ØMQ

입 문 서

Pieter Hintjens [저자] 박재도 [옮긴이]

제0.7판

ØMQ - 입문서
ØMQ - The Guide

Pieter Hintjens 저자
박재도 옮긴이

차 례

서문

0.1 번역 정보

이 책은 ØMQ 라이브러리의 입문서지만, 일반적인 메시징 시스템의 설계 방법을 배울 수 있도록 작성되어 있습니다. 멀티스레드 프로그래밍 및 네트워크 프로그래밍에서 일어나는 일반적인 문제 해결 방법 및 분산 응용 프로그램의 설계 방법 등을 배울 수 있습니다. 예를 들어, P2P(Peer-to-Peer) 응용프로그램과 분산 해시 테이블 등의 기반을 구현하고자 하는 분들도 추천합니다.

「ØMQ 가이드」는 피터 힌트젠스가 2015년 12월에 zeroMQ 3.2 버전 기준으로 작성한 「ØMQ - The Guide」의 한국어 번역입니다.

「ØMQ - The Guide」을 기반으로 한글로 번역하였으며, 깃허브 문서로 작성하기 위하여 「HAMANO Tsukasa」씨의 문서를 참조하였습니다. 예제는 ØMQ v3.2 기준(2015년)이지만 v4.3.2(2020년)에서 원도우 및 리눅스(와 유닉스) 운영체제에서도 사용할 수 있도록 소스 일부 수정하였습니다.

원서의 내용이 수정이 필요하거나 모호한 부분에 대하여 고인이 된 저자에게 문의할 수 없어, 일부 내용 및 소스 수정이 가해진 부분이 있으며 [옮긴이]에 표시해 두었습니다.

향후 ØMQ 기능 변경에 따라 해당 문서는 지속적으로 수정 및 타 개발 언어들에서도 적용할 수 있도록 하겠습니다.

오자/오역 및 의견이 있을 경우 「[@박재도](zzeddo@gmail.com)」 연락 부탁드립니다.

옮긴이(박재도)는 2000년대 초반부터 hp BASEstar Open이란 MOM(Message Oriented Middleware)를 국내 적용하였으며 2011년부터 하이텍크 산업(반도체/LCD/OLED)에서 TIB/Rendezvous(TIB/RV)를 적용하는 일을 하였습니다. 2012년에 zeroMQ를 처음 접하고 TIB/Rendezvous(TIB/RV) 대체하기 위한 프로젝트를 수행하였으며 2017년 CERN(European Organization for Nuclear Research)에서 사용하는 CMW(Control Middleware)가 zeroMQ 기반으로 구축된 것을 확인하고 기술 이전을 받아 국내 적용하기도 하였습니다.

현재 기존의 경험을 바탕으로 zeroMQ 기반의 SMW을 개발하고 있습니다.

0.2 100자로 ØMQ 설명하기

ZeroMQ(ØMQ, 0MQ, 혹은 zmq 알려짐)는 내장된 네트워킹 라이브러리와 유사하지만 동시성(concurrency) 프레임워크처럼 동작합니다. ØMQ는 다양한 전송계층 환경에서 단순 메시지를 전송하는 소켓을 제공하는데 전송수단으로 프로세스 내 통신(스레드 간), 프로세스 간 통신, TCP 그리고 멀티캐스트 등 다양합니다. 당신은 ØMQ 소켓을 N 대 N 방식으로 멀티캐스팅(pgm, epgm), 발행-구독(pub-sub), 작업 분배(pipeline), 그리고 요청-응답(request-reply) 등 패턴으로 사용할 수 있습니다. 이것은 클러스터 제품들에 적용해도 전혀 문제없는 높은 성능을 가지고 있습니다. 비동기 I/O 모델로 비동기 메시지 작업들을 구축하여 확장 가능한 멀티코어 응용프로그램을 제공합니다. 다양한 개발언어 API 있으며 거의 모든 운영체제에서 적용 가능합니다. ØMQ는 iMatix에서 개발했으며 LGPLv3 라이선스로 오픈소스입니다.

0.3 시작하기

일반적인 TCP 소켓을 가져다가 여러 가지 요소들(구소련 연방의 방사선 동위원소, 근육빵빵맨 등)을 혼합하여 ØMQ을 만들었으며, ØMQ 소켓은 네트워킹 세계(모두가 연결된)를 구원하는 슈퍼 히어로입니다.

그림 1 - 무서운 일

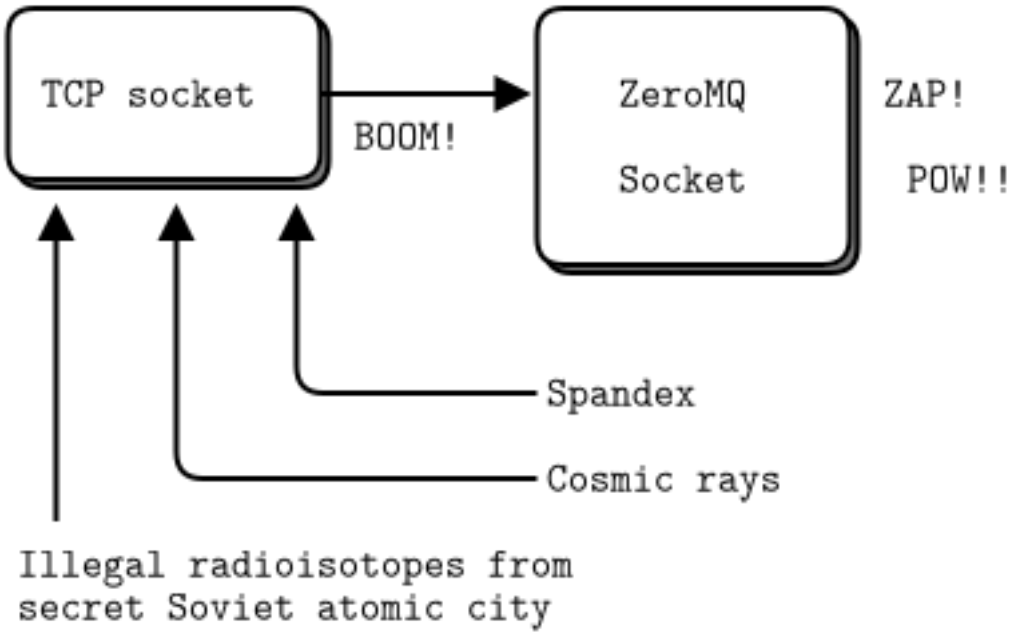


그림 1: 무서운 일

0.4 Zero의 철학

ØMQ의 0은 단지 절충안입니다. 한편에서는 이상한 이름으로 구굴 또는 트위터 검색에서의 가시성이 저하되고 있습니다. 다른 한편에서는 덴마크의 끔찍한 놈들이란 "ØMG røtfl" 의미로 혹은 "Ø는 이상한 모양 제로는 아니다"와 "Rødgrød med Fløde!"(크림이 오른 덴마크 간식) 등으로 표현하는 것은 잘못되었습니다.

원래 ØMQ의 zero의 의미는 "브러커 없는(zero broker)"를 (좀 더 정확하게) 의미하였으며 "지연 없는(zero latency)"로(가능한) 하는 것이었습니다. 이것을 위하여 서로 상반된 목표들로 둘려 싸이게 되었습니다 : 관리 없애기(zero administration), 비용 없애기(zero cost), 낭비 없애기(zero waste). "zero"는 미니멀리즘 문화를 참조하여 프로젝트에 반영하였으며 새로운 기능을 부가하기보다는 복잡성을 제거함으로 힘을 부여하였습니다.

0.5 대상 독자

이 책은 전문 프로그래머로 미래의 컴퓨팅을 지배할 거대한 분산 소프트웨어를 만드는 방법을 배우려는 사람을 대상으로 합니다. ØMQ가 여러 개발 언어를 사용할 수 있지만 여기에 있는 대부분의 예제는 C 언어로 되어 있기 때문에 C 코드를 읽을 수 있으며 네트워크상에서 응용프로그램을 확장하는 문제에 관심이 있고 최소한의 비용으로 최상의 결과가 필요하다고 가정합니다. 그렇지 않으면 ØMQ가 제공하는 절충안들을 인식하지 못할 것입니다. 우리는 가능한 분산 컴퓨팅과 네트워크 상의 모든 개념들을 표현하여 ØMQ을 사용할 수 있도록 하였습니다.

0.6 감사

이 문서를「도서」로 출판하기 위해 편집해 주신 Andy Oram에게 감사합니다。

작업에 도움을 주신 분들에게 감사합니다

Bill Desmarais, Brian Dorsey, Daniel Lin, Eric Desgranges, Gonzalo Diethelm, Guido Goldstein, Hunter Ford, Kamil Shakirov, Martin Sustrik, Mike Castleman, Naveen Chawla, Nicola Peduzzi, Oliver Smith, Olivier Chamoux, Peter Alexander, Pierre Rouleau, Randy Dryburgh, John Unwin, Alex Thomas, Mihail Minkov, Jeremy Avnet, Michael Compton, Kamil Kisiel, Mark Kharitonov, Guillaume Aubert, Ian Barber, Mike Sheridan, Faruk Akgul, Oleg Sidorov, Lev Givon, Allister MacLeod, Alexander D'Archangel, Andreas Hoelzlwimmer, Han Holl, Robert G. Jakabosky, Felipe Cruz, Marcus McCurdy, Mikhail Kulemin, Dr. Gergő Érdi, Pavel Zhukov, Alexander Else, Giovanni Ruggiero, Rick "Technoweenie", Daniel Lundin, Dave Hoover, Simon Jefford, Benjamin Peterson, Justin Case, Devon Weller, Richard Smith, Alexander Morland, Wadim Grasza, Michael Jakl, Uwe Dauernheim, Sebastian Nowicki, Simone Deponti, Aaron Raddon, Dan Colish, Markus Schirp, Benoit Larroque, Jonathan Palardy, Isaiah Peng, Arkadiusz Orzechowski, Umut Aydin, Matthew Horsfall, Jeremy W. Sherman, Eric Pugh, Tyler Sellon, John E. Vincent, Pavel Mitin, Min RK, Igor Wiedler, Olof Åkesson, Patrick Lucas, Heow Goodman, Senthil Palanisami, John Gal-

lagher, Tomas Roos, Stephen McQuay, Erik Allik, Arnaud Cogoluègnes, Rob Gagnon, Dan Williams, Edward Smith, James Tucker, Kristian Kristensen, Vadim Shalts, Martin Trojer, Tom van Leeuwen, Hiten Pandya, Harm Aarts, Marc Harter, Iskren Ivov Chernev, Jay Han, Sonia Hamilton, Nathan Stocks, Naveen Palli, Zed Shaw

1장-기본

0.7 세상을 구원하라

ØMQ를 어떻게 설명할 것인가? 우리 중 일부는 ØMQ가 하는 놀라운 것들을 말하는 것으로 시작합니다.

- 스테로이드를 먹은 스켓입니다.
- 라우팅 기능이 있는 우편함과 같습니다.
- 빠릅니다.

어떤 사람들은 깨달음의 순간을 공유하려 합니다.

- 모든 것이 분명해졌을 때 번쩍-우르릉-쾅쾅 깨달음(zap-pow-kaboom satori) 패러다임 전환의 순간을 느낍니다.
- 사물은 단순해지고 복잡성이 사라집니다
- 마음을 열게 합니다.

어떤 사람들은 다른 것과 비교하여 ØMQ를 설명하려고 합니다.

- 더 작고 단순하지만 여전히 익숙해 보입니다

개인적으로 저는 왜 우리가 ØMQ를 만들었는지를 기억하고 싶으며, 독자 여러분은 우리가 처음 길을 접어든 근처에 머물고 있을지 모르기 때문입니다.

- [옮긴이] 번쩍-우르릉-쾅쾅 깨달음(zap-pow-kaboom satori)에서 사토리는 불교의 용어로 해달의 경지인 깨달음 의미합니다. 일본에서 돈버는 것에도 출세에도 관심이 없는 젊은 세대를 "사토리 세대"라고 부릅니다.

그림 2: zap-pow-kaboom satori

프로그래밍은 예술로 치장된 과학이지만 우리 대부분은 소프트웨어의 물리학을 거의 이해하지 못하고 있으며 과학으로 가르치는 곳이 거의 없습니다. 소프트웨어의 물리학은 알고리즘이나, 자료 구조, 프로그래밍 언어나 추상화 등이 아닙니다. 이런 것들은 다만 우리가 그냥 만들어내고 사용하고 버리는 도구들일 뿐입니다. 소프트웨어의 진정한 물리학은 인간의 물리학입니다. 특히 복잡성 앞에서의 인간의 한계와 우리의 협력을 통하여 거대한 문제를 작은 조각들로 쪼개서 해결하려고 하는 욕구가 있습니다. 이것이 프로그래밍 과학입니다 : 사람들이 쉽게 이해하고 사용할 수 있는 구성요소들을 만들어냄으로써, 사람들이 그걸 이용하여 거대한 문제들을 해결하게 도와줍니다.

우리는 연결된 세상 속에 살고 있습니다. 그리고 현대의 소프트웨어들은 이런 세상 속을 항해할 수 있습니다. 따라서 미래의 거대한 솔루션들을 위한 구성요소들은 반드시 서로 연결되어야 하며 대규모로 병렬적이어야 합니다. 더 이상 프로그램이 그냥 "강력하고 침묵" 하기만 하면 되던 그런 시대는 지났습니다. 코드는 이젠 코드와 대화해야 합니다. 코드는

수다스러워야 하고 사교적이어야 하며 연결되어야 합니다. 코드는 반드시 인간의 두뇌처럼 작동해야 하고, 수조 개의 뉴런들이 서로에게 메시지를 던지듯이, 중앙 통제가 없는 대규모 병렬 네트워크로 단일 장애점 없어야 합니다. 그러면서도 극도로 어려운 문제들을 해결할 수 있어야 합니다. 그리고 코드의 미래가 인간의 두뇌처럼 보인다는 것은 우연이 아닙니다. 왜냐면 결국 네트워크도 단말에서 인간의 두뇌에 연결되며, 그 어떤 네트워크의 진화의 종착점도 인간의 두뇌이기 때문입니다.

당신이 스레드, 통신규약 또는 네트워크와 관련된 일을 하였다면 당신은 이것이 얼마나 불가능에 가까운 일인가를 알 것입니다. 이건 꿈과 같습니다. 실제 상황에서 몇 개의 프로그램들을 몇 개의 소켓을 통해 연결하는 것조차도 어렵고 불쾌합니다. 수조 원? 그 비용은 상상도 할 수 없을 정도입니다. 컴퓨터들을 연결시키는 것은 매우 어렵기 때문에 이를 수행하는 소프트웨어와 서비스들은 수십억 달러짜리의 사업입니다.

그래서 결국 우리는 이러한 세상에서 살게 되었습니다: 인터넷 보급과 속도는 상당히 발전되었는데, 그것을 제대로 활용하는 능력은 몇 년이나 뒤쳐져 있습니다. 우리는 1980년대에 한차례 소프트웨어 위기를 겼었으며, 프레드 브룩스와 같은 수준급 소프트웨어 엔지니어들은 "소프트웨어 업계에서 생산성, 신뢰성, 단순성 이 세 가지 척도 중에 어느 한 방면에서도 진보를 이루는 것에는「은총알이 존재하지 않는다」" 고 주장하였습니다.

브룩스가 놓쳤던 것은 자유롭게 사용 가능한 오픈 소스 소프트웨어들이었으며, 서로 효율적으로 지식을 공유함으로 위기를 해결할 수 있었습니다. 하지만 오늘날 우리는 또 다른 소프트웨어 위기에 직면하고 있지만 이번에는 이걸 얘기하는 사람은 거의 없습니다. 위기는 가장 크고 가장 부유한 기업들만이 "연결된 응용프로그램들을" 만들 수 있다는 점입니다. 물론 요즘에는 클라우드가 있지만 클라우드는 사적 소유이며 우리의 데이터와 우리의 지식은 개인 컴퓨터에서 점점 사라지고, 우리가 접근할 수 없고 우리가 그걸 상대로 경쟁조차 펼칠 수 없는 클라우드로 흘러 들어가고 있습니다. 우리의 소셜 네트워크를 누가 가지고 있을까요? 이것은 PC에서 대형 컴퓨터로 전환되는 혁명이 역으로 수행되고 있습니다.

우리는 이런 정치철학을「문명과 제국, 디지털 혁명」에서 다루고 있습니다 여기서 말하고자 하는 건, 인터넷이 대규모 연결된 프로그램에 잠재력을 제공하고 있음에도 불구하고, 현실은 이것을 충분히 이용할 수 있는 사람들은 극히 드물다는 점입니다. 그러므로 거대하고 흥미로운 문제들(건강, 의료, 교육, 경제, 물류 등의 분야들)을 해결되지 못하고 있으며 원인은 우리가 코드을 효과적으로 연결 못하고 개개인이 함께 협력하여 이러한 문제를 해결할 방법이

없었기 때문입니다.

- [옮긴이] 문화와 제국, 디지털 혁명은 피터 힌트젠스가 지은 책입니다. 링크 내 무료로 읽을 수 있고 PDF나 전자책 포맷으로 다운로드할 수 있습니다.

연결된 코드의 도전 문제를 해결하기 위한 많은 시도들도 있었습니다. 예를 들어 수천 개의 「국제 인터넷 표준화 기구(Internet Engineering Task Force, IETF)」 규격들, 그것들은 각각 퍼즐의 일부만을 해결합니다. 응용프로그램 개발자에게 있어서 HTTP는 간단하고 효과 좋은 해결책이었지만 HTTP는 개발자들에게 거대한 서버와 작고, 어리석은 클라이언트로 구성된 아키텍처를 권고하면서 문제를 더 악화시키게 되었습니다.

그래서 결과적으로 오늘날에도 사람들은 여전히 UDP, TCP, 사적 소유의 통신규약, HTTP와 웹소켓 등을 그대로 사용하고 있습니다. 여전히 고통스럽고 굼뜨고 확장이 어려우며 대체적으로 중앙화 되어 있습니다. 분산된 P2P 아키텍처도 존재하긴 하지만, 대부분 노는 일에만 기여하지 생산적인 데는 도움이 되지 않으며 스카이프나 비트토렌트로 데이터를 주고받는 사람은 많지 않습니다.

이 모든 현실은 다시 우리를 최초의 프로그래밍 과학에 대한 문제로 되돌려 보냅니다. 세상을 구원하기 위해 우리는 두 가지 일을 해야 합니다.

- 첫째, 일반적인 문제인 "어디서나, 코드와 코드 간에 연결시키기"
- 둘째, 모든 대책을 최대한 간단한 구성요소로 만들어 사람들이 쉽게 이해하고 사용하게 하기

말도 안 되게 간단하게 들립니다. 그리고 정말로 그럴지도 모릅니다. 하지만 이것이 바로 이 책의 전부입니다.

0.8 전제 조건

ØMQ 3.2 버전을 사용하고 리눅스와 유사한 운영체제를 사용한다고 가정합니다. 이 책의 기본 예제는 C 언어로 구성되어 있어 C 코드를 읽을 수 있어야 합니다. PUSH와 SUBSCRIBE와 같은 상수를 사용할 때 실제 프로그래밍 언어에서는 ZMQ_PUSH, ZMQ_SUBSCRIBE가 호출된다고 생각할 수 있습니다.

- [옮긴이] C 예제들을 윈도우 운영체제에서도 사용 가능하도록 변경하였습니다.

0.9 예제 받기

예제들은 「깃허브 공개 저장소」에 있으며, `git clone`명령으로 받을 수 있으며 동기화 이후 예제 폴더에서 다양한 개발 언어로 작성된 예제들을 확인 가능합니다.

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

0.10 물으면 얻을 것이다.

Hello World 예제로 시작하며, 클라이언트와 서버 프로그램을 만들도록 하겠습니다. 클라이언트에서 서버로 "Hello"을 보내면 서버에서 "World"를 클라이언트에게 응답합니다. 예제에서 ØMQ 소켓을 5555 포트로 오픈하여 요청에 대응합니다.

hwserver.c: Hello World 서버

```
//  Hello World server
#include <zmq.h>
#include <stdio.h>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>
#define sleep(n)    Sleep(n*1000)
#endif
#include <string.h>
#include <assert.h>

int main (void)
{
    //  Socket to talk to clients
```

```
    void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5555");
    assert (rc == 0);

    while (1) {
        char buffer [10];
        zmq_recv (responder, buffer, 10, 0);
        printf ("Received Hello\n");
        sleep (1);          //  Do some 'work'
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
```

그림 2 - 요청-응답

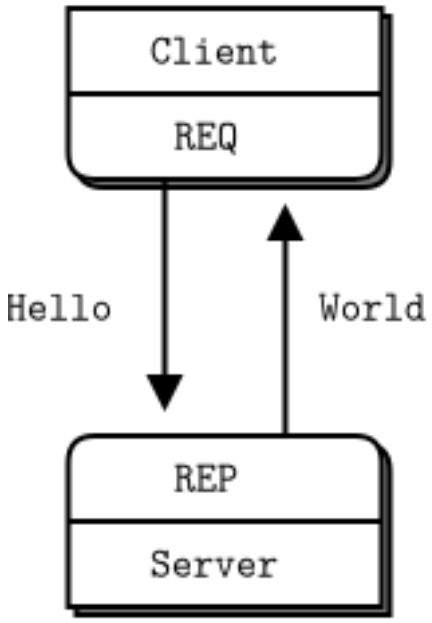


그림 3: Request-Reply

REQ-REP 소켓 쌍은 진행 순서가 고정되어 REQ 클라이언트는 루프상에서 zmq_send()를 호출하고 나서 zmq_recv()를 호출할 수 있으며, zmq_send()를 두 번 호출(예 : 두 개의 메시지를 연속으로 전송) 할 경우에는 오류 코드로 -1이 반환됩니다. 마찬가지로 REP 서버 측은 zmq_recv()를 호출하고 나서 zmq_send ()를 호출해야 합니다.

ØMQ는 참조 언어로서 C 언어를 사용하고 있기 때문에 샘플 코드에서 C 언어를 사용합니다. 여기에서는 C++ 서버 코드를 보고 비교하겠습니다.

hwserver.cpp: Hello World 서버

```
//
//  Hello World server in C++
//  Binds REP socket to tcp://*:5555
//  Expects "Hello" from client, replies with "World"
//
#include <zmq.hpp>
```

```
#include <string>
#include <iostream>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>
#define sleep(n)    Sleep(n*1000)
#endif

int main () {
    //  Prepare our context and socket
    zmq::context_t context (1);
    zmq::socket_t socket (context, ZMQ_REP);
    socket.bind ("tcp://*:5555");

    while (true) {
        zmq::message_t request;

        //  Wait for next request from client
        socket.recv (&request);
        std::cout << "Received Hello" << std::endl;

        //  Do some 'work'
        sleep(1);

        //  Send reply back to client
        zmq::message_t reply (5);
        memcpy (reply.data (), "World", 5);
        socket.send (reply);
```

```
    }
    return 0;
}
```

- [옮긴이] C++ 바인딩의 경우 두 개의 헤더 파일(“zmq.hpp”, “zmq_addon.hpp”)로 구성되어 있습니다.

ØMQ API는 C 언어와 C++에서 거의 동일하다는 것을 알 수 있으며, PHP와 Java 언어의 작성된 예제도 보도록 하겠습니다. 더 많은 것을 숨길 수 있고 코드를 더 쉽게 읽을 수 있습니다.

hwserver.php: Hello World 서버

```
<?php
/*
 *  Hello World server
 *  Binds REP socket to tcp://*:5555
 *  Expects "Hello" from client, replies with "World"
 * @author Ian Barber <ian(dot)barber(at)gmail(dot)com>
 */

$context = new ZMQContext(1);

//  Socket to talk to clients
$responder = new ZMQSocket($context, ZMQ::SOCKET_REP);
$responder->bind("tcp://*:5555");

while (true) {
    //  Wait for next request from client
    $request = $responder->recv();
    printf ("Received request: [%s]\n", $request);

    //  Do some 'work'
```

```
    sleep (1);

    //  Send reply back to client
    $responder->send("World");
}
```

hwserver.java: Hello World 서버

```
//
//  Hello World server in Java
//  Binds REP socket to tcp://*:5555
//  Expects "Hello" from client, replies with "World"
//

import org.ØMQ.SocketType;
import org.ØMQ.ZMQ;
import org.ØMQ.ZContext;

public class hwserver
{
    public static void main(String[] args) throws Exception
    {
        try (ZContext context = new ZContext()) {
            // Socket to talk to clients
            ZMQ.Socket socket = context.createSocket(SocketType.REP);
            socket.bind("tcp://*:5555");

            while (!Thread.currentThread().isInterrupted()) {
                byte[] reply = socket.recv(0);
                System.out.println(
                    "Received " + ": [" + new String(reply, ZMQ.CHARSET) + "]"
```

```
                );

                String response = "world";
                socket.send(response.getBytes(ZMQ.CHARSET), 0);

                Thread.sleep(1000); //  Do some 'work'
            }
        }
    }
}
```

- [옮긴이] Java의 경우 jzmq 바인딩으로 빌드 수행 시 오류가 발생하며, 해당 예제는 jeromq(ØMQ의 Java 구현) 라이브러리를 통하여 수행 필요합니다. Maven 레파지토리에서 원하는 버전의 jar 파일을 받을 수 있으며, 사용을 위해서는 "CLASSPATH" 환경변수에 등록 필요합니다.

다음은 클라이언트 코드입니다.

hwclient.c: Hello World 클라이언트

```
//  Hello World client
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>
#define sleep(n)    Sleep(n*1000)
#endif

int main (void)
```

```
{
    printf ("Connecting to hello world server...\n");
    void *context = zmq_ctx_new ();
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5555");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        char buffer [10];
        printf ("Sending Hello %d...\n", request_nbr);
        zmq_send (requester, "Hello", 5, 0);
        zmq_recv (requester, buffer, 10, 0);
        printf ("Received World %d\n", request_nbr);
    }
    zmq_close (requester);
    zmq_ctx_destroy (context);
    return 0;
}
```

ØMQ 소켓을 가졌지만 실제 구현이 단순하며, 이전에 보았듯이 초능력을 가지고 있습니다. 서버는 동시에 수천 개의 클라이언트들을 가질 수 있으며 작업을 쉽고 빠르게 할 수 있게 되었습니다. 재미를 위해 클라이언트를 시작한 다음 서버를 시작하시기 바랍니다. 모든 것이 여전히 작동하는지 확인한 다음 이것이 의미하는 바를 잠시 생각해보십시오.

- [옮긴이] 서버, 클라이언트 구동 순서에 관계없이 정상적으로 동작합니다.

이 두 프로그램이 실제로 무엇을 하고 있는지 간략하게 살펴보겠습니다. 이들은 우선 ØMQ 컨텍스트 ØMQ 소켓을 생성합니다. 서버는 REP 소켓을 포트 5555번으로 바인딩하여 매번 루프에서 요청을 기다리고 각 요청에 응답합니다. 클라이언트는 REQ 소켓을 포트 5555번에 연결하여 요청을 보내고 서버의 응답을 받습니다.

- [옮긴이] 빌드 및 테스트

```
> cl -EHsc hwserver.c libzmq.lib
> cl -EHsc hwclient.c libzmq.lib
> ./hwserver
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello

> ./hwclient
Connecting to hello world server...
Sending Hello 0...
Received World 0
Sending Hello 1...
Received World 1
Sending Hello 2...
Received World 2
Sending Hello 3...
Received World 3
Sending Hello 4...
Received World 4
Sending Hello 5...
Received World 5
Sending Hello 6...
```

```
Received World 6
Sending Hello 7...
Received World 7
Sending Hello 8...
Received World 8
Sending Hello 9...
Received World 9
```

서버를 죽이고(CTRL-C) 재시작하면 클라이언트는 정상적으로 동작하지 않을 것이다. 프로세스가 죽을 경우 복구는 쉽지 않은 문제이다. 신뢰성 있는 요청-응답은 복잡하지만 "4장-신뢰할 수 있는 요청-응답 패턴"에서 다루도록 하겠습니다.

프로그래밍된 뒤편에서는 많은 일이 일어나고 있지만 프로그래머들에게 중요한 것은 작성된 코드가 얼마나 짧고 멋있는지, 얼마나 자주 그것이 무거운 부하를 받더라도 충돌하지 않는지입니다. 이것이 ØMQ의 가장 간단한 사용 방법이며 RPC(remote procedure call)과 고전적인 클라이언트/서버 모델에 해당합니다.

0.11 문자열에 대한 간단한 주의 사항

ØMQ는 데이터의 바이트 크기 외에는 아무것도 모르기 때문에 개발자가 문자열을 안전하게 처리할 책임이 있습니다. 복잡한 데이터 타입이나 객체에 대하여 작업하기 위해 특화된 "통신규약 버퍼(Protocol Buffer)"라는 라이브러리가 있지만 문자열에 대하여서는 주의해야 합니다.

C 언어와 일부 다른 언어들에서 문자열은 한 개의 널(NULL(0)) 바이트로 끝이 나며 "Hello"와 같은 경우 추가 널(NULL(0)) 바이트를 추가합니다.

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

다른 언어에서 문자열을 보내면 해당 널(NULL(0)) 바이트를 포함하지 않을 것입니다. Python 언어에서는 널(NULL(0)) 바이트를 포함하지 않고 아래와 같이 보낼 수 있습니다.

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

이때 네트워크상에서 문자열의 길이와 개별 문자를 가진 문자열의 내용이 전송됩니다.

그림 3 - ØMQ 문자열(길이 + 개별문자)

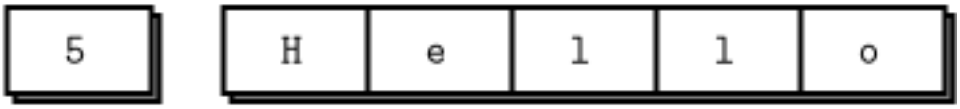


그림 4: ØMQ문자열

C 언어를 통해 코드를 본다면 문자열과 비슷한 것을 확인할 수 있으며 우연히 문자열처럼 동작할 수 있지만(운이 좋아 5 바이트(Hello)에 널(NULL(0)) 뒤따르는 경우) 적절한 문자열이 아닙니다. 클라이언트와 서버가 문자열 형식에 동의하지 않는다면 이상한 결과를 가져오게 됩니다.

ØMQ에서 C 언어로 문자열 데이터를 받는다면 안전하게 끝났는지(널(NULL(0)) 문자 포함) 보장할 수 없기 때문에 매번 여분의 공간을 가지고 공백으로 채워진 새로운 버퍼에 할당하여 문자열을 복사한 다음 널(NULL(0))로 올바르게 종료해야 합니다.

따라서 ØMQ 문자열은 길이가 지정되고 후행 널(NULL(0)) 없이 네트워크로 전송된다는 규칙을 설정하였습니다. 가장 간단한 경우(예제에서 작업 수행), ØMQ 문자열은 위의 그림과 같은 ØMQ 메시지 프레임에 매핑됩니다(길이와 일부 바이트들).

C 언어에서 ØMQ 문자열을 받아 유효한 C 문자열로 변환하는 함수를 정의하겠습니다.

255 문자열 길이 제한이 있는 s_recv() 함수

```
//  Receive ØMQ string from socket and convert into C string
//  Caller must free returned string. Returns NULL if the context
//  is being terminated.
static char *
s_recv (void *socket) {
    char buffer [256];
```

```
    int size = zmq_recv (socket, buffer, 255, 0);
    if (size == -1)
        return NULL;
    buffer[size] = '\0';

#if (defined (WIN32))
    return strdup (buffer);
#else
    return strndup (buffer, sizeof(buffer) - 1);
#endif
}
```

- [옮긴이] 256 바이트 문자열 길이 제약을 제거한 s_recv() 함수

```
//  Receive ØMQ string from socket and convert into C string
//  Caller must free returned string.
inline static char *
s_recv(void *socket, int flags = 0) {
    zmq_msg_t message;
    zmq_msg_init(&message);
    int rc = zmq_msg_recv(&message, socket, flags);
    if (rc < 0)
        return nullptr;           //  Context terminated, exit
    size_t size = zmq_msg_size(&message);
    char *string = (char*)malloc(size + 1);
    memcpy(string, zmq_msg_data(&message), size);
    zmq_msg_close(&message);
    string[size] = 0;
```

```
    return (string);
}
```

아래는 편리한 재사용 가능한 도우미 함수를 작성하였습니다. s_send()는 C 문자열을 받아 ØMQ 형식 문자열을 보내는 함수이며 재사용할 수 있도록 헤더 파일(zhelpers.h)에 포함합니다.

```
//  Convert C string to ØMQ string and send to socket
static int
s_send (void *socket, char *string) {
    int size = zmq_send (socket, string, strlen (string), 0);
    return size;
}
```

도우미 함수들은 C 언어에서 사용할 수 있도록 "zhelpers.h" 파일에 정의해 두었지만 소스가 상당히 길고 C 개발자에게만 흥미롭기 때문에 여유롭게 보시기 바랍니다.

0.12 ØMQ 버전 확인하기

ØMQ는 자주 버전이 변경되며 만약 문제가 있다면 다음 버전에서 해결될 수도 있습니다. ØMQ 버전 확인하는 방법은 다음과 같습니다.

version.c : ØMQ 버전 확인하기

```
//  Report ØMQ version
#include <zmq.h>
int main (void)
{
    int major, minor, patch;
    zmq_version (&major, &minor, &patch);
    printf ("Current ØMQ version is %d.%d.%d\n", major, minor, patch);
    return 0;
```

```
}
```

- [옮긴이] 빌드 및 테스트

```
> cl -EHsc version.c libzmq.lib
Microsoft (R) C/C++ 최적화 컴파일러 버전 19.16.27035(x64)
Copyright (c) Microsoft Corporation. All rights reserved.

version.c
Microsoft (R) Incremental Linker Version 14.16.27035.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:version.exe
version.obj
libzmq.lib
> ./version
Current ØMQ version is 4.3.2
```

0.13 메시지 송신하기

두 번째 고전적인 패턴으로 서버의 단방향 데이터 전송으로, 서버가 일련의 클라이언트들에게 변경정보들을 배포합니다. 예를 들면 우편번호, 온도, 상대습도 등으로 구성된 기상 변경정보들이 배포하여 실제 기상관측소처럼 임의의 값을 생성하도록 하였습니다.

아래의 서버 예제에서는 PUB 소켓을 생성하여 5556 포트를 사용하였습니다.

wuserver.c: 기상 정보 변경 서버

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

```c
#include "zhelpers.h"

int main (void)
{
    //  Prepare our context and publisher
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    int rc = zmq_bind (publisher, "tcp://*:5556");
    assert (rc == 0);

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

        //  Send message 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;
}
```

변경정보의 전송은 시작도 끝이 없이 반복되며, 마치 끝도 없는 방송(Broadcast) 같습니

다. 클라이언트 응용프로그램은 서버에서 발행되는 데이터를 들으며 특정 우편번호에 대한 것만 수신합니다. 기본적으로 뉴욕시의 우편번호(10001)가 설정되었지만 매개변수로 변경이 가능합니다.

wuclient.c: 기상 변경 클라이언트

```
//  Weather update client
//  Connects SUB socket to tcp://localhost:5556
//  Collects weather updates and finds avg temp in zipcode

#include "zhelpers.h"

int main (int argc, char *argv [])
{
    //  Socket to talk to server
    printf ("Collecting updates from weather server...\n");
    void *context = zmq_ctx_new ();
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    int rc = zmq_connect (subscriber, "tcp://localhost:5556");
    assert (rc == 0);

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

그림 4 - 발행-구독

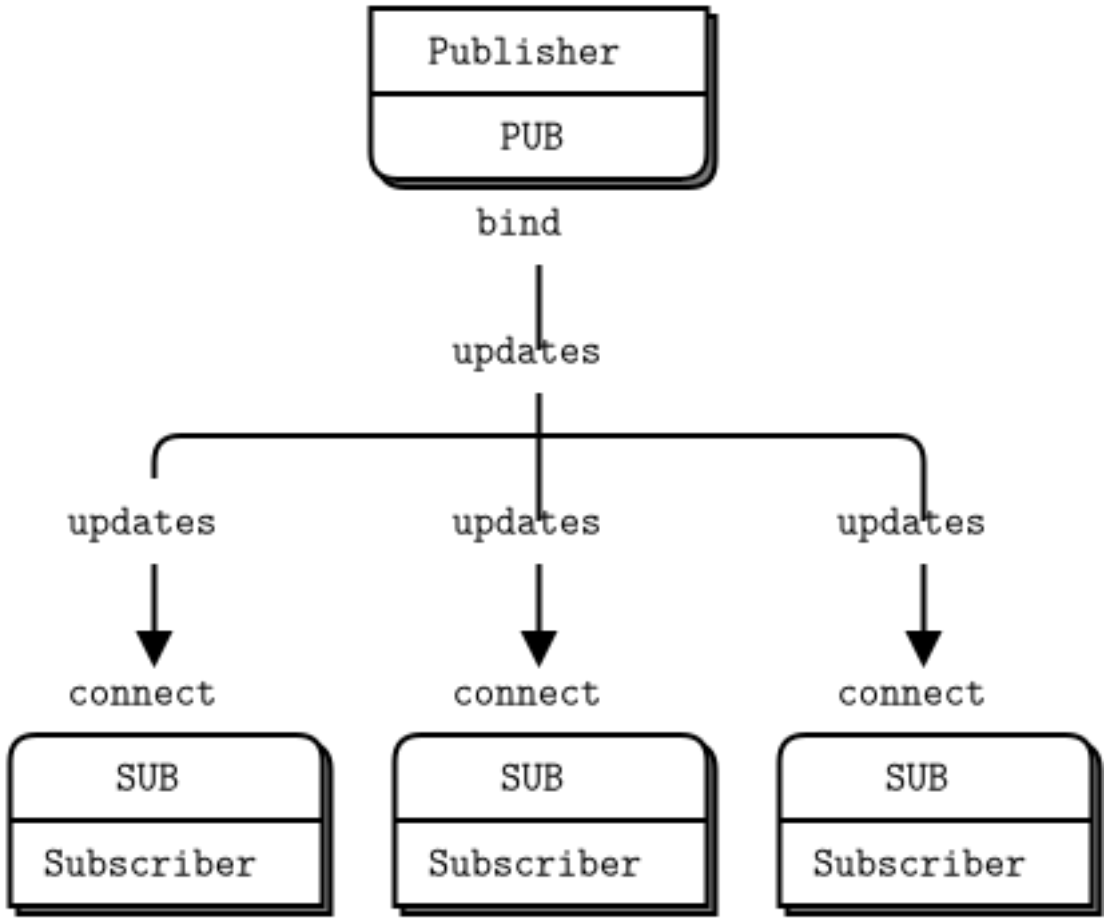


그림 5: Publish-Subscribe

클라이언트는 SUB 소켓을 사용할 때 반드시 `zmq_setsocket()`와 `ZMQ_SUBSCRIBE` 을 사용하여 구독을 설정해야 하며 미수행시 어떤 메시지도 받을 수 없습니다. 구독자는 다수의 구독을 설정할 수 있으며 특정 구독은 취소할 수 있습니다. 즉, 구독자에 설정된 구독과 일치하면 변경정보를 받습니다. 구독은 반드시 출력 가능한 문자열일 필요는 없으며, 동작 방법은 `zmq_setsocket()` 참조하시기 바랍니다.

PUB-SUB 소켓을 쌍으로 비동기로 작동하며 일반적으로 클라이언트 루프상에서 `zmq_recv()`를 호출합니다. SUB 소켓에 메시지를 보내려고 하면 오류가 발생합니다. 마찬가지로 PUB 소켓으로 `zmq_recv()`를 호출해서는 안됩니다.

이론적으로는 클라이언트, 서버에서 누가 `bind()`를 하던 `connect()`를 하는지는 문제가 되지 않지만 실제로는 PUB 소켓일 경우 `bind()`, SUB 소켓일 경우 `connect()`를 수행합니다.

- [옮긴이] 일반적으로 1:N 통신을 경우 1에 해당하는 단말은 'bind()', N에 해당하는 단말은 'connect()"을 수행합니다.

PUB-SUB 소켓에 대한 한 가지 중요한 사항으로 구독자가 메시지를 받기 시작하는 시기를 정확히 알 수 없습니다. 구독자를 기동하고 기다리는 동안 발행자를 기동 하면, 구독자는 항상 발행자가 보내는 첫 번째 메시지를 유실힙니다. 사유는 구독자가 발행자에게 연결할 때(작지면 0이 아닌 시간 소요) 발행자가 이미 메시지를 전송했기 때문입니다.

이 "더딘 결합(slow joiner)" 현상은 사람들을 놀라게 하며 나중에 자세히 설명하겠습니다. ØMQ는 백그라운드에서 비동기 I/O를 수행합니다. 백그라운드에서 두 개의 노드에서 아래의 순서로 통신을 수행합니다.

- 구독자는 단말에 연결(connect)하여 메시지를 수신하고 카운트합니다.
- 발행자는 단말에 바인딩(bind)하고 즉시 1,000개의 메시지들을 전송합니다.

그러면 구독자는 아마 아무것도 받지 못할 것입니다. 눈을 번쩍 뜨고 구독 필터를 올바르게 설정한 후 다시 시도하여도 구독자는 여전히 아무것도 받지 못합니다.

TCP 연결 생성은 네트워크와 연결 대상들 간의 경유 단계(hops)에 따라 몇 밀리초(milliseconds) 지연을 발생시키며, 그 시간 동안 ØMQ는 많은 메시지를 보낼 수 있습니다. 편의상 연결 설정에 5밀리초가 소요되고, 발행자가 초당 100만 메시지를 송신할 수 있다면 발행자와 연결에 필요한 5밀리초 사이에 5000개 메시지를 전송할 수 있습니다.

- [옮긴이] 홉(hops)은 컴퓨터 네트워크에서 출발지와 목적지 사이에 위치한 경로의 한 부분이다. 데이터 패킷은 브리지, 라우터, 게이트웨이를 거치면서 출발지에서 목적지로 경유한다. 패킷이 다음 네트워크 장비로 이동할 때마다 홉이 하나 발생한다. 홉 카운트는 데이터가 출발지와 목적지 사이에서 통과해야 하는 중간 장치들의 개수를 가리킨다.

"2장-소켓 및 패턴"에서 발행자와 구독자 간에 동기화하여 데이터 유실하지 않는 방법을 설명하겠습니다. 단순하고 무식하지만 발행자가 `sleep()`을 호출하여 지연시키는 방법도 있지만 사용할 경우 응용프로그램 지연이 발생할 수 있습니다.

동기화의 대안은 단순히 발행된 데이터 스트림이 무한하고 시작과 끝이 없다고 가정하여, 구독자가 시작되기 전에 무슨 일이 일어났는지 신경 쓰지 않는다고 가정합니다. 이것이 우리가 날씨 클라이언트 예제를 구축한 방법입니다.

정리하면, 클라이언트는 선택한 우편번호를 구독하고 해당 우편번호에 대한 100개 변경정보들을 수집합니다. 우편번호가 무작위로 분포하는 경우에는 약 1천만 변경정보가 발생합니다. 클라이언트를 시작한 후 서버를 시작하면 클라이언트는 문제없이 작동합니다. 서버를 중지하고 재시작해도 클라이언트는 계속 작동합니다. 클라이언트가 100의 변경정보들을 수집하면 평균을 계산하여 화면에 출력하고 종료합니다.

- [옮긴이] 빌드 및 테스트

```
> cl -EHsc wuserver.c libzmq.lib
> cl -EHsc wuclient.c libzmq.lib

> ./wuserver

> ./wuclient
Collecting updates from weather server...
Average temperature for zipcode '10001' was 35F
```

몇 가지 발행-구독(pub-sub) 패턴의 주요 사항들은 다음과 같습니다.

- 구독자는 한 개 이상의 발행자와 연결할 수 있다. 매번 하나의 연결을 사용하며 데이터가 도착하면 개별 대기열(fair-queued)에 쌓이며 개별 발행자로부터 수신된 데이터의 대기열은 다른 것에 영향을 주지 않습니다.
- 발행자에 어떤 구독자들도 연결하지 않았다면, 모든 메시지들은 버려집니다.
- TCP를 사용하고 구독자의 처리가 지연된다면 메시지는 발행자의 대기열에 추가됩니다. 이후 발행자를 최고수위 표시(HWM(high-water mark))를 사용하여 보호하는 방법을 알아보겠습니다.

- ØMQ v3.x부터, 발행자에서도 필터 기능이 추가되었습니다(tcp:// 혹은 ipc://). epgm:// 통신규약을 사용하여 구독자에서도 필터를 사용할 수 있습니다. ØMQ v2.x에서는 필터 기능은 구독자에서만 사용 가능했습니다.

아래는 2011 년에 구입한 인텔 i5 노트북에서 1천만 메시지를 수신하고 필터링하는 데 걸린 시간입니다.

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

real    0m4.470s
user    0m0.000s
sys     0m0.008s
```

0.14 나누어서 정복하라

그림 5 - 병렬 파이프라인(parallel pipeline)

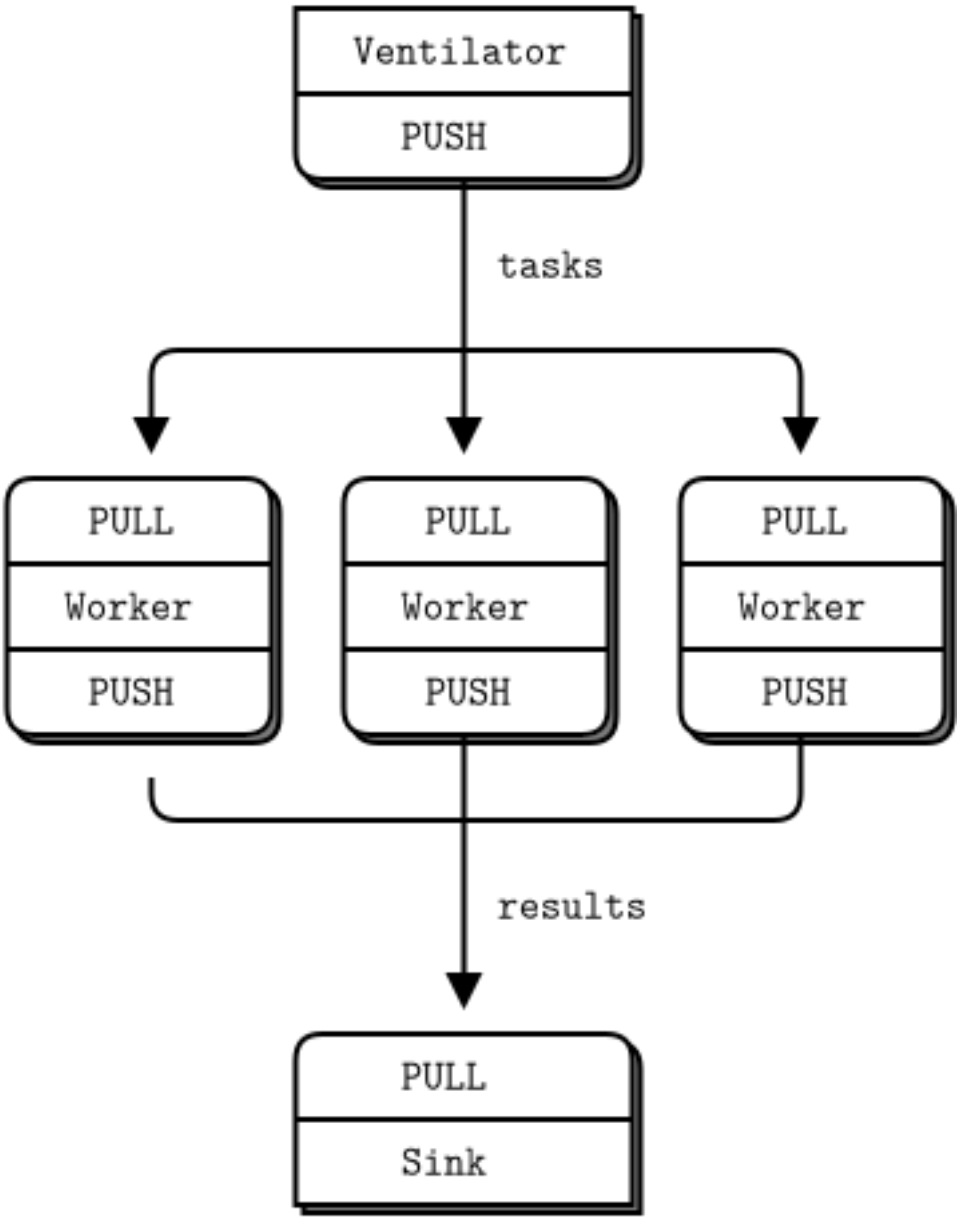


그림 6: 병렬 파이프라인

마지막 예는 작은 슈퍼 컴퓨터를 만들어 보겠으며, 슈퍼 컴퓨팅 응용프로그램은 전형적인 병렬 처리 모델입니다.

- 호흡기(ventilator)는 병렬 처리 가능한 작업들을 생성합니다.

- 일련의 작업자(Worker)들이 작업들을 처리합니다.
- 수집기(Sink)는 작업자 프로세스들로부터 작업 결과를 수집합니다.

실제로 작업자는 초고속 컴퓨터들에서 병렬 처리로 실행되며 GPU(그래픽 처리 장치)를 사용하여 어려운 연산을 수행합니다. 호흡기 소스는 100개의 작업할 메시지들을 만들며 각 메시지에는 작업자의 부하를 시간으로 전달(<100 msec)하여 해당 시간 동안 대기하게 합니다.

taskvent.c: 호흡기

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

#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

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

    //  Socket to send start of batch message on
    void *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...\n");
```

```
    //  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
    int task_nbr;
    int total_msec = 0;     //  Total expected cost in msecs
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        int workload;
        //  Random workload from 1 to 100msecs
        workload = randof (100) + 1;
        total_msec += workload;
        char string [10];
        sprintf (string, "%d", workload);
        s_send (sender, string);
    }
    printf ("Total expected cost: %d msec\n", total_msec);

    zmq_close (sink);
    zmq_close (sender);
    zmq_ctx_destroy (context);
    return 0;
}
```

작업자는 호흡기로부터 받은 메시지에 지정된 시간(밀리초)을 받아 시간만큼 대기하고 수집기에 메시지를 전달합니다.

taskwork.c 작업자

```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 (void)
{
    //  Socket to receive messages on
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  Socket to send messages to
    void *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;
}
```

수집기는 100개의 작업들의 결과를 수집하고 전체 경과 시간을 계산하여 작업자들의 개수에 따른 병렬 처리로 작업 시간 개선을 확인합니다(예 : 작업자가 1개일 때 3초이면, 작업자가 3개이면 1초).

tasksink.c: sinker

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

#include "zhelpers.h"

int main (void)
{
    //  Prepare our context and socket
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

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

    //  Start our clock now
    int64_t start_time = s_clock ();

    //  Process 100 confirmations
```

```
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if (task_nbr % 10 == 0)
            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;
}
```

평균 실행 시간은 대략 5초 정도입니다. 작업자의 개수를 1, 2, 4개로 했을 때의 결과는 다음과 같습니다.

- 1 worker: total elapsed time: 5034 msecs.
- 2 worker: total elapsed time: 2421 msecs.
- 4 worker: total elapsed time: 1018 msecs.
- [옮긴이] 빌드 및 테스트

```
> cl -EHsc taskvent.c libzmq.lib
> cl -EHsc taskwork.c libzmq.lib
> cl -EHsc tasksink.c libzmq.lib

// 1개의 worker로 작업할 경우
> ./taskvent
Press Enter when the workers are ready:
Sending tasks to workers...
Total expected cost: 4994 msec
> ./taskwork
86.42.46.52.23.51.69.75.37.93.4.67.41.23.35.65.16.77.74.26.53.52.74.10.60.96.
23.14.37.94.47.54.78.87.17.83.86.25.63.46.77.72.94.36.57.90.39.82.38.4.96.18.
84.66.12.56.59.42.70.29.54.31.21.22.70.28.25.44.46.56.44.75.92.79.53.42.27.
100.5.38.74.92.20.58.20.24.39.63.88.64.7.36.29.11.20.48.84.21.20.2.
> ./tasksink
:.........:.........:.........:.........
:.........:.........:.........:.........
:.........:.........
Total elapsed time: 5091 msec

// 2개의 worker로 작업할 경우
> ./taskvent
Press Enter when the workers are ready:
Sending tasks to workers...
Total expected cost: 5045 msec
> ./taskwork
87.79.38.90.83.51.13.26.74.33.64.81.18.96.36.28.95.2.27.12.79.10.43.94.83.49.
24.45.61.44.26.14.77.16.67.51.27.34.41.92.38.37.88.18.16.14.28.57.63.49.
> ./taskwork
```

```
12.17.65.21.79.83.82.9.12.97.27.36.78.47.34.65.49.6.51.96.70.18.57.55.32.84.
98.91.66.72.33.88.43.33.91.89.87.29.42.32.2.79.80.99.16.13.21.76.27.38.
> ./tasksink
:.........:.........:.........:.........
:.........:.........:.........:.........
:.........:.........
Total elapsed time: 2675 msec
```

코드상에서 특정 부분에 대하여 상세히 다루어 보겠습니다.

- 작업자는 상류의 호흡기와 하류의 수집기를 연결(connect)하며 자유롭게 작업자를 추가할 수 있습니다. 만약 작업자가 호흡기와 수집기에 바인딩(bind)를 수행할 경우 호흡기와 싱크 변경(및 추가) 시마다 작업자를 추가해야 합니다. 작업자에서 동적 요소로 연결(connect)을 수행하고 호흡기와 싱크는 정적 요소로 바인딩(bind)를 수행합니다.
- 모든 작업자가 시작할 때까지 호흡기의 작업 시작과 동기화해야 합니다. 이것은 ØMQ의 일반적인 원칙이며 간단한 해결 방법은 없습니다. `zmq_connect()`함수 수행 시 일정 정도의 시간이 소요되며, 여러 작업자가 호흡기에 연결할 때 첫 번째 작업자가 성공적으로 연결하여 메시지를 수신하는 동안 다른 작업자는 연결을 수행합니다. 동기화되어야만 시스템은 병렬로 동작합니다. 호흡기에서 작업자들이 동기화를 위하여 사용한 `getchar()`를 제거할 경우 어떤 일이 발생하는지 확인해 보시기 바랍니다.
- 호흡기의 PUSH 소켓은 작업자들에게 작업들을 분배하고 이것을 "부하 분산"이라고 부릅니다(호흡기에 작업자들을 연결하여 동기화되었다는 가정).
- 수집기의 PULL 소켓은 작업자들의 결과를 균등하게 수집하여 이를 "공정-대기열(fair-queuing)"이라고 합니다.

그림 6 - 공정 대기열

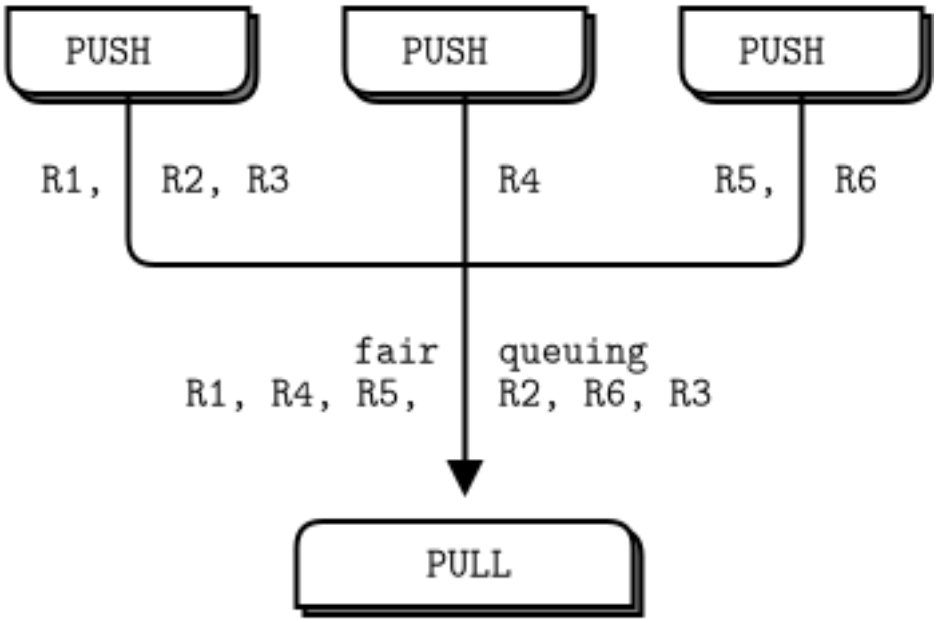


그림 7: Fair Queuing

파이프라인 패턴도 "더딘 결합(slow joiner)" 현상을 나타내며 PUSH 소켓이 올바르게 "부하 분산"을 수행하지 못하는 경우도 있습니다. PUSH-PULL을 사용하는 경우의 특정 작업자가 다른 작업자보다 많은 메시지를 처리하는 경우가 있습니다. 이는 특정 작업자의 PULL 소켓이 이미 호흡기에 연결되어 있어, 다른 작업자의 PULL 소켓에 연결하려는 동안 메시지를 처리하기 때문입니다. 올바른 "부하 분산"을 수행하기 위해서는 "3장-고급 요청-응답 패턴"을 참조하시기 바랍니다.

0.15 ØMQ 프로그램하기

예제를 통하여 당신은 QMQ 라이브러리를 사용하여 특정 응용프로그램을 작성하고 싶겠지만, 시작하기 전에 기본적인 권고 사항을 통해 많은 스트레스와 혼란을 줄일 수가 있습니다.

- 단계별로 ØMQ 배우기 : 단지 하나의 단순한 API이지만, 세상의 가능성을 숨기고 있습니다. 천천히 가능성을 취하고 자신의 것으로 만드십시오.

- 멋진 코드 작성 : 추악한 코드는 오류를 숨기고 있으며 다른 사람의 도움을 받기 어렵습니다. 의미 있는 단어로 변수의 명을 사용하면 변수가 하고자 하는 바를 전달할 수 있습니다. 좋은 코드를 작성하면 당신의 세계가 좀 더 편안해질 것입니다.
- 당신이 만든 것을 테스트하십시오 : 당신의 프로그램이 작동하지 않을 때, 당신은 탓할 다섯 줄이 무엇인지 알아야 합니다. 이것은 ØMQ가 마술을 부릴 때이며, 처음에는 몇 번 시도해도 동작하지 않다가 점차 익숙해지면 잘할 수 있습니다.
- 무엇인가 바라는 대로 동작하지 않을 때, 코드를 작은 조작으로 쪼개어 각각에 대하여 테스트를 하면 동작하지 않는 부분을 찾을 수 있습니다. ØMQ는 기본적으로 모듈라 코드로 만들 수 있게 하며 이것을 이용하십시오.
- 필요하다면 추상화(클래스, 메서드 등 무엇이든지)하십시오. 만약 단순히 복사/붙여넣기를 한다면 이것은 오류도 복사/붙여 넣기를 하는 것입니다.

0.15.1 컨텍스트 권한 얻기

ØMQ 응용프로그램은 컨텍스트를 생성하여 시작하고, 컨텍스트를 소켓을 생성하는데 사용합니다. C 언어에서는 `zmq_ctx_new()` 호출을 통해 컨텍스트를 생성합니다. 프로세스에서 정확히 하나의 컨텍스트를 생성하고 사용해야 합니다. 기능적으로 컨텍스트는 단일 프로세스 내의 모든 소켓들에 대한 컨테이너이며, inproc 소켓을 통하여 프로세스 내의 스레드들 간에 빠른 연결을 하게 합니다.

- [옮긴이] 컨텍스트 및 소켓의 생성

C 언어의 경우

```
void *context = zmq_ctx_new ();
void *responder = zmq_socket (context, ZMQ_REP);
int rc = zmq_bind (responder, "tcp://*:5555");
```

C++ 언어의 경우

```
zmq::context_t context (1);
zmq::socket_t socket (context, ZMQ_REP);
socket.bind ("tcp://*:5555");
```

실행 시 하나의 프로세스에서 2개의 컨텍스트를 가진다면 그들은 독자적인 ØMQ 인스턴스가 됩니다. 명시적으로 그런 상황이 필요하면 가능하지만 다음 사항을 기억하시기 바랍니다. 컨텍스트 생성(C 언어 : `zmq_ctx_new()`) 호출하여 프로세스 시작한 경우, 종료 시 컨텍스트 제거(C 언어 : `zmq_ctx_destory()`) 호출합니다.

`fork()`시스템 호출을 사용하는 경우, 각 프로세스는 자신의 컨텍스트를 필요로 합니다. 메인 프로세스에서 `zmq_ctx_new()`를 호출한 후`fork()`하면 자식 프로세스는 자신의 컨텍스트를 얻습니다. 일반적으로 주요 처리는 자식 프로세스에서 수행하고 부모 프로세스는 자식 프로세스를 관리하도록 합니다.

0.15.2 깨끗하게 종료하기

세련된 프로그래머는 세련된 암살자와 같은 모토를 공유합니다. : 항상 일이 끝나면 깨끗하게 정리하기. ØMQ를 사용할 경우, Python과 같은 개발언어는 자동으로 메모리 해제를 수행하지만, C 언어의 경우 종료 시 각 객체들에 대한 메모리 해제를 수행하지 않을 경우 메모리 누수 현상이나 불안정된 상태를 가질 수 있습니다.

메모리 누수도 그중 하나이며 ØMQ를 사용하는 응용프로그램을 종료 시에 주의해야 합니다. 그 이유는 만약 소켓을 오픈한 상태로 `zmq_ctx_destroy()` 함수를 호출하면 영구적으로 응용프로그램은 중단(hang)되게 됩니다. 그리고 모든 소켓을 닫은 후에 `zmq_ctx_destroy()` 호출 시 소켓을 닫기 전에 LINGER를 0으로 설정하지 않는 한 보류 중인 연결 또는 전송이 있으면 기본적으로 영원히 기다립니다.

- [옮긴이] 지연(LINGER)은 TCP 소켓의 단절 상태에 대응하기 위한 옵션으로 0일 경우 연결 상태를 종료하고 소켓 버퍼에 남아있는 데이터를 버리는 비정상 종료를 수행합니다.

ØMQ 객체들에서 주의해서 처리해야 하는 것은 메시지와 소켓, 컨텍스트 3가지입니다. 다행히도 단순한 프로그램에서 이들을 취급하는 것은 매우 간단합니다.

- 가능한 zmq_send()와 zmq_recv()를 사용하여 zmq_msg_t 객체의 사용을 회피해야 합니다.
- zmq_msg_recv()를 사용하여 메시지를 수신 시 zmq_msg_close()를 호출하여 수신된 메시지를 처리하자마자 항상 해제하십시오.
- 많은 소켓들을 열고(open)하고 닫고(close)할 경우 응용프로그램을 재설계가 필요합니다. 일부 경우에는 컨텍스트를 제거될 때까지 소켓 핸들이 유지될 수 있습니다.
- 프로그램을 종료할 때 소켓을 닫고 zmq_ctx_destroy()를 호출하여 컨텍스트를 제거합니다.

특히 C 언어를 사용할 경우에 해당되며, 일부 개발 언어(예 : Python)에서는 자동 객체 제거를 하며, 소켓과 컨텍스트 객체가 사용되는 범위를 벗어 나면 제거됩니다. 만약 예외(exception)를 사용할 경우 파이널(final) 블럭에서 자원의 해제가 필요합니다.

멀티스레드를 사용하는 경우, 이러한 처리는 더욱 복잡해집니다. 멀티스레드에 관해서는 다음 장에서 다루지만 경고를 무시하고 시도하려는 사람도 있기 때문에, 아래에서 멀티스레드 ØMQ 응용프로그램을 제대로 종료하기 위한 임시 가이드를 제공합니다.

- 첫째, 멀티스레드에서 동일 소켓을 사용하지 마십시오. 재미있을 것같다고 생각조차 마십시요. 제발 하지 마십시오.

- 둘째, 스레드들에 대한 지속되는 요청들에 대하여 개별적으로 소켓 종료해야 합니다. 적절한 방법으로(1초 정도) 지연(LINGER) 설정하여 소켓을 닫을 수 있습니다. 컨텍스트를 삭제할 때 언어 바인딩이 자동으로 작업을 수행하지 않으면, 언어 바인딩 개발자에게 수정 요청을 보내시기 바랍니다.

- 마지막으로 컨텍스트 제거하기입니다. 메인 스레드의 zmq_ctx_destory()는 컨텍스트를 공유하는 다른 스레드의 소켓이 모두 닫혀질 때까지 차단되며 오류를 발생합니다. 이러한 오류가 발생할 경우 지연(LINGER) 설정하고 해당 스레드의 소켓을 닫은 후에 종료하시기 바랍니다. 동일한 컨텍스트를 두 번 제거하지 마십시오. 메인 스레드의 zmq_ctx_destroy()는 알고 있는 모든 소켓이 안전하게 닫힐 때까지 차단됩니다.

끝났습니다. 이것은 매우 복잡하고 고통을 수반하지만 개발 언어별 ØMQ 바인딩을 개발하는 사람이 피땀을 흘려 자동으로 소켓을 닫아 주는 경우도 있어 반드시 이렇게 할 필요는 없을 수도 있습니다.

0.16 왜 ØMQ가 필요한가

ØMQ 동작 방식을 보았으며, 다시 "왜(why)"로 돌아가 보겠습니다.

오늘날 많은 응용프로그램들이 일종의 네트워크상(랜 혹은 인터넷)에서 각종 구성 요소들로 확장되고 있으며 많은 개발자들이 TCP와 UDP 통신규약을 사용하고 있습니다. 이러한 통신규약을 사용하지 어렵지는 않지만 메시지를 전송(A->B)하는 것과 신뢰성 있게 메시지를 전송하는 것에는 큰 차이가 있습니다.

원시(RAW) TCP를 사용할 때 직면하는 전통적인 문제들을 보도록 하겠습니다. 재사용 가능한 메시징 계층을 구현하기 위해서는 아래의 문제들을 해결해야 합니다.

- I/O를 어떻게 처리합니까?
- 응용프로그램이 차단할까요, 아니면 백그라운드에서 I/O를 처리합니까? 이것은 주요 설계 결정입니다. I/O를 차단하면 제대로 확장하기 어려운 아키텍처가 되며 백그라운드 I/O는 제대로 다루기 어렵습니다.
- 일시적으로 사라지는 동적 컴포넌트를 어떻게 다루어야 할까요?
- 일반적으로 작업 수행 단위들을 클라이언트와 서버, 머신으로 나누면 서버들은 유지되며 서버 간의 연결이 필요할 경우라든지, 매번 다시 연결 해야 하는 경우가 생길 수 있습니다.
- 네트워크상에서 메시지를 어떻게 표현합니까?
- 프레임 데이터를 쉽게 쓰고 읽게하며, 버퍼 오버플로우로부터 안전하고 작은 메시지에 대하여 효율적으로 처리 필요합니다.
- 즉시 전달할 수 없는 메시지를 어떻게 처리합니까?
- 온라인이 될 때까지 기다려야 하는 상황에서 해당 메시지를 버리든지, 데이터베이스에 넣든지 혹은 메모리 대기열에 관리해야 합니다.
- 어디에 메시지 대기열을 보관하나요?
- 대기열로부터 데이터 읽기가 지연되고 새로 대기열이 만들어져야 하면 어떤 전략으로 대응할 것인가.
- 메시지 유실에 어떻게 대응하나요?
- 신규 데이터를 기다리거나 재전송을 요청하거나 메시지 유실을 방지하기 위한 신뢰성

있는 계층을 만들어야 할지. 만약 해당 계층이 깨진다면 어떻게 할 것인지

- 다른 네트워크 전송계층을 사용하는 경우에는 어떻게 하나요?
- 예를 들면 TCP 유니케스트 대신에 멀티캐스트, IPV6 등의 전송계층 간의 통신을 위하여 응용프로그램 재작성해야 할지 다른 계층으로 전송계층을 추상화할 것인지
- 메시지들을 어떻게 라우팅 할까요?
- 동일한 메시지를 여러 개의 단말들에게 보낼 수 있는지, 원래 요청에 대하여 회신을 보낼 수 있는지
- 다른 개발 언어에서 어떻게 API를 작성할까요?
- 다른 개발언어에서도 상호운영성 보장이 가능한지, 효율적이고 안정적인 수단을 제공할 수 있는지, 라이브러리 재작성해야 하는지
- 다른 아키텍처(예 운영체제) 간에 어떻게 동일한 데이터를 표현할까요?
- 데이터 유형별 특정 엔코딩이 강제로 적용합니까? 이것이 상위 계층이 아닌 메시징 시스템의 역할은 얼마나 됩니까?
- 네트워크 장애를 어떻게 처리할지?
- 기다렸다가 재시도하기, 무시하기, 종료합니까?

하둡 주키퍼(Hadoop Zookeeper)는 C API로 작성되어 있으며 클라이언트/서버 네트워크 통신규약을 사용하며 SELECT(일정 주기) 대산에 POLL(이벤트 방식)을 사용하는 효율성을 가졌으며 범용 메세징 계층과 명시적인 네트워크 수준 통신규약을 사용하고 있었습니다. 하지만 이미 만들어 놓은 대상을 재사용하지 않고 계속 만드는 것(build wheel over and over)은 지극히 낭비적이었습니다.

하지만 어떻게 재사용 가능한 메시징 계층을 만들까요? 많은 프로젝트에서 이 기술이 요구하였지만 사람들은 아직도 TCP 소켓을 사용하여 많은 문제 목록을 계속 반복해서 해결하고 있습니다.

하지만 재사용 가능한 메세징 시스템을 만드는 것은 정말 어려운 것으로 밝혀졌으며 일부 "「FOSS 프로젝트」" 에서도 시도하였습니다. 그래서 상용 메세징 제품이 복잡하고 고비용이고 유연하지 못하고 예민한가에 대한 이유입니다. 2006년 "iMatix"는 AMQP를 설계하여 FOSS 개발자들에서 공유하였으며 아마도 메세징 시스템에 대한 첫 번째 재사용 방법을 제공하였습니다. AMQP는 다른 많은 설계들보다 잘 동작하지만 상대적으로 복잡하고 고비용이고 예민하여, 사용하기 위해서는 몇 주나 몇 달의 학습 기간이 필요합니다.

- [옮긴이] AMQP(Advanced Message Queuing Protocol)는 메시지 지향 미들웨어를 위한 개방형 표준 응용 계층 통신규약이며 AMQP 기반 제품은 RabbitMQ, ActiveMQ, ØMQ가 있습니다.

그림 7 - 태초의 메세징(Messaging as it Starts)

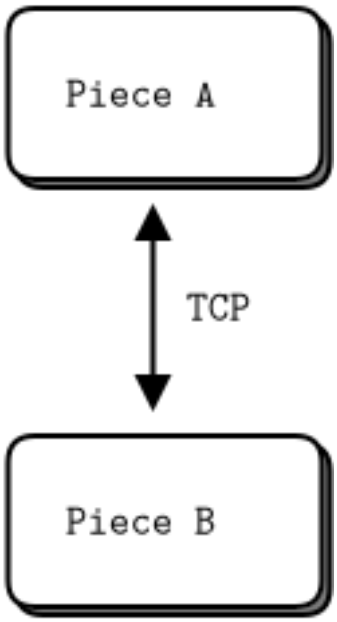


그림 8: 태초의 메세징

대부분의 메세징 프로젝트와 같이 AMQP에서도 재사용에 대한 오랫동안 지속된 문제를 해결하려 했으며, 새로운 개념으로 "브로커(Broker)"를 발명하였으며, 브로커는 주소 지정, 라우팅, 대기열 관리를 수행하였습니다. 그 결과 클라이언트/서버 통신규약과 일련의 API들은 응용프로그램들이 브로커를 통하여 소통할 수 있게 되었습니다. 브로커는 거대한 네트워크상에서 복잡성을 제거하는 데는 뛰어났지만, 주키퍼처럼 브로커 기반 메세징은 상황을 개선하기는 보다 더 나쁘게 만들었습니다. 이것은 브로커라는 부가적인 거대한 요소를 추가함으로 새로운 단일 장애점(SPOF, Single Point Of Failure)이 되었습니다. 브로커는 빠르게 병목 지점이 되어감에 따라 새로운 위험을 관리해야 했습니다. 소프트웨어적으로 이러한 이슈(장애조치)를 해결하기 위하여 브로커를 2개, 3개, 4개 추기 해야 했습니다. 이것을 결국 더욱 많은 조각으로 나누어지며, 복잡성을 증가하고, 중단 지점을 늘어가게 했습니다.

그리고 브로커 중심 설정은 이것을 위한 별도의 운영 조직을 필요하게 되었으며, 브로커가 매일 잘 동작하는지 모니터링하고 비정상적인 동작을 보일 때는 조정해야 했습니다. 더 많은 머신과 백업을 위한 추가적인 머신 그리고 이것을 관리하기 위한 사람 등, 브로커는 많은 데이터가 움직이는 거대한 응용프로그램에 대하여 오랫동안 여러 개의 팀을 통해 운영이 가능할 경우 적절합니다.

그림 8 - 변화된 메세징 (Messaging as it Becomes)

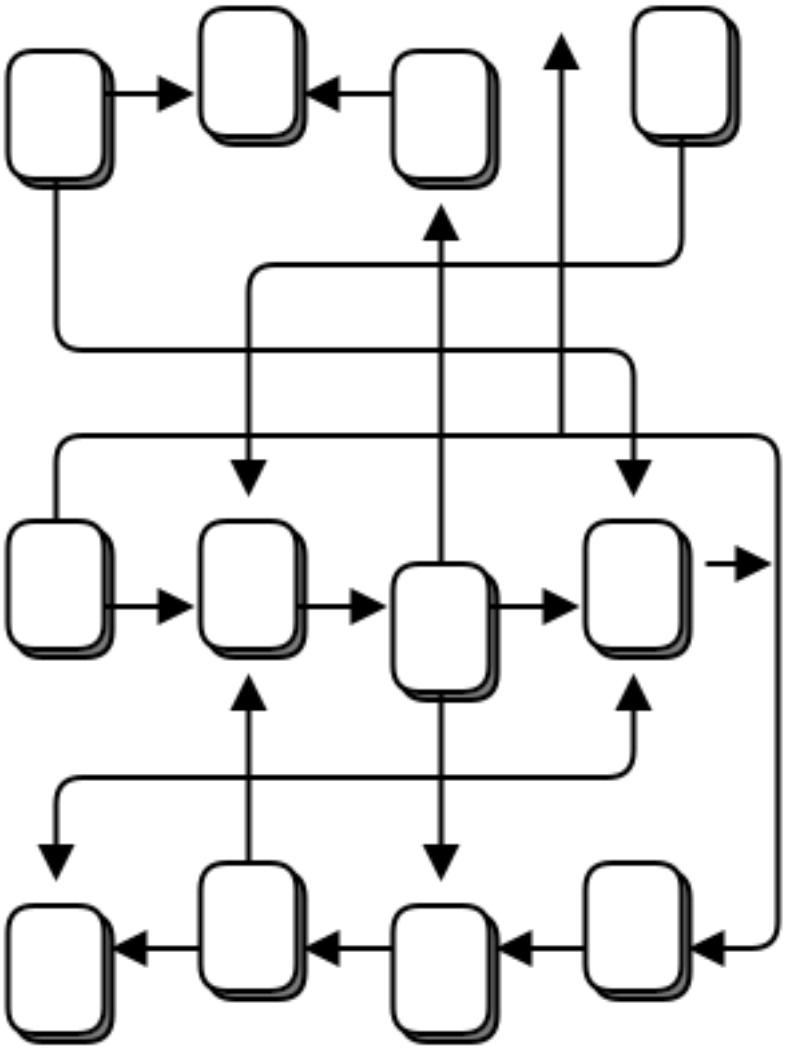


그림 9: Messaging as it Becomes

중소규모의 응용프로그램 개발자들이 네트워크 프로그래밍을 회피하거나 규모를 조정할 수 없는 모노리식(monolithic) 응용프로그램을 만들면서 덧에 빠지거나 잘못된 네트워크 프로그래밍으로 인하여 불안정하고 복잡한 응용프로그램을 만들어 유지 보수가 어렵게 됩니다. 또는 상용 메시징 제품을 의지하고 확장 가능하지만 투자 비용이 크거나 도태되는 기술일 경우도 있습니다. 지난 세기에서 메시징이 거대한 화두였지만 탁월한 선택은 없었으며 상용 메세징 제품을 지원하거나 라이선스를 판매하는 사람들에게는 축복이었지만, 사용자에게 비극이었기 때문입니다.

우리가 필요한 것은 메세징에 대한 작업은 단순하고 저비용이어야 하고 어떤 응용프로그램이나 운영체제에서도 동작해야 합니다. 이것은 어떤 의존성도 없는 라이브러리가 되어야 하며, 어떤 프로그램 언어에서도 사용 가능해야 합니다.

이것이 ØMQ입니다 : 내장된 라이브러리를 통하여 적은 비용과 효율적으로 네트워크상에서 대부분의 문제를 해결할 수 있습니다.

ØMQ의 특이점은 다음과 같습니다 :

- 백그라운드 스레드들에서 비동기 I/O 처리합니다.
- 백그라운드 스레드들은 응용프로그램 스레드들 간에 통신을 하며, 잠금 없는 자료 구조를 사용하여 동시성 ØMQ 응용프로그램은 잠금, 세마포어, 대기 상태와 같은 것들을 필요로 하지 않습니다.
- 서비스 구성요소는 동적 요구에 따라 들어오고 나갈 수 있으며 ØMQ는 자동으로 재연결할 수 있습니다.
- 서비스 구성요소를 어떤 순서로도 시작할 수 있으며 마치 서비스 지향 아키텍처처럼 네트워크상에서 언제든지 합류하거나 떠날 수 있습니다.
- 필요시 자동으로 메시지들을 대기열에 넣습니다.
- 지능적으로 수행하여 메시지를 대기열에 추가하기 전에 가능한 수신자에게 전송합니다.
- 가득 찬 대기열(Over-full Queue(HWM, 최고 수위 표시))을 다룰 수 있습니다.
- 대기열이 가득 차게 되면 ØMQ는 특정 메세징 종류(소위 “패턴”)에 따라 자동적으로 송신자의 메시지를 막거나 혹은 버릴 수 있습니다.
- 응용프로그램들을 임의의 전송계층상에서 상호 통신하게 합니다.
- TCP, PGM(multicast), inproc(in-process), ipc(inter-process) 등 다른 전송계층들을 사용하더라도 소스코드를 수정하지 않고 통신할 수 있습니다.

- 지연/차단된 수신자들을 메세징 패턴에 따라 다른 전략을 사용하여 안전하게 처리합니다.
- 요청-응답, 발행-구독과 같은 다양한 패턴을 통하여 메시지를 전송할 수 있습니다. 이러한 패턴은 네트워크의 구조와 위상을 생성하는 방법입니다.
- 단일 호출(`zmq_proxy()`)로 메시지를 대기열에 추가, 전달 또는 캡처하는 프록시를 만들 수 있습니다. 프록시는 네트워크의 상호 연결 복잡성을 줄일 수 있습니다.
- 네트워크 상의 단순한 프레이밍(Framing)을 사용하여 메시지를 전송한 상태 그대로 전달합니다. 10,000개 메시지를 전송하면 10,000개 메시지를 받게 됩니다.
- 메시지의 어떤 포맷도 강요하지 않습니다.
- 메시지들은 0에서 기가바이트로 거대할 수 있으며 데이터를 표현하기 위해서는 상위에 별도 제품을 사용하면 됩니다.
- 필요할 경우 자동 재시도를 통해 네트워크 장애를 지능적으로 처리합니다.
- 탄소 배출량을 줄입니다.
- CPU 자원을 덜 사용하여 전력을 덜 소모하게 하며, 오래된 컴퓨터도 사용할 수 있게 합니다. 엘 고어(Al Gore)는 ØMQ를 사랑할 겁니다.

사실 ØMQ는 나열한 것 이상의 것을 하며 네트워크 지원 응용프로그램 개발에 파괴적인 영향을 줄 수 있습니다. 표면적으로 소켓과 같은 API(`zmq_recv()`, `zmq_send()`)이지만 메시지 처리 절차는 내부적으로 일련의 메시지 처리 작업들로 쪼개져서 처리됩니다. 이것은 우아하고 자연스럽고 규모를 조정할 수 있으며 각각의 작업들은 임의의 전송계층에서 하나의 노드, 여러 개의 노드들과 매핑되어 처리됩니다. 2개의 노드들이 하나의 프로세스(노드는 스레드)에서, 2개의 노드들이 하나의 머신(노드는 프로세스)에서, 2개의 노드들이 하나의 네트워크(노드는 머신)에서 처리되며 응용프로그램 소스는 모두 동일합니다.

0.17 소켓 확장성

ØMQ의 확장성을 보기 위하여, 아래의 쉘 스크립터를 통하여 날씨 서버와 일련의 클라이언트들을 병렬로 시작해 보겠습니다.

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

클라이언트들을 실행하여 "top" 명령을 통하여 활성화된 프로세스를 볼 수 있으며 다음과 같이 보입니다(CPU 4 코어 머신).

```
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
```

날씨 서버는 단일 소켓을 가지고 병렬로 동작하는 5개 클라이언트들에게 데이터를 전송합니다. 우리는 수천 개의 동시성 클라이언트들을 가질 수도 있으며 서버는 클라이언트들을 보지 않으며 직접 통신하지도 않습니다. 이것이 ØMQ 소켓이 하나의 조그만 서버처럼 동작하는 것이며 조용히 클라이언트 요청들을 접수하여 네트워크에서 처리할 수 있을 만큼 빠르게 데이터를 전달합니다. 이것은 멀티스레드 서버로 CPU 자원을 최대한 쥐어짜도록 합니다.

0.18 ØMQ v2.2에서 ØMQ v3.2 업그레이드

0.18.1 호환되는 변경

이러한 변화는 기존의 응용프로그램 코드에 직접 영향을 주지 않습니다.

- PUB-SUB 필터링이 기존 구독자뿐만 아니라 발행자 측면에서도 수행할 수 있게 되었습니다. 이것은 많은 PUB-SUB 사용 사례에서 성능을 크게 개선합니다. 안전하게 v3.2과 v2.1/v2.2를 조합하여 사용 가능합니다.
- ØMQ v3.2에서 많은 신규 API가 추가되었습니다. (zmq_disconnect(), zmq_unbind(), zmq_monitor(), zmq_ctx_set()등)

0.18.2 호환되지 않는 변경

응용프로그램과 개발언어 바인딩에 영향을 주는 주요 변경 사항입니다.

- 송/수신 함수의 변경 : zmq_send()'와 'zmq_recv()는 서로 다른 함수지만 유사한 인터페이스를 가지고 있으며, 이전 기능들은 zmq_msg_send()와zmq_msg_recv()라는 함수로 제공됩니다.
- 증상 : 컴파일 오류가 발생합니다.
- 해결 방법 : 코드를 수정해야 합니다.
- zmq_send()'와 'zmq_recv()는 수행 성공 시 결과값으로 양수를 반환하며 오류가 발생하면 -1을 반환합니다. 버전 ØMQ v2는 성공 시 항상 0을 반환했습니다.
- 증상 : 정상적인 동작인데 오류가 발생한 것 같습니다.
- 해결 방법 : -1 반환값과 0이 아닌 반환값에 대하여 엄격한 테스트를 수행합니다.
- zmq_poll ()는 마이크로초가 아니라 밀리초 동안 대기합니다.
- 증상 : 응용프로그램의 응답이 지연됨(정확하게는 1000 배 느리게).
- 해결 방법 :zmq_poll ()를 호출할 때 새로 정의된ZMQ_POLL_MSEC 매크로를 이용하십시오.
- ZMQ_NOBLOCK 매크로는ZMQ_DONTWAIT로 이름이 변경되었습니다.
- 증상 : ZMQ_NOBLOCK 사용 시 컴파일 오류
- ZMQ_HWM 소켓 옵션은 ZMQ_SNDHWM과 ZMQ_RCVHWM로 분리되었습니다.
- 증상 : ZMQ_HWM 메크로 사용 시 컴파일 오류
- 대부분의 zmq_getsockopt() 옵션 값은 정수 값입니다.
- 증상 :zmq_setsockopt ()과 zmq_getsockopt ()를 실행할 때 오류가 발생합니다.

- ZMQ_SWAP 옵션은 제거되었습니다.
- 증상 : ZMQ_SWAP 사용 시 컴파일 오류.
- 해결 방법 : ZMQ_SWAP을 사용하는 코드를 다시 설계하십시오.

0.18.3 호환성 매크로

응용프로그램을 v2.x와 v3.2 모두에서 동작시키고 싶은 경우에 가능한 v3.2를 에뮬레이트하는 것이 좋습니다. 다음은 C/C++ 코드가 두 버전에서 정상적으로 작동하도록 도와주는 C 매크로 정의입니다(CZMQ에서 가져옴).

```
#ifndef ZMQ_DONTWAIT
# define ZMQ_DONTWAIT ZMQ_NOBLOCK
#endif
#if ZMQ_VERSION_MAJOR == 2
#   define zmq_msg_send(msg,sock,opt) zmq_send (sock, msg, opt)
#   define zmq_msg_recv(msg,sock,opt) zmq_recv (sock, msg, opt)
#   define zmq_ctx_destroy(context) zmq_term(context)
#   define ZMQ_POLL_MSEC 1000 // zmq_poll is usec
#   define ZMQ_SNDHWM ZMQ_HWM
#   define ZMQ_RCVHWM ZMQ_HWM
#elif ZMQ_VERSION_MAJOR == 3
#   define ZMQ_POLL_MSEC 1 // zmq_poll is msec
#endif
```

0.19 주의 - 불안정한 패러다임!

전통적인 네트워크 프로그래밍에서는 하나의 소켓이 하나의 쌍, 하나의 연결과 통신한다는 일반적인 가정을 기반으로 합니다. 물론 멀티캐스트 통신규약도 있지만 이것은 색다른 것입니다. 우리가 "하나의 소켓 = 하나의 연결"을 가정한다면 어떤 방법으로 우리의 아키텍처를 확장할

수 있을까요. 우리는 단일 소켓에 대하여 개별 동작하는 스레드들에 대한 로직을 만들어야 하고 각각의 스레드의 상태와 지능을 부여해야 합니다.

ØMQ 세계에서는 소켓들은 출입구로 빠르고 작은 백그라운드 통신 엔진으로 일련의 연결이 자동적으로 이루어지도록 관리합니다. 당신은 연결에 대한 오픈과 닫기 혹은 설정된 상태를 볼 수 없으며, 차단된 송/수신, 풀링을 사용하던지 간에 소켓과 소통할 수 있으며, 직접 연결 관리할 필요는 없습니다. 연결은 비공개이며 보이지 않으며 이것이 ØMQ 확장성의 핵심입니다.

이것은 소켓과 통신하는 코드의 변경 없이 주변에 있는 다양한 네트워크 통신규약의 연결들을 처리할 수 있게 합니다. ØMQ에 있는 메시징 패턴은 응용프로그램 코드에 있는 메시징 패턴보다 비용이 절감되고 확장성이 높아집니다.

하지만 일반적인 가정이 적용되지 않는 경우도 있습니다. 예제 코드를 읽을 때, 당신의 머리에 기존의 지식과 매핑하려고 할지도 모릅니다. "소켓"이라는 단어를 읽으면 '아, 이것은 다른 노드에 대한 연결을 나타냅니다"라고 생각하겠지만 잘못된 것입니다." 스레드"라는 단어를 읽으면 '아, 스레드가 다른 노드와의 연결을 나타냅니다"라고 생각할지 모르지만, 이 또한 잘못되었습니다.

이 가이드를 처음 읽고 있다면, 실제로 ØMQ의 코드를 사용할 수 있을 때까지 1,2일(혹은 3,4일) 소요됩니다. 특히, ØMQ가 어떻게 사물을 단순화하고 있는지에 대해 당신은 혼란스러워할 수 있으며 ØMQ에 일반적인 가정을 적용하려고 하면 동작하지 않을 것이라고 생각할 수 있습니다. 하지만 결국 진리와 깨달음을 경험하는 순간, 당신은 번쩍-우르릉-쾅쾅 깨달음(zap-pow-kaboom satori) 패러다임 전환을 하게 될 것입니다.

2장-소켓 및 패턴

1장 - 기본에서 몇 가지 ØMQ 패턴의 기본 예제로 ØMQ의 세계로 뛰어들었습니다 : 요청-응답, 발행-구독, 파이프라인. 이장에서는 우리의 손을 더럽히면서 실제 프로그램에서 이러한 도구들을 어떻게 사용하는지 배우게 될 것입니다.

다루는 내용은 다음과 같습니다.

- ØMQ 소켓을 만들고 사용하는 방법
- 소켓상에서 메시지 송/수신 방법
- ØMQ 비동기 I/O 모델상에서 응용프로그램 개발하기
- 하나의 스레드상에서 다중 소켓 처리하는 방법
- 치명적인 및 비치명적인 오류에 적절하게 대응하기
- Ctrl-C와 같은 인터럽트 처리하기
- ØMQ 응용프로그램을 깨끗하게 종료하기
- ØMQ 응용프로그램의 메모리 누수 여부를 확인하기
- 멀티파트 메시지 송/수신 방법
- 네트워크상에서 메시지 전달하기
- 단순 메시지 대기열 브로커 개발하기
- ØMQ에서 멀티스레드 응용프로그램 개발하기
- ØMQ를 스레드 간 신호에 사용하기
- ØMQ를 네트워크상 노드들 간 협업에 사용하기

- 발행-구독에서 메시지 봉투 생성하고 사용하기
- 메모리 오버플로우에 대비하여 최고수위 표시(HWM,high-water mark) 사용하기

0.20 소켓 API

솔직히 말하자면, ØMQ는 당신에게 일종의 유인 상술을 쓰는데, 그것에 대하여 사과하지 않겠습니다. ØMQ는 익숙한 소켓 기반 API를 제공하고 내부에 일련의 메시지 처리 엔진을 가지고 있으며 당신이 어떻게 분산 소프트웨어를 설계하고 개발하는지에 따라 점차 당신의 세계관으로 만들 수 있습니다.

- [옮긴이] 유인 상술(switch and bait)은 값싼 상품을 광고해서 소비자를 끌어들인 뒤 비싼 상품을 사게 하는 상술입니다.

소켓은 네트워크 프로그래밍에서는 표준 API이며 눈꺼풀처럼 눈알이 빰에 떨어지는 것을 막을 정도로 유용합니다. 특히 개발자에게 ØMQ 매력적인 점은 다른 개념 대신 소켓과 메시지를 사용한다란 것이며, 이러한 개념을 이끌어낸 Martin Sustrik에게 감사하고 싶습니다. "메시지 기반 미들웨어"로 방 전체를 긴장감으로 채울 것 문구를 "특별히 매운 소켓!"으로 변경하여 피자에 대한 이상한 갈망과 더 많은 것을 알고 싶게 합니다.

좋아하는 요리처럼, ØMQ 소켓도 쉽게 이해할 수 있습니다. ØMQ 소켓도 BSD 소켓과 같이 4개의 생명 주기가 있습니다.

- 소켓 생성 및 파괴를 합니다.
- 소켓의 운명의 생명 주기를 만듭니다(`zmq_socket()`, `zmq_close()` 참조).
- 소켓 구성 옵션 설정합니다.
- 필요할 경우 설정 필요합니다(`zmq_setsockopt()` 참조).
- 소켓 연결을 합니다.
- ØMQ 연결을 생성하여 소켓을 네트워크에 참여시킵니다(`zmq_bind()`, `zmq_connect()` 참조).
- 소켓을 통한 메시지 송/수신합니다.
- 다양한 통신 패턴에서 메세지 송신과 수신에 사용됩니다.(`zmq_msg_send()`, `zme_msg_recv()` 참조)

소켓은 항상 void 포인터이며, 메시지들은 구조체이며 C 언어에서 `zmq_msg_send()`에 메시지의 주소를 전달하거나 `zmq_msg_recv()` 메시지의 주소를 반환받습니다. 기억하기 쉽게 ØMQ에서 모든 소켓은 우리에게 속해 있지만 메시지들은 소스 코드상에 있습니다.

소켓 생성, 소멸 및 구성은 모든 객체에 대해 예상대로 동작하지만, ØMQ는 비동기식이며 소켓을 네트워크 토폴로지에 연결하는 방법에 따라 소켓을 사용하는데 영향을 미칩니다.

0.20.1 네트워크상에 소켓 넣기

2개의 노드 간의 연결을 생성하기 위하여, 한쪽 노드에 `zmq_bind()`를 그리고 다른 쪽 노드에 `zmq_connect()`를 사용할 수 있습니다. 일반적으로 `zmq_bind()`를 수행하는 노드를 서버(네트워크 주소가 고정됨)라고 하며 `zmq_connect()`를 수행하는 노드를 클라이언트(네트워크 주소가 모르거나 임시)라고 합니다. 우리는 "단말(endpoint)에 소켓을 바인딩"과 "단말에 소켓을 연결"라고 말하며, 단말은 네트워크 주소를 알 수 있어야 합니다.

ØMQ 연결은 전통적인 TCP 연결과 다소 차이가 있으며 주요한 차이는 다음과 같습니다.

- ØMQ는 임의의 전송방식(inproc, ipc, tcp, pgm, epgm)을 교차하여 사용할 수 있습니다.
- `zmq_inproc()`, `zmq_ipc()`, `zmq_tcp()`, `zmq_pgm()`, `zmq_epgm()` 참조
- 하나의 소켓에서 여러 개의 송/수신 연결들(connections)을 가질 수 있습니다.
- `zmq_accept()`가 같은 함수가 없습니다.
- 소켓이 단말에 바인딩되면 자동으로 연결을 수락합니다.
- ØMQ는 네트워크 연결을 백그라운드로 수행하며, 네트워크 연결이 끊기면 자동으로 재연결합니다(예를 들어 통신 대상이 사라졌다가 다시 돌아올 경우).
- 응용프로그램 코드가 이러한 연결에 대해 직접 작업하지 않습니다. : ØMQ 소켓에 내장되어 있습니다.

일반적인 네트워크 아키텍처상에서 클라이언트/서버 구성할 때, 서버는 정적 형태, 클라이언트는 동적 형태로, 각 노드에서 서버의 경우 `zmq_bind()`, 클라이언트에서는 `zmq_connect()`을 수행하지만 어떤 종류의 소켓(예 : REQ-REP, PUB-SUB, PUSH-PULL, ROUTER-DEALER)을 사용하는지에 연관되며, 특이한 네트워크 아키텍처에서는 예외가 있습니다. 이러한 소켓 유형을 나중에 살펴보도록 하겠습니다.

서버를 시작하기 전에 클라이언트를 수행하는 경우, 전통적인 네트워킹에서는 오류가 발생하지만, ØMQ의 경우 순서에 상관없이 시작과 중단을 할 수 있습니다. 클라이언트 노드에서 `zmq_connect()`를 수행하여 성공하면 소켓에 메시지를 전달하기 시작합니다. 일정 단계(희망하건대 메시지가 너무 많이 대기열에 쌓이게 되어 버리거나 차단되기 전까지)까지, 서버가 기동 되어 `zmq_bind()`를 하게 되면 ØMQ는 메시지를 전송하게 됩니다.

서버 노드는 다수의 단말들(통신규약과 네트워크 주소의 조합)을 바인딩할 수 있으며 이것도 하나의 소켓을 통해 가능합니다. 즉 서로 다른 전송방식(inproc, ipc, pgm, tcp 등)으로 연결을 수락한다.

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

UDP를 제외하고 대부분의 전송방식에서 동일한 단말을 두 번 바인딩할 수 없습니다. 그러나 ipc(프로세스 간 통신) 전송계층은 하나의 프로세스가 첫 번째 프로세스에서 이미 사용된 단말에 바인딩하게 합니다. 이것은 프로세스 간의 충돌에 대비하여 복구하기 위한 목적입니다.

ØMQ는 어느 쪽이 바인딩되고 어느 쪽이 연결되는지에 대해 중립을 유지하려 하지만 차이점이 있습니다. 나중에 더 자세히 살펴보겠습니다. 일반적으로 "서버"는 다소 고정된 단말에 바인딩하고 "클라이언트"는 동적 요소인 단말에 연결하는 것으로 생각합니다. 동적/정적 사용 모델에 따라 응용프로그램을 설계하십시오. 그러면 "그냥 작동"할 가능성이 높습니다.

소켓은 유형이 있고 소켓 유형에 따라 정의되는 의미는 네트워크 상에서 내부/외부 메시지 라우팅 정책이나 대기열 저장 등입니다. 특정 소켓 유형은 함께 사용할 수 없으며 예를 들면 PUB 소켓과 SUB 소켓.

ØMQ는 다양한 형태로 소켓을 연결할 수 있으며 메시징 대기열 시스템으로써 기본 역량을 제공합니다. 근본적으로 ØMQ는 다양한 구성 요소(소켓, 연결, 프록시 등)들을 필요에 따라 끼워 맞추어 필요한 네트워크 아키텍처를 정의할 수 있게 합니다.

0.20.2 메시지 송/수신

메시지 송/수신을 위하여 `zmq_msg_send()`와 `zmq_msg_recv()` 함수를 사용할 수 있습니다. 전통적인 방식의 명칭이지만 ØMQ I/O 모델은 기존 TCP 모델과는 확연히 차이가 있습니다.

Figure 9 - 1:1 TCP 소켓(TCP sockets are 1 to 1)

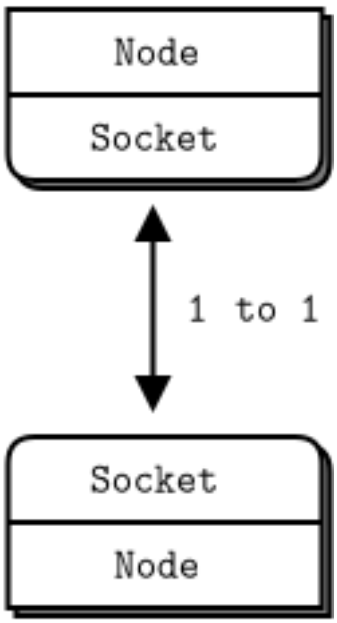


그림 10: 1:1 TCP 소켓

TCP 소켓과 ØMQ 소켓의 주요 차이는 데이터 처리 방식입니다.

- ØMQ 소켓은 메시지들을 전달합니다. 마치 UDP처럼. 다소 TCP의 바이트 스트림처럼 동작한다. ØMQ 메시지는 길이가 지정된 바이너리 데이터로 성능 최적화를 위해 다소 까다롭게 설계되었습니다.
- ØMQ 소켓의 I/O는 백그라운드 스레드로 동작합니다. 응용프로그램이 무엇을 바쁘게 처리하더라도 I/O 스래드는 메시지들을 입력 대기열로부터 수신하고, 출력 대기열로에서 송신하게 합니다.
- ØMQ 소켓들은 소켓 유형에 따라 1:N 라우팅이 내장되어 있다.

zmq_send() 함수는 실제 메시지를 소켓 연결에서 전송하지 않고 메시지를 대기열에 쌓아 I/O 스레드가 비동기로 전송할 수 있게 합니다. 이러한 동작은 일부 예외 상황을 제외하고는 차단되지 않습니다. 그래서 메시지를 zmq_send()로 전송할 때 반환값이 필요 없습니다.

0.20.3 유니케스트 전송방식

ØMQ는 일련의 유니케스트 전송방식(inproc, ipc, tcp)과 멀티캐스트 통신방식(epgm, pgm)을 제공합니다. 멀티캐스트는 진보된 기술로 나중에 다루지만 전개 비율에 따라 1:N 유니케스트가 불가하다는 것을 모른다면 시작조차 하지 마시기 바랍니다.

대부분의 경우 TCP 사용하며 "비연결성 TCP 전송계층"라고 하며, 대부분의 경우 충분히 빠르고 유연하고 간편합니다. 우리가 "비연결성"라고 부르는 것은 ØMQ의 TCP 전송방식은 단말이 연결을 하기전에 존재할 필요가 없기 때문입니다. 클라이언트들과 서버들은 언제든지 연결하거나 바인딩할 수 있으며, 나가고 들어오는 것이 가능하며 응용프로그램에 투명하게 유지됩니다.

프로세스 간 ipc 전송방식도 TCP처럼 "비연결성"이지만 한 가지 제약 사항은 윈도우 환경에서는 아직 동작하지 않습니다. 편의상 단말의 명칭들을 ".ipc" 확장명으로 사용하여 다른 파일명과 잠재적인 충돌을 피하도록 하겠습니다. 유닉스 시스템에서 ipc 단말을 사용하려 한다면 적절한 접근 권한을 생성이 필요하며 그렇지 않다면 다른 사용자 식별자(ID)하에 구동되는 프로세스 간에 공유는 허용되지 않을 것입니다. 그리고 모든 프로세스들은 파일들(예를 들면 동일 디렉터리에서 존재하는 파일들)에 대하서 접근할 수 있어야 합니다.

스레드 간 inproc 전송방식은 "연결된 신호 전송계층"이며 tcp 혹은 ipc 보다 훨씬 빠르지만 tcp나 ipc와 비교했을 때 특정 제약을 가지고 있습니다 : 서버는 반드시 클라이언트가 연결을 요청하기 전에 바인딩이 되어야 한다. 이것은 미래의 ØMQ 버전에서 개선되겠지만 현재는 어떻게 inproc 소켓을 사용해야 하는지를 정의합니다. 우리는 부모 스레드에서 소켓을 생성하고 바인딩하고 자식 스레드를 시작하여 소켓을 생성하고 연결하게 합니다.

0.20.4 ØMQ는 중립적인 전송수단이 아니다.

ØMQ을 처음 접하는 사람들의 공통적인 질문(이것은 나 스스로에게도 한 것임)은 "ØMQ로 XYZ 서버를 개발하는 방법은 무엇이지?"입니다. 예를 들면 "ØMQ로 HTTP 서버를 개발하는

방법은 무엇이지?"와 같으며 이는 HTTP 요청과 응답을 일반 소켓을 사용하는 것처럼, ØMQ 소켓을 통하여 더 빠르고 좋게 동일한 작업을 수행할 수 있어야 합니다.

예전에는 ØMQ는 "그렇게 동작하지 않는다"라며 ØMQ는 중립적인 전송수단(Neutral carrier)이 아니다고 했습니다 : 이것은 전송방식 통신규약(OSI 7 계층)에 프레이임을 부가하기 때문입니다. ØMQ의 프레이임은 기존의 통신규약들과 호환되지 않습니다. 예를 들어 HTTP 요청과 ØMQ 요청을 비교하면 둘 다 TCP/IP상에서 동작하지만 자신의 프레이임을 사용하는 경향이 있습니다.

- [옮긴이] 프레이임(framing)은 OSI 7 계층에서 데이터링크에서 정의하고 있으며 메시지 데이터 구성 방식으로 다양한 통신규약들에 의해 교환되고 운반되는 데이터 단위입니다

그림 10 - 네트워크상 HTTP

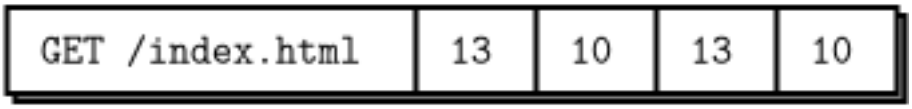


그림 11: 네트워크상 HTTP

HTTP 요청은 CR(0x0D)-LF(0x0A)을 프레이임 구분자로 사용하지만 ØMQ는 길이가 지정된 프레임을 사용합니다. 그래서 ØMQ를 사용하여 요청-응답 소켓 패턴으로 HTTP와 유사한 통신규약을 작성할 수 있지만 HTTP에서는 할 수 없습니다.

그림 11 - 네트워크상 ØMQ

그림 12: 네트워크상 ØMQ

v3.3 버전 이후로 ØMQ는 "ZMQ_ROUTER_RAW"라는 새로운 소켓 옵션을 가지고 ØMQ 프레이밍이 없이도 데이터를 읽고 쓸 수 있게 되었습니다. 즉 적절한 HTTP 요청과

응답을 읽고 쓸수 있게 되었다. “하딥 싱(Hardeep Singh)”은 자신의 ØMQ 응용프로그램에서 텔넷 서버에 연결할 수 있도록 이러한 변경에 기여했습니다. 글을 쓰는 시점에는 이것은 다소 실험적이지만 ØMQ가 어떻게 새로운 문제를 해결하기 위해 계속 발전하고 있는지 보여줍니다. 아마도 다음 패치에서는 사용 가능할 것으로 예상됩니다.

- [옮긴이] 현재(2020년 8월) ØMQ 4.3.2의 `zmq_setsockopt()`에서도 `ZMQ_ROUTER_RAW` 사용 가능합니다.

0.20.5 I/O 스레드들

이전에 ØMQ는 백그라운드 스레드를 통하여 I/O을 수행한다고 하였으며, 하나의 I/O 스레드(모든 소켓들에 대하여)에서 메시지 처리가 가능합니다(일부 극단적으로 대량의 I/O를 처리하는 응용프로그램의 경우 I/O 스레드 개수를 조정 가능함). 새로운 컨텍스트를 만들 때, 하나의 I/O 스레드가 시작되며, 일반적으로 하나의 I/O 스레드는 초당 1 기가바이트의 데이트를 입/출력할 수 있습니다. I/O 스레드의 개수를 늘리기 위해서는 소켓을 생성하기 전에 `zmq_ctx_set()` 함수를 사용할 수 있습니다.

```
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 소켓에서 한 번에 수십 개 혹은 수천 개의 연결을 처리할 수 있으며 이것이 ØMQ 기반 응용프로그램 작성에 근본적인 영향을 줍니다. 기존의 네트워크 응용프로그램에서는 원격 연결당 하나의 프로세스 혹은 하나의 스레드가 필요하였으며, 해당 프로세스 혹은 스레드에서 하나의 소켓으로 처리하였습니다. ØMQ를 사용하면 전체 구조를 단일 프로세스로 축소하고 필요에 따라 확장할 수 있습니다.

- [옮긴이] ØMQ 응용프로그램 작성 시 클라이언트/서버를 각각의 스레드로 작성하여 inproc를 통하여 테스트를 수행하고, 정상적인 경우 각 스레드는 프로세스로 분리하여 IPC나 TCP로 연결할 수 있습니다.

ØMQ를 스레드 간 통신(inproc)에 사용할 경우(예를 들어 멀티스레드 응용프로그램에서 외부 소켓 I/O가 없을 경우)에 한하여 I/O 스레드의 개수를 0으로 설정할 수 있습니다. 흥미롭지만 중요한 최적화는 아닙니다.

0.21 메시징 패턴

ØMQ 소켓 API를 감싸고 있는 갈색 포장지를 걷어내면 메시징 패턴의 세계가 놓여 있습니다. 만약 기업 메시징 시스템에 대한 배경이 있거나 UDP을 아신다면 다소 막연하지만 익숙할 것입니다. 그러나 ØMQ를 새로 접하시는 분에게는 놀라울 것입니다. 우리는 TCP 패러디엄을 사용하여 다른 노드와 1:1 매핑을 하였습니다.

ØMQ가 하는 일을 간단히 다음과 같습니다. * 데이터의 블록(메시지, 대용량 데이터(blobs))을 노드 간에 빠르고 효율적으로 전달합니다. * 노드를 스레드(inproc), 프로세스(ipc) 혹은 머신(tcp, pgm)으로 메핑 할 수 있습니다. * ØMQ는 응용프로그램에서 전송방식(in-process, inter-process, TCP, multicast)에 관계없이 단일 소켓 API를 제공합니다. * 노드들 간 연결이 끊기면 자동으로 재연결하게 합니다. * 송/수신 측에서 메시지들을 대기열에 관리하며, 이러한 대기열을 신중하게 관리하여 프로세스에 메모리가 부족하지 않도록 하며, 필요하다면 디스크에 저장합니다. * 소켓 오류를 처리합니다. * 모든 I/O 처리를 백그라운드 스레드가 처리합니다(기본 1개이며, 수량 조정 가능). * ØMQ는 노드 간의 통신에 잠금 없는 기술을 사용하여 잠금, 대기, 세마포어, 교착이 발생하지 않습니다.

- [옮긴이] 대기열에 설정한 HMW(최고수위 표시)를 초과할 경우 디스크에 저장하기 위해서는 `zmq_setsockopt()`에서 `ZMQ_SWAP`설정을 통해 수행합니다.

패턴이란 불리는 정확한 레시피에 따라 메시지들을 경유하고 대기열에 쌓을 수 있으며, 이러한 패턴들을 통하여 ØMQ에 지능을 제공합니다. 패턴들은 우리가 힘겹게 얻은 경험을 통하여 데이터와 작업을 분산하기 위한 최선의 방법을 제공합니다. ØMQ의 패턴은 하드코딩되어 있지만 향후 버전에서는 사용자 정의 가능 패턴이 허용될 수 있습니다.

ØMQ 패턴들은 소켓과 일치하는 유형의 쌍으로 구현됩니다. ØMQ 패턴을 이해하기 위해서 소켓 유형의 이해가 필요하며 어떻게 함께 동작(ØMQ 패턴, 소켓 유형)하는지 알아야 합니다.

내장된 핵심 ØMQ 패턴들은 다음과 같습니다.

- 요청-응답(REQ-REP)
- 일련의 클라이언트들에게 서비스들 연결을 하게 하며, 이것은 RPC(remote procedure call)와 작업 분배 패턴입니다.
- 발행-구독(PUSH-PULL)
- 일련의 발행자들과 구독자들 연결을 하게 하며, 데이터 분배 패턴입니다.
- 파이프라인(Pipeline)
- 노드들을 팬아웃/팬인(fan-out/fan-in) 패턴으로 연결하게 하며 여러 개의 절차들과 루프를 가질 수 있습니다. 이것은 병렬 작업 분배와 수집 패턴입니다.
- 독점적인 쌍(PAIR-PAIR)
- 2개의 소켓들을 상호 독점적으로 연결합니다. 이것은 하나의 프로세스에서 2개의 스레드 간에 연결하는 패턴이며 정상적인 소켓의 쌍과 혼동이 없어야 합니다.

1장 기본에서 3개의 패턴을 보았으며 "독점적인 쌍"은 다음 장에서 보도록 하겠습니다. `zmq_socket()` 매뉴얼에 패턴에 대한 설명이 되어 있으며 이해될 때까지 숙지할 필요가 있습니다. 다음은 연결-바인드 쌍에 대하여 유효한 소켓 조합입니다(어느쪽이든 바인딩을 수행할 수 있습니다.).

- PUB와 SUB
- REQ와 REP
- REQ와 ROUTER(주의 필요, REQ는 추가로 1개의 널(NULL(0)) 프레임을 넣는다.)
- DEALER와 REP(주의 필요, REP는 1개의 널(NULL(0)) 프레임으로 가정한다.)
- DEALER와 ROUTER
- DEALER와 DEALER
- ROUTER와 ROUTER
- PUSH와 PULL
- PAIR와 PAIR

XPUB와 XSUB 소켓도 있으며, 다음장에서 다루도록 하겠습니다(PUB와 SUB의 원시(raw) 버전과 유사). 다른 소켓 조합들은 신뢰할 수 없는 결과를 낳을 수 있으며, 미래의 ØMQ

에서는 오류를 생성하게 할 예정입니다. 물론 코드상에서 다른 소켓 유형을 연결할 수 있습니다 (즉, 한 소켓 유형에서 읽고 다른 소켓 유형에 쓰기).

0.21.1 고수준의 메시지 패턴들

ØMQ 에는 내장된 4개의 핵심 패턴이 있으며, 그들은 ØMQ API의 일부이며 C++ 라이브러리로 구현되어 있으며 모든 용도에 적절하게 동작함을 보장합니다.

- [옮긴이] 핵심 패턴들은 ØMQ RFC사이트(https://rfc.zeromq.org)에서 정의되어 있습니다.
- MDP(Majordome Protocol), TSP(Titanic Service Protocol), FLP(Freelance Protocol), CHP(Clustered Hashmap Protocol) 등이 있습니다.

위의 제목에서 ØMQ의 핵심 라이브러리가 아니고 ØMQ 패키지에 포함되지도 않았음에도 고수준의 메시지 패턴들을 넣은 것은, ØMQ 커뮤니티에서 자신만의 영역으로 존재하기 때문이다. 예를 들면 집사(Majordome) 패턴(4장-신뢰성 있는 요청-응답 패턴에서 소개)은 ØMQ 조직에서 깃허브 프로젝트(https://github.com/ØMQ/majordomo)로 자리 잡았습니다.

이 책에서 우리가 제공하고자 하는 일련의 고수준 패턴들은 작을 수도 있고(정상적으로 메시지 처리하는 방법) 거대(신뢰할 수 있는 PUB-SUB 아키텍처를 만드는 방법)할 수도 있습니다.

0.21.2 메시지와 작업하기

libzmq 핵심 라이브러리는 사실 메시지를 송/수신하는 2개의 API를 가지고 있습니다. `zmq_send()`와 `zmq_recv()` 함수이며, 이미 한줄짜리 코드로 보았습니다. `zmq_recv()`는 임의의 메시지 크기를 가지는 데이터를 처리하는 데는 좋지 않습니다. `zmq_recv()`는 응용프로그램의 코드 내에서 정의한 버퍼의 크기를 넘을 경우 크기 이상 메시지 데이터는 버리게 됩니다. 그래서 `zmq_msg_t` 구조체를 다룰 수 있도록 추가적인 API로 작업해야 합니다.

- 메시지 초기화 : `zmq_msg_init()`, `zmq_msg_init_size()`, `zmq_msg_init_data()`
- 메시지 송/수신 : `zmq_msg_send()`, `zmq_msg_recv()`

- 메시지 해제 : zmq_msg_close()
- 메시지 내용 접근 : zmq_msg_data(), zmq_msg_size(), zmq_msg_more()
- 메시지 속성 작업 : zmq_msg_get() , zmq_msg_set()
- 메시지 조작 : zmq_msg_copy() , zmq_msg_move()

네트워크상에서, ØMQ 메시지들은 메모리상 0에서 임의의 크기의 덩어리(blob) 입니다. 응용프로그램에서 해당 메시지를 처리하기 위해서는 통신규약 버퍼, msgpack, JSON 등으로 직렬화를 수행해야 합니다. 이식 가능한 데이터 표현을 현명하게 선택하기 위하여 장/단점을 이해해야 결정할 수 있습니다.

메모리상에서 ØMQ 메시지는 zmq_msg_t 구조체(개발 언어에 따라 클래스)로 존재하며 C 언어에서 ØMQ 메시지들을 사용하는 기본 규칙이 있습니다.

- zmq_msg_t 객체(데이터의 블록 아님)를 생성하고 전달할 수 있습니다.
- 메시지를 읽기 위해서 zmq_msg_init()을 호출하여 하나의 빈 메시지를 만들고 zmq_msg_recv()을 통하여 수신합니다.
- 새로운 데이터(구조체)에 메시지를 쓰기 위해서 zmq_msg_init_size()을 호출하여 메시지를 생성하고 동시에 구조체 크기의 데이터 블록을 할당합니다. 그리고 memcpy()을 사용하여 데이터를 채워 넣고 zmq_msg_send()를 통하여 메시지를 송신합니다.
- 메시지를 해제(파괴가 아님)를 위하여, zmq_msg_close()을 호출하여 참조를 없애고 결국 ØMQ가 메시지를 삭제하게 합니다.
- 메시지 내용을 접근하기 위하여 zmq_msg_data()을 사용하며, 메시지 내용에 얼마나 많은 데이터가 있는지 알기 위하여 zmq_msg_size()을 호출합니다.
- zmq_msg_move(), zmq_msg_copy(), zmq_msg_init_data()에 대하여 정확하게 사용법을 확인하여 사용해야 합니다.
- 메시지가 zmq_msg_send()을 통해 전송된 이후, ØMQ는 메시지를 청소합니다. 예를 들어 크기를 0으로 설정하여 동일 메시지를 두 번 보낼 수 없으며 데이터 전송 이후 메시지에 접근할 수 없습니다.
- 이러한 규칙은 zmq_send()와 zmq_recv()을 사용한 경우 적용되지 않으며, 이 함수들은 바이트의 배열을 전달하지 메시지 구조체를 전달하는 것은 아니기 때문입니다.

만약 동일 메시지를 한번 이상 보내고 싶을 경우, 추가적인 메시지를 만들어 zmq_msg_init()

으로 초기화하고 `zmq_msg_copy()`을 사용하고 첫 번째 메시지를 복사하여 생성할 수 있습니다. 이것은 데이터를 복사하는 것이 아니라 참조를 복사합니다. 그런 다음 메시지를 두 번(혹은 이상으로 더 많은 복사를 작성하는 경우) 보내며, 마지막 사본을 보내거나 닫을 때 메시지가 삭제됩니다.

ØMQ는 멀티파트 메시지들을 지원하며 일련의 데이터 프레임들을 하나의 네트워크상 메시지로 송/수신하게 하며 실제 응용프로그램에서 광범위하게 사용됩니다. "3장-개선된 요청-응답 패턴"에서 살펴보도록 하겠습니다.

프레임들(ØMQ 참조 매뉴얼에서는 메시지 파트들이고도 함)은 기본적인 ØMQ 메시지 형태이며 하나의 프레임은 일정 길이의 데이터의 블록이며 길이는 0 이상입니다. TCP 프로그래밍을 하셨다면 "네트워크 소켓에서 얼마나 많은 데이터를 읽어야 하는지?"란 물음에 대하여 ØMQ 프레임이 유용한 해답임을 알 수 있습니다.

ZMTP라고 불리는 와이어 레벨(wire-level) 통신규약이 존재하며 ØMQ가 TCP 연결상에서 프레임들을 어떻게 읽고 쓰는지를 정의합니다. 관심 있으면 한번 참조하기 바라며 사양서는 단순합니다.

원래는 하나의 ØMQ 메시지는 UDP처럼 하나의 프레임이었지만 나중에 멀티파트 메시지들로 확장하였으며, 단순히 일련의 프레임들이 있으며 "more" 비트을 1로 설정하고, 하나의 프레임 있으면 "more" 비트를 0으로 설정합니다. ØMQ API에서 "more" 플레그를 통해 메시지를 쓸 수 있으며, 메시지를 읽을 때 "more"을 체크하여 추가적인 동작을 할 수 있습니다.

저수준의 ØMQ API와 참조 매뉴얼에서는 메시지들과 프레임들 간에 모호함이 있어 용어를 정의하겠습니다.

- 하나의 메시지는 하나 이상의 파트(part)들이 될 수 있습니다.
- 여기서 파트는 프레임입니다.
- 각 파트는 `zmq_msg_t` 객체입니다.
- 저수준 ØMQ API를 통하여 각 파트를 개별적으로 송/수신할 수 있습니다.
- 고수준 ØMQ API에서는 전체 멀티파트 메시지를 전송하는 래퍼를 제공합니다.

메시지에 대하여 알아두면 좋을 것들이 있습니다.

- 길이가 0인 메시지를 전송할 수 있습니다. 예를 들어 하나의 스레드에서 다른 스레드로 신호를 전송하기 위한 목적입니다.

- ØMQ는 메시지의 모든 파트들(하나 이상)의 전송을 보증하거나, 하지 않을 수 있습니다.
- ØMQ는 메시지(1개 혹은 멀티파트)를 바로 전송하지 않고, 일정 시간 후에 보냅니다. 그래서 하나의 멀티파트 메시지는 설정한 메모리의 크기에 맞아야 합니다.
- 하나의 메시지(1개 혹은 멀티파트)는 반드시 메모리 크기에 맞아야 하며 파일이나 임의의 크기를 전송하려면 하나의 파트의 메시지로 쪼개어서 각 조각을 전송해야 합니다. 멀티파트 데이터를 사용하여 메모리 사용을 줄일 수는 없습니다.
- 메시지 수신이 끝나면 `zmq_msg_close()`을 반드시 호출하여야 하며, 일부 개발 언어에서는 변수의 사용 범위를 벗어나도 자동으로 삭제하지 않습니다. 메시지를 전송할 때는 `zmq_msg_close()`을 호출하지 마시기 바랍니다.
- [옮긴이] zhelpers.h에 정의된 s_dump() 함수를 통하여 멀티파트 메시지(multipart message) 출력이 가능하게 합니다.

