Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Real Time Programming

Laboratory work #2

Message Broker

Author:
Popa Eugeniu
std. gr. FAF-202

Supervisor:
Alexandru Osadcenco

Chișinău 2023

1 General Requirements

The goal for this project is to create an actor-based message broker application that would manage the communication between other applications named producers and consumers.

2 Minimal Features

- The message broker provides the ability to subscribe to publishers (if you are a consumer) and publish messages for subscribed consumers to receive (if you are a publisher);
- The message broker represents a dedicated TCP / UDP server;
- The message broker allows for clients to connect via telnet / netcat;
- The message broker provides the ability to subscribe to multiple topics (if you are a consumer) and publish messages on different topics (if you are a publisher);
- The project has an executable that can run the message broker with a single click / command.

```
defmodule Consumer do
  def serve(socket) do
    msg =
      with {:ok, data} <- read_line(socket),
      {:ok, command} <- Commands.parse(data),
      do: Commands.run(socket, command)
    write_line(socket, msg)
    serve(socket)
  end

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

  def init(_opts) do
    {:ok, %{}}
  end

  def read_line(socket) do
    :gen_tcp.recv(socket, 0)
  end

  defp send_message(socket, text) do
    :gen_tcp.send(socket, "#{text}\r\n")
  end

  def write_line(socket, {:ok, text}) do
    send_message(socket, text)
  end

  def write_line(socket, {:error, :unauthorized, action}) do
    send_message(socket, "As a consumer, you don't have permission to #{action}.")
  end

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

```
38        :already_subscribed_to_publisher -> send_message(socket, "Already subscribed to
      this publisher!")
39        :publisher_not_found -> send_message(socket, "No publisher found!")
40        _ -> write_line(socket, {:error, reason})
41      end
42    end
43
44    def write_line(_socket, {:error, :closed}) do
45      exit(:shutdown)
46    end
47
48    def write_line(socket, {:error, :unknown, reason}) do
49      send_message(socket, "Unknown #{reason}")
50    end
51
52    def write_line(socket, {:error, error}) do
53      send_message(socket, "Error #{inspect error}")
54      exit(error)
55    end
56
57    def assign_role(socket) do
58      write_line(socket, {:ok, "Creating the consumer..."})
59      result =
60        with consumer_role = "CONSUMER",
61        RoleManager.register_role(socket, consumer_role),
62        do: {:ok, "The consumer was created successfully. Execute a command to
      subscribe: "}
63      write_line(socket, result)
64      case result do
65        {:error, :unknown, _} -> assign_role(socket)
66        {:ok, _} -> Consumer.serve(socket)
67      end
68    end
69  end
```

This is an Elixir module named Consumer. It defines several functions related to serving a socket connection and handling commands.

serve(socket): This function is the main entry point for serving a socket connection. It continuously reads a line of data from the socket, parses the command, and executes it using the Commands module. The result of the command execution is then sent back to the socket. This function is recursively called to keep serving the socket.

start_link(opts): This function starts a GenServer process with the Consumer module. It takes opts as an argument, which represents the options for starting the process.

init(_opts): This function is a callback function required by the GenServer behavior. It is invoked when the GenServer process is started. In this case, it simply returns :ok, %, indicating that the process initialization is successful and the process state is an empty map.

read_line(socket): This function uses the :gen_tcp.recv/2 function to read a line of data from the socket. It expects the socket as an argument and returns the received data.

`send_message(socket, text)`: This function uses the `gen_tcp.send/2` function to send a text message to the socket. It concatenates the text argument with `\r\n` (carriage return and line feed) before sending.

write_line(socket, :ok, text): This function pattern matches on a tuple :ok, text and calls send_message/2 to send the text to the socket.

write_line(socket, :error, :unauthorized, action): This function pattern matches on a tuple :error, :unauthorized, action and sends a message to the socket indicating that the consumer doesn't have permission to perform the specified action.

write_line(socket, :error, :sub_manager, reason): This function pattern matches on a tuple :error, :sub_manager, reason. It further pattern matches on the reason to handle different cases. If reason is :already_subscribed, it sends a message indicating that the consumer is already subscribed to a topic. If reason is :already_subscribed_to_publisher, it sends a message indicating that the consumer is already subscribed to a publisher. If reason is :publisher_not_found, it sends a message indicating that no publisher was found. For any other reason, it recursively calls write_line/2 with :error, reason.

write_line(_socket, :error, :closed): This function pattern matches on a tuple :error, :closed and exits the process with :shutdown atom. It's typically used when the socket is closed.

write_line(socket, :error, :unknown, reason): This function pattern matches on a tuple :error, :unknown, reason and sends a message to the socket indicating that an unknown reason occurred.

write_line(socket, :error, error): This function pattern matches on a tuple :error, error and sends a message to the socket indicating an error occurred, along with the error details obtained from inspect/1 function. It then exits the process with the error atom.

assign_role(socket): This function is responsible for assigning the role of the consumer. It sends a message to the socket indicating the creation of the consumer and attempts to register the role using the RoleManager module. The result of the role registration is sent to the socket. If the result is :error, :unknown, _, indicating an unknown error occurred, the function is recursively called. If the result is :ok, _, indicating successful registration, Consumer.serve(socket) is called to start serving the socket.

Overall, this module provides a basic structure for serving socket connections, handling commands, and managing consumer roles. It relies on the Commands and RoleManager modules for command parsing, execution, and role registration.

