Libzmq Chapter 1 - Basics

From 탱이의 잡동사니

Contents

Fixing the World

ZeroMQ 를 어떻게 설명할 수 있을까? 우리들 중 누군가는 끝내주는 기능들에 대해서 이야기를 하고 한다. 스테로이드를 맞은 소켓, 라우팅 기능이 있는 우편함, 빠른 속도!

Starting Assumptions

우리는 최소한 여러분이 최소한 ZeroMQ 3.2 버전 이상을 사용하고 있다고 가정한다. 그리고 Linux 혹은 그와 상응하는 운영체제를 사용하고 있다고 가정한다. 또한, 여러분이 더도 말고, 덜도 말고 딱 C example code 를 읽을 수 있다고 가정한다. 그리고 앞으로 계속해서 나올 PUSH 혹은 SUBSCRIBE 라는 말이 때때로 ZMQ_PUSH 혹은 ZMQ_SUBSCRIBE 라는 것을 이해할 수 있을 것이라고 가정한다.

Getting the Examples

예제 코드들은 아래 github 저장소에서 확인할 수 있다. 가장 쉽게 예제코드를 다운받을 수 있는 방법은 아래의 명령어로 저장소를 복사하는 것이다.

```
git clone --depth=1 https://github.com/imatix/zguide.git
```

다음, examples 의 하위 디렉토리를 살펴보자. 각각의 언어별로 예제가 있는 것을 확인할 수 있을 것이다. 만약 여러분이 사용하는 언어의 예제가 없다면 submit a translation 할 것을 강추한다. 이것이 바로 이 문서가 어떻게 여러 사람들에게 유용할 수 있는지의 이유이다. 모든 예제들은 MIT/X11 라이센스를 가진다.

Ask and Ye Shall Receive

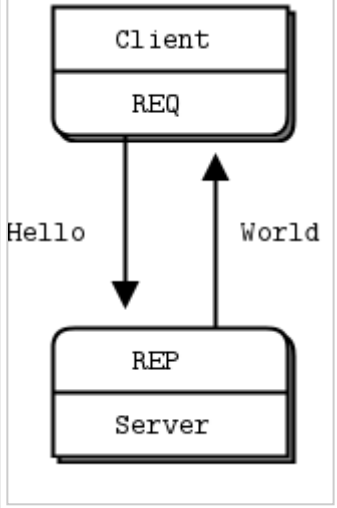
먼저 아래 코드를 보자. Hello World 예제부터 시작해볼 것이다. client 와 server 를 만들고, client 가 "Hello" 라고 server 에게 보내면, server 는 "World"라고 대답할 것이다. 여기 C 로 짜여진 서버가 있다. 포트 번호 5555 를 이용하여 ZeroMQ socket 을 열고, request 를 수신하고, "World" 라는 응답을 준다.

```c
// hwserver.c
// Hello World server

#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>
#include <string.h>

int main(int argc, char** argv)
{
    // Socket to talk to clients
    void* context;
    void* responder;
    int ret;

    context = zmq_ctx_new();
    assert(context != NULL);

    responder = zmq_socket(context, ZMQ_REP);
    assert(responder != NULL);

    ret = zmq_bind(responder, "tcp://*:5555");
    assert(ret == 0);

    while(1) {
        char buffer[10];

        memset(buffer, 0x00, sizeof(buffer));
        zmq_recv(responder, buffer, sizeof(buffer) - 1, 0);
        printf("Received message. message[%s]₩n", buffer);

        sleep(1);   // Do some 'work'
        zmq_send(responder, "World", 5, 0);
    }

    return 0;
}
```

아래는 client 쪽 소스내용이다.

```c
// hwclient.c
// Hello World client
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>

int main(int argc, char** argv)
{
    void* ctx;
    void* req;
    int req_cnt;
    int ret;

    printf("Connecting to hello world server..₩n");

    ctx = zmq_ctx_new();
    assert(ctx != NULL);
```

```
    req = zmq_socket(ctx, ZMQ_REQ);
    assert(req != NULL);

    zmq_connect(req, "tcp://localhost:5555");

    for(req_cnt = 0; req_cnt != 10; req_cnt++) {
        char buffer[100];

        printf("Sending Hello %d..₩n", req_cnt);
        sprintf(buffer, "Hello");
        zmq_send(req, buffer, strlen(buffer), 0);

        memset(buffer, 0x00, sizeof(buffer));
        ret = zmq_recv(req, buffer, sizeof(buffer) - 1, 0);
        printf("Received message. message[%s], cnt[%d], received_size[%d]₩n", buffer, req_cnt, ret);
    }

    zmq_close(req);
    zmq_ctx_destroy(ctx);

    return 0;
}
```



request-reply

REQ-REP socket pair 는 서로 굉장히 밀집되어 이뤄진다. client 는 루프를 돌면서 zmq_send() 를 호출하고, 이어 zmq_recv() 를 호출한다. 만약 한번에 두개 이상의 메시지를 전송/수신하고자 한다면 zmq_send() 혹은 zmq_recv()의 return code 는 -1(error) 를 반환할 것이다. 마찬가지로, server 쪽에서도 순서에 맞춰 zmq_recv() 후에 zmq_send() 를 호출한다.

ZeroMQ 는 C 를 reference language 로 사용하며, 앞으로 나올 예제들의 main language 가 될 것이다.

소스를 보면 알겠지만, 믿을 수 없을 정도로 굉장히 쉽고 간단하다. 게다가 앞서 이야기 했듯이, ZeroMQ 는 굉장한 힘을 가지고 있다. 하나의 server 에 1000 개 이상의 client 를 연동할 수도 있고, 정말 빠르게 동작할 것이다. 한번 재미삼아 client 를 serer 보다 먼저 실행시켜 보라. 잘 동작하지 않는가? 잠시 이게 무슨 의미인지 생각해보라.

잠시 위의 두개의 프로그램이 정확히 어떤 일을 하는지 알아보도록 하자. 작업을 위해 ZeroMQ context 를 생성하고, socket 을 생성했다. 지금은 무슨말인지 몰라도 된다. 뒷부분에가면 전부 이해가 될 것이다. 서버는 REP(reply) 소켓을 5555 포트에 bind 를 하고, loop 를 돌면서 요청을 기다리고, 요청이 들어오면 바로 응답을 전송한다. 클라이언트 는 서버로 요청을 전송하고, 응답을 수신한다.

만약 여러분이 서버를 죽이고(Ctrl + c), 재시작을 할 경우, 클라이언트가 정상적으로 작동하지 않을 수도 있다. 사실, 오류를 해결하고 복구하는 일은 그리 쉽지가 않는 일이다. 신뢰성있는 request-reply 연결을 만드는 것은 매우 복잡한 일이며, Chapter 4 - Reliable Request-Reply Patterns 에서 다룰 것이다.

