Multiprocessing with ZeroMQ

Common design patterns using ZeroMQ and how to use as a drop in replacement for Queue and Pipe… plus options for multicast and external communication.

Multiprocessing Introduction

- multiprocessing is a package that supports spawning processes using an API similar to the threading module.
- The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads.
- The multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.
- It runs on both Unix and Windows.

Multiprocessing API - interchangeable with Threading

```python
from multiprocessing import Process

class MyProcess(Process):
    def __init__(self):
        Process.__init__(self)
    def run(self):
        a, b = 0, 1
        for i in range(100000):
            a, b = b, a + b

if __name__ == "__main__":
    p = MyProcess()
    p.start()
    p.join()
```

```python
from threading import Thread

class MyThread(Thread):
    def __init__(self):
        Thread.__init__(self)
    def run(self):
        a, b = 0, 1
        for i in range(100000):
            a, b = b, a + b

if __name__ == "__main__":
    p = MyThread()
    p.start()
    p.join()
```

Multiprocessing API - interchangeable with Threading

```python
import sys
if len(sys.argv) > 1 and sys.argv[1] == "thread":
    from threading import Thread as Concurrent
else:
    from multiprocessing import Process as Concurrent

class MyConcurrent(Concurrent):
    def __init__(self):
        Concurrent.__init__(self)
    def run(self):
        a, b = 0, 1
        for i in range(100000):
            a, b = b, a + b

if __name__ == "__main__":
    p = MyConcurrent()
    p.start()
    p.join()
```

Multiprocessing Basics

```
import multiprocessing

def fun(name):
    print 'Hello', name

if __name__ == '__main__':
    p = multiprocessing.Process(target=fun, args=('Sean',))
    p.start()
    p.join()
```

Multiprocessing Basics - Daemon

```
import multiprocessing
import time

def fun(name):
    time.sleep(1)
    print 'Hello', name

if __name__ == '__main__':
    p = multiprocessing.Process(target=fun, args=('Sean',))
    p.daemon = True
    p.start()
    p.join()
```

Multiprocessing Basics - Daemon

```
import multiprocessing
import time

def fun(name):
    time.sleep(1)
    print 'Hello', name

if __name__ == '__main__':
    p = multiprocessing.Process(target=fun, args=('Sean',))
    p.daemon = True
    p.start()
    p.join()
```

Multiprocessing - Work distribution

- Typical work distribution done with a multiprocessing.Queue
  - Is a distributed version of Queue.Queue
  - Put work in, get work out
  - Use multiple Queues for bi-directional communication
- Using a multiprocessing.Manager, Queues can be used across the Network

Multiprocessing - Work distribution

```
import sys
import time
from  multiprocessing import
Process, Queue

def worker(q):
    for task_nbr in range(1000000):
        message = q.get()
    sys.exit(1)

def main():
    send_q = Queue()
    Process(target=worker,
args=(send_q,)).start()
    for num in range(1000000):
        send_q.put("MESSAGE")
```

```
if __name__ == "__main__":
    start_time = time.time()
    main()
    end_time = time.time()
    duration = end_time - start_time
    msg_per_sec = 1000000 / duration

    print "Duration: %s" % duration
    print "Messages Per Second: %s"
% msg_per_sec
```

Where does ZMQ fit in here?

- Glad you asked…
- First, let’s do a primer on ZMQ.

ZMQ Intro

- Intelligent socket library for messaging
- Many type of connection patterns
- Multi-platform, multi-language (40+)
- Very fast (10M msg/sec, 30µsec latency)
- Small (relative lib) <20K lines of C++
- Open source

