Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Real - Time Programming

Project 2

Message Broker

Author:
Daniel Pogorevici
std. gr. FAF-202

Supervisor:
Alex Osadcenco

Chișinău 2023

1 Code implementation

Minimal Features

For this laboratory work I focused and did only the minimal features.

First of all, I've set up a TCP server that listens for incoming connections, assigns a role to each connected client using a child process, and associates the child process with the client socket for message handling and I/O operations.

```
def accept(port) do
    {:ok, socket} = :gen_tcp.listen(port, [:binary, packet: :line, active: false, reuseaddr: true])
    Logger.info "Accepting connections on: #{port}"
    loop_acceptor(socket)
  end

  defp loop_acceptor(socket) do
    {:ok, client} = :gen_tcp.accept(socket)
    {:ok, pid} = Task.Supervisor.start_child(MessageBroker.TaskSupervisor, fn -> MessageBroker.Client.assign_role(client) end)
    :ok = :gen_tcp.controlling_process(client, pid)
    loop_acceptor(socket)
  end
```

Listing 1: Server Module.

Afterwards,I've set up the application module for the message broker. It starts the application supervisor and defines the child processes to be supervised, including the subscription manager, role manager, task supervisor, and the message broker server. The server sets up a TCP socket to accept incoming connections on a specified port, and the supervisor ensures the child processes are properly supervised and restarted if necessary.

```
def start(_type, _args) do
    port = String.to_integer(System.get_env("PORT") || "4040")

    children = [
      MessageBroker.SubscriptionManager,
      MessageBroker.RoleManager,
      {Task.Supervisor, name: MessageBroker.TaskSupervisor},
      Supervisor.child_spec({Task, fn -> MessageBroker.Server.accept(port) end}, restart: :permanent)
    ]

    opts = [strategy: :one_for_one, name: MessageBroker.Supervisor]
    Supervisor.start_link(children, opts)
  end
```

Listing 2: Application Module.

Next, I defined the client-side functionality for the message broker. It handles serving client connections, starting the client process, performing I/O operations with the client socket, assigning roles to clients, and concluding the role assignment process.

```
 def write_line(socket, {:error, :unauthorized, action}) do
    send_client(socket, "Unauthorized: As a #{MessageBroker.RoleManager.get_readable_role(socket)} you don't have permission to #{action}.")
  end

  def write_line(socket, {:error, :sub_manager, reason}) do
    case reason do
      :already_subscribed -> send_client(socket, "Already subscribed!")
```

```
      :not_subscribed -> send_client(socket, "You are not subscribed!")
      :not_subscribed_publisher -> send_client(socket, "You are not subscribed")
      :publisher_not_found -> send_client(socket, "No publisher found!")
      :already_subscribed_to_publisher -> send_client(socket, "Already subscribed!")
      _ -> write_line(socket, {:error, reason})
    end
  end

  def write_line(_socket, {:error, :closed}) do
    exit(:shutdown)
  end

  def write_line(socket, {:error, error}) do
    send_client(socket, "Error #{inspect error}")
    exit(error)
  end

  def assign_role(socket) do
    if MessageBroker.RoleManager.has_role?(socket) == false do
      write_line(socket, {:ok, "Type 'PUBLISHER' or 'SUBSCRIBER'"})

      msg =
        with {:ok, data} <- read_line(socket),
        {:ok, role} <- MessageBroker.RoleManager.check_and_assign(socket, String.trim
    (data)),
        do: conclude(socket, role)

      write_line(socket, msg)
      case msg do
        {:error, :unknown, _} -> assign_role(socket)
        {:ok, _} -> MessageBroker.Client.serve(socket)
      end
    end
  end

  def conclude(socket, role) do
    case role do
      :consumer ->
        {:ok, "Successfully registered as subscriber."}
      :producer ->
        write_line(socket, {:ok, "Enter your name:"})
        with {:ok, name} <- read_line(socket),
        :ok <- MessageBroker.SubscriptionManager.register_publisher(socket, String.
    trim(name)),
        do: {:ok, "Successfully registered as publisher."}
    end
  end
```

Listing 3: Client Module.

