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

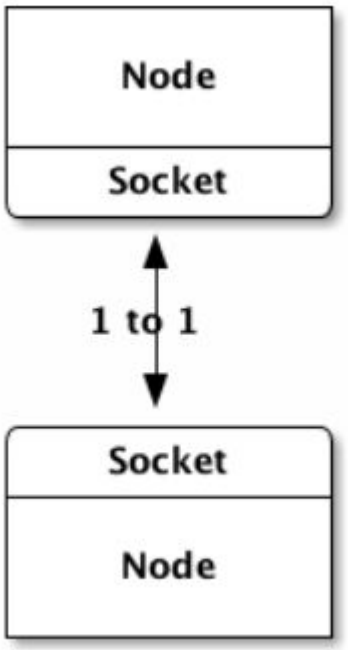
- Utiliser plusieurs protocoles:

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

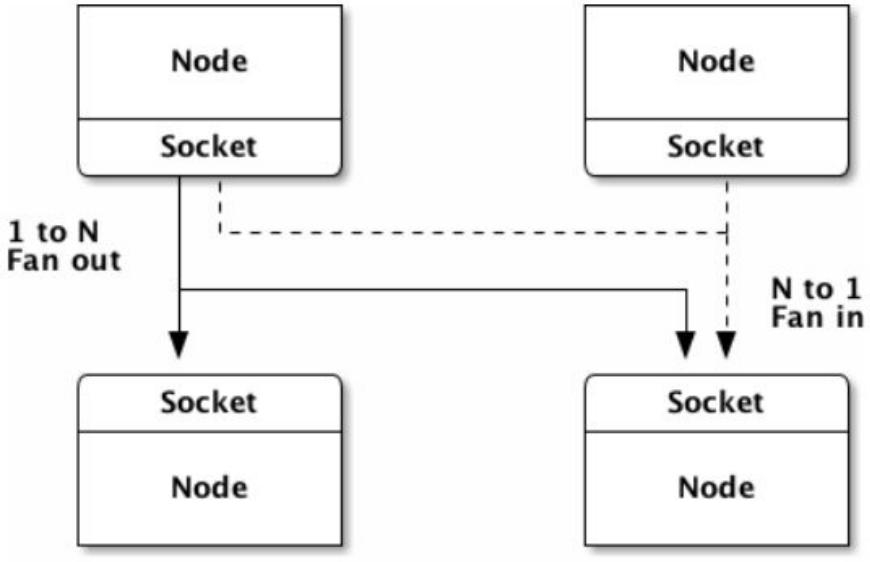
- Il n'y a pas de méthode *zmq.accept()*. Lorsqu'un socket est lié à un point, il commence automatiquement à accepter les connexions.

- Les sockets ZMQ transmettent(données binaires) des messages et non des octets (TCP) ou des trames (UDP)



TCP Socket : 1 to 1

- Un socket peut avoir de nombreuses connexions sortantes et entrantes



ZeroMQ Socket : N to N

Manipulation de plusieurs sockets

- Méthode zmq.Poller()

1. Connecter les différents tâches

2. Initialiser le zmq.Poller()

client.py

```python
import zmq
# Prepare our context and sockets
context = zmq.Context()

# Connect to task ventilator
receiver = context.socket(zmq.PULL)
receiver.connect("tcp://localhost:5557")

# Connect to weather server
subscriber = context.socket(zmq.SUB)
subscriber.connect("tcp://localhost:5556")
subscriber.setsockopt(zmq.SUBSCRIBE, "10001")

# Initialize poll set
poller = zmq.Poller()
poller.register(receiver, zmq.POLLIN)
poller.register(subscriber, zmq.POLLIN)
# Process messages from both sockets
while True:
    socks = dict(poller.poll())
    if receiver in socks and socks[receiver] == zmq.POLLIN:
        message = receiver.recv()
    # process task
    if subscriber in socks and socks[subscriber] == zmq.POLLIN:
        message = subscriber.recv()
```

Traitement des signaux d'interruption

- Lorsque le seveur reçoit Ctrl-C ou d'autres signaux d'interruption?

