Grygoruk Piotr 260299

Filip Strózik 260377

Rozproszone Systemy Informatyczne – Laboratorium nr 13

Prowadzący: Dr inż. Zbigniew Fryźlewicz

1. **Konfiguracja sieciowa**
   W celu wykonania ćwiczenia w konfiguracji dwumaszynowej skorzystaliśmy z sieci komputerowej eduroam. Koniecznym krokiem było wyłącznie firewalla, zarówno po stronie sendera jak i receivera.
   Broker był włączony na jednym laptopie i został udostępniony całej grupie zajęciowej, jego adres podczas zajęć to 10.182.17.252. Użytkownik dodany przez brokera to l:admin, h:admin. Dodanie użytkownika było kluczowe by zrealizować połączenie „z zewnątrz”.

Po wykonaniu powyższych kroków możliwe jest uruchomienie sendera oraz receivera w konfiguracji dwumaszynowej:

```
        var factory = new ConnectionFactory { HostName = "10.182.17.252", Port
= 5672, UserName = "admin", Password = "admin" };

        using var connection = factory.CreateConnection();
        using var channel = connection.CreateModel();

        channel.QueueDeclare(queue: "piotr_filip",

                    durable: true,
                    exclusive: false,
                    autoDelete: false,
                    arguments: null);
```

2. Budowa Programu Recieve (Consumer) w ramach projektu
   Budowe programu, można zacząć od stworzenia projektu programu w języku Java oraz dodania pakietu rabbitMQ oraz Gson do parsowania JSON’a.

Recv.java

```
import com.google.gson.GsonBuilder;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
import java.nio.charset.StandardCharsets;
import com.google.gson.Gson;
import java.time.LocalTime;
public class Recv {

    private final static String QUEUE_NAME = "filip_piotr";
    private final static int NO_SENDERS = 2;
    private static int endMarkerCount = 0;

    public static void main(String[] argv) throws Exception {

        MyData.info();

        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("10.182.17.252");
        factory.setPort(5672);
        factory.setUsername("admin");
        factory.setPassword("admin");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
```

Tutaj łączymy się z węzłem RabbitMQ na lokalnej maszynie. Gdybyśmy chcieli połączyć się z węzłem na innej maszynie, po prostu określilibyśmy tutaj jego nazwę hosta lub adres IP.

```
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
```

Deklarujemy kolejkę w celu zarządzania wiadomościami.

```
        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), StandardCharsets.UTF_8);
            System.out.println(" [x] Received '" + message + "'");
```

Definiujemy callback, który wywoływany jest za każdym razem, gdy otrzymamy wiadomość. Najpierw sprawdzamy czy wiadomością jest marker kończący przesyłanie wiadomości od senderów, jeżeli osiągnęliśmy ustaloną wyżej liczbę senderów to kończmy połączenie i następuje koniec działania programu, w przeciwnym wypadku następuje deserializacja wiadomości z formatu JSON na obiekt wiadomości oraz wypisanie pól wiadomości na ekran.

```
            if (message.equals("EndMarker")) {
                endMarkerCount++;
                if (endMarkerCount >= NO_SENDERS) {
                    System.out.println("Received " + NO_SENDERS + " end markers. Exiting...");
                    channel.basicCancel(consumerTag);
                    try {
                        channel.close();
                        connection.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
            else {
                Gson gson = new GsonBuilder()
                        .registerTypeAdapter(LocalTime.class, new LocalTimeAdapter())
                        .create();
                Message messageDeserialized = gson.fromJson(message, Message.class);

                System.out.println("Name: " + messageDeserialized.getName());
                System.out.println("Time: " + messageDeserialized.getTime());
                System.out.println("Counter: " + messageDeserialized.getCounter());
            }
        };
        channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> { });
    }
}
```

Message.java

```
import java.time.LocalTime;

public class Message {
    public LocalTime time;
    public String name;
    public int counter;

    public LocalTime getTime() {
        return this.time;
    }

    public String getName() {
```

```
        return this.name;
    }

    public int getCounter() {
        return this.counter;
    }
}
```

Klasa wiadomości służy do łatwiejszej deserializacji wiadomości w przypadku użycia biblioteki gson. W niej jest określona struktura i typy dostarczanej wiadomości.

