Libzmq Chapter 4 - Reliable Request-Reply Patterns

From 탱이의 잡동사니

Contents

Overview

Chapter 3 - Advanced Request-Reply Patterns 에서 ZeroMQ 의 좀더 심화된 request-reply pattern 을 확인할 수 있었다. 이번 챕터에서는 ZeroMQ 의 request-reply 패턴에서 신뢰성과 신뢰성 있는 메시징 패턴을 알아보도록 할 것이다.

이번 챕터에서는 재사용이 가능한 사용자 관점에서의 request-reply pattern 살펴볼 것이다. 즉, 아래의 내용들을 살펴볼 것이다.

- The Lazy Pirate pattern : reliable request-reply from the client side.
- The Simple Pirate pattern : reliable request-reply using load balancing.
- The Paranoid Pirate pattern : reliable request-reply with heart-beating.
- The Majordomo pattern : service-oriented reliable queuing.
- The Titanic pattern : disk-based/disconnected reliable queuing.
- The Binary Star pattern : primary-backup server failover.
- The Freelance pattern : brokerless reliable request-reply.

What is "Reliability"?

"신뢰성"을 이야기하는 대부분의 사람들은 그 참된 의미를 모르고 이야기를 한다. 오로지 실패의 측면에서 신뢰성을 정의할 수 있는데, 이 뜻은 우리가 실패의 의미를 잘 이해할 수만 있다면 실패를 신뢰할 수 있다는 뜻이라는 것이다. 그 이상도, 그 이하도 아닌 딱 그정도 인 것이다. 자 이제 ZeroMQ 에서 발생가능한 실패의 종류들에 대해 이야기 해 보자. 발생가능성이 큰 순서부터 이야기 할 것이다.

- Application 코드는 가장 최악의 공격자이다. 망가지고, 종료되고, 멈추고, INPUT 오류, 느린 동작, 메모리 누수 등과 같은 문제를 야기한다.
- 브로커와 같은 ZeroMQ 를 사용한 System 코드 역시 application 코드와 같이 엉망이 될 수 있다. System 코드는 application 코드보다 훨씬 더 신뢰성이 높아야 한다. 하지만 System 코드 역시 망가지고 부서진다. 특히 동작 속도가 느린 Client 에게 Queue Message 를 하는 경우 메모리 소모가 일어난다.

- Message Queue 역시 오버플로우될 수 있다. 일반적으로 느린 Client 에 대하여 엄청난 양의 데이터를 전송할 때 발생한다. Queue 에 오버플로우가 발생할 때, 추가되는 메시지를 그냥 무시하기 시작한다. 즉, "소실"되는 것이다.
- 네트워크도 실패할 수 있다(예를 들어 WIFI 에서 범위 밖으로 벗어나서 Switch-off 되는 경우를 생각해보자). 이런 경우, ZeroMQ 에서 자동적으로 재접속을 시도할 것이다. 하지만 그러는동안, 메시지는 소실 될 것이다.
- 하드웨어도 실패할 수 있으며 모든 프로세스에 영향을 줄 것이다.
- 드물지만, 네트워크 장비가 고장나서 실패가 발생 할 수도 있다.
- 데이터센터에 벼락이 치거나, 지진, 화재 그리고 전력 혹은 냉각 시스템 문제가 발생할 경우, 실패가 발생할 수도 있다.

위의 경우에 모두 대비할 수 있을 정도의 신뢰성있는 프로그램을 작성한다는 것은 말도 못하게 어렵고, 비싼 작업이다. 여기에서 그 부분을 다루는 건 범위를 벗어나므로 하지않을 것이다.

위에 설명한 사례 중, 처음 5가지의 경우가 전체 발생 실패의 거의 99.9%를 차지한다.

Designing Reliability

단순하게 이야기해서, 신뢰성이란, 코드가 정지되거나 망가지는 상황(즉, 코드가 죽는 상황)에서도 정상적으로 작동하게끔 하는 것이다. 하지만 이를 실제적으로 구현하기란 그리 단순하지가 않다. 즉, ZeroMQ 메시지 패턴 하나하나를 살펴보고 코드가 죽는 상황에서도 어떻게 작동이 되는지를 확인해야 한다.