```
s_dump (void *socket)
{
    int rc;

    zmq_msg_t message;
    rc = zmq_msg_init (&message);
    assert (rc == 0);

    puts ("----------------------------------------");
    //  Process all parts of the message
    do {
        int size = zmq_msg_recv (&message, socket, 0);
        assert (size >= 0);

        //  Dump the message as text or binary
        char *data = (char*)zmq_msg_data (&message);
```

```
        assert (data != 0);
        int is_text = 1;
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++) {
            if ((unsigned char) data [char_nbr] < 32
                || (unsigned char) data [char_nbr] > 126) {
                is_text = 0;
            }
        }

        printf ("[%03d] ", size);
        for (char_nbr = 0; char_nbr < size; char_nbr++) {
            if (is_text) {
                printf ("%c", data [char_nbr]);
            } else {
                printf ("%02X", (unsigned char) data [char_nbr]);
            }
        }
        printf ("\n");
    } while (zmq_msg_more (&message));

    rc = zmq_msg_close (&message);
    assert (rc == 0);
}
```

아직 `zmq_msg_init_data()`을 언급하지 않았지만 이것은 "zero-copy"며 문제를 일으킬 수 있습니다. 마이크로초 단축에 대해 걱정하기보다는 ØMQ에 대해 알아야 할 중요한 사항들이 많이 있습니다.

많은 API는 작업하기 피곤하며 각종 함수들을 성능을 위해 최적화하는 것도 단순하지 않습니다. ØMQ 매뉴얼에 익숙하기 전에 API를 잘못 사용하는 경우 오류가 발생할 수 있습

니다. 특정 개발 언어에 대한 바인딩을 통하여 API를 쉽게 사용하는 것도 하나의 방법입니다.

0.21.3 다중 소켓 다루기

지금까지의 예제들은 아래 항목의 루프를 수행하였습니다.

1. 소켓의 메시지를 기다림
2. 메시지를 처리
3. 반복

만약 다중 단말로부터 동시에 메시지를 받아 처리해야 한다면 하나의 소켓에 모든 단말들을 연결하여 ØMQ가 팬아웃 형태로 처리하는 것이 간단한 방법이며, 원격 단말들에도 동일한 패턴을 적용할 수 있습니다. 하지만 패턴이 다른 경우 즉 PULL 소켓을 PUB 단말에 연결하는 것은 잘못된 방법이다.

- [옮긴이] 가능한 패턴 : PUB/SUB, REQ/REP, REQ/ROUTER, DEALER/REP, DEALER/ROUTER, DEALER/DEALER, ROUTER/ROUTER, PUSH/PULL, PAIR/PAIR

실제로 한 번에 여러 소켓에서 읽으려면 `zmq_poll()`을 사용해야 합니다. 더 좋은 방법은 `zmq_poll()`을 프레임워크로 감싸서 이벤트 중심으로 반응하도록 변형하는 것이지만, 여기서 다루고 싶은 것보다 훨씬 더 많은 작업이 필요합니다.

- [옮긴이] zloop 리엑터(reactor)를 통하여 이벤트 중심으로 반응하도록 zmq_poll() 대체 가능합니다.

그럼 그렇게 하지 말라는 예로, 비차단(nonblocking) 소켓 읽기를 수행하는 방법을 보여드리겠습니다. 다음 예제는 2개의 소켓으로 메시지를 비차단 읽기로 수신하고 있으며 프로그램은 날씨 변경정보의 구독자와 병렬 작업의 작업자 역할을 수행하여 다소 혼란스럽습니다.

msreader.c: 다중 소켓 읽기

```
//  Reading from multiple sockets
//  This version uses a simple recv loop

#include "zhelpers.h"
```

```
int main (void)
{
    //  Connect to task ventilator
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  Connect to weather server
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 5);

    //  Process messages from both sockets
    //  We prioritize traffic from the task ventilator
    while (1) {
        char msg [256];
        while (1) {
            int size = zmq_recv (receiver, msg, 255, ZMQ_DONTWAIT);
            if (size != -1) {
                //  Process task
                printf("P");
            }
            else
                break;
        }
        while (1) {
            int size = zmq_recv (subscriber, msg, 255, ZMQ_DONTWAIT);
            if (size != -1) {
```

```
                //  Process weather update
                printf("S");
            }
            else
                break;
        }
        //  No activity, so sleep for 1 msec
        s_sleep (1);
    }
    zmq_close (receiver);
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

이런 접근에 대한 비용은 첫 번째 메시지에 대한 추가적인 지연이 발생합니다(메시지 처리하기에도 바쁜데 루푸의 마지막에 `sleep()`). 이러한 접근은 고속 처리가 필요한 응용프로그램에서 치명적인 지연을 발생합니다. `nanosleep()`를 사용할 경우 바쁘게 반복(busy-loop)되지 않는지 확인해야 합니다.

- [옮긴이] msreader.c에서 사용된 s_sleep()는 zhelpers.h에 정의되어 있음

```
//  Sleep for a number of milliseconds
static void
s_sleep (int msecs)
{
#if (defined (WIN32))
    Sleep (msecs);
#else
    struct timespec t;
    t.tv_sec  =  msecs / 1000;
```

```
    t.tv_nsec = (msecs % 1000) * 1000000;
    nanosleep (&t, NULL);
#endif
}
```

예제에서 2개의 루프에서 첫 번째 소켓(ZMQ_PULL)이 두 번째 소켓(ZMQ_SUB)보다 먼저 처리하게 하였습니다.

대안으로 이제 zmq_poll()을 사용하는 응용프로그램을 보도록 하겠습니다.

mspoller.c: 다중 소켓 폴러

```
//  Reading from multiple sockets
//  This version uses zmq_poll()

#include "zhelpers.h"

int main (void)
{
    //  Connect to task ventilator
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  Connect to weather server
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10000 ", 5);

    //  Process messages from both sockets
    while (1) {
        char msg [256];
```

```
        zmq_pollitem_t items [] = {
            { receiver,   0, ZMQ_POLLIN, 0 },
            { subscriber, 0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            int size = zmq_recv (receiver, msg, 255, 0);
            if (size != -1) {
                //  Process task
                printf("P");
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            int size = zmq_recv (subscriber, msg, 255, 0);
            if (size != -1) {
                //  Process weather update
                printf("S");
            }
        }
    }
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

- [옮긴이] “zmq_poll (items, 2, -1)”에서 3번째 인수가 “-1”일 경우, 이벤트가 발생할때까지 기다리게 됩니다.

`zmq_pollitem_t`는 4개의 맴버가 있으며 구조체입니다.(ØMQ 4.3.2).

```
typedef struct zmq_pollitem_t
{
    void *socket;
#if defined _WIN32
    SOCKET fd;
#else
    int fd;
#endif
    short events;
    short revents;
} zmq_pollitem_t;
```

- [옮긴이] msreader와 mspoller 테스트를 위해서는 taskvent(호흡기 서버)와 wuserver(기상 정보 변경 서버) 수행이 필요함

taskvent.c : 호흡기 서버 프로그램

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

#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

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

```
//  Socket to send start of batch message on
void *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...\n");

//  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
int task_nbr;
int total_msec = 0;     //  Total expected cost in msecs
for (task_nbr = 0; task_nbr < 100; task_nbr++) {
    int workload;
    //  Random workload from 1 to 100msecs
    workload = randof (100) + 1;
    total_msec += workload;
    char string [10];
    sprintf (string, "%d", workload);
    s_send (sender, string);
}
printf ("Total expected cost: %d msec\n", total_msec);
```

```
    zmq_close (sink);
    zmq_close (sender);
    zmq_ctx_destroy (context);
    return 0;
}
```

wuserver.c : 기상 정보 변경 서버

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

#include "zhelpers.h"

int main (void)
{
    //  Prepare our context and publisher
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    int rc = zmq_bind (publisher, "tcp://*:5556");
    assert (rc == 0);

    //  Initialize random number generator
    srandom ((unsigned) time (NULL));
    while (1) {
        //  Get values that will fool the boss
        int zipcode, temperature, relhumidity;
        zipcode     = randof (99999);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;
```

```
        //  Send message 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;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc mspoller.c libzmq.lib
 cl -EHsc msreader.c libzmq.lib
 cl -EHsc taskvent.c libzmq.lib
 cl -EHsc wuserver.c libzmq.lib

 ./taskvent
Press Enter when the workers are ready:
Sending tasks to workers...
Total expected cost: 4856 msec

 ./wuserver

 ./mspoller
SSSSSSSSSSSSSSSPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PSPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSS
```

```
 ./msreader
SSSSSSSSSSSSSSSSSSSPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSS
```

0.21.4 멀티파트 메시지들

ØMQ는 여러 개의 프레임들로 하나의 메시지를 구성하게 하며 멀티파트 메시지를 제공합니다. 현실에서는 응용프로그램은 멀티파트 메시지를 많이 사용하며 IP 주소 정보를 메시지에 내포하여 직열화 용도로 사용합니다. 우리는 이것을 응답 봉투에서 나중에 다루겠습니다.

이제부터 어떤 응용프로그램(프록시와 같은)에서나 멀티파트 메시지를 맹목적으로 안전하게 읽고 쓰는 방법을 설명하며, 프록시는 메시지를 별도의 검사 없이 전달하도록 합니다.

멀티파트 메시지 작업 시, 개별 파트는 zmq_msg 항목이며, 메시지를 5개의 파트들로 전송한다면 반드시 5개의 zmq_msg 항목들을 생성, 전송, 파괴를 해야 합니다. 이것을 미리 하거나 (zmq_msg를 배열이나 구조체에 저장하여) 혹은 전송 이후 차례로 할 수 있습니다.

zmq_msg_send()를 통하여 멀티파트 메시지에 있는 프레임을 전송하는 방법입니다(각각의 프레임은 메시지 객체로 수신합니다).

```
zmq_msg_send (&message, socket, ZMQ_SNDMORE);
...
zmq_msg_send (&message, socket, ZMQ_SNDMORE);
...
zmq_msg_send (&message, socket, 0);
```

zmq_msg_recv()을 통하여 메시지에 있는 모든 파트들을 수신하고 처리하는 예제입니다. 파트는 1개이거나 멀티파트로 가능합니다.

```
while (1) {
    zmq_msg_t message;
    zmq_msg_init (&message);
```

```
    zmq_msg_recv (&message, socket, 0);
    // Process the message frame
    ...
    zmq_msg_close (&message);
    if (!zmq_msg_more (&message))
        break; // Last message frame
}
```

멀티파트 메시지들에 대하여 알아야 될 몇 가지 사항은 다음과 같습니다.

- 멀티파트 메시지를 전송할 때, 첫 번째 파트(와 후속되는 파트들)는 마지막 파트를 전송하기 전에 네트워크상에서 실제 전송되어야 합니다.
- zmq_poll()을 사용한다면, 메시지의 첫 번째 파트가 수신될 때, 나머지 모든 파트들도 도착되어야 합니다.
- 메시지의 모든 파트들을 수신하거나, 하나도 받지 않을 수 있습니다.
- 메시지의 각각의 파트는 개별적인 zmq_msg 항목입니다.
- zmq_msg_more()을 통한 more 속성을 점검 여부에 관계없이 메시지의 모든 파트들을 수신할 수 있습니다.
- 데이터 전송 시, ØMQ는 메시지 대기열에 마지막 프레임이 올 때까지 쌓이게 하여, 마지막 프레임이 도착하면 대기열의 모든 프레임들을 전송합니다.
- 소켓을 닫는 경우를 제외하고는 메시지 전송 시 프레임들을 부분적으로 취소할 방법은 없습니다.
- [옮긴이] mbroker.c : zmq_poll()을 통한 송/수신 브로커에서 zmq_msg_t을 통해 메시지를 정의하고 멀티파트 메시지를 처리합니다.

```
//  Simple request-reply broker

#include "zhelpers.h"

int main (void)
{
    //  Prepare our context and sockets
    void *context = zmq_ctx_new ();
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (frontend, "tcp://*:5559");
    zmq_bind (backend,  "tcp://*:5560");

    //  Initialize poll set
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    //  Switch messages between sockets
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            while (1) {
                //  Process all parts of the message
                zmq_msg_init (&message);
                zmq_msg_recv (&message, frontend, 0);
                int more = zmq_msg_more (&message);
                zmq_msg_send (&message, backend, more? ZMQ_SNDMORE: 0);
```

```
            zmq_msg_close (&message);
            if (!more)
                break;      //  Last message part
        }
    }
    if (items [1].revents & ZMQ_POLLIN) {
        while (1) {
            //  Process all parts of the message
            zmq_msg_init (&message);
            zmq_msg_recv (&message, backend, 0);
            int more = zmq_msg_more (&message);
            zmq_msg_send (&message, frontend, more? ZMQ_SNDMORE: 0);
            zmq_msg_close (&message);
            if (!more)
                break;      //  Last message part
        }
    }
}
//  We never get here, but clean up anyhow
zmq_close (frontend);
zmq_close (backend);
zmq_ctx_destroy (context);
return 0;
}
```

0.21.5 중개자와 프록시

ØMQ는 지능의 탈중앙화를 목표로 만들었지만 네트워크상에서 중간이 텅 비었다는 것은 아니며, ØMQ를 통해 메시지 인식 가능한 인프라구조를 만들었습니다. ØMQ의 배관은 조그만 파이프에서 완전한 서비스 지향 브로커까지 확장될 수 있습니다. 메시지 업계에서는

이것을 중개자로 부르며 양쪽의 중앙에서 중개하는 역할을 수행하며, ØMQ에서는 문맥에 따라 프록시, 대기열, 포워더, 디바이스, 브로커라 부르겠습니다.

이러한 패턴은 현실세계에서 일반적이지만 왜 우리의 사회와 경제에서 실제 아무 기능도 없는 중개자로 채워져 있는 것은 거대한 네트워크 상에서 복잡성을 줄이고 비용을 낮추기 위함입니다. 현실세계에서 중계자는 보통 도매상, 판매 대리점, 관리자 등으로 불립니다.

0.21.6 동적 발견 문제

거대한 분산 아키텍처를 설계할 때 골 때리는 문제 중에 하나가 상대방을 찾는 것입니다. 어떻게 각 노드들이 상호 간을 인식할 수 있을지, 이것은 노드들이 네트워크상에서 들어오고 나가고를 반복하면서 특히 어렵게 되며 이러한 문제를 "동적 발견 문제"로 부르기로 하겠습니다.

- [옮긴이] 클라이언트에서 서비스(서버) 제공자를 찾기 위한 방법이 분산 시스템 구성에서 주요한 화두이며, 일부 상용 제품에서는 디렉터리 서비스(Directory Service)를 통해 문제를 해결합니다. 대표적인 제품으로 Apache LDAP이 있으며, NASA의 GMSEC의 경우 LDAP을 사용합니다.

동적 발견에 대한 몇 가지 해결 방법이 있지만, 가장 쉬운 해결 방법은 네트워크 아키텍처를 하드 코딩(혹은 구성 파일 사용)하여 사전에 정의하는 것이지만, 새로운 노드가 추가되면 네트워크 구성을 다시 하드 코딩(혹은 구성 파일 변경) 해야 합니다.

그림 12 - 소규모 Pub-Sub 네트워크

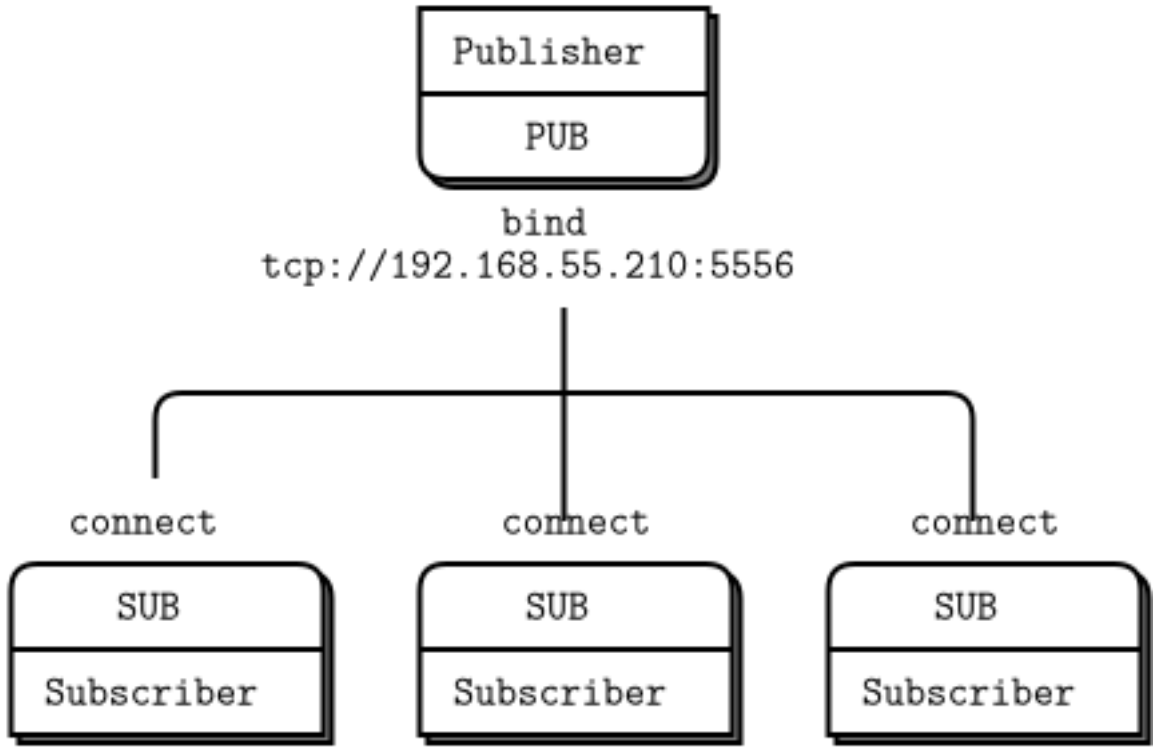


그림 13: Small-Scale Pub-Sub Network

실제 이런 해결법(하드 코딩(혹은 구성 파일 변경))은 현명하지 못하며 무너지기 쉬운 아키텍터로 이끌어 갑니다. 하나의 발행자에 대하여 100여 개의 구독자가 있다면 각 구독자가 연결하려는 발행자 단말 정보를 구성 정보화하여 발행자에 연결하려 할 것입니다. 발행자가가 고정되어 있고 구독자가 동적인 경우는 쉽지만, 새로운 발행자들을 추가할 경우 더 이상 이전 구성 정보를 사용할 수 없습니다. 만약 구성 정보 변경을 통하여 각 구독자를 발행자에 연결하게 한다면 동적 발견 회피 비용(구성 정보 변경 + 구독자 재기동)은 점점 더 늘어갈 것입니다.

그림 13 - 프록시를 통한 발행-구독 네트워크

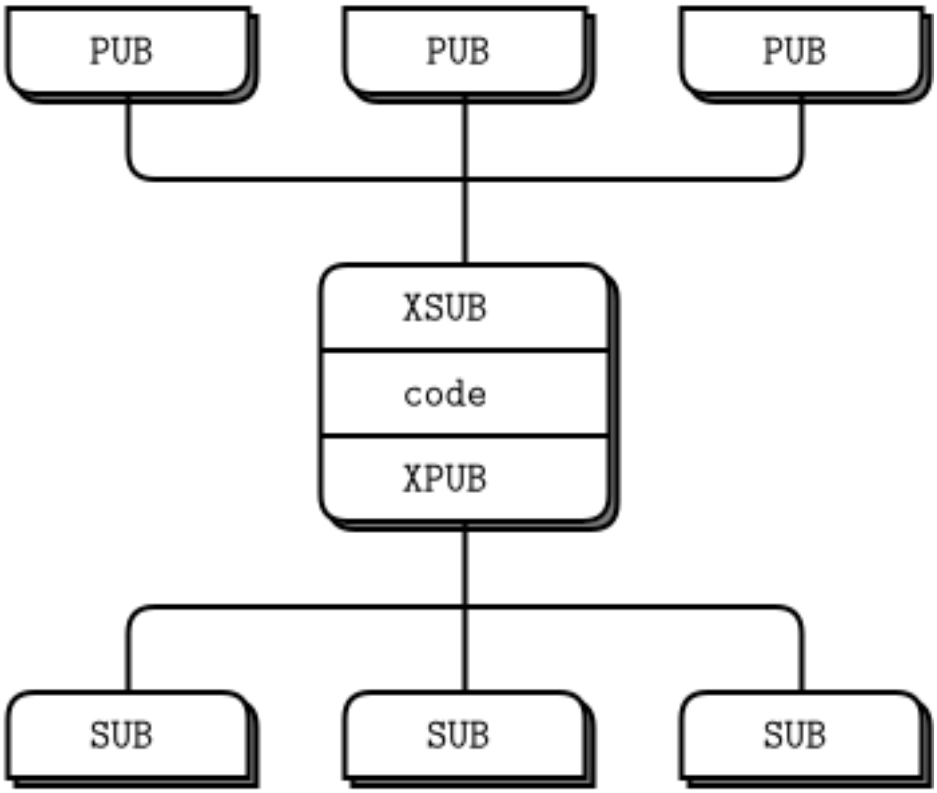


그림 14: Pub-Sub Network with a Proxy

이것에 대한 해결책은 중개자를 추가하는 것이며, 네트워크상 정적 지점에 다른 모든 노드들이 연결하게 합니다. 전통적인 메시징에서 이러한 역할은 메시지 브로커가 수행합니다. ØMQ는 이와 같은 메시지 브로커가 없지만 중개자들을 쉽게 만들 수 있습니다.

그럼 왜 ØMQ에서 메시지 브로커를 만들지 않았는지 궁금해질 것입니다. 초보자가 응용 프로그램을 작성할 때 메시지 브로커가 있다면 상당히 편리할 수 있습니다. 마치 네트워크상에서 성능을 고려하지 않은 스타 토폴로지가 편리한 것처럼 말입니다. 하지만 메시지 브로커는 탐욕스러운 놈입니다 : 특히 중앙 중개자 역할을 할 경우 그들은 너무 복잡하고 다양한 상태를 가지고, 결국 문제를 일으킵니다.

중개자를 단순히 상태가 없는 메시지 스위치들로 생각하면 좋을 것 같으며, 유사한 사례는 HTTP 프록시이며 클라이언트의 요청을 서버에 전달하고 서버의 응답을 클라이언트에 전달하는 역할 외에는 수행하지 않습니다. PUB-SUB 프록시를 추가하여 우리의 예제에서 동적

발견 문제를 해결할 수 있습니다. 프록시를 네트워크상 중앙에 두고 프록시를 발행자에게는 XSUB 소켓으로 구독자에게는 XPUB 소켓으로 오픈하고 각각(XSUB, XPUB)에 대하여 잘 알려진 IP 주소와 포트로 바인딩하게 합니다. 그러면 다른 프로세스들(발행자, 구독자)는 개별적으로 연결하는 것 대신에 프록시에 연결합니다. 이런 구성에서 추가적인 발행자나 구독자를 구성하는 것은 쉬운 일입니다.

그림 14 - 확장된 발행-구독

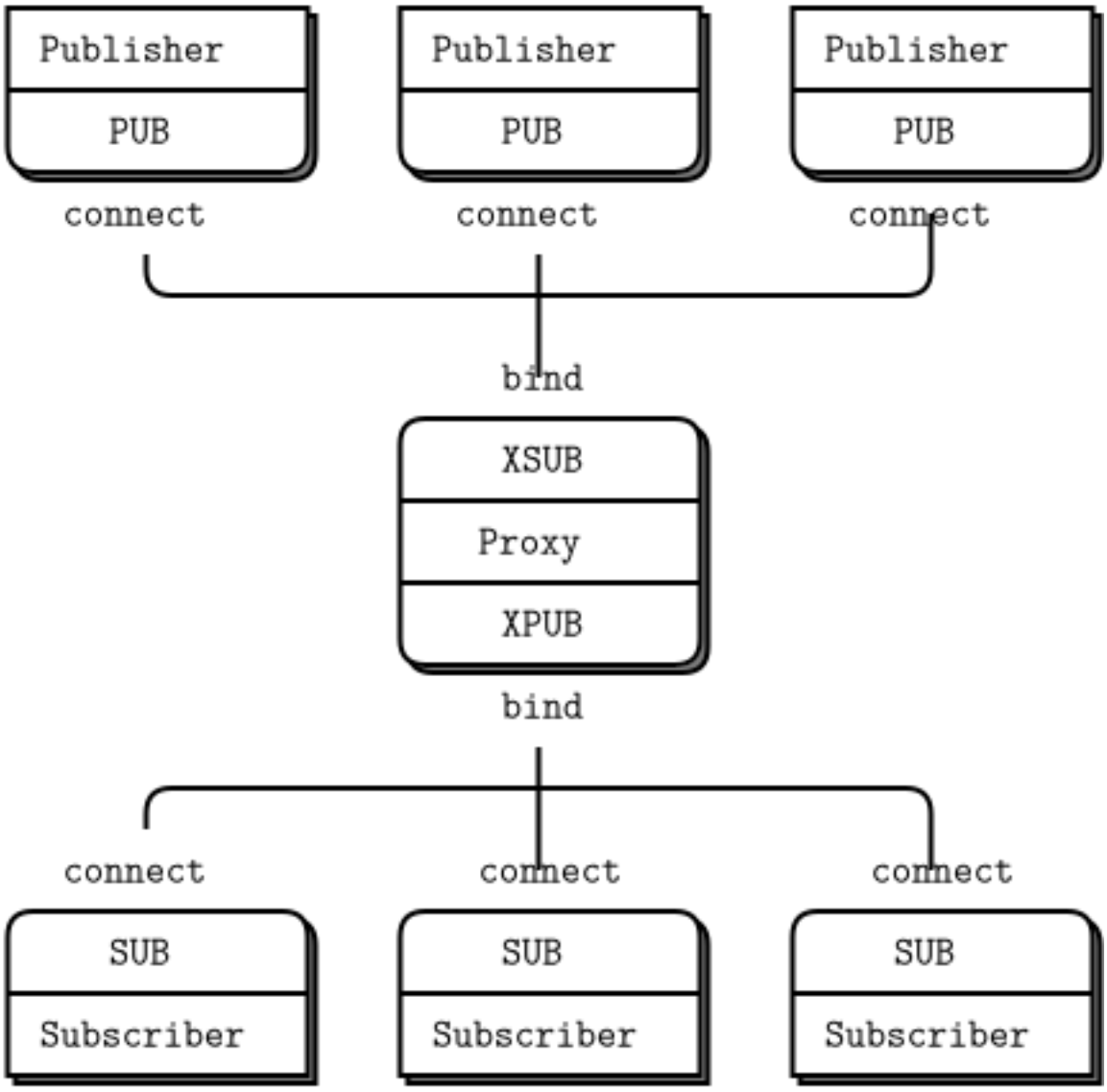


그림 15: Extended Pub-Sub

ØMQ가 구독 정보를 구독자들로부터 발행자들로 전달할 수 있는 XSUB과 XPUB 소켓에 대하여 살펴보도록 하겠습니다. XSUB와 XPUB은 특별한 메시지들에 대한 구독 정보를 공개하는 것 외에는 SUB와 PUB과 정확히 일치합니다. 프록시는 구독된 메시지들을 발행자들로부터 구독자들로 전달하며, XSUB 소켓으로 읽어서 XPUB 소켓에 쓰도록 합니다. 이것이 XSUB와 XPUB의 주요 사용법입니다.

- [옮긴이] XSUB는 eXtended subscriber, XPUB은 eXtended publisher이며, 다수의 발행자들과 구독자들을 연결하는 프록시(`zmq_proxy()`)를 통해 브로커 역할을 수행합니다.
- [옮긴이] XSUB와 XPUB을 테스트하기 위하여 기상 변경 정보를 전달하는 `wuserver`의 `bind()`를 `connect()`로 변경하여 테스트를 하겠습니다.

wuserver.c : 기상 변경 정보 발행

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

#include "zhelpers.h"

int main (void)
{
    //  Prepare our context and publisher
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    int rc = zmq_connect (publisher, "tcp://localhost:5556");
    assert (rc == 0);

    //  Initialize random number generator
    srandom ((unsigned) time (NULL));
    while (1) {
        //  Get values that will fool the boss
        int zipcode, temperature, relhumidity;
        zipcode     = randof (99999);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;
```

```
        //  Send message 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;
}
```

wuclient.c : 기상 변경 정보 구독

```
//  Weather update client
//  Connects SUB socket to tcp://localhost:5556
//  Collects weather updates and finds avg temp in zipcode

#include "zhelpers.h"

int main (int argc, char *argv [])
{
    //  Socket to talk to server
    printf ("Collecting updates from weather server...\n");
    void *context = zmq_ctx_new ();
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    int rc = zmq_connect (subscriber, "tcp://localhost:5557");
    assert (rc == 0);

    //  Subscribe to zipcode, default is NYC, 10001
    const char *filter = (argc > 1)? argv [1]: "10000";
    rc = zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE,
                         filter, strlen (filter));
```

```
    assert (rc == 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;
}
```

wuproxy.c : XSUB와 XPUB를 사용하여 중개자 역할 수행

```
//  Weather proxy device

#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();
```

```
    //  This is where the weather server sits
    void *frontend = zmq_socket (context, ZMQ_XSUB);
    zmq_bind (frontend, "tcp://*:5556");

    //  This is our public endpoint for subscribers
    void *backend = zmq_socket (context, ZMQ_XPUB);
    zmq_bind (backend, "tcp://*:5557");

    //  Run the proxy until the user interrupts us
    zmq_proxy (frontend, backend, NULL);

    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc wuserver.c libzmq.lib
 cl -EHsc wuclient.c libzmq.lib
 cl -EHsc wuproxy.c libzmq.lib

 ./wuserver

 ./wuproxy

 ./wuclient
Collecting updates from weather server...
```

```
Average temperature for zipcode '10000' was 18F
 ./wuclient
Collecting updates from weather server...
Average temperature for zipcode '10000' was 36F
```

0.21.7 공유 대기열(DEALER와 ROUTER 소켓)

Hello World 클라이언트/서버 응용프로그램에서, 하나의 클라이언트는 하나의 서비스와 통신할 수 있었습니다. 그러나 실제상황에서는 다수의 서비스와 다수의 클라이언트에 대하여 사용 가능해야 하며, 이를 통해 서비스의 성능을 향상할 수 있습니다(단 하나의 스레드가 아닌 다수의 스레드들, 프로세스들 및 노드들). 유일한 제약은 서비스는 무상태이며, 요청 중인 모든 상태는 데이터베이스와 같은 공유 스토리지에 저장되어야 합니다.

그림 15 - 요청 분배

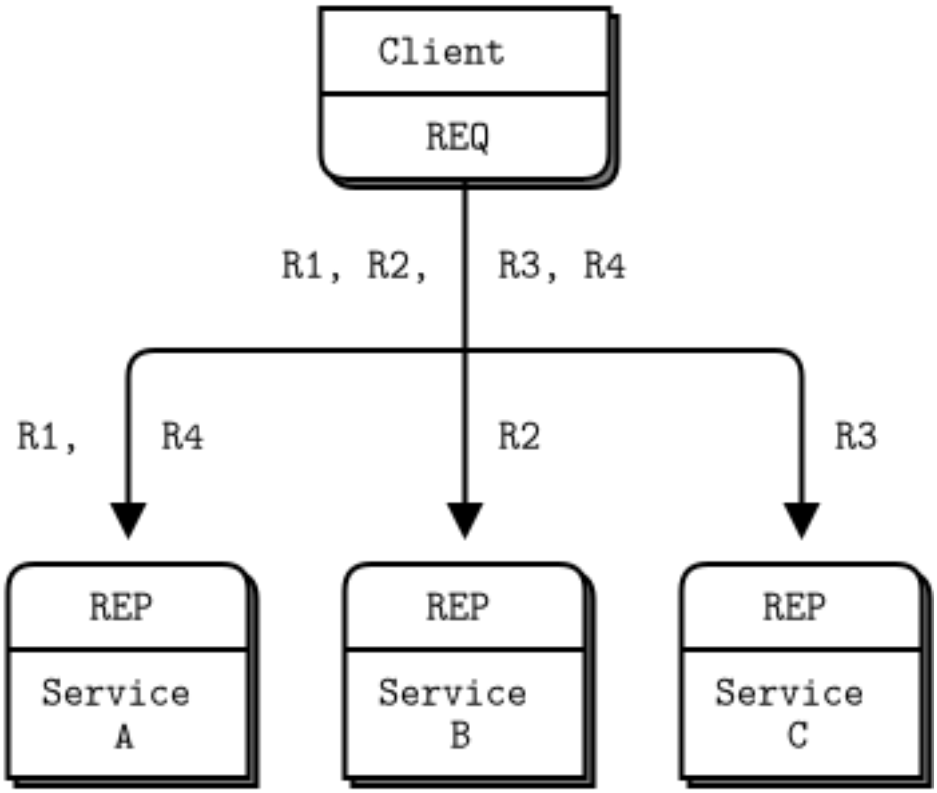


그림 16: Request Distribution

다수의 클라이언트들을 다수의 서버에 연결하는 방법에는 두 가지가 있습니다. 가장 강력한 방법은 각 클라이언트 소켓을 여러 서비스 단말에 연결하는 것입니다. 하나의 클라이언트 소켓은 여러 서비스 소켓에 연결할 수 있으며, REQ 소켓은 이러한 서비스들 간의 요청을 분배합니다. 다시 말해 하나의 클라이언트 소켓을 세 개의 서비스 단말에 연결한다고 하면 : 서비스 단말들(A, B 및 C)에 대하여 클라이언트는 R1, R2, R3, R4를 요청합니다. R1 및 R4는 서비스 A로 이동하고 R2는 B로 이동하고 R3은 서비스 C로 이동합니다.

- [옮긴이] 서버에서 서비스를 제공하기 때문에, 서버란 용어 대신 서비스로 대체하여 사용하기도 합니다.
- [옮긴이] 클라이언트가 REQ 소켓을 일련의 서버들의 REP 소켓에 연결하기 위한 예제는 다음과 같습니다.

hwclient_mc.c : REQ 소켓에 일련의 서버를 연결

```c
//  Hello World client
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>
#define sleep(n)    Sleep(n*1000)
#endif

int main (void)
{
    printf ("Connecting to hello world server...\n");
    void *context = zmq_ctx_new ();
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5555");
    zmq_connect (requester, "tcp://localhost:5556");
    zmq_connect (requester, "tcp://localhost:5557");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        char buffer [10];
        printf ("Sending Hello %d...\n", request_nbr);
        zmq_send (requester, "Hello", 5, 0);
        zmq_recv (requester, buffer, 10, 0);
        printf ("Received World %d\n", request_nbr);
    }
```

```
    zmq_close (requester);
    zmq_ctx_destroy (context);
    return 0;
}
```

hwserver_argv.c : 명령어 창에서 바인딩 주소를 받아 실행하는 서버

```
//  Hello World server

#include <zmq.h>
#include <stdio.h>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>
#define sleep(n)    Sleep(n*1000)
#endif
#include <string.h>
#include <assert.h>

int main (int argc, char *argv [])
{
    //  Socket to talk to clients
    void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, argv[1]);
    assert (rc == 0);

    while (1) {
        char buffer [10];
        zmq_recv (responder, buffer, 10, 0);
```

```
        printf ("Received Hello\n");
        sleep (1);          //  Do some 'work'
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc hwserver_argv.c libzmq.lib
 cl -EHsc hwclient_mc.c libzmq.lib
 ./hwclient_mc
Connecting to hello world server...
Sending Hello 0...
Received World 0
Sending Hello 1...
Received World 1
Sending Hello 2...
Received World 2
Sending Hello 3...
Received World 3
Sending Hello 4...
Received World 4
Sending Hello 5...
Received World 5
Sending Hello 6...
Received World 6
Sending Hello 7...
Received World 7
Sending Hello 8...
```

```
Received World 8
Sending Hello 9...
Received World 9
 ./hwserver_argv tcp://*:5555
Received Hello
Received Hello
Received Hello
Received Hello
 ./hwserver_argv tcp://*:5556
Received Hello
Received Hello
Received Hello
 ./hwserver_argv tcp://*:5557
Received Hello
Received Hello
Received Hello
```

이 디자인을 사용하면 더 많은 클라이언트를 쉽게 추가할 수 있습니다. 더 많은 서비스를 추가할 수도 있습니다. 각 클라이언트는 서비스에 요청들을 분배합니다. 그러나 각 클라이언트는 서비스 토폴로지(어디서 무슨 서비스를 제공하는지)를 알아야 합니다. 100개의 클라이언트들이 있고 3개의 서비스들을 추가하기로 결정한 경우 클라이언트들이 3개의 새로운 서비스에 대해 알 수 있도록 100 개의 클라이언트를 서비스들의 구성 정보를 변경하고 다시 시작해야 합니다.

슈퍼 컴퓨팅 클러스터에 자원이 부족하여 수백 개의 새로운 서비스 노드들을 추가해야 할 경우, 새벽 3시에 일어나 하고 싶지는 않으실 겁니다. 너무 많은 정적 조각들은 액체 콘크리트와 같습니다 : 지식이 분산되고 정적 조각이 많을수록 네트워크 토폴로지를 변경하는 것에는 많은 노력이 필요합니다. 우리가 원하는 것은 클라이언트들과 서비스들 사이에 네트워크 토폴로지에 대한 모든 지식을 중앙화하는 것입니다. 이상적으로는 토폴로지의 다른 부분을 건드리지 않고 원격 서비스 또는 클라이언트를 추가 및 삭제할 수 있어야 합니다.

따라서 우리는 이러한 유연성을 제공하는 작은 메시지 대기열 브로커를 작성하겠습니다.

브로커는 클라이언트의 프론트엔드와 서비스의 백엔드의 두 단말을 바인딩합니다. 그런 다음 `zmq_poll()`을 사용하여이 두 소켓의 활동을 모니터링하고 소켓에 메세지가 전달 되었을 때, 두 소켓 사이에서 메시지를 오가도록 합니다. 브로커는 명시적으로 어떤 대기열도 관리하지 않습니다 - ØMQd의 각 소켓에서 자동으로 처리하게 합니다.

REQ를 사용하여 REP와 통신할 때 엄격한 동기식 요청-응답가 필요하였습니다. 클라이언트가 요청을 보내면 서비스는 요청을 받아 응답을 보냅니다. 그런 다음 클라이언트는 응답을 받습니다. 만약 클라이언트나 서비스가 다른 작업을 시도하면(예 : 응답을 기다리지 않고 두 개의 요청을 연속으로 전송) 오류가 발생합니다.

그러나 우리의 브로커는 비차단적으로 동작하며. 명확하게 `zmq_poll()`을 이용하여 어떤 소켓에 이벤트를 기다릴 수 있지만, REP와 REQ 소켓은 사용할 수 없습니다.

그림 16 - 확장된 요청-응답

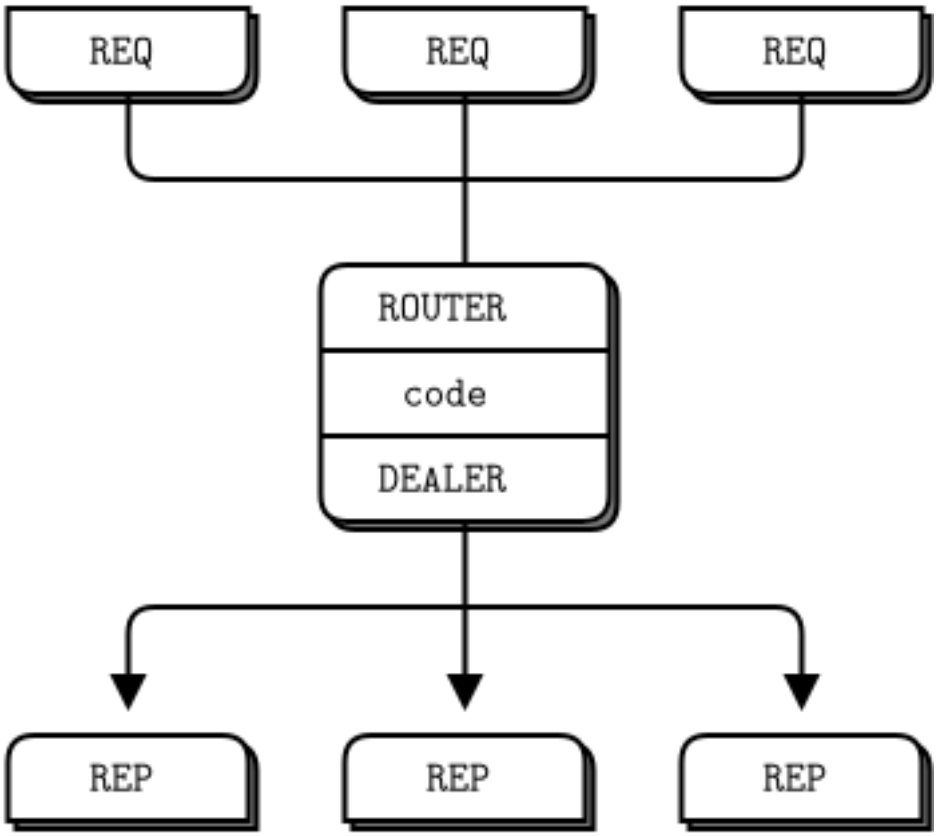


그림 17: Extended Request-Reply

다행히도 비차단 요청-응답을 수행할 수 있는 DEALER와 ROUTER라는 2개의 소켓이 있습니다. "3장-고급 요청-응답 패턴"에서 DEALER 및 ROUTER 소켓을 사용하여 모든 종류의 비동기 요청-응답 흐름을 구축하는 방법을 보도록 하겠습니다. 지금은 DEALER와 ROUTER가 중개자로 요청-응답을 확장하는 방법을 보도록 하겠습니다.

이 간단한 확장된 요청-응답 패턴으로, REQ 소켓은 ROUTER 소켓과 통신하고 DEALER 소켓은 REP 소켓과 통신합니다. DEALER와 ROUTER 사이에는 하나의 소켓에서 메시지를 가져와서 다른 소켓으로 보내는 코드(`zmq_proxy()`)가 존재해야 합니다.

요청-응답 브로커는 2개의 단말에 바인딩합니다. 하나는 클라이언트들을 연결(프론트엔드 소켓)하고 다른 하나는 작업자들을 연결(백엔드 소켓)하는 것입니다. 이 브로커를 테스트하기 위하여 백엔드 소켓에 연결하는 작업자들의 개수를 변경할 수도 있습니다. 다음은 요청하는

클라이언트 소스 코드입니다.

rrclient.c: 요청-응답 클라이언트

```
//  Hello World client
//  Connects REQ socket to tcp://localhost:5559
//  Sends "Hello" to server, expects "World" back

#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  Socket to talk to server
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5559");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        s_send (requester, "Hello");
        char *string = s_recv (requester);
        printf ("Received reply %d [%s]\n", request_nbr, string);
        free (string);
    }
    zmq_close (requester);
    zmq_ctx_destroy (context);
    return 0;
}
```

다음은 작업자(worker)의 코드입니다.

rrworker.c: 요청-응답 작업자

```
//  Hello World worker
//  Connects REP socket to tcp://localhost:5560
//  Expects "Hello" from client, replies with "World"

#include "zhelpers.h"
#ifndef _WIN32
#include <unistd.h>
#endif

int main (void)
{
    void *context = zmq_ctx_new ();

    //  Socket to talk to clients
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_connect (responder, "tcp://localhost:5560");

    while (1) {
        //  Wait for next request from client
        char *string = s_recv (responder);
        printf ("Received request: [%s]\n", string);
        free (string);

        //  Do some 'work'
        s_sleep (1000);

        //  Send reply back to client
        s_send (responder, "World");
    }
```

```
    //  We never get here, but clean up anyhow
    zmq_close (responder);
    zmq_ctx_destroy (context);
    return 0;
}
```

다음이 브로커 코드입니다. 멀티파트 메시지를 제대로 처리 할 수 있습니다.

rrbroker.c: 요청-응답 브로커

```
//  Simple request-reply broker

#include "zhelpers.h"

int main (void)
{
    //  Prepare our context and sockets
    void *context = zmq_ctx_new ();
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (frontend, "tcp://*:5559");
    zmq_bind (backend,  "tcp://*:5560");

    //  Initialize poll set
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    //  Switch messages between sockets
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
```

```
        if (items [0].revents & ZMQ_POLLIN) {
            while (1) {
                //  Process all parts of the message
                zmq_msg_init (&message);
                zmq_msg_recv (&message, frontend, 0);
                int more = zmq_msg_more (&message);
                zmq_msg_send (&message, backend, more? ZMQ_SNDMORE: 0);
                zmq_msg_close (&message);
                if (!more)
                    break;      //  Last message part
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            while (1) {
                //  Process all parts of the message
                zmq_msg_init (&message);
                zmq_msg_recv (&message, backend, 0);
                int more = zmq_msg_more (&message);
                zmq_msg_send (&message, frontend, more? ZMQ_SNDMORE: 0);
                zmq_msg_close (&message);
                if (!more)
                    break;      //  Last message part
            }
        }
    }
    //  We never get here, but clean up anyhow
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
```

```
    return 0;
}
```

그림 17 - 요청-응답 브로커

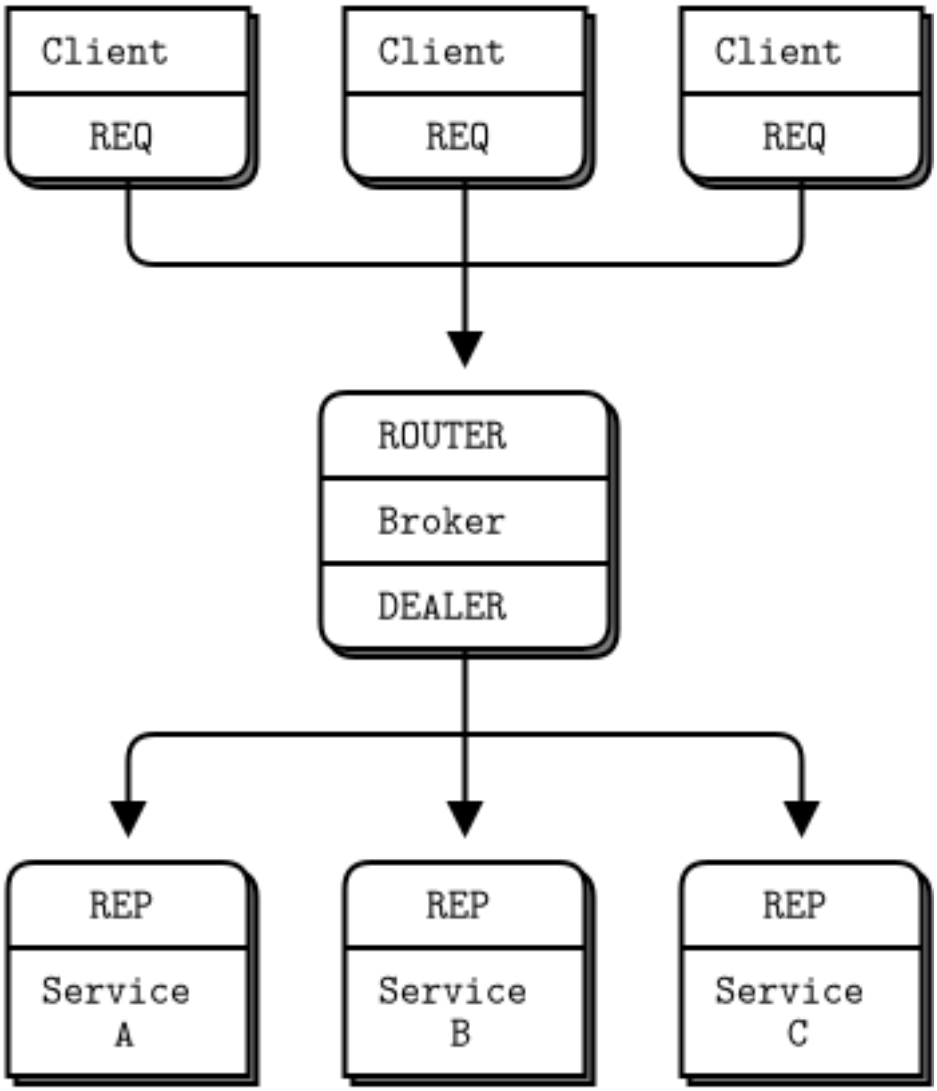


그림 18: Request-Reply Broker

요청-응답 브로커를 사용하면 클라이언트는 작업자를 보지 못하고 작업자는 클라이언트를 보지 않기 때문에 클라이언트/서버 아키텍처를 쉽게 확장할 수 있습니다. 유일한 정적 노드는

중간에 있는 브로커입니다.

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc rrclient.c libzmq.lib
 cl -EHsc rrworker.c libzmq.lib
 cl -EHsc rrproxy.c libzmq.lib

 ./rrbroker

 ./rrworker
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]
...

 ./rrclient
Received reply 0 [World]
Received reply 1 [World]
Received reply 2 [World]
Received reply 3 [World]
Received reply 4 [World]
...
```

- [옮긴이] 이런 구조에서는 작업자에서 제공하는 서비스를 구분해야 할 경우(설비 연계 서비스, 사용자 연계 서비스, 트랜젝션 처리 서비스 등) 개별 서비스에 대하여 요청/응답 브로커에서 분배합니다. 클라이언트에서 각 서비스를 제공하는 작업자들의 정보를 등록하여 찾아가게 하거나, 브로커에 서비스 추가/삭제하여 클라이언트가 브로커에

서비스를 요청하고 제공받게 할 수 있습니다. 작업자의 서비스들을 클라이언트에 정적으로 보관하거나, 브로커에서 서비스 관리하지 않기 위해서는 클라이언트에서 필요한 서비스를 찾을 수 있도록 서비스를 식별하기 위한 중앙의 저장소를 두어 운영할 수도 있습니다.

0.21.8 ØMQ의 내장된 프록시 함수

이전 섹션의 rrbroker의 코어 루프는 매우 유용하고 재사용 가능합니다. 적은 노력으로 PUB-SUB 포워더와 공유 대기열과 같은 작은 중개자를 구축할 수 있었습니다. ØMQ는 이 같은 기능을 감싼 하나의 함수 zmq_proxy()를 준비하고 있습니다.

```
zmq_proxy (frontend, backend, capture);
```

- [옮긴이] 아래의 코드를 zmq_proxy() 함수에서 대응할 수 있습니다.

```
    //  Initialize poll set
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    //  Switch messages between sockets
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            while (1) {
                //  Process all parts of the message
                zmq_msg_init (&message);
                zmq_msg_recv (&message, frontend, 0);
                int more = zmq_msg_more (&message);
```

```
            zmq_msg_send (&message, backend, more? ZMQ_SNDMORE: 0);
            zmq_msg_close (&message);
            if (!more)
                break;      //  Last message part
        }
    }
    if (items [1].revents & ZMQ_POLLIN) {
        while (1) {
            //  Process all parts of the message
            zmq_msg_init (&message);
            zmq_msg_recv (&message, backend, 0);
            int more = zmq_msg_more (&message);
            zmq_msg_send (&message, frontend, more? ZMQ_SNDMORE: 0);
            zmq_msg_close (&message);
            if (!more)
                break;      //  Last message part
        }
    }
}
```

2개의 소켓(또는 데이터를 캡처하려는 경우 3개 소켓)이 연결, 바인딩 및 구성되어야 합니다. `zmq_proxy()` 함수를 호출하면 rrbroker의 메인 루프를 시작하는 것과 같습니다. `zmq_proxy()`를 호출하도록 요청-응답 브로커를 다시 작성하겠습니다.

msgqueue.c: message queue broker

```
//  Simple message queuing broker
//  Same as request-reply broker but using shared queue proxy

#include "zhelpers.h"
```

```
int main (void)
{
    void *context = zmq_ctx_new ();

    //  Socket facing clients
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    int rc = zmq_bind (frontend, "tcp://*:5559");
    assert (rc == 0);

    //  Socket facing services
    void *backend = zmq_socket (context, ZMQ_DEALER);
    rc = zmq_bind (backend, "tcp://*:5560");
    assert (rc == 0);

    //  Start the proxy
    zmq_proxy (frontend, backend, NULL);

    //  We never get here...
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
cl -EHsc msgqueue.c libzmq.lib

./msgqueue
```

```
 ./rrworker
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]
...

 ./rrclient
Received reply 0 [World]
Received reply 1 [World]
Received reply 2 [World]
Received reply 3 [World]
Received reply 4 [World]
...
```

대부분의 ØMQ 사용자의 경우, 이 단계에서 "임의의 소켓 유형을 프록시에 넣으면 어떤 일이 일어날까?" 생각합니다. 짧게 대답하면 : 그것을 시도하고 무슨 일이 일어나는지 보아야 합니다. 실제로는 보통 ROUTER/DEALER, XSUB/XPUB 또는 PULL/PUSH을 사용합니다.

0.21.9 전송계층 간의 연결

ØMQ 사용자의 계속되는 요청은 "기술 X와 ØMQ 네트워크를 어떻게 연결합니까?" 입니다. 여기서 X는 다른 네트워킹 또는 메시징 기술입니다.

그림 18 - Pub-Sub 포워드 프록시 (Forwarder Proxy)

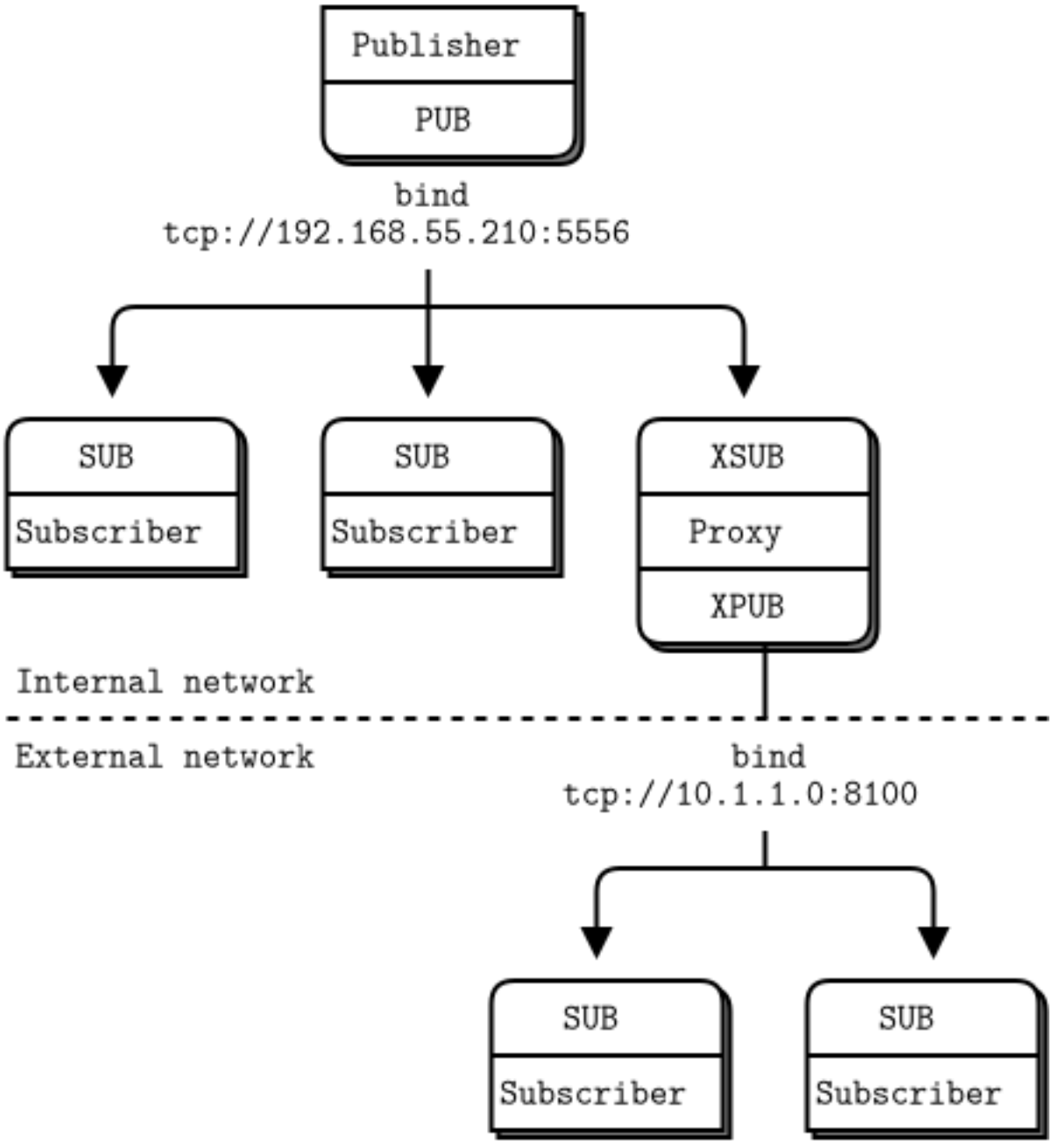


그림 19: Pub-Sub Forwarder Proxy

간단한 대답은 브리지(Bridge)를 만드는 것입니다. 브리지는 한 소켓에서 하나의 통신

규약을 말하고, 다른 소켓에서 두 번째 통신규약으로 변환하는 작은 응용프로그램입니다. 통신규약 번역기라고 할 수 있으며, ØMQ는 2개의 서로 다른 전송방식과 네트워크를 연결할 수 있습니다.

예제로, 발행자와 일련의 구독자들 간에 2개의 네트워크를 연결하는 작은 프록시를 작성하도록 하겠습니다. 프론트엔드 소켓은 날씨 서버가 있는 내부 네트워크를 향하며, 백엔드 소켓은 외부 네트워크의 일련의 구독자들을 향하게 합니다. 프록시의 프론트엔드 소켓에서 날씨 서비스에 구독하고 백엔드 소켓으로 데이터를 재배포하게 합니다.

wuproxy.c: 기상정보 변경 프록시

```
//  Weather proxy device

#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();

    //  This is where the weather server sits
    void *frontend = zmq_socket (context, ZMQ_XSUB);
    zmq_bind (frontend, "tcp://*:5556");

    //  This is our public endpoint for subscribers
    void *backend = zmq_socket (context, ZMQ_XPUB);
    zmq_bind (backend, "tcp://*:5557");

    //  Run the proxy until the user interrupts us
    zmq_proxy (frontend, backend, NULL);

    zmq_close (frontend);
    zmq_close (backend);
```

```
    zmq_ctx_destroy (context);
    return 0;
}
```

- [옮긴이] wuserver와 wuclient간에 wuproxy를 넣어서 테스트 가능합니다.

이전 프록시 예제와 매우 유사하지만, 중요한 부분은 프론트엔드와 백엔드 소켓이 서로 다른 2개의 네트워크에 있다는 것입니다. 예를 들어 이 모델을 사용하여 멀티캐스트 네트워크(pgm 전송방식)를 TCP 발행자에 연결할 수도 있습니다.

0.22 오류 처리와 ETERM

ØMQ의 오류 처리 철학은 빠른 실패와 회복의 조합입니다. 프로세스는 내부 오류에 대해 가능한 취약하고 외부 공격 및 오류에 대해 가능한 강력해야 대응해야 합니다. 유사한 예로 살아있는 세포는 내부 오류가 하나만 감지되면 스스로 죽지만, 외부로부터의 공격에는 모든 가능한 수단을 통하여 저항합니다.

어설션(assertion)은 ØMQ의 안정적인 코드에 필수적인 조미료입니다. 그들은 세포막에 있어야 하며 그런 막을 통하여 결함이 내부 또는 외부인지 구분하고, 불확실한 경우 설계 결함으로 수정해야 합니다.. C/C++에서 어설션은 오류 발생 시 응용프로그램을 즉시 중지합니다. 다른 언어에서는 예외 혹은 중지가 발생할 수 있습니다.

- [옮긴이] 어설션(assertion) 사용 사례(msgqueue.c에서 발취), assert()는 조건식이 거짓(false) 일 때 프로그램을 중단하며 참(true) 일 때는 프로그램이 계속 실행합니다.

```
    ...
    //  Socket facing clients
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    int rc = zmq_bind (frontend, "tcp://*:5559");
    assert (rc == 0);

    //  Socket facing services
```

```
    void *backend = zmq_socket (context, ZMQ_DEALER);
    rc = zmq_bind (backend, "tcp://*:5560");
    assert (rc == 0);
    ...
```

ØMQ가 외부 오류를 감지하면 호출 코드에 오류를 반환합니다. 드문 경우지만 오류를 복구하기 위한 명확한 전략이 없으면 메시지를 자동으로 삭제합니다.

지금까지 살펴본 대부분의 C 예제에서는 오류 처리가 없었습니다. 실제 코드는 모든 ØMQ 함수 호출에서 오류 처리를 수행해야 하며, C 이외의 다른 개발 언어 바인딩(예 : java)을 사용하는 경우 바인딩이 오류를 처리할 수 있습니다. C 개발 언어에서는 오류 처리 작업을 직접 해야 합니다. POSIX 관례로 시작하는 몇 가지 간단한 규칙이 있습니다.

- 객체 생성 함수의 경우 실패하면 NULL 값을 반환합니다.
- 데이터 처리하는 함수의 경우 처리되는 데이터의 바이트 크기를 반환하지만, 오류나 실패 시 -1을 반환합니다.
- 기타 처리 함수들의 경우 반환값이 0이면 성공, -1이면 실패입니다.
- 오류 코드는 `errno` 혹은 `zmq_errno(void)`에서 제공합니다.
- 로그를 위한 오류 메시지는 `zmq_strerror(int errnum)`를 통해 제공됩니다.

사용 예는 다음과 같습니다.

```
void *context = zmq_ctx_new ();
assert (context);
void *socket = zmq_socket (context, ZMQ_REP);
assert (socket);
int rc = zmq_bind (socket, "tcp://*:5555");
if (rc == -1) {
    printf ("E: bind failed: %s\n", strerror (errno));
    return -1;
}
```

- [옮긴이] hwserver.c를 수정(hwserver1.c)하여 "strerror(errno)" 확인하겠습니다.

hwserver1.c : Hellow World 서버 수정본

```
//  Hello World server

#include <zmq.h>
#include <stdio.h>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>
#define sleep(n)    Sleep(n*1000)
#endif
#include <string.h>
#include <assert.h>

int main (void)
{
    //  Socket to talk to clients
    void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5555");
    if (rc == -1){
        printf("E: bind failed: %s\n", zmq_strerror(zmq_errno()));
        return -1;
    }

    while (1) {
        char buffer [10];
        zmq_recv (responder, buffer, 10, 0);
        printf ("Received Hello\n");
```

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

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc hwserver1.c libzmq.lib
Microsoft (R) C/C++ 최적화 컴파일러 버전 19.16.27035(x64)
Copyright (c) Microsoft Corporation. All rights reserved.
hwserver1.c
Microsoft (R) Incremental Linker Version 14.16.27035.0
Copyright (C) Microsoft Corporation.  All rights reserved.
/out:hwserver1.exe
hwserver1.obj
libzmq.lib

 ./hwserver1

 ./hwserver1
E: bind failed: Address in use
```

치명적이지 않은 것으로 취급해야 하는 두 가지 주요 예외 조건이 있습니다 :

- 소스 코드 내에서 ZMQ_DONTWAIT 옵션이 포함된 메시지가 수신되고 대기 데이터가 없는 경우 ØMQ는 -1을 반환하고 errno를 EAGAIN으로 설정합니다.
- 하나의 스레드가 zmq_ctx_destroy()를 호출하고 다른 스레드가 여전히 차단 작업을 수행하는 경우 zmq_ctx_destroy() 호출은 컨텍스트를 닫고 모든 차단 호출은 -1로 종료되고 errno는 ETERM으로 설정됩니다.

C/C ++의 `assert()`는 최적화에 의해 완전히 제거되므로 `assert()`내에서 ØMQ API를 호출해서는 안됩니다. 최적화를 통해 모든`assert()`는 제거된 응용프로그램에서 개발자의 의도에 따라 ØMQ API 호출을 작성하여 오류가 발생할 경우 중단하게 해야 합니다.

- [옮긴이] assert() 매크로는 assert.h 헤더 파일에 정의되어 있으며, 원도우 VC 2017에서 디버그 모드에서만 동작하며 최적화된 릴리즈 모드에서는 제외됩니다.

그림 19 - 강제 종료 신호를 가진 병렬 파이프라인(Pipeline)

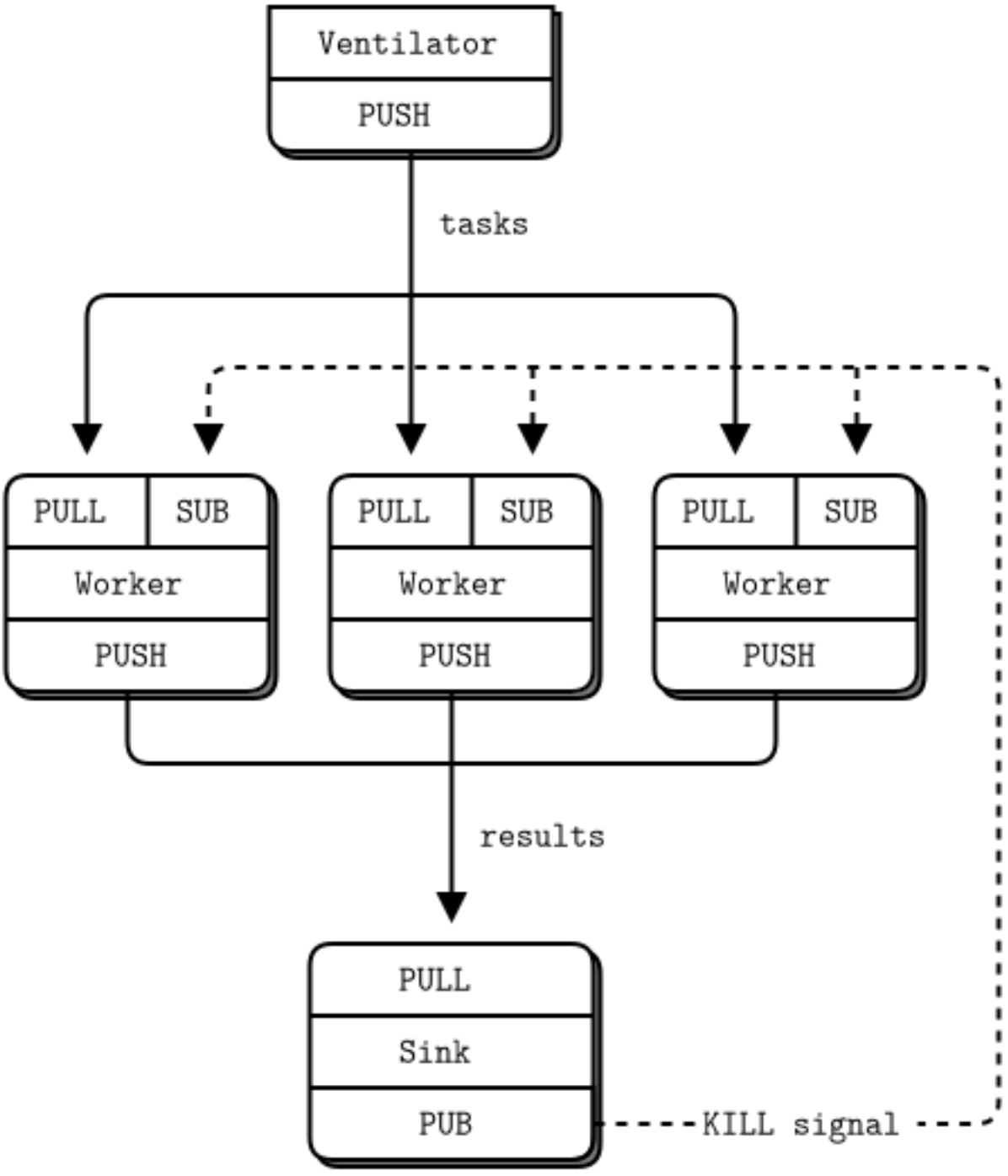


그림 20: Parallel Pipeline with Kill Signaling

프로세스를 깨끗하게 종료하는 방법을 알아보겠습니다. 이전 섹션의 병렬 파이프라인 예제를 보면, 백그라운드에서 많은 수의 작업자를 시작한 후 작업이 완료되면 수동으로 해당 작업자를 종료해야 했습니다. 이것을 작업이 완료되면 작업자에서 종료 신호를 보내 자동으로 종료하도록 하겠습니다. 이 작업을 수행하기 가장 좋은 대상은 작업이 완료된 시점을 알고 있는 수집기입니다.

작업자와 수집기를 연결하기 위하여 사용한 방법은 단방향 PUSH/PULL 소켓입니다. 다른 소켓 유형으로 전환하거나 다중 소켓을 혼합할 수 있습니다. 후자를 선택하여 : PUB-SUB 모델로 수집기에서 종료 메시지를 작업자에게 보냅니다.

- 수집기는 새로운 단말에 PUB 소켓을 생성합니다.
- 작업자들은 SUB 소켓을 수집기 단말에 연결합니다.
- 수집기가 일괄 작업의 종료 감지하면 PUB 소켓에 종료 신호를 보냅니다.
- 작업자들이 종료 메시지(kill message)를 감지하면 종료됩니다.

이러한 작업은 수집기에서 새로운 코드 추가가 필요합니다.

```
void *controller = zmq_socket (context, ZMQ_PUB);
zmq_bind (controller, "tcp://*:5559");
...
// 작업자(worker)를 종료시키는 신호를 전송
s_send (controller, "KILL");
```

작업자 프로세스에서는 2개의 소켓(호흡기에서 작업을 받아 오는 PULL 소켓과 제어 명령을 가져오는 SUB 소켓)을 이전에 보았던 zmq_poll() 기술을 사용해서 관리합니다.

taskwork2.c: 종료 신호를 처리하는 작업자

```
//  Task worker - design 2
//  Adds pub-sub flow to receive and respond to kill signal

#include "zhelpers.h"

int main (void)
```

```
{
    //  Socket to receive messages on
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

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

    //  Socket for control input
    void *controller = zmq_socket (context, ZMQ_SUB);
    zmq_connect (controller, "tcp://localhost:5559");
    zmq_setsockopt (controller, ZMQ_SUBSCRIBE, "", 0);

    //  Process messages from either socket
    while (1) {
        zmq_pollitem_t items [] = {
            { receiver, 0, ZMQ_POLLIN, 0 },
            { controller, 0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            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
```

```
        }
        //  Any waiting controller command acts as 'KILL'
        if (items [1].revents & ZMQ_POLLIN)
            break;                      //  Exit loop
    }
    zmq_close (receiver);
    zmq_close (sender);
    zmq_close (controller);
    zmq_ctx_destroy (context);
    return 0;
}
```

수정된 수집기 응용프로그램에서는 결과 수집이 완료되면 모든 작업자들에게 종료 메시지를 전송합니다.

tasksink2.c: 강제 종료(kill) 신호를 가진 병렬 작업 수집기(sink)

```
//  Task sink - design 2
//  Adds pub-sub flow to send kill signal to workers

#include "zhelpers.h"

int main (void)
{
    //  Socket to receive messages on
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    //  Socket for worker control
    void *controller = zmq_socket (context, ZMQ_PUB);
```

```
    zmq_bind (controller, "tcp://*:5559");

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

    //  Start our clock now
    int64_t start_time = s_clock ();

    //  Process 100 confirmations
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if (task_nbr % 10 == 0)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    printf ("Total elapsed time: %d msec\n",
        (int) (s_clock () - start_time));

    //  Send kill signal to workers
    s_send (controller, "KILL");

    zmq_close (receiver);
    zmq_close (controller);
    zmq_ctx_destroy (context);
```

```
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc taskwork2.c libzmq.lib
 cl -EHsc tasksink2.c libzmq.lib

 ./taskvent
Press Enter when the workers are ready:
Sending tasks to workers...
Total expected cost: 5086 msec



 ./taskwork2
13.22.47.94.99.65.5.88.38.22.36.27.3.47.98.36.66.43.82.93.5.91.41.
87.43.7.43.41.94.48.43.43.74.66.53.7.46.16.21.94.59.10.88.12.59.
40.27.43.26.44.51.3.4.59.85.81.96.16.72.71.86.26.46.37.45.20.8.27.
15.83.51.80.86.51.71.36.96.4.8.93.58.32.56.22.27.100.81.98.38.96.
91.27.9.82.94.57.52.33.92.39.



 ./tasksink2
:.........:.........:.........:.........:.........:.........:...
......:.........:.........:.........Total elapsed time: 5184 msec
```

0.23 인터럽트 신호 처리

실제 응용프로그램에서는 Ctrl-C나 SIGTERM과 같은 신호로 인한 인터럽트가 왔을 때 작업 종료 처리가 필요합니다. 기본적으로 이러한 인터럽트는 단순히 프로세스만 종료하지 메시지

대기열을 정리하지 않고, 파일을 깨끗하게 닫지 않습니다.

다양한 개발 언어에서 신호를 처리하는 방법을 보겠습니다.

- [옮긴이] 신호 처리를 위한 예제는 리눅스/유닉스에서 실행 필요합니다.

interrupt.c: Ctrl-C를 제대로 처리하는 방법

```
//  Shows how to handle Ctrl-C

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>

#include <zmq.h>

//  Signal handling
//
//  Create a self-pipe and call s_catch_signals(pipe's writefd) in your application
//  at startup, and then exit your main loop if your pipe contains any data.
//  Works especially well with zmq_poll.

#define S_NOTIFY_MSG " "
#define S_ERROR_MSG "Error while writing to self-pipe.\n"
static int s_fd;
static void s_signal_handler (int signal_value)
{
    int rc = write (s_fd, S_NOTIFY_MSG, sizeof(S_NOTIFY_MSG));
    if (rc != sizeof(S_NOTIFY_MSG)) {
        write (STDOUT_FILENO, S_ERROR_MSG, sizeof(S_ERROR_MSG)-1);
```

```
        exit(1);
    }
}

static void s_catch_signals (int fd)
{
    s_fd = fd;

    struct sigaction action;
    action.sa_handler = s_signal_handler;
    //  Doesn't matter if SA_RESTART set because self-pipe will wake up zmq_poll
    //  But setting to 0 will allow zmq_read to be interrupted.
    action.sa_flags = 0;
    sigemptyset (&action.sa_mask);
    sigaction (SIGINT, &action, NULL);
    sigaction (SIGTERM, &action, NULL);
}

int main (void)
{
    int rc;

    void *context = zmq_ctx_new ();
    void *socket = zmq_socket (context, ZMQ_REP);
    zmq_bind (socket, "tcp://*:5555");

    int pipefds[2];
    rc = pipe(pipefds);
    if (rc != 0) {
```

```
        perror("Creating self-pipe");
        exit(1);
    }
    int flags = fcntl(pipefds[0], F_GETFL, 0);
    if (flags < 0) {
        perror ("fcntl(F_GETFL)");
        exit(1);
    }
    rc = fcntl (pipefds[0], F_SETFL, flags | O_NONBLOCK);
    if (rc != 0) {
        perror ("fcntl(F_SETFL)");
        exit(1);
    }

    s_catch_signals (pipefds[1]);

    zmq_pollitem_t items [] = {
        { 0, pipefds[0], ZMQ_POLLIN, 0 },
        { socket, 0, ZMQ_POLLIN, 0 }
    };

    while (1) {
        rc = zmq_poll (items, 2, -1);
        if (rc == 0) {
            continue;
        } else if (rc < 0) {
            if (errno == EINTR) { continue; }
            perror("zmq_poll");
            exit(1);
```

```
        }

        // Signal pipe FD
        if (items [0].revents & ZMQ_POLLIN) {
            char buffer [1];
            read (pipefds[0], buffer, 1);  // clear notifying byte
            printf ("W: interrupt received, killing server...\n");
            break;
        }

        // Read socket
        if (items [1].revents & ZMQ_POLLIN) {
            char buffer [255];
            // Use non-blocking so we can continue to check self-pipe via zmq_poll
            rc = zmq_recv (socket, buffer, 255, ZMQ_DONTWAIT);
            if (rc < 0) {
                if (errno == EAGAIN) { continue; }
                if (errno == EINTR) { continue; }
                perror("recv");
                exit(1);
            }
            printf ("W: recv\n");

            // Now send message back.
            // ...
        }
    }

    printf ("W: cleaning up\n");
```

```
    zmq_close (socket);
    zmq_ctx_destroy (context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트(CentOS 7에서 수행함)

```
[gmsec@felix C]$ gcc -o interrupt interrupt.c -lzmq
[gmsec@felix C]$ gcc -o hwclient hwclient.c -lzmq
[gmsec@felix C]$ ./hwclient
Connecting to hello world server...
Sending Hello 0...
Received World 0
Sending Hello 1...
Received World 1
Sending Hello 2...

[gmsec@felix C]$ ./interrupt
W: recv
W: recv
^C
W: interrupt received, killing server...
W: cleaning up
```

이 프로그램은 `s_catch_signals()`함수가 Ctrl-C(SIGINT) 및 SIGTERM 신호를 잡을 수 있게 하며 `s_catch_signals()` 핸들러는 전역 변수 `s_interrupted`를 설정합니다. 신호 처리기 덕분에 응용프로그램이 자동으로 종료되지 않게 하여 의도에 따라 자원을 정리하고 종료할 수 있습니다. 이제 명시적으로 인터럽트 발생을 확인하고 올바르게 처리해야 합니다. 메인 코드 시작 시 `s_catch_signals()`(interrupt.c에서 복사)를 호출하여 신호 처리를 설정할 수 있습니다. 인터럽트는 다음과 같이 ØMQ API 호출에 영향을 미칩니다.

- 코드가 차단 호출(메시지 전송, 메시지 수신 또는 폴링)에 의해 차단된 경우, 신호가 도착하면 호출은 EINTR 반환합니다.
- s_recv()와 같은 래퍼 함수는 인터럽트 발생 시 NULL을 반환합니다.

따라서 EINTR 반환 코드나 NULL 반환 시 s_interrupted 전역 변수를 점검하십시오. 전형적인 사용 예시입니다.

```
s_catch_signals ();
client = zmq_socket (...);
while (!s_interrupted) {
    char *message = s_recv (client);
    if (!message)
        break;          //  Ctrl-C 사용될 경우
}
zmq_close (client);
```

s_catch_signals()를 호출하고 인터럽트를 테스트하지 않으면 응용프로그램이 Ctrl-C 및 SIGTERM에 무시하게 되어, 유용할지는 몰라도 비정상적인 종료에 의한 메모리 누수가 발생할 수 있습니다.

0.24 메모리 누수 탐지

오랫동안 구동되는 응용프로그램의 경우 올바르게 메모리 관리를 수행하지 않으면 사용 가능한 메모리 공간을 모두 차지하여 충돌이 발생합니다. 메모리 관리를 자동으로 수행하는 개발 언어를 사용한다면 축하할 일이지만 C 나 C++와 같이 메모리 관리가 필요한 개발 언어로 프로그래밍하는 경우 valgrind 사용을 통하여 프로그램에서 메모리 누수(leak)를 탐지할 수 있습니다.

유분투 혹은 데비안 운영체제에서 valgrind를 설치하려면 다음 명령을 실행하십시오.

```
sudo apt-get install valgrind
```

- [옮긴이] CentOS의 경우 아래와 같이 설치를 진행합니다.

```
[root@felix C]# yum install valgrind
Last metadata expiration check: 0:48:43 ago on Fri 17 Jan 2020 09:34:23 AM KST.
Package valgrind-1:3.14.0-10.el8.x86_64 is already installed.
Dependencies resolved.
================================================================================
 Package    Arch   Version      Repository     Size
================================================================================
Upgrading:
 valgrind   x86_64  1:3.15.0-9.el8   AppStream  12 M
 valgrind-devel x86_64 1:3.15.0-9.el8   AppStream   90 k
Transaction Summary
================================================================================
Upgrade  2 Packages

Total download size: 12 M
Is this ok [y/N]: y
Downloading Packages:
(1/2): valgrind-devel-3.15.0-9.el8.x86_64.rpm     992 kB/s |  90 kB     00:00
(2/2): valgrind-3.15.0-9.el8.x86_64.rpm      3.6 MB/s |  12 MB     00:03
--------------------------------------------------------------------------------
...
```

기본적으로 ØMQ는 valgrind가 많은 로그를 발생시키며, 이러한 경고를 제거하려면 다음 내용을 포함하는 vg.supp라는 파일을 작성하십시오.

```
{
   <socketcall_sendto>
   Memcheck:Param
   socketcall.sendto(msg)
   fun:send
   ...
}
{
    <socketcall_sendto>
    Memcheck:Param
    socketcall.send(msg)
    fun:send
    ...
}
```

Ctrl-C 수행 후에 응용프로그램이 완전히 종료되도록 수정하십시오. 자체 종료되는 응용프로그램에서는 필요하지 없겠지만, 오랫동안 구동되는 응용프로그램의 경우 필수이며, 그렇지 않을 경우 valgrind가 현재 할당된 모든 메모리에 대하여 경고를 표시합니다.

응용프로그램을 빌드 수행 시 -DDEBUG 옵션을 사용하면 valgrind가 메모리 누수 위치를 정확하게 알려 줄 수 있습니다.

- 마지막으로 'vagrind'을 아래와 같이 실행하십시오.

```
valgrind --tool=memcheck --leak-check=full --suppressions=vg.supp someprog
```

그리고 보고된 오류를 수정하면 아래와 같이 행복한 메시지가 출력됩니다.

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

- [옮긴이] CentOS의 경우 아래와 테스트를 수행합니다.

```
[zedo@sook C]$ ./interrupt
^CW: interrupt received, killing server...
W: cleaning up

[zedo@sook C]$ valgrind --tool=memcheck --leak-check=full --suppressions=vg.supp
 ./interrupt
==5157== Memcheck, a memory error detector
==5157== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5157== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5157== Command: ./interrupt
==5157==
^CW: interrupt received, killing server...
W: cleaning up
==5157==
==5157== HEAP SUMMARY:
==5157==     in use at exit: 0 bytes in 0 blocks
==5157==   total heap usage: 41 allocs, 41 frees, 23,038 bytes allocated
==5157==
==5157== All heap blocks were freed -- no leaks are possible
==5157==
==5157== For lists of detected and suppressed errors, rerun with: -s
==5157== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

0.25 ØMQ에서 멀티스레드

ØMQ는 멀티스레드 응용프로그램의 작성에 최적화되어 있습니다. 전통적인 소켓을 사용하셨다면, ØMQ 소켓 사용 시 약간의 재조정이 필요하지만, ØMQ 멀티스레딩은 기존에 알고 계셨던 멀티스레드 응용프로그램 작성 경험이 거의 필요하지 않습니다. 기존의 지식을 정원에

던져버리고 기름을 부어 태워 버리십시오. 책(지식)을 불태우는 경우는 드물지만, 동시성 프로그래밍의 경우 필요합니다.

ØMQ에서는 완벽한 멀티스레드 응용프로그램을 만들기 위해(그리고 문자 그대로), 뮤텍스, 잠금이 불필요하며 ØMQ 소켓을 통해 전송되는 메시지를 제외하고는 어떤 다른 형태의 스레드 간 통신이 필요하지 않습니다.

- [옮긴이] 전통적인 멀티스레드 응용프로그램에서는 스레드 간의 동기화를 위해 뮤텍스, 잠금, 세마포어를 사용하여 교착을 회피합니다.

"완벽한 멀티스레드 프로그램"은 작성하기 쉽고 이해하기 쉬운 코드로 어떤 프로그래밍 언어와 운영체제에서도 동일한 설계 방식으로 동작하며, CPU 수에 비례하여 자원의 낭비 없이 성능이 확장되어야 합니다.

멀티스레드 코드 작성하기 위한 기술(잠금 및 세마포어, 크리티컬 섹션 등)을 배우기 위해 수년의 시간을 보냈다면, 그것이 아무것도 아니라는 것을 깨닫는 순간 어처구니가 없을 것입니다. 30년 이상의 동시성 프로그래밍에서 배운 교훈이 있다면, 그것은 단지 "상태(& 자원)를 공유하지 마십시오"입니다. 이것은 한잔의 맥주를 같이 마시려고 하는 두 명의 술꾼들과 같습니다. 그들이 좋은 친구인지는 중요하지 않습니다. 조만간 그들은 싸우게 될 것입니다. 그리고 술꾼들과 테이블이 많을수록 맥주를 놓고 서로 더 많이 싸우게 됩니다. 멀티스레드 응용프로그램의 비극의 대다수는 술집에서 술꾼들의 싸움처럼 보입니다.

고전적인 공유 상태 멀티스레드 코드를 작성할 때 싸워야 할 일련의 이상한 문제들은 스트레스와 위험으로 직결되지 않는다면 재미있을 수도 있지만, 이러한 코드는 정상적으로 동작하는 것처럼 보여도 갑자기 중단될 수 있습니다. 세계 최고의 경험을 가진 대기업(Microsoft)에서 "멀티스레드 코드에서 발생 가능한 11 개의 문제" 목록을 발표하였으며, 잊어버린 동기화(forgotten synchronization), 부적절한 세분화(incorrect granularity), 읽기 및 쓰기 분열(read and write tearing), 잠금 없는 순서 변경(lock-free reordering), 잠금 호송(lock convoys), 2 단계 댄스(two-step dance) 및 우선순위 반전(priority inversion)을 다루고 있습니다.

- [옮긴이] 잠금 호송(lock convoys)은 잠금(Lock)을 소유한 스레드가 스케줄링에서 제외되어 지연됨으로써 그 잠금(Lock) 해제를 기다리는 다른 스레드(Thread)들도 함께 지연되는 현상입니다.

몰론 위에서는 11개가 아닌 7개의 문제만 나열하였습니다. 하지만 요지는 발전소 혹은 주식 시장과 같이 기간산업에 적용된 코드에서, 바쁜 목요일 오후 3시에 2 단계 잠금 호송으로 인하여 처리가 지연되기를 원하지는 않습니다. 용어가 실제로 의미하는 것에는 누구도 신경 쓰지 않으며 이것으로 프로그래밍이 되지도 않으며 더 복잡한 해결 방안으로 인한 더 복잡한 부작용과 싸우게 됩니다.

이와 같이 널리 사용되는 모델들은 전체 산업의 기반임에도 불구하고 근본적으로 잘못되었으며, 공유 상태 동시성(shared state concurrency)은 그중 하나입니다. 인터넷이 그러하듯 제한 없이 확장하고자 하는 코드는 메시지를 보내고 어떤 것도 공유하지 않는 것입니다(결함 있는(broken) 프로그래밍 모델에 대한 일반적인 오류는 제외).

ØMQ로 즐거운 멀티스레드 코드를 작성하려면 몇 가지 규칙을 따라야 합니다.

- 스레드 내에 데이터를 개별적으로 고립시키고 스레드들 간에 공유하지 않습니다. 예외는 ØMQ 컨텍스트로 스레드 안전합니다.
- 전통적인 병렬 처리인 뮤텍스, 크리티컬 섹션, 세마포어 등을 멀리 하며, 이것은 ØMQ 응용프로그램에 부적절합니다.
- 프로세스 시작 시 단 하나의 ØMQ 컨텍스트를 생성하여 inproc(스레드 간 통신) 소켓을 통해 연결하려는 모든 스레드들에 전달합니다.
- 응용프로그램에서 구조체를 생성하기 위하여 할당된 스레드들(attached threads)을 사용하며, inproc상에서 페어 소켓을 사용하여 부모 스레드에 연결합니다. 이러한 패턴은 부모 소켓을 바인딩 한 다음에 부모 소켓을 연결하는 자식 스레드를 만듭니다.
- 독립적인 작업을 위하여 자체 컨텍스트를 가진 해제된 스레드(detached threads) 사용합니다. TCP상에서 연결하며 나중에 소스 코드의 큰 변경 없이 독립적인 프로세스들로 전환 가능합니다.
- 모든 스레드들 간의 정보 교환은 ØMQ 메시지로 이루어지며, 어느 정도 공식적으로 정의할 수 있습니다.
- 스레드들 간에 ØMQ 소켓을 공유하지 말아야 하며, ØMQ 소켓은 스레드 안전하지 않습니다. 소켓을 다른 스레드로 전환하는 것은 기술적으로는 가능하지만 숙련된 기술이

필요합니다. 여러 스레드에서 하나의 소켓을 취급하는 유일한 방법은 마법 같은 가비지 수집을 가진 언어 바인딩 정도입니다.

- [옮긴이] 스레드 안전은 스레드를 통한 동시성 프로그래밍 수행 시, 교착, 경쟁조건을 감지하여 회피할 수 있게 합니다.

응용프로그램에서 2개 이상의 프록시를 시작해야 하는 경우, 예를 들어 각 스레드에서 각각의 프록시를 실행하려고 하면, 한 스레드에서 프록시 프론트엔드 및 백엔드 소켓을 작성한 후 다른 스레드의 프록시로 소켓을 전달하게 되면 오류가 발생하기 쉽습니다. 이것은 처음에는 작동하는 것처럼 보이지만 실제 사용에서는 무작위로 실패합니다. 알아두기 : 소켓을 만든 스레드를 제외하고 소켓을 사용하거나 닫지 마십시오.

이러한 규칙을 따르면 우아한 멀티스레드 응용프로그램을 쉽게 구축할 수 있으며 나중에 필요에 따라 스레드를 별도의 프로세스로 분리할 수 있습니다. 응용프로그램 로직은 요구되는 규모에 따라, 스레드들, 프로세스들 혹은 노드들로 확장할 수 있습니다.

ØMQ는 가상의 "그린"스레드 대신 기본 운영체제 스레드를 사용합니다. 장점은 새로운 스레드 API를 배울 필요가 없고 ØMQ 스레드가 운영체제에 매핑되어 사용 가능합니다. 인텔의 스레드 검사기와 같은 표준 도구를 사용하여 응용프로그램이 수행 중인 작업을 확인할 수 있습니다. 단점은 운영체제 기반 스레드 API가 항상 이식 가능한 것은 아니며 거대한 수의 스레드가 필요한 경우 일부 운영체제에서는 성능에 영향을 주게 됩니다.

실제 동작 방법으로 이전 Hello World 서버(hwserver)를 보다 유능한 것으로 변환하겠습니다. 원래 서버는 단일 스레드로 구동되었습니다. 요청당 작업이 적을 경우 문제 되지 않지만 : 하나의 ØMQ 스레드가 CPU 코어에서 최고 속도로 실행되며 대기 없이 많은 작업을 수행할 수 있습니다. 그러나 현실적인 서버는 요청마다 중요한 작업을 수행해야 합니다. 단일 CPU 코어로는 10,000개의 클라이언트의 요청을 하나의 스레드인 서버에서 처리하기에는 충분하지 않습니다. 그래서 현실적인 서버는 여러 작업자 스레드들을 구동시켜 10,000개의 클라이언트의 요청을 최대한 빨리 받아 작업자 스레드에 배포합니다. 작업자 스레드들은 작업을 분산 처리하여 결국 응답을 보냅니다.

물론 브로커와 외부 작업자 프로세스를 사용하여 모든 작업을 수행할 수 있지만, 16개 프로세스로 16 코어 CPU를 소모하기보다는 하나의 프로세스에서 멀티스레드를 구동하는 것이 더욱 쉽습니다. 추가로 하나의 프로세스에서 작업자 스레드들을 실행하면 네트워크 홉, 지연시간, 트래픽을 줄일 수 있습니다.

멀티스레드 버전의 Hello World 서비스는 기본적으로 하나의 프로세스 내에서 브로커와 작업자로 구성하게 합니다.

mtserver.c: 멀티스레드 서비스

```
//  Multithreaded Hello World server

#include "zhelpers.h"
#include <pthread.h>
#ifndef _WIN32
#include <unistd.h>
#endif

static void *
worker_routine (void *context) {
    //  Socket to talk to dispatcher
    void *receiver = zmq_socket (context, ZMQ_REP);
    zmq_connect (receiver, "inproc://workers");

    while (1) {
        char *string = s_recv (receiver);
        printf ("Received request: [%s]\n", string);
        free (string);
        //  Do some 'work'
        s_sleep (1000);
        //  Send reply back to client
        s_send (receiver, "World");
    }
    zmq_close (receiver);
    return NULL;
}
```

```
int main (void)
{
    void *context = zmq_ctx_new ();

    //  Socket to talk to clients
    void *clients = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (clients, "tcp://*:5555");

    //  Socket to talk to workers
    void *workers = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (workers, "inproc://workers");

    //  Launch pool of worker threads
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_routine, context);
    }
    //  Connect work threads to client threads via a queue proxy
    zmq_proxy (clients, workers, NULL);

    //  We never get here, but clean up anyhow
    zmq_close (clients);
    zmq_close (workers);
    zmq_ctx_destroy (context);
    return 0;
}
```

그림 20 - 멀티스레드 서비스

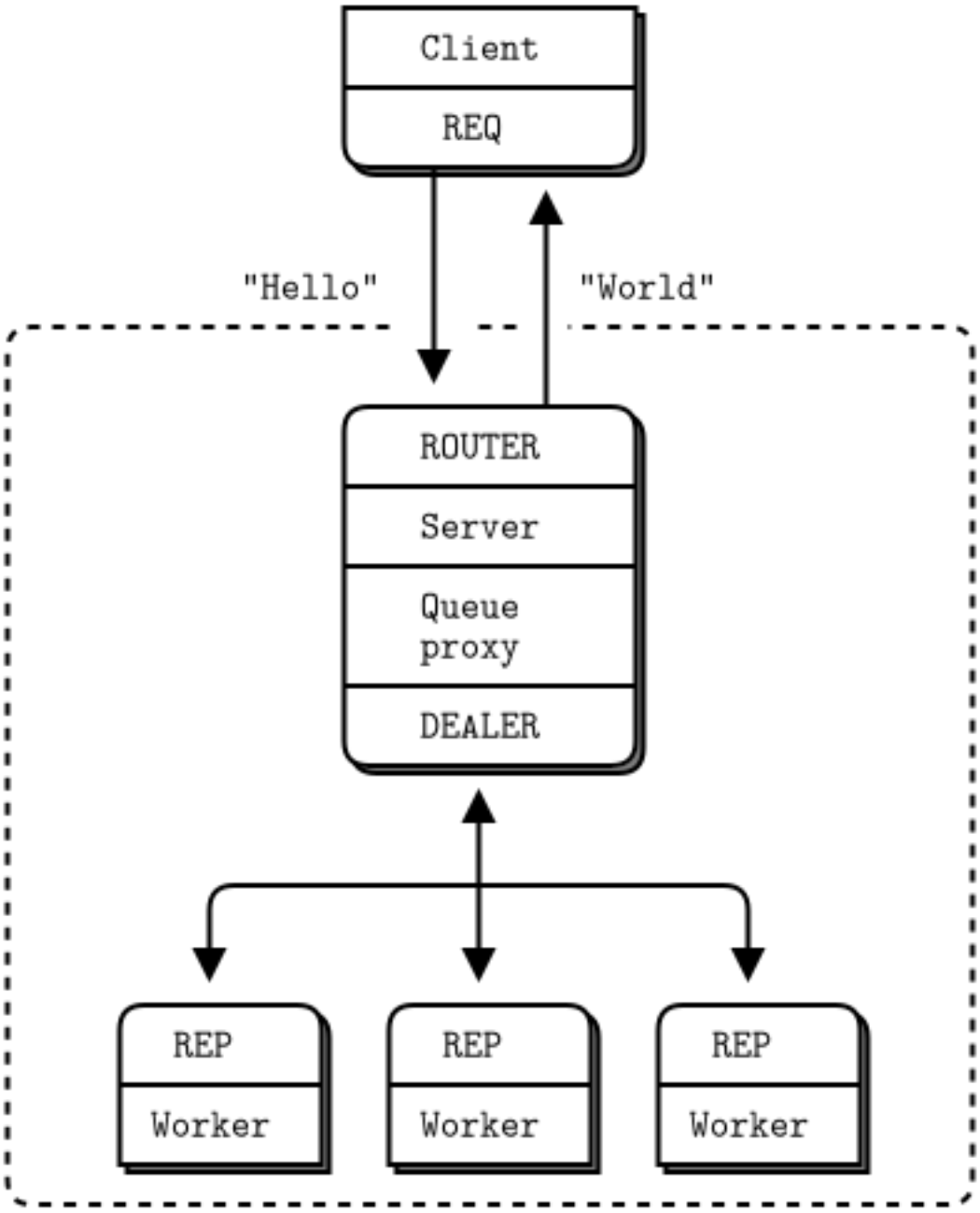


그림 21: 멀티스레드 서비스

- [옮긴이] 윈도우 운영체제에서 POSIX 스레드(pthread) 사용하기 위해서는 pthreads

Win32를 사용합니다. 응용프로그램 빌드 시 timespec 관련 오류(Error C2011 'timespec': 'struct' type redefinition) 발생할 경우 pthread.h" 파일에#define HAVE_STRUCT_TIMES 추가합니다. 사이트(ftp://sourceware.org/pub/pthreads-win32/dll-latest)에서 최신 파일(pthreadVC2.dll, pthreadVC2.lib, pthread.h, sched.h, semaphore.h)을 받아 설치합니다.

이제 코드를 보면 무엇을 하는지 아실 수 있으실 겁니다. 작동 방법 :

- 서버(mtserver)는 일련의 작업자 스레드들(worker threads)를 시작합니다. 각 작업자 스레드는 REP 소켓을 만들어 연결하여 소켓의 요청을 처리합니다. 작업자 스레드는 단일 스레드 서버와 같습니다. 유일한 차이점은 전송방식(tcp 대신 inproc)과 바인드-연결(bind-connect) 방향입니다.
- 서버는 클라이언트(hwclient)와 통신하기 위해 ROUTER 소켓을 생성하고 외부 인터페이스(tcp를 통해)에 바인딩(bind)합니다.
- 서버는 작업자들과 통신하기 위해 DEALER 소켓을 생성하고 내부 인터페이스 (inproc 통해)에 바인딩(bind)합니다.
- 서버는 두 소켓(ROUTER-DEALER)을 연결하는 프록시(zmq_proxy())를 시작합니다. 프록시는 모든 클라이언트에서 들어오는 요청을 작업자들에게 배포하며 작업자들의 응답을 클라이언트에 전달합니다.
- [옮긴이] 빌드 및 테스트

```
 cl -EHsc mtserver.c libzmq.lib pthreadVC2.lib
Microsoft (R) C/C++ 최적화 컴파일러 버전 19.16.27035(x64)
Copyright (c) Microsoft Corporation. All rights reserved.
mtserver.c
Microsoft (R) Incremental Linker Version 14.16.27035.0
Copyright (C) Microsoft Corporation.  All rights reserved.
/out:mtserver.exe
mtserver.obj
```

```
libzmq.lib
pthreadVC2.lib

 ./mtserver
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]
Received request: [Hello]

 ./hwclient
Connecting to hello world server...
Sending Hello 0...
Received World 0
Sending Hello 1...
Received World 1

 ./hwclient
Connecting to hello world server...
Sending Hello 0...
Received World 0
Sending Hello 1...
Received World 1
```

스레드들의 생성은 대부분 프로그래밍 언어에서 이식성이 없다는 점에 유의하십시오. POSIX 라이브러리 pthreads이지만 윈도우에서는 다른 API를 사용해야 됩니다. 예제에서는 `pthread_create()`를 호출하여 정의된 작업자의 기능을 수행하고 있습니다. '3장-고급 요청-응답 패턴'에서 이식 가능한 고급 API(CZMQ)로 변환 방법을 살펴보겠습니다.

여기서 "작업(작업자 스레드가 요청을 받아 응답)"은 1초 동안 대기(`s_sleep(1000)`)합니다. 우리는 다른 노드와 통신하는 것을 포함하여 작업자들에 대하여 무엇이든 할 수 있습니다. 멀티스레드 서버(mtserver)를 ØMQ 소켓과 노드처럼 보이게 하였습니다. 그리고 REQ-REP

통로는 REQ-ROUTER-queue-DEALER-REP로 사용되는 것을 확인하였습니다.

0.26 스레드들간의 신호(PAIR 소캣)

ØMQ로 멀티스레드 응용프로그램을 만들 때 스레드들 간 협업하는 방법에 대하여 궁금하실 겁니다. `sleep()` 함수를 삽입하거나 멀티스레딩 기술인 세마포어 혹은 뮤텍스와 같이 사용하고 싶더라도 ØMQ 메시지만 사용해야 합니다. 술꾼과 맥주병 이야기에서 언급한 것처럼 스레드들 간에 메시지 전달만 수행하고 공유하는 상태(& 자원)가 없어야 합니다.

준비되셨다면 서로 신호를 보내는 3개의 스레드를 만들어 봅시다. 예제에서 inproc 전송 방식로 PAIR 소켓을 사용합니다.

mtrelay.c: 멀티 스레드(MT) relay

```
//  Multithreaded relay

#include "zhelpers.h"
#include <pthread.h>

static void *
step1 (void *context) {
    //  Connect to step2 and tell it we're ready
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step2");
    printf ("Step 1 ready, signaling step 2\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}
```

```
static void *
step2 (void *context) {
    //  Bind inproc socket before starting step1
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step2");
    pthread_t thread;
    pthread_create (&thread, NULL, step1, context);

    //  Wait for signal and pass it on
    char *string = s_recv (receiver);
    printf ("Step 2 received sting from step 1 : %s\n", string);
    free (string);
    zmq_close (receiver);

    //  Connect to step3 and tell it we're ready
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step3");
    printf ("Step 2 ready, signaling step 3\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}

int main (void)
{
    void *context = zmq_ctx_new ();

    //  Bind inproc socket before starting step2
```

```
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step3");
    pthread_t thread;
    pthread_create (&thread, NULL, step2, context);

    //  Wait for signal
    char *string = s_recv (receiver);
    printf ("Step 3 received sting from step 2 : %s\n", string);
    free (string);
    zmq_close (receiver);

    printf ("Test successful!\n");
    zmq_ctx_destroy (context);
    return 0;
}
```

그림 21 - 릴레이(Relay)

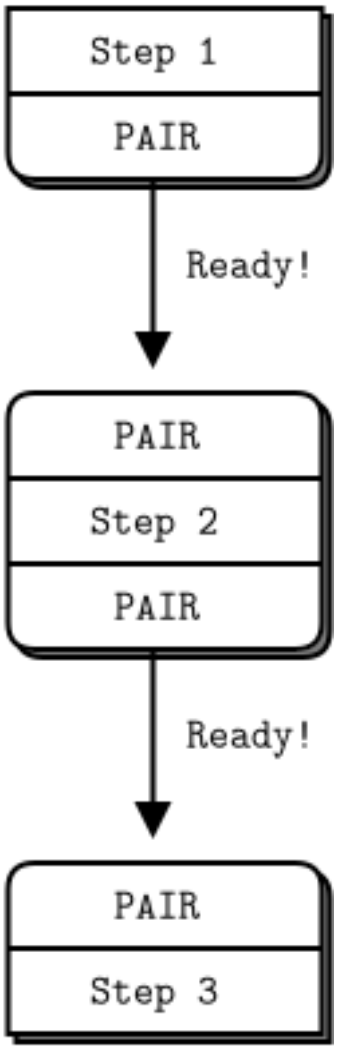


그림 22: Relay

이것은 ØMQ 멀티스레드을 위한 고전적인 패턴입니다.

- 2개의 스레드는 공유 컨텍스트를 통하여 inproc로 통신합니다.
- 부모 스레드는 하나의 소캣을 만들어 `inproc://step3`(혹은 step2)에 바인딩한 다음, 자식 스레드를 시작하여 컨텍스트를 전달합니다.
- 자식 스레드는 두 번째 소캣을 만들어서 `inproc://step3`(혹은 step2)에 연결한 다음, 준비된 부모 스레드에 신호를 보냅니다.

이 패턴(ZMQ_PAIR)을 사용하는 멀티스레드 코드는 프로세스에서는 사용할 수 없습니다. inproc 및 소켓 쌍(PAIR)을 사용하여 강하게 연결된 응용프로그램(스레드가 구조적으로 상호 의존적)을 작성할 수 있습니다. 낮은 지연 시간이 핵심적일 경우 사용하시기 바랍니다. 다른 디자인 패턴으로 느슨하게 연결된 응용프로그램의 경우, 스레드들이 자체 컨텍스트가 있고 ipc 또는 tcp를 통해 통신합니다. 느슨하게 연결된 스레드들은 독립적인 프로세스로 쉽게 분리 할 수 있습니다.

PAIR 소켓을 사용한 예제를 처음으로 보여 주었습니다. PAIR를 사용하는 이유는 다른 소켓 조합으로도 동작할 것 같지만 이러한 신호 인터페이스로 사용하면 부작용이 있습니다.

- 송신자는 PUSH를 사용하고 수신자 PULL을 사용할 수 있습니다. 이것은 단순하고 동작하지만 PUSH는 모든 사용 가능한 수신자에게 메시지를 분배합니다. 실수로 2개의 수신자를 시작한 경우(예 : 이미 첫 번째 실행 중이고 잠시 후(1초) 두 번째는 시작한 경우) 신호의 절반을 "손실"됩니다. PAIR는 하나 이상의 연결을 거부할 수 있는 장점이 있습니다. PAIR 독점적(exclusive)합니다.

- 송신자는 DEALER를, 수신자는 ROUTER를 사용할 수 있습니다. 그러나 ROUTER는 메시지를 "봉투"속에 넣으며, 0 크기 신호가 멀티파트 메시지(공백 구분자 + 데이터) 변환됩니다. 데이터에 신경 쓰지 않고 유효한 신호로 어떤 것도 처리하지 않고 소켓에서 한번 이상 읽지 않는다면 문제가 되지는 않습니다. 그러나 실제 데이터를 전송하기로 결정하면 ROUTER가 갑자기 잘못된 메시지를 제공하는 것을 알게 됩니다. 또한 DEALER는 ROUTER에서 전달된 메시지를 배포하여 PUSH와 동일한 위험(실수로 2개의 수신자를 시작)이 있습니다.

- 송신자에 PUB를 사용하고 수신자에 SUB를 사용할 수 있습니다. 그러면 메시지를 보낸 그대로 정확하게 전달할 수 있으며 PUB는 PUSH 또는 DEALER처럼 배포하지 않습니다. 그러나 빈 구독으로 구독자로 설정해야 하나 귀찮은 일입니다.

- [옮긴이] 빈 구독(empty subscrition)은 구독자가 발행자가 전송하는 모든 메시지를 수신 가능하도록 필터를 두지 않도록 설정합니다.

- 설정 예시 : `zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);`

이러한 이유로 PAIR 패턴은 스레드들 간의 협업을 위한 최상의 선택입니다.

0.27 노드 간의 협업

네트워크상에서 노드들 간의 협업하려 하면, PAIR 소켓을 적용할 수 없습니다. 이것은 스레드들과 노드들 간에 전략이 다른 몇 가지 영역 중 하나입니다. 주로 노드들은 왔다 갔다 하고 스레드들은 보통 정적입니다. PAIR 소켓은 원격 노드가 가고 다시 돌아오면 자동으로 재연결하지 않습니다.

그림 22 - 발행-구독 동기화

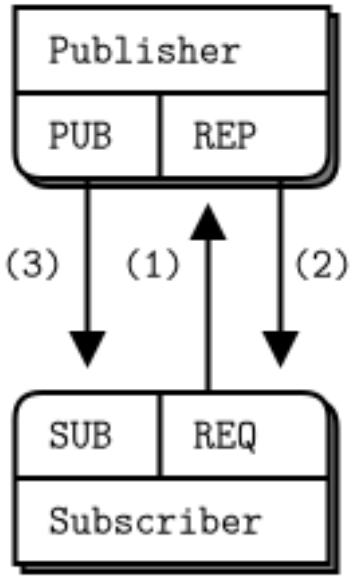


그림 23: Pub-Sub Synchronization

스레드들과 노드들 간의 두 번째 중요한 차이점은 일반적으로 스레드들의 수는 고정되어 있지만 노드들의 수는 가변적입니다. 노드 협업을 위해 이전 시나리오(날씨 서버 및 클라이언트)를 가져와서 구독자 시작 시 발행자에 대한 연결에 소요되는 시간으로 인하여 데이터가 유실하지 않도록 하겠습니다.

다음은 응용프로그램 동작 방식입니다.

- 발행자는 미리 예상되는 구독자들의 숫자를 알고 있으며 이것은 어딘가에서 얻을 수 있는 마법 숫자입니다.

- 발행자가 시작하면서 모든 구독자들이 연결될 때까지 기다립니다. 이것이 노드 협업 부분입니다. 각 구독자들이 구독하기 시작하면 다른 소켓으로 발행자에게 준비가 되었음을 알립니다.
- 발행자는 모든 구독자들이 연결되면 데이터 전송을 시작됩니다.
- [옮긴이] 마법 숫자(magic number)는 이름 없는 숫자 상수(unnamed numerical constants)로 상황에 따라 상수값이 변경(예 : 하드웨어 아키텍처에 따란 INT는 16 비트 혹은 32 비트로 표현)될 수 있을 때 지정합니다. 현실에서는 구독자들의 숫자가 실제 얼마인지 정확히 알기 어렵기 때문에 예제에서는 정해진 구독자의 숫자를 마법 숫자(magic number)로 지칭합니다.

이 경우에 REP-REQ 소켓을 통하여 구독자들과 발행자 간에 동기화하도록 하겠습니다. 아래는 발행자의 소스 코드입니다.

syncpub.c: 동기화된 발행자

- [옮긴이] 원도우 환경에서 테스트 위해 SUBSCRIBERS_EXPECTED을 10에서 2로 변경

```
//  Synchronized publisher

#include "zhelpers.h"
#define SUBSCRIBERS_EXPECTED  2  //  We wait for 2 subscribers

int main (void)
{
    void *context = zmq_ctx_new ();

    //  Socket to talk to clients
    void *publisher = zmq_socket (context, ZMQ_PUB);

    int sndhwm = 1100000;
    zmq_setsockopt (publisher, ZMQ_SNDHWM, &sndhwm, sizeof (int));
```

```
    zmq_bind (publisher, "tcp://*:5561");

    //  Socket to receive signals
    void *syncservice = zmq_socket (context, ZMQ_REP);
    zmq_bind (syncservice, "tcp://*:5562");

    //  Get synchronization from subscribers
    printf ("Waiting for subscribers\n");
    int subscribers = 0;
    while (subscribers < SUBSCRIBERS_EXPECTED) {
        //  - wait for synchronization request
        char *string = s_recv (syncservice);
        free (string);
        //  - send synchronization reply
        s_send (syncservice, "");
        subscribers++;
    }
    //  Now broadcast exactly 1M updates followed by END
    printf ("Broadcasting messages\n");
    int update_nbr;
    for (update_nbr = 0; update_nbr < 1000000; update_nbr++)
        s_send (publisher, "Rhubarb");

    s_send (publisher, "END");

    zmq_close (publisher);
    zmq_close (syncservice);
    zmq_ctx_destroy (context);
```

```
    return 0;
}
```

아래는 구독자 소스 코드입니다.

syncsub.c: 동기화된 구독자

```
//  Synchronized subscriber

#include "zhelpers.h"
#ifndef _WIN32
#include <unistd.h>
#endif

int main (void)
{
    void *context = zmq_ctx_new ();

    //  First, connect our subscriber socket
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5561");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);

    //  ØMQ is so fast, we need to wait a while...
    s_sleep (1000);

    //  Second, synchronize with publisher
    void *syncclient = zmq_socket (context, ZMQ_REQ);
    zmq_connect (syncclient, "tcp://localhost:5562");

    //  - send a synchronization request
```

```
    s_send (syncclient, "");

    //  - wait for synchronization reply
    char *string = s_recv (syncclient);
    free (string);

    //  Third, get our updates and report how many we got
    int update_nbr = 0;
    while (1) {
        char *string = s_recv (subscriber);
        if (strcmp (string, "END") == 0) {
            free (string);
            break;
        }
        free (string);
        update_nbr++;
    }
    printf ("Received %d updates\n", update_nbr);

    zmq_close (subscriber);
    zmq_close (syncclient);
    zmq_ctx_destroy (context);
    return 0;
}
```

아래 배시 쉘 스크립터는 10개의 먼저 구독자들을 구동하고 발행자를 구동합니다.

```
echo "Starting subscribers..."
for ((a=0; a<10; a++)); do
    syncsub &
done
```

```
echo "Starting publisher..."
syncpub
```

만족스러운 결과를 보여 줍니다.

```
Starting subscribers...
Starting publisher...
Received 1000000 updates
Received 1000000 updates
...
Received 1000000 updates
Received 1000000 updates
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc syncpub.c libzmq.lib
 cl -EHsc syncsub.c libzmq.lib

 ./syncpub
Waiting for subscribers
Broadcasting messages

 ./syncsub
Received 1000000 updates

 ./syncsub
Received 1000000 updates
```

'syncsub'에서 REQ/REP 통신가 완료될 때까지 SUB 연결이 완료되지 않는다고 하면 inproc 이외의 전송을 사용하는 경우 아웃바운드 연결이 어떤 순서로든 완료된다는 보장은

없습니다. 따라서 예제에서는 'syncsub'에서 SUB 소켓 연결하고 REQ/REP 동기화 전송 사이에 1초의 대기를 수행하였습니다.

보다 강력한 모델은 다음과 같습니다.

- 발행자가 PUB 소켓을 열고 Hello메시지(데이터 아님)를 보내기 시작합니다.
- 구독자는 SUB 소켓을 연결하고 Hello 메시지를 받으면 REQ/REP 소켓 쌍을 통해 발행자에게 알려줍니다.
- 발행자가 모든 필요한 확인 완료하며, 실제 데이터를 보내기 시작합니다.

0.28 제로 복사

ØMQ 메시지 API는 응용프로그램 버퍼의 데이터를 복사하지 않고 직접 메시지를 송/수신하게 합니다. 우리는 이것을 제로 복사라고 부르며 일부 응용프로그램의 성능을 향상할 수 있습니다.

제로 복사는 높은 빈도로 큰 메모리 공간(수천 바이트) 전송하는 특정한 경우 사용할 수 있습니다. 짧은 메시지 또는 낮은 빈도에서 제로 복사를 사용하면 별다른 효과 없이 코드를 더 복잡하고 엉망으로 만들 수 있습니다. 모든 최적화처럼 도움이 될 거라면 사용하고 전/후의 성능을 측정하십시오.

제로 복사(zero-copy)를 수행하려면 zmq_msg_init_data()를 사용하여 malloc() 또는 다른 할당자에 이미 할당된 데이터 블록을 참조하는 메시지를 생성하여 zmq_msg_send()을 통해 전달합니다. 메시지를 생성할 때, ØMQ는 메시지 전송이 완료되면 데이터 블록을 해제하기 위해 호출하는 기능도 전달합니다. 단순한 예로, 힙에 할당된 버퍼가 1,000 바이트로 가정합니다.

