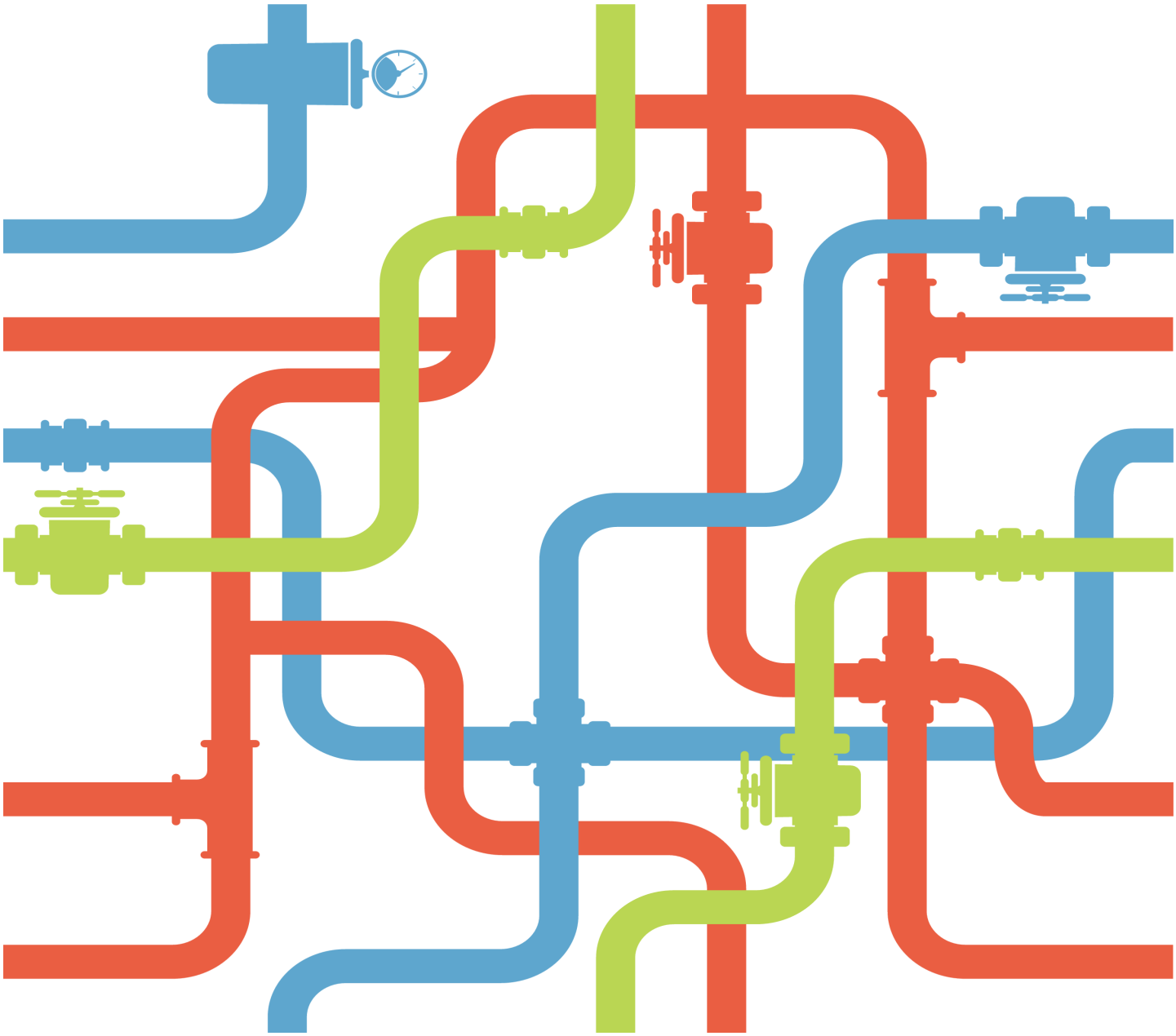
ØMQ

ガイドブック

Pieter Hintjens [著] 濱野 司 [訳]



第 0.6 版

ØMQガイドブック
ØMQ - The Guide

Pieter Hintjens 著
濱野　司 訳

目次

まえがき

訳者より

この本は ØMQ というライブラリの入門書という体裁になっていますが、もっと一般的なメッセージングシステムの設計方法を学べるように書かれています。マルチスレッドプログラミングおよびネットワークプログラミングで起こる一般的な問題の解決方法や、分散アプリケーションの設計方法などを学ぶことが出来ます。たとえば、P2P アプリケーションや分散ハッシュテーブルなどの基盤を実装したいと考えている方にもおすすめの本です。

この本の原文は全 8 章で構成されており、現在 4 章までの翻訳が完了しています。

誤字・誤訳等ありましたら@hamano まで連絡下さい。校正を手伝ってくれた亀井亜佐夫さんに感謝します。

ØMQ とは

ØMQ(ZeroMQ, 0MQ, zmq などとも呼ばれます) は組み込みネットワークライブラリと言うことも出来ますが、並行フレームワークの様にも機能します。ØMQ はプロセス内通信、プロセス間通信、TCP やマルチキャストの様な幅広い通信手段を用いてアトミックにメッセージを転送する通信ソケットを提供します。ソケットをファンアウト、Pub-Sub、タスク分散、リクエスト・応答の様なパターンで N 対 N で接続できます。非同期 I/O モデルにより、マルチコア環境でスケーラブルな非同期メッセージ処理を行うアプリケーションを構築可能で、製品クラスタを構成する上で十分高速です。ØMQ は多くのプログラミング言語向けの API を持ち、ほとんどの OS で動作します。ØMQ は iMatix で開発され、LGPLv3 ライセンスで配布されています。

事の発端

我々は通常の TCP を手にしている。それにソビエトの秘密研究所から盗まれた放射性同位元素が注入され、1950 年代の宇宙線が放射された。それはひどい変装趣味の漫画作家に渡り、全身タイツをまとい、筋肉が盛り上がった。ええ、こうして ØMQ ソケットはネットワーキングの世界を守るスーパーヒーローになったのです。

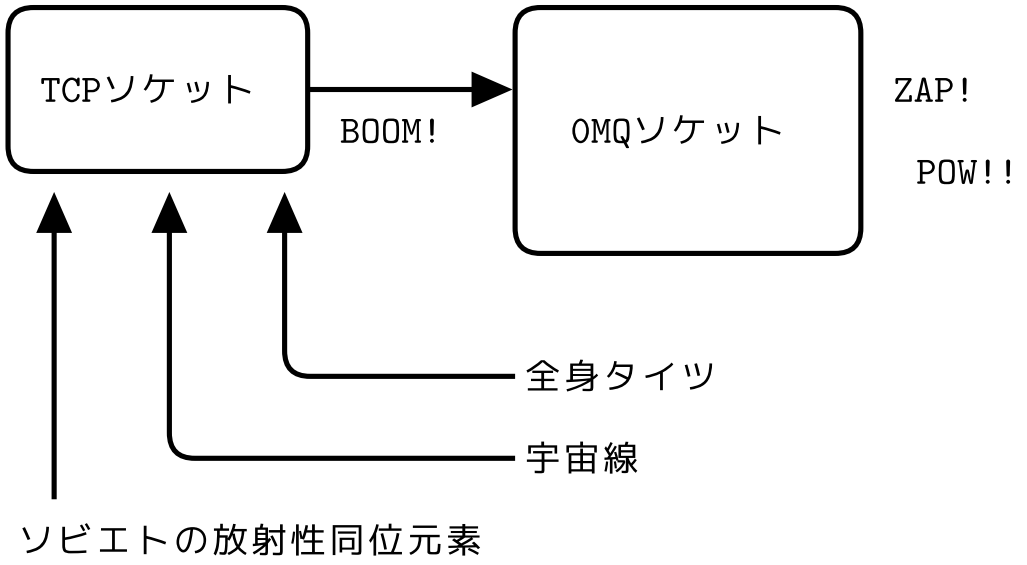


図 1　恐ろしいアクシデント

ゼロの哲学

ØMQ の Ø はトレードオフが本質です。まず、この奇妙な名前によって Google や Twitter の検索での可視性が低下しています。一方で、デンマークのひどい奴らが「ØMG(笑)」とか「Ø は変な形をしたゼロではない」とか「Rødgrød med Fløde!」(クリームがのったデンマークのおやつ) とか言ってくるのは明らかな侮辱であり、不愉快です。うん、どうやらこれはフェアなトレードだ。

元来、ØMQ のゼロは「仲介無し」や (出来るだけ)「遅延ゼロ」(に近づける) という意味を持っていました。以来それは「ゼロ管理」や「ゼロコスト」、「無駄がゼロ」といった別の目的を包含するようになりました。一般的な言葉で言うと、最小主義の文化がプロジェクトに浸透している事を示しています。私達は新たな機能を追加するというよりも、複雑さを取り除くことを重要視します。

対象読者

この本は、コンピューティングの未来を支配する大規模な分散ソフトウェアの作り方を学びたいプロのプログラマ向けに書かれています。ØMQ は多くのプログラミング言語で使えるにも関わらず、ほとんどのサンプルコードは C 言語で書かれているため、あなたが C 言語を読めることを想定しています。あなたがスケーラビリティの問題を気にしていることを想定しています。あなたが最小のコストで最良の結果を必要としている事を想定しています。そうしないと ØMQ のトレードオフについて認識できないからです。それ以外の ØMQ を使う上で必要なネットワークや分散コンピューティングなどの基本的な概念は出来るだけ説明するように心がけます。

謝辞

このテキストを書籍として出版する為に企画と編集を行なってくれた Andy Oram に感謝します。

以下の方々の貢献に感謝します:

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 Gallagher, 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

第 1 章

基礎

1.1 世界の修正

さてどうやって ØMQ を説明しましょうか。私達の中には、素晴らしい事柄を並べて説明を始める人もいます。それはソケットのステロイド化合物だ。それはメールボックスのルーティングの様だ。それは速い! その他には zap-pow-kaboom パラダイムシフトの悟りを開き、全てが明解になる瞬間を共有しようとする人もいます。物事は単純になります。複雑さは消え、心を開くのです...

もっと他の説明を試してみましょう。こちらは短くて単純ですがもっと馴染みやすいはずです。個人的に何故私達が ØMQ を作ったかという話を覚えておいて欲しいです。何故かというと、ほとんどの読者も同じ問題を抱えているはずだからです。

プログラミングは芸術としてドレスアップされた理学です。私達のほとんどはこれまで教わったことがないために、ソフトウェアの物理学を理解していません。ソフトウェアの物理学とはアルゴリズムやデータ構造、言語、抽象化などではありません。それらは唯の道具であり、作って使い捨てるものです。本当のソフトウェアの物理学とは人間の物理学です。具体的には、複雑性による私達の限界や巨大な問題を解決したいという欲求です。人々が簡単に理解して利用できるブロックを作り、協調して大きな問題を解決する事こそがプログラミングの理学です。

私達は接続された世界に住んでいて、現代のソフトウェアはこの世界を往来しなければなりません。ですから、ブロックは明日には巨大なシステムに接続され、大量に平行化されるかもしれないのです。コードは強く物静かであるだけでは不十分です。コードはコードと会話し、社交的なおしゃべりでなくてはなりません。コードは人間の脳にある何兆もの独立したニュー

ロンの様にお互いにメッセージを発し、中央制御が必要なく、単一障害点の存在しない超並列ネットワークを構成して動作しなければなりません。そうしてやっと、困難な問題を解決できるようになります。この様なコードの未来が人間の脳と似ていることは偶然ではありません。ネットワークのエンドポイントは幾つかのレベルで人間の脳と同じだからです。

この様なシステムをスレッドやプロトコル、ネットワークを 1 から構築して実装しようとした場合、到底不可能であることに気がつくでしょう。それは夢物語です。複数のプログラムが複数のソケットを利用して接続するプログラムは現実的には地味に厄介です。想像を絶するコストが掛かります。数十億ドル規模のビジネスでコンピューターを接続するソフトウェアやサービスを行うことは非常に困難になります。

私達は自分たちの扱える能力に見合った世界で生活しています。1980 年代にソフトウェア業界の重大局面がありました。フレデリック・ブルックスのような有名ソフトウェア・エンジニアが生産性、信頼性、単純性を大幅に改善する「銀の弾など存在しない」と断言した事です。

ブルックスはオープンソースソフトウェアが効果的に知識を共有するする事を可能にして重大局面を解決することを見落としていました。そして現在、ソフトウェア業界は別の重大局面に直面していますがこれについて話したがる人はあまり居ません。最も巨大で裕福な企業のみが、接続するアプリケーションを開発する余裕があります。それはプロプライエタリなクラウドです。我々のデータと知識はパーソナルコンピューターの中からクラウドの中に消えてしまい、我々自身もアクセス出来なくなっています。ソーシャルネットワークを自分自身で所有している人はいますか? これではまるでメインフレーム-パーソナルコンピューター革命を逆行しているようです。

政治哲学的な話はこの辺にしておいて他の本に譲る事にしますが、重要なのはインターネットは潜在的に大量のコードが接続しあうにも関わらず、現実では私達の手の届かない所にあるという事です。そしてこれは健康、教育、経済、流通などにおいて非常に興味深い問題を引き起こしますが、コードを接続する方法が無いので未だ解決出来ていません。したがって、これらの問題を解決するには脳を接続出来る相手と一緒に仕事するしかありません。

コードを接続するという問題に関して多くの試みが行われてきました。何千もの IETF の仕様はこれらの問題を解決するパズルの一部です。HTTP は恐らくアプリケーション開発者にとって単純明解な解決方法の一つでしょう。しかしそれは楽観的な開発者や設計者による、巨大なサーバーと貧弱なクライアントを前提とした考えであり、問題を悪化させるでしょう。

そして現在でも人々は UDP や TCP、プロプライエタリなプロトコル、HTTP、Web ソケットを使用してアプリケーションを接続しています。それは痛みを伴うほど遅く、拡張が難しく、本質的に中央集中型です。分散 P2P はほとんど娯楽のためであり、ビジネスで使うには難しいでしょう。Skype や Bittorrent とデータ通信を行うアプリケーションがどれほどあるでしょうか?

プログラミングの理学の話に立ち帰ると、世界を修正するために我々は 2 つの事を行う必要

があります。1 つ目は「何処でもコードとコードを接続出来るようにする方法」という一般的な問題を解決すること。2 つ目は人々が簡単に理解して利用できる単純なブロックでそれを包み込む事です。

それは馬鹿馬鹿しいほど単純に聞こえるし、多分きっとそうなんでしょう。しかしこれはとても肝心な事です。

1.2 前提条件

あなたが最新の ØMQ バージョン 3.2 を利用している事を想定しています。また、あなたが Linux マシンまたは類似の何かを利用していることを想定します。サンプルコードの既定の言語は C 言語ですので、あなたが多かれ少なかれ C 言語が読めることを想定しています。私が PUSH や SUBSCRIBE といった定数を書いた時、実際には ZMQ_PUSH や ZMQ_SUBSCRIBE という様に各プログラミング言語で使われる記述に読み替えて読んでください。

1.3 サンプルコードの取得

サンプルコードは GitHub の公開レポジトリから取得できます。全てのサンプルコードを取得する最も簡単な方法はレポジトリを clone することです。

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

続いて examples サブディレクトリを参照すると、プログラミング言語毎のディレクトリ見つけるでしょう。もしあなたのお気に入りのプログラミング言語が無い場合はコードを移植して送って頂ください。この様に多くの人々の協力でこのテキストは便利になりました。全てのサンプルコードは MIT/X11 ライセンスで公開されています。

1.4 尋ねよ、さらば受け取らん

さあ、コードから始めましょう。 もちろん最初は Hello World のサンプルコードから始めます。クライアントが「Hello」をサーバーに送信したら、サーバーは「World」を応答するクライアントとサーバーを作ってみましょう。ここでサーバーは ØMQ ソケットを TCP ポート 5555 番で待ち受け、リクエストを受け取ったら「World」を応答するコードを C 言語で実装しています:

hwserver.c: Hello World サーバー

```c
//  Hello World サーバー

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

int main (void)
{
    // クライアントと通信を行うソケット
    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); // 何らかの処理
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
```

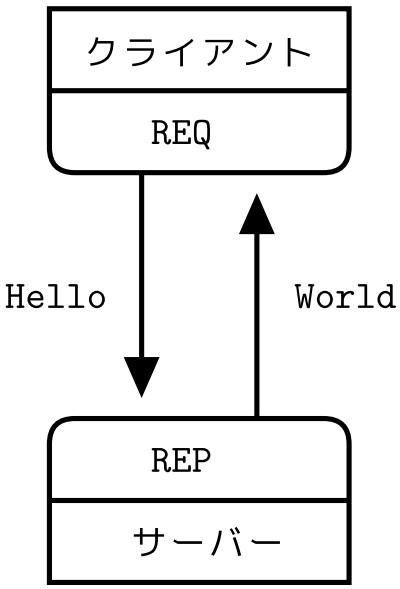


図 1.1　リクエストと応答

REQ-REP ソケットペアはロックステップ方式です。クライアントはループ内で zmq_send() を呼んでから zmq_recv() を発行します。それ以外のケース、例えば 2 回メッセージを送信した場合などでは zmq_send() や zmq_recv() で-1 が返ります。同様にサーバー側は zmq_recv() を呼んでから zmq_send() を発行する必要があります。

ØMQ はリファレンス言語として C 言語を採用しているので、サンプルコードでも C 言語を使います。ここでは C++ のコードを見て比べてみましょう。

hwserver.cpp: Hello World サーバー

```
//
// Hello World サーバー (C++ 版)
// REP ソケットを tcp://*:5555 でバインドします。
// クライアントが"Hello"を送信してきた時、"World"と応答します。
//
#include <zmq.hpp>
#include <string>
#include <iostream>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>
#endif

int main () {
    // コンテキストとソケットの準備
    zmq::context_t context (1);
    zmq::socket_t socket (context, ZMQ_REP);
    socket.bind ("tcp://*:5555");

    while (true) {
        zmq::message_t request;

        // クライアントからのリクエストを待機
        socket.recv (&request);
        std::cout << "Received Hello" << std::endl;

        // 何らかの処理
#ifndef _WIN32
    sleep(1);
#else
    Sleep (1);
#endif
```

```
        // クライアントに応答
        zmq::message_t reply (5);
        memcpy ((void *) reply.data (), "World", 5);
        socket.send (reply);
    }
    return 0;
}
```

ØMQ の API は C 言語と C++ でほとんど同じだという事が解ると思います。PHP と Java の例も見てみましょう。

hwserver.php: Hello World サーバー

```
<?php
/*
* Hello World サーバー (PHP)
* REP ソケットを tcp://*:5555 でバインドします。
* クライアントが"Hello"を送信してきた時、"World"と応答します。
* @author Ian Barber <ian(dot)barber(at)gmail(dot)com>
*/

$context = new ZMQContext(1);

// クライアントとの通信ソケット
$responder = new ZMQSocket($context, ZMQ::SOCKET_REP);
$responder->bind("tcp://*:5555");

while (true) {
    // クライアントカアラのリクエストを待機
    $request = $responder->recv();
    printf ("Received request: [%s]\n", $request);

    // 何らかの処理
    sleep (1);

    // クライアントに応答
    $responder->send("World");
}
```

hwserver.java: Hello World サーバー

```
//
// Hello World サーバー (Java)
```

```
// REP ソケットを tcp://*:5555 でバインドします。
// クライアントが"Hello"を送信してきた時、"World"と応答します。
//

import org.zeromq.ZMQ;

public class hwserver{

    public static void main (String[] args) throws Exception{
        ZMQ.Context context = ZMQ.context(1);
        // クライアントとの通信ソケット
        ZMQ.Socket socket = context.socket(ZMQ.REP);
        socket.bind ("tcp://*:5555");

        while (!Thread.currentThread ().isInterrupted ()) {
            byte[] reply = socket.recv(0);
            System.out.println("Received Hello");
            Thread.sleep(1000); // 何らかの処理
            String request = "World" ;
            socket.send(request.getBytes (), 0);
        }
        socket.close();
        context.term();
    }
}
```

以下はクライアントのコードです。

hwclient.c: Hello World クライアント

```
//  Hello World クライアント
#include <zmq.h>
#include <string.h>
#include <stdio.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_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;
}
```

さて、この例は現実的にあまりにも単純に見えますが、これまで学んできたように ØMQ ソケットはとんでもない力を秘めています。あなたは同時に数千のクライアントでこのサーバーに接続することができ、問題なく迅速に動作し続けるでしょう。戯れにサーバーを立ち上げてクライアントを実行し、どんな風に動作するか試してみてください。そしてこの意味を少し考えてみて下さい。

これら 2 つのプログラムが実際に何をしているか簡潔に説明しましょう。これらはまず ØMQ コンテキストと ØMQ ソケットを作成します。言葉の意味については後で説明しますのでまだ心配しなくて大丈夫です。サーバーは REP(応答) ソケットをポート 5555 番で bind します。サーバーはループの中でリクエストを待ち、リクエスト毎に応答します。クライアントはリクエストを送信し、サーバーからの応答を受け取ります。

サーバーを Ctrl-C で終了して再起動した場合、クライアントは適切に復旧しません。プロセスの異常終了から復旧することは簡単ではありません。信頼性の高いリクエスト-応答フローを構成することは十分複雑なので、これについては 4 章の「信頼性のあるリクエスト・応答パターン」で説明します。

我々プログラマにとってどんなに短く素敵なコードでも、裏側ではたくさんの事が起こっています。そしてそのおかげでどれだけ負荷を掛けてもクラッシュしません。これをリクエスト・応答パターンと呼びます。恐らく、ØMQ の最も単純な利用方法です。これは RPC とか、古典的なクライアント・サーバーモデルに対応します。

1.5 文字列に関する補足

ØMQ はデータについてサイズ以外の事は何も知りません。これは、プログラマがアプリケーション側で安全に読み戻せるようにする責任があるという事を意味します。オブジェクトや複雑なデータ構造を利用する事は Protocol Buffers の様なライブラリの役目です。文字列でさえ気を配ってやる必要があります。

C 言語や幾つかの言語では、文字列は NULL 文字で終端してます。“HELLO” という様な文字列を送信する際、以下の様に NULL 文字付きで送信出来ます。

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

しかしながらその他の言語では NULL 文字を含まない場合があります。例えば Python では、以下のようにして文字列を送信します。

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

この時、文字列の長さと文字列の内容がネットワーク上を流れます。

図 1.2　ØMQ 文字列

そしてもし C 言語のプログラムでこれを読むと、あなたは偶然文字列の様なものを受け取るでしょうが、これは正しい文字列ではありません。クライアントとサーバーで文字列フォーマットに関する合意がない場合、おかしな結果が得られるかもしれません。

C 言語で文字列を受信する際、文字列が安全に NULL 終端していると期待してはいけません。文字列を読み込む際には、新たに大きめの新しくバッファを確保し、コピーして適切に NULL 文字で終端させてやる必要があります。

それでは、NULL 終端していない ØMQ 文字列が送られてきた場合のルールを確立しましょう。最も単純なケースでは、先の図の様に ØMQ 文字列の長さと内容は ØMQ メッセージフレームにぴったり一致します。

以下のコードは、C 言語で受け取った ØMQ 文字列を適切な文字列としてアプリケーションに受け渡す為に何を行う必要があるのかを示しています。

```
// ソケットから 0MQ 文字列を受信して C 文字列に変換する
// 255 文字より長い文字列は打ち切る
static char *
s_recv (void *socket) {
    char buffer [256];
    int size = zmq_recv (socket, buffer, 255, 0);
    if (size == -1)
        return NULL;
    if (size > 255)
        size = 255;
```

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

モノ作り精神で作成したこの便利なヘルパー関数は有効に再利用することが出来ます。同様に、正しい ØMQ フォーマット文字列を送信する s_send 関数も書いてみましょう。そして再利用できるヘッダーファイルをパッケージングします。

その成果が zhelpers.h であり、これによって短く簡潔に ØMQ アプリケーションを書くことが出来ます。このソースコードは相当長いですが、興味がある C 開発者の方は余裕がある時に読んでみて下さい。

1.6 バージョン報告

ØMQ には幾つかのバージョンがあり、頻繁にバージョンアップします。もし問題に遭遇したとしても最新のバージョンで修正されていることが多いでしょう。ですから ØMQ のバージョンを正確に調べる方法を知っておくと役に立ちます。

以下はそれを行う小さなプログラムです:

version.c: ØMQ のバージョン報告

```
// 0MQ のバージョン報告

#include <zmq.h>

int main (void)
{
    int major, minor, patch;
    zmq_version (&major, &minor, &patch);
    printf ("Current 0MQ version is %d.%d.%d\n", major, minor, patch);
    return 0;
}
```

1.7 メッセージ配信

第二の典型的なパターンは、サーバーから複数のクライアントに更新をプッシュする一方方向のデータ配信です。それでは、郵便番号と気温、湿度からなる気象情報をプッシュ配信する例を見てみましょう。ここで利用する気象情報はランダムに生成した値を利用することにし

ます。

以下がサーバーのサンプルコードです。このアプリケーションは TCP 5556 番ポートを利用します。

wuserver.c: 気象情報更新サーバー

```
// 気象情報更新サーバー
// PUB ソケットを tcp://*:5556 でバインドし、
// ランダムな気象情報を配信します

#include "zhelpers.h"

int main (void)
{
    // コンテキストとパブリッシャーの準備
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    int rc = zmq_bind (publisher, "tcp://*:5556");
    assert (rc == 0);

    // 乱数生成器の初期化
    srandom ((unsigned) time (NULL));
    while (1) {
        // インチキな気象データを取得
        int zipcode, temperature, relhumidity;
        zipcode     = randof (100000);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;

        // 全てのサブスクライバーにメッセージを送信
        char update [20];
        sprintf (update, "%05d %d %d", zipcode, temperature, relhumidity);
        s_send (publisher, update);
    }
    zmq_close (publisher);
    zmq_ctx_destroy (context);
    return 0;
}
```

終わりの無い放送の様に、このストリームの配信に始まりと終わりはありません。

以下のクライアントアプリケーションはストリームの配信を聞き取り、特定の郵便番号に関するデータを収集します。デフォルトではニューヨークを指定しています。なぜならそこは冒険を始めるには絶好の場所だからです。

wuclient.c: 気象情報更新クライアント

```
// 気象情報更新クライアント
// SUB ソケットで tcp://localhost:5556 に接続して気象情報を収集し、
// 引き数に指定した郵便番号の平均値を求める

#include "zhelpers.h"

int main (int argc, char *argv [])
{
    // サーバーと通信するソケット
    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);

    // 購読する郵便番号、デフォルトで 10001(ニューヨーク)
    char *filter = (argc > 1)? argv [1]: "10001 ";
    rc = zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE,
                         filter, strlen (filter));
    assert (rc == 0);

    // 100 個分処理する
    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;
}
```

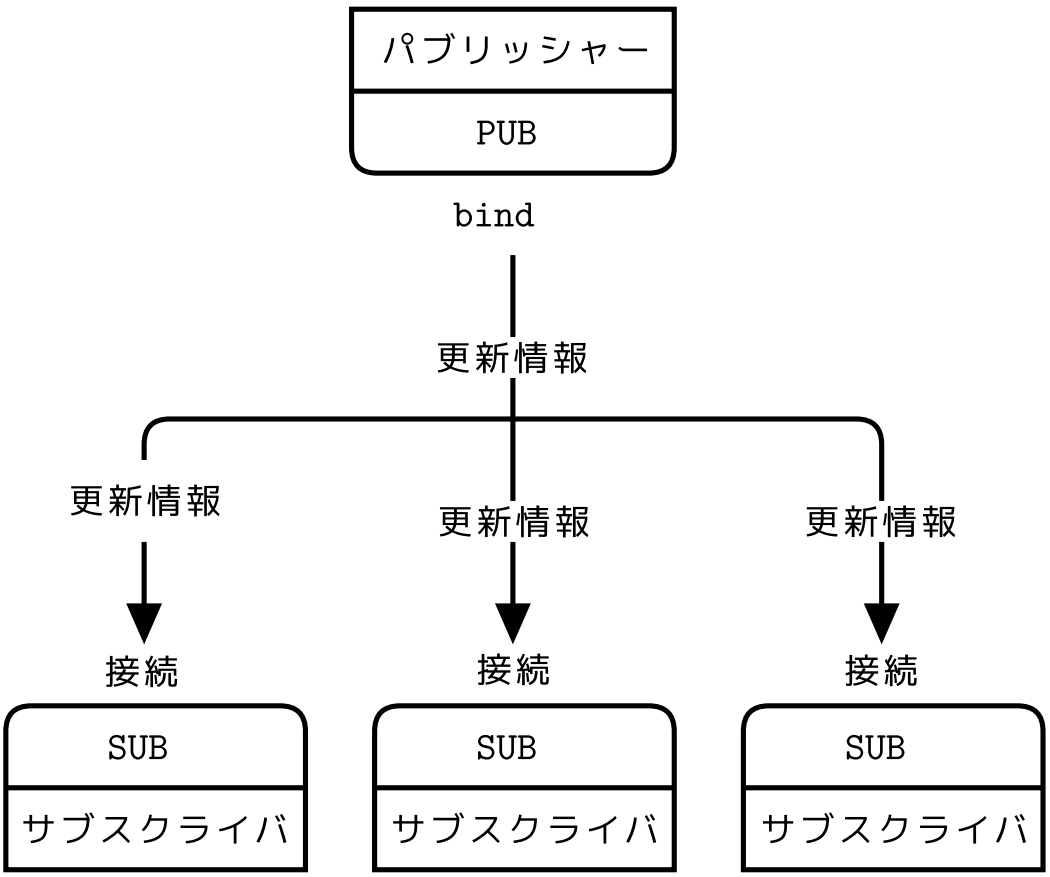


図 1.3　パブリッシュ・サブスクライブ

SUB ソケットを利用する際、このコードの様に zmq_setsockopt() で SUBSCRIBE を設定しなければならないことに注意して下さい。もし設定しなかった場合メッセージを受信できません。これはよくある初歩的なミスです。サブスクライバーは複数のサブスクリプションを設定できます。その際サブスクリプションに一致した更新のみ受信します。サブスクライバーは特定のサブスクリプションをキャンセルすることも出来ます。サブスクリプションは必ずしも印字可能な文字とは限りません。これがどの様に動作するかは zmq_setsockopt() のソースコードを読んで下さい。

PUB-SUB ソケットのペアは非同期で動作し、通常クライアントはループ内で zmq_recv() を呼び出します。SUB ソケットでメッセージを送信しようとするとエラーが発生します。同様に、PUB ソケットで zmq_recv() を呼んではいけません。

理論上はどちらが bind してどちらが接続しても問題ないはずです。しかし今の所ドキュメント化されていないので出来れば PUB で bind して SUB で接続して下さい。

PUB-SUB ソケットについて知るべき重要なことがもうひとつあります。それは、サブスクライバーがいつメッセージを受信し始めたかどうかを正確に知ることは出来ないという事です。サブスクライバーを起動し、しばらく経ってパブリッシャーを起動した場合でも

必ず最初のメッセージを取りこぼします。これは、サブスクライバがパブリッシャーに接続している間 (一瞬だがゼロでは無い時間) に、パブリッシャーがメッセージを配信している可能性が在るからです。

多くの人がこの「参加遅延症状」に遭遇するので私達はこれについての説明を頻繁に行います。ØMQ が非同期 I/O であることを思い出して下さい。2 ノードでこれを行う際、バックグラウンドでは以下の事を以下の順序で行います。

- サブスクライバはエンドポイントに接続し、メッセージを受信して数える。
- パブリッシャーはエンドポイントを bind し、即座に 1000 メッセージを送信する。

恐らくサブスクライバーは何も受信していないでしょう。フィルタが正しく設定されているか確認し、再度試してみて下さい。まだ何も受信出来ていないはずです。

TCP コネクションの作成およびハンドシェイクはネットワークやピア間のホップ数に応じて数ミリ秒の遅延を発生させます。ØMQ はこの間に多くのメッセージを送信できます。便宜上、コネクションの確立に 5 ミリ秒かかり、1 秒間に 100 万メッセージを処理できると仮定すると、パブリッシャーはサブスクライバが接続しているわずか 5 ミリ秒の間に、5000 メッセージを送信出来ることになります。

「2 章ソケットとパターン」ではパブリッシャーとサブスクライバを同期してサブスクライバの準備が整うまでパブリッシャーがデータを配信しないようにする方法を説明します。単純に sleep を入れて遅延させるという愚直な方法もありますが、実用のアプリケーションでこれをやると極めて不安定な上に遅いのでやらないで下さい。正しくこれをやる方法と、sleep を行うと何が起こるかは「2 章ソケットとパターン」まで待って下さい。

同期を行わない場合、サーバーは無限にデータを配信することを前提とし、サブスクライバは開始時に始まりと終わりを扱いません。これは天気クライアントの例で見てきた通りです。

まとめると、クライアントは指定した郵便番号の更新を 100 個収集します。郵便番号がランダムに分布している場合には、約 1 千万の更新が送られてくることになります。クライアントを開始した後に、サーバを起動してもクライアントは問題なく動作します。サーバーを好きなタイミングで再起動しても、クライアントは動作し続けます。クライアントが 100 の更新を収集すると、平均値を計算し、表示して終了します。

パブリッシュ・サブスクライブ (PUB-SUB) パターンの要点は以下の通りです。

- サブスクライバーは一つ以上のパブリッシャーに接続することが出来ます。一つのパブリッシャーが大量のメッセージを流して専有してしまわないように、到着したデータは制御されています。これを「平衡キューイング」と呼びます。
- パブリッシャーに接続しているサブスクライバーが居ない時、全てのメッセージは単純に破棄されます。

- TCP を利用していてサブスクライバが遅い場合、メッセージはパブリッシャーのキューに入れられます。「HWM(満杯マーク)」を利用してパブリッシャーを保護する方法については後で説明します。
- ØMQ v3.x 以降、ステートフルプロトコル (tcp: もしくは ipc:) を利用している場合にパブリッシャー側でフィルタリング出来るようになりました。epgm:// プロトコルを利用する場合は、サブスクライバ側でフィルタリングします。ØMQ v2.x では全てのフィルタリングはサブスクライバ側で行います。

これは、2011 年に買った Intel i5 の普通のノート PC で 1 千万のメッセージを受信してフィルタリングするのに掛かった時間です。

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

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

1.8 分割統治法

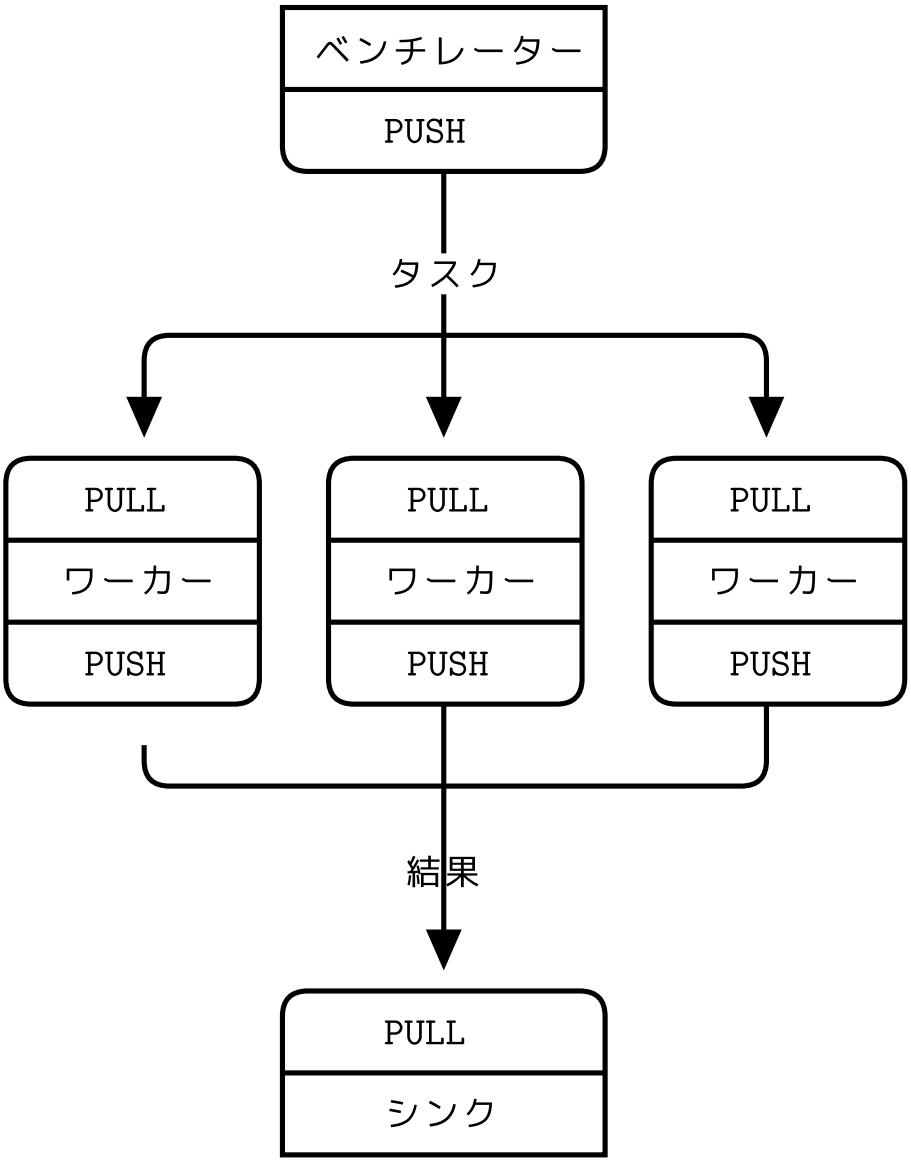


図 1.4　並行パイプライン

最後の例は小さなスパコンを作って計算してみましょう。そして沢山のコードばかりを見てきて疲れたでしょうからコーヒーでも飲んで休憩してください。スパコンのアプリケーションは典型的な並行分散処理モデルです。

- 「ベンチレーター」は並行に処理できるタスクを生成します。
- 「ワーカー」群はタスクを処理します。
- 「シンク」は「ワーカー」の処理結果を収集します。

実践ではワーカーは GPU などを搭載した高速マシンで実行されます。ベンチレーターは 100 のタスクを生成しワーカーに送信します。ワーカーは受け取った数値×ミリ秒の sleep を行います。

taskvent.c: 並行タスクベンチレーター

```
// タスクベンチレーター
// PUSH ソケット tcp://localhost:5557 をバインドし、
// ソケットを経由してワーカーに処理タスクを送信する

#include "zhelpers.h"

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

    // メッセージを送信するソケット
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_bind (sender, "tcp://*:5557");

    // シンクに処理の開始を通知するソケット
    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");

    // 処理の開始を表す「0」というメッセージを送信
    s_send (sink, "0");

    // 乱数生成器の初期化
    srandom ((unsigned) time (NULL));

    // 100 個のタスクを送信
    int task_nbr;
    int total_msec = 0;     //  Total expected cost in msecs
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        int workload;
        // 1〜100 ミリ秒かかるランダムな処理
        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;
}
```

以下はワーカーアプリケーションです。受信したメッセージの秒数分 sleep し、完了を通知します。

taskwork.c 並行タスクワーカー

```
// タスクワーカー
// PULL ソケットで tcp://localhost:5557 に接続し、
// ベンチレーターから仕事を貰う
// PUSH ソケットで tcp://localhost:5558 に接続し、
// シンクに対して処理結果を送信

#include "zhelpers.h"

int main (void)
{
    // メッセージ受信用ソケット
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    // メッセージ送信用ソケット
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sender, "tcp://localhost:5558");

    // タスクを処理し続ける
    while (1) {
        char *string = s_recv (receiver);
        printf ("%s.", string);      // 進行状況を表示
        fflush (stdout);
        s_sleep (atoi (string));     // なんらかの処理
        free (string);
        s_send (sender, "");         // シンクに処理結果を送信
    }
    zmq_close (receiver);
```

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

以下はシンクアプリケーションです。100 のタスクを収集し、処理にどれくらいの時間が掛かったかを計算します。この結果により、本当に並行処理が行われたどうかを確認できます。

tasksink.c: 並行タスクシンク

```
// シンクタスク
// PULL ソケットで tcp://localhost:5558 を bind し、
// ソケット経由で処理結果を収集

#include "zhelpers.h"

int main (void)
{
    // コンテキストとソケットの準備
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    // 処理が開始されるまで待機
    char *string = s_recv (receiver);
    free (string);

    // 処理時間の計測を開始
    int64_t start_time = s_clock ();

    // 100 個の処理結果を確認
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if ((task_nbr / 10) * 10 == task_nbr)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    // 処理時間を計測して表示
    printf ("Total elapsed time: %d msec\n",
        (int) (s_clock () - start_time));
```

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

平均的な実行時間は大体 5 秒程度です。 ワーカーを 1, 2, 4 個と増やした時の結果は以下の通りです。

- 1 ワーカー: total elapsed time: 5034 msecs.
- 2 ワーカー: total elapsed time: 2421 msecs.
- 4 ワーカー: total elapsed time: 1018 msecs.

