Systemy Rozproszone - Ćwiczenie 7

1 RabbitMQ - Hello World

Celem ćwiczenia jest przesłanie komunikatu od nadawcy do odbiorcy za pomocą brokera komunikatów RabbitMq. Serwer RabbitMq dostępny jest pod adresem: `213.184.8.82:15672`, login: student.

Pierwszym krokiem jest pobranie bibliotek klienckich i dołączenie ich do projektu. Ze strony `https://www.rabbitmq.com/download.html` pobierz Java Client (zip) i rozpakuj pliki klienta. W projekcie korzystającym z RabbitMq należy dodać bibliotekę `rabbitmq-client.jar` W Netbeans rozwiń drzewo projektu, PPM na `Libraries`, wybierz `Add JAR/folder` i wskaż lokalizację pliku `rabbitmq-client.jar`.

1.1 Nadawca wiadomości

Proces nadawcy najpierw łączy się z brokerem RabbitMq tworząc połączenie i kanał komunikacyjny. Następnie nadawca deklaruje kolejkę o nazwie hello rozszerzonej o twój numer indeksu (zob. dokumentacja queueDeclare).

Deklarownie kolejki jest idempotentne - broker utworzy kolejkę tylko wtedy jeżeli ta nie została wcześniej utworzona.

Po utworzeniu kolejki publikacja wiadomości odbywa się za pomocą metody channel.basicPublish. Wiadomość publikowana jest przez domyślną rozdzielnię do kolejki o nazwie `hello`. Teścią wiadomości jest łańcuch znaków - może to być json, xml lub dowolny inny format. Po wysłaniu wiadomości zamyka połączenie z brokerem.

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;

/* plik Send.java */

public class Send {

    private final static String QUEUE_NAME = "
        hello_numer_indeksu";
```

```
    public static void main(String[] argv) throws java.io
        .IOException {
        ConnectionFactory factory = new ConnectionFactory
            ();
        factory.setHost("213.184.8.82");
        factory.setUsername("student");
        factory.setPassword("xxx");
        factory.setVirtualHost("systemy_rozproszone");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare(QUEUE_NAME, false, false,
            false, null);
        String message = "Hello World!";
        channel.basicPublish("", QUEUE_NAME, null,
            message.getBytes());
        System.out.println(" [x] Sent '" + message + "'")
            ;
        channel.close();
        connection.close();
    }
}
```

1.2 Odbiorca wiadomości

W tym kroku utowrzysz proces będący odbiorcą wiadomości. Proces najpierw łączy się z brokerem RabbitMq tworząc połączenie i kanał komunikacyjny. Następnie odbiorca deklaruje kolejkę o nazwie `hello` i konsumenta wiadomości. Zwróć uwagę, że i nadawca i odbiorca deklarują kolejkę w ten sam sposób - dzięki temu nie ma znaczenia czy który proces zostanie uruchomiony jako pierwszy.

Samo pobieranie wiadomości z kolejki i ich przetwarzanie odbywa się w pętli `while`.

```
/* plik Recv.java */

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.QueueingConsumer;

public class Recv {

    private final static String QUEUE_NAME = "
        hello_numer_indeksu";

    public static void main(String[] argv)
```

```
            throws java.io.IOException,
            java.lang.InterruptedException {

        ConnectionFactory factory = new ConnectionFactory
            ();
        factory.setHost("213.184.8.82");
        factory.setUsername("student");
        // popros prowadzacego o haslo
        factory.setPassword("xxx");
        factory.setVirtualHost("systemy_rozproszone");

        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false,
            false, null);
        System.out.println(" [*] Waiting for messages. To
            exit press CTRL+C");
        QueueingConsumer consumer = new QueueingConsumer(
            channel);
        channel.basicConsume(QUEUE_NAME, true, consumer);

        while (true) {
            QueueingConsumer.Delivery delivery = consumer
                .nextDelivery();
            String message = new String(delivery.getBody
                ());
            System.out.println(" [x] Received '" +
                message + "'");
        }
    }
}
```

1.3 Ćwiczenia

- Uruchom nadawcę kilka razy bez uruchamiania odbiorcy. Następnie uruchom odbiorcę. Sprawdź czy komunikaty zostały dostarczone.
- Zmodyfikuj program tak, żeby nadawca wysyłał do odbiorcy treść wiadomości wpisanej z klawiatury
- Zmodyfikuj program tak, żeby przesyłać wiadomość zawierającą strukturę (np. tytuł wiadomości, treść wiadomości, data wysłania)

2 Rozdzielanie zadań

Tym razem stworzysz producenta i kilku konsumentów. Producent będzie umieszczać w kolejce zadania do wykonania (bliżej nieokreślone czasochłonne operacje), natomiast konsumenci (workerzy) będą pobierać zadania i wykonywać je.

2.1 Producent

Producent działa podobnie do tego z poprzedniego przykładu. Z linii poleceń pobierana jest wiadomość (metoda `getMessage`), która publikowana jest do kolejki $\texttt{task}_{q}ueue$.

