



IMT Atlantique

Bretagne-Pays de la Loire

École Mines-Télécom

Communication de ZeroMQ

Sheng Shen & Aziz Goudiaby

CHAPITRE 2

ZeroMQ Avancé

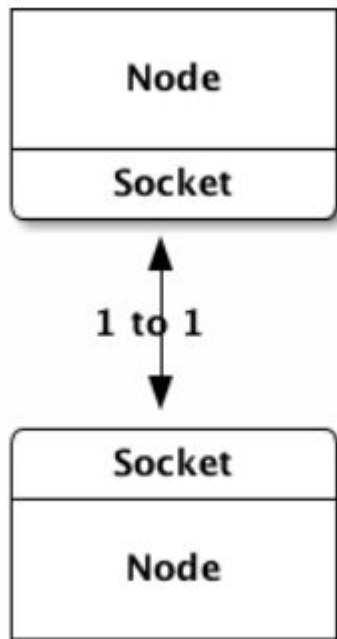


IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

2.1 Différences entre ZeroMQ et TCP

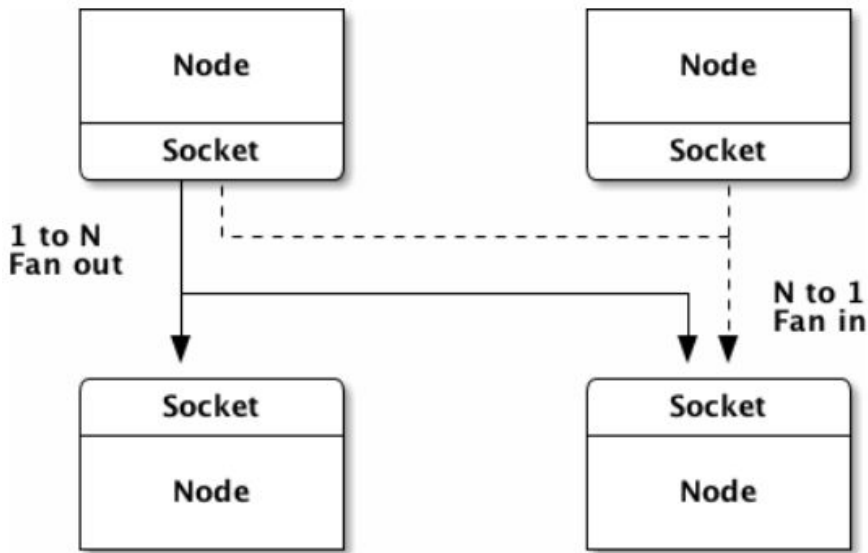
- Utiliser plusieurs protocoles:
inproc (in-process), ipc (inter-process), tcp, pgm (broadcast), epgm
- Lorsqu'un client connecte le ZeroMQ , il n'est pas obligé d'avoir un serveur qui connecte.
-Par exemple, nous pouvons ouvrir d'abord le client, puis le serveur
- Il n'y a pas de méthode *zmq.accept()*. Lorsqu'un socket est lié à un point, il commence automatiquement à accepter les connexions.
- Les sockets ZMQ transmettent des messages (données binaires) et non des octets (TCP) ou des trames (UDP)

2.1 Différences entre ZeroMQ et TCP



TCP Socket : 1 to 1

- Un socket peut avoir de nombreuses connexions sortantes et entrantes



ZeroMQ Socket : N to N

Méthode zmq.Poller()



- Connecter les différents tâches
- Initialiser le zmq.Poller() et register les différents tâches

client.py

```
1 import zmq
2 # Prepare our context and sockets
3 context = zmq.Context()
4
5 # Connect to task ventilator
6 receiver = context.socket(zmq.PULL)
7 receiver.connect("tcp://localhost:5557")
8
9 # Connect to weather server
10 subscriber = context.socket(zmq.SUB)
11 subscriber.connect("tcp://localhost:5556")
12 subscriber.setsockopt(zmq.SUBSCRIBE, "10001")
13
14 # Initialize poll set
15 poller = zmq.Poller()
16 poller.register(receiver, zmq.POLLIN)
17 poller.register(subscriber, zmq.POLLIN)
18 # Process messages from both sockets
19 while True:
20     socks = dict(poller.poll())
21     if receiver in socks and socks[receiver] == zmq.POLLIN:
22         message = receiver.recv()
23     # process task
24     if subscriber in socks and socks[subscriber] == zmq.POLLIN:
25         message = subscriber.recv()
```

