Implementing distributed applications with

ØMQ

...and some other bad guys...

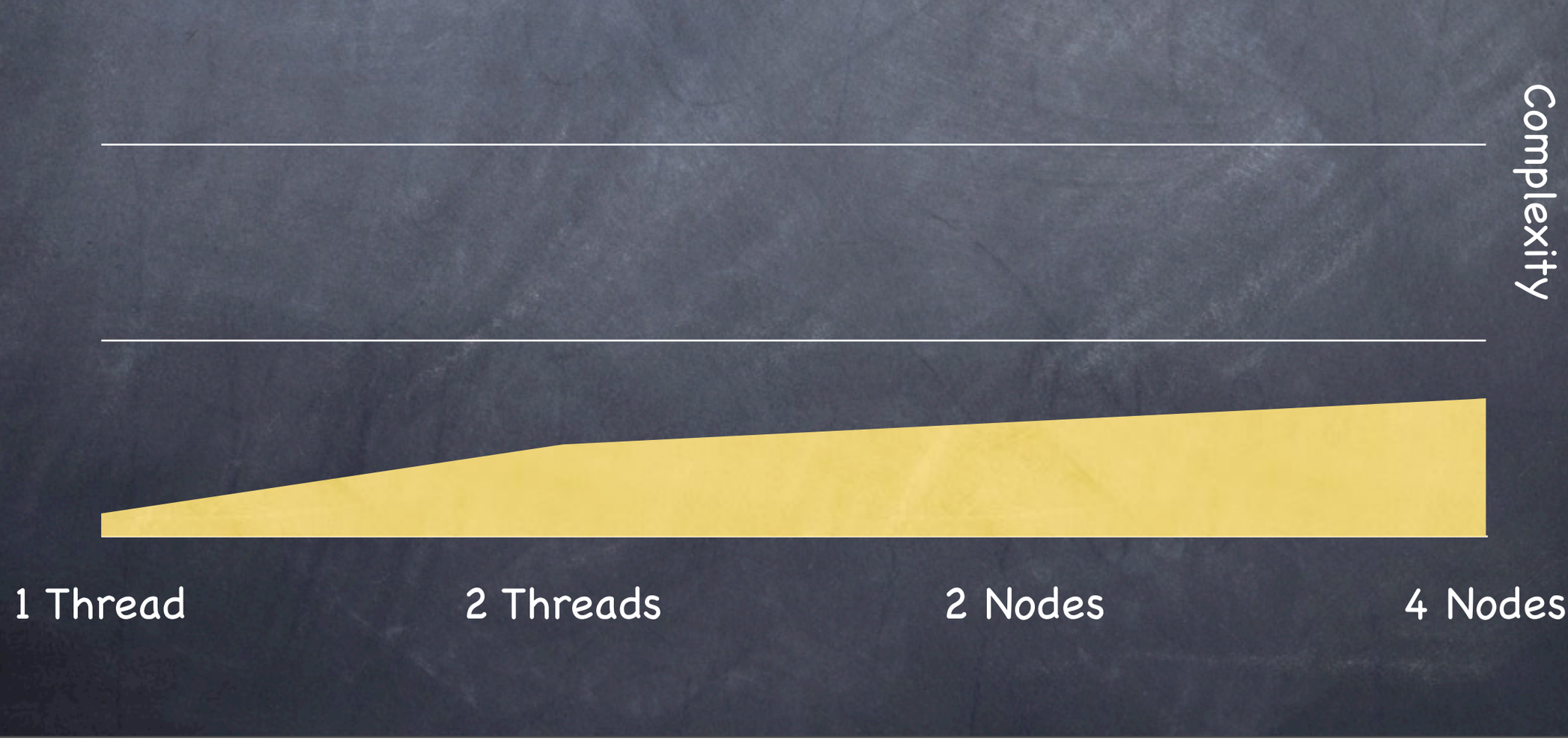
Crippa Francesco

BYTE-CODE Technology Solutions

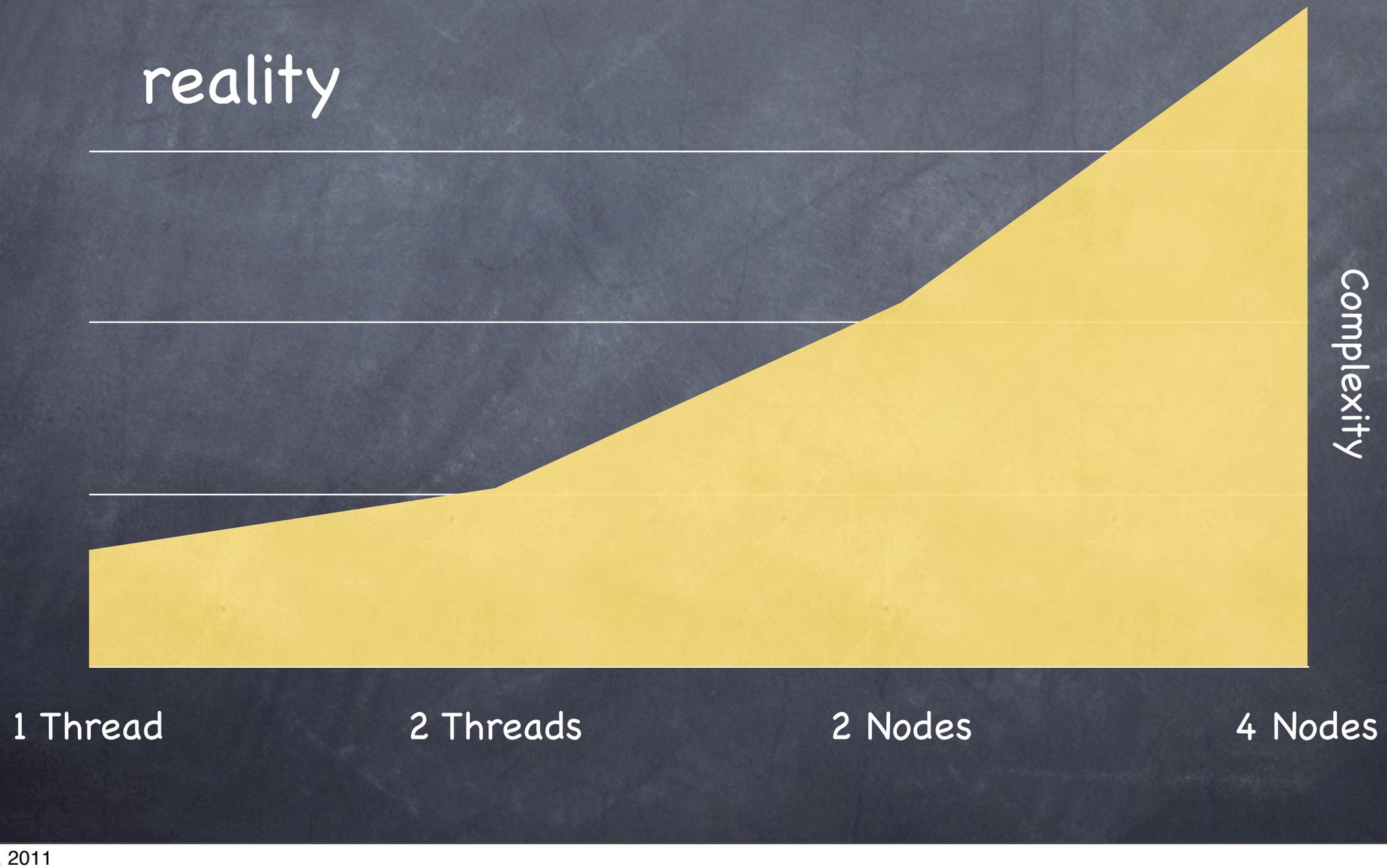
Scalability vs Complexity

Scalability vs Complexity

what we want



Scalability vs Complexity



The Q in ØMQ


Rosecrans Ave ½
Inglewood Ave 1½
Hawthorne Blvd 107 2¼
Rosecrans Ave
Manhattan Beach

The Ø in ØMQ

The Ø in ØMQ

- Zero Broker

The Ø in ØMQ

- Zero Broker
- Zero Latency (as close as possible...)

The Ø in ØMQ

- Zero Broker
- Zero Latency (as close as possible...)
- Zero administration

The Ø in ØMQ

- Zero Broker
- Zero Latency (as close as possible...)
- Zero administration
- Zero cost

The Ø in ØMQ

- Zero Broker
- Zero Latency (as close as possible...)
- Zero administration
- Zero cost
- Zero waste

Sockets

Sockets

- Unicast transports (inproc, ipc, tcp)

Sockets

- Unicast transports (inproc, ipc, tcp)
- Multicast transports (pgm or epgm)

Sockets

- Unicast transports (inproc, ipc, tcp)
- Multicast transports (pgm or epgm)
- connect() and bind() are independent

Sockets

- Unicast transports (inproc, ipc, tcp)
- Multicast transports (pgm or epgm)
- connect() and bind() are independent
- They are asynchronous (with queues)

Sockets

- Unicast transports (inproc, ipc, tcp)
- Multicast transports (pgm or epgm)
- connect() and bind() are independent
- They are asynchronous (with queues)
- They express a certain "messaging pattern"

Sockets

- Unicast transports (inproc, ipc, tcp)
- Multicast transports (pgm or epgm)
- connect() and bind() are independent
- They are asynchronous (with queues)
- They express a certain "messaging pattern"
- They are not necessarily one-to-one

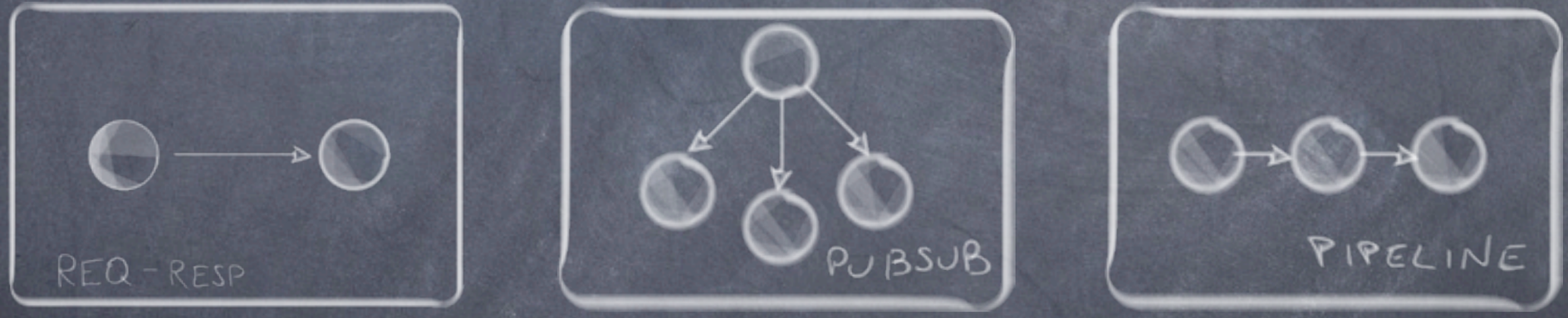
...and, of course...