```
import java.io.IOException;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.MessageProperties;

public class Producer {

    private static final String TASK_QUEUE_NAME = "
        task_queue_nr_indeksu";

    private static String getMessage(String[] strings
        ) {
        if (strings.length < 1) {
            return "Hello World...";
        }
        return joinStrings(strings, " ");
    }

    private static String joinStrings(String[]
        strings, String delimiter) {
        int length = strings.length;
        if (length == 0) {
            return "";
        }
        StringBuilder words = new StringBuilder(
            strings[0]);
        for (int i = 1; i < length; i++) {
            words.append(delimiter).append(strings[i
                ]);
        }
        return words.toString();
    }
```

```
    public static void main(String[] argv)
            throws java.io.IOException {

        ConnectionFactory factory = new
            ConnectionFactory();
        factory.setHost("213.184.8.82");
        factory.setUsername("student");
        factory.setPassword("srwmii");
        factory.setVirtualHost("systemy_rozproszone")
            ;

        Connection connection = factory.newConnection
            ();
        Channel channel = connection.createChannel();

        channel.queueDeclare(TASK_QUEUE_NAME, true,
            false, false, null);

        String message = getMessage(argv);

        channel.basicPublish("", TASK_QUEUE_NAME,
                MessageProperties.
                    PERSISTENT_TEXT_PLAIN,
                message.getBytes());
        System.out.println("␣[x]␣Sent␣'" + message +
            "'");

        channel.close();
        connection.close();
    }
}
```

2.2 Konsument

Zadaniem konsumenta jest pobieranie wiadomości z kolejki i przetwarzanie ich. W metodzie `doWork` symulujemy wykonywanie czasochłonnych operacji - każda kropka w wiadomości jest równoważna 1 sekundzie symulowanych obliczeń (w rzeczywistości może to być renderowanie, konwersja z jednego formatu do drugiego, budowanie archiwum, wyliczanie sum kontrolnych, itp.). Istotną sprawą jest chwila potwierdzenia wiadomości (wysłanie ACK) - odbiorca wiadomości może to zrobić zaraz po odebraniu wiadomościu lub po jej przetworzeniu. Jest to istotne w kontekście awarii workera - RabbitMq usuwa wiadomość po otrzymaniu ACK (patrz wykład dot. tolerowania awarii).

```
public class Worker {

    private static final String TASK_QUEUE_NAME = "
        task_queue_nr_indeksu";

    public static void main(String[] argv)
            throws java.io.IOException,
            java.lang.InterruptedException {

        ConnectionFactory factory = new
            ConnectionFactory();
        factory.setHost("213.184.8.82");
        factory.setUsername("student");
        factory.setPassword("srwmii");
        factory.setVirtualHost("systemy_rozproszone")
            ;
        Connection connection = factory.newConnection
            ();
        Channel channel = connection.createChannel();

        channel.queueDeclare(TASK_QUEUE_NAME, true,
            false, false, null);
        System.out.println(" [*] Waiting for messages
            . To exit press CTRL+C");

        //int prefetchCount = 1;
        //channel.basicQos(prefetchCount);

        QueueingConsumer consumer = new
            QueueingConsumer(channel);
        // czy wysylac ack przy odebraniu (true) czy
            po wykonaniu zadania
        boolean ackOnDelivery = true;
        channel.basicConsume(TASK_QUEUE_NAME,
            ackOnDelivery, consumer);

        while (true) {
            QueueingConsumer.Delivery delivery =
                consumer.nextDelivery();
            String message = new String(delivery.
                getBody());

            System.out.println(" [x] Received '" +
                message + "'");
            doWork(message);
            System.out.println(" [x] Done");
```

```
            if (!ackOnDelivery) {
                // wyslij ack dopiero po wykonaniu
                    zadania
                channel.basicAck(delivery.getEnvelope
                    ().getDeliveryTag(), false);
            }
        }
    }

    private static void doWork(String task) throws
        InterruptedException {
        for (char ch : task.toCharArray()) {
            if (ch == '.') {
                System.out.println("␣[x]␣␣␣Doing␣
                    something␣really␣time␣consuming!")
                    ;
                Thread.sleep(1000);
            }
        }
    }
}
```

2.3 Zadania

- Uruchom producenta kilkukrotnie, następnie uruchom konsumenta - zobacz w jaki sposób wykonywane będą zadania przez workera
- Uruchom kilku konsumentów, a następnie uruchom prducenta kilkukrotnie - zobacz w jaki sposób wykonywane będą zadania przez workerów
- Zmodyfikuj producenta tak, żeby za 1 uruchomieniem wysyłał kilka wiadomości, przy czym ich czasochłonność (liczba kropek) powinna być pobierana z linii poleceń
- Zmodyfikuj workera tak, żeby z zadanym prawopodobieństwem ulegał awarii (zasymuluj to przez rozłączenie i ponowne połączenie z RabbitMq w trakcie wykonywania `doWork`). Zobacz jaki wpływ ma ustawienie wartości `ackOnDelivery` w przypadku zawodnych workerów.
- Uruchom 2 konsumentów i wyślij od nich zadania naprzemiennie długie i krótkie. Zobacz w jaki sposób wykonywane będą zadania przez workerów. Odkomentuj linie używające metody `channel.basicQos`, sprawdź jaki wpływ miało to na workerów. basicQos określa ile niepotwierdzonch wiadomości może być przetrzymywanych w buforze odbiorczym. Domyślne ustawienia nie narzucają żadnych ograniczeń,

wieć RabbitMq będzie bezzwłocznie wysyłać wiadomości do workerów. Zastanów co się dzieje gdy prefetchCount ustawiony jest na 1 (wskazówki są tutaj)?