それでは、もっと詳しくコードの特徴を見ていきましょう。

- ワーカーは上流のベンチレーターと下流のシンクに接続します。これは自由にワーカーを追加できる機能を持っているという事を意味しています。もしワーカーが bind を行ったとすると、ワーカーを追加する度にベンチレーターとシンクの動作を変更しなければなりません。ベンチレーターとシンクがアーキテクチャの固定部品であり、ワーカーは動的な部品であると言えます。
- 全てのワーカーが起動するまで、処理の開始を同期させる必要があります。これは ØMQ のよくある落とし穴であり簡単な解決方法はありません。`zmq_connect()` 関数はどうしてもある程度の時間がかかってしまいます。複数のワーカーがベンチレーターに接続する際、最初のワーカーが正常に接続してメッセージを受信しても、他のワーカーはまだ接続中の状態になります。何らかの方法で、処理の開始を同期しなければシステムは並行に動作しません。試しに getchar による一時停止を削除して、何が起こるか確認してみましょう。
- ベンチレーターの PUSH ソケットはタスクを均等にワーカーに分散します (処理が開始されるまでに全てのワーカーは接続済みであると仮定します)。これはロードバランシングと呼ばれ、詳細は後ほど改めて説明します。
- シンクの PULL ソケットはワーカーからの処理結果を均等に収集します。これは平衡キューイングと呼びます。

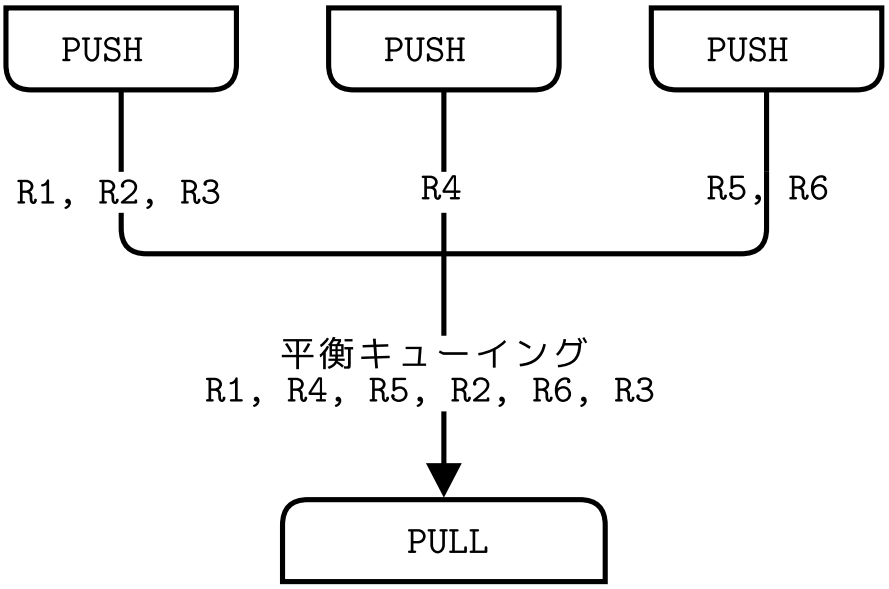


図 1.5 平衡キューイング

この様なパターンで「参加遅延病」が発症した場合、PUSH ソケットが適切にロードバランスしなくなる現象を引き起こします。PUSH と PULL を利用している場合、あるワーカーが他のワーカーより多くのメッセージを受け取ることになります。なぜならばある PULL ソケットは早く接続していて、その他のソケットが接続している間に多くのメッセージを受け取るからです。もし正確なロードバランシングを行いたい場合は「第 3 章- Advanced Request-Reply Patterns」を参照して下さい。

1.9 ØMQ プログラミング

幾つかのサンプルコードを見てきました。あなたは ØMQ でなにかアプリケーションを作りたくて仕方が無いのでしょう。それを始める前に、大きく深呼吸をして落ち着き、ストレスと混乱を避けるために幾つかの基本的なアドバイスに耳を傾けて下さい。

- 一歩ずつ ØMQ を学んで下さい。これはとてもシンプルな API ですが、あらゆる可能性が潜んでいます。起こりうる可能性を一つずつ学んでいってください。
- 素敵なコードを書いて下さい。醜いコードは問題を隠蔽し、他の人があなたを助けることを困難にします。変数名に無意味な名前を利用すると誰もあなたのコードを読めなくなるでしょう。変数の意味を伝えるのに適切な現実の世界の言葉を使って下さい。一貫したインデントと綺麗なレイアウトを使って下さい。素敵なコードを書くとあなたの世界はより快適になります。
- あなたが作ったものをテストして下さい。プログラムが動作しない時は何処に原因があ

るか特定する必要があります。ØMQ を初めて使い始めたばかりで上手く動作しない時は特に十分テストして下さい。

- 上手く動作しない所を見つけた時、個別にテストして切り分けを行って下さい。ØMQ は基本的なモジュールコードを作成できます。これはあなたの助けになるでしょう。
- 必要に応じてコードを上手く抽象化して下さい。同じコードをコピー&ペーストばかりしていたら、エラー箇所も増えてゆきます。

1.9.1 正しくコンテキストを取得する

ØMQ アプリケーションは常にコンテキストを作成し、それを利用してソケットを作成します。C 言語では zmq_ctx_new() を呼び出します。プロセス内に一つのコンテキストを作成し、それを利用します。技術的に言うと、コンテキストは単一プロセス内で全てのソケットをまとめるコンテナであり、プロセス内で高速にスレッド間を接続するプロセス内ソケットとして振る舞います。もし、1 つの実行プロセスが 2 つのコンテキスト持つと、それは ØMQ インスタンスが 2 に分離しているように見えます。あえてこうしたいのであれば問題ありませんが、そうでないのなら注意して下さい。

メインコードの最初で zmq_ctx_new() 呼び出して、終わりに zmq_ctx_destroy() を呼び出して下さい。

fork() システムコールを利用している場合、各プロセスは独自のコンテキストを必要とします。メインプロセスで zmq_ctx_new() を呼び出した後に fork() した場合、子プロセスは独自のコンテキストを得ます。一般的に、主な処理は子プロセスで行い、親プロセスは子プロセスを管理するだけでしょう。

1.9.2 正しく終了する

一流のプログラマは一流の殺し屋と同じ教訓を共有します。「仕事が終わったら後片付けしろ」という事です。ØMQ を Python の様な言語で利用している場合、オブジェクトは自動的に開放されます。しかし、C 言語の場合は慎重にオブジェクトを開放する必要があります。そうしなければメモリリークが発生したり、アプリケーションが不安定になったり、天罰が下ったりします。

メモリリークもその一つです。ØMQ はアプリケーションを終了することに関してとても気難しいです。その理由は、技術的かつ痛みを伴いますが、もしソケットをオープンしたまま zmq_ctx_destroy() 関数を呼び出した場合、永久にハングします。そしてもし、LINGER を 0 に設定せずに全てのソケットクローズした場合でも、zmq_ctx_destroy() で待たされるでしょう。

ØMQ で気配りする必要があるオブジェクトはメッセージとソケットとコンテキストの 3 つです。幸いなことに、単純なプログラムでこれらを扱うのはとても簡単です。

- 可能な限り zmq_send() と zmq_recv() を使って下さい。これらは zmq_msg_t オブジェクトの利用を避けることが出来ます。
- zmq_msg_recv() を使う場合、メッセージを受信したら zmq_msg_close() を呼ぶ前に出来るだけ早く開放して下さい。
- 多くのソケットをオープンしてクローズする場合、アプリケーションを再設計する必要性がある兆候です。幾つかのケースでは、コンテキストを開放するまでソケットが開放されなくなります。
- プログラムを終了する際、ソケットを閉じてから zmq_ctx_destroy() を呼んで下さい。これはコンテキストを破棄する関数です。

最後のケースは C 言語で開発する場合です。多くの言語では、スコープが外れた時にソケットやコンテキストなどのオブジェクトは自動的に開放されます。もし例外を利用する場合は「final」ブロックでこれらのリソースを開放すると良いでしょう。

マルチスレッドを利用している場合、これはもっと複雑になります。マルチスレッドに関しては次の章で扱いますが、警告を無視して試してみたい人もいるでしょう。以下は、マルチスレッドの ØMQ アプリケーションで正しく終了するための急しのぎのガイドです。

まず、複数のスレッドから同一のソケットを扱わないで下さい。冗談ではありません、やらないで下さい。次に、リクエスト中のソケットを接続を切る時は LINGER に小さい値 (1 秒程度) を設定し、それから接続を閉じて下さい。もしあなたの利用している言語バインディングがこれを行わない場合、修正してパッチを送ることを推奨します。

最後に、コンテキストを開放します。これを行うと、コンテキストを共有して送受信を行っている別のスレッドでエラーが返ります。エラーを拾い、LINGER を設定してソケットをクローズして下さい。同じコンテキストを 2 回開放しないで下さい。zmq_ctx_destroy() は全てのソケットが安全に閉じられるまでメインスレッドでブロックします。

おしまい!これはとても複雑で痛みを伴いますが、有能な言語バインディングの作者が自動的にソケットを閉じてくれるので必ずしもこれを行う必要はないでしょう。

1.10 なぜ ØMQ が必要なのか

これまで ØMQ の動作について見てきましたが、前に戻って「何故」の話に戻りましょう。

今日多くのアプリケーションは LAN やインターネットなどのネットワークを横断する機能を有しています。そして多くのアプリケーション開発者は最終的メッセージング機能を必要と

します。開発者の中にはメッセージキュー製品を利用する人もいますが、ほとんどの人は TCP や UDP を利用して自前で実装します。これらのプロトコルを利用するのは難しいことではありませんが、単に A から B へメッセージを送信する事と、信頼性のある方法でこれを行うのとでは大きな違いあがあります。

それでは生の TCP を利用して部品を接続する際に発生する典型的な問題を見て行きましょう。再利用可能なメッセージングレイヤを実装するにはこれらの問題を解決する必要があります。

- I/O 処理をどの様に行うか。ブロッキング I/O か非同期 I/O のどっちにする?これは重要な仕様判断です。ブロッキング I/O を選択するとスケーラビリティの無いアーキテクチャになります。一方、非同期 I/O を正しく実装するのはとても難しいです。
- 動的なコンポーネントをどの様に処理するか。例えば部品が一時的に停止した時どうしますか? 一般的にコンポーネントは「サーバー」と「クライアント」に別れていることが多いですが、サーバーが落ちてしまった時どうしますか? サーバーとサーバーが接続するような場合は? 数秒毎に再接続するようにしますか?
- メッセージをネットワーク上でどの様に表現するか。どの様にしてデータフレームを簡単に読み書きしたり、バッファーオーバーフローが起きないようにしたり、小さいメッセージを効果的に転送したり、パーティ用帽子をかぶった猫が踊っている巨大な動画を見ますか?
- 即座に配信できないメッセージをどの様に処理するか。例えばコンポーネントが一時的にオフラインである場合、メッセージを破棄しますか? データベースに入れておきますか? それともメモリーキューに入れておきますか?
- メッセージキューを何処に格納するか。キューが増えてきて読み込みが遅くなった時はどうしますか? その時の戦略は?
- 欠落したデータをどの様に処理するか。新しいデータを待ちますか? リクエストを再送しますか? 信頼性のあるレイヤでネットワークを構築すればメッセージは欠落しないって? そのレイヤ自体がクラッシュしたらどうするの?
- 複数のネットワークに配送する場合はどうする? TCP ユニキャストの代わりにマルチキャストとか IPv6 を使う? アプリケーションを書きなおしますか? ネットワークレイヤを抽象化しますか?
- どの様にメッセージをルーティングする? 同じメッセージを複数の相手に送れる? 元のリクエスト送信者に返信出来る?
- どうやって API をいろんな言語で実装する? ネットワークレベルのプロトコルを再実装する? ライブラリを再パッケージする? 前者ならどうやって安定したスタックを保証しますか? 後者ならどうやって相互運用性を保証しますか?

- 異なるアーキテクチャでどの様にデータを表現しますか? 特定のデータエンコーディングに統一しますか? 何処までがメッセージングシステムの仕事で何処からが上位アプリケーションレイヤの仕事でしょうか?
- ネットワークエラーをどの様に処理しますか? リトライしますか? 静かに無視しますか? 処理を中断しますか?

2013 年 1 月頃、典型的オープンソースプロジェクトである Hadoop Zookeeper の C API コード (src/c/src/zookeeper.c) を読んでみると、4,200 行のコードは謎めいていて、クライアント/サーバーの通信プロトコルはドキュメント化されていませんでした。それは select ではなく効率的な poll を利用している事が確認できました。Zookeeper はもっと一般的なメッセージングレイヤを利用し、ドキュメント化されたネットワークプロトコルを使ったほうが良いでしょうが、それはチームにとって車輪の再発明を繰り返す事になりとてつもなく無駄です。

しかしどうやって再利用可能なメッセージングレイヤを作るのでしょうか? 多くのプロジェクトでこの技術が必要とされているにも関わらず、何故人々は未だに TCP ソケットを直に触って先ほど挙げた問題を解決するために繰り返し苦労しているのでしょうか。

再利用可能なメッセージングシステムを作るのが本当に難しいということは、これを行う FOSS プロジェクトが少ないことや、商用メッセージング製品が複雑、高価で柔軟性が無く、不安定であることからも分ります。2006 年に iMatix 社は AMQP という再利用可能なメッセージングシステムを恐らく最初に FOSS 開発者に提供しました。AMQP はその他の設計より上手く動作していましたが比較的複雑で高価で不安定でした。数週間掛けて使い方を学び、数ヶ月掛けて安定したアーキテクチャを作り上げた結果、恐ろしいクラッシュが発生しなくなりました。

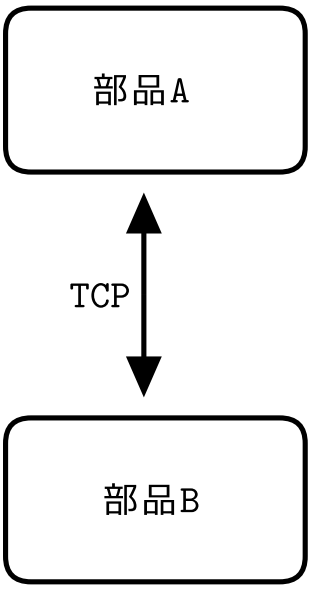


図 1.6　メッセージングのはじまり

多くのメッセージングプロジェクトと同様に、AMQP も先ほど挙げた問題をアドレッシング、ルーティング、キューングを行う「ブローカー」という新しい概念を用いて解決しようとしました。その結果、アプリケーションはブローカーに対して、クライアント/サーバープロトコルや、API を利用してドキュメント化されていないプロトコルをやり取りするようになりました。ブローカーは巨大で複雑なネットワークを縮小させる事に役立ちましたが、ブローカーをベースとしたメッセージングは Zookeeper の様な製品では必ずしも良い結果が得られませんでした。高性能なサーバーを追加していく内に、ブローカーが単一故障点になってしまったのです。あっという間にブローカーはボトルネックとなり、管理上のリスクとなりました。これをソフトウェアで解決する場合、第 2、第 3、第 4 のブローカーを追加し、フェイルオーバーの仕組みを作る必要がありました。人々がこれを行った結果、より多くの部品が増え、複雑になり、いろいろなものが壊れました。

そして中央ブローカーのセットアップには、専用の運用チームが必要でした。そして、ブローカーを昼夜構わず監視し、素行の悪いヤツを見つけて棒で叩く必要がありました。新しいサーバーが必要になり、さらにそのバックアップサーバーが必要になり、そのサーバーを管理する人材が必要になりました。この様な状況は、幾つものチームで数年に渡って運用する大規模なアプリケーションにおいては価値があるでしょう。

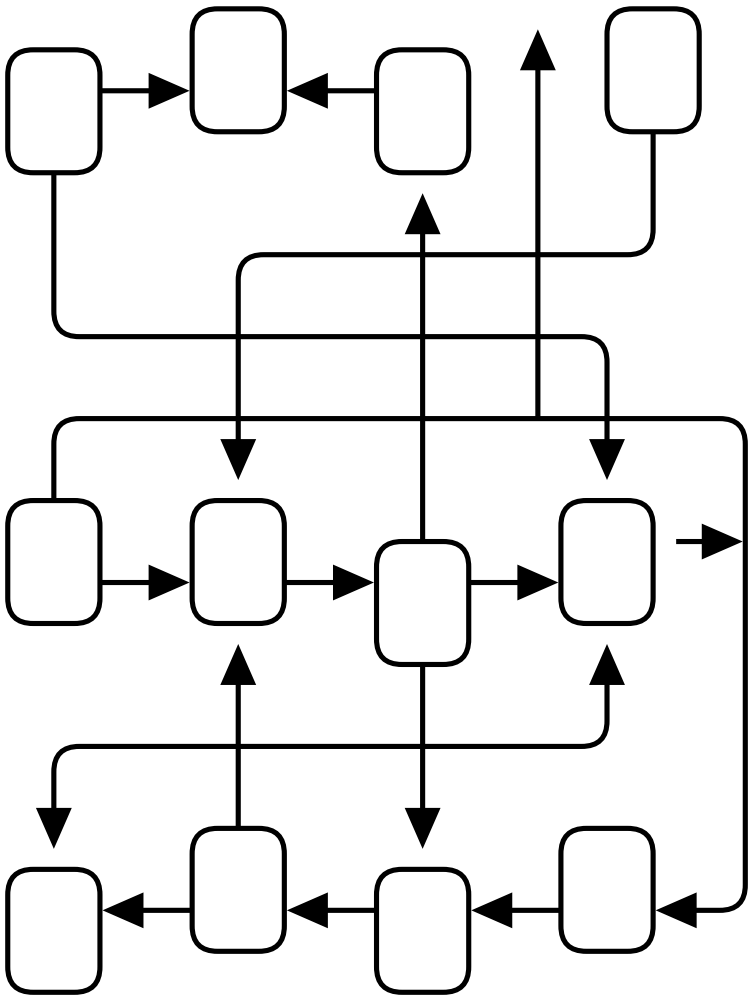


図 1.7　Messaging as it Becomes

つまり、中小規模のアプリケーション開発者にとってこれは罠なのです。ネットワークプログラミングを避けて一枚岩なアプリケーションを作るか、ネットワークプログラミングに挑戦して不安定で複雑なアプリケーションを作り、メンテナンスに苦しむか、メッセージング製品に頼り、スケーラブルだけど高価で壊れやすい技術を利用するという選択肢があります。前世紀のメッセージングが何故巨大であったかを考えると、これらは本当に良い選択肢ではありません。サポートやライセンス販売する人は大喜びでしょうが、ユーザーにとって良い事は一つもないからです。

私達に必要なのはシンプルかつ安価で様々なアプリケーションで動作するメッセージングを機能です。それは何にも依存せずリンクできるライブラリでなければなりません。追加の部品は必要ありません、すなわち追加のリスクはありません。それは、あらゆる OS とあらゆるプログラミング言語で動作しなければなりません。

そうして出来たのが ØMQ です。ØMQ はアプリケーションがネットワークを縦横無尽に横断する為に必要な問題を解決する、低コストで効率的な組み込みライブラリです。

仕様:

- I/O はバックグラウンドのスレッドで非同期に処理します。アプリケーションスレッドはロックフリーなデータ構造を利用して通信を行うので、ØMQ アプリケーションはロックやセマフォなどの同期処理を必要としません。
- コンポーネントを動的にできるように、ØMQ は自動的に再接続を行います。これにより、どの様な順番でコンポーネントを実行してもよくなります。そしていつでもネットワークに参加して離脱出来る、サービス指向アーキテクチャを作ることが出来ます。
- メッセージは必要に応じてキューに入れられます。それは賢く、メッセージは受信者に出来るだけ近いキューに入れられます。
- HWM(満杯マーク) と呼ばれる方法でキューが溢れないようにします。キューが一杯になった時、ØMQ は自動的に送信側をブロックするか、あるいはメッセージを捨てるかどうかをメッセージの種類によってコントロールできます。(これを「パターン」と呼びます。)
- アプリケーションは様々な通信手段を利用する事が出来ます。例えば、TCP, マルチキャスト、プロセス間通信、プロセス内通信など。異なる通信手段を利用するためにコードを修正する必要はありません。
- 受信側の読み込みが遅かったりブロックされている場合でも、メッセージパターンによって異なる戦略を利用して安全に処理します。
- リクエスト-応答パターンや、pub-sub パターンなど、様々なパターンを利用してメッセージをルーティング出来ます。これらのパターンによりネットワーク構造のトポロジーを構成できます。
- キューのプロキシを構成したり、メッセージを採取したり転送したり出来ます。プロキシは相互接続によるネットワークの複雑性を緩和します。
- 配送されたメッセージはフレーム境界を維持してそのまま送信されます。10K バイトのメッセージを書き込んだ場合、受信側では 10K バイトのメッセージを受け取ります。
- メッセージのフォーマットについては関わりません。データサイズはゼロかもしれませんし、何ギガバイトもの巨大サイズかもしれません。データの表現方法については msgpack や Google の protocol buffers などの好きなライブラリを選んで使って下さい。
- ネットワークエラーを賢く処理します。それが理にかなっている時は再試行を行います。
- 二酸化炭素排出量を削減します。サーバーの CPU 利用量と利用電力を減らします。そして古いサーバーを長く使い続けることが出来ます。アル・ゴアは ØMQ を気に入るでしょう。

実際のところ ØMQ はこれらの事よりもっと多くのことを行います。それはネットワー

ク機能を持ったアプリケーションの開発に多大な影響を及ぼします。一見、zmq_recv() や zmq_send() はソケット API と同じように見えますが、メッセージ処理タスクは即座に中央ループに入り、複数のタスクに分解されます。これは上品で自然な動作です。そしてこれらのタスクはノードに対応付けされ、任意の通信経路を経由してノードに転送されます。1 プロセスに 2 ノード配置する時、ノードとはスレッドを意味します。1 つのサーバーに 2 ノード配置する時、ノードはプロセスの事です。1 つのネットワークに 2 ノード配置する場合、ノードはサーバーを意味します。これらは全て同じように、アプリケーションを変更せず構成する事ができます。

1.11 ソケットスケーラビリティ

ØMQ のスケーラビリティを見てみましょう。これは天気配信サーバーを起動し、クライアントを並列に実行するシェルスクリプトです。

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

4 コアのマシンでクライアントの実行中に、top コマンドを利用すると以下のようなプロセス情報を確認できるでしょう。

```
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
```

ここで何が起こっているのか少し考えてみましょう。天気情報サーバーは 1 つのソケットを持ち、5 つのクライアントにデータを並行に送信しています。私達は並行クライアントを数千ほどに増やすことが出来ます。サーバーアプリケーションにこれらのコードは直接記述されていません。クライアントのリクエストを静かに受け付け、出来るだけ素早くネットワークにデータを配信する小さなサーバとして機能振る舞います。そしてそれはマルチスレッドサーバーであり、CPU リソースを無駄なく絞りとります。

1.12 ØMQ v2.2 から ØMQ v3.2 へのアップグレード

1.12.1 互換性のある変更

これらの変更は既存のアプリケーションコードに直接影響はありません。

- PUB-SUB フィルタリングをサブスクライバ側だけでなくパブリッシャーサイドでも行えるようになりました。これは多くの pub-sub ユースケースでパフォーマンスを大きく改善します。v3.2 と v2.1/v2.2 を組み合わせても安全です。
- ØMQ v3.2 で多くの新しい API が追加されました。(`zmq_disconnect()`, `zmq_unbind()`, `zmq_monitor()`, `zmq_ctx_set()`, など)

1.12.2 互換性の無い変更

アプリケーションや言語バインディングが影響を受ける主な変更です。

- `zmq_send()` と `zmq_recv()` メソッドのインターフェースが変更されました。古い関数は現在 `zmq_msg_send()` と `zmq_msg_recv()` という名前で提供されています。症状: コンパイルエラーが発生します。解決方法: コードを修正する必要があります。
- これらの 2 つのメソッドは、成功すると正の値、エラーが発生すると-1 を返します。バージョン 2 では成功時は常に 0 を返していました。症状: 正常な動作なのにエラーが発生したように見えてしまう。解決方法: エラー処理を厳密に -1 と非ゼロで判定すること。
- `zmq_poll()` はミリ秒ではなく、マイクロ秒待つようになりました。症状: アプリケーションの応答が止まって見える (正確には 1000 倍遅くなる)。解決方法: `zmq_poll()` を呼び出す時に、新しく定義された `ZMQ_POLL_MSEC` マクロを利用して下さい。
- `ZMQ_NOBLOCK` マクロは `ZMQ_DONTWAIT` という名前に変更になりました。症状: コンパイルエラー
- ZMQ_HWM ソケットオプションは、ZMQ_SNDHWM と ZMQ_RCVHWM に分割されました。症状: コンパイルエラー
- 全てではありませんが、ほとんどの `zmq_getsockopt()` オプションの値は整数値です。症状: `zmq_setsockopt()` や `zmq_getsockopt()` の実行時にエラーが発生します。
- `ZMQ_SWAP` オプションは廃止されました。症状: コンパイルエラー。解決方法: この機能を利用したコードを再設計して下さい。

1.12.3 互換性維持マクロ

アプリケーションを v2.x と v3.2 の両方で動作させたい場合があります。以下の C マクロ定義は、両方のバージョンで動作させる為に役立ちます。

```
#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
```

1.13 警告: 不安定なパラダイム!

従来のネットワークプログラミングは一般的に 1 ソケットに対して 1 つの接続、1 ピアと会話することを前提にして構築されています。マルチキャストプロトコルがありますが、これらはちょっと風変わりです。私達は「1 ソケット = 1 コネクション」を前提としたアーキテクチャを有る意味で拡張しました。論理的なスレッドを作成しそれぞれのスレッドが 1 ソケット、1 ピアとして機能します。これらのスレッドに情報や状態を格納します。

ØMQ の世界では、全てのコネクションの集合を自動的に管理する早くて小さい通信エンジンへの出入口です。あなたはオープンやクローズ、コネクションに設定された状態の設定を見ることが出来ません。送受信をブロッキングで行うかポーリングするかどうかはあなたがソケットと会話して決定します。コネクションはこれを自動的に管理しません。コネクションは隠蔽化しているため直接見えませんが、これが ØMQ のスケーラビリティの重要な鍵になります。

なぜならソケットと会話することで、ネットワークプロトコルやコネクション数を操作することが出来るからです。メッセージングパターンはあなたのアプリケーションコードで実装するよりも、ØMQ のレイヤで実装したほうがより拡張性が高まります。

ですので一般的な仮定が通用しない場合があります。サンプルコードを読む時に、あなたの頭の中で、既存の知識とマッピングしようとするかもしれません。「ソケット」という言葉を見た時、「ああ、これは別のノードへのコネクションを表すのね」と思うでしょうが誤りです。

「スレッド」という言葉を見た時、「ああ、スレッドが別ノードへのコネクションを制御しているのね」と思うかもしれませんが、これもまた誤りです。

このガイドブックを初めて読んでいるのなら、実際に ØMQ のコードを書けるようになるまで 1,2 日 (もしくは 3,4 日) かかるでしょう。特に、ØMQ がどの様に物事を単純化しているかについてあなたは混乱するかもしれません。あるいは ØMQ で一般的な仮定を適用しようとして上手く行かないかもしれません。そして全てが明らかになったその時、あなたは zap-pow-kaboom パラダイムシフトの真理と悟りの瞬間を経験するでしょう。

第 2 章

ソケットとパターン

第 1 章「基礎」では ØMQ の主要なパターンである「リクエスト・応答」、「pub-sub」、「パイプライン」などの基本的なサンプルコードを見てきました。この章では実際のプログラムでこれらのパターンをどの様に利用するかを手を動かしながら学んで行きましょう。

この章では以下の内容を学んでいきます。

- ØMQ ソケットを生成して、動作させる方法
- ØMQ ソケットを経由してメッセージを送受信する方法
- ØMQ の非同期 I/O モデルでアプリケーションを開発する方法
- 1 スレッドで複数のソケットで扱う方法
- 致命的、致命的でないエラーを適切に処理する方法
- Ctrl-C の様な割り込みシグナルを処理する方法
- ØMQ アプリケーションを正しく終了させる方法
- ØMQ アプリケーションでメモリリークチェックを行う方法
- マルチパートメッセージを送受信する方法
- 別のネットワークへメッセージを転送する方法
- 簡易メッセージキューブローカーの作成方法
- ØMQ でマルチスレッドアプリケーションを作る方法
- スレッド間でシグナルを利用する方法
- ØMQ でネットワークのノードを連携する方法
- pub-sub パターンにおけるメッセージエンベロープの作成と利用方法
- HWM(満杯マーク) を使ってメモリ溢れを防ぐ方法

2.1 ソケット API

正直に言うと、ØMQ には囮商法の様な所があるかもしれませんが謝罪はしません。その痛みはこれまでの痛みよりも良性の痛みだからです。私達がメッセージ処理エンジンの多くを隠蔽する事に尽力したことにより、ØMQ は親しみやすいソケットベースの API を提供しています。しかしその結果、分散ソフトウェアの実装や設計方法に関するあなたの考え方を変える必要があるかもしれません。

ソケット API はネットワークプログラミングの事実上の標準というだけでなく、目が飛び出るほど便利です。開発者にとって ØMQ の特に魅力的なところは、他の別の概念の代わりにソケットとメッセージを利用したという所です。これについては Martin Sustrik に賞賛を送りたいと思います。これにより「メッセージ指向ミドルウェア」というちょっと特殊な言葉は誰もが欲しがるピザの様な言葉に変化しました。

大好きな料理と同じように、ØMQ を飲み込むのは簡単です。ØMQ ソケットの操作は BSD ソケットとまったく同じように 4 つに分類する事が出来ます。

- ソケットの生成と開放。これはソケットの人生と責任範囲を示しています。(`zmq_socket()`, `zmq_close()`)
- 必要に応じてソケットオプションの取得や設定を行います。(`zmq_setsockopt()`, `zmq_getsockopt()`)
- ネットワークトポロジーにソケットを接続したり、待ち受けを行います。(`zmq_bind()`, `zmq_connect()`)
- ソケットを利用してメッセージの送受信を行います。(`zmq_send()`, `zmq_recv()`)

ソケットは常に void ポインタであり、メッセージは構造体であることに注意してください。[*1]つまり、C 言語ではソケットをそのまま渡しますが、`zmq_send()` や `zmq_recv()` はメッセージ構造体のポインタを渡して動作します。覚え方としては、「ソケットは私達の所有物である」が、「メッセージはあなたのコードの所有物である」と言えます。

ソケットの生成や破棄や設定はあなたの期待通りに動作します。ただし、ØMQ は柔軟に非同期で動作することを覚えておいて下さい。これはネットワークトポロジーへの接続方法や、ソケットの扱い方にいくらかの影響を与えます。

[*1] 訳注: メッセージ構造体は、古い API である zmq_msg_send() や zmq_msg_recv() に関しての説明だと思われる。

2.1.1 ソケットをトポロジーに接続する

2 つのノード間でコネクションを作成するためには、片方のノードで zmq_bind() を呼び、もう片方で zmq_connect() を呼び出します。一般則では、zmq_bind() したノードは「サーバー」と呼ばれ、公開されたネットワークアドレスが設定されています。一方、zmq_connect() を行うノードは「クライアント」と呼ばれ、動的なネットワークアドレスであっても構いません。従って私達は、「エンドポイントとしてソケットをバインドする」とか「ソケットをエンドポイントに接続する」という様な言い方をします。エンドポイントは公開されたネットワークアドレスになるでしょう。

