# Learning 0MQ

*Release 1.0*

**Ashish**

**May 31, 2017**

# Contents


**1 pyzmq code and examples** **3**
- 1.1 ØMQ and pyzmq Basics . . . 3
  - 1.1.1 Installation . . . 3
  - 1.1.2 ØMQ Version . . . 3
  - 1.1.3 ØMQ context . . . 3
  - 1.1.4 ØMQ sockets . . . 4
- 1.2 ØMQ Messaging Patterns . . . 4
  - 1.2.1 PAIR . . . 4
  - 1.2.2 Client / Server . . . 6
  - 1.2.3 Publish/Subscribe . . . 8
  - 1.2.4 Push/Pull . . . 10
- 1.3 ØMQ Devices . . . 13
  - 1.3.1 Queue . . . 13
  - 1.3.2 Forwarder . . . 16
  - 1.3.3 Streamer . . . 18
- 1.4 Multiprocess & pyzmq . . . 20
- 1.5 Polling and Sockets . . . 21
  - 1.5.1 ZMQ Poller . . . 22
  - 1.5.2 PyZmq Tornado Event Loop . . . 24
- 1.6 PyZmq Devices . . . 26
  - 1.6.1 Streamer . . . 26
  - 1.6.2 Queue . . . 28
  - 1.6.3 Monitor Queue . . . 30

**2 Various links for understanding ZeroMQ** **33**

**3 Competing or related products** **35**

**4 Code** **37**

**5 Acknowledgements** **39**

**6 Contact** **41**

ØMQ is a neat messaging library that allows you that allows you to build your own messaging infrastructure. ØMQ does not provide out of box messaging system experience like ActiveMQ or RabbitMQ. It is higher level concept than sockets and provides as low level conceptual framework as possible to build messaging systems. It can help build framework that scales where services could be handled by different applications that does one job well.

**Note**

These are my notes based on reading the excellent articles on web, ØMQ guide and pyzmq documentation and trying out examples. This is still a work in progress.

Distributed applications are also easier to change compared to a monolithic applications. It's expensive to try to define the behavior of each of many software applications that work together in a large network. It's much cheaper, and more flexible, to define the interfaces between them: the APIs. Messaging is an API that can be stretched across a network.

pyzmq provides python bindings for ØMQ and allows you to leverage ØMQ in python applications. I have been using pyzmq-static with virtualenv as it neatly provides isolated sandbox for my learning.

This tutorial is my journey into ØMQ programming with python. Hopefully, it will help you too.

CHAPTER 1

# pyzmq code and examples

The code in the following tutorial is contrived and does not represent any real world examples. These are also inspired by the code already provided by ØMQ guide and many other examples on articles posted on the web. I have worked on each of the example as a part of my learning. The structure represents, how I have been learning ØMQ.

## ØMQ and pyzmq Basics

### Installation

It is better to install it using virtualenv/virtualenwrapper:

```
pip install pyzmq-static
```

### ØMQ Version

2.1.7:

```
import zmq
print zmq.pyzmq_version()
```

### ØMQ context

Before using any ØMQ library functions, the caller must initialize a ØMQ context:

```
import zmq
import time
context = zmq.Context()
```

Contexts are thread safe unlike sockets. An application can create and manage multiple contexts.

### ØMQ sockets

zmq sockets are created from the initialized context:

```
socket = context.socket(zmq.REP)
```

- zmq sockets are of certain types which enable one of the various communication patterns.
- zmq socket type must be passed during socket creation.

## ØMQ Messaging Patterns

In distributed architecture, different parts of system interconnect and communicate with each other. These interconnecting systems viewed graphically represents a network topology.

Messaging patterns are network oriented architectural pattern that describes the flow of communication between interconnecting systems. ØMQ provides pre-optimized sockets which enables you to take advantage of these patterns.

Each pattern in ØMQ defines the constraints on the network topology. What systems can connect to each other and flow of communication between them. These patterns are designed to scale.

We will run through each of the pattern with an example.

### PAIR

It provides sockets that are close in behavior to conventional sockets.

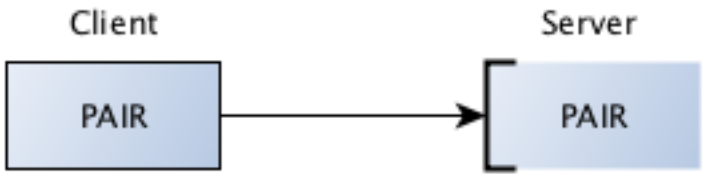
Conventional sockets allow:

- only strict one-to-one (two peers)
- many-to-one (many clients, one server)
- one-to-many (multicast) relationships

**Exclusive pair pattern**

Paired sockets are very similar to regular sockets.

- The communication is bidirectional.
- There is no specific state stored within the socket
- There can only be one connected peer.
- The server listens on a certain port and a client connects to it.




What this really shows is the simplicity of setup and the fact that you receive the complete message that was sent. There is no need to think whether you have read the complete message or not.

**pairserver.py**

```
import zmq
import random
import sys
import time

port = "5556"
context = zmq.Context()
socket = context.socket(zmq.PAIR)
socket.bind("tcp://*:%s" % port)

while True:
    socket.send("Server message to client3")
    msg = socket.recv()
    print msg
    time.sleep(1)
```

**pairclient.py**

```
import zmq
import random
import sys
import time

port = "5556"
context = zmq.Context()
socket = context.socket(zmq.PAIR)
socket.connect("tcp://localhost:%s" % port)

while True:
    msg = socket.recv()
    print msg
    socket.send("client message to server1")
    socket.send("client message to server2")
    time.sleep(1)
```

running it:

```
python pairserver.py <port>
python pairclient.py <port>
```

Each of them can send any number of messages to each other.

## Client / Server

**Request/Reply pattern**

Most basic pattern is client/server model, where client sends a request and server replies to the request.

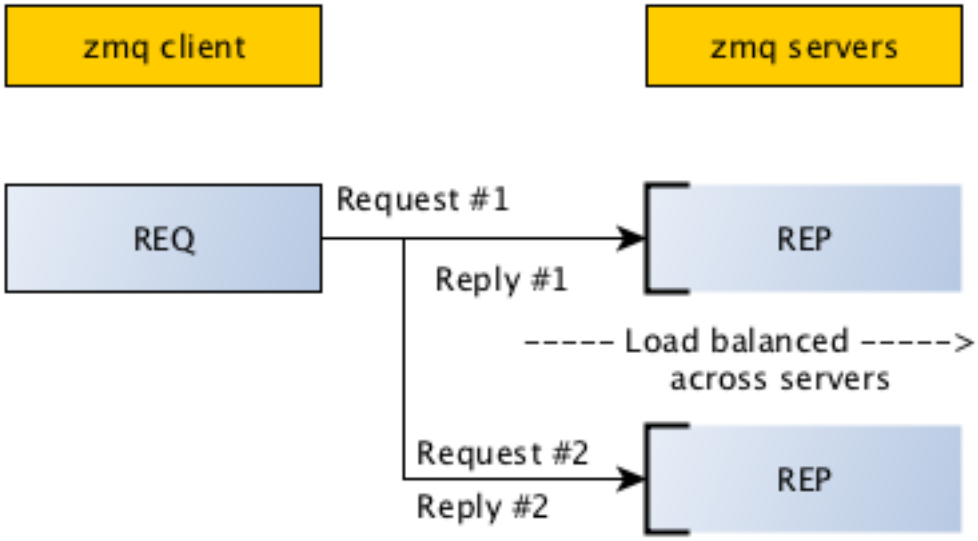
There is one difference from zmq.PAIR and other type of ZMQ sockets.

- ZMQ REQ sockets can connect to many servers.
- The requests will be interleaved or distributed to both the servers.

