

zeromq 中文教程

简介:

ØMQ (ZeroMQ, 0MQ, zmq),这一堆表达方式看哪个顺眼就选哪个吧, 都指的咱要讲的这玩意儿。

它出现的目的只有一个: 更高效的利用机器。好吧, 这是我个人的看法, 官方说法是: 让任何地方、任何代码可以互联。

应该很明白吧, 如果非要做联想类比, 好吧, 可以想成经典的 C/S 模型, 这个东东封装了所有底层细节, 开发人员只要关注代码逻辑就可以了。(虽然联想成 C/S, 但可不仅仅如此哦, 具体往下看)。

它的通信协议是 AMQP,具体的 Google 之吧, 在自由市场里, 它有一个对头 RabbitMQ, 关于那只"兔子", 那又是另外一个故事了。

C/S 模式:

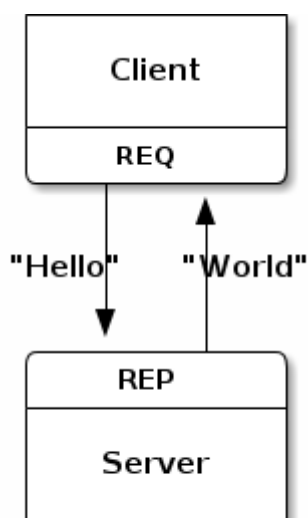


Figure 1 – Request-Reply


server

Python 代码 ☆

```
1. import zmq
2.
3. c = zmq.Context()
4. s = c.socket(zmq.REP)
5. #s.bind('tcp://127.0.0.1:10001')
6. s.bind('ipc:///tmp/zmq')
7.
8. while True:
```

```
9.     msg = s.recv_pyobj()
10.    s.send_pyobj(msg)
11. s.close()
```

client

Python 代码 

```
1. import zmq
2.
3. c = zmq.Context()
4. s = c.socket(zmq.REQ)
5. #s.connect('tcp://127.0.0.1:10001')
6. s.connect('ipc:///tmp/zmq')
7. s.send_pyobj('hello')
8. msg = s.recv_pyobj()
9. print msg
```

注意:

这个经典的模式在 zeroMQ 中是应答状态的, 不能同时 send 多个数据, 只能 ababab 这样。还有这里 send_pyobj 是 pyzmq 特有的, 用以传递 python 的对象, 通用的还是如同 socket 的 send~

pub/sub 模式:

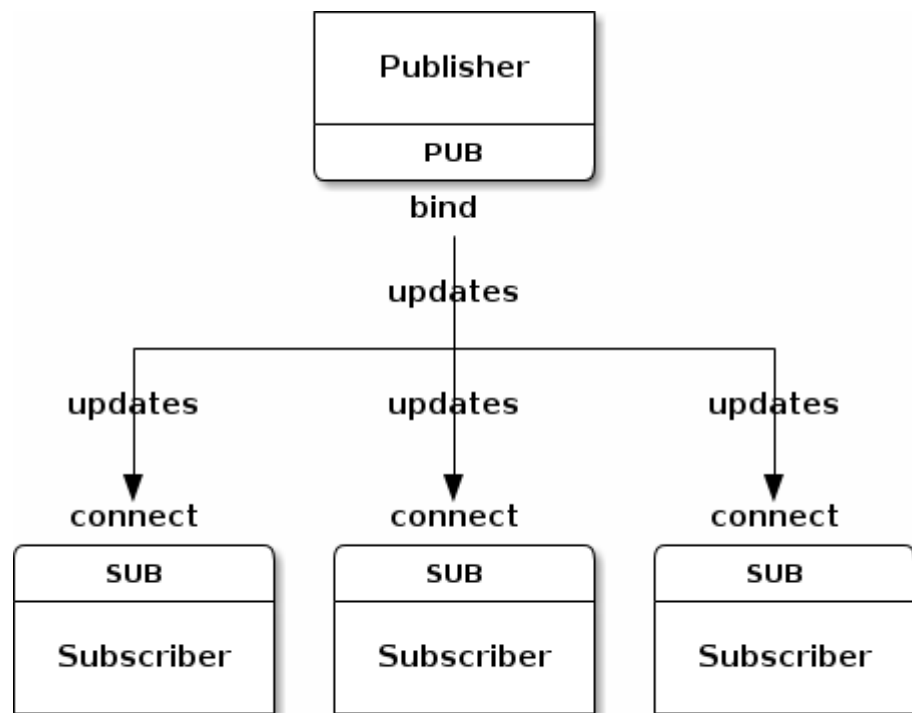


Figure 4 – Publish-Subscribe

发布端(pub)

Python 代码 ☆

```
1. import itertools
2. import sys
3. import time
4.
5. import zmq
6.
7. def main():
8.     if len(sys.argv) != 2:
9.         print 'usage: publisher <bind-to>'
10.        sys.exit(1)
11.
12.    bind_to = sys.argv[1]
13.
14.    all_topics = ['sports.general', 'sports.football', 'sports.basketbal
15.                  1',
16.                  'stocks.general', 'stocks.GOOG', 'stocks.AAPL',
17.                  'weather']
18.
19.    ctx = zmq.Context()
20.    s = ctx.socket(zmq.PUB)
```

```

20.     s.bind(bind_to)
21.
22.     print "Starting broadcast on topics:"
23.     print "    %s" % all_topics
24.     print "Hit Ctrl-C to stop broadcasting."
25.     print "Waiting so subscriber sockets can connect..."
26.     print
27.     time.sleep(1.0)
28.
29.     msg_counter = itertools.count()
30.     try:
31.         for topic in itertools.cycle(all_topics):
32.             msg_body = str(msg_counter.next())
33.             print '    Topic: %s, msg:%s' % (topic, msg_body)
34.             #s.send_multipart([topic, msg_body])
35.             s.send_pyobj([topic, msg_body])
36.             # short wait so we don't hog the cpu
37.             time.sleep(0.1)
38.     except KeyboardInterrupt:
39.         pass
40.
41.     print "Waiting for message queues to flush..."
42.     time.sleep(0.5)
43.     s.close()
44.     print "Done."
45.
46. if __name__ == "__main__":
47.     main()

```

订阅端(sub):

Python 代码 ☆

```

1. import sys
2. import time
3. import zmq
4.
5. def main():
6.     if len (sys.argv) < 2:
7.         print 'usage: subscriber <connect_to> [topic topic ...]'
8.         sys.exit (1)
9.
10.    connect_to = sys.argv[1]
11.    topics = sys.argv[2:]
12.

```

```

13.     ctx = zmq.Context()
14.     s = ctx.socket(zmq.SUB)
15.     s.connect(connect_to)
16.
17.     # manage subscriptions
18.     if not topics:
19.         print "Receiving messages on ALL topics..."
20.         s.setsockopt(zmq.SUBSCRIBE, '')
21.     else:
22.         print "Receiving messages on topics: %s ..." % topics
23.         for t in topics:
24.             s.setsockopt(zmq.SUBSCRIBE, t)
25.     print
26.     try:
27.         while True:
28.             #topic, msg = s.recv_multipart()
29.             topic, msg = s.recv_pyobj()
30.             print '    Topic: %s, msg:%s' % (topic, msg)
31.     except KeyboardInterrupt:
32.         pass
33.     print "Done."
34.
35. if __name__ == "__main__":
36.     main()

```

注意：

这里的发布与订阅角色是绝对的，即发布者无法使用 `recv`，订阅者不能使用 `send`，并且订阅者需要设置订阅条件“`setsockopt`”。

按照官网的说法，在这种模式下很可能发布者刚启动时发布的数据出现丢失，原因是用 `zmq` 发送速度太快，在订阅者尚未与发布者建立联系时，已经开始了数据发布（内部局域网没这么夸张的）。官网给了两个解决方案；1，发布者 `sleep` 一会再发送数据（这个被标注成愚蠢的）；2，（还没有看到那，在后续中发现的话会更新这里）。

官网还提供了一种可能出现的问题：当订阅者消费慢于发布，此时就会出现数据的堆积，而且还是在发布端的堆积，显然，这是不可以被接受的。至于解决方案，或许后面的“分而治之”就是吧。

push/pull 模式：

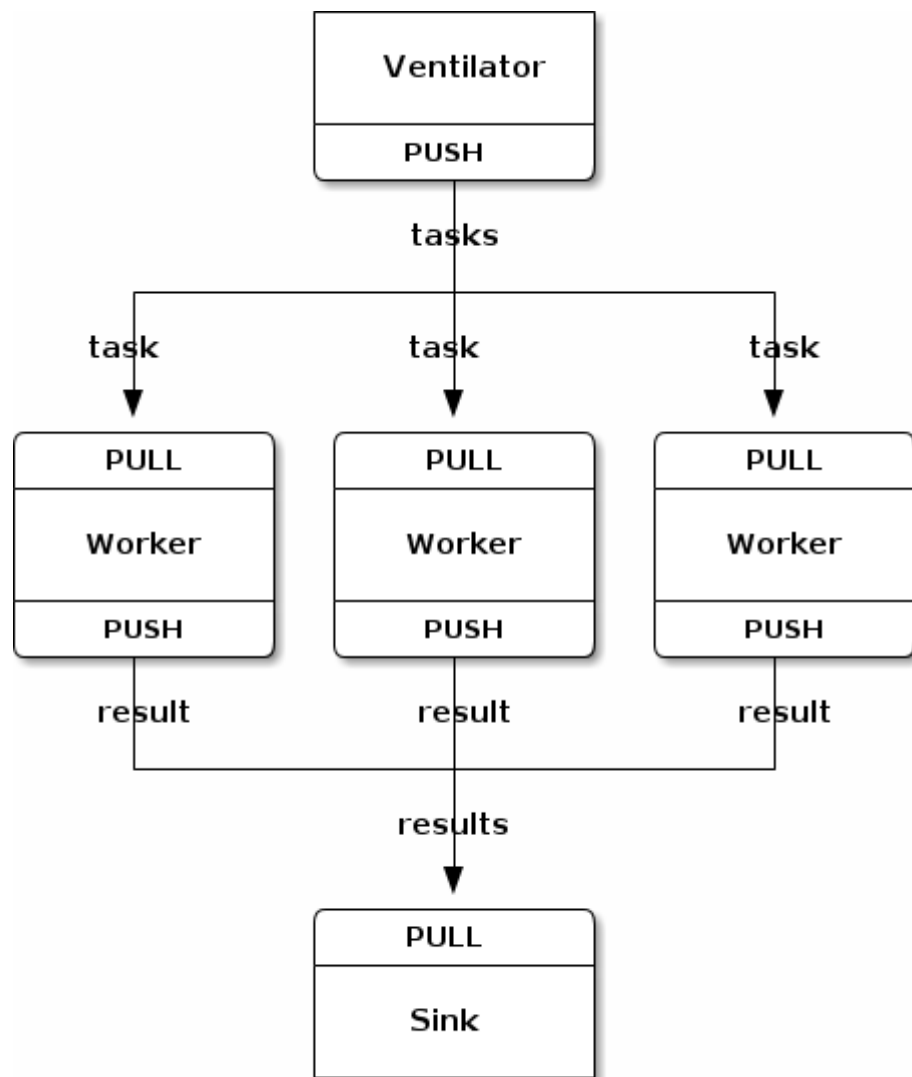


Figure 5 – Parallel Pipeline

模型描述:

- 1.上游(任务发布)
- 2.工人(中间，具体工作)
- 3.下游(信号采集或者工作结果收集)

上游代码:

Python 代码 ☆

```
1. import zmq
2. import random
3. import time
4.
5. context = zmq.Context()
6.
7. # Socket to send messages on
```

```

8. sender = context.socket(zmq.PUSH)
9. sender.bind("tcp://*:5557")
10.
11. print "Press Enter when the workers are ready: "
12. _ = raw_input()
13. print "Sending tasks to workers..."
14.
15. # The first message is "0" and signals start of batch
16. sender.send('0')
17.
18. # Initialize random number generator
19. random.seed()
20.
21. # Send 100 tasks
22. total_msec = 0
23. for task_nbr in range(100):
24.     # Random workload from 1 to 100 msec
25.     workload = random.randint(1, 100)
26.     total_msec += workload
27.     sender.send(str(workload))
28. print "Total expected cost: %s msec" % total_msec

```

工作代码:

Python 代码 ☆


```

1. import sys
2. import time
3. import zmq
4.
5. context = zmq.Context()
6.
7. # Socket to receive messages on
8. receiver = context.socket(zmq.PULL)
9. receiver.connect("tcp://localhost:5557")
10.
11. # Socket to send messages to
12. sender = context.socket(zmq.PUSH)
13. sender.connect("tcp://localhost:5558")
14.
15. # Process tasks forever
16. while True:
17.     s = receiver.recv()
18.
19.     # Simple progress indicator for the viewer

