Libzmq Chapter 2 - Sockets and Patterns

From 탱이의 잡동사니

Contents

Overview

챕터 1에서 간단한 ZeroMQ 의 기본 패턴들을 살펴보았다(request-reply, pub-sub, pipeline). 이번 챕터에서는 실제 프로그램에서 이를 어떻게 응용하는지를 살펴보도록 하자.

다음을 살펴볼 예정이다.

- ZeroMQ 소켓의 생성과 사용
- 소켓을 이용한 Messsage send/receive
- ZeroMQ 비동기 I/O 를 이용한 Application 작성
- 단일 thread 에서 다중 소켓 사용
- fatal/nonfatal error 관리
- Interrupt signal(ctrl+c) 관리
- ZeroMQ 종료 방법
- ZeroMQ 어플리케이션 메모리 누수 탐지
- Multipart 메시지 send/receive
- 네트워크 메시지 forward.
- 간단한 Message queuing broker 작성법
- ZeroMQ를 이용한 멀티 쓰레드 프로그램 작성법
- ZeroMQ 를 이용한 쓰레드 signal 전송
- ZeroMQ 를 이용한 네트워크 노드 조정
- pub-sub 에서의 message envelope
- 메모리 오버플로우를 막기 위한 HWM(high-water mark) 사용법.

The Socket API

정말 솔직하게 이야기하자면, ZeroMQ 는 일종의 switch-and-bait 방식으로 동작한다. ZeroMQ는 메시지 프로세싱과 관련한 내용을 숨기고 Socket-based API 를 제공하기때문에 큰 이점을 준다.

Socket 은 네트워크 프로그래밍에서의 사실상의 표준(De facto standard)이다. ZeroMQ 를 더욱 더 돋보이게 만드는 점은 어떤 특별한 컨셉을 적용하는 것이 아닌 일반적인 소켓을 사용한다는 점이다.

먹기 좋은 음식처럼, ZeroMQ 역시 받아들이기 쉽다. ZeroMQ 의 소켓의 사용은 BSD 소켓과 같이 크게 4가지 부분으로 나뉜다.

- 소켓을 생성하고 종료한다. Creating and destroying sockets. (zmq_socket(), zmq_close()).
- 필요할 경우, 소켓의 속성을 확인하고 설정한다. Configuring sockets by setting options on them and checking them if necessary. (zmq_setsockopt(), zmq_getsockopt())
- 소켓을 네트워크와 연결한다. Plugging sockets into the network topology by creating ZeroMQ connections to and from them. (zmq_bind(), zmq_connect())
- 소켓을 이용하여 메시지를 송신/수신 한다. Using the sockets to carry data by writing and receiving messages on them. (zmq_msg_send(), zmq_msg_recv())

소켓은 언제나 void pointer 이고, 메시지는 structure 라는 것을 기억하자. 이 때문에 c 에서는 zmq_msg_send() 와 zmq_msg_recv() 와 같이 메시지의 주소를 전달하는 것 만으로 ZeroMQ 를 사용할 수 있다.

소켓의 생성, 종료, 수정은 정확히 사용자가 생각한대로 동작한다. 하지만 ZeroMQ 는 비동기 방식이며 유연하게 동작한다는 것을 기억하자.

Plugging Sockets into the Topology

두 개의 노드사이에 연결을 설정하기 위해서는 한쪽의 노드에서 zmq_bind() 를 사용하고, 다른쪽에서는 zmq_connect() 를 사용해야 한다. 간단하게, zmq_bind() 를 사용하는 쪽이 server 가 되고, zmq_connect() 를 사용하는 쪽이 client 가 된다.

ZeroMQ 를 통한 connection 은 기존의 TCP connection 과는 약간의 다른점이 있다. 크게 다음과 같이 나타낼 수 있다.

- 여러가지 transport 계층을 사용할 수 있다(inproc, ipc, tcp, pgm, epgm, ...). zmq_inproc(), zmq_ipc(), zmq_tcp(), zmq_pgm(), zmq_epgm() 을 보라.
- 하나의 소켓이 여러개의 in/out connection 을 가질 수 있다.
- zmq_accept() 는 없다. 소켓이 connection 을 시도할 때, 자동으로 accept 가 된다.
- connection 이 끊어질 경우, 자동으로 재접속을 시도한다(네트워크의 연결이 끊어졌다 연결되었을 경우).
- ZeroMQ 를 사용하는 application 들은 이 connection 에 직접적으로 접근할 수 없다. 캡슐화 되어있다.

