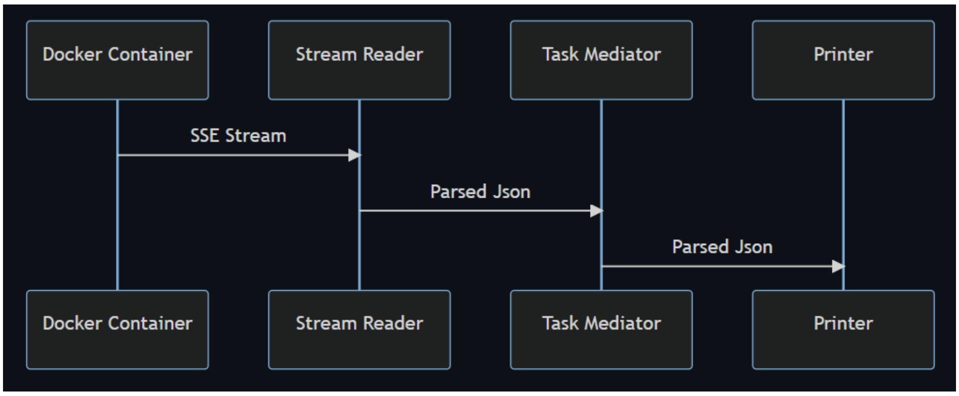

Figure 5: Message flow diagram

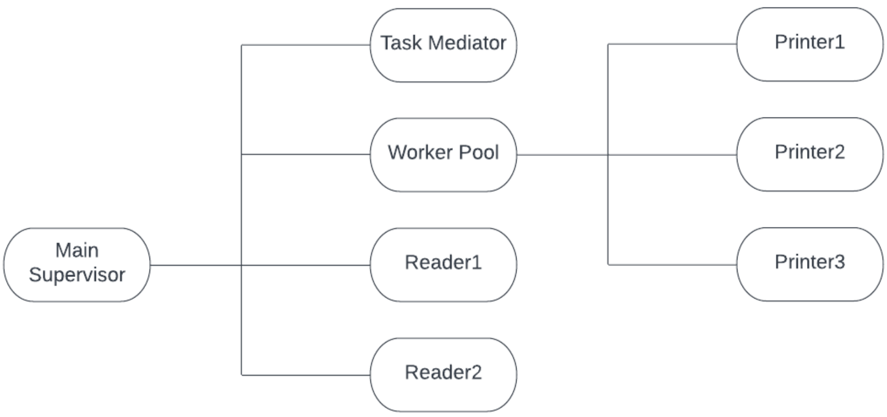

Figure 6: Supervision tree diagram

4 P1W4

Minimal Task Continue your Worker actor. Besides printing out the redacted tweet text, the Worker actor must also calculate two values: the Sentiment Score and the Engagement Ratio of the tweet. To compute the Sentiment Score per tweet you should calculate the mean of emotional scores of each word in the tweet text. A map that links words with their scores is provided as an endpoint in the Docker container. If a word cannot be found in the map, it's emotional score is equal to 0.

Main Task Break up the logic of your current Worker into 3 separate actors: one which redacts the tweet text, another that calculates the Sentiment Score and lastly, one that computes the Engagement Ratio.

Main Task Create 3 Worker Pools that would process the tweet stream in parallel. Each Pool will have the respective workers from the previous task.

```
defmodule Redactor do
  use GenServer

  def start_link(name) do
    IO.puts("Starting #{name}")
    GenServer.start_link(__MODULE__, name, name: name)
  end

  def init(name) do
    {:ok, name}
  end

  def handle_info(:kill, name) do
    IO.puts("Killing #{name}")
    exit(:kill)
    {:noreply, name}
  end

  def handle_info(data, name) do
    {:ok, bad_words_json} = File.read("./lib/bad_words.json")
    {:ok, bad_words_dict} = Poison.decode(bad_words_json)
    bad_words = bad_words_dict["BadWords"]
    tweet_words = String.split(data["message"]["tweet"]["text"], " ", trim: true)
    censored_words = Enum.map(tweet_words, fn word ->
      word_lowercase = String.downcase(word)
      if Enum.member?(bad_words, word_lowercase) do
        String.duplicate("*", String.length(word))
      else
        word
      end
    end)
    censored_tweet = Enum.join(censored_words, " ")
    IO.puts("Received tweet: #{censored_tweet}\n")
    min = 5
    max = 50
    lambda = Enum.sum(min..max) / Enum.count(min..max)
    Process.sleep(trunc(Statistics.Distributions.Poisson.rand(lambda)))
    {:noreply, name}
  end
end
```

```
defmodule RedactorPool do
  use Supervisor