```

```
20.     sys.stdout.write('.')
21.     sys.stdout.flush()
22.
23.     # Do the work
24.     time.sleep(int(s)*0.001)
25.
26.     # Send results to sink
27.     sender.send('')
```

下游代码:

Python 代码 

```
1. import sys
2. import time
3. import zmq
4.
5. context = zmq.Context()
6.
7. # Socket to receive messages on
8. receiver = context.socket(zmq.PULL)
9. receiver.bind("tcp://*:5558")
10.
11. # Wait for start of batch
12. s = receiver.recv()
13.
14. # Start our clock now
15. tstart = time.time()
16.
17. # Process 100 confirmations
18. total_msec = 0
19. for task_nbr in range(100):
20.     s = receiver.recv()
21.     if task_nbr % 10 == 0:
22.         sys.stdout.write(':')
23.     else:
24.         sys.stdout.write('.')
25.
26. # Calculate and report duration of batch
27. tend = time.time()
28. print "Total elapsed time: %d msec" % ((tend-tstart)*1000)
```

注意点:

这种模式与 pub/sub 模式一样都是单向的，区别有两点：

- 1，该模式下在没有消费者的情况下，发布者的信息是不会消耗的(由发布者进程维护)
- 2，多个消费者消费的是同一列信息，假设 A 得到了一条信息，则 B 将不再得到

这种模式主要针对在消费者能力不够的情况下，提供的多消费者并行消费解决方案(也算是之前的 pub/sub 模式的那个“堵塞问题”的一个解决策略吧)

由上面的模型图可以看出，这是一个 N:N 的模式，在 1:N 的情况下，各消费者并不是平均消费的，而在 N:1 的情况下，则有所不同，如下图：

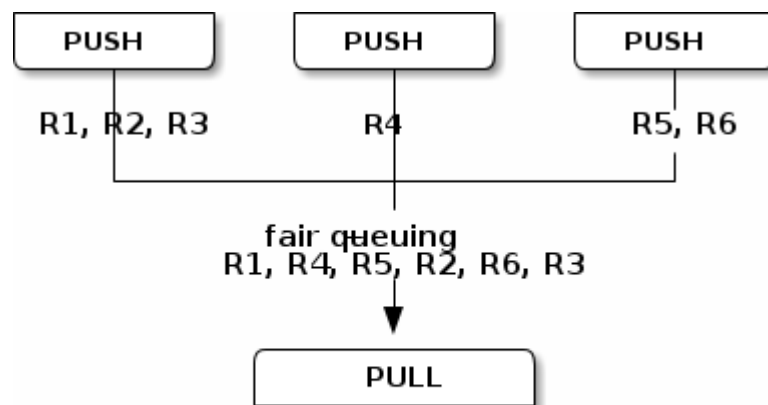


Figure 6 – Fair queuing

这种模式主要关注点在于，可以扩展中间 worker，来达到并发的目的。

既然是读书笔记，按照顺序，原作者在这里“唠嗑”了一大堆 Why，看的着实有些热血沸腾，幸而也是他的对头“兔子”的簇拥，略带客观的说：

zeroMQ 从某种层面上甚至都不能称为软件(或许连工具都称不上)。它只是一套组件，封装了复杂的网络、进程等连接环境，向上提供 API，由各个语言编写对应类库，正式运用环境是通过调用对应的类库来实现的。

从自由的角度来说，它是去中心化的，自由度极高；而从安全稳固的角度来看，则是有着无法避免的缺憾。(原本就没有完美)

额，有点偏了。

0mq 中的 0 从“零延迟”衍生为：零成本、零浪费、零管理，奉行简约主义的哲学。

你是否还在为网络编程中，为繁复的 socket 策略而纠结？甚或应此放弃或者逃避网络化设计？现在，ZeroMQ 这个新世纪网络编程的福音出现了，不再需要拘泥于底层连接的策略问题，全部交给 zeroMQ 吧，专注于你所专注，网络编程就是这么简单～

诸位在前面的例子中，已经可以发现所有的关系都是成对匹配出现的。

之前已经使用的几种模式：

req/rep(请求答复模式)：主要用于远程调用及任务分配等。

pub/sub（订阅模式）：主要用于数据分发。

push/pull(管道模式)：主要用于多任务并行。

除此之外，还有一种模式，是因为大多数人还有从"TCP"传统模式思想转变过来，习惯性尝试的独立成对模式(**1to1**).这个在后面会有介绍。

ZeroMQ 内置的有效绑定对：

- PUB and SUB
- REQ and REP
- REQ and XREP
- XREQ and REP
- XREQ and XREP
- XREQ and XREQ
- XREP and XREP
- PUSH and PULL
- PAIR and PAIR

非正常匹配会出现意料之外的问题(未必报错，但可能数据不通路什么的，官方说法是未来可能会有统一错误提示吧)，未来还会有更高层次的模式（当然也可以自己开发）。

由于 **zeroMQ** 的发送机制，发送到数据有两种状态(是否 Copy)，在非 Copy 下，一旦发送成功，发送端将不再能访问到该数据，Copy 状态则可以（主要用于重复发送）。还有就是所发送的信息都是保持在内存，故不能随意发送大数据(以防溢出)，推荐的做法是拆分逐个发送。(python 中的单条信息限制为 4M.)

—补充：

这样的发送需要额外标识 **ZMQ_SNDMORE**，在接收端可以通过 **ZMQ_RCVMORE** 来判断。

号外！

官方似乎野心勃勃啊，想将 **zeroMQ** 加入到 Linux kernel，若真做到可就了不得了。

之前已经讲过，**zeroMQ** 是可以多对多的，但需要成对匹配才行，即多个发布端都是同一种模式，而这里要涉及到的是，多个发布端模式不统一的情况。

文中先给出了一个比较"脏"的处理方式：

Python 代码 ☆

```

1. import zmq
2. import time
3.
4. context = zmq.Context()
5.
6. receiver = context.socket(zmq.PULL)
7. receiver.connect("tcp://localhost:5557")
8.
9. subscriber = context.socket(zmq.SUB)
10. subscriber.connect("tcp://localhost:5556")
11. subscriber.setsockopt(zmq.SUBSCRIBE, "10001")
12.
13. while True:
14.
15.     while True:
16.         try:
17.             rc = receiver.recv(zmq.NOBLOCK) #这是非阻塞模式
18.         except zmq.ZMQError:
19.             break
20.
21.     while True:
22.         try:
23.             rc = subscriber.recv(zmq.NOBLOCK)
24.         except zmq.ZMQError:
25.             break

```

显然，如此做既不优雅，还有出现单来源循环不止，另一来源又得不到响应的状况。

自然，官方也做了相应的封装，给了一个相对优雅的实现：

Python 代码 ☆

```

1. import zmq
2.
3. context = zmq.Context()
4.
5. receiver = context.socket(zmq.PULL)
6. receiver.connect("tcp://localhost:5557")
7.
8. subscriber = context.socket(zmq.SUB)
9. subscriber.connect("tcp://localhost:5556")
10. subscriber.setsockopt(zmq.SUBSCRIBE, "10001")
11.
12. poller = zmq.Poller()

```

```
13. poller.register(receiver, zmq.POLLIN)
14. poller.register(subscriber, zmq.POLLIN)
15.
16. while True:
17.     socks = dict(poller.poll())
18.
19.     if receiver in socks and socks[receiver] == zmq.POLLIN:
20.         message = receiver.recv()
21.
22.     if subscriber in socks and socks[subscriber] == zmq.POLLIN:
23.         message = subscriber.recv()
```

这种方式采用了平衡兼顾的原则，实现了类似于同一模式多发布端推送的“平衡队列”功能。

关掉一个进程有很多方式，而在 **ZeroMQ** 中则推崇通过使用信号通知，可控的卸载、关闭进程。在这里，要援引之前的“分而治之”例子(具体可以见[这里](#))。

例图：

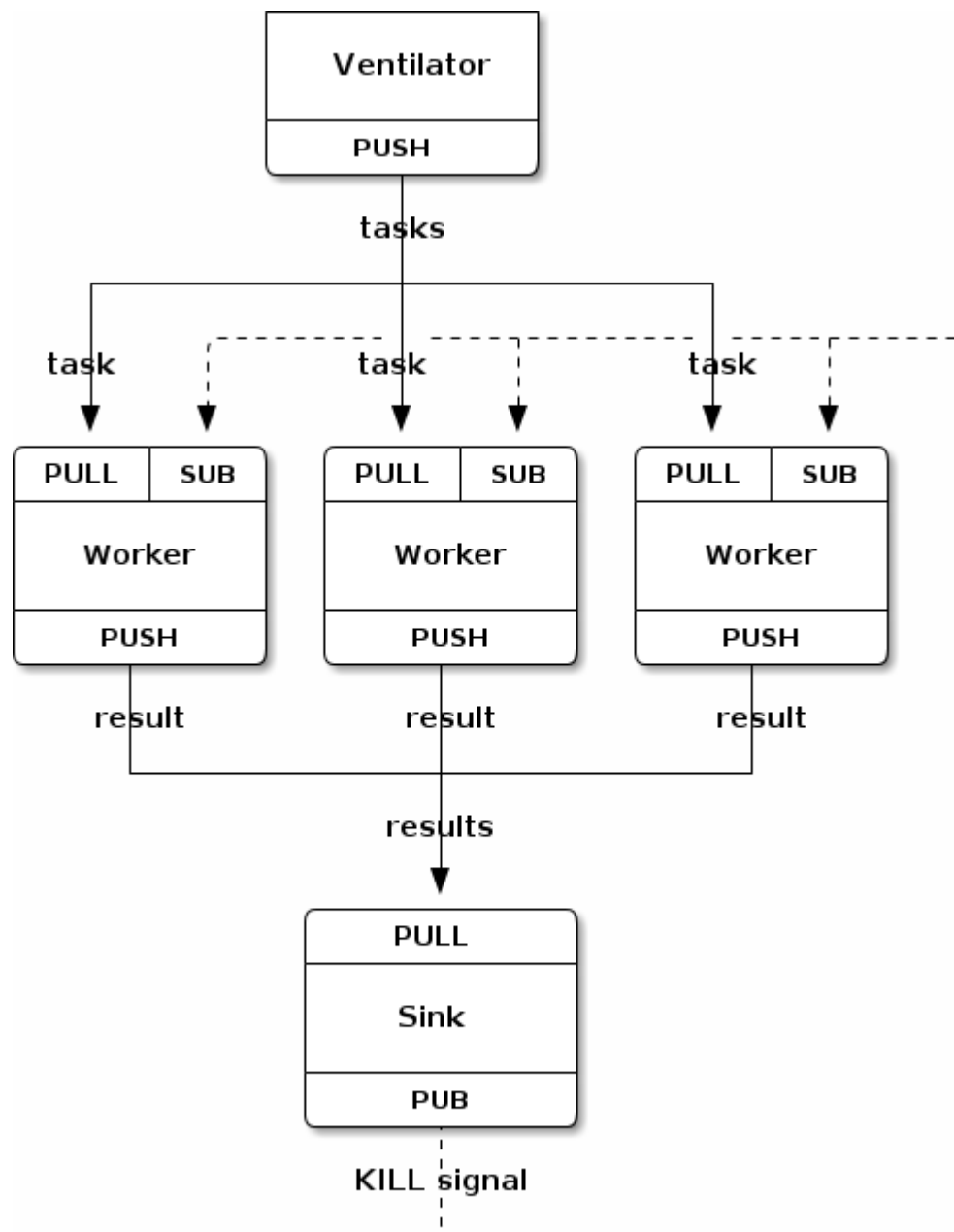


Figure 14 – Parallel Pipeline with Kill signaling

显然，信号发送是由能够掌握整个进度的"水槽"(下游)来控制，在原有基础上做少许变更即可。

Worker（数据处理）：

Python 代码 ☆

```

1. import sys
2. import time
3. import zmq
4.