```
defmodule Consumer.Server do
  require Logger

  def accept(port) do
    {:ok, socket} = :gen_tcp.listen(port, [:binary, packet: :line, active: false,
   reuseaddr: true])
    Logger.info "Consumer server - 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(TaskSupervisor, fn -> Consumer.
   assign_role(client) end)
    :ok = :gen_tcp.controlling_process(client, pid)
    loop_acceptor(socket)
  end
end
```

This is an Elixir module named Consumer.Server. It contains functions related to accepting socket connections and creating child processes to handle each connection.

accept(port): This function accepts incoming socket connections on the specified port. It uses the :gen_tcp.listen/2 function to create a listening socket on the given port. The options [:binary, packet: :line, active: false, reuseaddr: true] are passed to configure the socket. The :binary option indicates that the socket will handle binary data. The packet: :line option means that the socket will receive data in lines. The active: false option means the socket won't receive data actively but requires manual reading. The reuseaddr: true option allows reusing the address. Once the socket is successfully created, the function logs an info message using the Logger module and calls loop_acceptor/1 to start accepting connections.

loop_acceptor(socket): This function continuously accepts incoming connections on the socket obtained from :gen_tcp.listen/2. It uses the :gen_tcp.accept/1 function to accept a new connection and returns a tuple :ok, client containing the accepted client socket. It then starts a new child process using Task.Supervisor.start_child/2 from the TaskSupervisor supervisor. The child process executes the anonymous function fn -> Consumer.assign_role(client) end, which assigns the role to the client and starts serving the socket. The returned pid represents the process identifier of the child process.

:ok = :gen_tcp.controlling_process(client, pid): This line sets the controlling process of the client socket to the child process with pid. This means that the child process will handle communication on the socket.

loop_acceptor(socket): This line recursively calls loop_acceptor/1 to continue accepting new connections on the socket. This allows the server to accept multiple connections simultaneously.

Overall, this module sets up a server that listens for incoming socket connections on a specified port. Each incoming connection is accepted, and a child process is created to handle the connection. The child process assigns the role to the client and starts serving the socket. This approach allows for concurrent handling of multiple client connections. The server logs information about accepted connections using the Logger module.

```
defmodule Publisher do
  def serve(socket) do
    msg =
      with {:ok, data} <- read_line(socket),
      {:ok, command} <- Commands.parse(data),
      do: Commands.run(socket, command)
    write_line(socket, msg)
    serve(socket)
  end

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

  def init(_opts) do
    {:ok, %{}}
  end

  def read_line(socket) do
    :gen_tcp.recv(socket, 0)
  end

  defp send_message(socket, text) do
    :gen_tcp.send(socket, "#{text}\r\n")
  end

  def write_line(socket, {:ok, text}) do
    send_message(socket, text)
  end

  def write_line(socket, {:error, :unauthorized, action}) do
    send_message(socket, "As a publisher, you don't have permission to #{action}.")
  end