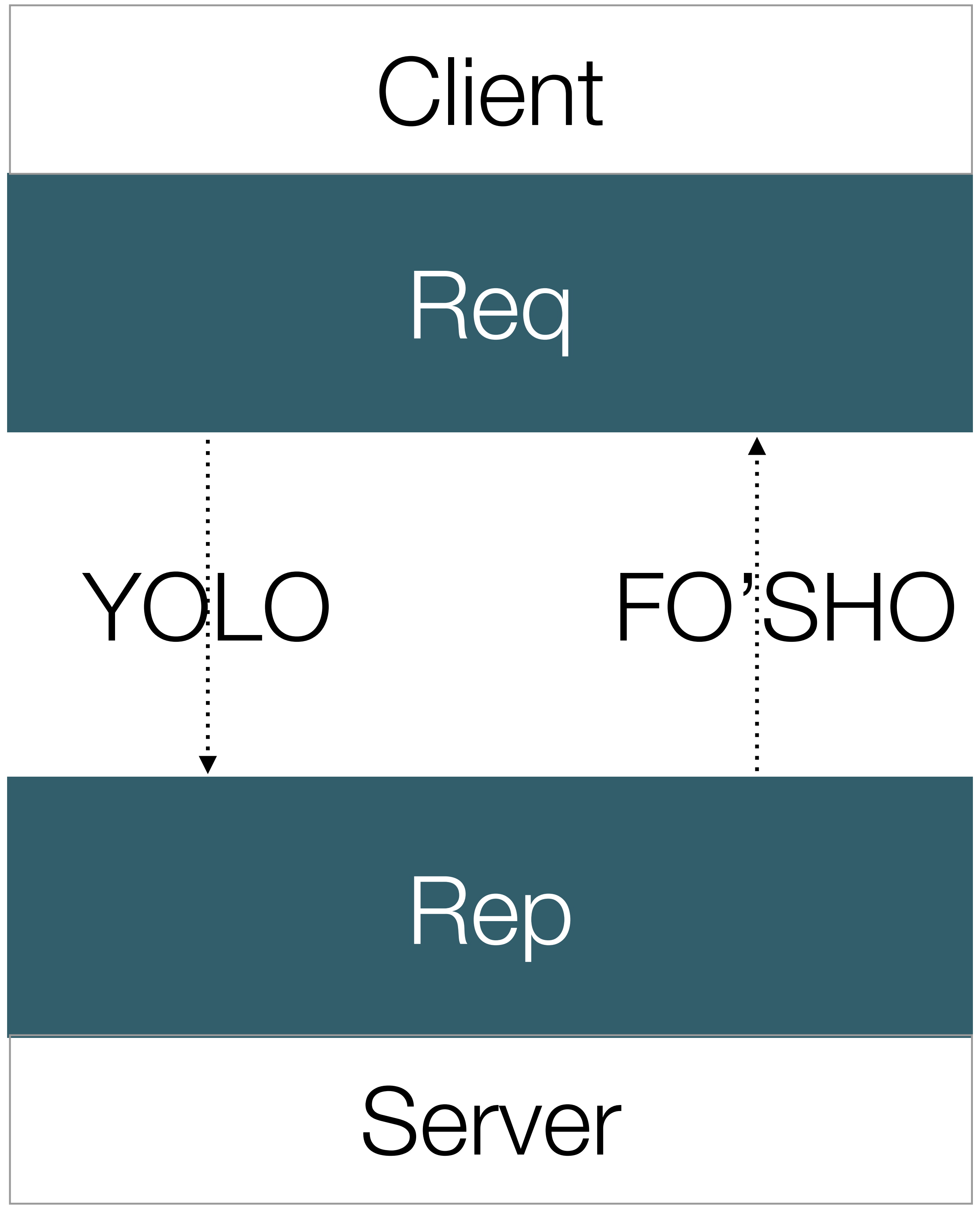
ZMQ Intro - Type of sockets*

- inproc:// - Threads in one process
- ipc:// - Multiple processes on one box
- tcp:// - Processes on a network
- pgm:// - Multicast group (rarely used)

ZMQ Intro - Features

- Queuing at both client and server (* this is HUGE)
- One zmq client socket connects to many zmq server sockets (* this is HUGE)
- Automatic TCP connect / reconnect (* this is nice)
- Zero-copy for large messages

