RELATÓRIO EXPERIMENTO 1

Para o desenvolvimento do experimento foi escolhido a linguagem JavaScript por questão de afinidade.

Tutorial one: “Hello World!”

Arquivos:

send.js

```
const amqp = require("amqplib")

async function sendMessage(){
    try {
        const connection = await amqp.connect("amqp://localhost")
        const channel = await connection.createChannel()

        const queue = "hello"
        const msg = "Hello world"

        channel.assertQueue(queue, {
            durable: false
        })

        channel.sendToQueue(queue, Buffer.from(msg))
        console.log(" [x] Sent %s", msg)

        setTimeout(() => {
            connection.close()
            process.exit(0)
        }, 500)
    } catch (error) {
        console.error(error)
    }
}

sendMessage()
```

receive.js

```js
const amqp = require("amqplib")

async function receiveMessage(){
    try {
        const connection = await amqp.connect("amqp://localhost")
        const channel = await connection.createChannel()

        const queue = "hello"

        channel.assertQueue(queue, {
            durable: false
        })

        console.log(" [*] Waiting for messages in %s. To exist press CTRL+C", queue)
        channel.consume(queue, (msg) => {
            console.log(" [x] Received %s", msg.content.toString())
        }, { noAck: true })
    } catch (error) {
        console.error(error)
    }
}

receiveMessage()
```

Acontecimentos: Nesse tutorial é criado dois arquivos um que mandara um “Hello World” para uma fila chama “hello” e o outro arquivo que irá consumir os dados dessa fila.

Saída:

send.js

```
C:\Users\gabri\OneDrive\Documentos\WebDesing11\exp1 mensageria\exemplo1>nod
e send.js
 [x] Sent Hello world
```

receive.js

```
e receive.js
 [*] Waiting for messages in hello. To exist press CTRL+C
 [x] Received Hello world
```

Tutorial two: Work Queues

Arquivos:

new_task.js

```
const amqp = require("amqplib")

async function sendMessage(){
    try {
        const connection = await amqp.connect("amqp://localhost")
        const channel = await connection.createChannel()

        const queue = "task_queue"
        const msg = process.argv.slice(2).join(" ") || "Hello World!"

        channel.assertQueue(queue, {
            durable: true
        })

        setInterval(() => {
            channel.sendToQueue(queue, Buffer.from(msg), { persistent: true })

            console.log(" [x] Sent %s", msg)

        }, 5000)

    } catch (error) {
        console.error(error)
    }
}

sendMessage()
```

worker.js

```
const amqp = require("amqplib")

async function sendMessage(){
    try {
        const connection = await amqp.connect("amqp://localhost")
        const channel = await connection.createChannel()

        const queue = "task_queue"

        channel.assertQueue(queue, {
            durable: true
        })

        channel.consume(queue, (msg) => {
            const secs = msg.content.toString().split(".").length - 1

            console.log(" [x] Received %s", msg.content.toString())
            setTimeout(() => {
                console.log(" [x] Done")
            }, secs * 1000)
        }, { noAck: true })
    } catch (error) {
        console.error(error)
    }
}

sendMessage()
```

Acontecimentos: Esse tipo de fila é usado quando quer evitar que uma fila fique ocupada com uma tarefa muito demorada, então se divide o consumo entre duas ou mais “workers”. Quando se executa o código com 2 terminais com o worker.js e 1 terminal com o new_task.js, acaba que cada tarefa vai intercalar entre os 2 workers.

Saída:

new_task.js

```
[x] Sent Hello World!
[x] Sent Hello World!
[x] Sent Hello World!
```

worker.js - 1

```
[x] Received Hello World!
[x] Done
[x] Received Hello World!
[x] Done
```

worker.js - 2

```
[x] Received Hello World!
[x] Done
```

Tutorial three: Publish/Subscribe

Arquivos:

emit_log.js

```
const amqp = require("amqplib")

async function sendMessage(){
    try {
        const connection = await amqp.connect("amqp://localhost")
        const channel = await connection.createChannel()

        const exchange = "logs"
        const msg = process.argv.slice(2).join(" ") || "Hello World!"

        channel.assertExchange(exchange, "fanout", {
            durable: false
        })

        channel.publish(exchange, "", Buffer.from(msg))
        console.log(" [x] Sent %s", msg)

        setTimeout(() => {
            connection.close()
            process.exit(0)
        }, 500)

    } catch (error) {
        console.error(error)
    }
}

sendMessage()
```