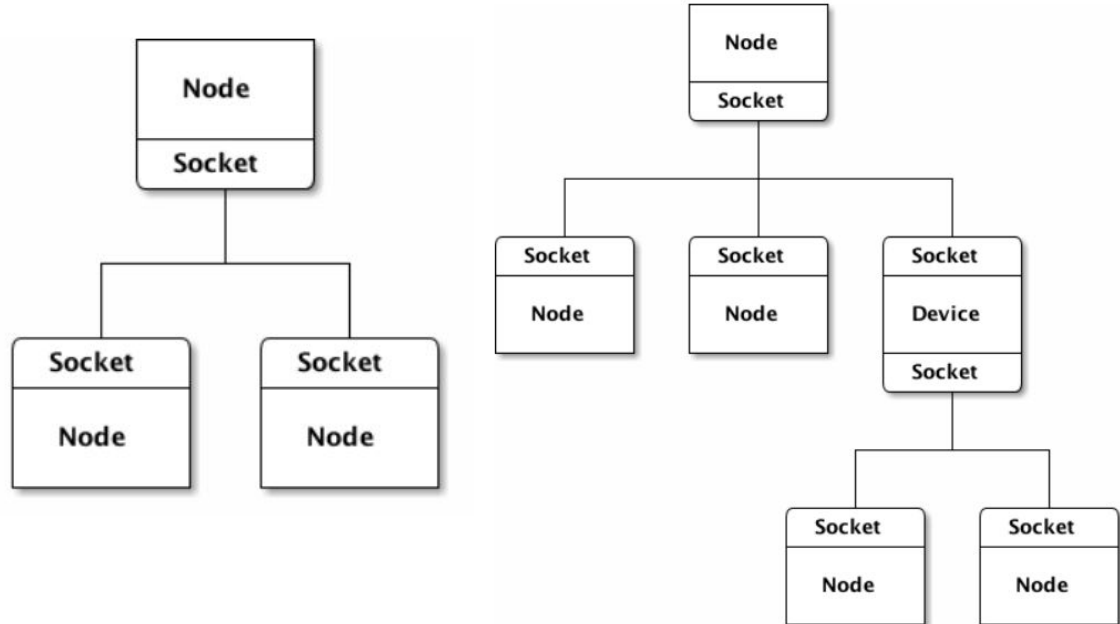
Lorsque l'application reçoit Ctrl-C ou des autres signaux d'interruption?

- Python exception
- zmq.ZMQError

```
1  import zmq
2  import signal
3
4  interrupted = False
5  def signal_handler(signum, frame):
6      global interrupted
7      interrupted = True
8
9  context = zmq.Context()
10 socket = context.socket(zmq.REP)
11 socket.bind("tcp://*:5558")
12
13 # python raise a KeyboardInterrupt
14 try:
15     socket.recv()
16 except KeyboardInterrupt:
17     print "W: interrupt received, proceeding..."
18
19 # or you can use a custom handler
20 counter = 0
21 signal.signal(signal.SIGINT, signal_handler)
22 while True:
23     try:
24         message = socket.recv(zmq.NOBLOCK)
25     except zmq.ZMQError:
26         pass
27     counter += 1
28     if interrupted:
29         print "W: interrupt received, killing server..."
30         break
31
```