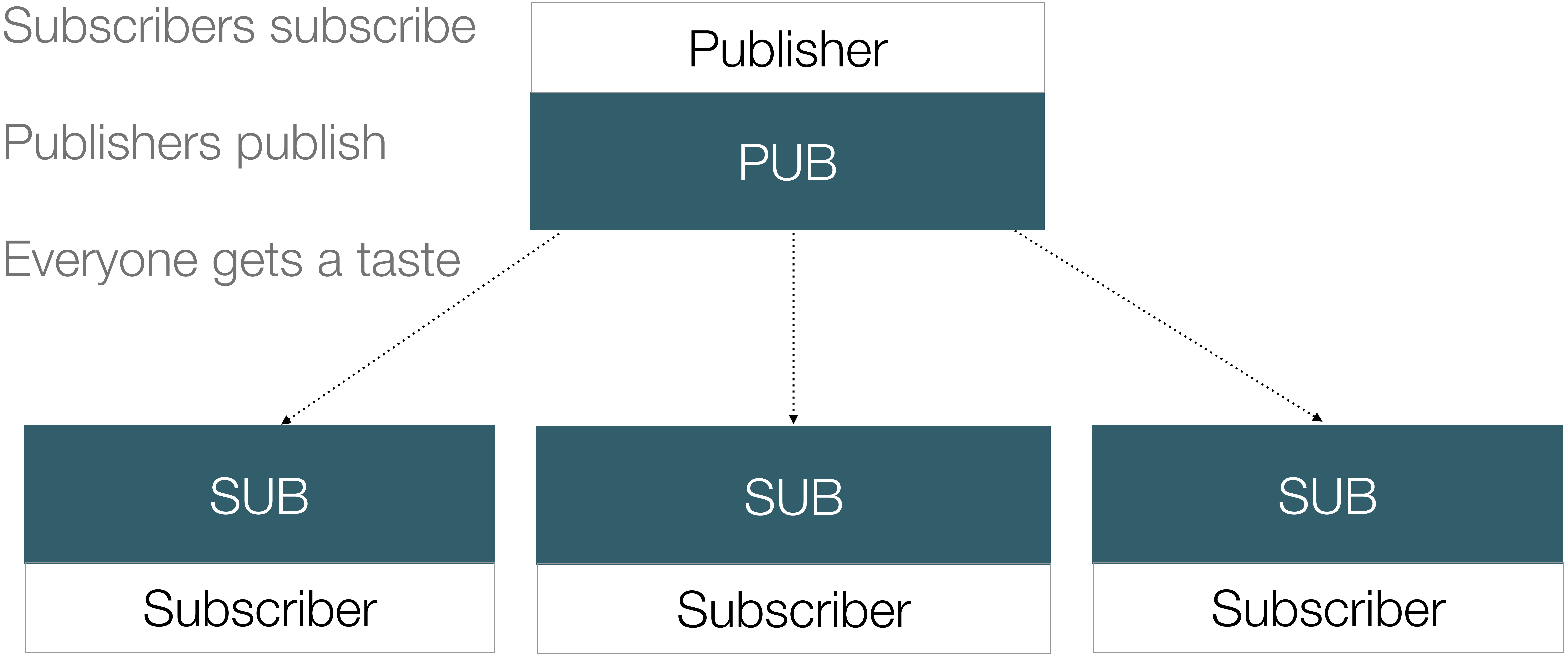
ZMQ Patterns - Request / Reply



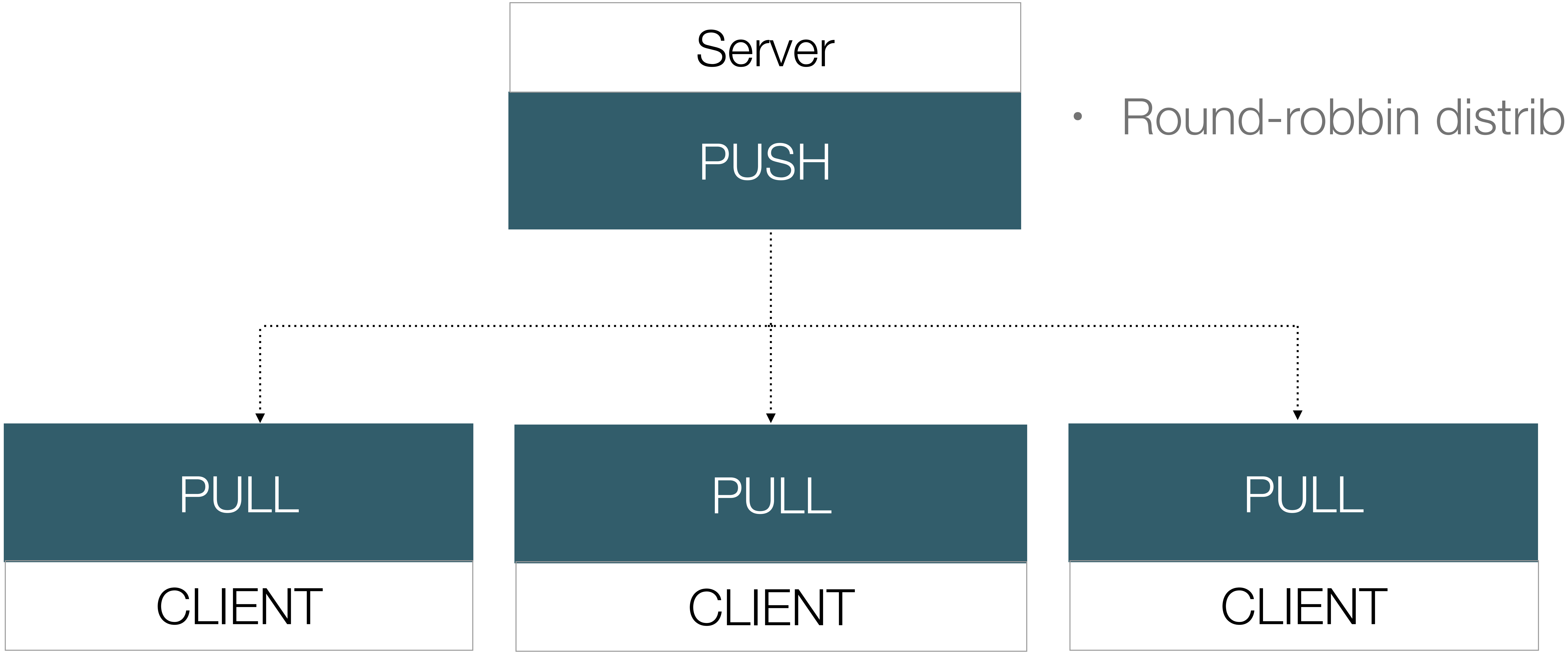
- Client makes a request
- Server sends a response