```
void my_free (void *data, void *hint) {
    free (data);
}
// Send message from buffer, which we allocate and ØMQ will free for us
zmq_msg_t message;
zmq_msg_init_data (&message, buffer, 1000, my_free, NULL);
```

```
zmq_msg_send (&message, socket, 0);
```

zmq_msg_send()로 메시지를 송신하고 zmq_msg_close()를 호출하지 않는 점에 주의하십시오. libzmq 라이브러리는 실제 메시지를 송신하고 자동으로 작업(메모리 해제)을 수행합니다.

수신에는 제로 복사를 수행할 방법은 없습니다 : ØMQ는 원하는 기간 동안 저장할 수 있는 버퍼를 제공하지만 데이터를 응용프로그램 버퍼에 직접 쓰지는 않습니다.

데이터를 쓸 때, ØMQ의 멀티파트 메시지는 제로 복사와 함께 잘 작동합니다. 전통적인 메시징에서는 전송할 하나의 버퍼와 함께 다른 버퍼 마살링해야 합니다. 이것은 데이터 복사를 의미합니다. ØMQ에서는 서로 다른 소스에서 여러 개의 버퍼를 개별 메시지 프레임으로 보낼 수 있습니다. 각 필드를 길이가 구분된 프레임으로 보냅니다. 응용프로그램에서는 일련의 송/수신 호출처럼 보입니다. 그러나 내부적으로 다중 프레임이 네트워크에 쓰이고 단일 시스템 호출로 다시 읽히므로 매우 효율적입니다.

- [옮긴이] 마살링(marshalling or marshaling)은 객체의 메모리상 표현을 다른 데이터 형태로 변경하는 절차입니다. 컴퓨터상에서 응용프로그램들 간의 데이터 통신 수행시 서로 다른 포맷에 대하여 전환하는 작업이 필요하며, 직렬화(Serialization)와 유사합니다.

0.29 발행-구독 메시지 봉투

발행-구독 패턴에서, 키를 개별 메시지 프레임으로 분할하고 봉투라고 했습니다. 발행-구독 봉투를 사용하기 위해서는 직접 만들어야 합니다. 선택 사항으로 이전 발행-구독 예제에서는 이런 작업을 수행하지 않았습니다. 발행-구독 봉투를 이용하려면 약간의 코드를 추가해야 합니다. 키와 데이터를 별도의 메시지 프레임에 나누어서 있어 실제 코드를 보면 바로 이해할 수 있습니다.

그림 23 - 개별 키를 가진 발행-구독 봉투

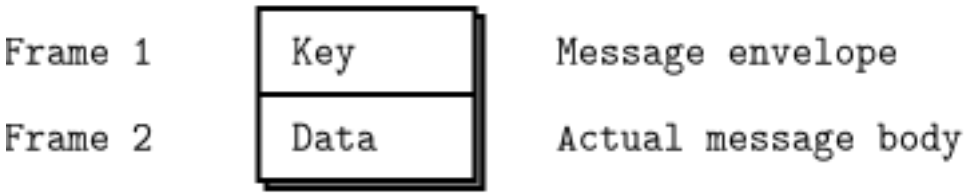


그림 24: Pub-Sub Envelope with Separate Key

구독자에서 구독시 필터를 사용하여 접두사 일치 수행하는 것을 기억하십시오. 즉, "XYZ로 시작하는 모든 메시지"를 찾아야 합니다. 분명한 질문으로 실수로 접두사 일치가 데이터와 일치하지 않도록 데이터에서 키를 구분하는 방법입니다. 가장 좋은 방법은 봉투를 사용하여 프레임 경계를 넘지 않으므로 일치 여부 확인하는 것입니다. 최소 예제로 발행-구독 봉투가 코드로 구현하였으며 발행자는 A와 B의 두 가지 유형의 메시지를 보냅니다.

봉투에는 메시지 유형이 있습니다.

psenvpub.c: Pub-Sub 봉투 발행자(enveloper publisher)

```
//  Pubsub envelope publisher
//  Note that the zhelpers.h file also provides s_sendmore

#include "zhelpers.h"
#ifndef _WIN32
#include <unistd.h>
#endif

int main (void)
{
    //  Prepare our context and publisher
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5563");
```

```
    while (1) {
        //  Write two messages, each with an envelope and content
        s_sendmore (publisher, "A");
        s_send (publisher, "We don't want to see this");
        s_sendmore (publisher, "B");
        s_send (publisher, "We would like to see this");
        s_sleep (1000);
    }
    //  We never get here, but clean up anyhow
    zmq_close (publisher);
    zmq_ctx_destroy (context);
    return 0;
}
```

구독자(subscriber)는 B 유형의 메시지만 원합니다.

psenvsub.c: Pub-Sub 봉투 구독자(enveloper subscriber)

```
//  Pubsub envelope subscriber

#include "zhelpers.h"

int main (void)
{
    //  Prepare our context and subscriber
    void *context = zmq_ctx_new ();
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5563");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "B", 1);
```

```
    while (1) {
        //  Read envelope with address
        char *address = s_recv (subscriber);
        //  Read message contents
        char *contents = s_recv (subscriber);
        printf ("[%s] %s\n", address, contents);
        free (address);
        free (contents);
    }
    //  We never get here, but clean up anyhow
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

2개의 프로그램을 실행하면 구독자(subscriber)는 아래와 같이 보입니다.

```
[B] We would like to see this
[B] We would like to see this
[B] We would like to see this
...
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc psenvpub.c libzmq.lib
 cl -EHsc psenvsub.c libzmq.lib

 ./psenvpub

 ./psenvsub
[B] We would like to see this
```

```
[B] We would like to see this
[B] We would like to see this
...
```

- [옮긴이] 다중 필터(multiple filters)를 아래와 같이 수행하면 “A’,”B” 2개 중에 한 개만 있어도 수신 가능함.

```
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "B", 1);
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "A", 1);
```

예제에서 구독 필터가 전체 멀티 파트 메시지(키와 데이터)를 거부하거나 수락함을 보여줍니다. 멀티파트 메시지의 일부만 받지 않습니다. 여러 발행자들을 구독하고 다른 소켓을 통해 데이터를 보낼 수 있도록 주소를 알고 싶다면 (이것은 일반적인 사용 사례임) 3 부분으로 된 메시지(key-address-data)를 만드십시오.

그림 24 - 서버 주소(Address) 가진 발행-구독 봉투

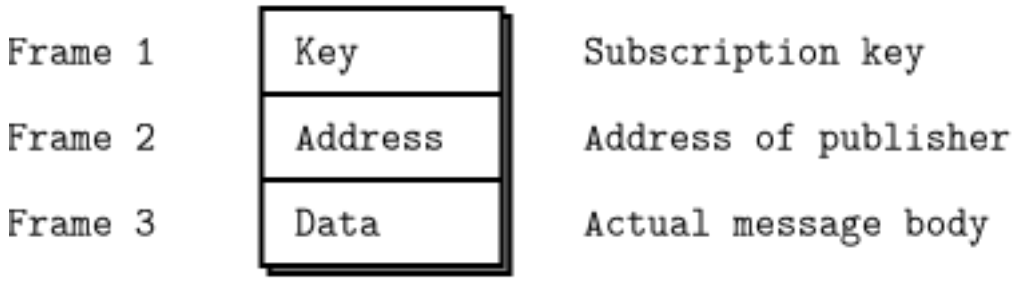


그림 25: Pub-Sub Envelope with Sender Address

0.30 최고수위 표시

프로세스에서 프로세스로 메시지를 빠르게 보낼 때에 곧 메모리가 쉽게 채워질 수 있다는 것을 알게 됩니다. 프로그램 상의 문제에 대하여 이해하고 예방 조치를 하지 않아 메세지 수신 프로세스의 어딘가에서 몇 초의 지연이 발생하면 메시지 데이터는 메모리 상의 백로그 전환되어 서버의 자원을 고갈시킬 수 있습니다.

문제는 다음과 같습니다 : 프로세스 A가 높은 빈도로 프로세스 B로 메시지를 전송하고 프로세스 B에서 메시지를 처리하고 있다고 가정하면, 갑자기 B 프로세스가 바쁜 상태(가비지 수집, CPU 과부하 등)가 되어 잠시 메시지를 처리할 수 없습니다. 많은 가비지 수집(역주 : java 가상 머신과 같이)의 경우 몇 초가 소요되거나 더 심각한 문제가 있는 경우 처리 시간이 지연될 수 있습니다. - 프로세스 A가 여전히 미친 듯이 메시지를 보내면 메시지는 어떻게 될까요? - 일부는 B의 네트워크 버퍼(buffer)에 있습니다. - 일부는 이더넷 네트워크 자체에 있으며 - 일부는 프로세스 A의 네트워크 버퍼에 있습니다. - 나머지는 나중에 신속하게 재전송할 수 있도록 프로세스 A의 메모리에 남아 있습니다. - 사전 예방을 하지 않을 경우 프로세스 A는 메모리가 고갈되어 오류가 발생할 수 있습니다.

이것은 메시지 브로커의 일관되고 고전적인 문제입니다. 더 아프게 만드는 것은 프로세스 B 오류이며 피상적이며 B는 일반적으로 A가 제어할 수 없는 사용자 작성 응용프로그램입니다.

해결 방법의 하나는 문제를 메시지를 계속 전달하는 상류에 알리는 것입니다. 프로세스 A는 다른 곳에서 메시지를 받고 있으며 "Stop!"이라고 전달됩니다. 이것을 흐름 제어라고 합니다. 타당해 보이지만 트위터 피드에 "Stop!"을 보내면 어떨까요? B 정상화될 때까지 전 세계에서 트윗을 중단하라고 지시합니까?

흐름 제어는 경우에 따라 작동하거나 하지 않을 수 있습니다. 전송계층은 응용프로그램 계층에 "stop"하도록 지시할 수 없습니다. 마치 지하철 시스템이 연계된 많은 계열사에게 "너무 바쁘니 직원을 30분 더 근무하도록 하십시오"하는 것처럼 할 수는 없습니다. 메시징의 해결책은 버퍼 크기에 제한을 두어 전송되는 메시지가 버퍼 제한에 도달하면 적절한 조치를 취하는 것입니다. 경우에 따라 메시지를 버리는 것이 해결책이 될 수도 있으며 다른 경우는 대기하는 것이 최선의 전략입니다.

ØMQ는 HWM(최고수위 표시) 개념을 사용하여 내부 파이프의 용량을 정의합니다. 내/외부 소켓에 대한 각 연결에서 소켓 유형에 따라 자체 파이프 및 송/수신을 위한 HWM이 있습니다. 일부 소켓(PUB, PUSH)은 송신 버퍼만 있으며 일부 소켓(SUB, PULL, REQ, REP)은 수신 버퍼만 있습니다. 일부 소켓(DEALER, ROUTER, PAIR)은 송신 및 수신 버퍼가 모두 있습니다.

ØMQ v2.x에서는 HWM은 기본적으로 제한이 없어 대량 데이터를 발행하는 발행자에는 치명적인 문제였습니다(옮긴이 : PUB 소켓 사용으로 메모리 버퍼 초과). ØMQ v3.x부터는 기본적으로 HWM을 1,000으로 설정되어 있습니다. 여전히 ØMQ v2.x를 사용하는 경우

메시지 크기 및 예상 구독자 성능을 고려하여 ØMQ v3.x과 일치하도록 HWM을 1,000으로 설정하거나 다른 수치로 조정해야 합니다.

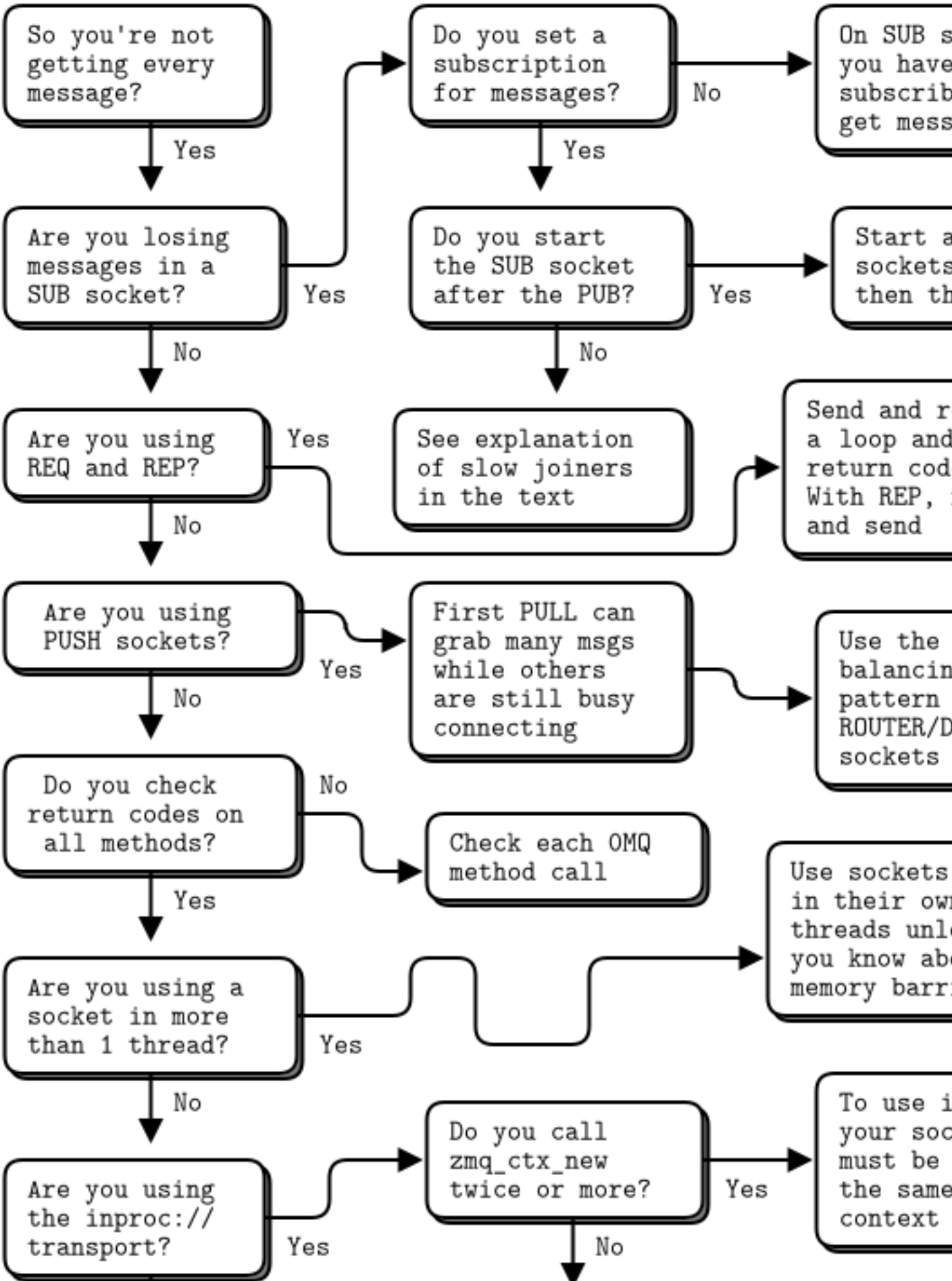
소켓이 HWM에 도달하면 소켓 유형에 따라 데이터를 차단하거나 삭제합니다. PUB 및 ROUTER 소켓은 HWM에 도달하면 데이터를 삭제하지만 다른 소켓 유형은 차단합니다. inproc 전송방식에서 송신자와 수신자는 동일한 버퍼를 공유하므로 실제 HWM은 양쪽에서 설정한 HWM의 합입니다.

마지막으로, HWM은 정확하지 않습니다. 기본적으로 최대 1,000개의 메시지를 받을 수 있지만 libzmq가 대기열을 구현하는 방식으로 인해 실제 버퍼 크기가 훨씬 작을(절반까지) 수 있습니다.

0.31 메시지 분실 문제 해결 방법

ØMQ를 사용하여 응용프로그램을 구축하면 “수신할 메시지가 손실되었습니다”와 같은 문제가 한번 이상 발생합니다 : 가장 일반적인 원인을 설명하는 다이어그램을 작성했습니다.

그림 25 - 메시지 분실 문제 해결 방법


So you're not getting every message?
Yes
Are you losing messages in a SUB socket?
Yes
No
Are you using REQ and REP?
Yes
No
Are you using PUSH sockets?
Yes
No
Do you check return codes on all methods?
No
Yes
Are you using a socket in more than 1 thread?
Yes
No
Are you using the inproc:// transport?
Yes
Do you set a subscription for messages?
No
Yes
Do you start the SUB socket after the PUB?
Yes
No
See explanation of slow joiners in the text
First PULL can grab many msgs while others are still busy connecting
Check each OMQ method call
Do you call zmq_ctx_new twice or more?
Yes
No
On SUB s
you have
subscrib
get mess
Start a
sockets
then th
Send and r
a loop and
return cod
With REP,
and send
Use the
balancin
pattern
ROUTER/D
sockets
Use sockets
in their ow
threads unl
you know ab
memory barr
To use i
your soc
must be
the same
context

도표가 이야기하는 바는 다음과 같습니다.

- SUB 소켓에서 ZMQ_SUBSCRIBE로 zmq_setsockopt()를 사용하여 구독을 설정하십시오. 그렇지 않으면 메시지를 받을 수 없습니다. 접두사로 메시지를 구독하기 때문에 필터에 “”(빈 구독)를 구독하면 모든 메시지를 받을 수 있습니다.
- PUB 소켓이 데이터 전송을 시작한 후 SUB 소켓을 시작하면(즉, PUB 소켓에 연결 설정) SUB 소켓 연결 전의 발행된 데이터를 잃게 됩니다. 이것이 문제이면 아키텍처를 수정하여 SUB 소켓이 먼저 시작한 후에 PUB 소켓이 발행하도록 하십시오
- SUB 및 PUB 소켓을 동기화하더라도 메시지가 손실될 수 있습니다. 이것은 실제로 연결이 생성될 때까지 내부 대기열이 생성되지 않기 때문입니다. 바인드/연결 방향을 바꾸어, SUB 소켓이 바인딩하고 PUB 소켓이 연결되면 기대한 것처럼 동작할 수 있습니다.
- REP 및 REQ 소켓을 사용하고 동기화된 송신/수신/송신/수신 순서를 지키지 않으면, ØMQ가 오류를 보고하지만 무시할 경우가 있습니다. 그러면 메시지를 잃어버린 것과 같습니다. REQ 또는 REP를 사용하는 경우 송신/수신 순서를 지켜야 하며 항상 실제 코드에서는 ØMQ 호출에서 오류가 있는지 확인하십시오.
- PUSH 소켓을 사용한다면, 연결할 첫 번째 PULL 소켓은 불공정하게 분배된 메시지들을 가지게 됩니다. 메시지의 정확한 순서은 모든 PULL 소켓이 성공적으로 연결된 경우에 발생하며 다소(몇 밀리초) 시간이 걸립니다. PUSH/PULL 소켓의 대안으로 낮은 데이터 빈도를 위해서는 ROUTER/DEALER 및 부하 분산 패턴을 고려하시기 바랍니다.
- 스레드 간에 소켓을 공유하지 마십시오. 무작위적인 이상 현상을 초래하고 충돌합니다.
- inproc을 사용하는 경우 두 소켓이 동일한 컨텍스트에 있는지 확인하십시오. 그렇지 않으면 연결 측이 실제로 실패합니다. 또한 먼저 바인딩한 다음 연결하십시오. inproc은 tcp와 같은 연결이 끊긴 전송방식이 아닙니다.
- ROUTER 소켓을 사용하는 경우, 우연히 잘못된 인식자(ID) 프레임을 보내어(또는 인식자(ID) 프레임을 보내지 않음) 메시지를 잃어버리기가 쉽습니다. 일반적으로 ROUTER 소켓에서 ZMQ_ROUTER_MANDATORY 옵션을 설정하는 것이 좋지만 모든 송신 호출에서 반환값을 확인해야 합니다.
- 마지막으로, 무엇이 잘못되었는지 알 수 없다면 문제를 재현하는 최소한의 테스트 사례를 만들어 문제가 발생하는지 테스트 수행하시기 바라며 ØMQ 커뮤니티에 도움을

요청하십시오.

- [옮긴이] `zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0)` 설정 시 모든 메시지 구독합니다.

3장-고급 요청-응답 패턴

2장-소켓 및 패턴에서 우리는 ØMQ의 새로운 측면에 대하여 일련의 작은 응용프로그램을 개발하여 ØMQ 기본적인 사용 방법을 살펴보았습니다. 3장에서는 ØMQ의 핵심 요청-응답 패턴 위에 만들어진 고급 패턴을 알아보며 작은 응용프로그램을 개발해 보겠습니다.

다루는 내용은 다음과 같습니다.

- 요청-응답 메커니즘의 작동 방법
- REQ, REP, DEALER 및 ROUTER 소켓 조합 방법
- 상세한 ROUTER 소켓 작동 방법
- 부하 분산 패턴
- 간단한 부하 분산 메시지 브로커 구축
- ØMQ를 위한 고급 API 설계
- 비동기 요청-응답 서버 구축
- 상세한 브로커 간 라우팅 예제

0.32 요청-응답 메커니즘

이미 멀티파트 메시지를 간략히 알아보았습니다. 이제 주요 사용 사례인 응답 메시지 봉투를 알아보겠습니다. 봉투는 데이터 자체를 건드리지 않고 하나의 주소로 데이터를 안전하게 포장하는 방법입니다. 봉투에서 응답 주소를 분리하여 메시지 내용 또는 구조에 관계없이

주소를 생성, 읽기 및 제거하는 API 및 프록시와 같은 범용 중개자를 작성할 수 있습니다.

- [옮긴이] 멀티파트 메시지는 여러 개의 프레임들로 하나의 메시지를 구성합니다.

요청-응답 패턴에서 봉투는 응답을 위한 반송 주소를 가지고 있습니다. 이것은 상태가 없는 ØMQ 네트워크가 어떻게 왕복 요청-응답 대화를 수행하는 방법입니다.

REQ 및 REP 소켓을 사용하면 봉투들을 볼 수 조차 없습니다. REQ/REP 소켓들은 자동으로 봉투를 처리하지만 대부분의 흥미로운 요청-응답 패턴의 경우 특히 ROUTER 소켓에서 봉투를 이해하고 싶을 것입니다. 우리는 단계적으로 작업해 나가겠습니다.

0.32.1 간단한 응답 봉투

요청-응답 교환은 요청 메시지와 이에 따른 응답 메시지로 구성됩니다. 간단한 요청-응답 패턴에는 각 요청에 대해 하나의 응답이 있습니다. 고급 패턴에서는 요청과 응답이 비동기적으로 동작할 수 있지만 응답 봉투는 항상 동일한 방식으로 동작합니다.

ØMQ 응답 봉투는 공식적으로 0개 이상의 회신 주소, 공백 구분자, 메시지 본문(0개 이상의 프레임들)으로 구성됩니다. 봉투는 순차적으로 여러 소켓에 의해 생성되어 함께 동작합니다. 우리는 이것을 구체적으로 살펴보겠습니다.

REQ 소켓을 통해 "Hello"를 보내는 것으로 시작하겠습니다. REQ 소켓은 주소가 없고 공백 구분자 프레임과 "Hello" 문자열이 포함된 메시지 프레임만 있는 가장 간단한 응답 봉투를 생성합니다. 이것은 2개의 프레임으로 구성된 메시지입니다.

그림 26 - 최소 봉투 요청(Request with Minimal Envelope)

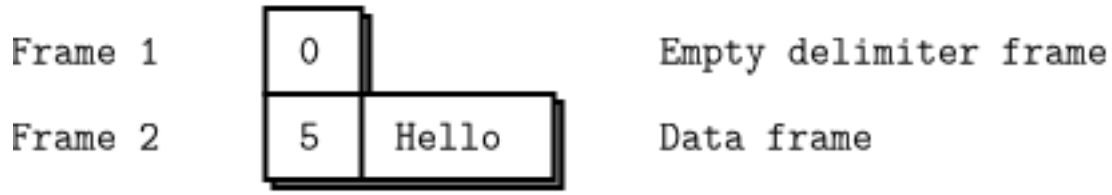


그림 27: 최소 봉투 요청

REP 소켓은 일치하는 작업을 수행합니다. 봉투에서 공백 구분자 프레임 제거하고 전체 봉투를 저장한 후 "Hello" 문자열을 응용프로그램에 전달합니다. 따라서 우리의 원래 Hello

World 예제는 내부적으로 요청-응답 봉투를 처리하여 응용프로그램에서는 이를 보지 못했습니다.

hwclient와 hwserver 사이에 흐르는 네트워크 데이터를 감시한다면, 보게 될 것은 다음과 같습니다. : 모든 요청과 모든 응답은 사실 두 개의 프레임으로 “하나의 공백 프레임”과 “본문” 입니다. 간단한 REQ-REP 대화에는 별 의미가 없는 것 같지만 ROUTER와 DEALER가 봉투를 다루는 방법을 살펴보면 그 이유를 알 수 있습니다.

- [옮긴이]
- 요청시 : hwclient-[“Hello”]->REQ->[“”+“Hello”]->REP->[“Hello”]->hwserver
- 은답시 : hwserver-[“World”]->REP-[“”+“World”]->REQ-[“World”]->hwclient

0.32.2 확장된 응답 봉투

이제 REQ-REP 쌍 사이에 ROUTER-DEALER 프록시 두어 확장하고 이것이 응답 봉투에 어떤 영향을 주는지 알아보겠습니다. 이것은 확장된 요청-응답 패턴으로 “2장-소켓 및 패턴”에서 보았습니다. 실제로 프록시 단계를 원하는 만큼 삽입할 수 있습니다. 동작 방식은 동일합니다.

그림 27 - 확장된 요청-응답 패턴

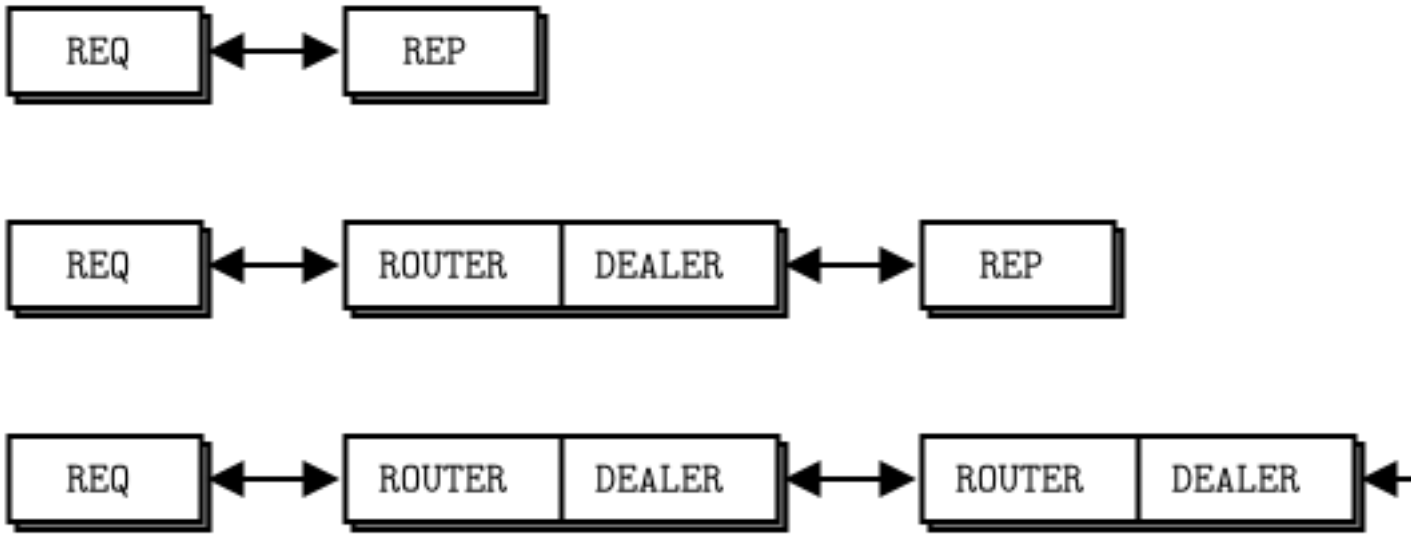


그림 28: Extended Request-Reply Pattern

가상 코드에서 프록시는 아래와 같이 동작합니다.

```
prepare context, frontend and backend sockets
while true:
    poll on both sockets
    if frontend had input:
        read all frames from frontend
        send to backend
    if backend had input:
        read all frames from backend
        send to frontend
```

ROUTER 소켓은 다른 소켓과 달리 모든 연결을 추적하고 호출자에게 이에 대해 알려줍니다. 호출자에게 알리는 방법은 수신된 각 메시지 선두에 연결 식별자(ID)를 붙이는 것입니다. 때때로 주소라고도 할 수 있는 식별자(ID)는 "이것은 연결에 대한 고유한 핸들"이란 의미를

가진 이진 문자열(binary string)입니다. 그래서 ROUTER 소켓을 통해 메시지를 보낼 때 먼저 식별자(ID) 프레임을 보냅니다.

'zmq_socket()' 매뉴얼에서는 이와 같이 설명합니다.

메시지를 수신할 때 ZMQ_ROUTER 소켓은 메시지를 응용프로그램에 전달하기 전에 발신 상대의 식별자(ID)를 포함하는 메시지 부분을 메시지 선두에 추가해야 합니다. 수신된 메시지는 연결된 모든 상대 사이의 공정 대기열에 놓입니다. ZMQ_ROUTER 소켓에서 메시지를 보낼 때 메시지의 첫 번째 부분을 제거하고 이를 사용하여 메시지가 라우팅 될 상대의 식별자(ID)가 되게 합니다.

지난 이야기지만, ØMQ v2.2 이전 버전의 식별자(ID)로 UUID를 사용하였으며, ØMQ 3.0 이후부터는 짧은 정수를 사용하고 있습니다. 이런 변화는 네트워크 성능을 개선하는 영향을 주지만, 다중 프록시 홉(hops)을 사용하는 경우는 영향은 미미한 것입니다. 주목할만한 영향은 libzmq에서 UUID 생성에 필요한 라이브러리에 의존성을 제거한 것입니다.

- [옮긴이] UUID(universally unique identifier)은 네트워크 상에서 서로 모르는 개체들을 식별하고 구별하기 위해서는 각각의 고유한 식별자로 32개의 십육진수(4bit)인 128 bits의 수로 표현됩니다(예 : 550e8400-e29b-41d4-a716-446655440000).

식별자들(IDs)은 이해하기 어려운 개념이지만 ØMQ 전문가가 되고 싶다면 필수입니다. ROUTER 소켓은 작동하는 각 연결에 대해 임의의 식별자(ID)를 만듭니다. ROUTER 소켓에 3개의 REQ 소켓이 연결되어 있는 경우 각각의 REQ 소켓에 대해 하나씩 3개의 임의의 ID를 생성합니다.

동작 가능한 예제로 계속하면 REQ 소켓에 3 바이트 식별자(ID) ABC가 있다고 하면 내부적으로 ROUTER 소켓이 ABC를 검색할 수 있는 해쉬 테이블 가지고 REQ 소켓에 대한 TCP 연결을 찾을 수 있게 합니다.

ROUTER 소켓에서 메시지를 받으면 3개의 프레임을 얻습니다.

그림 28 - 주소가 있는 요청(Request with One Address)

그림 29: Request with One Address

프록시 루프의 핵심은 "한 소켓에서 읽고 다른 소켓에 쓰기"이므로, 문자 그대로 3개의 프레임을 DEALER 소켓으로 보냅니다. 네트워크 트래픽을 엿보기한다면 3개의 프레임이 DEALER 소켓에서 REP 소켓으로 이동하는 것을 볼 수 있습니다. REP 소켓은 이전과 마찬가지로 새 응답 주소를 포함하여 전체 봉투를 제거하고 다시 한번 호출자에게 "Hello"를 전달합니다.

부수적으로 REP 소켓은 한 번에 하나의 요청-응답만 처리할 수 있으므로 엄격한 송/수신 주기를 지키지 않고 여러 개의 요청을 읽어 여러 개의 응답을 보내려고 하면 오류가 발생하는 원인입니다.

이제 반환 경로를 시각화할 수 있습니다. `hwserver`가 "World"를 다시 보내면 REP 소켓은 저장한 봉투에 감싸고 네트워크를 통해 DEALER 소켓으로 3개 프레임(ID + empty delimiter + body) 응답 메시지를 보냅니다.

그림 29 - 주소가 있는 응답

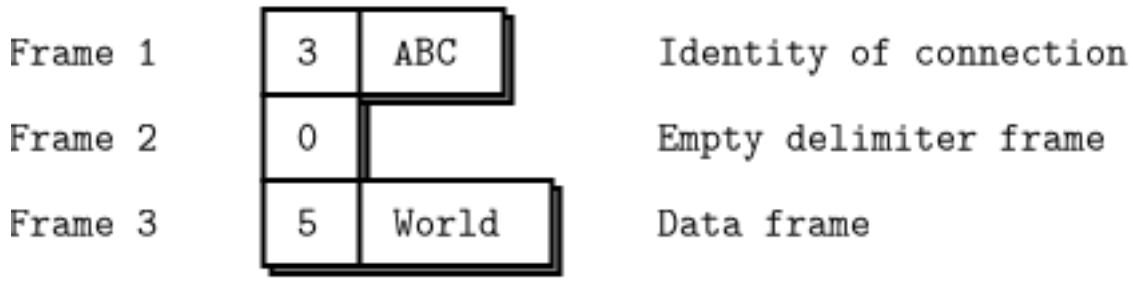


그림 30: Reply with One Address

DEALER는 3개의 프레임을 읽고 3개의 모든 프레임을 ROUTER 소켓에 전달됩니다. ROUTER는 첫 번째 메시지 프레임을 읽고 ABC라는 ID에 해당하는 연결을 찾습니다. 연결을 찾으면 나머지 2개 프레임(empty delimiter + body)을 네트워크에 보냅니다.

그림 30 - 최소 응답 봉투(Reply with Minimal Envelope)

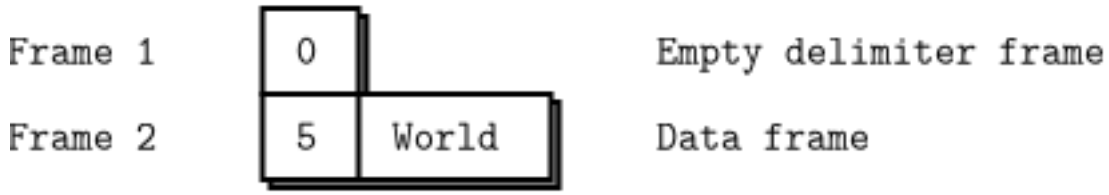


그림 31: Reply with Minimal Envelope

REQ 소켓은 전달된 메시지를 받아 첫 번째 프레임이 공백 구분자인지 확인하고 맞으면 REQ 소켓은 공백 구분자 프레임을 버리고 “World”를 호출한 응용프로그램에 전달합니다. 그러면 ØMQ를 시작했을 때의 놀라움으로 응용프로그램에서 “World”가 출력됩니다.

- [옮긴이] 송/수신시에 프레임 구성
- 송신 : APP -[“Hello”]-> REQ -[“”+“Hello”]-> ROUTER -[ID+“”+“Hello”]-> DEALER -[ID+“”+“Hello”]-> REP -[“Hello”]-> APP
- 수신 : APP -[“World”]-> REP -[ID+“”+“World”]-> DEALER -[ID+“”+“World”]-> ROUTER -[“”+“World”]-> REQ -[“World”]-> APP
- [옮긴이] 송/수신 시에 프레임 전달에 대한 테스트를 위하여 “test_frame.c”을 작성하였으며 czmq 라이브러리를 사용합니다.
- test_frame.c Hello World 예제 프로그램의 프레임 흐름

```
//  Multithreaded Hello World server
#include "czmq.h"
#define NBR_THREADS 1
static void *
```

```
worker_routine (zctx_t *ctx) {
    //  Socket to talk to dispatcher
    void *receiver = zsocket_new (ctx, ZMQ_REP);
    zsocket_connect (receiver, "inproc://workers");
    while (!zctx_interrupted) {
        zmsg_t *request = zmsg_recv (receiver);
        if(!request)
            break; // interrupted
        printf ("[REP-APP]Received request: \n");
        zmsg_dump (request);
        zmsg_destroy (&request);
        //  Do some 'work'
        zclock_sleep (1000);
        //  Send reply back to client
        zmsg_t *reply = zmsg_new ();
        zmsg_addstr (reply, "World");
        zmsg_send (&reply, receiver);
        zmsg_destroy (&reply);
    }
    zsocket_destroy(ctx, receiver);
    return NULL;
}

static void *
client_routine (zctx_t *ctx) {
    //  Socket to talk to dispatcher
    void *sender = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (sender, "inproc://clients");
    while (!zctx_interrupted) {
```

```
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "Hello");
        //  Send request to worker
        zmsg_send (&request, sender);
        zmsg_destroy (&request);
        zmsg_t *reply = zmsg_recv (sender);
        printf ("[REQ-APP]Received reply: \n");
        zmsg_dump (reply);
        zmsg_destroy (&reply);
    }
    zsocket_destroy (ctx, sender);
    return NULL;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    //  Socket to talk to clients
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "inproc://clients");
    //  Socket to talk to workers
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://workers");

    int thread_nbr;
    //  Launch pool of client threads
    for (thread_nbr = 0; thread_nbr < NBR_THREADS; thread_nbr++) {
        zthread_new (client_routine, ctx);
    }
```

```
    //  Launch pool of worker threads
     for (thread_nbr = 0; thread_nbr < NBR_THREADS; thread_nbr++) {
         zthread_new (worker_routine, ctx);
     }
      //  Initialize poll set
     zmq_pollitem_t items [] = {
         { frontend, 0, ZMQ_POLLIN, 0 },
         { backend,  0, ZMQ_POLLIN, 0 }
     };
     //  Connect work threads to client threads via a queue proxy
     while (!zctx_interrupted) {
         zmq_poll (items, 2, -1);
         if (items [0].revents & ZMQ_POLLIN) {
             //  Process all parts of the message
             zmsg_t *message = zmsg_recv (frontend);
             if(!message)
                 break; // interrupted
             printf ("[REQ-ROUTER]Received message: \n");
             zmsg_dump (message);
             zmsg_send (&message, backend);
             zmsg_destroy (&message);
         }
         if (items [1].revents & ZMQ_POLLIN) {
             zmsg_t *message = zmsg_recv (backend);
             if(!message)
                 break; // interrupted
             printf ("[REP-DEALER]Received message: \n");
             zmsg_dump (message);
             zmsg_send (&message, frontend);
```

```
            zmsg_destroy (&message);
        }
    }
    zsocket_destroy (ctx, backend);
    zsocket_destroy (ctx, frontend);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
S D:\git_store\zguide-kr\examples\C> cl -EHsc test_frame.c libzmq.lib czmq.lib
 ./test_frame
[REQ-ROUTER]Received message:
D: 20-09-04 16:20:48 [005] 0080000029     --> ID
D: 20-09-04 16:20:48 [000]                --> 공백 구분자
D: 20-09-04 16:20:48 [005] Hello          --> 데이터
[REP-APP]Received request:
D: 20-09-04 16:20:48 [005] Hello          --> 데이터
[REP-DEALER]Received message:
D: 20-09-04 16:20:49 [005] 0080000029     --> ID
D: 20-09-04 16:20:49 [000]                --> 공백 구분자
D: 20-09-04 16:20:49 [005] World          --> 데이터
[REQ-APP]Received reply:
D: 20-09-04 16:20:49 [005] World          --> 데이터
...
```

0.32.3 이것이 좋은 이유는?

솔직하게 엄격한 요청-응답 또는 확장된 요청-응답을 사용하는 경우는 다소 제한적입니다. 첫째, 오류가 있는 응용프로그램 코드로 인한 서버 실패와 같은 일반적인 오류로부터 쉽게

복구할 수 있는 방법이 없습니다. “4장-신뢰할 수 있는 요청-응답 패턴”에서 자세히 다루겠습니다. 하지만 4개의 소켓(REQ-ROUTER-DEALER-REP)이 봉투를 처리하는 방식과 서로 대화하는 방식을 파악하면 매우 유용한 작업을 수행할 수 있습니다. ROUTER가 응답 봉투를 사용하여 응답을 다시 라우팅 할 클라이언트 REQ 소켓을 결정하는 것을 보았습니다. 이제 다른 방식으로 표현해 보겠습니다.

- ROUTER가 메시지를 줄 때마다, ROUTER는 식별자(ID)로 어떤 상대로부터 왔는지 알려줍니다.
- 응용프로그램에서 해쉬 테이블로 식별자(ID)를 관리하여 도착된 새로운 상대를 추적할 수 있습니다.
- 메시지의 첫 번째 프레임으로 식별자(ID)를 접두사로 지정하면, ROUTER는 연결된 모든 상대에 비동기로 메시지들을 라우팅합니다.

ROUTER 소켓은 전체 봉투를 관여하지 않으며 공백 구분자에 대해서도 모릅니다. ROUTER 소켓이 관심을 갖는 것은 식별자(ID) 프레임으로 메시지를 보낼 연결(connection)을 아는 것입니다.

0.32.4 요청-응답 소켓 정리

정리하면 다음과 같습니다.

- REQ 소켓은 메시지 데이터 앞에 공백 구분자 프레임을 넣어 네트워크로 보냅니다. REQ 소켓은 동기식으로 REQ 소켓은 항상 하나의 요청을 보내고 하나의 응답을 기다립니다. REQ 소켓은 한 번에 하나의 상대와 통신합니다. REQ 소켓을 여러 상대에 연결하려면, 요청들은 한 번에 하나씩 각 상대에 분산되고 응답이 예상됩니다.
- REP 소켓은 모든 식별자(ID) 프레임과 공백 구분자를 읽고 저장한 다음 프레임을 호출자에게 전달합니다. REP 소켓은 동기식이며 한 번에 하나의 상대와 통신합니다. REP 소켓을 여러 개의 단말들에 연결하면 요청들은 상대로부터 공정한 형태로 읽히고, 응답은 항상 마지막 요청을 한 동일한 상대로 전송됩니다.
- DEALER 소켓은 응답 봉투를 인식하지 못하며 응답 봉투를 멀티파트 메시지처럼 처리합니다. DEALER 소켓은 비동기식이며 PUSH와 PULL이 결합된 것과 같으며 모든 연

결 간에 보내진 메시지(ROUTER->DEALER)를 배포하고 모든 연결에서 받은 메시지(REP->DEALER)를 순서대로 공정 대기열에 보관합니다.

- ROUTER 소켓은 DEALER처럼 응답 봉투를 인식하지 못합니다. ROUTER는 연결에 대한 식별자(ID)를 만들고 수신된 메시지의 첫 번째 프레임으로 생성한 식별자(ID)를 호출자에게 전달합니다. 반대로 호출자가 메시지를 보낼 때 첫 번째 메시지 프레임을 식별자(ID)로 사용하여 보낼 연결을 찾습니다. ROUTER는 비동기로 동작합니다.

0.33 요청-응답 조합

각각 특정 동작을 하는 4개의 요청-응답 소켓들(REQ, ROUTER, DELEAER, REP)이 있으며 그들이 단순하고 확장된 요청-응답 패턴들(request-reply patterns)로 연결되는 방법을 보았습니다. 이러한 소켓은 많은 문제를 해결하기 위해 사용될 수 있는 구성요소로 사용될 것입니다.

규정된 조합은 다음과 같습니다.

- REQ에서 REP
- DEALER에서 REP
- REQ에서 ROUTER
- DEALER에서 ROUTER
- DEALER에서 DEALER
- ROUTER에서 ROUTER

다음 조합은 유효하지 않습니다(이유를 설명하겠습니다).

- REQ에서 REQ
- REQ에서 DEALER
- REP에서 REP
- REP에서 ROUTER

의미를 상기하기 위한 몇 가지 팁들입니다. DEALER는 비동기 REQ 소켓과 같고 ROUTER는 비동기 REP 소켓과 같습니다. REQ 소켓을 사용하는 경우 DEALER를 사용할 수 있으며

봉투에 직접 읽고 쓸 수 있어야 합니다. REP 소켓을 사용하는 곳에 ROUTER를 사용할 수 있으며 식별자(ID) 프레임을 직접 관리해야 합니다.

REQ 및 DEALER 소켓을 "클라이언트"로, REP 및 ROUTER 소켓을 "서버"로 생각하십시오. 대부분 REP 및 ROUTER 소켓을 서버처럼 바인딩하고 REQ 및 DEALER 소켓을 클라이언트처럼 연결하려고 합니다. 항상 이렇게 간단하지는 않지만 시작하기에 깨끗하고 기억에 남을 것입니다.

0.33.1 REQ와 REP 조합

이미 REP 서버와 통신하는 REQ 클라이언트를 다루었지만 한 가지 측면을 살펴보겠습니다. REQ 클라이언트는 메시지 흐름을 시작해야 하며 REP 서버는 처음에 요청을 하지 않은 REQ 클라이언트와 통신할 수 없습니다. 기술적으로도 불가능하며, REP 서버에서 API 호출을 시도하면 EFSM 오류도 반환합니다.

0.33.2 DEALER와 REP 조합

이제 DEALER로 REQ 클라이언트를 바꾸며 여러 REP 서버들과 통신할 수 있는 비동기 클라이언트(asynchronous client)로 제공할 수 있습니다. DEALER를 사용하여 "Hello World" 클라이언트를 다시 작성하면 응답을 기다리지 않고 "Hello" 요청을 얼마든지 보낼 수 있습니다.

DEALER를 사용하여 REP 소켓과 통신할 때, REQ 소켓이 보낸 봉투(empty delimiter + body)를 정확하게 모방하며 그렇지 않으면 REP 소켓이 잘못된 메시지로 버립니다. 따라서 메시지를 보내려면 :

- MORE 플래그가 설정된 빈 메시지 프레임을 보내고
- 메시지 본문을 보냅니다.

그리고 REP 소켓에서 메시지를 받으면

- 첫 번째 프레임을 받고 비어 있지 않으면 전체 메시지를 버립니다.
- 다음 프레임을 수신하고 응용프로그램에 전달합니다.

0.33.3 REQ와 ROUTER 조합

REQ를 DEALER로 대체하는 것과 같이 REP를 ROUTER로 대체할 수 있습니다. 여러 REQ 클라이언트와 동시에 통신할 수 있는 비동기 서버를 제공합니다. ROUTER를 사용하여 "Hello World" 서버를 다시 작성하면 여러 "Hello" 요청을 병렬로 처리할 수 있습니다. 이것을 "2장-소켓 및 패턴"에서 mtserver 예제를 보았습니다.

- [옮긴이] mtserver 예제는 여러 개(예 : 10개)의 hwclient 요청을 ROUTER 소켓(asynchronous server)에서 받아 프록시(zmq_proxy())로 DEALER 소켓으로 전달하면 5개의 worker 스레드들이 받아 응답하는 형태입니다.
- [옮긴이] mtserver 예제에서는 worker에 대하여서만 스레드 구성하였으나, 다음 예제에서는 client도 스레드로 구성하여 테스트하였습니다.
- mtserver_client.c : 다중 서버와 클라이언트 스레드를 통한 테스트

```
//  Multithreaded Hello World server

#include "zhelpers.h"
#include <pthread.h>
#ifndef _WIN32
#include <unistd.h>
#endif

static void *
client_routine (void *context) {
    //  Socket to talk to dispatcher
    void *sender = zmq_socket (context, ZMQ_REQ);
    zmq_connect (sender, "inproc://clients");

    while (1) {
```

```
        s_send (sender, "Hello");
        char *string = s_recv (sender);
        printf ("Received reply: [%s]\n", string);
        free (string);
    }
    zmq_close (sender);
    return NULL;
}

static void *
worker_routine (void *context) {
    //  Socket to talk to dispatcher
    void *receiver = zmq_socket (context, ZMQ_REP);
    zmq_connect (receiver, "inproc://workers");

    while (1) {
        char *string = s_recv (receiver);
        printf ("Received request: [%s]\n", string);
        free (string);
        //  Do some 'work'
        s_sleep (1000);
        //  Send reply back to client
        s_send (receiver, "World");
    }
    zmq_close (receiver);
    return NULL;
}

int main (void)
```

```
{
    void *context = zmq_ctx_new ();

    //  Socket to talk to clients
    void *clients = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (clients, "inproc://clients");

    //  Socket to talk to workers
    void *workers = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (workers, "inproc://workers");

     //  Launch pool of client threads
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
        pthread_t client;
        pthread_create (&client, NULL, client_routine, context);
    }

    //  Launch pool of worker threads
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_routine, context);
    }
    //  Connect work threads to client threads via a queue proxy
    zmq_proxy (clients, workers, NULL);

    //  We never get here, but clean up anyhow
    zmq_close (clients);
    zmq_close (workers);
```

```
    zmq_ctx_destroy (context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc mtserver_client.c libzmq.lib pthreadVC2.lib

 ./mtserver_client
Received request: [Hello]
Received request: [Hello]
Received reply: [World]
Received request: [Hello]
Received reply: [World]
Received request: [Hello]
Received request: [Hello]
Received reply: [World]
Received reply: [World]
...
```

ROUTER 소켓은 두 가지 사용법이 있습니다.

- 프론트엔드와 백엔드 소켓들 간에 메시지를 전환하는 프록시로 사용
- 응용프로그램이 메시지를 읽고 그에 따라 행동하도록 사용

첫 번째 경우 ROUTER는 단순히 인위적인 식별자(ID) 프레임을 포함한 모든 프레임들을 읽고 맹목적으로 전달합니다. 두 번째 경우 ROUTER는 전송된 응답 봉투의 형식을 알아야 합니다. 다른 상대가 REQ 소켓이므로 ROUTER는 식별자 프레임(ID), 공백 프레임(empty delimiter) 및 데이터 프레임(body)을 가져옵니다.

0.33.4 DEALER와 ROUTER 조합

이제 REQ와 REP를 DEALER 및 ROUTER로 전환하여 가장 강력한 소켓 조합을 얻을 수 있습니다. DEALER는 ROUTER와 통신합니다. 이것은 비동기 클라이언트들(DEALER)가 비동기 서버들(ROUTER)와 통신하여 양측에서 메시지 형식을 완전히 제어하게 합니다.

DEALER와 ROUTER는 임의의 메시지 형식으로 작업할 수 있기 때문에 안전하게 사용하려면 프로토콜 디자이너가 되어야 합니다. 최소한 REQ/REP 응답 봉투를 모방할지 여부를 결정해야 합니다. 실제로 답장을 보내야 하는지 여부에 따라 다릅니다.

0.33.5 DEALER와 DEALER 조합

REP를 ROUTER와 교체할 수 있지만, DEALER가 단 하나의 상대와 통신하는 경우 DEALER를 REP로 교체할 수 있습니다.

REP를 DEALER로 교체하면 작업자가 갑자기 전체 비동기식으로 전환되어 원하는 수의 응답을 다시 보낼 수 있습니다. 이에 따른 대가는 응답 봉투를 직접 관리하고 올바르게 처리해야 해야 하며 그렇지 않으면 아무것도 작동하지 않습니다. 나중에 작동되는 예제를 보겠습니다. 지금 당장은 DEALER와 DEALER가 제대로 작동하기에는 까다로운 패턴 중 하나이며 다행히도 우리가 필요로 하는 경우는 드뭅니다.

0.33.6 ROUTER와 ROUTER 조합

이것은 N 대 N 연결에 완벽하게 들리지만 사용하기 가장 어려운 조합입니다. ØMQ에 익숙해질 때까지 피해야 합니다. “4장-신뢰할 수 있는 요청-응답 패턴”의 프리랜서 패턴에서 한 가지 예제를 보겠으며, “8장 - 분산 컴퓨팅을 위한 프레임워크”에서 P2P(peer-to-peer) 작업을 위한 ROUTER 설계를 대체하는 DEALER 설계를 보겠습니다.

0.33.7 잘못된 조합

대부분의 경우 클라이언트를 클라이언트에 연결하거나, 서버를 서버에 연결하는 것은 나쁜 생각이며 동작하지 않습니다. 그러나 일반적으로 모호한 경고를 하기보다는 자세히 설명하겠

습니다.

- REQ와 REQ 조합 : 양측은 서로에게 메시지를 보내는 것으로 시작하기를 원하며, 두 상대가 동시에 메시지를 교환하도록 시간을 정한 경우에만 작동합니다. 생각만 해도 머리가 아픕니다.
- REQ와 DEALER 조합 : 이론적으로는 이것을 할 수 있지만, DEALER에 두 번째 REQ를 추가하면 원래 상대에게 답장을 보낼 방법이 없기 때문에 REQ와 DEALER 간의 통신은 중단됩니다. 따라서 REQ 소켓은 혼란스럽고 다른 클라이언트를 위한 메시지를 반환합니다.
- REP와 REP 조합 : 양측은 상대방이 첫 번째 메시지를 보내기를 기다립니다.
- REP와 ROUTER 조합 : ROUTER 소켓은 이론적으로 REP 소켓이 연결되어 있고 해당 연결된 식별자(ID)를 알고 있는 경우 통신을 시작하고 적절한 형식의 요청을 할 수 있지만, 그것은 지저분하고 ROUTER와 DEALER 위에 아무것도 추가하지 않습니다.

ØMQ의 올바른 소켓의 조합에 대해 일반적인 사항은 항상 어느 한쪽이 단말에 바인딩하고 다른 쪽에서는 연결한다는 것입니다. 추가로, 어느 쪽에서 바인딩을 하거나 연결을 해도 상관없지만 자연스러운 패턴을 따라야 합니다. "확실(정적)한 존재"임이 기대되는 측면이 바인딩을 실시하여 서버, 브로커, 발행자, 수집가가 되며, '희미(동적)한 존재'는 연결을 실시하여 클라이언트나 작업자가 될 것입니다. 이를 기억해두면 더 좋은 ØMQ 아키텍처를 설계하는 데 도움이 됩니다.

0.34 ROUTER 소켓 알아보기

ROUTER 소켓을 좀 더 자세히 보겠습니다. ROUTER가 개별 메시지를 특정 연결들로 라우팅하는 것을 보았습니다. 이러한 연결들을 식별하는 방법과 메시지를 보낼 수 없는 경우 ROUTER 소켓이 수행하는 작업에 대해 자세히 설명하겠습니다.

0.34.1 식별자와 주소

ØMQ에서 식별자(ID)의 개념은 ROUTER 소켓이 다른 소켓에 대한 연결을 식별하는 방법입니다. 더 광범위하게, 식별자들(IDs)은 응답 봉투의 주소로 사용됩니다. 대부분의 경우, 식별자

(ID)는 로컬에서 ROUTER 소켓에 대한 해쉬 테이블의 색인 키로 사용됩니다. 독자적으로 상대는 물리적인 주소(네트워크의 단말 “tcp : //192.168.55.117 : 5670”)와 논리적인 주소(UUID 혹은 이메일 주소, 다른 고유한 키)를 가질 수 있습니다.

ROUTER 소켓을 사용하여 특정 상대와 통신하는 응용프로그램은 해쉬 테이블을 구축한 경우 논리 주소를 식별자(ID)로 변환할 수 있습니다. ROUTER 소켓은 상대가 메시지를 보낼 때 연결의 식별자(ID)만 알리기 때문에, 단지 상대에게 메시지로 응답할 수 있을 뿐이지, 자발적으로 상대와 통신할 수 없습니다.

이는 규칙을 뒤집고 상대가 ROUTER에 연결될 때까지 기다리지 않고 ROUTER를 상대에 연결하는 경우에도 마찬가지입니다. 그러나 ROUTER 소켓이 식별자(ID) 대신 논리 주소를 사용하도록 강제할 수 있습니다. `zmq_setsockopt()` 참조 페이지는 소켓 식별자(ID) 설정하는 방법이 설명되어 있으며 다음과 같이 작동합니다.

- ROUTER에 바인딩 혹은 연결하려는 상대 응용프로그램 소켓(DEALER 또는 REQ)에서 대한 옵션 ZMQ_IDENTITY 옵션을 설정합니다.
- 일반적으로 상대는 이미 바인딩된 ROUTER 소켓에 연결합니다. 그러나 ROUTER도 상대에 연결할 수도 있습니다.
- 연결 수행 시, 상대 소켓(REQ 혹은 DEALER)은 ROUTER 소켓에 “연결에 대한 식별자(ID)로 사용하십시오”라고 알려줍니다.
- 상대 소켓(REQ 혹은 DEALER)이 그렇게 하지 않으면, ROUTER는 연결에 대해 일반적인 임의의 식별자(ID)를 생성합니다.
- 이제 ROUTER 소켓은 해당 상대에서 들어오는 모든 메시지에 대하여 접두사 식별자 프레임으로 논리 주소를 응용프로그램에 제공합니다.
- ROUTER는 모든 보내는 메시지에 대하여서도 접두사 식별자 프레임을 논리 주소로 사용됩니다.
- [옮긴이] `zmq_setsockopt (client, ZMQ_IDENTITY, "PEER1", 5);`

다음은 ROUTER 소켓에 연결하는 2개의 상대의 간단한 예입니다. 하나는 논리적 주소 “PEER2”를 부과합니다.

identity.c: 식별자 검사

```
//  Demonstrate request-reply identities

#include "zhelpers.h"

int main (void)
{
    void *context = zmq_ctx_new ();
    void *sink = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (sink, "inproc://example");

    //  First allow ØMQ to set the identity
    void *anonymous = zmq_socket (context, ZMQ_REQ);
    zmq_connect (anonymous, "inproc://example");
    s_send (anonymous, "ROUTER uses a generated 5 byte identity");
    s_dump (sink);

    //  Then set the identity ourselves
    void *identified = zmq_socket (context, ZMQ_REQ);
    zmq_setsockopt (identified, ZMQ_IDENTITY, "PEER2", 5);
    zmq_connect (identified, "inproc://example");
    s_send (identified, "ROUTER socket uses REQ's socket identity");
    s_dump (sink);

    zmq_close (sink);
    zmq_close (anonymous);
    zmq_close (identified);
    zmq_ctx_destroy (context);
    return 0;
```

```
}
```

수행 결과는 다음과 같습니다.

```
----------------------------------------
[005] 006B8B4567
[000]
[026] ROUTER uses a generated UUID
----------------------------------------
[005] PEER2
[000]
[038] ROUTER uses REQ's socket identity
```

- [옮긴이] s_dump()는 zhelpers.h에 정의된 함수로 전달된 소켓에서 수신된 메시지의 내용을 멀티파트 메시지로 모두 출력하는 함수입니다.

```
//  Receives all message parts from socket, prints neatly
//
static void
s_dump (void *socket)
{
    int rc;

    zmq_msg_t message;
    rc = zmq_msg_init (&message);
    assert (rc == 0);

    puts ("----------------------------------------");
    //  Process all parts of the message
    do {
        int size = zmq_msg_recv (&message, socket, 0);
```

```
        assert (size >= 0);

        //  Dump the message as text or binary
        char *data = (char*)zmq_msg_data (&message);
        assert (data != 0);
        int is_text = 1;
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++) {
            if ((unsigned char) data [char_nbr] < 32
                || (unsigned char) data [char_nbr] > 126) {
                is_text = 0;
            }
        }

        printf ("[%03d] ", size);
        for (char_nbr = 0; char_nbr < size; char_nbr++) {
            if (is_text) {
                printf ("%c", data [char_nbr]);
            } else {
                printf ("%02X", (unsigned char) data [char_nbr]);
            }
        }
        printf ("\n");
    } while (zmq_msg_more (&message));

    rc = zmq_msg_close (&message);
    assert (rc == 0);
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc identity.c libzmq.lib


 ./identity
----------------------------------------
[005] 0080000029
[000]
[039] ROUTER uses a generated 5 byte identity
----------------------------------------
[005] PEER2
[000]
[040] ROUTER socket uses REQ's socket identity
```

0.34.2 ROUTER 오류 처리

ROUTER 소켓은 어디에도 보낼 수 없는 메시지들를 처리하는 다소 무식한 방법을 가지고 있습니다 : 조용히 메시지들을 버리기. 동작하는 코드에서 의미가 있는 태도이지만 디버깅을 어렵게 만듭니다. "첫 번째 프레임으로 식별자(ID) 보내기" 접근은 우리가 학습할 때 종종 실수할 정도로 까다로우며 ROUTER의 바위 같은 침묵은 그다지 건설적이지 않습니다.

ØMQ v3.2부터는 소켓 옵션을 설정하여 오류를 잡을 수 있습니다 : ZMQ_ROUTER_MANDATORY. ROUTER 소켓에 설정한 다음 상대(REQ 혹은 DEALER)로 메시지를 전송(send) 시 라우팅할 수 없는 식별자를 만나면 ROUTER 소켓에서 EHOSTUNREACH 오류를 알립니다.

0.35 부하 분산 패턴

이제 몇 가지 코드를 살펴보겠습니다. ROUTER 소켓에 REQ 소켓을 연결 한 다음 DEALER 소켓에 연결하는 방법을 살펴보겠습니다(REQ-ROUTER-DEALER). 2개의 예제는 부하 분산 패턴과 동일한 처리 방법을 따릅니다. 이 패턴은 단순히 응답 채널 역할을 하는 것보다 계획적인 라우팅을 위해 ROUTER 소켓을 사용하는 첫 번째 사례입니다.

부하 분산 패턴은 매우 일반적이며 이 책에서 여러 번 보았습니다. 간단한 라운드 로빈

라우팅(PUSH 및 DEALER 제공하는 것처럼)으로 주요 문제를 해결하였지만 라운드 로빈은 작업의 처리 시간이 고르지 않은 경우 비효율적이 될 수 있습니다.

우체국에 비유하면, 카운터 당 대기열이 하나 있고 우표를 사는 사람(빠르고 간단한 거래)이 있고 새 계정을 여는 사람(매우 느린 거래)이 있다면 우표 구매자가 대기열에 부당하게 기다리는 것을 발견하게 될 것입니다. 우체국에서와 마찬가지로 메시징 아키텍처가 불공평하면 사람들은 짜증을 낼 것입니다.

우체국에 대한 솔루션은 업무 처리 특성에 따라 느린 작업에 하나 또는 두 개의 카운터을 생성하고, 그 외의 작업은 다른 카운터가 선착순으로 클라이언트에게 서비스를 제공하도록 단일 대기열을 만드는 것입니다.

단순한 접근 방식의 PUSH와 DEALER를 사용하는 한 가지 이유는 순수한 성능 때문입니다. 미국의 주요 공항에 도착하면 이민국에서 사람들이 줄을 서서 기다리는 것을 볼 수 있습니다. 국경 순찰대원은 단일 대기열을 사용하는 대신 각 카운터로 미리 사람들을 보내 대기하게 합니다. 사람들이 미리 50야드를 걷게 하여 승객당 1~2분 시간을 절약 가능한 것은 모든 여권 검사는 거의 같은 시간이 걸리기에 다소 공정합니다. 이것이 PUSH 및 DEALER의 전략입니다. 이동 거리가 줄어들도록 미리 작업 부하를 보냅니다.

이것은 ØMQ에서 되풀이되는 주제입니다 : 세계의 문제는 다양하며 올바른 방법으로 각각 다른 문제를 해결함으로써 이익을 얻을 수 있습니다. 공항은 우체국이 아니며 문제의 규모가 다릅니다.

브로커(ROUTER)에 연결된 작업자(DEALER 또는 REQ) 시나리오로 돌아가 보겠습니다. 브로커는 작업자가 언제 준비되었는지 알고 있어야 하며, 매번 최저사용빈도(LRU) 작업자를 선택할 수 있도록 작업자 목록을 유지해야 합니다.

- [옮긴이] LRU(least recently used)는 최저사용빈도의 의미로 오랫동안 사용되지 않은 대상을 선택하는 알고리즘입니다.

솔루션은 실제로 간단합니다. 작업자가 시작할 때와 각 작업을 완료한 후에 “ready” 메시지를 보냅니다. 브로커는 이러한 메시지를 하나씩 읽습니다. 메시지를 읽을 때마다 마지막으로 사용한 작업자가 보낸 것입니다. 그리고 우리는 ROUTER 소켓을 사용하기 때문에 작업자에게 작업을 회신할 수 있는 ID가 있습니다.

작업에 대한 결과는 응답이 보내지고, 새로운 작업 요청에 따라 작업에 대한 응답은 보내

지기 때문에 요청-응답에 대한 응용입니다. 이들을 이해하기 위하여 예제 코드로 구현하겠습니다.

0.35.1 ROUTER 브로커와 REQ 작업자

일련의 REQ 작업자들과 통신하는 ROUTER 브로커를 사용하는 부하 분산 패턴의 예입니다.

rtreq.c: ROUTER와 REQ 통신

```
// 2015-01-16T09:56+08:00
//  ROUTER-to-REQ example

#include "zhelpers.h"
#include <pthread.h>

#define NBR_WORKERS 10

static void *
worker_task(void *args)
{
    void *context = zmq_ctx_new();
    void *worker = zmq_socket(context, ZMQ_REQ);

#if (defined (WIN32))
    s_set_id(worker, (intptr_t)args);
#else
    s_set_id(worker);          //  Set a printable identity.
#endif

    zmq_connect(worker, "tcp://localhost:5671");
```

```
    int total = 0;
    while (1) {
        //  Tell the broker we're ready for work
        s_send(worker, "Hi Boss");

        //  Get workload from broker, until finished
        char *workload = s_recv(worker);
        int finished = (strcmp(workload, "Fired!") == 0);
        free(workload);
        if (finished) {
            printf("Completed: %d tasks\n", total);
            break;
        }
        total++;

        //  Do some random work
        s_sleep(randof(500) + 1);
    }
    zmq_close(worker);
    zmq_ctx_destroy(context);
    return NULL;
}

//  .split main task
//  While this example runs in a single process, that is only to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.

int main(void)
```

```
{
    void *context = zmq_ctx_new();
    void *broker = zmq_socket(context, ZMQ_ROUTER);

    zmq_bind(broker, "tcp://*:5671");
    srandom((unsigned)time(NULL));

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create(&worker, NULL, worker_task, (void *)(intptr_t)worker_nbr);
    }
    //  Run for five seconds and then tell workers to end
    int64_t end_time = s_clock() + 5000;
    int workers_fired = 0;
    while (1) {
        //  Next message gives us least recently used worker
        char *identity = s_recv(broker);
        s_sendmore(broker, identity);
        free(identity);
        free(s_recv(broker));     //  Envelope delimiter
        free(s_recv(broker));     //  Response from worker
        s_sendmore(broker, "");

        //  Encourage workers until it's time to fire them
        if (s_clock() < end_time)
            s_send(broker, "Work harder");
        else {
            s_send(broker, "Fired!");
```

```
            if (++workers_fired == NBR_WORKERS)
                break;
        }
    }
    zmq_close(broker);
    zmq_ctx_destroy(context);
    return 0;
}
```

예제는 5초 동안 실행된 다음 각 작업자가 처리한 작업의 개수를 출력합니다. ROUTER 소켓을 통한 부하 분산으로 REQ 요청을 처리하면 작업의 공정한 분배를 기대할 수 있습니다.

```
Completed: 20 tasks
Completed: 18 tasks
Completed: 21 tasks
Completed: 23 tasks
Completed: 19 tasks
Completed: 21 tasks
Completed: 17 tasks
Completed: 17 tasks
Completed: 25 tasks
Completed: 19 tasks
```

예제에서 작업자와 통신하기 위하여 식별자(ID)와 공백 구분자 프레임으로 REQ-적합한 봉투를 생성해야 했습니다.

그림 31 - REQ에 대한 라우팅 봉투(Routing Envelope for REQ)

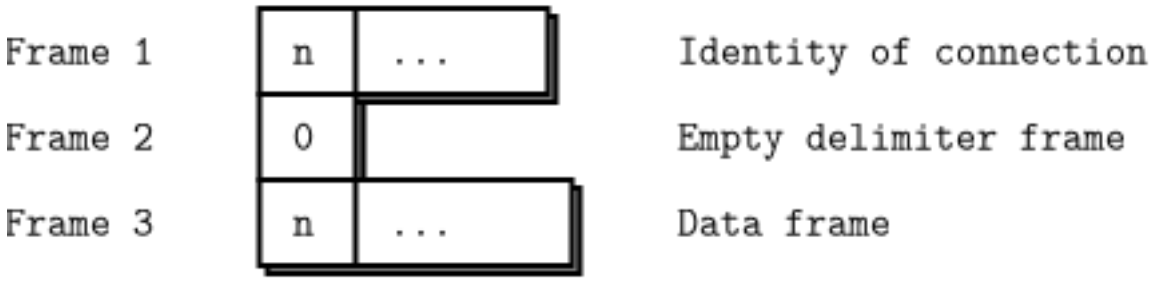


그림 32: RRouting Envelope for REQ

- [옮긴이] rtreq에 대하여 REQ 작업자에게 스레드 식별자(ID)를 출력하도록 수정하면 다음과 같습니다.

```
// 2015-01-16T09:56+08:00
//  ROUTER-to-REQ example

#include "zhelpers.h"
#include <pthread.h>

#define NBR_WORKERS 10

static void *
worker_task(void *args)
{
    void *context = zmq_ctx_new();
    void *worker = zmq_socket(context, ZMQ_REQ);

#if (defined (WIN32))
    s_set_id(worker, (intptr_t)args);
#else
```

```
    s_set_id(worker);          //  Set a printable identity.
#endif

    zmq_connect(worker, "tcp://localhost:5671");

    int total = 0;
    while (1) {
        //  Tell the broker we're ready for work
        s_send(worker, "Hi Boss");

        //  Get workload from broker, until finished
        char *workload = s_recv(worker);
        int finished = (strcmp(workload, "Fired!") == 0);
        free(workload);
        if (finished) {
            printf("[%Id] Completed: %d tasks\n", (intptr_t)args, total);
            break;
        }
        total++;

        //  Do some random work
        s_sleep(randof(500) + 1);
    }
    zmq_close(worker);
    zmq_ctx_destroy(context);
    return NULL;
}

//  .split main task
```

```
//  While this example runs in a single process, that is only to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.

int main(void)
{
    void *context = zmq_ctx_new();
    void *broker = zmq_socket(context, ZMQ_ROUTER);

    zmq_bind(broker, "tcp://*:5671");
    srandom((unsigned)time(NULL));

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create(&worker, NULL, worker_task, (void *)(intptr_t)worker_nbr);
    }
    //  Run for five seconds and then tell workers to end
    int64_t end_time = s_clock() + 5000;
    int workers_fired = 0;
    while (1) {
        //  Next message gives us least recently used worker
        char *identity = s_recv(broker);
        free(s_recv(broker));     //  Envelope delimiter
        free(s_recv(broker));     //  Response from worker
        s_sendmore(broker, identity);
        free(identity);
        s_sendmore(broker, "");
        //  Encourage workers until it's time to fire them
```

```
        if (s_clock() < end_time)
            s_send(broker, "Work harder");
        else {
            s_send(broker, "Fired!");
            if (++workers_fired == NBR_WORKERS)
                break;
        }
    }
    zmq_close(broker);
    zmq_ctx_destroy(context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc rtreq.c libzmq.lib pthreadVC2.lib


 ./rtreq
[2] Completed: 22 tasks
[8] Completed: 22 tasks
[7] Completed: 22 tasks
[9] Completed: 22 tasks
[0] Completed: 22 tasks
[4] Completed: 22 tasks
[6] Completed: 22 tasks
[1] Completed: 22 tasks
[5] Completed: 22 tasks
[3] Completed: 22 tasks
```

- [옮긴이] mtserver_client 예제에서는 REQ 클라이언트 스레드들과 REP 작업자 스레드들과 ROUTER-DEALER 간에 `zmq_proxy()`을 통해 그대로 전달하여, 식별자(ID) 및

공백 구분자 프레임 없이 데이터만으로 통신이 가능했습니다. 위의 예제는 ROUTER 소켓에서 받은 데이터를 직접 다루기 때문에 3개의 프레임들(ID + empty delimiter + body) 처리가 필요합니다.

0.35.2 ROUTER 브로커와 DEALER 작업자

어디서든 REQ를 사용할 수 있는 곳이면 DEALER를 사용할 수 있습니다. 2개의 구체적인 차이점이 있습니다 :

- REQ 소켓은 모든 데이터 프레임 앞에 공백 구분자 프레임을 보냅니다. DEALER는 그렇지 않습니다.
- REQ 소켓은 응답을 받기 전에 하나의 요청 메시지만 보냅니다. DEALER는 완전히 비동기적입니다.

동기와 비동기 동작은 엄격한 요청-응답을 수행하기 때문에 예제에 영향을 미치지 않습니다. 이는 "4장-신뢰할 수 있는 요청-응답 패턴"에서 다루며 오류 복구를 처리할 때 더 관련이 있습니다.

이제 똑같은 예제지만 REQ 소켓이 DEALER 소켓으로 대체되었습니다.

rtdealer.c: ROUTER에서 DEALER

```
// 2015-02-27T11:40+08:00
//  ROUTER-to-DEALER example

#include "zhelpers.h"
#include <pthread.h>
#define NBR_WORKERS 10

static void *
worker_task(void *args)
{
    void *context = zmq_ctx_new();
```

```
    void *worker = zmq_socket(context, ZMQ_DEALER);

#if (defined (WIN32))
    s_set_id(worker, (intptr_t)args);
#else
    s_set_id(worker);          //  Set a printable identity
#endif

    zmq_connect (worker, "tcp://localhost:5671");

    int total = 0;
    while (1) {
        //  Tell the broker we're ready for work
        s_sendmore(worker, "");
        s_send(worker, "Hi Boss");

        //  Get workload from broker, until finished
        free(s_recv(worker));     //  Envelope delimiter
        char *workload = s_recv(worker);
        //  .skip
        int finished = (strcmp(workload, "Fired!") == 0);
        free(workload);
        if (finished) {
            printf("[%d] Completed: %d tasks\n", (intptr_t)args, total);
            break;
        }
        total++;

        //  Do some random work
```

```
        s_sleep(randof(500) + 1);
    }
    zmq_close(worker);
    zmq_ctx_destroy(context);
    return NULL;
}

//  .split main task
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.

int main(void)
{
    void *context = zmq_ctx_new();
    void *broker = zmq_socket(context, ZMQ_ROUTER);

    zmq_bind(broker, "tcp://*:5671");
    srandom((unsigned)time(NULL));

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create(&worker, NULL, worker_task, (void *)(intptr_t)worker_nbr);
    }
    //  Run for five seconds and then tell workers to end
    int64_t end_time = s_clock() + 5000;
    int workers_fired = 0;
    while (1) {
```

```
        //  Next message gives us least recently used worker
        char *identity = s_recv(broker);
        s_sendmore(broker, identity);
        free(identity);
        free(s_recv(broker));     //  Envelope delimiter
        free(s_recv(broker));     //  Response from worker
        s_sendmore(broker, "");

        //  Encourage workers until it's time to fire them
        if (s_clock() < end_time)
            s_send(broker, "Work harder");
        else {
            s_send(broker, "Fired!");
            if (++workers_fired == NBR_WORKERS)
                break;
        }
    }
    zmq_close(broker);
    zmq_ctx_destroy(context);
    return 0;
}
//  .until
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc rtdealer.c libzmq.lib pthreadVC2.lib
 ./rtdealer
[9] Completed: 22 tasks
[7] Completed: 22 tasks
[3] Completed: 22 tasks
```

```
[1] Completed: 22 tasks
[5] Completed: 22 tasks
[0] Completed: 22 tasks
[4] Completed: 22 tasks
[8] Completed: 22 tasks
[6] Completed: 22 tasks
[2] Completed: 22 tasks
```

- [옮긴이] rtreq 예제에서 REQ 소켓을 사용한 작업자에서는 공백 구분자 없이 데이터만 송/수신하였지만, DEALER 사용한 작업자에서는 공백 구분자와 데이터 프레임을 송/수신하였습니다. main() 함수의 소스는 변경이 없지만 작업자 스레드에서 공백 구분자를 송/수신하도록 변경되었습니다.

- [옮긴이] DEALER-ROUTER 송/수신시에 프레임 구성은 다음과 같습니다.

- 송신 : APP-[“”+“Hello”]->DEALER-[“”+“Hello”]->ROUTER-[ID+“”+“Hello”]->APP

- 수신 : APP-[ID+“”+“World”]-ROUTER->[ID+“”+“World”]->DEALER-[“”+“World”]->APP

작업자가 DEALER 소켓을 사용하고 데이터 프레임 이전에 공백 구분자를 읽고 쓴다는 점을 제외하면 코드는 거의 동일합니다. 이것은 REQ 작업자와의 호환성을 유지하기 위한 접근 방식입니다.

DEALER에서 공백 구분자 프레임을 넣은 이유를 기억하십시오. REP 소켓에서 종료되는 다중도약 네트워크 확장 요청을 허용하여 응답 봉투에서 공백 구분자를 분리하여 데이터 프레임을 응용프로그램에 전달할 수 있습니다.

- [옮긴이] 다중도약 네트워크(multihop network)는 고정되어 있거나 이동하는 단말들을 도약해 무선통신 네트워크를 효율적으로 구성하는 기술입니다. 이 기술은 네트워크를 확장하지 않아도 가청 범위를 넓힐 수 있게 합니다.

메시지가 REP 소켓을 경유하지 않는다면 양쪽에 공백 구분자 프레임을 생략할 수 있으며 이렇게 함으로 간단합니다. 이것은 순수한 DEALER와 ROUTER 프로토콜을 이용하고 싶은 경우에 일반적인 설계 방법입니다.

- [옮긴이] DEALER-ROUTER 송/수신시에서 공백을 제거할 경우 프레임 구성은 다음과 같습니다.
- 송신 : APP-[“Hello”]->DEALER-[“Hello”]->ROUTER-[ID+“Hello”]->APP
- 수신 : APP-[ID+“World”]-ROUTER->[ID+“World”]->DEALER-[World“]->APP

rtdealer1.c : DEALER-ROUTER 송/수신에서 공백을 제거함

```
// 2015-02-27T11:40+08:00
//  ROUTER-to-DEALER example

#include "zhelpers.h"
#include <pthread.h>
#define NBR_WORKERS 10

static void *
worker_task(void *args)
{
    void *context = zmq_ctx_new();
    void *worker = zmq_socket(context, ZMQ_DEALER);

#if (defined (WIN32))
    s_set_id(worker, (intptr_t)args);
#else
    s_set_id(worker);          //  Set a printable identity
#endif

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

    int total = 0;
    while (1) {
```

```
        //  Tell the broker we're ready for work
        s_send(worker, "Hi Boss");

        //  Get workload from broker, until finished
        char *workload = s_recv(worker);
        //  .skip
        int finished = (strcmp(workload, "Fired!") == 0);
        free(workload);
        if (finished) {
            printf("[%Id] Completed: %d tasks\n", (intptr_t)args, total);
            break;
        }
        total++;

        //  Do some random work
        s_sleep(randof(500) + 1);
    }
    zmq_close(worker);
    zmq_ctx_destroy(context);
    return NULL;
}

//  .split main task
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.

int main(void)
{
```

```
    void *context = zmq_ctx_new();
    void *broker = zmq_socket(context, ZMQ_ROUTER);

    zmq_bind(broker, "tcp://*:5555");
    srandom((unsigned)time(NULL));

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create(&worker, NULL, worker_task, (void *)(intptr_t)worker_nbr);
    }
    //  Run for five seconds and then tell workers to end
    int64_t end_time = s_clock() + 5000;
    int workers_fired = 0;
    while (1) {
        //  Next message gives us least recently used worker
        char *identity = s_recv(broker);
        s_sendmore(broker, identity);
        free(identity);
        free(s_recv(broker));     //  Response from worker

        //  Encourage workers until it's time to fire them
        if (s_clock() < end_time)
            s_send(broker, "Work harder");
        else {
            s_send(broker, "Fired!");
            if (++workers_fired == NBR_WORKERS)
                break;
        }
```

```
    }
    zmq_close(broker);
    zmq_ctx_destroy(context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc rtdealer1.c libzmq.lib pthreadVC2.lib
 ./rtdealer1
[0] Completed: 21 tasks
[1] Completed: 21 tasks
[5] Completed: 22 tasks
[6] Completed: 20 tasks
[4] Completed: 17 tasks
[2] Completed: 24 tasks
[9] Completed: 23 tasks
[3] Completed: 18 tasks
[7] Completed: 26 tasks
[8] Completed: 19 tasks
```

0.35.3 부하 분산 메시지 브로커

이전 예제까지 절반 정도 완성되었습니다. 일련의 작업자들을 더미 요청 및 응답으로 관리할 수 있지만 클라이언트와 통신할 방법은 없습니다. 클라이언트 요청을 수락하는 두 번째 프론트엔드 ROUTER 소켓을 추가하고 이전 예제를 프론트엔드에서 백엔드로 메시지를 전환할 수 있는 프록시로 바꾸면, 유용하고 재사용 가능한 작은 부하 분산 메시지 브로커을 가지게 됩니다.

그림 32 - 부하 분산 브로커(Load Balancing Broker)

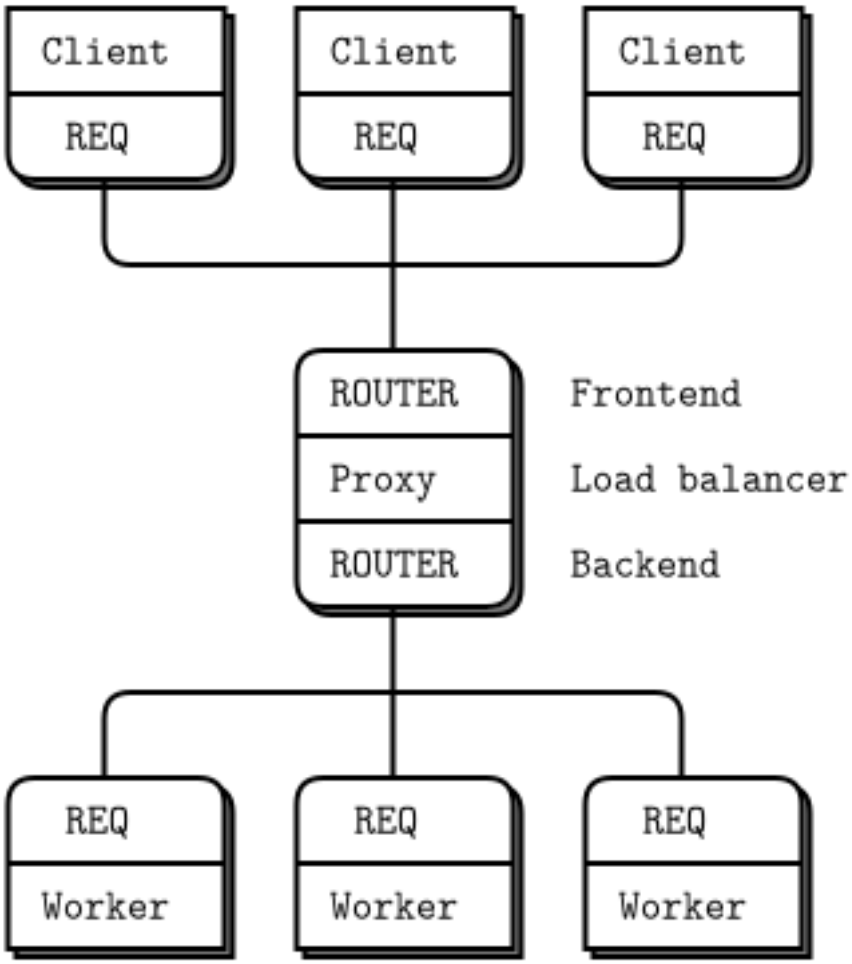


그림 33: Load Balancing Broker

이 브로커는 다음과 같은 작업을 수행합니다.

- 일련의 클라이언트들로부터의 연결을 받습니다.
- 일련의 작업자들로부터의 연결을 받습니다.
- 클라이언트의 요청을 받고 단일 대기열에 보관합니다.
- 부하 분산 패턴을 사용하여 이러한 요청들을 작업자에게 보냅니다.
- 작업자들로부터 응답을 받습니다.
- 이러한 응답을 원래 요청한 클라이언트로 다시 보냅니다.

브로커 코드는 상당히 길지만 이해할 가치가 있습니다.

lbbroker.c: 부하 분산 브로커

```
//  Load-balancing broker
//  Clients and workers are shown here in-process

#include "zhelpers.h"
#include <pthread.h>
#define NBR_CLIENTS 10
#define NBR_WORKERS 3

//  Dequeue operation for queue implemented as array of anything
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof(q) - sizeof(q[0]))

//  Basic request-reply client using REQ socket
//  Because s_send and s_recv can't handle ØMQ binary identities, we
//  set a printable text identity to allow routing.
//
static void *
client_task(void *args)
{
    void *context = zmq_ctx_new();
    void *client = zmq_socket(context, ZMQ_REQ);

#if (defined (WIN32))
    s_set_id(client, (intptr_t)args);
    zmq_connect(client, "tcp://localhost:5672"); // frontend
#else
    s_set_id(client); // Set a printable identity
    zmq_connect(client, "ipc://frontend.ipc");
```

```
#endif

    //  Send request, get reply
    s_send(client, "HELLO");
    char *reply = s_recv(client);
    printf("Client: %s\n", reply);
    free(reply);
    zmq_close(client);
    zmq_ctx_destroy(context);
    return NULL;
}

//  .split worker task
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//  This is the worker task, using a REQ socket to do load-balancing.
//  Because s_send and s_recv can't handle ØMQ binary identities, we
//  set a printable text identity to allow routing.

static void *
worker_task(void *args)
{
    void *context = zmq_ctx_new();
    void *worker = zmq_socket(context, ZMQ_REQ);

#if (defined (WIN32))
    s_set_id(worker, (intptr_t)args);
    zmq_connect(worker, "tcp://localhost:5673"); // backend
```

```
#else
    s_set_id(worker);
    zmq_connect(worker, "ipc://backend.ipc");
#endif

    //  Tell broker we're ready for work
    s_send(worker, "READY");

    while (1) {
        //  Read and save all frames until we get an empty frame
        //  In this example there is only 1, but there could be more
        char *identity = s_recv(worker);
        char *empty = s_recv(worker);
        assert(*empty == 0);
        free(empty);

        //  Get request, send reply
        char *request = s_recv(worker);
        printf("Worker: %s\n", request);
        free(request);

        s_sendmore(worker, identity);
        s_sendmore(worker, "");
        s_send(worker, "OK");
        free(identity);
    }
    zmq_close(worker);
    zmq_ctx_destroy(context);
    return NULL;
```

```
}

//  .split main task
//  This is the main task. It starts the clients and workers, and then
//  routes requests between the two layers. Workers signal READY when
//  they start; after that we treat them as ready when they reply with
//  a response back to a client. The load-balancing data structure is
//  just a queue of next available workers.

int main(void)
{
    //  Prepare our context and sockets
    void *context = zmq_ctx_new();
    void *frontend = zmq_socket(context, ZMQ_ROUTER);
    void *backend = zmq_socket(context, ZMQ_ROUTER);

#if (defined (WIN32))
    zmq_bind(frontend, "tcp://*:5672"); // frontend
    zmq_bind(backend, "tcp://*:5673"); // backend
#else
    zmq_bind(frontend, "ipc://frontend.ipc");
    zmq_bind(backend, "ipc://backend.ipc");
#endif

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++) {
        pthread_t client;
        pthread_create(&client, NULL, client_task, (void *)(intptr_t)client_nbr);
    }
```

```
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create(&worker, NULL, worker_task, (void *)(intptr_t)worker_nbr);
    }
    //  .split main task body
    //  Here is the main loop for the least-recently-used queue. It has two
    //  sockets; a frontend for clients and a backend for workers. It polls
    //  the backend in all cases, and polls the frontend only when there are
    //  one or more workers ready. This is a neat way to use ØMQ's own queues
    //  to hold messages we're not ready to process yet. When we get a client
    //  request, we pop the next available worker and send the request to it,
    //  including the originating client identity. When a worker replies, we
    //  requeue that worker and forward the reply to the original client
    //  using the reply envelope.

    //  Queue of available workers
    int available_workers = 0;
    char *worker_queue[10];

    while (1) {
        zmq_pollitem_t items[] = {
                { backend, 0, ZMQ_POLLIN, 0 },
                { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll(items, available_workers ? 2 : 1, -1);
        if (rc == -1)
            break;              //  Interrupted
```

```
    //  Handle worker activity on backend
    if (items[0].revents & ZMQ_POLLIN) {
        //  Queue worker identity for load-balancing
        char *worker_id = s_recv(backend);
        assert(available_workers < NBR_WORKERS);
        worker_queue[available_workers++] = worker_id;

        //  Second frame is empty
        char *empty = s_recv(backend);
        assert(empty[0] == 0);
        free(empty);

        //  Third frame is READY or else a client reply identity
        char *client_id = s_recv(backend);

        //  If client reply, send rest back to frontend
        if (strcmp(client_id, "READY") != 0) {
            empty = s_recv(backend);
            assert(empty[0] == 0);
            free(empty);
            char *reply = s_recv(backend);
            s_sendmore(frontend, client_id);
            s_sendmore(frontend, "");
            s_send(frontend, reply);
            free(reply);
            if (--client_nbr == 0)
                break;      //  Exit after N messages
        }
```

```
            free(client_id);
        }
        //  .split handling a client request
        //  Here is how we handle a client request:

        if (items[1].revents & ZMQ_POLLIN) {
            //  Now get next client request, route to last-used worker
            //  Client request is [identity][empty][request]
            char *client_id = s_recv(frontend);
            char *empty = s_recv(frontend);
            assert(empty[0] == 0);
            free(empty);
            char *request = s_recv(frontend);

            s_sendmore(backend, worker_queue[0]);
            s_sendmore(backend, "");
            s_sendmore(backend, client_id);
            s_sendmore(backend, "");
            s_send(backend, request);

            free(client_id);
            free(request);

            //  Dequeue and drop the next worker identity
            free(worker_queue[0]);
            DEQUEUE(worker_queue);
            available_workers--;
        }
    }
```

```
    zmq_close(frontend);
    zmq_close(backend);
    zmq_ctx_destroy(context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc lbbroker.c libzmq.lib pthreadVC2.lib

 ./lbbroker
Worker: HELLO
Worker: HELLO
Worker: HELLO
Client: OK
Worker: HELLO
Client: OK
Worker: HELLO
Worker: HELLO
Client: OK
Client: OK
Worker: HELLO
Worker: HELLO
Client: OK
Client: OK
Worker: HELLO
Client: OK
Worker: HELLO
Client: OK
Client: OK
```

```
Client: OK
```

- [옮긴이] 송/수신 시 구조
- 송신 : APP(client)->REQ->ROUTER(frontend)->ROUTER(backend)->REQ->APP(worker)
- 수신 : APP(worker)->REQ->ROUTER(backend)->ROUTER(frontend)->REQ->APP(client)
- [옮긴이] 송/수신 시에 멀티파트 메시지 구성
- 송신 : CLIENT -[“HELLO”]-> REQ -[“”+“HELLO”]-> ROUTER -[CID+“”+“HELLO”]-> logic -[WID+“”+CID+“”+“HELLO”]-> ROUTER -[“”+CID+“”+“HELLO”]-> REQ -[CID+“”+“HELLO”]-> WORKER
- 수신 : WORKER -[CID+“”+“OK”]-> REQ -[“”+CID+“”+“OK”]-> ROUTER ->[WID+“”CID+“”+“OK”]-> logic -[CID+“”+“OK”]-> ROUTER -[+“”+“OK”]-> REQ -[“OK”]-> CLIENT
- [옮긴이] dequeue 매크로에서 사용된 memmove() 함수는 source가 가리키는 곳부터 num 바이트만큼을 destination이 가리키는 곳으로 옮기는 역할을 수행하며, 큐(queue)의 q(0)에 q(1)부터 정해진 크기(sizeof(q)-sizeof(q[0]))의 데이터가 복사됩니다.

```
#include <string.h>  // C++ 에서는 <cstring>
void* memmove(void* destination, const void* source, size_t num);
```

- [옮긴이] 클라이언트와 작업자 스레드에서 식별자(ID)를 지정하지 않을 경우, ØMQ에서 ROUTER에서 자체 생성하고 바이너리 ID 형태로 `s_send()`, `s_recv()`에서 처리할 수 없습니다.

이 프로그램의 어려운 부분은 (a)각 소켓이 읽고 쓰는 봉투들과 (b)부하 분배 알고리즘입니다. 메시지 봉투 형식부터 시작하여 차례로 설명하겠습니다.

클라이언트에서 작업자와 전체 요청-응답 체인을 추적하도록 하겠습니다. 이 코드에서는 메시지 프레임을 더 쉽게 추적할 수 있도록 클라이언트 및 작업자 소켓의 식별자(ID)를

설정합니다. 실제로 ROUTER 소켓이 연결을 위한 식별자들(IDs)를 만들게 하였으며, 클라이언트의 ID는 “CLIENT”라 하고 작업자의 ID는 “WORKER”라고 가정하고 클라이언트 응용프로그램은 단일 프레임의 “Hello”를 보냅니다.

그림 33 - 클라이언트가 보낸 메시지(Message that Client Sends)

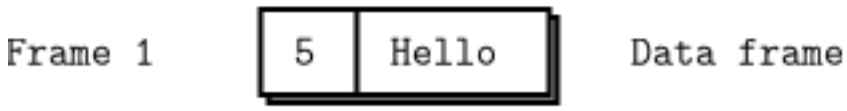


그림 34: Message that Client Sends

REQ 소켓이 공백 구분자 프레임을 추가하고 ROUTER 소켓이 연결 식별자(ID)를 추가하기 때문에 프록시는 프론트엔드 ROUTER 소켓에서 클라이언트 주소(ID), 공백 구분자 프레임(emtpty delimiter), 데이터 부분(body)을 읽습니다.

그림 34 - 프론트엔드가 받는 메시지(Message Coming in on Frontend)

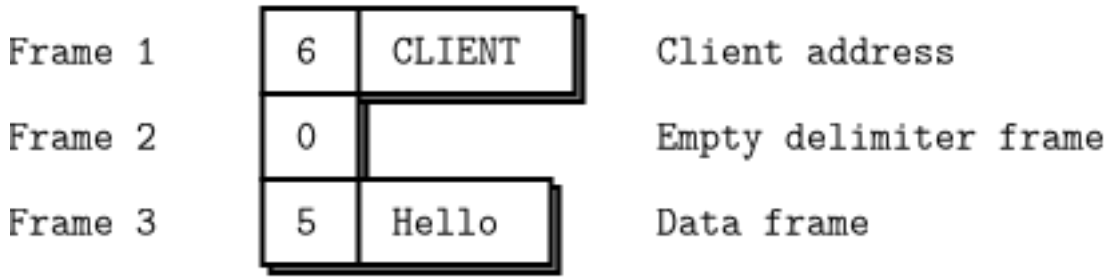


그림 35: Message Coming in on Frontend

그리고 브로커는 다른 쪽 끝에서 REQ를 유지하기 위해 선택된 작업자 식별자(ID)와 공백 구분자를 클라이언트로부터 전달된 메시지의 앞에 추가하여 작업자에게 전송합니다.

그림 35 - 백엔드로 보내는 메시지(Message Send to Backend)

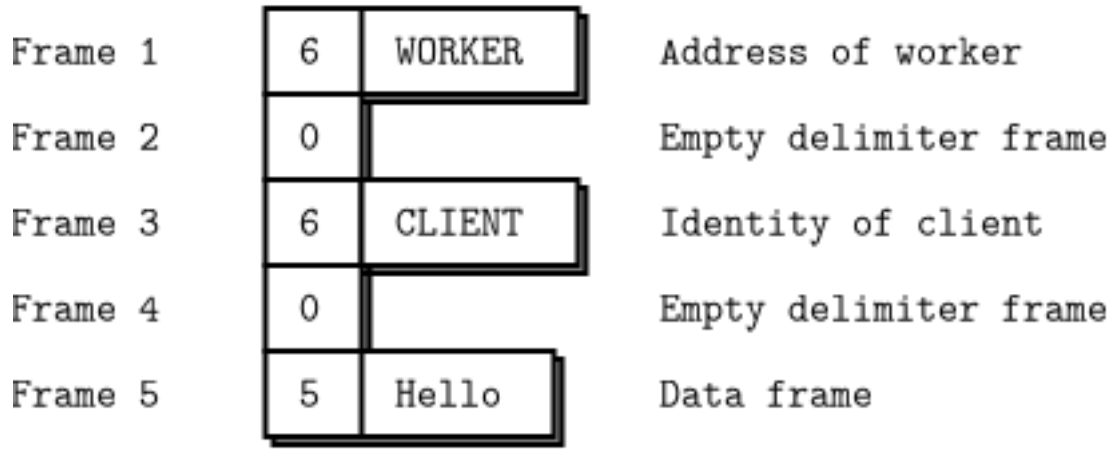


그림 36: Message Send to Backend

이 복잡한 봉투 스택은 백엔드 ROUTER 소켓에 의해 첫 번째 프레임(Worker ID)을 제거되고 작업자의 REQ 소켓은 공백 구분자을 제거하고 나머지는 작업자 응용 프로그램에 제공합니다.

그림 36 - 작업자에게 전달되는 메시지(Message Delivered to Worker)

그림 37: Message Delivered to Worker

작업자는 봉투(빈 메시지 프레임까지 포함하는 모든 부분)를 저장하며 데이터 부분으로 필요한 것을 할 수 있습니다. REP 소켓은 이 작업을 자동으로 수행하지만, REQ-ROUTER 패턴을 사용함으로 적절한 부하 분산을 할 수 있는 것에 주목하시기 바랍니다.

반대 방향으로는 메시지는 들어왔을 때와 동일합니다. 즉, 백엔드 소켓은 브로커에게 다섯

부분으로 메시지(WORKER ID + EMPTY + CLIENT ID + EMPTY + BODY)를 제공하고 브로커는 프론트엔드 소켓에 세 부분으로 메시지(CLIENT ID + EMPTY + BODY)를 보내고 클라이언트는 하나의 데이터 프레임(BODY) 메시지만 받습니다.

이제 부하 분산 알고리즘을 보겠습니다. 모든 클라이언트와 작업자가 REQ 소켓을 사용하고 작업자가 받은 메시지의 봉투를 올바르게 저장하고 응답해야 합니다. 알고리즘은 다음과 같습니다.

- 항상 백엔드를 폴링하는 zmq_pollitem_t 구조체의 배열을 통해 하나 이상의 작업자가 가용한 경우에만 프론트엔드를 폴링합니다(`zmq_poll(items, available_workers ? 2 : 1, -)`).
- 폴링 제한시간은 설정하지 않습니다.
- 백엔드의 활동은 작업자가 기동하여 "READY" 메시지나 클라이언트에 대한 응답이 있으면 작업자 대기열(`worker_queue[10]`)에 작업자 주소(WORKER_ID)를 저장하고 나머지(CLIENT ID + EMPTY + BODY)가 클라이언트 응답인 경우 프론트엔드를 통해 해당 클라이언트로 다시 전송합니다.
- 프론트엔드의 활동은 클라이언트 요청을 받으면 다음 작업자 식별자(ID, 마지막으로 사용된 작업자)를 가져와(DEQUEUE) 요청을 백엔드로 보냅니다. 이것은 작업자 주소(WORKER_ID), 공백 구분자와 클라이언트의 요청인 3개의 프레임을 전송하는 것을 의미합니다.

이제 작업자가 초기 "READY"메시지에서 제공하는 정보를 기반한 수정안으로 부하 분산 알고리즘을 재사용하고 확장할 수 있음을 알아야 합니다. 예를 들어 작업자는 시작하여 자체 성능 테스트를 수행하여 브로커에게 얼마나 빠른지 알려 줄 수 있습니다. 그러면 브로커는 가장 오래된 작업자가 아닌 가장 처리가 빠른 가용한 작업자를 선택할 수 있습니다.

0.36 ØMQ 고급 API

요청-응답을 패턴에 대한 화제를 벗어나 ØMQ API 자신에 대하여 보도록 하겠습니다. 이러한 우회하는 이유가 있습니다. 우리가 더 복잡한 예제를 작성함에 따라 저수준 ØMQ API가 점점 다루기 힘들기 시작합니다. 로드 밸런싱 브로커에서 작업자 스레드의 핵심 처리 방법은 다음과

같습니다.

```
while (true) {
    // Get one address frame and empty delimiter
    char *address = s_recv (worker);
    char *empty = s_recv (worker);
    assert (*empty == 0);
    free (empty);

    // Get request, send reply
    char *request = s_recv (worker);
    printf ("Worker: %s\n", request);
    free (request);

    s_sendmore (worker, address);
    s_sendmore (worker, "");
    s_send (worker, "OK");
    free (address);
}
```

이 코드를 재사용할 수 없는 이유는 봉투에서 하나의 응답 주소만 처리하며 이미 ØMQ API를 일부 감싸두었기((zhelpers.h 파일에서 정의) 때문입니다. 단순 메시지 API(libzmq)를 사용했다면 다음과 같이 작성해야 합니다.

```
while (true) {
    // Get one address frame and empty delimiter
    char address [255];
    int address_size = zmq_recv (worker, address, 255, 0);
    if (address_size == -1)
        break;
```

```
    char empty [1];
    int empty_size = zmq_recv (worker, empty, 1, 0);
    zmq_recv (worker, &empty, 0);
    assert (empty_size <= 0);
    if (empty_size == -1)
        break;

    // Get request, send reply
    char request [256];
    int request_size = zmq_recv (worker, request, 255, 0);
    if (request_size == -1)
        return NULL;
    request [request_size] = 0;
    printf ("Worker: %s\n", request);

    zmq_send (worker, address, address_size, ZMQ_SNDMORE);
    zmq_send (worker, empty, 0, ZMQ_SNDMORE);
    zmq_send (worker, "OK", 2, 0);
}
```

코드가 빨리 작성하기에는 너무 길고 이해하기도 너무 길어집니다. 지금까지 ØMQ 사용자로서 그것을 자세히 알아야 하기 때문에 저수준 API를 고수했습니다. 그러나 어려움이 된다면 해결해야 할 문제로 다루어야 합니다.

물론 수천 명의 사람들이 동의하고 의존하는 문서화된 공개 규약인 ØMQ API을 변경할 수는 없습니다. 대신에 지금까지의 경험, 특히 복잡한 요청-응답 패턴 작성 경험을 기반으로 상위 수준의 API(higher-level API)를 만들 수 있습니다.

우리가 원하는 것은 회신 주소의 개수에 상관없이 응답 봉투를 포함하여, 전체 메시지를 한방에 송/수신할 수 있는 API이며, 최소한의 코드 작성으로 우리가 원하는 것을 할 수 있어야 합니다.

좋은 메시지 API를 만드는 것은 상당히 어렵습니다. 용어 문제가 있습니다 : ØMQ는

멀티파트 메시지들과 개별 메시지 프레임들에 대하여 “메시지”를 사용합니다. 우리에게는 기대하는 데이터 타입이 다르다는 문제도 있습니다 : 메시지 내용이 출력 가능한 문자열 데이터로, 때로는 이진 블록으로 나올 수도 있으며 기술적인 문제로 거대한 데이터를 복사하지 않고 보낼 수도 있습니다.

좋은 API를 만드는 문제는 모든 개발 언어에 영향을 미칩니다. 이 책에서의 사용 사례는 C 언어입니다. 어떤 개발 언어를 사용하든 언어 바인딩(binding)이 C 바인딩보다 좋게(또는 더 나은) 만들 수 있도록 고민이 필요합니다.

0.36.1 고급 API의 특징

고급 API는 3개의 알기 쉬운 개념을 이용합니다 : * 문자열 도우미(이미 `zhelpers.h`에서 `s_send()`, `s_recv()` 사용) * 프레임(메시지 프레임) 및 * 메시지(하나 이상의 프레임 목록)이며 다음은 이러한 개념을 사용하여 다시 작성된 작업자 코드입니다.

```
while (true) {
    zmsg_t *msg = zmsg_recv (worker);
    zframe_reset (zmsg_last (msg), "OK", 2);
    zmsg_send (&msg, worker);
}
```

복잡한 메시지를 읽고 쓰는데 필요한 코드의 양을 줄이는 것은 굉장합니다: 결과는 읽고 이해하기 편해졌습니다. ØMQ 작업의 다른 측면을 고려해서 계속하겠습니다. 지금까지 ØMQ에 대한 경험에 기반한 고급 API의 요구 사항 목록은 다음과 같습니다.

- 소켓 자동 처리.
- 수동으로 소켓을 닫고, 일부(전부는 아님) 경우에 명시적으로 제한시간(linger timeout)을 정하는 것은 번거롭습니다. 컨텍스트를 닫을 때 자동으로 소켓을 닫는 방법이 있으면 좋을 것입니다.
- 이식 가능한 스레드 관리.
- 주요 ØMQ 응용프로그램은 스레드를 사용하지만 POSIX 스레드는 다른 운영체제에는 이식 가능하지 않습니다. 좋은 고급 API는 이식 가능한 계층을 만들어 세부 기능은

숨겨야 합니다.

- 부모와 자식 스레드들 간의 통신.
- 부모와 자신 스레드들 간의 통신 방법은 반복되는 문제입니다. 고급 API는 ØMQ 메시지 파이프를 제공해야 합니다(자동으로 PAIR 소켓과 inproc 사용).
- 이식 가능한 시간.
- 시간을 밀리초 단위의 해상도나 수 밀리초 간 대기 설정하는 것은 이식성이 없습니다. 현실적인 ØMQ 응용프로그램은 이식 가능한 시간이 필요하며, 고급 API에서 제공해야 합니다.
- `zmq_poll()`을 대체할 리엑터.
- 폴링 루프는 간단하지만 어색합니다. 이것들을 많이 사용하면 동일한 작업을 반복해서 수행합니다 : 타이머를 계산, 소켓이 준비되면 코드를 호출. 소켓 리더와 타이머가 있는 소켓 리엑터로 많은 반복 작업을 줄일 수 있습니다.
- Ctrl-C의 적절한 처리.
- 우리는 이미 인터럽트를 처리 방법을 보았습니다. 이러한 처리가 모든 응용프로그램에서 가능하면 유용합니다.

0.36.2 CZMQ 고급 API

요구 사항을 실제 C 언어에 구현하면 ØMQ 언어 바인딩인 CZMQ가 됩니다. 사실이 고급 바인딩(`zhelpers.h`)은 이전 버전의 예제에서도 개발되었습니다. ØMQ로 이식 가능한 계층을 통해 쉬운 사용과 해쉬 테이블 및 목록과 같은 자료 구조를 가진 컨테이너(C 개발언어)도 제공합니다. CZMQ는 또한 우아한 객체 모델 사용하여 멋진 코딩으로 이끌어줍니다.

- [옮긴이] ØMQ 4.3.2에서 CZMQ API을 사용하기 위하여 CZMQ 4.0.2가 존재하지만 가이드의 예제를 수행하기에는 기능이 변경되어 사용하기 어렵습니다(예 : zthread가 zactor로 기능이 승계되었지만 사용 방법이 다름). CZMQ 3.0.2를 설치할 경우 가이드 예제들 수행 가능합니다. CZMQ 3.0.2 소스는(https://github.com/zeromq/czmq/releases/tag/v3.0.2)에서 받을 수 있으며 Visual Studio 2017에서 빌드(czmq.dll, czmq.lib 생성)하여 테스트 수행합니다.

다음은 ØMQ 고급 API(C의 경우 CZMQ)를 사용하여 재작성된 부하 분산 브로커입니다.

lbbroker2.c: CZMQ을 사용한 부하 분산 브로커

```
//  Load-balancing broker
//  Demonstrates use of the CZMQ API

#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "READY"      //  Signals worker is ready

//  Basic request-reply client using REQ socket
//
static void *
client_task(void *args)
{
    zctx_t *ctx = zctx_new();
    void *client = zsocket_new(ctx, ZMQ_REQ);

#if (defined (WIN32))
    zsocket_connect(client, "tcp://localhost:5672"); // frontend
#else
    zsocket_connect(client, "ipc://frontend.ipc");
#endif

    //  Send request, get reply
    zstr_send(client, "HELLO");
    char *reply = zstr_recv(client);
    if (reply) {
        printf("Client: %s\n", reply);
```

```
        free(reply);
    }

    zctx_destroy(&ctx);
    return NULL;
}

//  Worker using REQ socket to do load-balancing
//
static void *
worker_task(void *args)
{
    zctx_t *ctx = zctx_new();
    void *worker = zsocket_new(ctx, ZMQ_REQ);

#if (defined (WIN32))
    zsocket_connect(worker, "tcp://localhost:5673"); // backend
#else
    zsocket_connect(worker, "ipc://backend.ipc");
#endif

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new(WORKER_READY, strlen(WORKER_READY));
    zframe_send(&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv(worker);
        if (!msg)
```

```
            break;              //  Interrupted
        zframe_print(zmsg_last(msg), "Worker: ");
        zframe_reset(zmsg_last(msg), "OK", 2);
        zmsg_send(&msg, worker);
    }
    zctx_destroy(&ctx);
    return NULL;
}

//  .split main task
//  Now we come to the main task. This has the identical functionality to
//  the previous {{lbbroker}} broker example, but uses CZMQ to start child
//  threads, to hold the list of workers, and to read and send messages:

int main(void)
{
    zctx_t *ctx = zctx_new();
    void *frontend = zsocket_new(ctx, ZMQ_ROUTER);
    void *backend = zsocket_new(ctx, ZMQ_ROUTER);

    // IPC doesn't yet work on MS Windows.
#if (defined (WIN32))
    zsocket_bind(frontend, "tcp://*:5672");
    zsocket_bind(backend, "tcp://*:5673");
#else
    zsocket_bind(frontend, "ipc://frontend.ipc");
    zsocket_bind(backend, "ipc://backend.ipc");
#endif
```

```
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new(client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new(worker_task, NULL);

    //  Queue of available workers
    zlist_t *workers = zlist_new();

    //  .split main load-balancer loop
    //  Here is the main loop for the load balancer. It works the same way
    //  as the previous example, but is a lot shorter because CZMQ gives
    //  us an API that does more with fewer calls:
    while (true) {
        zmq_pollitem_t items[] = {
                { backend, 0, ZMQ_POLLIN, 0 },
                { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll(items, zlist_size(workers) ? 2 : 1, -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle worker activity on backend
        if (items[0].revents & ZMQ_POLLIN) {
            //  Use worker identity for load-balancing
            zmsg_t *msg = zmsg_recv(backend);
            if (!msg)
```

```
            break;          //  Interrupted

#if 0
        // zmsg_unwrap is DEPRECATED as over-engineered, poor style
        zframe_t *identity = zmsg_unwrap(msg);
#else
        zframe_t *identity = zmsg_pop(msg);
        zframe_t *delimiter = zmsg_pop(msg);
        zframe_destroy(&delimiter);
#endif

        zlist_append(workers, identity);

        //  Forward message to client if it's not a READY
        zframe_t *frame = zmsg_first(msg);
        if (memcmp(zframe_data(frame), WORKER_READY, strlen(WORKER_READY)) == 0) {
            zmsg_destroy(&msg);
        } else {
            zmsg_send(&msg, frontend);
            if (--client_nbr == 0)
                break; // Exit after N messages
        }
    }
    if (items[1].revents & ZMQ_POLLIN) {
        //  Get client request, route to first available worker
        zmsg_t *msg = zmsg_recv(frontend);
        if (msg) {
#if 0
            // zmsg_wrap is DEPRECATED as unsafe
```

```
                zmsg_wrap(msg, (zframe_t *)zlist_pop(workers));
#else
                zmsg_pushmem(msg, NULL, 0); // delimiter
                zmsg_push(msg, (zframe_t *)zlist_pop(workers));
#endif

                zmsg_send(&msg, backend);
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size(workers)) {
        zframe_t *frame = (zframe_t *)zlist_pop(workers);
        zframe_destroy(&frame);
    }
    zlist_destroy(&workers);
    zctx_destroy(&ctx);
    return 0;
}
```

CZMQ가 제공하는 기능 중 하나는 깔끔함 인터럽트 처리입니다. Ctrl-C는 차단 ØMQ API 호출이 반환 코드 -1 및 EINTR로 errno을 설정하고 종료하게 합니다. 이러한 경우 CZMQ의 `recv()` 메서드는 NULL을 반환하여 다음과 같이 루프를 깔끔하게 종료할 수 있습니다.

```
while (true) {
    zstr_send (client, "Hello");
    char *reply = zstr_recv (client);
    if (!reply)
        break; // Interrupted
    printf ("Client: %s\n", reply);
```

```
    free (reply);
    sleep (1);
}
```

또는 zmq_poll()을 호출하여 반환 코드를 테스트합니다.

```
if (zmq_poll (items, 2, 1000 * 1000) == -1)
    break; // Interrupted
```

- [옮긴이] lbbroker2에서의 zmq_poll() 인터럽트 처리는 다음과 같습니다.

```
        //  Poll frontend only if we have available workers
        int rc = zmq_poll(items, zlist_size(workers) ? 2 : 1, -1);
        if (rc == -1)
            break;              //  Interrupted
```

이전 예제는 여전히 zmq_poll()을 사용합니다. 그렇다면 리엑터는 어떻게 된 것일까요? CZMQ zloop 리엑터는 간단하지만 기능적이며 다음의 것을 수행할 수 있습니다.

- 소켓에 처리 함수(zloop_reader())를 설정합니다. 즉, 소켓에 입력이 있을 때마다 호출되는 코드입니다.
- 소켓에서 처리 함수(zloop_reader())를 취소합니다.
- 특정 간격으로 한번 또는 여러 번 꺼지는 타이머를 설정합니다.
- 타이머를 취소합니다.

zloop는 내부적으로 zmq_poll()을 사용합니다. 처리 함수(zloop_reader())를 추가하거나 제거할 때마다 폴링 세트를 재구축되고, 다음 타이머와 일치하도록 폴링 제한시간을 계산합니다. 그리고 주의가 필요한 각 소켓 및 타이머에 대한 처리 함수(zloop_reader())와 타이머 함수(zloop_timer())를 호출합니다.

리액터 패턴을 사용하면 기존 코드가 변경되며 루프(while(true))가 제거됩니다.. 주요 논리는 다음과 같습니다.

```
zloop_t *reactor = zloop_new ();
zloop_reader (reactor, self->backend, s_handle_backend, self);
zloop_start (reactor);
zloop_destroy (&reactor);
```

메시지의 실제 처리는 전용 함수 또는 메서드 내에 있으며(s_handle_frontend(), s_handle_backend()) 이런 형태로 마음에 들지 않을 수 있습니다 : 그것은 취향의 문제입니다. 이 패턴은 타이머 처리와 소켓의 처리가 섞여 있는 경우에 도움이 됩니다. 이 책의 예제에서 간단한 경우 zmq_poll()을 사용하고 복잡한 경우 zloop를 사용하겠습니다.

