Laboratório 4: Fila de Mensagens

12/07/2021

O laboratório explorará aplicações com ZeroMQ e RabbitMQ. Caso não tenha familiaridade com a linguagem Python, em https://github.com/std29006/oo-java-e-python são apresentados pequenos exemplos em Python3 sobre conceitos de orientação a objetos que foram trabalhados em Java na disciplina POO29004. Em http://docente.ifsc.edu.br/mello/livros/python/python_cheat_sheet.pdf tem um material de referência rápida sobre a linguagem.

1 ZeroMQ

Os exercícios foram obtidos da documentação oficial do projeto[1] e do livro [1].

1.1 Preparação do ambiente com Python VirtualEnv

No Linux, além dos pacotes referentes para execução de *scripts* python3, é necessário ter o pacote `python3-dev` instalado. Abaixo é feito uso do virtualenv do python para instalar o pacote zmq.

```
python3 -m venv venv
source venv/bin/activate

pip install zmq
```

1.2 Hello World – Envio de strings com o padrão REQ-REP

Com esse padrão o servidor usa um socket do tipo REP e o cliente usa um socket do tipo REQ. Cliente precisa receber resposta do pedido antes que possa fazer um novo pedido para o mesmo servidor.

Os *sockets* ZeroMQ enviam os dados como uma sequência de *bytes*, bem como o total de bytes que foram enviados, e não faz qualquer tipo de tratamento. Ou seja, cabe ao desenvolvedor a responsabilidade para formatar esses dados de acordo com a linguagem de programação que esteja usando. Por exemplo, na linguagem C as *strings* devem ser terminadas com um caractere nulo, porém em outras linguagens isso não é obrigatório. Então, um programa em C ao receber uma string, deve reservar um espaço em memória com um espaço extra para armazenar o caractere nulo.

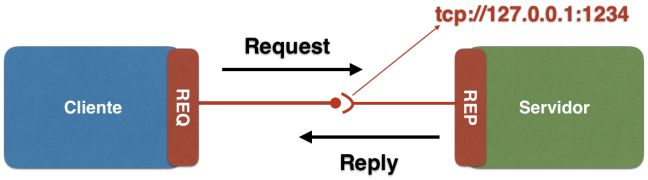

Figura 1: Padrão Reques-Reply

[1] http://zguide.zeromq.org

Listagem 1: Olá mundo – `servidor01.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import zmq

context = zmq.Context()

HOST = "*"
PORT1 = "50007"
PORT2 = "50008"

p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect

s  = context.socket(zmq.REP)    # create reply socket

s.bind(p1)                      # bind socket to address
s.bind(p2)                      # bind socket to address
while True:
  message = s.recv()            # wait for incoming message
  sMsg = message.decode()
  if not "STOP" in sMsg:        # if not to stop...
    print("Sending reply")
    s.send(str("Echo: " + sMsg).encode('utf-8'))
  else:                         # else...
    break                       # break out of loop and end
```

Listagem 2: Olá mundo – `cliente01.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import zmq

context = zmq.Context()

HOST = sys.argv[1] if len(sys.argv) > 1 else "localhost"
PORT = sys.argv[2] if len(sys.argv) > 2 else "50007"

p1 = "tcp://"+ HOST +":"+ PORT # how and where to connect
s  = context.socket(zmq.REQ)    # create request socket

s.connect(p1)                   # block until connected
s.send(b"Hello world")           # send message
message = s.recv()              # block until response
s.send(b"STOP")                  # tell server to stop
print("Reply {}".format(message)) # print result
```

Para executar os códigos acima:

```
python3 servidor01.py
```

```
python3 cliente01.py
```

1.3 Padrão Publish & Subscribe

Aplicativo produtor usa socket PUB e aplicativo consumidor (assinante) usa socket SUB. Quando uma mensagem é publicado, todos os assinantes ativos recebem a mesma. Se o produtor publicar uma mensagem e não houver qualquer assinante ativo, então essa mensagem será perdida. O assinante poderá assinar todas as mensagens publicadas por um produtor (SubscribeALL) ou somente mensagens específicas.

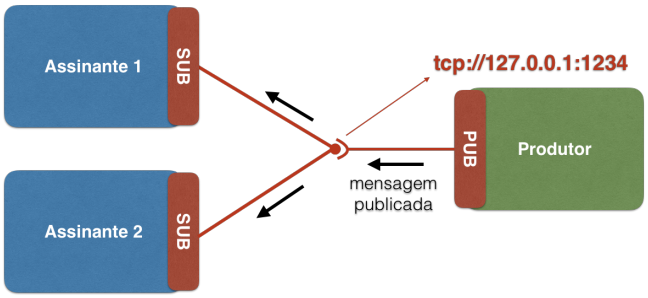

Figura 2: Padrão Publish & Subscribe

Listagem 3: Servidor de tempo multicast

```
#!/usr/bin/env python3
import zmq
import time

context = zmq.Context()
s = context.socket(zmq.PUB)        # create a publisher socket

HOST = "*"
PORT = "50009"

p = "tcp://" + HOST + ":" + PORT      # how and where to communicate
s.bind(p)                           # bind socket to the address

while True:
    time.sleep(5)
    msg = str("TIME " + time.asctime())
    s.send(msg.encode())
```

Listagem 4: Cliente para um servidor de tempo

```
#!/usr/bin/env python3
import zmq, sys

context = zmq.Context()
s = context.socket(zmq.SUB)          # create a subscriber socket

HOST = sys.argv[1] if len(sys.argv) > 1 else "localhost"
PORT = sys.argv[2] if len(sys.argv) > 2 else "50009"

p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
s.connect(p)                         # connect to the server
s.setsockopt(zmq.SUBSCRIBE, b"TIME")  # subscribe to TIME messages

for i in range(5):  # Five iterations
  time = s.recv()   # receive a message
  print(time.decode())
```

1.4 Padrão Pipeline

Pode ser usado para o processamento paralelo de tarefas, ou seja, distribuição de tarefas para um conjunto de processos trabalhadores. Para execução desse experimento, suba primeiro um processo coletor, depois 1 ou mais processos trabalhadores e por fim 1 processo distribuidor. Veja o tempo gasto para os casos com 1, 2 e 4 processos trabalhadores.