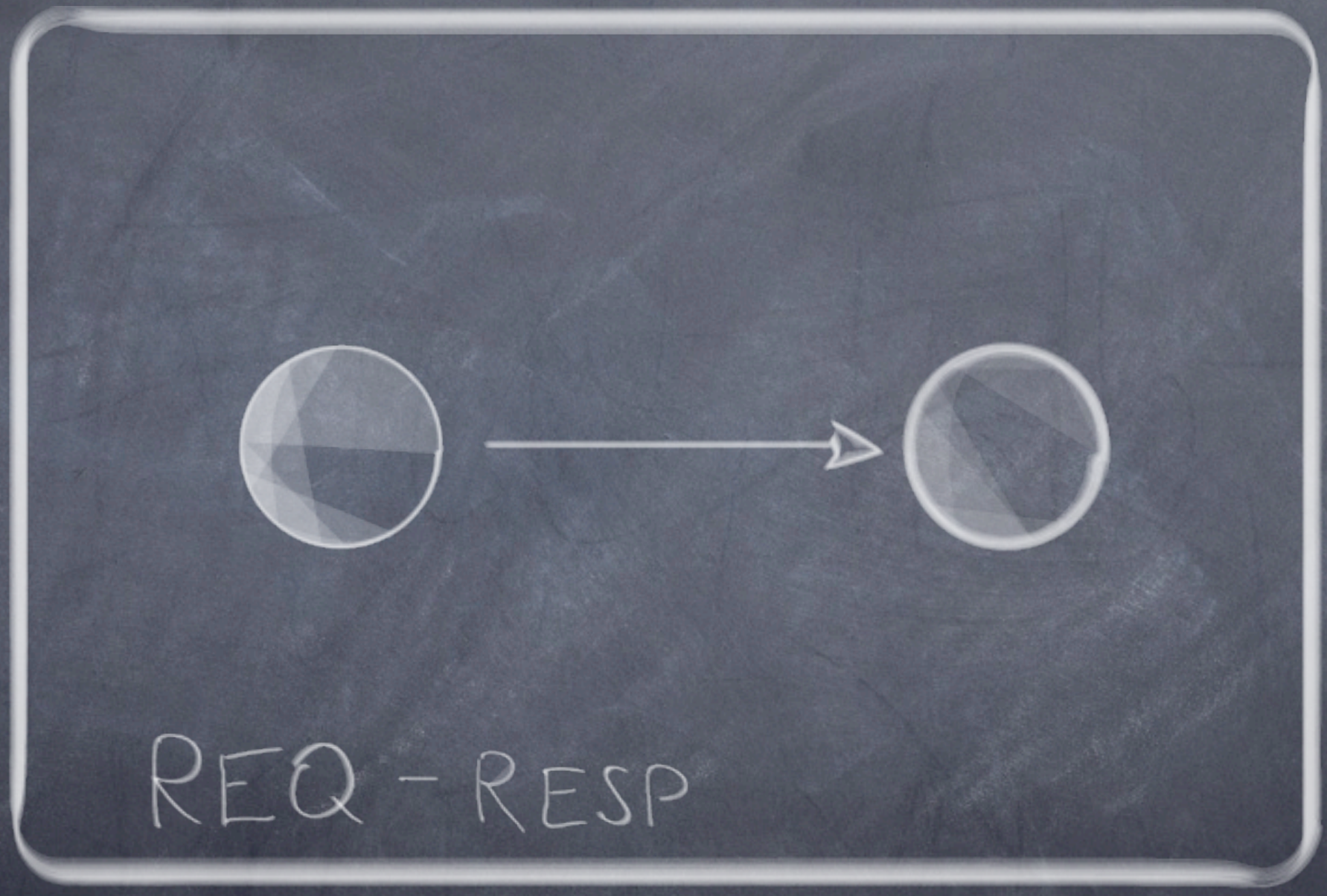
- Cross Platform (Linux, Windows, Mac, etc...)
- Multiple Languages (c, c++, python, java, ruby, erlang, php, perl, ada, c#, lua, scala, objective-c, go, haskell, racket, cl, basic...)
- OpenSource

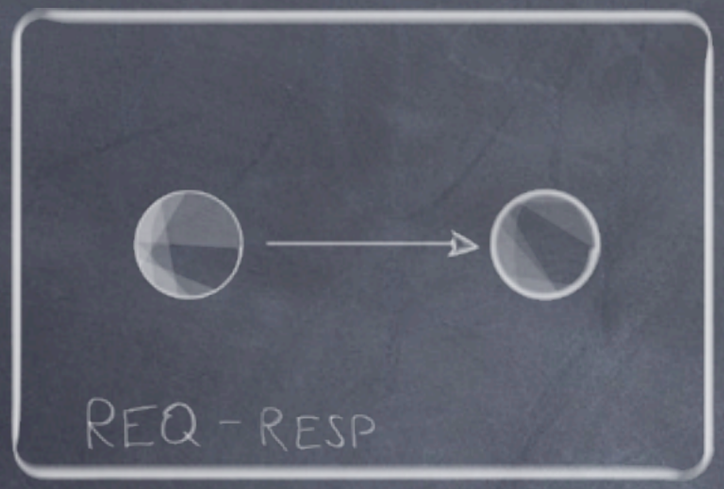
if you have a laptop...

- http://www.zeromq.org/
- http://zguide.zeromq.org/

Basic Message Patterns




REQ - RESP

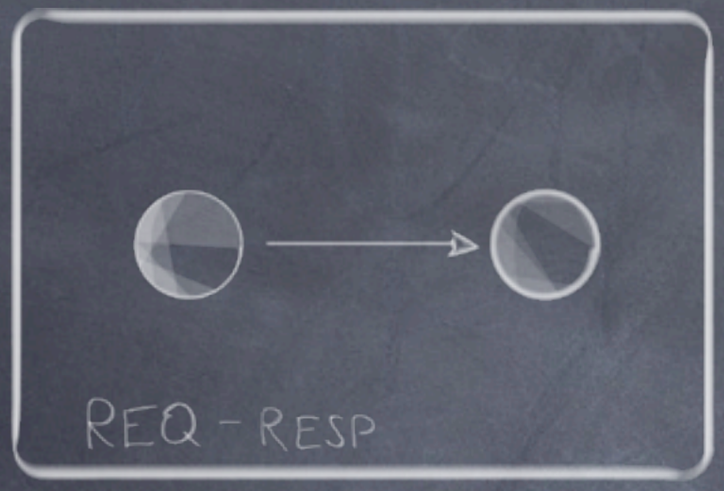


Server

hwserver.py

```
#
#   Hello World server in Python
#   Binds REP socket to tcp://*:5555
#   Expects "Hello" from client, replies with "World"
#
import zmq
import time

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    #  Wait for next request from client
    message = socket.recv()
    print "Received request: ", message

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

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

Line: 1 Column: 1 Python Django Soft Tabs: 4

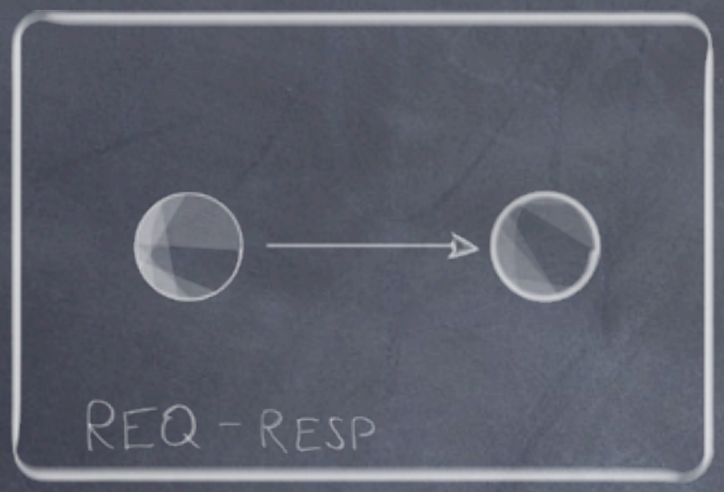


Client

hwclient.py

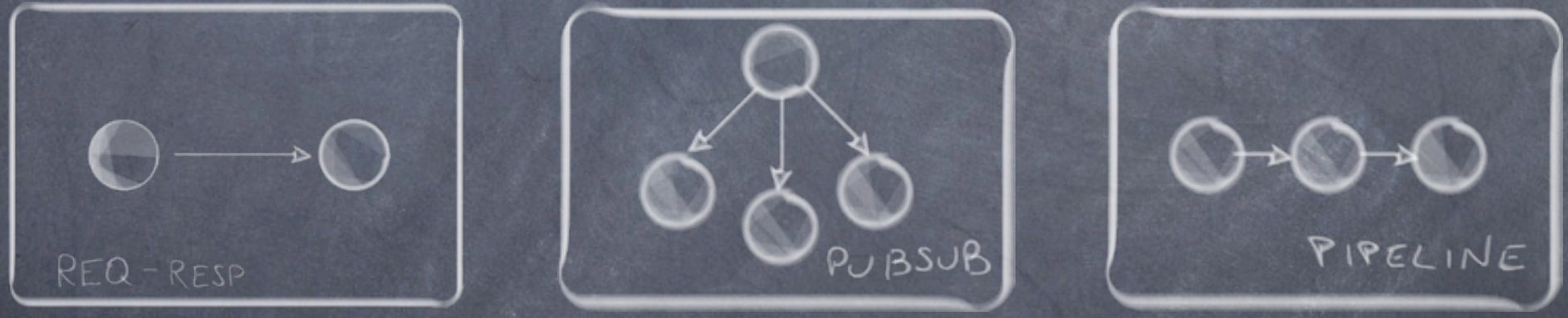
```
#
#   Hello World client in Python
#   Connects REQ socket to tcp://localhost:5555
#   Sends "Hello" to server, expects "World" back
#
import zmq

context = zmq.Context()

#  Socket to talk to server
print "Connecting to hello world server..."
socket = context.socket(zmq.REQ)
socket.connect ("tcp://localhost:5555")

#  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, "]"
```

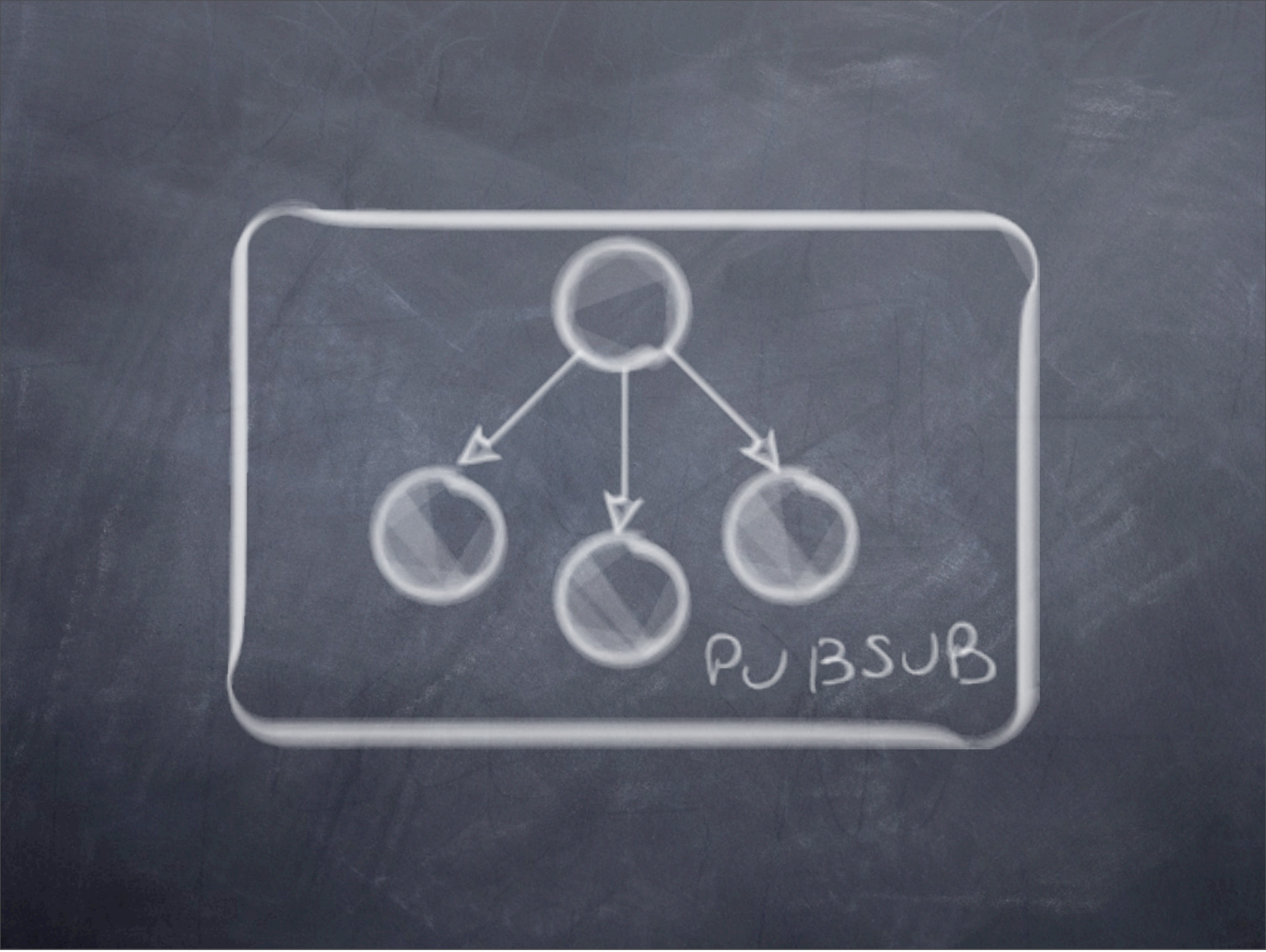
Line: 1 Column: 1 Python Django Soft Tabs: 4

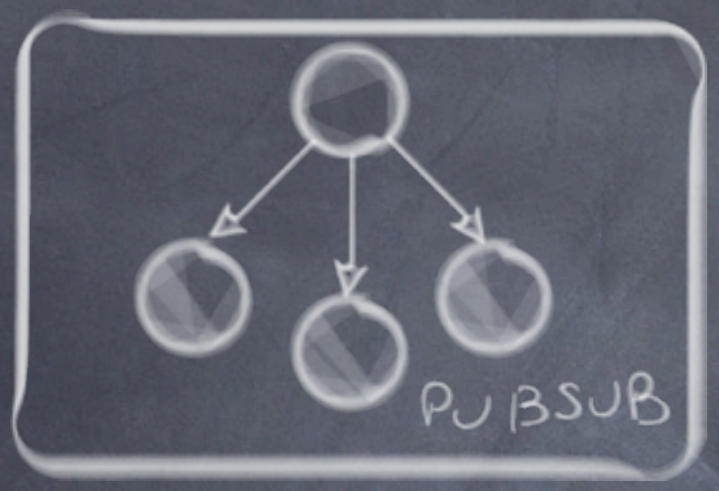


Demo

Basic Message Patterns




PUBSUB



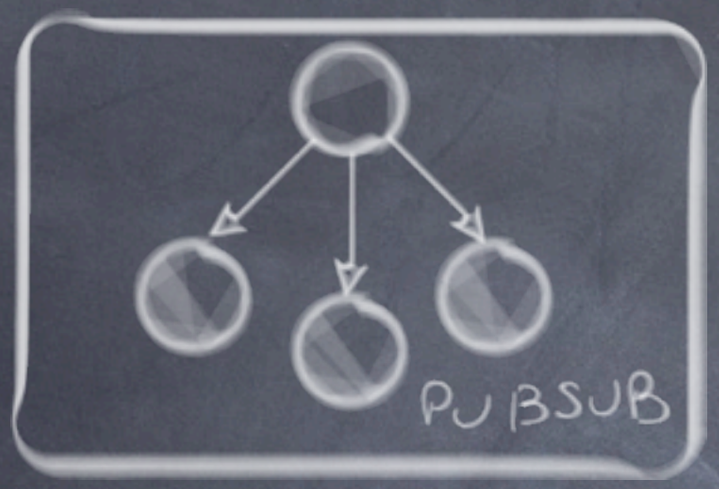
Server

wuserver.py