```

```
  def start_link(state) do
    IO.puts("Starting redactor pool")
    Supervisor.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(_args) do
    children = [
      %{
        id: :Redactor1,
        start: {Redactor, :start_link, [:Redactor1]},
        restart: :permanent
      },
      %{
        id: :Redactor2,
        start: {Redactor, :start_link, [:Redactor2]},
        restart: :permanent
      },
      %{
        id: :Redactor3,
        start: {Redactor, :start_link, [:Redactor3]},
        restart: :permanent
      },
    ]
    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

```
defmodule RedactorTaskMediator do
  use GenServer

  def start_link(state) do
    IO.puts("Starting redactor task mediator")
    GenServer.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(state) do
    {:ok, state}
  end

  def handle_info(data, state) do
    redactor = :"Redactor#{state}"
    if Process.whereis(redactor) != nil, do: send(redactor, data)
    state = state + 1
    if state >= 4 do
      {:noreply, 1}
    else
      {:noreply, state}
    end
  end
end
```

Redactor: This is a GenServer that redacts bad words from tweets. It takes in tweets in JSON format, redacts the bad words, and sleeps for a random amount of time between 5 and 50 seconds. It is meant to be used as part of a larger application that processes tweets.

RedactorPool: This is a Supervisor that manages a pool of three Redactor processes. It starts them and restarts them if they fail, using a "one for one" strategy.

RedactorTaskMediator: This is a GenServer that acts as a mediator between the application's main process and the RedactorPool. It receives tweets in JSON format, determines which Redactor process to send them to based on a round-robin scheduling algorithm, and sends them to that process. If a Redactor process is not available, it skips it and tries the next one.

```
defmodule SentimentScorer do
  use GenServer

  def start_link(name) do
    IO.puts("Starting #{name}")
    GenServer.start_link(__MODULE__, name, name: name)
  end

  def init(name) do
    {:ok, name}
  end

  def handle_info(:kill, name) do
    IO.puts("Killing #{name}")
    exit(:kill)
    {:noreply, name}
  end

  def handle_info(data, name) do
    tweet_words = String.split(data["message"]["tweet"]["text"], " ", trim: true)
    emotion_values_url = "localhost:4000/emotion_values"
    emotion_values = HTTPoison.get!(emotion_values_url).body
    emotion_values_strings = String.split(emotion_values, ["\n", "\t"]) |> Enum.map(&
    String.replace(&1, "\r", ""))
    emotion_values_pairs = Enum.chunk_every(emotion_values_strings, 2)
    emotion_values_dict = emotion_values_pairs |> Enum.reduce(%{}, fn [word, score],
    acc -> Map.put(acc, word, score) end)
    values = Enum.map(tweet_words, fn word ->
      case Map.get(emotion_values_dict, word) do
        nil -> 0
        val -> String.to_integer(val)
      end
    end)
    sum_of_values = Enum.reduce(values, 0, fn x, acc -> x + acc end)
    sentiment_score = sum_of_values / length(values)
    IO.puts("Sentiment score: #{sentiment_score}\n")
    min = 5
    max = 50
    lambda = Enum.sum(min..max) / Enum.count(min..max)
    Process.sleep(trunc(Statistics.Distributions.Poisson.rand(lambda)))
    {:noreply, name}
  end
end
```

```
defmodule SentimentScorerPool do
  use Supervisor

  def start_link(state) do
    IO.puts("Starting sentiment scorer pool")
    Supervisor.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(_args) do
    children = [
      %{
        id: :SentimentScorer1,
        start: {SentimentScorer, :start_link, [:SentimentScorer1]},
```