With socket zmq.PAIR, you could send any number of messages among connected peers or client/server.

- socket zmq.REQ will block on send unless it has successfully received a reply back.
- socket zmq.REP will block on recv unless it has received a request.

Each Request/Reply is paired and has to be successful.




**reqrep_server.py**

Provide port as command line argument to run server at two different ports.

```
import zmq
import time
import sys

port = "5556"
if len(sys.argv) > 1:
    port =  sys.argv[1]
    int(port)
```

**Server** is created with a socket type "**zmq.REP**" and is bound to well known port.

```
context = zmq.Context()
```

```
socket = context.socket(zmq.REP)
socket.bind("tcp://*:%s" % port)
```

It will block on recv() to get a request before it can send a reply.

```
socket.bind("tcp://*:%s" % port)

while True:
    #  Wait for next request from client
    message = socket.recv()
    print "Received request: ", message
    time.sleep (1)
    socket.send("World from %s" % port)
```

**reqrep_client.py**

Provide two ports of two different servers to connect to simultaneously.

```
import zmq
import sys

port = "5556"
if len(sys.argv) > 1:
    port =  sys.argv[1]
    int(port)

if len(sys.argv) > 2:
    port1 =  sys.argv[2]
    int(port1)
```

**Client** is created with a socket type "**zmq.REQ**". You should notice that the same socket can connect to two different servers.

```
context = zmq.Context()
print "Connecting to server..."
socket = context.socket(zmq.REQ)
socket.connect ("tcp://localhost:%s" % port)
if len(sys.argv) > 2:
    socket.connect ("tcp://localhost:%s" % port1)
```

You have to send a request and then wait for reply.

```

#  Do 10 requests, waiting each time for a response
for request in range (1,10):
    print "Sending request ", request,"..."
    socket.send ("Hello")
    #  Get the reply.
    message = socket.recv()
    print "Received reply ", request, "[", message, "]"
```

Executing the scripts:

```
python reqrep_server.py 5546
python reqrep_server.py 5556
python reqrep_client.py 5546 5556
```

Output:

```
Connecting to hello world server...
Sending request  1 ...
Received reply  1 [ World from 5556 ]
Sending request  2 ...
Received reply  2 [ World from 5546 ]
Sending request  3 ...
Received reply  3 [ World from 5556 ]
Sending request  4 ...
Received reply  4 [ World from 5546 ]
Sending request  5 ...
Received reply  5 [ World from 5556 ]
Sending request  6 ...
```

Any attempt to send another message to the socket (zmq.REQ/zmq.REP), without having received a reply/request will result in an error:

```
....
socket.send ("Hello")
socket.send ("Hello1")
....

Error: zmq.core.error.ZMQError: Operation cannot be accomplished in current state
```

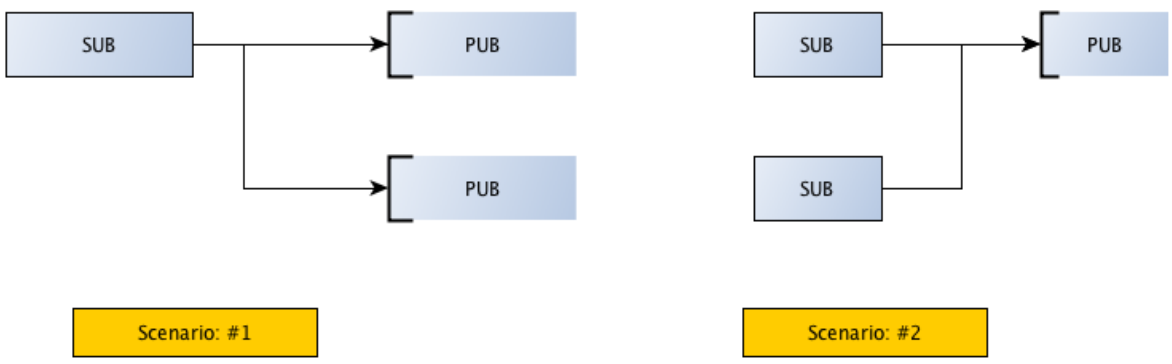
**Note:** If you kill the server (Ctrl-C) and restart it, the client won't recover properly.

## Publish/Subscribe

**Pub/Sub pattern**

Publish/Subscribe is another classic pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Messages are published without the knowledge of what or if any subscriber of that knowledge exists.

Scenario #2 is more known, general pattern where multiple subscribers subscribes to messages/topics being published by a publisher. It is scenario #1 which is more interesting. Just like ZMQ.REQ which can connect to multiple ZMQ.REP, ZMQ.SUB can connect to multiple ZMQ.PUB (publishers). No single publisher overwhelms the subscriber. The messages from both publishers are interleaved.



**pub_server.py**

Publishers are created with ZMQ.PUB socket types

```
import zmq
import random
import sys
import time

port = "5556"
if len(sys.argv) > 1:
    port =  sys.argv[1]
    int(port)

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:%s" % port)
```

Data is published along with a topic. The subscribers usually sets a filter on these topics for topic of their interests.

```
while True:
    topic = random.randrange(9999,10005)
    messagedata = random.randrange(1,215) - 80
    print "%d %d" % (topic, messagedata)
    socket.send("%d %d" % (topic, messagedata))
    time.sleep(1)
```

**sub_client.py**

Subscribers are created with ZMQ.SUB socket types. You should notice that a zmq subscriber can connect to many publishers.

```
import sys
import zmq

port = "5556"
if len(sys.argv) > 1:
    port =  sys.argv[1]
    int(port)

if len(sys.argv) > 2:
    port1 =  sys.argv[2]
    int(port1)

# Socket to talk to server
context = zmq.Context()
socket = context.socket(zmq.SUB)

print "Collecting updates from weather server..."
socket.connect ("tcp://localhost:%s" % port)

if len(sys.argv) > 2:
    socket.connect ("tcp://localhost:%s" % port1)
```

The current version of zmq supports filtering of messages based on topics at subscriber side. This is usually set via socketoption.

```
# Subscribe to zipcode, default is NYC, 10001
topicfilter = "10001"
socket.setsockopt(zmq.SUBSCRIBE, topicfilter)

# Process 5 updates
total_value = 0
for update_nbr in range (5):
    string = socket.recv()
    topic, messagedata = string.split()
    total_value += int(messagedata)
    print topic, messagedata

print "Average messagedata value for topic '%s' was %dF" % (topicfilter, total_value /
↪ update_nbr)
```

Pub/Sub communication is asynchronous. If a "publish" service has been started already and then when you start "subscribe" service, it would not receive a number of message that was published already by the pub services. Starting "publisher" and "subscriber" is independent of each other.