```
#
#   Weather update server
#   Binds PUB socket to tcp://*:5556
#   Publishes random weather updates
#

import zmq
import random

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:5556")

while True:
    zipcode = random.randrange(1,100000)
    temperature = random.randrange(1,215) - 80
    relhumidity = random.randrange(1,50) + 10

    socket.send("%d %d %d" % (zipcode, temperature, relhumidity))
```

Line: 1 Column: 1 Python Django Soft Tabs: 4

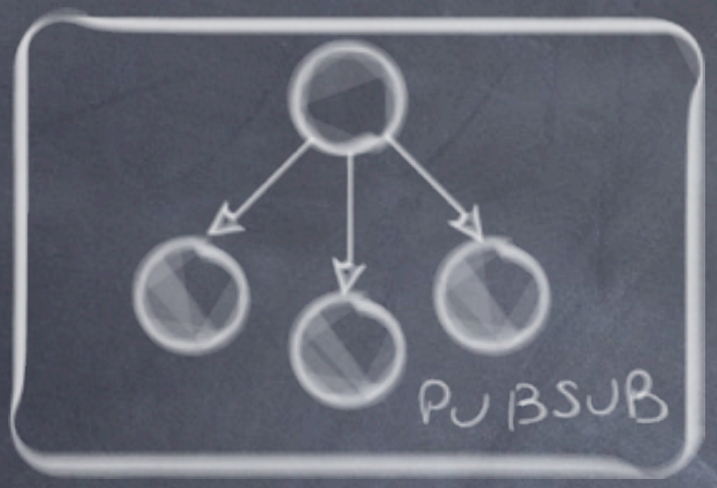
Client



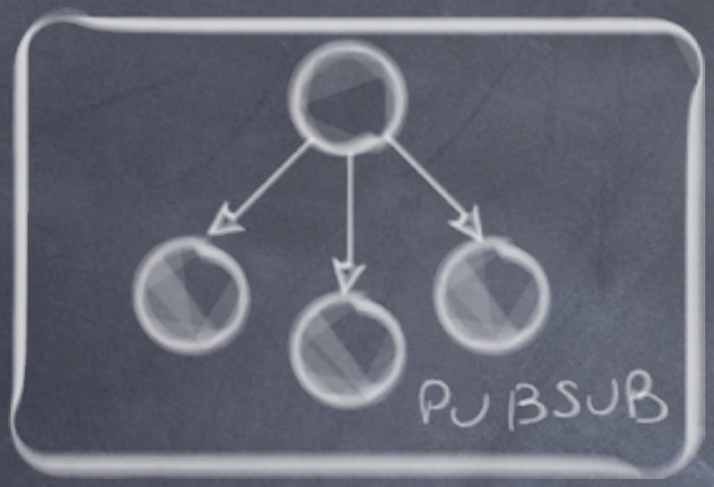
wuclient.py

```
#
#   Weather update client
#   Connects SUB socket to tcp://localhost:5556
#   Collects weather updates and finds avg temp in zipcode
#

import sys
import zmq

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

print "Collecting updates from weather server..."
socket.connect ("tcp://localhost:5556")

# Subscribe to zipcode, default is NYC, 10001
filter = sys.argv[1] if len(sys.argv) > 1 else "10001"
socket.setsockopt(zmq.SUBSCRIBE, filter)

# Process 5 updates
total_temp = 0
for update_nbr in range (5):
    string = socket.recv()
    zipcode, temperature, relhumidity = string.split()
    total_temp += int(temperature)