LocalTimeAdapter.java

```
import com.google.gson.TypeAdapter;
import com.google.gson.stream.JsonReader;
import com.google.gson.stream.JsonToken;
import com.google.gson.stream.JsonWriter;

import java.io.IOException;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public class LocalTimeAdapter extends TypeAdapter<LocalTime> {
    private final DateTimeFormatter formatter = DateTimeFormatter.ISO_TIME;

    @Override
    public void write(JsonWriter out, LocalTime value) throws IOException {
        if (value == null) {
            out.nullValue();
        } else {
            String formattedValue = formatter.format(value);
            out.value(formattedValue);
        }
    }

    @Override
    public LocalTime read(JsonReader in) throws IOException {
        if (in.peek() == JsonToken.NULL) {
            in.nextNull();
            return null;
        }
        String dateString = in.nextString();
        return LocalTime.parse(dateString, formatter);
    }
}
```

Klasa LocalTimeAdapter określa jaki format dat będzie odbierany. Uściślenie format w tym przypadku ISO_TIME jest konieczne w celu poprawnej deserializacji i serializacji czasu.

3. Budowa Programu Send (Publisher) w ramach projektu

Send.cs

```
using System;
```

```
using System.Text;
using RabbitMQ.Client;
using System.Threading;
using Newtonsoft.Json;

public class Program
{
    private const int DurationSeconds = 10;
    private const string EndMarkerMessage = "EndMarker";


    public static void Main()
    {
        MyData.Info();



        var factory = new ConnectionFactory { HostName = "10.182.17.252", Port
= 5672, UserName = "admin", Password = "admin" };
        using var connection = factory.CreateConnection();
        using var channel = connection.CreateModel();

        channel.QueueDeclare(queue: "piotr_filip",
                     durable: true,
                     exclusive: false,
                     autoDelete: false,
                     arguments: null);

        DateTime endTime = DateTime.Now.AddSeconds(DurationSeconds);
        int counter = 0;
        while (DateTime.Now < endTime)
        {
            // serialize message to JSON format
            string message = JsonConvert.SerializeObject(new { name = "Filip",
time = DateTime.Now.TimeOfDay, counter = counter++ });

            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish(exchange: "",
                                 routingKey: "piotr_filip",
                                 basicProperties: null,
                                 body: body);

            Console.WriteLine($" [x] Sent {message}");

            // Random sleep between 1 and 2 seconds
            Random rnd = new Random();
            int sleep = rnd.Next(1000, 2000);
            Thread.Sleep(sleep);
```

```
        }

        var endMarkerBody = Encoding.UTF8.GetBytes(EndMarkerMessage);
        channel.BasicPublish(exchange: "",
                             routingKey: "hello",
                             basicProperties: null,
                             body: endMarkerBody);

        Console.WriteLine($" [x] Sent end marker '{EndMarkerMessage}'");

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }
}
```

Ten kod jest prostą aplikacją kliencką, która wykorzystuje bibliotekę RabbitMQ.Client do komunikacji za pomocą protokołu RabbitMQ. Główną funkcjonalnością kodu jest wysyłanie wiadomości do kolejki brokera.
Wewnątrz klasy **Program** znajdują się dwie stałe: **DurationSeconds**, która określa czas trwania działania aplikacji w sekundach (ustawiona na 10), oraz **EndMarkerMessage**, która reprezentuje wiadomość końcową.

Metoda **Main()** jest punktem wejścia do aplikacji. W pierwszej linii wywoływana jest metoda **MyData.Info**. Następnie tworzony jest obiekt **ConnectionFactory** z adresem hosta RabbitMQ.

Dalej, za pomocą **using** tworzone są obiekty **connection** i **channel**, reprezentujące połączenie i kanał komunikacyjny z RabbitMQ. Dzięki użyciu **using**, zasoby zostaną automatycznie zwolnione po zakończeniu bloku.

Główna część kodu to pętla **while**, która działa przez określony czas (**DurationSeconds**). Wewnątrz pętli:

1. Tworzona jest wiadomość w formacie JSON, zawierająca nazwę ("Filip"), aktualny czas i licznik. Używamy do tego bibiolteki Newtonsoft.Json, która pozwala na proste użycie serializacji i deserializacji.
2. Wiadomość jest kodowana w formacie UTF-8 i przekształcana na tablicę bajtów.
3. Wywoływana jest metoda **BasicPublish** na obiekcie **channel**, która wysyła wiadomość do kolejki RabbitMQ o podanej nazwie wymiany (**""** oznacza domyślną wymianę), kluczu routingu ("hello") oraz bez żadnych dodatkowych właściwości.
4. Informacja o wysłanej wiadomości jest wyświetlana na konsoli.

Po każdym wysłaniu wiadomości następuje losowe opóźnienie między 1s a 2s, realizowane przez wywołanie **Thread.Sleep(sleep)**, gdzie **sleep** to losowo wygenerowany czas oczekiwania.