receive_log.js

```
const amqp = require('amqplib');

async function consumeMessages() {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();
    const exchange = 'logs';

    await channel.assertExchange(exchange, 'fanout', { durable: false });

    const { queue } = await channel.assertQueue('', { exclusive: true });

    console.log(" [*] Waiting for messages in %s. To exit press CTRL+C", queue);

    await channel.bindQueue(queue, exchange, '');

    channel.consume(queue, function(msg) {
      if (msg.content) {
        console.log(" [x] %s", msg.content.toString());
      }
    }, { noAck: true });
  } catch (error) {
    console.error(error);
  }
}

consumeMessages();
```

Acontecimentos: Nesse exemplo uma mensagem enviada é recebida por 2 ou mais consumidores, meio que uma ligação de um para muitos.

Saída:

emit_log.js

```
[x] Sent Hello World!
```

receive_log.js

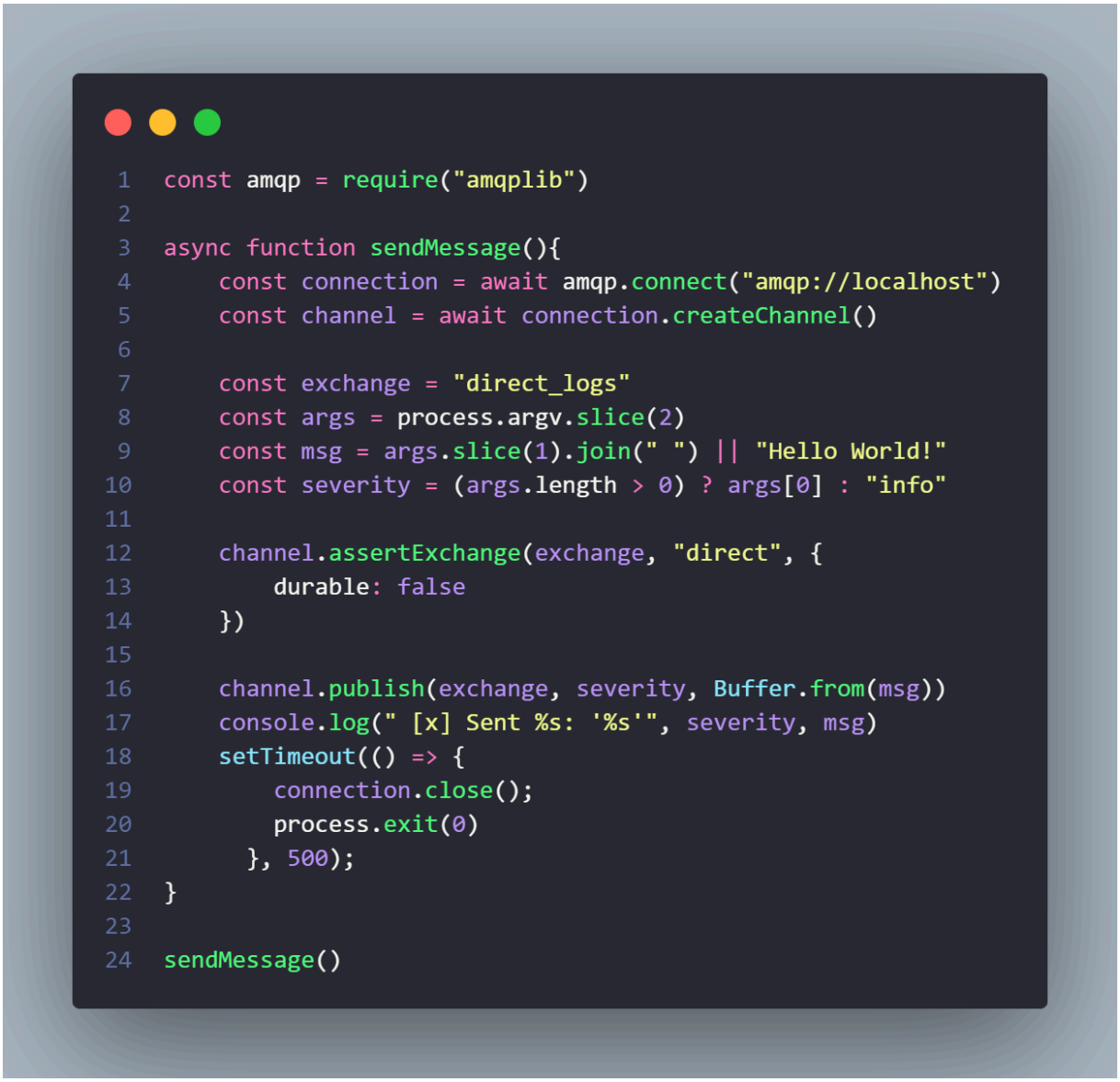
```
 [*] Waiting for messages in amq.gen-9hMsqPksRxO
pxk-0o6XqEA. To exit press CTRL+C
```

```
 [*] Waiting for messages in amq.gen-HfWLWx5bqvJ
UolOuUJAF6w. To exit press CTRL+C
 [x] Hello World!
```

