(/wiki/%EB%8C%80%EB%AC%B8)

%EB%A1%9C%EA%B7%B8%EC%9D%B8&returnto=ZeroMQ%2Flibzmq%EC%9D%98+%EB%82%B4%EB%B6%80+%EA%B5%AC%EC%A1%B0)

PGWiki 검색

문서 (/wiki/ZeroMQ/libzmq%EC%9D%98_%EB%82%B4%EB%B6%80_%EA%B5%AC%EC%A1%B0)

토론 (/w/index.php?title=%ED%86%A0%EB%A1%A0:ZeroMQ/libzmq%EC%9D%98_%EB%82%B4%EB%B6%80_%EA%B5%AC%EC%A1%B0&action=edit&redlink=1)

원본 보기 (/w/index.php?title=ZeroMQ/libzmq%EC%9D%98_%EB%82%B4%EB%B6%80_%EA%B5%AC%EC%A1%B0&action=edit)

역사 (/w/index.php?title=ZeroMQ/libzmq%EC%9D%98_%EB%82%B4%EB%B6%80_%EA%B5%AC%EC%A1%B0&action=history)

ZeroMQ/libzmq의 내부 구조

< ZeroMQ (/wiki/ZeroMQ)

목차

libzmq의 내부 아키텍처

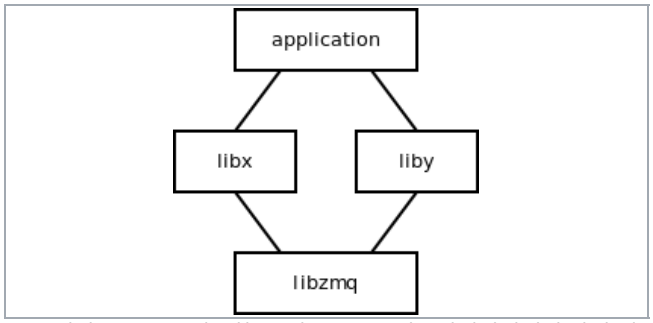
아래 내용은 ØMQ의 공식 위키를 토대로 작성되었다. [1]

상기 문서는 프로젝트에 참여할 마음이 있는 개발자들을 위해 작성된 것으로 보여지는데, 그러기엔 상당히 불친절하다.

오히려 그렇기에 적당히 핥아볼 사람들에게 유익한 것 같다 ——;

전역 상태

라이브러리에서 전역 변수를 사용함에 따라 생길 수 있는 문제의 예



위 예와 같이 다른 라이브러리인 libx, liby에서 libzmq를 링크하고, 이 libx, y를 애플리케이션에서 다시 링크한다면 컴파일 과정은 문제가 안 될 것이다.

만일 이 상황에서 libzmq에 전역 변수가 있다면 실제 동작은 어떨까? 산신령도 모를 일이다 ——;

그런 연유로, libzmq에서는 전역 변수를 사용하지 않는다.

대신, 라이브러리 사용자가 전역 상태를 명시적으로 만들 책임이 있다고 한다. 전역 상태를 포함하는 객체를 *문맥(context)*[2]이라고 부르는데, 이 문맥은 크게 2가지의 관점으로 바라볼 수 있다.

1. 사용자 관점
2. 라이브러리 자신(libzmq)의 관점

사용자 관점에서는 소켓들과 사용되는 입출력 스레드의 풀로 보여지고, 라이브러리 관점에서는 사용자가 필요할 수도 있는 모든 전역 상태를 저장하는 객체로 볼 수가 있다.

libzmq의 4가지 전송 방식 중 하나인 inproc 방식을 예로 들어, 이 방식에서는 가용 종단점의 목록을 저장하는데 문맥이 사용된다.

또한, 닫힌 소켓들의 목록은 여전히 메모리 상에 남게 되는데, 이는 아직 보내지지 않은 메시지를 문맥이 지연시키기 때문이다.