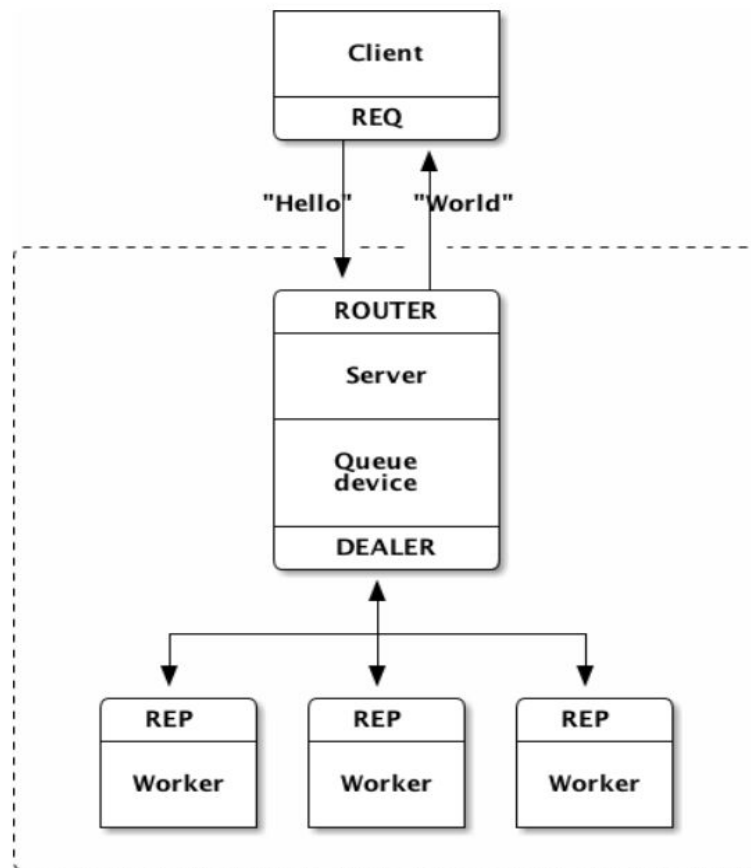
Périphérique

- le routage et l'adressage
- la fourniture de services
- la planification de files d'attente



Service Hello-World

- Le serveur démarre un ensemble de threads - Worker.
- Le serveur crée un socket ROUTER pour communiquer avec le client.
- Le serveur crée un socket DEALER pour communiquer avec le Worker à l'aide d'une interface interne (inproc).
- Le serveur démarre le périphérique interne QUEUE et connecte les sockets sur les deux ordinateurs.

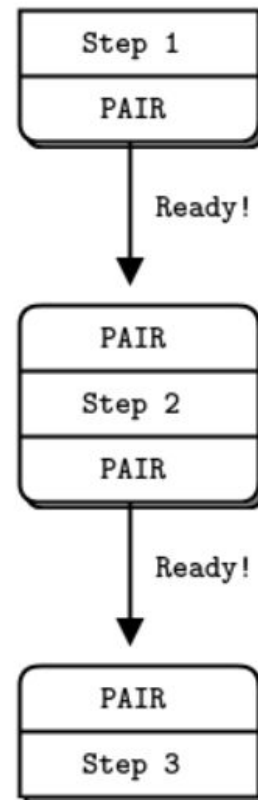


server.py

```
1 import time
2 import threading
3 import zmq
4 def worker_routine(worker_url, context):
5     """ Worker routine """
6     # Socket to talk to dispatcher
7     socket = context.socket(zmq.REP)
8     socket.connect(worker_url)
9     while True:
10         string = socket.recv()
11         print("Received request: [%s]\n" % (string))
12         # do some 'work'
13         time.sleep(1)
14         # send reply back to client
15         socket.send("World")
16 def main():
17     """ server routine """
18     url_worker = "inproc://workers"
19     url_client = "tcp://*:5555"
20     # Prepare our context and sockets
21     context = zmq.Context(1)
22     # Socket to talk to clients
23     clients = context.socket(zmq.ROUTER)
24     clients.bind(url_client)
25     # Socket to talk to workers
26     workers = context.socket(zmq.DEALER)
27     workers.bind(url_worker)
28     # Launch pool of worker threads
29     for i in range(5):
30         thread = threading.Thread(target=worker_routine, args=(url_worker, context, ))
31         thread.start()
32     zmq.device(zmq.QUEUE, clients, workers)
33     # We never get here but clean up anyhow
34     clients.close()
35     workers.close()
36     context.term()
37
```

- Pair Sockets
- Inproc transport

1. Deux threads communiquent via le protocole inproc en utilisant le même contexte;
2. Le step 1 crée un socket, se connecte au point de terminaison Inproc , puis démarre le thread enfant en lui transmettant l'objet de contexte.
3. Le step 2 crée un second socket, se connecte au noeud Inproc, puis envoie un signal prêt au thread 3.