ØMQ のコネクションは、古典的な TCP コネクションとは異なる点が幾つかあります。主要な違いは以下の通りです。

- ØMQ はプロセス内通信、プロセス間通信、TCP、pgm, epgm などの様々な通信方式を利用できます。(詳しくは zmq_inproc(), zmq_ipc(), zmq_tcp(), zmq_pgm(), zmq_epgm() の man ページを参照して下さい。)
- 一つのソケットで複数の送受信コネクションを扱うことが出来ます。
- zmq_accept() という関数は存在しません。ソケットがエンドポイントとしてバインドを行ったら、自動的に接続の受け付けを開始します。
- ネットワークコネクションはバックグラウンドで処理されます。そしてもし接続が切断された場合は自動的に再接続を行います。(接続相手が復旧した場合)
- アプリケーションから直接コネクションを触ることは出来ません。これらはソケットの中に隠蔽されています。

多くのアーキテクチャでは、いわゆるクライアント・サーバーモデルに従っており、サーバーは静的なコンポーネント、クライアントは動的に増減するコンポーネントとして構成されています。アドレッシングの問題として、一般的にサーバーのアドレスはクライアントに公開されている必要がありますが、その逆は必要ありません。ですので、どちらのノードが zmq_bind() を呼び出し、どちらのノードが zmq_connect() を呼び出すべきかどうかは殆どの場合明らかです。これは幾つかの例外的なネットワークアーキテクチャを除いて、どのソケット種別を利用するかどうかにも関連してきます。ソケット種別については後で説明します。

さて、サーバーを起動する前にクライアントを起動するような状況を想像してみましょう。伝統的なネットワークであれば明確なエラーが発生します。一方 ØMQ は部品を一時的に停止して、再開することが出来ます。クライアントノードが zmq_connect() を呼び出すと直ぐに接続を行い、メッセージをソケットに書き込み出来るようになります。サーバーが起動し、zmq_bind() が行われた段階で ØMQ はメッセージを配送します。

サーバーノードは一つのソケットに対して複数のエンドポイントを bind する事が出来ます。

(複数のプロトコルやアドレスを組み合わせる事も可能) これは異なる通信方式のネットワークを横断して接続を待ち受けることが出来るという事です。

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

UDP を除き、ほとんどの通信方式では同じエンドポイントを 2 度バインドする事は出来ません。IPC 通信方式でも 2 度 bind する事ができますが最初に bind を行ったプロセスのソケットは無効になります。これはプロセスのクラッシュから回復する為の手段です。

ØMQ は bind する側と接続する側について中立であろうとしますが、違いは明確です。後でもっと詳しく説明しますが結論だけ言うと、通常「サーバー」はトポロジーの静的な部品として位置づけられる固定的なエンドポイントと考えます。また、「クライアント」はエンドポイントに接続する動的に増減する部品です。多くの場合このモデルに基づいてアプリケーションを設計する事になるでしょう。

ソケットには幾つかの種類があります。ソケット種別はソケットの役割を定めたものであり、メッセージのルーティングや内側や外側でキューイングしたりする際のポリシーです。ソケットは例えばパブリッシャーソケットとサブスクライバーソケットの様に、特定のソケット種別との組み合わせで接続することが出来ます。また、ソケットは「メッセージングパターン」との組み合わせで機能します。これについては後ほど詳しく見ていきます。

異なる方法でソケットを接続するメッセージキューシステムこそが ØMQ の基本的な能力です。あとで出てきますが、上位レイヤに位置するプロキシ様なものがあります。しかし ØMQ の本質はレゴ・ブロックの様に部品を組み合わせることでネットワークアーキテクチャを構築できるという事です。

2.1.2 メッセージの送受信

メッセージの送受信を行うには、`zmq_msg_send()` 関数と `zmq_msg_recv()` 関数を利用します。慣習的な名前ではありますが、ØMQ の I/O モデルは古典的な TCP のものとは異なっている事を念頭に置いておく必要があります。

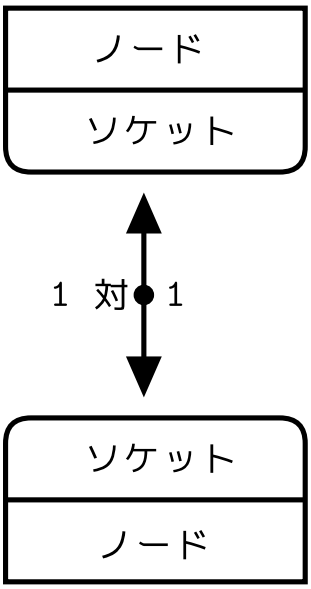


図 2.1 TCP ソケットは 1 対 1 に接続します

それでは、データを処理する際の TCP ソケットと ØMQ ソケットの主な違いを見て行きましょう。

zmq_send() 関数を呼んだ時、実際にメッセージが送出されるわけではありません。それはキューに入れられ、I/O スレッドによって非同期に送信されます。これは幾つか例外を除いてブロックされることはありません。ですので zmq_send() からアプリケーションの処理に戻ってきた時、メッセージは必ずしも送信されていないと思っていて下さい。

2.1.3 ユニキャストの通信方式

ØMQ はユニキャスト通信方式 (inproc, ipc, tcp) とマルチキャスト通信方式 (epgm, pgm) に対応しています。マルチキャストは高度なテクニックなので後で説明します。ファンアウト比の限界が分からないのに 1 対 N のユニキャストを使い始めるのは止めて下さい。

最も一般的なケースでは、非接続性の TCP 通信方式を使用します。この方式は柔軟性と移植性に優れていて、多くのケースで十分高速です。なぜ私達がこれを「非接続性」と呼ぶかと言うと、ØMQ の TCP 通信方式は接続先のエンドポイントが存在していなくても構わないからです。クライアントとサーバーはいつでも接続と bind を行うことが可能で、それが可能になった時点でアプリケーションからは透過的に接続が確立します。

プロセス間通信は TCP と同様に非接続性の通信方式です。この方式は今のところ Windows で動作しないという制限があります。慣例として、エンドポイント名に「.ipc」という拡張子を付けることで既存のファイルと競合することを避けています。UNIX ファイルシステムでは、IPC エンドポイントに適切なパーミッションが設定されている必要があります。そうしなけれ

ば異なるユーザー ID で動作しているプロセス間で IPC エンドポイントを共有できません。ですのでこれらのプロセスから IPC ファイルにアクセス出来る必要があります。

スレッド間通信やプロセス内通信は、接続性のある通信方式です。これは TCP や IPC よりはるかに高速です。この通信方式は TCP や IPC と比べて特別な制限があります。サーバーはクライアントが接続しにくるより前に bind していなければなりません。これは、将来のバージョンで改善されるかもしれませんが、現時点ではこういう制限があります。まず、ソケットを bind してから子スレッドを作成し接続を行って下さい。

2.1.4 ØMQ は中立キャリアではありません

初心者からのよくある質問に、「ØMQ で○○サーバーをどうやって作ればよいですか?」例えば「ØMQ で HTTP サーバーをどうやって作るの?」という質問を受けます。恐らく質問者は、普通の TCP ソケットで HTTP リクエストとレスポンスを転送できるのだから、ØMQ ソケットを利用して同じ事をより良く実現できるのではないか、と考えているのでしょう。

答えは、「その様な事は出来ない」です。ØMQ は中立キャリアではありません。ØMQ は転送プロトコルにフレーミングを強要します。既存のプロトコルは独自のフレーミングを利用しようとしますが、ØMQ のフレーミングとは互換性がありません。例として、HTTP リクエストと ØMQ リクエストを比較してみましょう。両者とも TCP 上のプロトコルです。

GET /index.html	13	10	13	10

図 2.2　HTTP の通信データ

ØMQ がフレームの長さを指定しているのに対し、HTTP リクエストはフレーミングの区切りに CR-LF を利用しています。ですから、ØMQ のリクエスト・応答ソケットパターンを利用して HTTP の様なプロトコルを実装したとしても、それは HTTP では無いのです。

5	H	E	L	L	O

図 2.3　ØMQ の通信データ

ただし ØMQ のバージョン 3.3 以降、ØMQ のフレーミングを利用せずソケットを読み書きすることが出来る、`ZMQ_ROUTER_RAW` というソケットオプションが追加されました。これを利用して、厳密に HTTP リクエストを実装することは可能です。 Hardeep Singh さんのこの貢献により彼の ØMQ アプリケーションから Telnet サーバーに接続することを可能にしました。

これはまだ実験的なものですが、これは ØMQ が新しい問題を解決して進化し続けていることを示しています。次はパッチはあなたの書いたパッチがマージされるかもしれませんよ。

2.1.5 I/O スレッド

先ほど述べたように ØMQ の I/O はバックグラウンドで行います。極端なアプリケーションを除いて、I/O スレッドは一つで十分です。新しいコンテキストを作成すると、一つのスレッドが起動します。一般的な経験則で言うと、一つのスレッドで 1 秒間に数ギガバイトのデータを扱う事ができます。I/O スレッドの数を増やしたい場合、ソケットを生成する前に `zmq_ctx_set()` を呼び出します。

```
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);
```

これまで一つのソケットで数千のコネクションを同時に扱える事を見てきました。これはあなたのアプリケーション開発に根本的な影響を与えます。伝統的なネットワークアプリケーションは、一つのプロセス、もしくは一つのスレッド毎にコネクション持ち、ソケットを扱います。ØMQ を利用すると、全ての構造を一つのプロセスでやり遂げるので、スケーラビリティの壁を打ち破る事ができます。

例外的にスレッド間通信 (例えば、マルチスレッドアプリケーションで外部との I/O ソケットを持たない場合) のみを利用している場合に I/O スレッドの数を 0 に設定することが出来ます。しかしこれは興味をそそるような重要な最適化ではありません。

2.2 メッセージングパターン

ØMQ のソケット API の包み紙の底にはメッセージングパターンが横たわっています。エンタープライズのメッセージング製品の経験があるか、もしくは UDP についてよく知っていればなんとなく解るでしょう。しかし殆どの ØMQ の初心者はこの事に驚くでしょう。彼らは既にソケットが 1 対 1 に別のノードに対応する TCP のパラダイムに慣れているからです。

ØMQ の動作について簡単に要約すると、ØMQ はデータの塊 (メッセージ) を効率良く迅速に転送します。ノードをスレッド、プロセス、別ノードに対応付ける事が出来ます。ØMQ は様々な通信手段 (プロセス内、プロセス間、TCP、マルチキャストなど) を扱う単一のソケット API を提供します。接続相手が一時的に居なくなった場合に再接続を行います。送信側と受信側の両方でメッセージを必要に応じてキューイングします。メモリやディスクを食いつぶした

りしないように、キューを慎重に管理します。ソケットのエラーを適切に処理します。全ての I/O はバックグラウンドのスレッドで処理されます。ノード間の通信にはロック・フリーのテクニックが使われていますのでロックやセマフォ、デッドロックが発生しません。

通してみると、パターンと呼ばれる明確なレシピに従ってメッセージをキューイングしたりルーティングしている事が分ります。これらのパターンによって ØMQ の知性が提供されます。これらは、分散データ処理に関して私達が経験によって苦労して得た最良の方法をカプセル化したものです。ØMQ のパターンは現在ハードコーディングされていますが、将来的にはユーザー定義のパターンを定義出来るようにするつもりです。

ØMQ のパターンはソケット種別のペアによって実装されます。言い換えると、ØMQ のパターンを理解するためにはソケット種別とそれがどの様に連携して動作するかを理解する必要があります。これは単に覚えるだけですのでここでの説明はこれくらいにしておきます。

ØMQ に組み込まれている主要なパターンは、

- リクエスト・応答: 複数のクライアントが複数のサービスに接続します。RPC(リモート・プロシージャ・コール) パターンやタスク分散処理パターンと言います。
- Pub-sub: 複数のサブスクライバが複数のパブリッシャーに接続します。これはデータ分散処理パターンと言います。
- パイプライン: 複数の層やループを持つファン・アウト/ファン・インパターンでノードを接続します。これは並行タスク分散処理パターン、コレクションパターンと言います。
- 排他的ペア: 2 つのソケットを排他的に接続します。これはプロセス内の 2 つのスレッドを接続するためのパターンです。**通常**のソケットペアと混同しないで下さい。

第 1 章「基礎」で最初の 3 つは既に見てきました。そして排他的ペアのパターンは後の章でやります。`zmq_socket()` の man ページはパターンについて詳しく説明していますのでよく理解できるまで何度か読み返すだけの価値はあります。以下は接続と bind を行う際に有効なソケットペアの組み合わせです。(どちら側でも bind 出来ます)

- PUB と SUB
- REQ と REP
- REQ と ROUTER
- DEALER と REP
- DEALER と ROUTER
- DEALER と DEALER
- ROUTER と ROUTER
- PUSH と PULL

- PAIR と PAIR

後ほど XPUB や XSUB というソケットも出てくるでしょう。これは PUB と SUB の raw ソケットのような物です。これ以外の組み合わせはドキュメント化されていなかったり、信頼できない結果が得られます。将来的なバージョンでは正しくエラーが返ってくるようになるでしょう。もちろん他のソケットタイプをブリッジして、一度ソケットから読み込んだメッセージを他のソケットに書き込むような事は可能です。

2.2.1 ハイレベル・メッセージングパターン

これらの主要なパターンが ØMQ で料理されます。 これらは ØMQ API の一部分であり、コア C++ ライブラリで実装され、全ての小売店で販売されるようになります。

これら基本的なメッセージパターンの上に、ハイレベルなメッセージングパターンを追加する事が出来ます。アプリケーションで利用している様々な言語で ØMQ の上にハイレベルなパターンを構築します。これらはコア・ライブラリの一部ではありませんので、ØMQ のパッケージには含まれていませんし、ØMQ コミュニティで配布しているわけではありません。例えば Majordomo パターンについては第 4 章の「信頼性のあるリクエスト・応答パターン」で詳しく解説しますが、これは別プロジェクトとして GitHub でホスティングされています。

この本の目的の一つは大小様々なハイレベルパターンを知る事で、メッセージを正しく処理し、信頼性のある pub-sub アーキテクチャを構築できるようにする為です。

2.2.2 メッセージの処理

libzmq のコアライブラリは、送受信を行う 2 つの API を持っています。zmq_send() と zmq_recv() 関数については既に簡単な使い方を見てきました。私達は時々 zmq_recv() を利用しますが、一定のバッファーサイズを超えたメッセージを切り捨てる為、任意のメッセージサイズを扱う際には都合が悪いことがあります。ですので、zmq_msg_t 構造体を渡す事の出来る、より高機能で複雑な 2 つ目の API が用意されています。

- メッセージの初期化: zmq_msg_init(), zmq_msg_init_size(), zmq_msg_init_data()
- メッセージの受信: zmq_msg_send(), zmq_msg_recv()
- メッセージの開放: zmq_msg_close()
- メッセージデータへのアクセス: zmq_msg_data(), zmq_msg_size(), zmq_msg_more()
- メッセージプロパティの取得、設定: zmq_msg_get(), zmq_msg_set()
- メッセージ操作: zmq_msg_copy(), zmq_msg_move()

通信経路上では、ØMQ のメッセージは 0 から任意のサイズのデータとしてメモリに格納されています。そして protocol buffers や msgpack、JSON、あるいはあなたのアプリケーションで利用できる独自の方法でシリアライゼーションを行うことができます。可搬性のあるデータ表現を選択する事は賢明な判断ですが、この決定にはトレードオフが伴うでしょう。

ØMQ メッセージは zmq_msg_t 構造体 (利用している言語によってはクラス) でメモリに格納されています。以下は ØMQ メッセージを C 言語で扱う上での基本原則です。

- メッセージとは生成、もしくは受信した zmq_msg_t オブジェクトの事です。バイト配列ではありません。
- メッセージを受信するには、まず zmq_msg_init() を呼び出して空のメッセージを生成し、zmq_msg_recv() に渡して受信する必要があります。
- メッセージを送信するには、まず zmq_msg_init_size() を呼び出してデータと同サイズのメッセージオブジェクトを作成します。そして memcpy() などを利用してデータをメッセージオブジェクトにコピーし、zmq_msg_send() に渡して送信します。
- メッセージオブジェクトを開放するには zmq_msg_close() を呼び出します。参照を開放し、ØMQ は最終的にメッセージは破壊します。
- メッセージの内容にアクセスするには zmq_msg_data() を利用します。メッセージのデータサイズを知りたい場合は zmq_msg_size() を呼び出します。
- zmq_msg_move(), zmq_msg_copy(), zmq_msg_init_data() などの関数は man ページを読み、何故これが必要なのかはっきりと理解できるまで利用してはいけません。
- zmq_msg_send() を読んでメッセージを送信すると、ØMQ はメッセージオブジェクトを初期化します。具体的にはサイズが 0 になります。同じメッセージを 2 度送信することは出来ませんし、送信後のメッセージにアクセスする事は出来ません。
- これらのルールは zmq_send() と zmq_recv() には当てはまりません。これらの関数はメッセージオブジェクトではなくバイト配列を受け取るからです。

もしも同じメッセージを 2 度以上送信したい場合、2 つ目のメッセージも zmq_msg_init() を呼び出して初期化し、zmq_msg_copy() を呼び出して 1 つ目のメッセージをコピーして下さい。これはデータそのものをコピーせず、参照をコピーします。これでメッセージを 2 度以上送信出来るようになり、最後のコピーが送信あるいは close された場合にメッセージが開放されます。

ØMQ は一つのメッセージに複数のフレームを含めて送受信を行う事が出来る、マルチパート・メッセージに対応しています。これは実際のアプリケーションでよく使われるので、第 3 章の「リクエスト・応答パターンの応用」で説明します。

フレーム (ØMQ の man ページで「message parts」とも呼ばれています) は ØMQ の基本的な転送フォーマットです。フレームは長さが決められたバイト配列であり、0 以上の長さを指

定できます。もしあなたが TCP プログラミングに慣れている場合、何故フレームが便利なのか理解できるはずです。ネットワークソケットからどれくらいのサイズのデータが送られてくるか予め知ることができるからです。

ØMQ が TCP 接続上でフレームを読み書きする方法を定義した ZMTP という転送プロトコルがあります。この仕様はとても短いですので、もしこれに興味があれば読んでみて下さい。

元々、ØMQ のメッセージは UDP の様に一つのフレームしか持っていませんでした。私達は後々マルチパートメッセージを扱えるようにこれを拡張しました。これはとても単純に一連のフレームが連続している場合はビット集合の「more」フラグをオンにします。これにより、ØMQ API はメッセージを受信する際に「more」フラグがあるかどうか確認する事ができます。

低レベル API やマニュアルの中には、「メッセージ」と「フレーム」という言葉について幾つか曖昧な点があるのでここで整理しておきます。

- メッセージは一つ以上の部品で構成されます。
- これらの部品はフレームと呼ばれます。
- これらの部品は zmq_msg_t オブジェクトです。
- 各部品は低レベル API を用いて別々に送受信することが出来ます。
- 高レベル API ではマルチパートメッセージをまとめて送信する事ができるラッパーを提供します。

メッセージについて以下のことも知っておくと良いでしょう。

- 長さ 0 サイズのメッセージを送ることが可能です。(例えばノードから別のノードに通知を送りたい場合など)
- ØMQ はメッセージの部品を全て転送するか、全く送信しないかのどちらかであることを保証します。
- メッセージは即座に送信されず、不確定なタイミングで送信されます。従ってマルチパートメッセージはメモリに収まらなければなりません。
- メッセージはメモリに収まる必要があります。もし、あなたが巨大なファイルを送りたい場合それらを分割してシングルパートメッセージとして送信する必要があります。マルチパートメッセージを利用した所でメモリの使用量は変わりありません。
- スコープが外れた時にオブジェクトを自動的に開放しない言語では、メッセージの受信が完了した際には `zmq_msg_close()` を呼び出す必要があります。メッセージを送信した後にこの関数を呼び出す必要はありません。

繰り返し言いますが、まだ `zmq_msg_init_data()` 関数はまだ利用しないで下さい。これはゼロコピーを行う手段であり、間違いなくあなたを悩ませるでしょう。細かいことを気にする前に、ØMQ についてもっと重要な事を学ぶ必要があります。

高レベル API を利用すると面倒なことが起きる場合があります。この関数は、単純さよりもパフォーマンスに最適化されているからです。注意深く man ページを読まずにこれらの関数を使用すると、間違いなく誤った使い方をしてしまうでしょう。ですのでこれらの API を簡単に使えるようにラップしてやることですが言語バインディングの重要な仕事になります。

2.2.3 複数のソケットを処理する (Handling Multiple Sockets)

これまでのサンプルコードでは、全てメインループで以下の処理を行っていました。

1. ソケットからのメッセージを待つ
2. メッセージを処理する
3. 繰り返し

もし複数のエンドポイントから同時に受信を行いたい場合はどうしたら良いのでしょうか? 最も単純な方法はファン・インとして全てのエンドポイントを一つのソケットで接続する方法です。これはリモートのエンドポイントが同じパターンである場合に有効ですが、PULL ソケットから PUB エンドポイントへと接続する場合に上手く行きません。

正しく複数のソケットから同時に受信するためには zmq_poll() を利用します。もっと良い方法は、zmq_poll() をラップしてイベントドリブンに反応するフレームワークを用いることだが、ここではそれについて取り上げません。

さて、やってはいけない例として泥臭いハックを見て行きましょう。その目的は、非ブロッキングでソケットを読み込む方法を学ぶ為です。この例では非ブロッキングで 2 つのソケットから読み込みを行う例を示します。ややこしいですが、気象情報のサブスクライバーと並行処理のワーカーの両方の機能を持ったプログラムを例に使用します。

msreader.c: Multiple socket reader in C

```
// 複数のソケットから受信する例
// これは単純な受信ループを行うバージョンです

#include "zhelpers.h"

int main (void)
{
    // ベンチレーターに接続します
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");
```

```
    // 気象情報サーバーに接続します
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

    // 両方のソケットからメッセージを受信します。
    // この際、ベンチレーターのソケットが優先されます。
    while (1) {
        char msg [256];
        while (1) {
            int size = zmq_recv (receiver, msg, 255, ZMQ_DONTWAIT);
            if (size != -1) {
                // ベンチレーターからのメッセージを処理
            }
            else
                break;
        }
        while (1) {
            int size = zmq_recv (subscriber, msg, 255, ZMQ_DONTWAIT);
            if (size != -1) {
                // 気象情報サーバーからのメッセージを処理
            }
            else
                break;
        }
        // メッセージはありませんので 1 ミリ秒スリープします。
        s_sleep (1);
    }
    zmq_close (receiver);
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

この方法の欠点は、ループの最後にある sleep によって一つ目のメッセージを読み込む前に遅延が発生してしまうことです。これは、ミリ秒以上の遅延を許容しないアプリケーションで問題になるでしょう。また、nanosleep() の様な関数を利用しても構いませんが、ビジーループが発生しないかどうか確認する必要があります。

この例では、2 つめのソケットよりも優先的に一つ目のソケットの読み込みが行われます。

さて次は、同じようなアプリケーションで zmq_poll() を使う例を見て行きましょう。

mspoller.c: Multiple socket poller in C

```
// 複数のソケットから受信する例
// zmq_poll() を利用するバージョンです

#include "zhelpers.h"

int main (void)
{
    // ベンチレーターに接続します
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    // 気象情報サーバーに接続します
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

    // 両方のソケットからメッセージを受信します。
    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) {
                // ベンチレーターからのメッセージを処理
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            int size = zmq_recv (subscriber, msg, 255, 0);
            if (size != -1) {
                // 気象情報サーバーからのメッセージを処理
            }
        }
    }
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

zmq_pollitem_t 構造体は 4 つのメンバ変数を持っています。

```
typedef struct {
    void *socket; // 監視する 0MQ ソケット
    int fd; // もしくは、監視するファイルディスクリプタ
    short events; // 監視するイベント
    short revents; // イベント発生後の zmq_poll() の返り値
} zmq_pollitem_t;
```

2.2.4 マルチパートメッセージ

ØMQ は幾つかのフレームをまとめたマルチパートメッセージを扱うことが出来ます。実際にはマルチパートメッセージを利用すると、宛先情報を付与して、シリアライズを行うため、処理が重くなります。後ほど、応答エンベロープの詳細について見ていきます。

今から学ぶことは、単純にアプリケーションから安全にマルチパートメッセージを読み書きする方法です。これは中身を読まずにメッセージを転送するプロキシの様なアプリケーションで必要になります。

マルチパートメッセージを扱うには、zmq_msg オブジェクトをそれぞれ処理する必要があります。例えば、5 つのフレームを送信するには、5 つのメッセージを生成し、それぞれの zmq_msg オブジェクトを開放する必要があります。その時 zmq_msg オブジェクトを配列で持つなどして、まとめて行っても構いませんし、一つずつ生成して送信しても構いません。

以下は、マルチパートメッセージを送信する方法です。

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

以下は、メッセージを受信し、各メッセージフレームを処理する方法です。

```
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
}
```

マルチパートについて以下の事を知っておいてください。

- マルチパートを送信する際、最初のメッセージフレームは実際には最後のメッセージフレームを送信する時にまとめて送信されます。
- zmq_poll() を利用している場合、最初のメッセージを受信した時には、もう残りのメッセージは全て到着しています。
- マルチパートメッセージは全て受信するか、全く受信しないかのどちらかです。
- メッセージフレームは zmq_msg オブジェクトで分割されます。
- zmq_msg_more を呼び出して more 属性を確認してもしなくても、全てのメッセージを受信することになります。
- 送信時、全てのメッセージフレームが送信され、最後のフレームが受信されるまで、メモリ上の ØMQ キューに保存されています。
- 送信したメッセージフレームを部分的に取り消すには、ソケットをクローズするしか方法はありません。

2.2.5 中継とプロキシ

ØMQ は知性の分散を目指しますが、ネットワークの中央に何もないというわけではありません。そこにはメッセージを扱うインフラや ØMQ で構築したインフラで満たされています。ØMQ は小さなパイプから、完全なサービス指向ブローカーまで様々な配管を行うことが可能です。メッセージング業界では、中央でメッセージを取り扱う役割を仲介者と呼びます。ØMQ ではこの役割の事を文脈に依存してプロキシ、キュー、フォワーダー、デバイス、ブローカと呼びます。

私達の社会や経済の中で巨大なネットワークを持つ仲介者が溢れている様に、この様なパターンは現実の世界では極めて一般的です。現実世界の言葉では、問屋、卸業者などと呼ばれます。

2.2.6 動的ディスカバリー問題

ディスカバリーは大きなアーキテクチャを設計する上で遭遇する問題の一つです。部品はどうやってその他の部品を見つければ良いのでしょうか。部品が動的に増減する場合、これは特に難しいので、私達はこれを「動的ディスカバリー問題」と呼んでいます。

動的ディスカバリー問題には幾つかの解決方法があります。最もシンプルな方法は、接続先をハードコーディングもしくは設定で指定することです。しかしこの方法では、新しい部品を追加した時にネットワークを再構成する必要があります。

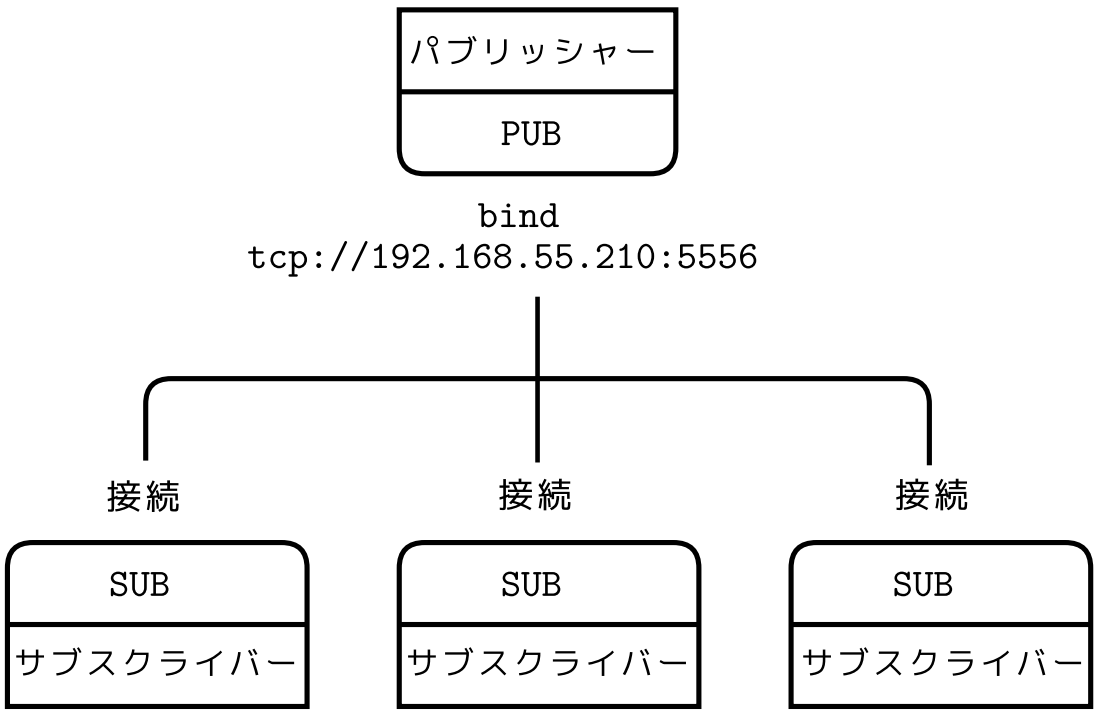


図 2.4　Small-Scale Pub-Sub Network

実際には、この方法は次第に脆く、扱いにくいアーキテクチャになるでしょう。例えば、1つのパブリッシャーと 100 のサブスクライバー居るとしましょう。各サブスクライバーがパブリッシャーに接続するために、各サブスクライバーにパブリッシャーのエンドポイントを設定する必要があります。サブスクライバーは動的な部品であり、パブリッシャーは静的な部品なので、まあこれは簡単です。しかし突然パブリッシャーを追加しなければならなくなった時、これは簡単な事ではありません。サブスクライバーからパブリッシャーへの接続が増え続けていった場合、動的ディスカバリー問題を回避するコストは高まっていきます。

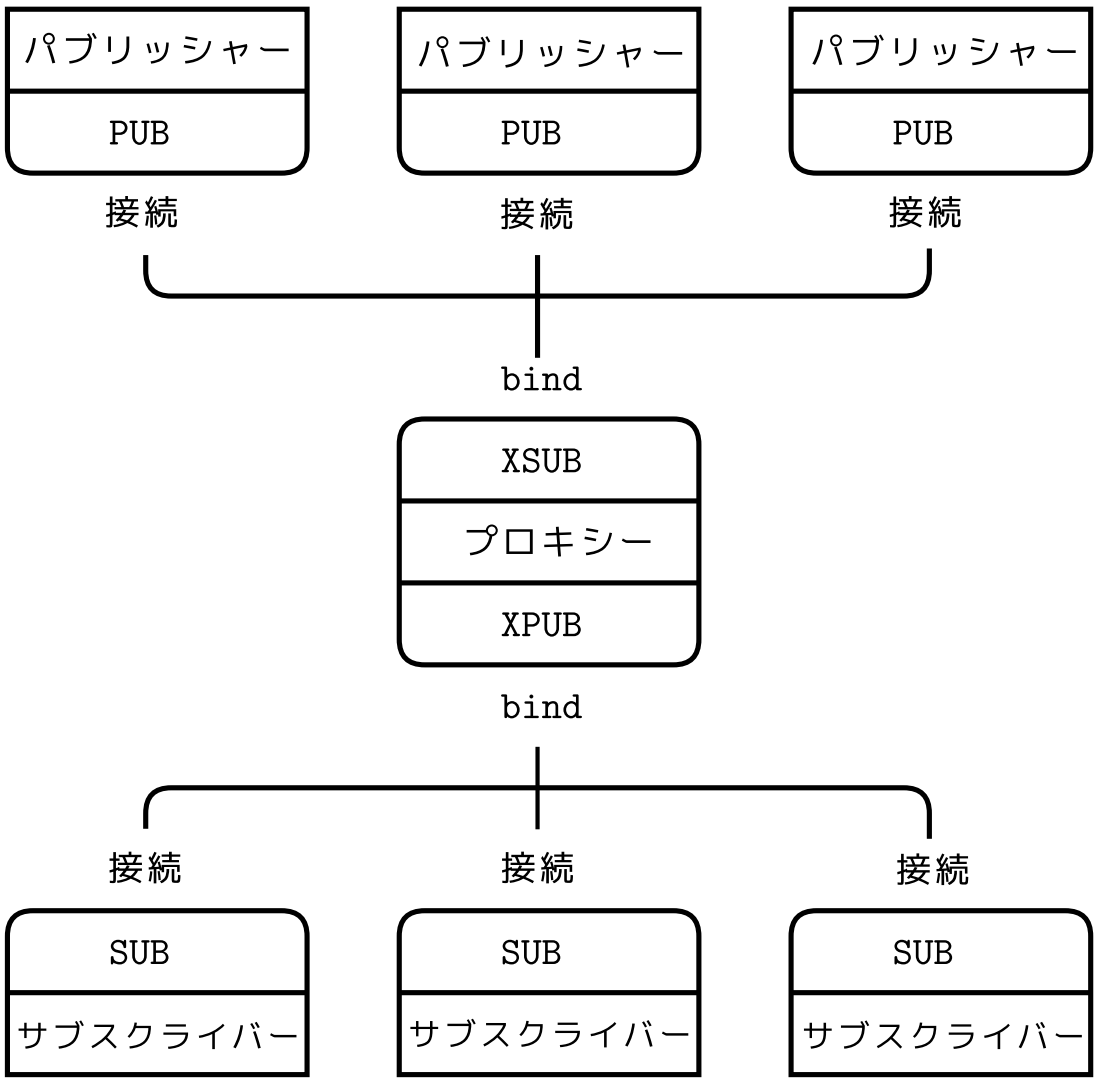


図 2.5　Pub-Sub Network with a Proxy

これには幾つかの解決方法がありますが、単純な解決方法は、ネットワークの静的なポイントとなる仲介者を導入することです。古典的なメッセージングシステムでは、この仕事はメッセージブローカーの役目とされていました。ØMQ ではメッセージブローカーは存在しませんが、とても簡単に仲介者を構築することが出来ます。

巨大なネットワークが最終的に仲介者を必要とするなら、なぜ単純にメッセージブローカーを導入しないのか不思議に思うかもしれません。入門者の場合、それは適切な妥協策です。パフォーマンスの事を無視すれば常にスター型トポロジが問題なく動作するでしょう。しかしながらブローカは強欲であり、様々な役割を中央集権した結果、どんどんステートフルで複雑になり、最終的には問題が発生します。

仲介者はステートレスなメッセージスイッチとして考えた方が良いでしょう。似たような例えとして HTTP プロキシがあります。これは特別な役割を持っていません。以下のサンプル

コードでは、pub-sub プロキシを追加することで動的ディスカバリー問題を解決します。ネットワークの中間にプロキシを配置し、そのプロキシは XSUB ソケットを開き、公開された IP アドレスとポートで XPUB ソケットを待ち受けます。そして、全てのプロセスは、プロキシに対して接続を行います。これにより、サブスクライバーやパブリッシャーを追加することが容易になります。

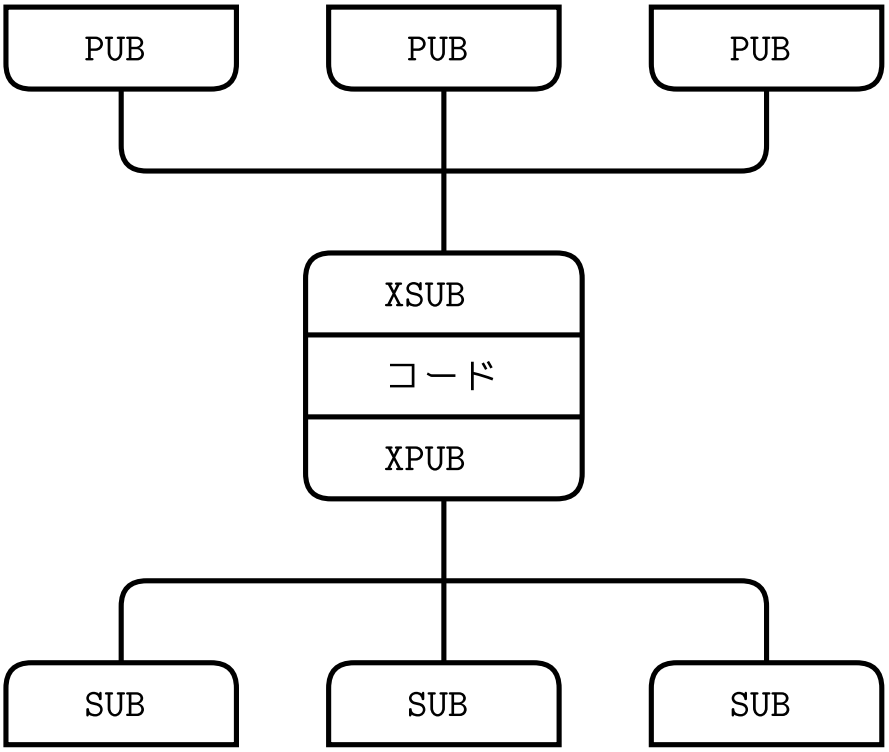


図 2.6　Extended Pub-Sub

接続をサブスクライバーからパブリッシャーに転送するためには XPUB ソケットと XSUB ソケットが必要です。XSUB と XPUB は特別に生のメッセージを転送するという点を除いて、SUB, PUB ソケットとまったく同じです。 プロキシは XSUB, XPUB ソケットを読み書きする事でサブスクライバー側からのメッセージをパブリッシャー側に転送します。これは XSUB, XPUB ソケットの主要な利用方法です。