- Python exception

- zmq.ZMQError

```python
import zmq
import signal

interrupted = False
def signal_handler(signum, frame):
    global interrupted
    interrupted = True

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5558")

# python raise a KeyboardInterrupt
try:
    socket.recv()
except KeyboardInterrupt:
    print "W: interrupt received, proceeding..."

# or you can use a custom handler
counter = 0
signal.signal(signal.SIGINT, signal_handler)
while True:
    try:
        message = socket.recv(zmq.NOBLOCK)
    except zmq.ZMQError:
        pass
    counter += 1
    if interrupted:
        print "W: interrupt received, killing server..."
        break
```

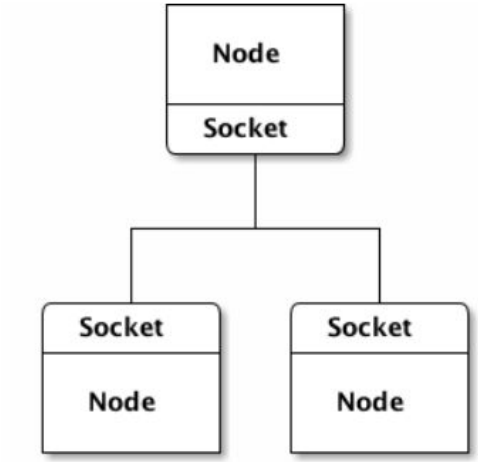
Proxy

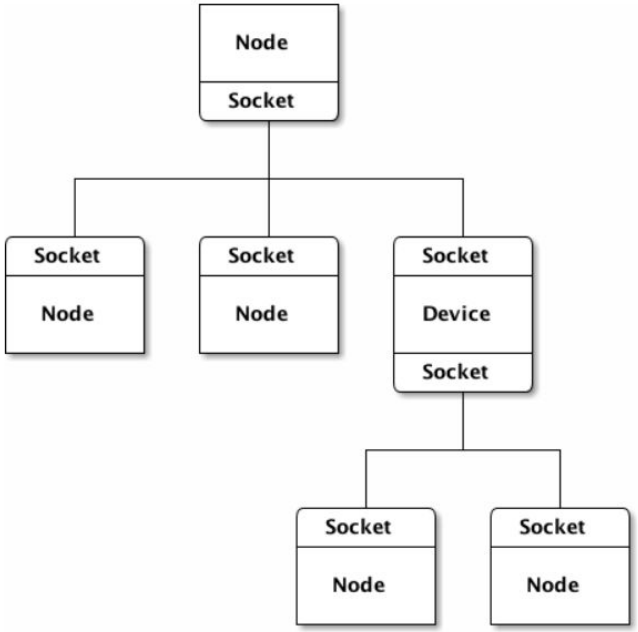
- Périphérique

  - le routage et l'adressage

  - la fourniture de services

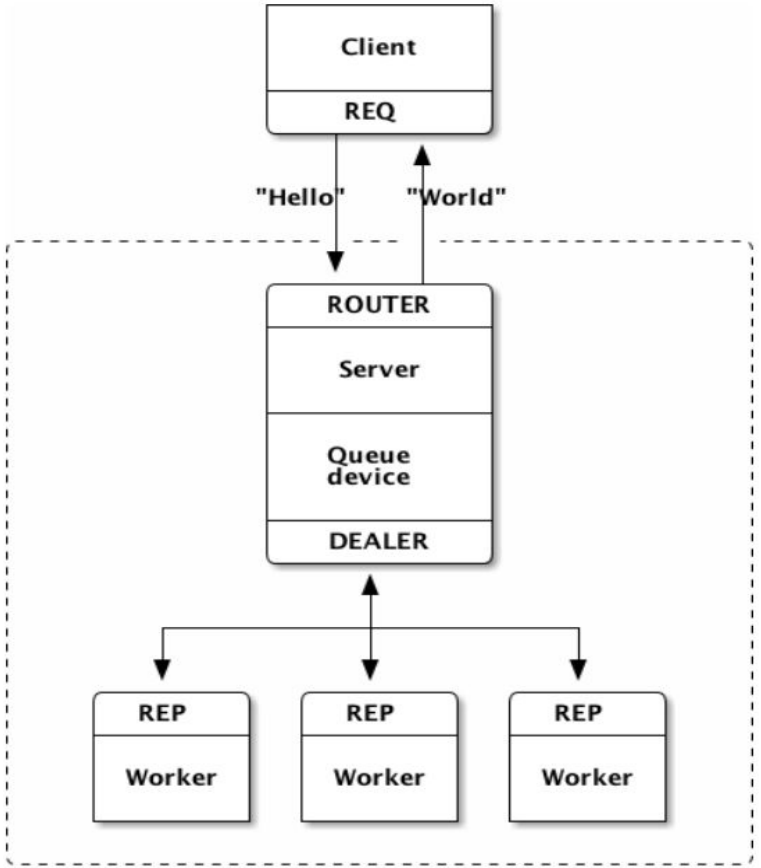
  - la planification de files d'attente





Multi-Threading (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

```
import time
import threading
import zmq

def worker_routine(worker_url, context=None):
    """Worker routine"""
    context = context or zmq.Context.instance()
    # Socket to talk to dispatcher