```

```

5. context = zmq.Context()
6.
7. receiver = context.socket(zmq.PULL)
8. receiver.connect("tcp://localhost:5557")
9.
10. sender = context.socket(zmq.PUSH)
11. sender.connect("tcp://localhost:5558")
12.
13. controller = context.socket(zmq.SUB)
14. controller.connect("tcp://localhost:5559")
15. controller.setsockopt(zmq.SUBSCRIBE, "")
16.
17. poller = zmq.Poller()
18. poller.register(receiver, zmq.POLLIN)
19. poller.register(controller, zmq.POLLIN)
20. while True:
21.     socks = dict(poller.poll())
22.
23.     if socks.get(receiver) == zmq.POLLIN:
24.         message = receiver.recv()
25.
26.         workload = int(message) # Workload in msecs
27.         time.sleep(workload / 1000.0)
28.         sender.send(message)
29.
30.         sys.stdout.write(".")
31.         sys.stdout.flush()
32.
33.     if socks.get(controller) == zmq.POLLIN:
34.         break

```

水槽(下游):

Python 代码 

```

1. import sys
2. import time
3. import zmq
4.
5. context = zmq.Context()
6.
7. receiver = context.socket(zmq.PULL)
8. receiver.bind("tcp://*:5558")
9.
10. controller = context.socket(zmq.PUB)

```

```

11. controller.bind("tcp://*:5559")
12.
13. receiver.recv()
14.
15. tstart = time.time()
16.
17. for task_nbr in xrange(100):
18.     receiver.recv()
19.     if task_nbr % 10 == 0:
20.         sys.stdout.write(":")
21.     else:
22.         sys.stdout.write(".")
23.     sys.stdout.flush()
24.
25. tend = time.time()
26. tdiff = tend - tstart
27. total_msec = tdiff * 1000
28. print "Total elapsed time: %d msec" % total_msec
29.
30. controller.send("KILL")
31. time.sleep(1)

```

注意:

正常情况下, 即使进程被关闭, 可能端口并没有被清除(那是有 ZeroMQ 维护的), 原文中调用了这么两句

zmq_close (server)

zmq_term (context)

python 中对应为 `zmq.close()`, `zmq.term()`, 不过 python 的垃圾回收会替俺们解决后顾之忧的~

写过"永不停歇"的代码的兄弟应该都或多或少遇到或考虑到内存溢出之类的问题, 那么, 在 ZeroMQ 的应用中, 又如何处理如是情况?

文中给出了类 C 这种需要自行管理内存的解决方案(虽然 python 的 GC 很强大, 不过, 关注下总没有坏处):

这里运用到了这个工具: **valgrind**

为了避免 **zeromq** 中的一些 **warning** 的干扰, 首先需要重新 **build** 下 **zermq**

- \$ cd zeromq
- \$ export CPPFLAGS=-DZMQ_MAKE_VALGRIND_HAPPY
- \$./configure
- \$ make clean; make

- `$ sudo make install`

然后：

```
valgrind --tool=memcheck --leak-check=full someprog
```

由此帮助，通过修正代码，应该可以得到如下令人愉快的信息：

```
==30536== ERROR SUMMARY: 0 errors from 0 contexts...
```

似乎这是技巧章了，与 ZeroMQ 关联度不是太大啊，读书笔记嘛，书上写了，就记录下，学习下。

前面所谈到的网络拓扑结构都是这样的：

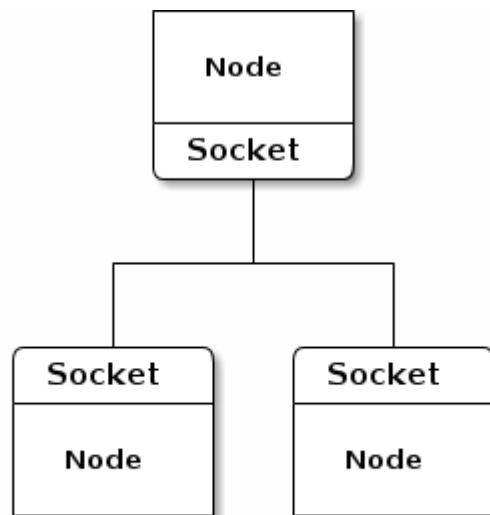


Figure 15 – Small scale ØMQ application

而在实际的应用中，绝大多数会出现这样的结构要求：

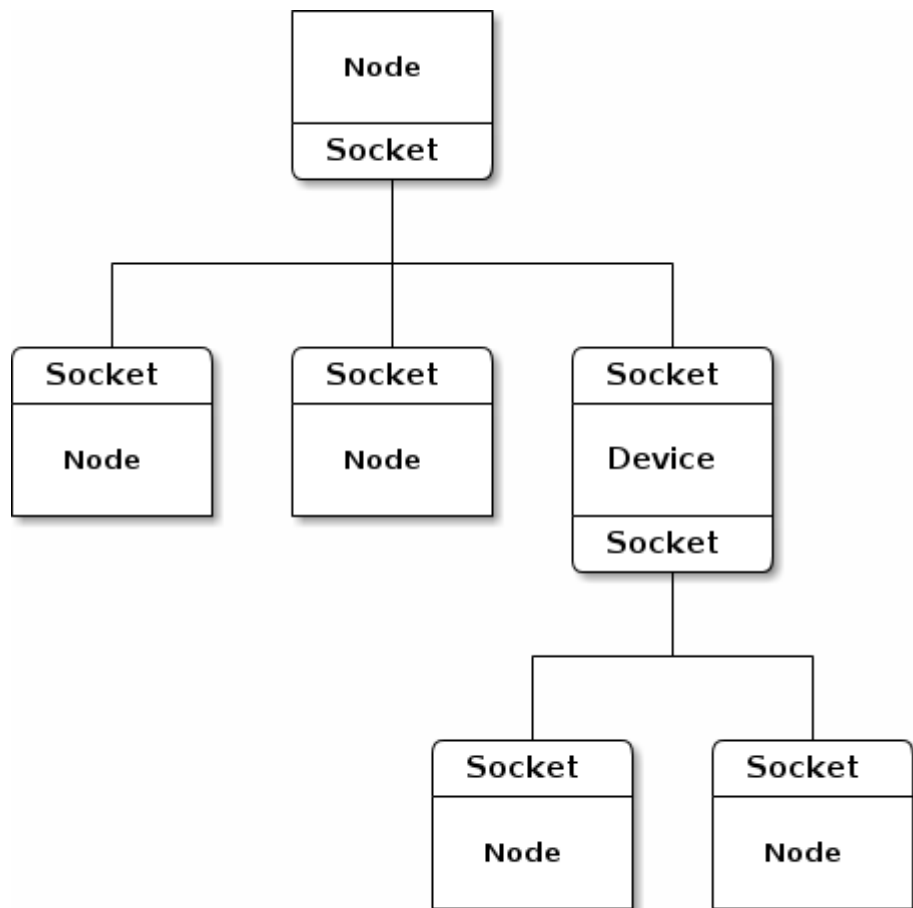


Figure 16 – Larger scale ØMQ application

zeroMQ 中自然也提供了这样的需求案例：

1.发布/订阅 代理模式：

Python 代码 ☆

```
1. import zmq
2.
3. context = zmq.Context()
4.
5. frontend = context.socket(zmq.SUB)
6. frontend.connect("tcp://192.168.55.210:5556")
7.
8. backend = context.socket(zmq.PUB)
9. backend.bind("tcp://10.1.1.0:8100")
10.
11. frontend.setsockopt(zmq.SUBSCRIBE, '')
12.
13. while True:
14.     while True:
```

```

15.     message = frontend.recv()
16.     more = frontend.getsockopt(zmq.RCVMORE)
17.     if more:
18.         backend.send(message, zmq.SNDMORE)
19.     else:
20.         backend.send(message)
21.         break # Last message part

```

注意代码，这个代理是支持大数据多包发送的。这个 proxy 实现了下图：

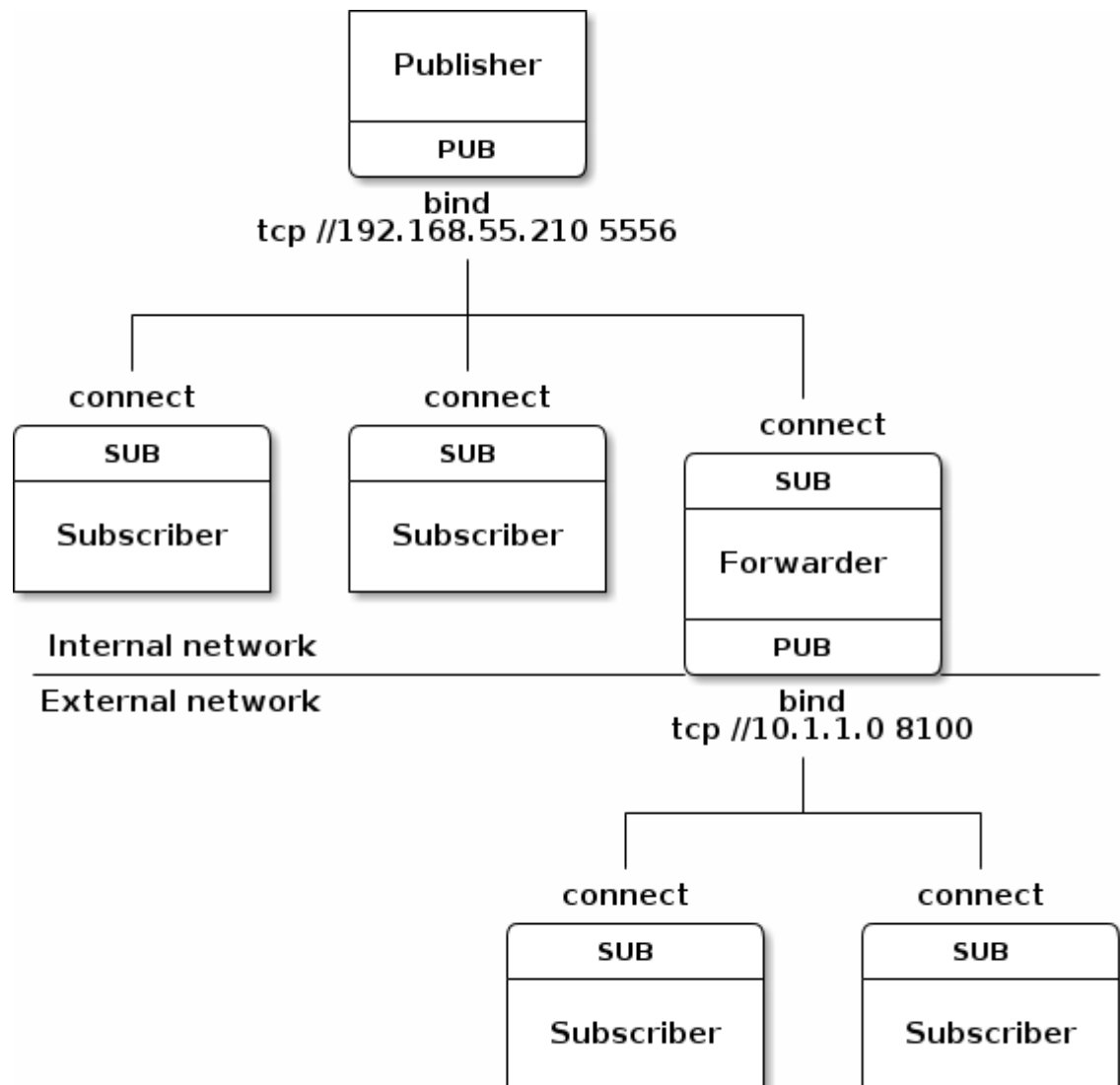


Figure 17 – Forwarder proxy device

2.请求/答复 代理模式：

因为 zeroMQ 天然支持"多对多",所以看似不需要代理啊，如下图：

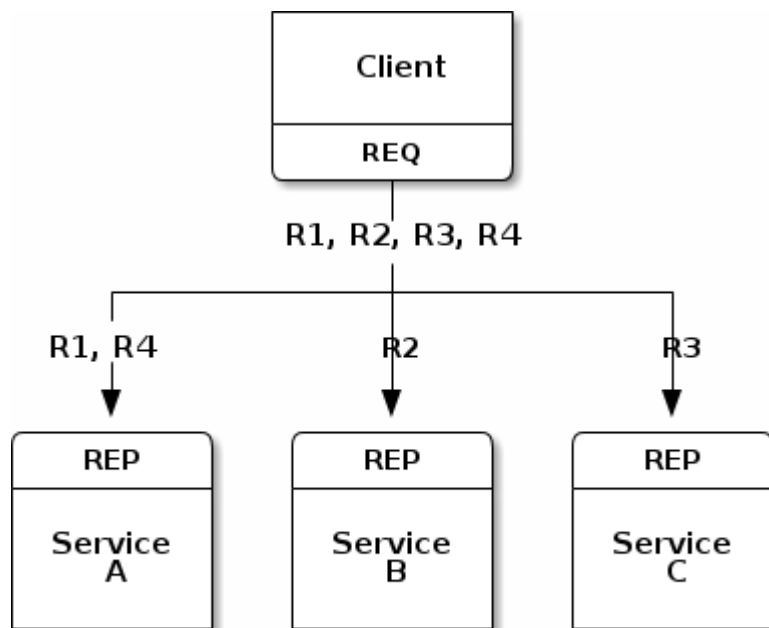



Figure 18 – Load balancing of requests

不过，这样会有一个问题，客户端需要知道所有的服务地址，并且在服务地址出现变迁时，需要通知客户端，这样迁移扩展的复杂度将无法预估。故，需要实现下图：




客户端：

Python 代码 

```
1. import zmq
2.
3. context = zmq.Context()
4. socket = context.socket(zmq.REQ)
5. socket.connect("tcp://localhost:5559")
6.
7. for request in range(1,10):
8.     socket.send("Hello")
9.     message = socket.recv()
10.     print "Received reply ", request, "[", message, "]"
```