실은 화면에는 안보이지만 짧은 코드 라인으로는 상상할 수 없는 많은 일들이 벌어지고 있다. 오류없이 잘 작동하고, 빠르고, 안정적으로 동작한다. 우리가 살편본 패턴은 request-reply pattern 이었다. 아마도 가장 간단한 ZeroMQ 사용법 중의 하나일 것이다. 이 패턴은 RPC 모델과 client/server 모델과 부합되는 패턴이다.

A Minor Note on String

ZeroMQ 는 소켓에 입력되는 데이터에 대해서 데이터의 길이(크기) 말고는 아무것도 알지 못한다. 그 뜻은, 다른 프로그램에서 수신한 데이터를 읽을 수 있도록 하는 것은 순전히 여러분의 책임이라는 뜻이다. 복잡한 데이터 타입을 다루는 특수 라이브러리(프로토콜 버퍼와 같은)부터 간단한 string 까지 모두 신경을 써야 한다.

C 와 같은 언어들에서는 string 의 끝을 NULL byte 로 표현한다. 만약 "Hello"라는 문자열을 NULL byte 와 함께 전송하고자 한다면 다음과 같이 해야 한다.

```
zmq_send(requester, "Hello", 6, 0);
```

하지만 만약 다른 프로그램 언어에서 string 을 전송한다면 NULL byte 가 포함되지 않을지도 모른다. 예를 들어, 위와 같은 문자열을 python 에서는 아래와 같이 전송한다.

```
socket.send("Hello")
```

이제 실제 데이터가 어떤식으로 저장되는지를 확인해보자.

A ZeroMQ string

그리고 만약 이 데이터를 C 프로그램에서 수신한다면, 처음에는 마치 string 처럼 보일 것이다. 그리고 어쩌면 실제 string 처럼 동작할 지도 모른다(운이 좋아서 문자열에 뒤에 NULL byte가 붙어있을지도 모른다). 하지만 이는 C 에서 사용하는 string 형식이 아니다. 때문에 만약 Server 와 Client 에서 string 형식에 대해 사전에 협의 없다면, 프로그램이 제대로 작동하지 않을 수도 있다.

간단하게, C 에서 ZeroMQ 를 통해서 string 데이터를 수신할 경우, 문자열의 끝에 문자열 종료문자가 있는지/없는지를 정확히 예측할 수 없다. 그렇기 때무에 매번 string 을 읽을 때마다, 새로이 버퍼를 할당하고, 버퍼에 string 종료 문자를 염두해서 크기를 살짝 더 키우는 것도 잊으면 안된다.

자, 이쯤에서 한가지 규칙을 정하자. ZeroMQ string 은 길이-지정 형태이며, 자체적으로 NULL 종료 문자를 포함하지 않은 상태로 전송한다. 여기 c 에서 ZeroMQ string 을 수신할 때 어떻게 하는지 예시를 나타내었다.

```
// Receivce ZeroMQ string from socket and convert into C string
// Chops string at 255 chars, if it's longer
static char*
s_recv(void* socket) {
  char buffer[256];

  int size = zmq_recv(socket, buffer, 255, 0);
  if(size == -1) {
    return NULL;
  }

  if(size > 255) {
    size = 255;
  }

  buffer[size] = 0;
  return strdup(buffer);
}
```

쉽고, 간단하다. 이와 비슷하게 s_send() 함수도 만들 수 있다. 앞으로의 예제를 위해 이런 간단 함수들을 모아 놓은 파일을 따로 만들어놓았다. 아래의 링크에서 확인이 가능하다

- https://github.com/imatix/zguide/blob/master/examples/C/zhelpers.h

Version Reporting

ZeroMQ 는 매우 자주 버전이 업그레이드 되어 배포된다. 만약 사용중 문제에 부딪혔다면, 아마도 최신 버전에서는 이미 문제가 해결되어 있을 것이다. 때문에 현재 사용중인 ZeroMQ 의 정확한 버전을 안다는 것은 꽤나 유용할 수도 있다.

여기에 버전 정보를 확인하는 프로그램이 있다.

```
// version.c
// Report OMQ version

#include <zmq.h>

int main(int argc, char** argv)
{
    int major, minor, patch;
    zmq_version(&major, &minor, &patch);
    printf("Current OMQ version is %d.%d.%dWn", major, minor, patch);

    return 0;
}
```

다음과 같이 정보를 나타낸다.

```
$ ./main

Current OMQ version is 3.2.5
```

Getting the Message Out

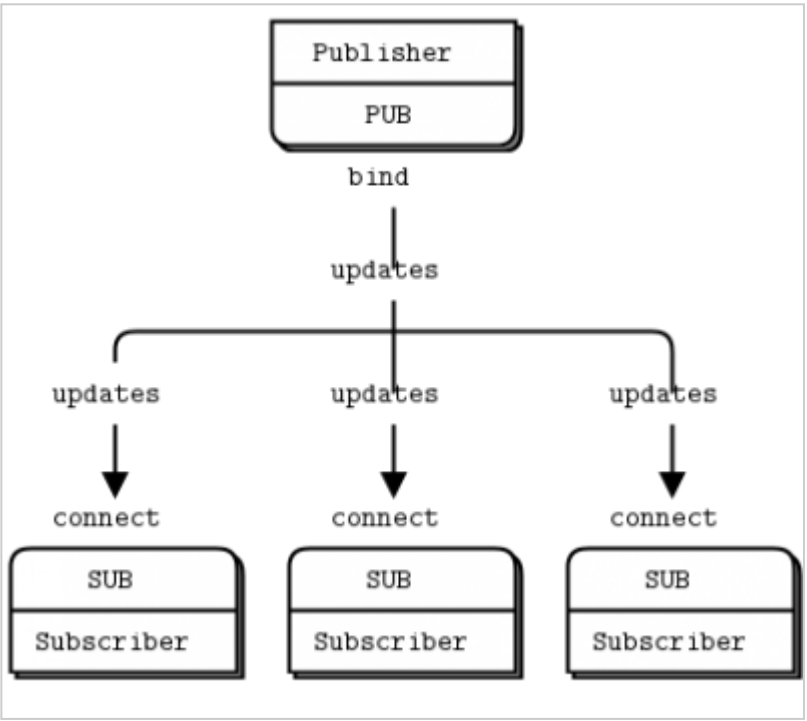
두번째로 알아볼 패턴은 one-way data distribution 이다. server 는 데이터를 입력하고 client 는 데이터를 수신하는 방식이다. 예제를 통해 알아보자. 서버는 지속적으로 zip 코드, 온도, 습도 등의 날씨 정보를 업데이트한다. 진짜 날씨처럼 보이도록 랜덤 변수로 변화를 주도록 한다.