A subscriber can in fact connect to more than one publisher, using one 'connect' call each time. Data will then arrive and be interleaved so that no single publisher drowns out the others.:

```
python pub_server.py 5556
python pub_server.py 5546
python sub_client.py 5556 5546
```

Other things to note:

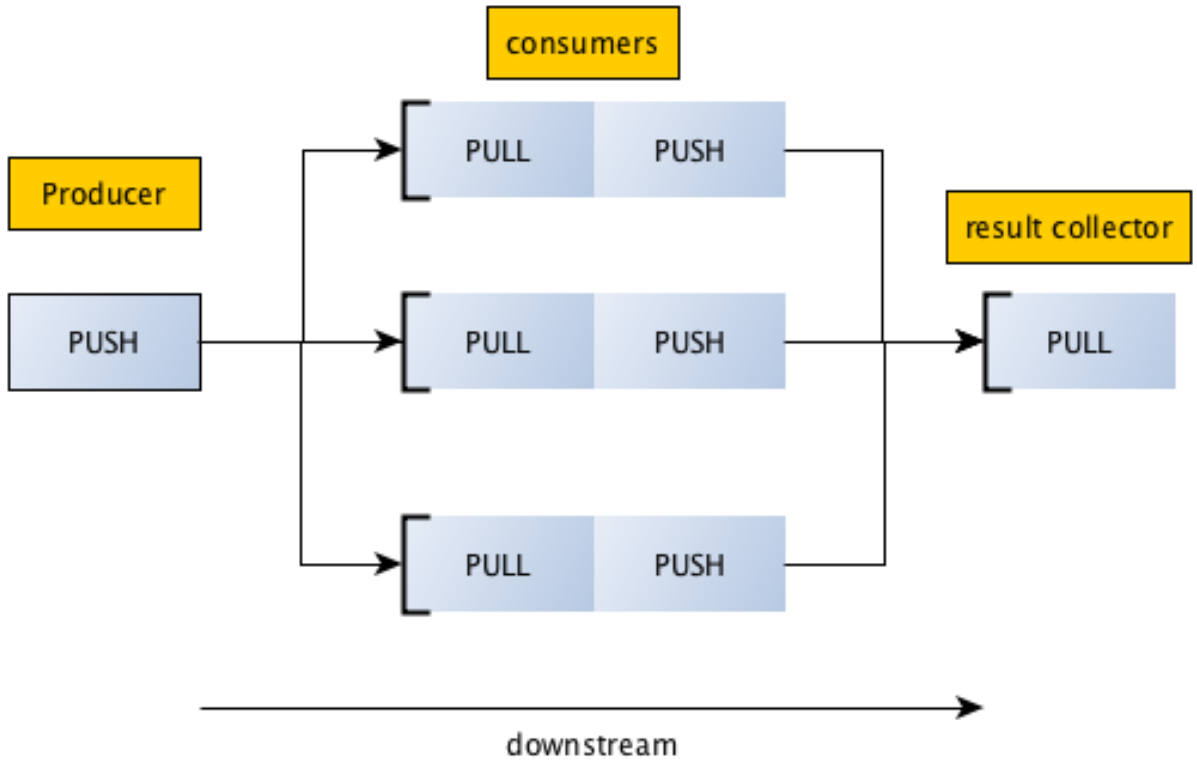
- A publisher has no connected subscribers, then it will simply drop all messages.
- If you're using TCP, and a subscriber is slow, messages will queue up on the publisher.
- In the current versions of ØMQ, filtering happens at the subscriber side, not the publisher side.

## Push/Pull

**Pipeline pattern**

Push and Pull sockets let you distribute messages to multiple workers, arranged in a pipeline. A Push socket will distribute sent messages to its Pull clients evenly. This is equivalent to producer/consumer model but the results computed by consumer are not sent upstream but downstream to another pull/consumer socket.

Data always flows down the pipeline, and each stage of the pipeline is connected to at least one node. When a pipeline stage is connected to multiple nodes data is load-balanced among all connected nodes.




**producer.py**

Producers are created with ZMQ.PUSH socket types. Producer is bound to well known port to which consumers can connect too.

```
import time
import zmq

def producer():
    context = zmq.Context()
    zmq_socket = context.socket(zmq.PUSH)
    zmq_socket.bind("tcp://127.0.0.1:5557")
    # Start your result manager and workers before you start your producers
    for num in xrange(20000):
        work_message = { 'num' : num }
        zmq_socket.send_json(work_message)

producer()
```

**consumer.py**

Producers are created with ZMQ.PULL socket types to pull requests from producer and uses a push socket to connect and push result to result collector.

```
import time
import zmq
import random

def consumer():
    consumer_id = random.randrange(1,10005)
```

```
    print "I am consumer #%s" % (consumer_id)
    context = zmq.Context()
    # recieve work
    consumer_receiver = context.socket(zmq.PULL)
    consumer_receiver.connect("tcp://127.0.0.1:5557")
    # send work
    consumer_sender = context.socket(zmq.PUSH)
    consumer_sender.connect("tcp://127.0.0.1:5558")

    while True:
        work = consumer_receiver.recv_json()
        data = work['num']
        result = { 'consumer' : consumer_id, 'num' : data}
        if data%2 == 0:
            consumer_sender.send_json(result)

consumer()
```

**resultcollector.py**

result collector are created with ZMQ.PULL socket type and act as consumer of results from intermediate consumers. They also are bound to well known port so that intermediate consumer can connect to it.

```
import time
import zmq
import pprint

def result_collector():
    context = zmq.Context()
    results_receiver = context.socket(zmq.PULL)
    results_receiver.bind("tcp://127.0.0.1:5558")
    collecter_data = {}
    for x in xrange(1000):
        result = results_receiver.recv_json()
        if collecter_data.has_key(result['consumer']):
            collecter_data[result['consumer']] = collecter_data[result['consumer']] +␣
↪1
        else:
            collecter_data[result['consumer']] = 1
        if x == 999:
            pprint.pprint(collecter_data)

result_collector()
```

We have to execute the programs on separate shells as all programs have a while loop that we will discard later:

```
python resultcollector.py
python consumer.py
python consumer.py
python producer.py
```

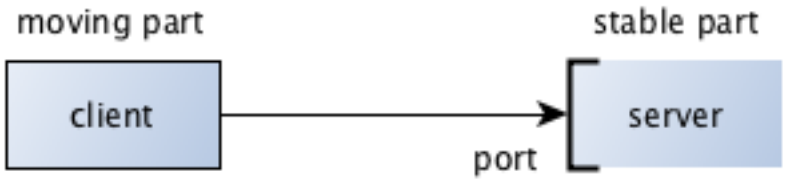
Results shows the distribution of transmitted result to result collector:

```
{   3362: 233,
    9312: 767
}
```

## ØMQ Devices

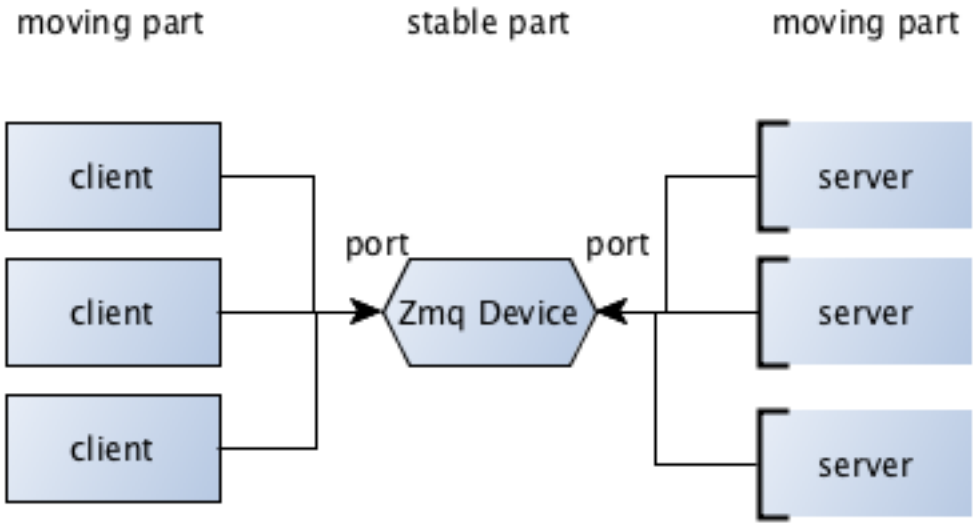
You must have noticed that you can bind a port to any of the ZMQ Socket types. In example for Push/Pull pattern, we bound ZMQ.PUSH (producer) and ZMQ.PULL (result collector) to well known ports.

In theory, most stable part of the network (server) will BIND on a specific port and have the more dynamic parts (client) CONNECT to that.




Some time both ends can be dynamic and it is not a good idea to provide well known ports to either of the ends.