병행성 모델

ØMQ의 병행성 모델은 멀티스레드를 사용했다는데도 병렬 처리 (/w/index.php?title=%EB%B3%91%EB%A0%AC_%EC%B2%98%EB%A6%AC&action=edit&redlink=1)를 조직할 목적으로 뮤텍스 (/w/index.php?title=%EB%AE%A4%ED%85%8D%EC%8A%A4&action=edit&redlink=1)나 조건 변수 (/w/index.php?title=%EC%A1%B0%EA%B1%B4_%EB%B3%80%EC%88%98&action=edit&redlink=1), 세마포어 (/w/index.php?title=%EC%84%B8%EB%A7%88%ED%8F%AC%EC%96%B4&action=edit&redlink=1) 등을 사용하지 않았다고 한다. (뒤에서 이에 대한 덧글이 붙으므로 참고)

내부 확정성과 병행성을 위해서 메시지 전달 (/w/index.php?title=%EB%A9%94%EC%8B%9C%EC%A7%80_%EC%A0%84%EB%8B%AC&action=edit&redlink=1) 방식을 사용했다고 하는데, (여기에서 eat our own dogfood라는 표현이 재미있다.) 대신에 각 객체가 자신의 스레드 내에서 존재하고, 다른 스레드가 이를 접근할 수 없도록 했다는 것이 그 골자다.

(한편으로는 내부에서까지 메시지 전달을 함으로써 생기는 비용에 대해 의문이 든다.)

방법이 아예없는 것은 아니고, 한 스레드가 메시지(사용자 레벨의 ØMQ 메시지와 식별하기 위해서 '명령(command)'이라고 부른다고 한다.)를 다른 스레드에게 보내는 것으로서 그 스레드의 객체와 상호작용을 할 수가 있다고 한다. (결국엔 객체 to 객체도 같은 방식...)

사용자의 관점에서 객체 간의 명령 전달은 쉬운 편이다. 그냥 'object_t' 클래스에서 상속받는 게 전부다. 들어오는 명령에 대해 핸들러를 정의하거나, 명령을 보낼 수도 있다. 모든 사용 가능한 명령은 command.hpp 파일에 있다고 한다.

대부분의 명령은 수행 중에 대상 객체가 사라지지 않음을 보장한다. 예외인 명령들도 있는데, 이 경우엔 송신자가 명령 자체를 전송하기 전에 대상 객체에 저장된 sent_seqnum이라는 카운터를 동기적으로 증가시키는 `inc_seqnum` 함수를 호출함으로써 보장이 된다. 대상 객체는 이 명령을 처리할 때, processed_seqnum이라는 카운터를 증가시키고, processed_seqnum이 sent_seqnum보다 작을 때 아직 모든 명령이 처리된 것이 아님을 알 수가 있다. 모든 처리는 object_t와 own_t 클래스에서 투명하게 이루어지며, 명령 송/수신자는 시퀀스 번호를 신경쓰지 않고 그냥 명령을 보내고 받으면 된다고 한다. [3]

몇몇 데이터의 일부는 임계 영역 (/w/index.php?title=%EC%9E%84%EA%B3%84_%EC%98%81%EC%97%AD&action=edit&redlink=1)에 싸여있는데, 이 영역을 선택하기 위한 2가지 규칙이 있다. [4]

1. 언제든, 어떤 스레드이던지 데이터에 접근할 필요가 있다. (말하자면 존재하는 inproc 종단점의 목록)
2. 데이터는 임계 영역으로 보호되고, 임계 경로 (/w/index.php?title=%EC%9E%84%EA%B3%84_%EA%B2%BD%EB%A1%9C&action=edit&redlink=1) 상에서 접근돼선 안된다. (메시지 전달 자체)

스레딩 모델

우선, OS의 관점에서 보는 ØMQ는 2가지의 스레드가 있다. [5]

- 애플리케이션 스레드 : ØMQ 외부에서 만들어지고, API 접근에 사용된다.
- I/O 스레드 : ØMQ 내부에서 만들어지고, 백그라운드에서 메시지를 주고 받는 데에 사용된다.

ØMQ의 관점에서 스레드는 '우편함(mailbox)'을 갖는 어떤 객체다. 여기서 우편함은 해당 스레드에 있는 모든 객체에 보냈던 명령들을 담고 있는 큐다.[6] 스레드는 이 우편함에서 명령을 찾아서 보낸 순서대로 하나씩 처리한다.