먼저 서버쪽 소스를 보자.

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

#include "zhelpers.h"

int main(int argc, char** argv)
{
    // Prepare our context and publisher
    void* context;
    void* publisher;
    int ret;

    context = zmq_ctx_new();
    publisher = zmq_socket(context, ZMQ_PUB);
    ret = zmq_bind(publisher, "tcp://*:5556");
    assert(ret == 0);

    // Initialize random number generator
    srandom((unsigned)time(NULL));
    while(1) {
        // Get value that will fool the boss
        int zipcode, temperature, relhumidity;
        zipcode = randof(100000);
        temperature = randof(215) - 80;
        relhumidity = randof(50) + 10;

        // Send mesage to all subscribers
        char update[20];
        sprintf(update, "%05d %d %d", zipcode, temperature, relhumidity);
        s_send(publisher, update);
    }
    zmq_close(publisher);
    zmq_ctx_destroy(context);

    return 0;

}
```

클라이언트쪽 소스이다. 기본값으로 10001 NewYork zipcode 를 설정하여 10001 zipcode 를 가진 데이터 모두를 수집한다.

```
// wuclient.c
// Weather updte client
// Conntes SUB socket to tcp://localhost:5556
// Collects weather updates and finds avg temp in zipcode

#include "zhelpers.h"

int main(int argc, char** argv)
{
    void* context;
    void* subscriber;
    int ret;

    // Socket to talk to server
    printf("Collecting updates from weather server..₩n");
    context = zmq_ctx_new();
    subscriber = zmq_socket(context, ZMQ_SUB);
    ret = zmq_connect(subscriber, "tcp://localhost:5556");
    assert(ret == 0);

    // Subscribe to zipcode, default is NYC, 10001
    char* filter = (argc > 1)? argv[1]:"10001";
    ret = zmq_setsockopt(subscriber, ZMQ_SUBSCRIBE, filter, strlen(filter));
    assert(ret == 0);

    // Process 100 updates
    int update_nbr;
    long total_temp = 0;
    for(update_nbr = 0; update_nbr < 100; update_nbr++) {
        char* string = s_recv(subscriber);

        int zipcode, temperature, relhumidity;
        sscanf(string, "%d %d %d", &zipcode, &temperature, &relhumidity);
        total_temp += temperature;
        free(string);
    }
    printf("Average temperature for zipcode '%s' was %dF₩n", filter, (int)(total_temp / update_nbr));

    zmq_close(subscriber);
    zmq_ctx_destroy(context);