많은 수의 아키텍쳐들이 client/server 모델을 따른다. 이 모델에서 보통 server 는 정적이고, client 는 동적으로 작동하게된다. 가끔 addressing 과 관련된 이슈들이 발생하게 되는데, server 는 필연적으로 client 에게 주소(위치)가 노출되는 반면에 client 는 그렇지 않아도 된다. 때문에 명확하게 어느쪽이 zmq_bind() 를 수행해야 하고(server), 어느쪽이 zmq_connect() 를 수행해야(client) 하는지를 알 수 있다. 물론 가끔 특별한 네트워크에서는 조금 다를 수 있다.

이제, server 를 시작하기전에 client 를 시작해보자. 전통적인 네트워킹에서라면 아마 시작과 동시에 에러를 확인할 수 있을 것이다. 하지만 ZeroMQ 는 어느쪽이 먼저 시작하던 상관없도록 해준다. client 에서 zmq_connect()를 함과 동시에, connection 이 생성되고 소켓을 통해 메시지를 전송할 수 있게 된다. 그리고 이후 server 가 실행되면(큐에 메시지가 가득차서 메시지를 discard 하기 전에) ZeroMQ 는 메시지를 전송하기 시작한다.

server는 하나의 소켓을 이용해서 여러개의 endpoint(프로토콜과 address 의 조합) 를 가질 수 있다. 즉, 서로 다른 transport 를 통한 connection 을 받아들일 수 있다는 뜻이다.

```
zmq_bind(socket, "tcp://*:5555");
zmq_bind(socket, "tcp://*:9999");
zmq_bind(socket, "inproc://somename");
```

대부분의 transport 들이 그러하듯, 하나의 endpoint 를 두번 할당할 수 있다.

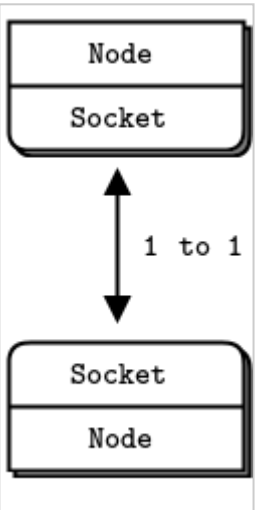
비록 ZeroMQ 에서 어느쪽이 bind를 하건, connect 를 하건 상관이 없도록 한다지만, 어느정도 차이점은 있다. 이 차이점에 대해서는 나중에 깊게 살펴볼 예정이다. 짧게 말하면, server 란, 좀더 고정적인 endpoint 를 bind 를 하는 쪽이며, client 는 동적인 부분을 담당하며 connect 를 하는 쪽이라고 보면 된다.

소켓에는 타입이 있다. 소켓의 타입은 소켓의 속성을 정의한다. 메시지의 내부/외부 라우팅, Queuing, etc 등을 정의한다. 여러개의 타입들을 연결할 수 잇다(예를 들면 publisher/subscriber). 소켓은 "message pattern" 의 형태로 동작하게 된다. 나중에 깊게 살펴 볼 것이다.

이와 같이 ZeroMQ 에서는 소켓들을 조금 다른 방식으로 connection 하게 되는데, 바로 이 점이 ZeroMQ 로 하여금 Message Queuing System 으로써의 동작하게 하는 것이다. 여기에 proxy 와 같은 계층이 위에 올라가게 된다(나중에 살펴볼 것이다). 하지만 기본적으로 ZeroMQ 를 이용하게되면, 아이들의 블럭 장난감처럼 네트워크 아키텍쳐를 마음껏 가지고 놀 수 있게 된다.

Sending and Receiving Messages

메시지를 송신하고, 수신하기 위해서는 zmq_msg_send() 와 zmq_msg_recv() 를 사용해야 한다. 전통적인 이름이지만, ZeroMQ 의 I/O 모델을 알게된다면, 기존의 TCP 와는 다른 방식으로 작동한다는 것을 알게 될 것이다.



