ZeroMQ에 대한 소개글 번역 컴퓨터, 게임, 인터넷 2012. 3. 16. 16:25

싱글 프로세스로 만들어진 파이썬 서버를 가급적 적게 수정하고 멀티프로세스화 하는 방법을 찾다가, ZeroMQ라는 네트워크 라이브러리를 우연히 발견하였다. 쉽고 독특한 접근 방법을 가지고 있는데, 애플리케이션에 네트워크 기능을 간단하게 추가하거나 서버의 확장성 구현을 시행착오 없이 하고 싶다면 ZeroMQ가 좋은 답이 될 수 있다.

다음은 인터넷에서 발견한 ZeroMQ에 대한 소개 글을 번역해 본 것이다. (원본은 http://nichol.as/zeromq-an-introduction 에서 볼 수 있다.)
한글 페이지 http://kr.zeromq.org/ 도 있으니 참고하면 좋을 것이다.

ZeroMQ에 대한 소개

ZeroMQ는 메시징 라이브러리이다. 그것은 많은 수고를 들이지 않고도 복잡한 커뮤니케이션 시스템을 설계할 수 있도록 해준다. ZeroMQ는 스스로를 효율적으로 설명하기 위해 지금까지 많은 노력을 해왔다. 처음에는 '메시징 미들웨어'로 소개되었지만, 나중에는 '스테로이드를 맞은 TCP' 그리고 이제는 '네트워크 스택의 새로운 레이어'라고 말한다.

나는 처음에 ZeroMQ를 이해하는데 어려움을 겪어서 기존의 내 생각을 완전히 버려야 했었다. 우선 그것은 RabbitMQ나 ActiveMQ같은 완전한 메시징 시스템은 아니다. 나는 린던 리서치의 사람들이 그것들을 비교한 것을 알고 있지만 상당히 다르다. 완전히 갖춰진 메시징 시스템들은 당장 사용하는 경험을 줄 수 있다. 포장을 벗기고, 설정하고, 모든 복잡한 내용을 알고 있다면 바로 진행할 수 있다.
ZeroMQ는 그러한 시스템이 전혀 아니다. 그것은 프로그램을 통하여 사용할 수 있는 단순한 메시징 라이브러리이다. 그것은 기본적으로 당신의 메시지 시스템을 빠르게 만들 수 있는 흥미로운 소켓 인터페이스를 제공한다.

나비처럼 날아서, 벌처럼 쏘라

그렇다면 저수준의 버클리 소켓이 아닌 ZeroMQ를 사용하는 이유는 무었인가? 내 생각에 답은 '균형'이다. 당신은 구현이 편한 고수준 접근과 유연성 및 퍼포먼스가 좋은 저수준 접근을 동시에 원할 수 있다. 그렇지만, 확장성 있는 시스템을 만들 때 소켓을 그대로 쓰는 것은 어렵기도 하고 귀찮다. 고수준의 시스템은 원래 의도된 환경에서 사용되면 정말 좋지만, 핵심 요소를 바꾸기 어렵고 사용 편의성이 퍼포먼스를 깍아 먹기도 한다. 이것은 메시지 시스템에만 제한된 문제가 아니다. 우리는 앞의 딜레마를 웹프레임웍들에서도 발견할 수 있다. 그것이 '마이크로 프레임웍'들이 인기를 얻고 있는 이유이기도 하다.

나는 ZeroMQ가 이러한 고수준과 저수준의 갭을 완벽하게 메꿔 준다고 믿는다. 대체 어떤 기능들이 있는 걸까?

퍼포먼스

ZeroMQ는 정말 빠르다. 그것은 대부분의 AMQP(역자: Advanced Message Queuing Protocol)들 보다 단위가 다를 정도로 빠르다. 이러한 퍼포먼스는 다음과 같은 테크닉들 때문에 가능하다:

- AMQP처럼 과도하게 복잡한 프로토콜이 없다.
- 신뢰성 있는 멀티캐스트나 Y-suite IPC 전송 같은 효율적인 전송을 활용한다.
- 지능적인 메시지 묶음을 활용한다. 이것은 0MQ로 하여금 프로토콜 오버헤드뿐만 아니라 시스템 호출을 줄여서 TCP/IP를 효율적으로 활용하게 한다.