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

  def write_line(socket, {:error, :unknown, reason}) do
```

```
    send_message(socket, "Unknown #{reason}")
  end

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

  def assign_role(socket) do
    write_line(socket, {:ok, "Creating the publisher..."})
    result =
      with publisher_role = "PUBLISHER",
      RoleManager.register_role(socket, publisher_role),
      do: register_publisher(socket)
    write_line(socket, result)
    case result do
      {:error, :unknown, _} -> assign_role(socket)
      {:ok, _} -> Publisher.serve(socket)
    end
  end

  def register_publisher(socket) do
    write_line(socket, {:ok, "Enter the publisher's name: "})
    with {:ok, name} <- read_line(socket),
    :ok <- SubscriptionManager.register_publisher(socket, String.trim(name)),
    do: {:ok, "The publisher was created successfully. Execute the command to publish
    : "}
    end
end
```

This is an Elixir module named Publisher. It defines functions related to serving a socket connection and handling commands specific to a publisher role.

serve(socket): This function is the main entry point for serving a socket connection as a publisher. It is similar to the Consumer.serve/1 function we discussed earlier. It reads a line of data from the socket, parses the command using the Commands module, executes the command, and sends the result back to the socket. It then recursively calls itself to continue serving the socket.

start_link(opts): This function starts a GenServer process with the Publisher module. It takes opts as an argument, representing the options for starting the process.

init(_opts): This function is a callback function required by the GenServer behavior. It is invoked when the GenServer process is started. In this case, it simply returns :ok, %, indicating that the process initialization is successful and the process state is an empty map.

read_line(socket): This function uses the :gen_tcp.recv/2 function to read a line of data from the socket. It expects the socket as an argument and returns the received data.

`send_message(socket, text)`: This function uses the `:gen_tcp.send/2` function to send a text message to the socket. It concatenates the text argument with `"\r\n"` (carriage return and line feed) before sending.

write_line(socket, :ok, text): This function pattern matches on a tuple :ok, text and calls send_message/2 to send the text to the socket.

write_line(socket, :error, :unauthorized, action): This function pattern matches on a tuple :error, :unauthorized, action and sends a message to the socket indicating that the publisher doesn't have permission to perform the specified action.

write_line(_socket, :error, :closed): This function pattern matches on a tuple :error, :closed and exits the process with :shutdown atom. It's typically used when the socket is closed.

write_line(socket, :error, :unknown, reason): This function pattern matches on a tuple :error, :unknown, reason and sends a message to the socket indicating that an unknown reason occurred.

write_line(socket, :error, error): This function pattern matches on a tuple :error, error and sends a message to the socket indicating an error occurred, along with the error details obtained from inspect/1 function. It then exits the process with the error atom.

assign_role(socket): This function is responsible for assigning the role of the publisher. It sends a message to the socket indicating the creation of the publisher and attempts to register the role using the RoleManager module. If the registration is successful, it calls register_publisher/1 to proceed with publisher-specific registration. The result is sent to the socket, and based on the result, either assign_role/1 is recursively called or Publisher.serve/1 is invoked to start serving the socket as a publisher.

register_publisher(socket): This function is called after the publisher role is successfully registered. It sends a message to the socket asking for the publisher's name, reads the input from the socket, and attempts to register the publisher using the SubscriptionManager module. If the registration is successful, it returns a tuple :ok, "The publisher was created successfully. Execute the command to publish: ".

Overall, this module provides functionality for serving a socket connection as a publisher. It handles command execution, role assignment, and registration of the publisher. It interacts with the Commands, RoleManager, and SubscriptionManager modules for command parsing, execution, and role/publisher registration, respectively.