아래 예제는 재작성된 부하 분산 브로커입니다. 이번에는 zloop를 사용합니다.

lbbroker3.c: zloop을 사용한 부하 분산 브로커

```
//  Load-balancing broker
//  Demonstrates use of the CZMQ API and reactor style
//
//  The client and worker tasks are identical from the previous example.
//  .skip

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      //  Signals worker is ready

//  Basic request-reply client using REQ socket
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
```

```
    zsocket_connect (client, "ipc://frontend.ipc");

    //  Send request, get reply
    while (true) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        s_sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  Worker using REQ socket to do load-balancing
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);
```

```
    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .until
//  Our load-balancer structure, passed to reactor handlers
typedef struct {
    void *frontend;             //  Listen to clients
    void *backend;              //  Listen to workers
    zlist_t *workers;           //  List of ready workers
} lbbroker_t;

//  .split reactor design
//  In the reactor design, each time a message arrives on a socket, the
//  reactor passes it to a handler function. We have two handlers; one
//  for the frontend, one for the backend:

//  Handle input from client, on frontend
int s_handle_frontend (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
```

```
    lbbroker_t *self = (lbbroker_t *) arg;
    zmsg_t *msg = zmsg_recv (self->frontend);
    if (msg) {
        zmsg_wrap (msg, (zframe_t *) zlist_pop (self->workers));
        zmsg_send (&msg, self->backend);

        //  Cancel reader on frontend if we went from 1 to 0 workers
        if (zlist_size (self->workers) == 0) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
            zloop_poller_end (loop, &poller);
        }
    }
    return 0;
}

//  Handle input from worker, on backend
int s_handle_backend (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    //  Use worker identity for load-balancing
    lbbroker_t *self = (lbbroker_t *) arg;
    zmsg_t *msg = zmsg_recv (self->backend);
    if (msg) {
        zframe_t *identity = zmsg_unwrap (msg);
        zlist_append (self->workers, identity);

        //  Enable reader on frontend if we went from 0 to 1 workers
        if (zlist_size (self->workers) == 1) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
            zloop_poller (loop, &poller, s_handle_frontend, self);
```

```
        }
        //  Forward message to client if it's not a READY
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
            zmsg_destroy (&msg);
        else
            zmsg_send (&msg, self->frontend);
    }
    return 0;
}

//  .split main task
//  And the main task now sets up child tasks, then starts its reactor.
//  If you press Ctrl-C, the reactor exits and the main task shuts down.
//  Because the reactor is a CZMQ class, this example may not translate
//  into all languages equally well.

int main (void)
{
    zctx_t *ctx = zctx_new ();
    lbbroker_t *self = (lbbroker_t *) zmalloc (sizeof (lbbroker_t));
    self->frontend = zsocket_new (ctx, ZMQ_ROUTER);
    self->backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (self->frontend, "ipc://frontend.ipc");
    zsocket_bind (self->backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);
```

```
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  Queue of available workers
    self->workers = zlist_new ();

    //  Prepare reactor and fire it up
    zloop_t *reactor = zloop_new ();
    zmq_pollitem_t poller = { self->backend, 0, ZMQ_POLLIN };
    zloop_poller (reactor, &poller, s_handle_backend, self);
    zloop_start  (reactor);
    zloop_destroy (&reactor);

    //  When we're done, clean up properly
    while (zlist_size (self->workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (self->workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&self->workers);
    zctx_destroy (&ctx);
    free (self);
    return 0;
}
```

- [옮긴이] 위의 코드에서는 ipc(프로세스 간) 소켓을 사용하였지만 윈도우 환경에서는 사용 불가하여 TCP 변경하고 `sleep()`함수도 `czmq.h`의 `zclock_sleep()`을 사용하도록 변경하였습니다.

```
//  Load-balancing broker
//  Demonstrates use of the CZMQ API and reactor style
//
//  The client and worker tasks are identical from the previous example.
//  .skip

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      //  Signals worker is ready

//  Basic request-reply client using REQ socket
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "tcp://localhost:5672");

    //  Send request, get reply
    while (true) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        zclock_sleep(1000);
```

```
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  Worker using REQ socket to do load-balancing
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "tcp://localhost:5673");

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
        zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
```

```
}

//  .until
//  Our load-balancer structure, passed to reactor handlers
typedef struct {
    void *frontend;             //  Listen to clients
    void *backend;              //  Listen to workers
    zlist_t *workers;           //  List of ready workers
} lbbroker_t;

//  .split reactor design
//  In the reactor design, each time a message arrives on a socket, the
//  reactor passes it to a handler function. We have two handlers; one
//  for the frontend, one for the backend:

//  Handle input from client, on frontend
int s_handle_frontend (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    lbbroker_t *self = (lbbroker_t *) arg;
    zmsg_t *msg = zmsg_recv (self->frontend);
    if (msg) {
        zmsg_wrap (msg, (zframe_t *) zlist_pop (self->workers));
        zmsg_send (&msg, self->backend);

        //  Cancel reader on frontend if we went from 1 to 0 workers
        if (zlist_size (self->workers) == 0) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
            zloop_poller_end (loop, &poller);
        }
```

```
    }
    return 0;
}

//  Handle input from worker, on backend
int s_handle_backend (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    //  Use worker identity for load-balancing
    lbbroker_t *self = (lbbroker_t *) arg;
    zmsg_t *msg = zmsg_recv (self->backend);
    if (msg) {
        zframe_t *identity = zmsg_unwrap (msg);
        zlist_append (self->workers, identity);

        //  Enable reader on frontend if we went from 0 to 1 workers
        if (zlist_size (self->workers) == 1) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
            zloop_poller (loop, &poller, s_handle_frontend, self);
        }
        //  Forward message to client if it's not a READY
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
            zmsg_destroy (&msg);
        else
            zmsg_send (&msg, self->frontend);
    }
    return 0;
}
```

```
//  .split main task
//  And the main task now sets up child tasks, then starts its reactor.
//  If you press Ctrl-C, the reactor exits and the main task shuts down.
//  Because the reactor is a CZMQ class, this example may not translate
//  into all languages equally well.

int main (void)
{
    zctx_t *ctx = zctx_new ();
    lbbroker_t *self = (lbbroker_t *) zmalloc (sizeof (lbbroker_t));
    self->frontend = zsocket_new (ctx, ZMQ_ROUTER);
    self->backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (self->frontend, "tcp://*:5672");
    zsocket_bind (self->backend, "tcp://*:5673");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  Queue of available workers
    self->workers = zlist_new ();

    //  Prepare reactor and fire it up
    zloop_t *reactor = zloop_new ();
    zmq_pollitem_t poller = { self->backend, 0, ZMQ_POLLIN };
    zloop_poller (reactor, &poller, s_handle_backend, self);
```

```
    zloop_start  (reactor);
    zloop_destroy (&reactor);

    //  When we're done, clean up properly
    while (zlist_size (self->workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (self->workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&self->workers);
    zctx_destroy (&ctx);
    free (self);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc lbbroker3.c libzmq.lib czmq.lib
 ./lbbroker3
D: 20-08-11 16:06:07 Worker: [005] HELLO
D: 20-08-11 16:06:07 Worker: [005] HELLO
Client: OK
D: 20-08-11 16:06:07 Worker: [005] HELLO
Client: OK
D: 20-08-11 16:06:07 Worker: [005] HELLO
Client: OK
D: 20-08-11 16:06:07 Worker: [005] HELLO
Client: OK
D: 20-08-11 16:06:07 Worker: [005] HELLO
Client: OK
Client: OK
```

```
...
```

응용프로그램에 Ctrl-C로 보낼 때 제대로 종료되도록 하는 것은 까다로울 수 있습니다. zctx 클래스를 사용하면 자동으로 신호 처리가 설정하지만 코드는 여전히 협력해야 합니다. `zmq_poll()`이 -1을 반환하거나 `zstr_recv()`, `zframe_recv()` 또는 `zmsg_recv()` 메서드가 NULL을 반환하는 경우 모든 루프를 중단해야 합니다. 중첩 루프가 있는 경우 바깥쪽 루프의 조건문으로 `!zctx_interrupted` 설정하는 것도 유용합니다.

자식 스레드들을 사용하는 경우, 자식 스레드들은 인터럽트를 받지 못하기 때문에 종료하기 위하여 다음 중 하나를 수행할 수 있습니다.

- 자식 스레드들과 동일한 컨텍스트를 공유(예 : PAIR 소켓)하는 경우 컨텍스트를 파기합니다. 이 경우 대기중인 모든 차단 호출은 ETERM으로 끝납니다.
- 자식 스레드들이 자신의 컨텍스트를 사용하는 경우 종료 메시지를 보냅니다. 이를 위해서는 소켓 간의 연결이 필요합니다.

0.37 비동기 클라이언트/서버 패턴

ROUTER에서 DEALER 예제에서 하나의 서버가 여러 작업자들과 비동기적으로 통신하는 1 대 N 사용 사례(mtserver와 hwclient(REQ) 사례)를 보았습니다. 역으로 다양한 클라이언트들이 단일 서버와 비동기식으로 통신하고 매우 유용한 N-to-1 아키텍처를 사용할 수 있습니다.

그림 37 - 비동기 클라이언트/서버(Asynchronous Client/Server)

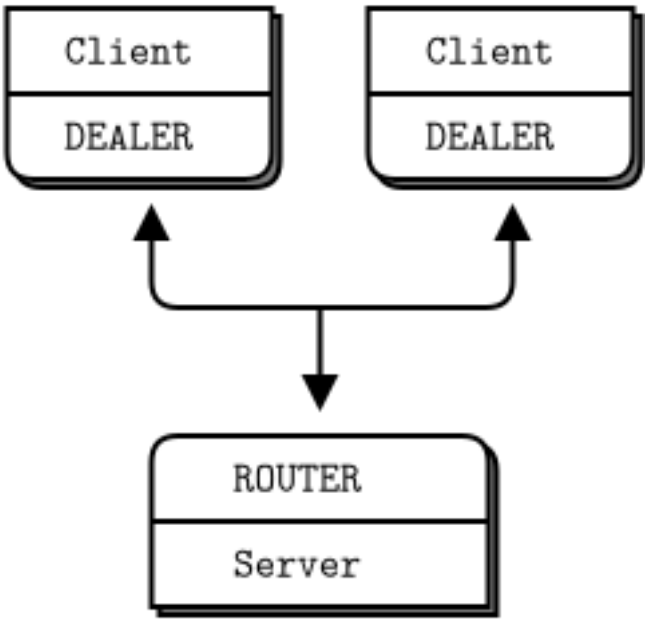


그림 38: Asynchronous Client/Server

동작 방식은 다음과 같습니다.

- 클라이언트는 서버에 연결하여 요청을 보냅니다.
- 각 요청에 대해 서버는 0개 이상의 응답을 보냅니다.
- 클라이언트는 응답을 기다리지 않고 여러 요청들을 보낼 수 있습니다.
- 서버는 새로운 요청을 기다리지 않고 여러 응답들을 보낼 수 있습니다.

동작 방식에 대한 소스 코드는 다음과 같습니다.

asyncsrv.c: 비동기 클라이언트(N)/서버(1)

```
//  Asynchronous client-to-server (DEALER to ROUTER)
//
//  While this example runs in a single process, that is to make
//  it easier to start and stop the example. Each task has its own
//  context and conceptually acts as a separate process.
```

```
#include "czmq.h"

//  This is our client task
//  It connects to the server, and then sends a request once per second
//  It collects responses as they arrive, and it prints them out. We will
//  run several client tasks in parallel, each with a different random ID.

static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_DEALER);

    //  Set random identity to make tracing easier
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    zsocket_set_identity (client, identity);
    zsocket_connect (client, "tcp://localhost:5570");

    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    int request_nbr = 0;
    while (true) {
        //  Tick once per second, pulling in arriving messages
        int centitick;
        for (centitick = 0; centitick < 100; centitick++) {
            zmq_poll (items, 1, 10 * ZMQ_POLL_MSEC);
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (client);
```

```
                zframe_print (zmsg_last (msg), identity);
                zmsg_destroy (&msg);
            }
        }
        zstr_sendf (client, "request #%d", ++request_nbr);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split server task
//  This is our server task.
//  It uses the multithreaded server model to deal requests out to a pool
//  of workers and route replies back to clients. One worker can handle
//  one request at a time but one client can talk to multiple workers at
//  once.

static void server_worker (void *args, zctx_t *ctx, void *pipe);

void *server_task (void *args)
{
    //  Frontend socket talks to clients over TCP
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5570");

    //  Backend socket talks to workers over inproc
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://backend");
```

```
    //  Launch pool of worker threads, precise number is not critical
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++)
        zthread_fork (ctx, server_worker, NULL);

    //  Connect backend to frontend via a proxy
    zmq_proxy (frontend, backend, NULL);

    zctx_destroy (&ctx);
    return NULL;
}

//  .split worker task
//  Each worker task works on one request at a time and sends a random number
//  of replies back, with random delays between replies:

static void
server_worker (void *args, zctx_t *ctx, void *pipe)
{
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "inproc://backend");

    while (true) {
        //  The DEALER socket gives us the reply envelope and message
        zmsg_t *msg = zmsg_recv (worker);
        zframe_t *identity = zmsg_pop (msg);
        zframe_t *content = zmsg_pop (msg);
        assert (content);
```

```
        zmsg_destroy (&msg);

        //  Send 0..4 replies back
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            //  Sleep for some fraction of a second
            zclock_sleep (randof (1000) + 1);
            zframe_send (&identity, worker, ZFRAME_REUSE + ZFRAME_MORE);
            zframe_send (&content, worker, ZFRAME_REUSE);
        }
        zframe_destroy (&identity);
        zframe_destroy (&content);
    }
}

//  The main thread simply starts several clients and a server, and then
//  waits for the server to finish.

int main (void)
{
    zthread_new (client_task, NULL);
    zthread_new (client_task, NULL);
    zthread_new (client_task, NULL);
    zthread_new (server_task, NULL);
    zclock_sleep (5 * 1000);    //  Run for 5 seconds then quit
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
cl -EHsc asyncsrv.c libzmq.lib czmq.lib

./asyncsrv
```

- [옮긴이] 윈도우 환경에서 테스트 수행 시 화면상에 아무것도 찍히지 않고 5초 후에 종료됩니다. 원인은 main()에서 클라이언트 스레드 호출 시 동시에 호출하였기 때문입니다.

```
int main (void)
{
    zthread_new (client_task, NULL);
    zthread_new (client_task, NULL);
    zthread_new (client_task, NULL);
    ...
```

클라이언트의 아래 randof() 함수에서 ID가 동일하게 생성되었기 때문입니다.

```
    ...
    //  Set random identity to make tracing easier
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    printf("client_task[%s]\n", identity);
    ...
```

```
./asyncsrv
client_task[9046-0052]
client_task[9046-0052]
client_task[9046-0052]
```

위의 추가로 srand() 함수를 시간을 통하여 randof() 초기화를 수행하여도 클라이언트 스레드가 동시에 수행될 경우 같은 값이 되는 경우가 발생하였습니다.

```
#include "zhelpers.h"
    ...
    //  Set random identity to make tracing easier
    char identity [10];
    srand(s_clock());
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    printf("client_task[%s]\n", identity);
    ...
```

아래 실행 결과에서는 2개의 스레드가 같은 ID가 되었습니다.

```
 ./asyncsrv
client_task[C0B6-F58A]
client_task[C0B6-F58A]
client_task[14B0-F590]
D: 20-08-12 14:31:24 C0B6-F58A[010] request #3
```

이를 해결하기 위하여 client thread 실행 시에 100 msec의 시간 간격을 두고 시작하게 합니다.

```
int main (void)
{
    zthread_new (client_task, NULL);
    s_sleep(100);
    zthread_new (client_task, NULL);
    s_sleep(100);
    zthread_new (client_task, NULL);
    s_sleep(100);
    ...
```

테스트 수행 결과는 다음과 같습니다.

```
 ./asyncsrv
client_task[BEC0-E8C8]
client_task[87DC-EB6A]
client_task[B95C-EDAE]
D: 20-08-12 14:37:22 B95C-EDAE[010] request #2
D: 20-08-12 14:37:23 B95C-EDAE[010] request #2
D: 20-08-12 14:37:24 BEC0-E8C8[010] request #3
D: 20-08-12 14:37:24 87DC-EB6A[010] request #3
D: 20-08-12 14:37:24 B95C-EDAE[010] request #3
```

- [옮긴이] 수정된 전체 aysncsrv.c 전체 코드는 다음과 같습니다.

```
//  Asynchronous client-to-server (DEALER to ROUTER)
//
//  While this example runs in a single process, that is to make
//  it easier to start and stop the example. Each task has its own
//  context and conceptually acts as a separate process.

#include "czmq.h"
#include "zhelpers.h"

//  This is our client task
//  It connects to the server, and then sends a request once per second
//  It collects responses as they arrive, and it prints them out. We will
//  run several client tasks in parallel, each with a different random ID.

static void *
client_task (void *args)
{
```

```
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_DEALER);

    //  Set random identity to make tracing easier
    char identity [10];
    srand(s_clock());
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    printf("client_task[%s]\n", identity);
    zsocket_set_identity (client, identity);
    zsocket_connect (client, "tcp://localhost:5570");

    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    int request_nbr = 0;
    while (true) {
        //  Tick once per second, pulling in arriving messages
        int centitick;
        for (centitick = 0; centitick < 100; centitick++) {
            zmq_poll (items, 1, 10 * ZMQ_POLL_MSEC);
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (client);
                zframe_print (zmsg_last (msg), identity);
                zmsg_destroy (&msg);
            }
        }
        zstr_sendf (client, "request #%d", ++request_nbr);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

```
//  .split server task
//  This is our server task.
//  It uses the multithreaded server model to deal requests out to a pool
//  of workers and route replies back to clients. One worker can handle
//  one request at a time but one client can talk to multiple workers at
//  once.

static void server_worker (void *args, zctx_t *ctx, void *pipe);

void *server_task (void *args)
{
    //  Frontend socket talks to clients over TCP
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5570");

    //  Backend socket talks to workers over inproc
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://backend");

    //  Launch pool of worker threads, precise number is not critical
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++)
        zthread_fork (ctx, server_worker, NULL);

    //  Connect backend to frontend via a proxy
    zmq_proxy (frontend, backend, NULL);
```

```
    zctx_destroy (&ctx);
    return NULL;
}

//  .split worker task
//  Each worker task works on one request at a time and sends a random number
//  of replies back, with random delays between replies:

static void
server_worker (void *args, zctx_t *ctx, void *pipe)
{
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "inproc://backend");

    while (true) {
        //  The DEALER socket gives us the reply envelope and message
        zmsg_t *msg = zmsg_recv (worker);
        zframe_t *identity = zmsg_pop (msg);
        zframe_t *content = zmsg_pop (msg);
        assert (content);
        zmsg_destroy (&msg);

        //  Send 0..4 replies back
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            //  Sleep for some fraction of a second
            zclock_sleep (randof (1000) + 1);
            zframe_send (&identity, worker, ZFRAME_REUSE + ZFRAME_MORE);
            zframe_send (&content, worker, ZFRAME_REUSE);
```

```
        }
        zframe_destroy (&identity);
        zframe_destroy (&content);
    }
}

//  The main thread simply starts several clients and a server, and then
//  waits for the server to finish.

int main (void)
{
    zthread_new (client_task, NULL);
    s_sleep(100);
    zthread_new (client_task, NULL);
    s_sleep(100);
    zthread_new (client_task, NULL);
    s_sleep(100);
    zthread_new (server_task, NULL);
    zclock_sleep (5 * 1000);    //  Run for 5 seconds then quit
    return 0;
}
```

이 예제는 다중 프로세스 아키텍처를 시뮬레이션하여 멀티스레드를 사용하여 하나의 프로세스에서 실행됩니다. 예제를 실행하면 서버에서 받은 응답을 출력하는 3개의 클라이언트(각각 임의의 식별자(ID)가 있음)가 표시됩니다. 주의 깊게 살펴보면 각 클라이언트 작업이 요청당 0개 이상의 응답을 받는 것을 볼 수 있습니다.

이 코드에 대한 설명입니다.

- 클라이언트는 초당 한 번씩 요청을 보내고 0개 이상의 응답을받습니다. `zmq_poll()`을 사용하여 작업을 수행하기 위해, 단순히 1초 제한시간으로 폴링할 수 없으며 마지막 응답을 받은 후 1초 후에 새 요청을 보내게 됩니다. 그래서 우리는 높은 빈도(1초당 100

회 폴링(1/100초(10밀리초) 간격))로 폴링합니다. 이는 거의 정확합니다.

- 서버는 작업자 스레드 풀(pool)을 사용하여 각 스레드가 하나의 요청을 동기적으로 처리합니다. 클라이언트(DEALER)는 내부 대기열을 사용하여 프론트엔드 소켓(ROUTER)에 연결하고 작업자(DEALER)도 내부 대기열을 사용하여 백엔드 소켓(DEALER)에 연결합니다. `zmq_proxy()` 호출하여 프런트 엔드와 백엔드 소켓 간에 통신하도록 합니다.

그림 38 - 비동기 서버 상세(Detail of Asynchronous Server)

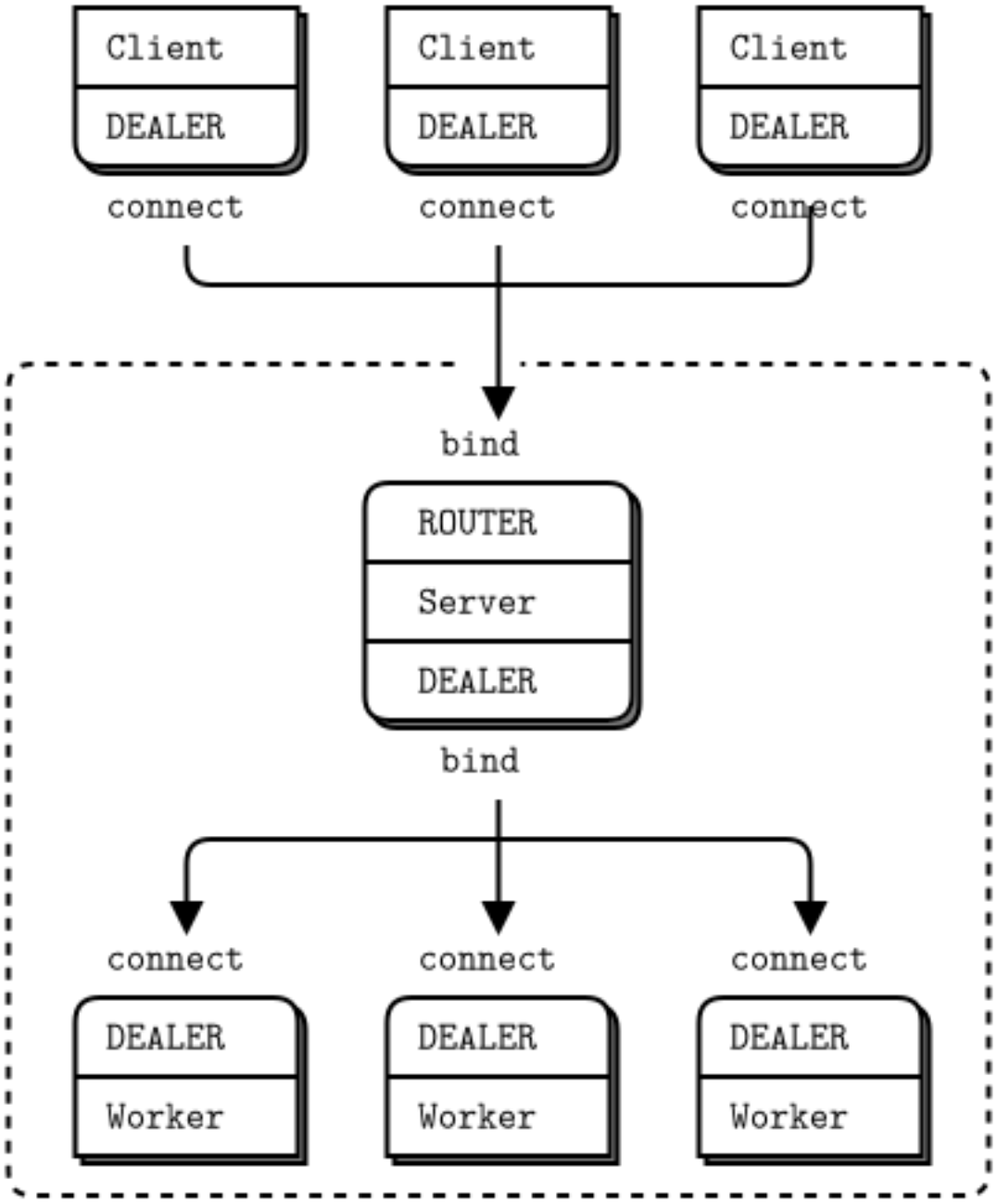


그림 39: Detail of Asynchronous Server

클라이언트와 서버간에 DEALER와 ROUTER 통신을 수행하고 있지만, 서버 메인 스레

드와 작업자 스레드들 간에는 내부적으로 DEALER와 DEALER를 수행하고 있습니다. REP 소켓을 사용했듯이 작업자들은 엄격하게 동기식입니다. 하지만 여러 회신을 보내려고 하기 때문에 비동기 소켓이 필요합니다. 회신들을 각 응답에 대하여 직접 라우팅하지 않기 위해서, 항상 요청을 보낸 단일 서버 스레드로 가게 합니다.

라우팅 봉투에 대해 생각해 봅시다. 클라이언트는 단일 프레임으로 구성된 메시지를 보냅니다. 서버 스레드는 2개 프레임 메시지(CLINT ID + DATA)를 받습니다. 2개 프레임을 작업자에게 보내면 일반 응답 봉투로 취급하고 2개 프레임 메시지(CLINT ID + DATA)로 반환합니다. 그러면 작업자는 첫 번째 프레임을 라우팅할 ID(CLIENT ID)로 두 번째 프레임을 클라이언트에 대한 응답으로 사용합니다.

이것은 아래와 같이 보입니다.

```
   client          server       frontend       worker
[ DEALER ]<---->[ ROUTER <----> DEALER <----> DEALER ]
          1 part         2 parts       2 parts
```

이제 소켓들에 대해 : 부하 분산 ROUTER와 DEALER 패턴을 작업자들 간의 통신에 사용할 수 있었지만 추가 작업이 있습니다. 이 경우 DEALER와 DEALER 패턴은 괜찮겠지만 단점은 각 요청에 대한 지연시간가 짧지만 작업 분산이 평준화되지 않을 위험이 있습니다. 이 경우 단순화시켜 대응합니다.

클라이언트와 통신 상태를 유지하는 서버를 구축할 때 고전적인 문제가 발생합니다. 서버가 클라이언트에 대한 통신 상태를 유지하지만, 클라이언트는 '희미(동적)한 존재(comes and goes)'지만 연결을 유지할 경우 결국 서버 자원은 부족하게 됩니다. 기본 식별자(ID)를 사용하여 동일한 클라이언트들이 계속 연결을 유지하더라도 각 연결은 새로운 연결처럼 보입니다.

위의 예제를 매우 짧은 시간(작업자가 요청을 처리하는 데 걸리는 시간) 동안만 통신 상태를 유지 한 다음 통신 상태를 버리는 방식으로 문제를 해결할 수 있습니다. 그러나 해결안은 많은 경우에 실용적이지 않습니다. 상태 기반 비동기 서버에서 클라이언트 상태를 적절하게 관리하려면 다음의 작업을 수행해야 합니다.

- 일정 시간 간격으로 클라이언트에서 서버로 심박을 보냅니다. 위의 예제에서는 클라이언트에서 1초당 한번 요청을 보냈으며, 심박으로 신뢰할 수 있게 사용할 수 있습니다.

- 클라이언트 식별자(ID)를 키로 사용하여 상태를 저장합니다.
- 클라이언트로부터 중단된 심박을 감지합니다. 클라이언트로부터 일정 시간(예 : 2초) 동안 요청이 없으면 서버는 이를 감지하고 해당 클라이언트에 대해 보유하고 있는 모든 상태를 폐기할 수 있습니다.

0.38 동작 예제 : 브로커 간 라우팅

지금까지 본 모든 것을 가져와 실제 응용프로그램으로 확장해 보겠습니다. 여러 번의 반복을 거쳐 단계별로 구축하겠습니다. 우량(VIP) 고객이 긴급하게 전화를 걸어 대규모 클라우드 컴퓨팅 시설의 설계를 요청합니다. 고객의 클라우드에 대한 비전은 많은 데이터 센터에 퍼져 있는 각 클라이언트들과 작업자들이 클러스터를 통해 하나로 동작하는 있습니다. 우리는 실전이 항상 이론을 능가한다는 것을 알만큼 똑똑하기 때문에 ØMQ를 사용하여 동작하는 시뮬레이션을 만들겠습니다. 우리의 고객은 자신의 상사가 마음을 바꾸기 전에 예산을 확정하기 바라며, 트위터에서 ØMQ에 대한 훌륭한 정보를 읽은 것 같습니다.

0.38.1 상세한 요구 사항

몇 잔의 에스프레소를 마시고 코드 작성에 뛰어들고 싶지만, 전체적으로 잘못된 문제에 대한 놀라운 해결책을 제공하기 전에, 자세한 사항을 확인하라고 마음속에서 무언가의 속삭임 있습니다. 그래서 고객에서 "클라우드로 어떤 일을 하고 싶으시나요?"라고 묻습니다.

고객의 요구사항은 다음과 같습니다.

- 작업자는 다양한 종류의 하드웨어에서 실행되지만, 어떤 작업도 처리할 수 있어야 합니다. 클러스터당 수백 개의 작업자가 있고 대략 12개의 클러스터가 있습니다.
- 클라이언트는 작업자에게 작업을 요청합니다. 각 작업은 독립적인 작업 단위로 클라이언트는 가용한 작업자를 찾아 가능한 한 빨리 작업을 보내려고 합니다. 많은 클라이언트가 존재하며 임의적으로 왔다 갔다 합니다.
- 클라우드에서 진짜 어려운 것은 클러스터를 언제든지 추가하고 제거할 수 있어야 하는 것입니다. 클러스터는 속해 있는 모든 작업자와 클라이언트들을 함께 즉시 클라우드를 떠나거나 합류할 수 있습니다.

- 자체 클러스터에 작업자가 없는 경우, 클라이언트의 작업은 클라우드에서 가용한 다른 클러스터의 작업자에게 전달됩니다.
- 클라이언트는 한 번에 하나의 작업을 보내 응답을 기다립니다. 제한시간(X초) 내에 응답을 받지 못하면 작업을 다시 전송합니다. 이것은 우리의 고려사항은 아니며 클라이언트 API가 이미 수행하고 있습니다.
- 작업자들은 한 번에 하나의 작업을 처리합니다. 그들은 매우 단순하게 작업을 처리합니다. 작업자들의 수행이 중단되면 그들을 기동한 스크립트에 의해 재시작됩니다.

위에서 설명한 것을 제대로 이해했는지 다시 확인합니다.

- "클러스터들 간에 일종의 초고속 네트워크 상호 연결이 있을 것입니다. 맞습니까?" 고객은 "예, 물론 우리는 바보가 아닙니다."라고 말합니다.
- "통신 규모는 어느 정도입니까?"라고 묻습니다. 고객은 "클러스터 당 최대 1,000개의 클라이언트들, 각 클라이언트는 초당 최대 10 개의 요청을 수행합니다. 요청은 작고 응답도 각각 1KB 이하로 작습니다."라고 응답합니다(20 Mbytes=20,000,000 bytes=1(Cluster) * 1,000(clients) * 10(requests) * 1,000(bytes) * 2(send/recv)).

그러면 고객의 요구 사항에 대하여 약간의 계산을 하고 이것이 일반 TCP상에서 작동할지 확인합니다. 클라이언트들 2,500 개 x 10/초(request) x 1,000바이트(date) x 2방향(send/recv) = 50 MBytes/초 또는 400 Mbit/초, 1Gb 대역폭의 TCP 네트워크에는 문제가 되지 않습니다.

이것은 간단한 문제로 특별한 하드웨어나 통신규약들이 필요하지 않고 다소 영리한 라우팅 알고리즘과 신중한 설계만 필요로 합니다. 먼저 하나의 클러스터(하나의 데이터 센터)를 설계하고 클러스터를 함께 연결하는 방법을 알아보겠습니다.

0.38.2 단일 클러스터 아키텍처

작업자들과 클라이언트들은 동기적으로 동작합니다. 부하 분산 패턴을 사용하여 작업들을 작업자들에게 전달하기 원하며 작업자들은 모두 동일한 기능을 수행합니다. 데이터센터에는 작업자는 익명이며 특정 서비스에 대한 개념이 없습니다. 클라이언트들은 직접 주소를 지정

하지 않습니다. 재시도(제한시간(X초) 내에 응답을 받지 못하면 클라이언트에서 작업을 다시 전송)가 자동으로 이루어지므로 통신에 대한 보증은 언급하지 않아도 좋을 것입니다.

우리가 이미 보았듯이 클라이언트들과 작업자들은 서로 직접 통신하지 않습니다. 동적으로 노드들을 추가하거나 제거하는 것이 불가능합니다. 따라서 우리의 기본 모델은 이전에 살펴본 요청-응답 메시지 브로커로 구성됩니다.

그림 39 - 클러스터 아키텍처

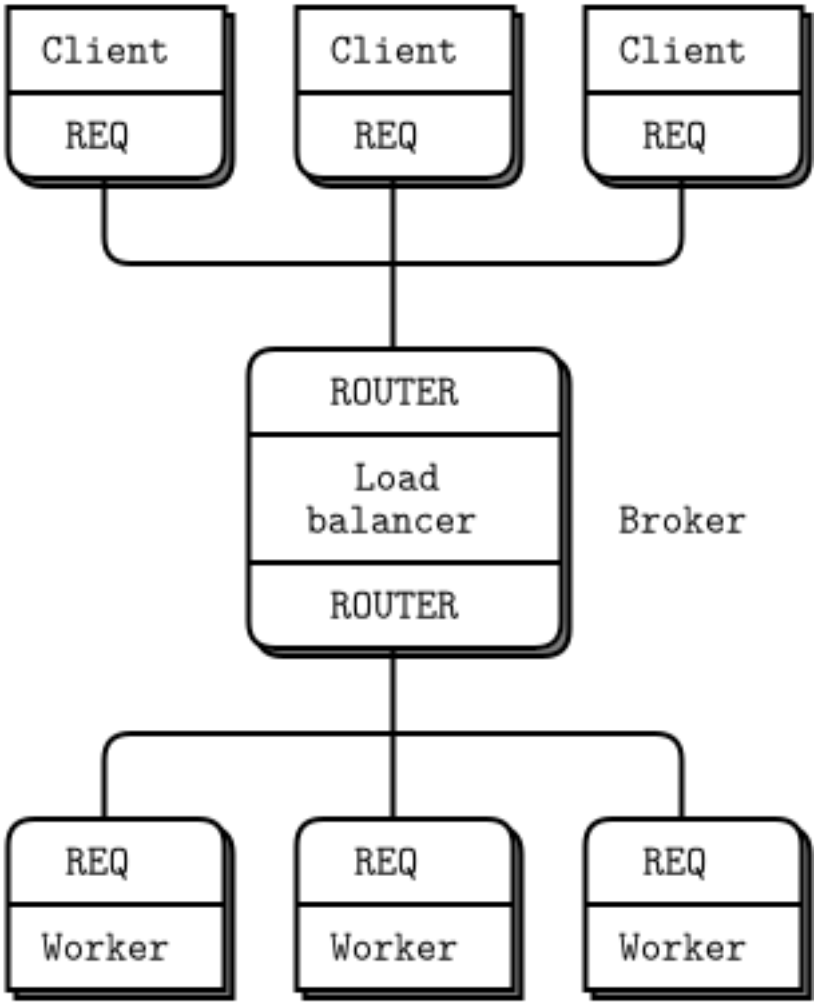


그림 40: Cluster Architecture

0.38.3 다중 클러스터로 확장

이제 하나 이상의 클러스터로 확장합니다. 각 클러스터에는 일련의 클라이언트들 및 작업자들이 있으며 이들을 함께 결합하는 브로커(broker)가 있습니다.

그림 40 - 다중 클러스터

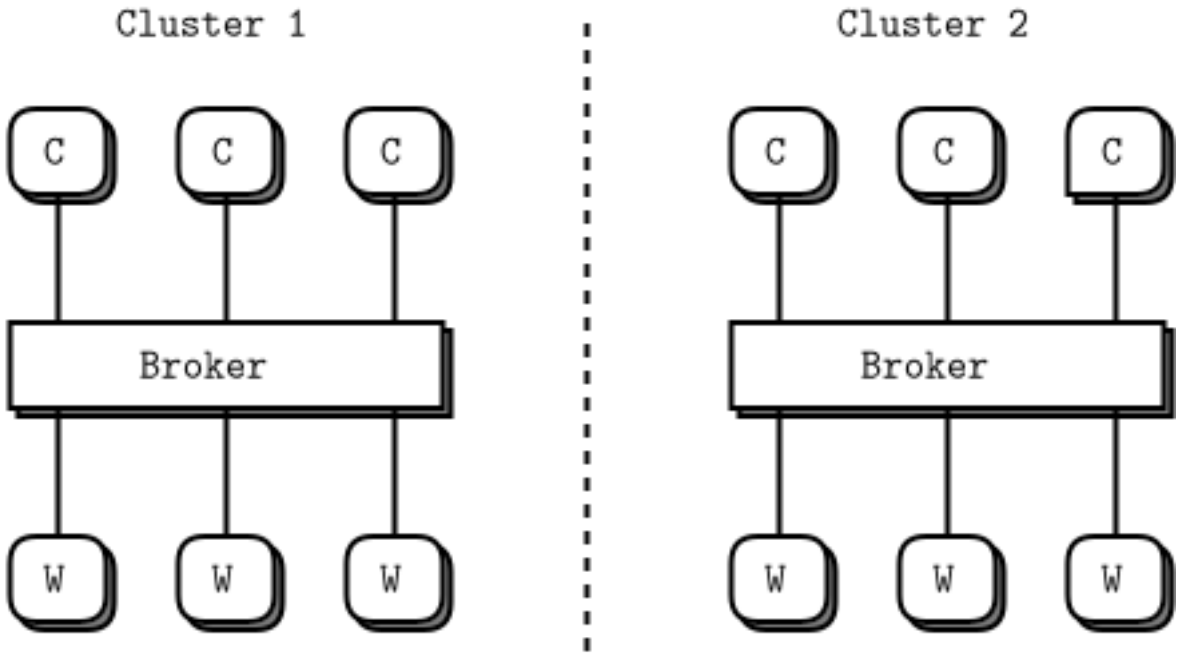


그림 41: Multiple Clusters

여기서 질문입니다 : 각 클러스터에 속해 있는 클라이언트들이 다른 클러스터의 작업자들과 통신하는 방법은 어떻게 될까요? 여기 몇 가지 방법이 있으며 각각 장단점이 있습니다.

- 클라이언트들은 직접 양쪽 브로커에 연결할 수 있습니다. 장점은 브로커들과 작업자들을 수정할 필요가 없습니다. 그러나 클라이언트들은 더 복잡해지고 전체 토폴로지를 알아야 합니다. 예를 들어 세 번째 또는 네 번째 클러스터를 추가하려는 경우 모든 클라이언트들이 영향을 받습니다. 영향으로 클라이언트의 라우팅 및 장애조치 로직을 변경해야 하며 좋지 않습니다.
- 작업자들은 직접 양쪽 브로커에 연결하려 하려 하지만 REQ 작업자는 하나의 브로커에만 응답할 수 있어 그렇게 할 수 없습니다. REP를 사용하려 하지만 REP는 부하

분산처럼 사용자 지정 가능한 브로커와 작업자 간 라우팅을 제공하지 않고 내장된 부하 분산만 제공하여 잘못된 것입니다. 유휴 작업자에게 작업을 분배하려면 정확하게 부하 분산이 필요합니다. 유일한 해결책은 작업자 노드들에 대해 ROUTER 소켓을 사용하는 것입니다. 이것을 "아이디어# 1"로 명명하겠습니다.

- 브로커들은 상호 간에 연결할 수 있습니다. 가장 적은 추가 연결을 생성하여 가장 깔끔해 보입니다. 동적으로 클러스터를 추가하기 어려지만 설계 범위를 벗어난 것 같습니다. 이제 클라이언트들과 작업자들은 실제 네트워크 토폴로지를 모르게 하고 브로커들 간에는 여유 용량이 있을 때 서로에게 알립니다. 이것을 "아이디어 #2"로 명명하겠습니다.

아이디어 #1을 분석해 보겠습니다. 이 모델에서는 작업자들이 양쪽 브로커들에 연결하고 하나의 브로커에서 작업을 수락합니다.

그림 41 - 아이디어#1 : 교차 연결된 작업자

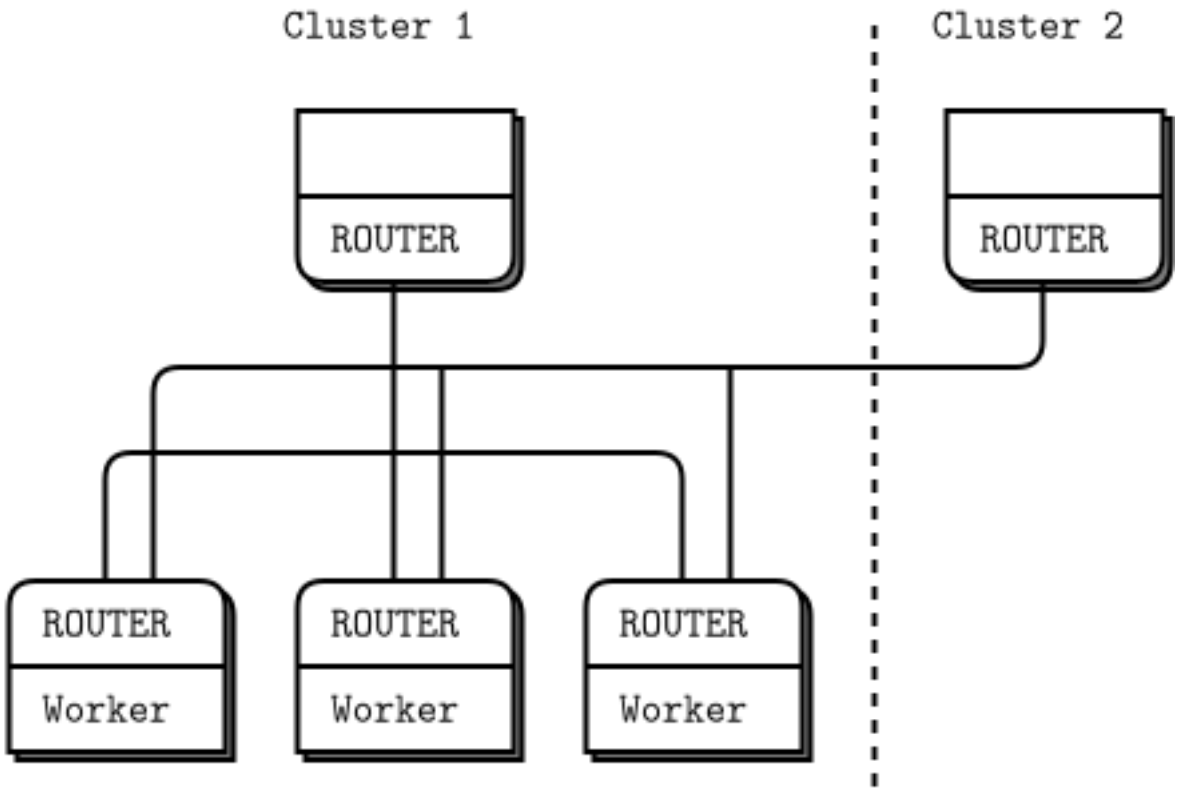


그림 42: Cross-connected Worker

좋은 방법으로 보이지만 우리가 원하는 것을 제공하지 않습니다. 클라이언트들은 가능하면 로컬 작업자들을 얻고 기다리는 것보다 더 나은 경우에만 원격 작업자를 얻습니다. 또한 작업자들은 양쪽 브로커들에게 “준비(READY)”신호를 보내고 다른 작업자는 유휴 상태로 있는 동안 한 번에 두 개의 작업들을 받을 수 있습니다. 이 디자인은 잘못된 것 같으며 원인은 다시 우리가 양쪽 말단에 라우팅 로직 넣어야 하기 때문입니다.

그럼, 아이디어# 2에서 우리는 브로커들을 상호 연결하고 우리가 익숙한 REQ 소켓을 사용하는 클라이언트들과 또는 작업자들을 손대지는 않습니다.

그림 42 - 상호 통신하는 브로커(Broker Talking to Each Other)

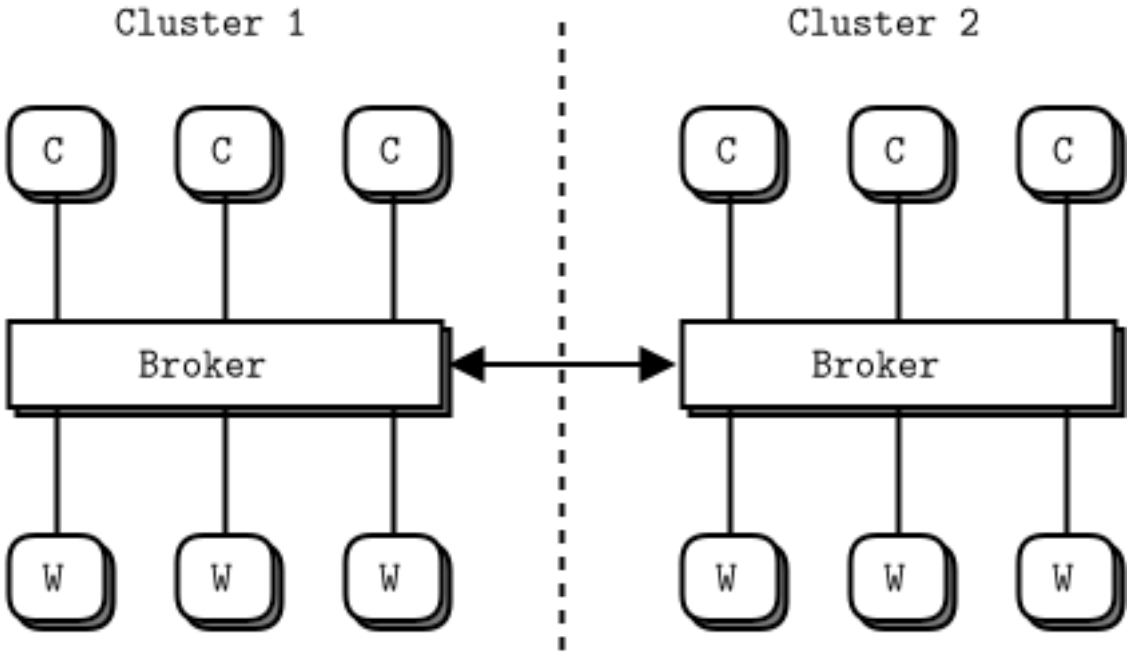


그림 43: Broker Talking to Each Other

이 디자인은 문제가 한 곳에서 해결되고 나머지 것들은 보이지 않기 때문에 매력적입니다. 기본적으로 브로커들은 서로에게 비밀 채널들을 열고 낙타 상인처럼 속삭입니다. “이봐, 나는 여유가 좀 있는데 클라이언트들이 너무 많을 경우 알려 주시면 우리가 대응하겠습니다.”

사실 이것은 더 복잡한 라우팅 알고리즘 일뿐입니다 : 브로커들은 서로를 위해 하청업체가 됩니다. 실제 코드를 작성하기 전에 이와 같은 설계를 좋아할 점들이 있습니다.

- 일반적인 경우(동일한 클러스터의 클라이언트들과 작업자들)를 기본으로 하고 예외적인 경우(클러스터들 간 작업을 섞음)에 대한 추가 작업을 수행합니다.
- 다른 유형의 작업에 대해 다른 메시지 흐름을 사용하게 합니다. 다르게 처리하게 하기 위함으로 예를 들면 서로 다른 유형의 네트워크 연결을 사용합니다.
- 부드럽게 확장할 수 있습니다. 3개 이상의 브로커들간의 상호 연결하는 것은 다소 복잡하게 되지만, 이것이 문제라고 판단되면 하나의 슈퍼 브로커를 추가하여 쉽게 해결할 수 있습니다.

이제 동작하는 예제를 만들겠습니다. 전체 클러스터를 하나의 프로세스로 압축합니다. 분명히 현실적이지는 않지만 시뮬레이션하기에는 단순하게 만들어 시뮬레이션을 정확하게 실제 프로세스들로 확장 할 수 있습니다. 이것이 ØMQ의 아름다움입니다 - 미시 수준에서 설계하고 거시 수준까지 확장 할 수 있습니다. 스레드들은 프로세스들이 된 다음 하드웨어 머신들로 점차 거시적으로 확장 가능 하지만, 패턴들과 논리는 동일하게 유지됩니다. 각 "클러스터" 프로세스들에는 클라이언트 스레드들, 작업자 스레드들 및 브로커 스레드가 포함됩니다.

이제 기본 모델을 잘 알게 되었습니다.

- REQ 클라이언트 스레드들은 작업부하들을 생성하여 브로커(ROUTER)로 전달합니다.
- REQ 작업자 스레드들은 작업부하들을 처리하고 결과들을 브로커(ROUTER)로 반환합니다.
- 브로커는 부하 분산 패턴을 사용하여 작업부하들을 대기열에 넣고 분배합니다.

0.38.4 페더레이션 및 상대 연결

브로커들을 상호 연결하는 방법에는 여러 가지가 있습니다. 우리가 원하는 것은 다른 브로커들에게 "우리는 여유 용량이 있어"라고 말한 다음 여러 작업들을 받는 것입니다. 또한 우리는 다른 브로커들에게 "그만, 우리는 여유 용량이 없어"라고 말할 수 있어야 합니다. 완벽할 필요는 없으며, 때로는 즉시 처리할 수 없는 작업들을 받은 다음 가능한 한 빨리 처리합니다.

가장 쉬운 상호 연결은 연합(Federation)이며, 브로커들이 클라이언트들과 작업자들을 서로 시뮬레이션하는 것입니다. 이를 수행하기 위해 클러스터의 백엔드를 다른 브로커의 프론트엔드 소켓에 연결합니다. 한 소켓을 단말에 바인딩하고 다른 단말에 연결이 모두 가능한지 확인하십시오.

그림 43 - 연합 모델에서 교차 연결된 브로커들

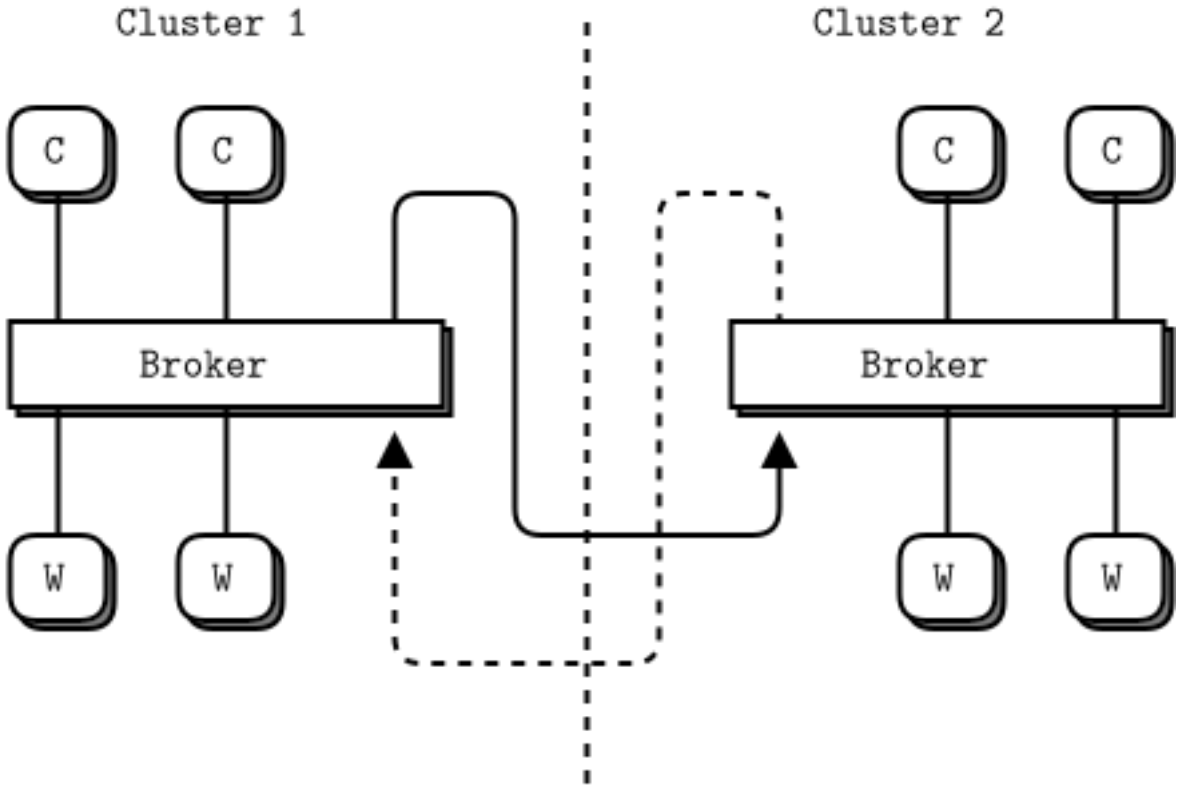


그림 44: Cross-connected Brokers in Federation Model

연합은 브로커들과 타당하고 좋은 처리 방식을 가진 단순한 로직을 제공합니다. 클라이언트들이 없을 때 다른 브로커에게 "준비(READY)"라고 알리고 하나의 작업을 받아들입니다. 유일한 문제는 이것이 너무 단순하다는 것입니다. 연합된 브로커는 한 번에 하나의 작업만 처리할 수 있습니다. 브로커가 잠금 단계 클라이언트와 작업자로 하게 되면 정의상 잠금 단계가 되며, 비록 많은 작업자들이 있어도 동시에 사용할 수 없습니다. 우리의 브로커들은 완전히 비동기적으로 연결되어야 합니다.

- [옮긴이] 잠근 단계 통신규약(lock-step protocol)은 동기식 요청-응답과 같이 클라이언트가 하나의 요청을 하면, 작업자가 요청을 받아 처리하고 응답할 때까지 대기하는 방식입니다. 특정 작업자들에 대하여 응답이 지연되는 현상이 발생하면 시스템의 자원 활용도 및 성능은 저하됩니다. 해결 방법으로는 요청에 대한 응답 지연 시 제한시간을

두거나, 비동기 요청-응답 처리가 있습니다.

페더레이션 모델은 다른 종류의 라우팅, 특히 부하 분산이나 라운드 로빈보다는 서비스 이름 및 근접성에 따라 라우팅하는 서비스 지향 아키텍처들(SOAs)에 적합합니다. 모든 용도에 적응하는 가능한 것은 아니지만 용도에 따라 사용할 수 있습니다.

연합 대신 상대 연결 방식에서는 브로커들이 서로를 명시적으로 인식하고 제한된 채널들을 통해 통신합니다. 상세히 설명하면 N개의 브로커들이 상호 연결한다고 가정하면, 각 브로커에는 (N-1) 개의 상대가 있으며 모든 브로커들은 정확히 동일한 코드와 로직을 사용합니다. 브로커들 간에는 2개의 고유한 정보 흐름이 있습니다.

- [상태 정보] 각 브로커는 언제든지 가용한 작업자 수를 상대들에게 알려야 합니다. 이것은 매우 간단한 정보로 일정 시간 간격으로 변경되는 수량 정보입니다. 발행-구독이 명백하고(정확한) 소켓 패턴입니다. 따라서 모든 브로커들은 PUB 소켓을 열어 상태 정보를 발행하고 모든 브로커들은 SUB 소켓도 열어 다른 모든 브로커들의 PUB 소켓에 연결하여 상대들의 상태 정보를 받습니다.
- [작업 위임/작업 수신] 각 브로커는 작업을 상대에게 위임하고 비동기적으로 응답들을 받을 수 있는 방법이 필요합니다. ROUTER 소켓을 사용하여 작업을 수행하며 다른 조합은 적용할 수 없습니다. 각 브로커에는 2개의 ROUTER 소켓을 가지고 하나는 작업들 수신용이고 다른 하나는 작업들 위임용입니다. 2개의 소켓을 사용하지 않는다면 매번 요청을 읽었는지 응답을 읽었는지에 알기 위해 더 많은 작업이 필요합니다. 이는 메시지 봉투에 더 많은 정보를 추가하는 것을 의미합니다.

그리고 클러스터에는 브로커와 클라이언트들 및 작업자들 간에 정보 흐름이 있습니다.

0.38.5 명명식

2개의 소켓들 x 3개의 흐름들 = 6개의 소켓으로 브로커에서 관리해야 합니다. 다중 소켓으로 다양한 소켓 유형이 섞여 있는 브로커에서 좋은 이름을 선정하는 것은 우리의 마음을 일관성 있게 유지하는 데 중요합니다. 소켓은 그들의 이름을 통해 무엇을 하는지 알 수 있어야 합니다. 몇 주 후에 추운 월요일 아침 커피를 마시기 전에, 코드를 읽으며 고통을 느끼지 않기 위해서입니다.

소켓에 대한 샤머니즘적인 명명식을 하겠습니다. 세 가지 흐름은 다음과 같습니다.

- [옮긴이] 샤머니즘(shamanism)은 초자연적인 존재와 직접적으로 소통하는 샤먼을 중심으로 하는 주술이나 종교입니다.
- [local] 브로커와 클라이언트들 및 작업자들 간의 로컬 요청-응답(Workload flow) 흐름.
- [cloud] 브로커와 상대 브로커들 간의 클라우드 요청-응답(Task deligation) 흐름.
- [state] 브로커와 상대 브로커들 간의 상태 흐름(State flow)

길이가 모두 같은 의미 있는 이름을 찾으면 코드가 잘 정렬되며 중요한 사항은 아니지만 세부 사항에 주의 깊게 살피는데 도움이 됩니다. 브로커의 각 흐름에는 2개의 소켓이 있으며 프론트엔드와 백엔드를 호출입니다. 우리는 이 이름들을 자주 사용했습니다. 프론트엔드는 정보 또는 작업들을 받습니다. 백엔드는 이를 다른 상대들로 보냅니다. 개념적 흐름은 앞에서 뒤로 진행됩니다(응답들은 뒤에서 앞으로 반대 방향으로 진행됩니다.).

본 입문서에서는 작성하는 모든 코드에서 다음 소켓 이름들을 사용합니다.

- 「local」 로컬에 작업부하 처리에 사용되는 localfe와 localbe
- 「cloud」 클라우드의 작업 위임/응답 처리에 사용되는 cloudfe와 cloudbe
- 「state」 브로커들간의 상태(작업자의 개수 전달)를 확인하기 위해 사용되는 statefe와 statebe

전송 방식은 ipc를 사용하는 것은 하나의 박스에서 모든 것을 시뮬레이션하기 때문입니다. 이것은 연결 측면에서 tcp처럼 작동한다는 장점이 있으며(즉, inproc과는 달리 연결이 끊어진 전송방식) 어려울 수 있는 IP 주소나 DNS 이름이 필요하지 않습니다. ipc 단말들로 무언가-로컬(something-local), 무언가-클라우드(something-cloud) 및 무언가-상태(something-state)라는 불리는 것을 사용합니다. 여기서 무언가(something)는 시뮬레이션하는 클러스터의 이름입니다.

이것은 일부 이름들에 대한 과분한 작업이라고 생각할 수 있습니다. 왜 그들을 s1, s2, s3, s4 등으로 부르지 않을까요? 대답은 당신의 두뇌가 완벽한 기계가 아니라면 코드를 읽을 때 많은 도움이 필요하며 우리는 의미 있는 이름들이 도움이 된다는 것을 알게 되었습니다. “6개의 다른 소켓”보다는 “3개의 흐름, 2개의 방향”으로 기억하는 것이 더 쉽습니다.

그림 44 - 브로커 소켓 배열(피어링)

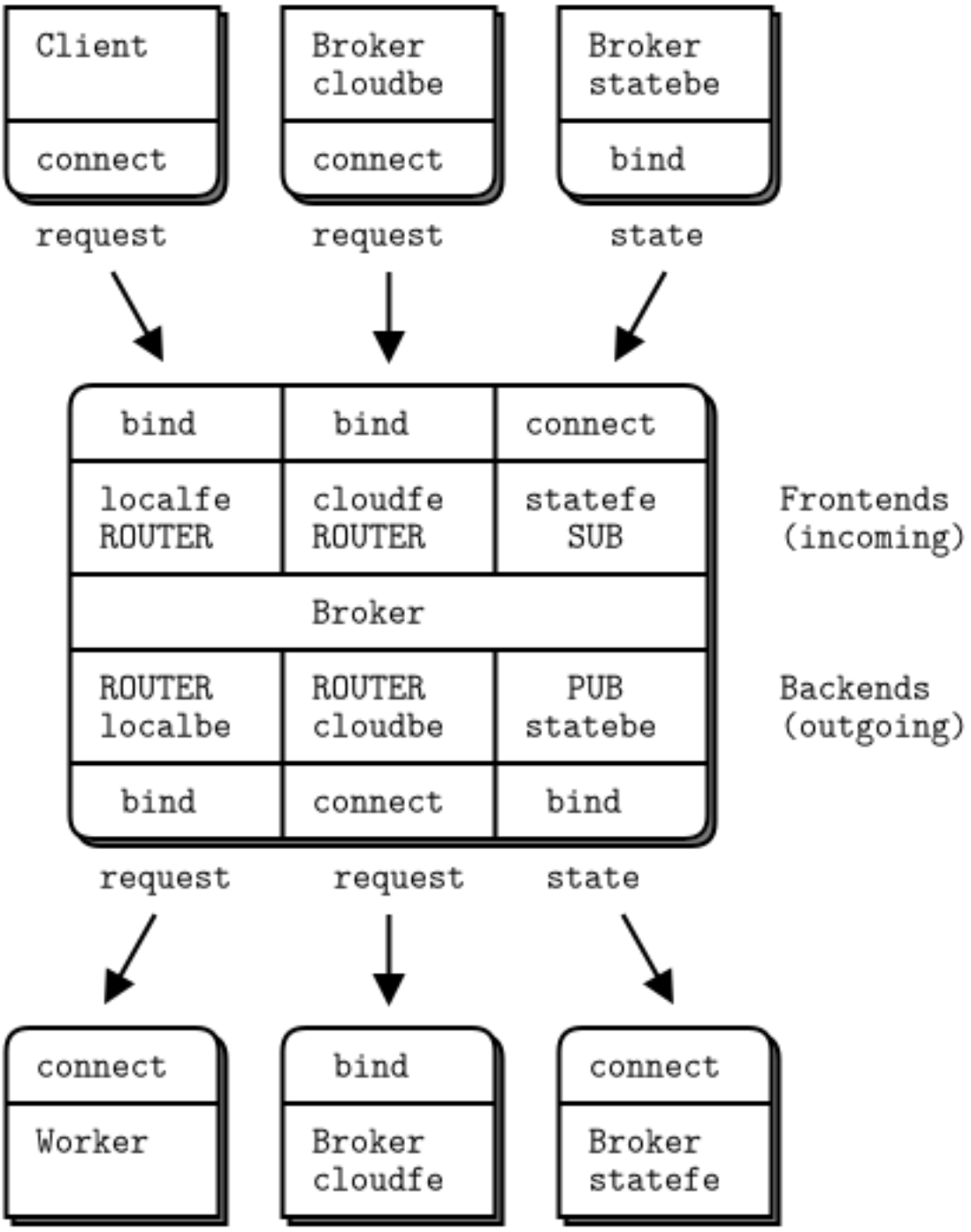


그림 45: Broker Socket Arrangement(Peering)

각 브로커의 cloudbe을 다른 모든 브로커들의 cloudfe에 연결하고 마찬가지로 각 브로커의 statebe에 다른 모든 브로커들의 statefe을 연결합니다.

0.38.6 상태 흐름에 대한 기본 작업

각 소켓 흐름은 방심한 사람들이 빠지기 쉬운 고유의 함정들이 있기 때문에, 한 번에 전체를 코드로 구축하기보다는 각 소켓에 대하여 하나씩 실제 코드로 테스트합니다. 각 흐름이 만족스러우면 전체 프로그램으로 통합할 수 있습니다. 상태 흐름(state flow)부터 시작하겠습니다.

그림 45 - 상태 흐름(The State Flow)

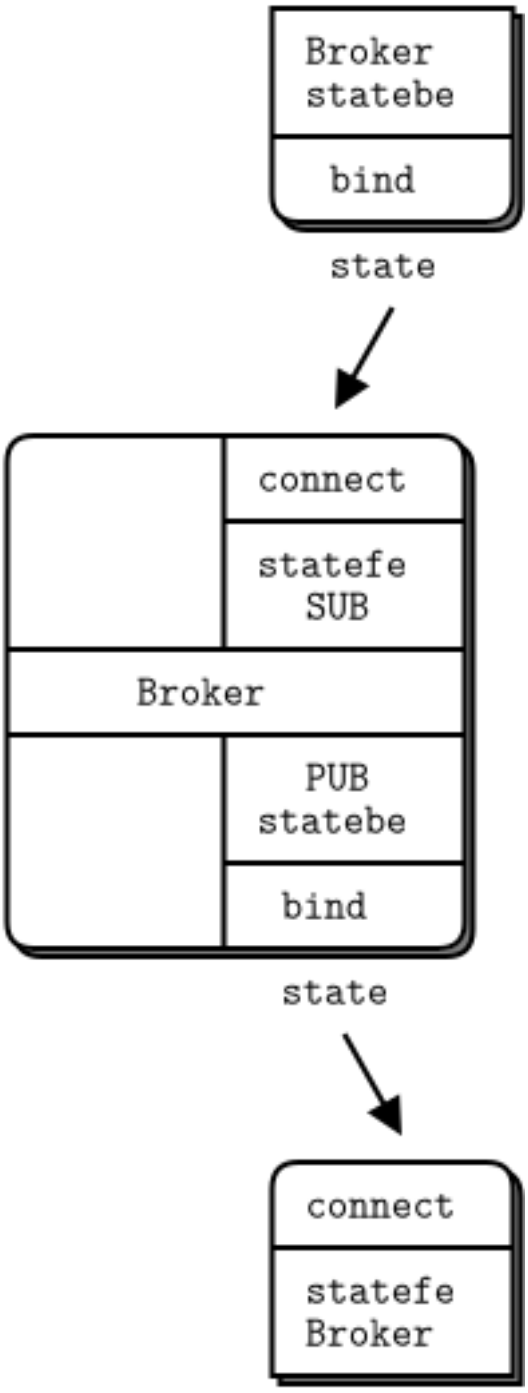


그림 46: The State Flow

상태 흐름이 코드상에서 동작하는 방식입니다.

peering1.c: state flow에 대한 기본작업

```
//  Broker peering simulation (part 1)
//  Prototypes the state flow

#include "czmq.h"

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}...\n");
        return 0;
    }
    char *self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    zctx_t *ctx = zctx_new ();

    //  Bind state backend to endpoint
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  Connect statefe to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (statefe, "");
```

```
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  .split main loop
    //  The main loop sends out status messages to peers, and collects
    //  status messages back from peers. The zmq_poll timeout defines
    //  our own heartbeat:

    while (true) {
        //  Poll for activity, or 1 second timeout
        zmq_pollitem_t items [] = { { statefe, 0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle incoming status messages
        if (items [0].revents & ZMQ_POLLIN) {
            char *peer_name = zstr_recv (statefe);
            char *available = zstr_recv (statefe);
            printf ("%s - %s workers free\n", peer_name, available);
            free (peer_name);
            free (available);
        }
        else {
            //  Send random values for worker availability
            zstr_sendm (statebe, self);
```

```
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

이 코드에서 주의할 사항은 다음과 같습니다.

- 각 브로커에는 식별자(ID)를 가지며 ipc 단말 이름들을 구성하는 데 사용합니다. 실제 브로커는 TCP상에서 동작하며 좀 더 복잡한 구성체계로 작업해야 합니다. 이 책의 뒷부분에서 보겠지만, 지금은 생성된 ipc 이름을 사용하여 TCP/IP 주소 또는 이름으로 발생하는 문제를 무시할 수 있습니다.
- 우리는 프로그램의 핵심으로 `zmq_poll()` 루프를 사용합니다. `zmq_poll()`을 통해 들어오는 메시지들을 처리하고 상태 메시지들을 보냅니다. 수신 메시지들을 1초 동안 받지 못할 경우 상태 메시지를 보냅니다. 메시지를 받을 때마다 상태 메시지를 보내면 다른 브로커들에서 많은 메시지를 처리(상태 메시지 받고/상태 메시지 보내고) 해야 합니다.
- 우리는 발신자 주소와 데이터로 구성된 2개 부분으로 구성된 PUB-SUB 메시지를 사용합니다. 작업을 보내려면 발행자의 주소를 알아야 하기 때문에 한 가지 방법으로 메시지의 일부로 명시적으로 포함하여 보내는 것입니다.
- 실행 중인 브로커들에 연결할 때 오래된 상태 정보를 받을 수 있기 때문에 구독자들에 식별자(ID)를 설정하지 않습니다.
- 우리는 발행자에게 HWM을 설정하지 않았지만 ØMQ v2.x를 사용한다면 설정하는 것이 좋습니다.

프로그램을 빌드하고 3개의 클러스터를 시뮬레이션하기 위해 세 번 실행합니다. DC1, DC2 및 DC3(임의적인 이름임)이라고 부르며, 이 3개 명령들을 별도의 창에서 실행합니다.

```
peering1 DC1 DC2 DC3  #  Start DC1 and connect to DC2 and DC3
peering1 DC2 DC1 DC3  #  Start DC2 and connect to DC1 and DC3
```

```
peering1 DC3 DC1 DC2  #  Start DC3 and connect to DC1 and DC2
```

프로그램들을 실행하면 각 클러스터가 상대 클러스터의 상태를 보고하는 것을 볼 수 있으며, 몇 초 후 그들은 모두 즐겁게 1초당 한 번씩 난수(0~9)를 출력합니다. 이것을 시도하고 3개의 브로커가 모두 일치하고 1초당 상태 변경정보에 동기화됨을 확인하시기 바랍니다.

실제로는 우리는 정기적으로 상태 메시지를 보내는 것이 아니라 상태가 변경될 때(예 : 작업자가 가용하거나 비가용하거나) 메시지를 보냅니다. 메시지로 인한 많은 통신 트래픽이 발생할 것 같지만, 상태 메시지의 크기는 작으며 내부 클러스터 간 연결 설정으로 매우 빠릅니다.

정확한 시간 간격으로 상태 메시지들을 보내려면 자식 스레드를 만들고 자식 스레드에서 statebe 소켓을 열어 메인 스레드에서 자식 스레드로 가용한 작업자의 변경 정보를 보내고 자식 스레드는 정기적인 메시지와 함께 상대 클러스터들에게 보내도록 합니다.

- [옮긴이] 원도우 환경에서는 ipc를 사용할 수 없기 때문에 inproc나 tcp로 코드를 변경하여 테스트가 필요합니다. peering1의 경우 여러 개의 프로세스들을 실행하여 프로세스 간 통신(ipc)하도록 설계되어 있어 프로세스 내(inproc)로 변경할 경우 각 스레드들 간에 컨텍스트가 공유될 수 있도록 수정 필요합니다.
- 프로세스 간 통신이 가능하도록 tcp로 변경하여 테스트를 수행합니다. tcp 소켓을 사용하도록 변경된 `peering1_tcp.c`입니다.

```
//  Broker peering simulation (part 1)
//  Prototypes the state flow

#include "czmq.h"

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
```

```
        printf ("syntax: peering1 me {you}...\n");
        return 0;
    }
    char *self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    zctx_t *ctx = zctx_new ();

    //  Bind state backend to endpoint
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "tcp://*:%s", self);

    //  Connect statefe to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (statefe, "");
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n", peer);
        zsocket_connect (statefe, "tcp://localhost:%s", peer);
    }
    //  .split main loop
    //  The main loop sends out status messages to peers, and collects
    //  status messages back from peers. The zmq_poll timeout defines
    //  our own heartbeat:

    while (true) {
        //  Poll for activity, or 1 second timeout
```

```
        zmq_pollitem_t items [] = { { statefe, 0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle incoming status messages
        if (items [0].revents & ZMQ_POLLIN) {
            char *peer_name = zstr_recv (statefe);
            char *available = zstr_recv (statefe);
            printf ("%s - %s workers free\n", peer_name, available);
            free (peer_name);
            free (available);
        }
        else {
            //  Send random values for worker availability
            zstr_sendm (statebe, self);
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc peering1_tcp.c libzmq.lib czmq.lib


 ./peering1_tcp 5555 5556 5557
I: preparing broker at 5555...
I: connecting to state backend at '5556'
```

```
I: connecting to state backend at '5557'
5556 - 4 workers free
5556 - 1 workers free
5556 - 9 workers free
5556 - 2 workers free
...
 ./peering1_tcp 5556 5555 5557
I: preparing broker at 5556...
I: connecting to state backend at '5555'
I: connecting to state backend at '5557'
5555 - 2 workers free
5555 - 4 workers free
5555 - 7 workers free
5555 - 0 workers free
5555 - 1 workers free
...
 ./peering1_tcp 5557 5556 5555
I: preparing broker at 5557...
I: connecting to state backend at '5556'
I: connecting to state backend at '5555'
5556 - 8 workers free
5556 - 2 workers free
5555 - 8 workers free
5556 - 7 workers free
5555 - 7 workers free
```

[옮긴이] inproc를 사용하여 스레드간 통신이 가능하도록 변경하여 테스트를 수행합니다. inproc 소켓을 사용하도록 변경된 peering1_inproc.c입니다.

```
//  Broker peering simulation (part 1)
//  Prototypes the state flow

#include "czmq.h"

static void
broker(char* argv[], zctx_t *ctx, void *pipe)
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    char *self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  Bind state backend to endpoint
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "inproc://%s-state", self);

    //  Connect statefe to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (statefe, "");
    int argc = 0;
    while(argv[++argc]);
    for (int argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n", peer);
        zsocket_connect (statefe, "inproc://%s-state", peer);
    }
```

```
    //  .split main loop
    //  The main loop sends out status messages to peers, and collects
    //  status messages back from peers. The zmq_poll timeout defines
    //  our own heartbeat:

    while (true) {
        //  Poll for activity, or 1 second timeout
        zmq_pollitem_t items [] = { { statefe, 0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle incoming status messages
        if (items [0].revents & ZMQ_POLLIN) {
            char *peer_name = zstr_recv (statefe);
            char *available = zstr_recv (statefe);
            printf ("%s - %s workers free\n", peer_name, available);
            free (peer_name);
            free (available);
        }
        else {
            //  Send random values for worker availability
            zstr_sendm (statebe, self);
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
}
```

```
int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}...\n");
        return 0;
    }
    zctx_t *ctx = zctx_new ();
    zthread_fork (ctx, broker, argv);
    zclock_sleep(100);
    // Change the order
    int m,n;
    for(n=2; n < argc; n++){
        char *temp = strdup(argv[1]);
        for(m = 2; m < argc; m++)
        {
            argv[m-1] = argv[m];
        }
        argv[m-1] = temp;
        zthread_fork (ctx, broker, argv);
        zclock_sleep(100);
    }
    zclock_sleep (10 * 1000);    //  Run for 10 seconds then quit
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc peering1_inproc.c libzmq.lib czmq.lib


 ./peering1_inproc me cat dog
I: preparing broker at me...
I: connecting to state backend at 'cat'
I: connecting to state backend at 'dog'
I: preparing broker at cat...
I: connecting to state backend at 'dog'
I: connecting to state backend at 'me'
I: preparing broker at dog...
I: connecting to state backend at 'me'
I: connecting to state backend at 'cat'
me - 3 workers free
me - 3 workers free
cat - 3 workers free
dog - 3 workers free
dog - 3 workers free
me - 5 workers free
cat - 3 workers free
...
```

0.38.7 로컬 및 클라우드 흐름에 대한 기본 작업

이제 로컬과 클라우드 소켓을 통해 작업들 흐름의 기본 작업을 수행하겠습니다. 이 코드는 클라이언트들과 상대 브로커들로부터 요청들 받아 무작위로 로컬 작업자들과 및 클라우드 상대들에 배포합니다.

그림 46 - 작업들의 흐름

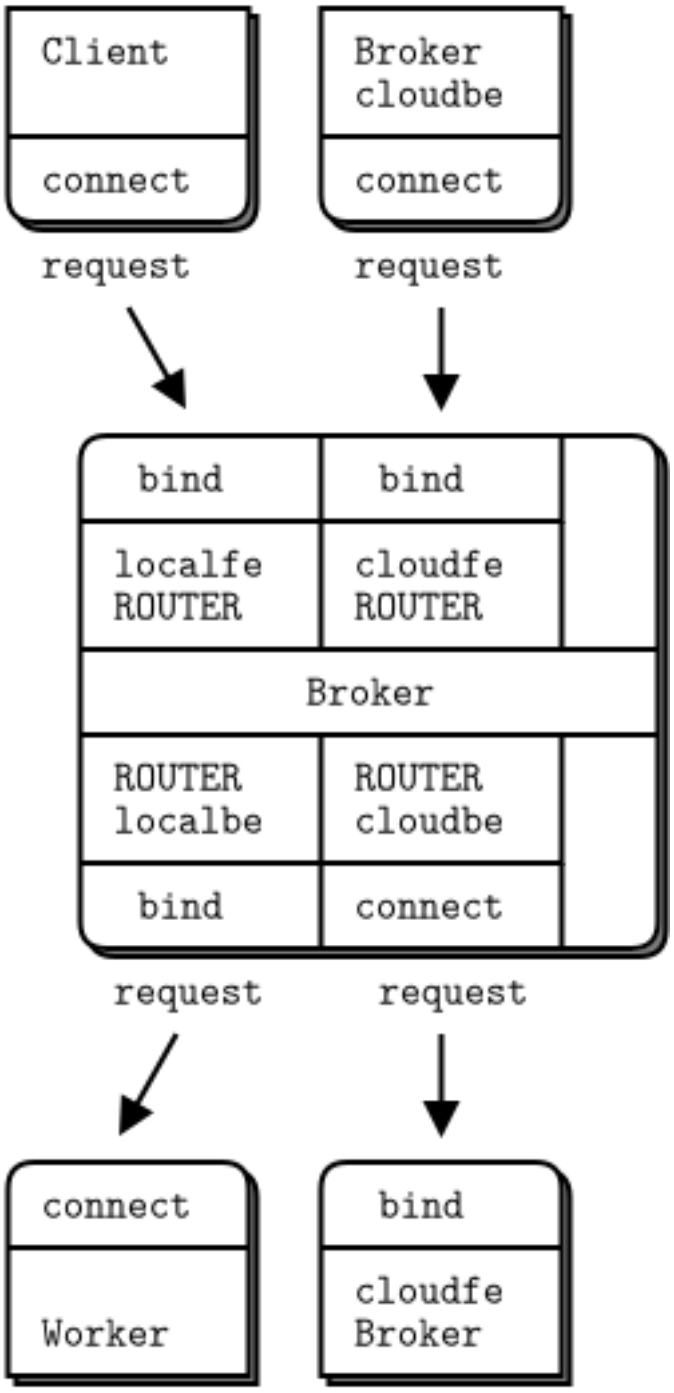


그림 47: The Flow of Tasks

조금 복잡해지는 코드로 작성하기 전에 핵심 라우팅 로직을 그리면서 간단하면서도 안정된 설계를 하겠습니다.

2개의 대기열이 필요합니다. 하나는 로컬 클라이언트들의 요청을 위한 것이고 다른 하나는 클라우드 클라이언트들의 요청을 위한 것입니다. 한 가지 옵션은 로컬 및 클라우드 프론트엔드에서 메시지를 가져와서 각각의 대기열들에서 퍼내는 것이지만 ØMQ 소켓은 이미 대기열이 존재하여 무의미합니다. 따라서 ØMQ 소켓 버퍼들을 대기열들로 사용합니다.

이것이 우리가 부하 분산 브로커(lbbroker)에서 사용한 기술이며 멋지게 동작했습니다. 요청을 보낼 곳(작업자들 혹은 상대 브로커들)이 있을 때만 2개의 프론트엔드들에서 읽습니다. 백엔드에서 요청에 대한 응답을 반환하므로 항상 백엔드 소켓을 읽을 수 있습니다. 백엔드가 우리와 통신하지 않는 한 프론트엔드를 감시할 필요가 없습니다.

- [옮긴이] 백엔드에서 처리 가능한 경우 프론트엔드로부터 요청을 받아 전달하며, 이전 예제(lbbroker)에서 처럼 작업자 대기열의 대기 상태를 보고 요청 처리할 수 있습니다.

주요 처리 과정은 다음과 같습니다.

- 특정 작업을 수행하기 위해 백엔드를 폴링하면 수신한 메시지는 작업자의 "준비(READY)" 상태이거나 응답일 수 있습니다. 응답인 경우 로컬 클라이언트 프론트엔드(localfe) 혹은 클라우드 프론트엔드(cloudfe)를 통해 반환합니다.
- 작업자가 응답하면 사용할 수 있게 되었으므로 대기열에 넣고 큐의 크기를 하나 증가시킵니다.
- 작업자들이 가용할 동안 프론트엔드에서 요청을 받아 로컬 작업자(localbe)에게 또는 무작위로 클라우드 상대(cloudbe)로 라우팅합니다.

작업자가 아닌 상대 브로커에 무작위로 작업을 보내는 것은 전체 클러스터에서 작업 분배를 시뮬레이션하기 위함입니다. 그다지 현명하지 않지만 이 단계에서는 괜찮습니다.

브로커 식별자(ID)를 사용하여 브로커들 간에 메시지를 전달합니다. 각 브로커에 있는 식별자(ID)는 단순한 프로그램 실행 명령에서 매개변수로 제공하는 이름입니다. 이러한 식별자(ID)는 클라이언트 노드들의 식별자(ID)와 중복되지 말아야 하며, 중복될 경우 응답을 클라이언트들 혹은 브로커로 반환할지 알 수 없습니다.

여기에서 실제로 작동하는 코드입니다. 흥미로운 부분은 "Interesting part)"이라는 주석으로 시작됩니다.

peering2.c: local과 cloud 간 흐름의 기본 구현

```
//  Broker peering simulation (part 2)
//  Prototypes the request-reply flow

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      //  Signals worker is ready

//  Our own name; in practice this would be configured per node
static char *self;

//  .split client task
//  The client task does a request-reply dialog using a standard
//  synchronous REQ socket:

static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);

    while (true) {
        //  Send request, get reply
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              //  Interrupted
```

```
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split worker task
//  The worker task plugs into the load-balancer using a REQ
//  socket:

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
```

```
        zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split main task
//  The main task begins by setting-up its frontend and backend sockets
//  and then starting its client and worker tasks:

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
        printf ("syntax: peering2 me {you}...\n");
        return 0;
    }
    self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    zctx_t *ctx = zctx_new ();

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
```

```
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%s'\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }
    //  Prepare local frontend and backend
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);
    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  Get user to tell us when we can start...
    printf ("Press Enter when all brokers are started: ");
    getchar ();

    //  Start local workers
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  Start local clients
    int client_nbr;
```

```
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);

    // Interesting part
    //  .split request-reply handling
    //  Here, we handle the request-reply flow. We're using load-balancing
    //  to poll workers at all times, and clients only when there are one
    //  or more workers available.

    //  Least recently used queue of available workers
    int capacity = 0;
    zlist_t *workers = zlist_new ();

    while (true) {
        //  First, route any waiting replies from workers
        zmq_pollitem_t backends [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 }
        };
        //  If we have no workers, wait indefinitely
        int rc = zmq_poll (backends, 2,
            capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle reply from local worker
        zmsg_t *msg = NULL;
        if (backends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
```

```
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);
            capacity++;

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  Or handle reply from peer broker
        else
        if (backends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker identity for anything
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_destroy (&identity);
        }
        //  Route reply to cloud if it's addressed to a broker
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
```

```
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  .split route client requests
        //  Now we route as many client requests as we have worker capacity
        //  for. We may reroute requests from our local frontend, but not from
        //  the cloud frontend. We reroute randomly now, just to test things
        //  out. In the next version, we'll do this properly by calculating
        //  cloud capacity:

        while (capacity) {
            zmq_pollitem_t frontends [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            rc = zmq_poll (frontends, 2, 0);
            assert (rc >= 0);
            int reroutable = 0;
            //  We'll do peer brokers first, to prevent starvation
            if (frontends [1].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (cloudfe);
                reroutable = 0;
            }
            else
            if (frontends [0].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (localfe);
                reroutable = 1;
            }
```

```
            else
                break;      //  No work, go back to backends

            //  If reroutable, send to cloud 20% of the time
            //  Here we'd normally use cloud status information
            //
            if (reroutable && argc > 2 && randof (5) == 0) {
                //  Route to random broker peer
                int peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [peer], strlen (argv [peer]));
                zmsg_send (&msg, cloudbe);
            }
            else {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                capacity--;
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

2개의 창에서 2개의 브로커 인스턴스를 시작하며 실행합니다.

```
peering2 me you
peering2 you me
```

코드의 대한 설명입니다.

- 적어도 C 언어로 작성된 코드에서 zmsg 클래스를 사용하면 구현이 쉬워지고 코드는 훨씬 짧아집니다. zmsg는 동작 가능한 추상화입니다. C 언어로 ØMQ 응용프로그램을 개발할 경우 CZMQ 라이브러리를 사용해야 합니다.
- 우리는 상대로부터 어떤 상태 정보도 받지 못하기 때문에 그들이 동작 중이라고 가정합니다. 코드에서 처음에는 모든 브로커들이 실행 중인지 확인하며, 실제로 브로커들의 상태를 보내지 않으면 브로커에게 어떤 메시지도 전송하지 않습니다.

정상적으로 실행되는 코드를 보면서 만족시킬 수 있습니다. 잘못 전달 된 메시지가 있는 경우 클라이언트는 결국 차단되고 브로커는 추적 정보 출력을 중지하며 브로커들을 중단함으로 확인할 수 있습니다. 다른 브로커가 클라우드에 요청을 보내고 클라이언트들은 하나씩 응답이 되돌아올 때까지 기다립니다.

- [옮긴이] 위의 프로그램을 TCP에서 구동할 수 있게 변경하였습니다.

```
    static void *
    client_task (void *args){
    ...
        zsocket_connect (client, "ipc://%s-localfe.ipc", self);
    ...
    }
    static void *
    worker_task (void *args){
    ...
        zsocket_connect (worker, "ipc://%s-localbe.ipc", self);
    ...
```

```
}
int main (int argc, char *argv []){
...
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);
...
    printf ("I: connecting to cloud frontend at '%d'\n", peer);
    zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
...
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);
...
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);
...
}
```

대응하는 TCP 포트를 지정하며 브로커의 peer 지정시에 주의가 필요합니다. * localfe : self * localbe : atoi(self) + 1 * couldfe : atoi(self) + 2 * couldbe : peer + 2

```
static void *
client_task (void *args){
...
    zsocket_connect (client, "tcp://localhost:%s", self);
...
}
static void *
worker_task (void *args){
...
    zsocket_connect (worker, "tcp://localhost:%d", atoi(self)+1);
...
}
int main (int argc, char *argv []){
...
```

```
        zsocket_bind (cloudfe, "tcp://*:%d", atoi(self)+2);
    ...
        printf ("I: connecting to cloud frontend at '%d'\n", atoi(peer)+2);
        zsocket_connect (cloudbe, "tcp://localhost:%d", atoi(peer)+2);
    ...
        zsocket_bind (localfe, "tcp://*:%s", self);
    ...
        zsocket_bind (localbe, "tcp://*:%d", atoi(self)+1);
    ...
    }
```

TCP 전송 방식으로 수정된 peering2_tcp.c 코드는 아래와 같습니다.

```
//  Broker peering simulation (part 2)
//  Prototypes the request-reply flow

#include "czmq.h"
#include "zhelpers.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      //  Signals worker is ready

//  Our own name; in practice this would be configured per node
static char *self;

//  .split client task
//  The client task does a request-reply dialog using a standard
//  synchronous REQ socket:

static void *
client_task (void *args)
```

```
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "tcp://localhost:%s", self);
    printf("client : tcp://localhost:%s\n", self);

    while (true) {
        //  Send request, get reply
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              //  Interrupted
        printf ("Client: %s\n", reply);
        free (reply);
        s_sleep (1000);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split worker task
//  The worker task plugs into the load-balancer using a REQ
//  socket:

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
```

```
    zsocket_connect (worker, "tcp://localhost:%d", atoi(self)+1);
    printf("worker : tcp://localhost:%d\n", atoi(self)+1);

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted

        zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split main task
//  The main task begins by setting-up its frontend and backend sockets
//  and then starting its client and worker tasks:

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
```

```
    //
    if (argc < 2) {
        printf ("syntax: peering2 me {you}...\n");
        return 0;
    }
    self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    zctx_t *ctx = zctx_new ();

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "tcp://*:%d", atoi(self)+2);
    printf("cloudfe : tcp://*:%d\n", atoi(self)+2);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%d'\n", atoi(peer)+2);
        zsocket_connect (cloudbe, "tcp://localhost:%d", atoi(peer)+2);
        printf("cloudbe : tcp://localhost:%d\n", atoi(peer)+2);
    }
    //  Prepare local frontend and backend
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
```

```
    zsocket_bind (localfe, "tcp://*:%s", self);
    printf("localfe : tcp://*:%s\n", self);
    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "tcp://*:%d", atoi(self)+1);
    printf("localbe : tcp://*:%d\n", atoi(self)+1);

    //  Get user to tell us when we can start...
    printf ("Press Enter when all brokers are started: ");
    getchar ();

    //  Start local workers
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  Start local clients
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);

    // Interesting part
    //  .split request-reply handling
    //  Here, we handle the request-reply flow. We're using load-balancing
    //  to poll workers at all times, and clients only when there are one
    //  or more workers available.

    //  Least recently used queue of available workers
    int capacity = 0;
    zlist_t *workers = zlist_new ();
```

```
    while (true) {
        //  First, route any waiting replies from workers
        zmq_pollitem_t backends [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 }
        };
        //  If we have no workers, wait indefinitely
        int rc = zmq_poll (backends, 2,
            capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle reply from local worker
        zmsg_t *msg = NULL;
        if (backends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_print (zmsg_last (msg), "Worker: ");
            zlist_append (workers, identity);
            capacity++;

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
```

```
        //  Or handle reply from peer broker
        else
        if (backends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker identity for anything
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_destroy (&identity);
        }
        //  Route reply to cloud if it's addressed to a broker
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  .split route client requests
        //  Now we route as many client requests as we have worker capacity
        //  for. We may reroute requests from our local frontend, but not from
        //  the cloud frontend. We reroute randomly now, just to test things
        //  out. In the next version, we'll do this properly by calculating
        //  cloud capacity:
```

```
        while (capacity) {
            zmq_pollitem_t frontends [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            rc = zmq_poll (frontends, 2, 0);
            assert (rc >= 0);
            int reroutable = 0;
            //  We'll do peer brokers first, to prevent starvation
            if (frontends [1].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (cloudfe);
                reroutable = 0;
            }
            else
            if (frontends [0].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (localfe);
                reroutable = 1;
            }
            else
                break;      //  No work, go back to backends

            //  If reroutable, send to cloud 20% of the time
            //  Here we'd normally use cloud status information
            //
            if (reroutable && argc > 2 && randof (5) == 0) {
                //  Route to random broker peer
                int peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [peer], strlen (argv [peer]));
                zmsg_send (&msg, cloudbe);
```

```
            }
            else {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                capacity--;
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

- [옮긴이] 빌드 및 테스트
- 수행시 각 포트 번호 간의 간격을 주고 실행합니다.

```
 cl -EHsc peering2_tcp.c libzmq.lib czmq.lib


 ./peering2_tcp 5560 5570
I: preparing broker at 5560...
cloudfe : tcp://*:5562
I: connecting to cloud frontend at '5572'
cloudbe : tcp://localhost:5572
localfe : tcp://*:5560
```

```
localbe : tcp://*:5561
Press Enter when all brokers are started:
client : tcp://localhost:5560
client : tcp://localhost:5560
worker : tcp://localhost:5561
...
D: 20-08-14 17:17:52 Worker: [001] 01
client : tcp://localhost:5560
D: 20-08-14 17:17:52 Worker: [001] 01
D: 20-08-14 17:17:52 Worker: [005] HELLO
D: 20-08-14 17:17:52 Worker: [001] 01
D: 20-08-14 17:17:52 Worker: [005] HELLO
D: 20-08-14 17:17:52 Worker: [002] OK
D: 20-08-14 17:17:52 Worker: [005] HELLO
D: 20-08-14 17:17:52 Worker: [002] OK
Client: OK
...


 ./peering2_tcp 5570 5560
I: preparing broker at 5570...
cloudfe : tcp://*:5572
I: connecting to cloud frontend at '5562'
cloudbe : tcp://localhost:5562
localfe : tcp://*:5570
localbe : tcp://*:5571
Press Enter when all brokers are started:
worker : tcp://localhost:5571
worker : tcp://localhost:5571
client : tcp://localhost:5570
```

```
...
D: 20-08-14 17:17:54 Worker: [001] 01
D: 20-08-14 17:17:54 Worker: [001] 01
D: 20-08-14 17:17:54 Worker: [005] HELLO
D: 20-08-14 17:17:54 Worker: [001] 01
D: 20-08-14 17:17:54 Worker: [005] HELLO
D: 20-08-14 17:17:54 Worker: [002] OK
D: 20-08-14 17:17:54 Worker: [005] HELLO
D: 20-08-14 17:17:54 Worker: [002] OK
D: 20-08-14 17:17:54 Worker: [005] HELLO
D: 20-08-14 17:17:54 Worker: [002] OK
D: 20-08-14 17:17:54 Worker: [005] HELLO
D: 20-08-14 17:17:54 Worker: [002] OK
D: 20-08-14 17:17:54 Worker: [005] HELLO
Client: OK
```

- [옮긴이] inproc 전송 방식으로 수정된 peering2_inproc.c 코드는 아래와 같습니다. peering2_inproc.c : inproc로 스레드간 통신으로 변경된 로컬 및 클라우드 작업 분배합니다.

```
//  Broker peering simulation (part 2)
//  Prototypes the request-reply flow

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      //  Signals worker is ready

//  .split client task
//  The client task does a request-reply dialog using a standard
```

```
//  synchronous REQ socket:

static void
client_task (char* argv[], zctx_t *ctx, void *pipe)
{
    char *self = argv [1];
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "inproc://%s-localfe", self);

    while (true) {
        //  Send request, get reply
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              //  Interrupted
        printf ("[%s]Client: %s\n", self, reply);
        free (reply);
        zclock_sleep (1000);
    }
}

//  .split worker task
//  The worker task plugs into the load-balancer using a REQ
//  socket:

static void
worker_task (char* argv[], zctx_t *ctx, void *pipe)
{
    char *self = argv [1];
```

```
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "inproc://%s-localbe", self);

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
        char msgbuffer[20];
        sprintf(msgbuffer, "[%s] Workers : ", self);
        zframe_print (zmsg_last (msg), msgbuffer);
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
}

//  broker task
//  The main task begins by setting-up its frontend and backend sockets
//  and then starting its client and worker tasks:

static void
broker(char* argv[], zctx_t *ctx, void *pipe)
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
```

```
    //
    char *self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "inproc://%s-cloud", self);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudbe, self);
    int argc = 0;
    while(argv[++argc]);
    for (int argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%s'\n", peer);
        zsocket_connect (cloudbe, "inproc://%s-cloud", peer);
    }
    //  Prepare local frontend and backend
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "inproc://%s-localfe", self);
    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "inproc://%s-localbe", self);

    //  Get user to tell us when we can start...
//    printf ("Press Enter when all brokers are started: ");
//    getchar ();
```

```
    //  Start local workers
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_fork (ctx, worker_task, argv);

    //  Start local clients
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_fork (ctx, client_task, argv);

    // Interesting part
    //  .split request-reply handling
    //  Here, we handle the request-reply flow. We're using load-balancing
    //  to poll workers at all times, and clients only when there are one
    //  or more workers available.

    //  Least recently used queue of available workers
    int capacity = 0;
    zlist_t *workers = zlist_new ();

    while (true) {
        //  First, route any waiting replies from workers
        zmq_pollitem_t backends [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 }
        };
        //  If we have no workers, wait indefinitely
        int rc = zmq_poll (backends, 2,
```

```
            capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle reply from local worker
        zmsg_t *msg = NULL;
        if (backends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);
            capacity++;

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  Or handle reply from peer broker
        else
        if (backends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker identity for anything
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_destroy (&identity);
        }
```

```
        //  Route reply to cloud if it's addressed to a broker
        for (int argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  .split route client requests
        //  Now we route as many client requests as we have worker capacity
        //  for. We may reroute requests from our local frontend, but not from
        //  the cloud frontend. We reroute randomly now, just to test things
        //  out. In the next version, we'll do this properly by calculating
        //  cloud capacity:

        while (capacity) {
            zmq_pollitem_t frontends [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            rc = zmq_poll (frontends, 2, 0);
            assert (rc >= 0);
            int reroutable = 0;
            //  We'll do peer brokers first, to prevent starvation
            if (frontends [1].revents & ZMQ_POLLIN) {
```

```
            msg = zmsg_recv (cloudfe);
            reroutable = 0;
        }
        else
        if (frontends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localfe);
            reroutable = 1;
        }
        else
            break;      //  No work, go back to backends

        //  If reroutable, send to cloud 20% of the time
        //  Here we'd normally use cloud status information
        //
        if (reroutable && argc > 2 && randof (5) == 0) {
            //  Route to random broker peer
            int peer = randof (argc - 2) + 2;
            zmsg_pushmem (msg, argv [peer], strlen (argv [peer]));
            zmsg_send (&msg, cloudbe);
        }
        else {
            zframe_t *frame = (zframe_t *) zlist_pop (workers);
            zmsg_wrap (msg, frame);
            zmsg_send (&msg, localbe);
            capacity--;
        }
    }
}
//  When we're done, clean up properly
```

```
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
}
int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}...\n");
        return 0;
    }
    zctx_t *ctx = zctx_new ();
    zthread_fork (ctx, broker, argv);
    zclock_sleep(100);
    // Change the order
    int m,n;
    for(n=2; n < argc; n++){
        char *temp = strdup(argv[1]);
        for(m = 2; m < argc; m++)
        {
            argv[m-1] = argv[m];
        }
        argv[m-1] = temp;
        zthread_fork (ctx, broker, argv);
        zclock_sleep(100);
```

```
    }
    zclock_sleep (10 * 1000);    //  Run for 10 seconds then quit
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc peering2_inproc.c libzmq.lib czmq.lib


 ./peering2_inproc dog cat
I: preparing broker at dog...
I: connecting to cloud frontend at 'cat'
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
[dog]Client: OK
D: 20-09-07 11:11:32 [dog] Workers : [005] HELLO
```

```
[dog]Client: OK
[dog]Client: OK
[dog]Client: OK
I: preparing broker at cat...
I: connecting to cloud frontend at 'dog'
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
[cat]Client: OK
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
[cat]Client: OK
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
[cat]Client: OK
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
[cat]Client: OK
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
[cat]Client: OK
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
[cat]Client: OK
D: 20-09-07 11:11:32 [cat] Workers : [005] HELLO
[cat]Client: OK
```

0.38.8 결합하기

위의 예제들을 하나의 패키지로 합치겠습니다. 이전에는 전체 클러스터를 하나의 프로세스로 실행합니다. 2가지 예제를 가져와서 하나로 병합하여 원하는 수의 클러스터들을 시뮬레이션할 수 있도록 하겠습니다.

이 코드는 270줄로 이전 2개의 예제를 합친 크기입니다. 이는 클라이언트들과 작업자들 및 클라우드 작업부하를 분산시키는 클러스터 시뮬레이션에 좋습니다. 다음은 코드입니다.

peering3.c: 클로스터 전체 시뮬레이션

```
//  Broker peering simulation (part 3)
//  Prototypes the full flow of status and tasks

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define WORKER_READY   "\001"      //  Signals worker is ready

//  Our own name; in practice, this would be configured per node
static char *self;

//  .split client task
//  This is the client task. It issues a burst of requests and then
//  sleeps for a few seconds. This simulates sporadic activity; when
//  a number of clients are active at once, the local workers should
//  be overloaded. The client uses a REQ socket for requests and also
//  pushes statistics to the monitor socket:

static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "ipc://%s-monitor.ipc", self);

    while (true) {
        sleep (randof (5));
```

```
        int burst = randof (15);
        while (burst--) {
            char task_id [5];
            sprintf (task_id, "%04X", randof (0x10000));

            //  Send request with random hex ID
            zstr_send (client, task_id);

            //  Wait max ten seconds for a reply, then complain
            zmq_pollitem_t pollset [1] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (pollset, 1, 10 * 1000 * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  Interrupted

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
                    break;              //  Interrupted
                //  Worker is supposed to answer us with our task id
                assert (streq (reply, task_id));
                zstr_sendf (monitor, "%s", reply);
                free (reply);
            }
            else {
                zstr_sendf (monitor,
                    "E: CLIENT EXIT - lost task %s", task_id);
                return NULL;
            }
        }
```

```
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split worker task
//  This is the worker task, which uses a REQ socket to plug into the
//  load-balancer. It's the same stub worker task that you've seen in
//  other examples:

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted

        //  Workers are busy for 0/1 seconds
        sleep (randof (2));
```

```
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split main task
//  The main task begins by setting up all its sockets. The local frontend
//  talks to clients, and our local backend talks to workers. The cloud
//  frontend talks to peer brokers as if they were clients, and the cloud
//  backend talks to peer brokers as if they were workers. The state
//  backend publishes regular state messages, and the state frontend
//  subscribes to all state backends to collect these messages. Finally,
//  we use a PULL monitor socket to collect printable messages from tasks:

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    if (argc < 2) {
        printf ("syntax: peering3 me {you}...\n");
        return 0;
    }
    self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  Prepare local frontend and backend
    zctx_t *ctx = zctx_new ();
```

```
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);

    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%s'\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }
    //  Bind state backend to endpoint
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  Connect state frontend to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (statefe, "");
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
```

```
        printf ("I: connecting to state backend at '%s'\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  Prepare monitor socket
    void *monitor = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (monitor, "ipc://%s-monitor.ipc", self);

    //  .split start child tasks
    //  After binding and connecting all our sockets, we start our child
    //  tasks - workers and clients:

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  Start local clients
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);

    //  Queue of available workers
    int local_capacity = 0;
    int cloud_capacity = 0;
    zlist_t *workers = zlist_new ();

    //  .split main loop
    //  The main loop has two parts. First, we poll workers and our two service
    //  sockets (statefe and monitor), in any case. If we have no ready workers,
    //  then there's no point in looking at incoming requests. These can remain
```

```
    //  on their internal ØMQ queues:

    while (true) {
        zmq_pollitem_t primary [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 },
            { statefe, 0, ZMQ_POLLIN, 0 },
            { monitor, 0, ZMQ_POLLIN, 0 }
        };
        //  If we have no workers ready, wait indefinitely
        int rc = zmq_poll (primary, 4,
            local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Track if capacity changes during this iteration
        int previous = local_capacity;
        zmsg_t *msg = NULL;     //  Reply from local worker

        if (primary [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);
            local_capacity++;

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
```

```
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  Or handle reply from peer broker
        else
        if (primary [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker identity for anything
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_destroy (&identity);
        }
        //  Route reply to cloud if it's addressed to a broker
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  .split handle state messages
        //  If we have input messages on our statefe or monitor sockets, we
        //  can process these immediately:
```

```
        if (primary [2].revents & ZMQ_POLLIN) {
            char *peer = zstr_recv (statefe);
            char *status = zstr_recv (statefe);
            cloud_capacity = atoi (status);
            free (peer);
            free (status);
        }
        if (primary [3].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (monitor);
            printf ("%s\n", status);
            free (status);
        }
        //  .split route client requests
        //  Now route as many clients requests as we can handle. If we have
        //  local capacity, we poll both localfe and cloudfe. If we have cloud
        //  capacity only, we poll just localfe. We route any request locally
        //  if we can, else we route to the cloud.

        while (local_capacity + cloud_capacity) {
            zmq_pollitem_t secondary [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            if (local_capacity)
                rc = zmq_poll (secondary, 2, 0);
            else
                rc = zmq_poll (secondary, 1, 0);
            assert (rc >= 0);
```

```
            if (secondary [0].revents & ZMQ_POLLIN)
                msg = zmsg_recv (localfe);
            else
            if (secondary [1].revents & ZMQ_POLLIN)
                msg = zmsg_recv (cloudfe);
            else
                break;      //  No work, go back to primary

            if (local_capacity) {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                local_capacity--;
            }
            else {
                //  Route to random broker peer
                int peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [peer], strlen (argv [peer]));
                zmsg_send (&msg, cloudbe);
            }
        }
        //  .split broadcast capacity
        //  We broadcast capacity messages to other peers; to reduce chatter,
        //  we do this only if our capacity changed.

        if (local_capacity != previous) {
            //  We stick our own identity onto the envelope
            zstr_sendm (statebe, self);
            //  Broadcast new capacity
```

```
            zstr_sendf (statebe, "%d", local_capacity);
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

적당히 큰 프로그램으로 코드를 작성하는데 대략 하루가 걸렸습니다. 주목할 점은 다음과 같습니다.

- 클라이언트 스레드들은 실패한 요청을 감지하고 보고 합니다. 응답을 기다리면서 잠시 후(약 10초) 아무것도 도착하지 않으면 오류 메시지를 출력합니다.
- 클라이언트 스레드들은 직접 인쇄하지 않고 대신 모니터 PUSH 소켓에 메시지를 보내면 메인의 PULL 소켓이 수집하고 출력합니다. 이것은 모니터링 및 로깅을 위해 ØMQ 소켓을 사용한 첫 번째 사례였습니다. 이 방법은 중요하므로 나중에 자세히 설명합니다.
- 클라이언트는 다양한 부하를 시뮬레이션하며 임의의 순간에 클러스터의 부하가 100% 가 되면 작업을 클라우드로 옮기게 합니다. 시뮬레이션 부하는 클라이언트들 및 작업자들의 수와 클라이언트 및 작업자 스레드들의 지연으로 제어합니다. 보다 현실적인 시뮬레이션을 위해 시뮬레이션 부하를 조정해 보시기 바랍니다.
- 메인 루프는 2개의 `zmq_pollitem_t`를 사용합니다. 실제로 정보, 백엔드 및 프론트엔드의 3가지를 사용합니다. 이전 기본 작업에서와 같이 백엔드에 처리 용량이 없으면 프런트 엔드 메시지를 받지 않습니다.

다음은 프로그램을 개발하면서 발생한 몇 가지 문제들입니다.

- 어딘가에서 요청들이나 응답들이 잃게 되어 클라이언트들이 멈출 수 있습니다. ROUTER 소켓은 전달할 수 없는 메시지들을 삭제합니다. 여기서 첫 번째 전략은 이러한 문제를 감지하고 보고하도록 클라이언트 스레드를 수정하는 것입니다. 둘째, 문제의 원인이 명확해질 때까지 메인 루프에서 모든 수신 후와 모든 송신 전에 `zmsg_dump()` 호출을 넣었습니다.
- 메인 루프가 하나 이상의 준비된 소켓들에서 잘못 읽게 되면 첫 번째 메시지가 유실되었습니다. 첫 번째 준비된 소켓에서만 읽음으로써 문제를 해결했습니다.
- zmsg 클래스가 UUID를 C 문자열로 제대로 인코딩하지 못했습니다. 이로 인해 UUID가 손상되어 0 바이트가 되었습니다. UUID를 출력 가능한 16진수 문자열로 인코딩하기 위해 zmsg를 수정하였습니다.

이 시뮬레이션은 클라스터 상대의 사라짐을 감지하지 않습니다. 여러 클러스터 상대들을 시작하고 하나(다른 상대들에게 작업자 수를 브로드캐스팅 수행함)를 중지하면, 다른 상대들은 사라진 상대에게 계속해서 작업을 보냅니다. 당신은 이것을 시도할 경우 분실된 요청들에 대해 불평(10초 대기 동안 응답이 오지 않으면 오류 메시지 출력)하는 클라이언트들을 확인할 수 있습니다. 해결책은 2개입니다. 첫째, 용량 정보(작업자 수)를 잠시 동안만 유지하여 상대가 사라지면 용량이 빠르게 0으로 설정합니다. 둘째, 요청-응답 체인에 신뢰성을 추가합니다. 다음 장에서 안정성에 대하여 설명하겠습니다.

- [옮긴이] peering3.c는 ipc 전송 방식을 사용하여 원도우에서 사용 가능한 tcp 전송 방식으로 변경하겠습니다. 변경을 위하여 바인드와 연결에 사용되는 포트 번호를 지정합니다.
- localfe : self
- localbe : atoi(self)+1
- cloudfe : atoi(self)+2
- cloudbe : atoi(peer)+2
- monitor : atoi(self)+2
- statefe : atoi(peer)+2
- statebe : atoi(self)+4
- 변경된 코드(peering3_tcp.c)는 다음과 같습니다.