단순성

API는 믿을 수 없을 정도로 간단하다. 그렇기에 소켓 버퍼에 계속 '값을 채워' 주어야 하는 생 소켓 방식에 비교하면 메시지를 보내는 것이 정말로 단순하다. ZeroMQ에서는 그냥 비동기 send 호출을 부르기만 하면, 메시지를 별도의 스레드의 큐에 넣고 필요한 모든 일을 해준다. 이러한 비동기 특성이 있기에 당신의 애플리케이션은 메시지가 처리되기를 기다리며 시간 낭비하지 않아도 된다. 0MQ의 비동기 특성은 이벤트 중심의 프레임웍에도 최적이다.

ZeroMQ의 단순한 와이어 프로토콜은 다양한 전송 프로토콜이 사용되는 요즘에 적합하다. AMQP를 쓴다면 그 위에 또 다른 프로토콜 레이어를 쓴다는 것은 좀 이상하게 느껴진다. 0MQ는 메시지를 그냥 Blob으로 보기에 당신이 메시지를 어떻게 인코드하든 상관없다. 단순히 JSON 메시지들을 보내던지, 아니면 BSON, Protocol Buffers나 Thrift 같은 바이너리 방식 메시지들도 괜찮다.

확장성

ZeroMQ 소켓들은 저수준처럼 보이지만 사실은 다양한 기능들을 제공한다. 예를 들어 하나의 ZeroMQ 소켓은 복수의 접점을 가질 수 있으며 그들 간에 자동으로 메시지 부하 분산을 수행한다. 또는 하나의 소켓으로 복수의 소스에서 메시지들을 받아들이는 게이트 역할을 할 수

도 있다.

ZeroMQ는 '브로커없는 설계' 방식을 따르기에 전체의 실패를 초래하는 단일 위치가 존재하지 않는다. 이것과 앞에서 설명한 단순함 그리고 퍼포먼스를 조합하면 당신의 애플리케이션에 분산처리 기능을 넣을 수 있다.

ZeroMQ로 메시징 레이어 구현하기

다음 섹션에서 나는 ZeroMQ를 사용해서 어떻게 메시징 레이어를 설계하는지 보여주겠다. 예제 코드를 위해서 Brian Granger가 만든 훌륭한 파이썬 바인딩인 PyZMQ를 사용하겠다.

ZeroMQ 메시징 레이어 구현은 세 단계로 접근한다:

1. 전송 방식 정하기
2. 하부 구조 잡기
3. 메시징 패턴 선택하기

전송 방식 정하기

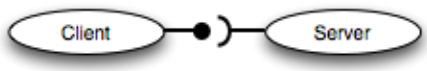
우선 전송 방식을 정하자. ZeroMQ는 4가지 다른 전송 방식을 제공한다:

1. *INPROC* 프로세스내 전송 모델
2. *IPC* 프로세스 간 전송 모델
3. *MULTICAST* PGM를 통한 멀티캐스트 (UDP로 싸여지기도 함)
4. *TCP* 네트워크 기반의 전송

TCP 전송은 대개 최선의 선택인데, 퍼포먼스가 괜찮고 안정적이다. 그렇지만, 서로 다른 머신간 통신이 아니라면 응답성을 더 좋게 하기위해 IPC나 INPROC 프로토콜을 살펴보는 것도 흥미롭다. MULTICAST 전송은 특별한 경우에 필요할 수 있다. 하지만 개인적으로 나는 확장을 할 때 어떻게 될지를 이해하는 것이 어려워서 멀티캐스트를 사용하는 것에 좀 조심스럽다. 얼마나 많은 멀티캐스트 그룹을 이 하드웨어와 저 하드웨어로 만들 수 있을지, 그리고 얼마나 많은 부하가 네트워크상의 스위치에 부과될 것인지 알아내는 것 같은 이슈들을 생각해 보라. 다른 전송 방식들은 모든 플랫폼에서 구현되지 않을 수도 있어서, 만약 당신의 코드가 크로스 플랫폼으로 동작하여야 한다면 TCP를 선택하는 것이 최선이다.