TCP sockets are 1 to 1

TCP 소켓과 ZeroMQ 소켓과의 가장 큰 차이점을 알아보도록 하자.

- ZeroMQ 소켓은 메시지를 전송한다. 바이트 스트림을 전송하는 TCP 보다는 UDP와 비슷하다.
- ZeroMQ 소켓은 I/O 작업을 백그라운드 쓰레드에서 수행한다. 이 뜻은 ZeroMQ 를 사용하는 어플리케이션이 바쁘게 작업하든 말든 상관없이 수신하는 메시지는 local input queue 로 오고, 송신하는 메시지는 local output queue 로 간다는 뜻이다.
- ZeroMQ 소켓은 이미 기본적으로(socket type에 따라) 1-to-N 라우팅 기능을 가지고 있다.

사실, zmq_bind() 메소드는 메시지를 바로 socket connection 으로 전송하지는 않는다. 단지 메시지를 queue 에 입력할 뿐이다. 이후 I/O thread 에서 비동기로 메시지를 송신한다. 몇몇 정말 특별한 경우를 제외하고는 blocking 을 하지 않는다. 때문에, zmq_send() 가 return 값을 반환했다고 실제로 메시지를 전송했음을 의미하지는 않는다.

Unicast Transport

ZeroMQ 는 여러 종류의 unicast transport(inproc, ipc, tcp) 와 multicast transport(epgm, pgm) 를 제공한다. multicast 는 보다 발전된 기술로, 나중에 살펴볼 것이다. 아직은 사용하려 하지 말기를 바란다.

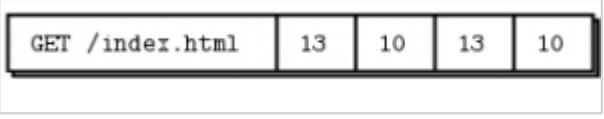
대부분의 경우, TCP(disconnected TCP) 를 사용한다. 빠르고, 편리하고, 유동적이다. Disconnected 라고 표현을 했는데, 왜냐하면 ZeroMQ 의 tcp transport 는 굳이 연결되는 상대방이 없어도 괜찮기 때문이다. client 와 server 는 언제든 서로 connect 와 bind 가 가능하며, 연결이 되고, 끊겨도 application 는 여전히 동작할 수 있다.

inter-process(ipc) transport 도 tcp 와 같이 disconnected 이다. 단 하나의 제한사항이 있을 뿐이다. 아직 Windows 에서는 작동하지 않는다. 관습에 따라 다른 파일 이름과 구분할 수 있도록 endpoint 이름을 ".ipc" 확장자로 표현한다. Unix 시스템에서 ipc endpoint 를 사용한다면 이를 생성하기 위한 적당한 permission 이 필요하다. 마찬가지로, 모든 프로세스들이 파일에 접근이 가능하도록 해야한다(ie. 같은 작업 디렉토리에서 동작).

ZeroMQ is Not a Newtural Carrier

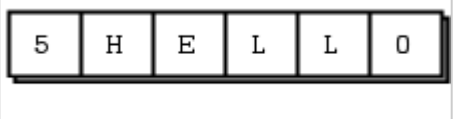
ZeroMQ 를 처음접하는 사람들이 하는 가장 많은 질문은 이것이다. "ZeroMQ 로 어떻게 XYZ server 를 만들 수 있나요?". 예를 들면, "ZeroMQ 로 어떻게 HTTP server 를 만들 수 있나요?"이다. 만약 일반적인 소켓으로 HTTP request 와 response 를 처리하고 있면, ZeroMQ 로도 똑같은 일을 하면서 훨씬 빠르고 좋게 만들 수 있다.

답은 "작동하는 것이 전부가 아닙니다"이다. ZeroMQ 는 단순히 전달자가 아니다. ZeroMQ 는 transport 프로토콜 에서 동작하는 frame 그 자체이다. ZeroMQ만의 frame 을 사용하기에 기존의 프로콜의 framing 과는비교할 수 없다. 예를 들어 TCP/IP 로 작동하는 HTTP request 와 ZeroMQ request 를 살펴보자.