- Request-reply

  만약 서버가 죽는다면(Request 도중), client 는 이를 바로 알아챌 수 있다(왜냐하면 Request 에 대한 응답을 하지 않으므로). 이후, 요청을 포기하거나 일정시간 대기 후 재시도 혹은 다른 서버를 찾아보거나 등의 행동을 취할 수 있다. 만약 Client 가 죽는 경우, 다른 어떤 이유로 인한 문제라고 간주할 수 있다.

- Pub-sub

  만약 client 가 죽는다면(약간의 데이터를 수신한채로), Server 로서는 알 방법이 없다. Pub-sub 패턴에서는 어떤 정보도 Client 에서 Server 로 전송되지 않는다. 하지만 Client는 Server 로 Out-of-band 방식으로 접속할 수 있다. Request-reply 패턴으로 접속해서 "내가 수신못한 데이터 전부를 다시 재전송 바랍니다"라고 메시지를 보낼 수 있다는 것이다. Server 가 죽는 경우는 여기의 범위를 벗어난다. Subscriber 들 역시 자체적으로 메시지 검증이 가능하므로, 만약 메시지 전송이 너무 느리거나 할 경우 별도의 행동을 취할 수 있다(경고를 보내는 등).

- Pipeline

  만약 worker 가 죽는 경우(작업중), ventilator 로써는 알 방법이 없다. Pipeline 은 굴러가는 기어와 같아서 오직 한 방향으로만 작업을 진행할 수 있다. 하지만 downstream collector 는 종료되지 못한 작업 내용을 알아챌 수 있으며, ventilator 에게 "324 번 작업을 다시 보내!"라고 메시지를 전송할 수 있다. 만약 ventilator 혹은 collector 가 죽은 경우, Upstream client 가 무엇이 되었든, Waiting 시간과 재전송의 작업을 피할 길이 없다. 비록 우아하지는 않지만, System code 가 죽지 않도록 하는 수밖에 없다.

여기에서는 request-reply 에 초점을 맞출 것이다. 가장 쉽기 때문이다.

기본적인 request-reply 패턴(REQ client 가 REP server 와 통신시, blocking send/receive 를 수행하는 것)은 대부분의 실패에서 가장 쉽게 다룰 수 있는 패턴이다. 만약 client가 request 를 보내는 동안 server 가 크래쉬를 일으킨다면, client는 영원히 멈춤상태가 될 것이다. 만약 네트워크에서 request 혹은 reply 를 유실했다면, 역시 client 는 영원히 멈춤상태가 된다.

하지만 고맙게도 ZeroMQ는 조용한 재접속 및 메시지 로드 밸런싱 등을 제공하기 때문에, Request-reply 는 여전히 TCP 보다 낫다. 하지만 역시 실제 상황에 적용하기에는 좀 부족한 면이 있다. 기본 request-reply 패턴에서 신뢰할만한 오직 한가지 경우는 네트워크나 서버가 죽을일이 없는 하나의 프로세스 안에서 동작하는 두개의 쓰레드끼리의 메시지 교환이다.

하지만 약간의 작업을 더한다면, 이 조잡한 작업은 실제 상황에서도 적용할 수 있는 훌륭한 reliable request-reply(RRR) 패턴이 된다. 이를 Pirate pattern 이라고 부른다.

Client 와 Server 는 크게 세가지 방법으로 연결될 수 있다. 신뢰성을 위해서는 각각의 방법마다 적절한 방법으로 접근해야 한다.

- Multiple clients talking to a single server(하나의 server와 여러 개의 client)

  Use case: a single well-known server to which clients need to talk.
  Type of failure we aim to handle: server crashes and restarts, and network disconnects.

- Multiple clients talking to a broker proxy that distributes work to multiple workers

  Use case: service-oriented transaction processing.
  Type of failure we aim to handle: worker crashes and restarts, worker busy looping, worker overload, queue crashes and restarts, and network disconnects.

- Multiple clients talking to multiple server with no intermediary proxies.

