Universidade Federal
do Espírito Santo

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Processamento Paralelo e Distribuído – Turmas 01 e 02 (EARTE) – 2022/1

Prof. Rodolfo da Silva Villaça – rodolfo.villaca@ufes.br

Laboratório III – Comunicação Indireta (Publish/Subscribe)

1. Objetivos

- Experimentar a implementação de sistemas de comunicação indireta por meio de *middleware Publish/Subscribe* (Pub/Sub) com Filas de Mensagens.
- Sincronizar a troca de mensagens entre os componentes do sistema.
- Utilizar *brokers* para gerenciamento da fila de mensagens na implementação de sistemas distribuídos.

2. Conceitos Básicos

Publish/Subscribe é um padrão arquitetural onde existem os *Publishers* (Publicadores) que enviam as mensagens e os *Subscribers* (Assinantes) que recebem as mensagens. Em uma maneira mais prática, sempre que houver algum evento, o publicador vai enviar uma mensagem para que os assinantes sejam notificados. O padrão arquitetural *Publish/Subscribe* (Pub/Sub) se resume na imagem a seguir: existe um publicador, que vai enviar uma mensagem em um canal (*Channel*) que redistribui uma cópia da mensagem para cada assinante. Em nosso caso, o canal é representado por um conjunto de filas gerenciados pelo *broker*.

Nenhuma das partes se conhece, muito menos sabem os detalhes de suas implementações. Apenas conhecem o endereço do *broker*. O modelo de comunicação Pub/Sub pode trazer baixo acoplamento e facilidade de escalabilidade em um ecossistema de software. Por ser assíncrono não haverá problemas com *timeouts* ou quebras de processamento por erros encadeados.

Mosquitto vs RabbitMQ

Mosquito é um *broker* de mensagens Pub/Sub, com licença de software livre e que implementa o protocolo MQTT (*Message Queue Telemetry Transport*). É leve e adequado para uso em todos os dispositivos, desde computadores de placa única de baixo consumo (Raspberry Pi, Arduino, ESP32, etc) até servidores completos. O protocolo MQTT fornece um método leve para implementação do modelo de comunicação indireta usando Pub/Sub.

O RabbitMQ também é um *broker* de mensagens Pub/Sub que oferece aos seus aplicativos uma plataforma comum para enviar e receber mensagens, e armazenamento persistente destas mensagens até serem recebidas pelos assinantes (comunicação assíncrona). *Advanced Messaging Queuing Protocol* (AMQP) é o protocolo base no qual foi construído o RabbitMQ.

Mosquitto e RabbitMQ pertencem à categoria "Fila de Mensagens" (*Message Queue*) da pilha de tecnologias para implementação de Sistemas Distribuídos baseados em Comunicação Indireta.

3. Atividade I: Uso do *broker* Mosquitto MQTT

a) Certificar-se de que o *broker* Mosquitto está instalado e em execução:
/etc/init.d/mosquitto status

b) Implementar o *Publisher* "Pub1" (*mqttpub1.py)*, em Python, que fará o envio de números aleatórios no intervalo [100..150] na fila "*rsv/light*" do *broker* MQTT. A ideia aqui é simular um sensor de luminosidade;

```
import paho.mqtt.client as mqtt
from random import randrange, uniform
import time

mqttBroker = "127.0.0.1"
#mqttBroker = "broker.emqx.io"

client = mqtt.Client("Node_1")
client.connect(mqttBroker)

while True:
   randNumber = uniform(100.0, 150.0)
   client.publish("rsv/light", randNumber)
   print("Just published " + str(randNumber) + " to topic rsv/light")
   time.sleep(1)
```

c) Implementar o Publisher "Pub2" (*mqttpub2.py)*, em Python, que fará o envio de números aleatórios no intervalo [15..25] na fila "*rsv/temp*" do *broker* MQTT. A ideia aqui é simular um sensor de temperatura;

```
import paho.mqtt.client as mqtt
from random import randrange, uniform
import time

mqttBroker = "127.0.0.1"
#mqttBroker = "broker.emqx.io"

client = mqtt.Client("Node_2")
client.connect(mqttBroker)

while True:
```

```
    randNumber = randrange(15, 25)
    client.publish("rsv/temp", randNumber)
    print("Just published " + str(randNumber) + " to topic rsv/temp")
    time.sleep(1)
```