ØMQ에 관한 2가지 스레드가 있는데 하나는 I/O 스레드이고, 다른 하나는 소켓(socket)이다.

I/O 스레드는 운영 체제 스레드와 대응되고, 명령 수신을 위한 하나의 우편함을 갖고 동작한다. 소켓은 묘하게 좀 더 복잡한 구석이 있다. 각 ØMQ 소켓은 명령 수신을 위한 자신의 우편함을 갖고, 분리된 스레드로서 처리된다. 실제로 하나의 애플리케이션 스레드는 다수의 소켓을 만들 수 있는데, 이 경우 다수의 ØMQ 스레드가 하나의 운영 체제 스레드에 대응된다. 여기서 더 복잡하게(!!!) 하면, ØMQ 소켓은 운영체제 스레드 간에서도 이주될 수가 있다.[7] 이 경우, ØMQ 스레드와 운영 체제 스레드 사이의 관계가 변화된다.

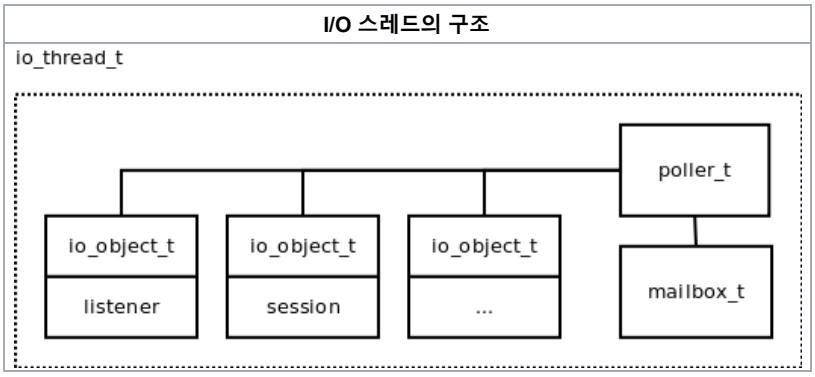
I/O 스레드

I/O 스레드는 비동기적으로 네트워크 트래픽을 처리하기 위해 백그라운드 상에서 동작하는 스레드다.

이 스레드는 io_thread_t 클래스로 구현되어있는데, thread_t 클래스와 object_t 클래스를 상속받는다.

각 I/O 스레드 내부에는 크게 3개의 객체가 존재한다.

I/O 스레드의 구조



1. poller_t : 폴링 메커니즘 구현
2. io_object_t : 입출력 이벤트 관리
3. mailbox_t : 우편함

poller_t는 각기 다른 운영 체제에서 구현되는 select(select_t), poll(poll_t), epoll(epoll_t)과 같은 폴링 메커니즘을 제공한다.

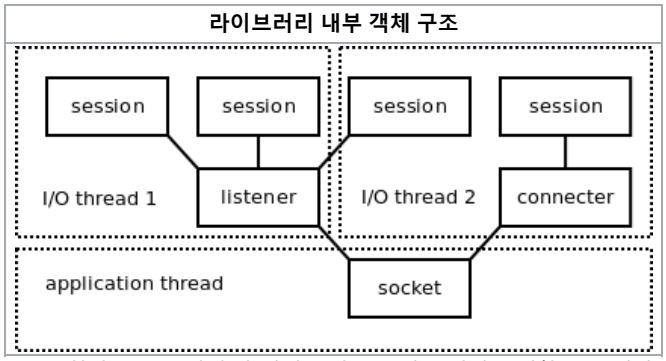
io_object_t는 add_fd()로 파일 기술자를 등록하고, 여기에서 입/출력 이벤트(in_event/out_event)가 발생하면 콜백을 호출하는 역할을 한다. 등록된 파일 기술자는 rm_fd()를 사용해서 제거할 수 있다. 또한, add_timer()를 사용해서 타이머를 추가할 수도 있는데, 이 타이머는 만료 시에 이벤트(timer_event)가 발생된다. 등록된 타이머는 cancel_timer()를 사용해서 취소할 수 있다.

io_thread_t는 자신의 파일 기술자를 폴러에 등록하게 되는데, 이 파일 기술자는 우편함과 연관되어 새로 도착하는 명령에 대해 in_event를 발생시키고, io_thread_t는 대상 객체에 이 명령을 전달한다.