```
//  Broker peering simulation (part 3)
//  Prototypes the full flow of status and tasks

#include "czmq.h"
#include "zhelpers.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define WORKER_READY   "\001"      //  Signals worker is ready

//  Our own name; in practice, this would be configured per node
static char *self;

//  .split client task
//  This is the client task. It issues a burst of requests and then
//  sleeps for a few seconds. This simulates sporadic activity; when
//  a number of clients are active at once, the local workers should
//  be overloaded. The client uses a REQ socket for requests and also
//  pushes statistics to the monitor socket:

static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "tcp://localhost:%s", self);
    printf("[d]client : tcp://localhost:%s\n", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "tcp://localhost:%d", atoi(self)+3);
    printf("[d]monitor : tcp://localhost:%d\n", atoi(self)+3);
```

```
    while (true) {
        s_sleep (randof (5)*1000);
        int burst = randof (15);
        while (burst--) {
            char task_id [5];
            sprintf (task_id, "%04X", randof (0x10000));

            //  Send request with random hex ID
            zstr_send (client, task_id);

            //  Wait max ten seconds for a reply, then complain
            zmq_pollitem_t pollset [1] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (pollset, 1, 10 * 1000 * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  Interrupted

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
                    break;              //  Interrupted
                //  Worker is supposed to answer us with our task id
                assert (streq (reply, task_id));
                zstr_sendf (monitor, "CLIENT recived reply : %s", reply);
                free (reply);
            }
            else {
                zstr_sendf (monitor,
                    "E: CLIENT EXIT - lost task %s", task_id);
                return NULL;
```

```
            }
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split worker task
//  This is the worker task, which uses a REQ socket to plug into the
//  load-balancer. It's the same stub worker task that you've seen in
//  other examples:

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "tcp://localhost:%d", atoi(self)+1);
    printf("[d]worker : tcp://localhost:%d\n", atoi(self)+1);
    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
```

```
        //  Workers are busy for 0/1 seconds
        s_sleep (randof (2)*1000);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split main task
//  The main task begins by setting up all its sockets. The local frontend
//  talks to clients, and our local backend talks to workers. The cloud
//  frontend talks to peer brokers as if they were clients, and the cloud
//  backend talks to peer brokers as if they were workers. The state
//  backend publishes regular state messages, and the state frontend
//  subscribes to all state backends to collect these messages. Finally,
//  we use a PULL monitor socket to collect printable messages from tasks:

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    if (argc < 2) {
        printf ("syntax: peering3 me {you}...\n");
        return 0;
    }
    self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));
```

```
    //  Prepare local frontend and backend
    zctx_t *ctx = zctx_new ();
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "tcp://*:%s", self);
    printf("[d]localfe : tcp://*:%s\n",self);

    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "tcp://*:%d", atoi(self)+1);
    printf("[d]localbe : tcp://*:%d\n", atoi(self)+1);

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "tcp://*:%d", atoi(self)+2);
    printf("[d]cloudfe : tcp://*:%d\n", atoi(self)+2);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%d'\n", atoi(peer)+2);
        zsocket_connect (cloudbe, "tcp://localhost:%d", atoi(peer)+2);
        printf("[d]cloudbe : tcp://localhost:%d\n", atoi(peer)+2);
    }
    //  Bind state backend to endpoint
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "tcp://*:%d", atoi(self)+4);
```

```
    printf("[d]statebe : tcp://*:%d\n", atoi(self)+4);

    //  Connect state frontend to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (statefe, "");
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%d'\n", atoi(peer)+4);
        zsocket_connect (statefe, "tcp://localhost:%d", atoi(peer)+4);
        printf("[d]statefe : tcp://localhost:%d\n", atoi(peer)+4);
    }
    //  Prepare monitor socket
    void *monitor = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (monitor, "tcp://*:%d", atoi(self)+3);
    printf("[d]monitor : tcp://*:%d\n", atoi(self)+3);

    //  .split start child tasks
    //  After binding and connecting all our sockets, we start our child
    //  tasks - workers and clients:

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    //  Start local clients
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);
```

```
    //  Queue of available workers
    int local_capacity = 0;
    int cloud_capacity = 0;
    zlist_t *workers = zlist_new ();

    //  .split main loop
    //  The main loop has two parts. First, we poll workers and our two service
    //  sockets (statefe and monitor), in any case. If we have no ready workers,
    //  then there's no point in looking at incoming requests. These can remain
    //  on their internal ØMQ queues:

    while (true) {
        zmq_pollitem_t primary [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 },
            { statefe, 0, ZMQ_POLLIN, 0 },
            { monitor, 0, ZMQ_POLLIN, 0 }
        };
        //  If we have no workers ready, wait indefinitely
        int rc = zmq_poll (primary, 4,
            local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Track if capacity changes during this iteration
        int previous = local_capacity;
        zmsg_t *msg = NULL;     //  Reply from local worker

        if (primary [0].revents & ZMQ_POLLIN) {
```

```
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);
            local_capacity++;

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  Or handle reply from peer broker
        else
        if (primary [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker identity for anything
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_destroy (&identity);
        }
        //  Route reply to cloud if it's addressed to a broker
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
```

```
        }
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  .split handle state messages
        //  If we have input messages on our statefe or monitor sockets, we
        //  can process these immediately:

        if (primary [2].revents & ZMQ_POLLIN) {
            char *peer = zstr_recv (statefe);
            char *status = zstr_recv (statefe);
            cloud_capacity = atoi (status);
            free (peer);
            free (status);
        }
        if (primary [3].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (monitor);
            printf ("%s\n", status);
            free (status);
        }
        //  .split route client requests
        //  Now route as many clients requests as we can handle. If we have
        //  local capacity, we poll both localfe and cloudfe. If we have cloud
        //  capacity only, we poll just localfe. We route any request locally
        //  if we can, else we route to the cloud.

        while (local_capacity + cloud_capacity) {
            zmq_pollitem_t secondary [] = {
```

```
            { localfe, 0, ZMQ_POLLIN, 0 },
            { cloudfe, 0, ZMQ_POLLIN, 0 }
        };
        if (local_capacity)
            rc = zmq_poll (secondary, 2, 0);
        else
            rc = zmq_poll (secondary, 1, 0);
        assert (rc >= 0);

        if (secondary [0].revents & ZMQ_POLLIN)
            msg = zmsg_recv (localfe);
        else
        if (secondary [1].revents & ZMQ_POLLIN)
            msg = zmsg_recv (cloudfe);
        else
            break;      //  No work, go back to primary

        if (local_capacity) {
            zframe_t *frame = (zframe_t *) zlist_pop (workers);
            zmsg_wrap (msg, frame);
            zmsg_send (&msg, localbe);
            local_capacity--;
        }
        else {
            //  Route to random broker peer
            int peer = randof (argc - 2) + 2;
            zmsg_pushmem (msg, argv [peer], strlen (argv [peer]));
            zmsg_send (&msg, cloudbe);
        }
```

```
        }
        //  .split broadcast capacity
        //  We broadcast capacity messages to other peers; to reduce chatter,
        //  we do this only if our capacity changed.

        if (local_capacity != previous) {
            //  We stick our own identity onto the envelope
            zstr_sendm (statebe, self);
            //  Broadcast new capacity
            zstr_sendf (statebe, "%d", local_capacity);
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

- [옮긴이] 빌드 및 테스트

```
PS C:\Users\zzedd>  cd D:\git_store\zguide-kr\examples\C
 cl -EHsc peering3_tcp.c libzmq.lib czmq.lib
 ./peering3_tcp 5560 5570
I: preparing broker at 5560...
[d]localfe : tcp://*:5560
[d]localbe : tcp://*:5561
```

```
[d]cloudfe : tcp://*:5562
I: connecting to cloud frontend at '5572'
[d]cloudbe : tcp://localhost:5572
[d]statebe : tcp://*:5564
I: connecting to state backend at '5574'
[d]statefe : tcp://localhost:5574
[d]monitor : tcp://*:5563
[d]worker : tcp://localhost:5561
...
[d]monitor : tcp://localhost:5563
[d]client : tcp://localhost:5560
CLIENT recived reply : 317C
[d]monitor : tcp://localhost:5563
[d]monitor : tcp://localhost:5563
CLIENT recived reply : CF08
CLIENT recived reply : 317C
[d]monitor : tcp://localhost:5563
CLIENT recived reply : 317C
CLIENT recived reply : 317C
CLIENT recived reply : 317C
CLIENT recived reply : CF08
...


 ./peering3_tcp 5570 5560
I: preparing broker at 5570...
[d]localfe : tcp://*:5570
[d]localbe : tcp://*:5571
[d]cloudfe : tcp://*:5572
I: connecting to cloud frontend at '5562'
```

```
[d]cloudbe : tcp://localhost:5562
[d]statebe : tcp://*:5574
I: connecting to state backend at '5564'
[d]statefe : tcp://localhost:5564
[d]monitor : tcp://*:5573
[d]client : tcp://localhost:5570
[d]worker : tcp://localhost:5571
[d]client : tcp://localhost:5570
...
[d]monitor : tcp://localhost:5573
[d]monitor : tcp://localhost:5573
CLIENT recived reply : 317C
CLIENT recived reply : CF08
[d]monitor : tcp://localhost:5573
CLIENT recived reply : 317C
CLIENT recived reply : 317C
CLIENT recived reply : 317C
```

- [옮긴이] Inproc를 사용할 수 있도록 변경된 코드(peering3_inproc.c)는 다음과 같습니다.

```
//  Broker peering simulation (part 3)
//  Prototypes the full flow of status and tasks

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define WORKER_READY   "\001"      //  Signals worker is ready

//  .split client task
```

```
//  This is the client task. It issues a burst of requests and then
//  sleeps for a few seconds. This simulates sporadic activity; when
//  a number of clients are active at once, the local workers should
//  be overloaded. The client uses a REQ socket for requests and also
//  pushes statistics to the monitor socket:

static void
client_task (char* argv[], zctx_t *ctx, void *pipe)
{
    char *self = argv [1];
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "inproc://%s-localfe", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "inproc://%s-monitor", self);

    while (true) {
        zclock_sleep (randof (5));
        int burst = randof (15);
        while (burst--) {
            char task_id [5];
            sprintf (task_id, "%04X", randof (0x10000));

            //  Send request with random hex ID
            zstr_send (client, task_id);

            //  Wait max ten seconds for a reply, then complain
            zmq_pollitem_t pollset [1] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (pollset, 1, 10 * 1000 * ZMQ_POLL_MSEC);
            if (rc == -1)
```

```
                break;          //  Interrupted

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
                    break;              //  Interrupted
                //  Worker is supposed to answer us with our task id
                assert (streq (reply, task_id));
                zstr_sendf (monitor, "[%s] Client received :  %s", self, reply);
                free (reply);
            }
            else {
                zstr_sendf (monitor,
                    "E: CLIENT EXIT - lost task %s", task_id);
                return;
            }
        }
    }
}

//  .split worker task
//  This is the worker task, which uses a REQ socket to plug into the
//  load-balancer. It's the same stub worker task that you've seen in
//  other examples:

static void
worker_task (char* argv[], zctx_t *ctx, void *pipe)
{
    char *self = argv [1];
```

```
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "inproc://%s-localbe", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "inproc://%s-monitor", self);

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted

        char *msgbuffer = zframe_strdup(zmsg_last (msg));
        zstr_sendf (monitor, "[%s] Worker received :  %s", self, msgbuffer);
        free(msgbuffer);
        //  Workers are busy for 0/1 seconds
        zclock_sleep (randof (2));
        zmsg_send (&msg, worker);
    }
}

//  .split main task
//  The main task begins by setting up all its sockets. The local frontend
//  talks to clients, and our local backend talks to workers. The cloud
//  frontend talks to peer brokers as if they were clients, and the cloud
//  backend talks to peer brokers as if they were workers. The state
```

```
//  backend publishes regular state messages, and the state frontend
//  subscribes to all state backends to collect these messages. Finally,
//  we use a PULL monitor socket to collect printable messages from tasks:

static void
broker(char* argv[], zctx_t *ctx, void *pipe)
{
    char *self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  Prepare local frontend and backend
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "inproc://%s-localfe", self);

    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "inproc://%s-localbe", self);

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "inproc://%s-cloud", self);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudbe, self);
    int argc = 0;
    while(argv[++argc]);
    for (int argn = 2; argn < argc; argn++) {
```

```
        char *peer = argv [argn];
        printf ("[%s] I: connecting to cloud frontend at '%s'\n", self, peer);
        zsocket_connect (cloudbe, "inproc://%s-cloud", peer);
    }
    //  Bind state backend to endpoint
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "inproc://%s-state", self);

    //  Connect state frontend to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (statefe, "");
    for (int argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("[%s] I: connecting to state backend at '%s'\n", self, peer);
        zsocket_connect (statefe, "inproc://%s-state", peer);
    }
    //  Prepare monitor socket
    void *monitor = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (monitor, "inproc://%s-monitor", self);

    //  .split start child tasks
    //  After binding and connecting all our sockets, we start our child
    //  tasks - workers and clients:

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_fork (ctx, worker_task, argv);

    //  Start local clients
```

```
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_fork (ctx, client_task, argv);

    //  Queue of available workers
    int local_capacity = 0;
    int cloud_capacity = 0;
    zlist_t *workers = zlist_new ();

    //  .split main loop
    //  The main loop has two parts. First, we poll workers and our two service
    //  sockets (statefe and monitor), in any case. If we have no ready workers,
    //  then there's no point in looking at incoming requests. These can remain
    //  on their internal 0MQ queues:

    while (true) {
        zmq_pollitem_t primary [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 },
            { statefe, 0, ZMQ_POLLIN, 0 },
            { monitor, 0, ZMQ_POLLIN, 0 }
        };
        //  If we have no workers ready, wait indefinitely
        int rc = zmq_poll (primary, 4,
            local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Track if capacity changes during this iteration
```

```
        int previous = local_capacity;
        zmsg_t *msg = NULL;     //  Reply from local worker

        if (primary [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);
            local_capacity++;

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  Or handle reply from peer broker
        else
        if (primary [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker identity for anything
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_destroy (&identity);
        }
        //  Route reply to cloud if it's addressed to a broker
        for (int argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
```

```
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  .split handle state messages
        //  If we have input messages on our statefe or monitor sockets, we
        //  can process these immediately:

        if (primary [2].revents & ZMQ_POLLIN) {
            char *peer = zstr_recv (statefe);
            char *status = zstr_recv (statefe);
            cloud_capacity = atoi (status);
            free (peer);
            free (status);
        }
        if (primary [3].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (monitor);
            printf ("%s\n", status);
            free (status);
        }
        //  .split route client requests
        //  Now route as many clients requests as we can handle. If we have
        //  local capacity, we poll both localfe and cloudfe. If we have cloud
        //  capacity only, we poll just localfe. We route any request locally
```

```
        //  if we can, else we route to the cloud.

        while (local_capacity + cloud_capacity) {
            zmq_pollitem_t secondary [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            if (local_capacity)
                rc = zmq_poll (secondary, 2, 0);
            else
                rc = zmq_poll (secondary, 1, 0);
            assert (rc >= 0);

            if (secondary [0].revents & ZMQ_POLLIN)
                msg = zmsg_recv (localfe);
            else
            if (secondary [1].revents & ZMQ_POLLIN)
                msg = zmsg_recv (cloudfe);
            else
                break;      //  No work, go back to primary

            if (local_capacity) {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                local_capacity--;
            }
            else {
                //  Route to random broker peer
```

```
                int peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [peer], strlen (argv [peer]));
                zmsg_send (&msg, cloudbe);
            }
        }
        //  .split broadcast capacity
        //  We broadcast capacity messages to other peers; to reduce chatter,
        //  we do this only if our capacity changed.

        if (local_capacity != previous) {
            //  We stick our own identity onto the envelope
            zstr_sendm (statebe, self);
            //  Broadcast new capacity
            zstr_sendf (statebe, "%d", local_capacity);
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
}

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    if (argc < 2) {
```

```
        printf ("syntax: peering3_inproc me {you}...\n");
        return 0;
    }
    zctx_t *ctx = zctx_new ();
    zthread_fork (ctx, broker, argv);
    zclock_sleep(100);
    // Change the order
    int m,n;
    for(n=2; n < argc; n++){
        char *temp = strdup(argv[1]);
        for(m = 2; m < argc; m++)
        {
            argv[m-1] = argv[m];
        }
        argv[m-1] = temp;
        zthread_fork (ctx, broker, argv);
        zclock_sleep(100);
    }
    zclock_sleep (10 * 1000);    //  Run for 10 seconds then quit
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

- [옮긴이] 빌드 및 테스트(peering3_inproc.c)

```
 ./peering3_inproc dog cat fox
I: preparing broker at dog...
[dog] I: connecting to cloud frontend at 'cat'
[dog] I: connecting to cloud frontend at 'fox'
[dog] I: connecting to state backend at 'cat'
```

```
[dog] I: connecting to state backend at 'fox'
I: preparing broker at cat...
[cat] I: connecting to cloud frontend at 'fox'
[dog] Worker received :  59AC
[cat] I: connecting to cloud frontend at 'dog'
[cat] I: connecting to state backend at 'fox'
[dog] Worker received :  59AC
[cat] I: connecting to state backend at 'dog'
I: preparing broker at fox...
[fox] I: connecting to cloud frontend at 'dog'
[dog] Client received :  EB34
[cat] Client received :  317C
[fox] I: connecting to cloud frontend at 'cat'
[fox] I: connecting to state backend at 'dog'
[dog] Worker received :  0240
[fox] I: connecting to state backend at 'cat'
...
```

3개의 스레드(dog, cat, fox)들이 컨텍스트를 공유하며 실행하는 것을 확인할 수 있습니다.

4장-신뢰할 수 있는 요청-응답 패턴

"3장-고급 요청-응답 패턴"은 ØMQ의 요청-응답 패턴의 고급 사용 방법을 예제와 함께 다루었습니다. 이 장에서는 신뢰성에 대한 일반적인 요구 사항을 보고 ØMQ의 핵심 요청-응답 패턴상에서 일련의 신뢰할 수 있는 메시징 패턴을 구축합니다.

이 장에서는 사용자관점의 요청-응답 패턴에 중점을 두고 있으며, 자체 ØMQ 아키텍처를 설계하는 데 도움이 되는 재사용 가능한 모델입니다.

- 게으른 해적 패턴(LPP) : 클라이언트 측의 신뢰할 수 있는 요청-응답
- 단순한 해적 패턴(SPP) : 부하 분산을 사용한 신뢰할 수 있는 요청-응답
- 편집증 해적 패턴(PPP) : 심박을 통한 신뢰할 수 있는 요청-응답
- 집사(majordomo) 패턴(MDP) : 서비스 지향 신뢰할 수 있는 메세지 대기열
- 타이타닉 패턴 : 디스크 기반 / 연결 해제된 신뢰할 수 있는 메세지 대기열
- 바이너리 스타 패턴 : 기본-백업 서버 장애조치
- 프리랜서 패턴 : 브로커 없는 신뢰할 수 있는 요청 응답

0.39 신뢰성이란 무엇인가?

"신뢰성"에 대해 말하는 대부분의 사람들은 의미하는 바를 모릅니다. 신뢰성은 장애의 관점에서만 정의할 수 있습니다. 즉, 일련의 잘 정의되고 이해된 장애를 다룰 수 있다면 이러한 장애의 관점에서 신뢰성이 있다고 할 수 있습니다. 그 이상도 이하도 아닙니다. 따라서 분산된

ØMQ 응용프로그램에서 가능한 장애 원인을 발생 확률에 따라 내림차순(확률이 높은 순)으로 보겠습니다.

- 응용프로그램 코드가 최악의 범죄자입니다. 충돌, 종료, 멈춤, 입력에 대한 무응답, 입력 처리 지연, 메모리 고갈 등이 발생할 수 있습니다.
- 시스템 코드 – ØMQ의 시스템 코드를 사용하여 작성한 브로커는 응용프로그램 코드와 같은 이유로 죽을 수 있습니다. 시스템 코드는 응용프로그램 코드보다 더 신뢰할 수 있어야 하지만 여전히 충돌 및 잘못될 수 있으며 특히 느린 클라이언트에 대한 메시지를 대기열에 넣으려고 하면 메모리가 부족할 수 있습니다.
- 메시지 대기열 초과로 인한 메시지 유실이 발생합니다. 처리가 느린 클라이언트와 함께 동작하는 브로커(시스템 코드)로 인해 메시지 대기열은 초과될 수 있습니다. 대기열이 초과되면 메시지를 버리기 시작하여 메시지 "유실"이 발생합니다.
- 하드웨어 장애 – 하드웨어는 장애가 발생하면 박스에서 실행되는 모든 프로세스들도 영향을 받게 됩니다.
- 네트워크 장애 – 네트워크는 색다르게 장애가 발생합니다. 예를 들어 스위치의 일부 포트가 죽거나 네트워크 상의 일부 지점에 접근할 수 없게 됩니다.
- 전체 데이터 센터 장애 – 데이터 센터는 번개, 지진, 화재, 일상적인 전력 장애, 냉각 장애로 영향을 받을 수 있습니다.

소프트웨어 시스템을 완전히 신뢰할 수 있게 만들어 이러한 모든 가능한 장애에 대응하게 하는 것은 엄청나게 어렵고 비용이 많이 드는 작업이며 이 책의 범위를 벗어납니다.

위 목록의 5개 사례는 대기업을 제외한 실제 요구사항의 99.9%를 해당됩니다(과학적 연구에 따르면 통계의 78%가 즉석해서 만들어진 것으며, 조작되지 않은 통계를 믿을 수 없게 합니다.). 대기업에 속해 있고 마지막 2가지 사안(네트워크, 데이터 센터)에 지출할 돈이 있다면 즉시 저희 회사(iMatix)에 연락 주시기 바랍니다. 저의 별장 뒷편의 공간을 호화로운 수영장으로 바꾸기를 기다리고 있습니다.

0.40 신뢰성 설계

주제를 단순화하면, 신뢰성은 "코드가 멈추거나 충돌할 때 정상적으로 동작하게 유지"하는 것이며 이런 장애 상황을 "죽음"으로 줄여 이야기합니다. 그러나 우리가 정상적으로 동작하고 싶은 것은 단순한 메시지보다 더 복잡합니다. 우리는 각각의 핵심 ØMQ 메시징 패턴을 통해 코드가 죽더라도 동작하는 방법을 살펴보겠습니다.

하나씩 살펴보겠습니다.

- 요청-응답 : 서버가(요청을 처리하는 동안) 죽으면, 클라이언트는 응답을 받지 못하기 때문에 장애를 알 수 있습니다. 그런 다음 잠시 대기하다가 다시 시도하거나 다른 서버를 찾는 등의 작업을 수행할 수 있습니다. 클라이언트가 죽는 거에 대해서는, 지금은 "다른 무엇인가의 문제"로 제외합니다.

- 발행-구독 : 클라이언트가 죽으면(일부 데이터를 얻음) 서버는 클라이언트가 죽은 사실을 알지 못합니다. 발행-구독은 클라이언트에서 서버로 어떤 정보를 보내지 않기 때문입니다. 그러나 클라이언트는 새로운 경로(예 : 요청-응답)로 서버에 접속하여 "내가 놓친 모든 것을 재전송해주십시오"라고 요구할 수 있습니다. 서버가 죽는 것은 이 책의 범위가 아닙니다. 구독자는 자신이 너무 느리게 동작하지 않는지 검증하면서, 느릴 경우 시스템 관리자에게 경고를 보내거나 자살할 수 있습니다.

- 파이프라인 : 작업자가(작업 중) 죽게 되면 호흡기(이전 예제 : ventilator, worker, sink)는 알지 못합니다. 파이프라인은 흐르는 시간처럼 한 방향으로만 동작합니다. 그러나 하류 수집기(sink)는 하나의 작업이 완료되지 않았음을 감지하고 호흡기에게 "이봐, 324 작업 다시 보내!"라는 메시지를 보낼 수 있습니다. 호흡기 또는 수집기가 죽으면, 상류 클라이언트(worker)가 보낸 작업 배치는 대기하는 데 지쳐서 전체 데이터를 재전송할 수 있습니다. 우아하지는 않지만 시스템 코드는 문제가 될 만큼 자주 죽지는 않습니다.

- [옮긴이] 그림 5 - 병렬 파이프라인(parallel pipeline)

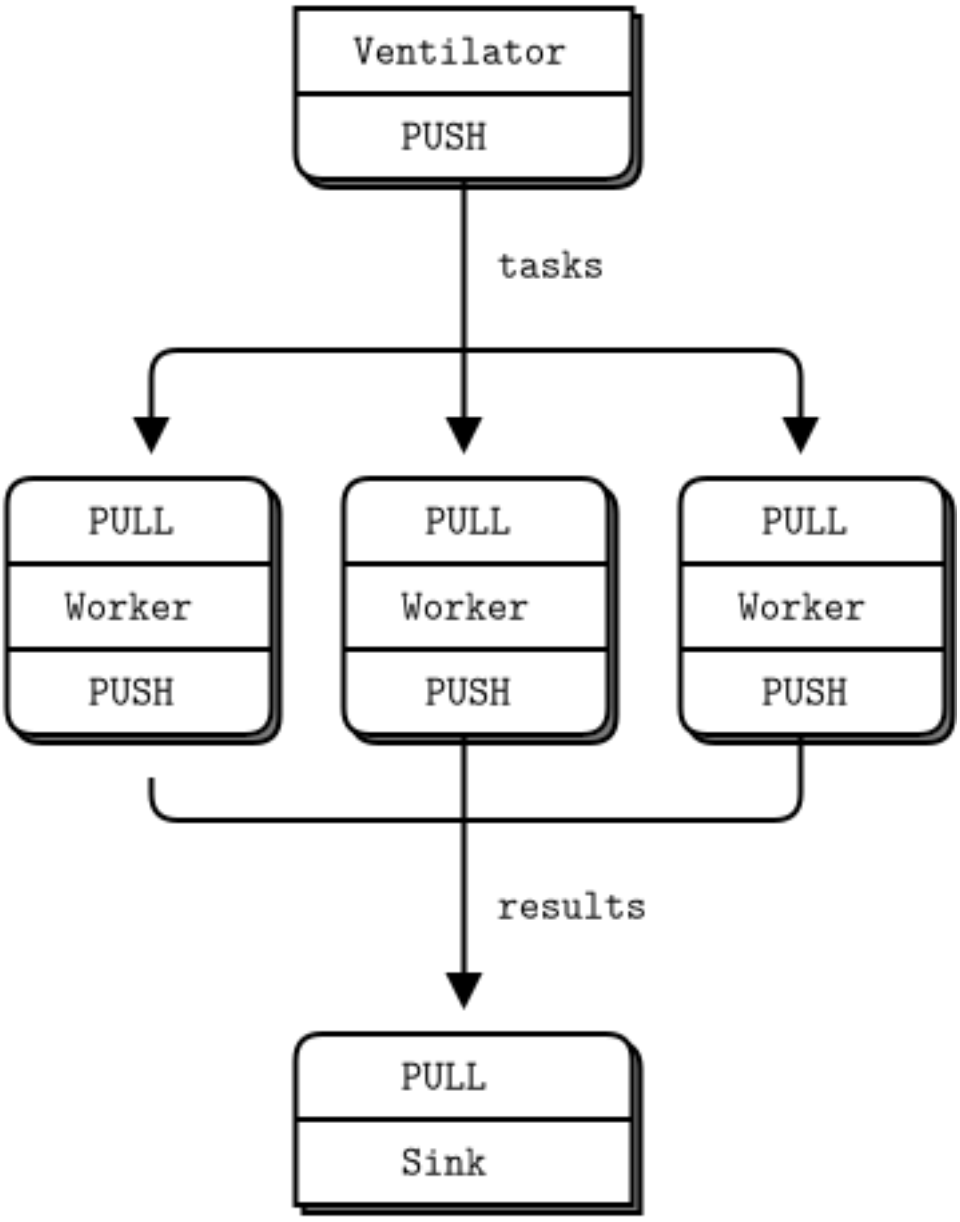


그림 48: 병렬 파이프라인

이 장에서는 신뢰성 있는 메시징에서 요청-응답에만 초점을 맞출 것입니다.

기본적인 요청-응답 패턴(하나의 REQ 클라이언트 소켓이 REP 서버 소켓에 동기식 송/수신)은 매우 일반적인 유형의 장애가 발생합니다. 요청을 처리하는 동안 서버가 충돌하거나

네트워크가 송/수신 데이터를 유실하면 클라이언트는 영원히 멈추게 됩니다.

OMQ의 요청-응답 패턴은 TCP보다 훨씬 좋습니다. ØMQ는 자동으로 상대들을 다시 연결하고 메시지를 부하 분산하는 등의 기능을 제공합니다. 그러나 실상황에서는 충분하지 않습니다. 기본적인 요청-응답 패턴을 신뢰할 수 있는 유일한 경우는 동일 프로세스에 있는 2개의 스레드들 간이며 장애가 될 수 있는 네트워크 또는 독자적인 서버 프로세스가 없어야 합니다.

그러나 약간의 추가 작업으로 이 겸손한 패턴은 분산된 네트워크에서 실제 작업을 위한 좋은 기반이 되며, 일련의 신뢰성 있는 요청-응답(reliable request-reply, RRR) 패턴들을 만들어 해적 패턴들(Pirate patterns)이라고 부르겠습니다.(그냥 농담입니다.)

경험상 클라이언트를 서버에 연결하는 방법에는 대략 3가지가 있습니다. 각각은 안정성에 대한 고유의 접근 방식이 필요합니다.

- 여러 개의 클라이언트들이 단일 서버와 직접 통신합니다.
- 사용 사례: 잘 알려진 단일 서버에 클라이언트들이 통신해야 합니다.
- 처리할 장애 유형 : 서버 충돌 및 재시작, 네트워크 연결 끊김.
- 여러 개의 클라이언트들이 브로커와 통신하며 브로커는 여러 개의 작업자들에게 작업을 분배합니다.
- 사용 사례 : 서비스 지향 트랜잭션 처리.
- 처리할 장애 유형 : 작업자 충돌 및 재시작, 작업자 바쁜 작업 처리, 작업자 과부하, 대기열 충돌 및 재시작, 네트워크 연결 끊김.
- 여러 개의 클라이언트가 브로커 없이 여러 개의 서버들과 통신합니다.
- 사용 사례 : 이름 확인과 같은 분산 서비스(예 : DNS).
- 처리할 장애 유형 : 서비스 충돌 및 재시작, 서비스 바쁜 작업 처리, 서비스 과부하 및 네트워크 연결 끊김.

각 접근 방식에는 장단점이 있으며 함께 사용하는 경우가 있습니다. 세 가지 모두 자세히 살펴보겠습니다.

0.41 클라이언트 측면의 신뢰성(게으른 해적 패턴)

클라이언트 코드의 일부 변경하여 매우 간단하고 신뢰성 있는 요청-응답을 얻을 수 있습니다. 우리는 이것을 게으른 해적 패턴이라고 부릅니다. 수신 차단을 수행하는 대신 다음을 수행합니다. :

- REQ 소켓을 폴링(poll)하고 응답이 도착했을 때만 수신합니다.
- 제한시간 내에 응답이 도착하지 않으면 요청을 재전송합니다.
- 여러 번의 요청들 후에도 여전히 응답이 없으면 트랜잭션을 포기합니다.

엄격한 송/수신 방식 외에 클라이언트에서 REQ 소켓을 사용하려 하면 오류가 발생합니다(기술적으로 REQ 소켓은 송/수신 핑-퐁(ping-pong)을 위해 작은 유한 상태 머신을 구현합니다. 오류 코드는 "EFSM" 입니다.). 해적 패턴으로 REQ 소켓을 사용하는 것을 어렵게 하며, 해적 패턴에서는 응답을 받기 전에 여러 요청(재시도)을 보낼 수 있어야 하기 때문입니다.

아주 무식한 방법으로 응답이 오기 전에 재요청 수행으로 인한 "EFSM" 오류가 발생하면 REQ 소켓을 닫고 다시 열어 요청하는 것입니다.

lpclient.c: 게으른 해적 클라이언트

```
#include <czmq.h>
#define REQUEST_TIMEOUT 2500 // msecs, (>1000!)
#define REQUEST_RETRIES 3 // Before we abandon
#define SERVER_ENDPOINT "tcp://localhost:5555"

int main()
{
    zsock_t *client = zsock_new_req(SERVER_ENDPOINT);
    printf("I: Connecting to server...\n");
    assert(client);

    int sequence = 0;
```

```
    int retries_left = REQUEST_RETRIES;
    printf("Entering while loop...\n");
    while(retries_left) // interrupt needs to be handled
    {
        // We send a request, then we get a reply
        char request[10];
        sprintf(request, "%d", ++sequence);
        zstr_send(client, request);
        int expect_reply = 1;
        while(expect_reply)
        {
            printf("Expecting reply....\n");
            zmq_pollitem_t items [] = {{zsock_resolve(client), 0, ZMQ_POLLIN, 0}};
            printf("After polling\n");
            int rc = zmq_poll(items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            printf("Polling Done.. \n");
            if (rc == -1)
                break; // Interrupted

            // Here we process a server reply and exit our loop if the
            // reply is valid. If we didn't get a reply we close the
            // client socket, open it again and resend the request. We
            // try a number times before finally abandoning:

            if (items[0].revents & ZMQ_POLLIN)
            {
                // We got a reply from the server, must match sequence
                char *reply = zstr_recv(client);
                if(!reply)
```

```
                break; // interrupted
            if (atoi(reply) == sequence)
            {
                printf("I: server replied OK (%s)\n", reply);
                retries_left=REQUEST_RETRIES;
                expect_reply = 0;
            }
            else
            {
                printf("E: malformed reply from server: %s\n", reply);
            }
            free(reply);
        }
        else
        {
            if(--retries_left == 0)
            {
                printf("E: Server seems to be offline, abandoning\n");
                break;
            }
            else
            {
                printf("W: no response from server, retrying...\n");
                zsock_destroy(&client);
                printf("I: reconnecting to server...\n");
                client = zsock_new_req(SERVER_ENDPOINT);
                zstr_send(client, request);
            }
        }
```

```
        }
        zsock_destroy(&client);
        return 0;
    }
}
```

대응하는 서버를 함께 구동합니다.

lpserver.c: 게으른 해적 서버

```
//  Lazy Pirate server
//  Binds REQ socket to tcp://*:5555
//  Like hwserver except:
//   - echoes request as-is
//   - randomly runs slowly, or exits to simulate a crash.

#include "zhelpers.h"
#ifndef _WIN32
#include <unistd.h>
#endif

int main (void)
{
    srandom ((unsigned) time (NULL));

    void *context = zmq_ctx_new ();
    void *server = zmq_socket (context, ZMQ_REP);
    zmq_bind (server, "tcp://*:5555");

    int cycles = 0;
    while (1) {
        char *request = s_recv (server);
```

```
        cycles++;

        //  Simulate various problems, after a few cycles
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: simulating a crash\n");
            break;
        }
        else
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: simulating CPU overload\n");
            s_sleep (2000);
        }
        printf ("I: normal request (%s)\n", request);
        s_sleep (1000);              //  Do some heavy work
        s_send (server, request);
        free (request);
    }
    zmq_close (server);
    zmq_ctx_destroy (context);
    return 0;
}
```

그림 47 - 게으른 해적 패턴(The Lazy Pirate Pattern)

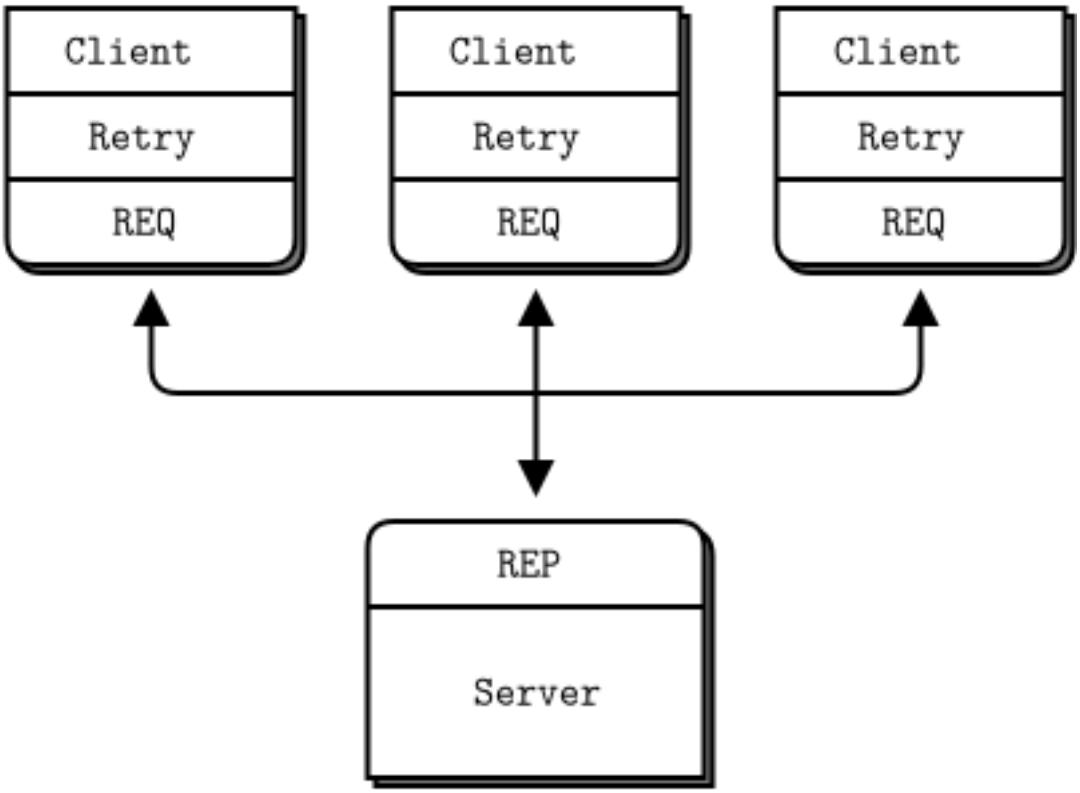


그림 49: The Lazy Pirate Pattern

이 테스트 케이스를 실행하려면 2개의 콘솔 창에서 클라이언트와 서버를 시작하십시오. 서버는 몇 개의 메시지들 받은 이후 무작위로 오동작합니다. 클라이언트의 응답을 확인할 수 있습니다. 다음은 서버부터의 전형적인 출력 예시입니다.

```
I: normal request (1)
I: normal request (2)
I: normal request (3)
I: simulating CPU overload
I: normal request (4)
I: simulating a crash
```

클라이언트의 출력은 다음과 같습니다.

```
I: connecting to server...
I: server replied OK (1)
I: server replied OK (2)
I: server replied OK (3)
W: no response from server, retrying...
I: connecting to server...
W: no response from server, retrying...
I: connecting to server...
E: server seems to be offline, abandoning
```

- [옮긴이] 윈도우 환경에서 구동 가능하도록 일부 수정(`sleep()` 함수)하고 서버(lpserver)를 수행하고 클라이언트(lpclient)를 구동하면 클라이언트는 1회 응답을 받고 종료됩니다.

```
 ./lpserver
I: normal request (1)


 ./lpclient
I: Connecting to server...
Entering while loop...
Expecting reply....
After polling
Polling Done..
I: server replied OK (1)
```

- [옮긴이] 원인은 "lpclient.c"에서 `while(retries_left)` 루프 내에서 소켓을 제거하기 때문이며 루프 밖으로 빼내어 프로그램 종료 시에 수행하도록 수정이 필요합니다.

```
    while(retries_left){
        ...
        zsock_destroy(&client);
        return 0;
    }
```

- [옮긴이] zsock_destroy() 함수의 수행 위치를 변경한 "lpclient.c"는 다음과 같습니다.

```
#include <czmq.h>
#define REQUEST_TIMEOUT 2500 // msecs, (>1000!)
#define REQUEST_RETRIES 3 // Before we abandon
#define SERVER_ENDPOINT "tcp://localhost:5555"

int main()
{
    zsock_t *client = zsock_new_req(SERVER_ENDPOINT);
    printf("I: Connecting to server...\n");
    assert(client);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    printf("Entering while loop...\n");
    while(retries_left) // interrupt needs to be handled
    {
        // We send a request, then we get a reply
        char request[10];
        sprintf(request, "%d", ++sequence);
        zstr_send(client, request);
        int expect_reply = 1;
```

```
        while(expect_reply)
        {
            printf("Expecting reply....\n");
            zmq_pollitem_t items [] = {{zsock_resolve(client), 0, ZMQ_POLLIN, 0}};
            printf("After polling\n");
            int rc = zmq_poll(items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            printf("Polling Done.. \n");
            if (rc == -1)
                break; // Interrupted

            // Here we process a server reply and exit our loop if the
            // reply is valid. If we didn't get a reply we close the
            // client socket, open it again and resend the request. We
            // try a number times before finally abandoning:

            if (items[0].revents & ZMQ_POLLIN)
            {
                // We got a reply from the server, must match sequence
                char *reply = zstr_recv(client);
                if(!reply)
                    break; // interrupted
                if (atoi(reply) == sequence)
                {
                    printf("I: server replied OK (%s)\n", reply);
                    retries_left=REQUEST_RETRIES;
                    expect_reply = 0;
                }
                else
                {
```

```
                printf("E: malformed reply from server: %s\n", reply);
            }
            free(reply);
        }
        else
        {
            if(--retries_left == 0)
            {
                printf("E: Server seems to be offline, abandoning\n");
                break;
            }
            else
            {
                printf("W: no response from server, retrying...\n");
                zsock_destroy(&client);
                printf("I: reconnecting to server...\n");
                client = zsock_new_req(SERVER_ENDPOINT);
                zstr_send(client, request);
            }
        }
    }
    zsock_destroy(&client);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 ./lpserver
I: normal request (1)
I: normal request (2)
I: normal request (3)
I: simulating a crash

 ./lpclient
I: Connecting to server...
Entering while loop...
Expecting reply....
After polling
Polling Done..
I: server replied OK (1)
Expecting reply....
After polling
Polling Done..
I: server replied OK (2)
Expecting reply....
After polling
Polling Done..
I: server replied OK (3)
Expecting reply....
After polling
Polling Done..
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
```

```
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
E: Server seems to be offline, abandoning
```

order: that no requests or replies are lost, and no replies come back more than once, or out of order. Run the test a few times until you're convinced that this mechanism actually works. You don't need sequence numbers in a production application; they just help us trust our design.

클라이언트는 각 메시지의 요청 순서에 따라 응답 순서가 정확히 왔는지 확인합니다 : 요청들이나 응답들이 유실되지 않거나, 응답들이 한번 이상 반환되지 않거나, 순서가 맞지 않을 경우를 확인합니다. 이 동작 방식이 실제로 동작한다고 확신할 때까지 테스트를 수행하시기 바랍니다. 양산 응용프로그램에서는 순서 번호는 필요하지 않지만 설계를 신뢰하는 데 도움이 됩니다.

클라이언트는 REQ 소켓의 엄격한 송/수신 주기를 지키기 위하여 무식한 강제 종료/재오픈을 수행합니다. REQ 소켓(sync) 대신에 DEALER 소켓(async)을 사용하고 싶겠지만 좋은 결정은 아닙니다. 첫째, 그것은 REQ 소켓 봉투들 모방하는 것이 복잡하고(그것이 무엇인지 잊었다면, 그것을 하고 싶지 않은 좋은 징조입니다), 둘째, 예상하지 못한 응답들을 받을 가능성이 있습니다.

- [옮긴이] DEALER의 경우 멀티파트 메시지(multipart message) 형태로 첫 번째 파트는 공백 구분자(empty delimter), 두 번째 파트는 데이터(body)로 REP 소켓에 데이터 전송 필요합니다.
- [옮긴이] REP 대신에 DEALER 소켓을 사용하는 "hwclient2.c"는 다음과 같습니다.

```
/**
**Hello World client
DEALER socket connect to tcp://localhost:5555
```

```
send "Hello" to server , expect receive "World"
*/
#include "zhelpers.h"
// #include <unistd.h>

int main (void)
{
    printf ("Connecting to hello world server...\n");
    void *context = zmq_ctx_new ();
    void *requester = zmq_socket (context, ZMQ_DEALER);
    zmq_connect (requester, "tcp://localhost:5555");
    int request_nbr;    //request number
    int reply_nbr = 0;  //receive respond number
    for (request_nbr = 0; request_nbr < 10; request_nbr++)
    {
        char buffer [10];
        memset(buffer,0,sizeof(buffer));
        printf ("Sending request msg: Hello NO=%d...\n", request_nbr+1);
        //send request msg to server
        s_sendmore(requester,"");  //send multi part msg,the first part is empty part
        zmq_send (requester, "Hello", 5, 0); //the second part is your request msg
        //receive reply msg
        int len;
        len = zmq_recv (requester, buffer, 10, 0);
        if(len == -1){
            printf("Error:%s\n", zmq_strerror(errno));
            exit(-1);
        }
        //if the first part you received is empty part,then continue receiving next part
```

```
        if (strcmp(buffer,"") == 0){
            memset(buffer,0,sizeof(buffer));
            len = zmq_recv(requester, buffer, 10, 0);
            if(len == -1){
                printf("Error:%s\n", zmq_strerror(errno));
                exit(-1);
            }
            printf("Received respond msg: %s NO=%d\n\n", buffer,++reply_nbr);
        }
        //if the first part you received is not empty part,discard the whole ZMQ msg
        else{
            printf("Discard the ZMQ message!\n");
        }
    }
    zmq_close(requester);
    zmq_ctx_destroy (context);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트
- hwserver는 REP소켓을 사용하며 수정 없이 사용 가능합니다.

```
 ./hwserver
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
Received Hello
```

```
Received Hello
Received Hello
Received Hello

 ./hwclient2
Connecting to hello world server...
Sending Hello 0...
Received World 0
Sending Hello 1...
Received World 1
Sending Hello 2...
Received World 2
Sending Hello 3...
Received World 3
Sending Hello 4...
Received World 4
Sending Hello 5...
Received World 5
Sending Hello 6...
Received World 6
Sending Hello 7...
Received World 7
Sending Hello 8...
Received World 8
Sending Hello 9...
Received World 9
```

클라이언트에서만 장애를 처리하는 것은 일련의 클라이언트들이 단일 서버와 통신하는 경우에만 동작합니다. 서버 장애를 처리할 수 있는 경우는 동일 서버를 재시작하여 복구하는 것을 의미합니다. 서버 하드웨어의 전원 공급 중단과 같은 영구적인 오류가 있는 경우 이 접근

방식이 동작하지 않습니다. 서버의 응용프로그램 코드는 보통 대부분의 아키텍처에서 가장 큰 장애 원인이므로 단일 서버에 의존하는 것은 좋은 생각이 아닙니다.

게으른 해적 패턴의 장단점은 다음과 같습니다.

- 장점 : 이해와 구현이 간단합니다.
- 장점 : 기존 클라이언트 및 서버 응용프로그램 코드로 쉽게 동작합니다.
- 장점 : ØMQ는 동작할 때까지 자동으로 재연결을 재시도합니다.
- 단점 : 백업 또는 대체 서버들로 장애조치하지 않습니다.
- [옮긴이] 하나의 프로세스에서 게으른 해적 클라이언트(lpclient)와 서버(lpserver)를 구동하기 위한 예제는 다음과 같습니다(lppattern.c). 클라이언트와 서버의 숫자를 각각 1개로 하는 것에 주의 필요합니다.

```
//  Lazy Pirate

#include <czmq.h>
#include "zhelpers.h"
#define NBR_CLIENTS 1
#define NBR_SERVERS 1
#define REQUEST_TIMEOUT 2500 // msecs, (>1000!)
#define REQUEST_RETRIES 3 // Before we abandon
#define SERVER_ENDPOINT "tcp://localhost:5555"

//  Lazy Pirate client using REQ socket
//
static void *
client_task (void *args)
{
    zsock_t *client = zsock_new_req(SERVER_ENDPOINT);
```

```
    printf("[C%Id] I: Connecting to server...\n", (intptr_t)args);
    assert(client);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    printf("[C%Id] Entering while loop...\n", (intptr_t)args);
    while(retries_left) // interrupt needs to be handled
    {
        char request[10];
        // We send a request, then we get a reply
        sprintf(request, "%d", ++sequence);
        zstr_send(client, request);
        int expect_reply = 1;
        while(expect_reply)
        {
            printf("[C%Id] Expecting reply....\n",  (intptr_t)args);
            zmq_pollitem_t items [] = {{zsock_resolve(client), 0, ZMQ_POLLIN, 0}};
            printf("[C%Id] After polling\n",  (intptr_t)args);
            int rc = zmq_poll(items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            printf("[C%Id] Polling Done.. \n", (intptr_t)args);
            if (rc == -1)
                break; // Interrupted

            // Here we process a server reply and exit our loop if the
            // reply is valid. If we didn't get a reply we close the
            // client socket, open it again and resend the request. We
            // try a number times before finally abandoning:

            if (items[0].revents & ZMQ_POLLIN)
```

```
            {
                // We got a reply from the server, must match sequence
                char *reply = zstr_recv(client);
                if(!reply)
                    break; // interrupted
                if (atoi(reply) == sequence)
                {
                    printf("[C%Id] I: server replied OK (%s)\n", (intptr_t)args, reply);
                    retries_left=REQUEST_RETRIES;
                    expect_reply = 0;
                }
                else
                {
                    printf("[C%Id] E: malformed reply from server: %s\n",(intptr_t)args, reply
                }
                free(reply);
            }
            else
            {
                if(--retries_left == 0)
                {
                    printf("[C%Id] E: Server seems to be offline, abandoning\n", (intptr_t)arg
                    break;
                }
                else
                {
                    printf("[C%Id] W: no response from server, retrying...\n", (intptr_t)args)
                    zsock_destroy(&client);
                    printf("[C%Id] I: reconnecting to server...\n", (intptr_t)args);
```

```
                    client = zsock_new_req(SERVER_ENDPOINT);
                    zstr_send(client, request);
                }
            }
        }
    }
    zsock_destroy(&client);
    return NULL;
}

//  Lazy Pirate server using REQ socket
//
static void *
server_task (void *args)
{
    srandom ((unsigned) time (NULL));

    void *context = zmq_ctx_new ();
    void *server = zmq_socket (context, ZMQ_REP);
    zmq_bind (server, "tcp://*:5555");

    int cycles = 0;
    while (1) {
        char *request = s_recv (server);
        cycles++;

        //  Simulate various problems, after a few cycles
        if (cycles > 3 && randof (3) == 0) {
            printf ("[S%Id] I: simulating a crash\n",(intptr_t)args);
```

```
            break;
        }
        else
        if (cycles > 3 && randof (3) == 0) {
            printf ("[S%Id] I: simulating CPU overload\n",(intptr_t)args);
            s_sleep (2000);
        }
        printf ("[S%Id] I: normal request (%s)\n", (intptr_t)args, request);
        s_sleep (1000);              //  Do some heavy work
        s_send (server, request);
        free (request);
    }
    zmq_close (server);
    zmq_ctx_destroy (context);
    return NULL;
}

int main (void)
{
    int server_nbr;
    for (server_nbr = 0; server_nbr < NBR_SERVERS; server_nbr++)
        zthread_new (server_task, (void *)(intptr_t)server_nbr);

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, (void *)(intptr_t)client_nbr);
    zclock_sleep (60 * 1000);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 ./lppattern
[C0] I: Connecting to server...
[C0] Entering while loop...
[C0] Expecting reply....
[C0] After polling
[S0] I: normal request (1)
[C0] Polling Done..
[C0] I: server replied OK (1)
[C0] Expecting reply....
[C0] After polling
[S0] I: normal request (2)
[C0] Polling Done..
[C0] I: server replied OK (2)
[C0] Expecting reply....
[C0] After polling
[S0] I: normal request (3)
[C0] Polling Done..
[C0] I: server replied OK (3)
[C0] Expecting reply....
[C0] After polling
[S0] I: normal request (4)
[C0] Polling Done..
[C0] I: server replied OK (4)
...
[C0] After polling
[S0] I: simulating CPU overload
[S0] I: normal request (9)
```

```
[C0] Polling Done..
[C0] W: no response from server, retrying...
[C0] I: reconnecting to server...
[C0] Expecting reply....
[C0] After polling
[S0] I: simulating a crash
[C0] Polling Done..
[C0] W: no response from server, retrying...
[C0] I: reconnecting to server...
[C0] Expecting reply....
[C0] After polling
[C0] Polling Done..
[C0] E: Server seems to be offline, abandoning
```

0.42 신뢰할 수 있는 큐잉(단순한 해적 패턴)

두 번째 접근 방식은 대기열 프록시를 사용하여 게으른 해적 패턴을 확장하여 투명하게 여러 대의 서버들과 통신하게 하며, 서버들은 좀 더 정확하게 "작업자들"라고 부를 수 있습니다. 최소한의 구동 모델인 단순한 해적 패턴부터 시작하여 단계별로 개발할 것입니다.

단순한 해적 패턴에서 작업자들은 상태 비저장입니다. 응용프로그램은 데이터베이스와 같이 공유 상태가 필요하지만 메시징 프레임워크를 설계할 당시에는 인지하지 못할 수도 있습니다. 대기열 프록시가 있다는 것은 클라이언트가 작업자들이 오가는 것을 인지하지 못하는 것을 의미합니다. 한 작업자가 죽을 경우, 다른 작업자가 인계받습니다. 이것은 멋지고 간단한 전송 방식이지만 하나의 실제 약점으로, 중앙 대기열 자체가 단일 실패 지점으로 관리해야 할 문제가 됩니다.

- [옮긴이] 상태 비저장(stateless)는 서버가 클라이언트의 이전 상태를 저장하지 않아 클라이언트의 이전 요청과도 무관한 각각의 요청을 독립적으로 처리하게 합니다.(예 : http)

그림 48 - 단순한 해적 패턴(The Simple Pirate Pattern)

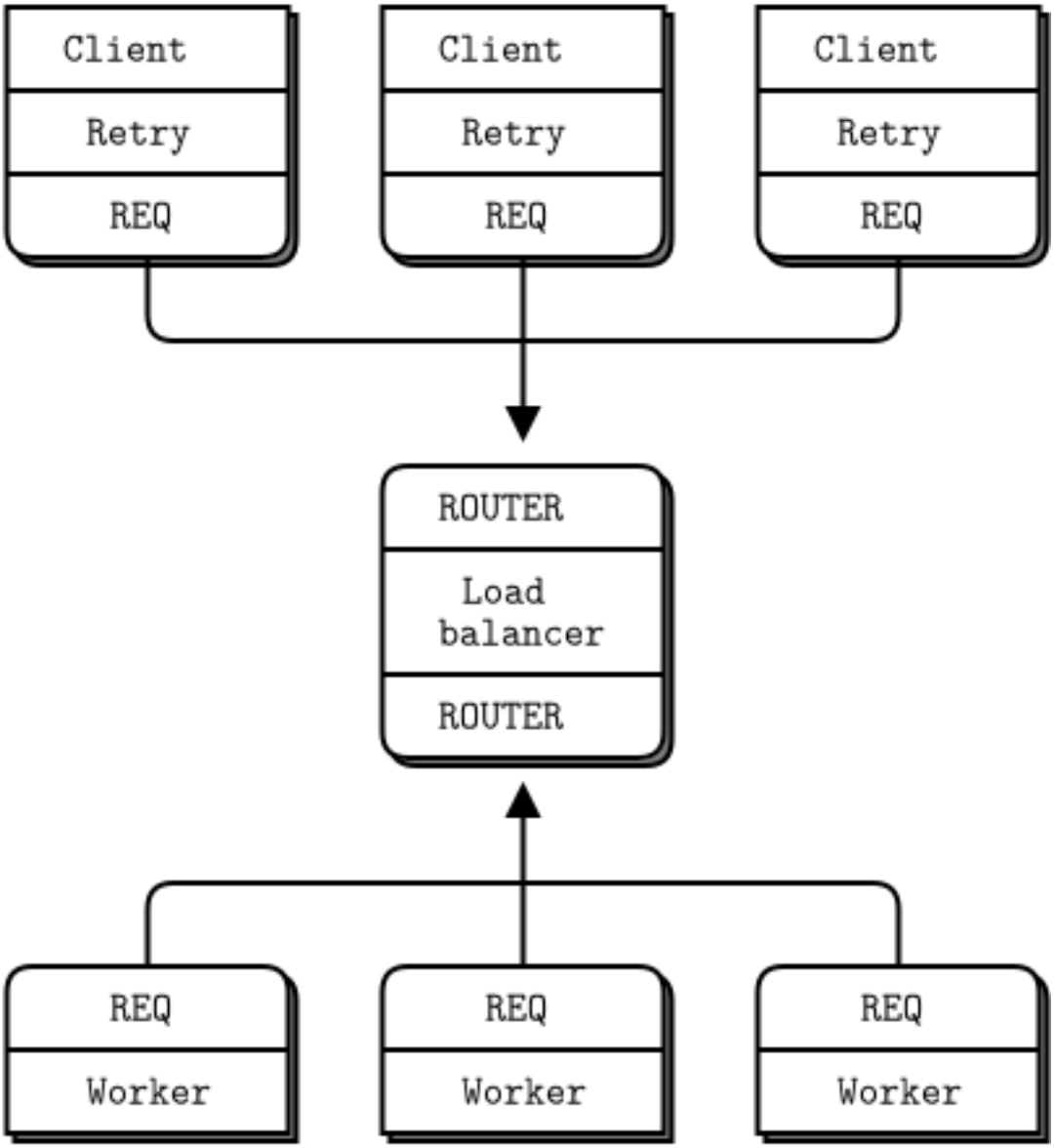


그림 50: The Simple Pirate Pattern

대기열 프록시의 기본은 “3장-고급 요청-응답 패턴”의 부하 분산 브로커입니다. “죽거나 차단된 작업자들을 처리하기 위해 최소한 해야 할 것은 무엇입니까?” 실제 해야 할 것은 의외로

적습니다. 우리는 이미 클라이언트들에서 재시도 기능을 부가한 적 있으며 따라서 부하 분산 패턴을 사용하면 꽤 잘 작동할 것입니다. 이것은 QMQ의 철학에 부합하며 P2P(peer-to-peer)와 같은 요청-응답 패턴 사이에 프록시를 추가하여 확장할 수 있습니다.

여기에는 특별한 클라이언트가 필요하지 않습니다. 이전의 게으른 해적 클라이언트(lpclient)를 사용할 수 있으며, 대기열은 부하 분산 브로커의 기본 작업과 동일합니다 :

spqueue.c: 단순한 해적 브로커

```
//  Simple Pirate broker
//  This is identical to load-balancing pattern, with no reliability
//  mechanisms. It depends on the client for recovery. Runs forever.

#include "czmq.h"
#define WORKER_READY   "\001"      //  Signals worker is ready

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  For clients
    zsocket_bind (backend,  "tcp://*:5556");    //  For workers

    //  Queue of available workers
    zlist_t *workers = zlist_new ();

    //  The body of this example is exactly the same as lbbroker2.
    //  .skip
    while (true) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
```

```
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            //  Use worker identity for load-balancing
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);

            //  Forward message to client if it's not a READY
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Get client request, route to first available worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
```

```
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
    //  .until
}
```

다음은 작업자 코드로, 게으른 해적 서버(lpserver)를 가져 와서 부하분산 패턴에 맞게 적용하였습니다.(REQ 소켓 및 "준비(ready)" 신호 사용).

spworker.c: 단순한 해적 작업자

```
//  Simple Pirate worker
//  Connects REQ socket to tcp://*:5556
//  Implements worker part of load-balancing

#include "czmq.h"
#include "zhelpers.h"
#define WORKER_READY   "\001"      //  Signals worker is ready

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
```

```
    //  Set random identity to make tracing easier
    srandom ((unsigned) time (NULL));
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen (identity));
    zsocket_connect (worker, "tcp://localhost:5556");

    //  Tell broker we're ready for work
    printf ("I: (%s) worker ready\n", identity);
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    int cycles = 0;
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted

        //  Simulate various problems, after a few cycles
        cycles++;
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating a crash\n", identity);
            zmsg_destroy (&msg);
            break;
        }
        else
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating CPU overload\n", identity);
            s_sleep (3000);
```

```
            if (zctx_interrupted)
                break;
        }
        printf ("I: (%s) normal reply\n", identity);
        s_sleep (1000);              //  Do some heavy work
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return 0;
}
```

이를 테스트하려면 순서에 관계없이 여러 개의 작업자들, 게으른 해적 클라이언트(lpclient) 및 대기열(spqueue)을 시작하십시오. 그러면 결국 작업자들이 모두 중단되는 것을 볼 수 있으며 클라이언트는 재시도(3회)를 수행한 후 포기합니다. 대기열은 멈추지 않으며 작업자들과 클라이언트들을 계속해서 재시작할 수 있습니다. 이 모델은 어떤 수량의 클라이언트들과 작업자라도 함께 작동합니다.

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc spqueue.c libzmq.lib czmq.lib
 cl -EHsc spworker.c libzmq.lib czmq.lib

 ./spqueue

 ./spworker
I: (7B44-8FFC) worker ready
...
I: (7B44-8FFC) simulating a crash

 ./spworker
I: (DB7E-8FC8) worker ready
```

```
...
I: (DB7E-8FC8) simulating CPU overload

 ./lpclient
I: Connecting to server...
Entering while loop...
Expecting reply....
After polling
Polling Done..
I: server replied OK (1)
Expecting reply....
After polling
Polling Done..
...
I: server replied OK (21)
Expecting reply....
After polling
Polling Done..
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
I: server replied OK (22)
Expecting reply....
After polling
Polling Done..
...
W: no response from server, retrying...
```

```
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
E: Server seems to be offline, abandoning
```

- [옮긴이] 윈도우 파워쉘(Powershell)에서 구동 중인 프로세스 검색 및 중지하는 방법은 다음과 같습니다.

```
 tasklist | findstr spqueue
spqueue.exe                  9276 Console                    1      4,812 K
 stop-process -name spqueue
## "taskkill /F /IM" 사용할 수도 있음
 tasklist | findstr spqueue
```

- [옮긴이] 하나의 프로세스에서 게으른 해적 클라이언트(lpclient)와 단순한 해적 브로커(spqueue), 단순한 해적 작업자(spworker)를 구동하기 위한 예제는 다음과 같습니다(sppattern.c). 작업자 스레드 기동시 시간 간격을 없을 경우 ID가 모두 동일하게 생성되어 프로그램이 정상 동작하지 않습니다.

```
//  Simple Pirate Pattern


#include <czmq.h>
#include "zhelpers.h"
#define NBR_CLIENTS 3
#define NBR_WORKERS 3
```

```
#define REQUEST_TIMEOUT 2500 // msecs, (>1000!)
#define REQUEST_RETRIES 3 // Before we abandon
#define FRONTEND "tcp://localhost:5555"
#define BACKEND "tcp://localhost:5556"
#define WORKER_READY   "\001"      //  Signals worker is ready


//  Lazy Pirate client using REQ socket
//
static void *
client_task (void *args)
{
    zsock_t *client = zsock_new_req(FRONTEND);
    printf("[C%Id] I: Connecting to server...\n", (intptr_t)args);
    assert(client);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    printf("[C%Id] Entering while loop...\n", (intptr_t)args);
    while(retries_left) // interrupt needs to be handled
    {
        char request[10];
        // We send a request, then we get a reply
        sprintf(request, "%d", ++sequence);
        zstr_send(client, request);
        int expect_reply = 1;
        while(expect_reply)
        {
            printf("[C%Id] Expecting reply....\n",  (intptr_t)args);
```

```
            zmq_pollitem_t items [] = {{zsock_resolve(client), 0, ZMQ_POLLIN, 0}};
            printf("[C%Id] After polling\n",  (intptr_t)args);
            int rc = zmq_poll(items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            printf("[C%Id] Polling Done.. \n", (intptr_t)args);
            if (rc == -1)
                break; // Interrupted

            // Here we process a server reply and exit our loop if the
            // reply is valid. If we didn't get a reply we close the
            // client socket, open it again and resend the request. We
            // try a number times before finally abandoning:

            if (items[0].revents & ZMQ_POLLIN)
            {
                // We got a reply from the server, must match sequence
                char *reply = zstr_recv(client);
                if(!reply)
                    break; // interrupted
                if (atoi(reply) == sequence)
                {
                    printf("[C%Id] I: server replied OK (%s)\n", (intptr_t)args, reply);
                    retries_left=REQUEST_RETRIES;
                    expect_reply = 0;
                }
                else
                {
                    printf("[C%Id] E: malformed reply from server: %s\n",(intptr_t)args, reply
                }
                free(reply);
```

```
            }
            else
            {
                if(--retries_left == 0)
                {
                    printf("[C%Id] E: Server seems to be offline, abandoning\n", (intptr_t)arg
                    break;
                }
                else
                {
                    printf("[C%Id] W: no response from server, retrying...\n", (intptr_t)args)
                    zsock_destroy(&client);
                    printf("[C%Id] I: reconnecting to server...\n", (intptr_t)args);
                    client = zsock_new_req(FRONTEND);
                    zstr_send(client, request);
                }
            }
        }
    }
    zsock_destroy(&client);
    return NULL;
}

//  Lazy Pirate server using REQ socket
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
```

```
    void *worker = zsocket_new (ctx, ZMQ_REQ);

    //  Set random identity to make tracing easier
    srand((unsigned) time (NULL));
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen (identity));
    zsocket_connect (worker, BACKEND);

    //  Tell broker we're ready for work
    printf ("[W%Id] I: (%s) worker ready\n", (intptr_t)args, identity);
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    int cycles = 0;
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted

        //  Simulate various problems, after a few cycles
        cycles++;
        if (cycles > 3 && randof (5) == 0) {
            printf ("[W%Id] I: (%s) simulating a crash\n", (intptr_t)args, identity);
            zmsg_destroy (&msg);
            break;
        }
        else
        if (cycles > 3 && randof (5) == 0) {
```

```
            printf ("[W%Id] I: (%s) simulating CPU overload\n", (intptr_t)args, identity);
            s_sleep (3000);
            if (zctx_interrupted)
                break;
        }
        printf ("[W%Id] I: (%s) normal reply\n", (intptr_t)args, identity);
        s_sleep (1000);              //  Do some heavy work
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  For clients
    zsocket_bind (backend,  "tcp://*:5556");    //  For workers

    //  Queue of available workers
    zlist_t *workers = zlist_new ();

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++){
        zthread_new (worker_task, (void *)(intptr_t)worker_nbr);
        s_sleep(1000);
    }
```

```
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, (void *)(intptr_t)client_nbr);

         //  The body of this example is exactly the same as lbbroker2.
    //  .skip
    while (true) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            //  Use worker identity for load-balancing
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);

            //  Forward message to client if it's not a READY
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
```

```
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Get client request, route to first available worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc sppattern.c libzmq.lib czmq.lib

 ./sppattern
[W0] I: (3B32-C588) worker ready
[W1] I: (8F2A-C58E) worker ready
```

```
[W2] I: (E324-C594) worker ready
[C0] I: Connecting to server...
[C0] Entering while loop...
[C0] Expecting reply....
[C0] After polling
[W0] I: (3B32-C588) normal reply
[C2] I: Connecting to server...
[C2] Entering while loop...
[C2] Expecting reply....
[C2] After polling
[C1] I: Connecting to server...
[C1] Entering while loop...
[C1] Expecting reply....
...
[C2] E: Server seems to be offline, abandoning
...
[W1] I: (8F2A-C58E) simulating a crash
[C0] Polling Done..
[C0] E: Server seems to be offline, abandoning
...
[W2] I: (E324-C594) simulating a crash
...
[W0] I: (3B32-C588) simulating a crash
[C1] Polling Done..
[C1] E: Server seems to be offline, abandoning
```

0.43 튼튼한 큐잉(편집증 해적 패턴)

그림 49 - 편집증 해적 패턴

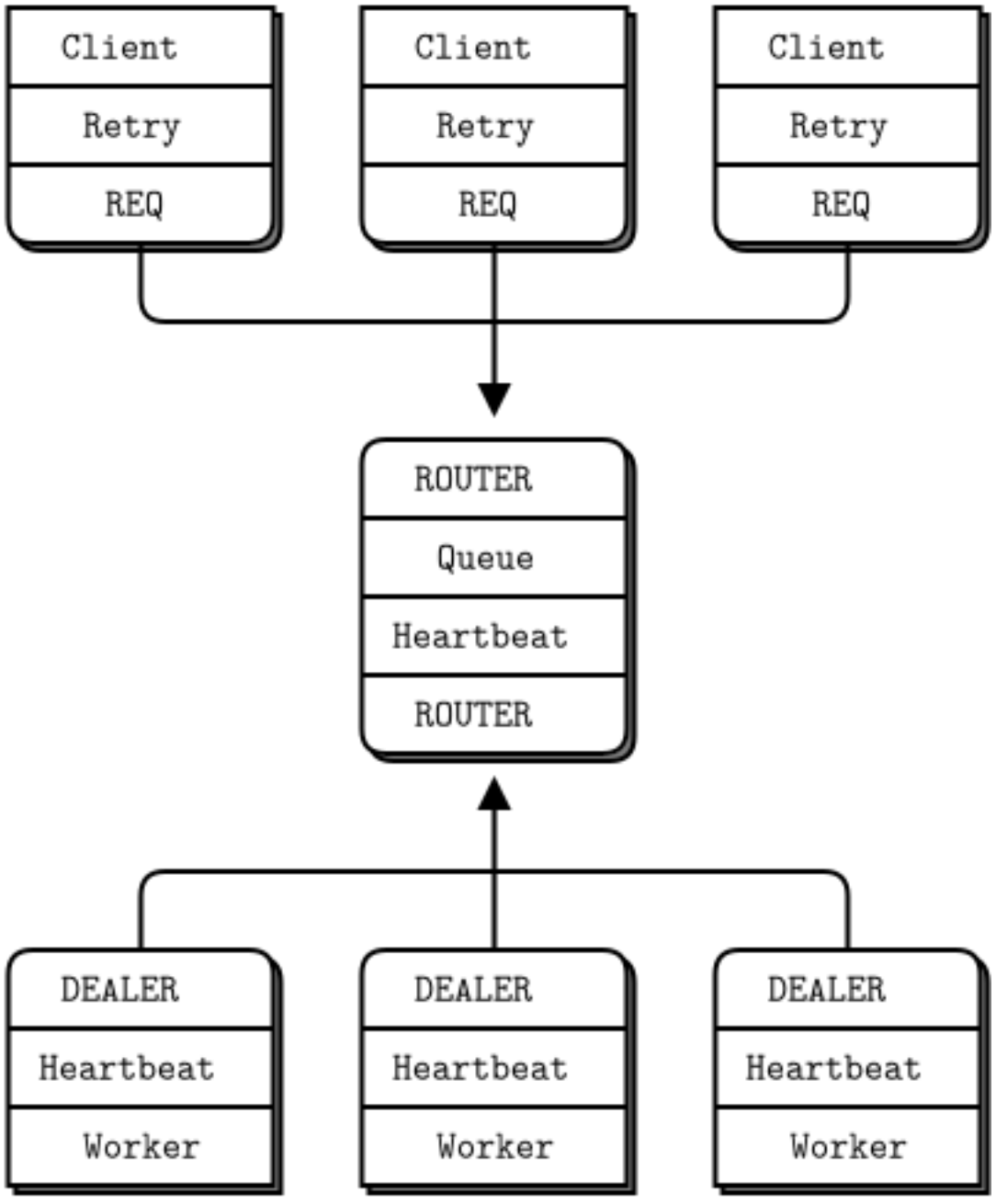


그림 51: Paranoid Pirate Pattern

단순한 해적 대기열 패턴은 기존 2가지 기존 패턴(게으른 해적, 부하 분산 브로커)의

조합이기 때문에 잘 작동합니다. 하지만 몇 가지 약점이 있습니다.

- [Heatbeat : Queue to Worker] 대기열 프록시 장애와 재시작의 경우 안정적이지 않습니다. 클라이언트는 복구되지만 작업자는 복구되지 않습니다. ØMQ는 작업자의 소켓을 자동으로 다시 연결하지만, 재시작된 대기열 프록시에 작업자는 준비(ready) 신호를 보내지 않았으므로 브로커의 리스트(zlist_t)상에 존재하지 않아 클라이언트의 요청을 처리할 수 없습니다. 이 문제를 해결하려면 대기열 프록시에서 작업자로 심박을 전송하여 작업자는 대기열 프록시가 장애가 일어난 시각을 알 수 있습니다.
- [Heatbeat : Worker to Queue] 대기열 프록시는 작업자 장애를 감지하지 않으므로, 작업자가 유휴 상태에서 죽으면 클라이언트가 대기열 프록시로 요청을 보낼 때까지 죽은 작업자를 대기열 프록시(zlist_t)에서 제거할 수 없습니다. 클라이언트는 아무것도 기다리지 않고 재시도합니다. 중대한 문제는 아니지만 좋지 않습니다. 제대로 된 작업을 수행하려면, 작업자가 대기열 프록시로 심박을 전송하여 대기열 프록시의 송/수신 단계에서 죽은 작업자를 감지할 수 있습니다.

위의 약점을 보완하여 제대로 된 편집증 해적 패턴으로 수정하겠습니다.

이전 예제에서는 작업자(spworker)에 대해 REQ 소켓을 사용했지만, 편집증 해적 작업자의 경우 DEALER 소켓으로 바꿉니다. DEALER 소켓을 사용함으로 REQ의 잠금 단계 송/수신(sync) 대신에 언제든지 메시지 송/수신할(async) 수 있는 장점이 있습니다. DEALER 소켓의 단점은 봉투(empty delimiter + body) 관리가 필요합니다(이 개념에 대한 배경 지식은 "3장-고급 요청-응답 패턴"을 참조하시기 바랍니다.).

우리는 계속 게으른 해적 클라이언트를 사용하겠으며, 편집증 해적 대기열 프록시의 코드는 다음과 같습니다. 다음과 같습니다.

ppqueue.c: 편집증 해적 대기열

```
//  Paranoid Pirate queue

#include "czmq.h"
#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  1000    //  msecs
```

```
//  Paranoid Pirate Protocol constants
#define PPP_READY       "\001"      //  Signals worker is ready
#define PPP_HEARTBEAT   "\002"      //  Signals worker heartbeat

//  .split worker class structure
//  Here we define the worker class; a structure and a set of functions that
//  act as constructor, destructor, and methods on worker objects:

typedef struct {
    zframe_t *identity;         //  Identity of worker
    char *id_string;            //  Printable identity
    int64_t expiry;             //  Expires at this time
} worker_t;

//  Construct new worker
static worker_t *
s_worker_new (zframe_t *identity)
{
    worker_t *self = (worker_t *) zmalloc (sizeof (worker_t));
    self->identity = identity;
    self->id_string = zframe_strhex (identity);
    self->expiry = zclock_time ()
                 + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS;
    return self;
}

//  Destroy specified worker object, including identity frame.
static void
s_worker_destroy (worker_t **self_p)
```

```
{
    assert (self_p);
    if (*self_p) {
        worker_t *self = *self_p;
        zframe_destroy (&self->identity);
        free (self->id_string);
        free (self);
        *self_p = NULL;
    }
}

//  .split worker ready method
//  The ready method puts a worker to the end of the ready list:

static void
s_worker_ready (worker_t *self, zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (streq (self->id_string, worker->id_string)) {
            zlist_remove (workers, worker);
            s_worker_destroy (&worker);
            break;
        }
        worker = (worker_t *) zlist_next (workers);
    }
    zlist_append (workers, self);
}
```

```
//  .split get next available worker
//  The next method returns the next available worker identity:

static zframe_t *
s_workers_next (zlist_t *workers)
{
    worker_t *worker = zlist_pop (workers);
    assert (worker);
    zframe_t *frame = worker->identity;
    worker->identity = NULL;
    s_worker_destroy (&worker);
    return frame;
}

//  .split purge expired workers
//  The purge method looks for and kills expired workers. We hold workers
//  from oldest to most recent, so we stop at the first alive worker:

static void
s_workers_purge (zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              //  Worker is alive, we're done here

        zlist_remove (workers, worker);
        s_worker_destroy (&worker);
        worker = (worker_t *) zlist_first (workers);
```

```
    }
}

//  .split main task
//  The main task is a load-balancer with heartbeating on workers so we
//  can detect crashed or blocked worker tasks:

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  For clients
    zsocket_bind (backend,  "tcp://*:5556");    //  For workers

    //  List of available workers
    zlist_t *workers = zlist_new ();

    //  Send out heartbeats at regular intervals
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    while (true) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1,
            HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
```

```
        if (rc == -1)
            break;              //  Interrupted

        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            //  Use worker identity for load-balancing
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  Interrupted

            //  Any sign of life from worker means it's ready
            zframe_t *identity = zmsg_unwrap (msg);
            worker_t *worker = s_worker_new (identity);
            s_worker_ready (worker, workers);

            //  Validate control message, or return reply to client
            if (zmsg_size (msg) == 1) {
                zframe_t *frame = zmsg_first (msg);
                if (memcmp (zframe_data (frame), PPP_READY, 1)
                &&  memcmp (zframe_data (frame), PPP_HEARTBEAT, 1)) {
                    printf ("E: invalid message from worker");
                    zmsg_dump (msg);
                }
                zmsg_destroy (&msg);
            }
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
```

```
            //  Now get next client request, route to next worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (!msg)
                break;          //  Interrupted
            zframe_t *identity = s_workers_next (workers);
            zmsg_prepend (msg, &identity);
            zmsg_send (&msg, backend);
        }
        //  .split handle heartbeating
        //  We handle heartbeating after any socket activity. First, we send
        //  heartbeats to any idle workers if it's time. Then, we purge any
        //  dead workers:
        if (zclock_time () >= heartbeat_at) {
            worker_t *worker = (worker_t *) zlist_first (workers);
            while (worker) {
                zframe_send (&worker->identity, backend,
                             ZFRAME_REUSE + ZFRAME_MORE);
                zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
                zframe_send (&frame, backend, 0);
                worker = (worker_t *) zlist_next (workers);
            }
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
        s_workers_purge (workers);
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        worker_t *worker = (worker_t *) zlist_pop (workers);
        s_worker_destroy (&worker);
```

```
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

대기열 프록시는 작업자들에게 심박 전송으로 부하 분산 패턴을 확장합니다. 심박은 바로 이해하기 어려운 "단순한(simple)" 것 중 하나입니다. 이에 대해서는 잠시 후에 설명하겠습니다.

다음 코드는 편집증 해적 작업자입니다.。

ppworker.c: 편집증 해적 작업자

```
//  Paranoid Pirate worker

#include "czmq.h"
#include "zhelpers.h"
#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  1000    //  msecs
#define INTERVAL_INIT       1000    //  Initial reconnect
#define INTERVAL_MAX       32000    //  After exponential backoff

//  Paranoid Pirate Protocol constants
#define PPP_READY       "\001"      //  Signals worker is ready
#define PPP_HEARTBEAT   "\002"      //  Signals worker heartbeat

//  Helper function that returns a new configured socket
//  connected to the Paranoid Pirate queue

static void *
s_worker_socket (zctx_t *ctx) {
```

```
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");

    //  Tell queue we're ready for work
    printf ("I: worker ready\n");
    zframe_t *frame = zframe_new (PPP_READY, 1);
    zframe_send (&frame, worker, 0);

    return worker;
}

//  .split main task
//  We have a single task that implements the worker side of the
//  Paranoid Pirate Protocol (PPP). The interesting parts here are
//  the heartbeating, which lets the worker detect if the queue has
//  died, and vice versa:

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = s_worker_socket (ctx);

    //  If liveness hits zero, queue is considered disconnected
    size_t liveness = HEARTBEAT_LIVENESS;
    size_t interval = INTERVAL_INIT;

    //  Send out heartbeats at regular intervals
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
```

```
    srandom ((unsigned) time (NULL));
    int cycles = 0;
    while (true) {
        zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        if (items [0].revents & ZMQ_POLLIN) {
            //  Get message
            //  - 3-part envelope + content -> request
            //  - 1-part HEARTBEAT -> heartbeat
            zmsg_t *msg = zmsg_recv (worker);
            if (!msg)
                break;          //  Interrupted

            //  .split simulating problems
            //  To test the robustness of the queue implementation we
            //  simulate various typical problems, such as the worker
            //  crashing or running very slowly. We do this after a few
            //  cycles so that the architecture can get up and running
            //  first:
            if (zmsg_size (msg) == 3) {
                cycles++;
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: simulating a crash\n");
                    zmsg_destroy (&msg);
                    break;
                }
```

```
            else
            if (cycles > 3 && randof (5) == 0) {
                printf ("I: simulating CPU overload\n");
                s_sleep (3000);
                if (zctx_interrupted)
                    break;
            }
            printf ("I: normal reply\n");
            zmsg_send (&msg, worker);
            liveness = HEARTBEAT_LIVENESS;
            s_sleep (1000);              //  Do some heavy work
            if (zctx_interrupted)
                break;
        }
        else
        //  .split handle heartbeats
        //  When we get a heartbeat message from the queue, it means the
        //  queue was (recently) alive, so we must reset our liveness
        //  indicator:
        if (zmsg_size (msg) == 1) {
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), PPP_HEARTBEAT, 1) == 0)
                liveness = HEARTBEAT_LIVENESS;
            else {
                printf ("E: invalid message\n");
                zmsg_dump (msg);
            }
            zmsg_destroy (&msg);
        }
```

```
            else {
                printf ("E: invalid message\n");
                zmsg_dump (msg);
            }
            interval = INTERVAL_INIT;
        }
        else
        //  .split detecting a dead queue
        //  If the queue hasn't sent us heartbeats in a while, destroy the
        //  socket and reconnect. This is the simplest most brutal way of
        //  discarding any messages we might have sent in the meantime:
        if (--liveness == 0) {
            printf ("W: heartbeat failure, can't reach queue\n");
            printf ("W: reconnecting in %zd msec...\n", interval);
            zclock_sleep (interval);

            if (interval < INTERVAL_MAX)
                interval *= 2;
            zsocket_destroy (ctx, worker);
            worker = s_worker_socket (ctx);
            liveness = HEARTBEAT_LIVENESS;
        }
        //  Send heartbeat to queue if it's time
        if (zclock_time () > heartbeat_at) {
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
            printf ("I: worker heartbeat\n");
            zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
            zframe_send (&frame, worker, 0);
        }
```

```
    }
    zctx_destroy (&ctx);
    return 0;
}
```

예제에 대하여 부가적인 설명은 다음과 같습니다.

- 코드에는 이전과 같이 장애 상황(crash, CPU overload)이 포함되어 있습니다. 이로 인해 (a) 디버그 하기가 매우 어렵고, (b) 재사용하기가 위험합니다. 디버그 하기 위해서는 장애 상황 비활성화가 필요합니다.
- 작업자는 게으른 해적 클라이언트용으로 설계한 것과 유사한 재연결 전략을 사용합니다. 두 가지 주요 차이점이 있습니다. (a) 지수 백오프(3회 응답 대기 후 "interval *= 2"하여 최대 32초간 대기)를 수행하고, (b) 무한히 재시도합니다 (클라이언트는 실패를 보고하기 전에 2차례 재시도).

아래의 bash 쉘스크립터를 통하여 클라이언트와 대기열 프로시, 작업자들을 시작합니다.

```
ppqueue &
for i in 1 2 3 4; do
    ppworker &
    sleep 1
done
lpclient &
```

작업자들이 중단되면서 하나씩 죽는 것을 볼 수 있으며 클라이언트는 결국 포기합니다. 대기열 프록시를 중지하고 재시작하면 클라이언트와 작업자들이 다시 연결하여 동작합니다. 그리고 당신이 대기열 프록시와 작업자들에 대하여 무엇을 하든 클라이언트는 요청 순서에 어긋나는 응답을 받지 않을 것입니다 : 전체 체인(workers-queue-client)이 동작하거나 클라이언트가 포기합니다.

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc ppqueue.c libzmq.lib czmq.lib
 cl -EHsc ppworker.c libzmq.lib czmq.lib

S D:\git_store\zguide-kr\examples\C> ./ppqueue

 ./ppworker
I: worker ready
I: worker heartbeat
I: normal reply
I: worker heartbeat
I: normal reply
I: worker heartbeat
I: simulating CPU overload
I: normal reply
I: worker heartbeat
I: simulating a crash

 ./lpclient
I: Connecting to server...
Entering while loop...
Expecting reply....
After polling
Polling Done..
I: server replied OK (1)
Expecting reply....
After polling
Polling Done..
I: server replied OK (2)
Expecting reply....
```

```
After polling
Polling Done..
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
I: server replied OK (3)
Expecting reply....
After polling
Polling Done..
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
W: no response from server, retrying...
I: reconnecting to server...
Expecting reply....
After polling
Polling Done..
E: Server seems to be offline, abandoning
```

- [옮긴이] 브로커(ppqueue)와 작업자(ppworker) 및 클라이언트(lpclient)를 하나의 프로세스에서 동작하도록 변경한 코드(pppattern.c)는 다음과 같습니다.

```
//  Simple Pirate Pattern

#include <czmq.h>
#include "zhelpers.h"
#define NBR_CLIENTS 3
#define NBR_WORKERS 3
#define REQUEST_TIMEOUT 2500 // msecs, (>1000!)
#define REQUEST_RETRIES 3 // Before we abandon
#define FRONTEND "tcp://localhost:5555"
#define BACKEND "tcp://localhost:5556"
#define WORKER_READY   "\001"      //  Signals worker is ready
#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  1000    //  msecs
#define INTERVAL_INIT       1000    //  Initial reconnect
#define INTERVAL_MAX       32000    //  After exponential backoff

//  Paranoid Pirate Protocol constants
#define PPP_READY       "\001"      //  Signals worker is ready
#define PPP_HEARTBEAT   "\002"      //  Signals worker heartbeat

//  Lazy Pirate client using REQ socket
//
static void *
client_task (void *args)
{
    zsock_t *client = zsock_new_req(FRONTEND);
    printf("[C%Id] I: Connecting to server...\n", (intptr_t)args);
    assert(client);
```

```
    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    printf("[C%Id] Entering while loop...\n", (intptr_t)args);
    while(retries_left) // interrupt needs to be handled
    {
        char request[10];
        // We send a request, then we get a reply
        sprintf(request, "%d", ++sequence);
        zstr_send(client, request);
        int expect_reply = 1;
        while(expect_reply)
        {
            printf("[C%Id] Expecting reply....\n",  (intptr_t)args);
            zmq_pollitem_t items [] = {{zsock_resolve(client), 0, ZMQ_POLLIN, 0}};
            printf("[C%Id] After polling\n",  (intptr_t)args);
            int rc = zmq_poll(items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            printf("[C%Id] Polling Done.. \n", (intptr_t)args);
            if (rc == -1)
                break; // Interrupted

            // Here we process a server reply and exit our loop if the
            // reply is valid. If we didn't get a reply we close the
            // client socket, open it again and resend the request. We
            // try a number times before finally abandoning:

            if (items[0].revents & ZMQ_POLLIN)
            {
                // We got a reply from the server, must match sequence
                char *reply = zstr_recv(client);
```

```
            if(!reply)
                break; // interrupted
            if (atoi(reply) == sequence)
            {
                printf("[C%Id] I: server replied OK (%s)\n", (intptr_t)args, reply);
                retries_left=REQUEST_RETRIES;
                expect_reply = 0;
            }
            else
            {
                printf("[C%Id] E: malformed reply from server: %s\n",(intptr_t)args, reply
            }
            free(reply);
        }
        else
        {
            if(--retries_left == 0)
            {
                printf("[C%Id] E: Server seems to be offline, abandoning\n", (intptr_t)arg
                break;
            }
            else
            {
                printf("[C%Id] W: no response from server, retrying...\n", (intptr_t)args)
                zsock_destroy(&client);
                printf("[C%Id] I: reconnecting to server...\n", (intptr_t)args);
                client = zsock_new_req(FRONTEND);
                zstr_send(client, request);
            }
```

```
            }
        }
    }
    zsock_destroy(&client);
    return NULL;
}

//  Helper function that returns a new configured socket
//  connected to the Paranoid Pirate queue

static void *
s_worker_socket (zctx_t *ctx) {
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, BACKEND);

    //  Tell queue we're ready for work
    printf ("I: worker ready\n");
    zframe_t *frame = zframe_new (PPP_READY, 1);
    zframe_send (&frame, worker, 0);

    return worker;
}

//  .split main task
//  We have a single task that implements the worker side of the
//  Paranoid Pirate Protocol (PPP). The interesting parts here are
//  the heartbeating, which lets the worker detect if the queue has
//  died, and vice versa:
//
```

```
static void *
worker_task (void *args)
{
  zctx_t *ctx = zctx_new ();
  void *worker = s_worker_socket (ctx);

  //  If liveness hits zero, queue is considered disconnected
  size_t liveness = HEARTBEAT_LIVENESS;
  size_t interval = INTERVAL_INIT;

  //  Send out heartbeats at regular intervals
  uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

  srandom ((unsigned) time (NULL));
  int cycles = 0;
  while (true) {
      zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN, 0 } };
      int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
      if (rc == -1)
          break;              //  Interrupted

      if (items [0].revents & ZMQ_POLLIN) {
          //  Get message
          //  - 3-part envelope + content -> request
          //  - 1-part HEARTBEAT -> heartbeat
          zmsg_t *msg = zmsg_recv (worker);
          if (!msg)
              break;          //  Interrupted
```

```
        //  .split simulating problems
        //  To test the robustness of the queue implementation we
        //  simulate various typical problems, such as the worker
        //  crashing or running very slowly. We do this after a few
        //  cycles so that the architecture can get up and running
        //  first:
        if (zmsg_size (msg) == 3) {
            cycles++;
            if (cycles > 3 && randof (5) == 0) {
                printf ("[W%Id] I: simulating a crash\n", (intptr_t)args);
                zmsg_destroy (&msg);
                break;
            }
            else
            if (cycles > 3 && randof (5) == 0) {
                printf ("[W%Id] I: simulating CPU overload\n", (intptr_t)args);
                s_sleep (3000);
                if (zctx_interrupted)
                    break;
            }
            printf ("[W%Id] I: normal reply\n", (intptr_t)args);
            zmsg_send (&msg, worker);
            liveness = HEARTBEAT_LIVENESS;
            s_sleep (1000);              //  Do some heavy work
            if (zctx_interrupted)
                break;
        }
        else
        //  .split handle heartbeats
```

```
            //  When we get a heartbeat message from the queue, it means the
            //  queue was (recently) alive, so we must reset our liveness
            //  indicator:
            if (zmsg_size (msg) == 1) {
                zframe_t *frame = zmsg_first (msg);
                if (memcmp (zframe_data (frame), PPP_HEARTBEAT, 1) == 0)
                    liveness = HEARTBEAT_LIVENESS;
                else {
                    printf ("[W%Id] E: invalid message\n", (intptr_t)args);
                    zmsg_dump (msg);
                }
                zmsg_destroy (&msg);
            }
            else {
                printf ("[W%Id] E: invalid message\n", (intptr_t)args);
                zmsg_dump (msg);
            }
            interval = INTERVAL_INIT;
        }
        else
        //  .split detecting a dead queue
        //  If the queue hasn't sent us heartbeats in a while, destroy the
        //  socket and reconnect. This is the simplest most brutal way of
        //  discarding any messages we might have sent in the meantime:
        if (--liveness == 0) {
            printf ("[W%Id] W: heartbeat failure, can't reach queue\n", (intptr_t)args);
            printf ("[W%Id] W: reconnecting in %zd msec...\n", (intptr_t)args, interval);
            zclock_sleep (interval);
```

```
            if (interval < INTERVAL_MAX)
                interval *= 2;
            zsocket_destroy (ctx, worker);
            worker = s_worker_socket (ctx);
            liveness = HEARTBEAT_LIVENESS;
        }
        //  Send heartbeat to queue if it's time
        if (zclock_time () > heartbeat_at) {
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
            printf ("[W%Id] I: worker heartbeat\n", (intptr_t)args);
            zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
            zframe_send (&frame, worker, 0);
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split worker class structure
//  Here we define the worker class; a structure and a set of functions that
//  act as constructor, destructor, and methods on worker objects:
typedef struct {
    zframe_t *identity;         //  Identity of worker
    char *id_string;            //  Printable identity
    int64_t expiry;             //  Expires at this time
} worker_t;

//  Construct new worker
static worker_t *
```

```
s_worker_new (zframe_t *identity)
{
    worker_t *self = (worker_t *) zmalloc (sizeof (worker_t));
    self->identity = identity;
    self->id_string = zframe_strhex (identity);
    self->expiry = zclock_time ()
                 + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS;
    return self;
}

//  Destroy specified worker object, including identity frame.
static void
s_worker_destroy (worker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        worker_t *self = *self_p;
        zframe_destroy (&self->identity);
        free (self->id_string);
        free (self);
        *self_p = NULL;
    }
}

//  .split worker ready method
//  The ready method puts a worker to the end of the ready list:

static void
s_worker_ready (worker_t *self, zlist_t *workers)
```

```
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (streq (self->id_string, worker->id_string)) {
            zlist_remove (workers, worker);
            s_worker_destroy (&worker);
            break;
        }
        worker = (worker_t *) zlist_next (workers);
    }
    zlist_append (workers, self);
}

//  .split get next available worker
//  The next method returns the next available worker identity:

static zframe_t *
s_workers_next (zlist_t *workers)
{
    worker_t *worker = zlist_pop (workers);
    assert (worker);
    zframe_t *frame = worker->identity;
    worker->identity = NULL;
    s_worker_destroy (&worker);
    return frame;
}

//  .split purge expired workers
//  The purge method looks for and kills expired workers. We hold workers
```

```
//  from oldest to most recent, so we stop at the first alive worker:

static void
s_workers_purge (zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              //  Worker is alive, we're done here

        zlist_remove (workers, worker);
        s_worker_destroy (&worker);
        worker = (worker_t *) zlist_first (workers);
    }
}

//  .split main task
//  The main task is a load-balancer with heartbeating on workers so we
//  can detect crashed or blocked worker tasks:
int main (void)
{
  zctx_t *ctx = zctx_new ();
  void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
  void *backend = zsocket_new (ctx, ZMQ_ROUTER);
  zsocket_bind (frontend, "tcp://*:5555");    //  For clients
  zsocket_bind (backend,  "tcp://*:5556");    //  For workers

  //  List of available workers
  zlist_t *workers = zlist_new ();
```

```
//  Send out heartbeats at regular intervals
uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

int worker_nbr;
for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++){
  zthread_new (worker_task, (void *)(intptr_t)worker_nbr);
  s_sleep(1000);
}

int client_nbr;
for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
  zthread_new (client_task, (void *)(intptr_t)client_nbr);

while (true) {
  zmq_pollitem_t items [] = {
    { backend,  0, ZMQ_POLLIN, 0 },
    { frontend, 0, ZMQ_POLLIN, 0 }
  };
  //  Poll frontend only if we have available workers
  int rc = zmq_poll (items, zlist_size (workers)? 2: 1,
    HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
  if (rc == -1)
    break;              //  Interrupted

  //  Handle worker activity on backend
  if (items [0].revents & ZMQ_POLLIN) {
    //  Use worker identity for load-balancing
    zmsg_t *msg = zmsg_recv (backend);
```

```
    if (!msg)
      break;          //  Interrupted

    //  Any sign of life from worker means it's ready
    zframe_t *identity = zmsg_unwrap (msg);
    worker_t *worker = s_worker_new (identity);
    s_worker_ready (worker, workers);

    //  Validate control message, or return reply to client
    if (zmsg_size (msg) == 1) {
      zframe_t *frame = zmsg_first (msg);
      if (memcmp (zframe_data (frame), PPP_READY, 1)
        &&  memcmp (zframe_data (frame), PPP_HEARTBEAT, 1)) {
        printf ("E: invalid message from worker");
        zmsg_dump (msg);
      }
      zmsg_destroy (&msg);
    }
    else
      zmsg_send (&msg, frontend);
  }
  if (items [1].revents & ZMQ_POLLIN) {
    //  Now get next client request, route to next worker
    zmsg_t *msg = zmsg_recv (frontend);
    if (!msg)
      break;          //  Interrupted
    zframe_t *identity = s_workers_next (workers);
    zmsg_prepend (msg, &identity);
    zmsg_send (&msg, backend);
```

```
    }
    //  .split handle heartbeating
    //  We handle heartbeating after any socket activity. First, we send
    //  heartbeats to any idle workers if it's time. Then, we purge any
    //  dead workers:
    if (zclock_time () >= heartbeat_at) {
      worker_t *worker = (worker_t *) zlist_first (workers);
      while (worker) {
        zframe_send (&worker->identity, backend,
          ZFRAME_REUSE + ZFRAME_MORE);
        zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
        zframe_send (&frame, backend, 0);
        worker = (worker_t *) zlist_next (workers);
      }
      heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
    }
    s_workers_purge (workers);
  }
  //  When we're done, clean up properly
  while (zlist_size (workers)) {
    worker_t *worker = (worker_t *) zlist_pop (workers);
    s_worker_destroy (&worker);
  }
  zlist_destroy (&workers);
  zctx_destroy (&ctx);
  return 0;
}
```

- [옮긴이] 빌드 및 테스트

~{.bash cl -EHsc pppattern.c libzmq.lib czmq.lib

./pppattern I: worker ready I: worker ready [W0] I: worker heartbeat [W1] I: worker heartbeat I: worker ready [W1] I: worker heartbeat [C1] I: Connecting to server… [C1] Entering while loop… [C1] Expecting reply…. … [C0] I: Connecting to server… [C0] Entering while loop… … [C2] I: Connecting to server… [C2] Entering while loop… … [W2] I: simulating a crash … [W0] I: simulating a crash … [W1] I: simulating a crash [C1] Polling Done.. [C1] E: Server seems to be offline, abandoning [C2] Polling Done.. [C2] W: no response from server, retrying… [C2] I: reconnecting to server… [C0] Polling Done.. [C0] W: no response from server, retrying… [C0] I: reconnecting to server… [C2] Expecting reply…. [C2] After polling [C0] Expecting reply…. [C0] After polling [C2] Polling Done.. [C2] E: Server seems to be offline, abandoning [C0] Polling Done.. [C0] W: no response from server, retrying… [C0] I: reconnecting to server… [C0] Expecting reply…. [C0] After polling [C0] Polling Done.. [C0] E: Server seems to be offline, abandoning

stop-process -name pppattern ~

0.44 심박

심박은 상대가 살았는지 죽었는지 알기 위한 문제를 해결합니다. 이것은 ØMQ에 한정된 문제가 아니며, TCP는 오랜 제한시간(30 분 정도)이 있어, 상대가 죽었는지, 연결이 끊어졌는지, 주말에 빨간 머리 아가씨와 프라하에 가서 보드카 한잔하는지 알 수가 없습니다.

심박을 제대로 얻는 것은 쉽지 않습니다. 편집증 해적 예제를 작성할 때, 심박이 제대로 작동하게 하는데 약 5시간이 소요되었습니다. 나머지 요청-응답 체인은 약 10분이 소요되었습니다. 특히 심박이 제대로 전송되지 않으면 상대들은 연결 해제되었다고 판단하여 “거짓 장애”를 생성하기 쉽습니다.

사람들이 ØMQ에서 심박에 사용하는 3가지 주요 해결책을 살펴보겠습니다.

0.44.1 무시하기

가장 일반적인 방법은 심박을 전혀 하지 않는 것입니다. 대부분의 ØMQ 응용프로그램은 아니지만 많은 응용프로그램이 이와 같이 합니다. 대다수의 경우 ØMQ는 상대들을 숨김으로

이를 권장합니다. 이 접근법은 어떤 문제들이 있을까요?

- 상대방들과 연결을 유지하는 응용프로그램에서 ROUTER 소켓을 사용하면 상대들이 연결을 해제 및 재연결을 지속적으로 수행하면 응용프로그램의 메모리(응용프로그램이 각 상대에 대해 보유하는 자원) 누수가 발생하고 점차 느려집니다.
- SUB 혹은 DEALER 소켓 기반 데이터를 수신자들로 사용할 때, 우리는 좋은 침묵(데이터 없음)과 나쁜 침묵(상대방이 죽었음)의 차이를 구분할 수 없습니다. 수신자가 상대방이 죽었다는 것을 알게 되면 백업 경로로 전환하여 데이터 수신을 지속할 수 있습니다.
- TCP 연결을 사용하는 경우 오랫동안 비활성화가 되면, 연결이 끊어지는 경우가 있습니다. 무언가를 보내면(기술적으로는 심박보다 "keep-alive") 네트워크가 살아 있게 됩니다.

0.44.2 단방향 심박

두 번째 옵션은 각 노드에서 1초마다 상대 노드들로 심박 메시지를 보내는 것입니다. 한 노드가 제한시간(보통 몇 초) 내에 다른 노드로부터 심박을 받지 못한다면 해당 상대를 죽은 것으로 처리됩니다. 이제 정말 좋은 것일까요? 이것은 잘 작동할 수도 있지만, 잘되지 않는 경우도 있습니다.

PUB-SUB 패턴이 단방향 심박을 사용할 수 있는 유일한 모델입니다. SUB 소켓은 PUB 소켓에 메시지를 전송할 수 없지만, PUB 소켓은 구독자들에게 "I'm alive"라는 메시지를 보낼 수 있습니다.

최적화를 고려하면, 실제 데이터가 없는 경우에만 심박를 보낼 수 있습니다. 또한 네트워크 활동이 문제인 경우(예 : 모바일 네트워크에서 통신으로 인해 배터리가 소모) 심박을 조금씩 느리게 보낼 수 있습니다. 수신자가 장애(통신의 급정지)를 감지할 수 있으면 괜찮습니다.

이러한 설계의 일반적인 문제는 다음과 같습니다.

- 많은 양의 데이터를 전송할 경우 해당 데이터로 인해 심박이 지연되므로 부정확할 수 있습니다. 심박이 지연되면 네트워크 정체로 인해 잘못된 제한시간과 연결 끊김이 발생할 수 있습니다. 따라서 발신자가 심박을 최적화 여부에 관계없이, 수신되는 데이터는 심박으로 취급합니다.

- PUB-SUB 패턴에서 사라진 수신자에 대한 메시지를 유실되는 동안, PUSH 및 DEALER 소켓은 메시지를 대기열에 넣습니다. 심박을 수신받는 상대가 죽었다가 다시 살아날 경우 대기열에 저장된 심박을 모두 받을 수 있습니다.
- 이 디자인은 심박 제한시간이 전체 네트워크에서 동일하다고 가정하지만 정확하진 않습니다. 일부 상대들은 장애를 신속하게 감지하기 위해 매우 공격적인 심박을 원할 것입니다. 그리고 일부는 네트워크를 수면 모드나 전력을 절약하기 위해서 매우 느슨한 심박을 원할 것입니다.

0.44.3 양방향 심박

세 번째 옵션은 핑-퐁(ping-pong)처럼 대화하는 것입니다. 한 상대가 다른 상대에게 `ping` 명령을 보내면 `pong` 명령으로 응답합니다. 2개의 명령 모두 추가적인 데이터는 없으며, 핑(pings)과 퐁(pongs)은 상호 연관되지 않습니다. "클라이언트"와 "서버"의 역할은 일부 네트워크에서 임의적이므로 보통 상대들끼리 핑(ping)을 보내고 응답으로 퐁(pong)을 기대할 수 있습니다. 그러나 제한시간은 동적 클라이언트에 알려진 네트워크 전송 방식에 의존적이기 때문에, 일반적으로 클라이언트에서 서버에 `ping`명령을 수행합니다.

이것은 모든 ROUTER 기반 브로커에서 동작하며, 두 번째 모델과 동일한 최적화를 사용하면 작업이 더욱 향상됩니다. 클라이언트로 수신되는 모든 데이터를 퐁(pong)으로 취급하고, 클라이언트에서 제한시간 내에 송신해야 할 데이터가 없는 경우 핑(ping)을 보냅니다.

0.44.4 편집증 해적 심박

편집증 해적 패턴에게는 두 번째 접근 방식(단방향 심박)을 선택했습니다. 가장 간단한 옵션이 아닐 수도 있습니다: 오늘 이것을 설계한다면 아마 핑-퐁(ping-pong) 방식(양뱡향 심박)을 선택할 것 같습니다. 그러나 원칙들은 비슷합니다. 심박 메시지는 비동기적으로 양방향(quueu-to-worker, worker-to-queue)으로 흐르며, 어느 쪽 상대(queue 혹은 worker)도 다른 쪽이 "죽었다"고 판단하고 통신를 중지할 수 있습니다.

작업자가 대기열 브로커로부터 심박(broker->worker)을 처리하는 방법은 다음과 같습니다.

- 대기열 브로커가 죽었다고 결정하기 전에, 허용 가능한 최대 심박 수인 liveness 변수를 정합니다. 3(HEARTBEAT_LIVENESS)에서 시작하며 요청 데이터나 심박이 수신되지 않을 때(1초 간격)마다 -1씩 감소합니다.
- zmq_poll() 루프에서 1초(timeout) 동안 대기합니다. 이것이 심박 시간 간격입니다.
- zmq_poll() 대기 시간 동안 수신 메시지(요청, 심박)가 있으면, liveness 변수를 최대 심박 수인 HEARTBEAT_LIVENESS로 재설정합니다.
- zmq_poll() 대기 시간 동안 메시지(요청, 심박)가 없으면, liveness 변수를 -1 감소(–liveness) 시킵니다.
- liveness 가 0에 도달하면 대기열 브로커이 죽은 것으로 판단합니다.
- 대기열 브로커가 죽으면, 소켓을 제거(zsocket_destroy())하고, 신규 소켓을 생성(zsocket_new())하고 재연결(zsocket_connect())합니다.
- 너무 많은 소켓을 열고 닫는 것을 방지하기 위해, 특정 시간간격 동안 대기 이후 재연결하며, 시간 간격은 32초(INTERVAL_MAX)에 도달 할 때까지 2배(interval *= 2) 늘립니다.

다음은 대기열 브로커가 작업자의 심박(worker->broker)을 처리하는 방식입니다.

- 다음 심박을 보낼 때를 계산합니다. 작업자들과 브로커와 통신하고 있기 때문에 단일 변수(heartbeat_at)입니다.
- zmq_poll() 루프에서 시간을 지날 때마다 브로커에서 모든 작업자들로 심박를 보냅니다.

아래는 작업자의 주요한 심박 처리 코드입니다.

```
#define HEARTBEAT_LIVENESS 3 // 3-5 is reasonable
#define HEARTBEAT_INTERVAL 1000 // msecs
#define INTERVAL_INIT 1000 // Initial reconnect
#define INTERVAL_MAX 32000 // After exponential backoff

...
// If liveness hits zero, queue is considered disconnected
size_t liveness = HEARTBEAT_LIVENESS;
```

```
size_t interval = INTERVAL_INIT;

// Send out heartbeats at regular intervals
uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

while (true) {
    zmq_pollitem_t items [] = { { worker, 0, ZMQ_POLLIN, 0 } };
    int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);

    if (items [0].revents & ZMQ_POLLIN) {
        ...
        // Receive any message from queue
        liveness = HEARTBEAT_LIVENESS;
        interval = INTERVAL_INIT;
        ...
    }
    else
    if (--liveness == 0) {
        zclock_sleep (interval);
        if (interval < INTERVAL_MAX)
            interval *= 2;
        zsocket_destroy (ctx, worker);
        ...
        liveness = HEARTBEAT_LIVENESS;
    }
    // Send heartbeat to queue if it's time
    if (zclock_time () > heartbeat_at) {
        heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        // Send heartbeat message to queue
```

```
    }
}
```

대기열 브로커의 코드도 동일하지만 각 작업자들에 대한 유효시간(heartbeat_at)을 관리합니다.

아래는 심박 구현을 위한 몇 가지 조언들입니다.

- zmq_poll() 혹은 반응자(zloop)를 응용프로그램의 주요 작업의 핵심으로 사용합니다.
- 상대들 간에 심박을 구축하여, 장애 상황을 시뮬레이션 및 테스트 한 다음 나머지 메시지 흐름을 구축합니다. 나중에 심박을 추가하는 것은 다소 까다롭습니다.
- 작업을 수행하면서 간단한 추적 기능을 사용하여, 명령어창에 출력하게 합니다. 상대들 간에 메시지 흐름을 추적하려면 zmsg가 제공하는 zmsg_dump() 기능을 사용하고 차이가 있는지 확인할 수 있도록 점차 메시지들에 번호를 지정합니다.
- 실제 응용프로그램에서 심박을 구성해서 상대와 통신해야 합니다. 일부 상대들은 최소 10 msec의 간격의 공격적인 심박을 원하며, 멀리 떨어진 상대들은 최대 30초의 심박을 원합니다.
- 각 상대들마다 다른 심박 간격이 있는 경우, 폴링 제한시간은 이들 중 가장 낮은(가장 짧은 시간) 것이어야 합니다. 무한한 제한시간을 사용하지 마십시오.
- 메시지와 동일한 소켓에서 심박을 수행하면, 심박을 통해 네트워크 연결을 유지(keep-alive)하는 역할도 합니다(일부 불친절한 방화벽은 통신이 수행되지 않은 접속을 끊어버리는 일이 있기 때문입니다).

0.45 계약과 통신규약

주의를 기울였다면 편집증 해적(PPP)은 단순한 해적 패턴(SPP)과 심박 때문에 상호 운영할 수 없다는 것을 알게 되었을 것입니다. 그러나 "상호 운용성(interoperability)"을 어떻게 정의할까요? 상호 운용성을 보장하려면, 일종의 계약(Contracts)이 필요하며 이러한 동의(agreement)를 통해 서로 다른 시간과 장소에 있는 서로 다른 팀들(사람들)이 동작이 보장되는 코드를 작성할 수 있게 합니다. 이를 "통신규약(Protocols)"이라고 합니다.

사양서(specifications) 없이 실험하는 것은 재미있지만, 실제 응용프로그램에서는 합리적인

기준이 아닙니다. 작업자를 다른 개발 언어로 작성하려면 어떻게 됩니까? 어떻게 동작하는지 보기 위해 소스 코드를 분석해야 하나요? 어떤 이유로 통신규약을 변경하려면 무엇을 해야 할까요? 원래는 간단한 통신규약도 점차 진화되어 복잡하게 됩니다.

계약의 부재은 일회용 응용프로그램의 전락하는 확실한 신호입니다. 통신규약에 대한 계약을 작성해 봅시다. 어떻게 하나요?

공개 ØMQ 계약을 위한 홈으로 만든 위키(rfc.zeromq.org)가 있습니다. 새로운 사양서를 만들기 위해서 위키에 등록하고 지침을 따르며 이것은 상당히 간단하지만 기술적인 문서를 작성하는 것이 모든 사람에게 식은 죽먹기는 아닙니다.

새로운 해적 패턴 통신규약을 작성하는 데 약 15분이 걸렸습니다. 큰 사양서는 아니지만 기본적인 행동을 이해하는 데는 충실합니다. 예를 들어 편집증 해적 패턴 통신 규약에 대하여 다른 사람이 작성한 코드가 위배될 경우 "당신이 작성한 대기열 브로커는 편집증 해적 패턴과 호환되지 않습니다. 수정하십시오!"라고 말할 수 있습니다.

편집증 해적 패턴을 실제 통신규약으로 바꾸려면 더 많은 작업이 필요합니다.

- READY 명령에 통신규약 버전 번호가 있어야 다른 버전의 편집증 해적 패턴을 구별할 수 있습니다.
- 현재 메시지들에서 READY 및 HEARTBEAT는 요청과 응답 메시지와 별 차이가 없습니다. 구별하기 위해 "메시지 유형" 부분에 포함하는 메시지 구조가 필요합니다.

0.46 서비스기반 신뢰성 있는 큐잉(MDP))

그림 50 - 집사(majordomo) 패턴

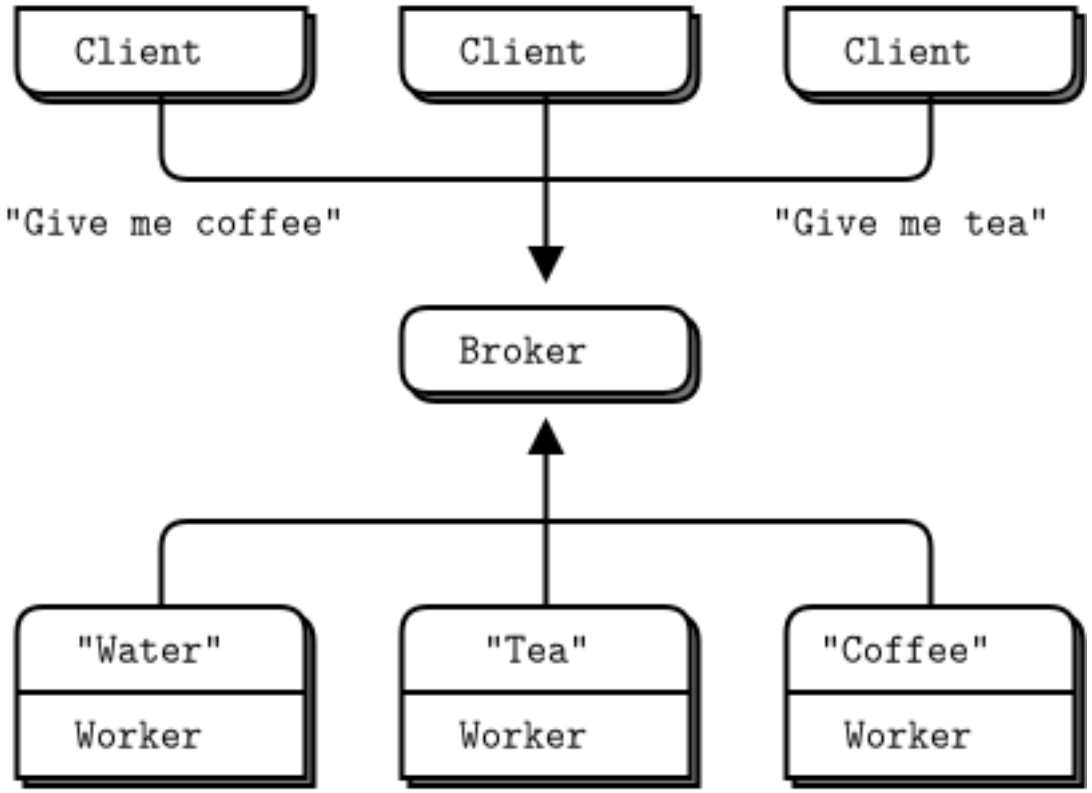


그림 52: majordomo

진보의 좋은 점은 변호사들과 위원회들이 관여하기 전에 빨리 진행하는 것입니다. 한 페이지짜리 MDP(Majordomo Protocol) 사양서는 PPP(Paranoid Pirate Protocol)를 보다 견고한 것으로 바꿉니다. 이것이 우리가 복잡한 아키텍처를 설계하는 방법입니다 : 계약서를 작성하는 것으로 시작하고 이를 구현할 소프트웨어를 작성합니다.

MDP는 한 가지 흥미로운 방식으로 PPP를 확장하고 개선합니다: 클라이언트가 보내는 요청에 "서비스 이름"을 추가하고 작업자에게 특정 서비스를 등록하도록 요청합니다. 서비스 이름을 추가하면 편집증 해적 브로커가 서비스 지향 브로커로 바뀝니다. MDP의 좋은 점은 동작하는 코드가 있으며, 이전 통신규약(PPP)도 있어, 명확한 문제를 해결한 정확한 일련의 개선 사례에서 나왔다는 것입니다. 이것으로 초안을 쉽게 작성할 수 있었습니다.

MDP를 구현하려면 클라이언트와 작업자를 위한 프레임워크를 작성해야 합니다. 모든 응용프로그램 개발자에게 사양서를 읽고 코드로 구현하도록 요청하는 것은 어려운 일이기

때문에 쉽게 사용할 수 있는 API를 제공하는 것이 좋습니다.

첫 번째 계약(MDP 자체)은 분산 아키텍처의 각 부분들이 상호 통신 방식을 정의하고, 두 번째 계약은 사용자 응용프로그램과 MDP 기술 프레임워크와 통신하는 방식을 정의합니다.

MDP에는 클라이언트 측과 작업자 측의 두 종류가 있습니다. 클라이언트 및 작업자 응용 프로그램을 모두 작성할 것이므로 2개의 API가 필요합니다. 다음은 간단한 객체 지향 접근 방식을 사용하여 설계한 클라이언트 API입니다.

```
mdcli_t *mdcli_new     (char *broker);
void     mdcli_destroy (mdcli_t **self_p);
zmsg_t  *mdcli_send    (mdcli_t *self, char *service, zmsg_t **request_p);
```

이게 다입니다. 브로커에 대한 세션을 열고, 요청 메시지를 보내고, 응답 메시지를 받고, 결국 연결을 닫습니다. 다음은 작업자 API에 대한 설계입니다.

```
mdwrk_t *mdwrk_new     (char *broker,char *service);
void     mdwrk_destroy (mdwrk_t **self_p);
zmsg_t  *mdwrk_recv    (mdwrk_t *self, zmsg_t *reply);
```

클라이언트와 다소 대칭적이지만 작업자 통신 방식은 약간 다릅니다. 작업자가 처음으로 `recv()`를 수행하면 null 응답을 전달하고 요청을 받으면 요청을 응답(echo)으로 새 요청을 받습니다.

클라이언트 및 작업자 API는 이미 개발 한 편집증 해적 코드를 기반으로 구성이 매우 단순하였습니다. 다음은 클라이언트 API입니다 :

mdcliapi.c: MDP 클라이언트 API

```
//  mdcliapi class - Majordomo Protocol Client API
//  Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

#include "mdcliapi.h"

//  Structure of our class
//  We access these properties only via class methods
```

```
struct _mdcli_t {
    zctx_t *ctx;                //  Our context
    char *broker;
    void *client;               //  Socket to broker
    int verbose;                //  Print activity to stdout
    int timeout;                //  Request timeout
    int retries;                //  Request retries
};

//  Connect or reconnect to broker

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_REQ);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s...", self->broker);
}

//  .split constructor and destructor
//  Here we have the constructor and destructor for our class:

//  Constructor

mdcli_t *
mdcli_new (char *broker, int verbose)
```

```
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  msecs
    self->retries = 3;              //  Before we abandon

    s_mdcli_connect_to_broker (self);
    return self;
}

//  Destructor

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  .split configure retry behavior
//  These are the class methods. We can set the request timeout and number
//  of retry attempts before sending requests:

//  Set request timeout

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}

//  Set request retries

void
mdcli_set_retries (mdcli_t *self, int retries)
{
    assert (self);
    self->retries = retries;
}

//  .split send request and wait for reply
//  Here is the {{send}} method. It sends a request to the broker and gets
//  a reply even if it has to retry several times. It takes ownership of
//  the request message, and destroys it when sent. It returns the reply
//  message, or NULL if there was no reply after multiple attempts:

zmsg_t *
```

```
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  Prefix request with protocol frames
    //  Frame 1: "MDPCxy" (six bytes, MDP/Client x.y)
    //  Frame 2: Service name (printable string)
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    if (self->verbose) {
        zclock_log ("I: send request to '%s' service:", service);
        zmsg_dump (request);
    }
    int retries_left = self->retries;
    while (retries_left && !zctx_interrupted) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->client);

        zmq_pollitem_t items [] = {
            { self->client, 0, ZMQ_POLLIN, 0 }
        };
        //  .split body of send
        //  On any blocking call, {{libzmq}} will return -1 if there was
        //  an error; we could in theory check for different error codes,
        //  but in practice it's OK to assume it was {{EINTR}} (Ctrl-C):

        int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
```

```
        if (rc == -1)
            break;          //  Interrupted

        //  If we got a reply, process it
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->client);
            if (self->verbose) {
                zclock_log ("I: received reply:");
                zmsg_dump (msg);
            }
            //  We would handle malformed replies better in real code
            assert (zmsg_size (msg) >= 3);

            zframe_t *header = zmsg_pop (msg);
            assert (zframe_streq (header, MDPC_CLIENT));
            zframe_destroy (&header);

            zframe_t *reply_service = zmsg_pop (msg);
            assert (zframe_streq (reply_service, service));
            zframe_destroy (&reply_service);

            zmsg_destroy (&request);
            return msg;     //  Success
        }
        else
        if (--retries_left) {
            if (self->verbose)
                zclock_log ("W: no reply, reconnecting...");
            s_mdcli_connect_to_broker (self);
```

```
        }
        else {
            if (self->verbose)
                zclock_log ("W: permanent error, abandoning");
            break;          //  Give up
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing client...\n");
    zmsg_destroy (&request);
    return NULL;
}
```

클라이언트 API가 어떻게 동작하는지, 100,000번 요청-응답 주기들을 수행하는 예제 테스트 프로그램으로 살펴보겠습니다.

mdclient.c: MDP 클라이언트

```
//  Majordomo Protocol client example
//  Uses the mdcli API to hide all MDP aspects

//  Lets us build this source without creating a library
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
```

```
        zmsg_pushstr (request, "Hello world");
        zmsg_t *reply = mdcli_send (session, "echo", &request);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  Interrupt or failure
    }
    printf ("%d requests/replies processed\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트 테스트 수행 시 -v 옵션을 주게 되면 처리 현황에 대한 정보를 받을 수 있습니다. 제한시간 2.5초 동안 2회 응답이 없어 프로그램은 종료합니다.

```
 cl -EHsc mdclient.c libzmq.lib czmq.lib

 ./mdclient -v
20-08-19 09:50:10 I: connecting to broker at tcp://localhost:5555...
20-08-19 09:50:10 I: send request to 'echo' service:
D: 20-08-19 09:50:10 [006] MDPC01
D: 20-08-19 09:50:10 [004] echo
D: 20-08-19 09:50:10 [011] Hello world
20-08-19 09:50:12 W: no reply, reconnecting...
20-08-19 09:50:12 I: connecting to broker at tcp://localhost:5555...
20-08-19 09:50:15 W: no reply, reconnecting...
20-08-19 09:50:15 I: connecting to broker at tcp://localhost:5555...
20-08-19 09:50:17 W: permanent error, abandoning
0 requests/replies processed
```

다음은 작업자 API입니다.

mdwrkapi.c: MDP 작업자 API

```
//  mdwrkapi class - Majordomo Protocol Worker API
//  Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

#include "mdwrkapi.h"

//  Reliability parameters
#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable

//  .split worker class structure
//  This is the structure of a worker API instance. We use a pseudo-OO
//  approach in a lot of the C examples, as well as the CZMQ binding:

//  Structure of our class
//  We access these properties only via class methods

struct _mdwrk_t {
    zctx_t *ctx;                //  Our context
    char *broker;
    char *service;
    void *worker;               //  Socket to broker
    int verbose;                //  Print activity to stdout

    //  Heartbeat management
    uint64_t heartbeat_at;      //  When to send HEARTBEAT
    size_t liveness;            //  How many attempts left
    int heartbeat;              //  Heartbeat delay, msecs
    int reconnect;              //  Reconnect delay, msecs
```

```
    int expect_reply;           //  Zero only at start
    zframe_t *reply_to;         //  Return identity, if any
};

//  .split utility functions
//  We have two utility functions; to send a message to the broker and
//  to (re)connect to the broker:

//  Send message to broker
//  If no msg is provided, creates one internally

static void
s_mdwrk_send_to_broker (mdwrk_t *self, char *command, char *option,
                        zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  Stack protocol envelope to start of message
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);
    zmsg_pushstr (msg, "");

    if (self->verbose) {
        zclock_log ("I: sending %s to broker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
```

```
    zmsg_send (&msg, self->worker);
}

//  Connect or reconnect to broker

void s_mdwrk_connect_to_broker (mdwrk_t *self)
{
    if (self->worker)
        zsocket_destroy (self->ctx, self->worker);
    self->worker = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->worker, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s...", self->broker);

    //  Register service with broker
    s_mdwrk_send_to_broker (self, MDPW_READY, self->service, NULL);

    //  If liveness hits zero, queue is considered disconnected
    self->liveness = HEARTBEAT_LIVENESS;
    self->heartbeat_at = zclock_time () + self->heartbeat;
}

//  .split constructor and destructor
//  Here we have the constructor and destructor for our mdwrk class:

//  Constructor

mdwrk_t *
mdwrk_new (char *broker,char *service, int verbose)
```

```
{
    assert (broker);
    assert (service);

    mdwrk_t *self = (mdwrk_t *) zmalloc (sizeof (mdwrk_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->service = strdup (service);
    self->verbose = verbose;
    self->heartbeat = 2500;     //  msecs
    self->reconnect = 2500;     //  msecs

    s_mdwrk_connect_to_broker (self);
    return self;
}

//  Destructor

void
mdwrk_destroy (mdwrk_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdwrk_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self->service);
        free (self);
        *self_p = NULL;
```

```
    }
}

//  .split configure worker
//  We provide two methods to configure the worker API. You can set the
//  heartbeat interval and retries to match the expected network performance.

//  Set heartbeat delay

void
mdwrk_set_heartbeat (mdwrk_t *self, int heartbeat)
{
    self->heartbeat = heartbeat;
}

//  Set reconnect delay

void
mdwrk_set_reconnect (mdwrk_t *self, int reconnect)
{
    self->reconnect = reconnect;
}

//  .split recv method
//  This is the {{recv}} method; it's a little misnamed because it first sends
//  any reply and then waits for a new request. If you have a better name
//  for this, let me know.

//  Send reply, if any, to broker and wait for next request.
```

```
zmsg_t *
mdwrk_recv (mdwrk_t *self, zmsg_t **reply_p)
{
    //  Format and send the reply if we were provided one
    assert (reply_p);
    zmsg_t *reply = *reply_p;
    assert (reply || !self->expect_reply);
    if (reply) {
        assert (self->reply_to);
        zmsg_wrap (reply, self->reply_to);
        s_mdwrk_send_to_broker (self, MDPW_REPLY, NULL, reply);
        zmsg_destroy (reply_p);
    }
    self->expect_reply = 1;

    while (true) {
        zmq_pollitem_t items [] = {
            { self->worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, self->heartbeat * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->worker);
            if (!msg)
                break;          //  Interrupted
            if (self->verbose) {
                zclock_log ("I: received message from broker:");
```

```
            zmsg_dump (msg);
        }
        self->liveness = HEARTBEAT_LIVENESS;

        //  Don't try to handle errors, just assert noisily
        assert (zmsg_size (msg) >= 3);

        zframe_t *empty = zmsg_pop (msg);
        assert (zframe_streq (empty, ""));
        zframe_destroy (&empty);

        zframe_t *header = zmsg_pop (msg);
        assert (zframe_streq (header, MDPW_WORKER));
        zframe_destroy (&header);

        zframe_t *command = zmsg_pop (msg);
        if (zframe_streq (command, MDPW_REQUEST)) {
            //  We should pop and save as many addresses as there are
            //  up to a null part, but for now, just save one...
            self->reply_to = zmsg_unwrap (msg);
            zframe_destroy (&command);
            //  .split process message
            //  Here is where we actually have a message to process; we
            //  return it to the caller application:

            return msg;     //  We have a request to process
        }
        else
        if (zframe_streq (command, MDPW_HEARTBEAT))
```

```
                ;               //  Do nothing for heartbeats
            else
            if (zframe_streq (command, MDPW_DISCONNECT))
                s_mdwrk_connect_to_broker (self);
            else {
                zclock_log ("E: invalid input message");
                zmsg_dump (msg);
            }
            zframe_destroy (&command);
            zmsg_destroy (&msg);
        }
        else
        if (--self->liveness == 0) {
            if (self->verbose)
                zclock_log ("W: disconnected from broker - retrying...");
            zclock_sleep (self->reconnect);
            s_mdwrk_connect_to_broker (self);
        }
        //  Send HEARTBEAT if it's time
        if (zclock_time () > self->heartbeat_at) {
            s_mdwrk_send_to_broker (self, MDPW_HEARTBEAT, NULL, NULL);
            self->heartbeat_at = zclock_time () + self->heartbeat;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing worker...\n");
    return NULL;
}
```

작업자 API가 어떻게 동작하는지, 에코(echo) 서비스로 구현된 예제 테스트 프로그램으로

살펴보겠습니다.

mdworker.c: MDP 작업자

```
//  Majordomo Protocol worker example
//  Uses the mdwrk API to hide all MDP aspects

//  Lets us build this source without creating a library
#include "mdwrkapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdwrk_t *session = mdwrk_new (
        "tcp://localhost:5555", "echo", verbose);

    zmsg_t *reply = NULL;
    while (true) {
        zmsg_t *request = mdwrk_recv (session, &reply);
        if (request == NULL)
            break;              //  Worker was interrupted
        reply = request;        //  Echo is complex... :-)
    }
    mdwrk_destroy (&session);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트
- 작업자는 응답을 대기하며 제한시간에 응답이 없으면 활성(liveness)을 -1 감소하고 심박을 보내고, 활성(liveness)이 0 되면 재접속하고 계속 대기합니다(무한 반복).