In such cases, you could connect them using zeromq's forwarding device. These devices can bind to 2 different ports and forward messages from one end to the other. The forwarding device can become the stable point in your network where each component can connect to.

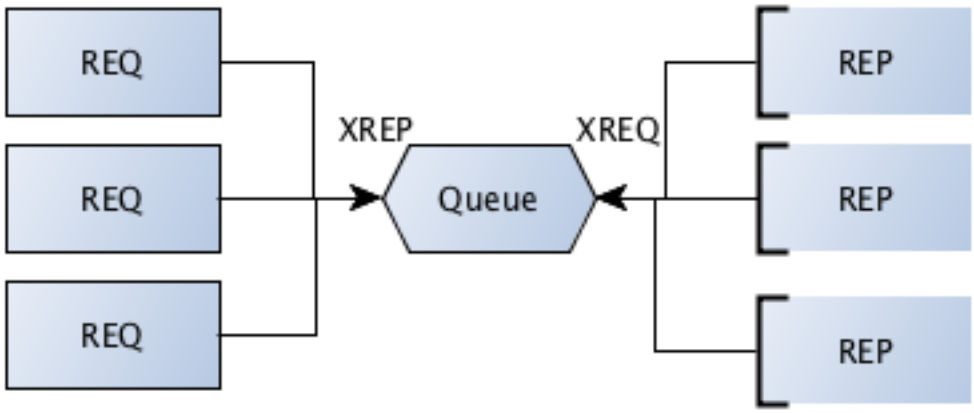



ZMQ provides certain basic devices to build complex topology with basic patterns:

### Queue

**Queue device**

This is the intermediary that sits between clients and servers, forwarding request to servers and relaying replies back to client. The ZMQ device takes a device type (ZMQ.QUEUE) and the two sockets bound to well known ports.




**queue_device.py**

```
import zmq

def main():

    try:
        context = zmq.Context(1)
        # Socket facing clients
        frontend = context.socket(zmq.XREP)
        frontend.bind("tcp://*:5559")
        # Socket facing services
        backend = context.socket(zmq.XREQ)
        backend.bind("tcp://*:5560")

        zmq.device(zmq.QUEUE, frontend, backend)
    except Exception, e:
        print e
        print "bringing down zmq device"
    finally:
        pass
        frontend.close()
        backend.close()
        context.term()

if __name__ == "__main__":
    main()
```

**Note:** ZMQ devices are full programs, devices include a while(True) loop and thus block execution permanently once invoked.

Here, you can see that client has not changed at all from our previous example by introduction of an intermediary ZMQ device.

**queue_client.py**

```
import zmq
import sys
import random

port = "5559"
context = zmq.Context()
print "Connecting to server..."
socket = context.socket(zmq.REQ)
socket.connect ("tcp://localhost:%s" % port)
client_id = random.randrange(1,10005)
#  Do 10 requests, waiting each time for a response
for request in range (1,10):
    print "Sending request ", request,"..."
    socket.send ("Hello from %s" % client_id)
    #  Get the reply.
    message = socket.recv()
    print "Received reply ", request, "[", message, "]"
```

Here, the only change to the server is that it is not bound to a well known port. Instead it connects to a well known port of the intermediary.

**queue_server.py**

```
import zmq
import time
import sys
import random

port = "5560"
context = zmq.Context()
socket = context.socket(zmq.REP)
socket.connect("tcp://localhost:%s" % port)
server_id = random.randrange(1,10005)
while True:
    #  Wait for next request from client
    message = socket.recv()
    print "Received request: ", message
    time.sleep (1)
    socket.send("World from server %s" % server_id)
```

Execute the following on different shells:

```
python  queue_device.py
python queue_server.py
python queue_server.py
python queue_client.py
python queue_client.py
```

If you run a single client, you can see that requests are load balanced among available server:

```
Connecting to server...
Sending request  1 ...
Received reply  1 [ World from server 7003 ]
Sending request  2 ...
Received reply  2 [ World from server 4411 ]
Sending request  3 ...
Received reply  3 [ World from server 7003 ]
Sending request  4 ...
```

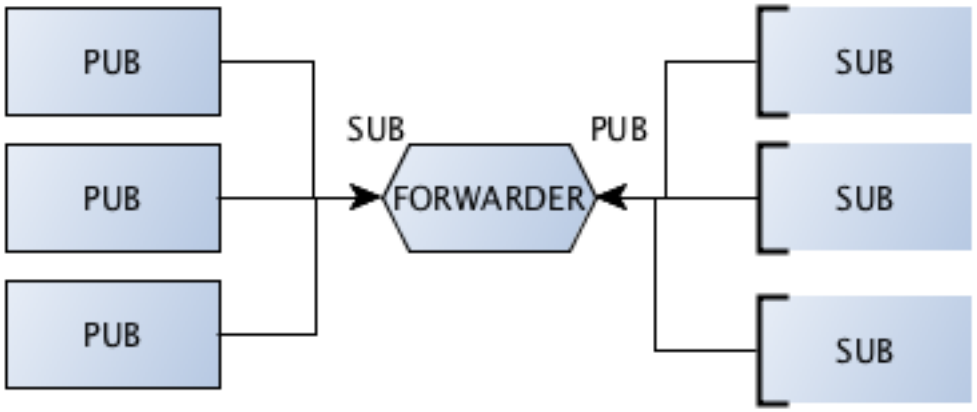
```
Received reply  4 [ World from server 4411 ]
Sending request  5 ...
Received reply  5 [ World from server 7003 ]
Sending request  6 ...
Received reply  6 [ World from server 4411 ]
Sending request  7 ...
Received reply  7 [ World from server 7003 ]
Sending request  8 ...
Received reply  8 [ World from server 4411 ]
Sending request  9 ...
Received reply  9 [ World from server 7003 ]
```

## Forwarder

**Forwarder device**

Just like QUEUE, which is like the request-reply broker, FORWARDER is like the pub-sub proxy server. It allows both publishers and subscribers to be moving parts and it self becomes the stable hub for interconnecting them.

FORWARDER collects messages from a set of publishers and forwards these to a set of subscribers.




You will notice that two zmq sockets, pub and sub are bound to well known ports. The frontend speaks to publishers and the backend speaks to subscribers. You should use ZMQ_FORWARDER with a ZMQ_SUB socket for the frontend and a ZMQ_PUB socket for the backend.

Another important thing to notice is that we want all the published messages to reach to the various subscribers, hence message filtering should be off in the forwarder device. See line no 11.

**forwarder_device.py**

```
import zmq

def main():

    try:
        context = zmq.Context(1)
        # Socket facing clients
```

```
        frontend = context.socket(zmq.SUB)
        frontend.bind("tcp://*:5559")

        frontend.setsockopt(zmq.SUBSCRIBE, "")

        # Socket facing services
        backend = context.socket(zmq.PUB)
        backend.bind("tcp://*:5560")

        zmq.device(zmq.FORWARDER, frontend, backend)
    except Exception, e:
        print e
        print "bringing down zmq device"
    finally:
        pass
        frontend.close()
        backend.close()
        context.term()

if __name__ == "__main__":
    main()
```

Only thing that changes here is that publisher connects to the intermediary and is not bound to any well known port.

**forwarder_server.py**

```
import zmq
import random
import sys
import time

port = "5559"
context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.connect("tcp://localhost:%s" % port)
publisher_id = random.randrange(0,9999)
while True:
    topic = random.randrange(1,10)
    messagedata = "server#%s" % publisher_id
    print "%s %s" % (topic, messagedata)
    socket.send("%d %s" % (topic, messagedata))
    time.sleep(1)
```

The subscribers are completely unaffected by introduction of intermediary - "forwarder device" and gains the ability to get messages from different publishers at no cost.