객체 트리

ØMQ 라이브러리에서 만드는 내부 객체들은 트리 구조로 조직된다.

라이브러리 내부 객체 구조

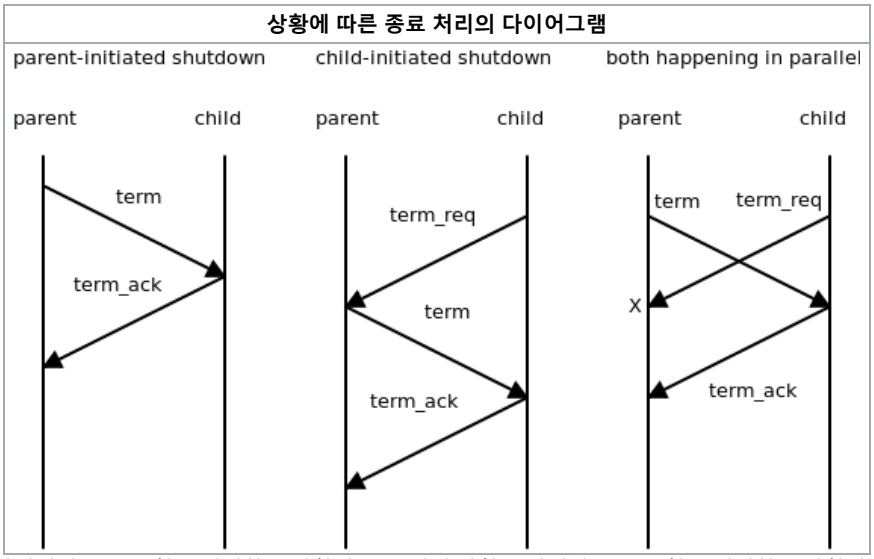


위 그림에서 알 수 있듯이, 트리의 루트는 항상 'socket'인데 각 객체는 다른 스레드 내에 존재할 수도 있다.

그림에선 애플리케이션 스레드와 I/O 스레드가 분리되어 애플리케이션 스레드에는 socket만이 있고, 이 socket을 2개의 I/O 스레드가 사용하고 있음을 알 수 있다.

이러한 객제 트리의 존재 이유는 결정론적 종료 메커니즘을 제공하기 위한 것이다. 객체는 종료를 위해 종료 요청을 모든 자식들에게 보내고 그 응답을 대기하는데, 이렇게 종료 요청과 응답을 주고 받는 과정에서 두 객체 사이에 진행 중인 명령을 효율적으로 플러시한다. 앞서 나온 대다수의 명령에서 시퀀스 번호가 사용될 필요가 없다는 것은, 이런 메커니즘을 사용하기 때문이다.

객체가 부모의 요청에 의한 것이 아니라 자식이 스스로 종료 처리를 하려는 경우[8]에 종료 처리는 더욱 복잡해진다.



첫번째 상황이 부모가 자식에게 종료 요청을 개시하는 상황이고, 두번째 상황은 자식이 종료 요청을 개시하는 상황이다.

마지막 세번째가 특이한 경우라고 할 수가 있겠는데, 동시에 개시한 경우 자식이 보낸 종료 요청(term_req)은 부모가 깔끔하게 씹어드신다.

객체 트리 메커니즘은 own_t 클래스[9]로 구현되는데, 객체 트리에 있는 모든 객체들은 명령을 송/수신할 수가 있다.[10]

리퍼 스레드

앞서 나온 종료 메커니즘에는 특별한 문제가 하나 있다.