Tutorial four: Routing

Arquivos:

emit_log_direct.js



```
const amqp = require("amqplib")

async function sendMessage(){
    const connection = await amqp.connect("amqp://localhost")
    const channel = await connection.createChannel()

    const exchange = "direct_logs"
    const args = process.argv.slice(2)
    const msg = args.slice(1).join(" ") || "Hello World!"
    const severity = (args.length > 0) ? args[0] : "info"

    channel.assertExchange(exchange, "direct", {
        durable: false
    })

    channel.publish(exchange, severity, Buffer.from(msg))
    console.log(" [x] Sent %s: '%s'", severity, msg)
    setTimeout(() => {
        connection.close();
        process.exit(0)
      }, 500);
}

sendMessage()
```

receive_log_direct.js

```js
const amqp = require("amqplib")

async function receiveMessage(){
    const args = process.argv.slice(2)
    if (args.length == 0) {
        console.log("Usage: receive_logs_direct.js [info] [warning] [error]");
        process.exit(1);
    }

    const connection = await amqp.connect("amqp://localhost")
    const channel = await connection.createChannel()

    const exchange = "direct_logs"

    channel.assertExchange(exchange, "direct", {
        durable: false
    })
    const { queue } = channel.assertQueue("", { exclusive: true })
    console.log(' [*] Waiting for logs. To exit press CTRL+C')
    args.forEach((severity) => {
        channel.bindQueue(queue, exchange, severity)
    })
    channel.consume(queue, (msg) => {
        console.log(" [x] %s: '%s'", msg.fields.routingKey, msg.content.toString())
    }, { noAck: true })
}

receiveMessage()
```

Acontecimentos: Nesse tutorial foi mostrado como direcionar apenas informações que contém um argumento específico para cada consumidor. Como por exemplo um consumidor que recebe apenas dados com o argumento “erro” e um consumidor que recebe os “erro” e “aviso”.

Saída:

emit_log_direct.js

```
xp1 mensageria\exemplo4>node emit_log_direct.js
error "apenas o consumidor que aceita erro"
 [x] Sent error: 'apenas o consumidor que aceita
 erro'
```

receive_log_direct.js - 1

```
 [*] Waiting for logs. To exit press CTRL+C
 [x] error: 'apenas o consumidor que aceita erro
'
```

receive_log_direct.js - 2

```
ct.js info
 [*] Waiting for logs. To exit press CTRL+C
```

Tutorial five: topics

Arquivos:

emit_log_topic.js

```
const amqp = require("amqplib")

async function sendMessage(){
    const connection = await amqp.connect("amqp://localhost")
    const channel = await connection.createChannel()

    const exchange = 'topic_logs'
    const args = process.argv.slice(2)
    const key = (args.length > 0) ? args[0] : 'anonymous.info'
    const msg = args.slice(1).join(' ') || 'Hello World!'

    channel.assertExchange(exchange, "topic", {
        durable: false
    })

    channel.publish(exchange, key, Buffer.from(msg))
    console.log(" [x] Sent %s:'%s'", key, msg)

    setTimeout(function() {
        connection.close();
        process.exit(0)
    }, 500)
}

sendMessage()
```