    socket = context.socket(zmq.REP)

    socket.connect(worker_url)

    while True:

        string  = socket.recv()

        print("Received request: [ %s ]" % (string))

        # do some 'work'
        time.sleep(1)

        #send reply back to client
        socket.send(b"World")
```

```
def main():
    """Server routine"""

    url_worker = "inproc://workers"
    url_client = "tcp://*:5555"

    # Prepare our context and sockets
    context = zmq.Context.instance()

    # Socket to talk to clients
    clients = context.socket(zmq.ROUTER)
    clients.bind(url_client)

    # Socket to talk to workers
    workers = context.socket(zmq.DEALER)
    workers.bind(url_worker)

    # Launch pool of worker threads
    for i in range(5):
        thread = threading.Thread(target=worker_routine, args=
(url_worker,))
        thread.start()

    zmq.proxy(clients, workers)

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

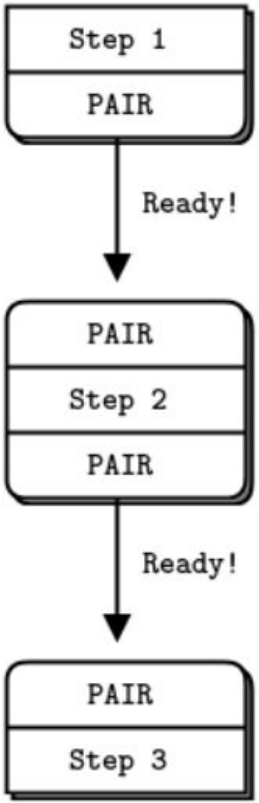
Signaling Between Threads

- 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.

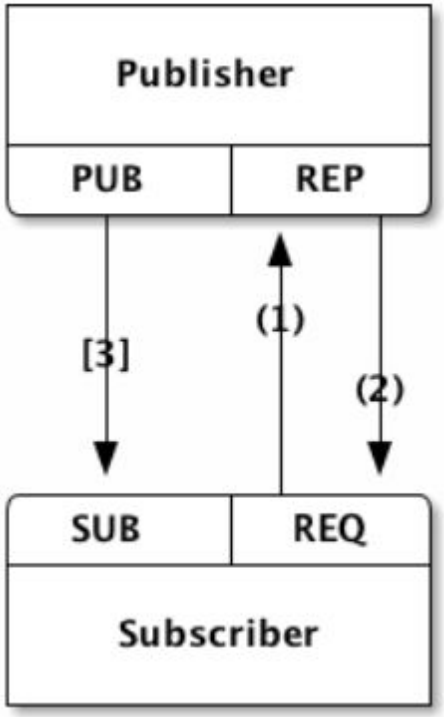


```
import threading
import zmq

def step1(context):
    """ step1 """