服务器端：

Python 代码 

```
1. import zmq
2.
3. context = zmq.Context()
```

```
4.socket = context.socket(zmq.REP)
5.socket.connect("tcp://localhost:5560")
6.
7.while True:
8.    message = socket.recv()
9.    print "Received request: ", message
10.    socket.send("World")
```

代理端:

Python 代码 ☆

```
1.import zmq
2.
3.context = zmq.Context()
4.frontend = context.socket(zmq.XREP)
5.backend = context.socket(zmq.XREQ)
6.frontend.bind("tcp://*:5559")
7.backend.bind("tcp://*:5560")
8.
9.poller = zmq.Poller()
10.poller.register(frontend, zmq.POLLIN)
11.poller.register(backend, zmq.POLLIN)
12.
13.while True:
14.    socks = dict(poller.poll())
15.
16.    if socks.get(frontend) == zmq.POLLIN:
17.        message = frontend.recv()
18.        more = frontend.getsockopt(zmq.RCVMORE)
19.        if more:
20.            backend.send(message, zmq.SNDMORE)
21.        else:
22.            backend.send(message)
23.
24.    if socks.get(backend) == zmq.POLLIN:
25.        message = backend.recv()
26.        more = backend.getsockopt(zmq.RCVMORE)
27.        if more:
28.            frontend.send(message, zmq.SNDMORE)
29.        else:
30.            frontend.send(message)
```

上面的代码组成了下面的网络结构:

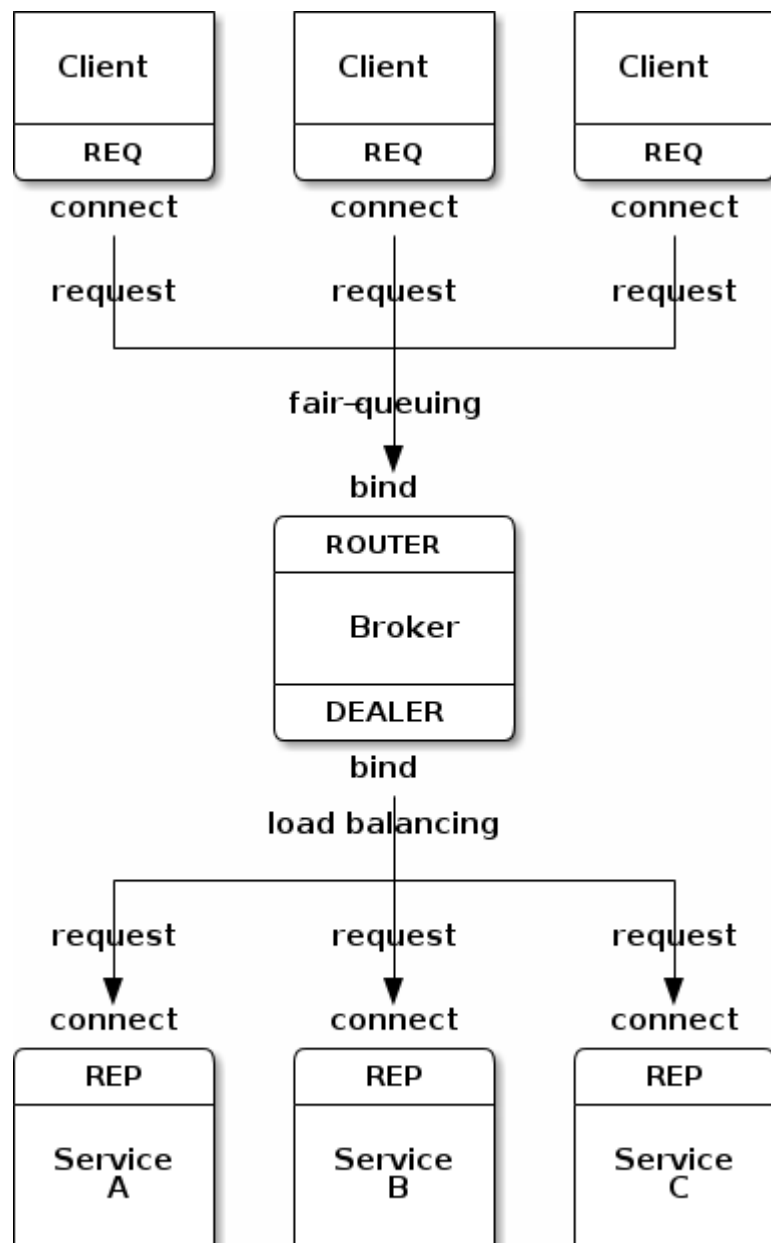


Figure 20 – Request reply broker

客户端与服务器端互相透明，世界一下清净了...

这节上了好多图和代码，绝对都是干货。不过，既然 0mq 已经想到了，为毛还要咱自己写代理捏？So,虽然上面的都是干货，或许，马上，就可以统统忘掉了。下面，展示下 0mq 自带的代理方案：

Python 代码 ☆

```
1. import zmq
2.
3. def main():
4.     context = zmq.Context(1)
```

```
5.
6.     frontend = context.socket(zmq.XREP)
7.     frontend.bind("tcp://*:5559")
8.
9.     backend = context.socket(zmq.XREQ)
10.    backend.bind("tcp://*:5560")
11.
12.    zmq.device(zmq.QUEUE, frontend, backend)
13.
14.    frontend.close()
15.    backend.close()
16.    context.term()
17.
18. if __name__ == "__main__":
19.     main()
```

这是应答模式的代理，官方提供了三种标准代理：

应答模式：queue XREP/XREQ

订阅模式：forwarder SUB/PUB

分包模式：streamer PULL/PUSH

特别提醒：

官方可不推荐代理混搭，不然责任自负。按照官方的说法，既然要混搭，还是自个儿写代理比较靠谱～

"或许，ZeroMQ 是最好的多线程运行环境！"官网如是说。

其实它想要支持的是那种类似 **erlang** 信号模式。传统多线程总会伴随各种"锁"出现各种稀奇古怪的问题。而 **zeroMQ** 的多线程致力于"去锁化"，简单来说，一条数据在同一时刻只允许被一个线程持有(而传统的是：只允许被一个线程操作)。而锁，是因为可能会出现的多线程同时操作一条数据才出现的副产品。从这里就可以很清晰的看出 **zeromq** 的切入点了，通过线程间的数据流动来保证同一时刻任何数据都只会被一个线程持有。

这里给出传统的应答模式的例子：

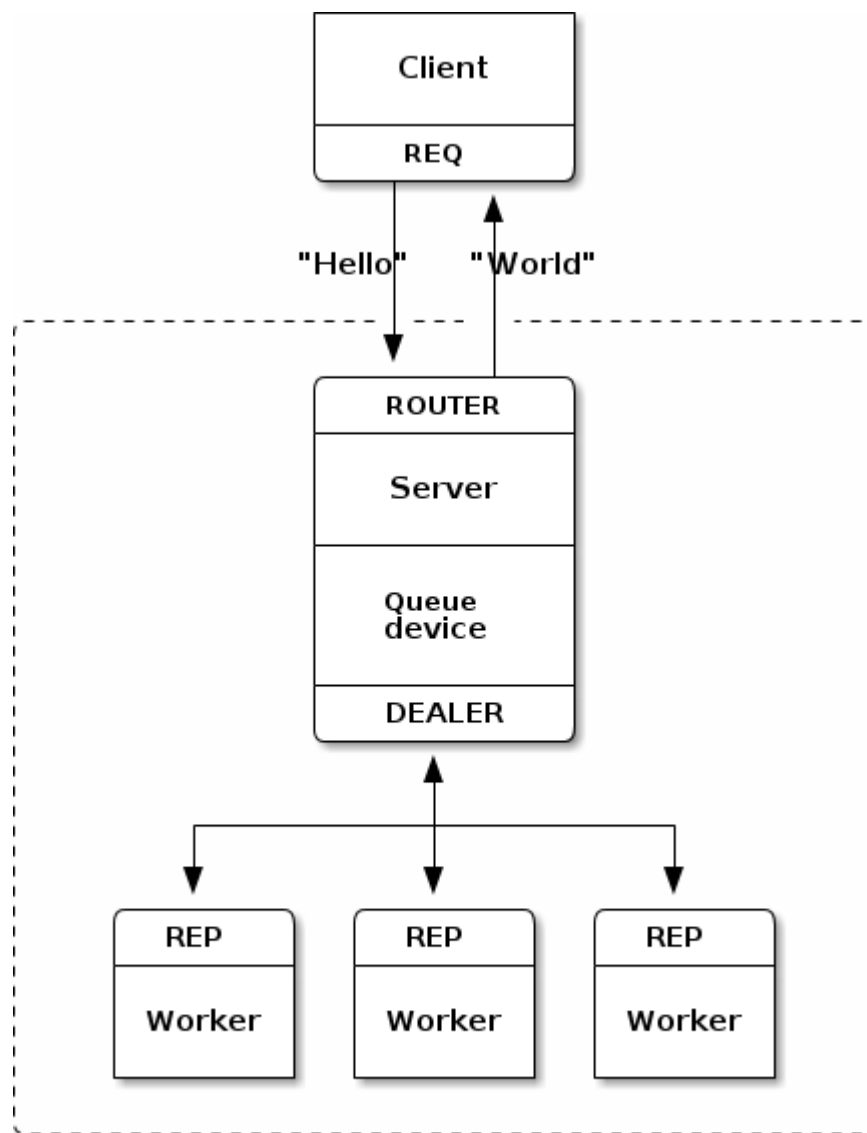


Figure 21 – Multithreaded server

Python 代码 ☆

```
1. import time
2. import threading
3. import zmq
4.
5. def worker_routine(worker_url, context):
6.
7.     socket = context.socket(zmq.REP)
8.
9.     socket.connect(worker_url)
10.
11.     while True:
12.
```

```

13.         string = socket.recv()
14.         print("Received request: [%s]\n" % (string))
15.         time.sleep(1)
16.
17.         socket.send("World")
18.
19. def main():
20.     url_worker = "inproc://workers"
21.     url_client = "tcp://*:5555"
22.
23.     context = zmq.Context(1)
24.
25.     clients = context.socket(zmq.XREP)
26.     clients.bind(url_client)
27.
28.     workers = context.socket(zmq.XREQ)
29.     workers.bind(url_worker)
30.
31.     for i in range(5):
32.         thread = threading.Thread(target=worker_routine, args=(url_wo
33.             rker, context, ))
34.         thread.start()
35.
36.     zmq.device(zmq.QUEUE, clients, workers)
37.
38.     clients.close()
39.     workers.close()
40.     context.term()
41.
42. if __name__ == "__main__":
43.     main()

```

这样的切分还有一个隐性的好处，万一要从多线程转为多进程，可以非常容易的把代码切割过来再利用。这里还给了一个用多线程不用多进程的理由：进程开销太大了（话说，python 是鼓励多进程替代线程的）。

上面代码给出的例子似乎没有子线程间的通信啊？既然支持用多线程，自然不会忘了这个：

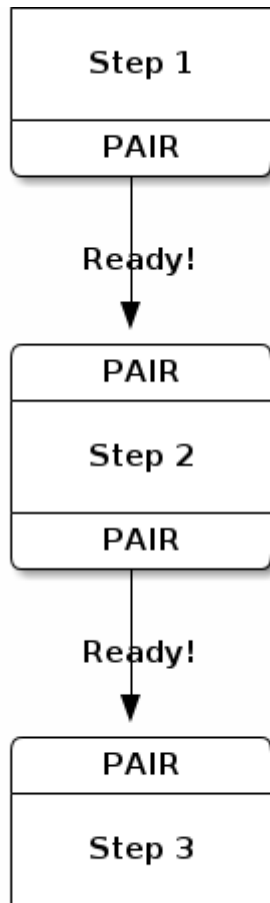


Figure 22 – The Relay Race

Python 代码 ☆

```
1. import threading
2. import zmq
3.
4. def step1(context):
5.     sender = context.socket(zmq.PAIR)
6.     sender.connect("inproc://step2")
7.     sender.send("")
8.
9. def step2(context):
10.    receiver = context.socket(zmq.PAIR)
11.    receiver.bind("inproc://step2")
12.
13.    thread = threading.Thread(target=step1, args=(context, ))
14.    thread.start()
15.
16.    string = receiver.recv()
17.
18.    sender = context.socket(zmq.PAIR)
```

```

19.     sender.connect("inproc://step3")
20.     sender.send("")
21.
22.     return
23.
24. def main():
25.     context = zmq.Context(1)
26.
27.     receiver = context.socket(zmq.PAIR)
28.     receiver.bind("inproc://step3")
29.
30.     thread = threading.Thread(target=step2, args=(context, ))
31.     thread.start()
32.
33.     string = receiver.recv()
34.
35.     print("Test successful!\n")
36.
37.     receiver.close()
38.     context.term()
39.
40.     return
41.
42. if __name__ == "__main__":
43.     main()