```
defmodule Publisher.Server do
  require Logger

  def accept(port) do
    {:ok, socket} = :gen_tcp.listen(port, [:binary, packet: :line, active: false, reuseaddr: true])
    Logger.info "Publisher server - 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(TaskSupervisor, fn -> Publisher.assign_role(client) end)
    :ok = :gen_tcp.controlling_process(client, pid)
    loop_acceptor(socket)
  end
end
```

This is an Elixir module named Publisher.Server. It contains functions related to accepting socket connections and creating child processes to handle each connection, similar to the Consumer.Server module we discussed earlier.

accept(port): This function accepts incoming socket connections on the specified port. It is similar to the Consumer.Server.accept/1 function. It uses the :gen_tcp.listen/2 function to create a listening socket on the given port with the specified options [:binary, packet: :line, active: false, reuseaddr: true]. Once the socket is successfully created, the function logs an info message using the Logger module and calls loop_acceptor/1 to start accepting connections.

loop_acceptor(socket): This function is similar to the Consumer.Server.loop_acceptor/1 function. It continuously accepts incoming connections on the socket obtained from :gen_tcp.listen/2. It uses the :gen_tcp.accept/1 function to accept a new connection and returns a tuple :ok, client containing the accepted client socket. It then starts a new child process using Task.Supervisor.start_child/2 from the TaskSupervisor supervisor. The child process executes the anonymous function fn -> Publisher.assign_role(client) end, which assigns the role to the client and starts serving the socket. The returned pid represents the process identifier of the child process.

:ok = :gen_tcp.controlling_process(client, pid): This line sets the controlling process of the

client socket to the child process with pid, similar to the Consumer.Server module. This means that the child process will handle communication on the socket.

loop_acceptor(socket): This line recursively calls loop_acceptor/1 to continue accepting new connections on the socket. This allows the server to accept multiple connections simultaneously.

Overall, this module sets up a server that listens for incoming socket connections on a specified port. Each incoming connection is accepted, and a child process is created to handle the connection. The child process assigns the role to the client and starts serving the socket. This approach allows for concurrent handling of multiple client connections. The server logs information about accepted connections using the Logger module. The code structure and behavior are similar to the Consumer.Server module, but specific to the publisher role.

```
defmodule RoleManager do
  use GenServer

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

  def init(_opts) do
    {:ok, %{}}
  end

  def assign_role(client, role) do
    GenServer.call(:role_manager, {:assign_role, client, role})
  end

  def get_role(client) do
    GenServer.call(:role_manager, {:get_role, client})
  end

  def register_role(client, input) do
    status = case input do
      "PUBLISHER" -> assign_role(client, :producer)
      "CONSUMER" -> assign_role(client, :consumer)
    end
    status
  end

  def check_role(client, required_role) do
    role = get_role(client)
    role == required_role
  end

  def handle_call({:assign_role, client, role}, _from, state) do
    new_state = Map.put(state, client, role)
    {:reply, {:ok, role}, new_state}
  end

  def handle_call({:get_role, client}, _from, state) do
    role = Map.get(state, client, :no_role)
    {:reply, role, state}
  end
end
```

This is an Elixir module named RoleManager. It is a GenServer implementation responsible for managing client roles and providing role-related functionalities.

use GenServer: This line indicates that the RoleManager module will use the GenServer behavior, allowing it to handle calls and maintain a state.

start_link(opts): This function starts a GenServer process with the RoleManager module. It takes opts as an argument, representing the options for starting the process. It uses GenServer.start_link/3 to start the process, providing the module name, opts, and setting the process name as :role_manager.

init(_opts): This function is a callback function required by the GenServer behavior. It is invoked when the GenServer process is started. In this case, it simply returns :ok, %, indicating that the process initialization is successful and the process state is an empty map.

assign_role(client, role): This function is used to assign a role to a specific client. It sends a synchronous call to the RoleManager GenServer process with the tuple :assign_role, client, role. This call is handled by the handle_call/3 function.

get_role(client): This function retrieves the role assigned to a specific client. It sends a synchronous call to the RoleManager GenServer process with the tuple :get_role, client. This call is also handled by the handle_call/3 function.

register_role(client, input): This function is responsible for registering a role for a client. It takes the client and input as arguments. Depending on the input value (which represents the role), it calls assign_role/2 with the appropriate role atom (:producer for "PUBLISHER" input or :consumer for "CONSUMER" input). The result of the assignment is returned.

check_role(client, required_role): This function checks whether a specific client has the required_role. It retrieves the role of the client using get_role/1 and compares it with the required_role. If they match, it returns true; otherwise, it returns false.

handle_call(:assign_role, client, role, _from, state): This function is the handler for the :assign_role, client, role call. It is invoked when assign_role/2 is called. It updates the state by associating the client with the assigned role in the state map. It returns :reply, :ok, role, new_state to respond to the caller, indicating a successful role assignment and providing the assigned role. The new_state is the updated state with the client and role added.

handle_call(:get_role, client, _from, state): This function is the handler for the :get_role, client call. It is invoked when get_role/1 is called. It retrieves the role associated with the client from the state map using Map.get/3. The role is then returned as the reply along with the current state. If no role is found for the client, :no_role is returned.

Overall, the RoleManager module provides functions to assign roles to clients, retrieve assigned roles, and check whether clients have specific roles. It uses a GenServer process to manage the state and handle the role-related operations.