    # Signal downstream to step 2
    sender = context.socket(zmq.PAIR)
    sender.connect("inproc://step2")

    sender.send("")



def step2(context):
    """ step2 """

    # Bind to inproc: endpoint, then start upstream thread
    receiver = context.socket(zmq.PAIR)


    receiver.bind("inproc://step2")

    thread = threading.Thread(target=step1, args=(context, ))
    thread.start()

    # Wait for signal
    string = receiver.recv()

    # Signal downstream to step 3
    sender = context.socket(zmq.PAIR)
    sender.connect("inproc://step3")
    sender.send("")

    return
```

```
def main():
    """ server routine """
    # Prepare our context and sockets
    context = zmq.Context(1)

    # Bind to inproc: endpoint, then start upstream thread
    receiver = context.socket(zmq.PAIR)
    receiver.bind("inproc://step3")

    thread = threading.Thread(target=step2, args=(context, ))
    thread.start()

    # Wait for signal
    string = receiver.recv()

    print("Test successful!\n")

    receiver.close()
    context.term()

    return
```

Node Coordination

- L'éditeur sait à l'avance combien d'abonnés il s'attend. Ce numéro peut être spécifié arbitrairement.
- L'éditeur démarre et attend que tous les abonnés se connectent. Il s'agit de la partie relative à la coordination des noeuds. Chaque abonné s'abonne puis indique à l'éditeur qu'il est prêt via un autre socket.
- Lorsque l'éditeur a tous les abonnés connectés, il commence à publier des données



publisher.py

```
#  Synchronized publisher
#
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)
    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('')
        subscribers += 1
        print "+1 subscriber"

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

    publisher.send('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, "")

    time.sleep(1)

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

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

    # 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 == 'END':
            break
        nbr += 1

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

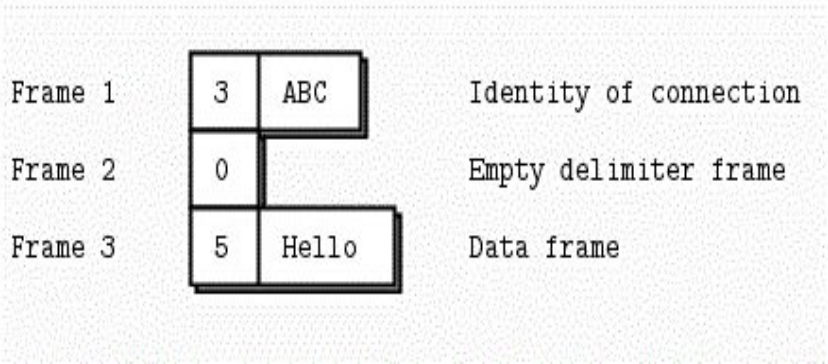
Pub-Sub Message Envelop

- 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.

```
import time
import zmq

def main():
    """ main method """

    # Prepare our context and publisher
    context    = zmq.Context(1)
    publisher = context.socket(zmq.PUB)
    publisher.bind("tcp://*:5563")

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

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

```
def main():
    """ main method """

    # Prepare our context and publisher
    context     = zmq.Context(1)

    subscriber = context.socket(zmq.SUB)
    subscriber.connect("tcp://localhost:5563")
    subscriber.setsockopt(zmq.SUBSCRIBE, "B")

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

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

Concept d'identité ZeroMQ : Router/Dealer



Multipart message

- RouterSocket

1er Cas

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

2ème Cas

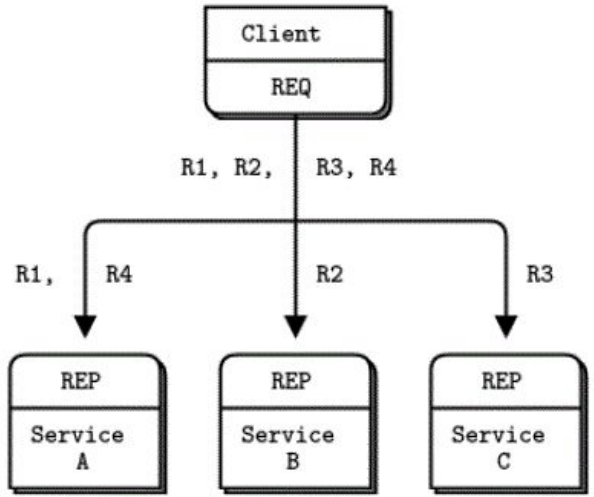
Client ne spécifie rien.
Utilise le mécanisme de generation aléatoire d'identité.