print "Average temperature for zipcode '%s' was %dF" % (
      filter, total_temp / update_nbr)
```

Line: 1 Column: 1 Python Django Soft Tabs: 4



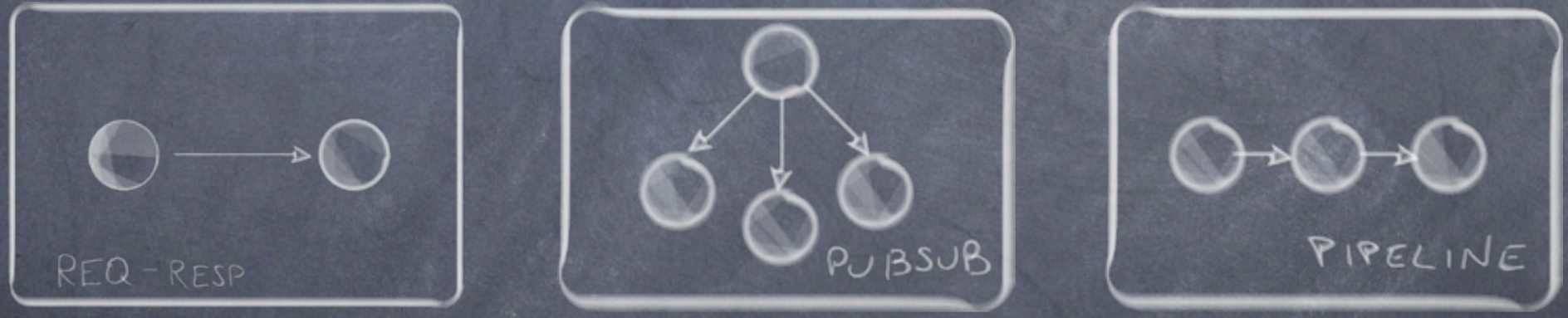
Demo

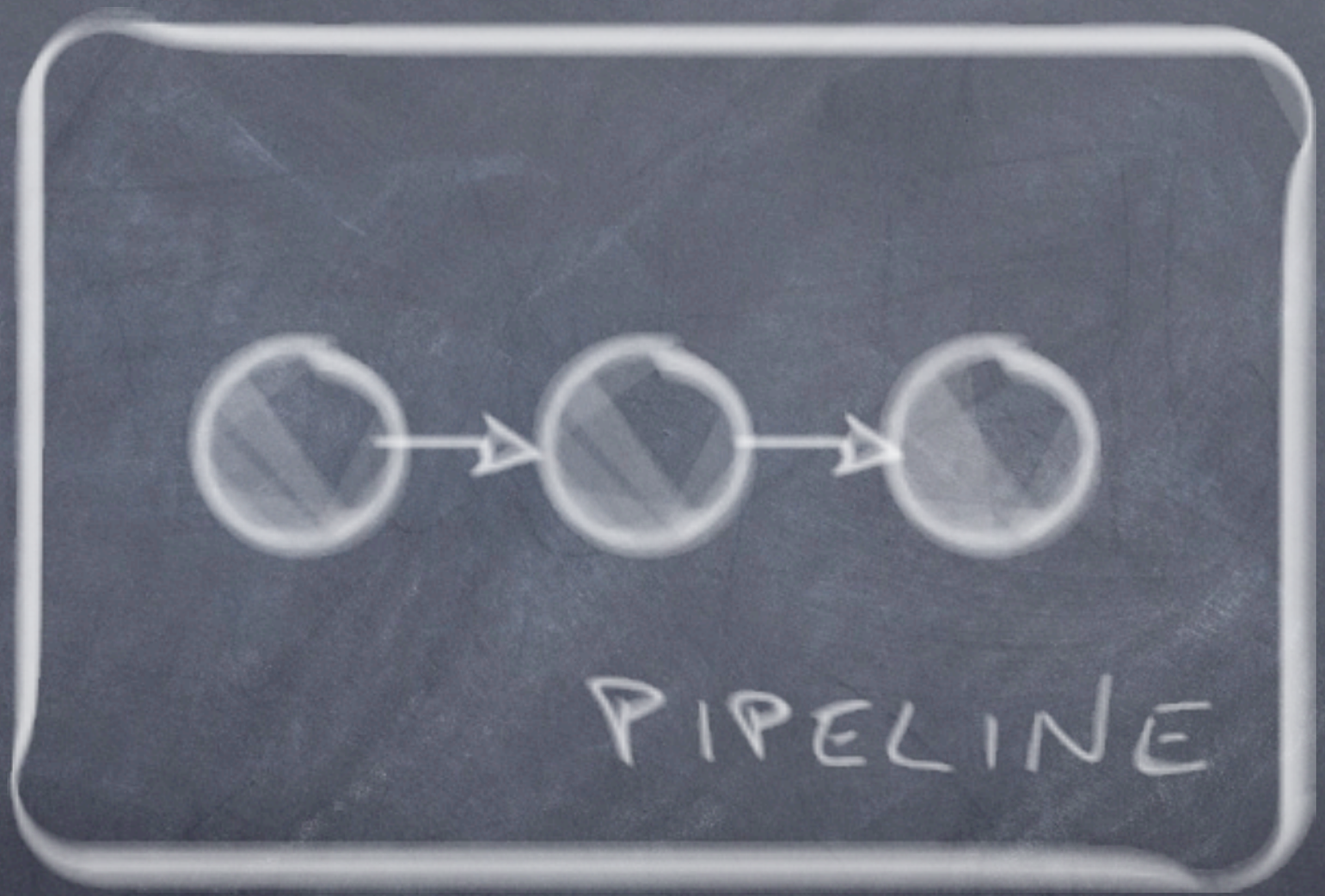


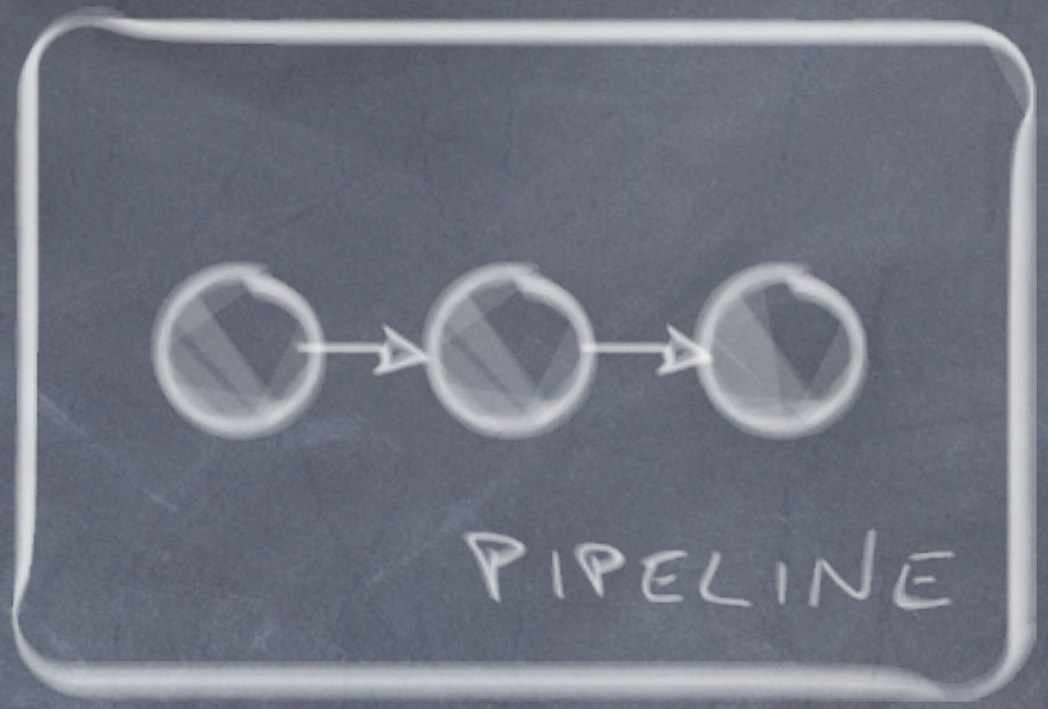
Publisher Subscriber

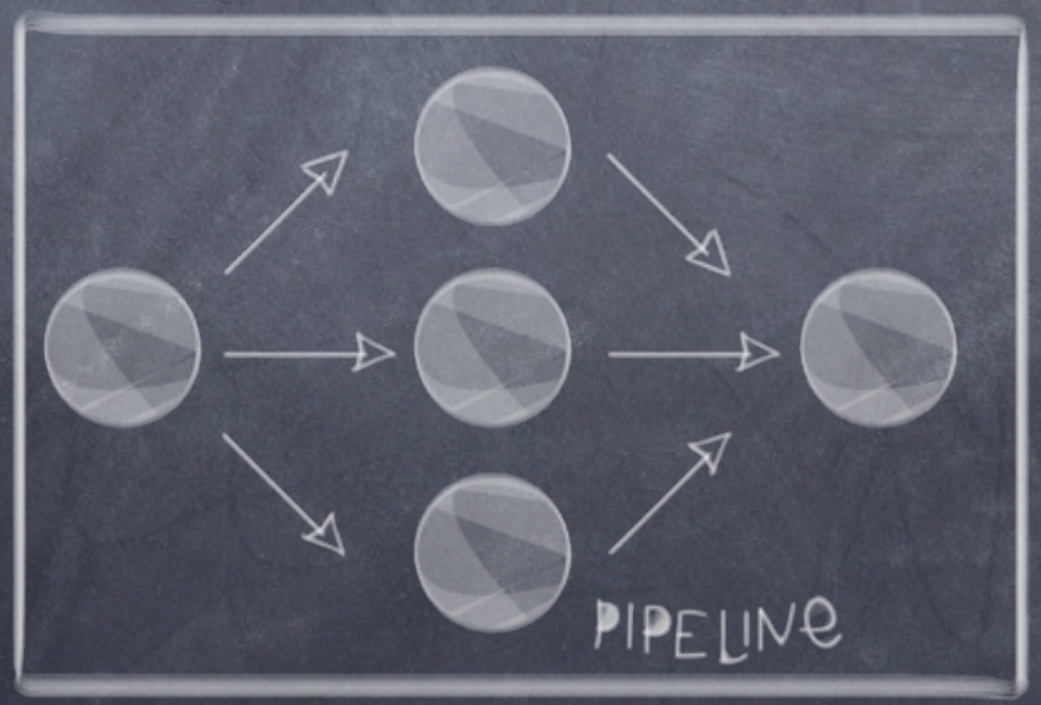
- The PUB-SUB socket pair is asynchronous
- when you use a **SUB** socket you must set a subscription using **zmq_setsockopt** and **SUBSCRIBE**
- "slow joiner" symptom

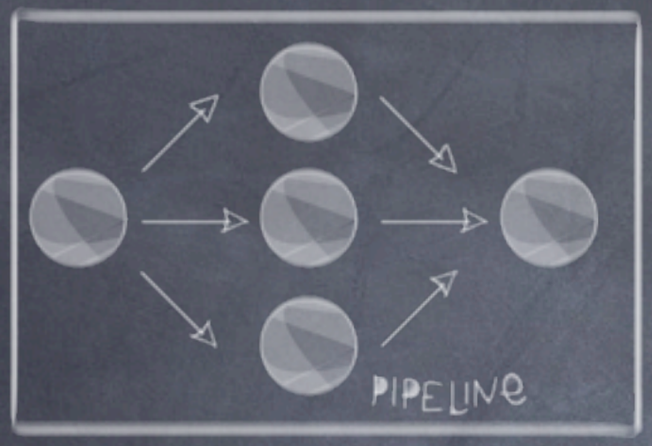
Basic Message Patterns




PIPELINE


PIPELINE


PIPELINE

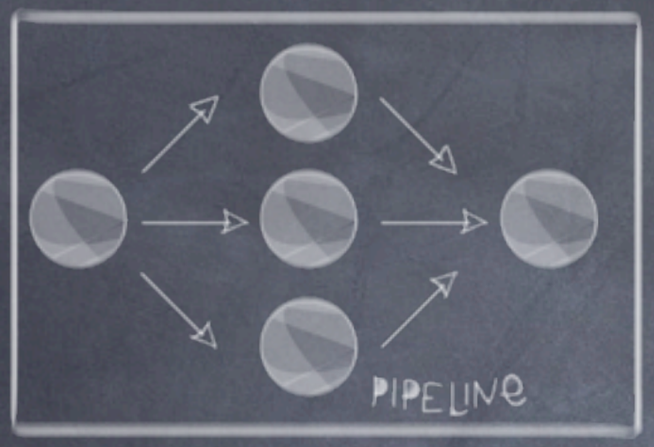


Ventilator

taskvent.py

```
# Task ventilator
# Binds PUSH socket to tcp://localhost:5557
# Sends batch of tasks to workers via that socket
#
# Author: Lev Givon <lev(at)columbia(dot)edu>

import zmq
import random
import time

context = zmq.Context()

# Socket to send messages on
sender = context.socket(zmq.PUSH)
sender.bind("tcp://*:5557")

print "Press Enter when the workers are ready: "
_ = raw_input()
print "Sending tasks to workers..."

# The first message is "0" and signals start of batch
sender.send('0')

# Initialize random number generator
random.seed()

# Send 100 tasks
total_msec = 0
for task_nbr in range(100):

    # Random workload from 1 to 100 msecs
    workload = random.randint(1, 100)
    total_msec += workload

    sender.send(str(workload))

print "Total expected cost: %s msec" % total_msec

# Give 0MQ time to deliver
time.sleep(1)
```

Line: 1 Column: 1 Python Django Soft Tabs: 4

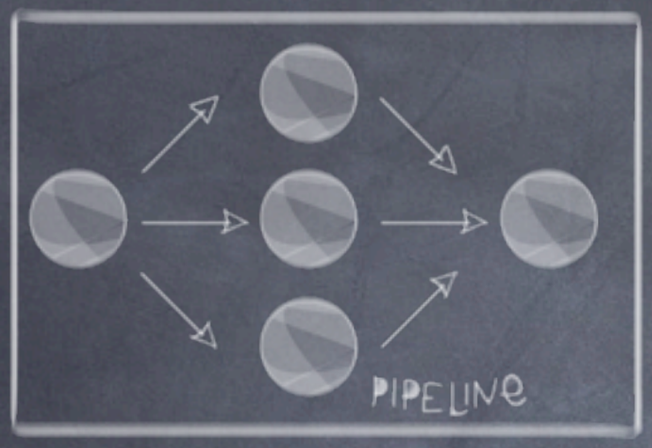


Worker

taskwork.py

```
# Task worker
# Connects PULL socket to tcp://localhost:5557
# Collects workloads from ventilator via that socket
# Connects PUSH socket to tcp://localhost:5558
# Sends results to sink via that socket
#
# Author: Lev Givon <lev(at)columbia(dot)edu>

import sys
import time
import zmq

context = zmq.Context()

# Socket to receive messages on
receiver = context.socket(zmq.PULL)
receiver.connect("tcp://localhost:5557")

# Socket to send messages to
sender = context.socket(zmq.PUSH)
sender.connect("tcp://localhost:5558")

# Process tasks forever
while True:
    s = receiver.recv()

    # Simple progress indicator for the viewer
    sys.stdout.write('.')
    sys.stdout.flush()

    # Do the work
    time.sleep(int(s)*0.001)

    # Send results to sink
    sender.send('')
```

Line: 1 Column: 1 Python Django Soft Tabs: 4

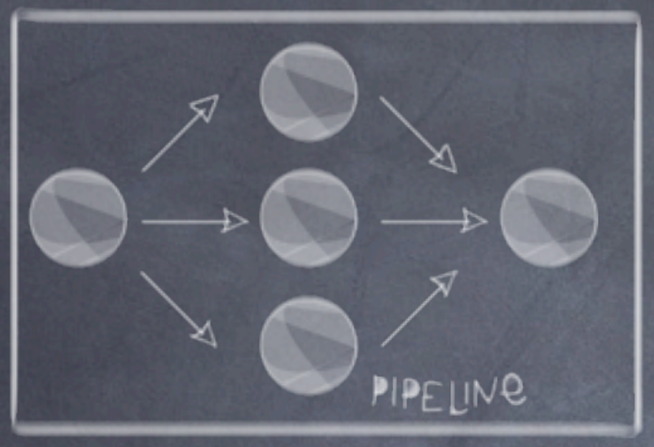


tasksink.py

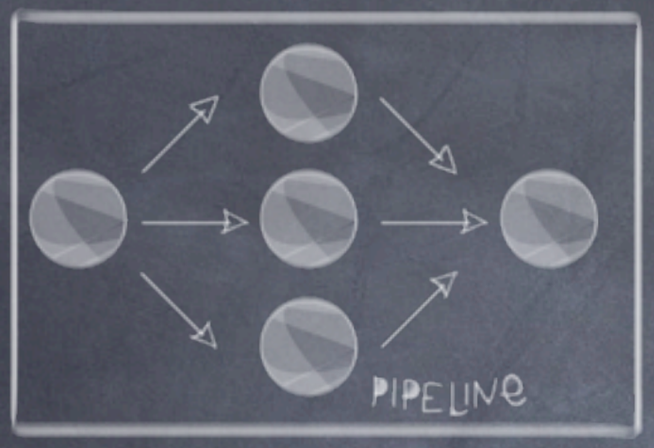
```
# Task sink
# Binds PULL socket to tcp://localhost:5558
# Collects results from workers via that socket
#
# Author: Lev Givon <lev(at)columbia(dot)edu>

import sys
import time
import zmq

context = zmq.Context()

# Socket to receive messages on
receiver = context.socket(zmq.PULL)
receiver.bind("tcp://*:5558")

# Wait for start of batch
s = receiver.recv()

# Start our clock now
tstart = time.time()

# Process 100 confirmations
total_msec = 0
for task_nbr in range(100):
    s = receiver.recv()
    if task_nbr % 10 == 0:
        sys.stdout.write(':')
    else:
        sys.stdout.write('.')

