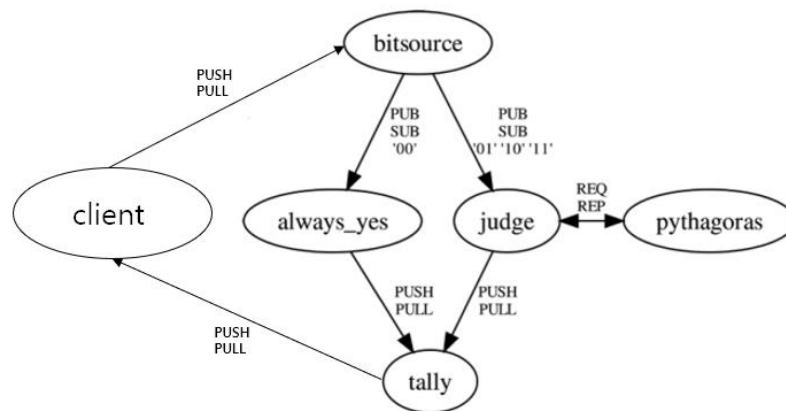


## Demonstrating ØMQ Messaging

### 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-Subscriber), 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 matplotlib 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.



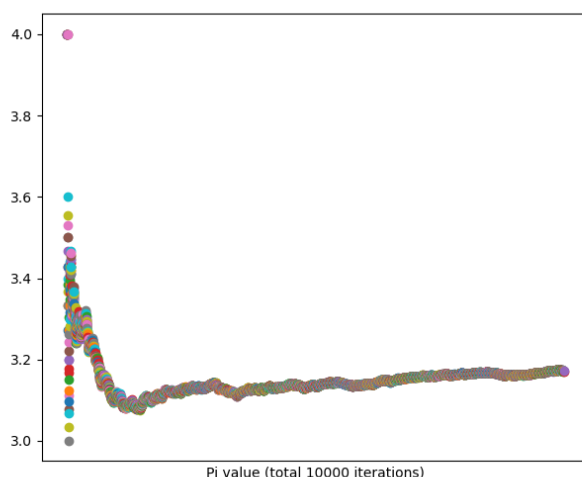
#### Test Cases

##### Realtime plotted graph from client

(N = 10000)

Final Pi value (after 10000 iterations)

→ 3.1652



## Before input (all terminals waiting)

```
(base) C:\Users\Rav\python queuepi_20172679.py client
Enter numbers of data points :

(base) C:\Users\Rav\python queuepi_20172679.py bitsource

(base) C:\Users\Rav\python queuepi_20172679.py judge
tcp://127.0.0.3:6702
tcp://127.0.0.5:6704
url : tcp://127.0.0.4:6703

(base) C:\Users\Rav\python queuepi_20172679.py always_yes
Input Url : tcp://127.0.0.3:6702
Output Url : tcp://127.0.0.5:6704

(base) C:\Users\Rav\python queuepi_20172679.py pythagoras
url : tcp://127.0.0.4:6703

(base) C:\Users\Rav\python queuepi_20172679.py tally
Input Url : tcp://127.0.0.5:6704
Output Url : tcp://127.0.0.1:6700
```

## After input (N = 10000)

```
Iteration 9972 Pi value : 3.164861610516042
Iteration 9973 Pi value : 3.1648453524516193
Iteration 9974 Pi value : 3.165029075595551
Iteration 9975 Pi value : 3.165112781954887
Iteration 9976 Pi value : 3.16479508222133
Iteration 9977 Pi value : 3.164972222110365
Iteration 9978 Pi value : 3.164962918420525
Iteration 9979 Pi value : 3.1650465918554965
Iteration 9980 Pi value : 3.165130263521042
Iteration 9981 Pi value : 3.1649131449754555
Iteration 9982 Pi value : 3.164896914265678
Iteration 9983 Pi value : 3.164880466793549
Iteration 9984 Pi value : 3.1650641025641026
Iteration 9985 Pi value : 3.1651477215623736
Iteration 9986 Pi value : 3.1652319239558947
Iteration 9987 Pi value : 3.16531490382197
Iteration 9988 Pi value : 3.1649979975971165
Iteration 9989 Pi value : 3.1650815987487234
Iteration 9990 Pi value : 3.165165165165165
Iteration 9991 Pi value : 3.1652487238514664
Iteration 9992 Pi value : 3.1649319455564453
Iteration 9993 Pi value : 3.1650155108576
Iteration 9994 Pi value : 3.165099094356614
Iteration 9995 Pi value : 3.1651825912959477
Iteration 9996 Pi value : 3.165266106442577
Iteration 9997 Pi value : 3.1648494848454536
Iteration 9998 Pi value : 3.165030060132
Iteration 9999 Pi value : 3.1651165116511653
Iteration 10000 Pi value : 3.1652

Sending : 21323299219934919840
Receiving : [3681257070, 1821635044]
Sending : 169700784895366836
Receiving : [623172039, 253055936]
Sending : 667742236118658745
Receiving : [2636957788, 68390767]
Sending : 7421307689743061233
Receiving : [3716239779, 6959561]
Sending : 1381523612329561592
Receiving : [1144749560, 3729811384]
Sending : 15221944515336189056
Receiving : [3989048153, 1671375826]
Sending : 187000231689899365
Receiving : [1937802349, 2776071727]
Sending : 11461652177238680330
Receiving : [2643215711, 77767628]
Sending : 7591691343265068935
Receiving : [4284612517, 2532298756]
Sending : 24619406489710734825
Receiving : [1337066470, 3069897882]
Sending : 11212073234165145924
Receiving : [19882686, 2477346065]
Sending : 6156517989429479391
Receiving : [393253305, 2214087653]
Sending : 2036723828998571434
Receiving : [242689745, 2907512919]
Sending : 8765646591154775596
Receiving : [2935551878, 1877004909]
Sending : 121406122568939425165

Sending : 3.1652313236533947
Receiving : Y
Sending : 3.16531490382197
Receiving : N
Sending : 3.1649979975971165
Receiving : Y
Sending : 3.1650815987487234
Receiving : N
Sending : 3.165165165165165
Receiving : Y
Sending : 3.1652487238514664
Receiving : N
Sending : 3.1649319455564453
Receiving : Y
Sending : 3.1650155108576
Receiving : Y
Sending : 3.165099094356614
Receiving : Y
Sending : 3.1651825912959477
Receiving : Y
Sending : 3.165266106442577
Receiving : Y
Sending : 3.1648494848454536
Receiving : Y
Sending : 3.165030060132
Receiving : Y
Sending : 3.1651165116511653
Receiving : Y
Sending : 3.1652
```

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

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