Furthermore, I defined the command parsing and execution logic for the message broker. It parses commands received from clients, constructs corresponding command tuples, and executes the appropriate actions based on the client's role and the parsed command.

```
case parts do
      ["PUBLISH" | [topic, message]] -> {:ok, {:publish, String.trim(topic), String.
    trim(message)}}
      ["SUBSCRIBE" | [topic]] -> {:ok, {:subscribe_topic, String.trim(topic)}}
      ["SUBSCRIBE_TO" | [name]] -> {:ok, {:subscribe_publisher, String.trim(name)}}
      ["UNSUBSCRIBE" | [topic]] -> {:ok, {:unsubscribe, String.trim(topic)}}
      ["UNSUBSCRIBE_FROM" | [name]] -> {:ok, {:unsubscribe_publisher, String.trim(
    name)}}
```

```
      _ -> {:error, :unknown, "command #{inspect data}."}
    end
  end

  def run(client, {:subscribe_topic, topic}) do
    if MessageBroker.RoleManager.check_role(client, :consumer) do
      status = MessageBroker.SubscriptionManager.subscribe_to_topic(client, topic)

      case status do
        :ok -> {:ok, "Subscribed to topic: #{inspect topic}."}
        _ -> status
      end
    else
      {:error, :unauthorized, "subscribe"}
    end
  end

  def run(client, {:subscribe_publisher, name}) do
    if MessageBroker.RoleManager.check_role(client, :consumer) do
      status = MessageBroker.SubscriptionManager.subscribe_to_publisher(client, name)

      case status do
        :ok -> {:ok, "Subscribed to publisher: #{inspect name}."}
        _ -> status
      end
    else
      {:error, :unauthorized, "subscribe"}
    end
  end

  def run(client, {:unsubscribe, topic}) do
    if MessageBroker.RoleManager.check_role(client, :consumer) do
      status = MessageBroker.SubscriptionManager.unsubscribe(client, topic)

      case status do
        :ok -> {:ok, "Unsubscribed from topic: #{inspect topic}."}
        _ -> status
      end
    else
      {:error, :unauthorized, "unsubscribe"}
    end
  end

  def run(client, {:unsubscribe_publisher, name}) do
    if MessageBroker.RoleManager.check_role(client, :consumer) do
      status = MessageBroker.SubscriptionManager.unsubscribe_from_publisher(client,
   name)

      case status do
        :ok -> {:ok, "Unsubscribed from publisher: #{inspect name}."}
        _ -> status
      end
    else
      {:error, :unauthorized, "unsubscribe"}
    end
  end

  def run(client, {:publish, topic, message}) do
    if MessageBroker.RoleManager.check_role(client, :producer) do
      status = MessageBroker.SubscriptionManager.publish(client, topic, message)
```

```
66
67        case status do
68          :ok -> {:ok, "Published to topic: #{inspect topic} the message: #{inspect
      message}."}
69          _ -> status
70        end
71      else
72        {:error, :unauthorized, "publish"}
73      end
74    end
```

Listing 4: Command Module.

Additionally, I have the Role Manager module, which acts as a server process responsible for managing roles in the message broker system. It allows clients to be assigned roles, retrieve their assigned roles, check their roles against required roles, and perform other role-related operations.

```
1   def get_readable_role(client) do
2     role = get_role(client)
3     case role do
4       :producer -> "PUBLISHER"
5       :consumer -> "SUBSCRIBER"
6       _ -> "NO ROLE"
7     end
8   end
9
10  def check_role(client, required_role) do
11    role = get_role(client)
12    role == required_role
13  end
14
15  def has_role?(client) do
16    get_role(client) != :no_role
17  end
18
19  def check_and_assign(client, input) do
20    status = case input do
21      "PUBLISHER" -> assign_role(client, :producer)
22      "SUBSCRIBER" -> assign_role(client, :consumer)
23      _ -> {:error, :unknown, "role #{inspect input}."}
24    end
25    status
26  end
27
28  def handle_call({:assign_role, client, role}, _from, state) do
29    new_state = Map.put(state, client, role)
30    {:reply, {:ok, role}, new_state}
31  end
32
33  def handle_call({:get_role, client}, _from, state) do
34    role = Map.get(state, client, :no_role)
35    {:reply, role, state}
36  end
```

Listing 5: Role Manager Module.

Lastly, I implemented the Subscription Manager module, which acts as a server process responsible for managing subscriptions to topics and publishers, handling unsubscriptions, registering publishers, and publishing messages to subscribers.

```
1     ...
2 def handle_call({:register, client, name}, _from, state) do
```

```
    publishers = Map.get(state, :publishers, %{})

    if Map.has_key?(publishers, name) do
      {:reply, {:error, :sub_manager, :already_registered}, state}
    else
      new_publishers = Map.put(publishers, name, client)
      new_state = Map.put(state, :publishers, new_publishers)
      {:reply, :ok, new_state}
    end
  end

  def handle_call({:publish, client, topic, message}, _from, state) do
    topic_subscribers = Map.get(state.topics, topic, [])

    pub_name = Enum.find_value(state.publishers, fn{key, value} -> value == client && key end)
    publisher_subscribers = Map.get(state.pub_sub, pub_name, [])

    all_subscribers = Enum.uniq(topic_subscribers ++ publisher_subscribers)

    Enum.each(all_subscribers, fn sub_socket -> send_message(sub_socket, pub_name, topic, message) end)
    {:reply, :ok, state}
  end

  defp send_message(socket, name, topic, message) do
    :gen_tcp.send(socket, "#{name} posted on topic [#{topic}] the message: #{inspect message}\r\n")
  end
```

Listing 6: Subscription Manager Module.