2.2.7 共有キュー (DEALER and ROUTER sockets)

Hello World クライアント・サーバーアプリケーションの例では、1 つのクライアントが 1 つのサービスに接続を行いました。実際のケースでは、複数のクライアントが複数のサービスに接続できる必要があります。これは、スレッド、プロセス、ノードを増やすことでスケールアップしてサービスを拡張することが出来ます。唯一の制限は、サービスはステートレスでな

ければなりません。全ての状態はデーターベースなどのストレージに格納されている必要があります。

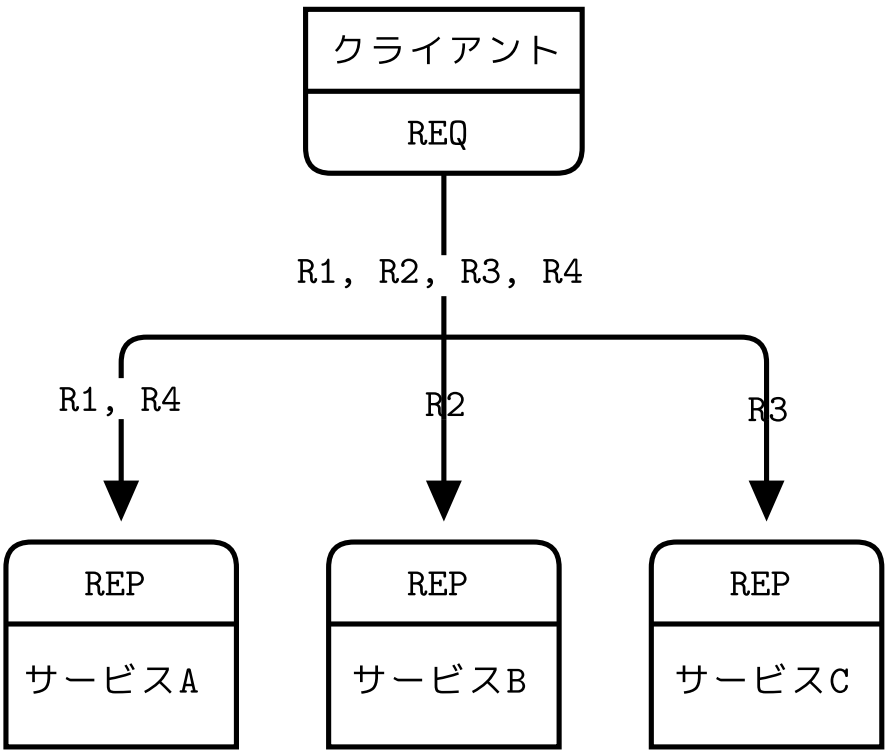


図 2.7　Request Distribution

複数のクライアントから複数のサーバーに接続する方法は 2 つあります。強引な方法だと複数のサービスに対してそれぞれのソケットを用意して接続し、分散したリクエストを行います。上記の図は、サービスのエンドポイント A, B, C に対して、クライアントは R1, R2, R3, R4 という 4 つのリクエストを行っています。そして、R1 と R4 はサービス A に、R2 はサービス B に、R3 はサービス C にリクエストが行われていることを示しています。

この設計でクライアントとサービスを追加するのは比較的簡単でしょう。各クライアントはサービスに対して分散してリクエストを送ります。しかし各クライアントはサービスのトポロジーを把握している必要があります。もし、100 のクライアント居て、新しいサービスを追加する場合、100 のクライアントにに対して再設定と再起動を行う必要があります。

突然スパコンのクラスタのリソース不足が発生し、数百のサービスノードを追加する必要が発生したとして、この様な作業を夜中の 3 時にやりたいとは思いません。多くの静的な部品は液体コンクリートの様な物です。知識が多くの静的部品に分散していると、トポロジーの変更が大変です。私達に必要なのは、クライアントとサービスの間に居て、トポロジーに関する全ての知識を持った存在です。これがあれば、トポロジーの他の部品に触れることなく、いつでもサービスやクライアントを追加したり削除したりできるはずです。

ですのでこの様な柔軟性を提供するちょっとしたメッセージキューブローカーを書いてみます。ブローカーはクライアント側のフロントエンドとサービス側のバックエンドの 2 つのエンドポイントを bind します。そして zmq_poll() を利用して 2 つのソケットの動作を監視して、メッセージを橋渡しします。ØMQ が自動的にキューを管理していますので、ブローカが直接それを行うことはありません。

REQ ソケットから REP ソケットに通信する際、厳密には同期的にやりとりを行います。サービスはリクエストを受信し、応答を返します。その後、クライアントは応答を受信します。もし、クライアントやサービスがこれ以外の動作 (例えば応答を待ってる時に 2 つ目のリクエストを送信するなど) を行うとエラーが返ります。

しかし今回のブローカーは非同期で行います。REP/REQ ソケットを利用せず、zmq_poll() を利用してソケットを監視します。

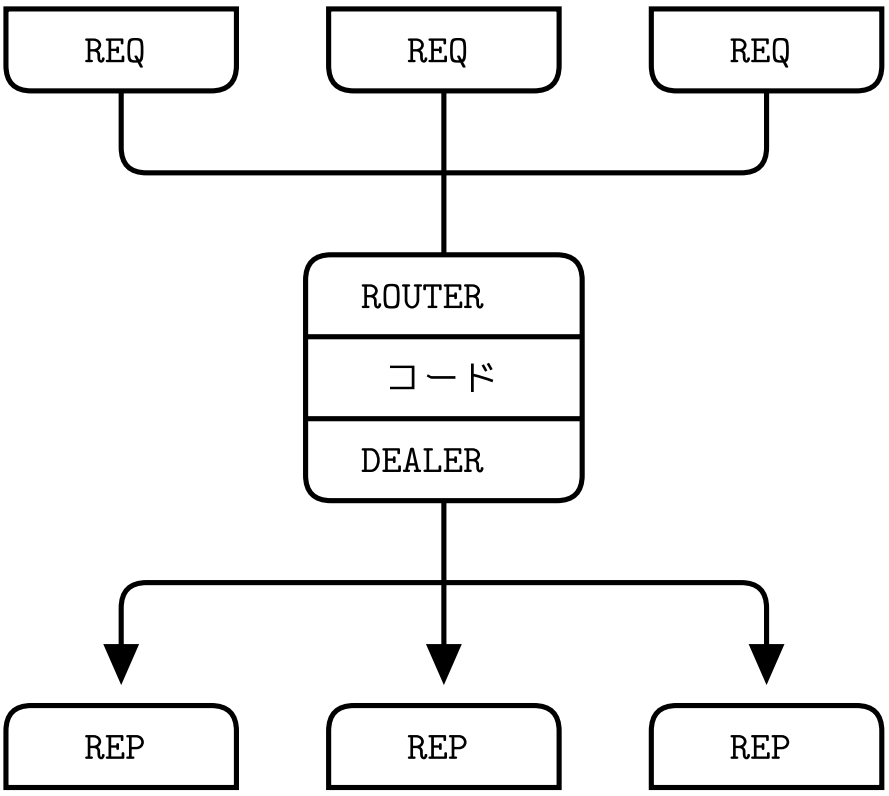


図 2.8　Extended Request-Reply

幸いなことに、リクエスト・応答を非ブロッキングで行う DEALER と ROUTER と呼ばれる 2 つのソケットがあります。第 3 章「リクエスト・応答パターンの応用」では DEALER と ROUTER ソケットを利用した様々な非同期のリクエスト・応答パターンを見ていきます。ここでは、リクエスト・応答パターンの仲介者として動作する簡単なブローカーを実装する方法として DEALER と ROUTER の説明を行います。

今回の単純なリクエスト・応答パターンでは、REQ ソケットは ROUTER ソケットと通信し、DEALER ソケットは REP ソケットと通信を行います。DEALER と ROUTER ソケットの間では、ソケットに届いたメッセージを、もう一方に転送するブローカーの様なコードが動作しています。

リクエスト・応答ブローカーは 2 つのエンドポイントを bind します。1 つ目はクライアントが接続してくるフロントエンド側のソケットです。もうひとつはワーカーが接続してくるバックエンド側のソケットです。ブローカーの動作を確認するために、バックエンドのワーカーの数を変更してみたくなるでしょう。以下はリクエストを行うクライアントのコードです。

rrclient.c: リクエスト・応答クライアント

```
// Hello World クライアント
// REQ ソケットで tcp://localhost:5559 に接続し、
//  「Hello」を送信して「World」を受信します。

#include "zhelpers.h"

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

    // サーバーとの通信ソケット
    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;
}
```

以下はワーカーのコードです。

rrworker.c: リクエスト・応答ワーカー

```
// Hello World ワーカー
// REP ソケットで tcp://localhost:5560 に接続し、
//  「Hello」を受信して「World」を応答します。

#include "zhelpers.h"

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

    // クライアントとの通信ソケット
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_connect (responder, "tcp://localhost:5560");

    while (1) {
        // クライアントからのリクエストを待ちます
        char *string = s_recv (responder);
        printf ("Received request: [%s]\n", string);
        free (string);

        // 何らかの処理
        sleep (1);

        // クライアントに応答
        s_send (responder, "World");
    }
    // この処理が行われることはありませんが、一応の終了処理です
    zmq_close (responder);
    zmq_ctx_destroy (context);
    return 0;
}
```

そして以下がブローカーのコードです。マルチパートメッセージも正しく処理できます。

rrbroker.c: リクエスト・応答ブローカー

```
// 単純なリクエスト・応答ブローカー

#include "zhelpers.h"

int main (void)
{
    // コンテキストとソケットの準備
```

```
    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");

    // ポーリングの準備
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    // メッセージの受け渡し
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            while (1) {
                // メッセージ全体を処理
                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;      // 最後のメッセージフレーム
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            while (1) {
                // メッセージ全体を処理
                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;      // 最後のメッセージフレーム
            }
        }
    }
    // この処理が行われることはありませんが、一応の終了処理です。
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
```

```
}
```

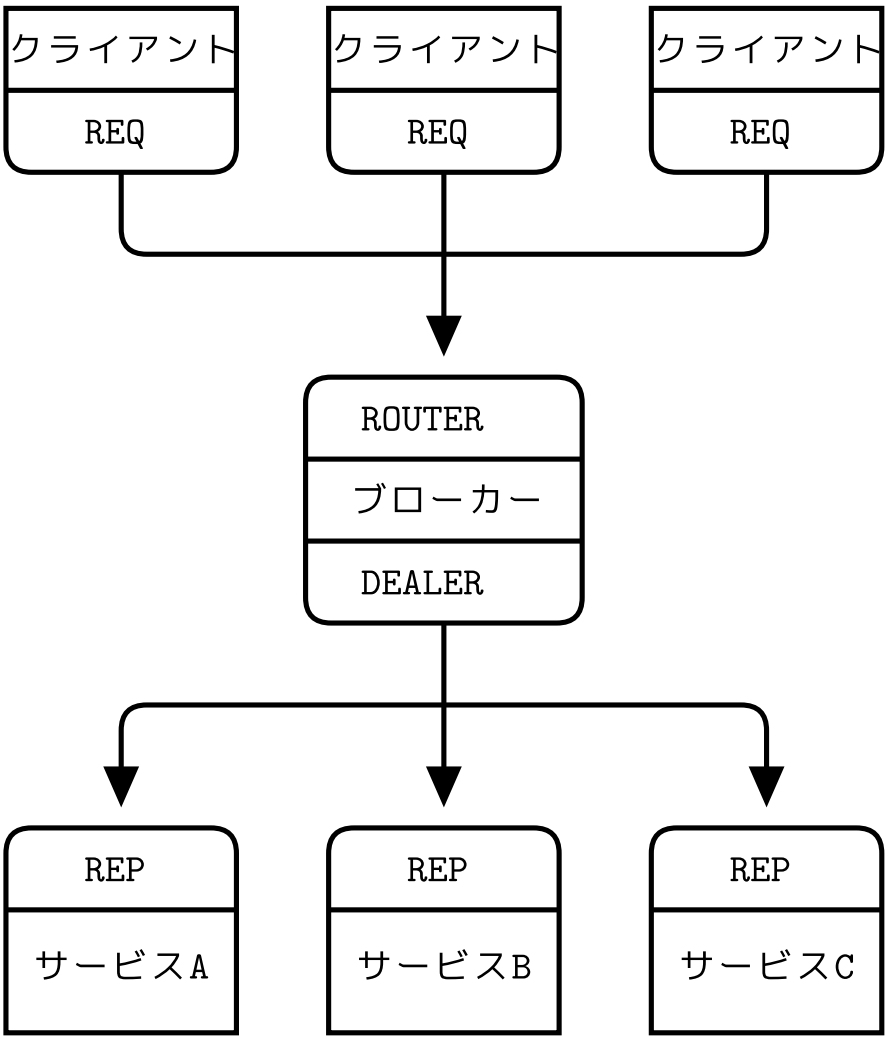


図 2.9　Request-Reply Broker

リクエスト・応答ブローカーを利用すると、クライアントは直接ワーカーの数を気にしなくて良くなるのでアーキテクチャを拡張し易くなります。これにより、静的なノードは中間にあるブローカのみとなります。

2.2.8 ØMQ の組み込みプロキシ関数

前節の rrbroker は非常に便利でメインループに注目すると再利用可能である事が分ります。キューを共有して pub-sub 転送行う仲介者もわずかな手間で実装出来ます。ØMQ はこの様な機能をラップした単一の関数 `zmq_proxy()` を用意しています。

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

この関数は 3 つの引数をとります (3 つ目の引き数はデータの採取が必要であれば)。zmq_proxy() 関数を呼び出すと、まさに rrbroker のメインループを実行します。それでは zmq_proxy を利用して、リクエスト・応答ブローカーを書きなおしてみましょう。

msgqueue.c: メッセージキューブローカー

```
// 単純なメッセージキューブローカー
// リクエスト・応答ブローカーと同じですが、zmq_proxy() を利用しています。

#include "zhelpers.h"

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

    // クライアント側ソケット
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    int rc = zmq_bind (frontend, "tcp://*:5559");
    assert (rc == 0);

    // サービス側ソケット
    void *backend = zmq_socket (context, ZMQ_DEALER);
    rc = zmq_bind (backend, "tcp://*:5560");
    assert (rc == 0);

    // プロキシの開始
    zmq_proxy (frontend, backend, NULL);

    // この処理は実行されません
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

あなたが典型的な ØMQ ユーザーであれば、この段階でこう考えるでしょう。「プロキシのソケット種別に何を指定しても良いのかな?」簡単な答えると、「通常は ROUTER/DEALER, XSUB/XPUB, PULL/PUSH の組み合わせしか利用しません。」

2.2.9 ブリッジ通信

ØMQ ユーザーからのよくある質問に、「どの様にして ØMQ ネットワークと通信技術 X と接続すれば良いですか?」というものがあります。X はその他の通信技術やメッセージングプラットフォームの事です。

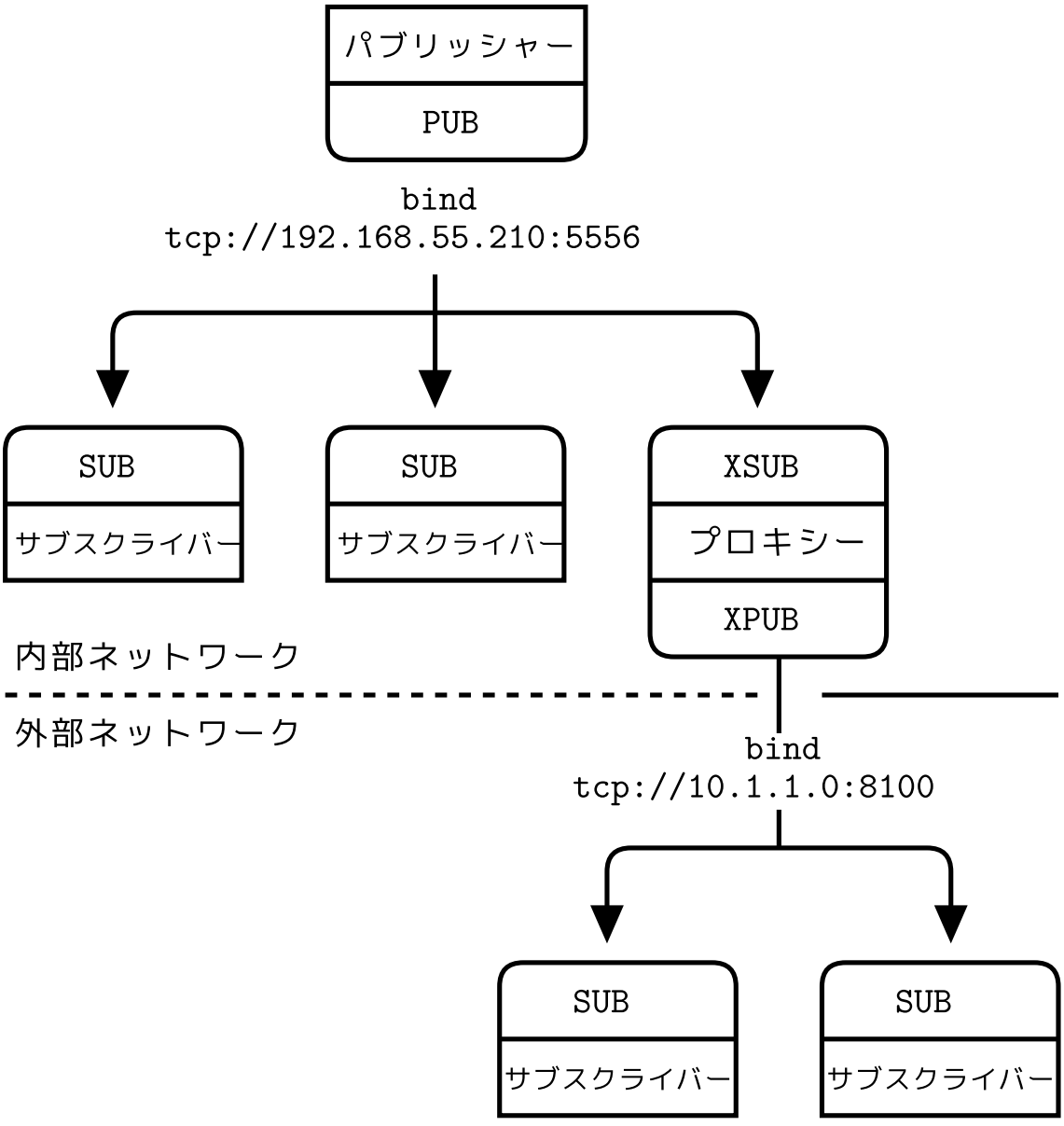


図 2.10　Pub-Sub Forwarder Proxy

単純な解決方法はブリッジを構築することです。ブリッジとは片方で 1 つのプロトコルに対応するソケットを持ち、もう片方で別のプロトコルに変換して接続を行う小さなアプリケーションです。これはプロトコルインタプリターと言っても構いません。ØMQ では 2 つの異な

る通信方式やネットワークをブリッジする事が可能です。

例として、パブリッシャーとサブスクライバーの間の 2 つのネットワークをブリッジする小さなプロキシを作ってみましょう。フロントエンドソケット (SUB) は気象情報サーバーが居る内部ネットワークに面しており、バックエンドソケット (PUB) は外部ネットワークに面しています。このプロキシはフロントエンドソケットで気象情報の更新を受信し、バックエンドソケットにデータを再配布します。

wuproxy.c: 気象情報更新プロキシ

```
// 気象情報プロキシ

#include "zhelpers.h"

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

    // 気象情報サーバー側ソケット
    void *frontend = zmq_socket (context, ZMQ_XSUB);
    zmq_connect (frontend, "tcp://192.168.55.210:5556");

    // サブスクライバー側のソケット
    void *backend = zmq_socket (context, ZMQ_XPUB);
    zmq_bind (backend, "tcp://10.1.1.0:8100");

    // プロキシの実行
    zmq_proxy (frontend, backend, NULL);

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

これはプロキシの時に見たサンプルコードとよく似ていますが、フロントエンドソケットとバックエンドソケットが異なるネットワークにある所がポイントです。この方法は、TCP のサブスクライバから受け取ったメッセージをマルチキャストネットワークに流すような場合にも利用できます。

2.3 エラー処理と ETERM

ØMQ におけるエラーハンドリングの哲学はフェイル・ファーストと回復力の組み合わせです。プロセスは内部エラーに関しては出来るだけ脆弱であるべきであり、外部要因のエラーや攻撃に対しては出来るだけ強固であるべきです。例えば生体細胞は、単一の内部エラーが発生すると自己崩壊するように出来ていますが、外部からの攻撃に対しては出来るだけ対抗しようとします。

アサーションは ØMQ コードを強固にする為に欠かせない調味料です。これには細胞壁のような壁があるはずです。もし障害が内部要因か外部要因かの区別がつかない場合、それは設計上の欠陥です。C/C++ ではアサーションはアプリケーションを直ちに停止させます。他の言語では例外を投げるか、あるいは終了するでしょう。

ØMQ が外部要因の障害を検出するとエラーをコードに返します。エラーからの復旧戦略がない場合、稀にメッセージを喪失してしまう可能性があります。

これまで見てきた C 言語のサンプルコードではエラー処理を行っていませんでした。実際のコードでは、ØMQ API の呼び出し毎にエラー処理を行う必要があります。もしあなたが C 言語でなくその他の言語のバインディングを利用している場合はバインディングがエラー処理を行ってくれるでしょう。C 言語ではそれを自分でやる必要があります。そのルールは POSIX の決まりごと似たような感じで単純です。

例:

```
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;
}
```

致命的なエラーとして扱ってはいけない例外的な状況が主に 2 つあります。

- ZMQ_DONTWAIT を指定してメッセージを受信しようとして実際に受信するデータが無かった場合、ØMQ は-1 を返して errno に EAGAIN をセットします。
- あるスレッドが zmq_ctx_destroy() を呼び出した際に、まだ別のスレッドがブロッキング処理中であった場合。zmq_ctx_destroy() はコンテキストを正しく開放し、ブロッキング処理を行っている関数は-1 を返して errno に ETERM をセットします。

C/C++ の assert() は最適化によって完全に取り除かれますので assert() 内で ØMQ API を呼び出してはいけません。最適化によって全ての assert() は削除され、アプリケーション上手く動作しなくなるでしょう。

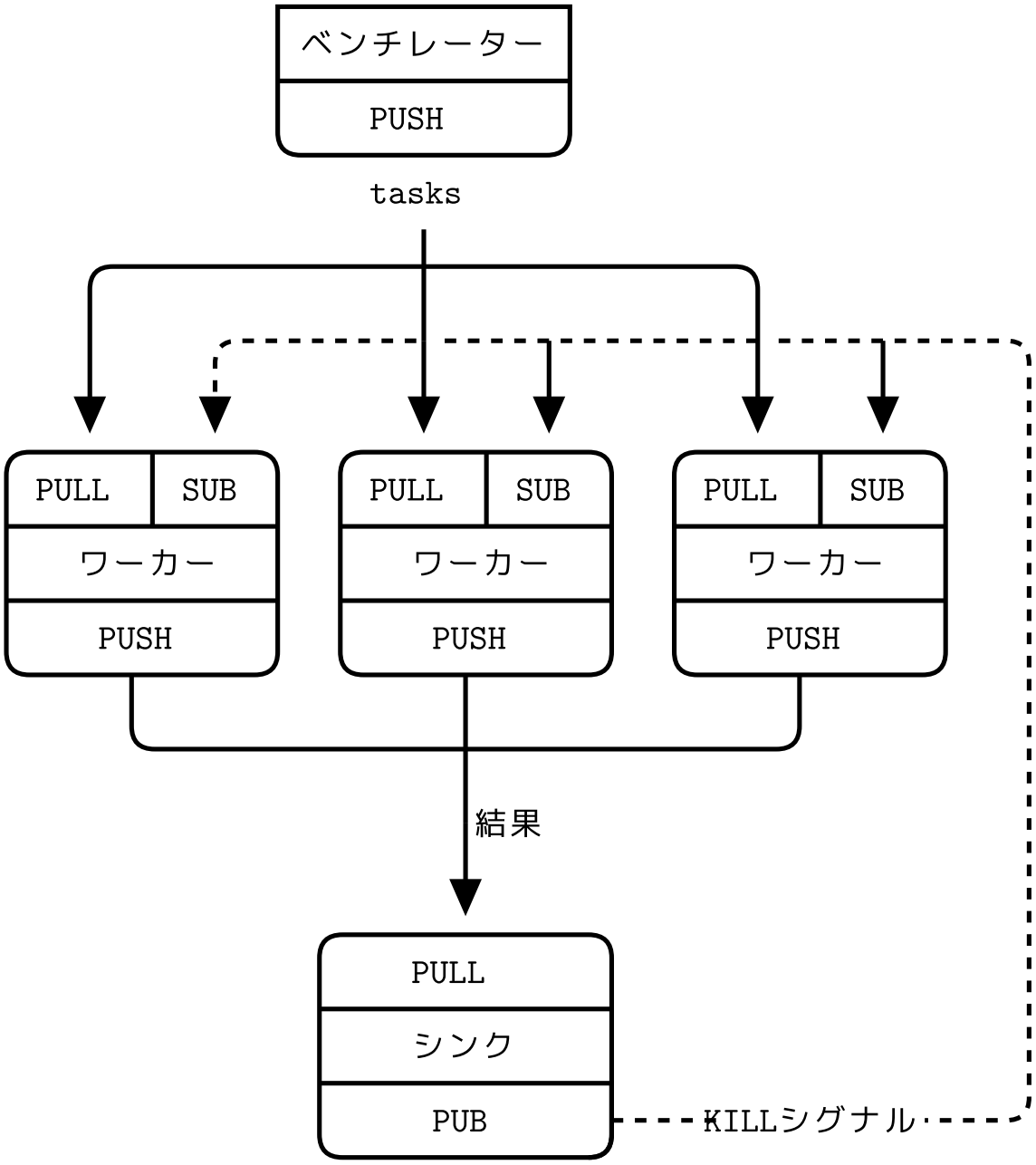


図 2.11　Parallel Pipeline with Kill Signaling

プロセスを行儀よく終了させる方法を見て行きましょう。以前の節で出てきた、並行パイプラインのサンプルコードを思い出しましょう。無事に処理が完了し、バックグラウンドで動作している大量のワーカーを終了させたいとします。こんな時はワーカーに対して終了メッセー

ジを送信してみましょう。この処理行うのに最適な部品はシンクです。なぜならシンクは処理の完了を知ることができるからです。

どの様にしてシンクからワーカーに接続すればよいでしょうか。PUSH/PULL ソケットは一方方向ですし、動作中の別のソケット種別に切り替えたりすることは出来ません。では pub-sub モデルでワーカーに終了メッセージを送る方法について説明しましょう。

- シンクで新しい PUB ソケットのエンドポイントを作成します。
- ワーカーで新しいエンドポイントを作成します。
- シンクで処理の完了を確認したら、PUB ソケットに対して KILL メッセージを送信します。
- ワーカーは KILL メッセージを受信し、終了します。

It doesn't take much new code in the sink: 追加のコードはそれほど必要ありません。

```
void *controller = zmq_socket (context, ZMQ_PUB);
zmq_bind (controller, "tcp://*:5559");
...
// ワーカーを終了させるシグナルを送信
s_send (controller, "KILL");
```

この場合ワーカープロセスは 2 つのソケットを先ほど学んだ zmq_poll() を使って管理します。1 つ目はタスクを受信を行うソケット、もうひとつは KILL メッセージなどの制御コマンドを受信するソケットです。

taskwork2.c: 終了シグナルを受け付ける並行タスクワーカー

```
// ワーカータスク - 再設計バージョン
// pub-sub ソケットを利用して終了シグナルを受信します

#include "zhelpers.h"

int main (void)
{
    // メッセージの受信用ソケット
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    // メッセージの送信用ソケット
    void *sender = zmq_socket (context, ZMQ_PUSH);
```

```
    zmq_connect (sender, "tcp://localhost:5558");

    // 制御コマンドを受信するソケット
    void *controller = zmq_socket (context, ZMQ_SUB);
    zmq_connect (controller, "tcp://localhost:5559");
    zmq_setsockopt (controller, ZMQ_SUBSCRIBE, "", 0);

    // 2 つのソケットからのメッセージを処理
    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);      // 進捗の表示
            fflush (stdout);
            s_sleep (atoi (string));     // 何らかの処理
            free (string);
            s_send (sender, "");         // 結果をシンクに送信
        }
        // 終了命令などの制御コマンドを処理します
        if (items [1].revents & ZMQ_POLLIN)
            break;                       // ループを抜ける
    }
    zmq_close (receiver);
    zmq_close (sender);
    zmq_close (controller);
    zmq_ctx_destroy (context);
    return 0;
}
```

こちらは、改修を行ったシンクアプリケーションです。結果の収集が完了した時に終了メッセージを全てのワーカーにブロードキャストしています。

tasksink2.c: Parallel task sink with kill signaling in C

```
// シンクタスク - 再設計バージョン
// pub-sub ソケットで終了シグナルを送信します

#include "zhelpers.h"

int main (void)
```

```
{
    // メッセージの受信用ソケット
    void *context = zmq_ctx_new ();
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    // ワーカーの制御用ソケット
    void *controller = zmq_socket (context, ZMQ_PUB);
    zmq_bind (controller, "tcp://*:5559");

    // 処理の開始を待ちます
    char *string = s_recv (receiver);
    free (string);

    // 計測開始
    int64_t start_time = s_clock ();

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

    // ワーカーに終了命令を送信
    s_send (controller, "KILL");

    zmq_close (receiver);
    zmq_close (controller);
    zmq_ctx_destroy (context);
    return 0;
}
```

2.4 割り込みシグナル処理

実際のアプリケーションでは Ctrl-C やその他のシグナルを受け取って行儀よく終了する必要があります。既定では、kill シグナルを受信すると即座に終了するので、メッセージはキューを溜めたままになり、ファイルなどもクローズされません。

以下はシグナルを処理する方法です。

interrupt.c: 正しく Ctrl-C を処理する方法

```
// Ctrl-C を処理する方法

#include <zmq.h>
#include <stdio.h>
#include <signal.h>

// シグナルハンドリング
//
// アプリケーションの開始時に s_catch_signals() を呼び出し、
// メインループの中で s_interrupted が 1 になったら終了します。
// zmq_poll() を利用した場合でも同様です。

static int s_interrupted = 0;
static void s_signal_handler (int signal_value)
{
    s_interrupted = 1;
}

static void s_catch_signals (void)
{
    struct sigaction action;
    action.sa_handler = s_signal_handler;
    action.sa_flags = 0;
    sigemptyset (&action.sa_mask);
    sigaction (SIGINT, &action, NULL);
    sigaction (SIGTERM, &action, NULL);
}

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

```
    s_catch_signals ();
    while (1) {
        // ここでブロックし、終了シグナルが来たら終了します
        char buffer [255];
        zmq_recv (socket, buffer, 255, 0);
        if (s_interrupted) {
            printf ("W: interrupt received, killing server...\n");
            break;
        }
    }
    zmq_close (socket);
    zmq_ctx_destroy (context);
    return 0;
}
```

このプログラムは s_catch_signals() を呼び出して Ctrl-C(SIGINT) や SIGTERM をトラップします。これらのシグナルを受信すると s_catch_signals() によってグローバル変数 s_interrupted を設定します。この場合、アプリケーションは自動的に終了しません。シグナルハンドラに感謝して下さい。代わりにリソースを開放して行儀よく終了する事が出来ます。これは明示的に割り込みを確認して正しく処理する必要があります。メインコードの最初でこの s_catch_signals() を呼び出して下さい。割り込みは以下の ØMQ API の呼び出しに影響します。

- 同期的なメッセージの送受信でブロッキングしている最中にシグナルを受信すると、その関数は EINTR を返します。
- s_recv() の様なラッパー関数は割り込みが入ると NULL を返します。

ですので関数の返り値が EINTR や NULL になっていないか確認し、必要に応じてグローバル変数 s_interrupted も確認して下さい。

以下は典型的なコード片です。

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

もしも s_catch_signals() を呼び出しておいて、s_interrupted をチェックしなかった場

合、Ctrl-C や SIGTERM は無視されるようになります。これは便利かもしれませんが一般的ではありません。

2.5 メモリリークの検出

長期間動作し続けるアプリケーションは適切にメモリを管理してやる必要があります、そうしなければ全てのメモリを使い果たして最後にはクラッシュしてしまうからです。自動的にメモリ管理を行う言語を利用しているそこのあなた、おめでとうございます。C 言語や C++ でプログラムを書く場合はメモリ管理の責任はプログラマにあります。以下は、valgrind を利用してプログラムのメモリリークを調査する為の簡単なチュートリアルです。

- valgrind をインストールするには、Ubuntu や Debian では以下のコマンドを実行します。

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

- valgrind は既定では多くの警告を表示します。問題の無い警告を無視するために以下の内容のファイル (vg.supp) を作成して下さい。

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

- Ctrl-C を押した時に行儀よくアプリケーションを終了するようにしてください。勝手に終了する場合これは必要無いですが、長期間動作し続けるアプリケーションでは必要不可欠です。行儀よく終了しなかった場合は valgrind が開放していないメモリ領域に関して警告を表示します。
- コンパイルオプションに-DDEBUG を付けてアプリケーションをビルドすると、valgrind

はメモリリークに関する警告をより詳細に教えてくれます。

- valgrind は以下のように実行して下さい。

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

そして、何も問題が見つからなかれば、以下のように素敵なメッセージが表示されます。

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

2.6 マルチスレッドと ØMQ

ØMQ はマルチスレッドアプリケーションを書く為の最適な方法を提供します。古典的なソケットを利用する場合と比べて ØMQ ソケットを使う場合はちょっとした調整を行えば良いだけで、あなたが知っているマルチスレッドプログラミングに関する知識をほとんど必要としません。既存の知識は庭に放り投げて、油を注いで燃やして下さい。

ØMQ では、完璧なマルチスレッドプログラムを作る為にmutex やロック、プロセス間通信などは必要あり

私の言う「完璧なマルチスレッドプログラム」とは、書き易く理解しやすいコードであり、どの様なプログラミング言語や OS でも同じ設計方法で機能し、CPU 数に比例して性能が向上して CPU リソースを無駄にすることが無いという意味を含んでいます。

もしあなたがロックやセマフォやクリティカルセクションなどのマルチスレッドプログラミングに関するテクニックを長年に渡って学んで来たのであれば、これらが如何に無駄な事だったかを思い知ることになるでしょう。私達が 30 年以上の並行プログラミングの経験から学んだたった一つのことは「状態を共有してはいけない」という事です。それは 2 人の酔っぱらいがビールを取り合っているようなものです。彼らが仲の良い友達同士であれば問題にはなりませんが、そうで無ければ遅かれ早かれ喧嘩になってしまうことでしょう。そして、そこに新しく酔っぱらいを追加するとかられもまたビールを巡って争いを初めます。マルチスレッドアプリケーション大多数は酔っぱらいが居る酒場の喧嘩のようなものです。

以下のリストは、古典的な共有メモリのマルチスレッドアプリケーションで発生る問題の数々です。これらの問題で発生するストレスとリスクに押しつぶされてしまったコードは突然動かなくなります。バグのあるプログラムに関する経験では世界一の大企業が「マルチスレッドプログラムでよく見られる 11 の問題」という文書を公開しました。この中では、同期忘れ、不適切な粒度、読み取りと書き込みの分裂、ロックフリーの並べ替え、ロックコンボイ、2 ステップダンス、優先順位の逆転といった問題が挙げられています。

はい、私は今 11 ではなく 7 つしか挙げませんでした。これはさして問題ではありません。

重要なのは電力網や株式市場のシステムで忙しい木曜日の午後 3 時に 2 ステップロックコンボイをやりたいのか、という事です。これはより複雑なハックで複雑な副作用と戦っているような物です。

広く使われているモデルは業界の基盤となっているにも関わらず、状態を共有するという根本的な欠陥があります。インターネット上のサービスの様に無制限に拡張させたい場合、欠陥のあるプログラミングモデルのよう共有するのではなく、メッセージの送信を行いましょう。

ØMQ で適切なマルチスレッドプログラムを書くには以下のルールに従う必要があります。

- データはスレッド毎に専有されており、複数のスレッドでデータが共有することはありません。ただし、スレッドセーフを保証している ØMQ コンテキストは例外です
- ミューテックスやクリティカルセクション、セマフォなどの古典的な並行メカニズムは一度忘れて下さい。これらは、ØMQ アプリケーションにおいてはアンチパターンです。
- ØMQ コンテクストはプログラムの開始に生成し、プロセス内通信ソケットを行う全てのスレッドにこれを渡して下さい。
- アプリケーションを骨組みになるスレッドは attached スレッドを作成して下さい。そしてプロセス内通信でペアのソケットを利用して親スレッドに接続して下さい。親スレッド側のソケットで bind して、子スレッド側のソケットで接続を行うのが定石です。
- 独立したタスクをシミュレートするには、detached スレッドを利用し、独立したコンテキストを作成して下さい。通信方式は TCP を利用することで、後に大きな修正を行うこと無くスタンドアローンのプロセスに移行できます。
- スレッド間の全てのやり取りはあなたの定義したメッセージで行います。
- ØMQ ソケットはスレッドセーフではありませんので ØMQ ソケットを複数のスレッドで共用しないで下さい。ソケットを別のスレッドに移行することは技術的には可能ですが熟練技術が必要です。複数のスレッドで一つのソケットを扱う唯一の場面は魔法のようなガーベジコレクションを持った言語バインディングくらいです。

例えばアプリケーションの中で 2 つ以上のプロキシを動作させたい場合、それぞれのスレッドでプロキシで動作させたいと思うかもしれません。1 つのスレッド内でエラーが発生した際に、ソケットを別のスレッドに渡すことが簡単にできてしまいますが、実際にこれをやるとランダムに失敗します。ソケットの生成を行ったスレッドでのみ close を行うということを忘れないで下さい。

これらのルールに従った場合、とてもエレガントなマルチスレッドアプリケーションを構築できます。後からスレッドではなくプロセスに分離することも可能です。これはアプリケーションロジックはスレッドでもプロセスでも別ノードでも、適切な規模に合わせてスケールアップできるという事を意味します。

ØMQ は仮想的な「グリーンスレッド」ではなく OS のネイティブスレッドを使用していま

す。これにはあなたが新しくスレッド API を学ばなくて良いという事や、ØMQ スレッドは明確に OS のスレッドと対応するという利点があります。また、Intel の ThreadChecker といった標準的なツールを利用してアプリケーションを観察することも出来ます。欠点はネイティブのスレッド API は必ずしも移植性があるとは限らないという点です。また、大量のスレッドを生成すると OS に負担が掛かってしまうでしょう。

それではこれらがどの様に動作するか実際に見て行きましょう。ずっと前に見た Hello World に幾つかの機能を追加します。元々のサーバーはシングルスレッドで動作していました。1 リクエストで行う処理が少なければ、1 コア分の CPU を利用して結構な処理を行うことができます。しかし、実際のサーバーでは 1 リクエストで多くの処理を行います。1 万クライアントが同時にアクセスしてきた場合にはシングルコアでは不十分でしょう。リクエストを受け付けたら即座にワーカースレッドに処理を分担し、最終的にはワーカースレッドが応答を返します。

もちろん、プロキシブローカーを利用して外部のワーカープロセスで全ての処理を行う事も可能ですが、16 コアの CPU を使い切るために 16 個のプロセスを起動するよりはマルチスレッド化した 1 つのプロセスを起動するほうが簡単でしょう。また、ワーカーをマルチスレッドで実行すると、余計なネットワークトラフィックやレイテンシが無くなります。

Hello World サービスのマルチスレッド版にはブローカとワーカーの機能が一つのプロセスに押し込まれています。

mtserver.c: サービスのマルチスレッド化