HTTP on the Wire

HTTP request 는 CR-LF 를 가장 간단한 framing 구분자로 사용한다. 반면에 ZeroMQ 는 length-specified frame 을 사용한다. 때문에 ZeroMQ를 통해서 HTTP 와 같은 프로토콜을 만들 수 있다. 하지만 HTTP 가 될 수는 없다.



ZeroMQ on the Wire

하지만 ZeroMQ v3.3 부터는 소켓 옵션 ZMQ_ROUTER_RAW 를 지원하는데, ZeroMQ framing 없이도 데이터를 read/write 할 수 있게 해준다. 이를 이용하면 보통의 HTTP request/response 처럼 데이터를 read/write 할 수 있다.

I/O Threads

ZeroMQ는 I/O 를 background thread 로 한다고 이야기 했었다. 하나의 I/O thread 로도 대부분의 application(엄청나게 빠른) 을 감당할 수 있다. 새로운 context 를 생성할 때, I/O thread 역시 시작된다. 일반적인 이야기로, 하나의 I/O thread 로 Gigabyte/sec (In/Out) 를 허용한다. I/O thread 의 숫자를 늘리고 싶다면 소켓을 생성하기 전에 zmq_ctx_set() 을 이용하면 된다.

```
int io_threads = 4;
void *context = zmq_ctx_new();
zmq_ctx_set(context, ZMQ_IO_THREADS, io_threads);
assert(zmq_ctx_get(context, ZMQ_IO_THREADS) == io_threads);
```

하나의 소켓이 한번에 수 십, 심지어는 수 천 개의 connection 을 허용하는 것을 보았다. 이것이 바로 ZeroMQ 를 이용해서 application 를 작성할 때 발생하는 일이다. 전통적인 네트워크 application 에서라면 하나의 connection 당, 하나의 thread 혹은 process 가 필요했을 것이다. 그리고 하나의 소켓만을 사용할 수 있었을 것이다. ZeroMQ 는 이런 방식을 완전히 벗어나서 전혀 새로운 방식으로 scaling 을 가능하게 해준다.

만약 ZeroMQ 를 inter-thread communications 로만 사용한다면(외부 소켓 I/O 가 없는 multithread application), I/O threads 를 0 으로 설정할 수도 있다.

Messaging Patterns

ZeroMQ socket API 의 가장 근간은 messaging pattern 이다. enterprise messaging 에 대한 배경지식이 있다면, 혹은 UDP 를 알고 있다면, 서로 약간 비슷하다고 할 수 있다. 하지만 ZeroMQ 를 처음 사용하는 사람들에게는 놀랄 일일 수 있다. 때문에 1-to-1 소켓 연결을 하는 TCP 패러다임을 사용 할 것이다.

간단히 ZeroMQ 가 무엇을 하는지 생각해보자. 데이터(메시지)를 노드로 전송하고, 빠르고, 효과적이다. 노드를 thread, process 등 에 비유할 수도 있다. ZeroMQ 는 application 으로 하여금 실제 transport 가 무엇이든 상관없이 단인 socket API 로 작동할 수 있도록 해준다(마치 in-process, inter-process, TCP 혹은 multicast).네트워크에 연결이 되고 떨어질때마다 자동적으로 상대방에게 재접속을 한다. Sender/Reciever 모두 메시지를 Queueing 한다. 메모리를 무한정 사용하는 것이 아닌, 프로세스에 무리가 없을 정도(out of memory)의 한계를 가진다. lock-free 테크닉을 사용하기에 node 끼리의 통신에 lock, wait, semaphore, deadlock 이 필요/발생하지 않는다.

하지만 결론적으로 본다면, pattern 에 따라 메시지를 route/queue 한다. 이 패턴들이 ZeroMQ 를 지능적으로 만들어준다. 패턴은 캡슐화되어 있으며, 데이터 전송과 작동과 관련된 어렵게 얻어낸 경험들의 집약체이다. ZeroMQ 의 패턴들은 hard-code 되어 있지만 나중의 버전에는 사용자가 정의한 패턴도 허용될 것이다.

ZeroMQ 의 패턴은 소켓들의 매칭 타입으로 구현된다. 달리 말해서, ZeroMQ 의 패턴을 이해하려면, 소켓의 타입과 어떻게 작동하는지를 이해해야 한다는 것이다.

built-in core ZeroMQ 패턴에는 다음과 같은 것들이 있다.