d) Implementar o *Subscriber* "Sub1" (*mqttsub1.py*), em Python, que assinará as filas "*rsv/temp*" e "*rsv/light*" no *broker* MQTT local (127.0.0.1) usando o Mosquitto;

```
mqttsub.py

import paho.mqtt.client as mqtt
import time

def on_message(client, userdata, message):
    print("received message: " ,str(message.payload.decode("utf-8")))

mqttBroker = "127.0.0.1"
#mqttBroker = "broker.emqx.io"

client = mqtt.Client("Node_3")
client.connect(mqttBroker)

client.loop_start()

client.subscribe("rsv/temp")
client.subscribe("rsv/light")
client.on_message=on_message

time.sleep(30000)
client.loop_stop()
```

e) Altere a variável *mqttBroker* de 127.0.0.1 (local) para o endereço de um broker público, cujo nome é "*broker.emqx.io*" e refaça o experimento.

- Observe (ou, provoque, se necessário) a interferência das publicações dos demais grupos no seu programa assinante (subscriber).
- Além disso, busque por apps para celulares Android/IPhone que sejam publicadores/ assinantes MQTT e experimente-os, interagindo com seus códigos em Python;

4. Atividade II: Uso do Rabbit MQ

a) Certificar-se de que o *broker RabbitMQ* está instalado e em execução:
/etc/init.d/rabbitmq-server status

b) Implementar o *Publisher* "Pub1" (*rabbitpub1.py)*, em Python, que fará o envio da mensagem "Hello World #1!" para a fila "*rsv/hello*" do *broker* AMQP RabbitMQ em *localhost* (127.0.0.1);

```
import pika

connection = pika.BlockingConnection(
```

```
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='rsv/hello')

channel.basic_publish(exchange='', routing_key='rsv/hello', body='Hello World #1!')
print("Sent 'Hello World #1!'")
connection.close()
```

c) Implementar o *Publisher* "Pub2" (*rabbitpub2.py)*, em Python, que fará o envio da string "Hello World #2!" para a fila "*rsv/hello*" do *broker* AMQP RabbitMQ em *localhost* (127.0.0.1);

```
import pika

connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='rsv/hello')

channel.basic_publish(exchange='', routing_key='rsv/hello', body='Hello World #2!')
print("Sent 'Hello World #2!'")
connection.close()
```