ZMQ Patterns - Publish / Subscribe

- Subscribers subscribe
- Publishers publish
- Everyone gets a taste



ZMQ Patterns - Push / Pull



- Round-robbin distribution

ZMQ Patterns - Router

- Router is just another socket type, but is used specifically for routing to another destination
- Each hop through a router toward request handler prepends routing information
- Each hop through a router toward a response handler pops the routing info from the head, uses it for routing an forwards the rest of the message

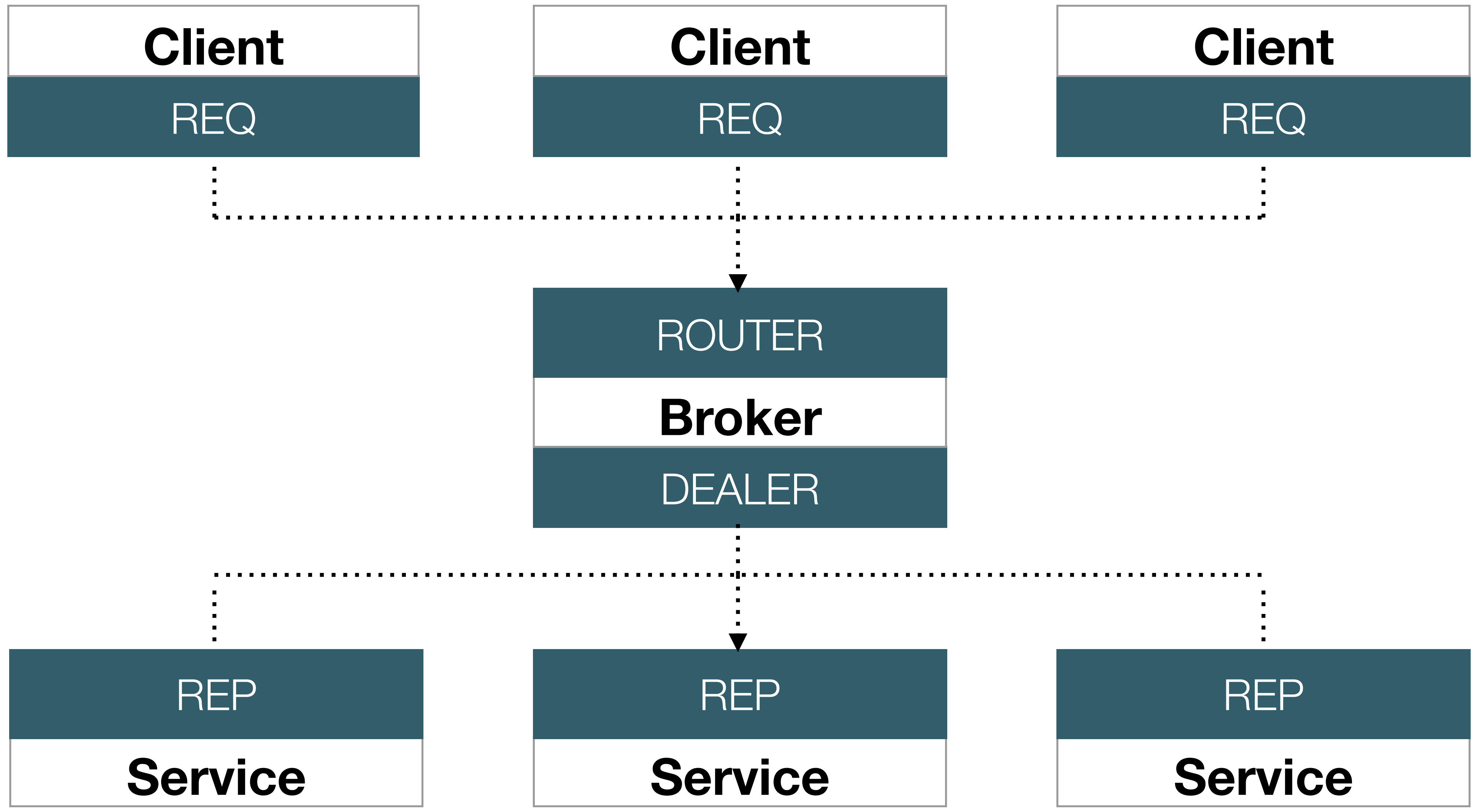
ZMQ Patterns - Dealer

- Dealer is async, bidirectional, round-robin
- Two main use-cases:
  - Work distribution via inproc:// or ipc://
  - Cross network distribution via tcp://
- Each receiver of dealer messages will reply to same dealer