- Request-reply

  여러 개의 client, 여러 개의 service 와 connect 한다. remote procedure call and task distribute pattern 이다.

- Pub-sub

  publisher 와 subscriber 의 connect 이다. data distribution pattern 이다.

- Pipeline

  여러개의 단계와 루프를 가질 수 있는 fan-out/fan-in pattern 이다. parallel task distribution 과 collection pattern 이다.

- Exclusive pair

  두 개의 소켓들에 배타적으로 connect 한다. 하나의 프로세스에서 두 개의 thread에 연결하는 패턴이다. 일바적인 소켓과 혼동하지 말자.

이미 Chapter 1 - Basic 에서 처음 3개의 패턴을 확인했다. Exclusive pair 패턴 역시 이번 챕터에서 확인할 것이다. zmq_socket() man page 를 살펴보면 패턴에 대해 잘 정리가 되었다. 다음은 사용가능한 소켓 조합이다.

- PUB and SUB
- REQ and REP
- REQ and ROUTER(take care, REQ inserts an extra null frame)
- DEALER and REP(take care, REP assumes a null frame)
- DEALER and ROUTER
- DEALER and DEALER

- ROUTER and ROUTER
- PUSH and PULL
- PAIR and PAIR

XPUB XSUB 소켓에 관한 내용도 볼 것이다. 위에 나열한 소켓 조합외의 다른 조합은 정상적이지 않은 결과를 가져올 것이다. 미래의 버전에서는 에러를 리턴할 것이다.

High-Level Messaging Patterns

Working with Messages

libzmq 에는 메지시 전송과 수신을 위한 API 가 두 개 있다. 바로 zmq_send(), zmq_recv() 이다. 하지만, zmq_recv() 는 메시지 길이와 관련해서 단점이 있다. 바로, 입력된 메시지 버퍼의 크기에 상관없이 메시지를 초기화 시켜버린다. 이 때문에 zmq_msg_t 구조체를 이용하는 다른 API 를 제공하고 있다.

- Initialize a message : zmq_msg_init(), zmq_msg_init_size(), zmq_msg_init_data()
- Sending and receiving a message : zmq_msg_send(), zmq_msg_recv()
- Release a message : zmq_msg_close()
- Access message content : zmq_msg_data(), zmq_msg_size(), zmq_msg_more()
- Work with message properties : zmq_msg_get(), zmq_msg_set()
- Message manipulation : zmq_msg_copy(), zmq_msg_move()

메모리 상에서, ZeroMQ 메시지는 zmq_msg_t structure(언어에 따라 혹은 class) 로 관리된다. C 에서 ZeroMQ 메시지를 기본 사용법이다.

- zmq_msg_t object 를 생성 한 뒤 data 를 넘겨주어야 한다.
- 메시지를 읽기 위해서는 zmq_msg_init() 를 이용해서 메시지를 초기화를 시킨다음에, zmq_msg_recv() 로 넘겨주면 된다.
- 메시지를 쓰기 위해서는 먼저 zmq_msg_init_size() 를 이용해서 메시지를 생성한 뒤, 데이터를 입력한다. 이후, zmq_msg_send() 로 넘겨준다.
- 메시지를 해제(not destroy) 하기 위해서는 zmq_msg_close() 를 이용하여 reference 를 없애면 된다. 이후, ZeroMQ 에서 메시지를 destroy 할 것이다.
- 메지디에 접근(access) 하기 위해서는 zmq_msg_data() 를 이용하면 된다. 메시지의 크기를 알고 싶다면, zmq_msg_size() 를 이용하면 된다.
- 정확한 목적 없이는 zmq_msg_move(), zmq_msg_copy(), zmq_msg_init_data() 를 사용하면 안된다.
- 메시지를 zmq_msg_send() 로 넘겨준 다음에는 ZeroMQ 에서 메시지를 초기화를 시킬 것이다. 하나의 메시지를 이용해서 두번의 zmq_msg_send() 를 이용할 수 없으며, 한번 전송한 메시지에 대해서 다시 접근할 수 없다.
- 이 규칙들은 zmq_send() 와 zmq_recv() 에는 적용되지 않는다. zmq_msg_t 를 사용하는 API 에 대해서만 적용된다.