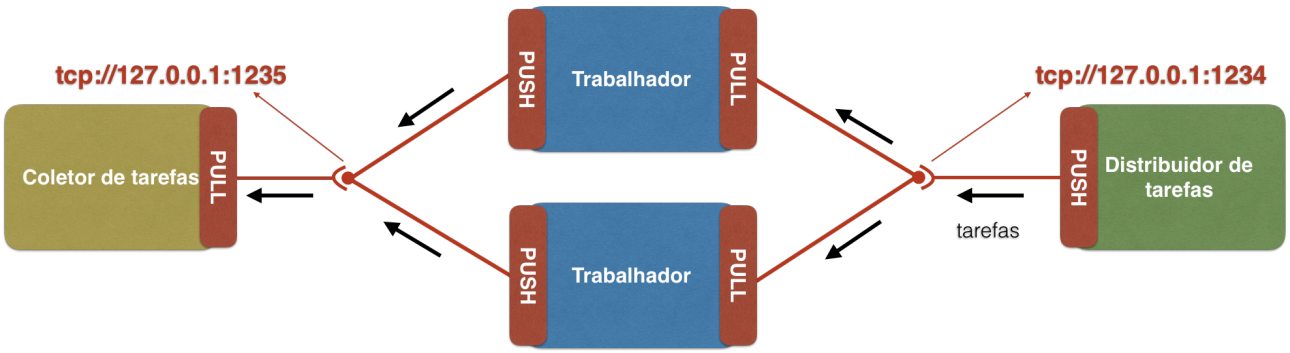

Figura 3: Padrão Pipeline

Listagem 5: Distribuidor de tarefas

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import zmq, time, random

try:
    raw_input
except NameError:
    # Python 3
    raw_input = input

HOST = "*"
HOSTC= "localhost"
PORTD = "50011"
PORTC = "50012"
pD = "tcp://" + HOST + ":" + PORTD
pC = "tcp://" + HOSTC + ":" + PORTC
context = zmq.Context()

# socket para distribuir tarefas
distribuidor = context.socket(zmq.PUSH)
distribuidor.bind(pD)

# socket para coletar respostas
coletor = context.socket(zmq.PUSH)
coletor.connect(pC)

print("Press Enter when the workers are ready: ")
_ = raw_input()
print("Sending tasks to workers...")

# The first message is "0" and signals start of batch
coletor.send(b'0')

# Initialize random number generator
random.seed()

# Send 100 tasks
total_msec = 0
```

```
for task_nbr in range(100):
    # Random workload from 1 to 100 msecs
    workload = random.randint(1, 100)
    total_msec += workload

    distribuidor.send_string(u'%i' % workload)

print("Total expected cost: %s msec" % total_msec)

# Give OMQ time to deliver
time.sleep(1)
```

Listagem 6: Processo trabalhador

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import sys
import time
import zmq

HOSTD = "localhost"
HOSTC = "localhost"
PORTD = "50011"
PORTC = "50012"

pD = "tcp://" + HOSTD + ":" + PORTD
pC = "tcp://" + HOSTC + ":" + PORTC
context = zmq.Context()

# Socket to receive messages on
receiver = context.socket(zmq.PULL)
receiver.connect(pD)

# Socket to send messages to
sender = context.socket(zmq.PUSH)
sender.connect(pC)

print("Worker ready...")
# Process tasks forever
while True:
    s = receiver.recv()

    # Simple progress indicator for the viewer
    sys.stdout.write('.')
    sys.stdout.flush()

    # Do the work
    time.sleep(int(s)*0.001)

    # Send results to coletor
    sender.send(b'')
```

Listagem 7: Processo coletor dos resultados

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import sys
import time
import zmq

HOST = "*"
```

```
PORTC = "50012"
pC = "tcp://" + HOST + ":" + PORTC
context = zmq.Context()

# Socket to receive messages on
receiver = context.socket(zmq.PULL)
receiver.bind(pC)

# Wait for start of batch
s = receiver.recv()

# Start our clock now
tstart = time.time()

# Process 100 confirmations
for task_nbr in range(100):
    s = receiver.recv()
    if task_nbr % 10 == 0:
        sys.stdout.write(':')
    else:
        sys.stdout.write('.')
    sys.stdout.flush()

# Calculate and report duration of batch
tend = time.time()
print("\nTotal elapsed time: %d msec" % ((tend-tstart)*1000))
```

2 RabbitMQ

Execute todos os exemplos (em Java) disponíveis em https://github.com/std29006/rabbitMQ. Se desejar, veja na página oficial do projeto RabbitMQ[2] exemplos para a linguagem Python.

Para a execução desses exemplos é necessário que tenha um servidor rabbitmq em execução. Você pode instalar o servidor em seu computador, seguindo as instruções apresentadas em http://www.rabbitmq.com/install-generic-unix.html. Outra opção, seria subir um contêiner Docker executando o comando abaixo:

```
docker run --rm -d --name rabbit -p 15672:15672 -p 5672:5672 rabbitmq:3-management-alpine
```

Para ver o painel de administração do rabbitMq, abra o navegador e acesse http://localhost:15672. O usuário e senha é `guest`.

Bibliografia

[1] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 3 edition, 2017.

[2] https://github.com/rabbitmq/rabbitmq-tutorials