    return 0;
}
```



Publish-Subscribe

SUB socket 을 사용한다면 반드시 위의 client 소스에서처럼 zmq_setsockopt() 함수와 SUBSCRIBE 옵션을 이용하여 subscription 을 설정해야한다는 것을 알아두자. 만약 어떠한 subscription 도 설정하지 않았다면, 아무런 메시지도 수신하지 못할 것이다. 처음이라면 누구나 흔히들 겪는 실수이다. subscriber 는 여러개의 subscription 들을 함께 추가함으로써, 한번에 여러개의 subscription 들을 설정할 수 있다. 반대로 subscriber 는 특정 subscription 을 제거할 수도 있다. subscription 은 반드시 print 가능한 string 일 필요는 없다. 자세한 내용은 zmq_setsockopt()[1]를 참고하면 된다.

PUB-SUB socket pair 는 비동기식으로 작동한다. client 는 loop에서 zmq_recv() 만을 수행한다. 만약 SUB socket 으로 메시지를 송신하려고 한다면 에러가 발생할 것이다. 마찬가지로, 서버쪽에서는 zmq_send() 로 데이터를 송신할 뿐이다. PUB socket 으로 zmq_recv() 를 하면 안된다.

이론적으로 ZeroMQ socket 에서는 어느쪽에서 bind 를 하건, connect 을 하건 상관이 없다. 하지만, 실제로는 약간 다른점이 있는데, 나중에 설명을 하도록 하겠다. 지금은, PUB에서 bind 를 하고, SUB 에서 connect를 한다고 알아두자.

PUB-SUB 소켓에 대해 한가지 더 알아두어야 할 점이 있다. 정확히 언제부터 메시지를 읽어들이는지 알 수 없다는 것이다. 설령 subscriber 를 실행한 다음, 나중에 publisher 를 실행한다고 하더라도, **subscriber 는 publisher 에서 송신하는 첫번째 메시지를 받지는 못한다**. 왜냐하면 subscriber 는 publisher 에게 connect 를 하는 시간이 있기때문에(아주 짧다고는 하지만 완전한 0이 아니기에) publisher 가 송신하는 첫번째 메시지를 수신할 수 없는 것이다.

이 "slow joiner" 현상 때문에, 때때로 사람들에게 왜 상세하게 설명을 해야 하는 경우가 있다. ZeroMQ가 비동기 I/O 라는 것을 기억하자.

여기 두 개의 노드(SUB/PUB)가 있고, 다음과 같이 동작한다고 가정해보자.

- Subscriber 는 다른쪽 endpoint 에 접속하여 메시지를 수신하고, 수신한 메시지의 갯수를 세아린다.
- Publisher 는 bind 후, 접속을 받아들이며, 즉시 1,000 개의 메시지를 전송한다.

그리고, subscriber는 아무런 메시지를 수신하지 못할 것이다. 맞다. 정확한 filter 를 설정해야 한다. filter 를 설정한 다음, 다시 시도를 해보아도 여전히 아무런 메시지를 수신하지 못할 것이다.

하나의 TCP connection 을 만들기 위해서는 handshaking 과 네트워크와 목적지와의 사이에 있는 홉 수에 따라 수 밀리세컨드 정도가 소요될 수 있다. 그 시간동안이면, ZeroMQ는 많은 수의 메시지를 전송할 수 있다. 연결 설정에 5 밀리초가 걸린다고 하고, 1초에 백만개의 메시지를 전송할 수 있는 링크라고 가정해보자. 이런 상황에서 publisher 는 천개의 메시지를 전송하는데 1 밀리초가 소요되고, subscriber 는 publisher 에 연결을 하는 데에만 5 밀리 초 가 소요 것이다.

Chapter 2 -Sockets and Patterns 에서 publisher 와 subscriber 를 어떻게 동기화 시키는지와, 이를 통해 publisher 에서 subscriber의 connection 이후에 데이터를 전송하도록 하는 방법을 살펴볼 것이다. 물론 publisher 에 slep과 같이 delay 를 주는 간단한 방법이 있다. 하지만 절대로 실제 application 에서 해서는 안된다. 속도가 정말로 느려질 것이다. 그러니 Chapter 2 - Socket and Patterns 에서 다른 방법을 살펴보도록 하자.

동기화를 하는 다른 방법으로 간단하게 전송되는 데이터 흐름이 무한하고, 시작과 끝이 없다고 생각하는 것이다. 그리고 subscriber 로 하여금, 메시지 전송의 시작과 끝에 의미를 두지 않도록 하는 것이다. 우리가 만든 날씨 client 가 바로 그 예이다.

client 는 zip code 를 subscriber 를 하며, 100 개의 업데이트를 메시지를 수신한다. 그 말인즉슨, 약 천만개 이상의 업데이트 메시지가 전송되었다는 뜻이다(zip code 가 랜덤이기에). client 를 먼저 시작한 뒤, server 를 시작할 수 있다. 잘 작동할 것이다. 종종 원하는 대로 서버를 재시작할 수도 있겠지만, 그래도 client 는 잘 작동할 것이다. client 시작 후, 100개의 메시지가 모두 모이면, 평균을 계산하고, 종료한다.

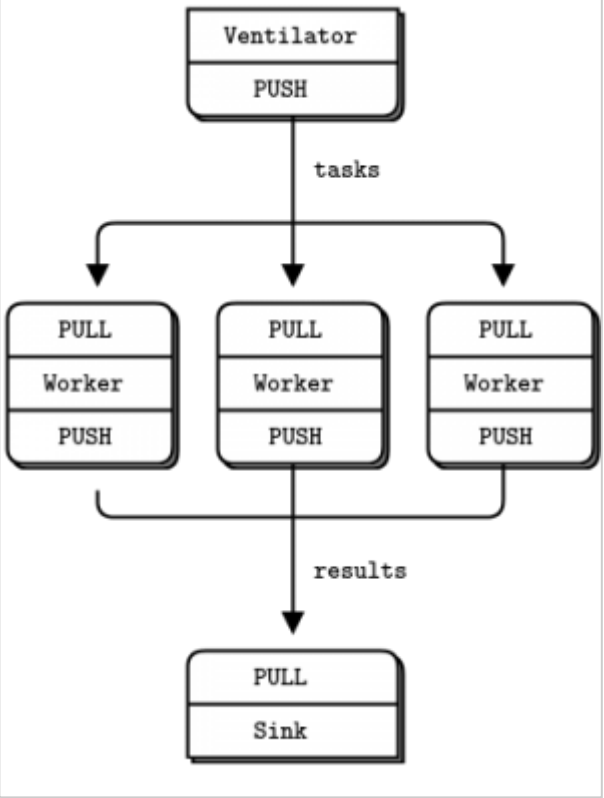
publish-subscriber(PUB-SUB) pattern 에서 몇가지 중요한 점은 다음과 같다.

- Sbuscrier 는 여러번의 connect 를 함으로써 하나 이상의 publisher 에 접속할 수 있다.
- 만약 publisher 에게 접속한 subscriber 가 하나도 없다면, 그냥 모든 메시지를 drop 할 것이다.
- 만약 TCP 와 subscriber 가 느리다면, 메시지들은 publisher 에서 Queueing 된다. 나중에 publisher 에서 "high-water mark" 를 통해 어떻게 이를 보호하는지 확인해볼 것이다.
- ZeroMQ v3.x 부터, tcp:// 혹은 ipc:// 를 통해 접속이 이루어지는 경우, filtering 은 publisher 쪽에서 발생하게 된다. epgm:// 프로토콜의 경우, subscriber 쪽에서 발생하게 된다. ZeroMQ v2.x 의 경우, 모든 filtering 은 publisher 에서 발생한다.

다음은 프로그램을 실행했을 때 걸린 시간이다. 약 천만개의 메시지가 발생했다.

```
$ time ./wuclient
Collecting updates from weather server..
Average temperature for zipcode '10001' was 28F

real    0m4.484s
user    0m0.000s
sys     0m0.008s
```

Divide and Conquer



Parallel Pipeline

조금전의 예제처럼 슈퍼컴퓨터 시뮬레이터를 만들어 보자. 이번에 만들 슈퍼 컴퓨터는 일반적인 병렬 컴퓨터와 같은 일을 수행할 것이다.

- Ventilator 는 입력된 작업을 병렬로 처리할 수 있도록 한다.
- 여러개의 Worker 들이 실제 작업을 처리한다.
- Sink 는 Worker 들로 부터 작업결과물을 종합한다.

실제적으로는 worker 들은 일종의 박스 안에서 빠른 속도를 연산 작업을 한다(아마도 GPU 등을 이용한). 100 개의 작업을 생성할 것이며, 각각의 메시지마다 worker 들에게 일정 시간의(msec) sleep 을 하도록 하게 할 것이다.

vantilator 쪽 소스이다.

```c
// taskvent.c
// Task ventilator
// Binds PUSH socket to tcp://localhost:5557
// Sends batch of tasks to workers via that socket

#include "zhelpers.h"

int main(int argc, char** argv)
{
    void* context;
    void* sender;
    void* sink;
    int task_nbr;
    int total_msec;
    int workload;

    context = zmq_ctx_new();

    // Socket to send messages on
    sender = zmq_socket(context, ZMQ_PUSH);
    zmq_bind(sender, "tcp://*:5557");

    // Socket to send start of batch messages on
    sink = zmq_socket(context, ZMQ_PUSH);
    zmq_connect(sink, "tcp://localhost:5558");

    printf("Press Enter when the workers are ready: ");
    getchar();
    printf("Sending tasks to workers...Wn");

    // The first message is "0" and signals start of batch
    s_send(sink, "0");

    // Initialize random number generator
    srandom((unsigned)time(NULL));

    // Send 100 tasks
    total_msec = 0;
    for(task_nbr = 0; task_nbr < 100; task_nbr++) {

        // Random workload from 1 to 100 msecs
        workload = randof(100) + 1;
        total_msec += workload;
        char string[10];
```

```
        sprintf(string, "%d", workload);
        s_send(sender, string);
    }

    printf("Total expected cost: %d msecWn", total_msec);

    zmq_close(sink);
    zmq_close(sender);
    zmq_ctx_destroy(context);

    return 0;
}
```

이번에는 worker 쪽 소스이다. 메시지를 수신하면, 주어진 시간만큼 sleep 후, 완료 신호를 전송한다.