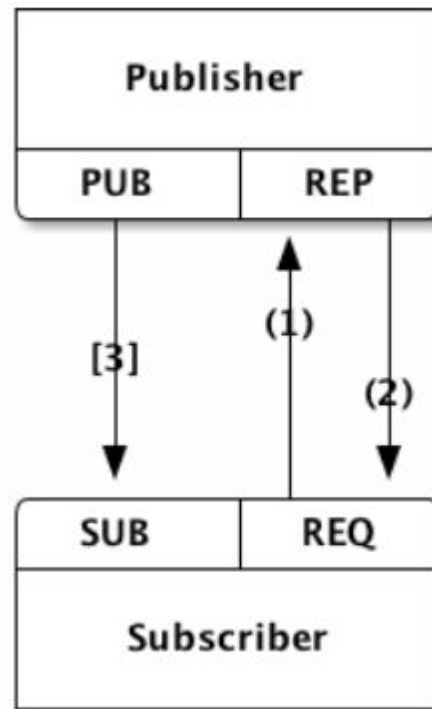


2.6 Signaling Between Threads

```
1  import threading
2  import zmq
3
4  def step1(context=None):
5      """Step 1"""
6      context = context or zmq.Context.instance()
7      # Signal downstream to step 2
8      sender = context.socket(zmq.PAIR)
9      sender.connect("inproc://step2")
10
11     sender.send(b"")
12
13  def step2(context=None):
14      """Step 2"""
15      context = context or zmq.Context.instance()
16      # Bind to inproc: endpoint, then start upstream thread
17      receiver = context.socket(zmq.PAIR)
18      receiver.bind("inproc://step2")
19
20     thread = threading.Thread(target=step1)
21     thread.start()
22
23     # Wait for signal
24     msg = receiver.recv()
25
26     # Signal downstream to step 3
27     sender = context.socket(zmq.PAIR)
28     sender.connect("inproc://step3")
29     sender.send(b"")
```

```
31  def main():
32      """ server routine """
33      # Prepare our context and sockets
34      context = zmq.Context.instance()
35
36     # Bind to inproc: endpoint
37     receiver = context.socket(zmq.PAIR)
38     receiver.bind("inproc://step3")
39
40     thread = threading.Thread(target=step2)
41     thread.start()
42
43     # Wait for signal
44     string = receiver.recv()
45
46     print("Test successful!")
47
48     receiver.close()
49     context.term()
```

- Le Publisher sait à l'avance combien de Subscriber s'attend. Ce numéro peut être spécifié arbitrairement.
- Le Publisher démarre et attend que tous les Subscriber se connectent. Chaque Subscriber s'abonne puis indique à l'éditeur qu'il est prêt via un autre socket.
- Lorsque le Publisher a tous les abonnés connectés, il commence à publier des données



2.7 Node Coordination

publisher.py

```
import zmq

# We wait for 10 subscribers
SUBSCRIBERS_EXPECTED = 10

def main():
    context = zmq.Context()

    # Socket to talk to clients
    publisher = context.socket(zmq.PUB)
    # set SNDHWM, so we don't drop messages for slow subscribers
    publisher.sndhwm = 1100000
    publisher.bind('tcp://*:5561')

    # Socket to receive signals
    syncservice = context.socket(zmq.REP)
    syncservice.bind('tcp://*:5562')

    # Get synchronization from subscribers
    subscribers = 0
    while subscribers < SUBSCRIBERS_EXPECTED:
        # wait for synchronization request
        msg = syncservice.recv()
        # send synchronization reply
        syncservice.send(b'')
        subscribers += 1
        print("+1 subscriber (%i/%i)" % (subscribers, SUBSCRIBERS_EXPECTED))

    # Now broadcast exactly 1M updates followed by END
    for i in range(1000000):
        publisher.send(b'Rhubarb')

    publisher.send(b'END')
```

subscriber.py

```
import time
import zmq

def main():
    context = zmq.Context()

    # First, connect our subscriber socket
    subscriber = context.socket(zmq.SUB)
    subscriber.connect('tcp://localhost:5561')
    subscriber.setsockopt(zmq.SUBSCRIBE, b'')
    time.sleep(1)

    # Second, synchronize with publisher
    syncclient = context.socket(zmq.REQ)
    syncclient.connect('tcp://localhost:5562')

    # send a synchronization request
    syncclient.send(b'')

    # wait for synchronization reply
    syncclient.recv()

    # Third, get our updates and report how many we got
    nbr = 0
    while True:
        msg = subscriber.recv()
        if msg == b'END':
            break
        nbr += 1

    print('Received %d updates' % nbr)
```