하부 구조 잡기

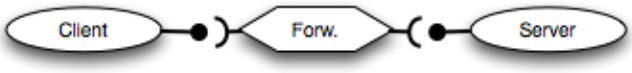
전송 방식을 정했으면 서로 다른 컴포넌트들이 서로 어떻게 연결될 것인지에 대해 생각해야 한다. 그것은 단순히 다음의 질문에 대한 답을 하는 것이다: “누가 누구에게 접속하는가?” 당신은 아마도 가장 안정적인 관계를 가진 것이 특정 포트를 열고 가장 가변적인 관계를 가진 것이 그것에 접속하기를 원할 것이다. 아래의 그림에서 우리는 서버가 어떻게 특정 포트를 열고, 클라이언트가 그것에 접속 하는지 묘사했다.



양 접점이 모두 비교적 가변적일 수 있어서 하나의 안정적인 접속점을 찾기 어려울 수도 있다. 만약 그렇다면, ZeroMQ가 제공하는 포워딩 디바이스(발송자)를 사용할 수 있다. 이 디바이스들은 서로 다른 두 포트를 열고 메시지를 한쪽에서 다른 쪽으로 포워딩 한다. 그렇게 함으로써, 포워딩 디바이스는 각 컴포넌트가 접속할 수 있는 안정적인 접점이 될 수 있다. ZeroMQ는 세 가지 종류의 디바이스들을 제공한다:

1. *QUEUE* 요청/답변(Request/Reply) 메시징 패턴을 위한 발송자
2. *FORWARDER* 발행/구독(Publish/Subscribe) 메시징 패턴을 위한 발송자
3. *STREAMER* 파이프라인의 단방향 메시징 패턴을 위한 발송자

우리는 아래의 그림에서 디바이스의 활용을 볼 수 있다. 여기에서 클라이언트와 서버는 서로 다른 포트를 열고 있는 발송자에 각각 접속한다. 이러한 디바이스를 사용하면 피어들의 리스트를 관리하지 않아도 되어서, 별도의 애플리케이션 로직이 필요 없게 된다.



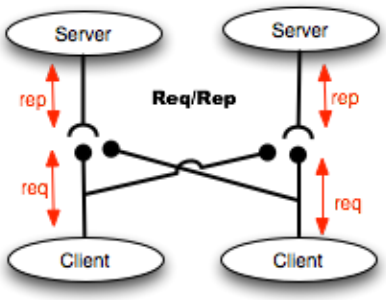
메시지 패턴을 선택하기

전 단계에서는 하부 구조를 정했지만, 메시지의 흐름을 명시하지 않았다. 이제 네트워크상의 각 컴포넌트 사이의 메시지 패턴에 대해 주의깊게 생각해볼 단계이다. 0MQ가 지원하는 패턴들은 다음과 같다:

1. *REQUEST/REPLY* 양방향, 부하 분산과 상태 기반
2. *PUBLISH/SUBSCRIBE* 다수의 구독자들에게 한번에 발행
3. *UPSTREAM / DOWNSTREAM* 파이프라인의 접점들에 단일 방향 데이터 배포
4. *PAIR* 두 피어들 간 배타적 커뮤니케이션

아래에서 좀 더 살펴보자.

요청과 답변



요청과 답변 방식은 아주 일반적이고 대부분의 서버에서 발견할 수 있다. 예를 들면: HTTP, POP 또는 IMAP이다. 이 패턴은 요청에 꼭 답변이 따라오는데 이를 위한 상태 정보를 가진다. 클라이언트는 소켓에 send()를 수행하여 요청을 시작하기에 REQ타입의 소켓을 사용한다. 서버는 REP타입의 소켓을 사용한다. 그것은 들어오는 요청을 읽기 위해서 recv()를 수행하여 시작하고, 그 후에 답변을 보낸다.

ZeroMQ는 단일 소켓에서 다수의 접점에 접속하는 것을 허용하여 이 패턴을 굉장히 단순화시켰다. ZeroMQ는 서로 다른 피어들의 요청을 자동으로 분산시킨다.