```

注意:

这里用到了一个新的端口类型: PAIR。专门为进程间通信准备的(文中还列了下为神马么用之前已经出现过的类型比如应答之类的)。这种类型及时,可靠,安全(进程间其实也是可以用的,与应答相似)。

上一篇讲到了线程间的协作,通过 zeroMQ 的 pair 模式可以很优雅的实现。而在各节点间(进程级),则适用度不高(虽然也能用)。这里给出了两个理由:

- 1.节点间是可以调节的,而线程间不是(线程是稳定的),pair 模式是非自动连接的。
- 2.线程数是固定的,可预估的。而节点则是变动、不可预估的。

由此得出结论: pair 适用于稳定、可控的环境。

所以,有了本章节。不知诸位还记得前面所讲的[发布/订阅模式](#),在那里曾说过这种模式是不太稳定的(主要是指初始阶段),容易在连接未建立前就发布、废弃部分数据。在这里,通过节点间的协作来解决那个难题。

模型图:

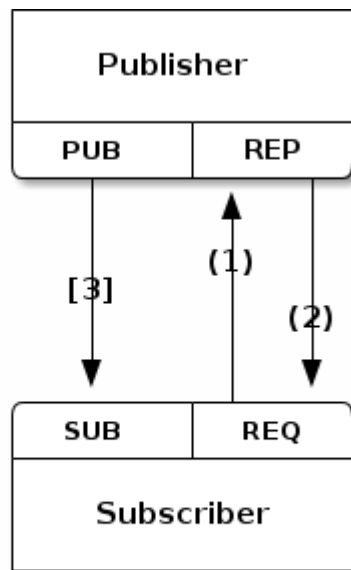


Figure 23 — Pub Sub Synchronization

发布端:

Python 代码

```
1. import zmq
2.
3. SUBSCRIBERS_EXPECTED = 2
4.
5. def main():
6.     context = zmq.Context()
7.
8.     publisher = context.socket(zmq.PUB)
9.     publisher.bind('tcp://*:5561')
10.
11.     syncservice = context.socket(zmq.REP)
12.     syncservice.bind('tcp://*:5562')
13.
14.     subscribers = 0
15.     while subscribers < SUBSCRIBERS_EXPECTED:
16.         msg = syncservice.recv()
17.         syncservice.send('')
18.         subscribers += 1
19.         print "+1 subscriber"
20.
21.     for i in range(1000000):
22.         publisher.send('Rhubarb')
23.
24.     publisher.send('END')
25.
```

```
26. if name == 'main':
27.     main()
```

订阅端:

Python 代码 ☆

```
1. import zmq
2.
3. def main():
4.     context = zmq.Context()
5.
6.     subscriber = context.socket(zmq.SUB)
7.     subscriber.connect('tcp://localhost:5561')
8.     subscriber.setsockopt(zmq.SUBSCRIBE, "")
9.
10.    syncclient = context.socket(zmq.REQ)
11.    syncclient.connect('tcp://localhost:5562')
12.
13.    syncclient.send('')
14.
15.    syncclient.recv()
16.
17.    nbr = 0
18.    while True:
19.        msg = subscriber.recv()
20.        if msg == 'END':
21.            break
22.        nbr += 1
23.
24.    print 'Received %d updates' % nbr
25.
26. if name == 'main':
27.     main()
```

由上例可见，通过应答模式解决了之前的困扰，如果还不放心的话，也可以通过发布特定参数，当订阅端得到时再应答，安全系数便又升了一级。不过这里有个大前提，得先通过某种方式得到或预估一个概念数来确保应用的可用性。

可能绝大多数接触 **zeromq** 的人都会对其去中心的自由感到满意，同时却又对数据传输的可靠性产生怀疑甚至沮丧(如果恰巧你也知道“兔子”的话)。

在这里，或许可以为此作出一些弥补，增强诸位使用它的信心。

zeromq 之所以传输的速度无以伦比，它的"zero copy"功不可没，在这种机制下，减少了数据的二次缓存和挪动，并且减少了通讯间的应答式回应。不过在快速的同时，也降低了数据传递的可靠性。而打开 copy 机制，则在牺牲一定速度的代价下提升了其稳定性。

除了 zero-copy 机制外，zeromq 还提供了一种命名机制，用以建立所谓的"Durable Sockets"。从之前的章节中已知，数据传输层面的事情已经由 zeromq 接管，那么在"Durable Sockets"下，即使你的程序崩溃，或者因为其他原因导致节点丢失(挂掉?)zeromq 会适当的为节点存储数据，以便当节点重新连上时，可以获取之前的数据

未启用命名机制时：

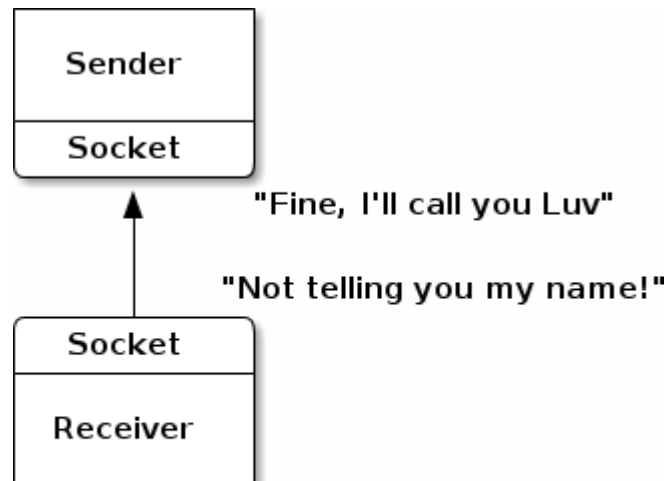


Figure 25 – Transient socket

启用后：

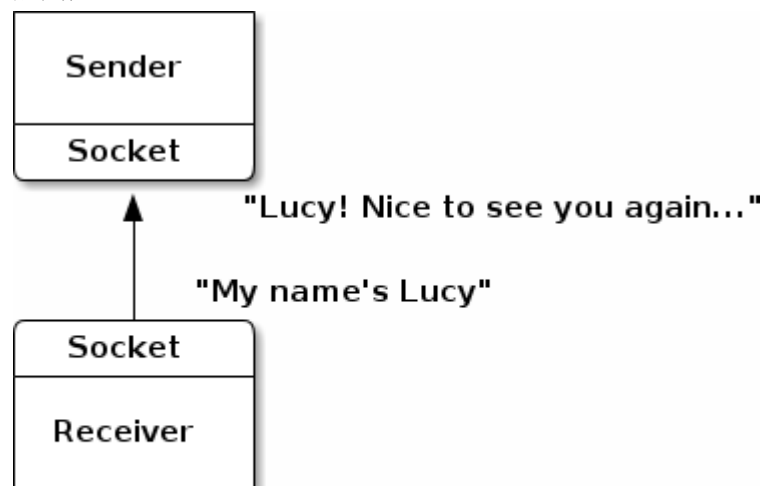


Figure 26 – Durable socket

相关设置：

c 代码 ☆

```
1. zmq_setsockopt (socket, ZMQ_IDENTITY, "Lucy", 4);
```

注意：

1. 如果要启用命名机制，必须在连接前设定名字。
2. 不要重名！
3. 在连接建立后不要再修改名字。
4. 最好不要随机命名。
5. 如果需要获知消息来源的名字，需要在消息发送时附加上 (xrep 会自动获取) 名字。

前面章节有介绍过当传输大数据时，建议分拆成多个小数据逐个发送，以防单条数据过大引发内存溢出等问题。同样的，这也适用于 发布订阅模式，这里用到了一个新名词：信封。

这种封装的数据结构看起来是这样的：

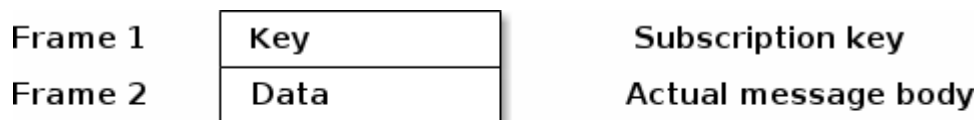


Figure 27 — Pub sub envelope with separate key

由于 key 的关系，不用担心出现 被分拆为多份的数据只被对应订阅方部分持有 这种尴尬的局面。

发布端：

Python 代码

```
1. import time
2. import zmq
3.
4. def main():
5.
6.     context = zmq.Context(1)
7.     publisher = context.socket(zmq.PUB)
8.     publisher.bind("tcp://*:5563")
9.
10.    while True:
11.        publisher.send_multipart(["A", "We don't want to see this"])
12.        publisher.send_multipart(["B", "We would like to see this"])
13.        time.sleep(1)
14.
```

```

15.     publisher.close()
16.     context.term()
17.
18. if name == "main":
19.     main()

```

订阅端：

Python 代码

```

1. import zmq
2.
3. def main():
4.
5.     context = zmq.Context(1)
6.     subscriber = context.socket(zmq.SUB)
7.     subscriber.connect("tcp://localhost:5563")
8.     subscriber.setsockopt(zmq.SUBSCRIBE, "B")
9.
10.    while True:
11.        [address, contents] = subscriber.recv_multipart()
12.        print("[%s] %s\n" % (address, contents))
13.
14.    subscriber.close()
15.    context.term()
16.
17. if name == "main":
18.     main()

```

如果想要尝试下上一章节的命名制，对数据做些许变动即可：

Frame 1	Key	Subscription key
Frame 2	Identity	Address of publisher
Frame 3	Data	Actual message body

Figure 28 – Pub sub envelope with sender address

[前文](#)曾提到过命名机制，事实上它是一把双刃剑。在能够持有数据等待重新连接的时候，也增加了持有数据方的负担(危险)，特别是在"发布/订阅"模式下，可谓牵一发而动全身。

这里先给出一组示例，在代码的运行过程中，通过重启消费者来观察发布者的进程状态。

发布端：

Python 代码 ☆

```
1. import zmq
2. import time
3.
4. context = zmq.Context()
5.
6. sync = context.socket(zmq.PULL)
7. sync.bind("tcp://*:5564")
8.
9. publisher = context.socket(zmq.PUB)
10. publisher.bind("tcp://*:5565")
11.
12. sync_request = sync.recv()
13.
14. for n in xrange(10):
15.     msg = "Update %d" % n
16.     publisher.send(msg)
17.     time.sleep(1)
18.
19. publisher.send("END")
20. time.sleep(1) # Give OMQ/2.0.x time to flush output
```

订阅端:

Python 代码 ☆

```
1. import zmq
2. import time
3.
4. context = zmq.Context()
5.
6. subscriber = context.socket(zmq.SUB)
7. subscriber.setsockopt(zmq.IDENTITY, "Hello")
8. subscriber.setsockopt(zmq.SUBSCRIBE, "")
9. subscriber.connect("tcp://localhost:5565")
10.
11. sync = context.socket(zmq.PUSH)
12. sync.connect("tcp://localhost:5564")
13. sync.send("")
14.
15. while True:
16.     data = subscriber.recv()
17.     print data
18.     if data == "END":
19.         break
```


订阅端得到的信息：

Java 代码 ☆

```
1.$ durasub
2.Update 0
3.Update 1
4.Update 2
5.^C
6.$ durasub
7.Update 3
8.Update 4
9.Update 5
10.Update 6
11.Update 7
12.^C
13.$ durasub
14.Update 8
15.Update 9
16.END
```

数据被发布者存储了，而发布者的内存占用也节节升高 (很危险啊)。所以是否使用命名策略是需要谨慎选择的。为了以防万一，zeromq 也提供了"高水位"机制，即当发送端持有数据达到一定数量就不再存储后面的数据，很好的控制了风险。这个机制也适当解决了[这里](#)的慢消费问题。

使用了 高水位 后的测试结果：

Java 代码 ☆

```
1.$ durasub
2.Update 0
3.Update 1
4.^C
5.$ durasub
6.Update 2
7.Update 3
8.Update 7
9.Update 8
10.Update 9
11.END
```

"高水位"封堵了内存崩溃的可能性，却是以数据丢失为代价的，zeromq 也为此配对提供了"swap"功能，将内存中的数据转存入硬盘，实现了"既不耗内存又不丢数据"。

实现代码:

Python 代码 ☆

```
1. import zmq
2. import time
3.
4. context = zmq.Context()
5.
6. # Subscriber tells us when it's ready here
7. sync = context.socket(zmq.PULL)
8. sync.bind("tcp://*:5564")
9.
10. # We send updates via this socket
11. publisher = context.socket(zmq.PUB)
12. publisher.bind("tcp://*:5565")
13.
14. # Prevent publisher overflow from slow subscribers
15. publisher.setsockopt(zmq.HWM, 1)
16.
17. # Specify the swap space in bytes, this covers all subscribers
18. publisher.setsockopt(zmq.SWAP, 25000000)
19.
20. # Wait for synchronization request
21. sync_request = sync.recv()
22.
23. # Now broadcast exactly 10 updates with pause
24. for n in xrange(10):
25.     msg = "Update %d" % n
26.     publisher.send(msg)
27.     time.sleep(1)
28.
29. publisher.send("END")
30. time.sleep(1) # Give OMQ/2.0.x time to flush output
```