2.8 Pub-Sub Message Envelopes

Frame 1

Key

Subscription key

Frame 2

Data

Actual message body

- En mode publication-abonnement, l'enveloppe contient des informations sur l'abonnement permettant de filtrer les messages qu'il n'est pas nécessaire de recevoir.

publisher.py

```
import time
import zmq

def main():
    # Prepare our context and publisher
    context = zmq.Context()
    publisher = context.socket(zmq.PUB)
    publisher.bind("tcp://*:5563")

    while True:
        # Write two messages, each with an envelope and content
        publisher.send_multipart([b"A", b"We don't want to see this"])
        publisher.send_multipart([b"B", b"We would like to see this"])
        time.sleep(1)

    # We never get here but clean up anyhow
    publisher.close()
    context.term()
```

subscriber.py

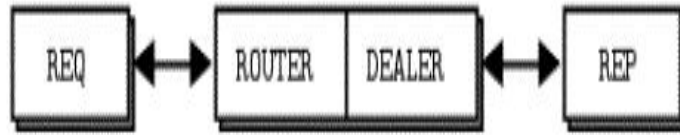
```
import zmq

def main():
    # Prepare our context and publisher
    context = zmq.Context()
    subscriber = context.socket(zmq.SUB)
    subscriber.connect("tcp://localhost:5563")
    subscriber.setsockopt(zmq.SUBSCRIBE, b"B")

    while True:
        # Read envelope with address
        [address, contents] = subscriber.recv_multipart()
        print("[%s] %s" % (address, contents))

    # We never get here but clean up anyhow
    subscriber.close()
    context.term()
```

Concept d'identité ZeroMQ : Router/Dealer



- RouterSocket

1er Cas

Client envoie d'abord une identité(connexion).
Insere une frame d'identité au début de chaque message avant de faire passer le message.

2ème Cas

Client ne spécifie rien.
Utilise le mécanisme de generation aléatoire d'identité.

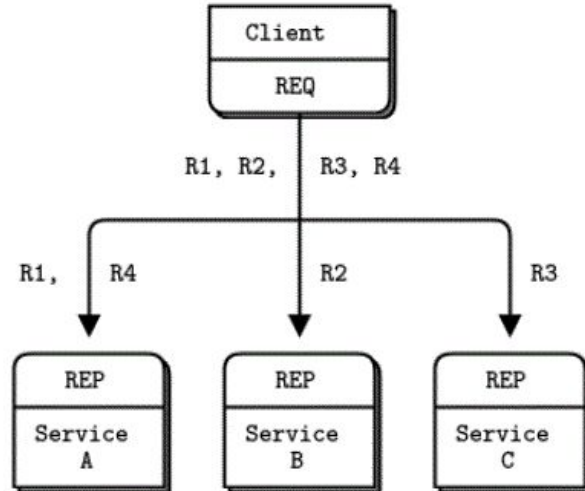
-DealerSocket

Idem que REQ mais asynchrone.

Frame 1	3	ABC	Identity of connection
Frame 2	0		Empty delimiter frame
Frame 3	5	Hello	Data frame

Files d'attentes partagés (Shared Queue)

Distribution de requete(Request Distribution)



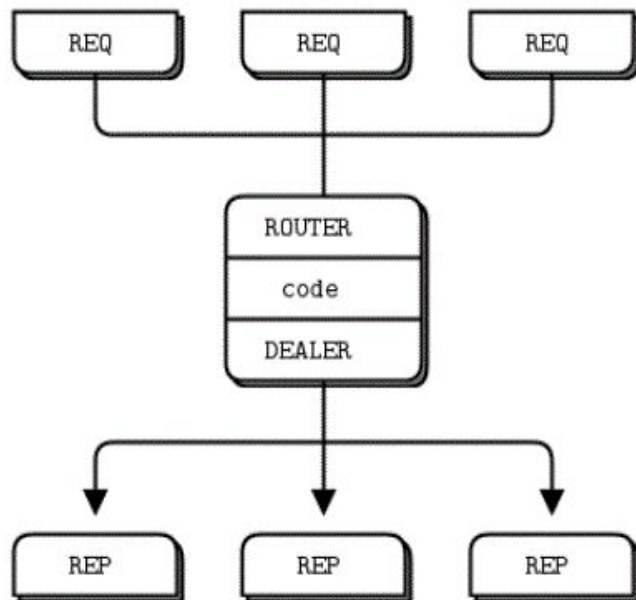
-Chaque client distribue ces demandes au service et doit connaître la topologie du réseau.