# Calculate and report duration of batch
tend = time.time()
print "Total elapsed time: %d msec" % ((tend-tstart)*1000)
```

Line: 1 Column: 1 Python Django Soft Tabs: 4

Sink

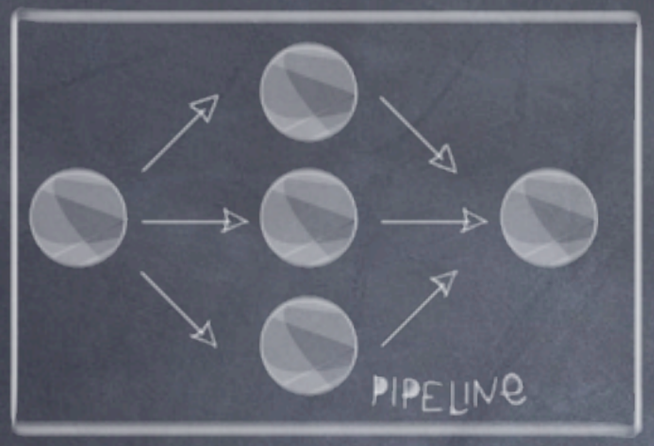


Demo

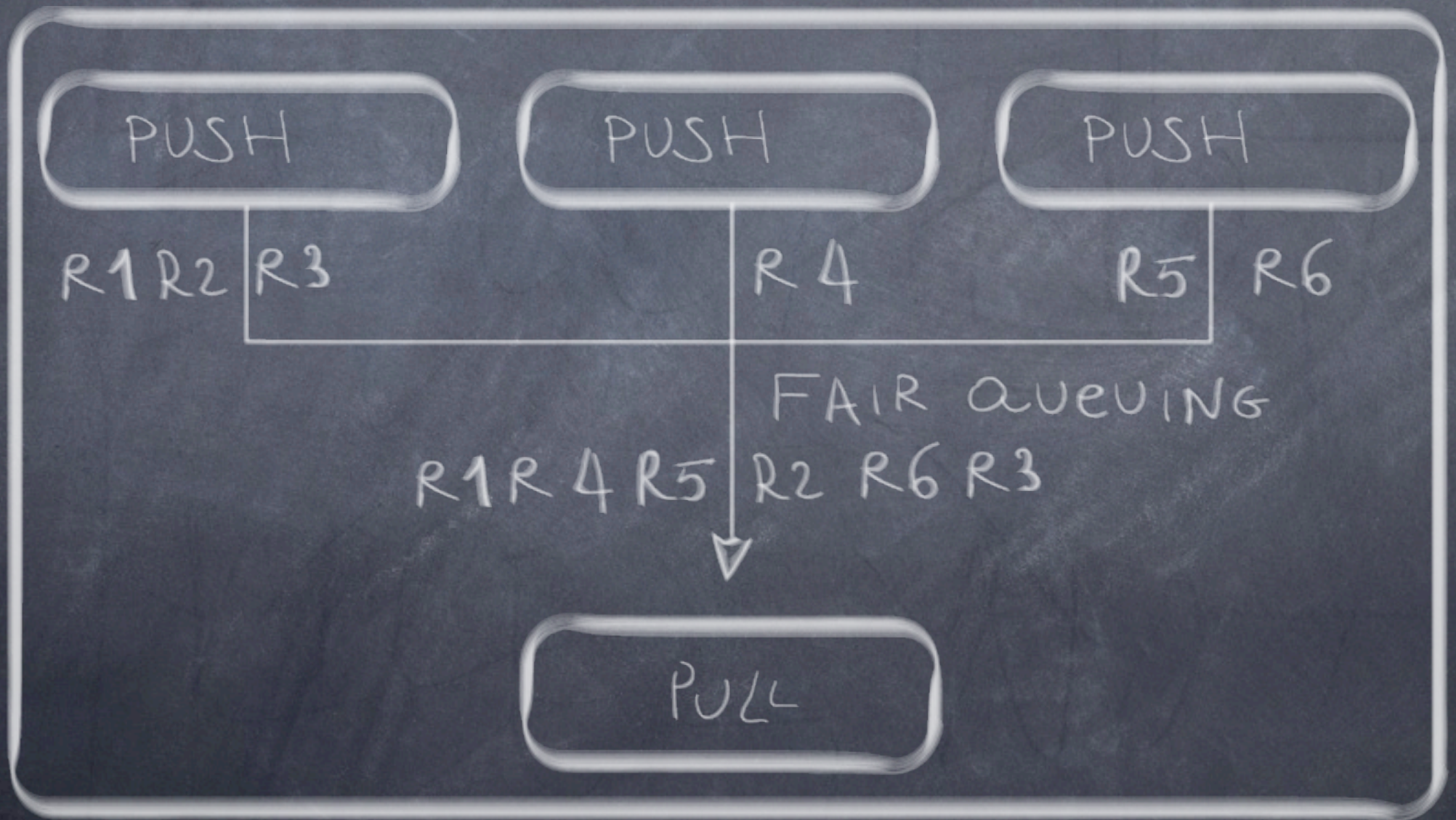


Workers

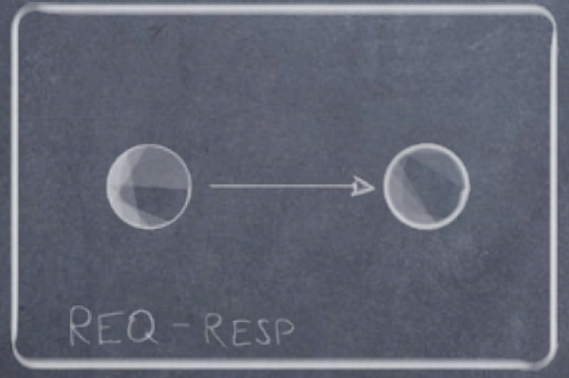
- Always synchronize the start of the batch
- The ventilator's PUSH socket distributes tasks to workers (**load balancing**)
- The sink's PULL socket collects results from workers evenly (**fair-queuing**)

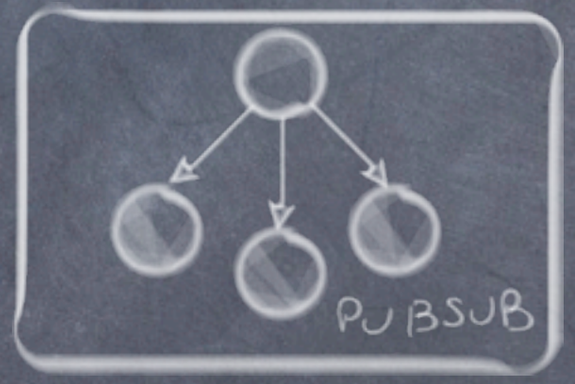


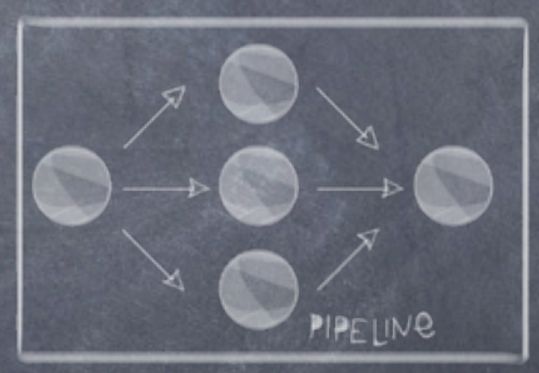
Fair-Queuing

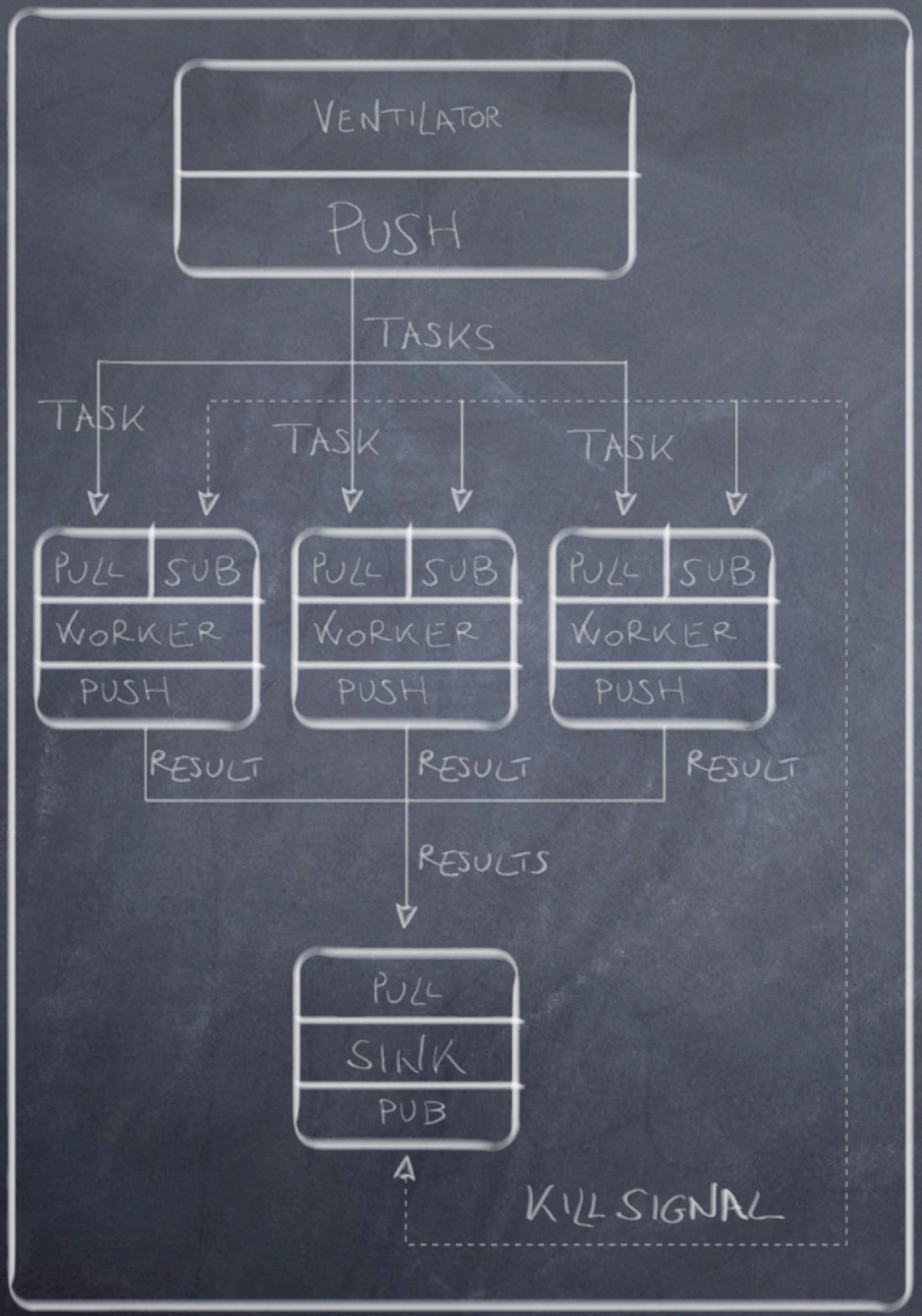


Basic Message Patterns

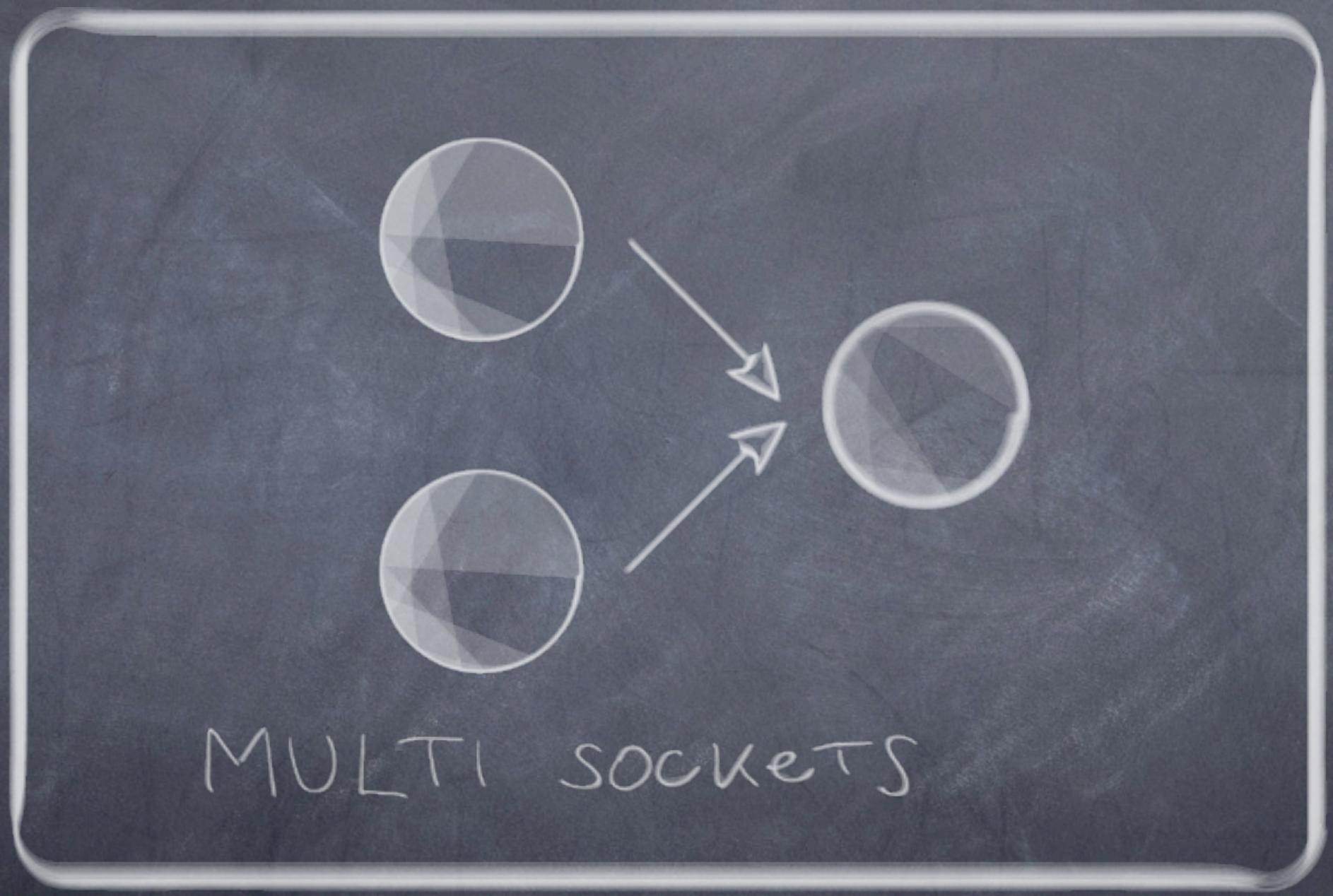


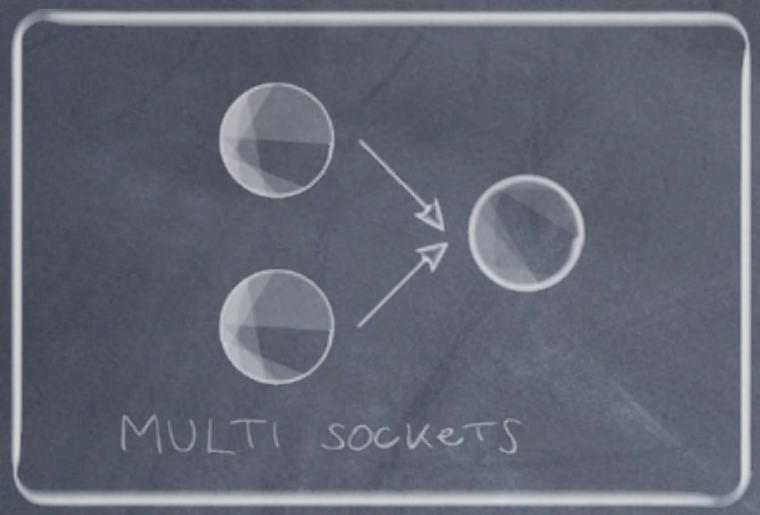






Send messages back

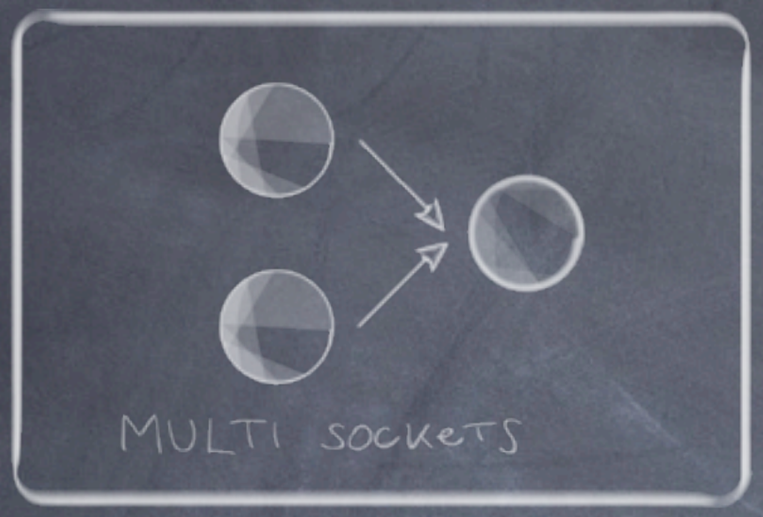

MULTI SOCKETS



mspoller.py