  Use case: distributed services such as name resolution.
  Type of failure we aim to handle: service crashes and restarts, service busy looping, service overload, and network disconnects.

각각의 접근방법에 따라 저마다의 trade-off 가 있으며, 두 개 이상의 경우를 묶어서도 사용한다.

Client-Side Reliability(Lazy Pirate Pattern)

request-reply 패턴에서 client를 약간만 수정하면 간단한 reliable request-reply 패턴을 만들 수 있다. 이를 Lazy Pirate Pattern 이라고 부른다. blocking receive 를 만드는 대신 다음을 할 것이다.

- Reply 가 확실히 도착했을 때에만 REQ Socket 을 Poll 하고 데이터를 수신한다.
- 지정된 timeout 시간동안 응답이 도착하지 않았을 경우, request 를 재전송 한다.
- 수 차례의 재시도에도 여전히 응답이 없을 경우, 트랜잭션을 포기한다.

만약 REQ socket 을 기본적인 send/receive 방식이 아닌 다른 방식으로 사용하려 한다면 에러를 발생할 것이다(기술적으로 REQ socket 은 send/receive ping-pong 을 위해 작은 유한-상태 머신으로 구현되어 있다. 그리고 에러 코드는 "EFSM"으로 불린다). 이 때문에 REQ socket 을 pirate pattern 으로 사용 시, reply 를 받기 까지 여러번의 request 를 전송해야 할 수도 있기에 약간 번거로울 수도 있다.

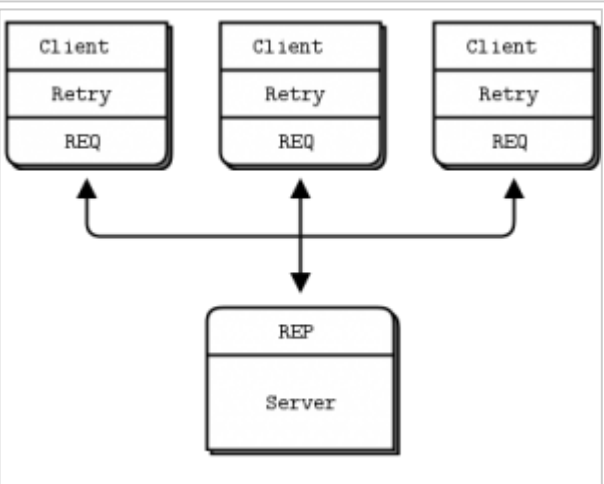
가장 쉽고 단순한 방법으로는 에러가 발생하면 REQ socket 을 닫고 다시 여는 것이다.

client.c

```
// Lazy Pirate client
// Use zmq_poll to do a safe request-reply
// To run, start client and then randomly kill/restart it.

#include <czmq.h>

#define REQUEST_TIMEOUT     2500    // msec, (> 1000!)
#define REQUEST_RETRIES     3       // Before we abandon
#define SERVER_ENDPOINT     "tcp://localhost:5555"

int main(int argc, char** argv)
{
    zctx_t* ctx;
    void* client;
    int sequence;
    int retries_left;

    ctx = zctx_new();
    client = zsocket_new(ctx, ZMQ_REQ);
    assert(client);

    printf("I: connecting to server....₩n");
    zsocket_connect(client, SERVER_ENDPOINT);

    sequence = 0;
    retries_left = REQUEST_RETRIES;
    while(retries_left && !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.
            }

            // Here we process a server reply and exit our loop if the
            // reply is valid. If we didn't a reply we close the client
            // socket and resend the request. We try a number of 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 {
                --retries_left;

                if(retries_left == 0) {
                    printf("E: server seems to be offline, abandoning.₩n");
                    break;
                }
                else {
                    printf("W: no response from server, retrying..₩n");
                    // Old socket is confused; close it and open a new one.
                    zsocket_destroy(ctx, client);
                    printf("I: reconnecting to serer...₩n");
                    client = zsocket_new(ctx, ZMQ_REQ);
                    zsocket_connect(client, SERVER_ENDPOINT);

                    // Send request again, on new socket
                    zstr_send(client, request);
                }
            }
        }
    }
    zctx_destroy(&ctx);
    return 0;
}
```

server.c

```
// 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"
#include <unistd.h>

int main(int argc, char** argv)
{
    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");
                sleep(2);
            }
        }

        printf("I: normal request (%s)₩n", request);
        sleep(1);
        s_send(server, request);
        free(request);
    }
    zmq_close(server);
    zmq_ctx_destroy(context);

    return 0;
}
```