```
defmodule Commands do
  def parse(line) do
    data = String.trim(line)
    parts = String.split(data, "/")
    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)}}
      _ -> {:error, :unknown, "command #{inspect data}."}
    end
  end

  def run(client, {:publish, topic, message}) do
    if RoleManager.check_role(client, :producer) do
      status = SubscriptionManager.publish(client, topic, message)
      case status do
        :ok -> {:ok, "Published the message: #{inspect message} to topic: #{inspect
    topic}."}
        _ -> status
      end
```

```
    else
      {:error, :unauthorized, "publish"}
    end
  end

  def run(client, {:subscribe_topic, topic}) do
    if RoleManager.check_role(client, :consumer) do
      status = 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 RoleManager.check_role(client, :consumer) do
      status = SubscriptionManager.subscribe_to_publisher(client, name)
      case status do
        :ok -> {:ok, "Subscribed to publisher: #{inspect name}."}
        _ -> status
      end
    else
      {:error, :unauthorized, "subscribe"}
    end
  end
end
```

This is an Elixir module named Commands. It defines functions for parsing and executing commands received from clients.

parse(line): This function takes a line of text as input and parses it into a command. It trims the line to remove leading and trailing whitespace, splits it into parts using "/" as the delimiter, and then pattern matches the parts to determine the command and its arguments. If the command matches one of the defined patterns, it returns :ok, parsed_command tuple, where parsed_command contains the command name and its arguments. If the command doesn't match any of the patterns, it returns :error, :unknown, "command <command_text>." tuple, indicating an unknown command.

run(client, :publish, topic, message): This function is invoked when the parsed command is :publish, topic, message. It checks if the client has the :producer role using RoleManager.check_role/2. If the client has the required role, it calls SubscriptionManager.publish/3 to publish the message to the specified topic. It then handles the status returned by SubscriptionManager.publish/3. If the status is :ok, it returns :ok, "Published the message: <message> to topic: <topic>." tuple. Otherwise, it returns the status itself.

run(client, :subscribe_topic, topic): This function is invoked when the parsed command is :subscribe_topic, topic. It checks if the client has the :consumer role using RoleManager.check_role/2. If the client has the required role, it calls SubscriptionManager.subscribe_to_topic/2 to subscribe the client to the specified topic. It then handles the status returned by SubscriptionManager.subscribe_to_topic/2. If the status is :ok, it returns :ok, "Subscribed to topic: <topic>." tuple. Otherwise, it returns the status itself.

run(client, :subscribe_publisher, name): This function is invoked when the parsed command is :subscribe_publisher, name. It checks if the client has the :consumer role using RoleManager.check_role/2. If the client has the required role, it calls SubscriptionManager.subscribe_to_publisher/2 to subscribe the client to the specified publisher.

It then handles the status returned by SubscriptionManager.subscribe_to_publisher/2. If the status is :ok, it returns :ok, "Subscribed to publisher: <name>." tuple. Otherwise, it returns the status itself.

The Commands module provides a way to parse and execute different commands received from clients. It checks the role of the client before executing certain commands, ensuring that clients have the appropriate permissions.

```
defmodule SubscriptionManager do
  use GenServer

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

  def init(_opts) do
    {:ok, %{topics: %{}, publishers: %{}, pub_sub: %{}}}
  end

  def register_publisher(client, name) do
    GenServer.call(:subscription_manager, {:register, client, name})
  end

  def publish(client, topic, message) do
    GenServer.call(:subscription_manager, {:publish, client, topic, message})
  end

  def subscribe_to_topic(client, topic) do
    GenServer.call(:subscription_manager, {:subscribe, client, topic})
  end

  def subscribe_to_publisher(client, publisher) do
    GenServer.call(:subscription_manager, {:subscribe_to_publisher, client, publisher
    })
  end

  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

  def handle_call({:subscribe, client, topic}, _from, state) do
    topics = Map.get(state, :topics, %{})
```