```
        restart: :permanent
      },
      %{
        id: :SentimentScorer2,
        start: {SentimentScorer, :start_link, [:SentimentScorer2]},
        restart: :permanent
      },
      %{
        id: :SentimentScorer3,
        start: {SentimentScorer, :start_link, [:SentimentScorer3]},
        restart: :permanent
      },
    ]
    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

```
defmodule SentimentScorerTaskMediator do
  use GenServer

  def start_link(state) do
    IO.puts("Starting sentiment scorer task mediator")
    GenServer.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(state) do
    {:ok, state}
  end

  def handle_info(data, state) do
    sentiment_scorer = :"SentimentScorer#{state}"
    if Process.whereis(sentiment_scorer) != nil, do: send(sentiment_scorer, data)
    state = state + 1
    if state >= 4 do
      {:noreply, 1}
    else
      {:noreply, state}
    end
  end
end
```

The SentimentScorer module is responsible for processing a single tweet and calculating its sentiment score. It uses an emotion values dictionary that it retrieves from a local web server to assign scores to individual words in the tweet. Once it has calculated the sentiment score, it sleeps for a random amount of time before returning control to the supervisor that manages it.

The SentimentScorerPool module defines a supervisor process that manages a pool of SentimentScorer processes. It starts three child processes, each with a unique ID and restart policy, and initializes the supervisor with these child processes using the Supervisor.init/2 function with the :one_for_one strategy. This ensures that if any of the child processes fail, only that child process will be restarted, not the entire pool.

The SentimentScorerTaskMediator module acts as a mediator between the client and the pool of SentimentScorer processes. It receives tweet data from the client, and then forwards it to the next available SentimentScorer process in the pool. It keeps track of the state of the pool, making sure that each process is used in a round-robin fashion. If a process is not available, it moves on to the next one until it finds an available process.

Together, these three modules form a simple sentiment scoring system that can be used to analyze large amounts of tweets and assign sentiment scores to each one. By managing the

SentimentScorer processes in a pool using the SentimentScorerPool module, the system can handle failures and ensure that there are always processes available to process incoming tweet data. The SentimentScorerTaskMediator module provides a way for clients to interact with the pool of SentimentScorer processes in a way that is transparent to them, making it easy to integrate the system into other applications.

```
defmodule EngagementRationer do
  use GenServer

  def start_link(name) do
    IO.puts("Starting #{name}")
    GenServer.start_link(__MODULE__, name, name: name)
  end

  def init(name) do
    {:ok, name}
  end

  def handle_info(:kill, name) do
    IO.puts("Killing #{name}")
    exit(:kill)
    {:noreply, name}
  end

  def handle_info(data, name) do
    favorite_count = data["message"]["tweet"]["retweeted_status"]["favorite_count"] || 0
    retweet_count = data["message"]["tweet"]["retweeted_status"]["retweet_count"] || 0
    followers_count = data["message"]["tweet"]["user"]["followers_count"]
    engagement_ratio =
      if followers_count == 0,
        do: 0,
        else: (favorite_count + retweet_count) / followers_count
    IO.puts("Engagement ratio: #{engagement_ratio}\n")
    min = 5
    max = 50
    lambda = Enum.sum(min..max) / Enum.count(min..max)
    Process.sleep(trunc(Statistics.Distributions.Poisson.rand(lambda)))
    {:noreply, name}
  end
end
```

```
defmodule EngagementRationerPool do
  use Supervisor

  def start_link(state) do
    IO.puts("Starting engagement rationer pool")
    Supervisor.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(_args) do
    children = [
      %{
        id: :EngagementRationer1,
        start: {EngagementRationer, :start_link, [:EngagementRationer1]},
        restart: :permanent
      },
      %{
```

```
        id: :EngagementRationer2,
        start: {EngagementRationer, :start_link, [:EngagementRationer2]},
        restart: :permanent
      },
      %{
        id: :EngagementRationer3,
        start: {EngagementRationer, :start_link, [:EngagementRationer3]},
        restart: :permanent
      },
    ]
    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