The Lazy Pirate Pattern

위 코드를 테스트하려면 서버와 클라이언트를 각기 다른 두 개의 콘솔창에서 실행한다. 서버는 몇번의 정상적인 도작 이후, 이상 행동을 일으키기 시작할 것이다. 이후 클라이언트에서 어떤 반응이 나오는지 확인이 가능하다. 여기 간단한 서버족 예제가 있다.

```
I: normal request (1)
I: normal request (2)
I: normal request (3)
I: simulating CPU overload
```

```
I: normal request (4)
I: simulating a crash
```

클라이언트쪽의 응답이다.

```
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
```

클라이언트는 순차적인 메시지를 전송하며 응답메지가 정확한 응답이 맞는지를 확인한다. 테스트 시작 후, 처음 몇번 동안은 정상적으로 잘 동작하는 것처럼 보여지지만 곧 우리가 예상한 결과를 만들어낸다. 실제 환경에서도 이런 순차적인 메시지를 작성할 필요는 없다. 어디까지나 테스트를 쉽게 하기 위해 만든 것 뿐이다.

클라이언트는 REQ socket 을 사용한다. 그리고 brute force close/reopen 을 수행하는데, 왜냐하면 REQ socket 은 제한적인 send/receive cycle 만 수행하도록 제한되어 있기 때문이다. 아마도 DEALER 를 대신 사용하려고 할 수도 있다. 하지만 결코 좋은 선택은 아닐 것이다. 먼저, REQ socket 처럼 뭔가 비밀스러운 작업을 하는데 거기에 추가적으로 가림막을 친다(이게 무슨말인지 모르겠다면 그냥 하지 않는 것이 좋을 것이다). 두번째로 원치않는 응답을 받을 가능성이 있다.

여러 client 에서 하나의 server로 메시지를 전송하는 경우, 오직 client 에서만 실패를 관리할 수 있다. 이는 서버 크래시를 처리할 수 있지만, 같은 서버가 재시작을 할 경우에만 유효하다. 만약 서버 장비에서 파워 서플라이 장애와 같은 영구 장애가 발생했을 경우에는 도저히 처리할 방법이 없다. 대부분의 아키텍쳐에서 server application code 는 실패의 가장 큰 원인이기 때문에 단일 서버로만 구성한다는 것은 별로 좋은 생각이 아니다.

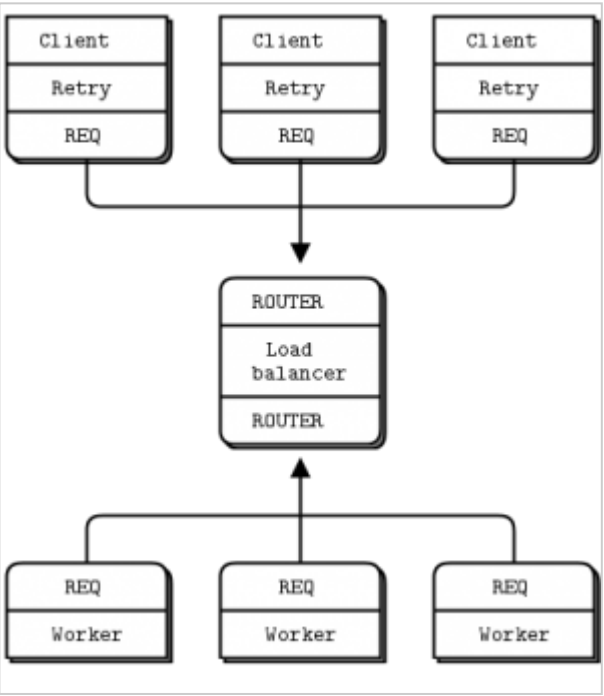
장점과 단점을 이야기하자면 다음과 같다.