```
    subscribers_to_topic = Map.get(topics, topic, [])
    if Enum.member?(subscribers_to_topic, client) do
      {:reply, {:error, :sub_manager, :already_subscribed}, state}
    else
      new_topic = Map.put(topics, topic, [client | subscribers_to_topic])
      new_state = Map.put(state, :topics, new_topic)
      {:reply, :ok, new_state}
    end
  end

  def handle_call({:subscribe_to_publisher, subscriber, publisher}, _from, state) do
    if Map.get(state.publishers, publisher, nil) == nil do
      {:reply, {:error, :sub_manager, :publisher_not_found}, state}
    else
      publisher_subscribers = Map.get(state.pub_sub, publisher, [])
      if Enum.member?(publisher_subscribers, subscriber) do
        {:reply, {:error, :sub_manager, :already_subscribed_to_publisher}, state}
      else
        new_state = %{state | pub_sub: Map.put(state.pub_sub, publisher, [subscriber
    | publisher_subscribers])}
        {:reply, :ok, new_state}
      end
    end
  end

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

This is an Elixir module named SubscriptionManager. It is a GenServer module that manages publishers, subscribers, and topic subscriptions.

start_link(opts): This function starts the SubscriptionManager GenServer by calling GenServer.start_link/3 with the module name, opts, and the name :subscription_manager.

init(_opts): This function is the initialization callback for the GenServer. It is invoked when the GenServer is started. It initializes the state of the GenServer with an empty map containing three key-value pairs: topics, publishers, and pub_sub. Each of these keys is associated with an empty map or list.

register_publisher(client, name): This function is used to register a publisher with the SubscriptionManager. It sends a call to the :subscription_manager process with the tuple :register, client, name. In the handle_call/3 callback, it checks if the publisher name already exists in the publishers map. If it does, it replies with :error, :sub_manager, :already_registered. Otherwise, it adds the publisher to the publishers map and replies with :ok.

publish(client, topic, message): This function is used to publish a message to subscribers of a topic. It sends a call to the :subscription_manager process with the tuple :publish, client, topic, message. In the handle_call/3 callback, it retrieves the subscribers for the specified topic and the subscribers for the publisher associated with the client. It combines these subscribers into a list of all_subscribers. It then iterates over each subscriber and sends a message to their socket using the send_message/4 function. Finally, it replies with :ok.

subscribe_to_topic(client, topic): This function is used to subscribe a client to a topic. It sends a call to the :subscription_manager process with the tuple :subscribe, client, topic. In the handle_call/3 callback, it retrieves the subscribers for the specified topic from the topics map. If the client is already a subscriber, it replies with :error, :sub_manager, :already_subscribed. Otherwise, it adds the client to the list of subscribers for the topic and updates the topics map in

the state. Finally, it replies with :ok.

subscribe_to_publisher(client, publisher): This function is used to subscribe a client to a specific publisher. It sends a call to the :subscription_manager process with the tuple :subscribe_to_publisher, client, publisher. In the handle_call/3 callback, it checks if the publisher exists in the publishers map. If not, it replies with :error, :sub_manager, :publisher_not_found. If the client is already a subscriber, it replies with :error, :sub_manager, :already_subscribed_to_publisher. Otherwise, it adds the client to the list of subscribers for the publisher in the pub_sub map and updates the state. Finally, it replies with :ok.

send_message(socket, name, topic, message): This is a private function used to send a message to a socket. It sends a formatted message using :gen_tcp.send/2, containing the publisher's name, the message, and the topic.