2 Diagrams

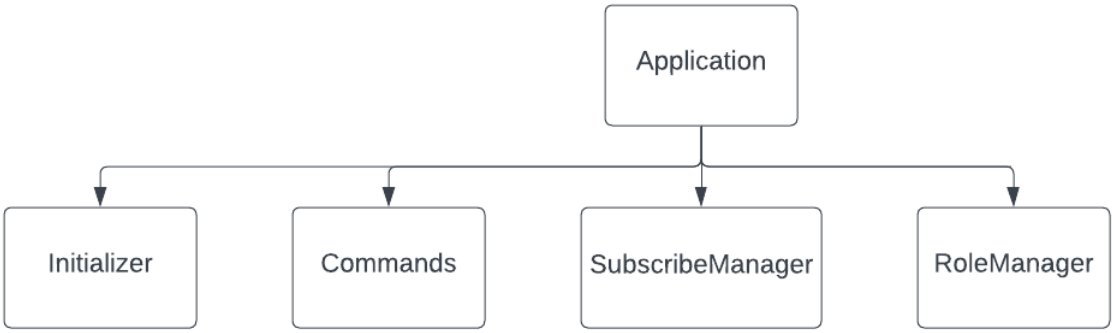

Figure 1: Supervision Tree.

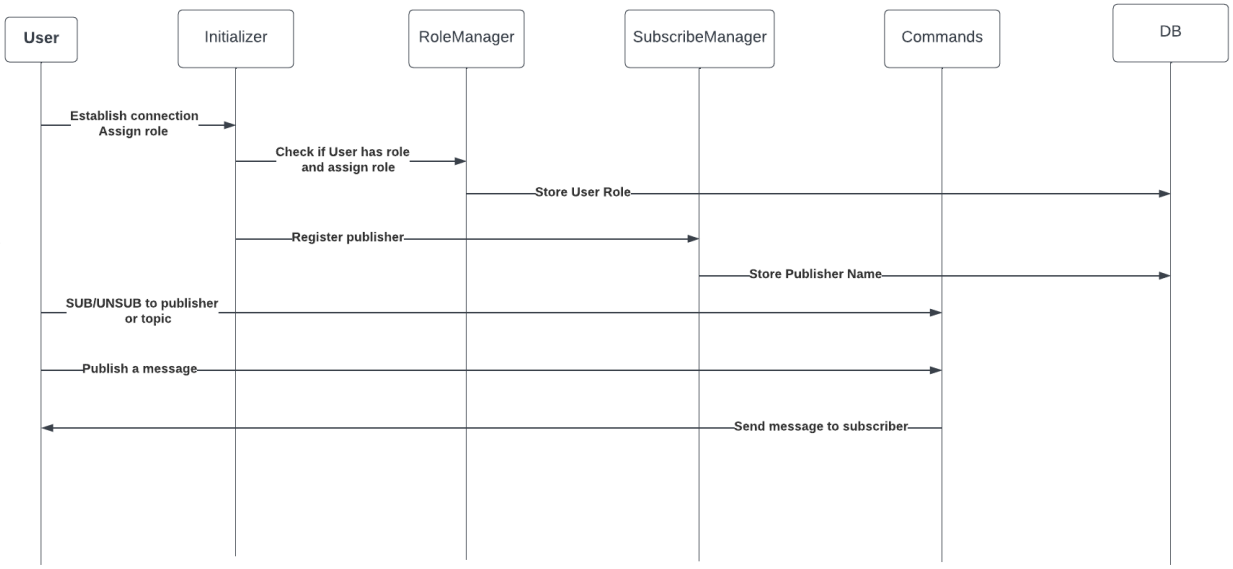

Figure 2: Message flow.

3 Conclusion

In conclusion, this laboratory work aimed to develop an actor-based message broker application that facilitates communication between producers and consumers. The project successfully implemented the minimal features required for a functional message broker.

The message broker application allows consumers to subscribe to publishers and receive messages published by them. It provides the ability for clients to connect via telnet or netcat, ensuring ease of access. The broker supports multiple topics, allowing consumers to subscribe to different topics and publishers to publish messages on various topics.

Overall, this laboratory work successfully accomplished the goal of creating a functional actor-based message broker application. It showcases the importance of reliable message communication and highlights the benefits of using an actor-based architecture for managing such communication tasks.