```
# encoding: utf-8
#
#   Reading from multiple sockets
#   This version uses zmq.Poller()
#
#   Author: Jeremy Avnet (brainsik) <spork(dash)zmq(at)theory(dot)org>
#

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()
        # process weather update
```

Line: 1 Column: 1 Python Django Soft Tabs: 4

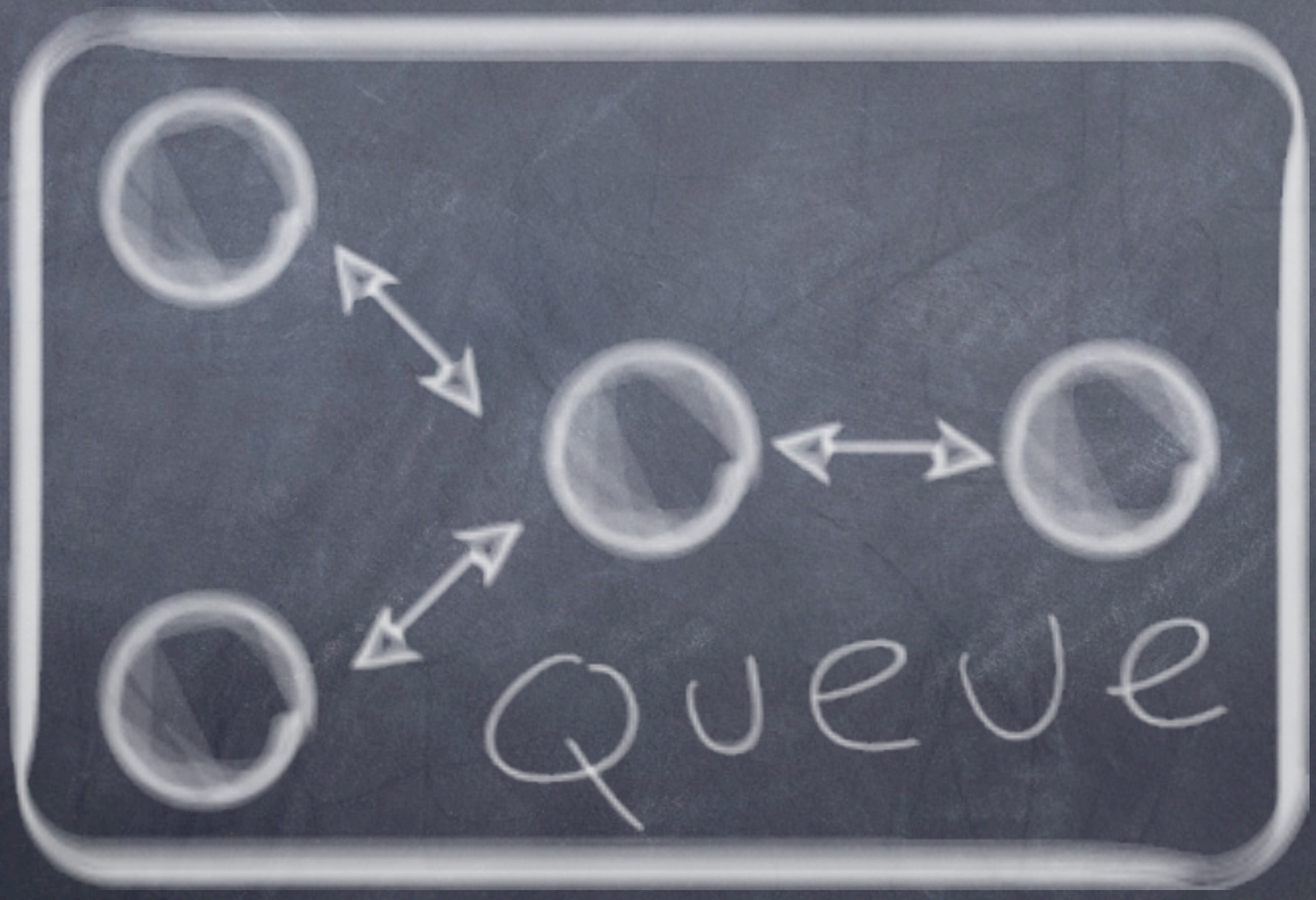


Demo

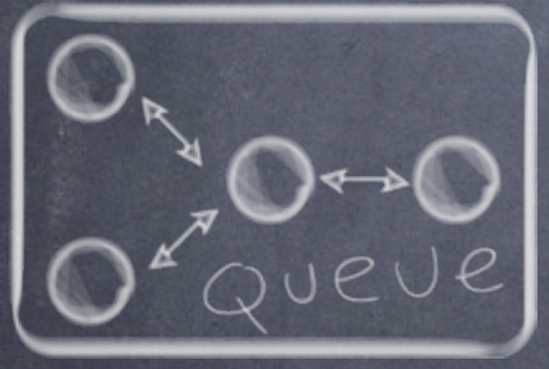
Allowed Patterns

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