The SubscriptionManager module provides functions for registering publishers, publishing messages to subscribers, and managing topic and publisher subscriptions. It maintains state using a GenServer process and handles calls from other processes to perform the necessary operations.

```
defmodule MessageBroker do
  use Application

  def start(_type, _args) do
    publisher_port = String.to_integer(System.get_env("PUBLISHER_PORT") || "4080")
    consumer_port = String.to_integer(System.get_env("CONSUMER_PORT") || "4081")
    children = [
      {Task.Supervisor, name: TaskSupervisor},
      Supervisor.child_spec({Task, fn -> Publisher.Server.accept(publisher_port) end}
    , restart: :permanent, id: :publisher_server),
      Supervisor.child_spec({Task, fn -> Consumer.Server.accept(consumer_port) end},
   restart: :permanent, id: :consumer_server),
      RoleManager,
      SubscriptionManager,
    ]
    opts = [strategy: :one_for_one, name: MessageBroker.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

This is an Elixir module named MessageBroker. It is an Application module that defines the startup behavior of the message broker system.

start(_type, _args): This function is the entry point of the application. It is invoked when the application starts. It takes two arguments _type and _args, although they are not used in this implementation. It configures and starts the supervision tree for the message broker system.

publisher_port: It retrieves the value of the environment variable "PUBLISHER_PORT" and converts it to an integer using String.to_integer/1. If the environment variable is not set, it defaults to port 4080. consumer_port: It retrieves the value of the environment variable "CONSUMER_PORT" and converts it to an integer using String.to_integer/1. If the environment variable is not set, it defaults to port 4081. children: It defines a list of child processes that will be supervised by the top-level supervisor. The child processes include: TaskSupervisor: A Task Supervisor that manages dynamic tasks. Publisher.Server.accept(publisher_port): A child specification for the Publisher.Server.accept/1 function, which starts a TCP server for publishers to connect on the specified port. Consumer.Server.accept(consumer_port): A child specification for the Consumer.Server.accept/1 function, which starts a TCP server for consumers to connect on