아래의 파이썬 코드는 REP소켓으로 5000번 포트를 오픈하는 서버를 만들 것이다. 그리고 그것은 들어오는 요청을 recv()로 받고 send()로 답변하는 과정을 반복할 것이다.

```
import zmq
context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://127.0.0.1:5000")

while True:
    msg = socket.recv()
    print "Got", msg
    socket.send(msg)
```

이 서버에 다수의 클라이언트가 접속하면 ZMQ소켓은 모든 요청을 동등하게 큐에 넣고 처리할 것이다. 만약 당신이 클라이언트들도 다수의 서버에 접속할 수 있기를 원한다면, 위의 코드를 가져와서 포트를 6000으로 바꾸고 그것을 별도의 서버로 실행시키자. 그 후에 다음의 클라이언트 코드는 양쪽 서버를 다 사용하게 한다:

```
import zmq
context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect("tcp://127.0.0.1:5000")
socket.connect("tcp://127.0.0.1:6000")

for i in range(10):
    msg = "msg %s" % i
    socket.send(msg)
    print "Sending", msg
    msg_in = socket.recv()
```

위의 코드는 모두 10개의 요청을 보내지만, 우리는 2개의 서로 다른 서버에 접속하고 있기에, 각 서버는 각각 5개의 요청들만 처리하면 된다. 정말 대단하지 않은가? 몇줄의 코드로 우리는 분산형 클라이언트/서버 모델을 만든 것이다.

만약 우리가 증가하는 요청들을 처리하기 위해서 추가 서버를 붙이려 하면 코드를 수정해야만 한다. 이것은 추가 서버로 요청 분산이 가능하다는 것을 알리기 위해 모든 클라이언트를 수정해야하기 때문에 아주 귀찮은 일이 될 수 있다.

이때가 바로 ZeroMQ의 디바이스가 필요한 상황이다. 클라이언트들로 하여금 다수의 서버에 곧바로 붙게 하는 대신, 하나의 포워딩 디바이스에 접속할 수 있다. 이 포워딩 디바이스는 모든 메시지를 접속된 서버들에게 재전달할 것이다.

클라이언트 출력의 예:

Sending msg 0
Sending msg 1
Sending msg 2
Sending msg 3
Sending msg 4

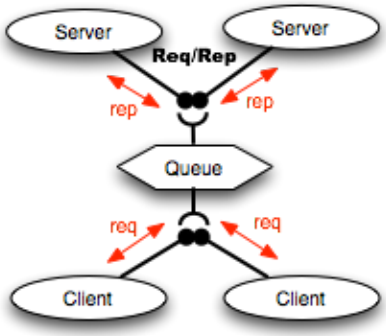
Sending msg 5
Sending msg 6
Sending msg 7
Sending msg 8
Sending msg 9

5000번 포트 서버 1의 출력 예:

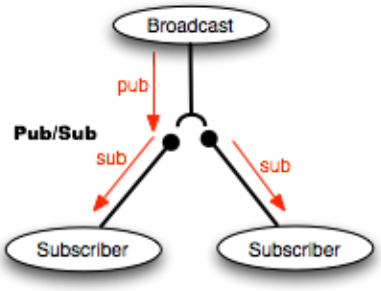
Got msg 0
Got msg 2
Got msg 4
Got msg 6
Got msg 8

6000번 포트 서버 2의 출력 예:

Got msg 1
Got msg 3
Got msg 5
Got msg 7
Got msg 9



발행과 구독



발행과 구독 패러다임은 지난 몇 해 동안 많은 관심을 끌어 왔었다. 푸쉬 메시지, XMPP 혹은 웹훅 같은 것들을 생각할 수 있다. 발행과 구독 패턴에서 컴포넌트들은 느슨하게 연결되어 있다. 이 방식은 구독자들에 대해서 걱정할 필요가 없기 때문에 확장에 크게 용이하다. 하지만 이 느슨한 연결은 완전히 이해하지 못하면 예상하지 못한 결과를 초래할 수도 있다. 발행과 구독 패러다임에 대한 좋은 비유는 라디오 방송국이다. 방송국에서 메시지를 발행하는 것은 특정 주파수를 통해서 보내는 것이다. 그 주파수를 구독하고 있는 청취자만 신호를 받게 된다. 그러나 또한 라디오처럼 방송이 끝난 후 채널을 맞추면 그것을 놓치게 되는 것이다.

다양한 메시지 패턴들이 하부 구조에는 비의존적이라는 것을 강조하고 싶다. 포트를 열고 접속한 피어들에게 발행하는 것이 가능지만, 또한 다수의 피어들에게 거꾸로 서버가 접속해 브로드캐스트하는 것도 가능하다. 전자는 라디오(누구나 청취할 수 있는)를 닮았고, 후자는 메가폰으로 사람들에게 소리를 지르는 것(선택된 그룹에게)과 유사하다. 양쪽 다 피어들은 구독하지 않는 것으로 메시지를 청취하지 않을 수 있다.