```
defmodule EngagementRationerTaskMediator do
  use GenServer

  def start_link(state) do
    IO.puts("Starting engagement rationer task mediator")
    GenServer.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(state) do
    {:ok, state}
  end

  def handle_info(data, state) do
    engagement_rationer = :"EngagementRationer#{state}"
    if Process.whereis(engagement_rationer) != nil, do: send(engagement_rationer,
   data)
    state = state + 1
    if state >= 4 do
      {:noreply, 1}
    else
      {:noreply, state}
    end
  end
end
```

The EngagementRationer module defines a GenServer process that calculates the engagement ratio of a Twitter post and sleeps for a random amount of time following a Poisson distribution before sending a response back to the main process. The EngagementRationerPool module uses the Supervisor behavior to start and supervise a pool of three EngagementRationer processes. Finally, the EngagementRationerTaskMediator module defines a GenServer process that receives incoming messages and forwards them to one of the EngagementRationer processes based on a round-robin scheduling algorithm.

Overall, these three modules work together to create a fault-tolerant and scalable system for processing incoming Twitter data in parallel, by distributing the workload across multiple worker processes.

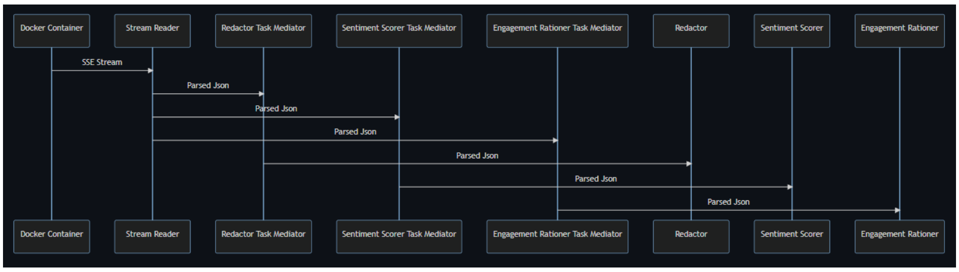

Figure 7: Message flow diagram

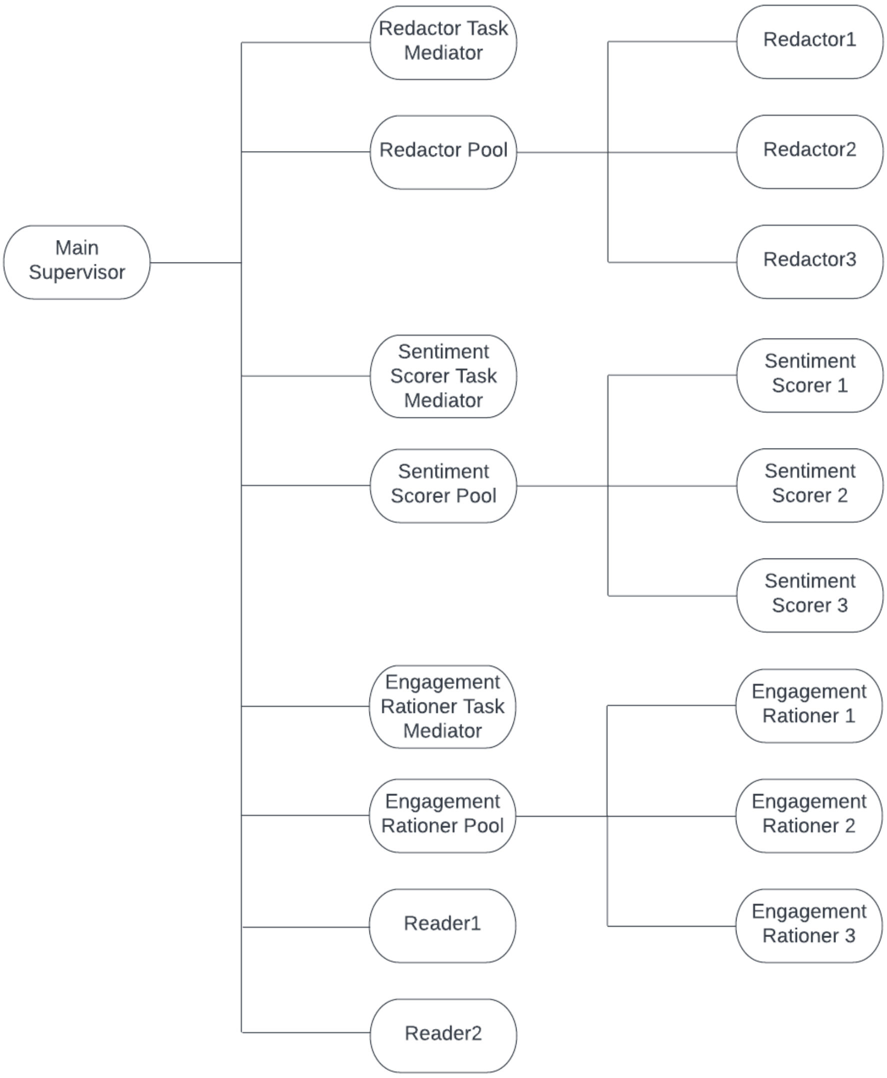

Figure 8: Supervision tree diagram

5 P1W5

Minimal Task Create an actor that would collect the redacted tweets from Workers and would print them in batches. Instead of printing the tweets, the Worker should now send them to the Batcher, which then prints them. The batch size should be parametrizable.

Main Task Continue your Batcher actor. If, in a given time window, the Batcher does not receive enough data to print a batch, it should still print it. Of course, the actor should retain any existing behaviour. The time window should be parametrizable.