**forwarder_subscriber.py**

```
import sys
import zmq

port = "5560"
# Socket to talk to server
context = zmq.Context()
socket = context.socket(zmq.SUB)
print "Collecting updates from server..."
socket.connect ("tcp://localhost:%s" % port)
topicfilter = "9"
socket.setsockopt(zmq.SUBSCRIBE, topicfilter)
```

```
for update_nbr in range(10):
    string = socket.recv()
    topic, messagedata = string.split()
    print topic, messagedata
```

Executing these programs from separate shell:

```
python forwarder_device.py
python forwarder_subscriber.py
python forwarder_server.py
python forwarder_server.py
```

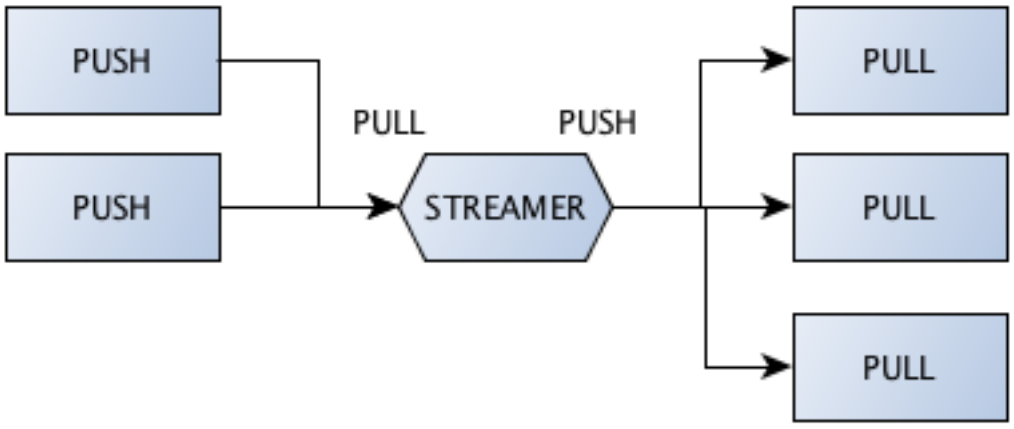
Output on the subscriber:

```
Collecting updates from server...
9 server#3581
9 server#9578
9 server#3581
9 server#9578
9 server#3581
9 server#9578
9 server#3581
9 server#3581
9 server#9578
9 server#3581
```

## Streamer

**Streamer device**

is a device for parallelized pipeline messaging. Acts as a broker that callects tasks from task feeders and supplies them to task workers.




**streamer_device.py**

```
import zmq

def main():

    try:
        context = zmq.Context(1)
        # Socket facing clients
        frontend = context.socket(zmq.PULL)
        frontend.bind("tcp://*:5559")

        # Socket facing services
        backend = context.socket(zmq.PUSH)
        backend.bind("tcp://*:5560")

        zmq.device(zmq.STREAMER, frontend, backend)
    except Exception, e:
        print e
        print "bringing down zmq device"
    finally:
        pass
        frontend.close()
        backend.close()
        context.term()

if __name__ == "__main__":
    main()
```

**task_feeder.py**

```
import time
import zmq

def producer():
    context = zmq.Context()
    zmq_socket = context.socket(zmq.PUSH)
    zmq_socket.connect("tcp://127.0.0.1:5559")
    # Start your result manager and workers before you start your producers
    for num in xrange(20000):
        work_message = { 'num' : num }
        zmq_socket.send_json(work_message)
        time.sleep(1)

producer()
```

**task_worker.py**

```
import sys
import time
import zmq
import random

def consumer():
    consumer_id = random.randrange(1,10005)
    print "I am consumer #%s" % (consumer_id)
    context = zmq.Context()
    # recieve work
```

```
    consumer_receiver = context.socket(zmq.PULL)
    consumer_receiver.connect("tcp://127.0.0.1:5560")
    while True:
        work = consumer_receiver.recv_json()
        data = work['num']
        result = { 'consumer' : consumer_id, 'num' : data}
        print result
consumer()
```

Execute these programs on separate shells:

```
python streamer_device.py
python task_feeder.py
python task_worker.py
python task_worker.py
```

Output on one of the workers:

```
I am consumer #8113
{'num': 1, 'consumer': 8113}
{'num': 3, 'consumer': 8113}
{'num': 5, 'consumer': 8113}
{'num': 7, 'consumer': 8113}
{'num': 9, 'consumer': 8113}
{'num': 11, 'consumer': 8113}
```

## Multiprocess & pyzmq

You will have noticed in the previous examples for the REQUEST-REPLY pattern that we executed the server and client separately. This was due to the fact that each program had a while loop that would run for ever. Only way is to invoke these little programs separately.

This served well to understand the various ØMQ patterns and devices. How ever, it is far better to use multiprocessing module.

This part of tutorial has nothing to do with ØMQ but how we use it with python programs.

**request_reply_processes.py**

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

def server(port="5556"):
    context = zmq.Context()
    socket = context.socket(zmq.REP)
    socket.bind("tcp://*:%s" % port)
    print "Running server on port: ", port
    # serves only 5 request and dies
    for reqnum in range(5):
        # Wait for next request from client
        message = socket.recv()
        print "Received request #%s: %s" % (reqnum, message)
        socket.send("World from %s" % port)
```

```
def client(ports=["5556"]):

    context = zmq.Context()
    print "Connecting to server with ports %s" % ports
    socket = context.socket(zmq.REQ)
    for port in ports:
        socket.connect ("tcp://localhost:%s" % port)
    for request in range (20):
        print "Sending request ", request,"..."
        socket.send ("Hello")
        message = socket.recv()
        print "Received reply ", request, "[", message, "]"
        time.sleep (1)


if __name__ == "__main__":
    # Now we can run a few servers
    server_ports = range(5550,5558,2)
    for server_port in server_ports:
        Process(target=server, args=(server_port,)).start()

    # Now we can connect a client to all these servers
    Process(target=client, args=(server_ports,)).start()
```

Now it easy to run the server and clients in one go.

The output shows how the requests are load balanced across available servers:

```
Running server on port:  5550
Running server on port:  5552
Running server on port:  5554
Running server on port:  5556
Connecting to server with ports [5550, 5552, 5554, 5556]
Sending request  0 ...
Received request #0: Hello
Received reply  0 [ World from 5550 ]
Sending request  1 ...
Received request #0: Hello
Received reply  1 [ World from 5552 ]
Sending request  2 ...
Received request #0: Hello
Received reply  2 [ World from 5554 ]
Sending request  3 ...
Received request #0: Hello
Received reply  3 [ World from 5556 ]
Sending request  4 ...
Received request #1: Hello
Received reply  4 [ World from 5550 ]
```

## Polling and Sockets

Using multiprocessing module helped us to launch the server, clients as processes from the same program. How ever, you would have noticed that this still suffered from one limitation. These processes would serve only one socket connection. How ever, in real world a process might be connected to multiple sockets and work on data received on both.

In such situation, it is better to poll for data on the sockets. ZMQ provides facility for polling sockets as you can not block on recv().

## ZMQ Poller

In this program, we will create a command server that tells when the worker should exit. Workers subscribes to a topic published by a publisher and prints it. It exits when it receives "Exit" message from the command server.

**zmqpolling.py**

PUSH server that sends command to workers to continue working or exit.

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

def server_push(port="5556"):
    context = zmq.Context()
    socket = context.socket(zmq.PUSH)
    socket.bind("tcp://*:%s" % port)
    print "Running server on port: ", port
    # serves only 5 request and dies
    for reqnum in range(10):
        if reqnum < 6:
            socket.send("Continue")
        else:
            socket.send("Exit")
            break
        time.sleep (1)
```