d) Implementar o *Subscriber* "Sub1", em Python, que assina a fila "*rsv/hello*" do broker AMQP RabbitMQ em *localhost* (127.0.0.1);

```
import pika, sys, os

def main():
    connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    channel.queue_declare(queue='rsv/hello')

    def callback(ch, method, properties, body):
        print("Received %r" % body)

    channel.basic_consume(queue='rsv/hello', on_message_callback=callback, auto_ack=True)

    print("Waiting for messages. To exit press CTRL+C")
    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

5. Tarefas

Com base nestes exemplos você precisará construir um protótipo similar a um minerador de criptomoedas do tipo *bitcoin*, porém no modelo de comunicação indireta baseado em Pub/Sub, com implementação em Python. Nesta tarefa você poderá escolher o *broker* a ser utilizado: Mosquitto OU RabbitMQ. Neste laboratório a arquitetura ainda seguirá o modelo cliente/servidor.

O servidor deverá ter o seguinte funcionamento:

a) Manter, localmente, enquanto estiver em execução, uma tabela com os seguintes registros:

TransactionID	*Challenge*	*Seed*	*Winner*
int	int	str	int

- *TransactionID*: Identificador da transação, representada por um valor inteiro (32 bits). Use valores incrementais começando em 0 (zero);
- *Challenge*: Valor do desafio criptográfico associado à transação, representado por um número [1..128], onde 1 é o desafio mais fácil. Gere desafios aleatórios, ou sequenciais, experimente as diferentes abordagens;
- *Seed:* String que, se aplicada a função de hashing SHA-1, solucionará o desafio criptográfico proposto;
- *Winner*: ClientID do usuário que solucionou o desafio criptográfico para a referida TransactionID (mesma linha da tabela). Enquanto o desafio não foi solucionado, considere que o ClientID = -1;

b) A tabela deverá ser impressa na tela sempre que houver alguma atualização;
c) Ao carregar, o servidor deverá gerar um novo desafio, com TransactionID=0;
d) Assinar a seguinte fila de mensagens no *broker* Pub/Sub:

Nome da Fila	*Codificação da Mensagem*	*Significado*
ppd/seed	"ClientID/ TransactionID/Seed", formato String*	Seed usada pelo ClientID para resolver o desafio associado à TransactionID. Importante: sempre receber em formato String (str, em Python), com separação usando "/", ou, definir uma mensagem em formato .json

* **Ponto extra se o grupo definir um arquivo .json para essa (e demais) codificações!!!**

d) Publicar nas seguintes filas de mensagens no *broker* Pub/Sub:

Nome da Fila	*Codificação da Mensagem*	*Significado*
ppd/challenge	"TransactionID/ Challenge", formato String*	Divulga o desafio (Challenge) associado ao TransactionID para os assinantes do tópico.
ppd/result	"TransactionID/ ClientID/Seed", formato String*	Divulga o resultado, com a semente (Seed), enviada pelo ClientID para o hashing SHA-1, que resolve o desafio proposto para a referida TransactionID.

d) O servidor deve ficar em loop, recebendo publicações de mensagens de seu interesse ("*ppd/seed*"), processando essas mensagens e i) publicando os resultados na fila "*ppd/result*" e ii) os novos desafios na fila "*ppd/challenge*".

O cliente deverá ter o seguinte funcionamento:

a) Assinar as seguintes filas de mensagens no *broker* Pub/Sub:

Nome da Fila	*Codificação da Mensagem*	*Significado*
ppd/challenge	"TransactionID/ Challenge", formato String*	Receber o desafio (Challenge) associado ao TransactionID.
ppd/result	"TransactionID/ ClientID/Seed", formato String*	Receber o resultado (Seed) do desafio proposto para a referida TransactionID, com a semente (Seed), enviada pelo ClientID.

b) Publicar na seguinte fila de mensagens no broker Pub/Sub:

Nome da Fila	*Codificação da Mensagem*	*Significado*
ppd/seed	"ClientID/ TransactionID/Seed", formato String*	Seed usada pelo ClientID para resolver o desafio associado à TransactionID.

c) Requisitos:

- Cada cliente deverá lançar pelo menos 2 (duas) *threads* (ou processos) concorrentes tentando resolver o desafio;
- O processo principal do cliente deverá permanecer aguardando publicações na fila de resultados "*ppd/result*";
- Caso o desafio seja resolvido durante o processamento de alguma *thread* (ou processo), este processo (ou *thread*) deverá ser encerrado/ interrompido;

- Caso encontre uma solução para um desafio ainda em aberto, o cliente deverá publicar na fila "*ppd/seed*".

d) O processo principal do cliente deverá permanecer em *loop*, esperando mensagens de novos desafios em "*ppd/challenge*" e resultados para os desafios em andamento (fila "*ppd/result*").

4. Instruções Gerais

1. O trabalho pode ser feito em grupos de 2 ou 3 alunos: não serão aceitos trabalhos individuais ou em grupos de mais de 3 alunos;
2. Os grupos deverão implementar os trabalhos usando Python;
3. Data de Entrega: 14/06/2022, 23:59h;
4. A submissão deverá ser feita por meio de um *link* com a disponibilização dos códigos no Github, em modo de acesso público ou, minimamente, que meu e-mail rodolfo.villaca@ufes.br tenha direito de acesso ao código;
5. A documentação deverá ser feita na própria página do Github através do arquivo README[1];
6. O grupo deverá gravar um vídeo de no máximo 5 min apresentando o funcionamento/teste do trabalho.

Bom trabalho!

1 https://guides.github.com/features/wikis/