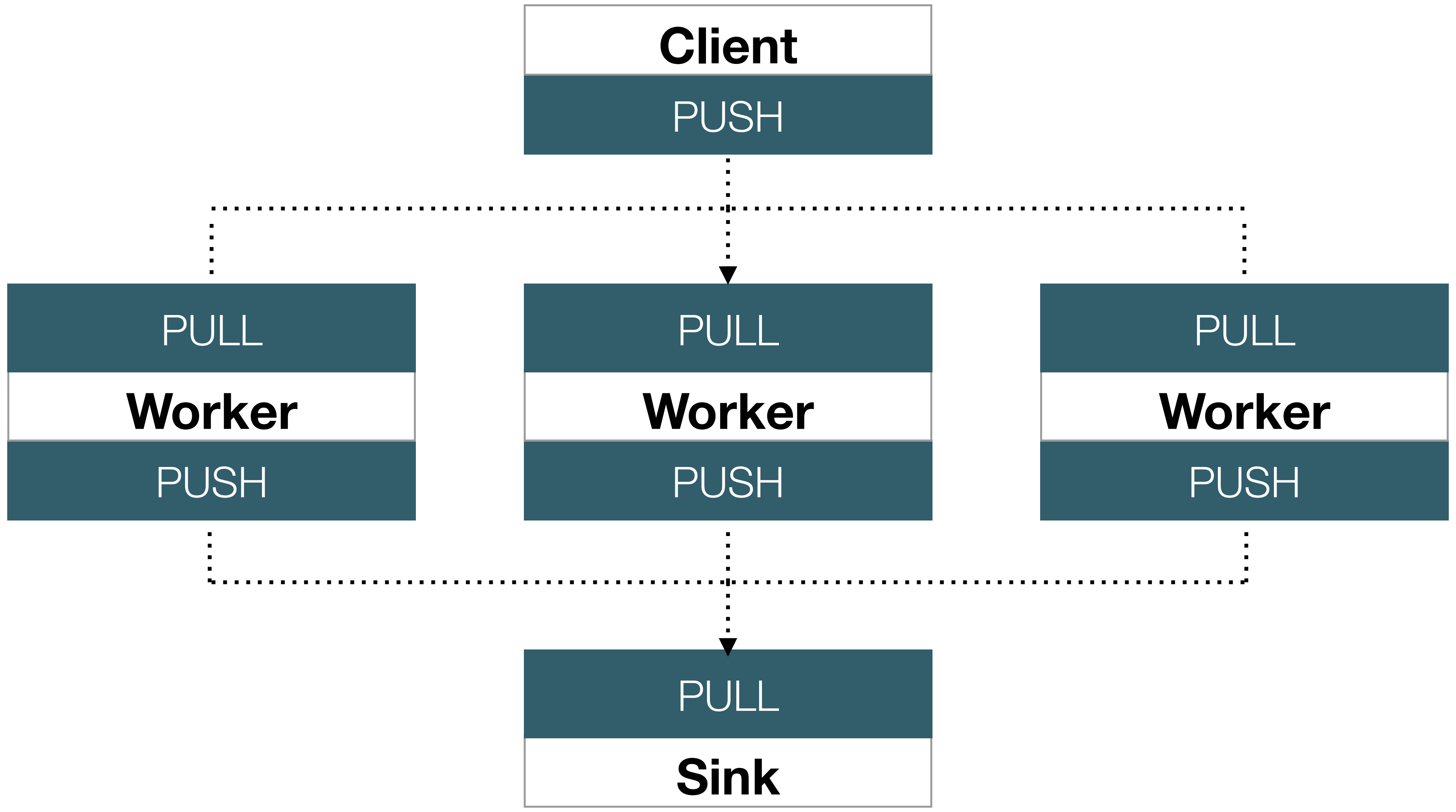
ZMQ Patterns - Valid Patterns

- REQ and REP
- PUB and SUB
- REQ and ROUTER
- DEALER and REP
- DEALER and ROUTER
- DEALER and DEALER
- ROUTER and ROUTER
- PUSH and PULL
- PAIR and PAIR