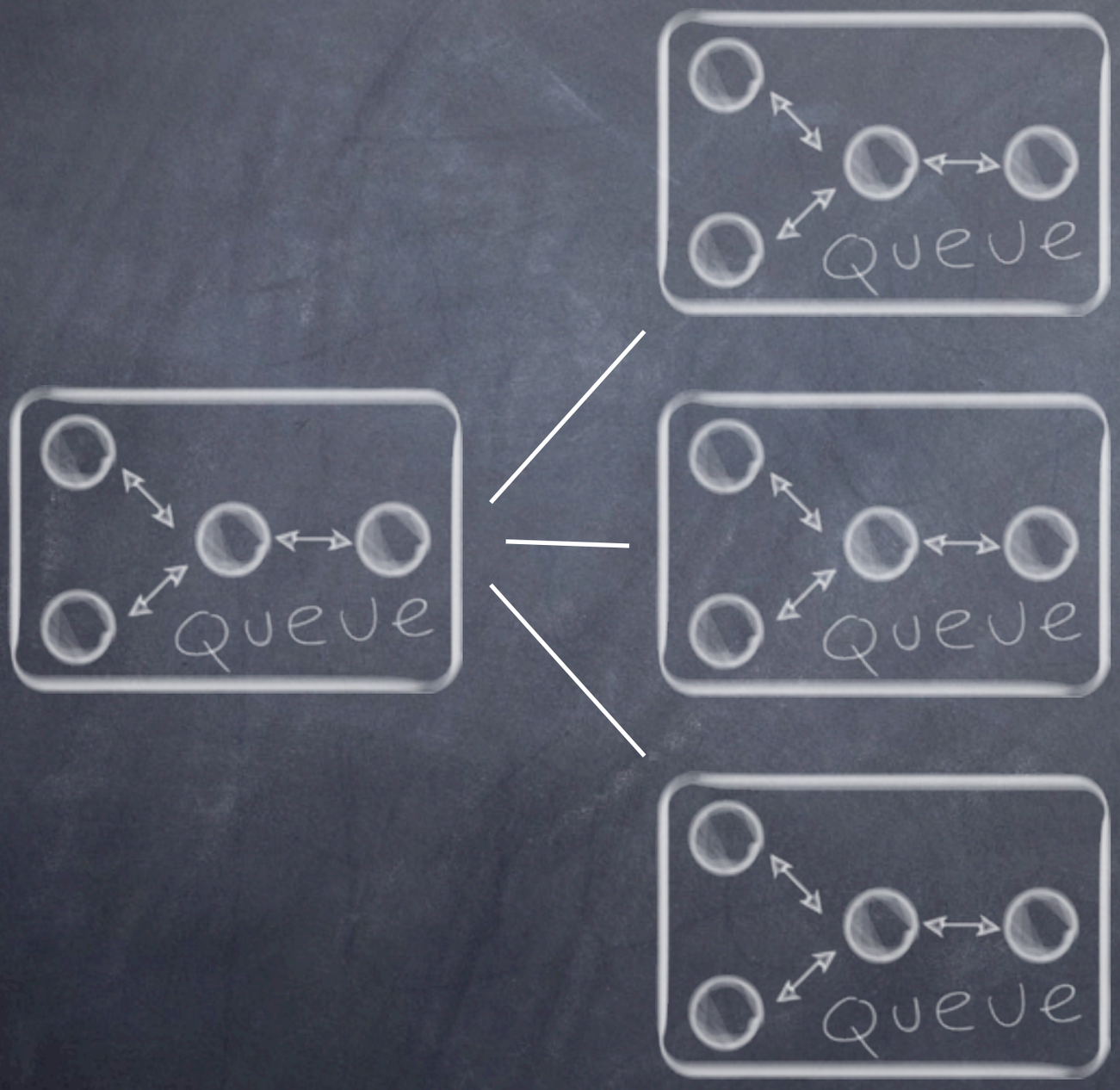
Scalability



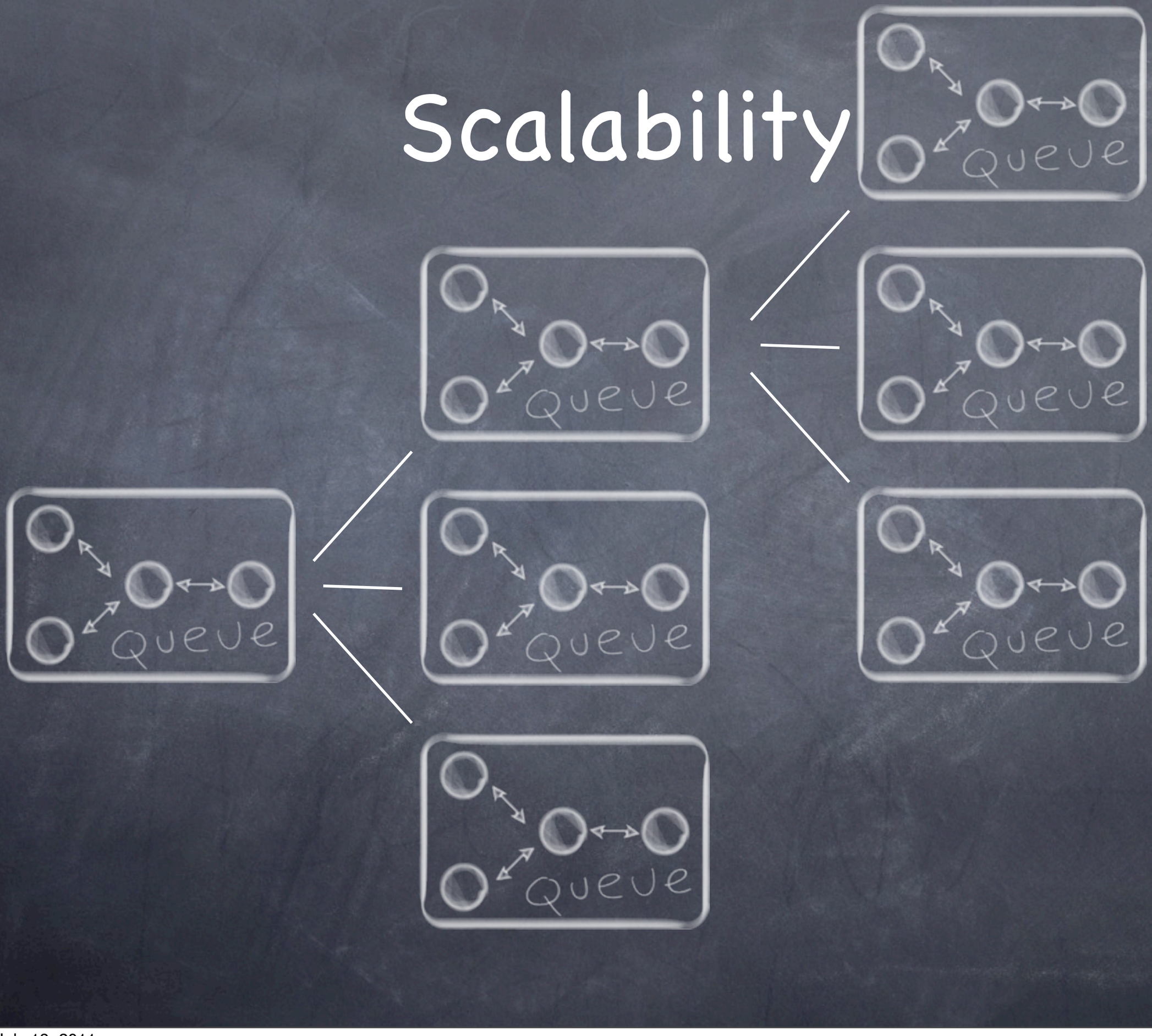
Scalability



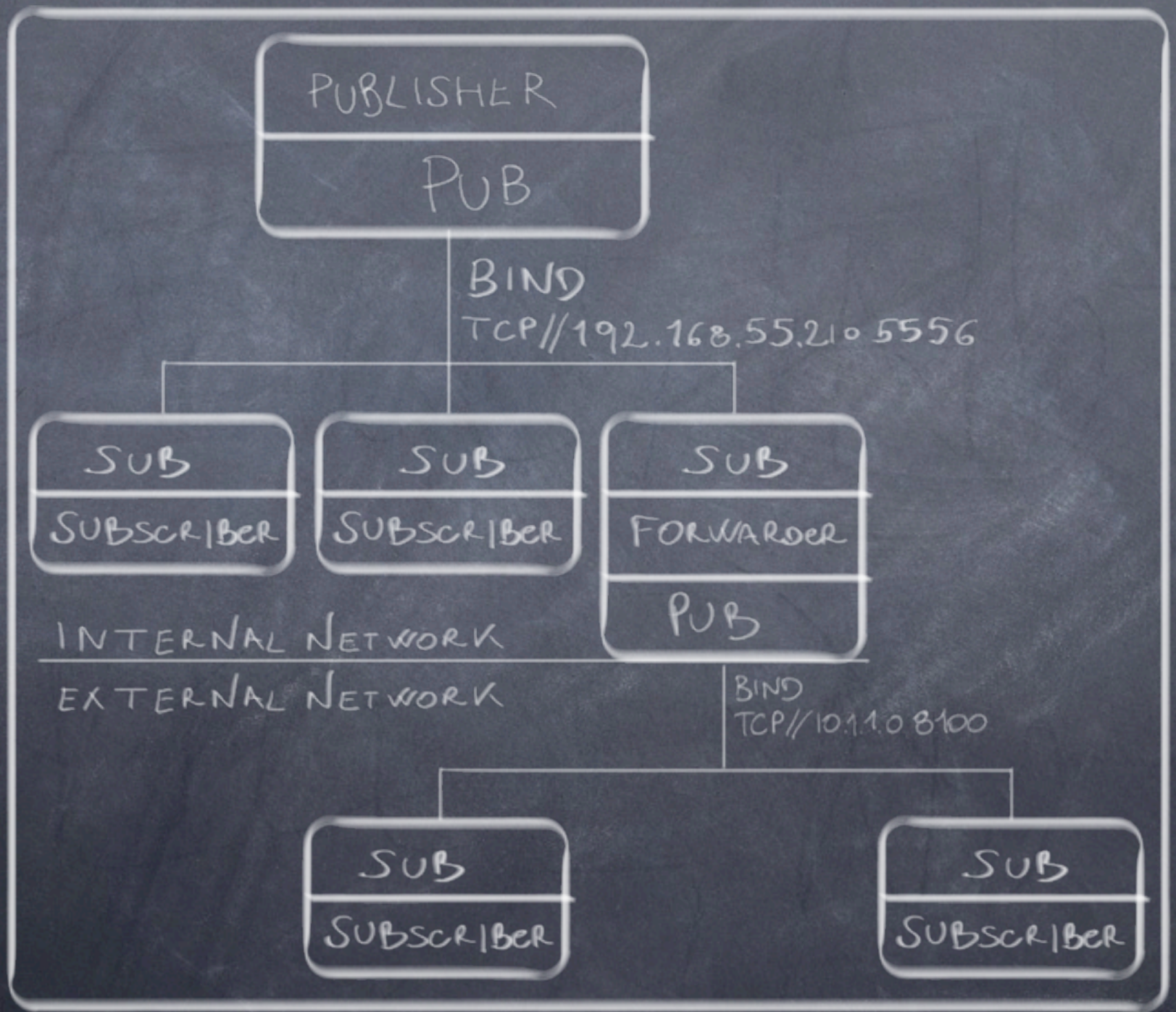
Scalability

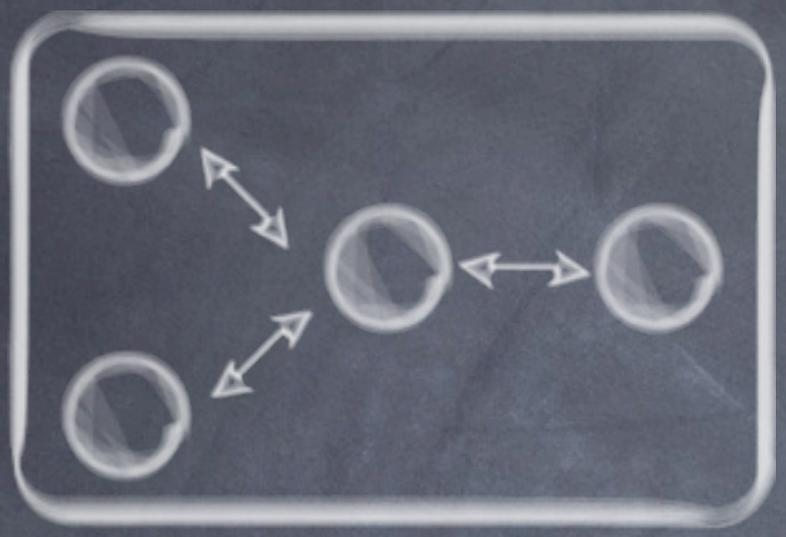


Scalability



A Publish-Subscribe Proxy





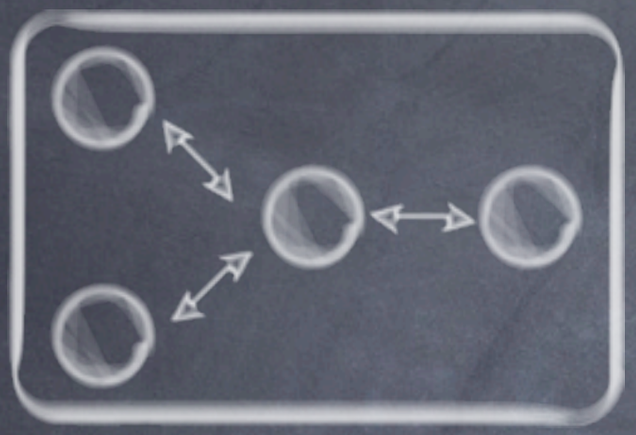
wuproxy.py

```
# Weather proxy device
#
# Author: Lev Givon <lev(at)columbia(dot)edu>

import zmq

context = zmq.Context()