```
 cl -EHsc mdworker.c libzmq.lib czmq.lib
 ./mdworker -v
20-08-19 10:37:13 I: connecting to broker at tcp://localhost:5555...
20-08-19 10:37:13 I: sending READY to broker
D: 20-08-19 10:37:13 [000]
D: 20-08-19 10:37:13 [006] MDPW01
D: 20-08-19 10:37:13 [001] 01
D: 20-08-19 10:37:13 [004] echo
20-08-19 10:37:18 I: sending HEARTBEAT to broker
D: 20-08-19 10:37:18 [000]
D: 20-08-19 10:37:18 [006] MDPW01
D: 20-08-19 10:37:18 [001] 04
20-08-19 10:37:21 W: disconnected from broker - retrying...
20-08-19 10:37:23 I: connecting to broker at tcp://localhost:5555...
20-08-19 10:37:23 I: sending READY to broker
D: 20-08-19 10:37:23 [000]
D: 20-08-19 10:37:23 [006] MDPW01
D: 20-08-19 10:37:23 [001] 01
D: 20-08-19 10:37:23 [004] echo
20-08-19 10:37:26 I: sending HEARTBEAT to broker
D: 20-08-19 10:37:26 [000]
D: 20-08-19 10:37:26 [006] MDPW01
D: 20-08-19 10:37:26 [001] 04
20-08-19 10:37:31 W: disconnected from broker - retrying...
20-08-19 10:37:33 I: connecting to broker at tcp://localhost:5555...
...
```

작업자 API에 대하여 몇 가지 주의할 점은 다음과 같습니다.

- API는 단일 스레드로 작업자가 백그라운드에서 심박을 보내지 않음을 의미합니다.

다행스럽게도 이것이 바로 우리가 원하는 것입니다: 작업자 응용프로그램이 중단되면 심박은 중지되고 브로커는 작업자에게 요청을 보내는 것을 중지합니다.

- 작업자 API는 지수 백-오프를 수행하지 않습니다. 복잡하게 할 필요가 없습니다.
- API는 오류보고를 수행하지 않습니다. 예상과 다르면 어설션(혹은 개발 언어에 따라 예외 처리)이 발생합니다. 임시 참조 구현에서는 이상적이며, 모든 통신규약 오류는 즉시 보여주어야 합니다. 실제 응용프로그램의 경우 API는 잘못된 메시지에 대하여 오류를 발생하고 종료하는 것이 아니라 예외 상황으로 처리하고 다음 메시지 처리가 필요합니다.
- [옮긴이] 지수 백 오프(exponential back-off)는 편집증 해적 패턴에서 작업자로 응답이 오지 않을 경우 활성(liveness)을 -1 감소시키면서 대기(sleep)하는 주기(interval)을 2^n으로 최대 32초까지 수행하는 기능입니다.

ØMQ는 상대가 사라졌다가 돌아오면 자동으로 소켓을 재연결하지만, 작업자 API가 소켓을 수동으로 닫고 새 소켓을 여는 이유에 대하여 궁금할 것입니다. 이해하기 위해 단순한 해적(SPP)과 편집증 해적(PPP) 작업자들을 되돌아보면, ØMQ는 브로커가 죽고 다시 돌아오면 자동으로 작업자를 다시 연결하지만, 브로커에 작업자들을 재등록하기에는 충분하지 않습니다.

- [옮긴이] 작업자에서 “READY” 신호를 통하여 브로커에 작업자를 등록하는 과정이 존재합니다.

적어도 두 가지 해결방안이 있습니다. - [작업자 측면] 여기서 사용하는 가장 간단한 방법은 작업자가 심박을 통하여 연결을 모니터링하다가 브로커가 죽었다고 판단되면 소켓을 닫고 새 소켓으로 시작하는 것입니다. - [브로커 측면] 다른 방법은 브로커에 대한 것으로, 브로커가 알 수 없는 작업자로부터 심박을 받으면 작업자에게 재등록을 요청하는 것입니다. 통신규약상에 지원이 필요합니다.

이제 MDP 브로커를 설계하겠습니다. 핵심 구조는 일련의 대기열로 하나의 서비스에 하나의 대기열입니다. 작업자가 나타날 때 이러한 대기열들을 생성합니다(작업자들이 사라지면 제거할 수 있지만 복잡해지기 때문에 지금은 잊어버립니다). 또한 하나의 서비스에 작업자 대기열을 유지합니다.

아래에 브로커 코드가 있습니다.

mdbroker.c: MDP 브로커

```
//  Majordomo Protocol broker
//  A minimal C implementation of the Majordomo Protocol as defined in
//  http://rfc.zeromq.org/spec:7 and http://rfc.zeromq.org/spec:8.

#include "czmq.h"
#include "mdp.h"

//  We'd normally pull these from config data

#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  2500    //  msecs
#define HEARTBEAT_EXPIRY    HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS

//  .split broker class structure
//  The broker class defines a single broker instance:

typedef struct {
    zctx_t *ctx;                //  Our context
    void *socket;               //  Socket for clients & workers
    int verbose;                //  Print activity to stdout
    char *endpoint;             //  Broker binds to this endpoint
    zhash_t *services;          //  Hash of known services
    zhash_t *workers;           //  Hash of known workers
    zlist_t *waiting;           //  List of waiting workers
    uint64_t heartbeat_at;      //  When to send HEARTBEAT
} broker_t;
```

```
static broker_t *
    s_broker_new (int verbose);
static void
    s_broker_destroy (broker_t **self_p);
static void
    s_broker_bind (broker_t *self, char *endpoint);
static void
    s_broker_worker_msg (broker_t *self, zframe_t *sender, zmsg_t *msg);
static void
    s_broker_client_msg (broker_t *self, zframe_t *sender, zmsg_t *msg);
static void
    s_broker_purge (broker_t *self);

//  .split service class structure
//  The service class defines a single service instance:

typedef struct {
    broker_t *broker;           //  Broker instance
    char *name;                 //  Service name
    zlist_t *requests;          //  List of client requests
    zlist_t *waiting;           //  List of waiting workers
    size_t workers;             //  How many workers we have
} service_t;

static service_t *
    s_service_require (broker_t *self, zframe_t *service_frame);
static void
    s_service_destroy (void *argument);
static void
```

```
    s_service_dispatch (service_t *service, zmsg_t *msg);

//  .split worker class structure
//  The worker class defines a single worker, idle or active:

typedef struct {
    broker_t *broker;           //  Broker instance
    char *id_string;            //  Identity of worker as string
    zframe_t *identity;         //  Identity frame for routing
    service_t *service;         //  Owning service, if known
    int64_t expiry;             //  When worker expires, if no heartbeat
} worker_t;

static worker_t *
    s_worker_require (broker_t *self, zframe_t *identity);
static void
    s_worker_delete (worker_t *self, int disconnect);
static void
    s_worker_destroy (void *argument);
static void
    s_worker_send (worker_t *self, char *command, char *option,
                   zmsg_t *msg);
static void
    s_worker_waiting (worker_t *self);

//  .split broker constructor and destructor
//  Here are the constructor and destructor for the broker:

static broker_t *
```

```
s_broker_new (int verbose)
{
    broker_t *self = (broker_t *) zmalloc (sizeof (broker_t));

    //  Initialize broker state
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->verbose = verbose;
    self->services = zhash_new ();
    self->workers = zhash_new ();
    self->waiting = zlist_new ();
    self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
    return self;
}

static void
s_broker_destroy (broker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        broker_t *self = *self_p;
        zctx_destroy (&self->ctx);
        zhash_destroy (&self->services);
        zhash_destroy (&self->workers);
        zlist_destroy (&self->waiting);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  .split broker bind method
//  This method binds the broker instance to an endpoint. We can call
//  this multiple times. Note that MDP uses a single socket for both clients
//  and workers:

void
s_broker_bind (broker_t *self, char *endpoint)
{
    zsocket_bind (self->socket, endpoint);
    zclock_log ("I: MDP broker/0.2.0 is active at %s", endpoint);
}

//  .split broker worker_msg method
//  This method processes one READY, REPLY, HEARTBEAT, or
//  DISCONNECT message sent to the broker by a worker:

static void
s_broker_worker_msg (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 1);     //  At least, command

    zframe_t *command = zmsg_pop (msg);
    char *id_string = zframe_strhex (sender);
    int worker_ready = (zhash_lookup (self->workers, id_string) != NULL);
    free (id_string);
    worker_t *worker = s_worker_require (self, sender);

    if (zframe_streq (command, MDPW_READY)) {
```

```
        if (worker_ready)               //  Not first command in session
            s_worker_delete (worker, 1);
        else
        if (zframe_size (sender) >= 4  //  Reserved service name
        &&  memcmp (zframe_data (sender), "mmi.", 4) == 0)
            s_worker_delete (worker, 1);
        else {
            //  Attach worker to service and mark as idle
            zframe_t *service_frame = zmsg_pop (msg);
            worker->service = s_service_require (self, service_frame);
            worker->service->workers++;
            s_worker_waiting (worker);
            zframe_destroy (&service_frame);
        }
    }
    else
    if (zframe_streq (command, MDPW_REPLY)) {
        if (worker_ready) {
            //  Remove and save client return envelope and insert the
            //  protocol header and service name, then rewrap envelope.
            zframe_t *client = zmsg_unwrap (msg);
            zmsg_pushstr (msg, worker->service->name);
            zmsg_pushstr (msg, MDPC_CLIENT);
            zmsg_wrap (msg, client);
            zmsg_send (&msg, self->socket);
            s_worker_waiting (worker);
        }
        else
            s_worker_delete (worker, 1);
```

```
    }
    else
    if (zframe_streq (command, MDPW_HEARTBEAT)) {
        if (worker_ready)
            worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
        else
            s_worker_delete (worker, 1);
    }
    else
    if (zframe_streq (command, MDPW_DISCONNECT))
        s_worker_delete (worker, 0);
    else {
        zclock_log ("E: invalid input message");
        zmsg_dump (msg);
    }
    free (command);
    zmsg_destroy (&msg);
}

//  .split broker client_msg method
//  Process a request coming from a client. We implement MMI requests
//  directly here (at present, we implement only the mmi.service request):

static void
s_broker_client_msg (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 2);     //  Service name + body

    zframe_t *service_frame = zmsg_pop (msg);
```

```
    service_t *service = s_service_require (self, service_frame);

    //  Set reply return identity to client sender
    zmsg_wrap (msg, zframe_dup (sender));

    //  If we got a MMI service request, process that internally
    if (zframe_size (service_frame) >= 4
    &&  memcmp (zframe_data (service_frame), "mmi.", 4) == 0) {
        char *return_code;
        if (zframe_streq (service_frame, "mmi.service")) {
            char *name = zframe_strdup (zmsg_last (msg));
            service_t *service =
                (service_t *) zhash_lookup (self->services, name);
            return_code = service && service->workers? "200": "404";
            free (name);
        }
        else
            return_code = "501";

        zframe_reset (zmsg_last (msg), return_code, strlen (return_code));

        //  Remove & save client return envelope and insert the
        //  protocol header and service name, then rewrap envelope.
        zframe_t *client = zmsg_unwrap (msg);
        zmsg_prepend (msg, &service_frame);
        zmsg_pushstr (msg, MDPC_CLIENT);
        zmsg_wrap (msg, client);
        zmsg_send (&msg, self->socket);
    }
```

```
    else
        //  Else dispatch the message to the requested service
        s_service_dispatch (service, msg);
    zframe_destroy (&service_frame);
}

//  .split broker purge method
//  This method deletes any idle workers that haven't pinged us in a
//  while. We hold workers from oldest to most recent so we can stop
//  scanning whenever we find a live worker. This means we'll mainly stop
//  at the first worker, which is essential when we have large numbers of
//  workers (we call this method in our critical path):

static void
s_broker_purge (broker_t *self)
{
    worker_t *worker = (worker_t *) zlist_first (self->waiting);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;                  //  Worker is alive, we're done here
        if (self->verbose)
            zclock_log ("I: deleting expired worker: %s",
                        worker->id_string);

        s_worker_delete (worker, 0);
        worker = (worker_t *) zlist_first (self->waiting);
    }
}
```

```
//  .split service methods
//  Here is the implementation of the methods that work on a service:

//  Lazy constructor that locates a service by name or creates a new
//  service if there is no service already with that name.

static service_t *
s_service_require (broker_t *self, zframe_t *service_frame)
{
    assert (service_frame);
    char *name = zframe_strdup (service_frame);

    service_t *service =
        (service_t *) zhash_lookup (self->services, name);
    if (service == NULL) {
        service = (service_t *) zmalloc (sizeof (service_t));
        service->broker = self;
        service->name = name;
        service->requests = zlist_new ();
        service->waiting = zlist_new ();
        zhash_insert (self->services, name, service);
        zhash_freefn (self->services, name, s_service_destroy);
        if (self->verbose)
            zclock_log ("I: added service: %s", name);
    }
    else
        free (name);

    return service;
```

```
}

//  Service destructor is called automatically whenever the service is
//  removed from broker->services.

static void
s_service_destroy (void *argument)
{
    service_t *service = (service_t *) argument;
    while (zlist_size (service->requests)) {
        zmsg_t *msg = zlist_pop (service->requests);
        zmsg_destroy (&msg);
    }
    zlist_destroy (&service->requests);
    zlist_destroy (&service->waiting);
    free (service->name);
    free (service);
}

//  .split service dispatch method
//  This method sends requests to waiting workers:

static void
s_service_dispatch (service_t *self, zmsg_t *msg)
{
    assert (self);
    if (msg)                    //  Queue message if any
        zlist_append (self->requests, msg);
```

```
    s_broker_purge (self->broker);
    while (zlist_size (self->waiting) && zlist_size (self->requests)) {
        worker_t *worker = zlist_pop (self->waiting);
        zlist_remove (self->broker->waiting, worker);
        zmsg_t *msg = zlist_pop (self->requests);
        s_worker_send (worker, MDPW_REQUEST, NULL, msg);
        zmsg_destroy (&msg);
    }
}

//  .split worker methods
//  Here is the implementation of the methods that work on a worker:

//  Lazy constructor that locates a worker by identity, or creates a new
//  worker if there is no worker already with that identity.

static worker_t *
s_worker_require (broker_t *self, zframe_t *identity)
{
    assert (identity);

    //  self->workers is keyed off worker identity
    char *id_string = zframe_strhex (identity);
    worker_t *worker =
        (worker_t *) zhash_lookup (self->workers, id_string);

    if (worker == NULL) {
        worker = (worker_t *) zmalloc (sizeof (worker_t));
        worker->broker = self;
```

```
        worker->id_string = id_string;
        worker->identity = zframe_dup (identity);
        zhash_insert (self->workers, id_string, worker);
        zhash_freefn (self->workers, id_string, s_worker_destroy);
        if (self->verbose)
            zclock_log ("I: registering new worker: %s", id_string);
    }
    else
        free (id_string);
    return worker;
}

//  This method deletes the current worker.

static void
s_worker_delete (worker_t *self, int disconnect)
{
    assert (self);
    if (disconnect)
        s_worker_send (self, MDPW_DISCONNECT, NULL, NULL);

    if (self->service) {
        zlist_remove (self->service->waiting, self);
        self->service->workers--;
    }
    zlist_remove (self->broker->waiting, self);
    //  This implicitly calls s_worker_destroy
    zhash_delete (self->broker->workers, self->id_string);
}
```

```
//  Worker destructor is called automatically whenever the worker is
//  removed from broker->workers.

static void
s_worker_destroy (void *argument)
{
    worker_t *self = (worker_t *) argument;
    zframe_destroy (&self->identity);
    free (self->id_string);
    free (self);
}

//  .split worker send method
//  This method formats and sends a command to a worker. The caller may
//  also provide a command option, and a message payload:

static void
s_worker_send (worker_t *self, char *command, char *option, zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  Stack protocol envelope to start of message
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);

    //  Stack routing envelope to start of message
```

```
    zmsg_wrap (msg, zframe_dup (self->identity));

    if (self->broker->verbose) {
        zclock_log ("I: sending %s to worker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->broker->socket);
}

//  This worker is now waiting for work

static void
s_worker_waiting (worker_t *self)
{
    //  Queue to broker and service waiting lists
    assert (self->broker);
    zlist_append (self->broker->waiting, self);
    zlist_append (self->service->waiting, self);
    self->expiry = zclock_time () + HEARTBEAT_EXPIRY;
    s_service_dispatch (self->service, NULL);
}

//  .split main task
//  Finally, here is the main task. We create a new broker instance and
//  then process messages on the broker socket:

int main (int argc, char *argv [])
{
```

```
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    broker_t *self = s_broker_new (verbose);
    s_broker_bind (self, "tcp://*:5555");

    //  Get and process messages forever or until interrupted
    while (true) {
        zmq_pollitem_t items [] = {
            { self->socket,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        //  Process next input message, if any
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->socket);
            if (!msg)
                break;          //  Interrupted
            if (self->verbose) {
                zclock_log ("I: received message:");
                zmsg_dump (msg);
            }
            zframe_t *sender = zmsg_pop (msg);
            zframe_t *empty  = zmsg_pop (msg);
            zframe_t *header = zmsg_pop (msg);

            if (zframe_streq (header, MDPC_CLIENT))
                s_broker_client_msg (self, sender, msg);
            else
```

```
            if (zframe_streq (header, MDPW_WORKER))
                s_broker_worker_msg (self, sender, msg);
            else {
                zclock_log ("E: invalid message:");
                zmsg_dump (msg);
                zmsg_destroy (&msg);
            }
            zframe_destroy (&sender);
            zframe_destroy (&empty);
            zframe_destroy (&header);
        }
        //  Disconnect and delete any expired workers
        //  Send heartbeats to idle workers if needed
        if (zclock_time () > self->heartbeat_at) {
            s_broker_purge (self);
            worker_t *worker = (worker_t *) zlist_first (self->waiting);
            while (worker) {
                s_worker_send (worker, MDPW_HEARTBEAT, NULL, NULL);
                worker = (worker_t *) zlist_next (self->waiting);
            }
            self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, shutting down...\n");

    s_broker_destroy (&self);
    return 0;
}
```

이것은 우리가 본 것 중 가장 복잡한 예제입니다. 거의 500라인의 코드입니다. 이것을 작성하고 다소 견고하게 만드는 데 2일이나 걸렸습니다. 그러나 이것은 아직 완전한 서비스 지향 브로커로는 짧은 편입니다.

브로커 코드에서 주의할 몇 가지 사항은 다음과 같습니다.

- MDP는 단일 ROUTER 소켓에서 클라이언트들과 작업자들을 모두 처리할 수 있습니다. 이는 브로커를 배포하고 관리하는 데는 좋습니다: 대부분의 프록시가 필요로 하는 두 개가 아닌 하나의 ØMQ 단말에 있습니다.
- 브로커는 모든 MDP/v0.1 사양을 구현하였으며, 브로커가 잘못된 명령을 보내는 경우에 연결 해제, 심박 및 나머지들을 포함합니다.
- 멀티스레드로 실행하기 위해 확장될 수 있으며, 각각의 스레드는 하나의 소켓과 하나의 클라이언트 및 작업자 집합을 관리합니다. 이것은 대규모 아키텍처로 분할하기 위한 흥미로운 주제가 될 수 있습니다. C 코드는 작업을 쉽게 하기 위해 이미 브로커 클래스를 중심으로 구성되어 있습니다.
- 기본(primary)/장애조치(failover) 또는 라이브(live)/라이브(live) 브로커 신뢰성 모델은 쉽지만, 브로커는 기본적으로 서비스 존재를 제외하고는 상태가 없기 때문에, 클라이언트들과 작업자들이 처음 선택한 브로커가 죽었을 경우 다른 브로커를 선택하는 것은 클라이언트들과 작업자들의 책임입니다.
- 예제는 2.5초간격 심박을 사용하며, 주로 추적을 활성화할 때 출력량을 줄이기 위해서입니다. 실제 값은 대부분의 LAN 응용프로그램에서 더 작을 수 있으나 모든 재시도는 혹시 서비스가 재시작되는 것을 고려해 충분히 늦추어져야 합니다(최소 10초).

현재 MDP 구현과 통신규약은 개선하고 확장하였으며, 현재 자체 Github 프로젝트로 자리 잡았습니다. 적절하게 사용 가능한 MDP 스택을 원한다면 GitHub 프로젝트를 사용하십시오.

- [옮긴이] 빌드 및 테스트
- 브로커(mdbroker)와 작업자(mdworker)를 실행하고 나서 클라이언트(mdclient)를 수행합니다.
- 클라이언트(mdclient)에서 루프를 100,000번 수행하는 것을 2로 변경합니다.
- 작업자(mdworker)에 등록된 서비스 이름이 “echo”와 클라이언트에서 요청하는 서비스 이름인 “echo”가 일치해야만 정상 구동됩니다.

```
 ./mdbroker -v
20-08-19 15:13:48 I: MDP broker/0.2.0 is active at tcp://*:5555
20-08-19 15:13:49 I: received message:
D: 20-08-19 15:13:49 [005] 0080000029   --> WID
D: 20-08-19 15:13:49 [000]
D: 20-08-19 15:13:49 [006] MDPW01
D: 20-08-19 15:13:49 [001] 01           --> MDPW_READY
D: 20-08-19 15:13:49 [004] echo         --> service
20-08-19 15:13:49 I: registering new worker: 0080000029
20-08-19 15:13:49 I: added service: echo
20-08-19 15:13:51 I: sending HEARTBEAT to worker
D: 20-08-19 15:13:51 [005] 0080000029   -> WID
D: 20-08-19 15:13:51 [000]
D: 20-08-19 15:13:51 [006] MDPW01
D: 20-08-19 15:13:51 [001] 04           --> MDPW_HEARTBEAT
...
20-08-19 15:13:55 I: received message:
D: 20-08-19 15:13:55 [005] 008000002A    --> CID
D: 20-08-19 15:13:55 [000]
D: 20-08-19 15:13:55 [006] MDPC01
D: 20-08-19 15:13:55 [004] echo          --> service
D: 20-08-19 15:13:55 [011] Hello world
20-08-19 15:13:55 I: sending REQUEST to worker
D: 20-08-19 15:13:55 [005] 0080000029   --> WID
D: 20-08-19 15:13:55 [000]
D: 20-08-19 15:13:55 [006] MDPW01
D: 20-08-19 15:13:55 [001] 02           --> MDPW_REQUEST
D: 20-08-19 15:13:55 [005] 008000002A   --> CID
D: 20-08-19 15:13:55 [000]
```

```
D: 20-08-19 15:13:55 [011] Hello world
20-08-19 15:13:55 I: received message:
D: 20-08-19 15:13:55 [005] 0080000029   --> WID
D: 20-08-19 15:13:55 [000]
D: 20-08-19 15:13:55 [006] MDPW01
D: 20-08-19 15:13:55 [001] 03           --> MDPW_REPLY
D: 20-08-19 15:13:55 [005] 008000002A   --> CID
D: 20-08-19 15:13:55 [000]
D: 20-08-19 15:13:55 [011] Hello world
...

 ./mdworker -v
20-08-19 15:13:49 I: connecting to broker at tcp://localhost:5555...
20-08-19 15:13:49 I: sending READY to broker
D: 20-08-19 15:13:49 [000]
D: 20-08-19 15:13:49 [006] MDPW01
D: 20-08-19 15:13:49 [001] 01               --> MDPW_READY
D: 20-08-19 15:13:49 [004] echo             --> service
20-08-19 15:13:51 I: sending HEARTBEAT to broker
...
20-08-19 15:13:55 I: received message from broker:
D: 20-08-19 15:13:55 [000]
D: 20-08-19 15:13:55 [006] MDPW01
D: 20-08-19 15:13:55 [001] 02              --> MDPW_REQUEST
D: 20-08-19 15:13:55 [005] 008000002A      --> CLIENT ID
D: 20-08-19 15:13:55 [000]
D: 20-08-19 15:13:55 [011] Hello world
20-08-19 15:13:55 I: sending REPLY to broker
D: 20-08-19 15:13:55 [000]
```

```
D: 20-08-19 15:13:55 [006] MDPW01
D: 20-08-19 15:13:55 [001] 03              --> MDPW_REPLY
D: 20-08-19 15:13:55 [005] 008000002A      --> CLIENT ID
D: 20-08-19 15:13:55 [000]
D: 20-08-19 15:13:55 [011] Hello world
...


 ./mdclient -v
20-08-19 15:13:55 I: connecting to broker at tcp://localhost:5555...
20-08-19 15:13:55 I: send request to 'echo' service:
D: 20-08-19 15:13:55 [006] MDPC01
D: 20-08-19 15:13:55 [004] echo
D: 20-08-19 15:13:55 [011] Hello world
20-08-19 15:13:55 I: received reply:
D: 20-08-19 15:13:55 [006] MDPC01
D: 20-08-19 15:13:55 [004] echo
D: 20-08-19 15:13:55 [011] Hello world
20-08-19 15:13:55 I: send request to 'echo' service:
D: 20-08-19 15:13:55 [006] MDPC01
D: 20-08-19 15:13:55 [004] echo
D: 20-08-19 15:13:55 [011] Hello world
20-08-19 15:13:55 I: received reply:
D: 20-08-19 15:13:55 [006] MDPC01
D: 20-08-19 15:13:55 [004] echo
D: 20-08-19 15:13:55 [011] Hello world
```

0.47 비동기 MDP 패턴

이전의 MDP 구현은 간단하지만 멍청합니다. 클라이언트는 섹시한 API로 감싼 단순한 해적 패턴입니다. 명령어창에서 클라이언트, 브로커 및 작업자를 실행하면 약 14초 내에 100,000 개의 요청을 처리(-v 옵션 제거)할 수 있으며, 이는 CPU 리소스 있는 한도에서 메시지 프레임들을 주변으로 복사 가능하기 때문입니다. 그러나 진짜 문제는 우리가 네트워크 왕복(round-trips)입니다. ØMQ는 네이글 알고리즘을 비활성화하지만 왕복은 여전히 느립니다.

- [옮긴이] 네이글(Nagle) 알고리즘은 TCP/IP 기반의 네트워크에서 특정 작은 인터넷 패킷 전송을 억제하는 알고리즘으로 작은 패킷을 가능한 모아서 큰 패킷으로 모아서 한 번에 전송합니다. 네트워크 전송의 효율을 높여주지만 실시간으로 운용해야 하는 응용프로그램에서는 오히려 지연을 발생시키게 됩니다.

이론은 이론적으로는 훌륭하지만 실제 해보는 것이 좋습니다. 간단한 테스트 프로그램으로 실제 왕복 비용을 측정해 봅시다. 프로그램에서 대량의 메시지들을 보내고, * 첫째 각 메시지에 대한 응답을 하나씩 기다리고, * 두 번째는 일괄 처리로, 모든 응답을 한꺼번에 읽게 합니다.

두 접근 방식 모두 동일한 작업을 수행하지만 결과는 매우 다릅니다. 클라이언트, 중개인 및 작업자를 동작하는 예제를 작성하겠습니다.

tripping.c: 왕복 데모(Round-trip demonstrator)

```
//  Round-trip demonstrator
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. The client task signals to
//  main when it's ready.

#include "czmq.h"

static void
client_task (void *args, zctx_t *ctx, void *pipe)
{
```

```
    void *client = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (client, "tcp://localhost:5555");
    printf ("Setting up test...\n");
    zclock_sleep (100);

    int requests;
    int64_t start;

    printf ("Synchronous round-trip test...\n");
    start = zclock_time ();
    for (requests = 0; requests < 10000; requests++) {
        zstr_send (client, "hello");
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d calls/second\n",
        (1000 * 10000) / (int) (zclock_time () - start));

    printf ("Asynchronous round-trip test...\n");
    start = zclock_time ();
    for (requests = 0; requests < 100000; requests++)
        zstr_send (client, "hello");
    for (requests = 0; requests < 100000; requests++) {
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d calls/second\n",
        (1000 * 100000) / (int) (zclock_time () - start));
    zstr_send (pipe, "done");
```

```
}

//  .split worker task
//  Here is the worker task. All it does is receive a message, and
//  bounce it back the way it came:

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");

    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  .split broker task
//  Here is the broker task. It uses the {{zmq_proxy}} function to switch
//  messages between frontend and backend:

static void *
broker_task (void *args)
{
    //  Prepare our context and sockets
```

```
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (frontend, "tcp://*:5555");
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "tcp://*:5556");
    zmq_proxy (frontend, backend, NULL);
    zctx_destroy (&ctx);
    return NULL;
}

//  .split main task
//  Finally, here's the main task, which starts the client, worker, and
//  broker, and then runs until the client signals it to stop:

int main (void)
{
    //  Create threads
    zctx_t *ctx = zctx_new ();
    void *client = zthread_fork (ctx, client_task, NULL);
    zthread_new (worker_task, NULL);
    zthread_new (broker_task, NULL);

    //  Wait for signal on client pipe
    char *signal = zstr_recv (client);
    free (signal);

    zctx_destroy (&ctx);
    return 0;
}
```

자체 보유한 개발 서버에서는 다음과 같은 결과를 얻을 수 있었습니다.

```
Setting up test...
Synchronous round-trip test...
 9057 calls/second
Asynchronous round-trip test...
 173010 calls/second
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc tripping.c libzmq.lib czmq.lib
 ./tripping
Setting up test...
Synchronous round-trip test...
 2359 calls/second
Asynchronous round-trip test...
 261096 calls/second
```

클라이언트 스레드는 시작하기 전에 잠시 대기를 수행(`zclock_sleep(100)`)합니다. 이것은 라우터 소켓의 "기능들"중 하나는 아직 연결되지 않은 상대의 주소로 메시지를 보내면 메시지가 유실됩니다. 예제에서는 부하 분산 메커니즘을 사용하지 않아, 일정 시간 동안 대기하지 않으면, 작업자 스레드의 연결이 지연되어 메시지가 유실되면 테스트가 엉망이 됩니다.

테스트 결과를 보았듯이, 간단한 예제의 경우에서 동기식으로 진행되는 왕복은 비동기식보다 20배(~60배) 더 느립니다. "물 들어올 때 노 저어라" 말처럼, 비동기식을 MDP에 적용하여 더 빠르게 만들 수 있는지 보겠습니다.

먼저 클라이언트 API인 `mdcli_send()`을 수정하여 송신(`mdcli_send()`)과 수신(`mdcli_recv()`)으로 분리합니다.

```
mdcli_t *mdcli_new (char *broker);
void mdcli_destroy (mdcli_t **self_p);
int mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p);
```

```
zmsg_t *mdcli_recv (mdcli_t *self);
```

동기식 클라이언트 API를 비동기식으로 재구성하는 것은 말 그대로 몇 분밖에 걸리지 않는 작업입니다.

mdcliapi2.c: 비동기 MDP 클라이언트 API

```
//  mdcliapi2 class - Majordomo Protocol Client API
//  Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

#include "mdcliapi2.h"

//  Structure of our class
//  We access these properties only via class methods

struct _mdcli_t {
    zctx_t *ctx;                //  Our context
    char *broker;
    void *client;               //  Socket to broker
    int verbose;                //  Print activity to stdout
    int timeout;                //  Request timeout
};

//  Connect or reconnect to broker. In this asynchronous class we use a
//  DEALER socket instead of a REQ socket; this lets us send any number
//  of requests without waiting for a reply.

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
```

```
    self->client = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s...", self->broker);
}

//  The constructor and destructor are the same as in mdcliapi, except
//  we don't do retries, so there's no retries property.
//  .skip
//  ---------------------------------------------------------------------
//  Constructor

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  msecs

    s_mdcli_connect_to_broker (self);
    return self;
}

//  Destructor
```

```
void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}

//  Set request timeout

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}

//  .until
//  .skip
//  The send method now just sends one message, without waiting for a
//  reply. Since we're using a DEALER socket we have to send an empty
//  frame at the start, to create the same envelope that the REQ socket
//  would normally make for us:
```

```
int
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  Prefix request with protocol frames
    //  Frame 0: empty (REQ emulation)
    //  Frame 1: "MDPCxy" (six bytes, MDP/Client x.y)
    //  Frame 2: Service name (printable string)
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    zmsg_pushstr (request, "");
    if (self->verbose) {
        zclock_log ("I: send request to '%s' service:", service);
        zmsg_dump (request);
    }
    zmsg_send (&request, self->client);
    return 0;
}

//  .skip
//  The recv method waits for a reply message and returns that to the
//  caller.
//  ---------------------------------------------------------------------
//  Returns the reply message or NULL if there was no reply. Does not
//  attempt to recover from a broker failure, this is not possible
//  without storing all unanswered requests and resending them all...
```

```
zmsg_t *
mdcli_recv (mdcli_t *self)
{
    assert (self);

    //  Poll socket for a reply, with timeout
    zmq_pollitem_t items [] = { { self->client, 0, ZMQ_POLLIN, 0 } };
    int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
    if (rc == -1)
        return NULL;            //  Interrupted

    //  If we got a reply, process it
    if (items [0].revents & ZMQ_POLLIN) {
        zmsg_t *msg = zmsg_recv (self->client);
        if (self->verbose) {
            zclock_log ("I: received reply:");
            zmsg_dump (msg);
        }
        //  Don't try to handle errors, just assert noisily
        assert (zmsg_size (msg) >= 4);

        zframe_t *empty = zmsg_pop (msg);
        assert (zframe_streq (empty, ""));
        zframe_destroy (&empty);

        zframe_t *header = zmsg_pop (msg);
        assert (zframe_streq (header, MDPC_CLIENT));
        zframe_destroy (&header);
```

```
        zframe_t *service = zmsg_pop (msg);
        zframe_destroy (&service);

        return msg;     //  Success
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing client...\n");
    else
    if (self->verbose)
        zclock_log ("W: permanent error, abandoning request");

    return NULL;
}
```

차이점은 다음과 같습니다.

- 클라이언트에서 REQ 대신 DEALER 소켓을 사용하여, 각 요청과 각 응답 전에 공백 구분자(empty delimiter) 프레임을 넣어 봉투(envelope)를 구성합니다.
- 요청을 재시도하지 않습니다. 응용프로그램을 재시도 필요한 경우 자체적으로 수행할 수 있습니다.
- 동기식 send(`mdcli_send()`) 메서드를 비동기식 send(`mdcli_send()`) 및 recv(`mdcli_recv()`) 메서드로 분리합니다.
- send 메서드는 비동기식이며 전송 후 즉시 결과를 반환하기 때문에 발신자는 응답을 받기 전에 많은 메시지들을 보낼 수 있습니다.
- recv 메서드는 하나의 응답을 기다렸다가(제한시간(2.5초) 내) 호출자에게 응답 메시지를 반환합니다.

다음은 대응하는 클라이언트 테스트 프로그램으로, 100,000개의 메시지들를 보낸 다음 100,000개의 메시지를 받습니다.

mdclient2.c: 비동기 MDP 클라이언트

```
//  Majordomo Protocol client example - asynchronous
//  Uses the mdcli API to hide all MDP aspects

//  Lets us build this source without creating a library
#include "mdcliapi2.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        mdcli_send (session, "echo", &request);
    }
    for (count = 0; count < 100000; count++) {
        zmsg_t *reply = mdcli_recv (session);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  Interrupted by Ctrl-C
    }
    printf ("%d replies received\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

프로토콜을 전혀 수정하지 않았기 때문에 브로커와 작업자의 코드는 변경되지 않았습니

다. 즉각적인 성능 향상이 있었습니다. 다음은 동기식 클라이언트에서 100,000개의 요청-응답 수행했던 결과입니다.

```
$ time mdclient
100000 requests/replies processed

real    0m14.088s
user    0m1.310s
sys     0m2.670s
```

다음은 비동기 클라이언트에서 하나의 작업자에서 수행한 결과입니다.

```
$ time mdclient2
100000 replies received

real    0m8.730s
user    0m0.920s
sys     0m1.550s
```

2배 정도 빠르며 나쁘진 않지만, 10명의 작업자 구동하여 처리 결과를 보도록 하겠습니다.

```
$ time mdclient2
100000 replies received

real    0m3.863s
user    0m0.730s
sys     0m0.470s
```

- [옮긴이] 비동기식으로 개선된 클라이언트(mdclient2) 수행 결과 기존 동기식 보다 약 2배 정도 성능의 차이가 있었습니다.
- [옮긴이] 빌드 및 테스트

```
cl -EHsc mdclient2.c libzmq.lib czmq.lib

./mdbroker

./mdworker

./mdclient2 -v
20-08-20 10:57:39 I: connecting to broker at tcp://localhost:5555...
20-08-20 10:57:39 I: send request to 'echo' service:
D: 20-08-20 10:57:39 [000]
D: 20-08-20 10:57:39 [006] MDPC01
D: 20-08-20 10:57:39 [004] echo
D: 20-08-20 10:57:39 [011] Hello world
...
```

완전히 비동기식은 아닌 것은 작업자는 마지막 사용 기준으로 메시지를 받기 때문입니다. 그러나 더 많은 작업자들을 통해 확장 가능합니다. 내 PC(4개의 코어(CPU))에서 8개 정도의 작업자들이 있으면 성능은 그다지 개선되지 않습니다. 그러나 클라이언트를 동기식을 비동기식으로 변경하는 몇 분의 작업으로 처리량이 4배 향상되었습니다. 브로커는 여전히 최적화되지 않았습니다. 제로 복사(zero-copy)를 수행하는 대신 메시지 프레임을 복사하는 데 대부분의 시간을 소비합니다. 그러나 우리는 매우 적은 노력으로 초당 25K의 안정적인 요청/응답 호출을 받고 있습니다.

그러나 비동기식 MDP 패턴이 장밋빛처럼 아름다운 것은 아닙니다. 그것은 근본적인 약점을 가지고 있는데, 더 많은 작업 없이는 브로커 장애에서 살아남을 수 없다는 것입니다. mdcliapi2 코드를 보면 장애 후에 재연결을 시도하지 않으며, 올바르게 재연결을 위해서는 다음의 작업이 필요합니다.

- 클라이언트의 모든 요청 번호에 매핑되는 응답 번호, 이상적으로 적용하려면 통신규약을 변경해야 합니다.
- 클라이언트 API의 모든 미해결 요청(예 : 응답이 미수신된 요청)을 추적하고 유지합니

다.

- 장애조치의 경우, 클라이언트 API가 모든 미해결 요청을 브로커에 재전송합니다.

핵심 구현 기준은 아니지만 성능의 대가는 복잡성이란 것을 의미합니다. MDP를 위해 구현할 가치가 있을까요? 수천 명의 클라이언트들을 지원하는 웹 프론트엔드의 경우 필요하겠지만, DNS와 같은 이름 조회 서비스의 경우 하나의 요청에서 세션이 완료되면 더 이상 필요하지 않습니다.

0.48 서비스 검색

우리는 훌륭한 서비스 지향 브로커를 가지고 있지만, 특정 서비스를 사용할 수 있는지 여부를 알 수는 없습니다. 클라이언트에서 요청이 실패했는지 여부(제한시간 내에 응답이 오지 않음)는 알지만 이유는 알 수 없습니다. 브로커에게 "에코 서비스가 실행 중입니까?"라고 물을 수 있으면 유용합니다. 가장 확실한 방법은 MDP/클라이언트 통신규약을 수정하여 브로커에 서비스 실행 여부를 묻는 명령을 추가하는 것입니다. 그러나 MDP/클라이언트는 단순하다는 매력이 있지만 서비스 검색 기능을 추가하면 MDP/작업자 통신규약만큼 복잡해집니다.

또 다른 방법은 이메일이 수행하는 작업으로, 전달할 수 없는 요청은 반환되게 합니다. 이것은 비동기식으로 운영되는 클라이언트와 브로커에서 잘 작동하지만 복잡성이 추가합니다. 요청에 따른 응답과 반환된 요청들을 제대로 구분해야 합니다.

기존에 MDP을 수정하는 대신에, MDP상에 추가적인 기능으로 구축해 보겠습니다. 서비스 검색은 그 자체로 서비스이며, "서비스 X 비활성화", "통계 제공"등과 같이 여러 관리 서비스들 중 하나와 같습니다. 우리가 원하는 것은 일반적이고 확장 가능한 솔루션이며 통신규약이나 기존 응용프로그램에는 영향이 없어야 합니다.

RFC로 MMI: Majordomo Management Interface는 MDP 통신규약 위에 구축된 작은 RFC입니다. 이미 브로커에서 구현(`s_broker_client_msg()`)했지만, 코드 전부를 읽지 않는다면 아마 놓쳤을 것입니다. 브로커에서 어떻게 작동하는지 설명하겠습니다.

- 클라이언트가 `mmi.`로 시작하는 서비스를 요청하면, 브로커는 작업자에게 전달하는 대신 내부적으로 처리합니다.
- 브로커에서는 서비스 검색 서비스로 `mmi.service` 처리합니다.

- 요청에 대한 반환값은 외부 서비스의 이름(작업자가 제공 한 실제 서비스)입니다.
- 브로커는 해당 서비스로 등록된 작업자가 존재 여부에 따라 “200”(OK) 또는 “404”(Not found)를 반환합니다.

응용프로그램에서 서비스 검색을 사용하는 방법은 다음과 같습니다.

mmiecho.c: MDP상 서비스 검색

```
//  MMI echo query example

//  Lets us build this source without creating a library
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  This is the service we want to look up
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");

    //  This is the service we send our request to
    zmsg_t *reply = mdcli_send (session, "mmi.service", &request);

    if (reply) {
        char *reply_code = zframe_strdup (zmsg_first (reply));
        printf ("Lookup echo service: %s\n", reply_code);
        free (reply_code);
        zmsg_destroy (&reply);
    }
```

```
    else
        printf ("E: no response from broker, make sure it's running\n");

    mdcli_destroy (&session);
    return 0;
}
```

- [옮긴이] 브로커(mdbroker)에 s_broker_client_msg()에 구현되어 있으며 "mmi.service"가 아닌 경우 작업자로 메시지를 전달(s_service_dispatch())하게 한다.
- [옮긴이] 빌드 및 테스트

```
 cl -EHsc mmiecho.c libzmq.lib czmq.lib

// 클라이언트에서 "echo" 서비스 등록 여부에 대하여 브로커에 "echo" 서비스가 등록된 경우
 ./mmiecho -v
20-08-20 11:40:46 I: connecting to broker at tcp://localhost:5555...
20-08-20 11:40:46 I: send request to 'mmi.service' service:
D: 20-08-20 11:40:46 [006] MDPC01
D: 20-08-20 11:40:46 [011] mmi.service
D: 20-08-20 11:40:46 [004] echo
20-08-20 11:40:46 I: received reply:
D: 20-08-20 11:40:46 [006] MDPC01
D: 20-08-20 11:40:46 [011] mmi.service
D: 20-08-20 11:40:46 [003] 200
Lookup echo service: 200

// 클라이언트에서 "nico" 서비스 등록 여부에 대하여 브로커에 "nico" 서비스가 미등록된 경우
 ./mmiecho -v
20-08-20 11:41:09 I: connecting to broker at tcp://localhost:5555...
20-08-20 11:41:09 I: send request to 'mmi.service' service:
```

```
D: 20-08-20 11:41:09 [006] MDPC01
D: 20-08-20 11:41:09 [011] mmi.service
D: 20-08-20 11:41:09 [004] nico
20-08-20 11:41:09 I: received reply:
D: 20-08-20 11:41:09 [006] MDPC01
D: 20-08-20 11:41:09 [011] mmi.service
D: 20-08-20 11:41:09 [003] 404
Lookup echo service: 404
```

"echo" 서비스로 등록된 작업자가 실행 혹은 실행되지 않는 상황에서 테스트하면 "200(OK)" 혹은 "404(Not found)"가 표시됩니다. 예제에서 브로커에서 MMI를 구현한 것은 조잡합니다. 예를 들어, 작업자가 사라지더라도 서비스는 "현재"상태로 유지됩니다. 사실 브로커는 제한 시간 후에 작업자가 없는 서비스를 제거해야합니다.

- [옮긴이] "echo" 서비스로 브로커에 등록된 작업자를 중지시키고 "mmiecho"을 두 차례 수행하면 "200(OK)"가 나오나 3번째 부터는 "404(Not found)"가 출력되는 것은 브로커에서 심박 수행시 "s_broker_purge()"을 통하여 대기중인 작업자들의 제한시간 초과(7.5초=2.5초(HEARTBEAT_INTERVAL) * 3(HEARTBEAT_LIVENESS))된 경우 삭제하기 때문입니다.

```
 ./mmiecho -v
20-08-20 11:50:16 I: connecting to broker at tcp://localhost:5555...
20-08-20 11:50:16 I: send request to 'mmi.service' service:
...
Lookup echo service: 200
 stop-process -name mdworker
 ./mmiecho -v
20-08-20 11:50:28 I: connecting to broker at tcp://localhost:5555...
20-08-20 11:50:28 I: send request to 'mmi.service' service:
...
Lookup echo service: 200
```

```
 ./mmiecho -v
20-08-20 11:50:29 I: connecting to broker at tcp://localhost:5555...
20-08-20 11:50:29 I: send request to 'mmi.service' service:
...
Lookup echo service: 200
 ./mmiecho -v
20-08-20 11:50:30 I: connecting to broker at tcp://localhost:5555...
20-08-20 11:50:30 I: send request to 'mmi.service' service:
...
Lookup echo service: 404
```

0.49 멱등성 서비스

멱등성은 약으로 복용하는 것이 아닙니다. 이것이 의미하는 바는 작업을 반복해도 안전하다는 것입니다. 시계의 시간을 확인하기는 멱등성입니다. 아이들에게 신용 카드를 빌려주기는 아닙니다(아이에 따라 결과가 달라짐). 많은 클라이언트-서버 사용 사례들은 멱등성이지만 일부는 그렇지 않습니다. 멱등성 사례의 다음과 같습니다.

- [옮긴이] 멱등성(□□□)은 연산을 여러 번 적용하더라도 결과가 달라지지 않는 성질입니다.
- 예) 절댓값 함수 - $\mathrm{abs}(\mathrm{abs}(x)) = \mathrm{abs}(x)$
- 상태 비저장 작업 배포는 멱등성입니다. 서버들이 상태 비저장 작업자들로써 역할로 요청에 의해 제공되는 상태를 기반으로 응답하는 파이프라인을 수행합니다. 이 경우 동일한 요청을 여러 번 실행해도 안전합니다(비효율적임).
- 이름 서비스는 멱등성입니다. 이름 서비스는 논리적 주소를 바인딩하거나 연결할 단말들로 변환합니다. 이러한 경우 동일한 조회 요청을 여러 번 수행해도 안전합니다.

멱등성 아닌 경우는 다음과 같습니다.

- 로깅 서비스. 동일한 로그 정보가 한번 이상 기록되는 것을 원하지 않습니다.
- 하류(downstream) 노드에 영향을 미치는 모든 서비스, 예 : 다른 노드들에 정보 전송. 해당 서비스가 동일한 요청을 한번 이상 받으면 하류 노드들이 중복 정보를 가지게 됩니다.
- 멱등성이 아닌 방식으로 공유 데이터를 수정하는 모든 서비스. 예 : 은행 계좌에서 인출하는 서비스는 추가 작업 없이는 멱등성이 아닙니다.

서버 응용프로그램들이 멱등성이 아닌 경우, 정확히 언제 충돌(종료)할 수 있는지 좀 더 신중하게 생각해야 합니다. 응용프로그램이 유휴 상태이거나 요청을 처리하는 동안 종료되는 경우는 그나마 괜찮지만, 금융권에서 클라이언트 요청에 의해 차변과 대변 처리 트랜잭션을 처리하다가 서버가 응답을 보내는 중에 죽는다면 문제입니다. 왜냐하면 서버에서 해당 트랜잭션을 처리했지만 클라이언트로 응답을 보내지 못했기 때문입니다.

응답이 클라이언트로 반환되는 순간 네트워크가 중단되면, 동일한 문제가 발생합니다. 클라이언트는 서버가 죽었다고 생각하고 재요청을 하며 서버는 동일한 작업을 두 번 하게 되어 우리가 원하지 않는 결과를 발생합니다.

멱등성이 아닌 작업을 처리하려면 중복 요청들을 감지하고 거부하는 표준적인 솔루션을 사용해야 하며, 다음을 의미합니다.

- 클라이언트는 모든 요청에 고유한 클라이언트 식별자(ID)와 고유한 메시지 번호로 넣어야 합니다.(client ID + Message No + request)
- 서버는 응답을 보내기 전에, 응답 메시지에 클라이언트 식별자(ID)와 메시지 번호의 조합하여 키로 저장합니다.(client ID + Message No + reply)
- 서버는 클라이언트로부터 요청을 받으면, 먼저 해당 클라이언트 ID와 메시지 번호에 대한 응답이 있는지 확인합니다(client ID + Message No + reply). 서버에 이미 응답이 있다면 요청을 처리하지 않고 기존 응답만 다시 보냅니다.

0.50 비연결 신뢰성(타이타닉 패턴)

MDP가 “신뢰할 수 있는” 메시지 브로커라는 사실을 알게 되면 하드디스크를 추가 할 수 있습니다. 결국 이것은 대부분의 기업 메시징 시스템에서 동작합니다. 이와 같이 유혹적인

생각은 다소 부정적일 수밖에 없는 것에 유감입니다. 하지만 냉소주의는 내 전문 분야 중 하나입니다. 그래서 아키텍처의 중심에 하드디스크 기반 브로커를 배치하여 지속성을 유지하지 않는 이유는 다음과 같습니다.

- 게으른 해적 클라이언트(LPP)는 잘 작동하였습니다. 클라이언트-서버 직접 연결(LPP)에서 부하 분산 브로커(SPP)까지 전체 범위의 아키텍처에서 동작합니다. 게으른 해적 패턴의 작업자는 상태 비저장이며 멱등성이라고 가정하는 경향이 있지만, 지속성이 필요한 경우 한계가 있습니다.
- 하드디스크는 느린 성능에서 부가적인 관리(새벽 6시 장애, 일과 시작 시 필연적인 고장 등), 수리 등 많은 문제를 가져옵니다. 일반적으로 해적 패턴의 아름다움은 단순성입니다. 해적 패턴의 요소들(클라이언트, 브로커, 작업자)은 충돌하지 않지만, 여전히 하드웨어가 걱정된다면 브로커가 없는 P2P(Peer-To-Peer) 패턴으로 이동할 수 있습니다. 이장의 뒷부분에서 설명하겠습니다.

그러나 이렇게 말했지만 , 하드디스크 기반 안정성이 필연적인 사용 사례는 비연결 비동기 네트워크입니다. 해적의 주요 문제, 즉 클라이언트가 실시간으로 응답을 기다려야 하는 문제를 해결합니다. 클라이언트와 작업자가 가끔씩만 연결되어 있는 경우(이메일과 유사하게 생각) 클라이언트들과 작업자들 간에 상태 비저장 네트워크를 사용할 수 없습니다. 상태를 중간에 두어야 합니다.

여기 타이타닉 패턴이 있으며, 우리는 메시지를 디스크에 기록하여 클라이언트들과 작업자들이 산발적으로 연결되어 있어도 메시지가 유실되지 않도록 합니다. MDP에서 브로커에 서비스 검색 기능(`s_broker_client_msg()`)을 추가한 것처럼 타이타닉을 별도의 통신규약으로 확장하는 대신 MDP 위에 계층화할 것입니다. 이것은 놀랍게도 게으르게 보이는 것은 발사 후 망각형(fire-and-forget) 신뢰성을 브로커가 아닌 전문화된 작업자에 구현하기 때문입니다. 작업자에 구현하기 좋은 이유는 다음과 같습니다.

- 우리가 나누고 정복하기 때문에 훨씬 쉽습니다. 브로커가 메시지 라우팅을 처리하고, 작업자는 신뢰성을 처리합니다.
- 특정 개발 언어로 작성된 브로커와 다른 개발 언어로 작성된 작업자를 혼합할 수 있습니다.

- 작업자를 통해 발사 후 망각형 기술로 독자적으로 진화시킵니다. 단 하나의 단점은 브로커와 하드디스크 간에 부가적인 네트워크 홉이지만 장점들은 단점을 능가하는 가치가 있습니다.

지속적인 요청-응답 아키텍처를 만드는 방법들은 많이 있지만, 우리는 간단하고 고통 없는 것을 지향하겠습니다. 몇 시간 동안 사용해 본 후 가장 간단한 설계는 "프록시 서비스"입니다. 즉, 타이타닉은 작업자들에게 영향을 주지 않습니다. 클라이언트가 즉시 응답을 원하면, 서비스에 직접 말하고 서비스가 가용하기를 기대합니다. 클라이언트가 기꺼이 기다리며 타이타닉과 대화하면서 "이봐, 친구, 내가 식료품을 사는 동안 이걸 처리해 주시겠어요?"라고 묻습니다.

그림 51 - 타이타닉 패턴

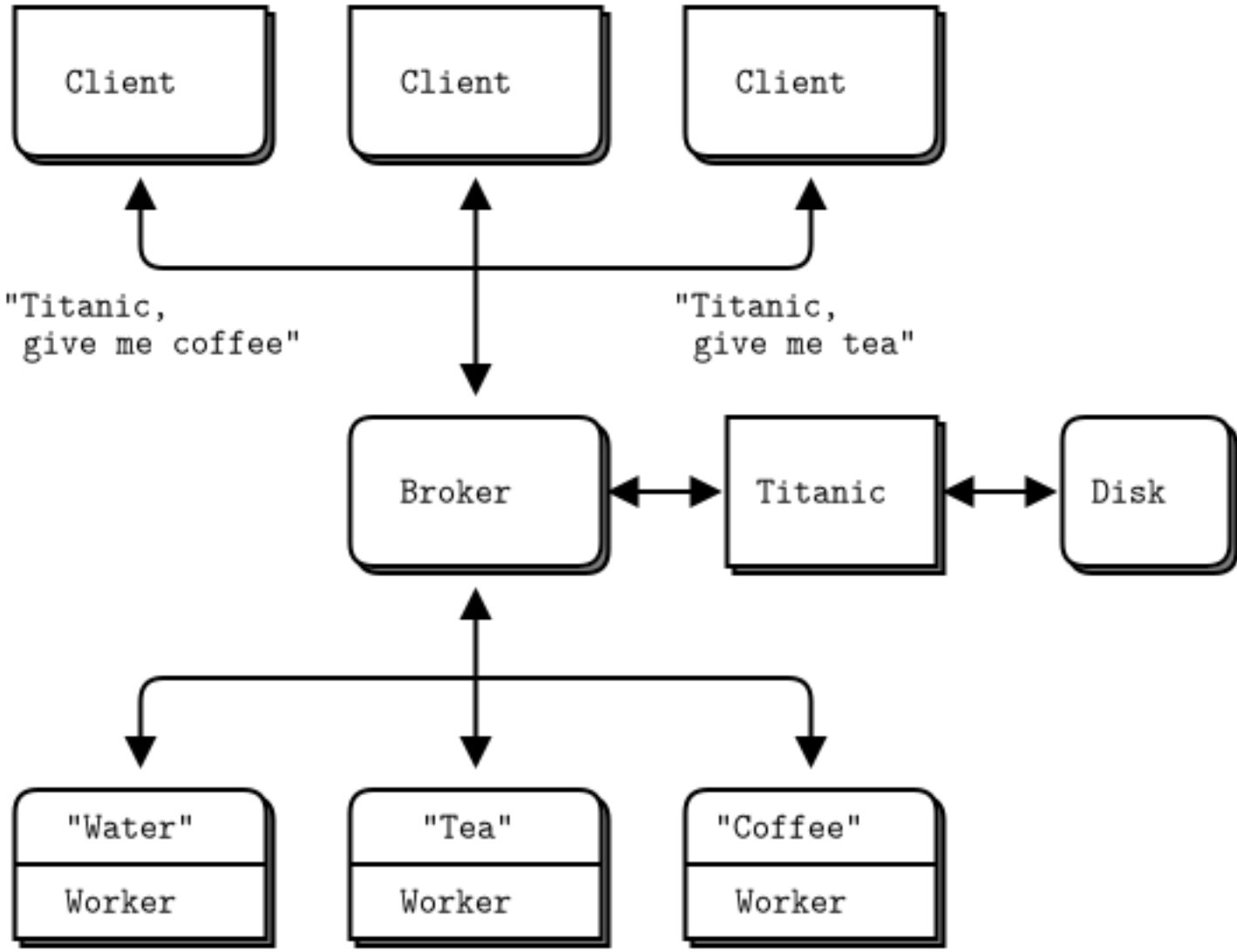


그림 53: The Titanic Pattern

타이타닉은 작업자이자 클라이언트이며, 클라이언트와 타이타닉 간의 대화는 다음과 같습니다.

- 클라이언트 : 나의 요청을 수락해 주세요. - 타이타닉 : 네, 수락했습니다.
- 클라이언트 : 저에게 응답이 있나요? - 타이타닉 : 네, 여기 있습니다. 혹은 아니요, 아직

없습니다.

- 클라이언트 : 네. 요청을 지워주신다면 감사하겠습니다. - 타이타닉 : 네, 지웠습니다.

한편 타이타닉과 브로커, 작업자 간의 대화는 다음과 같습니다.

- 타이타닉 : 안녕, 브로커, 커피 서비스가 있나요? - 브로커 : 음, 예, 있는 것 같아요
- 타이타닉 : 안녕, 커피 서비스(작업자), 이거 처리해 주세요.
- 커피(작업자) : 네, 여기 있습니다.
- 타이타닉 : 달콤해!

가능한 실패 시나리오를 통해 작업할 수 있습니다. 요청을 처리하는 동안 작업자가 중지하면 타이타닉은 계속 재요청합니다. 작업자의 응답이 어딘가에서 유실되면 타이타닉은 재요청합니다. 작업자를 통해 요청이 처리되었지만 클라이언트가 응답을 받지 못하면 클라이언트에서 다시 묻습니다. 요청 또는 응답을 처리하는 동안 타이타닉이 중지되면 클라이언트가 다시 시도합니다. 요청이 확실히 저장소에 기록되어 있으며 메시지는 유실되지 않습니다.

핸드 세이킹(요청-응답처럼 쌍으로 동작)은 현명하지만 파이프라인이 가능합니다. 예 : 클라이언트는 비동기 MDP 패턴을 사용하여 많은 작업을 수행한 다음 응답을 나중에 받습니다.

클라이언트가 응답을 요청하기 위한 방법이 필요합니다. 동일한 서비스를 요청하는 많은 클라이언트들이 있으며, 클라이언트들은 사라지거나 다른 식별자로 재등장합니다. 다음은 단순하지만 합리적이고 안전한 솔루션이 있습니다.

- 모든 요청은 UUID(Universally Unique ID)를 생성하며, 타이타닉이 요청을 대기열에 넣은 후 클라이언트로 반환합니다.
- 클라이언트가 응답을 요청할 때, 원래 요청에 대한 UUID를 지정해야 합니다.

실제로는, 클라이언트는 요청 UUID는 안전하게 로컬 데이터베이스에 저장될 것입니다.

타이타닉에 대한 공식 사양서를 작성하기 전에, 클라이언트가 타이타닉과 통신하는 방법을 고려하겠습니다. 하나는 단일 서비스를 사용하여 3개 요청 유형을 보내는 것입니다. 두 번째는 더 단순하게 그냥 3개의 서비스를 사용하는 것입니다.

- titanic.request : [titanic-client] 요청 메시지를 저장하고, 요청에 대한 UUID를 반환합니다.

- titanic.reply : [titanic-worker] 주어진 요청 UUID에 대해 가능한 응답을 가져옵니다.
- titanic.close : [titanic-client] 응답이 저장되고 처리되었는지 확인합니다.

ØMQ에 대한 멀티스레딩 경험을 통해 간단한 멀티스레드 작업자를 작성해 3개의 서비스를 제공합니다. 타이타닉에서 사용할 ØMQ 메시지와 프레임 설계를 하겠습니다. 이를 타이타닉 서비스 통신규약(TSP)으로 제공됩니다.

TSP를 사용하면 MDP를 통해 직접 서비스에 접근하는 것보다 클라이언트 응용프로그램에 더 많은 작업이 수행됩니다. 다음은 짧지만 강력한 "echo"클라이언트 예제입니다.

ticlient.c: 타이타닉 클라이언트

```
//  Titanic client example
//  Implements client side of http://rfc.zeromq.org/spec:9

//  Lets build this source without creating a library
#include "mdcliapi.c"

//  Calls a TSP service
//  Returns response if successful (status code 200 OK), else NULL
//
static zmsg_t *
s_service_call (mdcli_t *session, char *service, zmsg_t **request_p)
{
    zmsg_t *reply = mdcli_send (session, service, request_p);
    if (reply) {
        zframe_t *status = zmsg_pop (reply);
        if (zframe_streq (status, "200")) {
            zframe_destroy (&status);
            return reply;
        }
        else
```

```
        if (zframe_streq (status, "400")) {
            printf ("E: client fatal error, aborting\n");
            exit (EXIT_FAILURE);
        }
        else
        if (zframe_streq (status, "500")) {
            printf ("E: server fatal error, aborting\n");
            exit (EXIT_FAILURE);
        }
    }
    else
        exit (EXIT_SUCCESS);    //  Interrupted or failed

    zmsg_destroy (&reply);
    return NULL;        //  Didn't succeed; don't care why not
}

//  .split main task
//  The main task tests our service call by sending an echo request:

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  1. Send 'echo' request to Titanic
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");
    zmsg_addstr (request, "Hello world");
```

```
    zmsg_t *reply = s_service_call (
        session, "titanic.request", &request);

    zframe_t *uuid = NULL;
    if (reply) {
        uuid = zmsg_pop (reply);
        zmsg_destroy (&reply);
        zframe_print (uuid, "I: request UUID ");
    }
    //  2. Wait until we get a reply
    while (!zctx_interrupted) {
        zclock_sleep (100);
        request = zmsg_new ();
        zmsg_add (request, zframe_dup (uuid));
        zmsg_t *reply = s_service_call (
            session, "titanic.reply", &request);

        if (reply) {
            char *reply_string = zframe_strdup (zmsg_last (reply));
            printf ("Reply: %s\n", reply_string);
            free (reply_string);
            zmsg_destroy (&reply);

            //  3. Close request
            request = zmsg_new ();
            zmsg_add (request, zframe_dup (uuid));
            reply = s_service_call (session, "titanic.close", &request);
            zmsg_destroy (&reply);
            break;
```

```
        }
        else {
            printf ("I: no reply yet, trying again...\n");
            zclock_sleep (5000);      //  Try again in 5 seconds
        }
    }
    zframe_destroy (&uuid);
    mdcli_destroy (&session);
    return 0;
}
```

물론 이러한 기능은 어떤 종류의 프레임워크나 API로 감싸질 수 있습니다. 일반 응용프로그램 개발자에게 전체 메시징의 세부 사항을 배우도록 요청하는 것은 좋지 않습니다 : 이는 개발자들의 머리를 아프게 하고, 시간이 소요되며, 코드가 복잡하고 버그가 많게 되며 지능을 부가하기 어렵습니다.

예를 들어, 실제 응용프로그램에서 클라이언트는 각 요청을 차단할 수 있지만, 작업이 실행되는 동안 유용한 작업을 수행하고 싶어 합니다. 이를 위하여 백그라운드 스레드를 구축하고 스레드 간 통신하기 위해 주요한 연결 작업이 필요합니다. 이런 요구사항에 대하여 일반 개발자가 잘못 사용하지 않도록 단순한 API로 감싸 내부를 추상화하는 방법을 제공하는 것이, MDP에서 서비스 확인에서 사용한 동일한 방식입니다.

다음은 타이타닉 서버의 구현입니다. 타이타닉은 클라이언트와의 대화를 처리하기 위해 3개의 스레드들을 사용하여 3개의 서비스를 처리하며, 가장 확실한 접근 방식(메시지 당 하나의 파일)을 사용하여 디스크를 통한 완전한 지속성을 수행합니다. 너무 놀라울 만큼 단순합니다. 유일한 복잡한 부분은 디렉터리를 계속해서 읽는 것을 피하기 위해 모든 요청들에 대한 개별 대기열을 보유하게 하는 부분입니다.

titanic.c: 타이타닉 서버

```
//  Titanic service
//  Implements server side of http://rfc.zeromq.org/spec:9
```

```
//  Lets us build this source without creating a library
#include "mdwrkapi.c"
#include "mdcliapi.c"

#include "zfile.h"
#ifdef _WIN32
#pragma comment(lib, "rpcrt4.lib")  // UuidCreate - Minimum supported OS Win 2000
#include <windows.h>
#else
#include <uuid/uuid.h>
#endif

//  Return a new UUID as a printable character string
//  Caller must free returned string when finished with it
#ifdef _WIN32
static char *
s_generate_uuid (void)
{
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    UuidCreate(&uuid);
    char *uuidgenstr = zmalloc(sizeof(uuid_t));
    memcpy(uuidgenstr, &uuid, sizeof(uuid_t));
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuidgenstr [byte_nbr] & 8];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuidgenstr [byte_nbr] & 15];
    }
```

```
    free(uuidgenstr);
    return uuidstr;
}
#else
static char *
s_generate_uuid (void)
{
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    uuid_generate (uuid);
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuid [byte_nbr] >> 4];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuid [byte_nbr] & 15];
    }
    return uuidstr;
}
#endif

//  Returns freshly allocated request filename for given UUID

#define TITANIC_DIR ".titanic"

static char *
s_request_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.req", uuid);
    return filename;
```

```
}

//  Returns freshly allocated reply filename for given UUID

static char *
s_reply_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.rep", uuid);
    return filename;
}

//  .split Titanic request service
//  The {{titanic.request}} task waits for requests to this service. It writes
//  each request to disk and returns a UUID to the client. The client picks
//  up the reply asynchronously using the {{titanic.reply}} service:

static void
titanic_request (void *args, zctx_t *ctx, void *pipe)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.request", 0);
    zmsg_t *reply = NULL;

    while (true) {
        //  Send reply if it's not null
        //  And then get next request from broker
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  Interrupted, exit
```

```
        //  Ensure message directory exists
        zfile_mkdir (TITANIC_DIR);

        //  Generate UUID and save message to disk
        char *uuid = s_generate_uuid ();
        char *filename = s_request_filename (uuid);
        FILE *file = fopen (filename, "w");
        assert (file);
        zmsg_save (request, file);
        fclose (file);
        free (filename);
        zmsg_destroy (&request);

        //  Send UUID through to message queue
        reply = zmsg_new ();
        zmsg_addstr (reply, uuid);
        zmsg_send (&reply, pipe);

        //  Now send UUID back to client
        //  Done by the mdwrk_recv() at the top of the loop
        reply = zmsg_new ();
        zmsg_addstr (reply, "200");
        zmsg_addstr (reply, uuid);
        free (uuid);
    }
    mdwrk_destroy (&worker);
}
```

```
//  .split Titanic reply service
//  The {{titanic.reply}} task checks if there's a reply for the specified
//  request (by UUID), and returns a 200 (OK), 300 (Pending), or 400
//  (Unknown) accordingly:

static void *
titanic_reply (void *context)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.reply", 0);
    zmsg_t *reply = NULL;

    while (true) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  Interrupted, exit

        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
        if (zfile_exists (rep_filename)) {
            FILE *file = fopen (rep_filename, "r");
            assert (file);
            reply = zmsg_load (NULL, file);
            zmsg_pushstr (reply, "200");
            fclose (file);
        }
        else {
            reply = zmsg_new ();
```

```
            if (zfile_exists (req_filename))
                zmsg_pushstr (reply, "300"); //Pending
            else
                zmsg_pushstr (reply, "400"); //Unknown
        }
        zmsg_destroy (&request);
        free (uuid);
        free (req_filename);
        free (rep_filename);
    }
    mdwrk_destroy (&worker);
    return 0;
}

//  .split Titanic close task
//  The {{titanic.close}} task removes any waiting replies for the request
//  (specified by UUID). It's idempotent, so it is safe to call more than
//  once in a row:

static void *
titanic_close (void *context)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.close", 0);
    zmsg_t *reply = NULL;

    while (true) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
```

```
            break;      //  Interrupted, exit

        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
        zfile_delete (req_filename);
        zfile_delete (rep_filename);
        free (uuid);
        free (req_filename);
        free (rep_filename);

        zmsg_destroy (&request);
        reply = zmsg_new ();
        zmsg_addstr (reply, "200");
    }
    mdwrk_destroy (&worker);
    return 0;
}

//  .split worker task
//  This is the main thread for the Titanic worker. It starts three child
//  threads; for the request, reply, and close services. It then dispatches
//  requests to workers using a simple brute force disk queue. It receives
//  request UUIDs from the {{titanic.request}} service, saves these to a disk
//  file, and then throws each request at MDP workers until it gets a
//  response.

static int s_service_success (char *uuid);
```

```
int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    zctx_t *ctx = zctx_new ();

    void *request_pipe = zthread_fork (ctx, titanic_request, NULL);
    zthread_new (titanic_reply, NULL);
    zthread_new (titanic_close, NULL);

    //  Main dispatcher loop
    while (true) {
        //  We'll dispatch once per second, if there's no activity
        zmq_pollitem_t items [] = { { request_pipe, 0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted
        if (items [0].revents & ZMQ_POLLIN) {
            //  Ensure message directory exists
            zfile_mkdir (TITANIC_DIR);

            //  Append UUID to queue, prefixed with '-' for pending
            zmsg_t *msg = zmsg_recv (request_pipe);
            if (!msg)
                break;          //  Interrupted
            FILE *file = fopen (TITANIC_DIR "/queue", "a");
            char *uuid = zmsg_popstr (msg);
            fprintf (file, "-%s\n", uuid);
            fclose (file);
            free (uuid);
```

```
            zmsg_destroy (&msg);
        }
        //  Brute force dispatcher
        char entry [] = "?.....:.....:.....:.....:";
        FILE *file = fopen (TITANIC_DIR "/queue", "r+");
        while (file && fread (entry, 33, 1, file) == 1) {
            //  UUID is prefixed with '-' if still waiting
            if (entry [0] == '-') {
                if (verbose)
                    printf ("I: processing request %s\n", entry + 1);
                if (s_service_success (entry + 1)) {
                    //  Mark queue entry as processed
                    fseek (file, -33, SEEK_CUR);
                    fwrite ("+", 1, 1, file);
                    fseek (file, 32, SEEK_CUR);
                }
            }
            //  Skip end of line, LF or CRLF
            if (fgetc (file) == '\r')
                fgetc (file);
            if (zctx_interrupted)
                break;
        }
        if (file)
            fclose (file);
    }
    return 0;
}
```

```
//  .split try to call a service
//  Here, we first check if the requested MDP service is defined or not,
//  using a MMI lookup to the Majordomo broker. If the service exists,
//  we send a request and wait for a reply using the conventional MDP
//  client API. This is not meant to be fast, just very simple:

static int
s_service_success (char *uuid)
{
    //  Load request message, service will be first frame
    char *filename = s_request_filename (uuid);
    FILE *file = fopen (filename, "r");
    free (filename);

    //  If the client already closed request, treat as successful
    if (!file)
        return 1;

    zmsg_t *request = zmsg_load (NULL, file);
    fclose (file);
    zframe_t *service = zmsg_pop (request);
    char *service_name = zframe_strdup (service);

    //  Create MDP client session with short timeout
    mdcli_t *client = mdcli_new ("tcp://localhost:5555", false);
    mdcli_set_timeout (client, 1000);  //  1 sec
    mdcli_set_retries (client, 1);     //  only 1 retry

    //  Use MMI protocol to check if service is available
```

```
    zmsg_t *mmi_request = zmsg_new ();
    zmsg_add (mmi_request, service);
    zmsg_t *mmi_reply = mdcli_send (client, "mmi.service", &mmi_request);
    int service_ok = (mmi_reply
        && zframe_streq (zmsg_first (mmi_reply), "200"));
    zmsg_destroy (&mmi_reply);

    int result = 0;
    if (service_ok) {
        zmsg_t *reply = mdcli_send (client, service_name, &request);
        if (reply) {
            filename = s_reply_filename (uuid);
            FILE *file = fopen (filename, "w");
            assert (file);
            zmsg_save (reply, file);
            fclose (file);
            free (filename);
            result = 1;
        }
        zmsg_destroy (&reply);
    }
    else
        zmsg_destroy (&request);

    mdcli_destroy (&client);
    free (service_name);
    return result;
}
```

- [옮긴이] CentOS7상에서 “uuid” 라이브러리 설치가 필요하며, 빌드시 “uuid” 라이브러

리를 포함해야 합니다.

```
[zedo@sook C]$ sudo yum install uuid uuid-devel libuuid-devel
[zedo@sook C]$ cc -o ticlient ticlient.c -lzmq -lczmq -luuid
```

- [옮긴이] s_generate_uuid() 함수를 윈도우에서 사용하기 위해 아래와 같이 변경합니다.

```
#ifdef _WIN32
static char *
s_generate_uuid (void)
{
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    UuidCreate(&uuid);
    char *uuidgenstr = zmalloc(sizeof(uuid_t));
    memcpy(uuidgenstr, &uuid, sizeof(uuid_t));
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuidgenstr [byte_nbr] & 8];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuidgenstr [byte_nbr] & 15];
    }
    free(uuidgenstr);
    return uuidstr;
}
#else
static char *
s_generate_uuid (void)
{
    char hex_char [] = "0123456789ABCDEF";
```

```
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    uuid_generate (uuid);
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuid [byte_nbr] >> 4];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuid [byte_nbr] & 15];
    }
    return uuidstr;
}
#endif
```

- [옮긴이] 위에서 만든 UUID 생성함수를 테스트하기 위한 test_uuid.c 입니다.

```
#include <czmq.h>
#ifdef _WIN32
#pragma comment(lib, "rpcrt4.lib")  // UuidCreate - Minimum supported OS Win 2000
#include <windows.h>
#else
#include <uuid/uuid.h>
#endif
void main()
{
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    #ifdef _WIN32
    UuidCreate(&uuid);
    #else
    uuid_generate (uuid);
```

```
    #endif
    char *uuidgenstr = zmalloc(sizeof(uuid_t));
    memcpy(uuidgenstr, &uuid, sizeof(uuid_t));
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuidgenstr [byte_nbr] & 8];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuidgenstr [byte_nbr] & 15];
    }
    free(uuidgenstr);
    printf("uuidstr : %s\n", uuidstr);
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc tigen_uuid.c libzmq.lib czmq.lib

 ./tigen_uuid
uuidstr : 8A8B898B8B8E028989078989048A8B8D
```

- [옮긴이] 빌드 및 테스트 수행시 윈도우 환경에 클라이언트에서 titanic.request 수행 후에 titanic.reply 수행시 titanic이 종료되는 현상 발생하였습니다(확인 진행중).

- 윈도우 환경에서 실행할 경우

```
 ./mdbroker -v
20-08-22 07:12:34 I: MDP broker/0.2.0 is active at tcp://*:5555
...

 ./titanic -v
I: processing request 03018C8B01008800008D880205888E06
```

```
 ./mdworker -v
20-08-22 07:12:56 I: connecting to broker at tcp://localhost:5555...
20-08-22 07:12:56 I: sending READY to broker
...

 ./ticlient -v
20-08-22 06:58:17 I: connecting to broker at tcp://localhost:5555...
20-08-22 06:58:17 I: send request to 'titanic.request' service:
D: 20-08-22 06:58:17 [006] MDPC01
D: 20-08-22 06:58:17 [015] titanic.request
D: 20-08-22 06:58:17 [004] echo
D: 20-08-22 06:58:17 [011] Hello world
20-08-22 06:58:17 I: received reply:
D: 20-08-22 06:58:17 [006] MDPC01
D: 20-08-22 06:58:17 [015] titanic.request
D: 20-08-22 06:58:17 [003] 200
D: 20-08-22 06:58:17 [032] 03018C8B01008800008D880205888E06
D: 20-08-22 06:58:17 I: request UUID [032] 03018C8B01008800008D880205888E06
20-08-22 06:58:17 I: send request to 'titanic.reply' service:
D: 20-08-22 06:58:17 [006] MDPC01
D: 20-08-22 06:58:17 [013] titanic.reply
D: 20-08-22 06:58:17 [032] 03018C8B01008800008D880205888E06
20-08-22 06:58:20 W: no reply, reconnecting...
20-08-22 06:58:20 I: connecting to broker at tcp://localhost:5555...
20-08-22 06:58:22 W: no reply, reconnecting...
20-08-22 06:58:22 I: connecting to broker at tcp://localhost:5555...
20-08-22 06:58:25 W: permanent error, abandoning
```

- CentOS7 환경에서 실행할 경우

```
[zedo@sook C]$ ./mdbroker -v
20-09-25 08:53:29 I: MDP broker/0.2.0 is active at tcp://*:5555
20-09-25 08:53:32 I: received message:
D: 20-09-25 08:53:32 [005] 0080000029
D: 20-09-25 08:53:32 [000]
D: 20-09-25 08:53:32 [006] MDPW01
D: 20-09-25 08:53:32 [001] 01
D: 20-09-25 08:53:32 [013] titanic.reply
20-09-25 08:53:32 I: registering new worker: 0080000029
20-09-25 08:53:32 I: added service: titanic.reply
20-09-25 08:53:32 I: received message:
D: 20-09-25 08:53:32 [005] 008000002A
D: 20-09-25 08:53:32 [000]
D: 20-09-25 08:53:32 [006] MDPW01
D: 20-09-25 08:53:32 [001] 01
D: 20-09-25 08:53:32 [015] titanic.request
20-09-25 08:53:32 I: registering new worker: 008000002A
20-09-25 08:53:32 I: added service: titanic.request
20-09-25 08:53:32 I: received message:
D: 20-09-25 08:53:32 [005] 008000002B
D: 20-09-25 08:53:32 [000]
D: 20-09-25 08:53:32 [006] MDPW01
D: 20-09-25 08:53:32 [001] 01
D: 20-09-25 08:53:32 [013] titanic.close
20-09-25 08:53:32 I: registering new worker: 008000002B
...
20-09-25 08:54:29 I: received message:                    --> [titanic.request] ticlient -> b
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
```

```
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [015] titanic.request
D: 20-09-25 08:54:29 [004] echo
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: sending REQUEST to worker            --> [REQUEST] broker -> titinic
D: 20-09-25 08:54:29 [005] 006B8B456C
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 02
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [004] echo
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: received message:                    --> [REPLY] titanic -> broker(uuid)
D: 20-09-25 08:54:29 [005] 006B8B456C
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 03
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [003] 200
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
20-09-25 08:54:29 I: received message:                    --> [mmi.service] ticlient -> broke
D: 20-09-25 08:54:29 [005] 006B8B457B
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [011] mmi.service
D: 20-09-25 08:54:29 [004] echo
20-09-25 08:54:29 I: received message:                   --> [echo] ticlient -> broker
```

```
D: 20-09-25 08:54:29 [005] 006B8B457B
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [004] echo
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: sending REQUEST to worker            --> [REQUEST] broker -> mdworker
D: 20-09-25 08:54:29 [005] 006B8B456D
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 02
D: 20-09-25 08:54:29 [005] 006B8B457B
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: received message:                    --> [REPLY] mdworker -> broker
D: 20-09-25 08:54:29 [005] 006B8B456D
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 03
D: 20-09-25 08:54:29 [005] 006B8B457B
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: received message:                --> [titanic.reply] ticlient -> broker
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [013] titanic.reply
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
20-09-25 08:54:29 I: sending REQUEST to worker         --> [REQUEST] broker -> mdworker
D: 20-09-25 08:54:29 [005] 006B8B456A
```

```
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 02
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
20-09-25 08:54:29 I: received message:                 --> [REPLY] mdworker -> broker
D: 20-09-25 08:54:29 [005] 006B8B456A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 03
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [003] 200
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: received message:                --> [titanic.close] ticlient -> broker
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [013] titanic.close
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
20-09-25 08:54:29 I: sending REQUEST to worker       --> [REQUEST] broker -> mdworker
D: 20-09-25 08:54:29 [005] 006B8B4568
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 02
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
```

```
20-09-25 08:54:29 I: received message:                --> [REPLY] mdworker -> broker
D: 20-09-25 08:54:29 [005] 006B8B4568
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 03
D: 20-09-25 08:54:29 [005] 006B8B457A
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [003] 200

[zedo@sook C]$ ./mdworker -v
20-09-25 08:54:29 I: received message from broker:
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 02
D: 20-09-25 08:54:29 [005] 006B8B457B
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: sending REPLY to broker
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [006] MDPW01
D: 20-09-25 08:54:29 [001] 03
D: 20-09-25 08:54:29 [005] 006B8B457B
D: 20-09-25 08:54:29 [000]
D: 20-09-25 08:54:29 [011] Hello world

[zedo@sook C]$ ./titanic -v
20-09-25 08:54:29 I: received message:
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
I: processing request 79F1E824D1364A2F8D003A8D5868C1BC
```

```
[zedo@sook C]$ ./ticlient -v
20-09-25 08:54:29 I: connecting to broker at tcp://localhost:5555...
20-09-25 08:54:29 I: send request to 'titanic.request' service:
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [015] titanic.request
D: 20-09-25 08:54:29 [004] echo
D: 20-09-25 08:54:29 [011] Hello world
20-09-25 08:54:29 I: received reply:
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [015] titanic.request
D: 20-09-25 08:54:29 [003] 200
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
D: 20-09-25 08:54:29 I: request UUID [032] 79F1E824D1364A2F8D003A8D5868C1BC
20-09-25 08:54:29 I: send request to 'titanic.reply' service:
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [013] titanic.reply
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
20-09-25 08:54:29 I: received reply:
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [013] titanic.reply
D: 20-09-25 08:54:29 [003] 200
D: 20-09-25 08:54:29 [011] Hello world
Reply: Hello world
20-09-25 08:54:29 I: send request to 'titanic.close' service:
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [013] titanic.close
D: 20-09-25 08:54:29 [032] 79F1E824D1364A2F8D003A8D5868C1BC
20-09-25 08:54:29 I: received reply:
```

```
D: 20-09-25 08:54:29 [006] MDPC01
D: 20-09-25 08:54:29 [013] titanic.close
D: 20-09-25 08:54:29 [003] 200
```

테스트를 위해 mdbroker 및 titanic을 시작한 다음, ticlient를 실행합니다. 그리고 mdworker를 시작하면 클라이언트(ticlient)가 응답을 받고 행복하게 종료되는 것을 볼 수 있습니다.

코드상에 주목할 점은 다음과 같습니다.

- 일부 루프는 전송으로 시작되고, 다른 루프는 메시지 수신으로 시작됩니다. 이는 타이타닉이 서로 다른 역할들로 클라이언트와 작업자 역할을 수행하기 때문입니다.
- 타이타닉 브로커는 MMI 서비스 검색 통신규약을 사용하여 실행 중인 서비스들에게만 요청을 보냅니다. 우리의 작은 MDP 브로커의 MMI 구현은 매우 간소화되어 항상 작동하지는 않을 겁니다.
- inproc 연결(`request_pipe`)을 통해 titanic.request 서비스로부터 메인 디스패처로 새로운 요청 데이터를 보냅니다. 이렇게 하면 디스패처가 디스크 디렉터리를 읽고, 모든 요청 파일을 메모리에 적제하고, 파일을 날짜/시간별로 정렬할 필요가 없습니다.

이 예제에서 중요한 것은 성능이 아니라(테스트하지는 않았지만 확실히 나쁠 것으로 예상됨) 신뢰성 있는 계약서를 얼마나 잘 구현하는 것입니다. mdbroker 및 titanic 프로그램을 시작하고, ticlient를 시작한 다음, mdworker의 echo 서비스를 시작합니다. 4개의 프로그램에서 "-v" 옵션을 사용하여 실행하며 자세한 활동 추적할 수 있습니다. 언제든 클라이언트를 제외한 모든 부분을 중지하고 재시작할 수 있으며 메시지들의 손실은 없습니다.

실제 사례에서 타이타닉을 사용하고 싶다면 "성능을 개선하려면 어떻게 하나요?"라는 질문에 대응해야 합니다.

성능을 개선하기 위해 제안하는 구현 방안은 다음과 같습니다.

- 모든 데이터에 대하여 여러 파일들이 아닌 단일 파일을 사용하십시오. 운영체제는 일반적으로 많은 작은 파일들보다 몇 개의 큰 파일을 처리하는 것이 성능을 개선합니다.
- 새로운 요청이 연속적으로 쓰기 가능하도록 디스크 파일을 순환 버퍼로 구성합니다(순환이 완료 시, 겹치는 부분 덮어씀). 하나의 스레드로 디스크 파일에 최대 속도로 쓰고, 빠르게 동작할 수 있습니다.

- 메모리에 인덱스를 유지하고, 시작 시 디스크 버퍼에서 인덱스를 다시 생성합니다. 이렇게 하면 인덱스를 디스크에서 완전히 안전하게 유지하는 데 필요한 부가적인 하드디스크 접근을 줄입니다. 모든 메시지에 대하여 N 미리초(milliseconds) 단위로 `fsync()`로 메모리와 디스크 간의 동기화를 수행하지 않으면 시스템 장애 시 마지막 M개의 메시지를 유실할 수 있습니다.
- 하드디스크 대신에 메모리 디스크를 사용하십시오
- 전체 파일을 미리 할당하거나, 큰 덩어리로 할당하여 필요에 따라 순환 버퍼를 늘리거나 줄일 수 있습니다. 이렇게 하면 디스크 단편화가 방지되고 대부분의 읽기 및 쓰기가 연속적으로 이루어집니다.

빠른 키/값 저장소(해시 테이블)가 아닌 데이터베이스에 메시지들을 저장하는 것은 추천하지 않습니다. 데이터베이스를 통한 추상화의 대가는 원시 디스크 파일에 비해 10배에서 수천 배 성능 저하를 불러 옵니다.

타이타닉을 더욱 안정적으로 만들고 싶다면, 요청들을 백업 서버에 복제하십시오. 백업 서버는 현재 서버가 위치한 장소가 핵공격을 받을 경우 살아남을 수 있도록 멀리 떨어진 위치에 배치해야 하지만 복제를 위해 너무 많은 지연시간이 발생해서는 안됩니다.

타이타닉을 보다 빠르고 안정적으로 만들고 싶다면, 요청들과 응답들은 메모리에 저장하십시오. 이러면 연결이 끊어진 네트워크 상황에서도 지속성이 유지되지만, 타이타닉 서버 자체의 장애 시에는 메시지 유실이 발생할 수 있습니다.

0.51 고가용성 쌍 (바이너리 스타 패턴)

그림 52 - 고가용성 쌍, 정상 동작

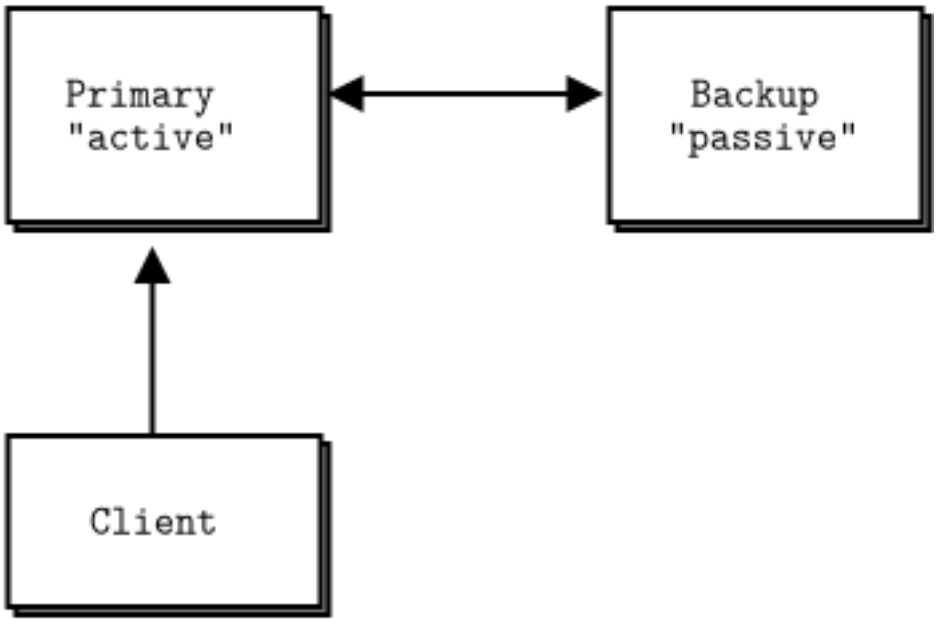


그림 54: High-Availability Pair, Normal Operation

바이너리 스타 패턴은 2개의 서버를 기본-백업으로 고가용성 쌍으로 배치합니다. 주어진 시간에 이들 중 하나(활성)는 클라이언트 응용프로그램의 연결을 수락합니다. 다른 하나(비활성)는 아무것도 하지 않지만, 2대의 서버들은 서로를 모니터링합니다. 활성 서버가 네트워크에서 사라지고 일정 시간이 지나면 비활성 서버가 활성 서버의 역할을 수행합니다.

우리는 OpenAMQ 서버를 위해 iMatix 사에서 바이너리 스타 패턴을 개발했으며, 다음과 같은 목표로 설계하였습니다.

- 똑바른 고가용성 솔루션을 제공합니다.
- 충분히 이해하고 사용할 만큼 단순해야 합니다.
- 필요할 때만 안정적으로 장애조치를 수행합니다.

바이너리 스타 쌍이 실행 중이라고 가정하고, 다음은 장애조치가 발생하는 여러 다른 시나리오입니다.

- 기본 서버로 실행되는 하드웨어에 치명적인 문제(전원 공급 장치가 고장 나거나, 기계에

불이 붙거나, 누군가가 실수로 전원을 뽑은 경우)가 발생하여 사라집니다. 응용프로그램은 서버가 사라짐을 확인하고 백업 서버에 연결합니다.

- 기본 서버가 있는 네트워크 영역에 장애(아마도 라우터에 순간 전압 상승으로 인한 기계적인 결함)가 발생하여 응용프로그램은 백업 서버로 연결하기 시작합니다.
- 기본 서버가 중지하거나 운영자에 의해 종료되어도 자동으로 재시작되지 않습니다.

그림 53 -장애 시 고가용성 쌍

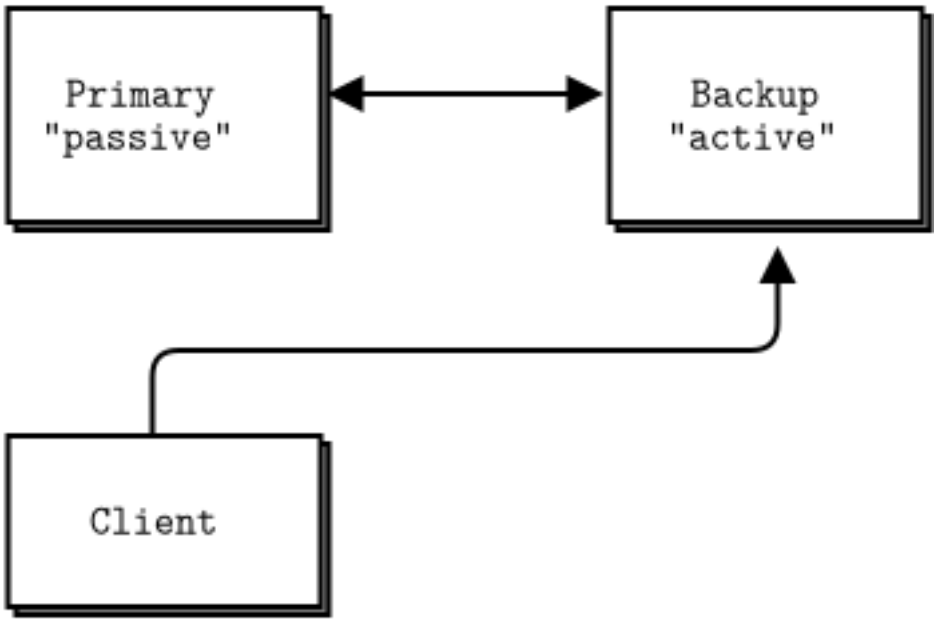


그림 55: High-availability Pair During Failover

장애 복구 절차는 다음과 같습니다.

1. 운영자는 기본 서버를 다시 시작하고 네트워크에서 사라지는 문제를 해결합니다.
2. 운영자는 백업 서버를 잠시 중지함으로 응용프로그램에 최소한의 중단을 수행합니다.
3. 응용프로그램이 기본 서버에 재연결되면, 운영자는 백업 서버를 다시 시작합니다.

복구(기본 서버를 활성으로 사용)는 수동으로 이루어지는 작업입니다. 자동 복구가 바람직하지 않다는 것을 고통스럽게 경험하였으며 여러 가지의 이유가 있습니다.

- 장애조치는 응용프로그램에 대한 서비스 중단을 발생하여, 10-30초 동안 지속 가능합니다. 실제 긴급 상황이라면 전체 중단되는 것보다는 낫습니다. 그러나 복구로 인하여 추가적으로 10~30초가 중단이 발생하면, 사용자가 네트워크를 사용하지 않거나 사용량이 적을 때 하는 것이 좋습니다.
- 긴급 상황이 발생하면. 문제 해결이 최우선 과제입니다. 자동 복구는 시스템 관리자에게 불확실성을 야기하며, 백업 서버에서도 동일한 문제가 발생 가능한지 확인해야 합니다.
- 네트워크가 장애조치 및 복구된 상황에서 자동 복구는 운영자가 발생한 상황을 분석하기 어렵게 합니다. 서비스가 중단되었지만 원인이 분명하지 않습니다.

그렇다고 해도 바이너리 스타 패턴은 기본 서버에 장애가 발생하면 백업 서버로 복구되며, 백업 서버에 장애 발생하면, (다시) 기본 서버로 복구되면서 사실상 자동 복구된 모습입니다.

바이너리 스타 쌍의 종료 프로세스는 다음 중 하나입니다.

- 비활성(백업) 서버를 중지한 다음 활성(기본) 서버를 중지하거나
- 순서에 관계없이 2대의 서버를 거의 동시에 중지하십시오.

활성(기본) 서버를 중지한 다음 비활성(백업) 서버를 중지 시, 장애조치 시간보다 지연되면 응용프로그램의 활성 서버에 연결이 해제 후 비활성 서버로 재연결되었다가, 비활성 서버가 중단되면 다시 연결 해제되어 사용자들에게 혼란을 야기합니다.

0.51.1 상세 요구사항

바이너리 스타는 단순하며 정확하게 작동합니다. 실제로 현재 설계는 세 번째 개정판입니다. 이전의 설계들은 너무 복잡하고 너무 많은 작업을 시도하여, 이해하고 사용하기 쉽게 그리고 충분히 신뢰 가능하도록 기능을 보완하였습니다.

고가용성 아키텍처에 대한 요구사항들은 다음과 같습니다.

- 장애조치는 치명적인 시스템 장애에 대한 보험을 제공하기 위함이며, 시스템 장애는 하드웨어 고장, 화재, 사고 등이 있습니다. 일반적인 서버 장애에서 복구하는 간단한 방법은 이미 다루었습니다.
- 장애조치 시간은 60초 미만이며 10초 이하가 바람직합니다.

- 장애조치는 자동으로 이루어지지만 복구는 수동으로 이루어져야 합니다. 응용프로그램이 백업 서버로 자동 전환되기를 바라지만, 운영자가 문제를 해결하고 응용프로그램을 다시 중단하기로 결정한 경우를 제외하고는 응용프로그램이 스스로 기본 서버로 다시 전환되는 것을 원하지 않습니다.
- 클라이언트 응용프로그램에 대한 통신규약은 개발자가 이해하기 쉽고 간단해야 합니다. 이상적으로는 클라이언트 API에서 숨겨져야 합니다.
- 네트워크 설계자를 위한 명확한 지침들이 제공되어, 2대의 서버가 모두 활성 상태로 생각되는 정신분열증 회피해야 합니다.
- 2대의 서버가 시작되는 순서에 대한 종속성이 없어야 합니다.
- PM(Planed Maintance)를 위하여 클라이언트 응용프로그램을 운영이 가능한 상태로, 2대 서버들 중 하나를 중지하고 재시작할 수 있어야 합니다(다시 재연결 발생).
- 운영자는 항상 2대의 서버를 모니터링할 수 있어야 합니다.
- 고속 전용 네트워크 연결을 사용하여 2대의 서버를 연결해야 합니다. 즉, 장애조치 동기화는 고정 IP 경로를 사용해야 합니다.

다음 일들을 가정합니다.

- 단일 백업 서버로도 충분한 보험을 제공합니다. 다중의 백업은 고려하지 않습니다.
- 기본 및 백업 서버는 동일하게 응용프로그램 부하를 처리할 수 있어야 합니다. 서버들 간의 부하 분산은 하지 않습니다.
- 상시 백업 서버가 아무것도 하지 않아도 되게 충분한 예산이 있습니다.

다음 일들을 가정하지 않습니다.

- 활성 백업 서버 또는 부하 분산 사용 : 바이너리 스타 쌍에서 백업 서버는 비활성 상태이며 기본 서버가 오프라인이 될 때까지 작업을 수행하지 않습니다.
- 모든 방식의 지속성 메시지들 또는 트랜잭션들 처리 : 안정화되지 않은(그리고 아마 신뢰할 수 없는) 서버 또는 바이너리 스타 쌍의 네트워크의 존재를 가정합니다.
- 네트워크의 자동 탐색 : 바이너리 스타 쌍은 수동 및 명시적으로 네트워크에서 정의되며 응용프로그램에 알려져 있습니다 (적어도 응용프로그램 구성 데이터에서).

- 서버들 간의 상태 혹은 메시지 복제 : 모든 서버 측 상태는 장애조치 시 응용프로그램에서 재생성되어야 합니다.

바이너리 스타에서 사용하는 주요 용어입니다.

- 기본(Primary) : 일반적으로 또는 처음에 활성화된 서버.
- 백업(Backup) : 일반적으로 비활성 상태의 서버. 기본(Primary) 서버가 네트워크에서 사라지고 클라이언트 응용프로그램이 백업 서버에 연결을 요청할 때 활성화됩니다.
- 활성(Active) : 클라이언트 연결을 수락하는 서버. 활성 서버는 최대 하나입니다.
- 수동(Passive) : 활성화된 서버가 사라지면 인계받는 서버. 바이너리 스타 쌍이 정상적으로 실행 중이면, 기본 서버가 활성 상태이고 백업 서버가 비활성 상태입니다. 장애조치가 발생하면 역할이 바뀝니다.

바이너리 스타 쌍을 구성하기 위하여 필요한 것은 다음과 같습니다.

1. 기본 서버에게 백업 서버가 있는 곳을 알려줍니다.
2. 백업 서버에게 기본 서버가 있는 곳을 알려줍니다.
3. 선택적으로, 장애조치 응답 시간은 두 대의 서버에 동일하게 설정하십시오.

주요 튜닝 매개변수로 서버가 상대 서버의 상태를 확인하는 빈도와 얼마나 빨리 장애조치를 활성화하는 시간이 있습니다. 예제에서 장애조치 제한시간 값은 기본적으로 2초입니다. 이 값을 줄이면 백업 서버가 더 빠르게 활성화되지만 기본 서버가 복구되는 경우에도 인계될 수 있습니다. 예를 들어, 기본 서버가 충돌로 재시작되는 셸 스크립트가 있다면 장애조치 제한시간은 기본 서버로 재시작되는데 필요한 시간보다 높아야 합니다.

클라이언트 응용프로그램이 바이너리 스타 쌍으로 제대로 작동하려면 다음과 같이 구현되어야 합니다.

1. 양쪽 서버 주소를 알고 있습니다.
2. 기본 서버에 연결하고 실패하면 백업 서버에 연결합니다.
3. 연결 실패를 인지하며, 일반적으로 심박을 사용합니다.
4. 기본 서버에 재연결 시도 후 백업 서버(순서대로)로 진행하며, 재시도 간의 지연시간(예: 2.5초)은 서버 장애조치 제한시간(예: 2초) 보다 커야 합니다.

5. 서버에서 필요한 모든 상태를 재생성합니다.
6. 메시지들에 대한 신뢰성이 필요한 경우, 장애조치 시 유실된 메시지들을 재전송합니다.

구현하는 것은 쉽지 않으며 , 보통 최종 사용자 응용프로그램에서 상세 구현을 숨기는 API로 추상화합니다.

바이너리 스타 패턴의 주요 제한 사항은 다음과 같습니다.

- 하나의 서버 프로세스는 하나의 바이너리 스타 쌍의 일부가 됩니다.
- 기본 서버는 하나의 백업 서버만 가질 수 있습니다.
- 비활성 서버는 특별한 작업을 수행하지 않습니다.
- 백업 서버는 기본 서버에서 구동되는 전체 응용프로그램의 부하들을 처리할 수 있어야 합니다.
- 장애조치 구성은 응용프로그램 실행 중에는 변경 불가합니다.
- 클라이언트 응용프로그램은 장애조치에 대응하기 위한 기능을 가지고 있어야 합니다.

0.51.2 정신분열증 방지

정신분열증은 기본-백업 서버가 동시에 활성이라고 생각할 때 발생합니다. 이로 인해 응용프로그램들은 서로 인식하지 못해 종료되게 됩니다. 바이너리 스타에는 정신분열을 감지하고 제거하는 알고리즘이 있으며, 3개의 방향 결정 메커니즘을 기반합니다(서버는 클라이언트 응용프로그램 연결 요청을 받고 상대 서버를 인식할 수 없을 때까지 활성 상태가 되지 않습니다).

그러나 3개의 방향 결정 알고리즘은 (잘못) 설계된 네트워크로 인하여 오동작의 가능성도 있습니다. 일반적인 시나리오는 두 개 건물 사이에 분산된 바이너리 스타 쌍으로, 각 건물에도 일련의 응용프로그램이 있고 두 개의 건물 사이에 단일 네트워크 연결이 존재합니다. 네트워크 연결을 끊으면 두 개의 세트의 클라이언트 응용프로그램들이 생성되고, 바이너리 스타 쌍의 절반에 속해 있는 클라이언트 응용프로그램들은 각각의 서버에 대해 장애조치하여 2대의 서버가 활성화됩니다.

정신분열 상황을 방지하려면 바이너리 스타 쌍을 전용 네트워크 링크를 사용하여 연결해야 하며, 동일한 스위치에 두 대의 서버들을 연결하거나 두 대의 서버 간에 크로스오버 케이블로 직접 연결하는 것이 좀 더 좋습니다.

- [옮긴이] 크로스오버 케이블은 스위치(혹은 허브)를 사용하지 않고 두 장비를 직접 연결 가능하도록 설계된 이더넷 케이블

바이너리 스타 아키텍처를 각각 응용프로그램 세트로 존재할 수 있는 2개의 섬으로 나누지 말아야 합니다. 이것은 네트워크 아키텍처의 공통적인 유형으로, 이러한 경우 고가용성 장애조치가 아닌 연합(federation) 모델을 사용해야 합니다.

- [옮긴이] 페더레이션 모델은 다른 종류의 라우팅, 특히 부하 분산(load balancing)이나 라운드 로빈(round robin)보다는 서비스 이름 및 근접성에 따라 라우팅하는 서비스 지향 아키텍처(service-oriented architectures(SOAs))에 적합합니다.

적절한 편집증적인 네트워크 구성은 하나가 아닌 2개의 사설망으로 클러스터 연결을 사용합니다. 그리고 클러스터에 사용되는 네트워크 카드는 메시지 전송에 사용되는 네트워크 카드와 다르게 하며, 서버 하드웨어에서 다른 경로에 있습니다. 목표는 네트워크(데이터 통신)의 가능한 오류와 클러스터(장애조치)의 가능한 장애를 분리하는 것입니다. 네트워크 포트 장애는 상대적 높습니다.

0.51.3 바이너리 스타 구현

더 이상 고민하지 말고, 여기에 바이너리 스타 서버의 개념 구현(Proof-of-concept)을 하였으며, 기본 및 백업 서버는 동일한 코드로 역할(기본 혹은 백업)을 지정하여 실행합니다.

bstarsrv.c: 바이너리 스타 서버

```
//  Binary Star server proof-of-concept implementation. This server does no
//  real work; it just demonstrates the Binary Star failover model.

#include "czmq.h"

//  States we can be in at any point in time
typedef enum {
    STATE_PRIMARY = 1,          //  Primary, waiting for peer to connect
    STATE_BACKUP = 2,           //  Backup, waiting for peer to connect
```

```
    STATE_ACTIVE = 3,           //  Active - accepting connections
    STATE_PASSIVE = 4           //  Passive - not accepting connections
} state_t;

//  Events, which start with the states our peer can be in
typedef enum {
    PEER_PRIMARY = 1,           //  HA peer is pending primary
    PEER_BACKUP = 2,            //  HA peer is pending backup
    PEER_ACTIVE = 3,            //  HA peer is active
    PEER_PASSIVE = 4,           //  HA peer is passive
    CLIENT_REQUEST = 5          //  Client makes request
} event_t;

//  Our finite state machine
typedef struct {
    state_t state;              //  Current state
    event_t event;              //  Current event
    int64_t peer_expiry;        //  When peer is considered 'dead'
} bstar_t;

//  We send state information this often
//  If peer doesn't respond in two heartbeats, it is 'dead'
#define HEARTBEAT 1000          //  In msecs

//  .split Binary Star state machine
//  The heart of the Binary Star design is its finite-state machine (FSM).
//  The FSM runs one event at a time. We apply an event to the current state,
//  which checks if the event is accepted, and if so, sets a new state:
```

```
static bool
s_state_machine (bstar_t *fsm)
{
    bool exception = false;

    //  These are the PRIMARY and BACKUP states; we're waiting to become
    //  ACTIVE or PASSIVE depending on events we get from our peer:
    if (fsm->state == STATE_PRIMARY) {
        if (fsm->event == PEER_BACKUP) {
            printf ("I: connected to backup (passive), ready active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: connected to backup (active), ready passive\n");
            fsm->state = STATE_PASSIVE;
        }
        //  Accept client connections
    }
    else
    if (fsm->state == STATE_BACKUP) {
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: connected to primary (active), ready passive\n");
            fsm->state = STATE_PASSIVE;
        }
        else
        //  Reject client connections when acting as backup
        if (fsm->event == CLIENT_REQUEST)
            exception = true;
```

```
    }
    else
    //  .split active and passive states
    //  These are the ACTIVE and PASSIVE states:

    if (fsm->state == STATE_ACTIVE) {
        if (fsm->event == PEER_ACTIVE) {
            //  Two actives would mean split-brain
            printf ("E: fatal error - dual actives, aborting\n");
            exception = true;
        }
    }
    else
    //  Server is passive
    //  CLIENT_REQUEST events can trigger failover if peer looks dead
    if (fsm->state == STATE_PASSIVE) {
        if (fsm->event == PEER_PRIMARY) {
            //  Peer is restarting - become active, peer will go passive
            printf ("I: primary (passive) is restarting, ready active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_BACKUP) {
            //  Peer is restarting - become active, peer will go passive
            printf ("I: backup (passive) is restarting, ready active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_PASSIVE) {
```

```
            //  Two passives would mean cluster would be non-responsive
            printf ("E: fatal error - dual passives, aborting\n");
            exception = true;
        }
        else
        if (fsm->event == CLIENT_REQUEST) {
            //  Peer becomes active if timeout has passed
            //  It's the client request that triggers the failover
            assert (fsm->peer_expiry > 0);
            if (zclock_time () >= fsm->peer_expiry) {
                //  If peer is dead, switch to the active state
                printf ("I: failover successful, ready active\n");
                fsm->state = STATE_ACTIVE;
            }
            else
                //  If peer is alive, reject connections
                exception = true;
        }
    }
    return exception;
}

//  .split main task
//  This is our main task. First we bind/connect our sockets with our
//  peer and make sure we will get state messages correctly. We use
//  three sockets; one to publish state, one to subscribe to state, and
//  one for client requests/replies:

int main (int argc, char *argv [])
```

```
{
    //  Arguments can be either of:
    //      -p  primary server, at tcp://localhost:5001
    //      -b  backup server, at tcp://localhost:5002
    zctx_t *ctx = zctx_new ();
    void *statepub = zsocket_new (ctx, ZMQ_PUB);
    void *statesub = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (statesub, "");
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    bstar_t fsm = { 0 };

    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: Primary active, waiting for backup (passive)\n");
        zsocket_bind (frontend, "tcp://*:5001");
        zsocket_bind (statepub, "tcp://*:5003");
        zsocket_connect (statesub, "tcp://localhost:5004");
        fsm.state = STATE_PRIMARY;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: Backup passive, waiting for primary (active)\n");
        zsocket_bind (frontend, "tcp://*:5002");
        zsocket_bind (statepub, "tcp://*:5004");
        zsocket_connect (statesub, "tcp://localhost:5003");
        fsm.state = STATE_BACKUP;
    }
    else {
        printf ("Usage: bstarsrv { -p | -b }\n");
        zctx_destroy (&ctx);
```

```
        exit (0);
    }
    //  .split handling socket input
    //  We now process events on our two input sockets, and process these
    //  events one at a time via our finite-state machine. Our "work" for
    //  a client request is simply to echo it back:

    //  Set timer for next outgoing state message
    int64_t send_state_at = zclock_time () + HEARTBEAT;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { statesub, 0, ZMQ_POLLIN, 0 }
        };
        int time_left = (int) ((send_state_at - zclock_time ()));
        if (time_left < 0)
            time_left = 0;
        int rc = zmq_poll (items, 2, time_left * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Context has been shut down

        if (items [0].revents & ZMQ_POLLIN) {
            //  Have a client request
            zmsg_t *msg = zmsg_recv (frontend);
            fsm.event = CLIENT_REQUEST;
            if (s_state_machine (&fsm) == false)
                //  Answer client by echoing request back
                zmsg_send (&msg, frontend);
            else
```

```
                zmsg_destroy (&msg);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Have state from our peer, execute as event
            char *message = zstr_recv (statesub);
            fsm.event = atoi (message);
            free (message);
            if (s_state_machine (&fsm))
                break;          //  Error, so exit
            fsm.peer_expiry = zclock_time () + 2 * HEARTBEAT;
        }
        //  If we timed out, send state to peer
        if (zclock_time () >= send_state_at) {
            char message [2];
            sprintf (message, "%d", fsm.state);
            zstr_send (statepub, message);
            send_state_at = zclock_time () + HEARTBEAT;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    //  Shutdown sockets and context
    zctx_destroy (&ctx);
    return 0;
}
```

클라이언트 코드는 다음과 같습니다.

bstarcli.c: 바이너리 스타 클라이언트

```
//  Binary Star client proof-of-concept implementation. This client does no
//  real work; it just demonstrates the Binary Star failover model.

#include "czmq.h"
#include "zhelpers.h"
#define REQUEST_TIMEOUT     1000    //  msecs
#define SETTLE_DELAY        2000    //  Before failing over

int main (void)
{
    zctx_t *ctx = zctx_new ();

    char *server [] = { "tcp://localhost:5001", "tcp://localhost:5002" };
    uint server_nbr = 0;

    printf ("I: connecting to server at %s...\n", server [server_nbr]);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, server [server_nbr]);

    int sequence = 0;
    while (!zctx_interrupted) {
        //  We send a request, then we work to get a reply
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            //  Poll socket for a reply, with timeout
```

```
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  Interrupted

            //  .split main body of client
            //  We use a Lazy Pirate strategy in the client. If there's no
            //  reply within our timeout, we close the socket and try again.
            //  In Binary Star, it's the client vote that decides which
            //  server is primary; the client must therefore try to connect
            //  to each server in turn:

            if (items [0].revents & ZMQ_POLLIN) {
                //  We got a reply from the server, must match sequence
                char *reply = zstr_recv (client);
                if (atoi (reply) == sequence) {
                    printf ("I: server replied OK (%s)\n", reply);
                    expect_reply = 0;
                    s_sleep (1000);  //  One request per second
                }
                else
                    printf ("E: bad reply from server: %s\n", reply);
                free (reply);
            }
            else {
                printf ("W: no response from server, failing over\n");

                //  Old socket is confused; close it and open a new one
                zsocket_destroy (ctx, client);
```

```
                server_nbr = (server_nbr + 1) % 2;
                zclock_sleep (SETTLE_DELAY);
                printf ("I: connecting to server at %s...\n",
                        server [server_nbr]);
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, server [server_nbr]);

                //  Send request again, on new socket
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

바이너리 스타의 테스트를 위하여 서버들과 클라이언트를 시작합니다. 시작하는 순서는 어느 쪽이 먼저라도 상관없습니다.

```
bstarsrv -p     # Start primary
bstarsrv -b     # Start backup
bstarcli
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc bstarsrv.c libzmq.lib czmq.lib
 cl -EHsc bstarcli.c libzmq.lib czmq.lib

 ./bstarsrv -p
I: Primary active, waiting for backup (passive)
I: connected to backup (passive), ready active
```

```
W: interrupted


 ./bstarsrv -b
I: Backup passive, waiting for primary (active)
I: connected to primary (active), ready passive
I: failover successful, ready active


 ./bstarcli
I: connecting to server at tcp://localhost:5001...
I: server replied OK (1)
I: server replied OK (2)
I: server replied OK (3)
I: server replied OK (4)
I: server replied OK (5)
W: no response from server, failing over
I: connecting to server at tcp://localhost:5002...
I: server replied OK (6)
I: server replied OK (7)
...
```

프로그램 실행 후에, 기본 서버를 종료하여 장애조치를 유발하고, 백업을 종료하여 기본 서버를 다시 시작함으로 복구할 수 있습니다. 장애조치 및 복구에 대응 및 반응하는 클라이언트에 주의해 주세요.

바이너리 스타는 유한 상태 머신에 의해 구동됩니다. 발생하는 이벤트들은 상대 상태이며 "Peer Active"은 다른 서버가 활성 상태라는 것을 알려 줍니다. "Client Request"은 서버가 클라이언트 요청을 받았음을 의미합니다. "Client Vote"는 비활성 상태의 서버가 클라이언트 요청을 받아 활성화 상태로 전환되거나, 상대가 2개의 심박을 받는 동안 비활성 상태임을 의미합니다.

서버는 상태 변경을 위해 PUB-SUB 소켓을 사용합니다. 다른 소켓 조합은 여기서 동작하지 않습니다. PUSH 및 DEALER 소켓은 메시지 수신 준비가 된 상대가 없는 경우 차단해

버립니다. PAIR 소켓은 상대가 사라지고 복귀하면 다시 연결되지 않습니다. ROUTER 소켓은 메시지를 보내기 전에 상대의 주소가 필요합니다.

그림 54 - 바이너리 스타 유한 상태 머신

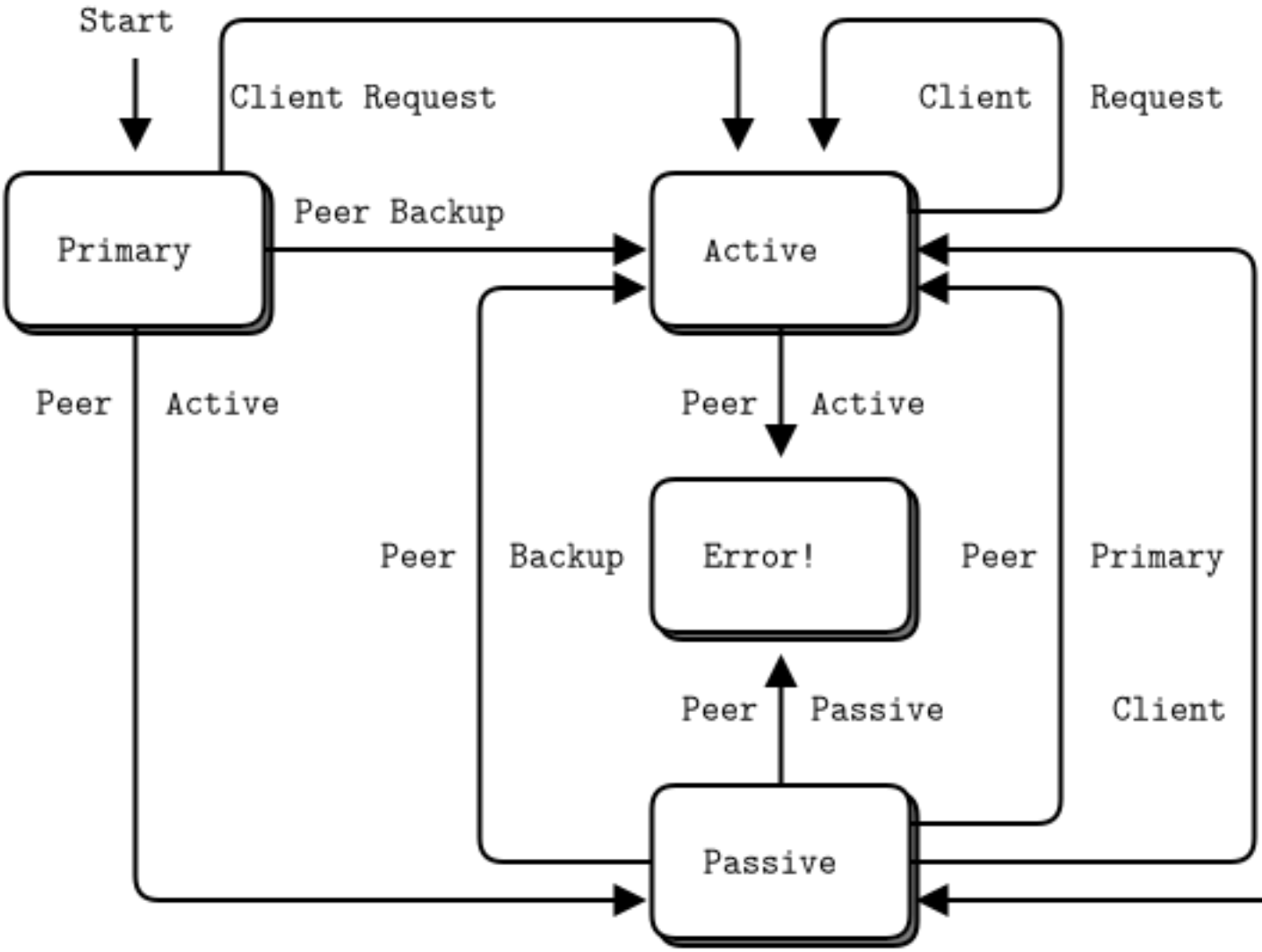


그림 56: Binary Star Finite State Machine

0.51.4 바이너리 스타 리엑터

바이너리 스타는 재사용 가능한 리엑터(reactor) 클래스로 만들면 유용하고 범용적입니다. 리엑터는 처리할 메시지가 있을 때마다 코드를 실행하고 호출합니다. 각 서버에 기능이 필요할 때마다 바이너리 스타 코드를 복사/붙여 넣기 하는 것보다 리엑터 API를 호출하는 것이 쉽습니다.

C 개발언어에서는 CZMQ 라이브러리의 zloop 클래스를 사용합니다. zloop는 소켓 및 타이머 이벤트에 반응하는 핸들러를 등록할 수 있습니다. 바이너리 스타 리엑터에서 클라이언트 요청(CLIENT_REQUEST)과 이벤트에 따른 상태 변경(ACTIVE -> PASSIVE 등)을 위한 핸들러를 제공합니다. 다음은 바이너리 스타 리엑터를 위한 API(bstar API) 입니다.

bstar.c: 바이너리 스타 핵심 클래스

```
//  bstar class - Binary Star reactor

#include "bstar.h"

//  States we can be in at any point in time
typedef enum {
    STATE_PRIMARY = 1,          //  Primary, waiting for peer to connect
    STATE_BACKUP = 2,           //  Backup, waiting for peer to connect
    STATE_ACTIVE = 3,           //  Active - accepting connections
    STATE_PASSIVE = 4           //  Passive - not accepting connections
} state_t;

//  Events, which start with the states our peer can be in
typedef enum {
    PEER_PRIMARY = 1,           //  HA peer is pending primary
    PEER_BACKUP = 2,            //  HA peer is pending backup
    PEER_ACTIVE = 3,            //  HA peer is active
    PEER_PASSIVE = 4,           //  HA peer is passive
```

```
    CLIENT_REQUEST = 5          //  Client makes request
} event_t;

//  Structure of our class

struct _bstar_t {
    zctx_t *ctx;                //  Our private context
    zloop_t *loop;              //  Reactor loop
    void *statepub;             //  State publisher
    void *statesub;             //  State subscriber
    state_t state;              //  Current state
    event_t event;              //  Current event
    int64_t peer_expiry;        //  When peer is considered 'dead'
    zloop_fn *voter_fn;         //  Voting socket handler
    void *voter_arg;            //  Arguments for voting handler
    zloop_fn *active_fn;        //  Call when become active
    void *active_arg;           //  Arguments for handler
    zloop_fn *passive_fn;         //  Call when become passive
    void *passive_arg;            //  Arguments for handler
};

//  The finite-state machine is the same as in the proof-of-concept server.
//  To understand this reactor in detail, first read the CZMQ zloop class.
//  .skip

//  We send state information every this often
//  If peer doesn't respond in two heartbeats, it is 'dead'
#define BSTAR_HEARTBEAT     1000        //  In msecs
```

```
//  Binary Star finite state machine (applies event to state)
//  Returns -1 if there was an exception, 0 if event was valid.

static int
s_execute_fsm (bstar_t *self)
{
    int rc = 0;
    //  Primary server is waiting for peer to connect
    //  Accepts CLIENT_REQUEST events in this state
    if (self->state == STATE_PRIMARY) {
        if (self->event == PEER_BACKUP) {
            zclock_log ("I: connected to backup (passive), ready as active");
            self->state = STATE_ACTIVE;
            if (self->active_fn)
                (self->active_fn) (self->loop, NULL, self->active_arg);
        }
        else
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: connected to backup (active), ready as passive");
            self->state = STATE_PASSIVE;
            if (self->passive_fn)
                (self->passive_fn) (self->loop, NULL, self->passive_arg);
        }
        else
        if (self->event == CLIENT_REQUEST) {
            // Allow client requests to turn us into the active if we've
            // waited sufficiently long to believe the backup is not
            // currently acting as active (i.e., after a failover)
            assert (self->peer_expiry > 0);
```

```
            if (zclock_time () >= self->peer_expiry) {
                zclock_log ("I: request from client, ready as active");
                self->state = STATE_ACTIVE;
                if (self->active_fn)
                    (self->active_fn) (self->loop, NULL, self->active_arg);
            } else
                // Don't respond to clients yet - it's possible we're
                // performing a failback and the backup is currently active
                rc = -1;
        }
    }
    else
    //  Backup server is waiting for peer to connect
    //  Rejects CLIENT_REQUEST events in this state
    if (self->state == STATE_BACKUP) {
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: connected to primary (active), ready as passive");
            self->state = STATE_PASSIVE;
            if (self->passive_fn)
                (self->passive_fn) (self->loop, NULL, self->passive_arg);
        }
        else
        if (self->event == CLIENT_REQUEST)
            rc = -1;
    }
    else
    //  Server is active
    //  Accepts CLIENT_REQUEST events in this state
    //  The only way out of ACTIVE is death
```

```
    if (self->state == STATE_ACTIVE) {
        if (self->event == PEER_ACTIVE) {
            //  Two actives would mean split-brain
            zclock_log ("E: fatal error - dual actives, aborting");
            rc = -1;
        }
    }
    else
    //  Server is passive
    //  CLIENT_REQUEST events can trigger failover if peer looks dead
    if (self->state == STATE_PASSIVE) {
        if (self->event == PEER_PRIMARY) {
            //  Peer is restarting - become active, peer will go passive
            zclock_log ("I: primary (passive) is restarting, ready as active");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_BACKUP) {
            //  Peer is restarting - become active, peer will go passive
            zclock_log ("I: backup (passive) is restarting, ready as active");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_PASSIVE) {
            //  Two passives would mean cluster would be non-responsive
            zclock_log ("E: fatal error - dual passives, aborting");
            rc = -1;
        }
        else
```

```
        if (self->event == CLIENT_REQUEST) {
            //  Peer becomes active if timeout has passed
            //  It's the client request that triggers the failover
            assert (self->peer_expiry > 0);
            if (zclock_time () >= self->peer_expiry) {
                //  If peer is dead, switch to the active state
                zclock_log ("I: failover successful, ready as active");
                self->state = STATE_ACTIVE;
            }
            else
                //  If peer is alive, reject connections
                rc = -1;
        }
        //  Call state change handler if necessary
        if (self->state == STATE_ACTIVE && self->active_fn)
            (self->active_fn) (self->loop, NULL, self->active_arg);
    }
    return rc;
}

static void
s_update_peer_expiry (bstar_t *self)
{
    self->peer_expiry = zclock_time () + 2 * BSTAR_HEARTBEAT;
}

//  Reactor event handlers...