```
defmodule Batcher do
  use GenServer

  def start_link({state, batch_size, timeout}) do
    IO.puts("Starting batcher")
    GenServer.start_link(__MODULE__, {state, batch_size, timeout}, name: __MODULE__)
  end

  def init({state, batch_size, timeout}) do
    Process.register(spawn_link(fn -> loop(timeout) end), :checker)
    {:ok, {state, batch_size}}
  end

  def handle_call(:get_state, _, state) do
    {:reply, state, state}
  end

  def handle_info(:set_state, {batch_size}) do
    {:noreply, {[], batch_size}}
  end

  def handle_info(tweet, {state, batch_size}) do
    new_state = state ++ [tweet]
    case length(new_state) == batch_size do
      true ->
        send(:checker, {:batch_full, new_state})
        {:noreply, {[], batch_size}}
      false ->
        {:noreply, {new_state, batch_size}}
    end
  end

  def loop(timeout) do
    receive do
      {:batch_full, state} ->
        IO.puts("Batcher is full. Printing the data...\n")
        print_tweets(state)
        loop(timeout)
    after
      timeout ->
        state = GenServer.call(Batcher, :get_state)
        send(Batcher, :set_state)
        IO.puts("Timeout reached. Printing the data...\n")
        print_tweets(state)
        loop(timeout)
    end
  end