어떤 객체가 종료할 때 zmq_close 호출은 POSIX와 유사하게 바로 반환이 되지만, 자식 객체와 모든 핸드 쉐이킹 (/wiki/TCP) 과정이 완료됐다고 확신할 수가 없다는 점이다.

zmq_close에서 반환된 스레드가 libzmq의 코드가 아닌 전혀 다른 코드를 실행하고 있을 수도 있고, 다시는 여기로 넘어오지 않을 수도 있다. 따라서, 이 소켓은 애플리케이션 스레드 대신에 모든 핸드 쉐이킹을 제어할 수 있는 작업 스레드로 이주되야 한다.

논리적으로는 소켓을 I/O 스레드 중의 하나로 이주시킬 수도 있을 것이다. 하지만, ØMQ는 I/O 스레드가 없이도 초기화 될 수 있다는 점을 기억하자![11] 그렇기에 이 작업을 할 전용 스레드가 필요하고 그게 바로 리퍼 스레드다.

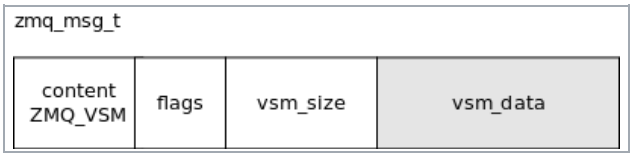
리퍼 스레드는 *reaper_t* 클래스로 구현되며, 소켓은 리퍼 스레드에 *reap* 명령을 송신하고, 명령을 수신한 리퍼 스레드는 이 소켓을 깔끔하게 종료되도록 관리한다.

메시지

메시지의 조건은 꽤 복잡한 편인데, 이는 아주 크거나 작은 메시지의 효율적인 전송을 구현하기 위한 것이다.

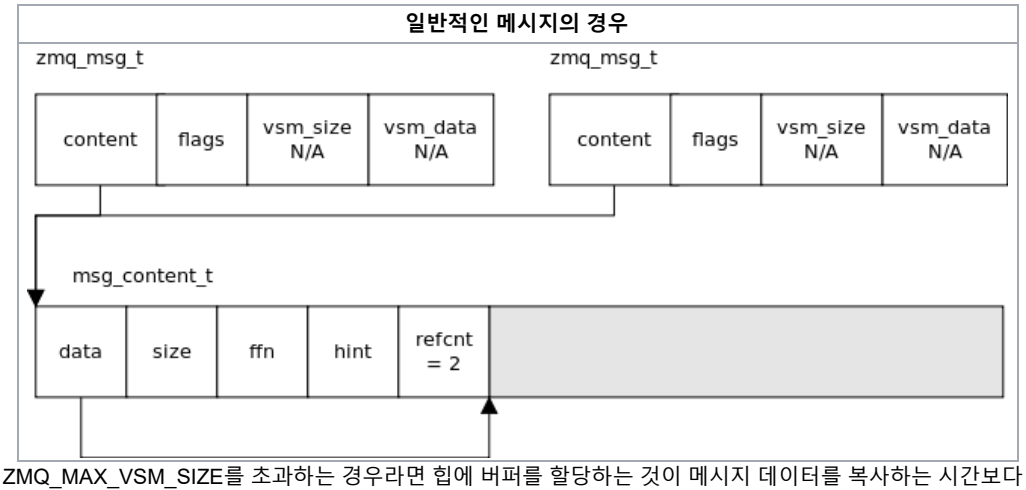
- 매우 작은 메시지는 힙에 있는 공유 데이터를 유지하는 것보다 메시지를 복사하는 비용이 더 싸다. 그래서 이런 메시지는 따로 연관 버퍼를 가지지 않고, *zmq_msg_t* 구조체에 직접 데이터가 저장된다.[12]
- inproc 통신을 사용할 때, 메시지는 복사되지 않는다. 따라서, 어떤 한 스레드에 보내진 버퍼는 다른 스레드에서 수신하고 할당을 해제해 버린다.
- 메시지는 참조 계수 (/w/index.php?title=%EC%B0%B8%EC%A1%B0_%EA%B3%84%EC%88%98&action=edit&redlink=1)를 지원한다. 어떤 메시지가 다수의 다른 TCP 연결에서 발행됐다면, 이걸 송신하는 모든 스레드(혹은 TCP 연결)는 이 버퍼를 복사하는게 아니라 같은 버퍼에 접근해서 보낸다.
- 같은 방법을 사용자에게도 적용할 수 있는데, 사용자는 내용의 복사를 거치지 않고 같은 물리적 버퍼를 다수의 ØMQ 소켓으로 보낼 수가 있다.
- 사용자는 데이터를 복사할 필요 없이 애플리케이션-지정 할당[13] 메커니즘에 의해 할당된 버퍼를 보낼 수 있다.