//  Publish our state to peer
```

```
int s_send_state (zloop_t *loop, int timer_id, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    zstr_sendf (self->statepub, "%d", self->state);
    return 0;
}

//  Receive state from peer, execute finite state machine
int s_recv_state (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    char *state = zstr_recv (poller->socket);
    if (state) {
        self->event = atoi (state);
        s_update_peer_expiry (self);
        free (state);
    }
    return s_execute_fsm (self);
}

//  Application wants to speak to us, see if it's possible
int s_voter_ready (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    //  If server can accept input now, call appl handler
    self->event = CLIENT_REQUEST;
    if (s_execute_fsm (self) == 0)
        (self->voter_fn) (self->loop, poller, self->voter_arg);
    else {
```

```
        //  Destroy waiting message, no-one to read it
        zmsg_t *msg = zmsg_recv (poller->socket);
        zmsg_destroy (&msg);
    }
    return 0;
}

//  .until
//  .split constructor
//  This is the constructor for our {{bstar}} class. We have to tell it
//  whether we're primary or backup server, as well as our local and
//  remote endpoints to bind and connect to:

bstar_t *
bstar_new (int primary, char *local, char *remote)
{
    bstar_t
        *self;

    self = (bstar_t *) zmalloc (sizeof (bstar_t));

    //  Initialize the Binary Star
    self->ctx = zctx_new ();
    self->loop = zloop_new ();
    self->state = primary? STATE_PRIMARY: STATE_BACKUP;

    //  Create publisher for state going to peer
    self->statepub = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->statepub, local);
```

```
    //  Create subscriber for state coming from peer
    self->statesub = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_set_subscribe (self->statesub, "");
    zsocket_connect (self->statesub, remote);

    //  Set-up basic reactor events
    zloop_timer (self->loop, BSTAR_HEARTBEAT, 0, s_send_state, self);
    zmq_pollitem_t poller = { self->statesub, 0, ZMQ_POLLIN };
    zloop_poller (self->loop, &poller, s_recv_state, self);
    return self;
}

//  .split destructor
//  The destructor shuts down the bstar reactor:

void
bstar_destroy (bstar_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        bstar_t *self = *self_p;
        zloop_destroy (&self->loop);
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  .split zloop method
//  This method returns the underlying zloop reactor, so we can add
//  additional timers and readers:

zloop_t *
bstar_zloop (bstar_t *self)
{
    return self->loop;
}

//  .split voter method
//  This method registers a client voter socket. Messages received
//  on this socket provide the CLIENT_REQUEST events for the Binary Star
//  FSM and are passed to the provided application handler. We require
//  exactly one voter per {{bstar}} instance:

int
bstar_voter (bstar_t *self, char *endpoint, int type, zloop_fn handler,
             void *arg)
{
    //  Hold actual handler+arg so we can call this later
    void *socket = zsocket_new (self->ctx, type);
    zsocket_bind (socket, endpoint);
    assert (!self->voter_fn);
    self->voter_fn = handler;
    self->voter_arg = arg;
    zmq_pollitem_t poller = { socket, 0, ZMQ_POLLIN };
    return zloop_poller (self->loop, &poller, s_voter_ready, self);
}
```

```
//  .split register state-change handlers
//  Register handlers to be called each time there's a state change:

void
bstar_new_active (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->active_fn);
    self->active_fn = handler;
    self->active_arg = arg;
}

void
bstar_new_passive (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->passive_fn);
    self->passive_fn = handler;
    self->passive_arg = arg;
}

//  .split enable/disable tracing
//  Enable/disable verbose tracing, for debugging:

void bstar_set_verbose (bstar_t *self, bool verbose)
{
    zloop_set_verbose (self->loop, verbose);
}

//  .split start the reactor
```

```
//  Finally, start the configured reactor. It will end if any handler
//  returns -1 to the reactor, or if the process receives SIGINT or SIGTERM:

int
bstar_start (bstar_t *self)
{
    assert (self->voter_fn);
    s_update_peer_expiry (self);
    return zloop_start (self->loop);
}
```

추상화된 API를 통하여 바이너리 스타 서버는 짧은 코드로 작성 가능합니다.

bstarsrv2.c: 바이너리 스타 서버 (핵심 클래스 (bstar) 사용)

```
//  Binary Star server, using bstar reactor

//  Lets us build this source without creating a library
#include "bstar.c"

//  Echo service
int s_echo (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    zmsg_t *msg = zmsg_recv (poller->socket);
    zmsg_send (&msg, poller->socket);
    return 0;
}

int main (int argc, char *argv [])
{
    //  Arguments can be either of:
```

```
    //      -p  primary server, at tcp://localhost:5001
    //      -b  backup server, at tcp://localhost:5002
    bstar_t *bstar;
    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: Primary active, waiting for backup (passive)\n");
        bstar = bstar_new (BSTAR_PRIMARY,
            "tcp://*:5003", "tcp://localhost:5004");
        bstar_voter (bstar, "tcp://*:5001", ZMQ_ROUTER, s_echo, NULL);
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: Backup passive, waiting for primary (active)\n");
        bstar = bstar_new (BSTAR_BACKUP,
            "tcp://*:5004", "tcp://localhost:5003");
        bstar_voter (bstar, "tcp://*:5002", ZMQ_ROUTER, s_echo, NULL);
    }
    else {
        printf ("Usage: bstarsrvs { -p | -b }\n");
        exit (0);
    }
    bstar_start (bstar);
    bstar_destroy (&bstar);
    return 0;
}
```

- [옮긴이] assert()에 지정한 조건식이 거짓(false) 일 때 프로그램을 중단하며 참(true) 일 때는 프로그램이 계속 실행합니다.

- [옮긴이] 빌드 및 테스트

```
 ./bstarsrv2 -p
I: Primary active, waiting for backup (passive)
20-08-23 10:39:19 I: request from client, ready as active
 ./bstarsrv2 -p
I: Primary active, waiting for backup (passive)
20-08-23 10:39:38 I: connected to backup (passive), ready as active
 ./bstarsrv2 -p
I: Primary active, waiting for backup (passive)
20-08-23 10:39:58 I: connected to backup (active), ready as passive
20-08-23 10:40:26 I: failover successful, ready as active

 ./bstarsrv2 -b
I: Backup passive, waiting for primary (active)
20-08-23 10:39:38 I: connected to primary (active), ready as passive
20-08-23 10:39:48 I: failover successful, ready as active
 ./bstarsrv2 -b
I: Backup passive, waiting for primary (active)
20-08-23 10:40:38 I: connected to primary (active), ready as passive
20-08-23 10:40:48 I: failover successful, ready as active

 ./bstarcli
I: connecting to server at tcp://localhost:5001...
I: server replied OK (1)
I: server replied OK (2)
I: server replied OK (3)
I: server replied OK (4)
W: no response from server, failing over
I: connecting to server at tcp://localhost:5002...
I: server replied OK (5)
```

```
...
I: server replied OK (39)
W: no response from server, failing over
I: connecting to server at tcp://localhost:5001...
I: server replied OK (40)
...
I: server replied OK (58)
W: no response from server, failing over
I: connecting to server at tcp://localhost:5002...
I: server replied OK (59)
...
```

0.52 브로커 없는 신뢰성(프리랜서 패턴)

ØMQ를 "브로커 없는 메시징"이라고 설명하면서, 브로커 기반 신뢰성에 너무 집중하는 것은 모순처럼 보입니다. 메시징에서 실생활에서와 같이 중개인(middleman)은 부담이자 장점입니다. 실제로 대부분의 메시징 아키텍처는 분산과 중개 메시징을 혼합하여 각 메시징의 장점을 얻을 수 있습니다. 각 메시징의 장단점을 자유롭게 선택할 수 있을 때 최상의 결과를 얻을 수 있습니다. 마치 혼자서 저녁 식사를 위한 와인 1병을 사기 위해 10분 걸어서 편의점에 가는 것과, 친구들과의 파티를 위해 5종류의 와인을 사려 20분 정도 운전해서 홈플러스까지 가는 이유와 같습니다.
현실 세계 경제에 기본은 소요되는 시간, 에너지 및 비용에 민감하며 상대적 평가를 통해 선택되며, 최적의 메시지 기반 아키텍처에도 마찬가지입니다.

이것이 ØMQ가 브로커 중심 아키텍처를 강요하지 않는 이유입니다. ØMQ는 브로커(일명 프록시)를 구축할 수 있는 도구를 주어 지금까지 12개 정도 여러 패턴들을 구축했습니다.

- [옮긴이] 이전에 학습한 메세지 패턴을 다음과 같습니다.
- REQ-REP
- PUB-SUB

- REQ-ROUTER
- DEALER-REP
- DEALER-ROUTER
- DEALER-DEALER
- ROUTER-ROUTER
- PUSH-PULL
- PAIR-PAIR
- LPP : Lazy Pirate Pattern
- SPP : Simple Pirate Pattern
- PPP : Paranoid Pirate Pattern
- MDP : Majordomo Pattern

따라서 이장을 마무리하면서 우리는 지금까지 만들어온 브로커 기반 신뢰성을 해체하고 프리랜스 패턴이라고 불리는 분산형 P2P(peer-to-peer) 아키텍처로 돌아가겠습니다. P2P의 핵심 사용 사례는 이름 확인 서비스(name resolution service)입니다. 이것은 ØMQ 아키텍처의 공통적인 문제입니다 : 연결할 단말을 어떻게 아시나요? 코드상 하드 코딩된 TCP/IP 주소는 매우 취약하기 때문에 주소를 구성 파일에 넣게 되면 관리가 끔찍해집니다. 모든 PC 또는 휴대폰의 웹브라우저에서 "google.com"이 "74.125.230.82"라는 것을 알기 위해 구성 파일을 직접 만들어야 한다고 생각해 보십시오.

- [옮긴이] DNS(domain name system)는 네트워크에서 도메인이나 호스트 네임을 숫자로 되어 있는 IP 주소로 해석해 주는 TCP/IP 네트워크 서비스입니다.

여기서 구현하는 ØMQ 이름 서비스는 다음의 기능을 수행해야 합니다 :

- 논리적 이름으로 바인딩 단말과 연결 단말로 확인합니다. 현실적인 이름 서비스는 다중 바인딩 단말들을 제공하고 가능하면 다중 연결 단말들도 제공합니다.
- 다중 병렬 환경들(예 : "테스트", "생산")을 코드 수정 없이 관리할 수 있습니다.
- 신뢰성이 보장돼야 합니다. 이름 서비스를 사용할 수 없으면 응용프로그램이 네트워크에 연결할 수 없기 때문입니다.

서비스 지향 MDP 브로커 뒤에 이름 서비스를 배치하는 것은 현명한 생각이지만, 클라이언트가 직접 연결 가능한 이름 서비스 서버로 노출하는 것이 훨씬 간단합니다. 올바른 작업을 위하여 이름 서비스가 글로벌 네트워크 단말이 되기 위해서 하드 코딩된 소스코드 또는 구성 파일이 필요합니다.

그림 55 - 프리랜서 패턴

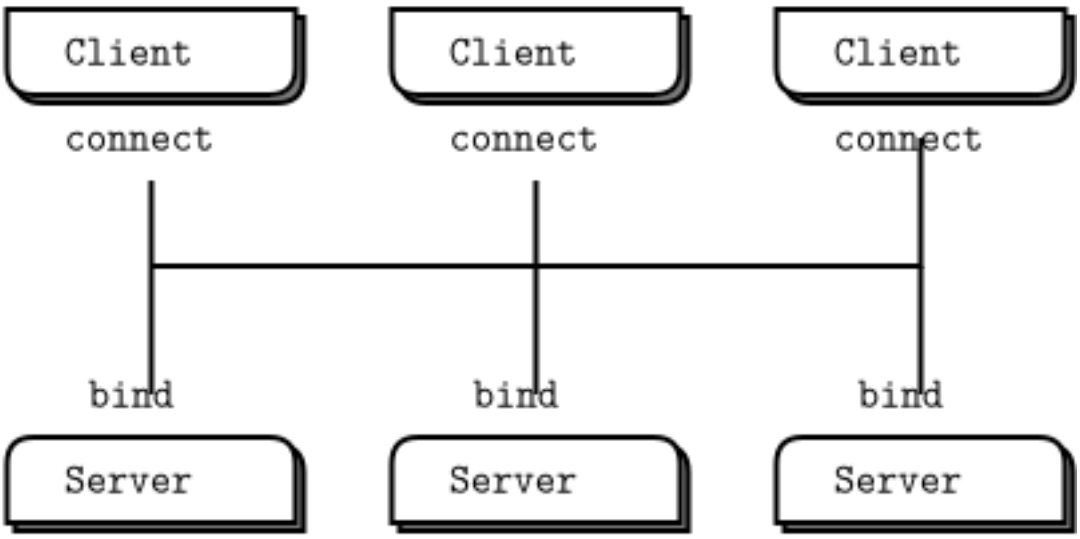


그림 57: The Freelance Pattern

처리하려는 장애 유형은 서버 충돌 및 재시작, 서버의 과도한 루프, 서버 과부하, 네트워크 문제입니다. 안정성을 확보하기 위해 이름 서버들의 저장소를 생성하여 서버 하나가 충돌하거나 사라지면, 클라이언트가 다른 서버에 연결하게 합니다. 실제로는 저장소은 2개면 충분하지만 예제에서는 저장소의 크기는 제한이 없다고 가정합니다.

- [옮긴이] 다중 서버 분산 아키텍처상에서 클라이언트 요청을 부하 분산하여 처리하기 위해 클라이언트에서 특정 서비스(서버) 접근을 지정할 경우 서비스를 제공하는 서버들 간에 부하 분산이 제대로 되지 않을 수도 있습니다. 부하 분산이 가능한 분산 처리를 클라이언트에서 구성할지 MDP와 같은 브로커를 통해 진행할지 고민이 필요합니다.

이 아키텍처에서는 대규모 집합의 클라이언트들이 소규모 서버들에 직접 연결됩니다. 서

버들은 각각의 주소로 바인딩합니다. 작업자가 브로커에 연결하는 MDP와 같은 브로커 기반 접근 방식과 근본적으로 다릅니다. 클라이언트에는 선택 가능한 몇 가지 옵션들이 있습니다.

- REQ 소켓과 게으른 해적 패턴(응답이 없을 경우 재시도)을 사용하십시오. 쉽지만 추가 기능으로 클라이언트들이 죽은 서버들에 계속해서 재연결을 하지 않게 하는 기능입니다.
- DEALER 소켓을 사용하고 응답을 받을 때까지 여러 개의 요청들을 비동기로 수행합니다(연결된 모든 서버들로 부하 분산됨). 효과적이지만 우아하지는 않습니다.
- 클라이언트가 특정 서버에 주소를 지정할 수 있도록 ROUTER 소켓을 사용합니다. 그러나 어떻게 클라이언트는 서버 소켓의 식별자(ID)를 인식할까요? 서버가 먼저 클라이언트를 핑(ping)하거나(복잡함), 서버가 하드 코딩된 고정 식별자(ID)를 가지고, 사전에 클라이언트가 인지하게 합니다.(서버들 구성이 변경(추가/삭제)될 때마다 클라이언트는 구성 정보 변경 필요하여 사용이 불편합니다.)

다음 절부터는 구현을 위한 개별 방법 설명하겠습니다.

0.52.1 모델 1: 간단한 재시도와 장애조치

클라이언트가 선택 가능한 3가지 방법들은 단순하고 무식하고 복잡하고 귀찮게 나타납니다. 우선 간단한 것부터 시작하여 꼬임을 해결해 봅시다. 게으른 해적 가져와서 다중 서버 단말들에 동작하도록 재작성합니다.

먼저 인수로 바인딩 할 단말을 지정하여 하나 이상의 서버을 시작하십시오.

flserver1.c: 프리랜서 서버, 모델 1

```
//  Freelance server - Model 1
//  Trivial echo service

#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
```

```
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        return 0;
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: echo service is ready at %s\n", argv [1]);
    while (true) {
        zmsg_t *msg = zmsg_recv (server);
        if (!msg)
            break;          //  Interrupted
        zmsg_send (&msg, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

그런 다음 하나 이상의 단말을 인수로 지정하여 클라이언트를 시작합니다.

flclient1.c: 프리랜서 클라이언트, 모델 1

```
//  Freelance client - Model 1
//  Uses REQ socket to query one or more services

#include "czmq.h"
#define REQUEST_TIMEOUT     1000
#define MAX_RETRIES         3       //  Before we abandon
```

```
static zmsg_t *
s_try_request (zctx_t *ctx, char *endpoint, zmsg_t *request)
{
    printf ("I: trying echo service at %s...\n", endpoint);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, endpoint);

    //  Send request, wait safely for reply
    zmsg_t *msg = zmsg_dup (request);
    zmsg_send (&msg, client);
    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
    zmsg_t *reply = NULL;
    if (items [0].revents & ZMQ_POLLIN)
        reply = zmsg_recv (client);

    //  Close socket in any case, we're done with it now
    zsocket_destroy (ctx, client);
    return reply;
}

//  .split client task
//  The client uses a Lazy Pirate strategy if it only has one server to talk
//  to. If it has two or more servers to talk to, it will try each server just
//  once:

int main (int argc, char *argv [])
{
    zctx_t *ctx = zctx_new ();
```

```
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = NULL;

    int endpoints = argc - 1;
    if (endpoints == 0)
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
    else
    if (endpoints == 1) {
        //  For one endpoint, we retry N times
        int retries;
        for (retries = 0; retries < MAX_RETRIES; retries++) {
            char *endpoint = argv [1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  Successful
            printf ("W: no response from %s, retrying...\n", endpoint);
        }
    }
    else {
        //  For multiple endpoints, try each at most once
        int endpoint_nbr;
        for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_nbr++) {
            char *endpoint = argv [endpoint_nbr + 1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  Successful
            printf ("W: no response from %s\n", endpoint);
        }
```

```
    }
    if (reply)
        printf ("Service is running OK\n");

    zmsg_destroy (&request);
    zmsg_destroy (&reply);
    zctx_destroy (&ctx);
    return 0;
}
```

예제는 다음과 같이 실행됩니다.

```
flserver1 tcp://*:5555 &
flserver1 tcp://*:5556 &
flclient1 tcp://localhost:5555 tcp://localhost:5556
```

- [옮긴이] 빌드 및 테스트

```
*** 1대의 서버에 대하여 테스트 수행하였을때
 ./flserver1 tcp://*:5559
I: echo service is ready at tcp://*:5559

 ./flclient1 tcp://localhost:5559
I: trying echo service at tcp://localhost:5559...
Service is running OK
 ./flclient1 tcp://localhost:5559

*** 서버가 모두 없고 클라이언트만 실행했을때
 ./flclient1 tcp://localhost:5559
I: trying echo service at tcp://localhost:5559...
W: no response from tcp://localhost:5559, retrying...
```

```
I: trying echo service at tcp://localhost:5559...
W: no response from tcp://localhost:5559, retrying...
I: trying echo service at tcp://localhost:5559...
W: no response from tcp://localhost:5559, retrying...

** 2대의 서버에 대하여 테스트 수행하였을 때
 ./flserver1 tcp://*:5559
I: echo service is ready at tcp://*:5559

 ./flserver1 tcp://*:5560
I: echo service is ready at tcp://*:5559

 ./flclient1 tcp://localhost:5559 tcp://localhost:5560
I: trying echo service at tcp://localhost:5559...
Service is running OK
```

- [옮긴이] 현재의 로직은 클라이언트는 여러 개의 서버들에 요청을 보내고 단 1개의 응답을 받으면 종료하도록 되어 있습니다. 서버들에 대하여 모두 응답을 받거나 혹은 제한 시간 만료에 따른 NULL 응답을 처리하고 싶다면 클라이언트(flclient1) main() 함수의 "break"를 변경해야 합니다.
- 서버들로부터 단 1개의 응답이 있을 경우 프로그램 종료

```
        //  For multiple endpoints, try each at most once
        int endpoint_nbr;
        for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_nbr++) {
            char *endpoint = argv [endpoint_nbr + 1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  Successful
            printf ("W: no response from %s\n", endpoint);
```

```
    }
```

- 서버들로부터 모든 응답을 처리하도록 클라이언트(flclient1) 수정한 경우

```
    //  For multiple endpoints, try each at most once
    int endpoint_nbr;
    for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_nbr++) {
        char *endpoint = argv [endpoint_nbr + 1];
        reply = s_try_request (ctx, endpoint, request);
        if (reply)
            continue;
        printf ("W: no response from %s\n", endpoint);
    }
```

```
    //  For multiple endpoints, try each at most once
    int endpoint_nbr;
    for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_nbr++) {
        char *endpoint = argv [endpoint_nbr + 1];
        reply = s_try_request (ctx, endpoint, request);
        if (reply)
            printf("receive reply from %s\n", endpoint);
        else
            printf ("W: no response from %s\n", endpoint);
    }
```

- [옮긴이] 수정된 클라이언트(flclient1)의 소스는 다음과 같습니다.

```
//  Freelance client - Model 1
//  Uses REQ socket to query one or more services

#include "czmq.h"
```

```
#define REQUEST_TIMEOUT     1000
#define MAX_RETRIES         3       //  Before we abandon

static zmsg_t *
s_try_request (zctx_t *ctx, char *endpoint, zmsg_t *request)
{
    printf ("I: trying echo service at %s...\n", endpoint);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, endpoint);

    //  Send request, wait safely for reply
    zmsg_t *msg = zmsg_dup (request);
    zmsg_send (&msg, client);
    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
    zmsg_t *reply = NULL;
    if (items [0].revents & ZMQ_POLLIN)
        reply = zmsg_recv (client);

    //  Close socket in any case, we're done with it now
    zsocket_destroy (ctx, client);
    return reply;
}

//  .split client task
//  The client uses a Lazy Pirate strategy if it only has one server to talk
//  to. If it has two or more servers to talk to, it will try each server just
//  once:
```

```
int main (int argc, char *argv [])
{
    zctx_t *ctx = zctx_new ();
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = NULL;

    int endpoints = argc - 1;
    if (endpoints == 0)
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
    else
    if (endpoints == 1) {
        //  For one endpoint, we retry N times
        int retries;
        for (retries = 0; retries < MAX_RETRIES; retries++) {
            char *endpoint = argv [1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                continue;           //  Successful
            printf ("W: no response from %s, retrying...\n", endpoint);
        }
    }
    else {
        //  For multiple endpoints, try each at most once
        int endpoint_nbr;
        for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_nbr++) {
            char *endpoint = argv [endpoint_nbr + 1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
```

```
                continue;
            printf ("W: no response from %s\n", endpoint);
        }
    }
    if (reply)
        printf ("Service is running OK\n");

    zmsg_destroy (&request);
    zmsg_destroy (&reply);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 ./flserver1 tcp://*:5600
I: echo service is ready at tcp://*:5600

 ./flserver1 tcp://*:5601
I: echo service is ready at tcp://*:5601

 ./flclient1 tcp://localhost:5600 tcp://localhost:5601
I: trying echo service at tcp://localhost:5600...
I: trying echo service at tcp://localhost:5601...
Service is running OK
```

기본 접근 방식은 게으른 해적이지만, 클라이언트는 하나의 성공적인 응답만 받는 것을 목표로 합니다. 클라이언트 프로그램은 단일 서버 혹은 여러 서버들을 실행하는지에 따라 2 가지 처리 방식이 있습니다.

- 단일 서버에서 클라이언트는 게으른 해적과 동일하게 수차례(3번) 재시도합니다.

- 여러 서버에서 클라이언트는 서버들의 응답을 받기 위해 최대 1번 시도합니다.

이는 게으른 해적의 주요 약점인 백업 또는 대체 서버로 장애조치할 수 없는 문제를 해결합니다.

그러나 설계에는 단점이 있습니다. 클라이언트에서 많은 소켓을 연결하고 기본 이름 서버가 죽으면 각 클라이언트는 고통스러운 제한시간으로 인한 지연을 경험합니다.

- [옮긴이] 현실에서 클라이언트가 가용한 모든 서버들에 동일한 작업 요청을 하고 응답을 기다리는 것은 서버 자원에 대한 낭비를 초래하고, 서버가 죽을 경우 제한시간 지연이 발생합니다.

0.52.2 모델 2: 잔인한 엽총 학살

두 번째 옵션으로 클라이언트를 DEALER 소켓을 사용하도록 전환하겠습니다. 여기서 우리의 목표는 특정 서버의 상태(죽고, 살고)에 관계없이 가능한 한 빨리 응답을 받는 것입니다. 클라이언트는 다음과 같은 접근 방식을 가집니다.

- 클라이언트가 준비가 되면, 모든 서버들에 연결합니다.
- 요청이 있을 때, 서버들의 대수만큼 요청을 보냅니다.
- 클라이언트는 첫 번째 응답을 기다렸다가 받아들입니다.
- 다른 서버들의 응답들은 무시합니다.

서버가 기동 중에 실제로 일어날 일은 ØMQ가 클라이언트의 요청들을 분배하여 각 서버들이 하나의 요청을 받고 하나의 응답을 보내도록 합니다. 서버가 오프라인이고 연결이 끊어지면, ØMQ는 다른 나머지 서버로 요청을 분배합니다. 따라서 서버는 경우에 따라 동일한 요청을 두 번 이상 받을 수 있습니다.

클라이언트는 여러 번의 응답들을 받지만 정확한 횟수로 응답되었는지를 보장할 수 없습니다. 요청들 및 응답들은 유실될 수 있습니다(예 : 요청을 처리하는 동안 서버가 죽는 경우).

따라서 요청들에 번호를 매겨서 요청 번호와 일치하지 않는 응답은 무시합니다. 모델 1(간단한 재시도와 장애조치) 서버는 에코 서버라서 작동하지만 우연은 필연이 아니기에 이해를 위한 좋은 기초가 아닙니다. 따라서 모델 2 서버를 만들어 요청에 대한 정확한 응답 번호와

“OK”로 반환하게 합니다. 메시지들은 2개의 부분으로 구성되어 있습니다 : 시퀀스 번호(응답 번호)와 본문(OK)

하나 이상의 서버들을 시작할 때, 바인딩할 단말을 지정합니다.

flserver2.c: 프리랜서 서버, 모델2

```
//  Freelance server - Model 2
//  Does some work, replies OK, with message sequencing

#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        return 0;
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: service is ready at %s\n", argv [1]);
    while (true) {
        zmsg_t *request = zmsg_recv (server);
        if (!request)
            break;          //  Interrupted
        //  Fail nastily if run against wrong client
        assert (zmsg_size (request) == 2);

        zframe_t *identity = zmsg_pop (request);
        zmsg_destroy (&request);
```

```
        zmsg_t *reply = zmsg_new ();
        zmsg_add (reply, identity);
        zmsg_addstr (reply, "OK");
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

클라이언트를 시작하며, 연결한 단말들을 인수로 지정합니다.

flclient2.c: 프리랜서 클라이언트, 모델2

```
//  Freelance client - Model 2
//  Uses DEALER socket to blast one or more services

#include "czmq.h"

//  We design our client API as a class, using the CZMQ style
#ifdef __cplusplus
extern "C" {
#endif

typedef struct _flclient_t flclient_t;
flclient_t *flclient_new (void);
void        flclient_destroy (flclient_t **self_p);
void        flclient_connect (flclient_t *self, char *endpoint);
zmsg_t     *flclient_request (flclient_t *self, zmsg_t **request_p);
```

```
#ifdef __cplusplus
}
#endif

//  If not a single service replies within this time, give up
#define GLOBAL_TIMEOUT 2500

int main (int argc, char *argv [])
{
    if (argc == 1) {
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
        return 0;
    }
    //  Create new freelance client object
    flclient_t *client = flclient_new ();

    //  Connect to each endpoint
    int argn;
    for (argn = 1; argn < argc; argn++)
        flclient_connect (client, argv [argn]);

    //  Send a bunch of name resolution 'requests', measure time
    int requests = 10000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flclient_request (client, &request);
```

```
        if (!reply) {
            printf ("E: name service not available, aborting\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("Average round trip cost: %d usec\n",
        (int) (zclock_time () - start) / 10);

    flclient_destroy (&client);
    return 0;
}

//  .split class implementation
//  Here is the {{flclient}} class implementation. Each instance has a
//  context, a DEALER socket it uses to talk to the servers, a counter
//  of how many servers it's connected to, and a request sequence number:

struct _flclient_t {
    zctx_t *ctx;        //  Our context wrapper
    void *socket;       //  DEALER socket talking to servers
    size_t servers;     //  How many servers we have connected to
    uint sequence;      //  Number of requests ever sent
};

//  Constructor

flclient_t *
flclient_new (void)
```

```
{
    flclient_t
        *self;

    self = (flclient_t *) zmalloc (sizeof (flclient_t));
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_DEALER);
    return self;
}

//  Destructor

void
flclient_destroy (flclient_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flclient_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  Connect to new server endpoint

void
flclient_connect (flclient_t *self, char *endpoint)
{
```

```
    assert (self);
    zsocket_connect (self->socket, endpoint);
    self->servers++;
}

//  .split request method
//  This method does the hard work. It sends a request to all
//  connected servers in parallel (for this to work, all connections
//  must be successful and completed by this time). It then waits
//  for a single successful reply, and returns that to the caller.
//  Any other replies are just dropped:

zmsg_t *
flclient_request (flclient_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);
    zmsg_t *request = *request_p;

    //  Prefix request with sequence number and empty envelope
    char sequence_text [10];
    sprintf (sequence_text, "%u", ++self->sequence);
    zmsg_pushstr (request, sequence_text);
    zmsg_pushstr (request, "");

    //  Blast the request to all connected servers
    int server;
    for (server = 0; server < self->servers; server++) {
        zmsg_t *msg = zmsg_dup (request);
```

```
        zmsg_send (&msg, self->socket);
    }
    //  Wait for a matching reply to arrive from anywhere
    //  Since we can poll several times, calculate each one
    zmsg_t *reply = NULL;
    uint64_t endtime = zclock_time () + GLOBAL_TIMEOUT;
    while (zclock_time () < endtime) {
        zmq_pollitem_t items [] = { { self->socket, 0, ZMQ_POLLIN, 0 } };
        zmq_poll (items, 1, (endtime - zclock_time ()) * ZMQ_POLL_MSEC);
        if (items [0].revents & ZMQ_POLLIN) {
            //  Reply is [empty][sequence][OK]
            reply = zmsg_recv (self->socket);
            assert (zmsg_size (reply) == 3);
            free (zmsg_popstr (reply));
            char *sequence = zmsg_popstr (reply);
            int sequence_nbr = atoi (sequence);
            free (sequence);
            if (sequence_nbr == self->sequence)
                break;
            zmsg_destroy (&reply);
        }
    }
    zmsg_destroy (request_p);
    return reply;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc flserver2.c libzmq.lib czmq.lib
 cl -EHsc flclient2.c libzmq.lib czmq.lib

*** 1대의 서버를 대상으로 한 경우
 ./flserver2 tcp://*:5560
I: service is ready at tcp://*:5560

 ./flclient2 tcp://localhost:5560
Average round trip cost: 151 usec

*** 2대의 서버를 대상으로 한 경우
 ./flserver2 tcp://*:5560
I: service is ready at tcp://*:5560
 ./flserver2 tcp://*:5561
I: service is ready at tcp://*:5561

 ./flclient2 tcp://localhost:5560 tcp://localhost:5561
Average round trip cost: 152 usec

*** 모든 서버가 죽은 경우
S D:\git_store\zguide-kr\examples\C> ./flclient2 tcp://localhost:5560 tcp://localhost:5561
E: name service not available, aborting
Average round trip cost: 250 usec
```

- [옮긴이] 클라이언트에서 1만 번의 요청-응답 처리 시간(round trip cost)에 대하여 모든 서버가 죽은 경우, 1만 번이 아닌 1회에 대한 요청-응답 처리 시간(round trip cost)이 산정됩니다. 즉 제한시간인 2.5초(2.5초/10,000=0.25 msec=250 usec)가 산정됩니다.

클라이언트 구현에서 주의할 사항들은 다음과 같습니다.

- 클라이언트는 작은 클래스 기반 API로 구성되어 ØMQ 컨텍스트와 소켓을 생성하고 서버와 통신하는 어려운 작업을 은폐하였습니다. 즉, 산탄총이 목표로 발사되어 무수한 산탄이 나가는 것을 클라이언트에서 서버로 "말하기"라고 할 수 있습니다.
- 클라이언트는 몇 초(2.5초) 내에 응답하는 서버를 찾지 못하면 추적을 포기합니다.
- 클라이언트는 DELAER 소켓을 사용하므로, 유효한 REP 봉투(empty delimiter + body)를 만들어야 합니다. 즉, 메시지 앞에 빈 메시지 프레임을 추가해야 합니다.

클라이언트는 10,000개의 이름 확인 요청들(서버가 딱히 아무것도 하지 않는 가짜 요청)을 수행하고 평균 비용(시간)을 측정합니다. 내 테스트 컴퓨터에서 한 서버와 통신하는 데 약 60 마이크로초(usec)가 소요되었습니다. 3대의 서버와 통신하면 약 80 마이크로초(usec)가 소요됩니다.

산탄총 접근에 대한 장단점은 다음과 같습니다.

- 장점 : 단순하고 만들기 쉬우며 이해하기 쉽습니다.
- 장점 : 최소한 하나의 서버가 실행되면 장애조치 작업을 수행하고 빠르게 작동합니다.
- 단점 : 불필요한 네트워크 트래픽을 생성합니다.
- 단점 : 서버들에 대한 우선순위를 지정할 수 없습니다 (예 : Primary, Secondary).
- 단점 : 서버는 한 번에 최대 하나의 요청을 수행할 수 있습니다.
- [옮긴이] 모델 1(단순 응답 및 장애조치)에 이어 모델 2(샷건 살인)의 경우도 클라이언트가 가용한 모든 서버들에 동일한 작업 요청-응답 처리를 하고 있으며 서버 자원에 대한 낭비가 발생합니다. 현실 세계에서는 모델 1과 모델 2는 적절하지 않습니다.

0.52.3 모델 3: 복잡하고 불쾌한 방법

샷건 접근 방식이 실현하기에 너무 좋은 것처럼 보입니다. 과학적으로 가능한 대안들을 보도록 하겠습니다. 모델 3(복잡하고 불쾌한 옵션)을 탐구하며, 마침내 실제 적용하기 위해서는 잔인하더라도 복잡하고 불쾌한 방법이 선호되는 것을 깨닫게 됩니다. 이것은 삶의 이야기와 같이

단순하거나 무식한 방법으로 살고 싶지만, 인생에서는 복잡하고 불쾌한 방법으로 해결해야 하는 경우가 많기 때문입니다.

클라이언트의 주요 문제를 ROUTER 소켓으로 전환하여 해결할 수 있습니다. 특정 서버가 죽은 경우 요청 피하는 것이 일반적으로 똑똑한 방법입니다. ROUTER 소켓으로 전환하여 서버가 단일 스레드 같아지는 문제를 해결합니다.

그러나 2개의 익명 소켓들(식별자(ID)를 설정하지 않은) 사이에서 ROUTER와 ROUTER 통신을 수행하는 것은 불가능합니다. 양측은 첫 번째 메시지를 받을 때만 식별자(다른 상대에 대한)를 생성하므로, 처음 메시지를 받을 때까지 상대방과 통신을 할 수 없습니다. 이 수수께끼에서 벗어나는 유일한 방법은 메시지 수신 후 식별자 생성이 아닌 하드 코딩된 식별자를 한 방향으로 사용하는 것입니다. 클라이언트/서버의 경우 올바른 처리하는 방법은 클라이언트가 서버의 식별자를 미리 "알게"하는 것입니다. 다른 방법으로 수행하는 것은 복잡하고 불쾌하며 미쳐 버릴 것입니다. 많은 클라이언트가 개별적으로 실행되기 때문입니다. 미쳤고, 복잡하고, 불쾌한 것은 대량학살 독재자에게는 중요한 속성이지만 소프트웨어에게는 끔찍한 것입니다.

관리할 또 다른 개념을 만드는 대신 연결 단말을 식별자(ID)로 사용합니다. 이것은 클라이언트/서버 모두에게 고유한 식별자이며 샷건 모델에서 사용한 이력을 통해 양측이 동의할 수 있습니다. 두 개의 ROUTER 소켓을 연결하는 교활하고 효과적인 방법입니다.

ØMQ 식별자(ID)의 동작 방식은 서버 ROUTER 소켓은 식별자(ID)를 ROUTER 소켓에 바인딩하기 전에 설정합니다. 클라이언트가 연결되면 양쪽이 실제 메시지를 보내기 전에 식별자(ID) 교환을 위한 통신을 수행합니다.

- [옮긴이] 클라이언트에서 `zmq_setsockopt()`, `zsocket_set_identity()`를 통해 식별자 지정 가능합니다.

식별자(ID)를 설정하지 않은 클라이언트 ROUTER 소켓은 서버에 null 식별자를 보내면 서버는 임의의 UUID를 생성하여 자체적으로 사용할 클라이언트를 지정합니다. 서버는 자신의 식별자(우리가 사용하기로 한 단말 문자열(예 : tcp://localhost:5556))를 클라이언트에 보냅니다.

클라이언트에서 서버로 연결되면 클라이언트가 메시지를 서버로 전송 가능함을 의미합니다(클라이언트에서 서버 단말(예 : tcp://localhost:5556)을 식별자(ID)로 지정하여 ROUTER 소켓에서 전송). 그것은 `zmq_connect()`를 수행하고 잠시 시간이 지난 후에 연결됩니다. 여기

에 한 가지 문제가 있습니다: 우리는 서버가 실제로 가용하여 연결을 완료할 수 있는 시점을 모릅니다. 서버가 온라인 상태이면 몇 밀리초(msec)가 걸릴 수 있지만 서버가 다운되고 시스템 관리자가 점심을 먹으러 가면 지금부터 한 시간이 걸릴 수도 있습니다.

여기에 작은 역설이 있습니다. 서버가 언제 연결되고 가용한 시점을 알아야 합니다. 프리랜서 패턴에서는 앞부분에서 본 브로커 기반 패턴과 달리 서버가 말을 할 때까지 침묵합니다. 따라서 서버가 온라인이라고 말할 때까지 서버와 통신할 수 없으며, 우리가 요청하기 전까지는 통신할 수 없습니다.

해결책은 모델 2의 샷건 접근을 약간 혼합하는 것이며, 방법은 인지 가능한 모든 것에 대하여 산탄총을 발사(무해하게)하여 움직이는 것이 있다면 살아 있다는 것을 아는 것입니다. 우리는 실제 요청을 실행하는 것이 아니라 일종의 핑-퐁(ping-pong) 심박입니다.

이것은 우리를 다시 통신규약 영역으로 가져와서, 프리랜서 클라이언트와 서버가 핑-퐁(ping-pong) 명령과 요청-응답을 수행하는 방법을 정의하는 사양서로 만들었습니다. - Freelance Protocol

서버로 구현하는 것은 짧고 달콤합니다. 다음은 에코 서버입니다.

flserver3.c: 프리랜서 서버, 모델3

```
//  Freelance server - Model 3
//  Uses an ROUTER/ROUTER socket but just one thread

#include "czmq.h"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    zctx_t *ctx = zctx_new ();

    //  Prepare server socket with predictable identity
    char *bind_endpoint = "tcp://*:5555";
    char *connect_endpoint = "tcp://localhost:5555";
```

```
    void *server = zsocket_new (ctx, ZMQ_ROUTER);
    zmq_setsockopt (server,
        ZMQ_IDENTITY, connect_endpoint, strlen (connect_endpoint));
    zsocket_bind (server, bind_endpoint);
    printf ("I: service is ready at %s\n", bind_endpoint);

    while (!zctx_interrupted) {
        zmsg_t *request = zmsg_recv (server);
        if (verbose && request)
            zmsg_dump (request);
        if (!request)
            break;          //  Interrupted

        //  Frame 0: identity of client
        //  Frame 1: PING, or client control frame
        //  Frame 2: request body
        zframe_t *identity = zmsg_pop (request);
        zframe_t *control = zmsg_pop (request);
        zmsg_t *reply = zmsg_new ();
        if (zframe_streq (control, "PING"))
            zmsg_addstr (reply, "PONG");
        else {
            zmsg_add (reply, control);
            zmsg_addstr (reply, "OK");
        }
        zmsg_destroy (&request);
        zmsg_prepend (reply, &identity);
        if (verbose && reply)
            zmsg_dump (reply);
```

```
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

그러나 프리랜서 클라이언트는 거대해졌습니다. 명확하게 하기 위해 예제 응용프로그램과 어려운 작업을 수행하는 클래스로 나뉩니다. 다음은 메인 응용프로그램입니다.

flclient3.c: 프리랜서 클라이언트, 모델 3

```
//  Freelance client - Model 3
//  Uses flcliapi class to encapsulate Freelance pattern

//  Lets us build this source without creating a library
#include "flcliapi.c"

int main (void)
{
    //  Create new freelance client object
    flcliapi_t *client = flcliapi_new ();

    //  Connect to several endpoints
    flcliapi_connect (client, "tcp://localhost:5555");
    flcliapi_connect (client, "tcp://localhost:5556");
    flcliapi_connect (client, "tcp://localhost:5557");

    //  Send a bunch of name resolution 'requests', measure time
    int requests = 1000;
```

```
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flcliapi_request (client, &request);
        if (!reply) {
            printf ("E: name service not available, aborting\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("Average round trip cost: %d usec\n",
        (int) (zclock_time () - start) / 10);

    flcliapi_destroy (&client);
    return 0;
}
```

그리고 여기에 MDP 브로커만큼 복잡하고 거대한 클라이언트 API 클래스가 있습니다.

flcliapi.c: 프리랜서 클라이언트 API

```
//  flcliapi class - Freelance Pattern agent class
//  Implements the Freelance Protocol at http://rfc.zeromq.org/spec:10

#include "flcliapi.h"

//  If no server replies within this time, abandon request
#define GLOBAL_TIMEOUT  3000    //  msecs
//  PING interval for servers we think are alive
#define PING_INTERVAL   2000    //  msecs
```

```
//  Server considered dead if silent for this long
#define SERVER_TTL      6000    //  msecs

//  .split API structure
//  This API works in two halves, a common pattern for APIs that need to
//  run in the background. One half is an frontend object our application
//  creates and works with; the other half is a backend "agent" that runs
//  in a background thread. The frontend talks to the backend over an
//  inproc pipe socket:

//  Structure of our frontend class

struct _flcliapi_t {
    zctx_t *ctx;        //  Our context wrapper
    void *pipe;         //  Pipe through to flcliapi agent
};

//  This is the thread that handles our real flcliapi class
static void flcliapi_agent (void *args, zctx_t *ctx, void *pipe);

//  Constructor

flcliapi_t *
flcliapi_new (void)
{
    flcliapi_t
        *self;

    self = (flcliapi_t *) zmalloc (sizeof (flcliapi_t));
```

```
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, flcliapi_agent, NULL);
    return self;
}

//  Destructor

void
flcliapi_destroy (flcliapi_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flcliapi_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  .split connect method
//  To implement the connect method, the frontend object sends a multipart
//  message to the backend agent. The first part is a string "CONNECT", and
//  the second part is the endpoint. It waits 100msec for the connection to
//  come up, which isn't pretty, but saves us from sending all requests to a
//  single server, at startup time:

void
flcliapi_connect (flcliapi_t *self, char *endpoint)
{
```

```
    assert (self);
    assert (endpoint);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, endpoint);
    zmsg_send (&msg, self->pipe);
    zclock_sleep (100);      //  Allow connection to come up
}

//  .split request method
//  To implement the request method, the frontend object sends a message
//  to the backend, specifying a command "REQUEST" and the request message:

zmsg_t *
flcliapi_request (flcliapi_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);

    zmsg_pushstr (*request_p, "REQUEST");
    zmsg_send (request_p, self->pipe);
    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *status = zmsg_popstr (reply);
        if (streq (status, "FAILED"))
            zmsg_destroy (&reply);
        free (status);
    }
    return reply;
```

```
}

//  .split backend agent
//  Here we see the backend agent. It runs as an attached thread, talking
//  to its parent over a pipe socket. It is a fairly complex piece of work
//  so we'll break it down into pieces. First, the agent manages a set of
//  servers, using our familiar class approach:

//  Simple class for one server we talk to

typedef struct {
    char *endpoint;             //  Server identity/endpoint
    uint alive;                 //  1 if known to be alive
    int64_t ping_at;            //  Next ping at this time
    int64_t expires;            //  Expires at this time
} server_t;

server_t *
server_new (char *endpoint)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));
    self->endpoint = strdup (endpoint);
    self->alive = 0;
    self->ping_at = zclock_time () + PING_INTERVAL;
    self->expires = zclock_time () + SERVER_TTL;
    return self;
}

void
```

```
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->endpoint);
        free (self);
        *self_p = NULL;
    }
}

int
server_ping (const char *key, void *server, void *socket)
{
    server_t *self = (server_t *) server;
    if (zclock_time () >= self->ping_at) {
        zmsg_t *ping = zmsg_new ();
        zmsg_addstr (ping, self->endpoint);
        zmsg_addstr (ping, "PING");
        zmsg_send (&ping, socket);
        self->ping_at = zclock_time () + PING_INTERVAL;
    }
    return 0;
}

int
server_tickless (const char *key, void *server, void *arg)
{
    server_t *self = (server_t *) server;
```

```
    uint64_t *tickless = (uint64_t *) arg;
    if (*tickless > self->ping_at)
        *tickless = self->ping_at;
    return 0;
}

//  .split backend agent class
//  We build the agent as a class that's capable of processing messages
//  coming in from its various sockets:

//  Simple class for one background agent

typedef struct {
    zctx_t *ctx;                //  Own context
    void *pipe;                 //  Socket to talk back to application
    void *router;               //  Socket to talk to servers
    zhash_t *servers;           //  Servers we've connected to
    zlist_t *actives;           //  Servers we know are alive
    uint sequence;              //  Number of requests ever sent
    zmsg_t *request;            //  Current request if any
    zmsg_t *reply;              //  Current reply if any
    int64_t expires;            //  Timeout for request/reply
} agent_t;

agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
```

```
    self->pipe = pipe;
    self->router = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->servers = zhash_new ();
    self->actives = zlist_new ();
    return self;
}

void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        zhash_destroy (&self->servers);
        zlist_destroy (&self->actives);
        zmsg_destroy (&self->request);
        zmsg_destroy (&self->reply);
        free (self);
        *self_p = NULL;
    }
}

//  .split control messages
//  This method processes one message from our frontend class
//  (it's going to be CONNECT or REQUEST):

//  Callback when we remove server from agent 'servers' hash table

static void
```

```
s_server_free (void *argument)
{
    server_t *server = (server_t *) argument;
    server_destroy (&server);
}

void
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);

    if (streq (command, "CONNECT")) {
        char *endpoint = zmsg_popstr (msg);
        printf ("I: connecting to %s...\n", endpoint);
        int rc = zmq_connect (self->router, endpoint);
        assert (rc == 0);
        server_t *server = server_new (endpoint);
        zhash_insert (self->servers, endpoint, server);
        zhash_freefn (self->servers, endpoint, s_server_free);
        zlist_append (self->actives, server);
        server->ping_at = zclock_time () + PING_INTERVAL;
        server->expires = zclock_time () + SERVER_TTL;
        free (endpoint);
    }
    else
    if (streq (command, "REQUEST")) {
        assert (!self->request);    //  Strict request-reply cycle
        //  Prefix request with sequence number and empty envelope
```

```
        char sequence_text [10];
        sprintf (sequence_text, "%u", ++self->sequence);
        zmsg_pushstr (msg, sequence_text);
        //  Take ownership of request message
        self->request = msg;
        msg = NULL;
        //  Request expires after global timeout
        self->expires = zclock_time () + GLOBAL_TIMEOUT;
    }
    free (command);
    zmsg_destroy (&msg);
}

//  .split router messages
//  This method processes one message from a connected
//  server:

void
agent_router_message (agent_t *self)
{
    zmsg_t *reply = zmsg_recv (self->router);

    //  Frame 0 is server that replied
    char *endpoint = zmsg_popstr (reply);
    server_t *server =
        (server_t *) zhash_lookup (self->servers, endpoint);
    assert (server);
    free (endpoint);
    if (!server->alive) {
```

```
        zlist_append (self->actives, server);
        server->alive = 1;
    }
    server->ping_at = zclock_time () + PING_INTERVAL;
    server->expires = zclock_time () + SERVER_TTL;

    //  Frame 1 may be sequence number for reply
    char *sequence = zmsg_popstr (reply);
    if (atoi (sequence) == self->sequence) {
        zmsg_pushstr (reply, "OK");
        zmsg_send (&reply, self->pipe);
        zmsg_destroy (&self->request);
    }
    else
        zmsg_destroy (&reply);
}

//  .split backend agent implementation
//  Finally, here's the agent task itself, which polls its two sockets
//  and processes incoming messages:

static void
flcliapi_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    zmq_pollitem_t items [] = {
        { self->pipe, 0, ZMQ_POLLIN, 0 },
        { self->router, 0, ZMQ_POLLIN, 0 }
```

```
    };
    while (!zctx_interrupted) {
        //  Calculate tickless timer, up to 1 hour
        uint64_t tickless = zclock_time () + 1000 * 3600;
        if (self->request
        &&  tickless > self->expires)
            tickless = self->expires;
        zhash_foreach (self->servers, server_tickless, &tickless);

        int rc = zmq_poll (items, 2,
            (tickless - zclock_time ()) * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Context has been shut down

        if (items [0].revents & ZMQ_POLLIN)
            agent_control_message (self);

        if (items [1].revents & ZMQ_POLLIN)
            agent_router_message (self);

        //  If we're processing a request, dispatch to next server
        if (self->request) {
            if (zclock_time () >= self->expires) {
                //  Request expired, kill it
                zstr_send (self->pipe, "FAILED");
                zmsg_destroy (&self->request);
            }
            else {
                //  Find server to talk to, remove any expired ones
```

```
                while (zlist_size (self->actives)) {
                    server_t *server =
                        (server_t *) zlist_first (self->actives);
                    if (zclock_time () >= server->expires) {
                        zlist_pop (self->actives);
                        server->alive = 0;
                    }
                    else {
                        zmsg_t *request = zmsg_dup (self->request);
                        zmsg_pushstr (request, server->endpoint);
                        zmsg_send (&request, self->router);
                        break;
                    }
                }
            }
        }
        //  Disconnect and delete any expired servers
        //  Send heartbeats to idle servers if needed
        zhash_foreach (self->servers, server_ping, self->router);
    }
    agent_destroy (&self);
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc flserver3.c libzmq.lib czmq.lib
 cl -EHsc flclient3.c libzmq.lib czmq.lib

 ./flserver3 -v
I: service is ready at tcp://*:5555
```

```
D: 20-08-25 14:42:43 [005] 0080000029
D: 20-08-25 14:42:43 [004] PING
D: 20-08-25 14:42:43 [005] 0080000029
D: 20-08-25 14:42:43 [004] PONG
D: 20-08-25 14:45:00 [005] 0080000029
D: 20-08-25 14:45:00 [001] 1
D: 20-08-25 14:45:00 [011] random name
D: 20-08-25 14:45:00 [005] 0080000029
D: 20-08-25 14:45:00 [001] 1
D: 20-08-25 14:45:00 [002] OK
D: 20-08-25 14:45:00 [005] 0080000029
D: 20-08-25 14:45:00 [001] 2
D: 20-08-25 14:45:00 [011] random name
D: 20-08-25 14:45:00 [005] 0080000029
D: 20-08-25 14:45:00 [001] 2
D: 20-08-25 14:45:00 [002] OK
...


 ./flclient3 -v
I: connecting to tcp://localhost:5555...
I: connecting to tcp://localhost:5556...
I: connecting to tcp://localhost:5557...
Average round trip cost: 152 usec
```

프리랜서 API 구현은 상당히 정교하며 이전에 보지 못한 몇 가지 기술을 사용합니다.

- 멀티스레드 API : 클라이언트 API는 두 부분으로 구성되며, 응용프로그램 스레드에서 실행되는 동기식 flcliapi 클래스와 백그라운드 스레드로 실행되는 비동기 agent 클래스입니다. ØMQ가 멀티스레드 응용프로그램들을 쉽게 만드는 방법을 생각하면 flcliapi(동기식) 및 agent(비동기식) 클래스는 inproc 소켓을 통해 메시지들로 통신합니다. 모든 ØMQ 측면(예 : 컨텍스트 생성 및 삭제)은 API에 은폐되어 있습니다. 실제로 agent는

미니 브로커처럼 동작하여 백그라운드에서 서버와 통신하면서, 요청 시 가용한 서버을 찾기 위해 최선을 다할 수 있습니다.

- 무지연 폴링 타이머(Tickless poll timer) : 이전 폴링 루프에서 우리는 항상 일정한 시간 간격(예 : 1초)을 사용했는데, 이는 단순하지만 저전력 클라이언트(노트북 또는 휴대폰 등)에 적합하지 않습니다. 재미와 지구를 구하기 위해 에이전트가 무지연 타이머 사용하면, 무지연 타이머는 예상되는 다음 제한시간을 기준으로 폴링 지연을 계산합니다. 적절한 구현은 순서가 지정된 제한시간 목록을 유지하며, 모든 제한시간을 확인하고 다음 제한시간까지 폴링 지연을 수행합니다.

- [옮긴이] 모델 1(단순 응답 및 장애조치), 모델 2(샷건 살인)에 이어 모델 3(복잡하고 불쾌한) 경우 기존 모델과 달리 클라이언트가 가용한 모든 서버들 중에서 가용한 서버 한 대로만 작업 요청-응답 처리를 수행하여, 서버 자원 부분에서 낭비는 발생하지 않지만 무수히 많은 클라이언트들이 요청을 수행할 경우 부하 분산을 불가한 구조입니다. 현실 세계에서는 서버 자원 및 부하 분산 측면에서 모델 1, 모델 2, 모델 3도 적절하게 보이지 않습니다. 분산 환경에서 브로커 없이 사용하기 위해서는 서버들(서비스들)의 존재를 클라이언트들에서 알기 위해 이름 확인 서비스(Naming Resoution)을 통해 특정 서버(서비스) 확인하고 접속하여 서비스를 수행하다가 서버(서비스)가 죽은 경우 다시 이름 확인 서비스를 통해 동일한 서비스를 제공하는 서버에 접속하여 처리 필요합니다. 하지만 이경우 부하 분산 문제가 있으며 잘못하면 특정 서버로 부하가 집중될 수 있기 때문에 주의가 필요합니다. 해결 방법으로 HAProxy와 같은 부하 분산 도구를 사용할 수 있지만 HAProxy의 장애가 발생할 경우 장애조치 시간이 문제가 될 수 있습니다. 물론 MDP 형태도 가능하지만, 브로커 경유에 따른 성능 지연이 발생하며, 실시간 처리가 중요한 응용프로그램에서는 이슈가 될 수 있습니다.

0.53 결론

이 장에서는 다양한 신뢰성 있는 요청-응답 메커니즘을 보았으며, 각각의 특정 비용과 장점이 있었습니다. 예제 코드는 최적화되어 있지는 않지만 대부분 실제 사용이 가능합니다. 모든 다른 패턴 중에서 두 개의 패턴이 양산용으로 사용하기에 돋보였으며, 첫째는 브로커 기반 안정성

을 위한 집사(Majordomo) 패턴과, 둘째는 브로커 없는 안정성을 위한 프리랜서(Freelance) 패턴입니다.

5장 - 고급 발행-구독 패턴

"3장 - 고급 요청-응답 패턴" 및 "4장 - 신뢰할 수 있는 요청-응답 패턴"에서 ØMQ의 요청-응답 패턴의 고급 사용을 보았습니다. 그 모든 것을 이해하셨다면 축하드리며, 이 장에서는 발행-구독(publish-subscribe)에 중점을 두고, ØMQ의 핵심인 발행-구독 패턴을 성능, 안정성, 변경정보 배포 및 모니터링을 위한 상위 수준 패턴으로 확장합니다.

다루는 내용은 다음과 같습니다.

- 발행-구독을 사용해야 할 때
- 너무 느린 구독자를 처리하는 방법(자살하는 달팽이 패턴)
- 고속 구독자 설계 방법(블랙박스 패턴)
- 발행-구독 네트워크를 모니터링하는 방법(에스프레소(Espresso) 패턴)
- 공유 카-값 저장소를 만드는 방법(복제 패턴)
- 리엑터를 사용하여 복잡한 서버를 단순화하는 방법
- 바이너리 스타 패턴을 사용하여 서버에 장애조치를 추가하는 방법

0.54 발행-구독의 장점과 단점

ØMQ의 저수준 패턴은 서로 다른 특성을 가지고 있습니다. 발행-구독은 멀티캐스트 혹은 그룹 메시징과 같은 오래된 메시징 문제를 해결합니다. 그것은 ØMQ를 특징짓는 꼼꼼한 단순성과 잔인한 무관심의 고유한 혼합을 가지고 있습니다. 발행-구독이 만드는 장단점이 어떻게 도움이

되는지, 필요한 경우 어떻게 사용해야 할지 이해할 가치가 있습니다.

첫째, PUB 소켓은 각 메시지를 “all of many”로 보내는 반면 PUSH 및 DEALER 소켓은 메시지를 “one of many”로 수신자들에게 순차적으로 전달합니다. 단순히 PUSH를 PUB로 바꾸거나 역으로 하더라도 모든 것이 똑같이 동작하기를 바랄 수는 없습니다. 사람들이 바꾸어 사용(PUSH, PUB)하는 것을 자주 제안하기 때문에 문제는 반복됩니다.

좀 더 깊게 생각하면 발행-구독은 확장성을 목표로 합니다. 이는 많은 수신자들에게 빠르게 대량의 데이터를 송신하는 것입니다. 초당 수백만 개의 메시지를 수천 개의 단말로 송신해야 하는 경우, 소수의 수신자들에게 초당 몇 개의 메시지를 보내는 경우보다 발행-구독의 기능에 고마워할 것입니다.

확장성을 얻기 위해 발행-구독(pub-sub)은 푸시-풀(push-pull)과 동일하게 백-채터(back-chatter)를 제거합니다. 백-채터는 수신자가 발신자에게 응답하지 않음을 의미합니다. 일부 예외적인 경우로, SUB 소켓은 PUB 소켓에 구독을 보내지만 익명이며 드물게 일어납니다.

백-채터를 제거하는 것은 실제 확장성에 필수적입니다. 발행-구독 패턴은 네트워크 스위치에 의해 처리되는 PGM 멀티캐스트 통신규약에 말끔하게 매핑될 수 있습니다. 즉, 구독자는 발행자에 연결하지 않고, 발행자가 메시지를 보내는 네트워크 스위치 상의 멀티캐스트 그룹에 연결합니다.

백-채터를 제거하면, 전체 메시지 흐름은 훨씬 단순해져, 많은 사람들이 단순한 API와 단순한 통신규약을 사용할 수 있습니다. 그러나 백-채터의 제거는 송신자들과 수신자들를 조정할 수 없게 되며 의미하는 바는 다음과 같습니다.

- 발행자들은 구독자들의 연결 성공 시점이 초기 연결 혹은 네트워크 장애 후에 재연결인지 모릅니다.
- 구독자들은 발행자들에게 “전송 메시지 속도를 조정해 주세요”와 같은 어떤 메시지도 송신할 수 없습니다. 발행자들은 최대 속도로 메시지를 전송하면, 구독자의 성능의 따라 메시지 계속 받거나 유실하게 됩니다.
- 발행자들은 구독자들이 프로세스 충돌, 네트워크 중단 등으로 사라진 시점을 모릅니다.

위의 단점은 안정적인 멀티캐스트를 수행하기 위해 해결이 필요합니다. ØMQ 발행-구독 패턴은 구독자가 연결 중이거나, 네트워크 장애가 발생하거나, 발행자의 송신하는 많은 메시지를 구독자 또는 네트워크가 처리할 수 없는 경우 메시지가 유실됩니다.

하지만 긍정적인 면에서 안정적인 멀티캐스팅의 괜찮은 사용 사례가 많다는 것입니다. 백-채터가 필요할 때, ROUTER-DEALER(보통 수준의 메시지 처리량일 경우 주로 사용)를 사용하도록 전환하거나 동기화를 위한 별도의 채널을 추가할 수 있습니다(이에 대한 예는 나중에 살펴보겠습니다).

발행-구독은 라디오 방송과 같이 특정 채널에 가입하기 전에는 모든 것을 놓치며, 수신 품질에 따라 정보의 양이 달라집니다. 놀랍게도 이 모델은 실제 정보 배포에 완벽하게 매핑되어 유용하게 널리 사용되고 있습니다. 페이스북(Facebook)과 트위터(Twitter), BBC 세계 서비스, 스포츠 방송을 생각해 보십시오.

요청-응답과 같이 무엇이 잘못될 수 있는지의 관점에서 신뢰성을 정의하겠습니다. 다음은 발행-구독의 전형적인 장애 사례입니다.

- 구독자가 늦게 가입하여 발행자가 이미 보낸 메시지들을 유실합니다.
- 구독자가 메시지들를 너무 느리게 처리하여, 대기열이 가득 차면 유실됩니다.
- 구독자가 자리를 비운 동안 메시지들를 유실할 수 있습니다.
- 구독자가 장애 후 재시작되면 이미 받은 데이터들을 유실할 수 있습니다.
- 네트워크가 과부하가 되어 데이터들을 유실할 수 있습니다(특히 PGM 통신의 경우).
- 네트워크의 처리 속도가 너무 느려 발행자의 대기열이 가득 차고 발행자에게 장애가 발생할 수 있습니다(ØMQ v3.2 이전).

더 많은 것이 잘못될 수 있지만 이것들은 우리가 현실적인 시스템에서 보는 전형적인 장애입니다. ØMQ v3.x부터 ØMQ 대기열의 내부 버퍼(소위 HWM(high-water mark))에 대한 기본적인 제한을 적용하므로 의도적으로 HWM을 무한으로 설정하지 않으면 발행자의 대기열이 가득 차는 장애는 발생하지 않습니다.

- [옮긴이] HWM은 `zmq_setsockopt()`에서 송신의 경우 `ZMQ_SNDHWM`, 수신의 경우 `ZMQ_RCVHWM`으로 설정합니다.

- PUB, PUSH : 송신 버퍼 존재

- SUB, PULL, REQ, REP : 수신 버퍼 존재

- DEALER/ROUTER/PAIR : 송신 및 수신 버퍼 존재

항상 단순한 것은 아니지만 장애 사례들에는 해결책이 있습니다. 신뢰성은 대부분의 경우 필요하지 않는 복잡한 기능을 요구하지만, ØMQ는 기본적으로 제품 형태로 제공하려고 시도하진 않습니다(신뢰성을 적용하기 위한 범용적인 설계 방안은 존재하지 않습니다.).

0.55 발행-구독 추적 (에스프레소 패턴)

발행-구독 네트워크를 추적하는 방법을 보면서 이장을 시작하겠습니다. "2장 - 소켓 및 패턴"에서 전송 계층 간의 다리 역할을 수행하는 간단한 프록시를 보았습니다. `zmq_proxy()` 메서드에는 3개 인수는 소켓들로 전송 계층 간 연결할 프론트앤드 소켓과 백엔드 소켓, 모든 메시지를 송신할 캡처 소켓입니다.

- [옮긴이] int zmq_proxy (void frontend_, void backend_, void *capture_)

다음 코드는 매우 단순합니다.

espresso.c: 에스프레소 패턴

```
//  Espresso Pattern
//  This shows how to capture data using a pub-sub proxy

#include "czmq.h"

//  The subscriber thread requests messages starting with
//  A and B, then reads and counts incoming messages.

static void
subscriber_thread (void *args, zctx_t *ctx, void *pipe)
{
    //  Subscribe to "A" and "B"
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:6001");
```

```
    zsocket_set_subscribe (subscriber, "A");
    zsocket_set_subscribe (subscriber, "B");

    int count = 0;
    while (count < 5) {
        char *string = zstr_recv (subscriber);
        if (!string)
            break;              //  Interrupted
        free (string);
        count++;
    }
    zsocket_destroy (ctx, subscriber);
}

//  .split publisher thread
//  The publisher sends random messages starting with A-J:

static void
publisher_thread (void *args, zctx_t *ctx, void *pipe)
{
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:6000");

    while (!zctx_interrupted) {
        char string [10];
        sprintf (string, "%c-%05d", randof (10) + 'A', randof (100000));
        if (zstr_send (publisher, string) == -1)
            break;              //  Interrupted
        zclock_sleep (100);     //  Wait for 1/10th second
```

```
    }
}

//  .split listener thread
//  The listener receives all messages flowing through the proxy, on its
//  pipe. In CZMQ, the pipe is a pair of ZMQ_PAIR sockets that connect
//  attached child threads. In other languages your mileage may vary:

static void
listener_thread (void *args, zctx_t *ctx, void *pipe)
{
    //  Print everything that arrives on pipe
    while (true) {
        zframe_t *frame = zframe_recv (pipe);
        if (!frame)
            break;              //  Interrupted
        zframe_print (frame, NULL);
        zframe_destroy (&frame);
    }
}

//  .split main thread
//  The main task starts the subscriber and publisher, and then sets
//  itself up as a listening proxy. The listener runs as a child thread:

int main (void)
{
    //  Start child threads
    zctx_t *ctx = zctx_new ();
```

```
    zthread_fork (ctx, publisher_thread, NULL);
    zthread_fork (ctx, subscriber_thread, NULL);

    void *subscriber = zsocket_new (ctx, ZMQ_XSUB);
    zsocket_connect (subscriber, "tcp://localhost:6000");
    void *publisher = zsocket_new (ctx, ZMQ_XPUB);
    zsocket_bind (publisher, "tcp://*:6001");
    void *listener = zthread_fork (ctx, listener_thread, NULL);
    zmq_proxy (subscriber, publisher, listener);

    puts (" interrupted");
    //  Tell attached threads to exit
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 ./espresso
D: 20-08-26 13:43:30 [002] 0141
D: 20-08-26 13:43:31 [002] 0142
D: 20-08-26 13:43:31 [007] A-30398
D: 20-08-26 13:43:31 [007] B-14730
D: 20-08-26 13:43:32 [007] A-11907
D: 20-08-26 13:43:33 [007] B-53933
D: 20-08-26 13:43:33 [007] A-84011
D: 20-08-26 13:43:33 [002] 0041
D: 20-08-26 13:43:33 [002] 0042
```

에스프레소는 생성한 리스너 스레드는 PAIR 소켓을 읽고 출력을 합니다. PAIR 소켓은 파이프(pipe)의 한쪽 끝이며 다른 쪽 끝(추가 PAIR)은 `zmq_proxy()`에 전달하는 소켓입니다.

실제로 관심 있는 메시지를 필터링하여 추적하려는 대상(에스프레소 패턴의 이름처럼)의 본질을 얻습니다.

- [옮긴이] 에스프레소(Espresso)는 곱게 갈아 압축한 원두가루에 뜨거운 물을 고압으로 통과시켜 뽑아낸 이탈리안 정통 커피로 아주 진한 향과 맛이 특징이다.

구독자 스레드는 "A" 및 "B"를 포함한 메시지 구독하며, 수신된 메시지가 5 개째 소켓을 제거하고 종료합니다. 예제를 실행하면 리스너는 2개의 구독 메시지(/01)들, 5개의 데이터 메시지들("A", "B" 포함)과 2개의 구독 취소 메시지(/00)를 출력하고 조용해집니다.

```
[002] 0141
[002] 0142
[007] B-91164
[007] B-12979
[007] A-52599
[007] A-06417
[007] A-45770
[002] 0041
[002] 0042
```

이것은 발행자 소켓이 구독자가 없을 때 데이터 전송을 중지하는 방법을 깔끔하게 보여줍니다. 발행자 스레드가 여전히 메시지를 보내고 있지만 소켓은 조용히 그들을 버립니다.

- [옮긴이] 구독자는 발행자에게 구독 시와 구독 취소 시 이벤트(event)를 보내며 보내는 메시지는 바이트(byte) 형태로 첫 번째 바이트(HEX 코드)는 "00"은 구독, "01"은 구독 취소이며 나머지 바이트들은 토픽(sizeof(event)-1)으로 구성됩니다.

```
[002] 0141      --> "01" 구독, 토픽 : "41" A
[002] 0142      --> "01" 구독, 토픽 : "41" B
[007] B-91164
[007] B-12979
[007] A-52599
```

```
[007] A-06417
[007] A-45770
[002] 0041       --> "00" 구독 취소, 토픽 : "41" A
[002] 0042       --> "00" 구독 취소, 토픽 : "41" B
```

0.56 마지막 값 캐싱

상용 발행-구독 시스템을 사용했다면 빠르고 즐거운 ØMQ 발행-구독 모델에서 누락된 일부 익숙한 기능들을 알 수 있습니다. 이 중 하나는 마지막 값 캐싱(LVC : Last Value Caching)입니다. 이것은 새로운 구독자가 네트워크에 참여할 때 누락한 메시지 받을 수 있는 방식으로 문제를 해결합니다. 이론은 발행자들에게 새로운 구독자가 참여했고 특정 토픽에 구독하려는 시점을 알려주면, 발행자는 해당 토픽에 대한 마지막 메시지를 재송신할 수 있습니다.

- [옮긴이] TIB/RV(TIBCO Rendezvous)는 TIBCO사에서 개발한 상용 메시지 소프트웨어로 요청/응답(Request/reply), 브로드케스팅(Broadcasting), 신뢰성 메시징(reliable messaging), 메시지 전달 보장(Certified Messaging), 분산 메시지(Distributed Message), 원격 통신(Remote communication)등의 기능을 제공합니다.

발행자들이 새로운 구독자들이 참여하는 것을 알지 못하는 것은 대규모 발행-구독 시스템에서는 데이터의 양으로 인해 거의 불가능하기 때문입니다. 정말 대규모 발행-구독 네트워크를 만들려면 PGM과 같은 통신규약을 통해 이더넷 스위치의 기능을 활용하여 수천 명의 구독자에게 데이터를 멀티캐스트 해야 합니다. 발행자가 각각 수천 명의 구독자들에게 TCP 유니캐스트를 시도하는 것으로 확장할 수 없으며, 이상한 불협화음, 불공정한 배포(일부 구독자가 다른 구독자보다 먼저 메시지를 받음), 네트워크 정체 등으로 보통 좋지 않은 결과를 겪습니다.

PGM은 단방향 통신규약입니다. 발행자는 네트워크 스위치의 멀티캐스트 주소로 메시지를 보내면 모든 관심 있는 구독자들에게 다시 브로드캐스트 합니다. 발행자는 구독자가 가입하거나 탈퇴하는 시점을 결코 알 수 없습니다. 이 모든 작업은 스위치에서 발생하며, 네트워크 스위치의 프로그래밍을 다시 작성하고 싶지는 않습니다.

그러나 낮은 대역폭을 가지는 저용량 네트워크에 수십 명의 구독자의 제한된 토픽가 있으면 TCP를 사용할 수 있으며, XSUB 및 XPUB 소켓은 에스프레소 패턴에서 본 것처럼 상호 통신합니다.

"ØMQ를 사용하여 LVC(last value cache)를 만들 수 있을까요?"에 대한 대답은 "예"이며 발행자와 구독자들 사이에 프록시를 만들면 됩니다. PGM 네트워크 스위치와 유사하지만 ØMQ는 우리가 직접 프로그래밍할 수 있습니다.

최악의 시나리오를 가정한 발행자와 구독자를 만드는 것부터 시작하겠습니다. 이 발행자는 병리학적입니다. 발행자는 각각 1000개의 토픽들에 즉시 메시지를 보내고 1초마다 임의의 토픽에 하나의 변경정보를 전송합니다. 구독자는 연결하고 토픽을 구독합니다. 마지막 값 캐핑(LVC)이 없으면 구독자는 데이터를 얻기 위해 평균 500초를 기다려야 합니다. TV 드라마에서 그레고르라는 탈옥한 죄수가 8.3분 내에 구독자의 데이터를 얻게 하지 못한다면 장난감 토끼로저에게서 머리를 날려버리겠다고 위협하는 것처럼 말입니다.

다음은 발행자 코드입니다. 일부 주소에 연결하는 명령 줄 옵션이 없으면 단말에 바인딩되며, 나중에 마지막 값 캐시(LVC)에 연결에 사용합니다.

pathopub.c : 병리학적인 발행자

```
//  Pathological publisher
//  Sends out 1,000 topics and then one random update per second

#include "czmq.h"
#include "zhelpers.h"

int main (int argc, char *argv [])
{
    zctx_t *context = zctx_new ();
    void *publisher = zsocket_new (context, ZMQ_PUB);
    if (argc == 2)
        zsocket_bind (publisher, argv [1]);
    else
        zsocket_bind (publisher, "tcp://*:5556");
```

```
    //  Ensure subscriber connection has time to complete
    s_sleep (1000);

    //  Send out all 1,000 topic messages
    int topic_nbr;
    for (topic_nbr = 0; topic_nbr < 1000; topic_nbr++) {
        zstr_sendfm (publisher, "%03d", topic_nbr);
        zstr_send (publisher, "Save Roger");
    }
    //  Send one random update per second
    srandom ((unsigned) time (NULL));
    while (!zctx_interrupted) {
        s_sleep (1000);
        zstr_sendfm (publisher, "%03d", randof (1000));
        zstr_send (publisher, "Off with his head!");
    }
    zctx_destroy (&context);
    return 0;
}
```

구독자 코드는 다음과 같습니다.

pathosub.c : 병리학적인 구독자

```
//  Pathological subscriber
//  Subscribes to one random topic and prints received messages

#include "czmq.h"

int main (int argc, char *argv [])
{
```

```
    zctx_t *context = zctx_new ();
    void *subscriber = zsocket_new (context, ZMQ_SUB);
    if (argc == 2)
        zsocket_connect (subscriber, argv [1]);
    else
        zsocket_connect (subscriber, "tcp://localhost:5556");

    srandom ((unsigned) time (NULL));
    char subscription [5];
    sprintf (subscription, "%03d", randof (1000));
    zsocket_set_subscribe (subscriber, subscription);

    while (true) {
        char *topic = zstr_recv (subscriber);
        if (!topic)
            break;
        char *data = zstr_recv (subscriber);
        assert (streq (topic, subscription));
        puts (data);
        free (topic);
        free (data);
    }
    zctx_destroy (&context);
    return 0;
}
```

먼저 구독자를 실행하고 다음 발행자를 실행하면 구독자가 기대한 대로 “Save Roger”를 출력합니다.

```
./pathosub &
./pathopub
```

- [옮긴이] 빌드 및 테스트
- "pathsub" 실행하면 생성된 토픽에 해당되는 "Sava Roger"을 수신하고 임의의 긴 시간 (1초~16.7분) 후에 "Off with his head!"을 수신하게 됩니다.

```
 cl -EHsc pathopub.c libzmq.lib czmq.lib
 cl -EHsc pathosub.c libzmq.lib czmq.lib

 ./pathopub

 ./pathosub
Save Roger
Off with his head!
```

두 번째 구독자를 실행하면 Roger의 곤경("Save Roger"을 수신하지 못하고 오랜 시간 후에 "Off with his head!" 수신)을 알게 되며 데이터 수신하기까지 오랜 시간을 기다려야 합니다. 여기에 마지막 값 캐시(LVC)가 있습니다. 프록시가 2개의 소켓에 바인딩하고 양쪽의 메시지를 처리합니다.

lvcach.c : 마지막 값 저장 프록시

```
//  Last value cache
//  Uses XPUB subscription messages to re-send data

#include "czmq.h"

int main (void)
{
    zctx_t *context = zctx_new ();
```

```
    void *frontend = zsocket_new (context, ZMQ_SUB);
    zsocket_connect (frontend, "tcp://*:5557");
    void *backend = zsocket_new (context, ZMQ_XPUB);
    zsocket_bind (backend, "tcp://*:5558");

    //  Subscribe to every single topic from publisher
    zsocket_set_subscribe (frontend, "");

    //  Store last instance of each topic in a cache
    zhash_t *cache = zhash_new ();

    //  .split main poll loop
    //  We route topic updates from frontend to backend, and
    //  we handle subscriptions by sending whatever we cached,
    //  if anything:
    while (true) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { backend,  0, ZMQ_POLLIN, 0 }
        };
        if (zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC) == -1)
            break;              //  Interrupted

        //  Any new topic data we cache and then forward
        if (items [0].revents & ZMQ_POLLIN) {
            char *topic = zstr_recv (frontend);
            char *current = zstr_recv (frontend);
            if (!topic)
                break;
```

```
            char *previous = zhash_lookup (cache, topic);
            if (previous) {
                zhash_delete (cache, topic);
                free (previous);
            }
            zhash_insert (cache, topic, current);
            zstr_sendm (backend, topic);
            zstr_send (backend, current);
            free (topic);
        }
        //  .split handle subscriptions
        //  When we get a new subscription, we pull data from the cache:
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *frame = zframe_recv (backend);
            if (!frame)
                break;
            //  Event is one byte 0=unsub or 1=sub, followed by topic
            byte *event = zframe_data (frame);
            if (event [0] == 1) {
                char *topic = zmalloc (zframe_size (frame));
                memcpy (topic, event + 1, zframe_size (frame) - 1);
                printf ("Sending cached topic %s\n", topic);
                char *previous = zhash_lookup (cache, topic);
                if (previous) {
                    zstr_sendm (backend, topic);
                    zstr_send (backend, previous);
                }
                free (topic);
            }
```

```
            zframe_destroy (&frame);
        }
    }
    zctx_destroy (&context);
    zhash_destroy (&cache);
    return 0;
}
```

프록시와 발행자를 순서대로 실행합니다.

```
./lvcache &
./pathopub tcp://localhost:5557
```

그리고 원하는 수의 구독자들을 실행 시 프록시의 5558 포트에 연결합니다.

```
./pathosub tcp://localhost:5558
```

- [옮긴이] 빌드 및 테스트
- "lvcahe.c" 소스코드의 이상과 pathopub 인수가 잘못되어 정상 동작하지 않습니다.

```
 ./lvcache
Sending cached topic 000


 ./pathopub tcp://localhost:5557


 ./pathosub tcp://localhost:5558
```

"lvcahe.c"에서 프론트엔드 소켓에 접속하기 위한 부분이 잘못되었기 때문입니다.

```
// 수정전
    zsocket_connect (frontend, "tcp://*:5557");
```

```
// 수정후
    zsocket_connect (frontend, "tcp://localhost:5557");
```

pathopub 수행 시 인수도 수정 필요합니다.

```
// 수정전
 ./pathopub tcp://localhost:5557
// 수정후
 ./pathopub tcp://*:5557
```

- [옮긴이] 수정된 마지막 값 캐싱 (lvcache.c)

```
//  Last value cache
//  Uses XPUB subscription messages to re-send data

#include "czmq.h"

int main (void)
{
    zctx_t *context = zctx_new ();
    void *frontend = zsocket_new (context, ZMQ_SUB);
    zsocket_connect (frontend, "tcp://localhost:5557");
    void *backend = zsocket_new (context, ZMQ_XPUB);
    zsocket_bind (backend, "tcp://*:5558");

    //  Subscribe to every single topic from publisher
    zsocket_set_subscribe (frontend, "");

    //  Store last instance of each topic in a cache
    zhash_t *cache = zhash_new ();
```

```
    //  .split main poll loop
    //  We route topic updates from frontend to backend, and
    //  we handle subscriptions by sending whatever we cached,
    //  if anything:
    while (true) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { backend,  0, ZMQ_POLLIN, 0 }
        };
        if (zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC) == -1)
            break;              //  Interrupted

        //  Any new topic data we cache and then forward
        if (items [0].revents & ZMQ_POLLIN) {
            char *topic = zstr_recv (frontend);
            char *current = zstr_recv (frontend);
            if (!topic)
                break;
            char *previous = zhash_lookup (cache, topic);
            if (previous) {
                zhash_delete (cache, topic);
                free (previous);
            }
            zhash_insert (cache, topic, current);
            zstr_sendm (backend, topic);
            zstr_send (backend, current);
            free (topic);
        }
        //  .split handle subscriptions
```

```
        //  When we get a new subscription, we pull data from the cache:
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *frame = zframe_recv (backend);
            if (!frame)
                break;
            //  Event is one byte 0=unsub or 1=sub, followed by topic
            byte *event = zframe_data (frame);
            if (event [0] == 1) {
                char *topic = zmalloc (zframe_size (frame));
                memcpy (topic, event + 1, zframe_size (frame) - 1);
                printf ("Sending cached topic %s\n", topic);
                char *previous = zhash_lookup (cache, topic);
                if (previous) {
                    zstr_sendm (backend, topic);
                    zstr_send (backend, previous);
                }
                free (topic);
            }
            zframe_destroy (&frame);
        }
    }
    zctx_destroy (&context);
    zhash_destroy (&cache);
    return 0;
}
```

- [옮긴이] pathopub 인수 변경 후 테스트

```
 ./lvcache
Sending cached topic 045

 ./pathopub tcp://*:5557

 ./pathosub tcp://localhost:5558
Save Roger

 ./pathosub tcp://localhost:5558
Off with his head!
```

각 구독자는 “Save Roger”를 보고 받으며, 탈옥한 죄수 그레고르는 다시 감옥으로 돌아가 따뜻한 우유 한잔과 저녁 식사를 하게 되었습니다. 그는 진정 범죄를 저지르기보다는 관심받기를 원한 것이었습니다.

참고 : 기본적으로 XPUB 소켓은 중복 구독들을 보고하지 않습니다. 이는 XPUB를 XSUB에 그냥 연결할 때 원하는 것입니다. 우리의 예제는 임의의 토픽들을 사용하여 몰래 처리하므로 작동하지 않을 가능성은 백만분의 1입니다. 실제 LVC 프록시에서는 ZMQ_XPUB_VERBOSE 옵션을 사용하며 “6장 - ØMQ 커뮤니티”에서 연습문제로 구현하겠습니다.

0.57 느린 구독자 감지(자살하는 달팽이 패턴)

실제 생활에서 발행-구독 패턴을 사용할 때 발생하는 일반적인 문제는 느린 구독자입니다. 이상적인 세상에서 우리는 발행자에서 구독자로 데이터를 전속력으로 전송합니다. 실제로 구독자 응용프로그램은 종종 인터프리터 개발 언어로 작성되었거나 혹은 많은 작업을 처리하거나 잘못된 코드로 인한 오류로 발행자가 전송하는 데이터를 처리하지 못하는 상황이 발생할 수 있습니다.

- [옮긴이] 인터프리터(interpreter) 개발 언어는 단말기를 통하여 컴퓨터와 대화하면서 작성할 수 있는 프로그래밍 언어로 PYTHON, BASIC, Prolog. LISP, LOGO, R 등이 있습니다.

느린 구독자를 처리하는 방법으로 이상적인 해결책은 구독자를 더 빠르게 만드는 것이지만 응용프로그램을 최적화하는 작업에 많은 시간이 소요될 수 있습니다. 느린 구독자를 개선하기 위한 몇 가지 전통적인 전략들은 다음과 같습니다.

- 발행자의 메시지 대기열에 보관
- 몇 시간 동안 이메일을 읽지 않을 때 Gmail이 하는 일입니다. 그러나 대용량 메시징에서 구독자가 많고 성능상의 이유로 발행자 대기열을 디스크의 데이터를 저장할 수 없는 경우 발행자의 메모리 부족과 충돌이 발생할 수 있습니다.
- 구독자의 메시지 대기열에 보관
- 이것은 훨씬 낫고, 네트워크의 대역폭이 가능하다면 ØMQ가 기본적으로 하는 일입니다. 누군가 메모리가 부족하고 충돌이 발생하면 발행자가 아닌 구독자가 될 것입니다.
- 이것은 "피크(Peaky)" 스트림에 적합하며 메시징이 많아 구독자가 한동안 처리할 수 없지만 메시징이 적어지면 처리 가능합니다. 그러나 일반적으로 너무 느린 구독자에게는 해결책이 아닙니다.
- 잠시 동안 신규 메시지를 수신 중단.
- 수신 편지함의 저장 공간을 초과하는 경우 Gmail에서 수행하는 작업입니다. 신규 메시지는 거부되거나 삭제됩니다. 이것은 발행자의 관점에서는 훌륭한 전략이며, ØMQ에서 발행자가 HWM(High Water Mark)을 설정 시 적용됩니다. 그러나 여전히 느린 구독자를 개선하지는 못합니다. 단지 메시지 전송을 하지 못한 간격(GAP)을 가지게 됩니다.
- 느린 구독자를 연결을 끊어 처벌하기.
- Hotmail에 2주 동안 로그인하지 않았을 때 수행하는 것이며, 내가 15번째 Hotmail 계정을 사용한 이유입니다. 이것은 잔인한 전략으로 구독자가 앉아서 주의를 기울이게 하며, 이상적이지만 ØMQ는 이를 수행하지 않는 이유는 발행자 응용프로그램에는 구독자가 보이지 않기 때문입니다.

어떤 전통적인 전략들도 적합하지 않아, 창의력을 발휘해야 합니다. 발행자와의 연결을 끊는 대신 구독자가 자살하도록 설득하겠습니다. 이것은 자살하는 달팽이 패턴입니다. 구독자가 너무 느리게 실행되고 있음을 감지할 때, 한번 울고 죽게 합니다. "너무 느리다"는 판단은 구성 옵션으로 의미하는 바는 "만약 너무 천천히 당신이 여기 온다면, 내가 알아야 하니까

크게 소리쳐. 내가 고칠 수 있어!"입니다.

구독자는 "너무 느리다"를 감지하는 방법으로 한가지는 메시지를 순서대로 나열하고(순서대로 번호 지정) 발행자에서 HWM을 사용하는 것입니다. 그리고 구독자가 간격(즉, 번호가 연속적이지 않음)을 감지하면 잘못됨을 알게 되며, HWM을 "이 수준에 도달하면 울고 죽기" 수준으로 조정합니다.

이 해결책에는 2가지 문제가 있습니다. 첫째, 발행자들이 많은 경우, 메시지 순서를 지정하는 방법이며 해결책은 각 발행자는 고유한 식별자(ID)를 가지고 순서에 추가하는 것입니다.(PUB ID + SEQUENCE) 둘째, 구독자들이 ZMQ_SUBSCRIBE 필터를 사용하면 정의에 따라 간격이 생기며, 우리의 귀중한 메시지 순서는 소용이 없습니다.

일부 사용 사례들은 필터를 미사용 해야지 메시지 순서가 작동합니다. 그러나 보다 일반적인 해결책은 발행자가 각 메시지에 타임스탬프를 찍는 것입니다. 구독자가 메시지를 받으면 시간을 확인하고 그 차이가 1초 이상이면 "울고 죽기" 작업을 수행하며, 먼저 일부 운영자 컴퓨터 화면에 신호를 보냅니다.

자살하는 달팽이 패턴은 특히 구독자가 자신의 클라이언트들과 서비스 수준 계약(SLA, Service Level Agreements)을 맺고 있고 특정 최대 지연 시간을 보장해야 하는 경우에 효과적입니다. 최대 지연 시간을 보장하기 위해 구독자를 중단하는 것은 건설적인 방법은 아니지만, 가정 설정문(assertion) 모델입니다. 오늘 중단하면 문제는 해결되지만 지연되는 데이터가 하류(구독자들)로 흐르도록 허용하면 문제가 더 넓게 퍼져 많은 손상을 발생하지만 레이더(문제를 감지하는 수단)로 감지하는데 오래 걸릴 수 있습니다.

- [옮긴이] 가정 설정문(Assertions)은 프로그램상에서 실수는 일어난다고 가정하고 이에 대비하여 방지하기 위해 가정 설정문(Assertion)의 조건을 설정하고 참(TRUE)이면 아무것도 하지 않지만, 거짓(FALSE)이면 즉시 프로그램을 정지시키는 방어적 프로그래밍(defensive programming)의 수단입니다.

자살하는 달팽이에 대한 작은 예제입니다.

suisnail.c: 자살하는 달팽이

```
//  Suicidal Snail

#include "czmq.h"
```

```
//  This is our subscriber. It connects to the publisher and subscribes
//  to everything. It sleeps for a short time between messages to
//  simulate doing too much work. If a message is more than one second
//  late, it croaks.

#define MAX_ALLOWED_DELAY   1000    //  msecs

static void
subscriber (void *args, zctx_t *ctx, void *pipe)
{
    //  Subscribe to everything
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5556");

    //  Get and process messages
    while (true) {
        char *string = zstr_recv (subscriber);
        printf("%s\n", string);
        int64_t clock;
        int terms = sscanf (string, "%" PRId64, &clock);
        assert (terms == 1);
        free (string);

        //  Suicide snail logic
        if (zclock_time () - clock > MAX_ALLOWED_DELAY) {
            fprintf (stderr, "E: subscriber cannot keep up, aborting\n");
            break;
```

```
        }
        //  Work for 1 msec plus some random additional time
        zclock_sleep (1 + randof (2));
    }
    zstr_send (pipe, "gone and died");
}

//  .split publisher task
//  This is our publisher task. It publishes a time-stamped message to its
//  PUB socket every millisecond:

static void
publisher (void *args, zctx_t *ctx, void *pipe)
{
    //  Prepare publisher
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");

    while (true) {
        //  Send current clock (msecs) to subscribers
        char string [20];
        sprintf (string, "%" PRId64, zclock_time ());
        zstr_send (publisher, string);
        char *signal = zstr_recv_nowait (pipe);
        if (signal) {
            free (signal);
            break;
        }
        zclock_sleep (1);            //  1msec wait
```

```
    }
}

//  .split main task
//  The main task simply starts a client and a server, and then
//  waits for the client to signal that it has died:

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *pubpipe = zthread_fork (ctx, publisher, NULL);
    void *subpipe = zthread_fork (ctx, subscriber, NULL);
    free (zstr_recv (subpipe));
    zstr_send (pubpipe, "break");
    zclock_sleep (100);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc suisnail.c libzmq.lib czmq.lib

 ./suisnail
13242959186075
13242959186077
13242959186079
13242959186081
...
13242959203828
```

```
13242959203830
13242959203832
E: subscriber cannot keep up, aborting
```

자살하는 달팽이 예제에서 몇 가지 주목할 사항은 다음과 같습니다.

- 여기에 있는 메시지는 밀리초(msec) 단위의 현재 시스템 시각으로만 구성됩니다. 현실적인 응용프로그램에서는 적어도 메시지는 "타임스탬프가 있는 메시지 헤더"와 "데이터가 있는 메시지 본문"으로 구성되어야 합니다.
- 예제에서는 단일 프로세스에서 2개의 스레드인 발행자와 구독자가 있습니다. 실제로는 별도의 프로세스들입니다. 스레드는 편의상 데모를 위하여 사용합니다.

0.58 고속 구독자 (블랙 박스 패턴)

이제 구독자를 더 빠르게 만드는 한 가지 방법을 보겠습니다. 발행-구독의 일반적인 사용 사례는 증권거래소에서 발생하는 시장 데이터와 같은 대용량 데이터 스트림을 배포하는 것입니다. 일반적인 설정에는 발행자가 증권거래소에 연결되어 주가를 여러 구독자들에게 보냅니다. 구독자가 적으면 TCP를 사용할 수 있습니다. 많은 구독자의 경우 안정적인 멀티캐스트, 즉 PGM을 사용합니다.

그림 56 - 단순 블랙박스 패턴

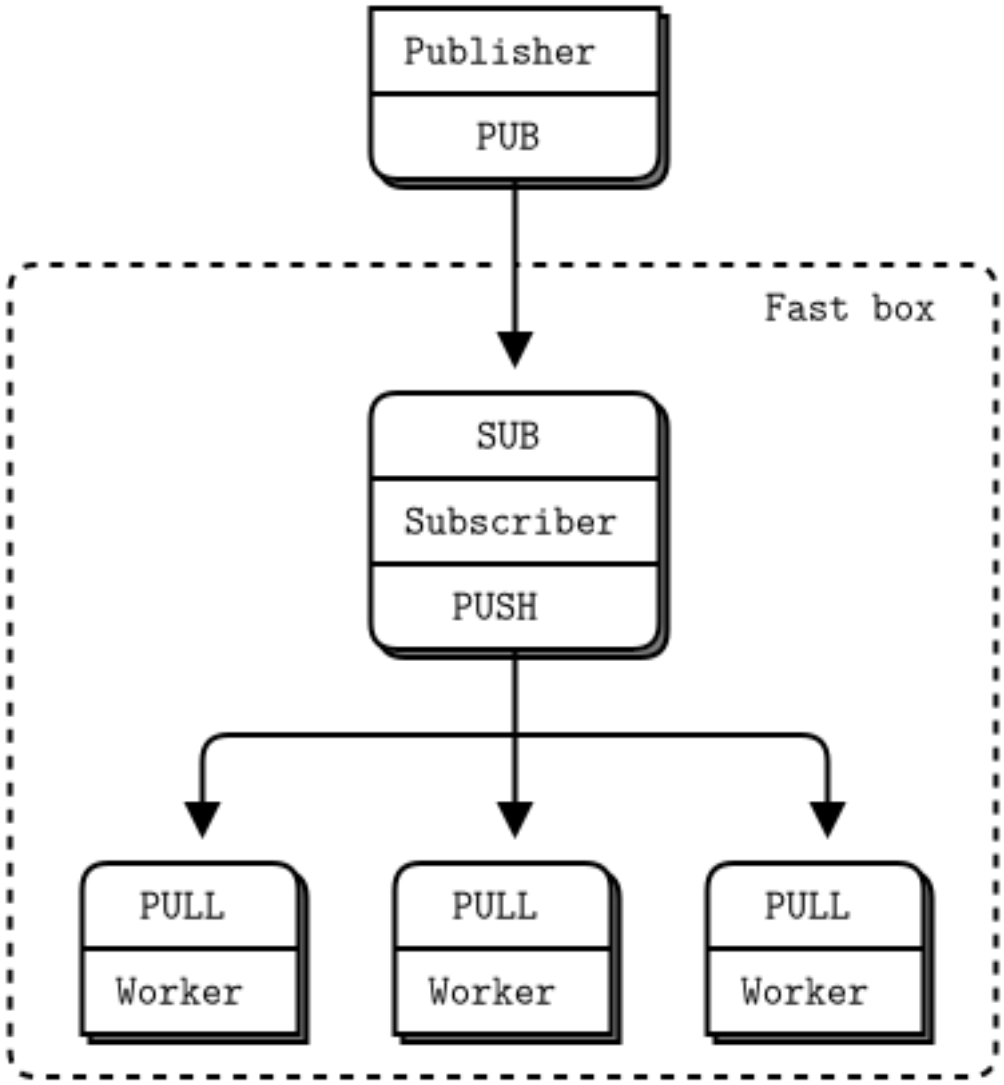


그림 58: The Simple Black Box Pattern

전달 데이터(feed)가 초당 평균 10만 개의 100 바이트 메시지들이 있다고 가정합니다. 시장 데이터를 필터링한 후 구독자에게 전송할 필요가 없게 하면 일반적인 비율입니다. 이제 하루의 데이터(대략 8시간에 250GB)를 저장하고, 시뮬레이션 네트워크(소규모 구독자들의 그룹)에서 재현하기로 합니다. ØMQ 응용프로그램에서 초당 10만 개의 메시지들의 처리는 쉽지만, 더 빠르게 처리하고 싶습니다.

그래서 아키텍처상에서 발행자와 구독자를 위한 각각에 하나의 몫음으로 박스를 설치하였습니다. 설치된 컴퓨터는 8개 코어를 사용하며 12개 발행자를 대응합니다.

그리고 구독자에게 데이터를 전달하면서 다음 2가지에 대하여 주목하게 됩니다.

1. 우리가 메시지에 대한 일정 작업을 수행할 경우, 구독자는 처리 속도가 느려지면서 발행자가 전달하는 메시지들을 처리할 수 없게 됩니다.
2. 신중하게 최적화하고 TCP를 조정한 후에, 발행자와 구독자의 메시지 처리 한계는 초당 6백만개 입니다.

가장 먼저 할 일은 구독자를 멀티 스레드 설계로 분할하여 한 스레드 집합에서는 메시지들의 처리를 하며 다른 스레드는 집합에서는 메시지를 읽게 합니다. 일반적으로 우리는 동일한 방식으로 모든 메시지 처리하고 싶지 않기에 구독자가 메시지에 대한 필터링(토픽(TOPIC))을 수행합니다. 메시지가 어떤 기준과 일치하면 구독자는 작업자를 호출하여 작업을 처리하게 합니다. ØMQ 용어로 작업자 스레드에 메시지를 보내는 것을 의미합니다.

따라서 구독자는 대기열 장치처럼 보입니다. 다양한 소켓을 사용하여 구독자와 작업자들을 연결할 수 있습니다. 단방향 통신과 작업자들 모두 동일하다고 가정하면 PUSH 및 PULL 소켓을 사용하고 모든 라우팅(작업자들에게 분배) 작업을 ØMQ에 위임할 수 있습니다. 이것은 가장 간단하고 빠른 접근 방식입니다.

구독자는 TCP 또는 PGM을 통해 발행자와 통신하며, 구독자는 `inproc://`를 통해 하나의 프로세스에 있는 모두 작업자들(스레드들)과 통신합니다.

그림 57 - 미친 블랙박스 패턴

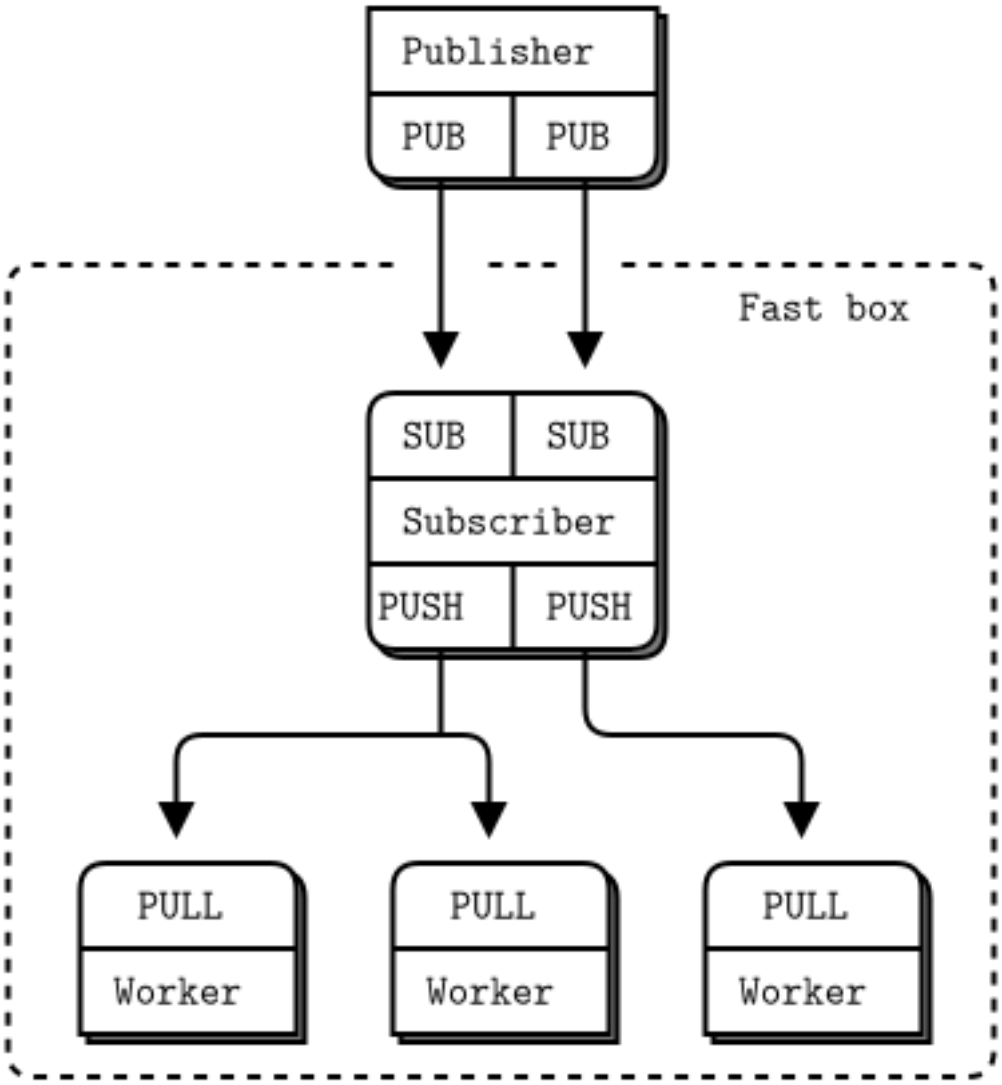


그림 59: Mad Black Box Pattern

이제 한계를 돌파해 봅시다. 구독자 스레드는 CPU의 100%에 도달한 이유는 하나의 스레드로 구성하여 2 이상의 CPU 코어를 사용할 수 없습니다. 단일 스레드는 초당 2백만, 6백만 혹은 이상의 메시지들에서 처리 한계에 도달합니다. 작업을 분할하여 멀티스레드를 통한 병렬로 실행하겠습니다.

이러한 접근은 많은 고성능 메시지 처리 제품에서 사용되고 있으며 샤딩(sharding)이라

합니다. 샤딩을 사용하여 작업을 병렬 및 독립 스트림으로 분할합니다(한 스트림에서 토픽 키의 절반, 다른 스트림에서 토픽 키의 절반). 많은 스트림을 사용할 수 있지만 유휴 CPU 코어가 없으면 성능이 확장되지 않습니다. 이제 하나의 메시지 스트림을 2개의 메시지 스트림으로 분할하는 방법을 보겠습니다.

- [옮긴이] 병렬 샤딩(Parallel Sharding)은 데이터가 서로 공유되지 않도록 완전히 분리하여 독립적으로 동작하는 샤딩(Sharding)을 여러 개로 묶어서 병렬 처리를 통해 성능 및 확장성을 확보하는 방식입니다.

2개의 메시지 스트림들에 대하여 최고 속도로 작업하기 위하여 ØMQ을 다음과 같이 구성합니다.

- 하나가 아닌 2개의 I/O 스레드들.
- 2개의 네트워크 인터페이스(NIC), 구독자 당 하나씩.
- 개별 I/O 스레드는 지정된 네트워크 인터페이스(NIC)에 바인딩
- 2개의 구독자 스레드는 지정된 CPU 코어들에 바인딩.
- 2개의 SUB 소켓은 구독자 스레드당 하나씩 지정.
- 나머지 CPU 코어들은 작업자 스레드들에 할당.
- 작업자 스레드들은 가입자의 2개의 PUSH 소켓들에 연결

이상적으로는 아키텍처에서 완전히 부하 처리가 가능한 스레드 수를 코어 수와 일치시키려고 했습니다. 스레드가 코어 및 CPU 주기를 놓고 싸우기(Time sharing) 시작하면 스레드를 추가하는 비용이 메시지 처리량보다 큽니다(더 많은 메시지 처리하기 위해 더 많은 CPU 코어 구매). 더 많은 I/O 스레드를 생성하는 것에는 이점이 없습니다.

- [옮긴이] 고속 구독자에 대한 테스트를 위한 예제입니다.

hssub.c : 고속 구독자

```
// high speed subscriber


#include "czmq.h"
#define NBR_WORKERS 3
```

```
//  .split publisher task
static void
publisher (void *args, zctx_t *ctx, void *pipe)
{
    //  Prepare publisher
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");

    while (true) {
        //  Send current clock (msecs) to subscribers
        char string [20];
        sprintf (string, "%" PRId64, zclock_time ());
        zstr_send (publisher, string);
        char *signal = zstr_recv_nowait (pipe);
        if (signal) {
            free (signal);
            break;
        }
        zclock_sleep (1);            //  1msec wait
    }

}

// subscriber task
static void
subscriber (void *args, zctx_t *ctx, void *pipe)
{
    //  Subscribe to everything
```

```
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5556");

    //  Socket to send messages to
    void *sender = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_bind (sender, "tcp://*:5557");

    //  Get and send messages
    while (true) {
        char *topic = zstr_recv (subscriber);
        zstr_send(sender, topic);
        free (topic);
    }
}
// worker task
static void
worker(void *args, zctx_t *ctx, void *pipe)
{
    //  Subscribe to everything
    void *worker = zsocket_new (ctx, ZMQ_PULL);
    zsocket_connect (worker, "tcp://localhost:5557");

    //  Get and send messages
    while (true) {
        char *string = zstr_recv (worker);
        printf ("[W%Id]Receive: [%s]\n", (intptr_t)args, string);
        free (string);
    }
```

```
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *pubpipe = zthread_fork (ctx, publisher, NULL);
    void *subpipe = zthread_fork (ctx, subscriber, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++){
        zthread_fork (ctx, worker, (void *)(intptr_t)worker_nbr);
        //zclock_sleep(100);
    }
    free (zstr_recv (subpipe));
    zstr_send (pubpipe, "break");
    zclock_sleep (100);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트
- 3개의 작업자들에게 라운드로빈 형태로 메시지가 전달되는 것을 확인 가능합니다.

```
 cl -EHsc hssub.c libzmq.lib czmq.lib

 ./hssub
[W0]Receive: [13242984839620]
[W1]Receive: [13242984839622]
[W2]Receive: [13242984839624]
[W0]Receive: [13242984839626]
[W1]Receive: [13242984839628]
```

```
[W2]Receive: [13242984839630]
[W0]Receive: [13242984839632]
[W1]Receive: [13242984839634]
[W2]Receive: [13242984839636]
...
```

0.59 신뢰할 수 있는 발행-구독 (복제 패턴)

좀 더 큰 작업 예제로, 신뢰할 수 있는 발행-구독 아키텍처를 만드는 문제를 보겠습니다. 우리는 이것을 단계적으로 개발하며 목표는 일련의 응용프로그램들이 일부 공통 상태를 공유하게 합니다. 기술적 도전 과제들은 다음과 같습니다.

- 수천 또는 수만 개의 클라이언트 응용프로그램들이 있습니다.
- 임의로 네트워크에 가입하고 탈퇴합니다.
- 이러한 응용프로그램들은 하나의 최종 일관성 상태를 공유해야 합니다.
- 모든 응용프로그램은 언제든지 상태를 변경할 수 있습니다.

상태의 변경이 상당히 적은 양이라고 가정하고 실시간 처리 목표는 없습니다. 전체 상태는 메모리에 저장하고, 일부 사용 사례는 다음과 같습니다.

- 클라우드 서버들의 그룹에서 공유하는 구성.
- 플레이어 그룹이 공유하는 일부 게임 상태.
- 환율 데이터가 실시간으로 변경되고 응용프로그램에서 가용함.

0.59.1 중앙집중형과 분산형

우리가 해야 할 첫 번째 결정은 중앙 서버 혹은 분산 서버로 작업할지 여부입니다. 결과적인 설계에서 큰 차이를 만들며, 장단점은 다음과 같습니다.

- 개념적으로 중앙 서버로 작업하는 것이 네트워크가 태생적으로 비대칭형이기 때문에 이해하기 더 쉽습니다. 중앙 서버를 사용하면 검색, 바인딩과 연결 등과 관련된 모든 질문들을 피할 수 있습니다.

- 일반적으로 완전히 분산된 아키텍처는 기술적으로 더욱 어렵지만 의외로 간단한 통신 규약으로 끝납니다. 즉, 각 노드가 올바른 방식으로 서버 및 클라이언트 역할을 해야 하며 이는 섬세합니다. 올바르게 수행하면 중앙 서버를 사용하는 것보다 결과는 더 간단합니다. "4장 - 신뢰할 수 있는 요청-응답 패턴"의 프리랜서 패턴에서 보았습니다.
- 중앙 서버는 대량 데이터 사용 사례에서 병목 현상을 겪을 수 있습니다. 초당 수백만 개의 메시지를 처리해야 한다면, 당장 분산 처리 환경을 목표로 해야 합니다.
- 역설적이게도 중앙집중식 아키텍처는 분산형 아키텍처보다 더 쉽게 더 많은 노드로 확장됩니다. 즉, 하나의 서버에 10,000개의 노드들을 연결하는 것이 노드들 간에 상호 연결하는 것보다 쉽습니다.

그래서 복제 패턴의 경우 중앙 서버를 통해 상태 변경을 전송하면 일련의 클라이언트들의 응용프로그램에서 사용합니다.

0.59.2 상태를 키-값 쌍으로 표시

우리는 한 번에 하나씩 문제를 해결하면서 단계적으로 복제(Clone)를 개발할 것입니다. 먼저 일련의 클라이언트들에서 공유 상태를 변경하는 방법을 보겠습니다. 우리는 상태를 표현하고 변경한 방법을 결정해야 합니다. 가장 단순한 형식은 키-값 저장소(해시 테이블)로 하나의 키-값 쌍은 공유 상태를 변경하는 기본 단위가 됩니다.

"1장 - 기본"에서 날씨 서버 및 클라이언트 예제로 간단한 발행-구독 패턴으로 구현하였습니다. 이제 서버(wuserver)를 변경하여 키-값 쌍을 보내고 클라이언트에서 해시 테이블에 저장하겠습니다. 이를 통해 고전적인 발행-구독 모델을 사용하여 한 서버에서 일련의 클라이언트들로 변경정보를 보낼 수 있습니다.

변경정보는 신규 키-값 쌍이거나 기존 키의 수정된 값 혹은 삭제된 키입니다. 지금은 전체 저장소가 메모리에 저장할 수 있고 응용프로그램들이 해시 테이블이나 사전을 사용하는 것처럼 키로 접근한다고 가정합니다. 더 큰 저장소와 일종의 지속성이 필요한 경우 상태를 데이터베이스에 저장할 수 있지만 여기서는 관련이 없습니다.

다음은 서버 코드입니다.

clonesrv1.c: 복제 서버, 모델 1

```
//  Clone server Model One

#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and publisher socket
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");
    zclock_sleep (200);

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    while (!zctx_interrupted) {
        //  Distribute as key-value message
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_store   (&kvmsg, kvmap);
    }
    printf (" Interrupted\n%d messages out\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

다음은 클라이언트 코드입니다.

clonecli1.c: 복제 클라이언트, 모델 1

```
//  Clone client Model One

#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and updates socket
    zctx_t *ctx = zctx_new ();
    void *updates = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (updates, "");
    zsocket_connect (updates, "tcp://localhost:5556");

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;

    while (!zctx_interrupted) {
        kvmsg_t *kvmsg = kvmsg_recv (updates);
        if (!kvmsg)
            break;          //  Interrupted
        kvmsg_store (&kvmsg, kvmap);
        sequence++;
    }
    printf (" Interrupted\n%d messages in\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] clonecli.c에서 `while(true)`에 대하여 사용자 인터럽트를 받을 수 있도록 `while (!zctx_interrupted)`로 변경하였습니다.

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc clonesrv1.c libzmq.lib czmq.lib
 cl -EHsc clonecli1.c libzmq.lib czmq.lib

 ./clonesrv1
 Interrupted
14938939 messages out

 ./clonecli1
 Interrupted
1119580 messages in
```

Figure 58 - 상태 변경정보 발행

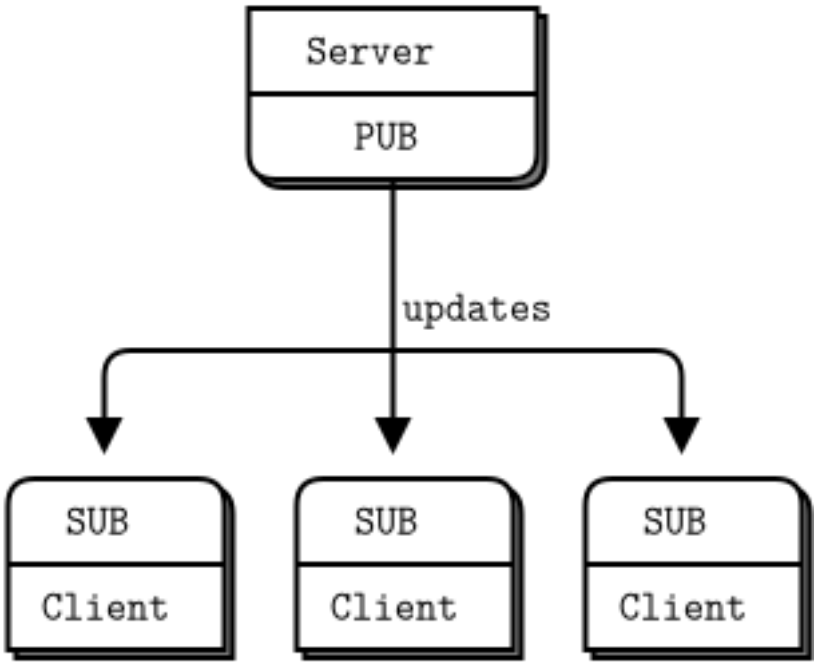


그림 60: Publishing State Updates

첫 번째 모델에 대한 몇 가지 주목할 점은 다음과 같습니다.