  def print_tweets(tweets) do
    Enum.each(tweets, fn tweet -> IO.puts("Received tweet: #{tweet}\n") end)
  end
end
```

The code defines a Batcher module which is a GenServer that receives incoming tweets and batches them together until a predefined batch_size is reached. Once a batch is full, the Batcher sends the batch of tweets to a separate process, :checker, and resets its internal state.

The Batcher is initialized with a tuple of state, batch_size, timeout. The state is initially an empty list, batch_size determines how many tweets are needed for a full batch, and timeout is the amount of time the Batcher waits before sending any pending tweets to :checker.

When the Batcher receives a tweet via handle_info(tweet, state, batch_size), it adds the tweet to its internal state list and checks whether the state list has reached the batch_size. If the state list has reached the batch_size, it sends the list of tweets to :checker, and resets its internal state back to an empty list. If the state list has not reached the batch_size, it simply returns the updated state list.

The loop function is a separate process that runs in parallel and checks if the Batcher has received enough tweets to form a full batch. If loop receives a message :batch_full, state, it prints out the batch of tweets and then continues to wait for the next batch. If the loop function has not received a full batch within the timeout period, it sends a message to Batcher to request its current state. Once it receives the state, it sends a :set_state message back to the Batcher process to reset its internal state to an empty list, and then prints out the tweets in the batch.

The print_tweets function simply prints out each tweet in the list of tweets it is given.

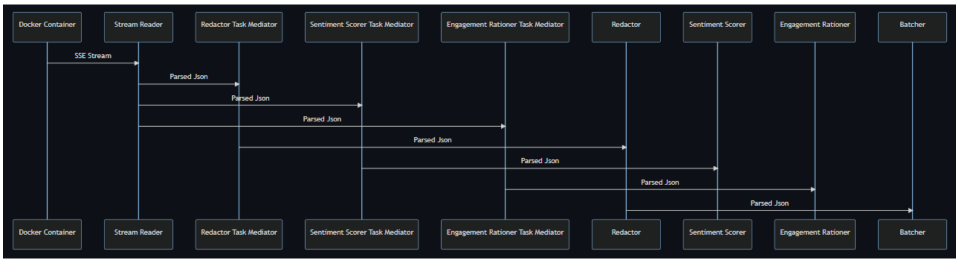

Figure 9: Message flow diagram

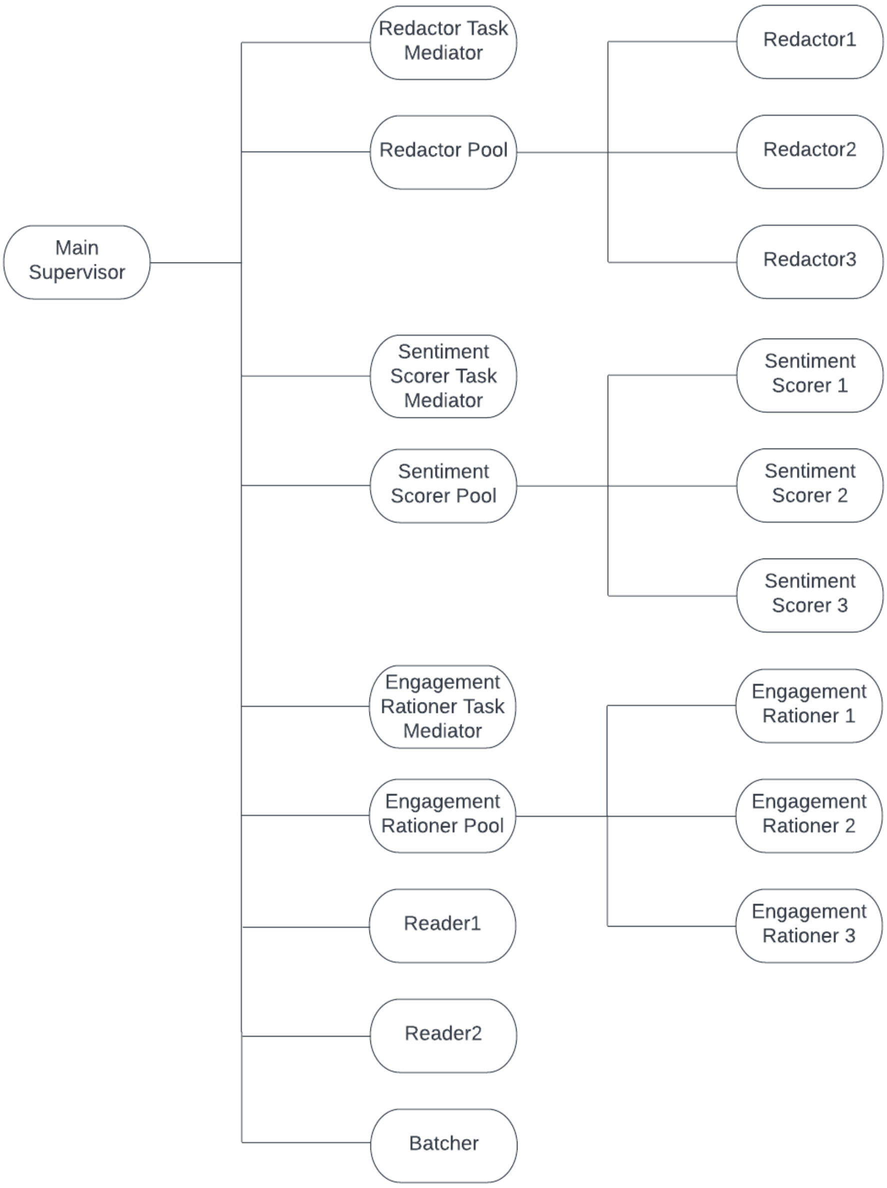

Figure 10: Supervision tree diagram

6 P1W6

Minimal Task Create a database that would store the tweets processed by your system.

Minimal Task Continue your Batcher actor. Instead of printing the batches of tweets, the actor should now send them to the database, which will persist them.