```
// Hello World サーバーのマルチスレッド化

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

static void *
worker_routine (void *context) {
    // ディスパッチャーと通信するソケット
    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);
        // なんらかの処理
        sleep (1);
        // クライアントへの応答
```

```
        s_send (receiver, "World");
    }
    zmq_close (receiver);
    return NULL;
}

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

    // クライアントと通信するソケット
    void *clients = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (clients, "tcp://*:5555");

    // ワーカーと通信するソケット
    void *workers = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (workers, "inproc://workers");

    // ワーカーのスレッドプールを起動
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_routine, context);
    }
    // キュープロキシでワーカースレッドとクライアントスレッドを接続します
    zmq_proxy (clients, workers, NULL);

    // この処理が行われることはありませんが、一応の終了処理です
    zmq_close (clients);
    zmq_close (workers);
    zmq_ctx_destroy (context);
    return 0;
}
```

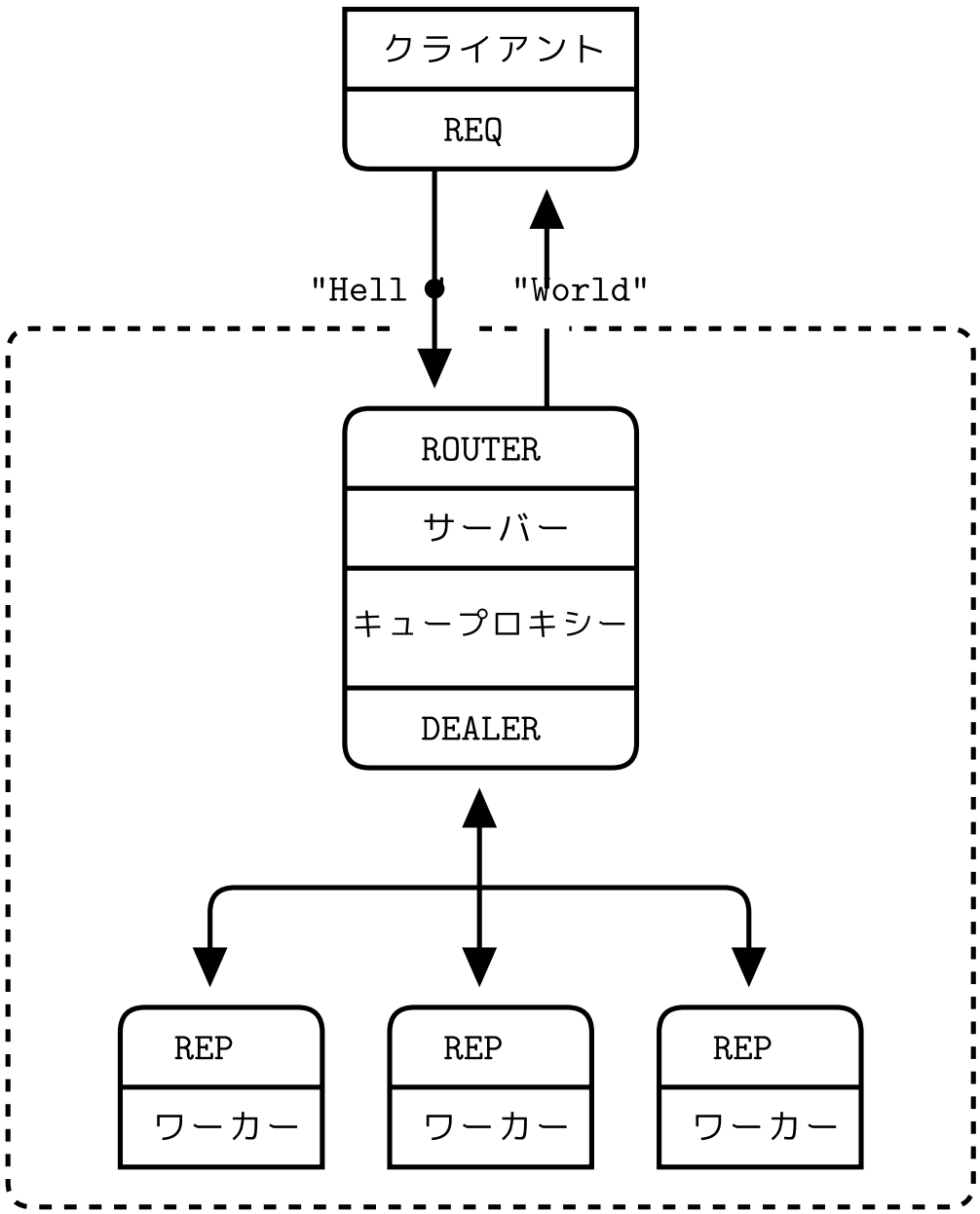


図 2.12　サーバーのマルチスレッド化

もうコードを見れば何をやっているか理解できる頃でしょう。一応説明しておくと、

- サーバーは複数のワーカースレッドを開始し、それぞれのワーカースレッドで REP ソケットを作成してこのソケット経由でリクエストを処理します。ワーカースレッドはシングルスレッド版のサーバーと同様です。唯一の違いは転送方式が TCP ではなくプロセス内通信であることと、bind-接続の方向性くらいです。
- サーバーは ROUTER ソケットを生成して bind を行い、外部インターフェースである TCP を利用してクライアントと通信します。

- サーバーは DEALER ソケットを生成して bind を行い、内部インターフェースであるプロセス内通信を利用してワーカースレッドと通信します。
- サーバーは 2 つのソケットをつなぐプロキシを開始します。プロキシは全てのクライアントから受け付けたリクエストをワーカーに均等に分担します。プロキシは元のクライアントに対して応答を返します。

スレッドの生成は多くのプログラミング言語で移植性が無いことに注意して下さい。POSIX ライブラリに pthreads がありますが、Windows では異なる API を使わなくてはなりません。このサンプルコードでは、`pthread_create()` を呼び出して定義されたワーカー処理の関数を実行しています。第 3 章「リクエスト・応答パターンの応用」では移植性のある API でこれをラップする方法を見ていきます。

ここでの「仕事」は単に 1 秒間停止しているだけです。ワーカーでは、他のノードと通信することを含めてあらゆる処理を行うことが出来ます。これはマルチスレッドサーバーが ØMQ ソケットやノードと同等である事を表しています。リクエスト・応答の経路は REQ-ROUTER-queue-DEALER-REP を経由します。

2.7 スレッド間の通知 (PAIR ソケット)

実際に ØMQ でマルチスレッドアプリケーションを作り始めると、スレッド同士の連携をどの様に行うかという問題に遭遇するでしょう。この時あなたはつい魔が差して sleep 命令を入れたり、マルチスレッドのテクニックであるセマフォやミューテックスを使用したくなるかもしれませんが、この時使うべき手段は ØMQ メッセージだけです。酔っぱらいとビール瓶の話を思い出して下さい。

それでは、3 つのスレッドでお互いに準備完了を通知するコードを書いてみましょう。この例ではプロセス内通信を行う PAIR ソケットを利用します。

mtrelay.c: マルチスレッド relay

```
// マルチスレッドリレー

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

static void *
step1 (void *context) {
    // step2 に接続して、準備完了を通知します
```

```
    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) {
    // step1 を開始する前にプロセス内通信で bind します
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step2");
    pthread_t thread;
    pthread_create (&thread, NULL, step1, context);

    // step1 からのメッセージを待ちます
    char *string = s_recv (receiver);
    free (string);
    zmq_close (receiver);

    // step3 に接続して、準備完了を通知します
    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 ();

    // step2 を開始する前にプロセス内通信で bind します
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step3");
    pthread_t thread;
    pthread_create (&thread, NULL, step2, context);

    // step2 からのメッセージを待ちます
    char *string = s_recv (receiver);
    free (string);
```

```
    zmq_close (receiver);

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

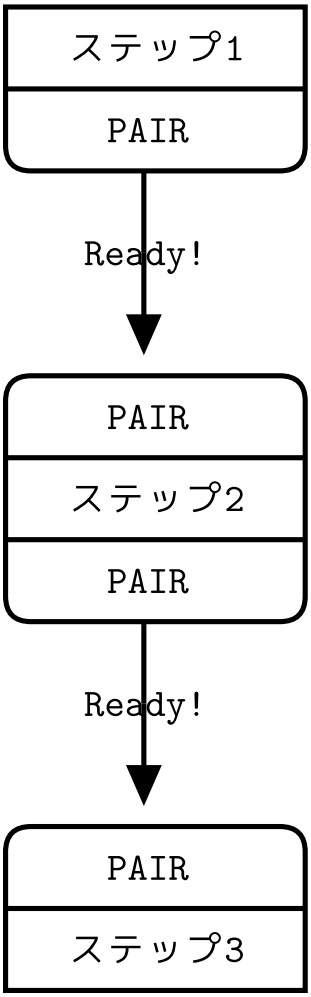


図 2.13　リレー

ØMQ のマルチスレッドの古典的なパターンは以下の通りです。

- 2 つのスレッドで共有しているコンテキストを利用してプロセス内通信を行います。
- 親スレッドはソケットを 1 つ作成し、プロセス内通信のエンドポイントとして bind し、コンテキストを渡して子スレッドを起動します。
- 子スレッドは 2 つ目のソケットを作成し、親スレッドのエンドポイントに接続します。そして親スレッドに準備完了を通知します。

マルチスレッドのコードでこのパターンを利用すると外部プロセスに拡張出来ないことに注意して下さい。プロセス内通信でペアソケットを利用すると、例えば片方のスレッドが構造的に相互依存している様な結合の強いアプリケーションを構築できます。低レイテンシを維持することがが極めて重要な場合にはこれは最適です。他のデザインパターンはスレッドは独自のコンテキストを持ち、IPC や TCP を経由して通信を行うので疎結合なアプリケーションに向いています。疎結合なスレッドは簡単に別プロセスに分離することができます。

PAIR ソケットを利用したサンプルコードはこれが初めてです。ここで PAIR ソケットを使った理由はなんだと思いますか? 他のソケットの組み合わせでも上手く動作するように見えますが、これらを通知のインターフェースとして利用すると副作用があります。

- 送信側で PUSH、受信側で PULL ソケットを使用するとします。これは単純に動作するように見えますが PUSH はメッセージを全ての受信者に配信する事を思い出して下さい。受信者が 2 つ居て、片方が起動していない場合、半分の通知が失われてしまうことになります。PAIR は排他的であり 2 つ以上の接続を許可しませんのでこの様な事が起こりません。
- 送信側で DEALER、受信側で ROUTER ソケットを使用するとします。ROUTER はメッセージをエンベロープに包装します。これはサイズが 0 の通知用メッセージであってもマルチパートメッセージに包まれてしまう事を意味します。メッセージの内容に関して関知せずソケットからデータを読み取らない場合は問題ありませんが、内容を持ったデータを送信する事になった場合、ROUTER から誤ったデータを読み取ってしまいます。DEALER はソケットを通知で利用する際には PUSH ソケットと同様にリスクがあります。
- 送信側で PUB、受信側で SUB ソケットを使用するとします。PUSH や DEALER やと異なり、この場合完全に正しくメッセージを配送することができますが、空のメッセージを受信できるようにサブスクライバ側の設定を行わなければならないのが面倒です。

これらの理由により、スレッド間の連携を行うためには PAIR ソケットを利用するのが最適です。

2.8 ノードの連携

ネットワーク上のノードを連携する際、PAIR ソケットでは上手く動作しません。スレッドとノードの戦略が異なっている部分の 1 つです。ノードは落ちてたり動いてたりするのに対し、スレッドは固定的であることが主な違いです。PAIR ソケットは接続相手と一時的に接続が切れた場合に自動的に再接続を行いません。

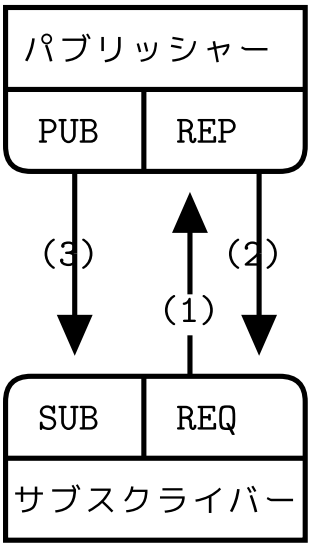


図 2.14　Pub-Sub Synchronization

スレッドとノードで異なる 2 つ目の重要な点は、一般的にスレッドの数は固定であるのに対してノードの数は可変である事です。以前見た気象情報サーバーとクライアントのシナリオで、起動時にデータを喪失しないよう確実に配信を行えるようノードの連携を行ってみましょう。

このアプリケーションは以下のように動作します。

- パブリッシャーは接続してくるサブスクライバー数を想定しているとします。これはとりあえずハードコーディングしていますが何でも構いません。
- パブリッシャーが起動すると、想定している全てのサブスクライバーが接続してくるまで待ちます。ここからがノードの連携処理で、各サブスクライバーはパブリッシャーに対してもう一方の SUB ソケットが準備完了していることを通知します。
- 全てのサブスクライバーが接続したらデータの配信を開始します。

このケースでは、サブスクライバーとパブリッシャーの同期を行うために REQ-REP ソケットを利用します。以下はパブリッシャーのコードです。

syncpub.c: 同期パブリッシャー

```
// 同期パブリッシャー

#include "zhelpers.h"
#define SUBSCRIBERS_EXPECTED  10  // 10 個のサブスクライバーを待ちます
```

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

    // クライアントと通信するソケット
    void *publisher = zmq_socket (context, ZMQ_PUB);

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

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

    // シグナルを受信するソケット
    void *syncservice = zmq_socket (context, ZMQ_REP);
    zmq_bind (syncservice, "tcp://*:5562");

    // サブスクライバと同期
    printf ("Waiting for subscribers\n");
    int subscribers = 0;
    while (subscribers < SUBSCRIBERS_EXPECTED) {
        // 同期リクエストを待つ
        char *string = s_recv (syncservice);
        free (string);
        // 同期応答を送信
        s_send (syncservice, "");
        subscribers++;
    }
    // 100 万回の更新を配信して終了
    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;
}
```

こちらはサブスクライバーです。

syncsub.c: 同期サブスクライバー

```c
// 同期サブスクライバー

#include "zhelpers.h"

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

    // まずサブスクライバーソケットで接続します
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5561");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);

    // 0MQ は速過ぎるので少し待ちます
    sleep (1);

    // 次にパブリッシャーと同期します
    void *syncclient = zmq_socket (context, ZMQ_REQ);
    zmq_connect (syncclient, "tcp://localhost:5562");

    // 同期リクエストを送信
    s_send (syncclient, "");

    // 同期応答を待ちます
    char *string = s_recv (syncclient);
    free (string);

    // 配信データを受信し、受け取った数を報告する
    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;
}
```

以下の Bash スクリプトで 10 個のサブスクライバーとパブリッシャーを起動します。

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

上手く行けば以下の出力が得られるはずです。

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

REQ/REP のやり取りが完了した時点では SUB ソケットの接続が完了しているとは限りません。転送方式にプロセス内通信を利用している場合を除き、接続完了の順序は保証されていません。そのため、SUB ソケットの接続後、REQ/REP 同期を行うまでの間に 1 秒間の強制的な sleep を行っています。

より確実なモデルにする為には、

- パブリッシャーは PUB ソケットを bind し、実際のデータではない「Hello」メッセージを送信します。
- サブスクライバーは SUB ソケットで接続を行い「Hello」メッセージを受信した時に REQ/REP ソケットペアを経由して受信に成功した旨をパブリッシャーに伝えます。
- そうして、パブリッシャーが全ての確認メッセージを確認した後に、実際のデータの配信を開始します。

2.9 ゼロコピー

ØMQ のメッセージ API はアプリケーションのバッファからコピーを行わずに直接送受信を行うことが出来ます。私達はこれをゼロコピーと呼んでいて、アプリケーションのパフォーマンスを改善する事ができます。

巨大なメモリブロックを頻繁に送信する様な特定のケースでは、ゼロコピーを利用することを検討すると良いでしょう。ただし、小さいメッセージや送受信が低頻度であればゼロコピー

はコードが複雑になるだけで有意な効果を得られないかもしれません。

ゼロコピーを行う為には zmq_msg_init_data() 関数に malloc() やその他のメモリアロケータで確保したメモリブロックを渡してメッセージオブジェクトを生成します。そして、そのメッセージを zmq_msg_send() に渡して送信します。メッセージオブジェクトを生成する際、メッセージの送信完了時にメモリを開放する為の関数も同時に渡します。以下はヒープ上に 1,000 バイトのバッファを確保する単純なサンプルコードです。

```
void my_free (void *data, void *hint) {
    free (data);
}
// Send message from buffer, which we allocate and 0MQ will free for us
zmq_msg_t message;
zmq_msg_init_data (&message, buffer, 1000, my_free, NULL);
zmq_msg_send (&message, socket, 0);
```

メッセージの送信後に zmq_msg_close() を呼んではならないことに注意して下さい。libzmq はメッセージを送信した後は自動的にこれを行います。

受信時にゼロコピーを行う方法はありません。ØMQ の受信バッファは望む限りいつまでも残しておくことができますが、ØMQ はアプリケーションのバッファに直接書き込む事はありません。

ゼロコピーは ØMQ のマルチパートメッセージの送信時に適しています。従来のメッセージングでは複数のバッファを一つのバッファーにまとめて送信する必要がありました。これはデータをコピーしなければならないことを意味しています。ØMQ では個別のメッセージフレームを元にしたマルチパートメッセージを送信することが可能です。アプリケーションから見ると一連の送受信呼び出しを行っているように見えますが、内部的には 1 度のシステムコール呼び出しでマルチパートメッセージを呼び出してネットワークに送信するため、非常に効率的です。

2.10 Pub-Sub メッセージエンベロープ

pub-sub パターンでは、フィルタの対象であるキーをエンベロープと呼ばれるメッセージフレームに分けて送信することができます。pub-sub エンベロープを利用する場合は明示的にこのメッセージフレームを生成して下さい。この機能は任意ですので以前の pub-sub のサンプルコードでは利用しませんでした。pub-sub エンベロープを利用するには、ちょっとしたコードを追加する必要があります。実際のコードを見れば直ぐに理解できると思いますが、キーとデータが別々のメッセージフレームに別けているだけです。

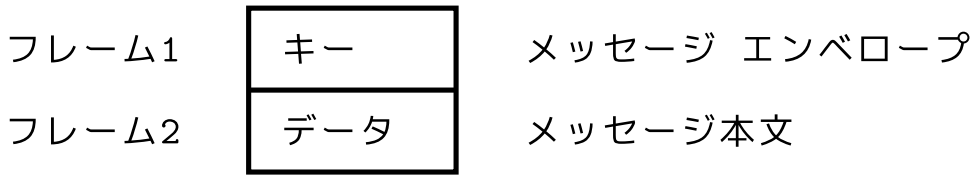


図 2.15 Pub-Sub Envelope with Separate Key

サブスクライバーは前方一致でフィルタリングを行っている事を思い出して下さい。誤ってデータと一致しないようにキーとデータをどうやって区切るのか、という疑問を抱くのは当然のことです。最適な解決方法はエンベロープを使うことです。フレーム境界を超えて一致することはありません。以下は pub-sub エンベロープを利用する最小のサンプルコードです。パブリッシャーは 2 種類のタイプのメッセージ (A と B) を送信しています。

エンベロープはメッセージ種別を保持しています。

psenvpub.c: Pub-Sub エンベロープパブリッシャー

```
// Pub-sub エンベロープパブリッシャー
// zhelpers.h ファイルに s_send() と s_sendmore() が定義されています

#include "zhelpers.h"

int main (void)
{
    // コンテキストとパブリッシャーの準備
    void *context = zmq_ctx_new ();
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5563");

    while (1) {
        // エンベロープとコンテンツを書き込みます
        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");
        sleep (1);
    }
    // ここの処理は実行されません
    zmq_close (publisher);
    zmq_ctx_destroy (context);
    return 0;
}
```

サブスクライバーはメッセージ種別 B のみを受信します。

psenvsub.c: Pub-Sub エンベロープサブスクライバー

```
// Pub-Sub エンベロープサブスクライバー

#include "zhelpers.h"

int main (void)
{
    // コンテキストとサブスクライバーの準備
    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) {
        // エンベロープを読み込みます
        char *address = s_recv (subscriber);
        // コンテンツを読み込みます
        char *contents = s_recv (subscriber);
        printf ("[%s] %s\n", address, contents);
        free (address);
        free (contents);
    }
    // ここの処理は実行されません
    zmq_close (subscriber);
    zmq_ctx_destroy (context);
    return 0;
}
```

2 つのプログラムを動作させると、サブスクライバー側では以下の結果が得られるでしょう。

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

このサンプルコードではキーとデータを含むマルチパートメッセージの取捨選択を行っています。マルチパートメッセージの一部を取得する必要はありません。複数のパブリッシャーから更新情報を受け取っている場合、メッセージの送信元を識別したいと思うかもしれません。そんな時は 3 つで構成されるマルチパートメッセージを作成しください。

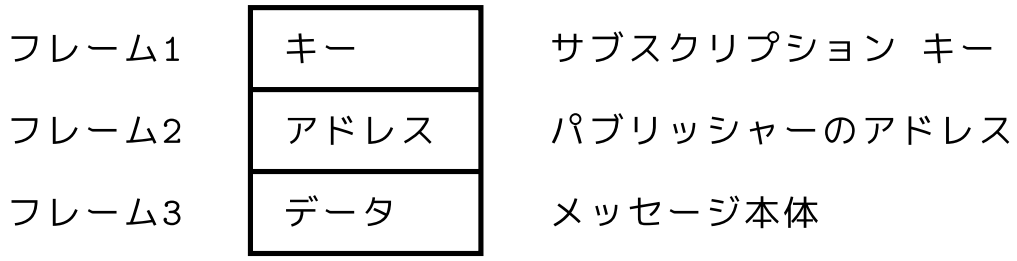


図 2.16　Pub-Sub Envelope with Sender Address

2.11 満杯マーク

プロセスからプロセスに大量のメッセージを送信する際、メモリが貴重な資源であることに気がつくでしょう。問題を理解し対策を講じない限り、プロセスのバックログは膨れ上がり、数秒間の遅延が発生引き起こすでしょう。

プロセス A からプロセス B に対して短期間に大量のメッセージを送信した場合を想像して下さいプロセス B は突然 CPU 負荷やガーベジコレクションなどの理由により高負荷に陥り、メッセージを処理出来なくなったとします。重いガーベジコレクションは数秒かかるでしょうし、深刻な問題が発生した場合はもっと時間がかかるかもしれません。プロセス A が必死にメッセージを送り続けた場合メッセージはどうなるでしょうか。幾つかは B 側のネットワークバッファにとどまります。幾つかはイーサネット回線上にとどまるでしょう。幾つかは A 側のネットワークバッファにとどまります。残りの部分は後ほど迅速に再送できるように A 側のメモリにとどまります。何らかの対応を行わなければメモリ不足に陥り、クラッシュしてしまうでしょう。

これは、メッセージブローカーが持つ古典的な問題と同様です。この問題の痛い所は、プロセス B の障害によってプロセス A が制御不能に陥ってしまうことです。

解決方法のひとつは問題を上流に伝えることです。つまり、プロセス A に対して「送信を止めろ」という旨を何らかの方法で伝えてやります。これはフロー制御と呼ばれています。この方法はもっともらしく見えますが、例えば Twitter のタイムラインで全世界に対して「つぶやきを止めろ」という事は妥当でしょうか?

フロー制御は上手く場合もありますが、上手くいかない事もあります。通信レイヤはアプリケーションレイヤに対して「止めろ」というようなことは出来ません。例えば地下鉄は「仕事を始めるのを 30 分送らせてくれ」といった事を行ってきますが、迷惑な話です。メッセージングシステムにおけるこの解決方法はバッファサイズに上限を設定し、この上限に達した場合に合理的な動作を行う事です。あるケースではメッセージを投げ捨ててしまう方が良い場合もあるし、ある時は待つことが最良の戦略である場合もあります。

ØMQ は HWM(満杯マークという) 概念を用いてパイプの容量を定義します。各コネクションはソケットの外部か内部に個別のパイプを持っていて、HWM は送信時と受信時にソケット種別に応じて制限を掛けます。PUB, PUSH などのソケットは送信バッファのみを持っていて、SUB, PULL, REQ, REP などのバッファは受信バッファを持っています。 DEALER, ROUTER, PAIR などのバッファに関しては送信と受信の両方バッファを持っています。

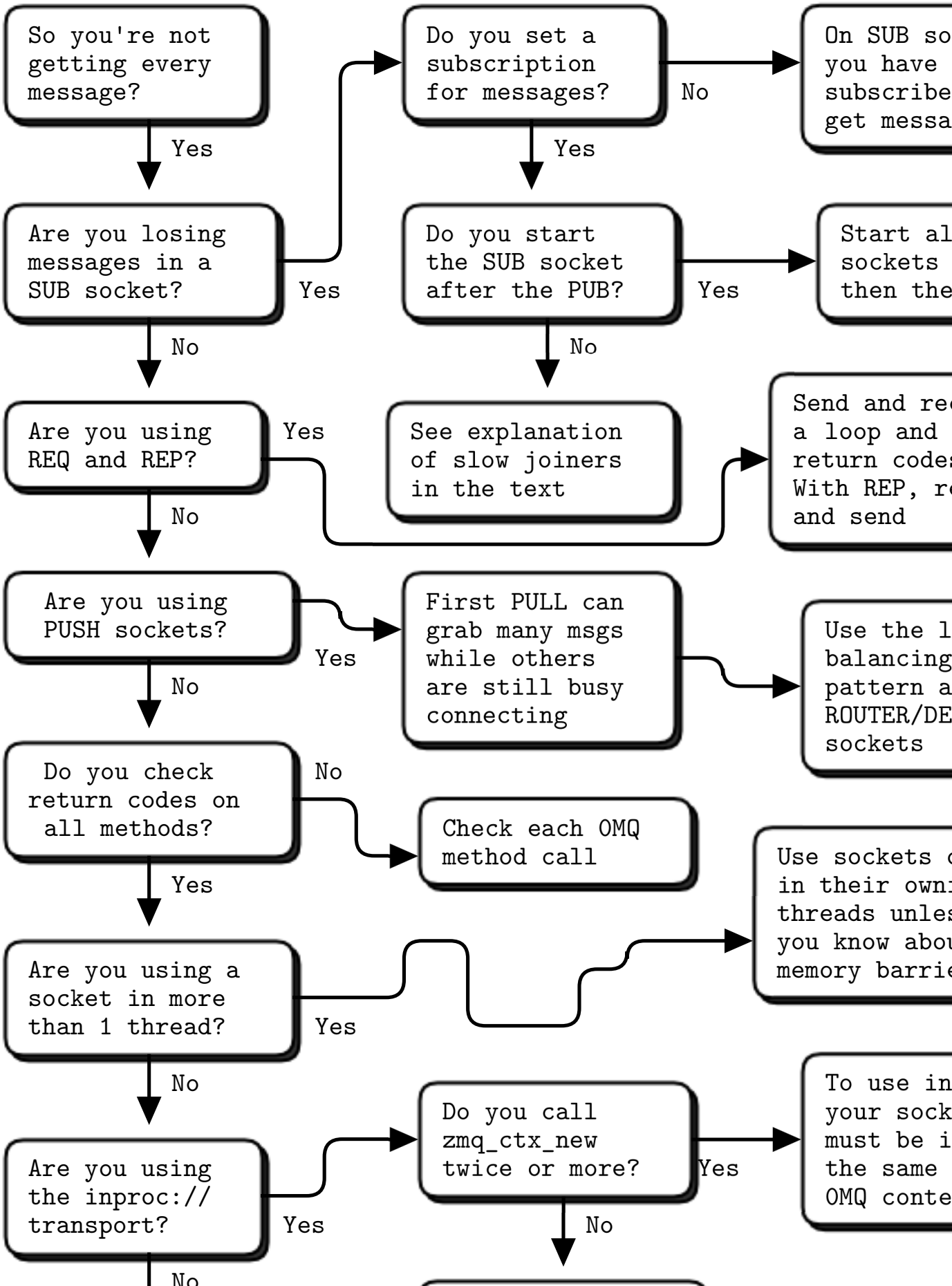
ØMQv2.x では、既定で HWM は無制限でした。これは単純でしたが大量配信を行うパブリッシャーにとって致命的でした。 ØMQ v3.x では規定で 1,000 という合理的な値が設定されています。もしあなたがまだ ØMQ v2.x を利用しているのなら、常に HWM に 1,000 あるいはサブスクライバーのパフォーマンスに適切な値を設定しておいたほうが良いでしょう。

ソケットが HWM の上限に達した場合ソケット種別に応じてメッセージをブロックするか捨てるかが決まります。HWM の上限に達した際、PUB と ROUTER ソケットはメッセージを捨て、その他のソケット種別の場合ブロックします。プロセス内通信を行う場合、送信側と受信側で同じバッファを共有していますので両サイドで設定した HWM の合計が実際の HWM の上限となります。

最後に、HWM は正確ではありません。デフォルトでは 1,000 個までのメッセージを受け取るはずですが、libzmq はキューとして実装されているので実際のバッファーサイズはこれより半分程度小さいことがあります。

2.12 メッセージ喪失問題の解決方法

ØMQ でアプリケーションを開発していると、受信するはずメッセージが喪失してしまうという問題に遭遇するでしょう。そこで私達はよくあるメッセージ喪失問題の解決フローをまとめました。


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
No
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 so
you have
subscribe
get messa
Start al
sockets
then the
Send and re
a loop and
return code
With REP, r
and send
Use the l
balancing
pattern a
ROUTER/DE
sockets
Use sockets
in their own
threads unle
you know abo
memory barri
To use in
your sock
must be i
the same
OMQ conte

この図は以下のことを表しています。

- SUB ソケットは、zmq_setsockopt() で ZMQ_SUBSCRIBE を設定しなければ更新メッセージを受信できません。これは意図的な仕様です、理由は更新メッセージはプレフィックスでフィルタリングを行っているため、既定のフィルタ「」(空文字列) では全てを受信してしまうからです。
- SUB ソケットが PUB ソケットに対して接続を確立した後に、PUB ソケットがメッセージを送信を開始した場合でもメッセージを失ってしまいます。これが問題になる場合、まず最初に SUB ソケットを開始して、その後、PUB ソケットで配信するようなアーキテクチャを構成しなければなりません。
- SUB ソケットと PUB ソケットを同期させる場合でもメッセージを喪失してしまう可能性があります。これは、実際に接続が行われるまで内部キューが作成されていないという事実によるものです。bind と接続の方向性は切り替えることができますので PUB ソケットから接続を行った場合、さらに幾つかの期待通りに動作しない場合があるでしょう。
- REP ソケットと REQ ソケットを利用していて、送信、受信、送信、受信という順番で同期が行われていない場合、ØMQ はメッセージを無視ししてエラーを報告するでしょう。この場合でも、メッセージの喪失した様に見えます。REQ や REP ソケットを利用する場合は、送信/受信の順番を常にコードの中で固定し、エラーを確認するようにしてください。
- PUSH ソケットを利用してメッセージを分配する場合、最初に接続した PULL ソケットは不公平な数のメッセージを受け取るかもしれません。メッセージの分配は正確には PULL ソケットが接続に成功して数ミリ秒掛かってしまうからです。少ない配信頻度でもっと正確な分配を行いたい場合は ROUTER/DEALER のロードバランシングパターンを利用して下さい。
- 複数のスレッドでソケットを共有している場合...、そんなことをしてはいけません、これをやるとランダムで奇妙なクラッシュを引き起こしてしまうでしょう。
- プロセス内通信を行っている場合、共有したひとつのコンテキストで両方のソケットを作成してください。そうしないと接続側は常に失敗します。またプロセス内通信は TCP のような非接続通信方式と異なりますので最初に bind を行なってから接続してください。
- ROUTER ソケットで不正な形式の identity フレームを送信してしまったり、identity フレームを送信し忘れたりしてしまうようなアクシデントによりメッセージを喪失しやすくなります。一般的に、ZMQ_ROUTER_MANDATORY オプションを ROUTER ソケットに設定することは良いアイディアですが、送信 API 呼び出しの返り値を確認す

るようにしてください。

- 最後に、なぜうまく行かないのか判断できない場合、問題を再現させる小さなテストコードを書いてコミュニティで質問してみるとよいでしょう。

第 3 章

リクエスト・応答パターンの応用

「第 2 章 - ソケットとパターン」では ØMQ を使った一連の小さなアプリケーションを開発しながら ØMQ の新しい側面を探って来ました。この章では引き続き同じような方法で ØMQ のコアとなるリクエスト・応答パターンの応用方法について探っていきます。

この章では、

- どの様にリクエスト・応答のメカニズムが動作するか
- REQ、REP、DEALER、ROUTER などのソケットを組み合わせる方法
- どの様に ROUTER ソケットが動作するか、とその詳細
- 負荷分散パターン
- 負荷分散メッセージブローカーを構築する
- 高レベルリクエスト・応答サーバーの設計
- 非同期なルリクエスト・応答サーバーの構築
- 内部ブローカーのルーティング例

3.1 リクエスト・応答のメカニズム

これまでマルチパートメッセージについて簡単に学んできました。ここでは応答メッセージエンベロープという主要なユースケースについて見ていきます。エンベロープはデータ本体に触れること無くデータに宛先を付けてパッケージ化する方法です。宛先をエンベロープに分離することで、メッセージ本体の構造に関わらず宛先を読み書き、削除を行うことの出来る汎用的な API や仲介者を構築することが可能になります。

リクエスト・応答パターンでは、応答する際の返信アドレスをエンベロープに記述します。

これにより ØMQ ネットワークは状態を持たずにリクエスト・応答の一連のやりとり実現出来ます。

REQ、REP ソケットを利用する際、わざわざエンベロープを参照する必要はありません。これらはソケットが自動的に行なってくれます。しかしここはリクエスト・応答パターンの面白い所ですし、とりわけ ROUTER ソケットのエンベロープついて学んでおいて損は無いでしょう。これからそれらを一歩一歩学んでいきます。

3.1.1 単純な応答パケット

リクエスト・応答のやり取りはリクエストメッセージとそれに対する応答メッセージかで成立します。単純なリクエスト・応答パターンでは各リクエストに対して 1 回の応答を行います。もっと高度なパターンだと、リクエストと応答は非同期で行われます。しかしながら応答エンベロープはいつも同じように動作します。

ØMQ の応答エンベロープは正確には 0 以上の返信先アドレス、続いて空のフレーム (エンベロープの区切り)、そしてメッセージ本体 (0 以上のフレーム) で構成されます。エンベロープは複数のソケット動作する中で生成されます。これをもっと具体的に見ていきます。

「Hello」というメッセージを REQ ソケットで送信する場合を考えます。REQ ソケットはアドレスを持たない空の区切りフレームと「Hello」というメッセージフレームから構成される最も単純な応答エンベロープを生成します。これは 2 つのフレームで構成されたメッセージです。

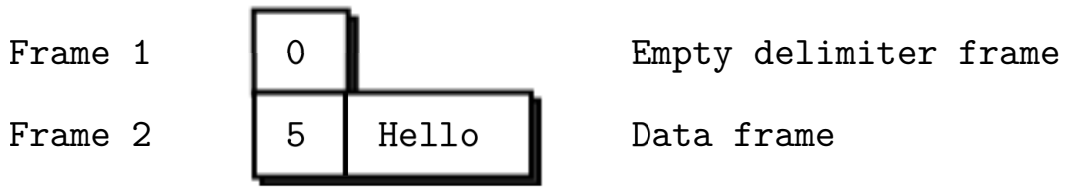


図 3.1　最小の応答エンベロープ

REP ソケットは区切りフレームを含む全体のエンベロープを退避します。そして残りの「Hello」という文字列がアプリケーションに渡されます。最初の Hello World のサンプルコードはリクエスト・応答のエンベロープは内部的に処理されていますのでアプリケーションでこれを意識する事はありません。

hwclient と hwserver の間を流れるネットワークデータを監視してみると、全てのリクエストと応答は空のフレームとメッセージ本体の 2 つのフレームで構成されていることを確認できるでしょう。この様に単純なリクエスト・応答のやり取りではエンベロープは付加されていません。しかし、ROUTER と DEALER ソケットの処理を監視すると、エンベロープに宛先が付

加されているのを確認できるはずです。

3.1.2 拡張された応答エンベロープ

それでは、REQ-REP ソケットペアを拡張した ROUTER-DEALER プロキシで応答エンベロープにどの様な影響があるか見て行きましょう。これは第 2 章の「ソケットとパターン」で既に見た、拡張されたリクエスト・応答パターンと同じ仕組みで、プロキシを幾つでも挿入することが出来ます。

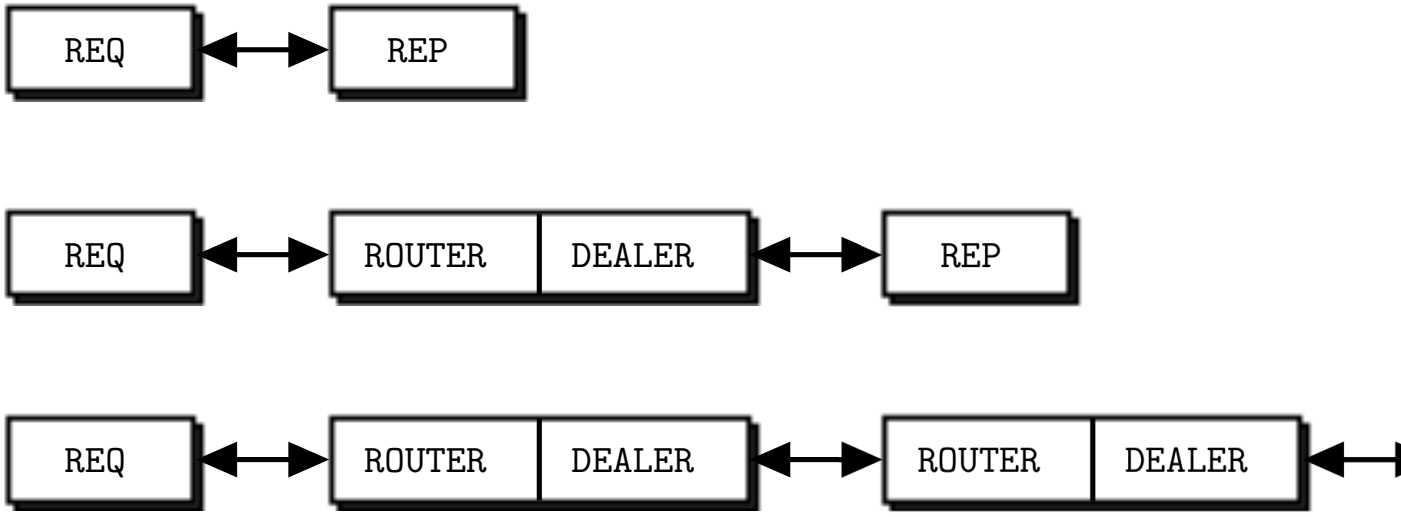


図 3.2　拡張されたリクエスト・応答パターン

プロキシは擬似コードで以下の様に動作します。