-Si on a 100 Clients et on ajoute 3 services supplémentaires nécessite une reconfiguration et un redémarrage des 100 clients pour qu'ils soient informés du changement.

-Architecture de Publisher/Workers
100 Publishers et 100 Workers (100*100 connexions).

Files d'attentes partagés (Shared Queue)

REQ-REPLY élargi(Extended REQ-REPLY)



Broker: Rappel : serveur de messagerie.

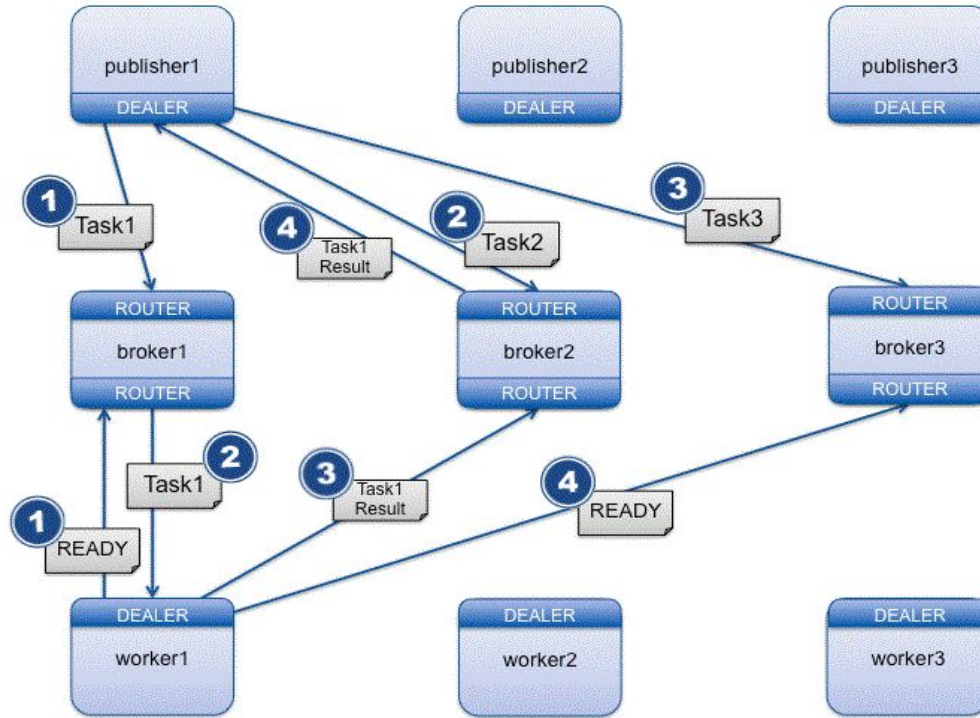
Le broker utilise `zmq_poll()` pour surveiller l'activité des sockets ROUTER et DEALER.

En REQ-REP élargi, Dealer et Router étant asynchrone, permet de faire une réponse-requête non bloquante.

Dealer et Router ont permis d'étendre la REQ-REP à un intermédiaire à savoir le broker.

Router reçoit ou génère l'ID du client de façon à pouvoir distinguer les requêtes venant des clients.