注意点:

高水位与交换区的设定, 是需要根据实际运用状态来确定的, 高水位设的过小, 会影响到速度。

如果是数据存储端崩溃了, 那么, 所有数据将彻底消失。

关于高水位的特别说明:

除了 PUB 型会在达到高水位丢弃后续数据外, 其他类型的都会以阻塞的形式来应对后续数据。

线程间的通信, 高水位是通信双方共同设置的总和, 如果有一方没有设置, 则高水位规则不会起到作用。

整整一大章全部讲的应答模式的进阶, 应该很重要吧(简直是一定的)。

上一节讲到了发布/订阅模式 关于封装的话题，在应答模式中也是如此，不过这个动作已经被底层(zeromq)接管，对应用透明。而其中普通模式与 X 模式又有区别，例如：req 连接 X rep:

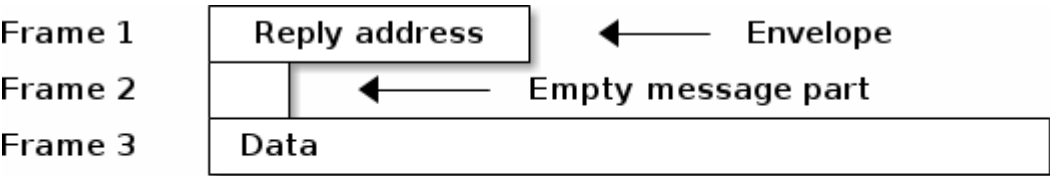


Figure 29 – Single hop request-reply envelope

说明：
第三部分是实际发送的数据
第二部分是 REQ 向 XREP 发送请求时底层附加的
第一部分是 XREP 自身地址
注意：
前文已经说过，XREP 其实用以平衡负载，所以这里由它对请求数据做了封装操作，如果通过多个 XREP,数据结构说明：
第三部分是实际发送的数据
第二部分是 REQ 向 XREP 发送请求时底层附加的
第一部分是 XREP 自身地址
注意：
前文已经说过，XREP 其实用以平衡负载，所以这里由它对请求数据做了封装操作，如果通过多个 XREP,数据结构就会变成这个样子：

(Next envelope will go here)

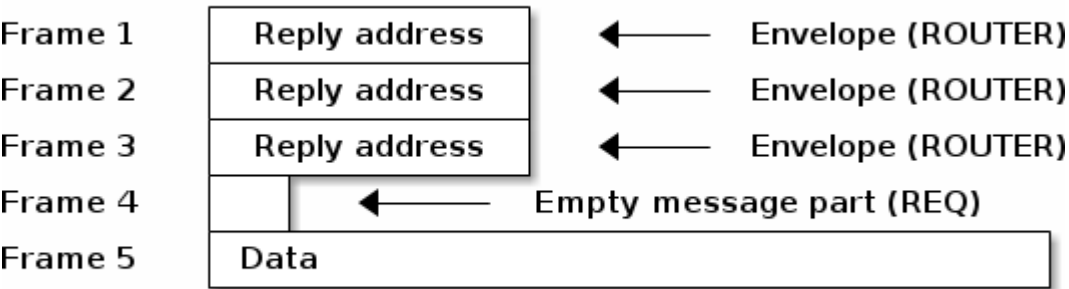


Figure 30 – Multihop request-reply envelope

同时，如果没有启用命名机制，XREP 会自动赋予临时名字：

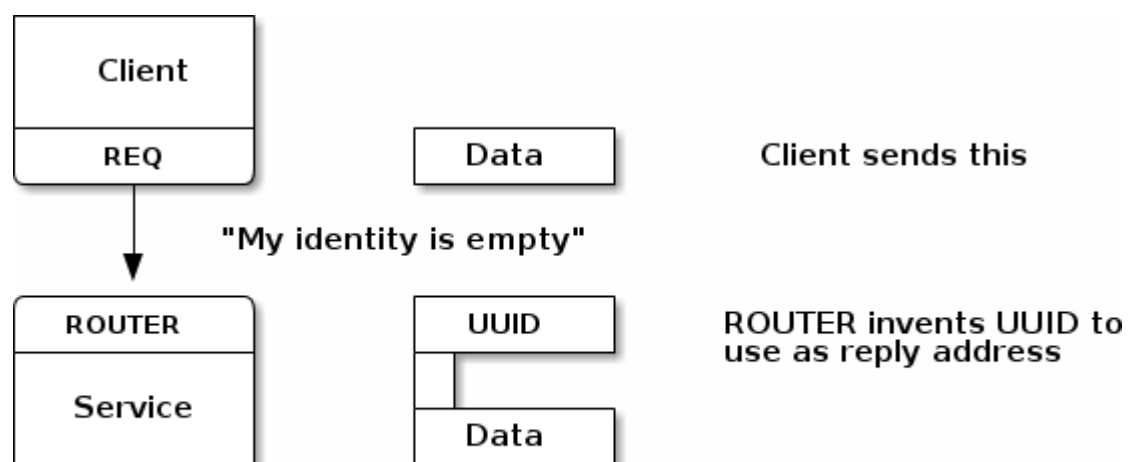


Figure 31 – ROUTER invents a UUID for transient sockets

不然，就是这样了：

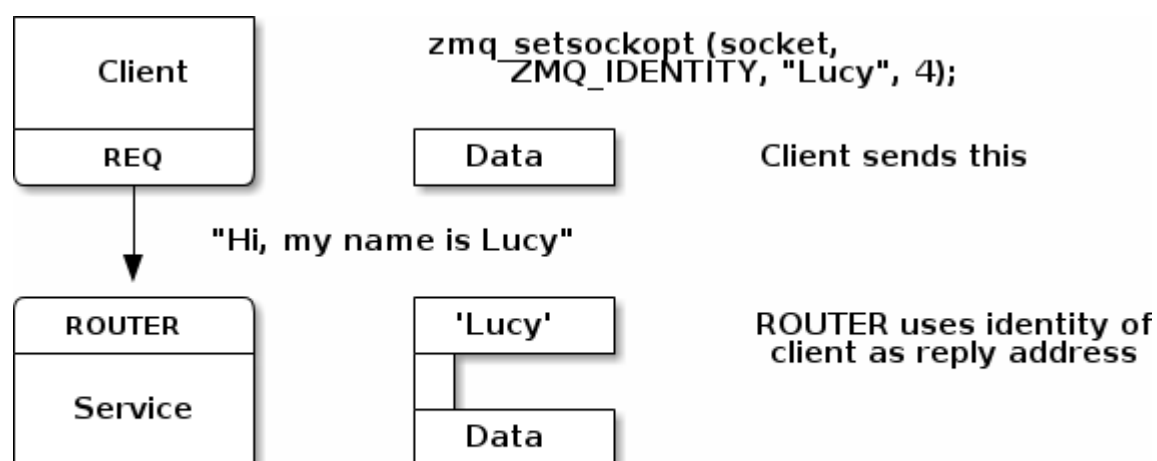


Figure 32 – ROUTER uses identity if it knows it

这里给出一个验证代码：

Python 代码 ☆

```
1. import zmq
2. import zhelpers
3.
4. context = zmq.Context()
5.
6. sink = context.socket(zmq.XREP)
7. sink.bind("inproc://example")
8.
9. # First allow OMQ to set the identity
10. anonymous = context.socket(zmq.XREQ)
```

```

11. anonymous.connect("inproc://example")
12. anonymous.send("XREP uses a generated UUID")
13. zhelpers.dump(sink)
14.
15. # Then set the identity ourself
16. identified = context.socket(zmq.XREQ)
17. identified.setsockopt(zmq.IDENTITY, "Hello")
18. identified.connect("inproc://example")
19. identified.send("XREP socket uses REQ's socket identity")
20. zhelpers.dump(sink)

```

上面的代码用到的 zhelpers:

Python 代码 ☆

```

1. from random import randint
2.
3. import zmq
4.
5.
6. # Receives all message parts from socket, prints neatly
7. def dump(zsocket):
8.     print "-----"
9.     for part in zsocket.recv_multipart():
10.         print "[%03d]" % len(part),
11.         if all(31 < ord(c) < 128 for c in part):
12.             print part
13.         else:
14.             print "".join("%x" % ord(c) for c in part)
15.
16.
17. # Set simple random printable identity on socket
18. def set_id(zsocket):
19.     identity = "%04x-%04x" % (randint(0, 0x10000), randint(0, 0x1000
20.         0))
21.     zsocket.setsockopt(zmq.IDENTITY, identity)

```

在上一节中已经提到 XREP 主要工作是包装数据，打上标记以便方便的传递数据。那么，换个角度来看，这不就是路由么！其实在[优雅的扩展](#)中有介绍过。在这里针对 XREP 模式做深入的探索。

首先，得要理一下其中几种类型的差别(相似的名字真是坑爹啊):

REQ,官网称之为"老妈类型"，因为它负责主动提出请求，并且要求得到答复(严格同步的)

REP,"老爸类型"，负责应答请求，(从不主动，也是严格同步的)

XREQ,"分销类型"，负责对进出的数据排序，均匀的分发给接入的 REP 或者 XREP

XREP,"路由类型",将信息转发至任何与他有连接的地方,可以和任何类型相连,不过看起来,天然的和老妈比较亲密。

传统的看法是,应答模式自然得同步的。不过在这里,显然是可以做到异步的(只要"老爸"或者"老妈"不处在整个线路的中间位置)。

通常定制路由会用到以下四种通讯连接:

XREP-to-XREQ.

XREP-to-REQ.

XREP-to-REP.

XREP-to-XREP.

在这几种基本连接下,定制路由完全看各人的想象力了。不过在即将到来的各种通讯的详解前,还是得要申明一下:

自定义路由有风险,使用需谨慎啊!

首先要介绍的是 XREP-XREQ 模式:

这是比较简单的一种模式, XREQ 会用到三种情景: 1, 汇总, 2, 代理分发, 3, 响应答复。这里要注意,如果 XREQ 用于响应答复,最好只有一个 XREP 与它相连,因为 XREQ 不会指定发送目标,而会将数据均衡的摊派给所有与它有连接关系的 XREP。

这里给出一个汇总式的例子:

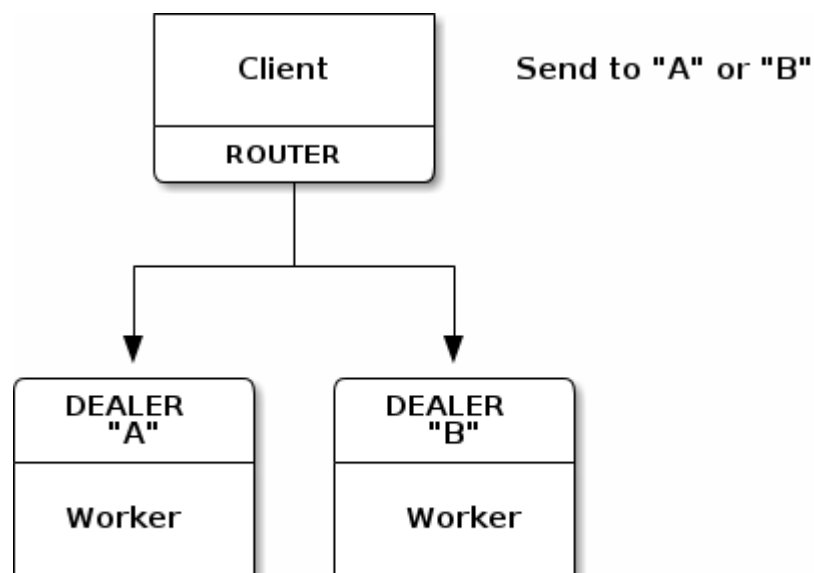


Figure 33 – Router to dealer custom routing

Python 代码 ☆

```
1. import time
2. import random
3. from threading import Thread
```

```
4. import zmq
5.
6. def worker_a(context):
7.     worker = context.socket(zmq.XREQ)
8.     worker.setsockopt(zmq.IDENTITY, 'A')
9.     worker.connect("ipc://routing.ipc")
10.
11.     total = 0
12.     while True:
13.         request = worker.recv()
14.         finished = request == "END"
15.         if finished:
16.             print "A received:", total
17.             break
18.         total += 1
19.
20. def worker_b(context):
21.     worker = context.socket(zmq.XREQ)
22.     worker.setsockopt(zmq.IDENTITY, 'B')
23.     worker.connect("ipc://routing.ipc")
24.
25.     total = 0
26.     while True:
27.         request = worker.recv()
28.         finished = request == "END"
29.         if finished:
30.             print "B received:", total
31.             break
32.         total += 1
33.
34. context = zmq.Context()
35. client = context.socket(zmq.XREP)
36. client.bind("ipc://routing.ipc")
37.
38. Thread(target=worker_a, args=(context,)).start()
39. Thread(target=worker_b, args=(context,)).start()
40.
41. time.sleep(1)
42.
43. for _ in xrange(10):
44.     if random.randint(0, 2) > 0:
45.         client.send("A", zmq.SNDMORE)
46.     else:
47.         client.send("B", zmq.SNDMORE)
```

```

48.
49.     client.send("This is the workload")
50.
51. client.send("A", zmq.SNDMORE)
52. client.send("END")
53.
54. client.send("B", zmq.SNDMORE)
55. client.send("END")
56.
57. time.sleep(1) # Give 0MQ/2.0.x time to flush output