Publisher that publishes for topics "8","9","10" in random order.

```
def server_pub(port="5558"):
    context = zmq.Context()
    socket = context.socket(zmq.PUB)
    socket.bind("tcp://*:%s" % port)
    publisher_id = random.randrange(0,9999)
    print "Running server on port: ", port
    # serves only 5 request and dies
    for reqnum in range(10):
        # Wait for next request from client
        topic = random.randrange(8,10)
        messagedata = "server#%s" % publisher_id
        print "%s %s" % (topic, messagedata)
        socket.send("%d %s" % (topic, messagedata))
        time.sleep(1)
```

Worker that works on messages received for topic "9". We setup zmq poller to poll for messages on the socket connection to both command server and publisher.

```
def client(port_push, port_sub):
    context = zmq.Context()
    socket_pull = context.socket(zmq.PULL)
```

```
    socket_pull.connect ("tcp://localhost:%s" % port_push)
    print "Connected to server with port %s" % port_push
    socket_sub = context.socket(zmq.SUB)
    socket_sub.connect ("tcp://localhost:%s" % port_sub)
    socket_sub.setsockopt(zmq.SUBSCRIBE, "9")
    print "Connected to publisher with port %s" % port_sub
    # Initialize poll set
    poller = zmq.Poller()
    poller.register(socket_pull, zmq.POLLIN)
    poller.register(socket_sub, zmq.POLLIN)
```

We poll the sockets to check if we have messages to recv and work on it. Worker continues working until it receives exit condition.

```
    # Work on requests from both server and publisher
    should_continue = True
    while should_continue:
        socks = dict(poller.poll())
        if socket_pull in socks and socks[socket_pull] == zmq.POLLIN:
            message = socket_pull.recv()
            print "Recieved control command: %s" % message
            if message == "Exit":
                print "Recieved exit command, client will stop recieving messages"
                should_continue = False

        if socket_sub in socks and socks[socket_sub] == zmq.POLLIN:
            string = socket_sub.recv()
            topic, messagedata = string.split()
            print "Processing ... ", topic, messagedata
```

Finally, we fire up all the processes.

```


if __name__ == "__main__":
    # Now we can run a few servers
    server_push_port = "5556"
    server_pub_port = "5558"
    Process(target=server_push, args=(server_push_port,)).start()
    Process(target=server_pub, args=(server_pub_port,)).start()
    Process(target=client, args=(server_push_port,server_pub_port,)).start()
```

Output of the program:

```
Running server on port:  5556
Running server on port:  5558
8 server#2739
Connected to server with port 5556
Connected to publisher with port 5558
Recieved control command: Continue
9 server#2739
Processing ...  9 server#2739
Recieved control command: Continue
9 server#2739
Processing ...  9 server#2739
Recieved control command: Continue
9 server#2739
```

```
Processing ...  9 server#2739
Recieved control command: Continue
8 server#2739
Recieved control command: Continue
8 server#2739
Recieved control command: Continue
8 server#2739
Recieved control command: Exit
Recieved exit command, client will stop recieving messages
8 server#2739
9 server#2739
8 server#2739
```

## PyZmq Tornado Event Loop

ØMQ Poller can be used to serve and communicate with multiple sockets. How ever, with ØMQ Poller, you end up with explicit blocks (under if loop) for handling the sockets. Each socket registered with ØMQ Poller has to have an explicit "if block" to handle it.

PyZmq includes the tornado ioloop and adapts its IOStream class into ZMQStream for handling poll events on ØMQ sockets. You can register callbacks to receive and send data.

Before you do this, you must have tornado module installed:

```
pip install tornado
```

We will be redoing the previous program to take advantage of the ZMQStream and Tornado ioloop.

**pyzmq_stream_poller.py**

You must first install PyZMQ's IOLoop.

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

from zmq.eventloop import ioloop, zmqstream
ioloop.install()
```

We have left the command server and the topic publisher same as before.

```
def server_push(port="5556"):
    context = zmq.Context()
    socket = context.socket(zmq.PUSH)
    socket.bind("tcp://*:%s" % port)
    print "Running server on port: ", port
    # serves only 5 request and dies
    for reqnum in range(10):
        if reqnum < 6:
            socket.send("Continue")
        else:
            socket.send("Exit")
            break
        time.sleep (1)
```

```
def server_pub(port="5558"):
    context = zmq.Context()
    socket = context.socket(zmq.PUB)
    socket.bind("tcp://*:%s" % port)
    publisher_id = random.randrange(0,9999)
    print "Running server on port: ", port
    # serves only 5 request and dies
    for reqnum in range(10):
        # Wait for next request from client
        topic = random.randrange(8,10)
        messagedata = "server#%s" % publisher_id
        print "%s %s" % (topic, messagedata)
        socket.send("%d %s" % (topic, messagedata))
        time.sleep(1)
```

Message handlers are separated from the worker logic. Also note, that we stop the event loop once the worker receives the "Exit" command.

```
def getcommand(msg):
        print "Received control command: %s" % msg
        if msg[0] == "Exit":
                print "Received exit command, client will stop receiving messages"
                should_continue = False
                ioloop.IOLoop.instance().stop()

def process_message(msg):
        print "Processing ... %s" % msg
```

Here, you can see that we use ZMQStream class to register callbacks. The callbacks are the handlers that we had written earlier. The "If blocks" in previous program has been converted to callbacks registered with tornado event loop. There are no explicit socket handling blocks here.

```
def client(port_push, port_sub):
        context = zmq.Context()
        socket_pull = context.socket(zmq.PULL)
        socket_pull.connect ("tcp://localhost:%s" % port_push)
        stream_pull = zmqstream.ZMQStream(socket_pull)
        stream_pull.on_recv(getcommand)
        print "Connected to server with port %s" % port_push

        socket_sub = context.socket(zmq.SUB)
        socket_sub.connect ("tcp://localhost:%s" % port_sub)
        socket_sub.setsockopt(zmq.SUBSCRIBE, "9")
        stream_sub = zmqstream.ZMQStream(socket_sub)
        stream_sub.on_recv(process_message)
        print "Connected to publisher with port %s" % port_sub
        ioloop.IOLoop.instance().start()
        print "Worker has stopped processing messages."


if __name__ == "__main__":
    # Now we can run a few servers
    server_push_port = "5556"
```

```
    server_pub_port = "5558"
    Process(target=server_push, args=(server_push_port,)).start()
    Process(target=server_pub, args=(server_pub_port,)).start()
    Process(target=client, args=(server_push_port,server_pub_port,)).start()
```

In the output, you should notice that client has exited prior to the publishers which keeps publishing without any subscribers to process these messages:

```
Running server on port:  5556
Running server on port:  5558
8 server#2028
Connected to server with port 5556
Connected to publisher with port 5558
Received control command: ['Continue']
9 server#2028
Processing ... ['9 server#2028']
Received control command: ['Continue']
8 server#2028
Received control command: ['Continue']
8 server#2028
Received control command: ['Continue']
8 server#2028
Received control command: ['Continue']
9 server#2028
Processing ... ['9 server#2028']
Received control command: ['Continue']
9 server#2028
Processing ... ['9 server#2028']
Received control command: ['Exit']
Received exit command, client will stop receiving messages
Worker has stopped processing messages.
8 server#2028
8 server#2028
9 server#2028
```

# PyZmq Devices

ØMQ devices are full fledged programs. They have an embedded while loop which block execution once invoked. See Devices in PyZMQ.

pyzmq provides convenient classes for launching devices in a background thread or processes.

## Streamer

Here we will use the **ProcessDevice** to create a STREAMER device for pipelining server and workers.

**streamerdevice.py**

```
import time
import zmq
from zmq.devices.basedevice import ProcessDevice
from multiprocessing import Process

frontend_port = 5559
backend_port = 5560
```

```
number_of_workers = 2
```

The key difference here is that while *zmq.device* take Socket objects as arguments, *zmq.devices.basedevice.ProcessDevice* takes socket types.