```
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 ソケットは他のソケットとは異なり、全ての接続をトラッキングして接続元を通知します。メッセージを受信すると、メッセージの頭に接続 ID を頭に付与する事で接続元を通知します。この ID はアドレスとも言われ、コネクションに対するユニークな ID になりま

す。。ROUTER ソケット経由でメッセージを送信すると、まずこの ID フレームが送信されます。

`zmq_socket()` の man ページには以下のように書かれています。

> “ ZMQ_ROUTER ソケットがメッセージを受信すると、メッセージフレームの先頭に元々の接続 ID を追加します。受信したメッセージは全ての接続相手の中から均等にキューイングします。ZMQ_ROUTER ソケットから送信を行う時、最初のメッセージフレームの ID ををを削除してメッセージをルーティングします。

歴史的な情報ですが、ØMQ v2.2 以前はこの ID に UUID を利用していましたが、ØMQ 3.0 以降からは短い整数を利用しています。これはネットワークパフォーマンスに少なからず影響を与えますが、多段のプロキシを利用している場合は影響は微々たるものでしょう。最も大きな影響は libzmq が UUID ライブラリに依存しなくなったことくらいです。

ID は理解しにくい概念ですが、ØMQ のエキスパートになる為には不可欠です。ROUTER ソケットはコネクション毎にランダムな ID を生成します。ROUTER ソケットに対して 3 つの REQ ソケットが接続したとすると、それぞれ異なる 3 つの ID が生成されるでしょう。

引き続き動作の説明を続けると、REQ ソケットが 3 バイトの ID「ABC」を持っていたとすると、内部的には、ROUTER ソケットは「ABC」というキーワードで検索して TCP コネクションを得ることのできるハッシュテーブルを持っていることを意味します。

ROUTER ソケットからメッセージを受信すると 3 つのフレームを受け取ることになります。

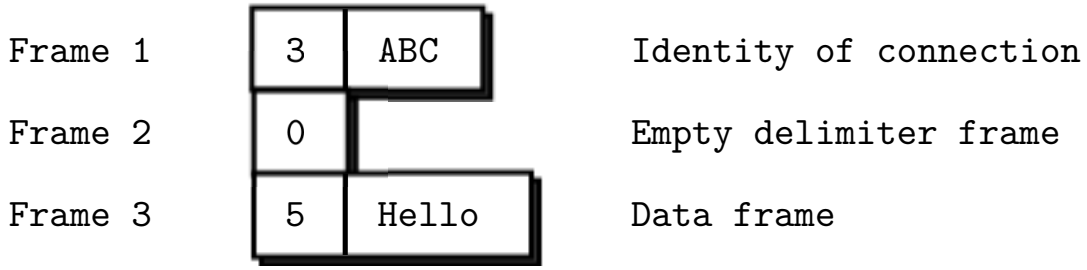


図 3.3　アドレス付きのリクエスト

プロキシのメインループでは「ソケットから読み取ったメッセージを他の相手に転送する処理」を繰り返していますので、DEALER ソケットからは 3 つのフレームが出ていく事になります。ネットワークトラフィックを監視すると、DEALER ソケットから REP ソケットに向けて 3 つのフレームが飛び出してくるのを確認できるでしょう。REP ソケットは新しい応答アドレスを含むエンベロープ全体を取り除き、「Hello」というメッセージをアプリケーションに返します。

繰り返しになりますが、REP ソケットは同時にに 1 回のリクエスト・応答のやりとりしか行うことが出来ません。複数のリクエストや応答をいっぺんに送ってしまうと、エラーが発生しますので、送受信の順序を守って 1 つずつ行なって下さい。

これで、応答経路をイメージできるようになったはずです。hwserver が「World」というメッセージを返信する時、REP ソケットは退避していた、エンベロープを再び付加して 3 フレームのメッセージを DEALER ソケットに対して送信します。

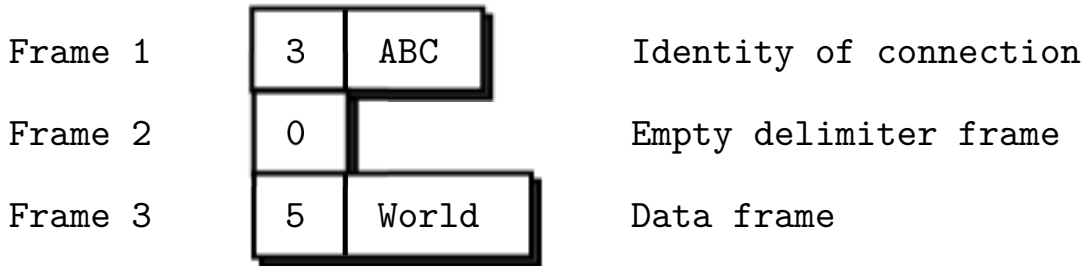


図 3.4　アドレス付きの応答

ここで、DEALER は 3 つのフレームを受信し、全てのフレームは ROUTER ソケットに渡されます。ROUTER は最初のメッセージフレームを読み取り、ABC という ID に対応する接続を検索します。接続が見つかったら、残りの 2 フレームをネットワークに送り出します。

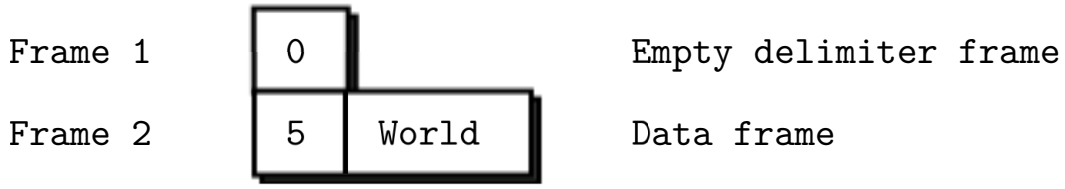


図 3.5　最小の応答エンベロープ

REQ ソケットはメッセージを受信し、最初のフレームが空の区切りフレームであることを確認し、これを破棄します。そして、「World」というメッセーがアプリケーションに渡され、ØMQ を始めてみた時の驚きとともに表示されます。

3.1.3 なにかいい事あるの?(What's This Good For?)

正直に言うと、素のリクエスト・応答パターンや拡張したリクエスト・応答パターンには幾つかの制限があります。ひとつ例を挙げると、サーバー側のアプリケーションのバグに起因したクラッシュなどの一般的な障害から回復する簡単な方法がありません。これは第 4 章の「信頼性のあるリクエスト・応答パターン」で詳しく解説します。

さておき、4 つのソケットがどの様な方法でエンベロープを扱い、お互いに会話するかを理解しておくことは大変有用です。これまで、ROUTER がどの様に応答エンベロープを利用してクライアントの REQ ソケットに応答するかを見てきましたので、簡単にまとめておきます。

- ROUTER がメッセージを受け取ると、接続元である相手を ID として記録します。
- 接続相手は、ID をキーとしたハッシュテーブルで保持します。
- ROUTER はメッセージの最初のフレームを ID として非同期でルーティングします。

ROUTER ソケットはエンベロープ全体については関知しません。例えば区切りフレームについては何も知りません。メッセージを送信する為の接続先を知るために ID フレームのみを参照します。

3.1.4 リクエスト・応答ソケットのまとめ

まとめると、

- REQ ソケットはメッセージデータの先頭に空の区切りフレームを付けてネットワークに送信します。REQ ソケットは同期的に、ひとつのリクエストを送信したら応答が返ってくるまで待つ必要があります。REQ ソケットが通信できる相手は同時に 1 つだけです。もし、複数の相手に接続した場合リクエストは分散され、同時に 1 つの相手からの応答を期待します。
- REP ソケットは全ての ID フレームと空の区切りフレームを読み込み、退避します。そして残りのフレームがアプリケーションに渡されます。REP ソケットも同期的であり、同時に 1 つの相手としか通信を行いません。REP ソケットに複数の相手が接続してきた場合は接続相手からの要求メッセージを均等に受信し、常に受信した相手に対して応答を返します。
- DEALER ソケットは応答エンベロープやマルチパートメッセージ処理に関しては無関心です。DEALER ソケットは PUSH ソケットと PULL ソケットの組み合わせの様に非同期です。メッセージは全ての接続相手に対して分散して送信し、受信時は全ての接続相手から均等にキューイングを行います。
- ROUTER ソケットは DEALER ソケットと同様に、応答エンベロープに関しては無関心です。このソケットはメッセージを受信すると、接続元を特定する ID を最初のフレームに追加します。逆に、このソケットから送信する際、最初のフレームの ID を参照して送信先を決定します。ROUTERS ソケットも非同期です。

3.2 リクエスト・応答の組み合わせ

リクエスト・応答ソケットにはそれぞれ異なる振る舞いをする 4 つのソケットがあり、これらの簡単な利用方法や、拡張されたリクエスト・応答パターンの利用方法を見てきました。これらのソケットを活用することで、多くの問題を解決するブロックを構築できるでしょう。

正しいソケットの組み合わせは以下の通りです。

- REQ から REP
- DEALER から REP
- REQ から ROUTER
- DEALER から ROUTER
- DEALER から DEALER
- ROUTER から ROUTER

And these combinations are invalid (and I'll explain why): そして以下の組み合わせは不正です。(理由は後ほど説明します)

- REQ から REQ
- REQ から DEALER
- REP から REP
- REP から ROUTER

ここでは、意味を覚えるためのヒントを幾つか紹介します。DEALER は非同期になった REQ ソケットの様なもので、ROUTER は REP ソケットの非同期版と言えます。REQ ソケットを使う場合のみ DEALER ソケットを使うことが出来、メッセージのエンベロープを読み書きする必要があります。REP ソケットを利用する場合のみ、ROUTER を配置することが出来、ID を管理する必要があります。

REQ ソケットと DEALER ソケット側の事を「クライアント」、REP ソケットと ROUTER ソケット側の事を「サーバー」として見ることができます。多くの場合、REP ソケットと ROUTER ソケットで bind を行うでしょうし、REQ ソケットと DEALER ソケットが接続を行います。いつもこの様に単純だとは限りませんが、大体こんな風に覚えておけば良いでしょう。

3.2.1 REQ と REP の組み合わせ

既に私達は REQ クライアントが REP サーバーと通信する仕組みについて見てきましたが, ここでは、ちょっと別の側面を見て行きましょう。メッセージフローは REQ クライアントが

開始する必要があります。REP サーバーまずリクエストを受け取らなければ、REQ クライアントに対して通信を行うことは出来ません。技術的にそれは不可能であり、もしこれをやろうとすると、API は EFSM エラーを返します。

3.2.2 DEALER と REP の組み合わせ

それでは REQ クライアントを DEALER ソケットに置き換えてみましょう。これは複数の REP サーバーと通信可能な非同期なクライアントを実現できます。例えば「Hello World」クライアントを DEALER で書き直した場合、応答を待たずに複数の「Hello」リクエストを送信可能です。

DEALER ソケットから REP ソケットに対して通信行う場合、REQ ソケットから送信が行われたように正確にエミュレートしする必要があります。そうしなければ REP ソケットは不正なメッセージとみなして破棄してしまうでしょう。すなわち、以下のように送信する必要があります。

- MORE フラグをセットして、空のフレームを送信
- 続いてメッセージ本体を送信

そして受信時は、

- 受信した最初のフレームが空でなければ、メッセージ全体を破棄します。
- 空フレームに続くフレームをアプリケーションに渡します。

3.2.3 REQ と ROUTER の組み合わせ

REQ ソケットを DEALER ソケットに置き換えたのと同様に、REP ソケットを ROUTER ソケットに置き換える事が出来ます。これは複数の REQ クライアントに対して同時に通信可能な非同期なサーバーを実現できます。例えば「Hello World」サーバーを ROUTER ソケットで書き直した場合、複数の「Hello」リクエストを並行に処理することが可能です。これは既に第 2 章の「Sockets and Patterns mtserver」の例で見てきました。

ROUTER ソケットは明確に 2 つの用途で利用できます。

- フロントエンドとバックエンドソケットの間でメッセージを中継するプロキシとして
- メッセージ受信するアプリケーションとして

最初のケースでは ROUTER ソケットは ID フレームを含む全てのフレームを受信し、盲目的にメッセージを通過させます。2 番目のケースでは ROUTER ソケットは応答エンベロープ

の形式を意識する必要があります。相手が REQ ソケットだとすると、ROUTER ソケットはまず ID フレームと空フレームを受信し、それからデータフレームを受け取ります。

3.2.4 DEALER と ROUTER の組み合わせ

そして、REQ ソケットと REP ソケットの組み合わせを DEALER ソケットと ROUTER ソケットという強力な組み合わせに置き換えることが可能です。これは非同期なクライアントと、非同期なサーバーを実現可能で、両側でメッセージエンベロープの形式を意識する必要があります。

なぜなら、DEALER ソケットと ROUTER ソケットの両側で自由なメッセージフォーマットを利用できるので、これらを安全に扱いたい場合は少し慎重にプロトコル設計を行う必要があります。最低限、あなたは REQ/REP ソケットの応答エンベロープをエミュレートするかどうかを決める必要があります。この決定は、応答を必ず返す必要があるかどうかに関わってきます。

3.2.5 DEALER と DEALER の組み合わせ

REP ソケットを ROUTER ソケットに置き換える事が可能ですが、通信相手が 1 つの場合に限り、REP ソケットを DEALER ソケットに置き換えることも可能です。

REP ソケットを DEALER ソケットで置き換えた場合、ワーカーは完全に非同期に応答を返すようになるでしょう。対価として、応答エンベロープを自分で管理して正しく取得する必要がする必要があります。そうしなければまったく動作しません。後ほど実際に動作する例を見ていきますが、この DEALER ソケットと DEALER ソケットの組み合わせはトリッキーなパターンの一つであり、これが必要となるケースは稀でしょう。

3.2.6 ROUTER と ROUTER の組み合わせ

これは完全な N 対 N 接続のように思うかもしれませんが、これは最も扱いにくい組み合わせです。ØMQ を使いこなせる様になるまで、この使い方は避けたほうが無難です。第 4 章信頼性のあるリクエスト・応答パターン「」ではこれを利用したフリーランス・パターンをという例を見ていきます。また、第 8 章「分散コンピューティング・フレームワーク」では P2P 機能を設計するする為の DEALER 対 ROUTER 通信の代替としてとして紹介します。

3.2.7 不正な組み合わせ

クライアントとクライアント、サーバーとサーバーで接続しようとする試みは、ほとんどの場合上手く動作しません。しかし、ここでは曖昧な警告で終わらせるのではなく具体的に説明しておきます。

- REQ と REQ の組み合わせ: 両者ともメッセージの送信を開始しようとします。そしてこれが正しく動作するのは、両者がぴったり同時にリクエストを送信した場合のみです。これについて考えると頭痛がします。
- REQ と DEALER の組み合わせ: 理論上これを行うことは可能ですが、2 つ目の REQ を追加した時に破綻します。なぜなら DEALER には元々の相手に応答を送信する機能が存在しないからです。従って、REQ ソケットは混乱してしまい誤ったクライアントにメッセージを返してしまう可能性があります。
- REP と REP の組み合わせ: お互いに最初のメッセージを待ち続けるでしょう。
- REP と ROUTER の組み合わせ: 相手が REP ソケットだという事が判っている場合、ROUTER ソケットは理論上対話を開始することが可能であり、正しい形式のリクエストを送信することが出来ます。それは DEALER と ROUTER の組み合わせと比べてややこしいだけで良いことは一つもありません。

ØMQ の正しいソケットの組み合わせについて一貫して言えることは、常にどちらかがエンドポイントとして bind し、もう片方が接続を行うという事です。なお、どちらが bind を行いどちらが接続を行っても構わないのですが、自然なパターンに従うのが良いでしょう。「存在が確か」である事を期待される側が bind を行い、サーバーやブローカー、パブリッシャーとなるでしょう。一方、「現れたり消えたり」する側が接続を行い、クライアントやワーカーとなるでしょう。これを覚えておくと、より良い ØMQ アーキテクチャを設計するのに役立ちます。

3.3 ROUTER ソケットの詳細

ROUTER ソケットについてもう少し詳しく見ていきましょう。これまでに、個別のメッセージを特定の接続にルーティングする機能について見てきました。ここでは、コネクションの識別方法についての詳細と、ROUTER が何を行い、どんな時にメッセージを送信できないかについて説明します。

3.3.1 ID とアドレス

ØMQ における ID は ROUTER ソケットが他のソケットへのコネクションを識別するための概念です。もっと大ざっぱに言うと、ID は応答エンベロープのアドレスとして利用されます。多くの場合、この ID は ROUTER がハッシュテーブルの検索に利用するための局所的なものです。ID はネットワークのエンドポイント「tcp://192.168.55.117:5670」の様な物理的なアドレスとは独立した論理的なアドレスになります。

ROUTER ソケットはメッセージの全てを読み込める為、ROUTER ソケットを利用した場合のみ接続相手が送信した接続 ID を知ることが出来ます。これを利用して、必要に応じて論理的なアドレスから ID に変換するハッシュテーブルを構築することが出来ます。

これとは反対に、ROUTER 側から接続を行う場合も同様です。そして、この ID の代わりに論理的な ID を強制的に利用する事も可能です。zmq_setsockopt の man ページではこれを「ソケット ID の設定」と呼んでいます。これは以下の様に動作します。

- アプリケーションは bind や接続を行う前にソケット (DEALER、もしくは REQ) に対して ZMQ_IDENTITY オプションを設定します。
- 通常、bind 済みの ROUTER ソケットに対して接続が行われます。しかし、ROUTER ソケットは接続しに行く事も可能です。
- 接続時、接続相手は ROUTER ソケットに対して「この接続 ID を利用してね」と伝えます。
- 接続相手がこれを伝えなかった場合、ROUTER 側でランダムな接続 ID を生成します。
- ROUTER ソケットは受け取ったメッセージに対して論理アドレスを付加します。
- そして ROUTER ソケットから出ていくメッセージには ID フレームが付加されていることを期待します。

以下のサンプルコードは、2 つのソケットでルーターソケットに対して接続を行い、片方のソケットに「PEER2」という論理アドレスを設定する単純な例です。

identity.c: ID チェック

```
// リクエスト応答 ID のデモ

#include "zhelpers.h"

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

```
    void *sink = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (sink, "inproc://example");

    // 0MQ が自動で ID を設定します
    void *anonymous = zmq_socket (context, ZMQ_REQ);
    zmq_connect (anonymous, "inproc://example");
    s_send (anonymous, "ROUTER uses a generated UUID");
    s_dump (sink);

    // ID を明示的に設定する場合
    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;
}
```

このプログラムは以下の出力を行います。

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

3.3.2 ROUTER のエラー処理

ROUTER ソケットはメッセージを送信できない場合に黙って捨てるという荒っぽい挙動を行います。これは実際のコードでは合理的な動作ですがデバッグが難しくなるのが難点です。

この最初のフレームに ID を含めて送信する方式は、注意しなければ誤った結果が得られたり、ROUTER は黙ってメッセージを捨てるので混乱してしまうかもしれません。

ØMQ v3.2 以降、このエラーを検知できる ZMQ_ROUTER_MANDATORY ソケットオプションが追加されました。ROUTER ソケットにこれを設定すると、ルーティング出来ない ID に対して送信した場合にソケットが EHOSTUNREACH エラーを通知します。

3.4 負荷分散パターン

それではコードを見て行きましょう。これから REQ ソケットや DEALER ソケットで ROUTER ソケットに接続する方法を見ていきます。この 2 つのパターンは同じく負荷分散パターンというロジックに従っています。単純な応答を行うのではなく、意図的にルーティングを行う例としてこのパターンは初めて紹介することになります。

負荷分散パターンは極めて一般的であり、この本の中で何度か出てくるでしょう。PUSH と DEALER ソケットとは異なり、負荷分散は単純なラウンドロビンを利用しますが、ラウンドロビンはタスクの処理時間が均等でない場合に非効率になる事があります。

郵便局で例えてみましょう。、郵便局の同じ窓口に切手を買いに来た人々 (速いトランザクション) と新規口座を開設しに来た人々 (非常に遅いトランザクション) が並んでいるとしましょう。そうすると、切手を買いに来た人が不当に待たされてしまうことに気がつくでしょう。あなたのメッセージングアーキテクチャがこの様な郵便局と同じだった場合、人々はイライラしてしまいます。

この郵便局の問題の解決方法は、行列が混雑してきた際に、遅い手続きの窓口を別に開設し、速い手続きの窓口は引き続き先着順で処理する事です。

PUSH と DEALER ソケットがこの様な単純な方式を利用するのは単にパフォーマンスが理由です。米国の主要な空港に到着すると、入国管理の所で長い行列をが出来ていることがよくあるでしょう。警備の人は人々をあらかじめ 1 つではなく複数に分けて行列を作ります。人々は 1,2 分程度時間をかけて 50 ヤードほどの行列を歩きます。これは公平な方法です。なぜなら全てのパスポートチェックは大体同じ時間で完了するからです。この様に前もってキューを分ける事で、移動距離を短くすることが PUSH と DEALER ソケットの戦略です。

これは、ØM で繰り返し議論されてきたテーマです。現実世界の問題は多様化しており、異なる問題にはそれぞれ正しい解決方法があります。空港は郵便局と異なるように、問題の規模はそれぞれ異なるのです。

それでは、ブローカー (ROUTER ソケット) に対してワーカー (DEALER や REQ ソケット) が接続する例に戻りましょう。ブローカーはワーカーの準備が完了したことを知っていて、ワーカーの一覧を保持する必要があります。

これを行う方法は簡単です。ワーカーは起動時に「準備完了」メッセージを送信し、その後仕事を行います。ブローカーは最も古いものから順にメッセージを 1 つずつ読み込んでいきます。そして、今回は ROUTER ソケットを利用しているので、ワーカーに返信するための ID を取得しています。

これはリクエストに対して応答を返していることから、リクエスト・応答パターンの応用と言えます。これらを理解する為のサンプルコードを示します。

3.4.1 ROUTER ブローカーと REQ ワーカー

これは ROUTER ブローカーを利用して REQ ワーカー群と通信を行う負荷分散パターンのサンプルコードです。

rtreq.c: ROUTER ソケット対 REQ ソケット

```
// ROUTER ソケット対 REQ ソケット

#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);
    s_set_id (worker);          // ID の設定
    zmq_connect (worker, "tcp://localhost:5671");

    int total = 0;
    while (1) {
        // 準備が完了したことをブローカーに通知
        s_send (worker, "Hi Boss");

        // ブローカーからタスクを受け取る
        char *workload = s_recv (worker);
        int finished = (strcmp (workload, "Fired!") == 0);
        free (workload);
        if (finished) {
            printf ("Completed: %d tasks\n", total);
            break;
        }
        total++;

        // ランダムな処理タスク
        s_sleep (randof (500) + 1);
    }
    zmq_close (worker);
    zmq_ctx_destroy (context);
    return NULL;
}
```

```
// このサンプルコードは簡単に実行できるようにシングルプロセスで動作する
// 様にしましたが、各スレッドは独立したコンテキストを持っていますので、
// 概念的にはプロセスを別けた状況をシミュレートしています。

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, NULL);
    }
    // 5 秒間実行した後にワーカーに終了を通知します
    int64_t end_time = s_clock () + 5000;
    int workers_fired = 0;
    while (1) {
        // ワーカーからメッセージを取得します
        char *identity = s_recv (broker);
        s_sendmore (broker, identity);
        free (identity);
        free (s_recv (broker));     // エンベロープの区切り
        free (s_recv (broker));     // ワーカーからの応答
        s_sendmore (broker, "");

        // 解雇する時までワーカーを励まし続ける
        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;
}
```

このサンプルコードを実行して 5 秒程度待つと、各ワーカーが処理したタスクの数を出力します。ルーティングが機能していれば、タスクは均等に分散されているはずです。

```
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
```

この例では、REQ ソケットと通信を行うために、ID フレームと空のエンベロープフレームを加えたメッセージを作成する必要があります。

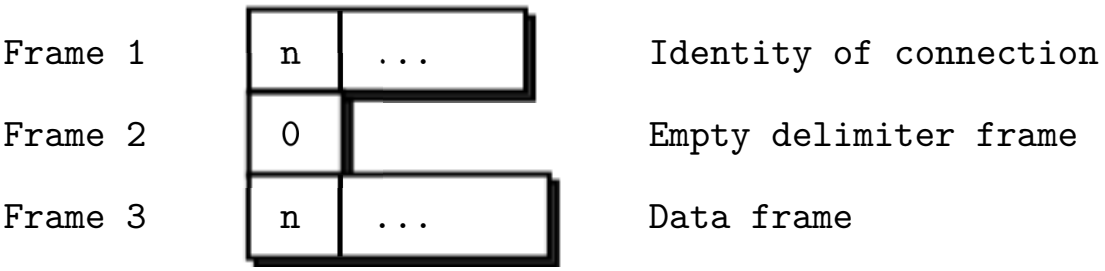


図 3.6 REQ ソケットと通信するためのルーティングエンベロープ

3.4.2 ROUTER ブローカーと DEALER ワーカー

REQ ソケットの代わりに DEALER ソケットを利用することも可能です。これらには 2 つの明確な違いあがあります。

- REQ ソケットは常にデータフレームの前に空の区切りフレームを付けて送信していましたが DEALER ソケットはこれを行いません。
- REQ ソケットは受信を行うまでに 1 つのメッセージしか送信できません。しかし DEALER 完全に非同期ですのでこれが可能です。

同期から非同期に切り替える場合でも、リクエスト・応答パターンという事に変わりありませんのでサンプルコードに大きな影響を与えません。この組み合わせはエラーからの復旧に関連していますので、後の第 4 章「信頼性のあるリクエスト・応答パターン」でも出てきます。

それでは REQ ソケットを DEALER ソケットに置き換えたまったく同じ動作を行うサンプルコードを見てみましょう。

rtdealer.c: ROUTER 対 DEALER

```
//  ROUTER 対 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);
    s_set_id (worker);          // ID の設定
    zmq_connect (worker, "tcp://localhost:5671");

    int total = 0;
    while (1) {
        // 準備が完了したことをブローカーに通知
        s_sendmore (worker, "");
        s_send (worker, "Hi Boss");

        // ブローカーからタスクを受け取る
        free (s_recv (worker));     // エンベロープの区切り
        char *workload = s_recv (worker);
        //  .skip
        int finished = (strcmp (workload, "Fired!") == 0);
        free (workload);
        if (finished) {
            printf ("Completed: %d tasks\n", total);
            break;
        }
        total++;

        // ランダムな処理タスク
        s_sleep (randof (500) + 1);
    }
    zmq_close (worker);
    zmq_ctx_destroy (context);
    return NULL;
}

// このサンプルコードは簡単に実行できるようにシングルプロセスで動作する
// 様にしましたが、各スレッドは独立したコンテキストを持っていますので、
```

```
// 概念的にはプロセスを別けた状況をシミュレートしています。

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, NULL);
    }
    // 5 秒間実行した後にワーカーに終了を通知します
    int64_t end_time = s_clock () + 5000;
    int workers_fired = 0;
    while (1) {
        // ワーカーからメッセージを取得します
        char *identity = s_recv (broker);
        s_sendmore (broker, identity);
        free (identity);
        free (s_recv (broker));     // エンベロープの区切り
        free (s_recv (broker));     // ワーカーからの応答
        s_sendmore (broker, "");

