Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Real - Time Programming

Project 1

Stream Processing with Actors

Author:
Daniel Pogorevici
std. gr. FAF-202

Supervisor:
Alex Osadcenco

Chișinău 2023

1 Code implementation

Week 1

For the first week I created a reader actor which reads the SSE streams and sends them to the printer actor.

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

  def init(url) do
    HTTPoison.get!(url, [], recv_timeout: :infinity, stream_to: self())
    {:ok, url}
  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: "event: \"message\"\n\ndata: {\"message\": panic
    }\n\n"},
        url
      ) do
    {:noreply, url}
  end

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

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

Listing 1: Reader Actor.

Also I created an actor that would print on the screen the tweets it receives from the SSE Reader and I only print the text of the tweet to save on screen space and also simulating some load on the actor by sleeping every time a tweet is received.

```
def start_link(sleep_time) do
    GenServer.start_link(__MODULE__, {sleep_time}, name: __MODULE__)
  end

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

  def handle_info(json, {sleep_time}) do
    Process.sleep(sleep_time)
    IO.puts("#{json["message"]["tweet"]["text"]}")
    {:noreply, {sleep_time}}
  end
```

Listing 2: Printer Actor.

Lastly I have the supervisor in which I initialize both readers and printer.

```
def start_link(init_arg \\ :ok) do
    Supervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    children = [
      {Printer, 1000},
      %{id: :reader1,start: {Reader, :start_link, [:reader1, "localhost:4000/tweets/1"]}},
      %{id: :reader2,start: {Reader, :start_link, [:reader2, "localhost:4000/tweets/2"]}}
    ]
    Supervisor.init(children, strategy: :rest_for_one)
  end
```

Listing 3: Supervisor Actor.

Week 2

In the second week I added worker pool which substitutes the printer actor. I initialize 3 copies of the Printer actor which I supervise by the pool.

```
def start_link(init_arg \\ :ok) do
    Supervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    children = [
      %{id: :printer1,start: {Printer, :start_link, [:printer1, 50]}},
      %{id: :printer2,start: {Printer, :start_link, [:printer2, 50]}},
      %{id: :printer3,start: {Printer, :start_link, [:printer3, 50]}}
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
```

Listing 4: Worker Pool.

Additionally, I have the Load Balancer actor which mediates the tasks and sends them in a Round Robin fashion to printer actors.

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

  def init(number) do
    {:ok, {0, number}}
  end

  def handle_info(message, {current, number}) do
    id = :"printer#{current + 1}"
    if Process.whereis(id) != nil, do:
    send(id, message)
    {:noreply, {rem(current + 1, number), number}}
  end
```

Listing 5: Load Balancer Actor.

Week 3

In the third I added only the additional functionality to the printer actor which checks for any bad words that shouldn't be printed and I replace all of them with stars.

```
def init({id, sleep_time}) do
    bad_words = ["arse","arsehole", ...]
    {:ok, {id, sleep_time, bad_words}}
  end

  def handle_info(json, {id, sleep_time, bad_words}) do
    Process.sleep(sleep_time)
    tweet_text = json["message"]["tweet"]["text"]
    filtered_text = filter_bad_words(tweet_text, bad_words)
    IO.puts("#{id}: #{filtered_text}")
    {:noreply, {id, sleep_time, bad_words}}
  end

  defp filter_bad_words(text, bad_words) do
    Enum.reduce(bad_words, text, fn word, acc ->
      String.replace(acc, ~r/\b#{word}\b/, String.duplicate("*", String.length(word))
    )
    end)
  end
```

Listing 6: Printer Actor.

Week 4

In the fourth week I created 2 additional actors. The first one is engagement rationer in which I calculate the engagement ratio of each tweet.

```
  def start_link(id) do
    IO.puts("#{id} is starting")
    GenServer.start_link(__MODULE__, {id}, name: id)
  end

  @impl true
  def init({id}) do
    {:ok, {id}}
  end

  @impl true
  def handle_info({:msg, {msg_id, json}}, {id}) do
    favourites = json["message"]["tweet"]["retweeted_status"]["favorite_count"] || 0
    retweets = json["message"]["tweet"]["retweeted_status"]["retweet_count"] || 0
    followers = json["message"]["tweet"]["user"]["followers_count"]

    eng_ratio =
      if followers == 0,
        do: 0,
        else: (favourites + retweets) / followers

    IO.puts("Engagement ratio:  #{eng_ratio}")

    {:noreply, {id}}
  end
```

Listing 7: Engagement Rationer Actor.

Second actor is the sentiment scorer in which I calculate the sentiment score per tweet.

```
def init({id}) do
   score_map =
     HTTPoison.get!("localhost:4000/emotion_values").body
     |> String.split("\r\n", trim: true)
     |> Enum.reduce([], fn pair, acc ->
       map_values =
         pair
         |> String.split("\t", trim: true)

       acc ++ [%{word: Enum.at(map_values, 0), score: Enum.at(map_values, 1)}]
     end)

   {:ok, {id, score_map}}
 end

 def handle_info({:msg, {msg_id, json}}, {id, score_map}) do
   words = String.split(json["message"]["tweet"]["text"], " ", trim: true)
   sum =
     words
     |> Enum.reduce(0, fn word, acc ->
       case Enum.find(score_map, fn %{:word => value} ->
              value == word |> String.downcase()
            end) do
         nil ->
           acc

         word_found ->
           acc + String.to_integer(word_found[:score])
       end
     end)

   score = sum / length(words)

   IO.puts("Sentiment score: #{score}")

   {:noreply, {id, score_map}}
 end
```

Listing 8: Engagement Rationer Actor.

Another modification is in the Worker Pool. This time I modified it in order to have a generic initialization of workers, allowing to set an arbitrary number of worker pools. Also I created 3 worker pool which process the tweet stream in parallel.

```
def start_link(module, id, nr_of_workers, worker_args) do
   Supervisor.start_link(__MODULE__, {module, id, nr_of_workers, worker_args}, name:
   :"#{id}Pool")
 end

 def init({module, id, nr_of_workers, worker_args}) do
   children =
     1..nr_of_workers
     |> Enum.reduce([], fn number, acc ->
       acc ++
         [
           %{
             id: :"#{id}#{number}",
             start: {module, :start_link, [:"#{id}#{number}"] ++ worker_args}
           }
         ]
```

```
16        end)
17
18      Supervisor.init(children, strategy: :one_for_one)
19    end
```

Listing 9: Worker Pool Actor.

2 Conclusion

In conclusion, this project aimed to build a functional stream processing system that could handle complex applications. Throughout the weeks, the project required the presentation of two diagrams, the Message Flow Diagram, and the Supervision Tree Diagram. These diagrams were used to analyze the message exchange between actors of the system and monitor structures of the application.s

In addition, the project focused on stream processing with actors, a popular paradigm for building highly concurrent and scalable systems. By using actors, the system was able to process streams of data in parallel, with each actor responsible for a specific task or subset of data. This allowed for efficient use of resources and enabled the system to handle large volumes of data in real-time.

Overall, the project provided a valuable learning experience in stream processing with actors and helped the team to develop a deeper understanding of the principles and best practices involved in building highly concurrent and scalable systems.