다음의 코드는 다섯 개의 축구 이벤트들에 대한 브로드캐스팅 서버를 어떻게 만들 수 있는지 보여준다:

```
import zmq
from random import choice
context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://127.0.0.1:5000")

countries = ['netherlands','brazil','germany','portugal']
events = ['yellow card', 'red card', 'goal', 'corner', 'foul']

while True:
    msg = choice( countries ) +" "+ choice( events )
    print "->",msg
    socket.send( msg )
```

이 서버는 서로 다른 나라들에 대한 계속되는 축구 이벤트들을 생성하여 PUB소켓 타입을 통해 푸쉬한다. 다음과 같은 출력이 나올 것이다:

-> portugal corner
-> portugal yellow card
-> portugal goal
-> netherlands yellow card
-> germany yellow card
-> brazil yellow card
-> portugal goal

-> germany corner
...

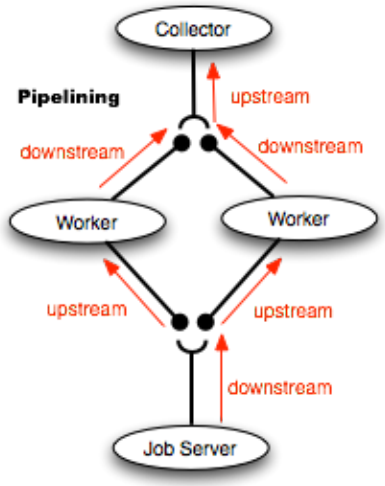
만약 당신이 네덜란드와 독일에 대한 이벤트들에만 관심이 있다면, 그 특정 메시지만 구독하는 클라이언트를 만들 수 있다:

```
import zmq

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect("tcp://127.0.0.1:5000")
socket.setsockopt(zmq.SUBSCRIBE, "netherlands")
socket.setsockopt(zmq.SUBSCRIBE, "germany")

while True:
    print  socket.recv()
```

클라이언트는 SUB 소켓을 만들어서, 5000번 포트의 브로드캐스트 서버에 접속하고 'netherlands' 혹은 'germany'로 시작하는 메시지들만 구독한다. 결과는 다음과 같을 것이다:

netherlands red card
netherlands goal
netherlands red card
germany foul
netherlands yellow card
germany foul
netherlands goal
netherlands corner
germany foul
netherlands corner
...

파이프라인 만들기

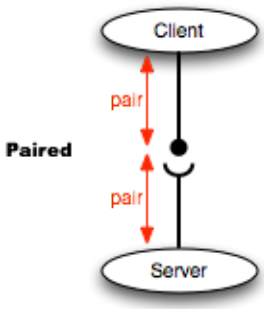


파이프라인 패턴은 요청/응답 패턴과 상당히 유사하다. 차이점은 응답이 요청한 사람에게 가는 것이 아니라 파이프로 내려보내 진다는 것이다. 이 방식은 병렬 데이터 처리가 필요한 경우 자주 사용된다. 예를 들어 얼굴 인식 시스템이 있다고 해보자. 실제로 인식 작업을 하는 것(워커)가 있고 그쪽으로 인식 대상 이미지들을 분배해 주는 것(잡 서버)가 있다. 워커는 인식 작업이 끝나면 파이프라인으로 내려보내 결과를 모으는 쪽(컬렉터)으로 보내준다.

왼쪽의 그림에서 워커는 *UPSTREAM* 소켓에서 메시지를 받고 작업이 끝나면 *DOWNSTREAM* 쪽으로 보낸다. 메시지들을 두 개의 다른 소켓 타입들 사이에서 전달하는 것이다.

잡 서버는 단순히 일들을 하나의 소켓에 붙은 여러 접점으로 내려보내기만 할 수 있다. ZeroMQ와 최신의 PyZMQ는 복사 없이 메시지를 전달할 수 있다. 이것은 IO 싸이클의 낭비 없이 큰 메시지들을 보낼 때 아주 좋다.

짝지어진 소켓



짝지어진 소켓은 통신이 양방향인 점이 정규 소켓과 비슷하다. 소켓과 연결된 상태 정보는 없고, 오직 하나의 피어만을 가진다. 대부분의 일반적인 문제들은 앞에서 언급된 패턴들로 해결되기에, 문제 해결을 단순히 하기 위해 이 패턴을 선택하기 전에 앞의 패턴들을 먼저 검토하기를 추천한다.