        // 解雇する時までワーカーを励まし続ける
        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;
}
```

このコードはワーカーが DEALER ソケットを利用して、データフレームの前に空フレームを付けて送信していることを除いて殆ど同じです。この方法は REQ ワーカーと互換性を保ちたい場合に役立ちます。

一方で、空の区切りフレームの存在意義を忘れないで下さい。それは終端にある REP ソ

ケットが応答エンベロープとデータフレームを区別するためのものです。

もし、メッセージが REP ソケットを経由しないのであれば、両側でこの区切り文字を省略する事が可能で、こうする事でより単純になります。これは純粋な DEALER と ROUTER プロトコルを利用したい場合に一般的な設計です。

3.4.3 負荷分散メッセージブローカー

前回のサンプルコードは複数のワーカーを管理し、擬似的なリクエストと応答を行うことが出来ましたが、これだけでは十分で無い場合があります。ワーカーからクライアントに対して問い合わせを行うことが出来ないからです。2 つ目のフロントエンド ROUTER ソケットを追加し、これでクライアントからのリクエストを受け付け、フロントエンドからバックエンドにメッセージを転送するプロキシを用意します。こうすることで、便利で再利用可能な負荷分散メッセージブローカーを作成することが出来ます。

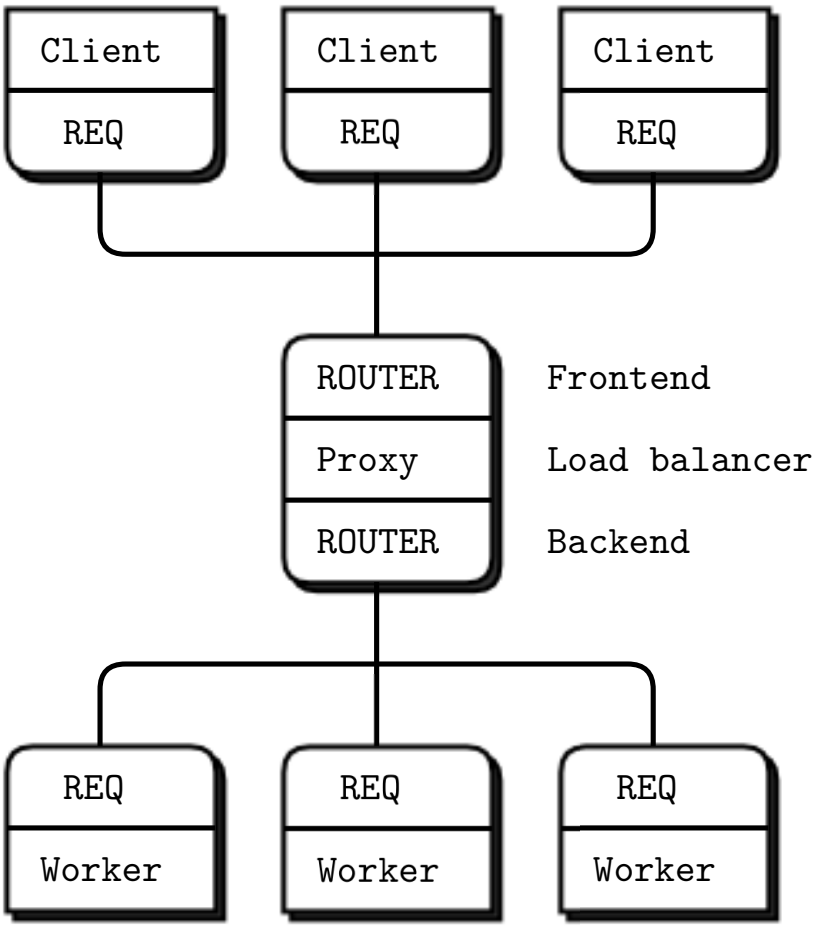


図 3.7　負荷分散ブローカー

このブローカーは以下のように動作します。

- クライアントからの接続を受け付けます。
- ワーカーからの接続を受け付けます。
- クライアントからのリクエストは単一のキューで保持します。
- これらリクエストは負荷分散パターンを利用してワーカーに送信します。
- ブローカーはワーカーからの応答を受け取ります。
- リクエストを行ったクライアントに応答を返します。

このサンプルコードはそこそこ長いですが、理解する価値はあるでしょう。

lbbroker.c: 負荷分散ブローカー

```
// 負荷分散ブローカー
// ここではクライアントとワーカーはプロセス内通信を行います。

#include "zhelpers.h"
#include <pthread.h>
#define NBR_CLIENTS 10
#define NBR_WORKERS 3

// キューから取り出す操作
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) - sizeof (q [0]))

// REQ ソケットを利用する基本的なリクエスト応答クライアントです
// s_send と s_recv はバイナリの ID を扱うことが出来ませんので ID には必ずテキ
// スト文字列を設定するようにしてください。

static void *
client_task (void *args)
{
    void *context = zmq_ctx_new ();
    void *client = zmq_socket (context, ZMQ_REQ);
    s_set_id (client);          // テキストの ID を設定
    zmq_connect (client, "ipc://frontend.ipc");

    // リクエストを送信し、応答を受信
    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;
}

// このサンプルコードは簡単に実行できるようにシングルプロセスで動作する
// 様にしましたが、各スレッドは独立したコンテキストを持っていますので、
// 概念的にはプロセスを別けた状況をシミュレートしています。

// こちらが REQ ソケットを利用して負荷分散を行うワーカータスクです。
static void *
worker_task (void *args)
{
    void *context = zmq_ctx_new ();
    void *worker = zmq_socket (context, ZMQ_REQ);
    s_set_id (worker);          // テキストの ID を設定
    zmq_connect (worker, "ipc://backend.ipc");

    //準備が完了したことをブローカーに通知
    s_send (worker, "READY");

    while (1) {

        // 本来なら空フレーム以前のフレームを全て読み取るべきです。今回
        // の例では空フレーム以前のフレームはひとつしかありませんので、
        // この様にしています。
        char *identity = s_recv (worker);
        char *empty = s_recv (worker);
        assert (*empty == 0);
        free (empty);

        // リクエストを受信し、応答を送信
        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;
}

// ここからメインタスクです。クライアントとワーカーを起動してリクエスト
```

```
// をルーティングします。
// Workers は起動時に READY というシグナルを送信し、クライアントからのリク
// エストを受付け応答を返します。
// 負荷分散アルゴリズムには簡単なキューを利用してワーカーを振り分けます。

int main (void)
{
    // コンテキストとソケットの準備
    void *context = zmq_ctx_new ();
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (frontend, "ipc://frontend.ipc");
    zmq_bind (backend,  "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++) {
        pthread_t client;
        pthread_create (&client, NULL, client_task, NULL);
    }
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_task, NULL);
    }

    // LRU キューを利用した負荷分散を行います。
    // ここではクライアントからのリクエストを受け付けるフロントエンドと
    // ワーカーにタスクを振り分けるバックエンドの 2 つのソケットを利用し
    // ます。
    // バックエンドのソケットは常に監視して、1 つ以上のワーカーが存在す
    // る場合のみフロントエンドソケットを監視します。
    // これはワーカーが存在しない場合に処理を保留するための賢い方法です。
    // クライアントからのリクエストを受け取ると、ID を含んだままのメッセー
    // ジをワーカーに対して振り分けます。
    // ワーカーからの応答を受け取ると、応答エンベロープを参照して元々の
    // クライアントに応答します。

    // ワーカーリストのキュー
    int available_workers = 0;
    char *worker_queue [10];

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
```

```
        };
        // ワーカーが存在する場合のみフロントエンドソケットを監視します
        int rc = zmq_poll (items, available_workers ? 2 : 1, -1);
        if (rc == -1)
            break;              // 割り込み

        // バックエンドのワーカーを監視します
        if (items [0].revents & ZMQ_POLLIN) {
            // ワーカーの ID を負荷分散キューに追加します
            char *worker_id = s_recv (backend);
            assert (available_workers < NBR_WORKERS);
            worker_queue [available_workers++] = worker_id;

            // 2 番目は空フレーム
            char *empty = s_recv (backend);
            assert (empty [0] == 0);
            free (empty);

            // 3 番目のフレームは READY もしくはクライアントへの応答です
            char *client_id = s_recv (backend);

            // クライアントへの応答であれば、元々のクライアントに応答します
            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);
        }

        // クライアントからのリクエストを受け付ける処理です

        if (items [1].revents & ZMQ_POLLIN) {
            // クライアントからのリクエストを受け取ると、最後に利用され
            // たワーカーを選んでルーティングします。
            // クライアントリクエストは [ID][からフレーム][リクエストの内容]
            // という順序です。
            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);

            // 負荷分散キューからワーカー ID を削除します
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            available_workers--;
        }
    }
    zmq_close (frontend);
    zmq_close (backend);
    zmq_ctx_destroy (context);
    return 0;
}
```

このプログラムの難しい所は、(a) 各ソケットでエンベロープを読み書きを行なっている事と、(b) 負荷分散アルゴリズムです。まずはエンベロープのフォーマットから説明します。

それでは、クライアントがリクエストを行い、ワーカーが応答を返す流れを見て行きましょう。このコードでは、メッセージフレームを追跡し易くする為にクライアントとワーカーの ID を設定しています。実際には ROUTER ソケットが接続 ID を割り振ることも出来るでしょう。ここでは、クライアントの ID を「CLIENT」、ワーカーの ID を「WORKER」だと仮定しましょう。まず、クライアント側のアプリケーションが「Hello」という単一のメッセージを送信します。

図 3.8　クライアントが送信するメッセージ

REQ ソケットが空の区切りフレームを追加し、ルーターソケットが接続 ID を追加するので、ブローカーはこのアドレスと区切りフレーム、データフレームを読み込みます。

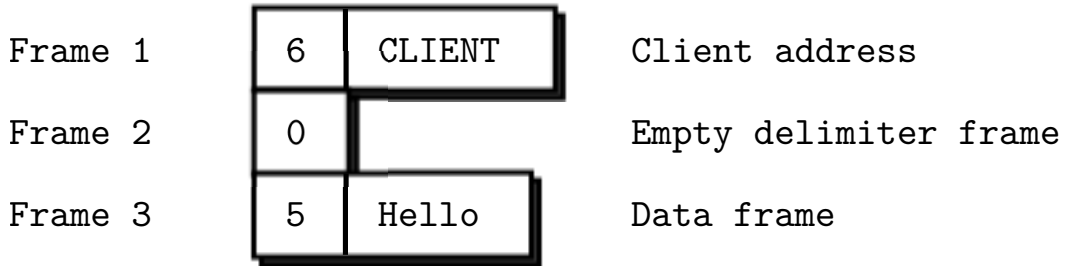


図 3.9　フロントエンドで受け取るメッセージ

ブローカーはこのメッセージに送信先に選んだワーカーのアドレスと、区切りフレームを先頭に追加てワーカーに送信します。

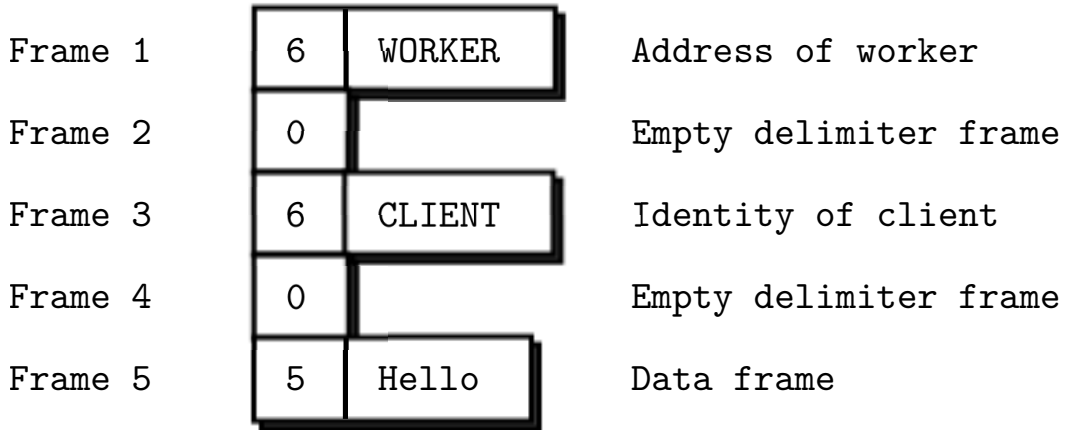


図 3.10　バックエンドに届いたメッセージ

この積み重なった複雑なエンベロープは、まずバックエンドの ROUTER ソケットで最初のフレームが取り除かれます。次にワーカー側の REQ ソケットで空の区切りフレームが取り除かれ、残りがワーカー側のアプリケーションに渡ります。

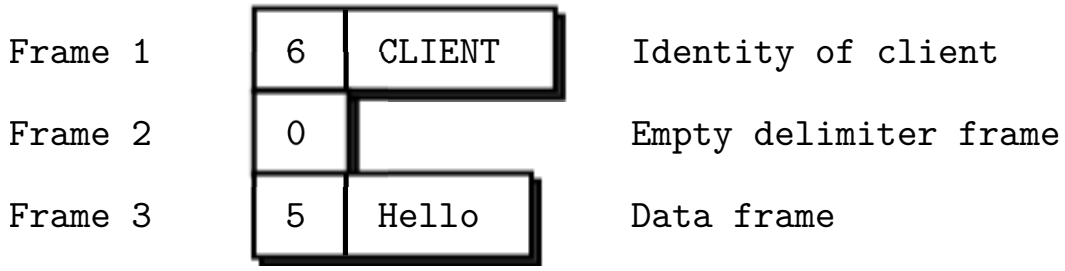


図 3.11　ワーカーに到達したメッセージ

ワーカーが必要とするのはデータ部ですが、区切りフレームを含むエンベロープ全体を保持

しておく必要があります。ここでは、REQ-ROUTER パターンを利用して負荷分散を行なっているので、REP ソケットがこれを自動的に行うことに注意して下さい。

帰りの経路は来た時と同じです。すなわち、ブローカーのバックエンドソケットで 5 つのフレームになり、ブローカーのフロントエンドは 3 つのフレームが送信されます。そしてクライアントは一つのデータフレームが渡されます。

それでは負荷分散アルゴリズムを見て行きましょう。クライアントとワーカーで REQ ソケットを利用する必要があり、ワーカーは、受け取ったエンベロープを正しく保持して応答する必要があります。このアルゴリズムは、

- zmq_pollitem_t 構造体の配列を作成してバックエンドを常にポーリングします。そして 1 つ以上ワーカーが存在する場合のみ、フロントエンドをポーリングします。
- ポーリングのタイムアウトは設定しません。
- バックエンドにワーカーからメッセージが送られて来た場合「READY」というメッセージかクライアントへの応答を受け取る可能性があります。どちらの場合でも最初のフレームはワーカーのアドレスですのでワーカーキューに格納します。残りの部分があればフロントエンドソケットを経由してクライアントに応答します。
- フロントエンドにメッセージが送られてきた場合、最後に利用されたワーカーを選択し、リクエストをバックエンドに送信します。この時、ワーカーのアドレス、区切りフレーム、データフレームという 3 つのフレームを送信します。

これまでの情報を元にして様々な負荷分散アルゴリズムに拡張できることに気がついたと思います。例えば、ワーカーが起動した後に自分自身でパフォーマンステストを走らせると、ブローカはどのワーカーが一番早いか知ることが出来ます。こうすることでブローカは最も速いワーカーを選択することが可能です。

3.5 ØMQ の高級 API

ここでリクエスト・応答パターンの話題から外れ、ØMQ API 自身の話になりますがこれには理由があります。このまま低レベルな ØMQ を使ってもっと複雑なサンプルコードを書くと可読性が低下してしまうからです。先ほどの負荷分散ブローカーのワーカースレッドの主要な処理を見て下さい。

```
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);
}
```

このコードはたった 1 つの応答アドレスしか読み取っていないので、再利用可能ではありません。そして、既に ØMQ API のヘルパー関数を利用していますが、純粋な libzmq の API を利用する場合は以下のように書く必要があるでしょう。

```
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);
}
```

そしてこのコードは長すぎるため、理解するのに時間が掛かってしまいます。これまではØMQ に慣れるためにあえて低レベルな API を利用してきましたが、そろそろその必要もなくなって来ました。

もちろん、既に多くの人々に周知されている ØMQ API を私達が勝手に変更することは出来ません。その代わりに私達の経験に基づいて高級 API を用意しています。特にこれはより複雑なリクエスト・応答パターンを書くために役立ちます。

私達が欲しいのは複数の応答エンベロープを含むメッセージを一発で送受信するための API です。これがあれば、やりたいことを最小のコードで記述することが出来ます。

良質なメッセージ API を設計するのはとても難しいことです。まず私達は用語に関する問題を抱えています。「メッセージ」という用語はマルチパートメッセージを表すこともあるし個別のメッセージフレームを表す場合もあります。期待するデータ種別が異なるという問題があります。メッセージは大抵の場合印字可能な文字列でしょうが、バイナリデータでる場合もあります。そして、技術的な挑戦として、巨大なデータをコピーせずに送信したい場合があります。

私の場合は C 言語ですが、良質な API を設計するための努力は全ての言語に影響を与えます。あなたがどのプログラミング言語を利用するにしても、より良い言語バインディングを作れるように考えています。

3.5.1 高級 API の機能

高級 API では、3 つの解かりやすい概念を利用します。文字列ヘルパー (既に出てきた s_send や s_recv の様なもの)、フレーム (メッセージフレーム)、そしてメッセージ (1 つ以上のフレームで構成される) です。これらの概念を利用してワーカーのコードを書き直してみます。

```
while (true) {
    zmsg_t *msg = zmsg_recv (worker);
    zframe_reset (zmsg_last (msg), "OK", 2);
    zmsg_send (&msg, worker);
}
```

素晴らしいことに、複雑なメッセージを読み書きする為に必要なコードを削減することが出来ました。これでかなりコードが読み易くなったでしょう。今後 ØMQ の他の機能についてはこんな風に説明します。

以下は私の経験を元に設計した高級 API の要件リストです。

- ソケットの自動処理。私は手動でソケットを閉じたり、明示的に linger のタイムアウトを設定するのが面倒になりました。ソケットはコンテキストをクローズする時に自動的

にクローズしてくれるのが望ましいでしょう。

- 移植性のあるスレッド管理。多くの ØMQ アプリケーションはスレッドを利用しますが、POSIX スレッドには移植性がありません。ですので高級 API でこの移植レイヤを隠蔽出来るのが望ましいです。
- 親スレッドから子スレッドへのパイプ接続。どの様にして親スレッドと子スレッド同士で通知を行うかという問題は度々発生します。高レベル API は PAIR ソケットとプロセス内通信を利用するメッセージパイプを提供します。
- 移植性のある時刻の取得方法。既におおよそミリ秒の精度で時刻を取得する方法はありますが移植性がありません。実際のアプリケーションでは移植性のある API が求められます。
- リアクターパターンによる zmq_poll() の置き換え。poll ループは単純ですがやや不格好です。大抵の場合、タイマーを設定して、ソケットから読み出すという単純なコードになりがちです。単純なリアクターパターンを導入して余計なな繰り返し作業を削減します。
- Ctrl-C を適切に処理する。既に割り込みを処理する方法を見てきましたが、これは全てのアプリケーションで必要とされる処理です。

3.5.2 CZMQ 高級 API

CZMQ はこのような要件リストを C 言語で実現した高級な言語バインディングです。これによって、より良い記述と移植性の高いより良い記述が可能になります。また、C 言語に限り、ハッシュやリストなどのコンテナを提供します。CZMQ はソースコードを親しみやすくする、エレガントなオブジェクトモデルを導入しています。

以下は、負荷分散ブローカーを C 言語の高級 API(CZMQ) で書き直したものです。

lbbroker2.c: 高級 API を利用した負荷分散ブローカー

```
// 負荷分散ブローカー (その 2)
// CZMQ API の利用方法を紹介します

#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      // 準備完了シグナル

// REQ ソケットを利用した基本的なリクエスト・応答クライアント
```

```
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    // リクエストを送信し、応答を受信します
    while (true) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

// REQ ソケットを利用する負荷分散されたワーカー
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    // ブローカーに準備完了を通知
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    // 到達したメッセージを処理します
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              // 割り込み
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

```
// ここからメインタスクです。今回は CZMQ ライブラリを利用していますが、機
// 能的には前回の lbbroker とまったく同じです。

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, "ipc://frontend.ipc");
    zsocket_bind (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);

    // ワーカーの負荷分散キュー
    zlist_t *workers = zlist_new ();

    // ここから負荷分散のメインループです。前回のサンプルコードと同様ですが、
    // CZMQ を利用しているのでかなり短くなています。
    while (true) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };

        // ワーカーが存在する場合のみフロントエンドソケットを監視します
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              // 割り込み

        // ワーカーとの通信を処理します
        if (items [0].revents & ZMQ_POLLIN) {
            // ワーカーの ID を負荷分散キューに追加します
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          // 割り込み
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);

            // 準備完了通知でなければクライアントに転送します
            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) {
            // クライアントからのリクエストをワーカーにルーティングします
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    // 終了処理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

CZMQ がやっていることのひとつはに割り込み処理があります。通常の ØMQ のブロッキング API は、Ctrl-C を押した時には errno に EINTR を設定して処理を中断しますが、高級 API の受信関数は単純に NULL を返します。ですので、この様な単純なループだけで行儀よくに終了することが出来ています。

```
while (true) {
    zstr_send (client, "Hello");
    char *reply = zstr_recv (client);
    if (!reply)
        break; // Interrupted
    printf ("Client: %s\n", reply);
    free (reply);
    sleep (1);
}
```

あと、zmq_poll() を呼び出す時は返り値を確認して下さい。

```
if (zmq_poll (items, 2, 1000 * 1000) == -1)
    break; // Interrupted
```

先ほどのサンプルコードではまだ zmq_poll() を使用しています。リアクターはどうなったのでしょうか。CZMQ の zloop リアクターは単純かつ機能的です。

これは以下のことを行えます。

- ソケットに対して処理関数をセット出来ます。これはソケットにメッセージが到達した時に呼び出されるコードの事です。
- ソケットと処理関数を取り外します。
- 指定した間隔でタイムアウトを発生させるタイマーを設定出来ます。
- タイマーのキャンセル出来ます。

zloop は内部的に zmq_poll() を利用しています。これは、処理関数を設定し、次のタイムアウト時間を計算してソケットの監視します。そして、処理関数と必要に応じてタイマー関数を呼び出します。

リアクターパターンを利用することで、ループが除去されます。メインループのコードはこんな風になるでしょう。

```
zloop_t *reactor = zloop_new ();
zloop_reader (reactor, self->backend, s_handle_backend, self);
zloop_start (reactor);
zloop_destroy (&reactor);
```

実際のメッセージ処理は指定した処理関数で行われます。このパターンはタイマー処理とソケットの処理が混ざっている場合に役立ちます。このスタイルが気に入らない場合もあるでしょうから好みに合わせて使用して下さい。この本では、単純なケースでは zmq_poll() を利用し、より複雑なケースでは zloop を利用しています。

以下の負荷分散ブローカーは zloop を利用して改めて書きなおしたものです。

lbbroker3.c: zloop を利用した負荷分散ブローカー

```
// 負荷分散ブローカー (その 3)
// CZMQ API とリアクターの利用方法を紹介します
//
// クライアントとワーカータスクは前回のサンプルコードとまったく同じです

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      // 準備完了シグナル
```

```
// REQ ソケットを利用した基本的なリクエスト・応答クライアント
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    // リクエストを送信し、応答を受信します
    while (true) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

// REQ ソケットを利用する負荷分散されたワーカー
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    // ブローカーに準備完了を通知
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    // 到達したメッセージを処理します
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              // 割り込み
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
```

```
    return NULL;
}

// リアクターハンドラーに渡す構造体です
typedef struct {
    void *frontend;             //  Listen to clients
    void *backend;              //  Listen to workers
    zlist_t *workers;           //  List of ready workers
} lbbroker_t;

// リアクターの設計では、メッセージが到達した際にこのハンドラー関数が呼
// び出されます。
// ここではフロントエンドとバックエンドの 2 つのハンドラを定義します。

// クライアントからの入力を処理するフロントエンドハンドラ
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);

        // ワーカーが居ない場合フロントエンドの読み込みを中止する
        if (zlist_size (self->workers) == 0) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
            zloop_poller_end (loop, &poller);
        }
    }
    return 0;
}

// ワーカーからの入力を処理するバックエンドハンドラ
int s_handle_backend (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    // ワーカーの ID を負荷分散キューに追加します
    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);

        // ワーカー数が 0 から 1 になった際にフロントエンドの監視を有効にする
        if (zlist_size (self->workers) == 1) {
            zmq_pollitem_t poller = { self->frontend, 0, ZMQ_POLLIN };
```

```
            zloop_poller (loop, &poller, s_handle_frontend, self);
        }
        // 準備完了通知でなければクライアントに転送します
        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;
}

// メインタスクではタスクの準備をしてリアクターを開始します。
// Ctrl-C が押された場合 reactor は終了してメインタスクに戻ります。
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);

    // ワーカーの負荷分散キュー
    self->workers = zlist_new ();

    // リアクターの準備と実行
    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);

    // 終了処理
    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;
}
```

Ctrl-C を送信すると、アプリケーションは行儀よく終了します。zctx_new() でコンテキストを作成した場合、自動的にシグナルハンドラが設定されますので、アプリケーションはこれに連動しなければなりません。従来の方法だと、zmq_poll の返り値が-1 であるかどうかや、zstr_recv, zframe_recv, zmsg_recv などの返り値が NULL かどうかを確認しなければなりませんでしたがもはや必要ありません。ネストしたループがある場合には、zctx_interrupted を利用して割り込みを確認する事ができます。

子スレッドでは、割り込みシグナルを受け取ることが出来ません。子スレッドに終了させるには以下の方法があります。

- 共有しているコンテキストを破棄します。そうするとブロッキングしている処理が ETERM を設定して戻ってきます。
- 独自のコンテキストを利用している場合にはメッセージを送信して終了を通知します。もちろんソケット同士で接続しておく必要があります。

3.6 非同期クライアント・サーバーパターン

ROUTER から DEALER に接続する例では、単一のサーバーが複数のワーカーに非同期で 1 対多の通信を行う例を見てきました。これとは逆に、複数のクライアントが単一のサーバーに非同期で通信を行う多対 1 のアーキテクチャも簡単に構築することが出来ます。

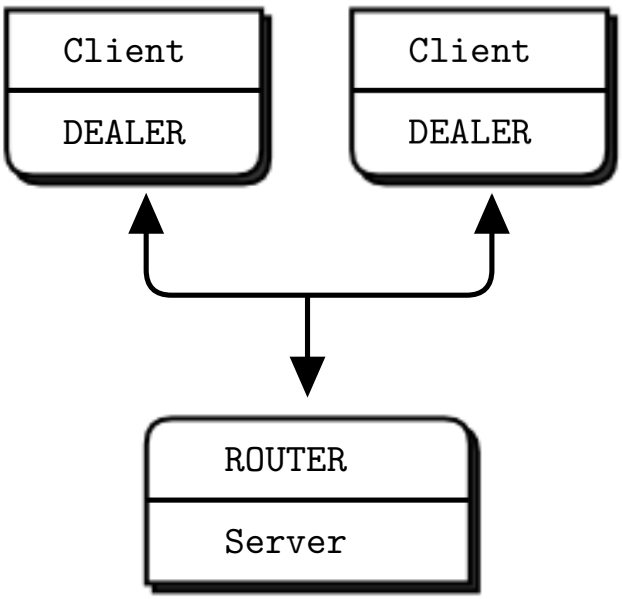


図 3.12　非同期なクライアント・サーバー

これは以下のように機能します。

- クライアント側がサーバーに対してリクエストを送信します。
- サーバーは各リクエストに対して、0 以上の応答を返します。
- クライアントは応答を待たずに複数のリクエストを送信することが出来ます。
- サーバーは新しいリクエストを待たずに複数の応答を返すことが出来ます。

以下にサンプルコードを示します。

asyncsrv.c: 非同期なクライアント・サーバー

```
// 非同期なクライアント・サーバー (DEALER 対 ROUTER)
//
// このサンプルコードは簡単に実行できるようにシングルプロセスで動作する
// 様にしましたが、各スレッドは独立したコンテキストを持っていますので、
// 概念的にはプロセスを別けた状況をシミュレートしています。

#include "czmq.h"

// こちらはクライアントタスクです。
// サーバーに接続たら、1 秒毎にリクエストを送信し、応答が到達した時点で
// メッセージを表示します。
// 複数のクライアントを並行して実行するので、ランダムな ID を設定していま
```

```
// す。

static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_DEALER);

    // トレースしやすいようにランダムな ID を設定
    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) {
        // 1 秒間ポーリングする
        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_send (client, "request #%d", ++request_nbr);
    }
    zctx_destroy (&ctx);
    return NULL;
}

// こちらがサーバータスクです
// リクエストを複数のワーカーに振り分けられるマルチサーバーモデルを利用
// しています。ワーカーは同時に 1 つのリクエストしか処理できませんが、ク
// ライアントは同時に複数のワーカーに対して処理を振り分けられます。

static void server_worker (void *args, zctx_t *ctx, void *pipe);

void *server_task (void *args)
{
    // クライアントと通信するフロントエンドソケット (TCP)
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
```

```
    zsocket_bind (frontend, "tcp://*:5570");

    // ワーカーと通信するバックエンドソケット (プロセス内通信)
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://backend");

    // 起動するワーカー数です、この数値に大して意味はありません
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++)
        zthread_fork (ctx, server_worker, NULL);

    // バックエンドソケットとフロントエンドソケットをプロキシで接続します
    zmq_proxy (frontend, backend, NULL);

    zctx_destroy (&ctx);
    return NULL;
}

// ワーカータスクがリクエストを受け付けると、ランダムな遅延を発生させて、
// ランダムな数の応答を返します。

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) {
        // DEALER ソケットからはエンベロープとメッセージを受け取ります
        zmsg_t *msg = zmsg_recv (worker);
        zframe_t *identity = zmsg_pop (msg);
        zframe_t *content = zmsg_pop (msg);
        assert (content);
        zmsg_destroy (&msg);

        // 0〜4 個の応答を返信
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            // 1 秒以下のスリープ
            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);
```

```
    }
}

// メインスレッドはクライアントやサーバーを起動するだけです、サーバーを
// 5 秒間実行したら終了します。

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);    // 5 秒間実行したら終了
    return 0;
}
```

このサンプルコードは単一プロセスで動作しますが、ここでのマルチスレッドはマルチプロセスアーキテクチャをシミュレートしていると思って見て下さい。サンプルコードを実行すると、3 つのクライアントはサーバーに対してリクエストを行い、応答を出力します。注意深く見ると、クライアントは 0 以上の応答を受け取っていることが分かるでしょう。

コードに補足すると、

- クライアントは 1 秒毎に 1 つのリクエストを送信し、複数の応答を受け取ります。これには zmq_poll() を利用しますが、単純に 1 秒のタイムアウトしまうと 1 秒間何も出来なくなってしまいますので、高頻度 (100 分の 1 秒に 1 回の頻度) でポーリングを行うようにします。
- サーバーはワーカースレッドを複数用意していてリクエストを同期的に処理します。接続をフロントエンドソケットでキューイングし、zmq_proxy() を呼び出してバックエンドソケットに接続します。

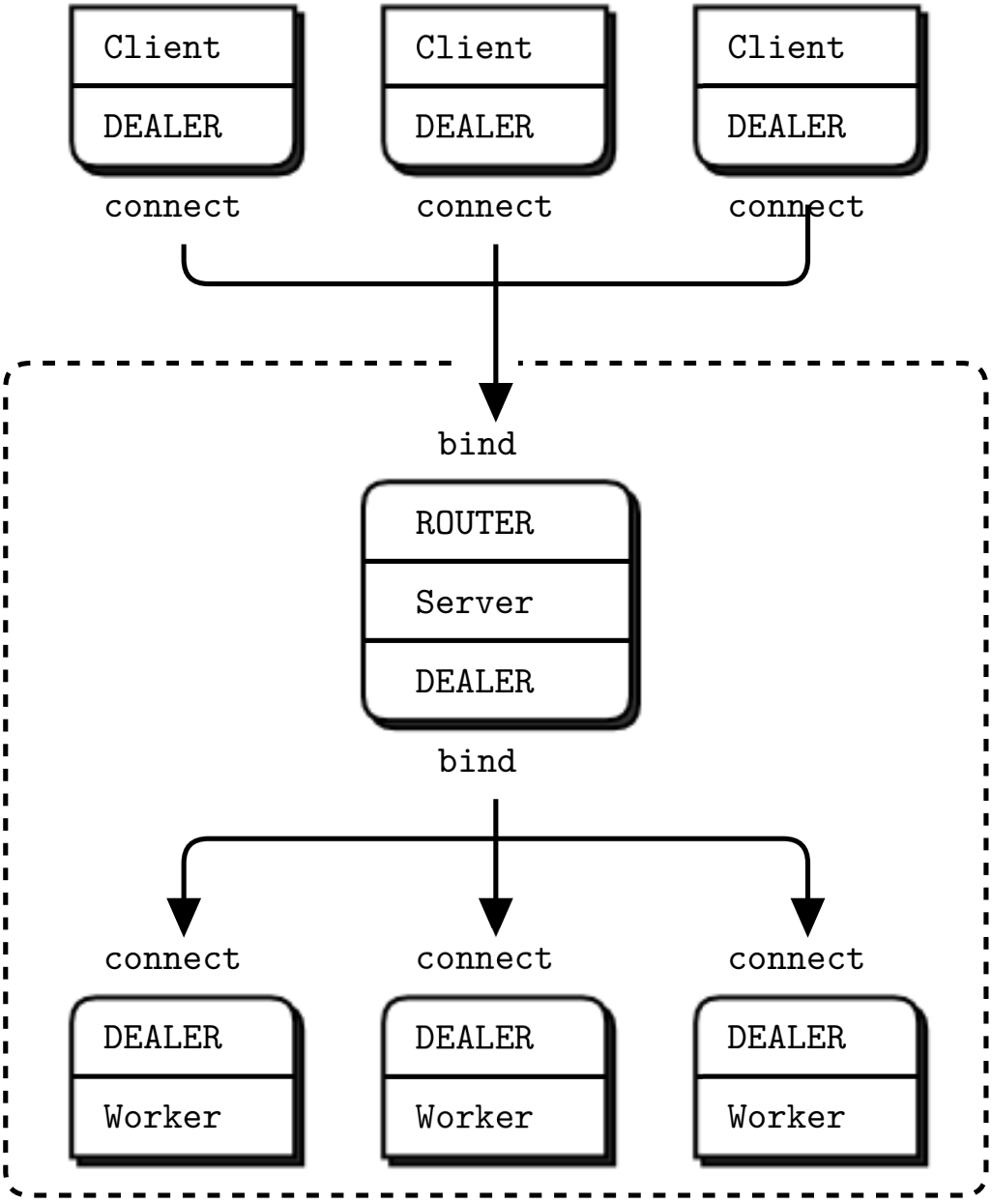


図 3.13　非同期サーバーの詳細

クライアントとサーバー間では DEALER 対 ROUTER の通信を行っていますが、内部的なサーバーとワーカーの通信では、DEALER 対 DEALER の通信を行っていることに注意して下さい。もしワーカーが完全に同期的に動作する場合は REP ソケットを利用するでしょう。しかしここでは複数の応答を行うために非同期なソケットが必要です。応答をルーティングするようなことはやりたくないので、単一のサーバーに対して応答を返すようにしてやります。

ルーティングのエンベロープについて考えてみましょう。クライアントは単一のフレームからなるメッセージを送信し、サーバースレッドはクライアントの ID が付け加えられた 2 つの

フレームを受信します。この 2 つのフレームをワーカーに送信すると、通常の応答エンベロープとして扱われ 2 つのフレームが返ってきます。そして最初のフレームはクライアントの ID としてルーテイングし、後続のフレームをクライアントに応答します。

以下の様になります