```
defmodule Database do
  use GenServer

  def start_link(state) do
    IO.puts("Database starting")
    GenServer.start_link(__MODULE__, state, name: __MODULE__)
  end

  def init(state) do
    :ets.new(:tweets, [:ordered_set, :protected, :named_table])
```

```
    {:ok, state}
  end

  def handle_call({:save_tweet, tweets}, _from, state) do
    new_state = Enum.reduce(tweets, state, fn tweet, acc ->
      :ets.insert(:tweets, {acc, tweet})
      acc + 1
    end)
    # if rem(new_state, 100) == 0 do
    #   IO.puts("State is now #{new_state}")
    #   print_database()
    # end
    {:reply, :ok, new_state}
  end

  def print_database do
    IO.puts("Database Table")
    table = :ets.tab2list(:tweets)
    Enum.each(table, fn {id, tweet} ->
      IO.puts("Id: #{id} Tweet: #{tweet}")
    end)
  end
end
```

This is a GenServer module named Database which provides the functionality to save tweets to an ETS table named :tweets.

The module has a start_link function which starts the GenServer process and returns its pid. It takes an initial state as its argument.

The init function is called when the process is started. Here, a new ETS table is created with the name :tweets and properties [ordered_set, protected, named_table].

The module provides a handle_call function which is called when a client sends a synchronous request to the server. Here, when the client sends the message :save_tweet, tweets to the server, it inserts each tweet in the ETS table using :ets.insert function. The new state of the server is returned as a reply to the client along with :ok.