만약 하나의 메시지를 한번 더 전송하고자 한다면, 먼저 새로운 메시지를 생성하고, zmq_msg_init() 로 초기화를 한 다음, zmq_msg_copy() 를 이용해서 복사를 해야 한다. zmq_msg_copy() 는 실제로는 데이터를 복사하지는 않지만 reference 를 복사한다. 이후, 두 번째 메시지를 전송할 수 있다. 이후, 메시지에 대한 reference 가 모두 없어졌을 때, 실제로 삭제가 된다.

이 뿐만이 아니라, ZeroMQ 는 multipart 메시지도 지원한다. 즉, 여러 개의 frame list 를 전송하거나 수신할 때 하나의 메시지로 다룰 수 있도록 해준다. Chapter 3- Advanced Request-Reply Patterns 에서 살펴볼 것이다.

ZeroMQ 에서 Frame("부분 메시지") 은 ZeroMQ 메시지의 가장 기본이다. Frame 은 길이가 있는 데이터 블럭이다. 길이는 0이 될 수 도 있고 그 이상이 될 수도 있다.

ZeroMQ 가 TCP 연결을 통해서 데이터를 읽고, 전송 할 때는 ZMTP 라는 프로토콜을 사용한다. 간단히 설명하면 다음과 같다.

원래 ZeroMQ 가 메시지는 UDP 와 같이 하나의 프레임이었다. 하지만 이후에 간단히 "more" bit 를 추가함으로써, 여러개의 메시지로 나누어서 전송할 수 있도록 하였다. 이후 사용자들로 하여금, more bit 설정을 통해 더 수신할 메시지가 있는지 없는지 여부를 확인할 수 있도록 하였다.

- 메시지는 하나 혹은 그 이상의 파트로 구성될 수 있다.
- 이런 메시지 파트 역시 "frame" 으로 불린다.
- 각각의 파트는 zmq_msg_t 객체이다.
- Low-level API 를 통해서 각각의 파트를 개별적으로 전송/수신이 가능하다.
- Higher-level API 를 이용하면 부분 메시지 전체를 한번에 전송/수신이 가능하다.

다음은 알아두면 좋은 내용들이다.

- 0 크기의 메시지를 보낼 수 있다(시그널 및 다른 목적을 위해 사용할 수도 있다).
- ZeroMQ 는 모든 메시지(frame)를 전송하거나, 아예 전송하지 않음을 보장한다.
- ZeroMQ 는 메지시 전송시, 즉시 전송하지 않는다. 약간의 규칙적이지 않는 전송 지연 시간을 가지며, 이 때문에 multipart 메시지 들은 반드시 메모리에 입력될 수 있어야 한다.
- 메시지 수신이후 반드시 zmq_msg_close() 를 사용해야 한다. 전송시에는 사용하지 않아도 된다.

여러번 이야기를 했지만, 아직은 zmq_msg_init_data() 를 사용하면 안된다. 이는 zero-copy method 이며, 사용에 많은 주의가 필요하다.

Handling Multiple Sockets

지금까지의 모든 예제들에서, 메인 루프는 다음과 같이 동작했다.

```
├ Wait for message on socket.
├ Process message.
├ Repeat.
```

만약 한번에 여러개의 endpoint 로부터 데이터를 읽어들이려고 한다면 어떻게 해야할까? 가장 간단한 방법은 하나의 소켓으로 하여금 모든 접속을 받아들이게 한 다음, ZeroMQ 로 fan-in 하는 것이다. 다른 쪽 endpoint 에서도 같은 방식의 패턴을 사용한다면 유효한 방식이다. 하지만 같은 방식의 패턴이 아니라면 유효하지 않다.

사실, 여러개의 소켓으로터 한번에 데이터를 읽어들이기 위해서는 zmq_poll() 을 사용해야 한다. 더 나은 방법으로는 프레임 워크 안에서 zmq_poll 을 wrapper 로 만들어서 event-driven reactor 로 사용하는 방법이다. 하지만 손과 시간이 걸리는 작업이다.

아래는 두 개의 소켓으로부터 nonblocking read 를 하는 예제이다.

See also

- http://zguide.zeromq.org/page:all#Chapter-Sockets-and-Patterns - Chapter 2 - Sockets and Patterns