-DealerSocket

Idem que REQ mais asynchrone.

Files d’attentes partagés (Shared Queue)

Distribution de requete(Request Distribution)



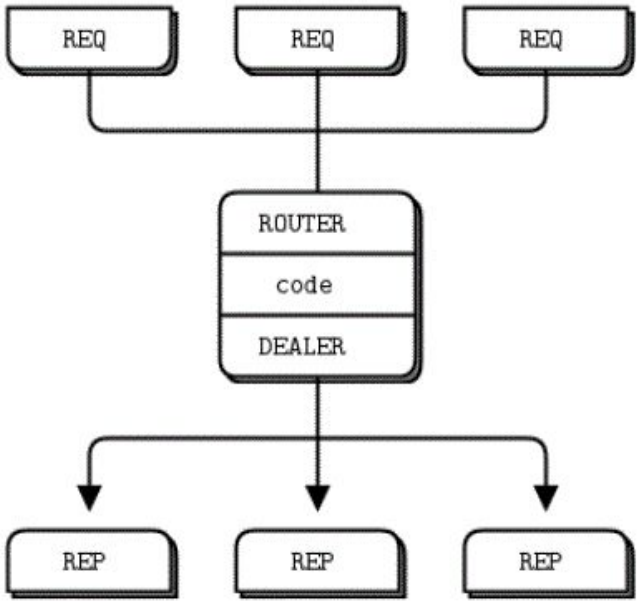
-Chaque client distribue ces demandes au service et doit connaître la topologie du réseau.

-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.

-Architectture 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.

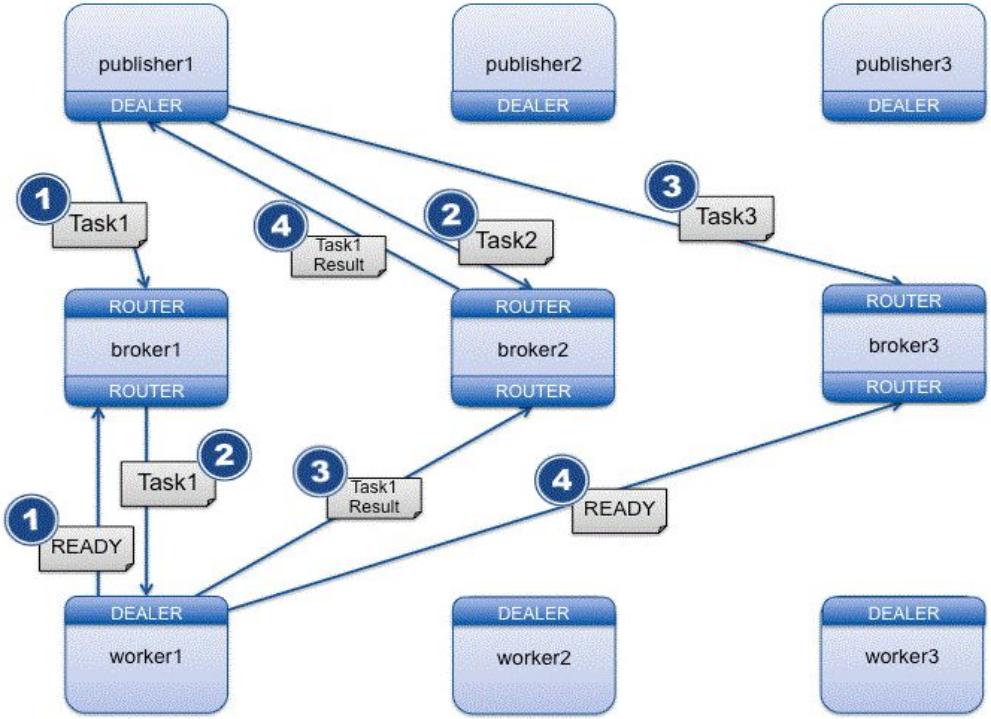
Le broker utilise zmq_poll () pour surveiller l'activité des sockets ROUTER et DEALER.