- 모든 어려운 작업은 `kvmsg` 클래스에서 수행되었습니다. 이 클래스는 키-값 메시지 객체들로 동작하며, 멀티파트(multipart) ØMQ 메시지는 3개의 프레임으로 구성됩니다 : 키(ØMQ 문자열), 순서 번호(네트워크 바이트 순서의 64 비트 값(`int64_t`)), 바이너리 본문(모든 항목을 포함)
- 서버는 임의의 4자리 키(0~9999)로 메시지를 생성하므로, 크지만 방대하지는 않은 해시 테이블(1만 개 항목들)을 시뮬레이션할 수 있습니다.
- 현재 버전에서는 삭제를 구현하지 않습니다. 모든 메시지는 삽입 또는 변경입니다.
- 서버는 소켓을 바인딩하고 200 밀리초 동안 일시 중지합니다. 이는 구독자가 서버의 소켓에 연결할 때 메시지를 유실하는 느린 참여 증후군(slow joiner syndrome)을 방지합니다. 이후 버전의 복제 코드에서는 제거하겠습니다.
- 소켓에 대하여 코드상에서 발행자와 구독자라는 용어를 사용할 것입니다. 이것은 나중에 다른 일을 하는 다중 소켓들로 작업할 때 도움이 됩니다.

다음은 현재 동작하는 가장 간단한 형식의 kvmsg 클래스입니다.

kvsimple.c : 키-값 메시지 클래스

```
//  kvsimple class - key-value message class for example applications

#include "kvsimple.h"
#include "zlist.h"

//  Keys are short strings
#define KVMSG_KEY_MAX   255

//  Message is formatted on wire as 3 frames:
//  frame 0: key (ØMQ string)
//  frame 1: sequence (8 bytes, network order)
//  frame 2: body (blob)
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_BODY      2
#define KVMSG_FRAMES    3

//  The kvmsg class holds a single key-value message consisting of a
//  list of 0 or more frames:

struct _kvmsg {
    //  Presence indicators for each frame
    int present [KVMSG_FRAMES];
    //  Corresponding ØMQ message frames, if any
    zmq_msg_t frame [KVMSG_FRAMES];
    //  Key, copied into safe C string
    char key [KVMSG_KEY_MAX + 1];
```

```
};

//  .split constructor and destructor
//  Here are the constructor and destructor for the class:

//  Constructor, takes a sequence number for the new kvmsg instance:
kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    kvmsg_set_sequence (self, sequence);
    return self;
}

//  zhash_free_fn callback helper that does the low level destruction:
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  Destroy message frames if any
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);
```

```
        //  Free object itself
        free (self);
    }
}

//  Destructor
void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}

//  .split recv method
//  This method reads a key-value message from socket, and returns a new
//  {{kvmsg}} instance:

kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
    kvmsg_t *self = kvmsg_new (0);

    //  Read all frames off the wire, reject if bogus
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
```

```
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_msg_recv (&self->frame [frame_nbr], socket, 0) == -1) {
            kvmsg_destroy (&self);
            break;
        }
        //  Verify multipart framing
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsocket_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    return self;
}

//  .split send method
//  This method sends a multiframe key-value message to a socket:

void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
```

```
        zmq_msg_t copy;
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_msg_send (&copy, socket,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}

//  .split key methods
//  These methods let the caller get and set the message key, as a
//  fixed string and as a printf formatted string:

char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
```

```
    else
        return NULL;
}

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}

void
kvmsg_fmt_key (kvmsg_t *self, char *format, ...)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}
```

```
//  .split sequence methods
//  These two methods let the caller get and set the message sequence number:

int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}

void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
```

```
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}

//  .split message body methods
//  These methods let the caller get and set the message body as a
//  fixed string and as a printf formatted string:

byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
```

```
}

void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}

void
kvmsg_fmt_body (kvmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}

//  .split size method
//  This method returns the body size of the most recently read message,
```

```
//  if any exists:

size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}

//  .split store method
//  This method stores the key-value message into a hash map, unless
//  the key and value are both null. It nullifies the {{kvmsg}} reference
//  so that the object is owned by the hash map, not the caller:

void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (self->present [FRAME_KEY]
        &&  self->present [FRAME_BODY]) {
            zhash_update (hash, kvmsg_key (self), self);
            zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
        }
```

```
        *self_p = NULL;
    }
}

//  .split dump method
//  This method prints the key-value message to stderr for
//  debugging and tracing:

void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self));
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd] ", size);
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}
```

```
//  .split test method
//  It's good practice to have a self-test method that tests the class; this
//  also shows how it's used in applications:

int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg: ");

    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "inproc://kvmsg_selftest");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "inproc://kvmsg_selftest");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  Test send and receive of simple message
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
```

```
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  Shutdown and destroy all objects
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

- [옮긴이] “kvsimpler.c”의 `kvmsg_test()`에서 ipc를 사용하고 있으나 윈도우 환경에서는 동작할 수 없어 inproc로 변경하여 테스트를 수행합니다.(inproc는 윈도우 및 Linux에서 동작 가능)

```
// 변경전
    int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
// 변경후
    int rc = zmq_bind (output, "inproc://kvmsg_selftest");
    assert (rc == 0);
```

```
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "inproc://kvmsg_selftest");
```

- [옮긴이] "kvsimpler"에 대한 테스트를 수행하기 위한 "kvsimtest.c"는 다음과 같습니다.

```
#include "kvsimple.c"

void main()
{
    kvmsg_test(1);
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc kvsimtest.c libzmq.lib czmq.lib

 ./kvsimtest
 * kvmsg: [seq:1][key:key][size:4] 626F6479
[seq:1][key:key][size:4] 626F6479
OK
```

나중에 실제 응용프로그램에서 동작하는 보다 정교한 kvmsg 클래스를 만들겠습니다.

서버와 클라이언트 모두 해시 테이블을 유지하지만, 첫 번째 모델은 서버를 시작하기 전에 모든 클라이언트들을 구동하고 클라이언트들이 충돌하지 않는 경우에만 제대로 작동합니다. 그것은 매우 인위적입니다.

0.59.3 대역 외 스냅샷 얻기

이제 두 번째 문제가 있습니다. 늦게 가입하는 클라이언트들과 혹은 장애조치 이후 재시작한 클라이언트들을 처리하는 방법입니다.

늦게 가입(또는 재시작)한 클라이언트들에 대하여 서버가 이미 발행한 메시지들을 받을 수 있게 하려면 서버 상태의 스냅샷을 얻어야 합니다. “메시지”의 의미를 “순서화된 키-값 쌍”으로 줄인 것처럼 “상태 정보”의 의미도 “해시 테이블” 줄일 수 있습니다. 서버 상태 정보를 얻기 위해 클라이언트는 DEALER 소켓을 열고 명시적으로 요청합니다.

이 작업을 수행하기 위해 타이밍 문제를 해결해야 합니다. 상태 스냅샷을 얻기는 일정 시간이 걸리며 스냅샷이 크면 상당히 오래 걸릴 수 있습니다. 스냅샷에 올바르게 변경정보를 적용해야 하지만 서버는 변경정보 전송을 언제 시작할지 모릅니다. 한 가지 방법은 클라이언트가 구독을 시작하고 첫 번째 변경정보를 받은 다음 “변경정보 N에 대한 상태(state for update N)”를 요청하는 것입니다. 이렇게 하려면 서버가 각 변경정보에 대해 하나의 스냅샷을 저장해야 하므로 실용적이지 않습니다.

그림 59 - 상태 복제

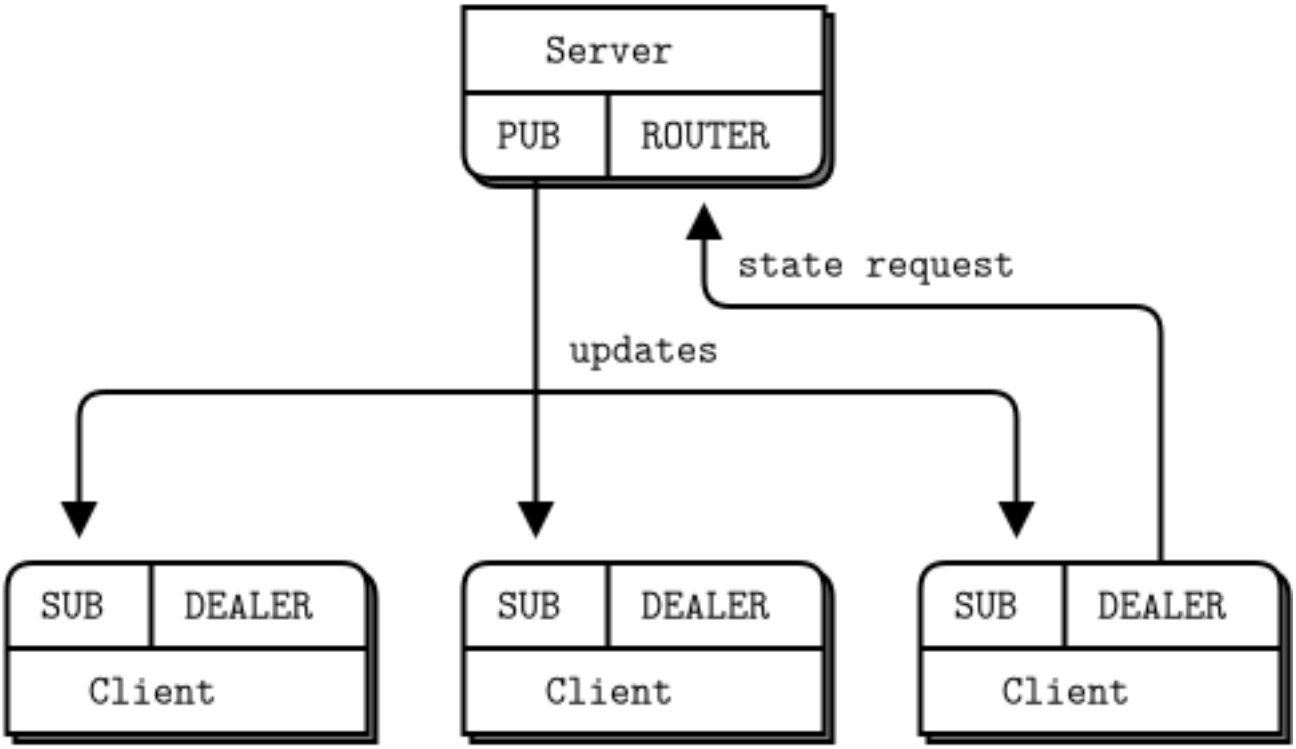


그림 61: State Replication

따라서 클라이언트의 동기화는 다음과 같이 수행합니다.

- 클라이언트는 먼저 변경정보들을 구독한 다음 상태 요청을 하면, 상태가 이전 오래된 변경정보보다 최신이 됩니다.
- 클라이언트는 서버가 상태로 응답할 때까지 대기하고 모든 변경정보를 대기열에 넣습니다. 변경정보를 대기열에 넣기만 하고 처리하지 않도록 하는 것은 대기열을 읽지 않음으로써 가능합니다 : ØMQ는 변경정보들을 소켓 대기열에 넣어 보관합니다.
- 클라이언트가 상태 변경을 받으면 대기열에 넣어 두었던 변경정보 읽기를 다시 시작합니다. 그러나 상태 변경 시점보다 오래된 변경정보들은 모두 삭제됩니다. 따라서 상태 변경 전에 대기열에 최대 200 개의 변경정보가 포함된 경우 클라이언트는 최대 201개의 변경정보를 삭제합니다.
- 그런 다음 클라이언트는 자체 상태 스냅샷에 변경정보를 적용합니다.

다음의 서버 예제는 ØMQ의 자체 내부 대기열을 이용하는 단순한 모델입니다.

clonesrv2.c : 복제 서버, 모델 2

```
//  Clone server - Model Two

//  Lets us build this source without creating a library
#include "kvsimple.c"

static int s_send_single (const char *key, void *data, void *args);
static void state_manager (void *args, zctx_t *ctx, void *pipe);

int main (void)
{
    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
```

```
    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    //  Start state manager and wait for synchronization signal
    void *updates = zthread_fork (ctx, state_manager, NULL);
    free (zstr_recv (updates));

    while (!zctx_interrupted) {
        //  Distribute as key-value message
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_send     (kvmsg, updates);
        kvmsg_destroy (&kvmsg);
    }
    printf (" Interrupted\n%d messages out\n", (int) sequence);
    zctx_destroy (&ctx);
    return 0;
}

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
} kvroute_t;

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
```

```
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  Send identity of recipient first
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}

//  .split state manager
//  The state manager task maintains the state and handles requests from
//  clients for snapshots:

static void
state_manager (void *args, zctx_t *ctx, void *pipe)
{
    zhash_t *kvmap = zhash_new ();

    zstr_send (pipe, "READY");
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");

    zmq_pollitem_t items [] = {
        { pipe, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
```

```
    int64_t sequence = 0;       //  Current snapshot version number
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1 && errno == ETERM)
            break;              //  Context has been shut down

        //  Apply state update from main thread
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (pipe);
            if (!kvmsg)
                break;          //  Interrupted
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        //  Execute state snapshot request
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted

            //  Request is in second frame of message
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: bad request, aborting\n");
                break;
            }
            //  Send state snapshot to client
```

```
            kvroute_t routing = { snapshot, identity };

            //  For each entry in kvmap, send kvmsg to client
            zhash_foreach (kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            printf ("Sending state shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    zhash_destroy (&kvmap);
}
```

clonecli2.c : 복제 클라이언트, 모델 2

```
//  Clone client - Model Two

//  Lets us build this source without creating a library
#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and subscriber
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
```

```
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5557");

    zhash_t *kvmap = zhash_new ();

    //  Get state snapshot
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (true) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("Received snapshot=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  Done
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    //  Now apply pending updates, discard out-of-sequence messages
    while (!zctx_interrupted) {
        kvmsg_t *kvmsg = kvmsg_recv (subscriber);
        if (!kvmsg)
            break;          //  Interrupted
        if (kvmsg_sequence (kvmsg) > sequence) {
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
```

```
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] clonecli2에서 “ICANHAZ?” 메시지를 clonesrv2로 보내면 서버는 해시 테이블에 저장한 변경정보들을 `s_send_single()`통하여 모두 전송하고 “KTHXBAI” 메시지를 전송합니다. clonecli2에서 “KTHXBAI”을 받으면 해당 sequence를 기준으로 clonesrv2에서 발행된 변경정보의 sequence와 비교하여 이후의 것들만 받아 해시 테이블에 보관합니다.(이전 정보는 폐기)

- [옮긴이] “ICANHAZ?”는 “I Can has?”(가져도 될까요?)이며 “KTHXBAI”는 “Ok, Thank you, goodbye”(예, 고마워요, 잘 있어요)를 의미합니다.

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc clonesrv2.c libzmq.lib czmq.lib
 cl -EHsc clonecli2.c libzmq.lib czmq.lib

 ./clonesrv2
Sending state shapshot=1401876

 ./clonecli2
Received snapshot=1401876
```

2개의 프로그램에서 몇 가지 주목할 사항은 다음과 같습니다.

- 서버는 2가지 작업을 수행합니다. 하나의 스레드(clonesrv2)는 변경정보(무작위로)를 생성하여 메인 PUB 소켓으로 보내고, 다른 스레드(state_manager)는 파이프(PAIR)에 변경정보를 받아 해시 테이블에 보관하고 ROUTER 소켓에서 클라이언트의 상태 요청들을 처리합니다. 메인 스레드(clonesrv2)와 자식 스레드(state_manager)는 `inproc://` 연결을 통해 PAIR 소켓을 통해 통신합니다.
- 클라이언트는 정말 간단합니다. C 언어에서는 약 50 줄의 코드로 구성됩니다. 많은 무거운 작업이 kvmsg(kvsimple) 클래스에서 수행됩니다. 그럼에도 불구하고 기본 복제 패턴은 처음에 보였던 것보다 구현하기가 쉽습니다.
- 우리는 상태를 직렬화하기 위해 화려한 것을 사용하지 않습니다. 해시 테이블은 일련의 kvmsg 객체를 보유하고 서버는 이러한 객체를 일련의 메시지로 클라이언트 요청(ICANHAZ) 시에 보냅니다. 여러 클라이언트들이 한 번에 상태를 요청하면, 각각 다른 스냅샷을 얻습니다.
- 우리는 클라이언트에 정확히 하나의 서버가 있다고 가정하며 서버가 실행 중이어야 합니다. 서버에 장애가 발생하면 어떻게 되는지는 여기서 다루지 않습니다.

현재, 2개 프로그램은 실제 작업을 수행하지 않지만 상태를 올바르게 동기화합니다. 깔끔한 예제로 PAIR-PAIR, PUB-SUB 및 ROUTER-DEALER와 같은 다양한 패턴을 혼합하는 방법을 다루었습니다.

0.59.4 클라이언트들로부터 변경정보 재발행

두 번째 모델에서 키-값 저장소에 대한 변경 사항은 서버 자체적으로 나왔습니다. 중앙집중식 모델에서는 유용하며, 예를 들어 서버에 중앙 구성 파일을 두고 각 노드들의 로컬 저장소에 사용하기 위해 배포합니다. 더 흥미로운 모델은 서버가 아닌 클라이언트로부터 변경정보들을 받는 것이며, 이럴 경우 서버는 상태 비저장(stateless) 브로커가 되며, 몇 가지 이점을 제공합니다.

- 우리는 서버의 안정성에 대해 덜 걱정합니다. 서버에 충돌이 발생하면 재시작하여 클라이언트에 신규 값을 제공할 수 있습니다.
- 키-값 저장소를 사용하여 활성 동료 간에 지식(저장소)을 공유할 수 있습니다.

클라이언트에서 다시 서버로 변경정보를 보내기 위해, 다양한 소켓 패턴을 사용할 수 있지만 가장 간단한 솔루션은 PUSH-PULL 조합입니다.

클라이언트들이 서로에게 직접 변경정보를 발행하지 않는 이유는 지연 시간(latency)이 줄어들지만 일관성(consistency)을 보장할 수 없기 때문입니다. 변경정보를 받는 사람에 따라 변경정보 순서를 변경한다면 일관된 공유 상태를 얻을 수 없습니다. 서로 다른 키들을 변경하는 두 개의 클라이언트들이 있다고 가정하면 잘 작동하지만 두 개의 클라이언트들이 거의 동시에 동일한 키를 변경하려고 하면 동일한 키에 다른 값으로 변경되어 일관성이 없어집니다.

여러 위치에서 동시 변경이 발생할 때 일관성을 유지하기 위한 몇 가지 전략들이 있습니다. 우리는 모든 변경 사항을 중앙 집중화하는 접근 방식을 사용할 것입니다. 클라이언트가 변경하는 정확한 시간에 관계없이, 모든 변경 사항은 서버를 통해 전달되므로 변경정보를 받는 순서에 따라 단일 순서가 적용됩니다.

그림 60 - 변경정보 재발행

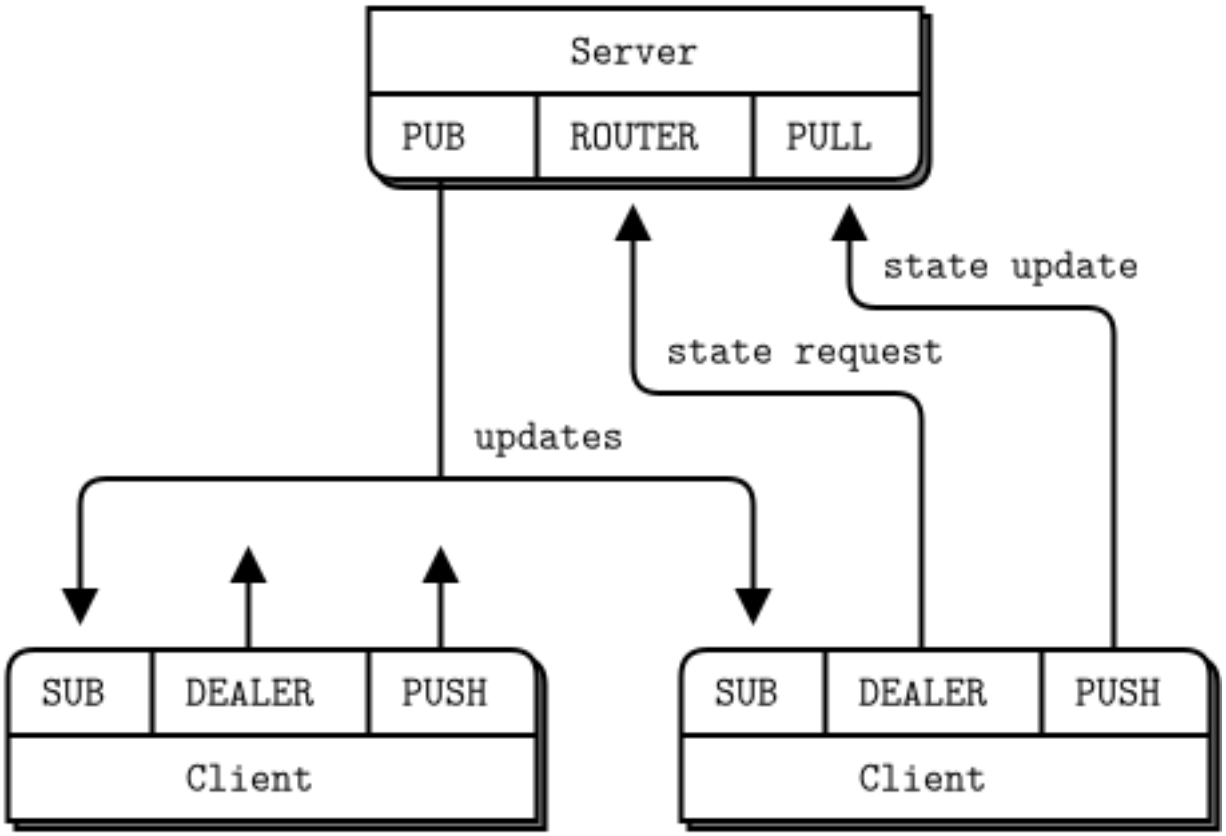


그림 62: Republishing Updates

모든 변경 사항을 조정하기 위해, 서버는 모든 변경정보에 고유한 순서 번호를 추가할 수 있습니다. 고유한 순서 번호를 통해 클라이언트는 네트워크 정체(cogestion) 및 대기열 오버플로우(overflow)를 포함하여 심각한 장애를 감지할 수 있습니다. 클라이언트는 수신 메시지 스트림에 구멍(순서 번호가 연결되지 않는 부분)이 있음을 발견하면 조치를 취할 수 있습니다. 클라이언트가 서버에 접속하여 누락된 메시지를 요청하는 것이 합리적으로 보이지만 실제로는 유용하지 않습니다. 만약 구멍 네트워크 처리 부하에 따른 스트레스로 인한 것이라면, 네트워크에 더 많은 스트레스를 가하면 상황이 악화됩니다. 클라이언트가 할 수 있는 일은 사용자에게 "계속할 수 없음"이라고 경고하고 중지하고, 누군가가 문제의 원인을 수동으로 확인할 때까지 재시작하지 않는 것입니다.

클라이언트에서 상태 변경들을 생성하겠습니다. 다음은 서버의 코드입니다.

clonesrv3.c : 복제 서버, 모델 3

```
//  Clone server - Model Three

//  Lets us build this source without creating a library
#include "kvsimple.c"

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
} kvroute_t;

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  Send identity of recipient first
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}

int main (void)
{
    //  Prepare our context and sockets
```

```
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    //  .split body of main task
    //  The body of the main task collects updates from clients and
    //  publishes them back out to clients:

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  Apply state update sent from client
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  Interrupted
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
```

```
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: publishing update %5d\n", (int) sequence);
        }
        //  Execute state snapshot request
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted

            //  Request is in second frame of message
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: bad request, aborting\n");
                break;
            }
            //  Send state snapshot to client
            kvroute_t routing = { snapshot, identity };

            //  For each entry in kvmap, send kvmsg to client
            zhash_foreach (kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            printf ("I: sending shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
```

```
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    printf (" Interrupted\n%d messages handled\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}
```

다음은 클라이언트의 코드입니다.

clonecli3: Clone client, Model Three in C

```
//  Clone client - Model Three

//  Lets us build this source without creating a library
#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and subscriber
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5557");
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");
```

```
    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  .split getting a state snapshot
    //  We first request a state snapshot:
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (true) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: received snapshot=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  Done
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    //  .split processing state updates
    //  Now we wait for updates from the server and every so often, we
    //  send a random key-value update to the server:

    int64_t alarm = zclock_time () + 1000;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 } };
        int tickless = (int) ((alarm - zclock_time ()));
        if (tickless < 0)
            tickless = 0;
```

```
        int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Context has been shut down

        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (subscriber);
            if (!kvmsg)
                break;          //  Interrupted

            //  Discard out-of-sequence kvmsgs, incl. heartbeats
            if (kvmsg_sequence (kvmsg) > sequence) {
                sequence = kvmsg_sequence (kvmsg);
                kvmsg_store (&kvmsg, kvmap);
                printf ("I: received update=%d\n", (int) sequence);
            }
            else
                kvmsg_destroy (&kvmsg);
        }
        //  If we timed out, generate a random kvmsg
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
    }
    printf (" Interrupted\n%d messages in\n", (int) sequence);
```

```
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] clonecli3에서 1초마다 보내는 변경정보를 clonesrv3은 클라이언트들에 발행하며 clonecli3 중지하면 clonesrv3도 더 이상 변경정보를 발행하지 않습니다.

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc clonesrv3.c libzmq.lib czmq.lib
 cl -EHsc clonecli3.c libzmq.lib czmq.lib

 ./clonesrv3
I: sending shapshot=0
I: publishing update     1
I: publishing update     2
I: publishing update     3
I: sending snapshot=3
I: publishing update     4
I: publishing update     5
...

 ./clonecli3
I: received snapshot=0
I: received update=1
I: received update=2
I: received update=3
I: received update=4
I: received update=5
...
```

```
 ./clonecli3
I: received snapshot=3
I: received update=4
I: received update=5
...
```

세 번째 설계에서 몇 가지 주목할 점은 다음과 같습니다.

- 서버가 단일 작업(clonesrv3)으로 축소되었으며 클라이언트로부터 수신되는 변경정보를 위한 PULL 소켓, 상태 요청 및 스냅샷 전달을 위한 위한 ROUTER 소켓, 클라이언트에게 변경정보 송신을 위한 PUB 소켓을 관리합니다.
- 클라이언트는 간단한 무지연(tickless) 타이머를 사용하여 1초에 한 번씩 서버에 무작위 업데이트를 보냅니다. 실제 구현에서는 응용프로그램 코드에서 업데이트를 유도합니다.

0.59.5 하위트리와 작업

클라이언트들의 수가 증가하면 공유 스토어의 규모도 커질 것입니다. 모든 클라이언트에게 모든 스냅샷을 보내는 것은 합리적이지 않습니다. 이것은 발행-구독의 고전적인 이야기입니다 : 클라이언트들의 수가 매우 적으면 모든 메시지들을 모든 클라이언트들에게 보낼 수 있습니다. 아키텍처를 커지면 비효율적이 되며 클라이언트는 다양한 영역에 전문화되어 있습니다.

따라서 서버의 공유 저장소로 작업할 때에도 일부 클라이언트들은 해당 저장소의 일부에서만 작업하기를 원하여, 일부 저장소를 하위트리(subtree)라고 합니다. 클라이언트는 상태 요청 시 하위 트리를 요청해야 하며 변경정보들을 구독할 때 동일한 하위트리를 지정해야 합니다.

트리에 대한 몇 가지 공통적인 문법들이 있으며, 하나는 경로 계층(path hierarchy)이고 다른 하나는 토픽 트리(topic tree)입니다. 다음과 같이 보입니다.

- 경로 계층 : /some/list/of/paths
- 토픽 트리 : some.list.of.topics

예제에서는 경로 계층을 사용하고, 클라이언트와 서버를 확장하여 클라이언트가 단일 하위트리로 작업하게 합니다. 단일 하위트리로 작업하는 방법을 알게 되면 용도에 따라 다중 하위트리들을 처리하도록 확장할 수 있습니다.

다음은 모델 3의 작은 변형으로 하위트리를 구현하는 서버의 코드입니다.

clonesrv4.c : 복제 서버, 모델 4

```
//  Clone server - Model Four

//  Lets us build this source without creating a library
#include "kvsimple.c"

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
    char *subtree;          //  Client subtree specification
} kvroute_t;

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  Send identity of recipient first
        zframe_send (&kvroute->identity,
```

```
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}

//  The main task is identical to clonesrv3 except for where it
//  handles subtrees.
//  .skip

int main (void)
{
    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
```

```
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  Apply state update sent from client
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  Interrupted
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_dump(kvmsg);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: publishing update %5d\n", (int) sequence);
        }
        //  Execute state snapshot request
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted
            //  .until
            //  Request is in second frame of message
            char *request = zstr_recv (snapshot);
            char *subtree = NULL;
            if (streq (request, "ICANHAZ?")) {
                free (request);
                subtree = zstr_recv (snapshot);
            }
            //  .skip
            else {
                printf ("E: bad request, aborting\n");
```

```
                break;
            }
            //  .until
            //  Send state snapshot to client
            kvroute_t routing = { snapshot, identity, subtree };
            //  .skip

            //  For each entry in kvmap, send kvmsg to client
            zhash_foreach (kvmap, s_send_single, &routing);

            //  .until
            //  Now send END message with sequence number
            printf ("I: sending shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    //  .skip
    printf (" Interrupted\n%d messages handled\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}
```

하위트리의 저장소의 내용을 구독하는 클라이언트의 코드입니다.

clonecli4.c : 복제 클라이언트, 모델 4

```
//  Clone client - Model Four

//  Lets us build this source without creating a library
#include "kvsimple.c"

//  This client is identical to clonecli3 except for where we
//  handles subtrees.
#define SUBTREE "/client/"
//  .skip

int main (void)
{
    //  Prepare our context and subscriber
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (subscriber, "");
    //  .until
    zsocket_connect (subscriber, "tcp://localhost:5557");
    zsocket_set_subscribe (subscriber, SUBTREE);
    //  .skip
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));
```

```
//  .until
//  We first request a state snapshot:
int64_t sequence = 0;
zstr_sendm (snapshot, "ICANHAZ?");
zstr_send  (snapshot, SUBTREE);
//  .skip
while (true) {
    kvmsg_t *kvmsg = kvmsg_recv (snapshot);
    if (!kvmsg)
        break;          //  Interrupted
    if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
        sequence = kvmsg_sequence (kvmsg);
        printf ("I: received snapshot=%d\n", (int) sequence);
        kvmsg_destroy (&kvmsg);
        break;          //  Done
    }
    kvmsg_store (&kvmsg, kvmap);
}
int64_t alarm = zclock_time () + 1000;
while (!zctx_interrupted) {
    zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 } };
    int tickless = (int) ((alarm - zclock_time ()));
    if (tickless < 0)
        tickless = 0;
    int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
    if (rc == -1)
        break;              //  Context has been shut down
```

```
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (subscriber);
            if (!kvmsg)
                break;          //  Interrupted

            //  Discard out-of-sequence kvmsgs, incl. heartbeats
            if (kvmsg_sequence (kvmsg) > sequence) {
                sequence = kvmsg_sequence (kvmsg);
                kvmsg_dump(kvmsg);
                kvmsg_store (&kvmsg, kvmap);
                printf ("I: received update=%d\n", (int) sequence);

            }
            else
                kvmsg_destroy (&kvmsg);
        }
        //  .until
        //  If we timed out, generate a random kvmsg
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%s%d", SUBTREE, randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
        //  .skip
    }
    printf (" Interrupted\n%d messages in\n", (int) sequence);
```

```
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트
- clonecli4에서 필터링을 통해 "SUBTREE"가 포함되어 발행(publish)된 kvmsg 객체를 받아 해시 테이블에 저정하며, 1초 간격으로 상태 요청에 사용될 kvmsg의 key에 "SUBTREE"을 포함하여 보냅니다.

```
 cl -EHsc clonesrv4.c libzmq.lib czmq.lib
 cl -EHsc clonecli4.c libzmq.lib czmq.lib

 ./clonesrv4
I: sending shapshot=0
I: publishing update     1
I: publishing update     2
I: publishing update     3
I: publishing update     4
...

 ./clonecli4
I: received snapshot=0
I: received update=1
I: received update=2
I: received update=3
I: received update=4
...
```

- [옮긴이] kvm_dump()을 통하여 메시지 내용을 확인하면 다음과 같습니다.

```
 ./clonesrv4
I: sending shapshot=0
[seq:1][key:/client/5174][size:6] 393939323337
I: publishing update     1
[seq:2][key:/client/2364][size:6] 373230383235
I: publishing update     2
[seq:3][key:/client/4486][size:6] 393134363732
I: publishing update     3
[seq:4][key:/client/4894][size:6] 353832373333
I: publishing update     4
...

 ./clonecli4
I: received snapshot=0
[seq:1][key:/client/5174][size:6] 393939323337
I: received update=1
[seq:2][key:/client/2364][size:6] 373230383235
I: received update=2
[seq:3][key:/client/4486][size:6] 393134363732
I: received update=3
[seq:4][key:/client/4894][size:6] 353832373333
I: received update=4
```

0.59.6 임시값들

임시값은 정기적으로 갱신(refresh)되지 없으면 자동으로 만료(expire)되는 값입니다. 등록 서비스에 복제가 사용된다고 생각하면, 임시값을 동적 값으로 사용할 수 있습니다. 노드는 네트워크에 가입하고 주소를 게시하고 이를 정기적으로 갱신합니다. 노드가 죽으면 그 주소는 결국 제거됩니다.

임시값들에 대한 일반적인 추상화는 이를 세션에 연결하고 세션이 종료될 때 삭제하는 것입니다. 복제에서 세션은 클라이언트들에 의해 정의되며 클라이언트가 죽으면 종료됩니다. 더 간단한 대안은 유효시간(TTL(Time to Live))을 임시값에 포함시켜 서버에서 TTL에 정해진 시간 동안 값이 갱신되지 않을 경우 만료하는 데 사용합니다.

좋은 디자인 원칙은 가능한 본질이 아닌 개념을 만들지 않는 것입니다. 매우 많은 수의 임시값들이 있는 경우 세션은 더 나은 성능을 제공합니다. 소수의 임시값들을 사용하는 경우 각 값에 TTL을 설정하는 것이 좋습니다. 대량의 임시값들을 사용하는 경우 이를 세션에 포함하고 한꺼번에 만료하는 것이 더 효율적입니다. 이번 단계의 주제에서는 세션을 제외하겠습니다.

이제 임시값을 구현합니다. 먼저 키-값 메시지에서 TTL을 인코딩하는 방법이 필요하며 프레임으로 추가할 수 있습니다. 속성으로 ØMQ 프레임을 사용할 때의 문제점은 매번 신규 속성을 추가할 때마다 메시지 구조를 변경할 경우 ØMQ 버전 간의 호환성을 깨뜨립니다. 따라서 메시지에 속성 프레임을 추가하고 속성값을 가져오고 입력할 수 있는 코드를 작성해 보겠습니다.

다음으로, "이 값을 삭제하십시오"라고 말하는 방법이 필요합니다. 지금까지는 서버와 클라이언트는 항상 맹목적으로 해시 테이블에 신규 값을 넣거나 기존 값을 변경했습니다. 앞으로는 해시 테이블에서 값이 비어 있으면 "키 삭제"를 의미하게 하겠습니다.

다음은 기존 `kvsimple` 보다 좀 더 완벽한 버전의 `kvmsg` 클래스이며, 속성들 프레임(나중에 필요할 UUID 프레임을 추가)을 구현하였습니다. 또한 필요한 경우 해시 테이블에서 키(key)를 삭제함으로 빈 값(empty value)을 처리합니다.

kvmsg.c : 키-값 메시지 클래스

```
//  kvmsg class - key-value message class for example applications

#include "kvmsg.h"
#ifdef _WIN32
#pragma comment(lib, "rpcrt4.lib")  // UuidCreate - Minimum supported OS Win 2000
#include <windows.h>
#else
#include <uuid/uuid.h>
#endif
```

```
#include "zlist.h"

//  Keys are short strings
#define KVMSG_KEY_MAX   255

//  Message is formatted on wire as 5 frames:
//  frame 0: key (ØMQ string)
//  frame 1: sequence (8 bytes, network order)
//  frame 2: uuid (blob, 16 bytes)
//  frame 3: properties (ØMQ string)
//  frame 4: body (blob)
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_UUID      2
#define FRAME_PROPS     3
#define FRAME_BODY      4
#define KVMSG_FRAMES    5

//  Structure of our class
struct _kvmsg {
    //  Presence indicators for each frame
    int present [KVMSG_FRAMES];
    //  Corresponding ØMQ message frames, if any
    zmq_msg_t frame [KVMSG_FRAMES];
    //  Key, copied into safe C string
    char key [KVMSG_KEY_MAX + 1];
    //  List of properties, as name=value strings
    zlist_t *props;
    size_t props_size;
```

```
};

//  .split property encoding
//  These two helpers serialize a list of properties to and from a
//  message frame:

static void
s_encode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    if (self->present [FRAME_PROPS])
        zmq_msg_close (msg);

    zmq_msg_init_size (msg, self->props_size);
    char *prop = zlist_first (self->props);
    char *dest = (char *) zmq_msg_data (msg);
    while (prop) {
        strcpy (dest, prop);
        dest += strlen (prop);
        *dest++ = '\n';
        prop = zlist_next (self->props);
    }
    self->present [FRAME_PROPS] = 1;
}

static void
s_decode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
```

```
    self->props_size = 0;
    while (zlist_size (self->props))
        free (zlist_pop (self->props));

    size_t remainder = zmq_msg_size (msg);
    char *prop = (char *) zmq_msg_data (msg);
    char *eoln = memchr (prop, '\n', remainder);
    while (eoln) {
        *eoln = 0;
        zlist_append (self->props, strdup (prop));
        self->props_size += strlen (prop) + 1;
        remainder -= strlen (prop) + 1;
        prop = eoln + 1;
        eoln = memchr (prop, '\n', remainder);
    }
}

//  .split constructor and destructor
//  Here are the constructor and destructor for the class:

//  Constructor, takes a sequence number for the new kvmsg instance:
kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    self->props = zlist_new ();
```

```
    kvmsg_set_sequence (self, sequence);
    return self;
}

//  zhash_free_fn callback helper that does the low level destruction:
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  Destroy message frames if any
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        //  Destroy property list
        while (zlist_size (self->props))
            free (zlist_pop (self->props));
        zlist_destroy (&self->props);

        //  Free object itself
        free (self);
    }
}

//  Destructor
void
kvmsg_destroy (kvmsg_t **self_p)
```

```
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}

//  .split recv method
//  This method reads a key-value message from the socket and returns a
//  new {{kvmsg}} instance:

kvmsg_t *
kvmsg_recv (void *socket)
{
    //  This method is almost unchanged from kvsimple
    //  .skip
    assert (socket);
    kvmsg_t *self = kvmsg_new (0);

    //  Read all frames off the wire, reject if bogus
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_msg_recv (&self->frame [frame_nbr], socket, 0) == -1) {
            kvmsg_destroy (&self);
```

```
            break;
        }
        //  Verify multipart framing
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsocket_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    //  .until
    if (self)
        s_decode_props (self);
    return self;
}

//  Send key-value message to socket; any empty frames are sent as such.
void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    s_encode_props (self);
    //  The rest of the method is unchanged from kvsimple
    //  .skip
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
        zmq_msg_init (&copy);
```

```
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_msg_send (&copy, socket,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}
//  .until

//  .split dup method
//  This method duplicates a {{kvmsg}} instance, returns the new instance:

kvmsg_t *
kvmsg_dup (kvmsg_t *self)
{
    kvmsg_t *kvmsg = kvmsg_new (0);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr]) {
            zmq_msg_t *src = &self->frame [frame_nbr];
            zmq_msg_t *dst = &kvmsg->frame [frame_nbr];
            zmq_msg_init_size (dst, zmq_msg_size (src));
            memcpy (zmq_msg_data (dst),
                    zmq_msg_data (src), zmq_msg_size (src));
            kvmsg->present [frame_nbr] = 1;
        }
    }
    kvmsg->props_size = zlist_size (self->props);
    char *prop = (char *) zlist_first (self->props);
```

```
    while (prop) {
        zlist_append (kvmsg->props, strdup (prop));
        prop = (char *) zlist_next (self->props);
    }
    return kvmsg;
}

//  The key, sequence, body, and size methods are the same as in kvsimple.
//  .skip

//  Return key from last read message, if any, else NULL
char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}
```

```
//  Set message key as provided
void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}

//  Set message key using printf format
void
kvmsg_fmt_key (kvmsg_t *self, char *format, ...)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}

//  Return sequence nbr from last read message, if any
```

```
int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}

//  Set message sequence number
void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
```

```
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}

//  Return body from last read message, if any, else NULL
byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}

//  Set message body
void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
```

```
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}

//  Set message body using printf format
void
kvmsg_fmt_body (kvmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}

//  Return body size from last read message, if any, else zero
size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
```

```
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}
//  .until

//  .split UUID methods
//  These methods get and set the UUID for the key-value message:

byte *
kvmsg_uuid (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_UUID]
    &&  zmq_msg_size (&self->frame [FRAME_UUID]) == sizeof (uuid_t))
        return (byte *) zmq_msg_data (&self->frame [FRAME_UUID]);
    else
        return NULL;
}

#ifdef _WIN32
//  Sets the UUID to a randomly generated value
void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
```

```
    UuidCreate(&uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid_t));
    memcpy (zmq_msg_data (msg), &uuid, sizeof (uuid_t));
    self->present [FRAME_UUID] = 1;
}
#else
void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
    uuid_generate (uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid));
    memcpy (zmq_msg_data (msg), uuid, sizeof (uuid));
    self->present [FRAME_UUID] = 1;
}
#endif

//  .split property methods
//  These methods get and set a specified message property:

//  Get message property, return "" if no such property is defined.
char *
kvmsg_get_prop (kvmsg_t *self, char *name)
```

```
{
    assert (strchr (name, '=') == NULL);
    char *prop = zlist_first (self->props);
    size_t namelen = strlen (name);
    while (prop) {
        if (strlen (prop) > namelen
        &&  memcmp (prop, name, namelen) == 0
        &&  prop [namelen] == '=')
            return prop + namelen + 1;
        prop = zlist_next (self->props);
    }
    return "";
}

//  Set message property. Property name cannot contain '='. Max length of
//  value is 255 chars.
void
kvmsg_set_prop (kvmsg_t *self, char *name, char *format, ...)
{
    assert (strchr (name, '=') == NULL);

    char value [255 + 1];
    va_list args;
    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);

    //  Allocate name=value string
```

```
    char *prop = malloc (strlen (name) + strlen (value) + 2);

    //  Remove existing property if any
    sprintf (prop, "%s=", name);
    char *existing = zlist_first (self->props);
    while (existing) {
        if (memcmp (prop, existing, strlen (prop)) == 0) {
            self->props_size -= strlen (existing) + 1;
            zlist_remove (self->props, existing);
            free (existing);
            break;
        }
        existing = zlist_next (self->props);
    }
    //  Add new name=value property string
    strcat (prop, value);
    zlist_append (self->props, prop);
    self->props_size += strlen (prop) + 1;
}

//  .split store method
//  This method stores the key-value message into a hash map, unless
//  the key and value are both null. It nullifies the {{kvmsg}} reference
//  so that the object is owned by the hash map, not the caller:

void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
```

```
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (kvmsg_size (self)) {
            if (self->present [FRAME_KEY]
            &&  self->present [FRAME_BODY]) {
                zhash_update (hash, kvmsg_key (self), self);
                zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
            }
        }
        else
            zhash_delete (hash, kvmsg_key (self));

        *self_p = NULL;
    }
}

//  .split dump method
//  This method extends the {{kvsimple}} implementation with support for
//  message properties:

void
kvmsg_dump (kvmsg_t *self)
{
    //  .skip
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
```

```
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self));
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        //  .until
        fprintf (stderr, "[size:%zd] ", size);
        if (zlist_size (self->props)) {
            fprintf (stderr, "[");
            char *prop = zlist_first (self->props);
            while (prop) {
                fprintf (stderr, "%s;", prop);
                prop = zlist_next (self->props);
            }
            fprintf (stderr, "]");
        }
        //  .skip
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}
//  .until

//  .split test method
//  This method is the same as in {{kvsimple}} with added support
```

```
//  for the uuid and property features of {{kvmsg}}:

int
kvmsg_test (int verbose)
{
    //  .skip
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg: ");

    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "inproc://kvmsg_selftest");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "inproc://kvmsg_selftest");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  .until
    //  Test send and receive of simple message
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
```

```
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose){
        kvmsg_dump (kvmsg);
        byte *uuid = kvmsg_uuid(kvmsg);
        int size = sizeof(uuid);
        fprintf (stderr, "UUID :");
        for (int char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", uuid [char_nbr]);
        fprintf (stderr, "\n");
    }
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  Test send and receive of message with properties
    kvmsg = kvmsg_new (2);
    kvmsg_set_prop (kvmsg, "prop1", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value2");
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
```

```
    kvmsg_destroy (&kvmsg);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    kvmsg_destroy (&kvmsg);
    //  .skip
    //  Shutdown and destroy all objects
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
//  .until
```

- [옮긴이] "kvmsg.c"는 윈도우에서 실행되지 않아 수정이 필요합니다.

1. ipc 전송 방법을 inproc로 변경합니다.

- 변경전 : "ipc://kvmsg_selftest.ipc"
- 변경후 : "inproc://kvmsg_selftest"

2. uuid를 윈도우 환경에서 사용할 수 있도록 4장 "titanic.c" 예제를 참조한다.

```
#ifdef _WIN32
#pragma comment(lib, "rpcrt4.lib")  // UuidCreate - Minimum supported OS Win 2000
#include <windows.h>
#else
#include <uuid/uuid.h>
```

```
#endif
...
#ifdef _WIN32
//  Sets the UUID to a randomly generated value
void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
    UuidCreate(&uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid_t));
    memcpy (zmq_msg_data (msg), &uuid, sizeof (uuid_t));
    self->present [FRAME_UUID] = 1;
}
#else
void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
    uuid_generate (uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid));
    memcpy (zmq_msg_data (msg), uuid, sizeof (uuid));
```

```
    self->present [FRAME_UUID] = 1;
}
#endif
```

- [옮긴이] 테스트를 위하여 "kvmsgtest.c" 코드는 다음과 같습니다.

```
#include "kvmsg.c"

void main()
{
    kvmsg_test(1);
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc kvmsgtest.c libzmq.lib czmq.lib

 ./kvmsgtest
 * kvmsg: [seq:1][key:key][size:4] 626F6479
[seq:1][key:key][size:4] 626F6479
[seq:2][key:key][size:4] [prop1=value1;prop2=value2;]626F6479
[seq:2][key:key][size:4] [prop1=value1;prop2=value2;]626F6479
UUID :5301DB3A6170784D
OK
```

모델 5 클라이언트는 모델 4와 거의 동일합니다. 이제 전체 kvmsg 클래스를 사용하고 각 메시지에 임의의 ttl 속성(초 단위로 측정)을 설정합니다.

```
kvmsg_set_prop (kvmsg, "ttl", "%d", randof (30));
```

0.59.7 리엑터 사용

지금까지 우리는 서버에서 폴(poll) 루프를 사용했습니다. 다음 서버 모델에서는 리엑터로 전환하여 사용합니다. C 언어에서는 CZMQ의 zloop 클래스를 사용합니다. 리액터를 사용하면 코드가 더 장황해지지만 서버의 각 부분이 개별 리엑터 핸들러에 의해 처리되기 때문에 쉽게 이해할 수 있습니다.

단일 스레드를 사용하고 리엑터 핸들러에 서버 객체를 전달합니다. 서버를 여러 개의 스레드들로 구성하여 각 스레드는 하나의 소켓 또는 타이머를 처리하지만 스레드가 데이터를 공유하지 않을 경우 더 잘 작동합니다. 이 경우 모든 작업은 서버의 해시 맵을 중심으로 이루어지며 하나의 스레드가 더 간단합니다.

3개의 리엑터 핸들러는 다음과 같습니다.

- 하나는 ROUTER 소켓을 통해 오는 스냅샷 요청들을 처리합니다.
- 하나는 PULL 소켓을 통해 오는 클라이언트들의 변경정보를 처리합니다.
- 하나는 TTL이 경과된 임시값을 만료 처리(값을 공백으로 처리)합니다.

clonesrv5.c : 복제 서버, 모델 5

```
//  Clone server - Model Five

//  Lets us build this source without creating a library
#include "kvmsg.c"

//  zloop reactor handlers
static int s_snapshots (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_collector (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int s_flush_ttl (zloop_t *loop, int timer_id, void *args);

//  Our server is defined by these properties
typedef struct {
    zctx_t *ctx;                //  Context wrapper
```

```
    zhash_t *kvmap;             //  Key-value store
    zloop_t *loop;              //  zloop reactor
    int port;                   //  Main port we're working on
    int64_t sequence;           //  How many updates we're at
    void *snapshot;             //  Handle snapshot requests
    void *publisher;            //  Publish updates to clients
    void *collector;            //  Collect updates from clients
} clonesrv_t;

int main (void)
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));
    self->port = 5556;
    self->ctx = zctx_new ();
    self->kvmap = zhash_new ();
    self->loop = zloop_new ();
    zloop_set_verbose (self->loop, false);

    //  Set up our clone server sockets
    self->snapshot  = zsocket_new (self->ctx, ZMQ_ROUTER);
    zsocket_bind (self->snapshot,  "tcp://*:%d", self->port);
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    self->collector = zsocket_new (self->ctx, ZMQ_PULL);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  Register our handlers with reactor
    zmq_pollitem_t poller = { 0, 0, ZMQ_POLLIN };
    poller.socket = self->snapshot;
```

```
    zloop_poller (self->loop, &poller, s_snapshots, self);
    poller.socket = self->collector;
    zloop_poller (self->loop, &poller, s_collector, self);
    zloop_timer (self->loop, 1000, 0, s_flush_ttl, self);

    //  Run reactor until process interrupted
    zloop_start (self->loop);

    zloop_destroy (&self->loop);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);
    return 0;
}

//  .split send snapshots
//  We handle ICANHAZ? requests by sending snapshot data to the
//  client that requested it:

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
    char *subtree;          //  Client subtree specification
} kvroute_t;

//  We call this function for each key-value pair in our hash table
static int
s_send_single (const char *key, void *data, void *args)
```

```
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        zframe_send (&kvroute->identity,    //  Choose recipient
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}

//  .split snapshot handler
//  This is the reactor handler for the snapshot socket; it accepts
//  just the ICANHAZ? request and replies with a state snapshot ending
//  with a KTHXBAI message:

static int
s_snapshots (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (poller->socket);
    if (identity) {
        //  Request is in second frame of message
        char *request = zstr_recv (poller->socket);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
```

```
            free (request);
            subtree = zstr_recv (poller->socket);
        }
        else
            printf ("E: bad request, aborting\n");

        if (subtree) {
            //  Send state socket to client
            kvroute_t routing = { poller->socket, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            zclock_log ("I: sending shapshot=%d", (int) self->sequence);
            zframe_send (&identity, poller->socket, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, poller->socket);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
        zframe_destroy(&identity);
    }
    return 0;
}

//  .split collect updates
//  We store each update with a new sequence number, and if necessary, a
//  time-to-live. We publish updates immediately on our publisher socket:
```

```
static int
s_collector (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (poller->socket);
    if (kvmsg) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
        if (ttl)
            kvmsg_set_prop (kvmsg, "ttl",
                "%" PRId64, zclock_time () + ttl * 1000);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing update=%d", (int) self->sequence);
    }
    return 0;
}

//  .split flush ephemeral values
//  At regular intervals, we flush ephemeral values that have expired. This
//  could be slow on very large data sets:

//  If key-value pair has expired, delete it and publish the
//  fact to listening clients.
static int
s_flush_single (const char *key, void *data, void *args)
{
```

```
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing delete=%d", (int) self->sequence);
    }
    return 0;
}

static int
s_flush_ttl (zloop_t *loop, int timer_id, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    if (self->kvmap)
        zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}
```

kvmsg클래스에 TTL 속성을 부가한 클라이언트 소스는 다음과 같습니다.

clonecli5.c : 복제 클라이언트, 모델 5

```
//  Clone client - Model Five

//  Lets us build this source without creating a library
#include "kvmsg.c"
```

```
#define SUBTREE "/client/"

int main (void)
{
    //  Prepare our context and subscriber
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_set_subscribe (subscriber, "");
    zsocket_connect (subscriber, "tcp://localhost:5557");
    zsocket_set_subscribe (subscriber, SUBTREE);
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  Get state snapshot
    int64_t sequence = 0;
    zstr_sendm (snapshot, "ICANHAZ?");
    zstr_send  (snapshot, SUBTREE);
    while (true) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: received snapshot=%d\n", (int) sequence);
```

```
            kvmsg_destroy (&kvmsg);
            break;          //  Done
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    int64_t alarm = zclock_time () + 1000;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 } };
        int tickless = (int) ((alarm - zclock_time ()));
        if (tickless < 0)
            tickless = 0;
        int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Context has been shut down

        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (subscriber);
            if (!kvmsg)
                break;          //  Interrupted

            //  Discard out-of-sequence kvmsgs, incl. heartbeats
            if (kvmsg_sequence (kvmsg) > sequence) {
                sequence = kvmsg_sequence (kvmsg);
                kvmsg_store (&kvmsg, kvmap);
                printf ("I: received update=%d\n", (int) sequence);
            }
            else
                kvmsg_destroy (&kvmsg);
        }
```

```
        //  If we timed out, generate a random kvmsg
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%s%d", SUBTREE, randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_set_prop (kvmsg, "ttl", "%d", randof (30));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
    }
    printf (" Interrupted\n%d messages in\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

- [옮긴이] 빌드 및 테스트

```
 cl -EHsc clonesrv5.c libzmq.lib czmq.lib
 cl -EHsc clonecli5.c libzmq.lib czmq.lib

 ./clonesrv5
20-08-29 07:27:56 I: sending shapshot=0
20-08-29 07:27:57 I: publishing update=1
20-08-29 07:27:58 I: publishing update=2
...
20-08-29 07:28:08 I: publishing update=12
20-08-29 07:28:08 I: publishing delete=13
20-08-29 07:28:09 I: publishing update=14
```

```
20-08-29 07:28:10 I: publishing update=15
20-08-29 07:28:11 I: publishing update=16
20-08-29 07:28:12 I: publishing update=17
20-08-29 07:28:12 I: publishing delete=18
...

 ./clonecli5
I: received snapshot=0
I: received update=1
I: received update=2
I: received update=3
I: received update=4
...
```

0.59.8 신뢰성을 위한 바이너리 스타 패턴 추가

지금까지 살펴본 복제 모델은 비교적 간단했습니다. 이제 우리는 불쾌하고 복잡한 영역에 들어가서 다른 에스프레소를 마시게 될 것입니다. "신뢰성 있는" 메시징을 만드는 것이 복잡한 만큼 "실제로 이것이 필요한가?"라는 의문이 있어야 합니다. "신뢰성이 없는" 혹은 "충분히 좋은 신뢰성을 가진" 상황에서 벗어날 수 있다면 비용과 복잡성 측면에서 큰 승리를 한 것입니다. 물론 때때로 일부 데이터가 유실할 수 있지만 좋은 절충안입니다. 하지만 한 모금의 에스프레소는 정말 좋습니다. 복잡성의 세계로 뛰어들어 가겠습니다.

이전 모델(리엑터와 TTL)로 작업할 때 서버를 중지하고 재시작하면 복구된 것처럼 보이지만 물론 적절한 현재 상태 대신 빈 상태에 변경정보들을 반영합니다. 네트워크에 가입하는 모든 신규 클라이언트는 전체 이력 데이터 대신 최신 변경정보만 받습니다.

우리가 서버가 죽거나 충돌하면 복구하는 방법이 필요합니다. 또한 서버가 일정 시간 동안 작동하지 않는 경우 백업을 제공해야 합니다. 누군가 "신뢰성" 구현이 필요하면 우선 처리해야 하는 장애들에 대한 목록 요구해야 하며, 우리의 경우 다음과 같습니다.

- 서버 프로세스가 충돌하여 자동 또는 수동으로 재시작됩니다. 프로세스는 상태를 잃고 어딘가에서 다시 가져와야 합니다.

- 서버 머신(하드웨어)이 죽고 상당한 시간 동안 오프라인 상태입니다. 클라이언트는 어딘가의 대체 서버로 전환해야 합니다.
- 서버 프로세스 또는 머신이 네트워크에서 연결 해제됩니다(예 : 스위치가 죽거나 데이터센터가 무응답). 어제든지 정상화될 수 있지만 그동안 클라이언트들은 대체 서버가 필요합니다.

첫 번째 단계는 대체 서버를 추가하는 것입니다."4장 - 신뢰할 수 있는 요청-응답 패턴"의 바이너리 스타 패턴을 사용하여 기본 및 백업으로 구성할 수 있습니다. 바이너리 스타는 리엑터를 사용하므로 이전 서버 모델(리엑터+TTL)을 재구성하기에는 유용합니다.

기본 서버가 충돌하더라도 변경정보들이 손실되지 않도록 해야 합니다. 가장 간단한 방법은 클라이언트의 상태 변경정보를 2개 서버(기본-백업)에 모두 보내는 것입니다. 그런 다음 백업 서버는 클라이언트 역할을 수행하며, 모든 클라이언트가 수행하는 것처럼 서버로부터 변경정보들을 수신하여 상태를 동기화합니다. 또한 백업 서버는 클라이언트로부터 새로운 변경정보를 받지만 해시테 이블에 저장하지 않고 잠시 동안 가지고 있습니다.

따라서 모델 6은 모델 5에 비해 다음과 같은 변경 사항이 적용되었습니다.

- 클라이언트가 서버로 전송하는 상태 변경정보에 PUSH-PULL 패턴 대신 PUB-SUB 패턴을 사용합니다. 이렇게 하면 2대의 서버로 변경정보가 동일하게 펼쳐져 갈 수 있습니다. 그렇지 않으면 2개의 DEALER 소켓들을 사용해야 합니다.
- 서버 변경정보 메시지(클라이언트로 가는)에 심박을 추가하여, 클라이언트가 기본 서버의 죽음을 감지하게 하여, 클라이언트가 백업 서버로 전환하게 합니다.
- 2대의 서버를 바이너리 스타 bstar 리엑터 클래스를 사용하여 연결합니다. 바이너리 스타는 클라이언트들이 활성화 상태로 여기는 서버에 명시적인 요청하는 투표에 의존합니다. 투표 메커니즘으로 스냅샷 요청들을 사용할 것입니다.
- 모든 변경정보 메시지에 고유의 식별자로 UUID 필드를 추가하였습니다. 클라이언트는 상태 변경정보에서 생성하고, 서버는 다시 클라이언트에서 변경정보를 받아하여 전파합니다.
- 비활성 서버는 "보류 목록(Pending list)"에
- 클라이언트에서 수신되었지만 아직 활성 서버에서는 수신되지 않은 변경정보들을 저장하거나,

- 활성 서버에서는 수신되었지만 클라이언트에서는 아직 수신되지 않은 변경정보들을 저장합니다.
- 보류 목록은 오래된 것부터 최신 순서로 정렬되어 있으므로 최신 것부터 변경정보들을 쉽게 제거할 수 있습니다.

그림 61 - 복제 클라이언트 유한 상태 머신

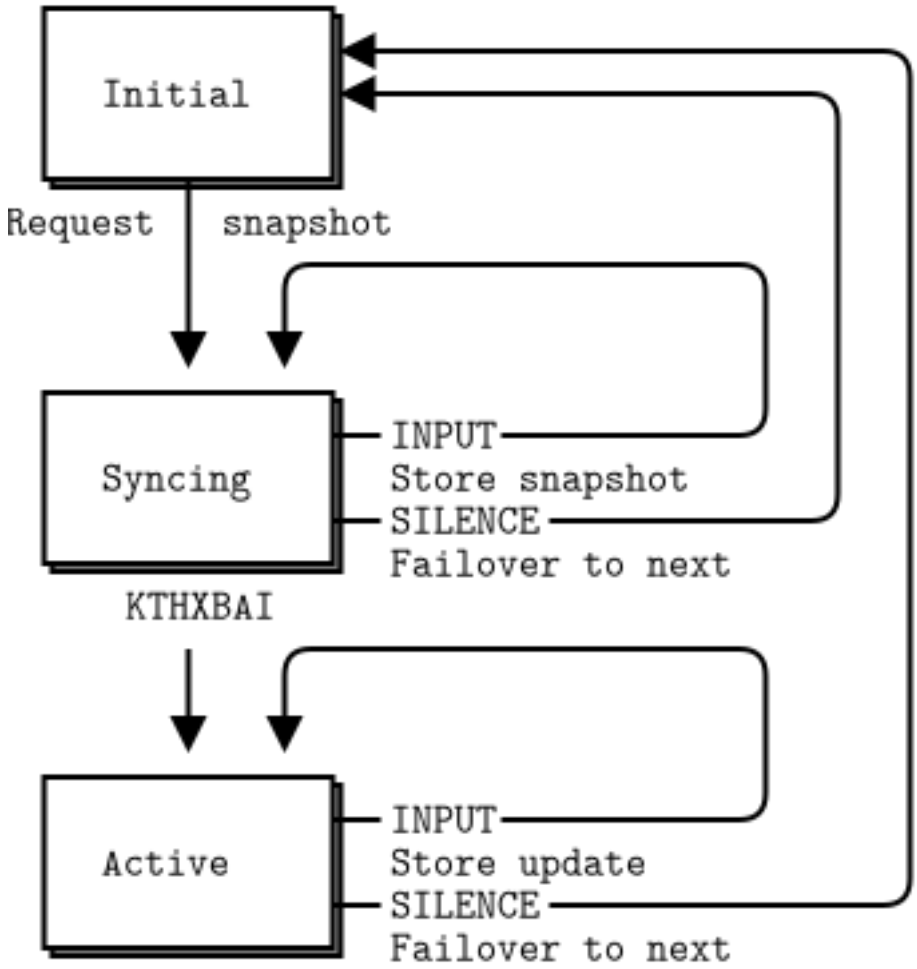


그림 63: Clone Client Finite State Machine

클라이언트 로직을 유한 상태 머신으로 설계하는 것이 유용합니다. 클라이언트는 다음 3 가지 상태(INITIAL, SYNCING, ACTIVE)를 순환합니다.

- [옮긴이] 기존 클라이언트 예제에서도 클라이언트가 시작되면 서버에 상태 요청(ICANHAZ)하여 snapshot 받아 오면, 클라이언트에 snapshot을 반영하고 서버에 완료(KTHXBAI)를 보낸 이후, 클라이언트에서 상태 변경정보를 보내면 서버가 각 클라이언트에서 변경정보를 전송하도록 하였습니다.

- [INITIAL] 클라이언트는 소켓(SUB, DEALER, PUB)을 열고 연결하고, 첫 번째 서버에서 스냅샷을 요청합니다. 클라이언트들의 요청 폭주를 방지하기 위해 주어진 서버에 2번만 요청합니다. 하나의 요청이 유실되는 것은 불행이지만, 두 번째까지 유실되는 것은 부주의입니다.

- [SYNCING] 클라이언트는 서버로부터 응답(스냅샷 데이터)을 기다렸다가 수신하여 저장합니다. 정해진 제한 시간 내에 서버로부터 응답이 없으면 다음 서버로 장애조치합니다.

- [ACTIVE] 클라이언트가 스냅샷을 받고, 서버의 변경정보들을 기다렸다가 처리합니다. 다시 정해진 제한 시간(timeout)에 서버로부터 응답이 없으면 다음 서버로 장애조치합니다.

클라이언트는 영원히 반복됩니다. 시작 또는 장애조치 중에 일부 클라이언트들은 기본 서버와 통신을 시도하고 다른 클라이언트들은 백업 서버와 통신을 시도할 것입니다. 바이너리 스타 상태 머신은 이것을 정확하게 처리합니다. 소프트웨어가 정확하다는 것을 증명하는 것은 어렵지만 대신에 우리는 소프트웨어가 틀렸다는 것을 증명할 수 없을 때까지 두드려 봅니다.

장애조치는 다음과 같이 이루어집니다.

- 클라이언트는 기본 서버가 더 이상 심박를 보내지 않음을 감지하고 죽었다는 결론을 내립니다. 클라이언트는 백업 서버에 연결하고 신규 상태 스냅샷을 요청합니다.
- 백업 서버는 클라이언트로부터 스냅샷 요청을 받기 시작하고 기본 서버의 죽음을 감지하여 기본 서버로 인계합니다.
- 백업 서버는 "보류 목록"을 자신의 해시 테이블에 적용한 다음, 클라이언트의 상태 스냅샷 요청을 처리하기 시작합니다.

기본 서버가 다시 온라인 상태가 되면 다음을 수행합니다.

- 비활성 상태로 서버를 시작하고, 복제 클라이언트로 백업 서버에 연결합니다.
- SUB 소켓을 통해 클라이언트들로부터 변경정보들을 수신하기 시작합니다.

몇 가지 가정들은 다음과 같습니다.

- 하나 이상의 서버가 계속 실행됩니다. 2대의 서버들이 충돌하면 모든 서버 상태가 유실되고 복구할 방법이 없습니다.
- 여러 클라이언트들이 동일한 해시 테이블의 키를 동시에 변경하지 않습니다. 클라이언트들의 변경정보들은 다른 순서로 2대 서버들에 도달합니다. 따라서 백업 서버는 기본 서버와 다른 순서의 변경정보들을 보류 목록(Pending List)의 반영할 수 있습니다. 한 클라이언트의 변경정보들은 항상 동일한 순서로 2대 서버들에 도달하므로 안전합니다.

따라서 바이너리 스티 패턴을 사용하는 고가용성 서버 쌍의 아키텍처에는 2대의 서버와 서버들과 통신하는 일련의 클라이언트들이 있습니다.

그림 62 - 고가용성 복제 서버 쌍

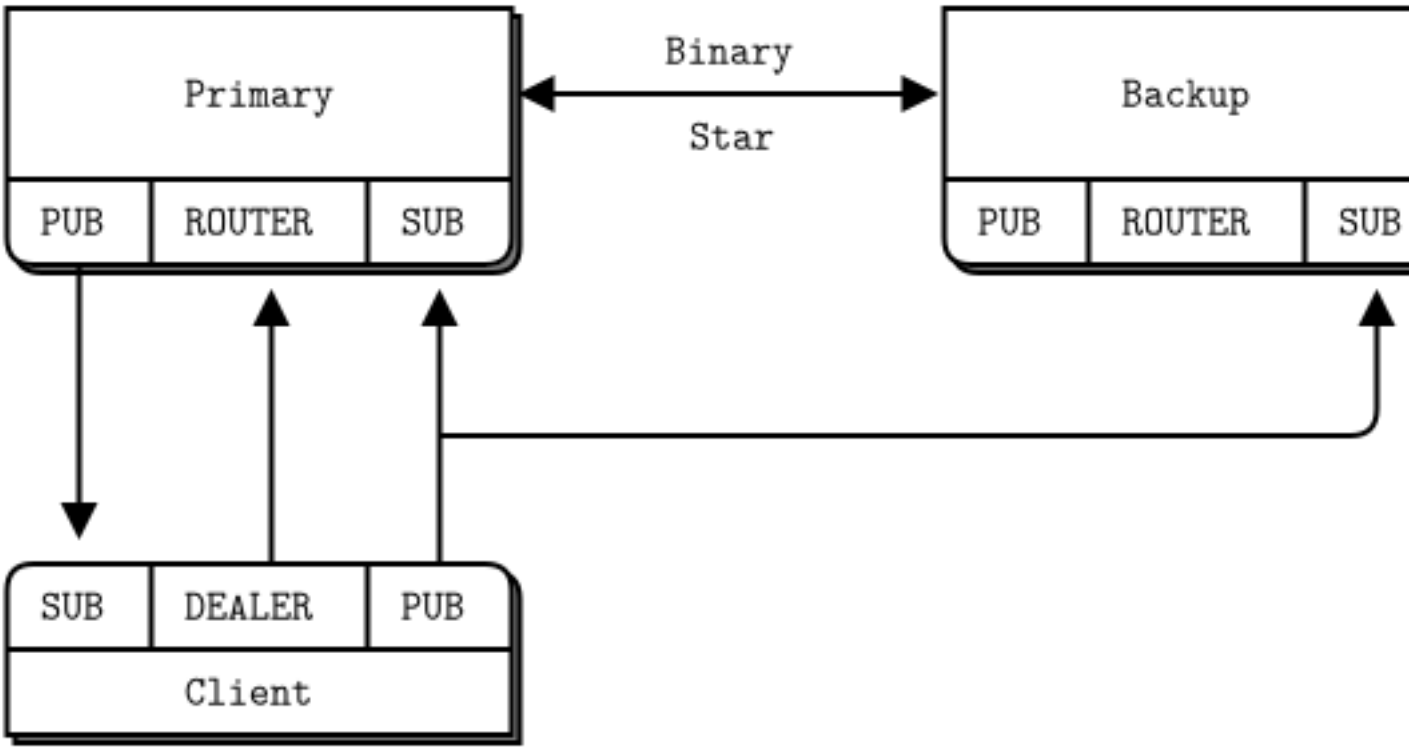


그림 64: High-availability Clone Server Pair

여기에 6번째로 복제 서버의 마지막 모델(모델 6)의 코드입니다.

clonesrv6.c : 복제 서버, 모델 6

```
//  Clone server Model Six

//  Lets us build this source without creating a library
#include "bstar.c"
#include "kvmsg.c"

//  .split definitions
//  We define a set of reactor handlers and our server object structure:
```

```
//  Bstar reactor handlers
static int
    s_snapshots   (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int
    s_collector   (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int
    s_flush_ttl   (zloop_t *loop, int timer_id, void *args);
static int
    s_send_hugz   (zloop_t *loop, int timer_id, void *args);
static int
    s_new_active  (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int
    s_new_passive (zloop_t *loop, zmq_pollitem_t *poller, void *args);
static int
    s_subscriber  (zloop_t *loop, zmq_pollitem_t *poller, void *args);

//  Our server is defined by these properties
typedef struct {
    zctx_t *ctx;                //  Context wrapper
    zhash_t *kvmap;             //  Key-value store
    bstar_t *bstar;             //  Bstar reactor core
    int64_t sequence;           //  How many updates we're at
    int port;                   //  Main port we're working on
    int peer;                   //  Main port of our peer
    void *publisher;            //  Publish updates and hugz
    void *collector;            //  Collect updates from clients
    void *subscriber;           //  Get updates from peer
    zlist_t *pending;           //  Pending updates from clients
```

```
    bool primary;               //  true if we're primary
    bool active;                //  true if we're active
    bool passive;               //  true if we're passive
} clonesrv_t;

//  .split main task setup
//  The main task parses the command line to decide whether to start
//  as a primary or backup server. We're using the Binary Star pattern
//  for reliability. This interconnects the two servers so they can
//  agree on which one is primary and which one is backup. To allow the
//  two servers to run on the same box, we use different ports for
//  primary and backup. Ports 5003/5004 are used to interconnect the
//  servers. Ports 5556/5566 are used to receive voting events (snapshot
//  requests in the clone pattern). Ports 5557/5567 are used by the
//  publisher, and ports 5558/5568 are used by the collector:

int main (int argc, char *argv [])
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));
    if (argc == 2 && streq (argv [1], "-p")) {
        zclock_log ("I: primary active, waiting for backup (passive)");
        self->bstar = bstar_new (BSTAR_PRIMARY, "tcp://*:5003",
                                 "tcp://localhost:5004");
        bstar_voter (self->bstar, "tcp://*:5556",
                     ZMQ_ROUTER, s_snapshots, self);
        self->port = 5556;
        self->peer = 5566;
        self->primary = true;
    }
```

```
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        zclock_log ("I: backup passive, waiting for primary (active)");
        self->bstar = bstar_new (BSTAR_BACKUP, "tcp://*:5004",
                                 "tcp://localhost:5003");
        bstar_voter (self->bstar, "tcp://*:5566",
                     ZMQ_ROUTER, s_snapshots, self);
        self->port = 5566;
        self->peer = 5556;
        self->primary = false;
    }
    else {
        printf ("Usage: clonesrv6 { -p | -b }\n");
        free (self);
        exit (0);
    }
    //  Primary server will become first active
    if (self->primary)
        self->kvmap = zhash_new ();

    self->ctx = zctx_new ();
    self->pending = zlist_new ();
    bstar_set_verbose (self->bstar, true);

    //  Set up our clone server sockets
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_set_subscribe (self->collector, "");
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
```

```
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  Set up our own clone client interface to peer
    self->subscriber = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_set_subscribe (self->subscriber, "");
    zsocket_connect (self->subscriber,
                     "tcp://localhost:%d", self->peer + 1);

    //  .split main task body
    //  After we've setup our sockets, we register our binary star
    //  event handlers, and then start the bstar reactor. This finishes
    //  when the user presses Ctrl-C or when the process receives a SIGINT
    //  interrupt:

    //  Register state change handlers
    bstar_new_active (self->bstar, s_new_active, self);
    bstar_new_passive (self->bstar, s_new_passive, self);

    //  Register our other handlers with the bstar reactor
    zmq_pollitem_t poller = { self->collector, 0, ZMQ_POLLIN };
    zloop_poller (bstar_zloop (self->bstar), &poller, s_collector, self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_flush_ttl, self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_send_hugz, self);

    //  Start the bstar reactor
    bstar_start (self->bstar);

    //  Interrupted, so shut down
    while (zlist_size (self->pending)) {
```

```
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_destroy (&kvmsg);
    }
    zlist_destroy (&self->pending);
    bstar_destroy (&self->bstar);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);

    return 0;
}

//  We handle ICANHAZ? requests exactly as in the clonesrv5 example.
//  .skip

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
    char *subtree;          //  Client subtree specification
} kvroute_t;

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
static int
s_send_single (const char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
```

```
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        zframe_send (&kvroute->identity,    //  Choose recipient
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}

static int
s_snapshots (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (poller->socket);
    if (identity) {
        //  Request is in second frame of message
        char *request = zstr_recv (poller->socket);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (poller->socket);
        }
        else
            printf ("E: bad request, aborting\n");

        if (subtree) {
            //  Send state socket to client
```

```
            kvroute_t routing = { poller->socket, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            zclock_log ("I: sending shapshot=%d", (int) self->sequence);
            zframe_send (&identity, poller->socket, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, poller->socket);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
        zframe_destroy(&identity);
    }
    return 0;
}
//  .until

//  .split collect updates
//  The collector is more complex than in the clonesrv5 example because the
//  way it processes updates depends on whether we're active or passive.
//  The active applies them immediately to its kvmap, whereas the passive
//  queues them as pending:

//  If message was already on pending list, remove it and return true,
//  else return false.
static int
s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg)
```

```
{
    kvmsg_t *held = (kvmsg_t *) zlist_first (self->pending);
    while (held) {
        if (memcmp (kvmsg_uuid (kvmsg),
                    kvmsg_uuid (held), sizeof (uuid_t)) == 0) {
            zlist_remove (self->pending, held);
            return true;
        }
        held = (kvmsg_t *) zlist_next (self->pending);
    }
    return false;
}

static int
s_collector (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (poller->socket);
    if (kvmsg) {
        if (self->active) {
            kvmsg_set_sequence (kvmsg, ++self->sequence);
            kvmsg_send (kvmsg, self->publisher);
            int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
            if (ttl)
                kvmsg_set_prop (kvmsg, "ttl",
                    "%" PRId64, zclock_time () + ttl * 1000);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: publishing update=%d", (int) self->sequence);
```

```
        }
        else {
            //  If we already got message from active, drop it, else
            //  hold on pending list
            if (s_was_pending (self, kvmsg))
                kvmsg_destroy (&kvmsg);
            else
                zlist_append (self->pending, kvmsg);
        }
    }
    return 0;
}

//  We purge ephemeral values using exactly the same code as in
//  the previous clonesrv5 example.
//  .skip
//  If key-value pair has expired, delete it and publish the
//  fact to listening clients.
static int
s_flush_single (const char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
```

```
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing delete=%d", (int) self->sequence);
    }
    return 0;
}

static int
s_flush_ttl (zloop_t *loop, int timer_id, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    if (self->kvmap)
        zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}
//  .until

//  .split heartbeating
//  We send a HUGZ message once a second to all subscribers so that they
//  can detect if our server dies. They'll then switch over to the backup
//  server, which will become active:

static int
s_send_hugz (zloop_t *loop, int timer_id, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_new (self->sequence);
    kvmsg_set_key  (kvmsg, "HUGZ");
```

```
    kvmsg_set_body (kvmsg, (byte *) "", 0);
    kvmsg_send     (kvmsg, self->publisher);
    kvmsg_destroy (&kvmsg);

    return 0;
}

//  .split handling state changes
//  When we switch from passive to active, we apply our pending list so that
//  our kvmap is up-to-date. When we switch to passive, we wipe our kvmap
//  and grab a new snapshot from the active server:

static int
s_new_active (zloop_t *loop, zmq_pollitem_t *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    self->active = true;
    self->passive = false;

    //  Stop subscribing to updates
    zmq_pollitem_t poller = { self->subscriber, 0, ZMQ_POLLIN };
    zloop_poller_end (bstar_zloop (self->bstar), &poller);

    //  Apply pending list to own hash table
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
```

```
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing pending=%d", (int) self->sequence);
    }
    return 0;
}

static int
s_new_passive (zloop_t *loop, zmq_pollitem_t *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zhash_destroy (&self->kvmap);
    self->active = false;
    self->passive = true;

    //  Start subscribing to updates
    zmq_pollitem_t poller = { self->subscriber, 0, ZMQ_POLLIN };
    zloop_poller (bstar_zloop (self->bstar), &poller, s_subscriber, self);

    return 0;
}

//  .split subscriber handler
//  When we get an update, we create a new kvmap if necessary, and then
//  add our update to our kvmap. We're always passive in this case:

static int
s_subscriber (zloop_t *loop, zmq_pollitem_t *poller, void *args)
{
```

```
    clonesrv_t *self = (clonesrv_t *) args;
    //  Get state snapshot if necessary
    if (self->kvmap == NULL) {
        self->kvmap = zhash_new ();
        void *snapshot = zsocket_new (self->ctx, ZMQ_DEALER);
        zsocket_connect (snapshot, "tcp://localhost:%d", self->peer);
        zclock_log ("I: asking for snapshot from: tcp://localhost:%d",
                    self->peer);
        zstr_sendm (snapshot, "ICANHAZ?");
        zstr_send (snapshot, ""); // blank subtree to get all
        while (true) {
            kvmsg_t *kvmsg = kvmsg_recv (snapshot);
            if (!kvmsg)
                break;          //  Interrupted
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                kvmsg_destroy (&kvmsg);
                break;          //  Done
            }
            kvmsg_store (&kvmsg, self->kvmap);
        }
        zclock_log ("I: received snapshot=%d", (int) self->sequence);
        zsocket_destroy (self->ctx, snapshot);
    }
    //  Find and remove update off pending list
    kvmsg_t *kvmsg = kvmsg_recv (poller->socket);
    if (!kvmsg)
        return 0;
```

```
    if (strneq (kvmsg_key (kvmsg), "HUGZ")) {
        if (!s_was_pending (self, kvmsg)) {
            //  If active update came before client update, flip it
            //  around, store active update (with sequence) on pending
            //  list and use to clear client update when it comes later
            zlist_append (self->pending, kvmsg_dup (kvmsg));
        }
        //  If update is more recent than our kvmap, apply it
        if (kvmsg_sequence (kvmsg) > self->sequence) {
            self->sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: received update=%d", (int) self->sequence);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    else
        kvmsg_destroy (&kvmsg);

    return 0;
}
```

이 모델은 수백 줄의 코드에 불과하지만 작동하기 위해 많은 시간이 소요되었습니다. 정확히 말하면 모델 6을 구현하는데 대략 1주일이 걸렸으며 "신이시여, 이건 예제로 하기에는 너무 복잡합니다." 하며 삽질하였습니다. 우리는 작은 응용프로그램에 지금까지 적용한 거의 개념들을 모아 조립하였습니다. 바이너리 스타, TTL, 유한 상태 머신, 리엑터, 장애조치, 임시값, 하위트리 등이 있습니다. 나를 놀라게 한 것은 앞선 설계가 꽤 정확하다는 것입니다. 여전히 많은 소켓 흐름을 작성하고 디버깅하는 것은 매우 어렵습니다.

리엑터 기반 설계는 코드에서 많은 지저분한 작업을 제거하여, 남은 것을 더 간단하고 이해하기 쉽게 합니다. "4장 - 신뢰할 수 있는 요청-응답 패턴"의 bstar 리액터를 재사용하였습

니다. 전체 서버가 하나의 스레드로 실행되므로 스레드 간 이상함이 일어나지 않습니다 - 모든 핸들러들에 전달된 구조체 포인터(self)만으로 즐거운 작업을 수행합니다. 리엑터 사용 시 하나의 좋은 효과는 폴링 루프와 약연결(loosely coupled)된 코드는 재사용하기 훨씬 쉽다는 것입니다. 모델 6의 큰 덩어리는 모델 5에서 가져왔습니다.

각 기능들을 하나씩 하나씩 구현하여, 다음 기능으로 넘어가기 전에 각 기능이 정상적으로 동작하게 하였습니다. 4~5개의 주요 소켓 흐름이 있기 때문에 많은 디버깅과 테스트가 필요했습니다. 디버깅은 컴퓨터 화면에 메시지를 덤프하여 수행하였습니다. ØMQ 응용프로그램을 단계별로 진행하기 위해 고적적인 디버거(gdb 등)를 사용하지 마십시오. 무슨 일이 일어나고 있는지 이해하려면 메시지 흐름을 확인해야 합니다.

테스트를 위해 항상 메모리 누수와 잘못된 메모리 참조를 잡기 위해 Valgrind를 사용하였습니다. C 언어의 주요 관심사로 컴파일러에서 가비지(garbage) 수집을 하지 않아 코드상에서 명시적으로 메모리 해제를 않으면 메모리 누수가 발생할 수 있습니다. 이때 kvmsg 및 CZMQ와 같은 적절하고 일관된 추상화를 사용하면 엄청난 도움이 됩니다.

- [옮긴이] "clonecli6.c" 상태(INITIAL, SYNCING, ACTIVE)를 PUSH-PULL을 PUB-SUB로 변경한 클라이언트의 마지막 모델(모델 6)로 다음 주제에서 설명합니다.

- [옮긴이] 빌드 및 테스트

```
cl -EHsc clonesrv6.c libzmq.lib czmq.lib
cl -EHsc clonecli6.c libzmq.lib czmq.lib

./clonesrv6 -p
20-08-29 16:13:24 I: primary active, waiting for backup (passive)
D: 20-08-29 16:13:24 zloop: register SUB poller (000001EEAEB11AC0, 0)
D: 20-08-29 16:13:24 zloop: register timer id=2 delay=1000 times=0
D: 20-08-29 16:13:24 zloop: register timer id=3 delay=1000 times=0
D: 20-08-29 16:13:24 zloop polling for 986 msec
D: 20-08-29 16:13:25 zloop: call timer handler id=1
...
```

```
20-08-29 16:13:41 I: connected to backup (passive), ready as active
D: 20-08-29 16:13:41 zloop: cancel SUB poller (000001EEAE518160, 0)
D: 20-08-29 16:13:41 zloop polling for 346 msec
D: 20-08-29 16:13:41 zloop: call timer handler id=1
...
D: 20-08-29 16:14:03 zloop: interrupted

 ./clonesrv6 -b
20-08-29 16:13:40 I: backup passive, waiting for primary (active)
D: 20-08-29 16:13:40 zloop: register SUB poller (000001F9CDC93690, 0)
D: 20-08-29 16:13:40 zloop: register timer id=2 delay=1000 times=0
D: 20-08-29 16:13:40 zloop: register timer id=3 delay=1000 times=0
D: 20-08-29 16:13:40 zloop polling for 985 msec
D: 20-08-29 16:13:40 zloop: call SUB socket handler (000001F9CD681270, 0)
...
D: 20-08-29 16:13:41 zloop: call SUB socket handler (000001F9CD681270, 0)
20-08-29 16:13:41 I: connected to primary (active), ready as passive
...
0-08-29 16:14:07 I: failover successful, ready as active
D: 20-08-29 16:14:07 zloop: cancel SUB poller (000001F9CD694FA0, 0)

 ./clonecli6
20-08-29 16:13:51 I: adding server tcp://localhost:5556...
20-08-29 16:13:51 I: waiting for server at tcp://localhost:5556...
20-08-29 16:13:51 I: adding server tcp://localhost:5566...
20-08-29 16:13:51 I: received from tcp://localhost:5556 snapshot=0
20-08-29 16:13:52 I: received from tcp://localhost:5556 update=1
20-08-29 16:13:53 I: received from tcp://localhost:5556 update=2
...
```

```
20-08-29 16:14:07 I: server at tcp://localhost:5556 didn't give HUGZ
20-08-29 16:14:07 I: waiting for server at tcp://localhost:5566...
```

0.59.9 클러스터된 해시맵 통신규약

모델 6 서버는 이전 모델(모델 5(임시값 + TTL))에 바이너리 스타 패턴을 매우 많이 혼합시켰지만, 모델 6 클라이언트는 좀 더 많이 복잡합니다. 클라이언트에 대해 알아보기 전에 최종 통신규약을 보면, 클러스터된 해시맵 통신규약으로 ØMQ RFC 사이트에 사양서로 작성해 두었습니다.

- [옮긴이] [클러스터된 해시 맵 통신규약](https://rfc.zeromq.org/spec/12/)은 일련의 클라이언트들 간에 공유하기 위한 클러스터 전반의 키-값 해시 맵과 메커니즘을 정의합니다.

대략적으로 이와 같은 복잡한 통신규약을 설계하는 방법에는 두 가지가 있습니다. 첫 번째 방법은 각 흐름을 자체 소켓 집합으로 분리하는 것입니다. 이것이 우리가 여기서 사용한 접근 방식입니다. 장점은 각 흐름이 단순하고 깔끔하다는 것입니다. 단점은 한 번에 다중 소켓 흐름을 관리하는 것이 매우 복잡할 수 있습니다. 리엑터를 사용하면 더 단순해지지만 그래도 정상적으로 동작하게 하기 위해 함께 조정할 부분들이 많습니다.

두 번째 방법은 모든 것에 단일 소켓 쌍을 사용하는 것입니다. 이 경우 서버에는 ROUTER를, 클라이언트들에는 DEALER를 사용하여 해당 연결상에서 모든 작업을 수행했습니다. 이러한 방법은 더 복잡한 통신규약을 만들지만 적어도 복잡성은 모두 한곳에 집중되어 있습니다. "7장 - ØMQ 활용한 고급 아키텍처"에서 ROUTER-DEALER 조합을 통해 수행되는 통신규약의 예를 살펴보겠습니다.

CHP 사양서를 살펴보겠습니다. "SHOULD", "MUST" 및 "MAY"는 통신규약 사양서에서 요구 사항 수준을 나타내기 위해 사용되는 핵심 용어라는 점에 주의하십시오.

0.59.9.1 목표

CHP는 ØMQ 네트워크에 연결된 클라이언트들의 클러스터에서 신뢰할 수 있는 발행-구독에 대한 기반을 제공합니다. "해시 맵" 추상화를 키-값 쌍으로 구성하여 정의합니다. 모든

클라이언트들은 언제든지 키-값 쌍을 수정할 수 있으며, 변경 사항은 모든 클라이언트들에게 전파됩니다. 클라이언트는 언제든지 네트워크에 참여할 수 있습니다.

0.59.9.2 아키텍처

CHP는 일련의 클라이언트와 서버 응용프로그램들을 연결합니다. 클라이언트들은 서버에 연결합니다. 클라이언트는 서로를 보지 못합니다. 클라이언트들은 언제든지 클러스터에 들어오고 나갈 수 있습니다.

0.59.9.3 포트들과 접속들

서버는 3개의 포트를 오픈하며 다음과 같습니다. * 스냅샷 포트(ØMQ ROUTER 소켓)로 포트 번호는 P. * 발행자 포트(ØMQ PUB 소켓)로 포트 번호는 P+1. * 수집자 포트(ØMQ SUB 소켓)로 포트 번호는 P+2.

클라이언트는 적어도 2개의 연결들이 오픈되어야 합니다. * 스냅샷 연결(ØMQ DEALER 소켓)로 포트 번호는 P. * 구독자 연결(ØMQ SUB 소켓)로 포트 번호는 P+1. 클라이언트는 3번째 연결을하여 해시맵 변경을 서버로 전달하면, 서버에서 각 클라이언트들에게 전파합니다. * 발행자 연결(ØMQ PUB 소켓)으로 포트 번호는 P+2

명령(SUBTREE, SET, GET, CONNECT)에 있는 추가 프레임은 아래에서 설명합니다.

0.59.9.4 상태 동기화

```
[클라이언트] ICANHAZ 명령
----------------------------------
Frame 0: "ICANHAZ?"
Frame 1: 하위트리 사양서(예 : "/client/")
```

2개 프레임들은 모두 ØMQ 문자열입니다. 하위트리 사양은 공백(“”)일 수 있지만 공백이 아닐 경우 (clone_subtree()로 설정) 슬래시(/) 이후 구성되는 경로명이며, 슬래시(/)로 끝납니다.(예 : “/client/”) 서버는 스냅샷 포트(ROUTER)로부터 “ICANHAZ?” 받고 0개 이상의 KVSYNC 명령으로 스냅샷 포트에 kvmsg를 전송 한 다음 “KTHXBAI” 명령을 전송합니다.

서버는 “ICANHAZ?” 명령에 의해 제공되는 클라이언트 식별자(ID)를 각 메시지들의 선두에 부여합니다. KVSYNC 명령은 다음과 같이 단일 키-값 쌍을 지정합니다.

```
[섭버] KVSYNC 명령
----------------------------------
Frame 0: 키, ØMQ 문자열
Frame 1: 순서 번호, 8 바이트 네트워크 순서
Frame 2: <공백>                  --> UUID
Frame 3: <공백>                  --> PROPERTIES
Frame 4: 값, 이진대형객체(Blob : Binary Large Object)
```

The KTHXBAI command takes this form:

순서 번호는 중요하지 않으며 0일 수도 있습니다. KTHXBAI 명령은 다음과 같은 형태입니다.

```
[서버] KTHXBAI 명령
----------------------------------
Frame 0: "KTHXBAI"
Frame 1: 순서 번호, 8 바이트 네트워크 순서
Frame 2: <공백>
Frame 3: <공백>
Frame 4: 서브트리 사양서(예 : "/client/")
```

순서 번호는 이전에 보낸 KVSYNC 명령의 가장 높은 순서 번호입니다. 클라이언트의 스냅샷 소켓(DEALER)를 통해 KTHXBAI 명령을 수신하면, 구독자 연결(SUB)에서 메시지를 수신하고 적용하기 시작합니다(SHOULD).

0.59.9.5 서버에서 클라이언트로 전달되는 변경정보들

서버에 해시 맵에 대한 변경정보가 있을 때 발행자 소켓(PUB)에 KVPUB 명령으로 브로드캐스트 해야 합니다. KVPUB 명령의 형식은 다음과 같습니다.

```
[서버] KVPUB 명령
----------------------------------
Frame 0: 키, ØMQ 문자열
Frame 1: 순서 번호, 8 바이트 네트워크 순서
Frame 2: UUID, 16 바이트
Frame 3: 추가 속성들, ØMQ 문자열
Frame 4: 값, 이진대형객체(Blob : Binary Large Object)
```

순서 번호는 반드시 증가해야 합니다. 클라이언트는 순서 번호가 마지막으로 수신된 KTHXBAI 또는 KVPUB 명령보다 크지 않은 모든 KVPUB 명령을 폐기됩니다. UUID는 선택 사항이며 "Frame 2"는 공백일 수 있습니다(크기 0). 추가 속성들은 0개 이상의 "name = value" 형태와 뒤에 개행 문자로 구성됩니다. 키-값 쌍에 추가 속성들이 없는 경우 속성 필드는 공백입니다.

다른 변경정보들이 없는 경우 서버는 일정한 간격(예 : 1초당 한번)으로 HUGZ 명령을 발행자 소켓(PUB)으로 보내야 합니다. HUGZ 명령의 형식은 다음과 같습니다.

- [옮긴이] HUGZ는 상대편 서버 및 클라이언트들에게 보냅니다.

```
[서버] HUGZ 명령
----------------------------------
Frame 0: "HUGZ"
Frame 1: 00000000
Frame 2: <공백>
Frame 3: <공백>
Frame 4: <공백>
```

클라이언트는 일정한 간격으로 HUGZ 메시지가 없을 경우, 서버에 장애가 발생했다는 표시로 사용됩니다(아래 안정성 참조).

0.59.9.6 클라이언트에서 서버로 전달되는 변경정보들

클라이언트가 해시 맵에 대한 변경정보를 가지고 있을 때, KVSET 명령으로 발행자 연결(PUB)을 통해 변경정보를 서버에 보낼 수 있습니다. KVSET 명령의 형식은 다음과 같습니다.

```
[클라이언트] KVSET 명령
---------------------------------
Frame 0: 키, ØMQ 문자열
Frame 1: 순서 번호, 8 바이트 네트워크 순서
Frame 2: UUID, 16 바이트
Frame 3: 추가 속성들, ØMQ 문자열
Frame 4: 값, 이진대형객체(Blob : Binary Large Object)
```

If the value is empty, the server MUST delete its key-value entry with the specified key. The server SHOULD accept the following properties:

순서 번호는 중요하지 않으며 0일 수 있습니다. UUID는 고유한 식별자로 신뢰성 있는 서버 아키텍처에서 사용됩니다. 만약 값이 공백(서버에서 TTL에 의한 값을 공백으로 치환)이면, 서버는 해당 키에 대한 키-값 항목을 삭제해야 합니다. 서버는 다음 속성을 수락해야 합니다. * TTL(Time to Live) : 유효시간(초)을 지정합니다. KVSET 명령에 ttl 속성이 있는 경우, 서버는 키-값 쌍을 삭제하고 값이 비어있는 kvsmg를 KVPUB 명령을 통해 클라이언트들로 전송해야 하며, 클라이언트에서는 TTL이 만료되었을 때 삭제합니다.

0.59.9.7 안정성

To assist server reliability, the client MAY:

CHP는 기본 서버가 실패할 경우 백업 서버가 인계받는 이중 서버 구성으로 사용할 수 있습니다. CHP는 이런 장애조치에 사용되는 메커니즘을 정의하지 않지만 바이너리 스타 패턴이 도움이 될 수 있습니다. 서버 안정성을 지원하기 위해 클라이언트는 다음을 수행할 수 있습니다.

- 모든 KVSET 명령에서 UUID를 설정합니다.

- 일정 기간 동안 HUGZ 메시지 부재 감지하고 이를 현재 서버가 실패했음을 나타내는 표시로 사용합니다.
- 백업 서버에 연결하고 상태 동기화(ICANHAZ~KTXBAI)를 다시 요청합니다.

0.59.9.8 확장성 및 성능

CHP는 브로커의 많은 수(수천)의 클라이언트들로 확장 가능하도록 설계되었으며, 단지 브로커상의 시스템 자원에 의해 제한받습니다. 모든 클라이언트들의 변경정보들이 단일 서버를 통과하기 때문에, 전체 처리량은 피크시 초당 수백만 개의 변경정보들로 제한될 수 있으며, 아마도 더 적을 수 있습니다.

0.59.9.9 보안

CHP는 인증, 접근 제어 또는 암호화 메커니즘을 구현하지 않았으며, 이러한 메커니즘이 필요한 환경에서 사용해서는 안됩니다.

0.59.10 멀티스레드 스택과 API 구축

지금까지 사용한 클라이언트 스택은 CHP 통신규약을 제대로 처리할 만큼 똑똑하지 않습니다. 심박을 시작하자마자 백그라운드 스레드에서 실행할 수 있는 클라이언트 스택이 필요합니다. "4장 - 신뢰할 수 있는 요청-응답 패턴"의 마지막 있는 프리렌서 패턴에서 멀티스레드 API를 사용했지만 자세히 설명하지 않았습니다. 멀티스레드 API는 CHP와 같은 복잡한 ØMQ 통신규약을 구현할 때 매우 유용합니다.

그림 63 - 멀티스레드 API

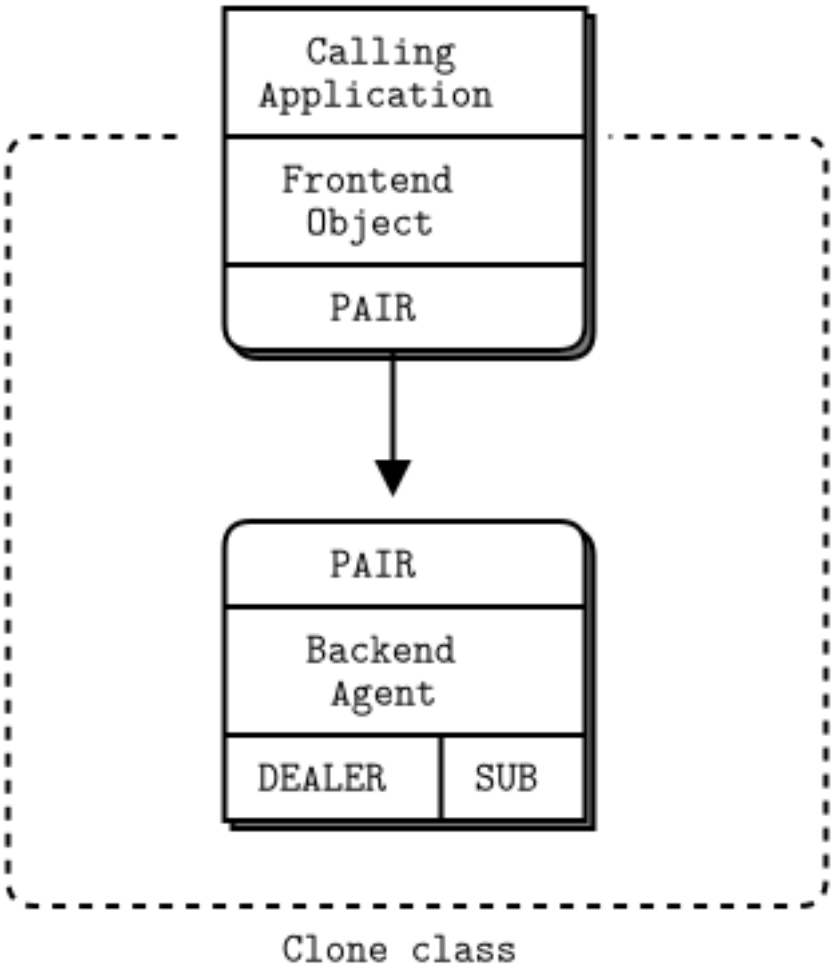


그림 65: Multithreaded API

중요한 통신규약을 만들고 응용프로그램에서 제대로 구현될 것으로 기대한다면, 대부분의 개발자는 잘못된 길로 갈 수가 있습니다. 당신은 통신규약이 너무 복잡하고, 너무 섬세하며, 너무 사용하기 어렵다고 불평하는 많은 불행한 사람들과 함께 남겨질 것입니다. 개발자들에게 호출할 간단한 API를 제공할 수 있다면, 당신은 API를 구매하려 할 것입니다.

멜티스레드 API는 2개의 PAIR 소켓으로 연결된 프론트엔드 개체와 백그라운드 에이전트로 구성됩니다. 이와 같이 2개의 PAIR 소켓을 연결하는 것은 매우 유용하여 ØMQ의 C 개발언어에서 제공하는 고수준의 바인딩인 CZMQ에서 수행합니다. 이것은 "신규 스레드를 생성시에 메시지를 보내는 데 사용할 수 있는 파이프를 사용"하는 방법입니다.

- [옮긴이] “clone.c”에서 아래와 같이 사용합니다.
- self->pipe = zthread_fork (self->ctx, clone_agent, NULL);

본 가이드에서 볼 수 있는 멀티스레드 API는 모두 동일한 형식을 가집니다.

- 객체의 생성자(clone_new())는 컨텍스트(Context)를 생성하고 파이프로 연결된 백그라운드 스레드(clone_agent())를 시작합니다. 파이프의 한쪽 끝을 잡고 있으므로 백그라운드 스레드에 명령을 보낼 수 있습니다.
- 백그라운드 스레드(clone_agent())는 에이전트를 시작하여 기본적으로 zmq_poll 루프를 통하여 파이프 소켓과 다른 소켓들(DEALER, SUB 소켓)을 읽습니다.
- 메인 응용프로그램 스레드와 백그라운드 스레드는 ØMQ 메시지를 통하여 통신합니다. 규칙에 따라 프론트엔드는 다음과 같이 문자열 명령(CONNECT, GET, SET 등)을 보내면 클래스의 각 메서드가 백엔드 에이전트에 전송되는 메시지로 전환(agent_control_message())합니다.

```
void
clone_connect (clone_t *self, char *address, char *service)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, address);
    zmsg_addstr (msg, service);
    zmsg_send (&msg, self->pipe);
}
```

- 메서드에 반환 코드가 필요한 경우 에이전트의 응답 메시지를 기다릴 수 있습니다.
- 에이전트가 비동기 이벤트들을 프론트엔드로 다시 보내야 하는 경우, recv() 메서드를 추가하여 프론트엔드 파이프에서 메시지를 기다립니다.
- 프론트엔드 파이프 소켓 핸들을 노출하여 추가 폴링 루프에 통합할 수 있습니다. 그렇지 않으면 모든 recv() 메서드가 응용프로그램을 차단합니다.

clone 클래스는 "4장 - 신뢰할 수 있는 요청-응답 패턴"의 프리랜서 패턴의 flcliapi 클래스와 동일한 구조를 가지며 복제 클라이언트의 마지막 모델(모델 5(임시값 + TTL))에서 고가용성 기능을 추가합니다. ØMQ가 없으면 이런 종류의 멀티스레드 API 설계는 몇 주동안 정말 힘든 작업이 될 것입니다. ØMQ에서는 하루나 이틀 정도의 작업이었습니다.

clone 클래스의 실제 API 메서드는 매우 간단합니다.

```
//  Create a new clone class instance
clone_t *
    clone_new (void);

//  Destroy a clone class instance
void
    clone_destroy (clone_t **self_p);

//  Define the subtree, if any, for this clone class
void
    clone_subtree (clone_t *self, char *subtree);

//  Connect the clone class to one server
void
    clone_connect (clone_t *self, char *address, char *service);

//  Set a value in the shared hashmap
void
    clone_set (clone_t *self, char *key, char *value, int ttl);

//  Get a value from the shared hashmap
char *
    clone_get (clone_t *self, char *key);
```

복제 클라이언트의 모델 6 코드가 있으며 clone 클래스를 사용하여 얇은 껍질에 불과하게

되었습니다.

clonecli6.c : 복제 클라이언트, 모델 6

```
//  Clone client Model Six

//  Lets us build this source without creating a library
#include "clone.c"
#define SUBTREE "/client/"

int main (void)
{
    //  Create distributed hash instance
    clone_t *clone = clone_new ();

    //  Specify configuration
    clone_subtree (clone, SUBTREE);
    clone_connect (clone, "tcp://localhost", "5556");
    clone_connect (clone, "tcp://localhost", "5566");

    //  Set random tuples into the distributed hash
    while (!zctx_interrupted) {
        //  Set random value, check it was stored
        char key [255];
        char value [10];
        sprintf (key, "%s%d", SUBTREE, randof (10000));
        sprintf (value, "%d", randof (1000000));
        clone_set (clone, key, value, randof (30));
        sleep (1);
    }
    clone_destroy (&clone);
```

```
    return 0;
}
```

하나의 서버 단말을 지정하는 연결 방법에 유의하십시오. 내부적으로 우리는 실제로 3개의 포트들과 통신을 필요하며, CHP 통신규약에서 알 수 있듯이 3개의 포트는 연속 포트 번호입니다.

- 서버 상태(snapshot) 라우터(ROUTER 소켓)는 포트 번호 P.
- 서버 변경정보 발행자(PUB)는 포트 번호 P + 1.
- 서버 변경정보 구독자(SUB)는 포트 번호 P + 2.

따라서 3개의 연결을 하나의 논리적 기능(이는 3개의 개별 ØMQ 연결 호출로 구현)으로 수행할 수 있습니다.

복제 스택에 대한 소스 코드로 마무리하겠습니다. 이것은 복잡한 코드이지만 프론트엔드 객체 클래스와 백엔드 에이전트로 분리하면 이해하기 더 쉽습니다. 프론트엔드는 문자열 명령들("SUBTREE", "CONNECT", "SET", "GET")을 에이전트에 전송합니다. 에이전트는 이러한 명령들을 처리하고 서버와 통신합니다. 에이전트의 처리 로직은 다음과 같습니다.

1. 첫 번째 서버에서 스냅샷을 가져와서 시작
2. 구독자 소켓에서 읽기로 전환하여 서버로부터 스냅샷을 받을 때
3. 서버로부터 스냅샷을 받지 못하면 두 번째 서버로 장애조치합니다.
4. 파이프(pipe)와 구독자(SUB) 소켓에서 폴링합니다.
5. 파이프(pipe)에 입력이 있으면 프론트엔드 개체에서 전달된 제어 메시지("SUBTREE", "CONNECT", "SET", "GET")를 처리합니다.
6. 구독자(SUB)에 대한 입력이 있으면 변경정보를 저장하거나 적용하십시오.
7. 일정 시간 내에 서버에서 아무것도 받지 못한다면 장애조치합니다.
8. Ctrl-C로 프로세스가 중단될 때까지 반복합니다.

아래는 실제 `clone` 클래스가 구현된 코드입니다.

clone.c: 복제 클래스

```
//  clone class - Clone client API stack (multithreaded)

#include "clone.h"
//  If no server replies within this time, abandon request
#define GLOBAL_TIMEOUT  4000    //  msecs

//  ---------------------------------------------------------------------
//  Synchronous part, works in our application thread

//  Structure of our class

struct _clone_t {
    zctx_t *ctx;                //  Our context wrapper
    void *pipe;                 //  Pipe through to clone agent
};

//  This is the thread that handles our real clone class
static void clone_agent (void *args, zctx_t *ctx, void *pipe);

//  .split constructor and destructor
//  Here are the constructor and destructor for the clone class. Note that
//  we create a context specifically for the pipe that connects our
//  frontend to the backend agent:

clone_t *
clone_new (void)
{
    clone_t
        *self;
```

```
    self = (clone_t *) zmalloc (sizeof (clone_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, clone_agent, NULL);
    return self;
}

void
clone_destroy (clone_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        clone_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  .split subtree method
//  Specify subtree for snapshot and updates, which we must do before
//  connecting to a server as the subtree specification is sent as the
//  first command to the server. Sends a [SUBTREE][subtree] command to
//  the agent:

void clone_subtree (clone_t *self, char *subtree)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
```

```
    zmsg_addstr (msg, "SUBTREE");
    zmsg_addstr (msg, subtree);
    zmsg_send (&msg, self->pipe);
}

//  .split connect method
//  Connect to a new server endpoint. We can connect to at most two
//  servers. Sends [CONNECT][endpoint][service] to the agent:

void
clone_connect (clone_t *self, char *address, char *service)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, address);
    zmsg_addstr (msg, service);
    zmsg_send (&msg, self->pipe);
}

//  .split set method
//  Set a new value in the shared hashmap. Sends a [SET][key][value][ttl]
//  command through to the agent which does the actual work:

void
clone_set (clone_t *self, char *key, char *value, int ttl)
{
    char ttlstr [10];
    sprintf (ttlstr, "%d", ttl);
```

```
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SET");
    zmsg_addstr (msg, key);
    zmsg_addstr (msg, value);
    zmsg_addstr (msg, ttlstr);
    zmsg_send (&msg, self->pipe);
}

//  .split get method
//  Look up value in distributed hash table. Sends [GET][key] to the agent and
//  waits for a value response. If there is no value available, will eventually
//  return NULL:

char *
clone_get (clone_t *self, char *key)
{
    assert (self);
    assert (key);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "GET");
    zmsg_addstr (msg, key);
    zmsg_send (&msg, self->pipe);

    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *value = zmsg_popstr (reply);
        zmsg_destroy (&reply);
```

```
        return value;
    }
    return NULL;
}

//  .split working with servers
//  The backend agent manages a set of servers, which we implement using
//  our simple class model:

typedef struct {
    char *address;              //  Server address
    int port;                   //  Server port
    void *snapshot;             //  Snapshot socket
    void *subscriber;           //  Incoming updates
    uint64_t expiry;            //  When server expires
    uint requests;              //  How many snapshot requests made?
} server_t;

static server_t *
server_new (zctx_t *ctx, char *address, int port, char *subtree)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));

    zclock_log ("I: adding server %s:%d...", address, port);
    self->address = strdup (address);
    self->port = port;

    self->snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (self->snapshot, "%s:%d", address, port);
```

```
    self->subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "%s:%d", address, port + 1);
    zsocket_set_subscribe (self->subscriber, subtree);
    zsocket_set_subscribe (self->subscriber, "HUGZ");
    return self;
}

static void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->address);
        free (self);
        *self_p = NULL;
    }
}

//  .split backend agent class
//  Here is the implementation of the backend agent itself:

//  Number of servers to which we will talk to
#define SERVER_MAX      2

//  Server considered dead if silent for this long
#define SERVER_TTL      5000    //  msecs

//  States we can be in
```

```
#define STATE_INITIAL       0   //  Before asking server for state
#define STATE_SYNCING       1   //  Getting state from server
#define STATE_ACTIVE        2   //  Getting new updates from server

typedef struct {
    zctx_t *ctx;                //  Context wrapper
    void *pipe;                 //  Pipe back to application
    zhash_t *kvmap;             //  Actual key/value table
    char *subtree;              //  Subtree specification, if any
    server_t *server [SERVER_MAX];
    uint nbr_servers;           //  0 to SERVER_MAX
    uint state;                 //  Current state
    uint cur_server;            //  If active, server 0 or 1
    int64_t sequence;           //  Last kvmsg processed
    void *publisher;            //  Outgoing updates
} agent_t;

static agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->kvmap = zhash_new ();
    self->subtree = strdup ("");
    self->state = STATE_INITIAL;
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    return self;
}
```

```
static void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        int server_nbr;
        for (server_nbr = 0; server_nbr < self->nbr_servers; server_nbr++)
            server_destroy (&self->server [server_nbr]);
        zhash_destroy (&self->kvmap);
        free (self->subtree);
        free (self);
        *self_p = NULL;
    }
}

//  .split handling a control message
//  Here we handle the different control messages from the frontend;
//  SUBTREE, CONNECT, SET, and GET:

static int
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);
    if (command == NULL)
        return -1;      //  Interrupted
```

```
    if (streq (command, "SUBTREE")) {
        free (self->subtree);
        self->subtree = zmsg_popstr (msg);
    }
    else
    if (streq (command, "CONNECT")) {
        char *address = zmsg_popstr (msg);
        char *service = zmsg_popstr (msg);
        if (self->nbr_servers < SERVER_MAX) {
            self->server [self->nbr_servers++] = server_new (
                self->ctx, address, atoi (service), self->subtree);
            //  We broadcast updates to all known servers
            zsocket_connect (self->publisher, "%s:%d",
                address, atoi (service) + 2);
        }
        else
            zclock_log ("E: too many servers (max. %d)", SERVER_MAX);
        free (address);
        free (service);
    }
    else
    //  .split set and get commands
    //  When we set a property, we push the new key-value pair onto
    //  all our connected servers:
    if (streq (command, "SET")) {
        char *key = zmsg_popstr (msg);
        char *value = zmsg_popstr (msg);
        char *ttl = zmsg_popstr (msg);
```

```
        //  Send key-value pair on to server
        kvmsg_t *kvmsg = kvmsg_new (0);
        kvmsg_set_key  (kvmsg, key);
        kvmsg_set_uuid (kvmsg);
        kvmsg_fmt_body (kvmsg, "%s", value);
        kvmsg_set_prop (kvmsg, "ttl", ttl);
        kvmsg_send     (kvmsg, self->publisher);
        kvmsg_store    (&kvmsg, self->kvmap);
        free (key);
        free (value);
        free (ttl);
    }
    else
    if (streq (command, "GET")) {
        char *key = zmsg_popstr (msg);
        kvmsg_t *kvmsg = (kvmsg_t *) zhash_lookup (self->kvmap, key);
        byte *value = kvmsg? kvmsg_body (kvmsg): NULL;
        if (value)
            zmq_send (self->pipe, value, kvmsg_size (kvmsg), 0);
        else
            zstr_send (self->pipe, "");
        free (key);
    }
    free (command);
    zmsg_destroy (&msg);
    return 0;
}

//  .split backend agent
```

```
//  The asynchronous agent manages a server pool and handles the
//  request-reply dialog when the application asks for it:

static void
clone_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    while (true) {
        zmq_pollitem_t poll_set [] = {
            { pipe, 0, ZMQ_POLLIN, 0 },
            { 0,    0, ZMQ_POLLIN, 0 }
        };
        int poll_timer = -1;
        int poll_size = 2;
        server_t *server = self->server [self->cur_server];
        switch (self->state) {
            case STATE_INITIAL:
                //  In this state we ask the server for a snapshot,
                //  if we have a server to talk to...
                if (self->nbr_servers > 0) {
                    zclock_log ("I: waiting for server at %s:%d...",
                        server->address, server->port);
                    if (server->requests < 2) {
                        zstr_sendm (server->snapshot, "ICANHAZ?");
                        zstr_send  (server->snapshot, self->subtree);
                        server->requests++;
                    }
                    server->expiry = zclock_time () + SERVER_TTL;
```

```
                self->state = STATE_SYNCING;
                poll_set [1].socket = server->snapshot;
            }
            else
                poll_size = 1;
            break;

        case STATE_SYNCING:
            //  In this state we read from snapshot and we expect
            //  the server to respond, else we fail over.
            poll_set [1].socket = server->snapshot;
            break;

        case STATE_ACTIVE:
            //  In this state we read from subscriber and we expect
            //  the server to give HUGZ, else we fail over.
            poll_set [1].socket = server->subscriber;
            break;
    }
    if (server) {
        poll_timer = (server->expiry - zclock_time ())
                   * ZMQ_POLL_MSEC;
        if (poll_timer < 0)
            poll_timer = 0;
    }
    //  .split client poll loop
    //  We're ready to process incoming messages; if nothing at all
    //  comes from our server within the timeout, that means the
    //  server is dead:
```

```
    int rc = zmq_poll (poll_set, poll_size, poll_timer);
    if (rc == -1)
        break;              //  Context has been shut down

    if (poll_set [0].revents & ZMQ_POLLIN) {
        if (agent_control_message (self))
            break;          //  Interrupted
    }
    else
    if (poll_set [1].revents & ZMQ_POLLIN) {
        kvmsg_t *kvmsg = kvmsg_recv (poll_set [1].socket);
        if (!kvmsg)
            break;          //  Interrupted

        //  Anything from server resets its expiry time
        server->expiry = zclock_time () + SERVER_TTL;
        if (self->state == STATE_SYNCING) {
            //  Store in snapshot until we're finished
            server->requests = 0;
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                self->state = STATE_ACTIVE;
                zclock_log ("I: received from %s:%d snapshot=%d",
                    server->address, server->port,
                    (int) self->sequence);
                kvmsg_destroy (&kvmsg);
            }
            else
```

```
                    kvmsg_store (&kvmsg, self->kvmap);
            }
            else
            if (self->state == STATE_ACTIVE) {
                //  Discard out-of-sequence updates, incl. HUGZ
                if (kvmsg_sequence (kvmsg) > self->sequence) {
                    self->sequence = kvmsg_sequence (kvmsg);
                    kvmsg_store (&kvmsg, self->kvmap);
                    zclock_log ("I: received from %s:%d update=%d",
                        server->address, server->port,
                        (int) self->sequence);
                }
                else
                    kvmsg_destroy (&kvmsg);
            }
        }
        else {
            //  Server has died, failover to next
            zclock_log ("I: server at %s:%d didn't give HUGZ",
                    server->address, server->port);
            self->cur_server = (self->cur_server + 1) % self->nbr_servers;
            self->state = STATE_INITIAL;
        }
    }
    agent_destroy (&self);
}
```

복제 클라이언트의 모델 6 까지 구현하였으며, 이장을 마지막으로 기본적인 구현 방법 마무리 하였습니다.

후기

0.60 라이센스

나는 많은 사람들이 본 가이드의 텍스트를 자신들의 작업에서 재사용하기를 바랍니다 : 잡지, 책, 프레젠테이션 등. 조건은 누군가가 본 가이드를 재창작할 경우 다른 사람도 또 다시 재창작할 수 있어야 합니다. 누군가가 재창작된 대상을 통하여 수익을 얻는 것은 나에게 영광이며 이의는 없습니다. 본 가이드는 cc-by-sa 라이센스로 공개하고 있습니다.

예제 소스 코드는 GPL로 공개하였지만 곧 제대로 운용되지 않는 것을 알 수 있었습니다. 예제 소스 코드의 조각들을 다양한 용도로 재사용하여 ØMQ가 광범위하게 사용되길 희망하였지만 잘 되지 않았습니다. 그래서 예제 코드의 라이선스를 MIT/X11로 전환하였습니다. 규모가 크고 더 복잡한 예제 코드에 관해서는 LGPL하에서 운용됩니다.

그리고 예제 코드는 집사(Majordomo)처럼 독립적인 프로젝트로 전환을 시작했으며, LGPL을 사용하였습니다. 거듭 말하지만, 보급 및 재 편집성이 적합하기 때문입니다. 라이센스는 도구이며, 의도(intent)를 가지고 사용되지만 이데올로기(ideology)는 없습니다.