VSM에 대한 zmq_msg_t의 구조



매우 작은 메시지(VSM, Very Small Message)에 대한 버퍼는 zmq_msg_t 구조체의 일부인 것을 볼 수 있다. 이 경우, 할당을 위한 별도의 작업이 필요치 않고 *content* 필드에 ZMQ_VSM 상수를 넣음으로써 VSM이라는 것을 식별할 수가 있다. *vsm_size*에는 메시지의 길이가 지정되고, *vsm_data*에는 메시지 데이터가 저장된다.

*vsm_data*의 최대 크기는 기본 30 바이트로 지정되어 있으며, 이는 ZMQ_MAX_VSM_SIZE 상수로 조정이 가능하다.[14]



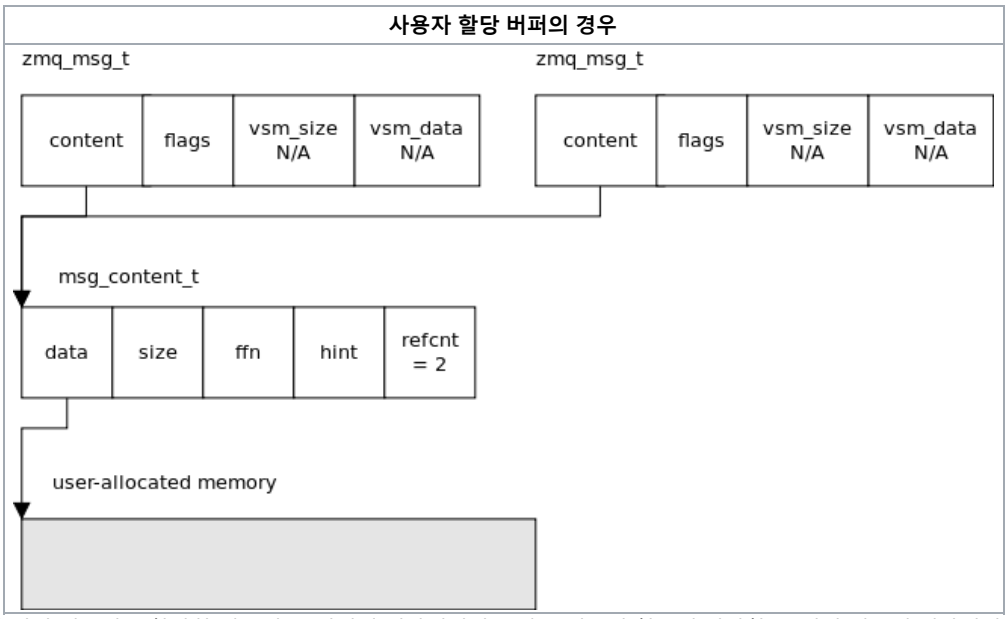
메시지 크기가 ZMQ_MAX_VSM_SIZE를 초과하는 경우라면 힙에 버퍼를 할당하는 것이 메시지 데이터를 복사하는 시간보다 더 싸다고 가정한다. 따라서 ØMQ는 힙에 버퍼를 할당한 다음에 zmq_msg_t 구조체가 그 주소를 가리키도록 한다.

힙에 할당된 구조체는 *msg_content_t*이며, 이 구조체는 할당된 청크와 관련된 모든 메타데이터를 저장한다.

