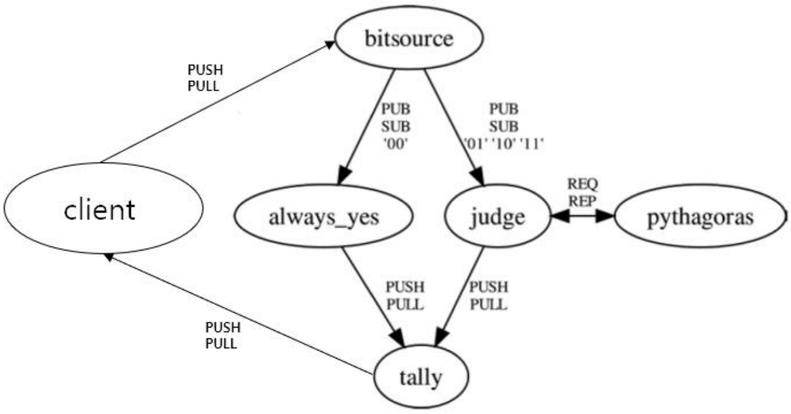
20172679 Cho Hong Rae (조홍래)

Computer Communications Assignment #1

High Level Design



This figure shows a ØMQ messaging fabric linking six different worker nodes all designed to be capable of running on multiple terminals on a local machine. The base source code was used from Listing 8-3 (https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter08/queuepi.py), and was given modifications to allow it to communicate between sockets in different terminals without using the previously provided start_thread method. In addition, apart from the five existing worker nodes, an additional client node was included to receive an input number of data points from the user and carry it on during the program execution.

The main purpose of this program is to compute the value of Pi using a Monte Carlo method. Using randomly generated 32bit binary digits, the workers evaluate whether it is inside of the area of a hypothetical circle depending on the prefix of the binary digits. After deciding on the previous mentioned evaluation, the value of Pi is calculated in the penultimate worker and then sent to the client worker where a graph is plotted in real time using the number of iterations as its x value and the calculated value of Pi as its y value. To calculate the estimate value of Pi the following formula was used.

$$\pi \approx 4 \times (\text{number of points in the circle} / \text{total number of points})$$

This program also demonstrates a ØMQ messaging topology that allows the communication sockets that are in play to use methods such as pub-sub(Publisher-Subcriber), push-pull and req-rep(Request-Reply). ØMQ messaging also has asynchronous characteristics that allow the events occurring in the program to run independently from the main program flow.

Logical View and Process View

1. Client

> **Client key statements**
>
> output_socket = zmq.Context().socket(zmq.PUSH)
>
> input_socket = zmq.Context().socket(zmq.PULL)
>
> plt.plot(x, y, 'o')

When the client process first starts, it receives an input value N from the user which represents the total data points to be generated. It then passes the N value through an output socket using the communication method of Push and Pull. The client then receives information from the tally using an input socket using the communication type Push and Pull. The reason Push and Pull method was implemented was due to its pipelining mechanism allowing it to do some load-balancing when sending relatively bigger amounts of data due to the input N value. While receiving the calculated Pi values from the tally worker, the client worker plots the number of iterations as the x-value and the given Pi value as the y value onto a real time graph using the matplot library. Also a try-catch block handles invalid input values into N to handle exceptions.

2. Bitsource

> **Bitsource key statements**
>
> input_socket = zmq.Context().socket(zmq.PUSH)
>
> output_socket = zmq.Context().socket(zmq.PUB)
>
> output_socket.send_string(ones_and_zeros(B*2))

The Bitsource worker receives the value of N from the client worker using the input socket generated with the Push and Pull method. The output socket of the Bitsource worker is generated with the Pub and Sub method which allows it to send its message to multiple subscribers at once (in this case it is workers always_yes and judge). Then using a loop statement, the output socket sends randomly generated 32 bit binary digits to its subscriber sockets.

3. Always_yes

> **Always_yes key statements**
>
> input_socket = zmq.Context().socket(zmq.SUB)
>
> output_socket = zmq.Context().socket(zmq.PUSH)
>
> input_socket.recv_string() / output_socket.send_string('Y')

The always_yes worker only subscribes to 32 bits from the Bitsource worker that start with '00'. This guarantees that the given digit is inside the circle and sends 'Y' to the tally worker using the output socket generated with the Push and Pull communication method.

4. Judge

Judge key statements

input_socket = zmq.Context().socket(zmq.SUB)

pytha_socket = zmq.Context().socket(zmq.REQ)

output_socket = zmq.Context().socket(zmq.PUSH)

The judge worker, similar to the always_yes worker, subscribes to the binary digits published by the Bitsource worker but only accepts digits that have a prefix of '01', '10' or '11'. Then it uses another socket to request to the Pythagoras worker to calculate the sum-of-squares of number sequences. Then the judge worker determines whether each input coordinate is within the unit circle and then sends 'Y' or 'N' depending on that determined value using the output socket to tally.

5. Pythagoras

Pythagoras key statements

socket = zmq.Context(). Socket(zmq.REP)

numbers = socket.recv_json()

socket.send_json(sum(n*n for n in numbers))

The Pythagoras worker receives a request from the judge worker to calculate the sum-of squares for number sequences. It uses a socket generated with the communication method of Request and Reply.

6. Tally

Tally key statements

decision = input_socket.recv_string()

q+=1 if decision == 'Y' then p+=4

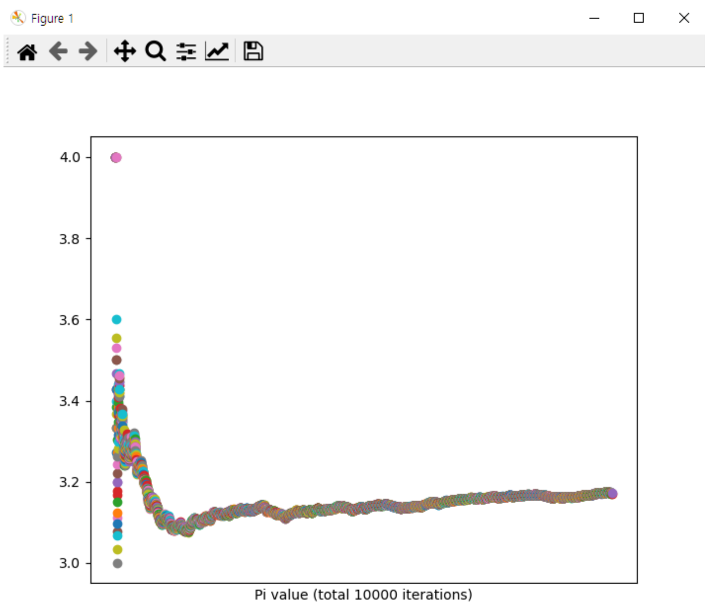
output_socket.send_string()

The tally worker receives all the Y and N decisions from both the judge worker and always_yes worker using an input socket generated with the communication method Push and Pull. After calculating the Pi value using the decision values, the tally worker then proceeds to use the output socket generated with the communication method of Push and Pull to send the number of iterations and Pi value to the client worker.

Test Cases

Realtime plotted graph from client

(N = 10000)



Final Pi value (after 10000 iterations)

➔ **3.1652**

Before input (all terminals waiting)

After input (N = 10000)

References: https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter08/queuepi.py

https://www.101computing.net/estimating-pi-using-the-monte-carlo-method