```
// taskworker.c
// 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

#include "zhelpers.h"

int main(int argc, char** argv)
{
    // Socket to receive messages on
    void* context;
    void* receiver;
    void* sender;

    context = zmq_ctx_new();
    receiver = zmq_socket(context, ZMQ_PULL);
    zmq_connect(receiver, "tcp://localhost:5557");

    // Socket to send messages to
    sender = zmq_socket(context, ZMQ_PUSH);
    zmq_connect(sender, "tcp://localhost:5558");

    // Process tasks forever
    while(1) {
        char* string = s_recv(receiver);
        printf("%s.", string);  // Show progress
        fflush(stdout);
        s_sleep(atoi(string));  // Do the work
        free(string);
        s_send(sender, "");     // Send results to sink
    }

    zmq_close(receiver);
    zmq_close(sender);
    zmq_ctx_destroy(context);

    return 0;
}
```

sink 쪽 소스이다. 100 개의 작업 결과를 수집한 뒤, 전체 작업 시간을 계산한다. 이를 이용하여 작업이 실제로 병렬로 처리되었는지를 확인할 수 있다.

```
// tasksink.c
// Task sink
// Binds PULL socket to tcp://localhost:5558
// Collects results from workers via that socket

#include "zhelpers.h"

int main(int argc, char** argv)
{
    // Prepare our context and socket
    void* context;
    void* receiver;
    char* string;
    int task_nbr;
    int64_t start_time;

    context = zmq_ctx_new();
```

```
    receiver = zmq_socket(context, ZMQ_PULL);

    zmq_bind(receiver, "tcp://*:5558");

    // Wait for start of batch
    string = s_recv(receiver);
    free(string);

    // Start our clock now
    start_time = s_clock();

    // Process 100 confirmations

    for(task_nbr = 0; task_nbr < 100; task_nbr++) {
        char* string = s_recv(receiver);
        free(string);
        if((task_nbr / 10) * 10 == task_nbr) {
            printf(":");
        }
        else {
            printf(".");
        }

        fflush(stdout);
    }

    // Calculate and report duration of batch
    printf("Total elapsed time: %d msec₩n", (int)(s_clock() - start_time));

    zmq_close(receiver);
    zmq_ctx_destroy(context);