```

传递的数据结构:

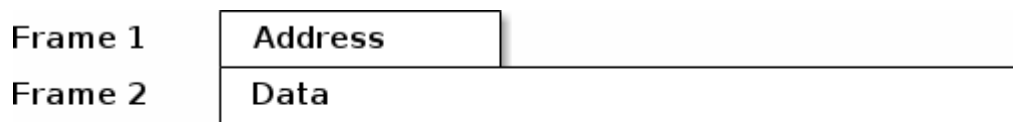


Figure 34 – Routing envelope for dealer

因为这是无应答的，比较简单，如果要应答的话，会稍微麻烦些，需要用到前面讲的 **POLL** 来调度。在代码中有一行 **sleep**，主要是为了等待接收端准备就绪，否则有可能像“发布/订阅”那样，丢失数据。除了 **XREP** 与 **PUB** 外，其他类型都不会存在这种问题(都会阻塞等待)。

注意:

在路由模式下，永远是不安全的，想要得到保障，就应该在得到路由信息时答复路由(回应一下)。

XREP-REQ 模式:

典型的“老妈模式”，只有当她真的要听你说时，她才能听的进去。所以首先，得要 **REQ** 告诉你“她准备好了，你可以讲了”，然后，你才能倾吐...

一般来说与 **XREQ** 一样，一个 **REQ** 只能连接一个 **XREP**（除非你想做容错，不过，不建议那样）。

实例模型:

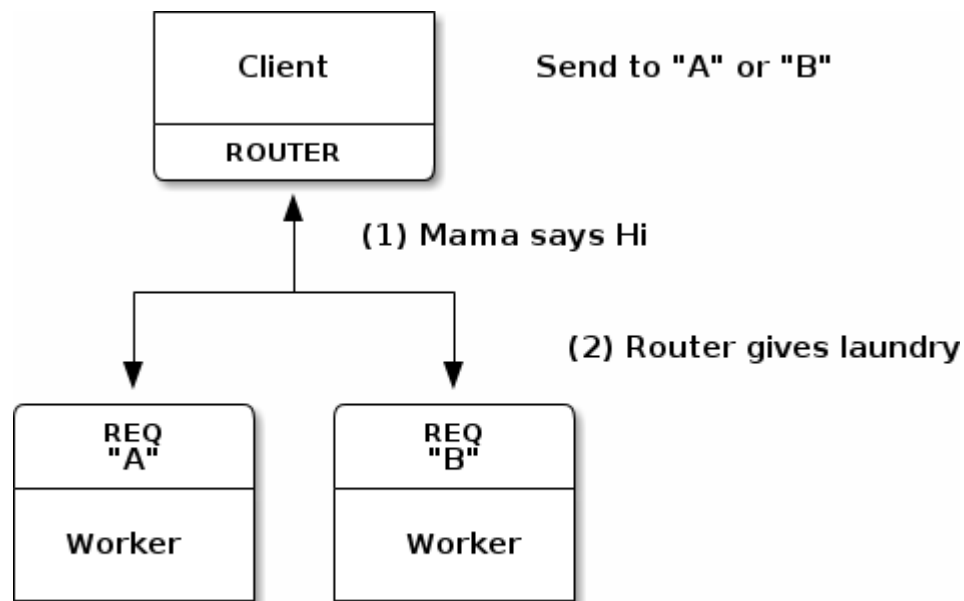


Figure 35 – Router to mama custom routing

Python 代码 ☆

```
1. import time
2. import random
3. from threading import Thread
4.
5. import zmq
6.
7. import zhelpers
8.
9. NBR_WORKERS = 10
10.
11. def worker_thread(context):
12.     worker = context.socket(zmq.REQ)
13.
14.     # We use a string identity for ease here
15.     zhelpers.set_id(worker)
16.     worker.connect("ipc://routing.ipc")
17.
18.     total = 0
19.     while True:
20.         # Tell the router we're ready for work
21.         worker.send("ready")
22.
23.         # Get workload from router, until finished
24.         workload = worker.recv()
25.         finished = workload == "END"
```

```

26.         if finished:
27.             print "Processed: %d tasks" % total
28.             break
29.         total += 1
30.
31.         # Do some random work
32.         time.sleep(random.random() / 10 + 10 ** -9)
33.
34. context = zmq.Context()
35. client = context.socket(zmq.XREP)
36. client.bind("ipc://routing.ipc")
37.
38. for _ in xrange(NBR_WORKERS):
39.     Thread(target=worker_thread, args=(context,)).start()
40.
41. for _ in xrange(NBR_WORKERS * 10):
42.     # LRU worker is next waiting in the queue
43.     address = client.recv()
44.     empty = client.recv()
45.     ready = client.recv()
46.
47.     client.send(address, zmq.SNDMORE)
48.     client.send("", zmq.SNDMORE)
49.     client.send("This is the workload")
50.
51. # Now ask mama to shut down and report their results
52. for _ in xrange(NBR_WORKERS):
53.     address = client.recv()
54.     empty = client.recv()
55.     ready = client.recv()
56.
57.     client.send(address, zmq.SNDMORE)
58.     client.send("", zmq.SNDMORE)
59.     client.send("END")
60.
61. time.sleep(1) # Give OMQ/2.0.x time to flush output

```

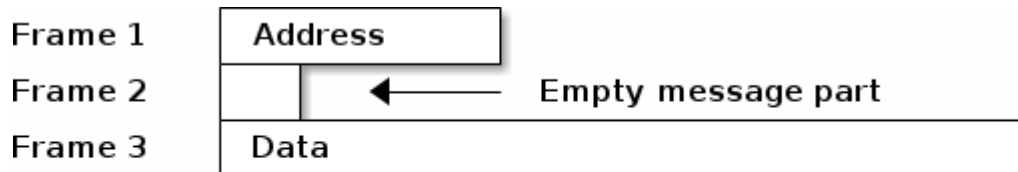


Figure 36 – Routing envelope for mama (REQ)

注意点：

如果"老妈"没有和你主动联系，那么就不要再向她发一个字！

XREP-REP 模式：

这种模式并不属于经典的应用范畴，通常的做法是 XREP-XREQ-REP，由“分销商”来负责数据的传递。不过既然有这两种类型，不妨试着联通看看～

实例模型：

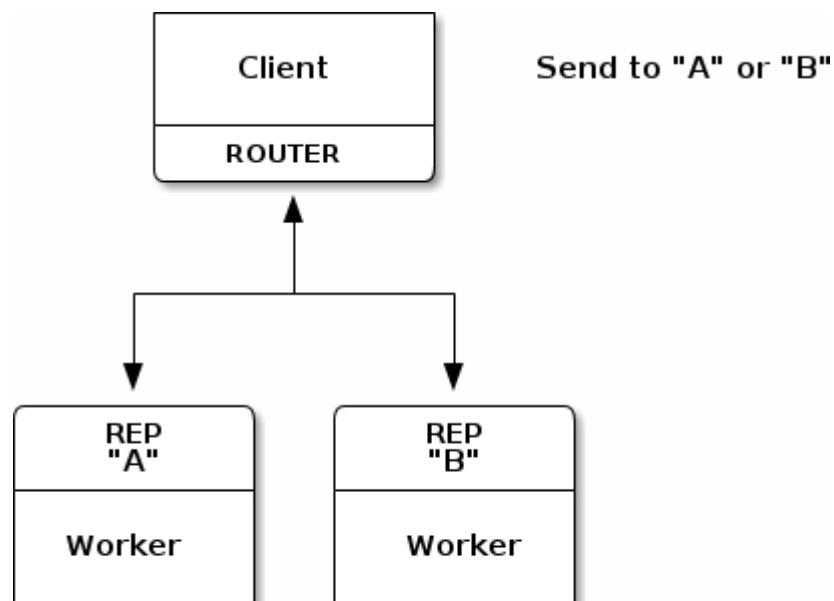


Figure 37 – Router to papa custom routing

Python 代码 ☆

```
1. import time
2.
3. import zmq
4.
5. import zhelpers
6.
7. context = zmq.Context()
8. client = context.socket(zmq.XREP)
9. client.bind("ipc://routing.ipc")
10.
11. worker = context.socket(zmq.REP)
12. worker.setsockopt(zmq.IDENTITY, "A")
13. worker.connect("ipc://routing.ipc")
14.
15. # Wait for sockets to stabilize
```

```

16.time.sleep(1)
17.
18.client.send("A", zmq.SNDMORE)
19.client.send("address 3", zmq.SNDMORE)
20.client.send("address 2", zmq.SNDMORE)
21.client.send("address 1", zmq.SNDMORE)
22.client.send("", zmq.SNDMORE)
23.client.send("This is the workload")
24.
25.# Worker should get just the workload
26.zhelpers.dump(worker)
27.
28.# We don't play with envelopes in the worker
29.worker.send("This is the reply")
30.
31.# Now dump what we got off the XREP socket...
32.zhelpers.dump(client)

```

数据结构:

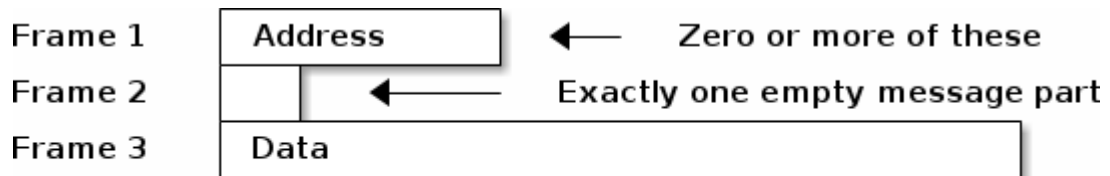


Figure 38 — Routing envelope for papa aka REP

注意:

因为 REP 不像 REQ 那样,他是被动的,所以在往 REP 传递数据时,先得确定他已经存在,不然数据可就丢了。

从经典到超越经典。

首先,先回顾下经典:

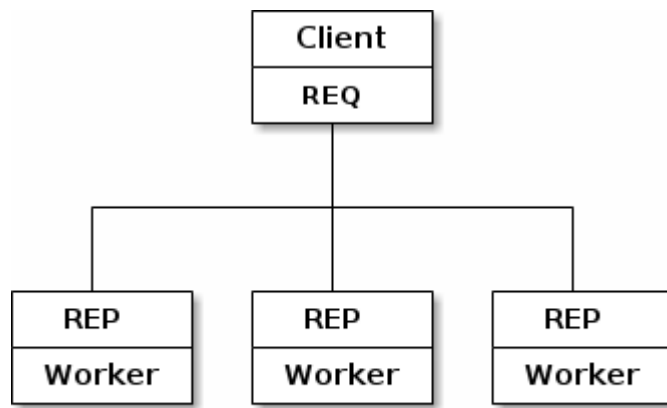


Figure 39 – Basic request-reply

然后，扩展：

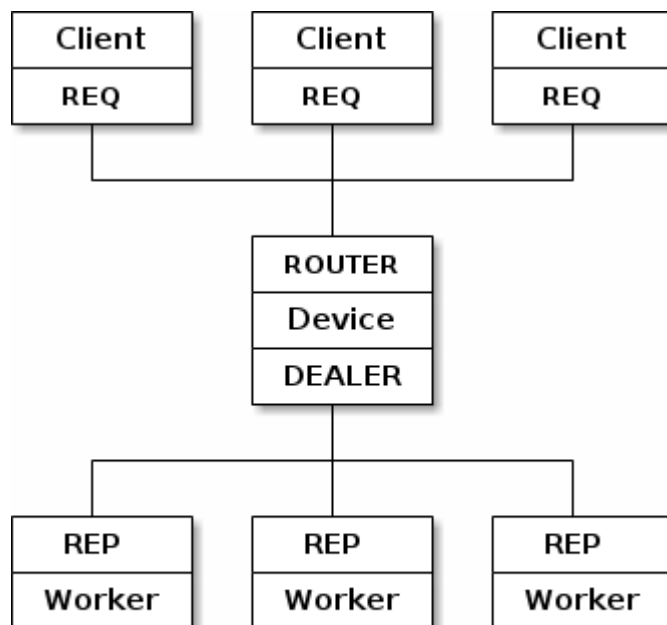


Figure 40 – Stretched request-reply

然后，变异：

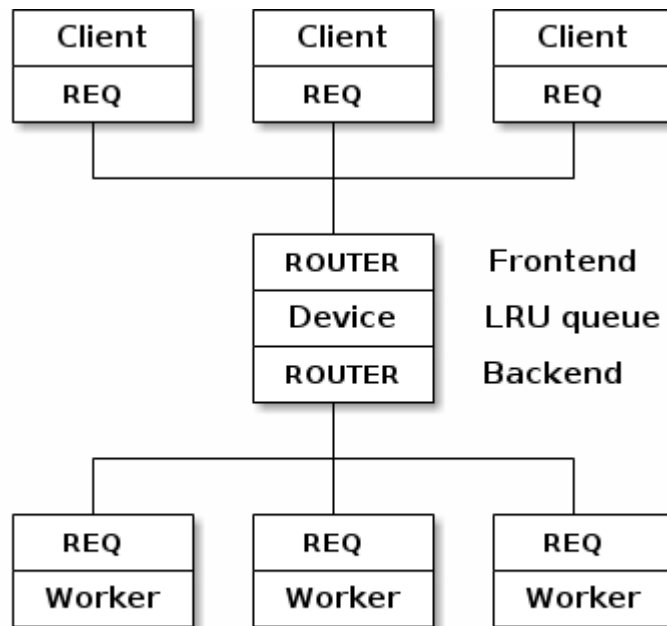


Figure 41 – Stretched request-reply with LRU

Python 代码 ☆

```

1. import threading
2. import time
3. import zmq
4.
5. NBR_CLIENTS = 10
6. NBR_WORKERS = 3
7.
8. def worker_thread(worker_url, context, i):
9.     """ Worker using REQ socket to do LRU routing """
10.
11.     socket = context.socket(zmq.REQ)
12.
13.     identity = "Worker-%d" % (i)
14.
15.     socket.setsockopt(zmq.IDENTITY, identity) #set worker identity
16.
17.     socket.connect(worker_url)
18.
19.     # Tell the broker we are ready for work
20.     socket.send("READY")
21.
22.     try:
23.         while True:

```

```

24.
25.         # python binding seems to eat empty frames
26.         address = socket.recv()
27.         request = socket.recv()
28.
29.         print("%s: %s\n" %(identity, request))
30.
31.         socket.send(address, zmq.SNDMORE)
32.         socket.send("", zmq.SNDMORE)
33.         socket.send("OK")
34.
35.     except zmq.ZMQError, zerr:
36.         # context terminated so quit silently
37.         if zerr.strerror == 'Context was terminated':
38.             return
39.         else:
40.             raise zerr
41.
42.
43. def client_thread(client_url, context, i):
44.     """ Basic request-reply client using REQ socket """
45.
46.     socket = context.socket(zmq.REQ)
47.
48.     identity = "Client-%d" % (i)
49.
50.     socket.setsockopt(zmq.IDENTITY, identity) #Set client identity. Makes tracing easier
51.
52.     socket.connect(client_url)
53.     # Send request, get reply
54.     socket.send("HELLO")
55.     reply = socket.recv()
56.     print("%s: %s\n" % (identity, reply))
57.     return
58.
59. def main():
60.     """ main method """
61.
62.     url_worker = "inproc://workers"
63.     url_client = "inproc://clients"
64.     client_nbr = NBR_CLIENTS
65.
66.     # Prepare our context and sockets

```

```

67.     context = zmq.Context(1)
68.     frontend = context.socket(zmq.XREP)
69.     frontend.bind(url_client)
70.     backend = context.socket(zmq.XREP)
71.     backend.bind(url_worker)
72.
73.
74.
75.     # create workers and clients threads
76.     for i in range(NBR_WORKERS):
77.         thread = threading.Thread(target=worker_thread, args=(url_w
orker, context, i, ))
78.         thread.start()
79.
80.     for i in range(NBR_CLIENTS):
81.         thread_c = threading.Thread(target=client_thread, args=(url
_client, context, i, ))
82.         thread_c.start()
83.
84.     # Logic of LRU loop
85.     # - Poll backend always, frontend only if 1+ worker ready
86.     # - If worker replies, queue worker as ready and forward repl
y
87.     # to client if necessary
88.     # - If client requests, pop next worker and send request to it
89.
90.     # Queue of available workers
91.     available_workers = 0
92.     workers_list      = []
93.
94.     # init poller
95.     poller = zmq.Poller()
96.
97.     # Always poll for worker activity on backend
98.     poller.register(backend, zmq.POLLIN)
99.
100.    # Poll front-end only if we have available workers
101.    poller.register(frontend, zmq.POLLIN)
102.
103.    while True:
104.
105.        socks = dict(poller.poll())
106.        # Handle worker activity on backend

```



```

107.         if (backend in socks and socks[backend] == zmq.POLLI
N):
108.
109.             # Queue worker address for LRU routing
110.             worker_addr = backend.recv()
111.
112.             assert available_workers < NBR_WORKERS
113.
114.             # add worker back to the list of workers
115.             available_workers += 1
116.             workers_list.append(worker_addr)
117.
118.             # Second frame is empty
119.             empty = backend.recv()
120.             assert empty == ""
121.
122.             # Third frame is READY or else a client reply address
S
123.             client_addr = backend.recv()
124.
125.             # If client reply, send rest back to frontend
126.             if client_addr != "READY":
127.
128.                 # Following frame is empty
129.                 empty = backend.recv()
130.                 assert empty == ""
131.
132.                 reply = backend.recv()
133.
134.                 frontend.send(client_addr, zmq.SNDMORE)
135.                 frontend.send("", zmq.SNDMORE)
136.                 frontend.send(reply)
137.
138.                 client_nbr -= 1
139.
140.                 if client_nbr == 0:
141.                     break # Exit after N messages
142.
143.             # poll on frontend only if workers are available
144.             if available_workers > 0:
145.
146.                 if (frontend in socks and socks[frontend] == zmq.POL
LIN):

```

```

147.             # Now get next client request, route to LRU worke
r
148.             # Client request is [address][empty][request]
149.             client_addr = frontend.recv()
150.
151.             empty = frontend.recv()
152.             assert empty == ""
153.
154.             request = frontend.recv()
155.
156.             # Dequeue and drop the next worker address
157.             available_workers -= 1
158.             worker_id = workers_list.pop()
159.
160.             backend.send(worker_id, zmq.SNDMORE)
161.             backend.send("", zmq.SNDMORE)
162.             backend.send(client_addr, zmq.SNDMORE)
163.             backend.send(request)
164.
165.             #out of infinite loop: do some housekeeping
166.             time.sleep (1)
167.
168.             frontend.close()
169.             backend.close()
170.             context.term()
171.
172.
173.     if name == "main":
174.         main()

```

client 发出的数据结构:

Frame 1

5	HELLO
---	-------

 Data part

Figure 42 – Message that client sends

路由处理成:

Frame 1	6	CLIENT	Identity of client
Frame 2	0		Empty message part
Frame 3	5	HELLO	Data part

Figure 43 – Message coming in on frontend

再转给 worker 成:

Frame 1	6	WORKER	Identity of worker
Frame 2	0		Empty message part
Frame 3	6	CLIENT	Identity of client
Frame 4	0		Empty message part
Frame 5	5	HELLO	Data part

Figure 44 – Message sent to backend

工人处理的数据:

Frame 1	6	CLIENT	Identity of client
Frame 2	0		Empty message part
Frame 3	5	HELLO	Data part

Figure 45 – Message delivered to worker

由 worker 到 client 是一个逆序过程，不过因为两边都是 REQ 类型，所以其实是一致的。

[补]:

通常，上层的 api 会帮我们做一些事，免去了逐步封装数据的麻烦，比如在 python 中，最终代码会是这个样子:

Python 代码 ☆

```

1. import threading
2. import time
3. import zmq
4.
5. NBR_CLIENTS = 10
6. NBR_WORKERS = 3
7.
8. def worker_thread(worker_url, context, i):

```

```

9.     """ Worker using REQ socket to do LRU routing """
10.
11.     socket = context.socket(zmq.REQ)
12.
13.     identity = "Worker-%d" % (i)
14.
15.     socket.setsockopt(zmq.IDENTITY, identity) #set worker identity
16.
17.     socket.connect(worker_url)
18.
19.     # Tell the borker we are ready for work
20.     socket.send("READY")
21.
22.     try:
23.         while True:
24.
25.             [address, request] = socket.recv_multipart()
26.
27.             print("%s: %s\n" %(identity, request))
28.
29.             socket.send_multipart([address, "", "OK"])
30.
31.     except zmq.ZMQError, zerr:
32.         # context terminated so quit silently
33.         if zerr.strerror == 'Context was terminated':
34.             return
35.         else:
36.             raise zerr
37.
38.
39. def client_thread(client_url, context, i):
40.     """ Basic request-reply client using REQ socket """
41.
42.     socket = context.socket(zmq.REQ)
43.
44.     identity = "Client-%d" % (i)
45.
46.     socket.setsockopt(zmq.IDENTITY, identity) #Set client identity. Makes tracing easier
47.
48.     socket.connect(client_url)
49.
50.     # Send request, get reply

```

```

51.     socket.send("HELLO")
52.
53.     reply = socket.recv()
54.
55.     print("%s: %s\n" % (identity, reply))
56.
57.     return
58.
59.
60. def main():
61.     """ main method """
62.
63.     url_worker = "inproc://workers"
64.     url_client = "inproc://clients"
65.     client_nbr = NBR_CLIENTS
66.
67.     # Prepare our context and sockets
68.     context = zmq.Context(1)
69.     frontend = context.socket(zmq.XREP)
70.     frontend.bind(url_client)
71.     backend = context.socket(zmq.XREP)
72.     backend.bind(url_worker)
73.
74.
75.
76.     # create workers and clients threads
77.     for i in range(NBR_WORKERS):
78.         thread = threading.Thread(target=worker_thread, args=(url_w
orker, context, i, ))
79.         thread.start()
80.
81.     for i in range(NBR_CLIENTS):
82.         thread_c = threading.Thread(target=client_thread, args=(url
_client, context, i, ))
83.         thread_c.start()
84.
85.     # Logic of LRU loop
86.     # - Poll backend always, frontend only if 1+ worker ready
87.     # - If worker replies, queue worker as ready and forward repl
y
88.     # to client if necessary
89.     # - If client requests, pop next worker and send request to it
90.
91.     # Queue of available workers

```

```

92.     available_workers = 0
93.     workers_list      = []
94.
95.     # init poller
96.     poller = zmq.Poller()
97.
98.     # Always poll for worker activity on backend
99.     poller.register(backend, zmq.POLLIN)
100.
101.     # Poll front-end only if we have available workers
102.     poller.register(frontend, zmq.POLLIN)
103.
104.     while True:
105.
106.         socks = dict(poller.poll())
107.
108.         # Handle worker activity on backend
109.         if (backend in socks and socks[backend] == zmq.POLLI
N):
110.
111.             # Queue worker address for LRU routing
112.             message = backend.recv_multipart()
113.
114.             assert available_workers < NBR_WORKERS
115.
116.             worker_addr = message[0]
117.
118.             # add worker back to the list of workers
119.             available_workers += 1
120.             workers_list.append(worker_addr)
121.
122.             # Second frame is empty
123.             empty = message[1]
124.             assert empty == ""
125.
126.             # Third frame is READY or else a client reply address
S
127.             client_addr = message[2]
128.
129.             # If client reply, send rest back to frontend
130.             if client_addr != "READY":
131.
132.                 # Following frame is empty
133.                 empty = message[3]

```

```

134.             assert empty == ""
135.
136.             reply = message[4]
137.
138.             frontend.send_multipart([client_addr, "", reply])
139.
140.             client_nbr -= 1
141.
142.             if client_nbr == 0:
143.                 break # Exit after N messages
144.
145.             # poll on frontend only if workers are available
146.             if available_workers > 0:
147.
148.                 if (frontend in socks and socks[frontend] == zmq.POLLIN):
149.                     # Now get next client request, route to LRU worker
150.                     # Client request is [address][empty][request]
151.
152.                     [client_addr, empty, request] = frontend.recv_multipart()
153.
154.                     assert empty == ""
155.
156.                     # Dequeue and drop the next worker address
157.                     available_workers -= 1
158.                     worker_id = workers_list.pop()
159.
160.                     backend.send_multipart([worker_id, "", client_addr, request])
161.
162.
163.             #out of infinite loop: do some housekeeping
164.             time.sleep(1)
165.
166.             frontend.close()
167.             backend.close()
168.             context.term()
169.
170.
171. if name == "main":
172.     main()

```