- 장점 : 만들기 쉽고 이해하기 쉽다.
- 장점 : server/client 모두 큰 변경없이 동작이 가능하다.
- 장점 : 다시 재동작이 될 때까지 ZeroMQ 자체에서 재시도/재접속을 시도한다.
- 단점 : 특정 서버 장애를 복구할 수 없다.

Basic Reliable Queueing(Simple Pirate Pattern)

Lazy Pirate pattern 두 번째 확장은 Queue proxy 와 다중 server(worker) 조합이다. 최소한의 working model(Simple Pirate pattern)을 이용해서 한번 살펴보자.

모든 Pirate pattern 에서 worker 는 상태가 없다(stateles). 만약 application 에서 Shared database 와 같은 어떤 상태 공유를 요청한다면, 이 메시징 프레임워크 안에서는 알 방법이 없다. Queue proxy 를 가진다는 것은 worker 들이 client 가 전혀 모르게 왔다/갔다 할 수 있음을 의미하는 것이다. 만약 하나의 worker 가 죽는 경우, 다른 worker 가 이를 넘겨 받게 된다. 꽤 괜찮은 방법이다. 단순한 구성에서도 중앙 Queue 라고 불리는 오직 하나의 약점이 있을 뿐이다(manage problem/single point of failure).



The Simple Pirate Pattern

Queue Proxy 의 기본은 load balancing broker 이다. Blocked worker/Dead worker 를 처리하기 위한 가장 최소한의 방법은 무엇일까? 많은 노력을 할 필요는 없다. 이미 client 에는 retry 매커니즘이 있다. 그렇기에 load balancing pattern 을 이용한다면 잘 처리하도록 할 수 있을 것이다. 이는 ZeroMQ 의 철학과도 맞아떨어지며, 순전히 중간에 있는 proxy를 통하여 peer-to-peer pattern 을 마치 request-reply pattern 처럼 작성할 수 있는 것이다.

무슨 특수한 client 가 필요한 것이 아니다. 앞서 작성한 Lazy Pirate client 를 다시금 사용할 것이다. 아래에 load balancing broker 역할을 하는 queue 프로그램이 있다.

spqueue.c

```
// Simple Pirate broker
// This is identical to load-balancing pattern, with no reliability
// mechanisms. It depends on the client for recovery. Run forever.

#include <czmq.h>
#define WORKER_READY    "WOO1"      // signals worker is ready

int main(int argc, char** argv)
{
    zctx_t* ctx;
    void* frontend;
    void* backend;

    ctx = zctx_new();
    frontend = zsocket_new(ctx, ZMQ_ROUTER);
    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.
    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);
            }
        }
    }

    return 0;
}
```

이번에는 worker 이다. Lazy Pirate server 를 load balancing pattern 에 적용시킨 것이다(REQ "ready" 를 사용한다).

```
// Simple Pirate worker
// Connects REQ socket to tcp://*.5556
// Implements worker part of load-balancing

#include <czmq.h>
#define WORKER_READY    "\001"      // Signals worker is ready

int main(int argc, char** argv)
{
    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;
        }

        if((cycles > 3) && (randof(5) == 0)) {
            printf("I: (%s) simulating CPU overload\n", identity);
            sleep(3);
            if(zctx_interrupted) {
                break;
            }
        }
```

```
        printf("I: (%s) normal reply₩n", identity);
        sleep(1);
        zmsg_send(&msg, worker);
    }
    zctx_destroy(&ctx);

    return 0;
}
```

테스트를 하기 위해서는 worker, client, queue 들을 순서에 상관없이 실행시키면 된다. 일단 시작하게되면 어느 순간에 worker 에서 장애가 나는 것을 확인할 수 있고, client 는 retry 후 포기하는 것을 확인할 수 있다. Queue 는 계속해서 작동 할 것이다. 이후 worker/client 를 재시작하면 곧바로 다시 작동하는 것을 볼 수 있다. 참고로 worker/client 가 몇개가 되어도 상관없이 잘 작동한다.

Robust Reliable Queueing(Paranoid Pirate Pattern