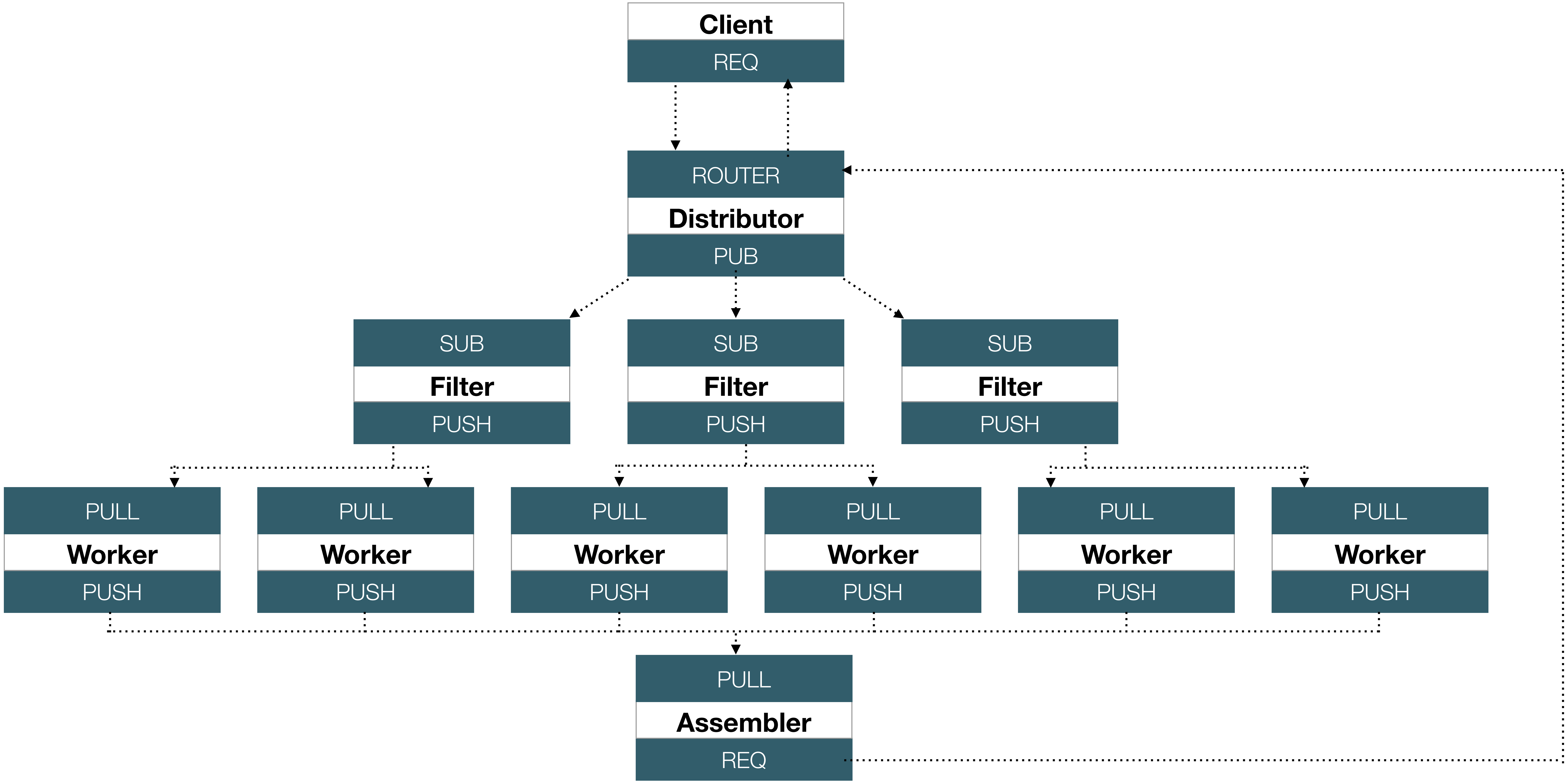
Complex Pipeline - Load Balancer



Complex Pipeline - Work Distributor



Complex Pipeline - Distributed Composition



Why are we even talking about ZMQ?

- multiprocessing is better with ZMQ pairs in place of Queue and Pipe
    - HERASY!
- Build your application using zmq for multiprocessing communication, and distribution across the network becomes simple
    - Your 8 core box can only really do 8 things at once
    - What if you have 100 things to do…
- Oh, and it is faster… MUCH FASTER

Why are we even talking about ZMQ?

Does speed matter?

```
import sys
import time
from  multiprocessing import Process, Queue

def worker(q):
    for task_nbr in range(1000000):
        message = q.get()
    sys.exit(1)

def main():
    send_q = Queue()
    for _ in range(5):
        Process(target=worker,
args=(send_q,)).start()
    for num in range(1000000):
        send_q.put("MESSAGE")
```

```
if __name__ == "__main__":
    start_time = time.time()
    main()
    end_time = time.time()
    duration = end_time - start_time
    msg_per_sec = 1000000 / duration

    print "Duration: %s" % duration
    print "Messages Per Second: %s" %
msg_per_sec
```

Why are we even talking about ZMQ?

Me thinks… yes!

```
import sys
import zmq
from  multiprocessing import Process
import time

def worker():
    context = zmq.Context()
    work_receiver =
context.socket(zmq.PULL)
    work_receiver.connect("ipc:///tmp/
foo.sock")

    for task_nbr in range(0, 1000000):
        message = work_receiver.recv()
    print "EXITED"

    sys.exit(1)
```

```
def main():
    Process(target=worker,