```
streamerdevice  = ProcessDevice(zmq.STREAMER, zmq.PULL, zmq.PUSH)
```

For each configuration method (bind/connect/setsockopt), the proxy methods are prefixed with "in_" or "out_" corresponding to the frontend and backend sockets.

```
streamerdevice.bind_in("tcp://127.0.0.1:%d" % frontend_port )
streamerdevice.bind_out("tcp://127.0.0.1:%d" % backend_port)
streamerdevice.setsockopt_in(zmq.IDENTITY, 'PULL')
streamerdevice.setsockopt_out(zmq.IDENTITY, 'PUSH')
```

Finally, you can start the device in background.

```
streamerdevice.start()
```

Server and workers in the pipeline have been kept relatively simple for illustration purposes.

```
def server():
    context = zmq.Context()
    socket = context.socket(zmq.PUSH)
    socket.connect("tcp://127.0.0.1:%d" % frontend_port)

    for i in xrange(0,10):
        socket.send('#%s' % i)

def worker(work_num):
    context = zmq.Context()
    socket = context.socket(zmq.PULL)
    socket.connect("tcp://127.0.0.1:%d" % backend_port)

    while True:
        message = socket.recv()
        print "Worker #%s got message! %s" % (work_num, message)
        time.sleep(1)

for work_num in range(number_of_workers):
    Process(target=worker, args=(work_num,)).start()
time.sleep(1)

server()
```

The requests are farmed out to workers in load balanced manner:

```
Worker #1 got message! #0
Worker #0 got message! #1
Worker #1 got message! #2
Worker #0 got message! #3
Worker #1 got message! #4
Worker #0 got message! #5
Worker #1 got message! #6
```

```
Worker #0 got message! #7
Worker #1 got message! #8
Worker #0 got message! #9
```

## Queue

Here we will use the **ProcessDevice** to create a QUEUE device for connecting client and server.

**queuedevice.py**

```
import time
import zmq
from zmq.devices.basedevice import ProcessDevice
from multiprocessing import Process
import random

frontend_port = 5559
backend_port = 5560
number_of_workers = 2
```

As noted earlier, we do not pass socket instance but socket type to ProcessDevice. Also here, we observe the constraint on request/reply pattern by setting the high water mark to 1.

```
queuedevice = ProcessDevice(zmq.QUEUE, zmq.XREP, zmq.XREQ)
queuedevice.bind_in("tcp://127.0.0.1:%d" % frontend_port)
queuedevice.bind_out("tcp://127.0.0.1:%d" % backend_port)
queuedevice.setsockopt_in(zmq.HWM, 1)
queuedevice.setsockopt_out(zmq.HWM, 1)
queuedevice.start()
time.sleep (2)
```

Server waits on a request to which it replies.

```
def server(backend_port):
    print "Connecting a server to queue device"
    context = zmq.Context()
    socket = context.socket(zmq.REP)
    socket.connect("tcp://127.0.0.1:%s" % backend_port)
    server_id = random.randrange(1,10005)
    while True:
        message = socket.recv()
        print "Received request: ", message
        socket.send("Response from %s" % server_id)
```

Client makes a request and waits for a reply.

```
def client(frontend_port, client_id):
    print "Connecting a worker #%s to queue device" % client_id
    context = zmq.Context()
    socket = context.socket(zmq.REQ)
    socket.connect("tcp://127.0.0.1:%s" % frontend_port)
    #  Do 10 requests, waiting each time for a response
```

```
    for request in range (1,5):
        print "Sending request #%s" % request
        socket.send ("Request fron client: %s" % client_id)
        #  Get the reply.
        message = socket.recv()
        print "Received reply ", request, "[", message, "]"
```

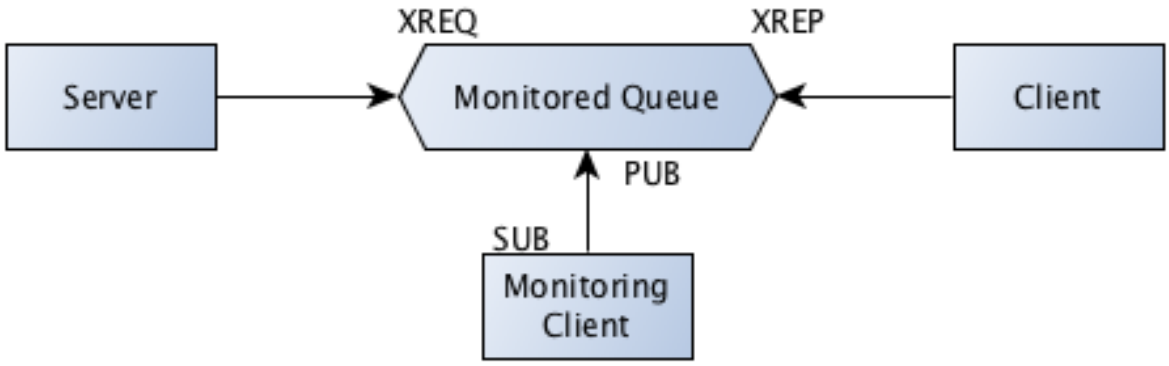
We have already started our device. Now we will bring up the server, before bringing up the client. Clients make a few request to server connected to our device.

```
Process(target=server, args=(backend_port,)).start()

time.sleep(2)

for client_id in range(number_of_workers):
    Process(target=client, args=(frontend_port, client_id,)).start()
```

Output:

```
Connecting a server to queue device
Connecting a worker #0 to queue device
Sending request #1
Connecting a worker #1 to queue device
Received request:  Request fron client: 0
Received reply  1 [ Response from 6548 ]
Sending request #2
Received request:  Request fron client: 0
Sending request #1
Received reply  2 [ Response from 6548 ]
Sending request #3
Received request:  Request fron client: 0
Received request:  Request fron client: 1
Received reply  3 [ Response from 6548 ]
Sending request #4
Received request:  Request fron client: 0
Received reply  1 [ Response from 6548 ]
Sending request #2
Received request:  Request fron client: 1
Received reply  4 [ Response from 6548 ]
Received reply  2 [ Response from 6548 ]
Sending request #3
Received request:  Request fron client: 1
Received reply  3 [ Response from 6548 ]
Sending request #4
Received request:  Request fron client: 1
Received reply  4 [ Response from 6548 ]
```

## Monitor Queue




**monitoredqueue.py**

MonitoredQueue allows you to create a Queue device. The messages in/out of the queue are published on a third socket.

```
import time
import zmq
from zmq.devices.basedevice import ProcessDevice
from zmq.devices.monitoredqueuedevice import MonitoredQueue
from zmq.utils.strtypes import asbytes
from multiprocessing import Process
import random

frontend_port = 5559
backend_port = 5560
monitor_port = 5562
number_of_workers = 2
```

MonitoredQueue accepts in/out socket type (type and not socket) like a typical ØMQ device. It also accepts a third socket types which conveniently can be a zmq.PUB type. This allows the communication on in/out socket to be published on a third socket for monitoring purposes.

Also you should read the following: pyzmq and unicode. As it says, PyZMQ is a wrapper for a C library and you should be passing in bytes and not string which in python 3 would be unicode strings. We will correct some of our examples later on for this purpose.