The module also has a print_database function which prints the entire ETS table. It converts the ETS table to a list using :ets.tab2list, and then uses Enum.each to iterate through each entry in the list and print it to the console.

The commented-out code block shows a potential way to log and print the state of the server. It prints the current state of the server every time the number of tweets in the ETS table is a multiple of 100.

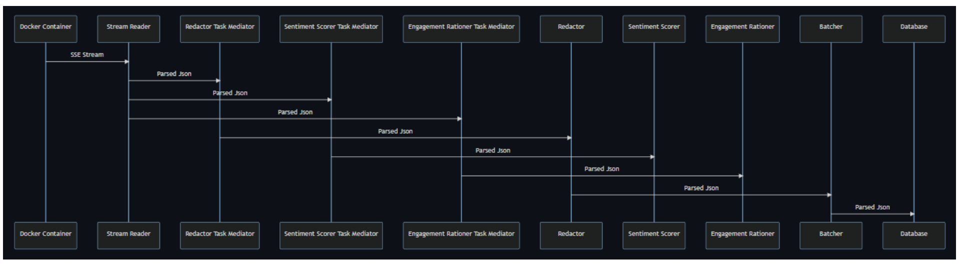

Figure 11: Message flow diagram

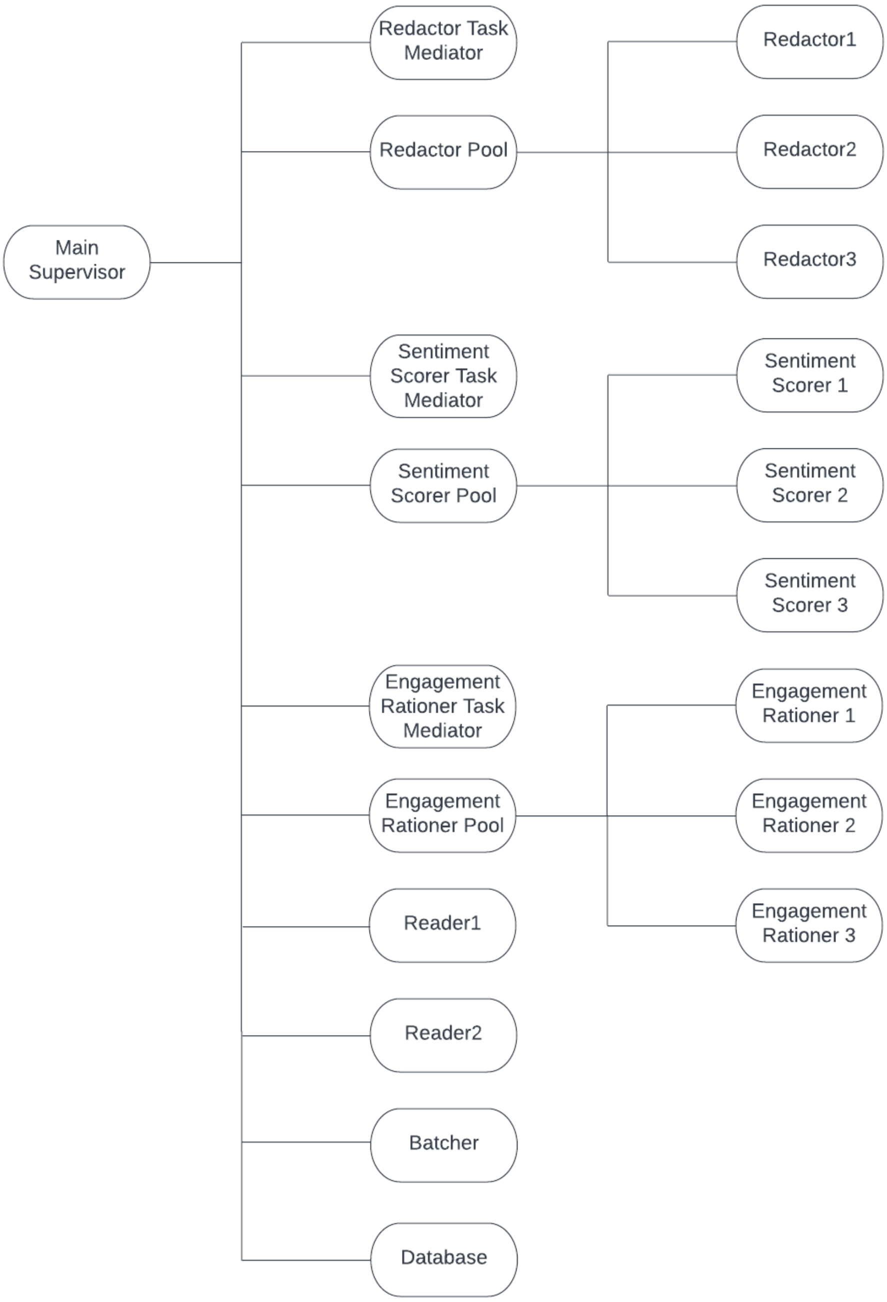

Figure 12: Supervision tree diagram

7 Conclusion

In summary, the goal of this project was to create a functional stream processing system that could handle complex applications. Two diagrams, the Message Flow Diagram and the Supervision Tree Diagram, were utilized to analyze message exchange between actors and monitor structures of the application. The project focused on stream processing with actors, a popular paradigm for building scalable and concurrent systems. With actors, the system could process streams of data in parallel, with each actor handling specific tasks or subsets of data, allowing for efficient resource utilization and real-time processing of large volumes of data. The project provided a valuable learning experience in stream processing with actors, helping the team develop a deeper understanding of the principles and best practices involved in building scalable and concurrent systems.