the specified port. RoleManager: A child specification for the RoleManager GenServer, responsible for managing roles (producer/consumer) of connected clients. SubscriptionManager: A child specification for the SubscriptionManager GenServer, responsible for managing topic subscriptions and message routing. opts: It specifies the supervision strategy and the name of the supervisor.

In this case, the strategy is :one_for_one, meaning that if a child process fails, only that specific child process will be restarted. The name of the supervisor is set to MessageBroker.Supervisor. Supervisor.start_link(children, opts): It starts the supervisor with the specified children and options. The MessageBroker module sets up the supervision tree for the message broker system, which includes supervisors for handling tasks, TCP servers for publishers and consumers, and the GenServers for managing roles and subscriptions.

3 Diagrams

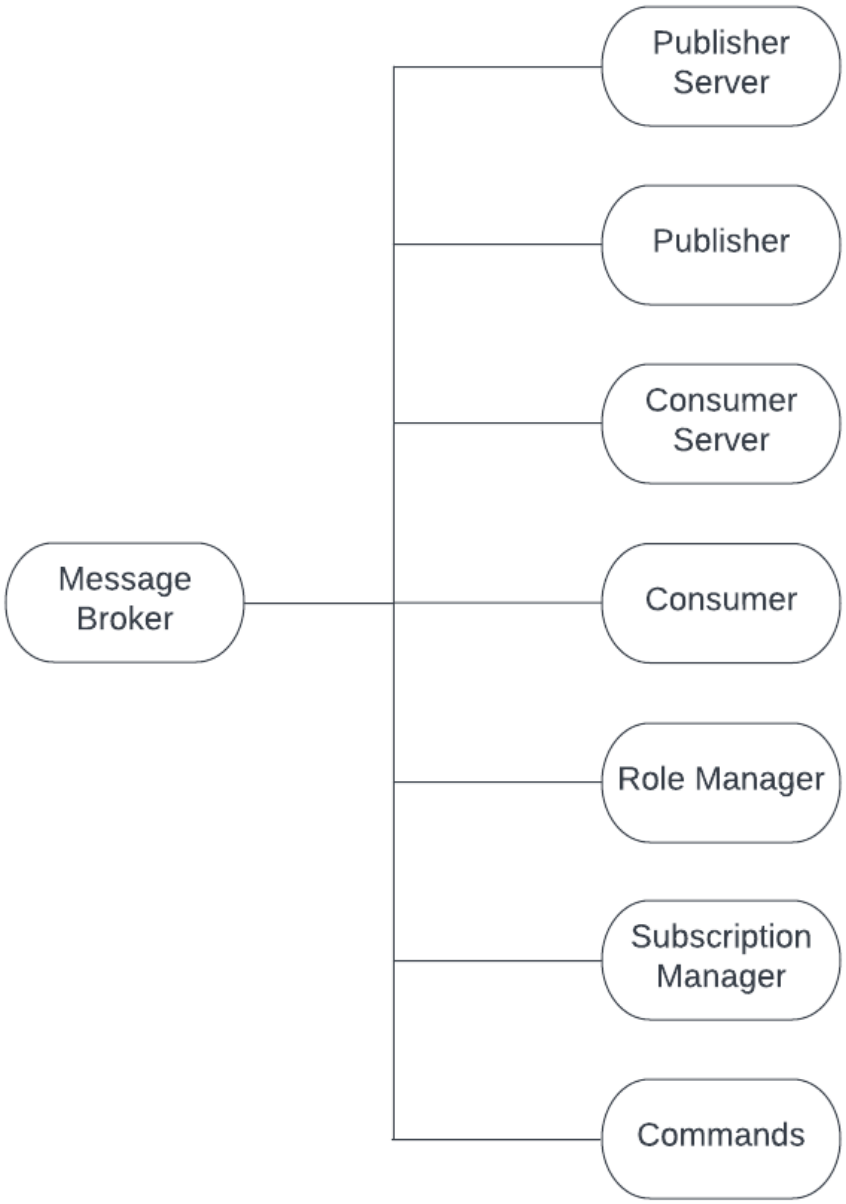

Figure 1: Supervision tree diagram

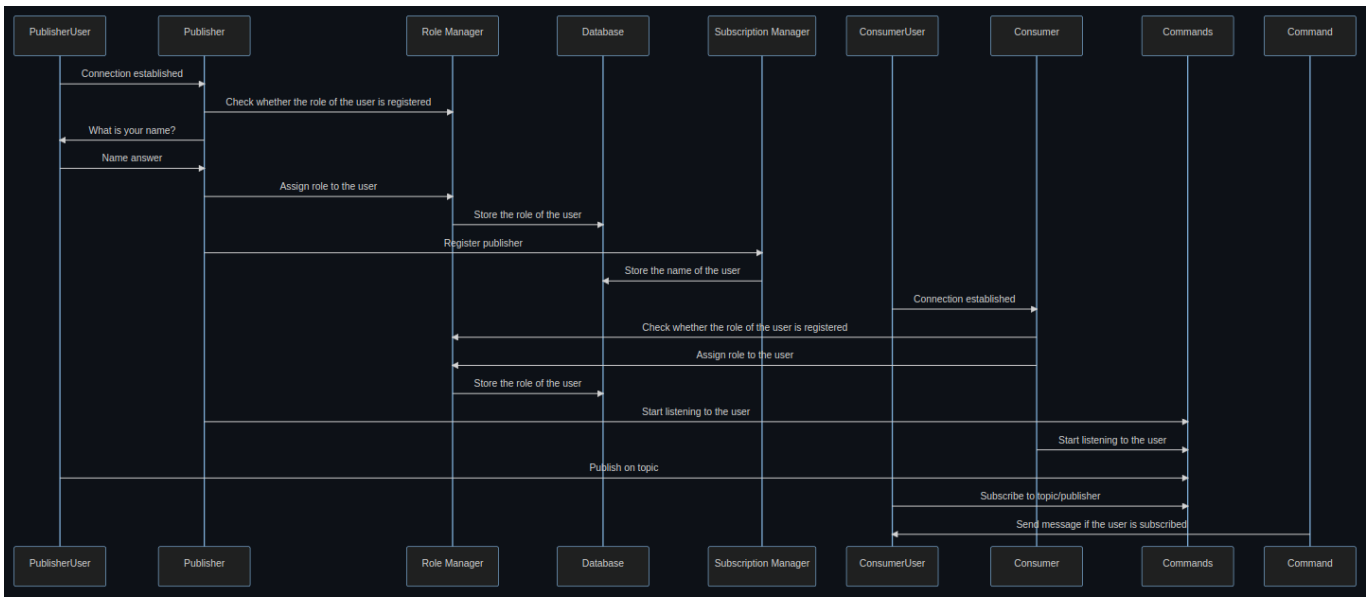

Figure 2: Message flow diagram

4 Conclusion

In summary, I can confidently say that I have successfully developed an actor-based message broker application that manages the communication between producers and consumers. The message broker provides essential functionalities such as subscribing to publishers and publishing messages for subscribed consumers to receive.

The message broker is implemented as a dedicated TCP/UDP server, allowing clients to connect using telnet or netcat. This enables seamless communication between different applications.

One of the key features of the message broker is its support for multiple topics. Consumers can subscribe to multiple topics, while publishers can publish messages on different topics. This flexibility allows for efficient message routing and ensures that consumers receive only the relevant messages.

To make the message broker easily accessible, I have created an executable file that can be run with a single click or command. This simplifies the process of starting and managing the message broker.

Overall, this project has provided me with valuable hands-on experience in developing an actor-based message broker application. I have learned the importance of efficient message handling, topic-based communication, and creating user-friendly interfaces for seamless interaction.