왼쪽의 그림은 짝지어진 소켓의 구조를 보여준다. 서버는 특정 포트를 열고 클라이언트는 거기에 접속한다. 빨간 선들은 메시지의 흐름을 의미하는데, 양쪽 접점은 *PAIR* 소켓을 사용하고 메시지는 양 방향으로 오고 갈 수 있다.

다음의 코드는 이것을 어떻게 구현하는지 보여준다. 하나의 포트를 한쪽에서 오픈한다.

```
import zmq
```

```
context = zmq.Context()
socket = context.socket(zmq.PAIR)
socket.bind("tcp://127.0.0.1:5555")
```

그리고 다른 쪽은 거기로 접속한다.

```
import zmq
context = zmq.Context()
socket = context.socket(zmq.PAIR)
socket.connect("tcp://127.0.0.1:5555")
```

ZeroMQ 그리고 미래

이 글에서 나는 ZeroMQ에 관한 간략한 소개를 하였다. 나는 당신이 지금쯤 이 작지만 위대한 라이브러리에 대한 내 생각을 공유했기를 바란다. 이 라이브러리가 소소하게 느껴질지는 몰라도 '새로운 메시지 레이어'가 되겠다는 큰 비전을 가지고 있고, 생각해보면 그 목표는 그다지 무리한 것도 아니다. 확장성의 이슈들은 대부분 통신과 이식성에 관한 이슈들이고, ZeroMQ는 이 문제들을 해결해 줄 수 있다.

예를 들어 Redis, Cassandra, TokyoTyrant, Postgres, MongoDB, DabbleDB, CouchDB, HBase, 그 외의 어떠한 최신 데이터베이스도 당신의 요구에 맞지 않아서 새로운 것을 만들기로 했다고 가정하자. 당신은 당신의 데이터에 딱 맞는 메모리 구조, 그리고 정말 빠른 인덱서도 만들었다. 이제 개별 클라이언트들이 통신할 수 있는 메시지 레이어만 있으면 된다. 가급적 다양한 프로그래밍 언어를 지원하고 클러스터링 기능도 있으면 좋을 것이다. 물론 그 모든 것들을 당신 스스로 만들 수도 있겠지만, 정말 할 일이 많을 것이다.

직접 만드는 것 보다 간단한 해결책은 당신의 데이터베이스를 ZeroMQ서버로 만들고 메시지 프로토콜(예를 들어 JSON)을 선택하는 것이다. 지금까지 본 것처럼, 그러한 기능들을 ZeroMQ로 구현하는 것은 정말 쉽고, 무엇보다도 ZeroMQ가 메시지들을 전달하는 방식 때문에 즉각적인 확장성을 가질 수 있다. 그리고 다양한 클라이언트들이 당신의 서버와 통신하게 하는 것도 정말 쉽다. 기본적으로 당신이 할 일은 지원 가능한 15개 언어 중 하나를 선택하고, 같은 메시지 프로토콜을 사용하기만 하면 되는 것이다. 현재 다음의 언어들을 위한 ZeroMQ 바인딩이 지원되고 있다: Ada, C, C++, Common Lisp, Erlang, Go, Haskell, Java, Lua, .NET, OOC, Perl, PHP, Python 그리고 Ruby.

ZeroMQ는 우리들의 컴포넌트들이 어떻게 연결되는지에 관한 새로운 방식이 될 수 있다. ZeroMQ의 가능성을 이해하고 있는 사람의 예로서 Mongrel2 프로젝트를 진행하는 Zed Shaw를 들 수 있겠다. 당신은 Mongrel2를 일반적인 HTTP클라이언트와 ZeroMQ 컴포넌트의 차이를 매꾸는데 사용할 수 있다. 당신이 웹소켓, Comet 혹은 플래쉬 기반의 소켓을 사용해보았다면 이 프로젝트가 얼마나 대단한지 바로 알 수 있을 것이다. 이런 구현 방식의 다른 예는 페이스북의 BigPipe인데, ZeroMQ에 연결된 서로 다른 컴포넌트들에 의해 페이지릿이 투명하게 생성될 수 있다.