receive_log_topic.js

```
const amqp = require("amqplib")

async function receiveMessage(){
    const args = process.argv.slice(2)
    if (args.length == 0) {
        console.log("Usage: receive_logs_topic.js <facility>.<severity>")
        process.exit(1)
    }
    const connection = await amqp.connect("amqp://localhost")
    const channel = await connection.createChannel()

    const exchange = "topic_logs"

    channel.assertExchange(exchange, "topic", {
        durable: false
    })

    const { queue } = await channel.assertQueue("", { exclusive: true })
    console.log(' [*] Waiting for logs. To exit press CTRL+C')

    args.forEach((key) => {
        channel.bindQueue(queue, exchange, key);
    })

    channel.consume(queue, (msg) => {
        console.log(" [x] %s:'%s'", msg.fields.routingKey, msg.content.toString())
    }, { noAck: true })
}

receiveMessage()
```

Acontecimentos: Funciona bem parecido com o tutorial quatro, porém agora é possível fazer roteamento com base em vários critérios.
Caso a fila possua um “#” ela receberá todas as mensagens;
Caso tenha um “*” + “algum texto” ele receberá todos desse “texto”;
Caso só tenha o “.texto” será algo do tipo 1 para 1.

Saída:

emit_log_direct.js

```
C:\Users\gabri\OneDrive\Documentos\WebDesingII\exp1 mensageria\exemplo5>nod
e emit_log_topic.js "kern.critical" "A critical kernel error"
 [x] Sent kern.critical:'A critical kernel error'
```

receive_log_direct.js

```
e receive_logs_topics.js "#"
 [*] Waiting for logs. To exit press CTRL+C
 [x] kern.critical:'A critical kernel error'
```

Tutorial six: RPC

Arquivos:

rpc_client.js

```
const amqp = require("amqplib")

async function receiveMessage(){
    const args = process.argv.slice(2)
    if (args.length == 0) {
        console.log("Usage: rpc_client.js num")
        process.exit(1)
    }
    const connection = await amqp.connect("amqp://localhost")
    const channel = await connection.createChannel()

    const { queue } = await channel.assertQueue("", { exclusive: true })

    const correlationId = generateUuid()
    const num = parseInt(args[0])

    console.log(' [x] Requesting fib(%d)', num)
    channel.consume(queue, (msg) => {
        if(msg.properties.correlationId == correlationId) {
            console.log(' [.] Got %s', msg.content.toString())
            setTimeout(() => {
                connection.close()
                process.exit(0)
            }, 500)
        }
    }, { noAck: true })

    channel.sendToQueue("rpc_queue",
        Buffer.from(num.toString()), {
            correlationId: correlationId,
            replyTo: queue
        })
}

function generateUuid() {
    return Math.random().toString() +
        Math.random().toString() +
        Math.random().toString()
}

receiveMessage()
```

rpc_server.js

```
const amqp = require("amqplib")

async function sendMessage(){
    const connection = await amqp.connect("amqp://localhost")
    const channel = await connection.createChannel()

    const queue = "rpc_queue"

    channel.assertQueue(queue, { durable: false })
    channel.prefetch(1)
    console.log(' [x] Awaiting RPC requests')
    channel.consume(queue, (msg) => {
        const n = parseInt(msg.content.toString())
        console.log(" [.] fib(%d)", n)

        const r = fibonacci(n)

        channel.sendToQueue(msg.properties.replyTo,
            Buffer.from(r.toString()), {
                correlationId: msg.properties.correlationId
            })

        channel.ack(msg)
    })
}

function fibonacci(n) {
    if (n == 0 || n == 1)
      return n;
    else
      return fibonacci(n - 1) + fibonacci(n - 2);
}

sendMessage()
```

Acontecimentos: Nesse exemplo é demonstrado como fazer uma fila que manda uma mensagem ou uma chamada a uma função e espera um retorno, para fins educativos foi utilizado uma função que retorna o último valor de fibonacci de acordo com o número enviado. Caso seja pedido para executar o fibonacci passando o número 60, vai demorar MUITO, mas fica ali esperando terminar.

Saída:
rpc_client.js

```
de rpc_client.js 5
 [x] Requesting fib(5)
 [.] Got 5
```

rpc_server.js

```
 [x] Awaiting RPC requests
 [.] fib(5)
```

Dificuldades:

A maior dificuldade sem dúvidas foi fazer o rabbitMQ server funcionar, além disso foi meio estranho entender teoricamente o que mensageria queria dizer mas ao realizar os tutoriais ficou mais claro.