En REQ-REP élargi ,Dealer et Router étant asynchrone permets 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 genere l'ID du client de façon à pouvoir distinguer les requêtes venant des clients.

Utilisation de Broker avec la combinaison Router-Router

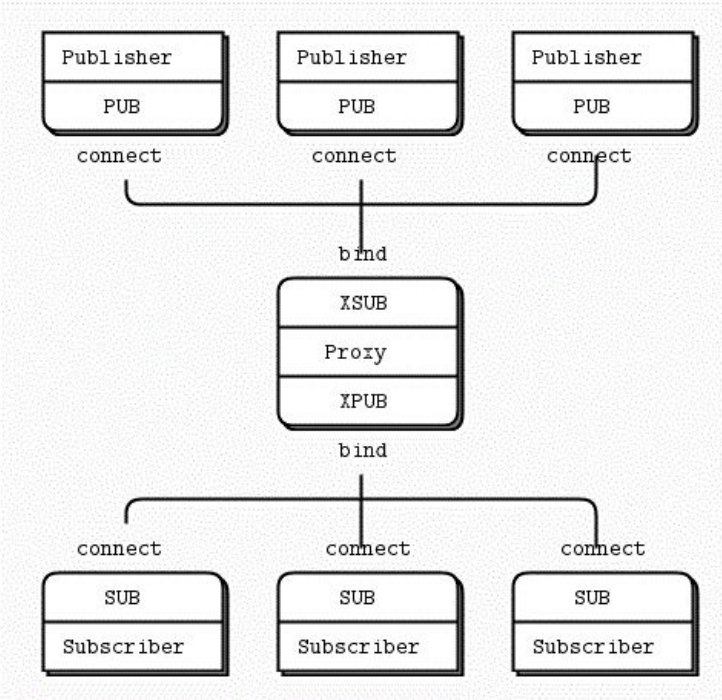


-Chaque publisher peut publier une tache sur n'importe quel broker

-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.

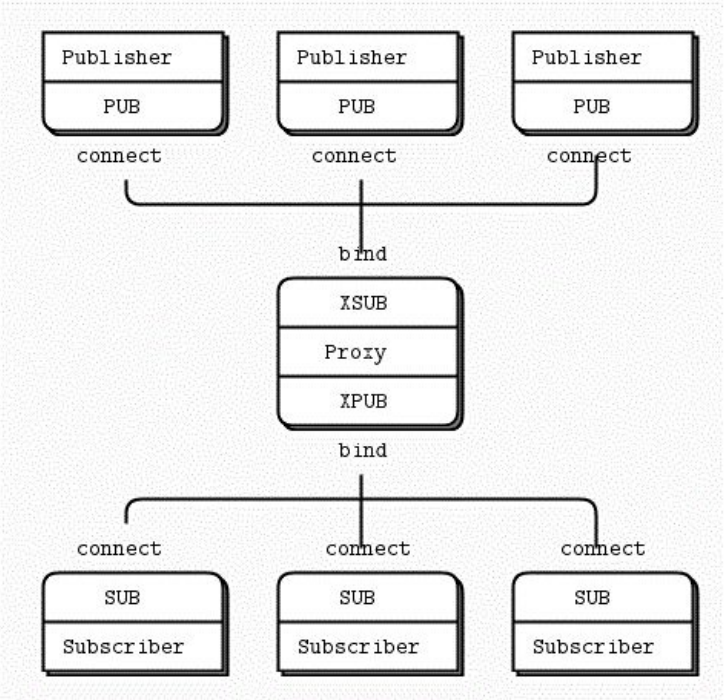
-->Si on ajoute un client, aucun problème
-->Si on ajoute un serveur ,ça se complique.

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.