    return 0;
}
```

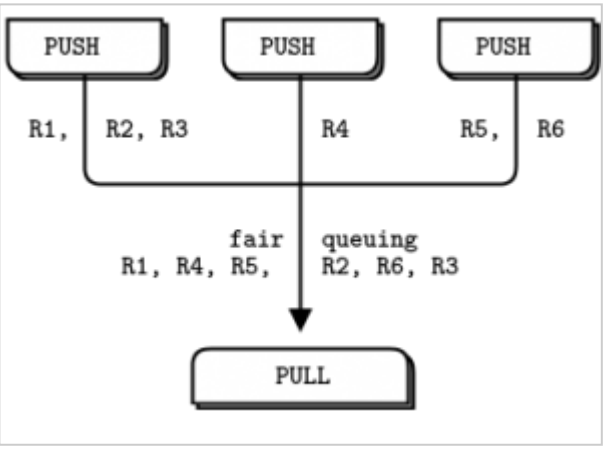
다음은 결과 내용이다.

- 1 worker : Total elapsed time: 5580 msec
- 2 workers : Total elapsed time: 2686 msec
- 4 workers : Total elapsed time: 1340 msec
- 8 workers : Total elapsed time: 743 msec

위의 코드를 조금 더 자세히 들여다 보자.

- worker 들은 ventilator 에 upstream 으로 접속하고, sink 에는 downstream 으로 접속한다. 이 뜻은, worker 들은 유동적으로 추가할 수 있다는 뜻이다. 만약 worker 들이 endpoint 에서 충돌하게 된다면, 어쩌면 더 많은 ventilator/sink 가 필요하게 될 것이다. 이번 예제에서는 ventialtor 와 sink 는 정적인 부분이라고 가정하고, worker 들은 동적인 부분이라고 가정했다.
- 예제에서 모든 worker 들이 동시에 작업을 수행할 수 있도록 동기화를 했다. 이러한 동기를 위한 batch signal 전송 기법은 ZeroMQ 에서는 매우 일반적인 기법이며, 이를 대체할만한 다른 쉬운 방법은 없다. zmq_connect() 함수의 실행에는 어느정도의 시간이 소요된다. 때문에, worker 들이 ventilator 에 접속할 때, 첫번째 worker 가 접속 성공이후, 작업 메시지 전부를 수신받아버릴 수도 있다. 간단히, 이런 동기화 기법을 사용하지 않은채로 프로그램을 동작시킬 경우, 프로그램이 병렬로 작동하지 않을 것이다. ventilator 에서 wait 하는 부분을 삭제하고 프로그램을 동작시켰을 때, 어떤 현상이 나타나는지 확인해보라.
- ventilator 의 PUSH socket 은 작업을 worker 들에게 분배를 한다(이미 모든 worker 들이 접속해있는 상태라고 가정하자). 이를 로드 밸런신(load balancing)이라고 하며, 나중에 다시 살펴볼 것이다.
- sink 의 PULL socket 은 worker 들로부터 작업 결과를 골고루 수집한다. 이를 fair-queuing 이라고 한다.

Pipeline pattern 은 "slow joiner" 현상을. 만약 PUSH 와 PULL 을 사용한다면, worker 들 중 하나는 다른 worker 에 비해 더 많은 메시지를 수신하게 될 것이다. 왜냐하면 PULL socket 은 보다 빨리 join 되기 때문에 다른 소켓들에 비해 많은 양의 메시지들을 수신하기 때문이다. 만약 로드 밸런싱(load



Fair Queuing

balancing)을 원한다면 Chapter 3 - Advanced Request-Reply Patterns 에서 load balancing 패턴을 보길바란다.

Programming with ZeroMQ

이제껏 예제들을 봤으니 지금쯤이면 ZeroMQ 를 이용해서 어플리케이션을 만들고 싶을 것이다. 하지만 시작하기 전에, 심호흡을 하고, 진정하고, 당신의 스트레스와 고민을 줄여줄 아래의 글들을 읽어보도록 하자.

- ZeroMQ 을 한걸음씩 배우자. 단순한 API 로 보일지도 모른다. 하지만 내부적으로 수 많은 가능성을 가지고 있다. 천천히, 한번에 하나씩, 마스터하길 바란다.
- 읽기 좋은 코드를 만들자. 읽기 나쁜 코드는 문제점들을 숨기고, 다른사람들이 알아보기 힘들게 만든다. 의미없은 변수 이름을 작성할 수도 있다. 하지만 사람들이 그 코드를 봤을때는 혼란스러울 수 있다. 알아보기 쉬운 변수명을 작성하도록 하자. 그리고 일관성있는 들여쓰기 역시 읽기 좋은 코드를 만든다.
- 만들고자 했던 내용으로 테스트를 하자. 당신의 프로그램이 제대로 작동하지 않을 때, 어떤 라인에서 문제가 발생했는지 알 수 있을 것이다. 특히 ZeroMQ 를 사용할 경우, 단 몇번의 테스트만으로도 어디가 문제인지 바로 알 수가 있다.
- 만약 프로그램이 의도한대로 작동하지 않는다면, 코드를 잘게 쪼개도록 하자. 그리고 하나씩 테스트를 해서 어느 부분이 정상적으로 작동하지 않는지를 확인하자. ZeroMQ 는 코드를 잘게 쪼갤 수 있도록 도와준다. 엄청난 이점이다.
- 필요한 만큼 추상화(클래스, 메소드, 뭐가되었든)를 하자. 코드를 복사하고 붙여넣기를 하게되면, 정확히 그만큼의 에러도 같이 복사하고 붙여넣기를 하는 것이다.

Getting the Context Right

ZeroMQ 어플리케이션은 항상 context 를 만들고, 이를 이용해서 socket 을 생성한다. C 에서는 zmq_ctx_new()를 호출한다. 하나의 프로세스에서는 반드시 하나의 context 만 생성해야 한다. 기술적으로, context 는 프로세스의 모든 소켓들을 위한 컨테이너와 같은 역할을 하며, inproc 소켓(프로세스 내에서 가장 빠른 쓰레드 연결)들의 Transport 역할을 한다. 만약 실행중인 프로세스에 2 개의 context 가 있다면, 두개의 다른 ZeroMQ instance 처럼 작동할 것이다. 만약 반드시 그렇게 해야만 하는 이유가 있다면, 2개 이상의 context 를 생성해도 된다. 하지만 그 외의 경우라면 다음을 기억하자.

- 프로세스의 시작에 zmq_ctx_new() 를 한번만 호출하고, 프로세스의 마지막에 zmq_ctw_destroy() 를 호출하자.

만약 fork() 시스템 콜을 사용중이라면, zmq_ctx_new() 를 fork() 이후, child 프로세스의 시작부분에서 호출 하면 된다.

Making a Clean Exit

유명한 프로그래밍 명언이 있다. "always clean-up when you finish the job". ZeroMQ 를 사용할 때, 가령 python 의 경우, 할당된 메모리는 자동적으로 해제된다. 하지만 C 를 사용할 경우, 메모리 해제에 주의를 기울여야 한다. 그렇지 않으면, 메모리 누수, 불안정한 어플리케이션, 그리고 업보가 쌓일 것이다.

ZeroMQ 에서의 메모리 관리는 굉장히 엄격하다. 그 이유는 굉장히 기술적인 부분이며 고통스럽다. 결론적으로 만약 하나의 socket 이라도 open 되어 있는 상태에서 zmq_ctx_destroy() 를 할 경우, 영원히 대기상태가 될 것이다. 그리고, 만약 모든 socket 을 close 한 상태라도, zmq_ctx_destroy() 은 기본적으로 현재 pending 중인 connect 가 있다면 작업이 종료될 때 까지 대기하게 된다. 또, close 함 socket 이라도 close 전 LINGER 를 0으로 설정하지 않은 상태에서 전송할 메시지가 있는 경우 이 역시 작업이 종료될 때 까지 계속 대기하게 된다.

즉, ZeroMQ 에서는 messages, sockets, contexts 들에 대해 굉장히 신경을 써야 한다. 하지만 다행히도, 굉장히 쉽다.

- zmq_msg_t object 사용을 지양하고, 가능한 zme_send() 와 zmq_recv() 를 사용한다.
- zmq_msg_recv() 를 사용할 경우, 가능한 빨리 zmq_msg_close() 를 통해 수신한 메모리 해제를 해야한다.
- 만약 많은 수의 socket 을 열고, 닫기를 해야할 경우, 아마도 어플리케이션 자체를 다시 설계해야하는 신호일지도 모른다. 어떤 경우, 소켓이 해제되지 않고, context 가 해제될 때 해제되는 경우가 있다.
- 프로그램을 종료할 때, 모든 socket 을 close 하고 zmq_ctx_destroy() 를 호출해야 한다. 이는 context 를 종료시키기 때문이다.

이 사항들은 c 개발자들에게 해당되는 말이다. 메모리관리가 자동으로 되는 언어들의 경우, socket 과 context 는 scope 를 벗어나는 순간 메모리 해제된다. 만약 exceptions 를 사용할 경우, 다른 일반적인 리소스들과 같이 반드시 final 같은 블럭을 이용해서 별도로 사용한 내용을 청소해 주어야 한다.

만약 다중 쓰레드를 이용할 경우, 훨씬 더 복잡해진다. 아무리 많은 경고에도 결국에는 다중 쓰레드에서 ZeroMQ 를 사용할 사람들 때문에, 다음 챕터에서 다중 쓰레드 사용과 관련된 내용을 다룰 것이다. 아래에 간략하게 그 까다로움을 조금이나마 나타내었다.

먼저 여러 쓰레드에서 같은 socket 을 사용하면 안된다. 제발 왜 같은 소켓을 사용해야만 하는지 설명하려고 하지 말기를 바란다. 그냥 하지 말아라. 그 다음으로 ongoing request 중인 socket 들을 일일이 shut down 해야 한다. 가장 일반적인 방법은 낮은 LINGER 를 설정(1 sec)하는 것이다. If your language binding doesn't do this for you automatically when you destroy a context, I'd suggest sending a patch.

Finally, destroy the context. This will cause any blocking receives or polls or sends in attached threads (i.e., which share the same context) to return with an error. Catch that error, and then set linger on, and close sockets in that thread, and exit. Do not destroy the same context twice. The zmq_ctx_destroy in the main thread will block until all sockets it knows about are safely closed.