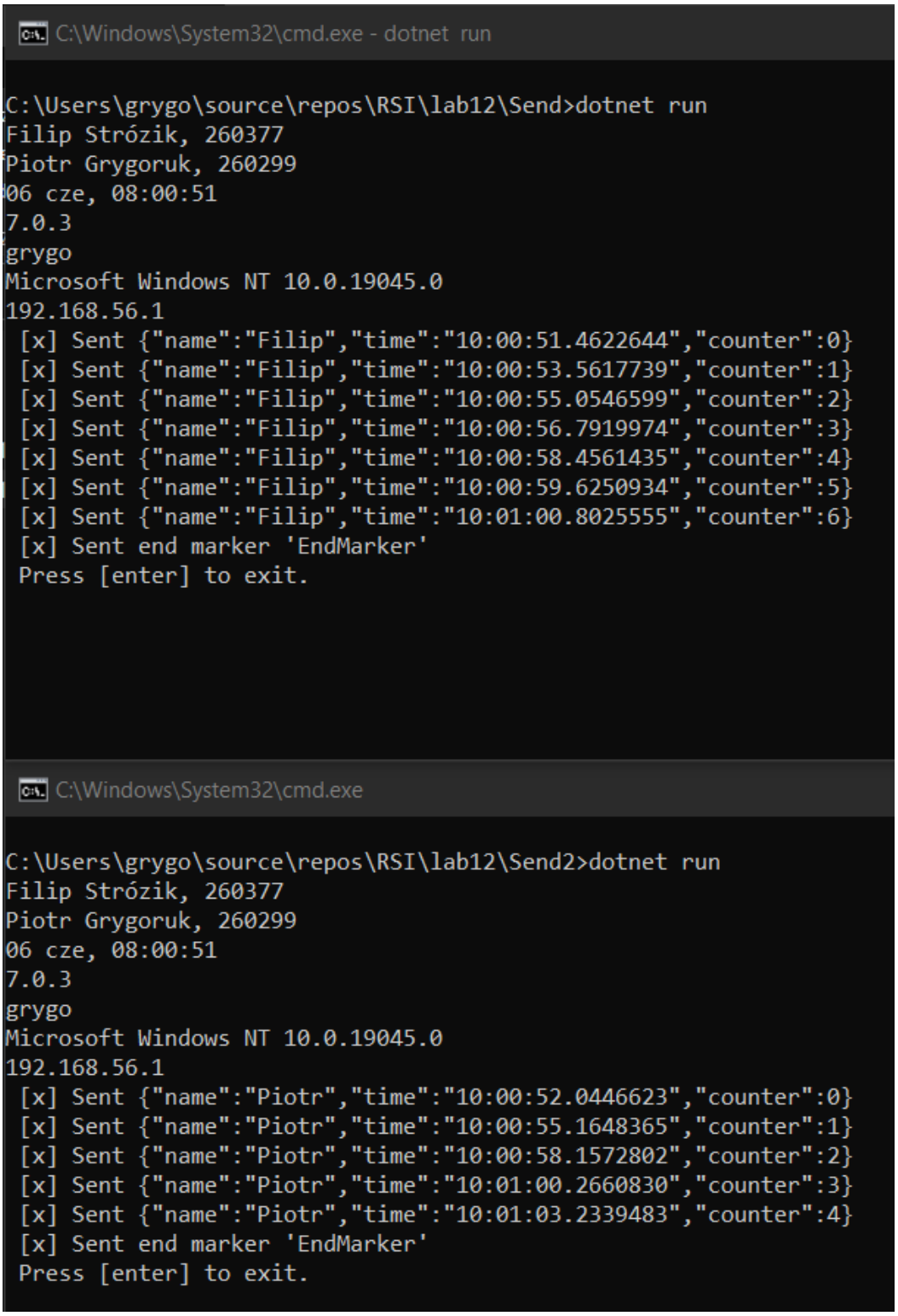
Po zakończeniu pętli wysyłającej, wysyłana jest wiadomość końcowa, która ma treść "EndMarker". Zasób **channel** używany jest ponownie do wysłania tej wiadomości.

Send2.cs

Działanie analogiczne do Send.cs (ze zmienionym opóźnieniem wysyłania wiadomości.)