args=()).start()
    context = zmq.Context()
    ventilator_send =
context.socket(zmq.PUSH)
    ventilator_send.bind("ipc:///tmp/
foo.sock")
    for num in range(0, 1000000):
        ventilator_send.send("MESSAGE")

if __name__ == "__main__":
    start_time = time.time()
    main()
    end_time = time.time()
    duration = end_time - start_time
    msg_per_sec = 1000000 / duration
```

Why are we even talking about ZMQ?

Speed difference

```
> python mp_with_queue.py
Duration: 12.5342438221
Messages Per Second: 79781.4382896

> python mp_with_zmq.py
Duration: 0.588519096375
Messages Per Second: 1699180.207
```

21x Faster!!!

Why is it so much faster?

- Queue needs to pickle data
  - Queue allows you to send objects which are serialized / deserialized (via pickle)
  - ZMQ sends bytes
- This means there is a trade-off as you lose some of the niceties of built in Queue by going back to bytes.
- But you can do this yourself (with pickle if you want)
- But in a polyglot system (like the one I work in) you definitely DON’T want pickle anyway.

Example time

Be sure to check out…

- github.com/CrowdStrike/cs.eyrie
    - Library written by internal team for abstraction of event flow handling.
    - Pollers are not needed as Tornado event loop handles recv, send and messages are handled in callbacks.

CROWDSTRIKE

We’re Hiring!

- Great pay, benefits, stock options, and bonus plan
- Top notch developers
- Challenging problems
- Large scale distributed cloud architecture
- We use Python
  - along with Scala, Go and C++
- “Hilarious quotes” wiki page from IRC logs

crowdstrike.com/about-us/careers/