Voila! It's complex and painful enough that any language binding author worth his or her salt will do this automatically and make the socket closing dance unnecessary.

Why We Needed ZeroMQ

맛보기이긴 하지만 ZeroMQ 를 살펴보았다. 이제 "왜"라는 질문으로 돌아가보자.

오늘날의 많은 어플리케이션들은 LAN/internet 과 같은 네트워크와 관련된 컴포넌트들을 가지고 있다. 즉, 결국 개발자들은 네트워크를 통한 메시징과 관련된 작업들을 해야하는데, 각각의 개발자마다 다루는 방식이 다르다. 이런 작업들이 비록 그렇게 어려운 작업은 아니지만, 바이트를 전송하는 것과 신뢰성있는 메시징을 하는것과는 큰 차이가 있다.

TCP 를 사용할 경우, 어떤 문제점들에 직면하게되는지 간단히 살펴보도록 하자. 어떤 종류의 메시징 계층이든, 결국에는 비슷한 문제를 해결해야만 할 것이다.

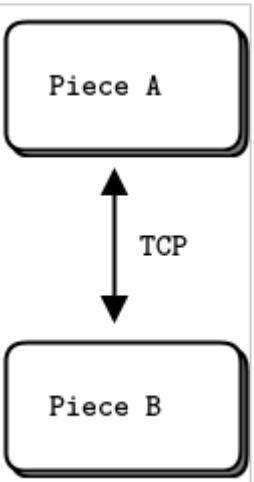
- I/O를 어떻게 다룰 것인가? 만약 어플리케이션이 먹통(block)일 경우, 혹은 I/O를 백그라운드로 관리할 수 있는가? Blocking I/O는 결국 Scale 의 문제에 부딪히게 된다. 반면에 백그라운드 작업은 굉장히 다루기가 힘들다는 문제가 있다.
- 동적 컴포넌트(Dynamic component)들을 어떻게 다룰 것인가? 예를 들어 일시적으로 중지를 시켜야하는 경우라면? 서버/클라이언트를 정확하게 구분하고 서버가 항상 작동하도록 할 수 있을까? 매 초마다 재접속을 시도할 수 있을까?
- 네트워크로 전송된 메시지를 어떻게 나타낼 수 있을까? 메시지를 프레임단위로 쪼개게 되면, 읽기, 쓰기 및 버퍼 오버플로우에 대처하기가 쉬워지겠지만, 대용량 비디오 스트리밍과 같은 경우는 어떻게 할 수 있을까?
- 만약 메시지를 전송할수 없다면 어떻게 할 수 있을까? 컴포넌트가 다시 정상적으로 작동할 때까지 기다려야 한다면? 메시지를 포기해야 할까? 데이터베이스에 쌓아두어야 할까? 메모리 큐에 입력해야 할까?
- 메시지 큐를 어디에 적재할 수 있을까? 만약 컴포넌트의 Queue Reading 속도가 너무 느릴 경우 어떻게 할 수 있을까? 어떤 전략을 사용할 수 있을까?
- 잃어버린 메시지는 어떻게 관리할 수 있을까? 새로운 데이터를 기다려야 할까? 재전송을 요청해야 할까? 아니면 별도의 신뢰성을 보장하는 새로운 계층을 만들어야 할까? 만약 계층 내에서 충돌이 발생하면 어떻게 해야 할까?
- 만약 다른 종류의 Network transport 계층을 사용한다면 어떻게 될까? 예를 들어, TCP 단일 전송 대신, 다중 전송을 사용한다면? 혹은 IPv6? 어플리케이션을 새로 만들어야 할까? 가상 Transport 계층을 만들어야 할까?
- 메시지를 어떻게 Route 할 수 있을까? 여러명에게 같은 메시지를 전송할 수 있을까? 최초 요청자에게 정확한 메시지를 응답할 수 있을까?
- 어떻게 다른 언어들에 대한 API 를 제공할 수 있을까? wire-level 프로토콜을 재구현해야할까? 패키지 방식의 라이브러리로 제공해야 할까? 안정성문제는 없을까? 어떻게 보증할 수 있을까?
- 서로 다른 아키텍쳐 사이의 데이터 전송시 아무런 문제가 없을까? 반드시 인코딩/디코딩을 하도록 해야할까?
- 네트워크 에러는 어떻게 대처할 수 있을까? 재전송해야할까? 기다려야할까? 그냥 무시해야할까? 프로그램 종료를 해야할까?

Hadoop Zookeeper 와 같은 오픈소스를 들여다보자. 2013년 1월 경 소스를 확인하면, 서버/클라이언트 네트워크 통신 프로토콜을 위해 약 4,200 줄의 신비스럽고(미스테리한) 문서화되지 않은 소스코드가 있었다. select 대신 poll 을 사용한 것으로 보아 굉장히 효율적인 것이라는 것을 알수 있다.하지만 Zookeeper 는 단순히 작동하는 것 이상으로 잘 정리된 문서와, 잘 가상화된 계층 레이어가 있었어야만 했다. 매번 사람들이 이를 재구현해야만 한다면 얼마나 큰 낭비이겠는가?

그런데, 어떻게 재사용이 가능한 메시지 계층을 만들 수 있을까? 왜 많은 프로젝트에서 이 기술이 필요한 걸까? 사람들이 TCP 소켓을 사용할때마다 늘 이런 작업들을 처리해야만 하는 걸까?

재사용이 가능한 메시징 시스템을 만든다는 것은 굉장히 어려운 일이다. 이 때문에 굉장히 적은 수의 프로젝트들만이 이 메시징 시스템을 만드는 시도를 했으며, 이런 이유로 상업용 메시징 제품들은 복작하고, 비싸고, 유연하지 못하고, 조잡했다. 이러던 중, 2006년 iMatrix 에서 AMQP 를 디자인했는데 여전히 복잡하고, 비싸고, 조잡했다. 그리고 사용에 많은 시간이 걸렸다.

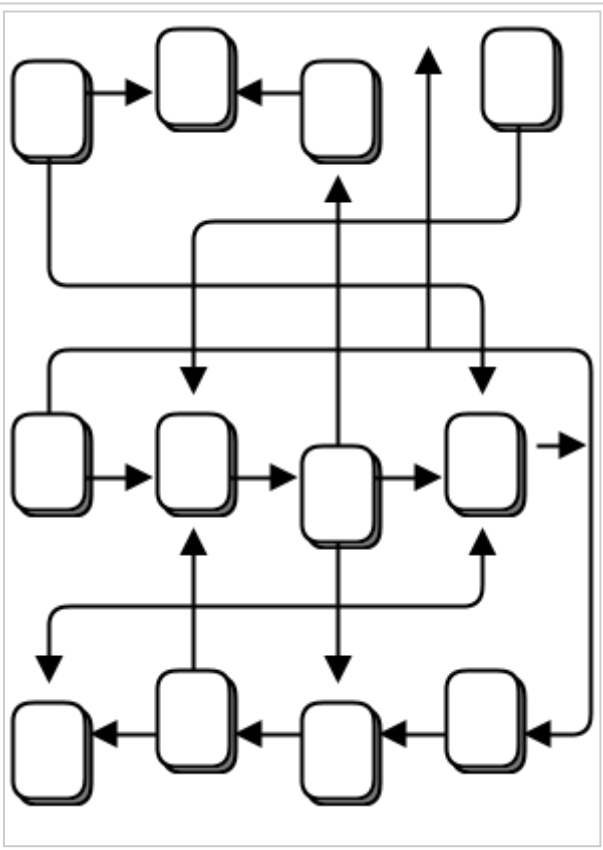
AMQP와 같은 대부분의 메시징 프로젝트들은 위에 나열한 여러가지 문제를 해결하면서도 재사용이 가능하도록 노력했다. 이들은 "broker"라는 새로운 컨셉을 만들었는데, addressing, routing, queuing 과 같은 작업을 관리한다. 이는 서버/클라이언트 프로토콜 혹은 API 등에서 프로토콜의 최상단에서 어플리케이션이 프토콜을 사용할 수 있도록 해주는 역할을 했다. broker는 대규모 네트워크에서의 복잡도를 놀랄정도로 줄여주었다. 하지만 Zookeeper 와 같은 제품에서는 좋은 결과를 얻지 못했다. 큰 시스템에 하나의 broker 를 추가한다는 것은 병목현상과 관리요소가 늘어난다는 것을 의미했다. 여러개의 broker 를 추가하는 방식으로 극복할 수 있었지만 여전히 복잡도가 높아지고, 고장날 위험이 높아지는 부담이 뒤따랐다.



Messaging as it Starts

그리고 broker 중심의 방식은 그것을 관리하는 팀이 필요했다. 말 그대로 밤/낮으로 관리가 필요했다.

때문에 작거나 중간 크기의 어플리케이션 개발자들은 함정에 빠졌다. 불필요한 네트워크 프로그래밍을 피하기위해서는 어플리케이션의 크기를 작고 단순하게 유지해야만 했다. 아니면 어렵고 까다로운 네크워크 프로그래밍과 크기가 크고 복잡한 어플리케이션을 개발해야만 했다. 어느쪽도 쉽지 않았다. 혹은 비싸면서 성능이 좋지않은 사용제품을 구입해야만 했다.


Messaging as it Becomes

우리가 필요한 것은 간단하고, 비용이 들지않고, 아무 어플리케이션과도 잘 작동되는 메시징이었다. 쉽게 링크가 가능하면서도 의존성이 없는 라이브러리, 추가 위험성이 없으면서도 어떤 OS 및 언어와도 잘 작동하는 메시징 말이다.

그게 바로 ZeroMQ 이다. 효과적이고, 즉시 사용이 가능하면서 비용이 없는 메시징 라이브러이다.

- 백그라운드 쓰레드로 비동기 I/O를 수행한다. lock-free data structure 를 사용하기에 ZeroMQ 어플리케이션은 lock, semaphore 및 다른 어떤 대기 상태가 필요없이 동시 사용이 가능하다.
- 컴포넌트들은 동적으로 활성/비활성 될 수 있으며, ZeroMQ에서는 자동적으로 재접속을 수행한다. 이 말은 순서에 상관없이 컴포넌트의 시작이 가능하다는 것이다. 이를 이용하면 네트워크에서 언제든 추가/해제가 가능한 "service-oriented architecture"(SOAs)를 구현할 수 있다.
- 필요할 경우, 자동적으로 메시지를 Queuing 한다. 지능적으로 수행되며 수신쪽에서 최대한 Queueing 이 가능한 정도에 맞추도록 push 된다.
- Over-full queue 을 다룰 수 있는 여러가지 방법을 제공한다("high water mark"). Queue 가 꽉 찼을 때, ZeroMQ 는 어떤 메시징을 하고 있느냐에 따라("pattern") 자동적으로 Sender 를 Block 하거나 메시지를 버린다.
- 어플리케이션에서 임의의 Transport 계층을 사용할 수 있도록 해준다. TCP, multicast, in-process, inter-process, etc.. 서로 다른 Transport 계층 사용을 위해 코드를 재작성할 필요가 없게끔 해준다.
- 메시징 패턴에 따라 서로 다른 전략으로 slow/block readers safe 를 한다.
- 여러가지 패턴(request-reply, pub-sub, ...)을 통한 메시지 route 를 제공한다.
- queue proxy, forward, capture 등을 제공한다.
- 정확하게 메시지를 수신할 수 있도록 한다. 10k 메시지를 보냈다면, 10k 메시지를 수신한다.
- 어떤 포멧도 강요하지 않는다. 0 에서 기가 바이트까지 전송이 가능하다.
- 지능적으로 에러를 관리한다. 자동 재전송과 같은 기능을 수행한다.
- CPU 를 적게 사용한다.

사실, ZeroMQ 는 이보다 더 많은 일들을 수행한다.

Socket Scalability

ZeroMQ 가 얼마나 감당할 수 있는지 확인해보자. weather server 와 여러개의 client 를 실행하는 간단한 스크립트이다.