Utilisation de Broker avec la combinaison Router-Router



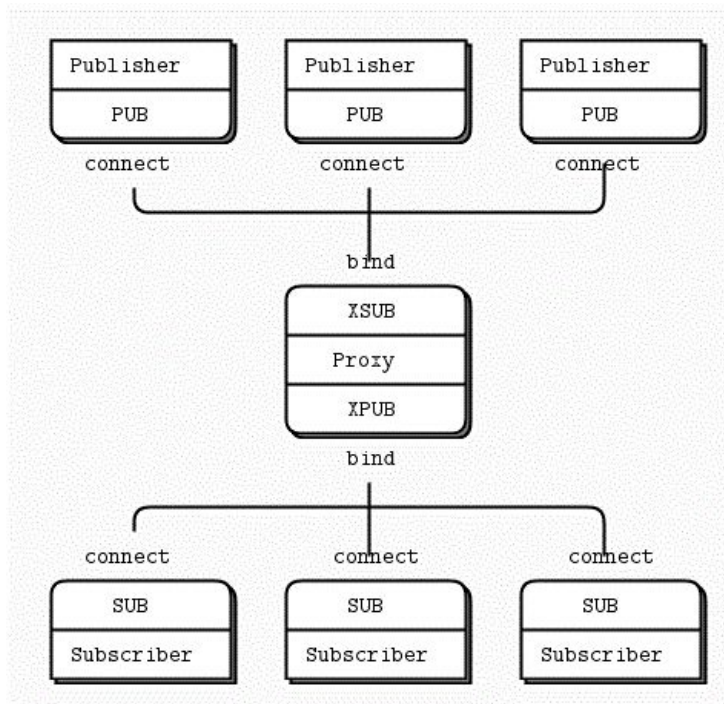
-Chaque publisher peut publier une tâche sur n'importe quel broker

-Chaque Worker peut récupérer une tâche et publier le résultat sur n'importe quel broker.

-Les brokers sont capables de router le résultat d'une tâche vers le publisher qui l'a publié.

Problème de découverte dynamique

Proxy HTTP



-Pour une architecture distribuée qui s'agrandit :
L'un des problèmes que l'on rencontre c'est :
Comment chaque noeud connaît la présence des autres?

Imaginons si nous avons une centaine client et serveur. On configure chaque client pour qu'il connaisse l'adresse du serveur.

-->Si on ajoute un client, aucun problème

-->Si on ajoute un serveur ,ça se complique.

Solution:

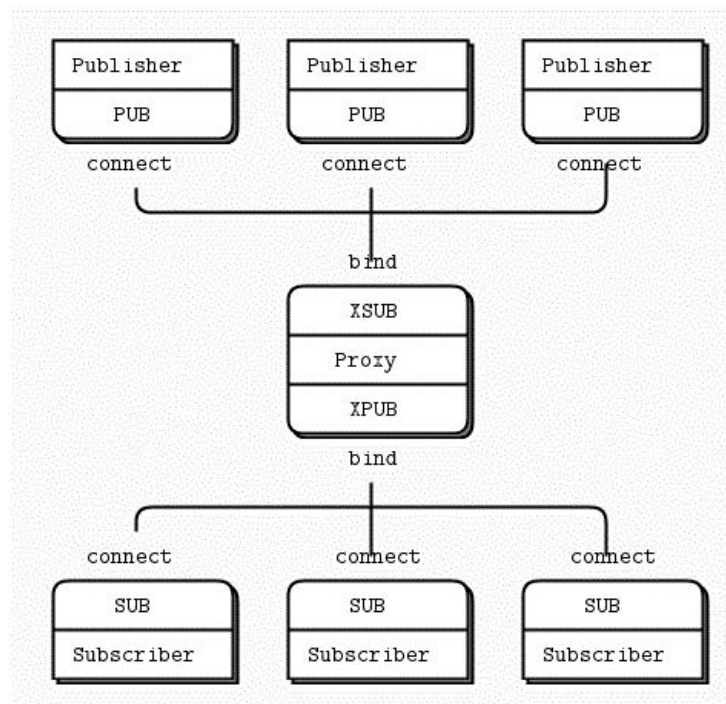
Le Proxy = centre de notre réseau qui:

-Ouvre des XSUB ,XPUB Sockets et attache chacune à une adresse statique du réseau.

XSUB/XPUB:similaire SUB/PUB.

Problème de découverte dynamique

XPUB/XSUB:similare PUB/SUB mais



-Chacun se connecte au proxy à la place de se connecter directement aux autres.

----> Facilite l'ajout des subscriber et des Publishers.

tutorial ZeroMQ : <http://zguide.zeromq.org/page:all>