```
def monitordevice():
    in_prefix=asbytes('in')
    out_prefix=asbytes('out')
    monitoringdevice = MonitoredQueue(zmq.XREP, zmq.XREQ, zmq.PUB, in_prefix, out_
↪prefix)

    monitoringdevice.bind_in("tcp://127.0.0.1:%d" % frontend_port)
    monitoringdevice.bind_out("tcp://127.0.0.1:%d" % backend_port)
    monitoringdevice.bind_mon("tcp://127.0.0.1:%d" % monitor_port)

    monitoringdevice.setsockopt_in(zmq.HWM, 1)
    monitoringdevice.setsockopt_out(zmq.HWM, 1)
```

```
    monitoringdevice.start()
    print "Program: Monitoring device has started"
```

This is a simple server that receives a request and sends a reply.

```
def server(backend_port):
    print "Program: Server connecting to device"
    context = zmq.Context()
    socket = context.socket(zmq.REP)
    socket.connect("tcp://127.0.0.1:%s" % backend_port)
    server_id = random.randrange(1,10005)
    while True:
        message = socket.recv()
        print "Server: Received - %s" % message
        socket.send("Response from server #%s" % server_id)
```

This is a simple client that sends a request, receives and prints the reply.

```
def client(frontend_port, client_id):
    print "Program: Worker #%s connecting to device" % client_id
    context = zmq.Context()
    socket = context.socket(zmq.REQ)
    socket.connect("tcp://127.0.0.1:%s" % frontend_port)
    request_num = 1
    socket.send ("Request #%s from client#%s" % (request_num, client_id))
    #  Get the reply.
    message = socket.recv_multipart()
    print "Client: Received - %s" % message
```

This is a monitoring client that connects to the publisher socket in the device and publishes the monitoring information.

```
def monitor():
    print "Starting monitoring process"
    context = zmq.Context()
    socket = context.socket(zmq.SUB)
    print "Collecting updates from server..."
    socket.connect ("tcp://127.0.0.1:%s" % monitor_port)
    socket.setsockopt(zmq.SUBSCRIBE, "")
    while True:
        string = socket.recv_multipart()
        print "Monitoring Client: %s" % string
```

Here we just start device, server, client and monitoring clients as separate process.

```
monitoring_p = Process(target=monitordevice)
monitoring_p.start()
server_p = Process(target=server, args=(backend_port,))
server_p.start()
monitorclient_p = Process(target=monitor)
monitorclient_p.start()
time.sleep(2)

for client_id in range(number_of_workers):
    Process(target=client, args=(frontend_port, client_id,)).start()
```

```
time.sleep(10)
server_p.terminate()
monitorclient_p.terminate()
monitoring_p.terminate()
```

Output:

```
Program: Server connecting to device
Starting monitoring process
Collecting updates from server...
Program: Worker #0 connecting to device
Program: Worker #1 connecting to device
Server: Received - Request #1 from client#0
Monitoring Client: ['in', '\x00\xcb\xc5J9<$E9\xac\xf6\r:\x82\x92EU', '', 'Request #1␣
↪from client#0']
Monitoring Client: ['out', '\x00\xcb\xc5J9<$E9\xac\xf6\r:\x82\x92EU', '', 'Response␣
↪from server #4431']
Client: Received - ['Response from server #4431']
Server: Received - Request #1 from client#1
Monitoring Client: ['in', "\x00\r'C\x0f\xf6TO\x84\xbe\xe3\x85\xf6(\x07<\xab", '',
↪'Request #1 from client#1']
Client: Received - ['Response from server #4431']
Monitoring Client: ['out', "\x00\r'C\x0f\xf6TO\x84\xbe\xe3\x85\xf6(\x07<\xab", '',
↪'Response from server #4431']
```

CHAPTER 2

# Various links for understanding ZeroMQ

**Main Links**

- Google Search :)
- Zguide
- ZeroMQ introduction
- Whats wrong with AMQP
- Restful messaging services

**Source Code**

- zguide examples in Python

**Applications**

- zeromq based logger
- Distributed MapReduce with ZeroMQ
- Python web framework using 0MQ

**Videos**

- Advanced networking at Pycon
- zeromq is the answer

**Presentations**

- zeromq super sockets
- zeromq is the answer (pres)
- Distributed application using 0MQ

**Articles**

- Python Multiprocessing with ZeroMQ
- Python work queue with zeromq

- Que handler and 0MQ
- PupSub example with 0MQ
- Learning 0MQ example
- Messaging a high level intro
- ZMQ pupsub
- Gevent, ZeroMQ, WebSockets, and Flot

**Stackoverflow**

- ActiveMQ or RabbitMQ or ZeroMQ

# CHAPTER 3

## Competing or related products

RestMQ

```
Redis based message queue, uses HTTP as transport, JSON to format a
minimalist protocol and is organized as REST resources.
It is built using Python, Twisted, Cyclone (a Tornado implementation over twisted)
and Redis.
```

**These are not the same**

Some of these are much more and different from what ZeroMQ is. Some of these are message/queue/broker based design that offer many features like persistence, reliability etc

HornetMQ ( HornetMQ architecture ):

```
HornetQ is an open source project to build a multi-protocol, embeddable,
very high performance, clustered, asynchronous messaging system.
HornetQ is an example of Message Oriented Middleware (MoM) .
http://en.wikipedia.org/wiki/Message_oriented_middleware
It is written in Java
```

Apache Kafka ( Kafka Design ):

```
Kafka is a distributed publish-subscribe messaging system. It is designed to support␣
↪persistent messaging, high-throughput,
explicit support for partitioning messages over Kafka servers & support for parallel␣
↪data load into Hadoop.
```

MSMQ ( MSMQ architecture )

```
Microsoft Message Queuing, or MSMQ, is technology for asynchronous messaging.␣
↪Whenever there's need for two or more applications
(processes) to send messages to each other without having to immediately know results,
↪ MSMQ can be used.
```

ActiveMQ ( activemq architecture )

```
It is open source messaging and Integration Patterns server. It is JMS compliant. It␣
↪supports many cross-platform clients &
protocols. It supports OpenWire & Stomp.
```

RabbitMQ ( RabbitMQ details ):

```
RabbitMQ provides robust messaging for applications. It is the leading implementation␣
↪of AMQP,
the open standard for business messaging, and, through plug-ins, supports STOMP, HTTP␣
↪for lightweight web messaging,
and other protocols. See the usage of rabbitmq in openstack: http://nova.openstack.
↪org/devref/rabbit.html
```

httpsqs

```
HTTPSQS is a Simple Queue Service based on HTTP GET/POST protocol.
```

Amazon SQS

```
Amazon Simple Queue Service (Amazon SQS) offers a reliable, highly scalable, hosted␣
↪queue for storing messages
as they travel between computers. By using Amazon SQS, developers can simply move␣
↪data between distributed components
of their applications that perform different tasks, without losing messages or␣
↪requiring each component to be always available.

An interesting characteristics:
When a message is received, it becomes "locked" while being processed. This keeps␣
↪other computers from processing the message
simultaneously. If the message processing fails, the lock will expire and the message␣
↪will be available again. In the case
where the application needs more time for processing, the "lock" timeout can be␣
↪changed dynamically via the
ChangeMessageVisibility operation.
```

- genindex
- modindex
- search

CHAPTER 4

# Code

Code is present as folders inside each of the chapters. You can obtain them through github pyzmqnotes project.

CHAPTER 5

# Acknowledgements

Learning and experimenting with ØMQ through python adaptor pyzmq has been a very exciting experience. Thanks to all those who have contributed to it. Many folks have written articles and blogged on ØMQ. I wouldn't have stumbled on this great tool without that. I have tried to mention some of the excellent articles that I happened to read on this subject in the reference section.

I always begrudged writing documents till Sphinx came along. It is quite exciting to be able to share my notes through readthedocs. Of course, all these has been made very easy through github.

CHAPTER 6

# Contact

Do send your suggestions/corrections for improvement at “ashish.vid” at gmail dot com (Ashish Vidyarthi).