```
wuserver &
wuclient 12345 &
wuclient 23456 &
wuclient 34567 &
wuclient 45678 &
wuclient 56789 &
```

Client 가 작동함에따라, top 명령어로 프로세스가 다음과 같이 작동하는 것을 확인할 수 있다.

```
PID  USER  PR  NI  VIRT  RES  SHR S %CPU %MEM   TIME+  COMMAND
7136  ph   20   0 1040m 959m 1156 R  157 12.0 16:25.47 wuserver
7966  ph   20   0 98608 1804 1372 S   33  0.0  0:03.94 wuclient
7963  ph   20   0 33116 1748 1372 S   14  0.0  0:00.76 wuclient
7965  ph   20   0 33116 1784 1372 S    6  0.0  0:00.47 wuclient
7964  ph   20   0 33116 1788 1372 S    5  0.0  0:00.25 wuclient
7967  ph   20   0 33072 1740 1372 S    5  0.0  0:00.35 wuclient
```

잠시 위의 결과가 무엇을 의미하는지 생각해보자. weather server 는 단일 소켓을 가지고 있다. 그리고 5 개의 client 에게 병렬로 데이터를 전송하고 있다. 심지어 몇 천개의 client 를 연결할 수도 있다. 하지만 server 쪽에서는 이를 볼 수 없다. 이를 보면 ZeroMQ 는 마치 하나의 작은 서버로써 수 많은 client 의 요청과 데이터 전송을 묵묵히 수행하고 있는 것이다.

Upgrading from ZeroMQ v2.2 to ZeroMQ v3.2

Compatible Changes

Incompatible Changes

Suggested Shim Macros

Warning: Unstable Paradigms!

See also

- http://zguide.zeromq.org/page:all#toc6 - Chapter 1 - Basics

Reference

1. ↑ http://api.zeromq.org/3-2:zmq_setsockopt