4. Uruchomienie

Send i Send2:


C:\Windows\System32\cmd.exe - dotnet run
C:\Users\grygo\source\repos\RSI\lab12\Send>dotnet run
Filip Strózik, 260377
Piotr Grygoruk, 260299
06 cze, 08:00:51
7.0.3
grygo
Microsoft Windows NT 10.0.19045.0
192.168.56.1
[x] Sent {"name":"Filip","time":"10:00:51.4622644","counter":0}
[x] Sent {"name":"Filip","time":"10:00:53.5617739","counter":1}
[x] Sent {"name":"Filip","time":"10:00:55.0546599","counter":2}
[x] Sent {"name":"Filip","time":"10:00:56.7919974","counter":3}
[x] Sent {"name":"Filip","time":"10:00:58.4561435","counter":4}
[x] Sent {"name":"Filip","time":"10:00:59.6250934","counter":5}
[x] Sent {"name":"Filip","time":"10:01:00.8025555","counter":6}
[x] Sent end marker 'EndMarker'
Press [enter] to exit.
C:\Windows\System32\cmd.exe
C:\Users\grygo\source\repos\RSI\lab12\Send2>dotnet run
Filip Strózik, 260377
Piotr Grygoruk, 260299
06 cze, 08:00:51
7.0.3
grygo
Microsoft Windows NT 10.0.19045.0
192.168.56.1
[x] Sent {"name":"Piotr","time":"10:00:52.0446623","counter":0}
[x] Sent {"name":"Piotr","time":"10:00:55.1648365","counter":1}
[x] Sent {"name":"Piotr","time":"10:00:58.1572802","counter":2}
[x] Sent {"name":"Piotr","time":"10:01:00.2660830","counter":3}
[x] Sent {"name":"Piotr","time":"10:01:03.2339483","counter":4}
[x] Sent end marker 'EndMarker'
Press [enter] to exit.

Receive:

```
Filip Strózik, 260377
Piotr Grygoruk, 260299
06 Jun, 10:04:58
17.0.6
filip
Windows 11
10.182.4.243
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received '{"name":"Filip","time":"10:05:12.6482830","counter":0}'
Name: Filip
Time: 10:05:12.648283
Counter: 0
 [x] Received '{"name":"Filip","time":"10:05:13.8895312","counter":1}'
Name: Filip
Time: 10:05:13.889531200
Counter: 1
 [x] Received '{"name":"Piotr","time":"10:05:14.0177944","counter":0}'
Name: Piotr
Time: 10:05:14.017794400
Counter: 0
 [x] Received '{"name":"Filip","time":"10:05:15.5187664","counter":2}'
Name: Filip
Time: 10:05:15.518766400
Counter: 2
```

```
 [x] Received '{"name":"Piotr","time":"10:05:16.5598323","counter":1}'
Name: Piotr
Time: 10:05:16.559832300
Counter: 1
 [x] Received '{"name":"Filip","time":"10:05:17.4071516","counter":3}'
Name: Filip
Time: 10:05:17.407151600
Counter: 3
 [x] Received '{"name":"Filip","time":"10:05:18.6156484","counter":4}'
Name: Filip
Time: 10:05:18.615648400
Counter: 4
 [x] Received '{"name":"Piotr","time":"10:05:18.7412923","counter":2}'
Name: Piotr
Time: 10:05:18.741292300
Counter: 2
 [x] Received '{"name":"Filip","time":"10:05:20.4253974","counter":5}'
Name: Filip
Time: 10:05:20.425397400
Counter: 5
 [x] Received '{"name":"Piotr","time":"10:05:21.0520706","counter":3}'
Name: Piotr
Time: 10:05:21.052070600
Counter: 3
 [x] Received '{"name":"Filip","time":"10:05:21.6954972","counter":6}'
Name: Filip
Time: 10:05:21.695497200
Counter: 6
 [x] Received 'EndMarker'
 [x] Received '{"name":"Piotr","time":"10:05:23.8849134","counter":4}'
Name: Piotr
Time: 10:05:23.884913400
Counter: 4
 [x] Received 'EndMarker'
Received 2 end markers. Exiting...
```

5. Opis protokołu RabbitMQ

RabbitMQ jest oprogramowaniem open source, które implementuje protokół AMQP (Advanced Message Queuing Protocol). Jest serwerem kolejek wiadomości, który pozwala na komunikację między aplikacjami. RabbitMQ jest napisany w Erlangu, ale posiada interfejsy do wielu języków oprogramownia. AMQP jest protokołem typu publish-subscribe, co oznacza, że wiadomości są wysyłane do kolejnki a następnie do subskrybentów tejże kolejki. Zaawansowany protokół kolejkowania wiadomości został zaprojektowany w celu zapewnienia

funkcji takich jak open source, standaryzacja, niezawodność, interoperacyjność i bezpieczeństwo. Pomaga połączyć organizację, czas, przestrzeń i technologie. Protokół jest binarny, z funkcjami takimi jak negocjacje, wielokanałowość, przenośność, wydajność i asynchroniczne przesyłanie komunikatów.

Źródła:
https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html
https://www.youtube.com/watch?v=bfVddTJNiAw&t=2089s
https://hub.docker.com/_/rabbitmq