# This is where the weather server sits
frontend = context.socket(zmq.SUB)
frontend.connect("tcp://192.168.55.210:5556")

# This is our public endpoint for subscribers
backend = context.socket(zmq.PUB)
backend.bind("tcp://10.1.1.0:8100")

# Subscribe on everything
frontend.setsockopt(zmq.SUBSCRIBE, '')

# Shunt messages out to our own subscribers
while True:
    while True:

        # Process all parts of the message
        message = frontend.recv()
        more = frontend.getsockopt(zmq.RCVMORE)
        if more:
            backend.send(message, zmq.SNDMORE)
        else:
            backend.send(message)
            break # Last message part
```

Line: 1 Column: 1 Python Django Soft Tabs: 4

Built-in Devices

- QUEUE (request-reply broker.)
- FORWARDER (pub-sub proxy server)
- STREAMER (like FORWARDER but for pipeline flows)

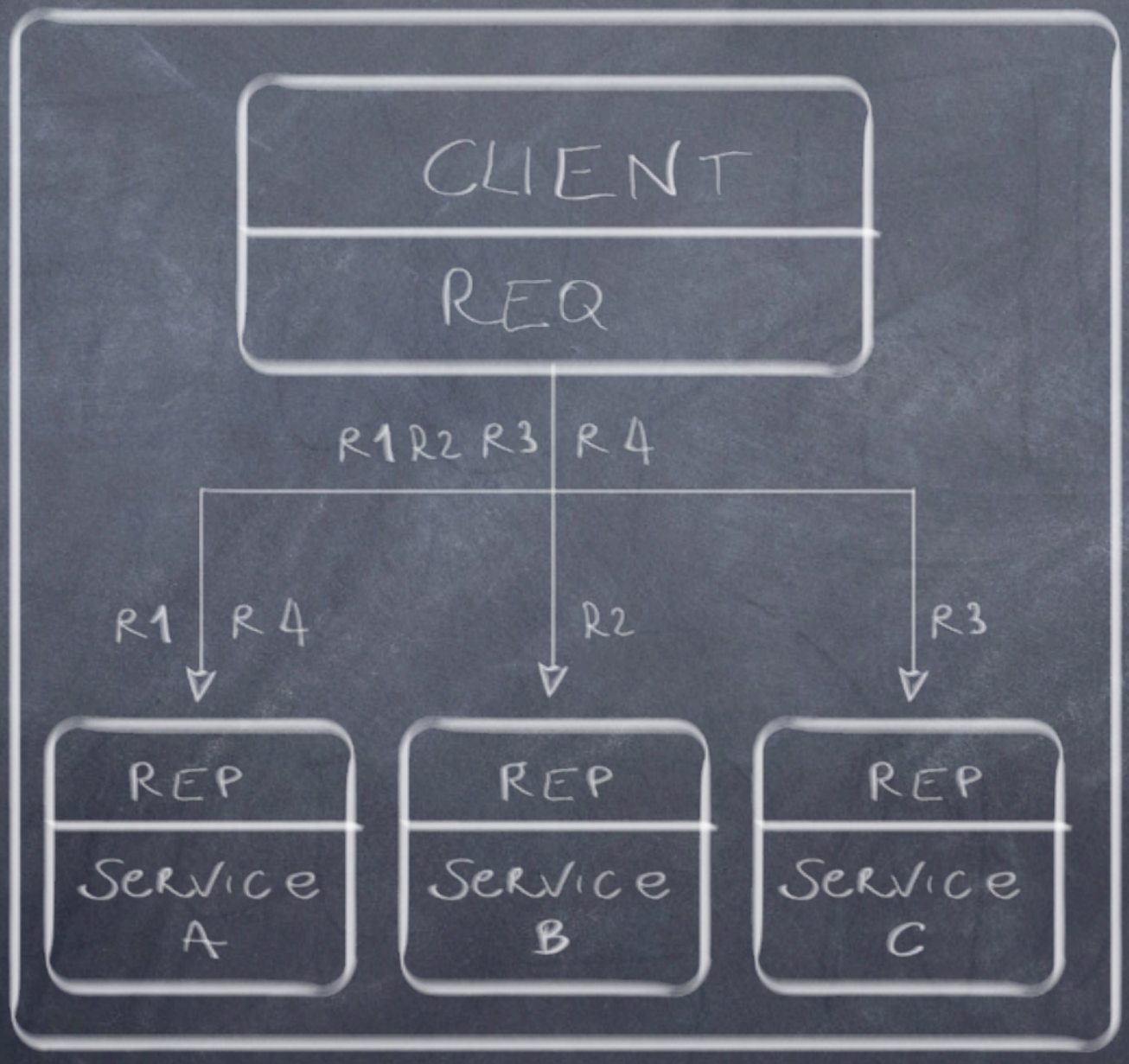


msgqueue.py

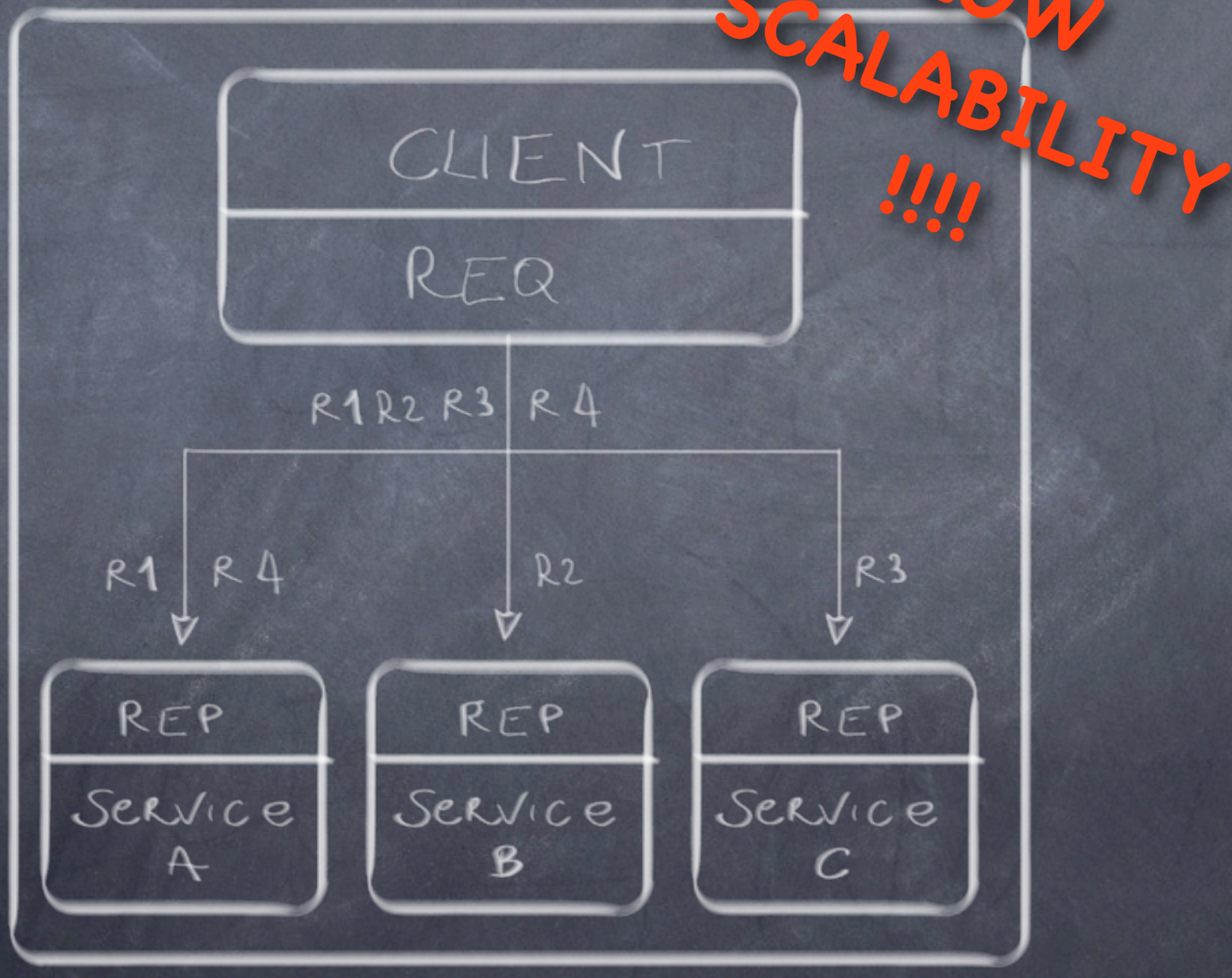
```
"""

   Simple message queuing broker
   Same as request-reply broker but using QUEUE device

   Author: Guillaume Aubert (gaubert) <guillaume(dot)aubert(at)gmail(dot)com>

"""


import zmq

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

    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)

    # We never get here...
    frontend.close()
    backend.close()
    context.term()


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

Line: 1 Column: 1 Python Django Soft Tabs: 4

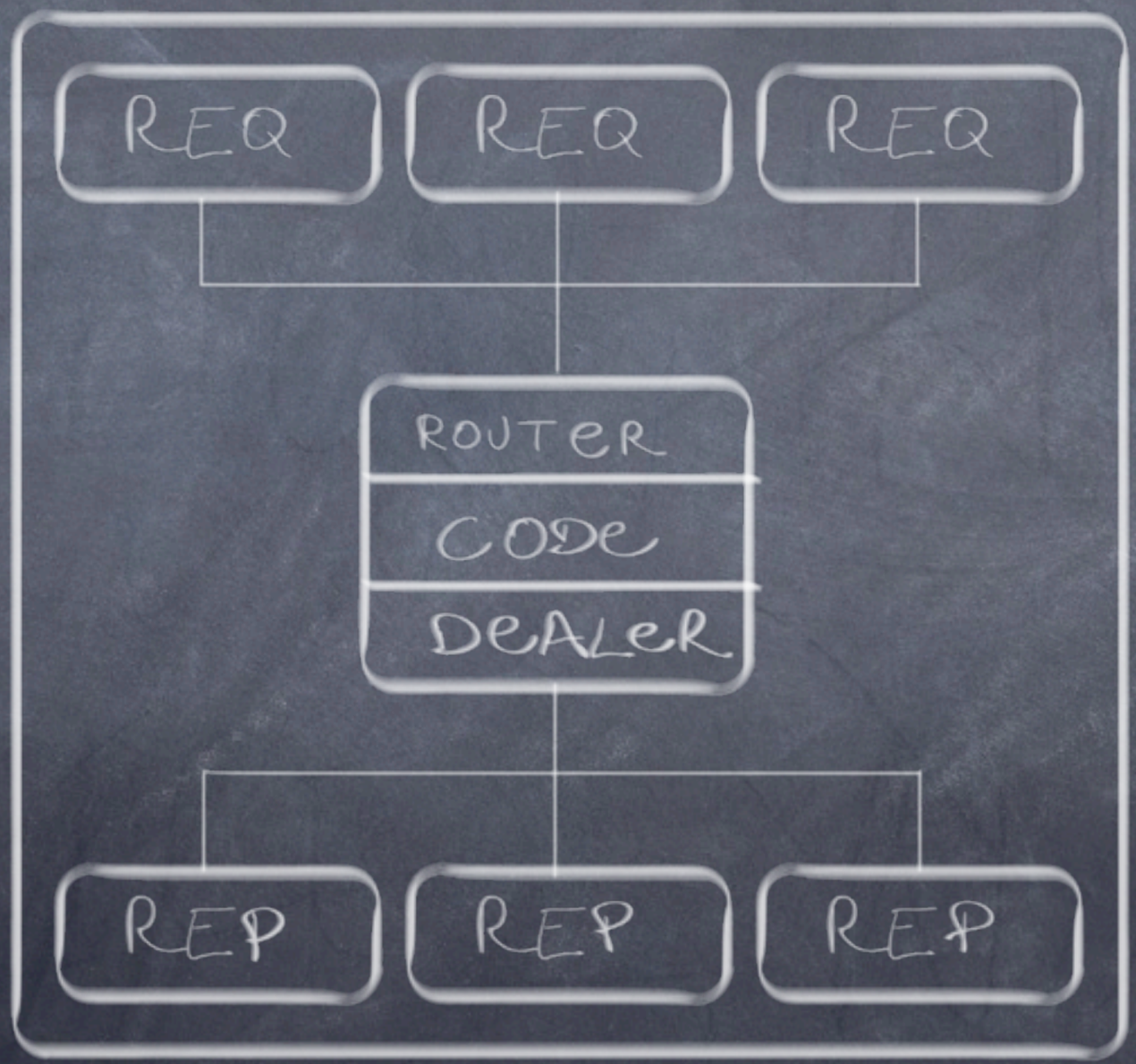
Dynamic Scalability...



Dynamic Scalability...



Dynamic Scalability...



ØMQ : Sockets = Python : C++