- data : 할당된 버퍼의 주소를 저장
- size : 버퍼의 크기를 저장
- ffn : 할당 해제에 사용할 함수의 포인터
- hint : 할당 해제 함수에 전달된 인자
- refcnt : 이 *msg_content_t*에 대한 참조 계수

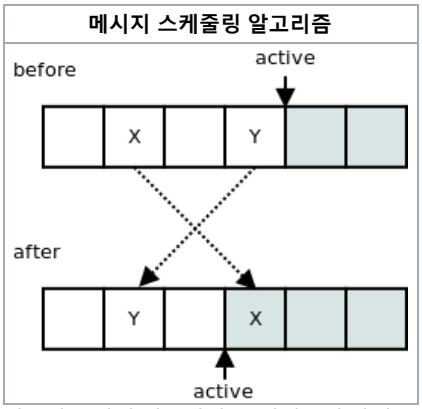
버퍼와 그 메타데이터는 *zmq_msg_t* 인스턴스 간에 공유될 수도 있는데, 이 버퍼를 가리키는 *zmq_msg_t*가 없을 때 할당을 해제하기 위해 버퍼의 참조 계수를 유지한다.

또한 위 그림에서도 나타나는 것처럼, *zmq_msg_copy*는 실제 데이터를 복사하는게 아니라 새로 *zmq_msg_t*를 할당한 다음에 같은 버퍼를 가리키기만 하는 동작을 하게 된다.



끝으로 사용자가 직접 메모리를 할당한 경우에는, 버퍼의 메타데이터를 같은 메모리 청크에 저장할 수 없기 때문에 메타데이터에 대한 청크를 따로 할당해야 한다.

파이프



ØMQ에서 메시지는 파이프 배열 상에서 모든 스케줄링 동작이 이루어지고, 파이프의 상태는 2가지로 표현된다.

- *능동(active)* : 메시지를 송/수신할 수 있는 상태
- *수동(passive)* : 파이프에서 메시지를 읽을 수가 없어서 더 이상 메시지를 송신할 수 없는 상태

위와 같은 표현 방식일 때, 1만 개의 파이프가 있고 이 중에 하나가 수동 상태일 경우를 생각해보면 이 하나를 위해서 9,999개의 파이프를 확인해야 하는 문제가 있다.

그렇기에 ØMQ에서는 위 그림에서 보이는 바와 같이 능동 상태에 있는 파이프를 수동 상태에 있는 파이프와 배열 상의 위치를 교환함으로써 항상 능동 상태의 파이프가 앞에 나타나는 알고리즘을 사용했다.

이런 알고리즘 덕분에 O(1)의 시간복잡도를 유지할 수 있으며, 일일이 능동 상태의 파이프를 찾지 않아도 되는 이점을 얻게 된 것이다.

주석

1. Internal Architecture of libzmp (http://zeromq.org/whitepapers:architecture)
2. 문맥은 **ctx_t** 클래스로 구현돼있다.
3. 근데 왜 이걸 설명한거야...
4. 이 내용은 애매한 부분이 있는데, 그럼 앞서 나온 뮤텍스, 조건 변수, 세마포어 등이 사용되지 않았다는 말은 뭔가?
5. OS 독립적 방식으로 스레드를 만들 수 있는 단순한 클래스인 thread_t로 구현된다.
6. 우편함은 'mailbox_t' 클래스로 구현된다.
7. 여기서 말하는 운영 체제 스레드는 여기에 대응되는 I/O 스레드를 말하는 것으로 보여진다. 아닌가?
8. TCP 종료 후에 세션 객체가 종료되는 등의 경우가 해당될 수 있다.
9. object_t를 상속한다.
10. 그 역은 성립하지 않는데, 객체 트리의 일부가 아닌 파이프나 종단점의 경우가 그 예다.
11. in-process 통신이 그 예다.
12. 이렇게 함으로써 메모리 동적 할당/해제에 대한 비용을 줄이는 것뿐 만이 아니라, 스택에 할당된 메모리 영역을 사용함으로써 속도 향상에도 도움이 된다고 판단하는 것 같다.
13. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.7663&rep=rep1&type=pdf (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.7663&rep=rep1&type=pdf)
14. 물론, 라이브러리의 코드를 수정하여 새로 빌드하는 것은 필요하다..——;

분류 (/wiki/%ED%8A%B9%EC%88%98:%EB%B6%84%EB%A5%98): ØMQ (/wiki/%EB%B6%84%EB%A5%98:%C3%98MQ)