```
    client          server       frontend       worker
  [ DEALER ]<---->[ ROUTER <----> DEALER <----> DEALER ]
            1 part         2 parts       2 parts
```

ここで ROUTER から DEALER への負荷分散パターンをを利用することも出来ますが余計な作業が必要です。この場合では、各リクエストのレイテンシが少ない DEALER から DEALER へのパターンを利用するのが最も適切ではありますが、分散が平均化されないリスクがありますのでこれらがトレードオフになります。

クライアントとステートフルなやりとりを行うサーバーを構築する際、あなたは古典的な問題に遭遇するでしょう。サーバーがクライアント毎の状態を保持する場合、クライアントが接続を繰り返す内にリソースを食いつぶしてしまうという問題です。既定の ID を利用すると、こういう事になってしまいます。

これは短い時間だけ状態を保持し、一定の時間が経過した場合に状態を捨てることでこの問題を回避することが出来ます。しかしこれは多くの場合で実用的ではありません。ステートフルな非同期サーバーでは以下のようにしてクライアントの状態を適切に管理する必要があります。

- クライアントからサーバーに対して定期的に疎通確認を行います。先ほどの例では 1 秒間に一度の疎通確認を行うことが出来ます。
- クライアント ID をキーとして状態を保持します。
- 疎通確認が失敗し、例えば 2 秒間クライアントからのリクエストが行われない場合は保持しているクライアントの状態を破棄します。

3.7 ブローカー間ルーティングの実例

それでは、これまで見てきたものを実際のアプリケーションに応用してみましょう。これらを一歩一歩説明しながら作っていきます。私達の顧客が緊急に私達を呼び出して大規模なクラウドコンピューティング施設を設計するように要求してきたとします。彼らは多くのデータセンターにわたって動作するクライアントとワーカーのクラスターが協調することで全体が機能するクラウドを構想しています。我々には理論に裏付けされた知識と経験が十分にあるので、私達は ØMQ を使用してシミュレーションを行うことを提案します。その顧客は自分の上司が心

変わりする前に、Twitter 上での ØMQ の賞賛を読ませて予算を確保することに同意させます。

3.7.1 要件の確認

エスプレッソでも飲んでコードを書き始めたいところですが、重大な問題が発生する前により詳細な要件を確認しろと心の中で何かが囁きます。そこで顧客に「クラウドでどんな事をやりたいのですか?」と尋ねます。

顧客はこう答えます。

- ワーカーは様々なハードウェアで動作し、あらゆる処理を行います。クラスターは数十個ほどあり、そのクラスター毎に数百ほどのワーカーを持っています。
- クライアントは独立した処理タスクを生成し、即座に空いているワーカーを見つけてタスクを送信します。膨大な数のクライアントが存在し、任意のタイミングで増えたり減ったりもします。
- 本当に難しい所は任意のタイミングでクラスターを追加したり外したり出来るようにすることです。クラスターは全てのワーカーとクライアントを引き連れて瞬時に追加したり外すことが出来なくてはなりません。
- クラスターにワーカーが存在しない場合、クライアントの処理タスクは他のクラスターに存在するワーカーに送信します。
- クライアントがひとつのタスクを送信すると、応答が返ってくるまで待ちます。一定時間待って応答が帰ってこなかった場合は再送します。これについてはクライアントの API が勝手に行ってくれるので特に何もする必要はありません。
- ワーカーは 1 度にひとつのタスクしか処理しません。もしワーカーがクラッシュしてしまった場合は起動したスクリプトで再起動します。

これを正確に理解するために再確認します。

- 「そのクラスター間のネットワーク接続は十分高速なんでしょうね?」と尋ねます。「もちろん、そこまで我々は馬鹿じゃない」と顧客は言います。
- 「通信料はどれくらいですか?」と尋ねます。「1 クラスターあたりのクライアント数は最大 1000 台程度で、各クライアントはせいぜい秒間 10 リクエスト程度でしょう。リクエストと応答のサイズは小さく、1K バイトを超えないでしょう。」と答えました。

これを聞いて私達は通常の TCP で上手く動作するか簡単に計算します。 2,500 クライアント x 10/秒 x 1,000 バイト x 2 方向 = 50MB/秒 ～400Mb/秒。1Gb ネットワークで問題なさそうだ。

これは特別なハードウェアやプロトコルを利用しなければ簡単な問題です。ただし、特殊な

ルーティングアルゴリズムを使用する場合は注意して設計して下さい。まず 1 つのクラスター (データセンター) で設計し、続いて複数のクラスター間の接続方法を考えます。

3.7.2 単一クラスターのアーキテクチャ

ワーカーとクライアントは同期的に通信します。ここでは負荷分散パターンを利用してタスクをワーカーにルーティングします。ワーカーは全て同一のサービスを提供します。ワーカーは匿名であり、固定的なアドレスを持ちません。再試行を自動的に行いますので通信の保証については特に考えなくても良いでしょう。

これまで検討してきたように、ノードの追加や削除が動的に行えなくなるのでクライアントとワーカーは直接通信しません。従ってこれまでに見てきたリクエスト・応答のメッセージブローカーを基本的なモデルとします。

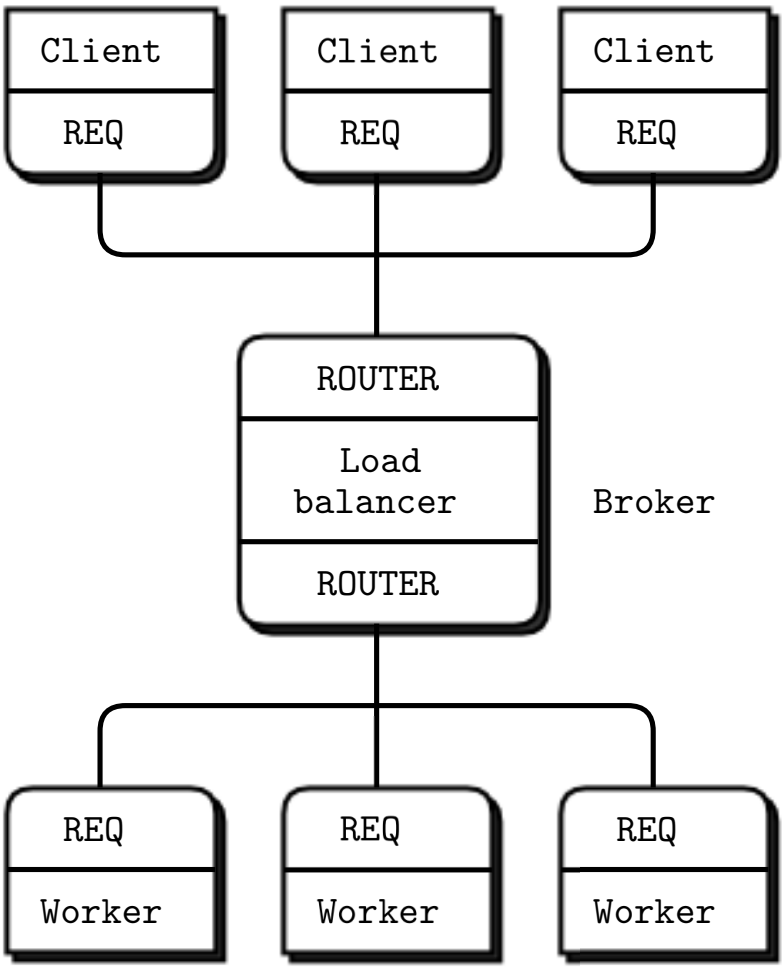


図 3.14　クラスターのアーキテクチャ

3.7.3 複数クラスターへの拡張

さて、複数のクラスターへ拡張してみましょう。各クラスターは接続されたクライアントとワーカーとブローカーで構成されます。

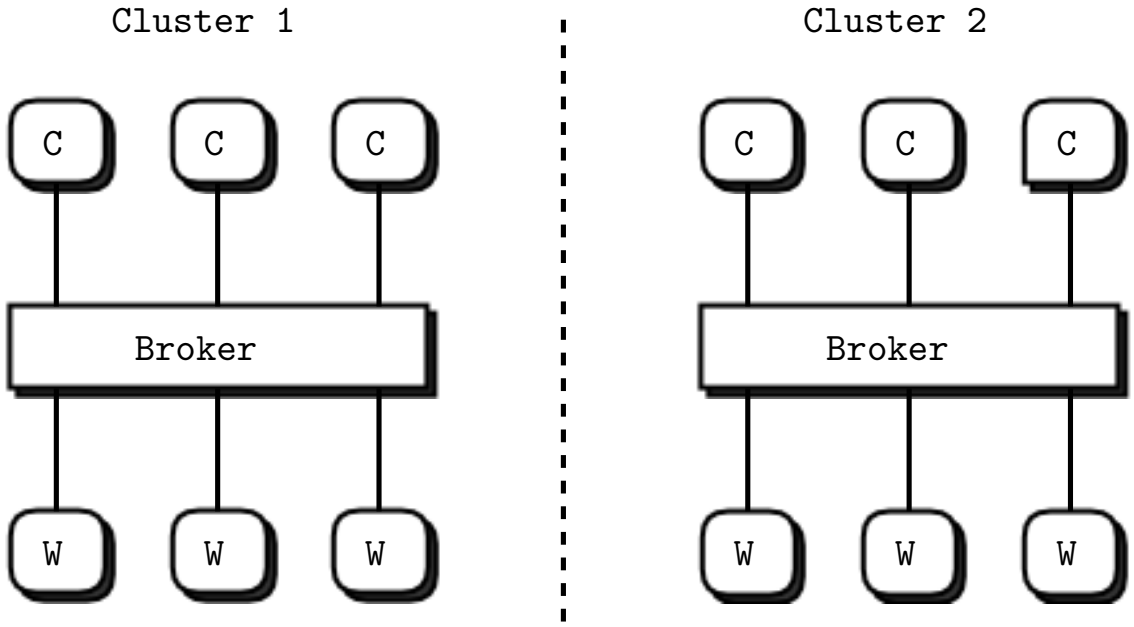


図 3.15　複数のクラスター

ここで問題です。クライアントはどの様にして他のクラスターに居るワーカーと通信すればよいのでしょうか。これには幾つかの方法があり、長所と短所があります。

- クライアントを他のブローカーに直接接続する方法。これにはブローカーとワーカーを変更する必要がないという利点があります。しかしクライアントはトポロジーを意識する必要がありますのでより複雑になります。例えば 3 つ目、4 つめのクラスターを追加する際に全てのクライアントが影響を受けます。これはルーティングやフェイルオーバーのロジックにも影響してしまうのであまり良くありません。
- ワーカーが他のブローカに接続する方法。残念ながら REQ ソケットのワーカーは 1 つのブローカーにしか応答を返さないのでこれを行うことが出来ません。そこで REP ソケットを使おうとするかもしれませんが、REP ソケットは負荷分散を行うようなルーティング機能を提供していません。これでは空いているワーカーを探して分散する機能を実現できません。唯一の方法は ROUTER ソケットを利用することです。これをアイディア#1 としておきます。
- ブローカー同士を相互に接続する方法。これは接続数を少なくできるので賢い方法の様

に見えます。クラスターを動的に追加することが難しくなりますが、これは許容範囲でしょう。クライアントとワーカーは実際のネットワークトポロジに関して何も知らなくて構いません。またブローカーはお互いの処理容量を教えあうことが可能です。これをアイディア#2 とします。

アイディア#2 について説明していきましょう。このモデルでは、ワーカーは 2 つのブローカーに接続してタスクを受け付けています。

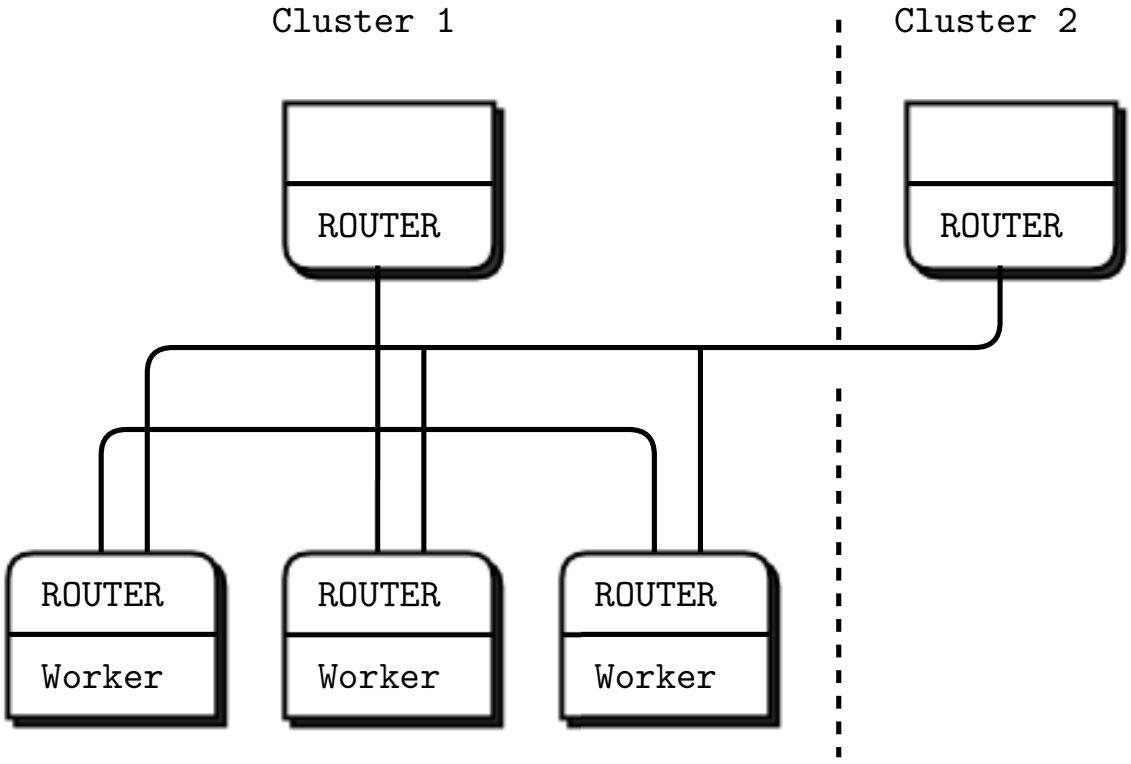


図 3.16　アイディア#1: ワーカーのクロス接続

これはもっともな方法に見えますが私達が求めているものとは少し違います。クライアントはまずローカルクラスターのワーカーを利用し、これが利用できない場合にリモートクラスターのワーカーを使用して欲しいのです。また、ワーカーが 2 つのブローカーに対して「準備完了」シグナルを送信すると、同時に 2 つのタスクを受け取る可能性があります。どうやらこれは設計に失敗した様だ。

ならばアイディア#2 で行きます。ブローカー同士の相互接続しますが、クライアントとワーカーが REQ ソケットを利用していることには触れないで下さい。

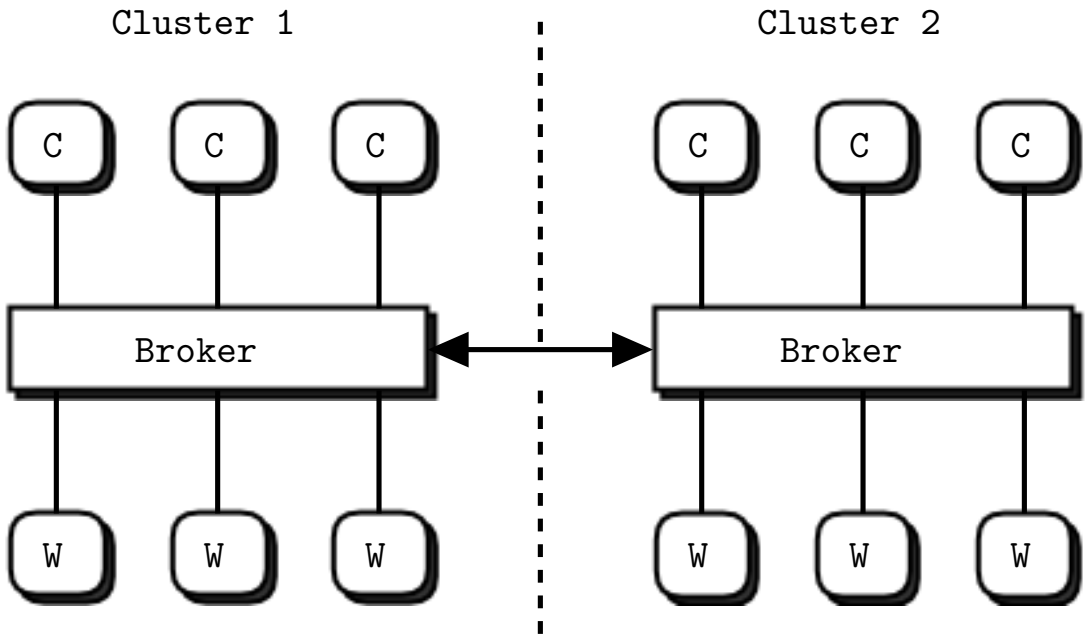


図 3.17　アイディア#2: ブローカーの相互接続

この設計は、全体を隠蔽化して局所的なクラスター内で自己完結している所が魅力的です。ブローカーは常時専用の回線で接続し、こんな風にお互いに囁き合っています。「おい、俺の処理容量は空きがあるぜ、そっちが忙しいようならこちらでタスクを引き受けるよ。」

実際にはこれはブローカーがお互いに下請け業者となるという高度なルーティングアルゴリズムです。この設計にはまだまだ特徴があります。

- クライアントとワーカーは既定ではいつも通りの動作を行います。タスクが他のクラスタに問い合わせるような場合は例外的に特別な処理を行います。
- 処理の種別に応じて異なるメッセージの経路を利用出来るようになります。これは異なるネットワーク接続を使い分ける事を意味しています。
- 高い拡張性。3 つ以上のブローカーを相互接続する場合は複雑になってきますが、もしこれが問題になる場合、超越的なブローカーを配置すれば良いでしょう。

それでは実際に動作するコードを書いてみましょう。ここでは 1 クラスターを 1 つのプロセスに押し込めます。これは現実的ではありませんが、単純なシミュレートだと思って下さい。このシミュレーションからマルチプロセスに拡張することは簡単です。ミクロのレベルで行った設計をマクロのレベルに拡張できる事こそが ØMQ の美学です。パターンやロジックはそのままで、スレッドをプロセスに移行し、さらに別サーバーで動作させることが可能です。これから作る「クラスター」プロセスにはクライアントスレッドとワーカースレッド、およびブローカースレッドが含まれています。

基本的なモデルは既に知っている通りです。

- REQ クライアントスレッドはタスクを生成し、ブローカーに渡します。(REQ ソケットから ROUTER ソケットへ)
- REQ ワーカースレッドはタスクを処理し、ブローカーに応答します。(REQ ソケットから ROUTER ソケットへ)
- ブローカーはキューイングし、負荷分散パターンを利用してタスクを分散します。

3.7.4 フェデレーションとピア接続

ブローカーを相互接続するには幾つかの方法があります。私達が欲しいのは「自分の処理容量」を別のブローカに伝え、複数のタスクを受け取る機能です。また、別のブローカーに「もう一杯だ、送らないでくれ」とつらえる機能も必要です。これは完璧である必要はありません、タスクを受け付けたら可能な限り早く処理できれば良しとします。

最も単純な相互接続を行う方法はフェデレーションモデルです。これはお互いにクライアントとワーカーをシミュレートします。これはブローカーのフロントエンドから、別のブローカーのバックエンドに接続することで実現します。この時ソケットが bind と接続の両方を行えるかどうかを確認して下さい。

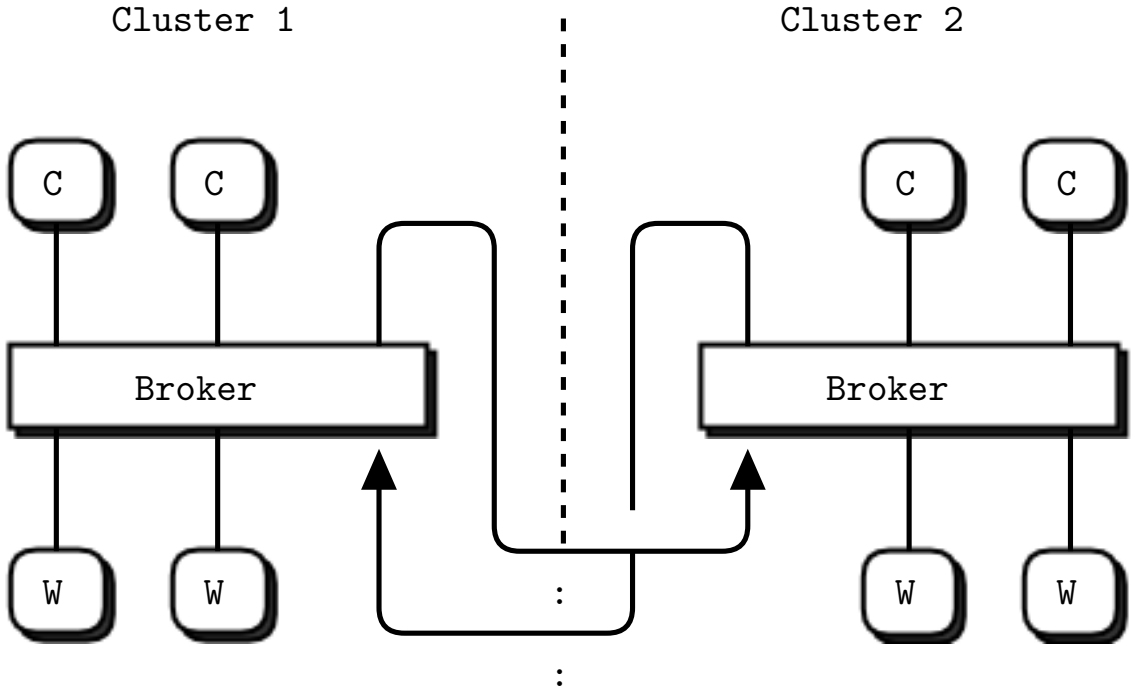


図 3.18　フェデレーションモデルによるブローカーの相互接続

これは双方のブローカーにとって単純なロジックであり、そこそこ良いメカニズムです。ワーカーが居ない場合でも他のクラスターのブローカーが「準備完了」メッセージを通知し、タスクを受け付けます。唯一の問題はこれが単純すぎるという所です。フェデレーションのブローカーは一度に 1 つのタスクしか処理できません。ブローカーがロックステップなクライアントとワーカーをエミュレートするならば、ブローカーもそのままロックステップとなり、たとえ沢山のワーカーが居たとしても同時に利用することが出来ません。ブローカーは完全に非同期で接続する必要があります。

フェデレーションモデルは様々な種類のルーティング、特にサービス指向アーキテクチャに最適です。サービス指向アーキテクチャは負荷分散ではなく、サービス種別に応じてルーティングを行います。ですので全ての用途に適応するわけではありませんが用途によっては有効です。

フェデレーションではなくピア接続を行う方法を紹介します。この方法ではブローカーは特別な接続を通じてお互いを認識しています。詳しく言うと、N 個のブローカーで相互接続を行いたい場合、(N - 1) 個のピアが存在し、これらは同じコードで動作しています。この時、ブローカー同士の間で 2 種類の情報の経路が存在します。

- 各ブローカーは常にピアに対して自信の処理容量 (ワーカーの数) を通知する必要があります。これはワーカーの数に変更が有った場合のみ通知される単純な情報になるでしょう。この様な用途に適したソケットパターンは pub-sub です。ですから全てのブローカーは SUB ソケットを利用して、各ブローカーの PUB ソケットに接続してピアから情報を受け取ることになるます。
- 各ブローカーはタスクを非同期でピアに委託し、応答を受け取る必要があります。これにはは ROUTER ソケットを利用します。これ以外の選択肢は無いでしょう。ここでブローカーは 2 種類のソケットを扱うことになります。ひとつはタスクを受信するためのソケットでもうひとつはタスクを委任するためのソケットです。2 つのソケットを利用しない場合は幾つかの作業が必要です。この場合メッセージエンベロープに付加的な情報を追加する必要があるでしょう。

もちろんこれらの情報の経路に加えて、クラスター内のワーカーとクライアントとの接続もあります。

3.7.5 命名の儀式

3 つの通信経路× 2 ソケットという事でブローカーは合計 6 つのソケットを管理する必要があります。多くのソケットを扱う際に混乱しないようにするためには良い名前を付けることが不可欠です。ソケットが何を行い、どの様な役割を持っているかを元に名前を決定します。そ

うしなければ後々寒い月曜の朝に苦んでコードを読むことになるでしょう。

それでは、ソケットの命名の儀式を行いましょう。 3 つの通信経路は、

- ブローカーとクライアント、ワーカーの間でリクエスト・応答を行う通信経路を「local」と呼びます。
- ブローカー同士の間でリクエスト・応答を行う通信経路を「cloud」と呼びます。
- ブローカー同士で状態を通知する通信経路を「state」と呼びます。

同じ長さの名前を付けるとコードがキレイに整うのでいい感じです。これは些細なことですが、細部への気配りです。ブローカーは各通信経路にそれぞれフロントエンドとバックエンドソケットを持ちます。フロントエンドからは状態やタスクを受信し、バックエンドにこれらを送信します。リクエストはフロントエンドからバックエンドへ、応答はバックエンドからフロントへ返されると考えて下さい。

という訳でこのチュートリアルでは以下のソケット名を使用します。

- 「local」の通信経路で利用するソケットは「localfe」と「localbe」
- 「cloud」の通信経路で利用するソケットは「cloudfe」と「cloudbe」
- 「state」の通信経路で利用するソケットは「statefe」と「statebe」

ここでは 1 サーバーで全てをシミュレートしているので、通信方式は全て IPC(プロセス間通信) を利用します。これは TCP で言う所の接続性を持ち、IP アドレスや DNS 名を必要としません。そして、ここではエンドポイントを呼ぶ時はシミュレートするクラスター名を付けて〜の local、〜の cloud、〜の state という言い方をします。

ソケットに名前を付ける作業が面倒になって、単純に S1, S2, S3, S4 で良いのではないかと考えていませんか? あなたの脳は完璧な機械ではありませんのでコードを読むときには名前による手助けが必要です。「6 種類のソケット」と覚えるより、「3 つの経路と、2 つの方向性」と覚える方が簡単でしょう。

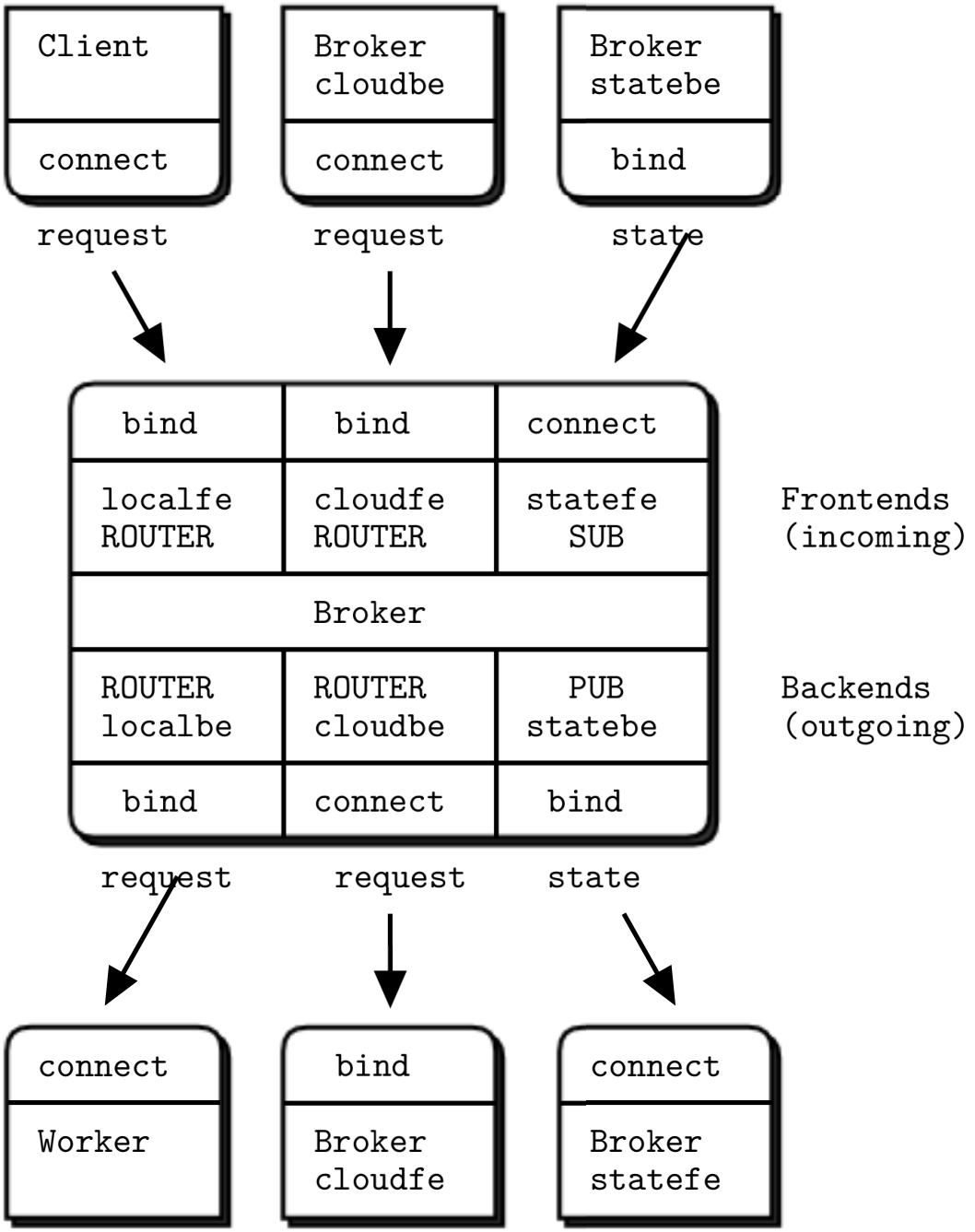


図 3.19　ブローカーが利用するソケット

各ブローカーの cloudbe ソケットは、他のブローカーの cloudfe ソケットに対して接続を行い、これと同様に statebe ソケットで、他のブローカの statefe ソケットに接続を行っています。

3.7.6 状態通知の仮実装

ソケットの通信経路には所々罠が仕掛けられていますので全てのコードが出来上がるのを待たず、ひとつずつテストを行っていきます。各経路で問題がないことを確認してからプログラム全体で動作確認を行います。ここでは状態通知経路を実装します。

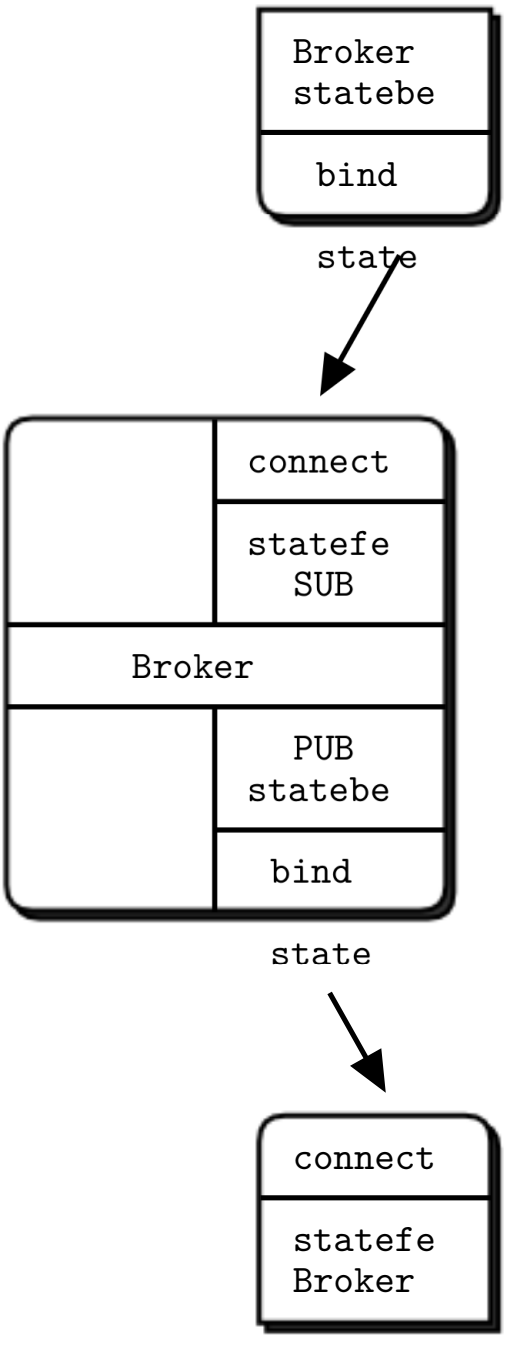


図 3.20　状態通知経路

これがコードです。

peering1.c: state flow のプロトタイプ

```
// ブローカー同士の通信 (パート 1)
// state flow のプロトタイプ

#include "czmq.h"

int main (int argc, char *argv [])
{
    // 最初の引き数は自分自身のブローカー名を指定し、
    // 残りの引き数は、他のブローカーを指定します。

    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 ();

    // statebe エンドポイントを bind します
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    // 全ブローカーに statefe ソケットで接続します
    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);
    }

    // メインループではブローカー同士でステータスメッセージをやりとりし
    // ます。zmq_poll はハートビートの間隔でタイムアウトするようにしてい
    // ます。

    while (true) {
        // 1 秒間ポーリング
```

```
        zmq_pollitem_t items [] = { { statefe, 0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        // 受信メッセージの処理
        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 {
            // 仮のワーカー数を送信
            zstr_sendm (statebe, self);
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

このコードの注意点は、

- 各ブローカーは IPC エンドポイントの名前に利用する ID を持っています。実際には TCP などの別の通信方法が利用され、アドレスは設定ファイルなどから読み込むでしょう。これについては本書の後のほうで出てきますので、ひとまずここは IPC を利用します。
- プログラムの主要な部分は zmq_poll() ループです。ここで受信したメッセージを処理し、状態情報を配信しています。メッセージを受信せず、1 秒が経過した場合にのみ状態情報を配信します。もしも 1 つのメッセージを受信する度にメッセージを送信した場合、メッセージの嵐が発生するでしょう。
- 状態を通知するための pub-sub メッセージは、送信者のアドレスとデータからなる 2 つのメッセージフレームで構成します。送信者のアドレスはタスクを送信するために必要なものです。
- 既に動作しているブローカに接続した際に、古い状態情報を取得してしまう可能性があるので、ソケットにサブスクライバーの ID は設定していません。
- ここでは HWM を設定していませんが、もし ØMQ v2.x を使用しているのであれば設定したほうが良いでしょう。

このプログラムをビルドしたら 3 回実行して 3 つのクラスターをシミュレートします。何でも構いませんがここではそれぞれのクラスターを DC1, DC2, DC3 と呼びます。3 つのターミナルを開いて以下のコマンドを実行してみましょう。

```
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
```

各クラスターは 1 秒毎に接続相手の仮の状態情報を出力します。実際にこれを試してみて、3 つのブローカーが 1 秒間隔で状態情報を同期できていることを確認してみましょう。

実際には、一定間隔で状態メッセージを送信するのではなく、ワーカーに増減が有った場合などの状態に変更があった場合のみ送信したいと思うかもしれません。しかしこのメッセージは十分小さく、クラスター間の接続十分速い事は確認しましたので気する必要はないでしょう。

もし、状態メッセージ正確な間隔で送信したい場合は子スレッドを生成して statebe ソケットを扱えば良いでしょう。また、ワーカーの数に更新があった場合にメインスレッドから子スレッドに通知を行い、定期的なメッセージと合わせて送信しても良いでしょう。これ以上事はここでは取り上げません。

3.7.7 タスクの通信経路を仮実装

では、local や cloud ソケットを経由するタスクの経路を実装してみましょう。このコードはクライアントから受信したタスクを、ローカルのワーカーや別のクラウドに分散して送信します。ここでは仮にランダムに送信先を決定します。

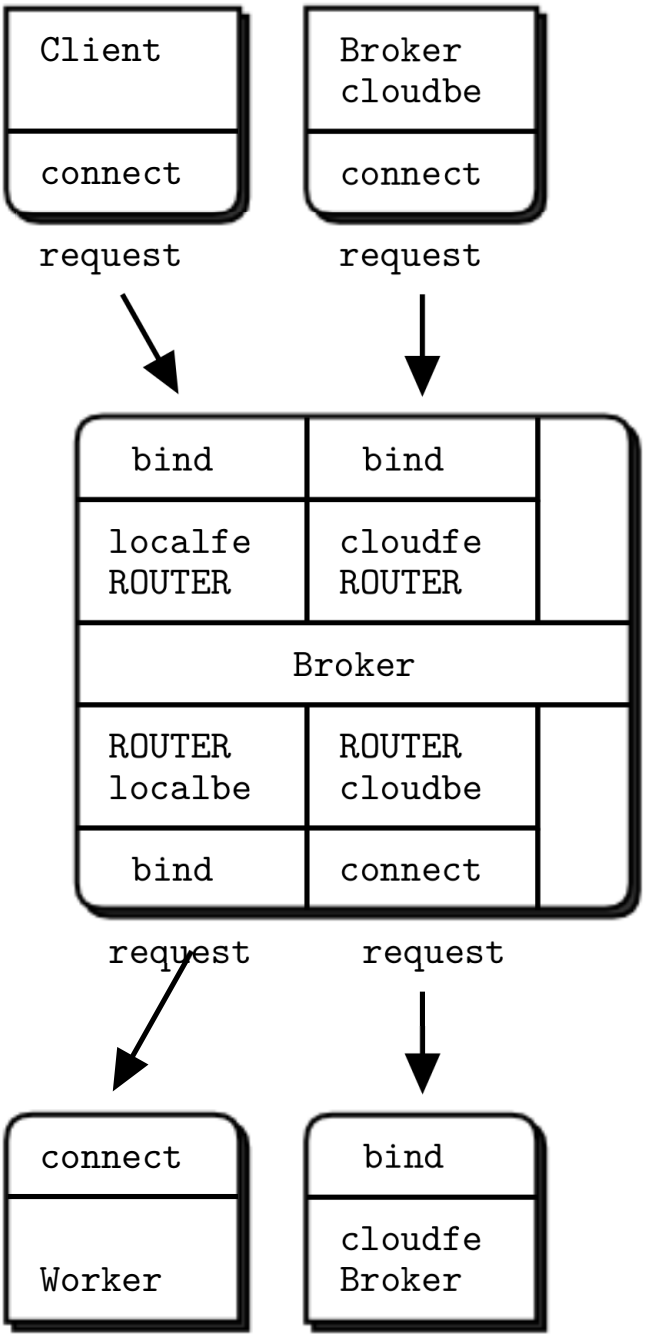


図 3.21　タスクの通信経路

コードが若干複雑になってきているので、まずはルーティングのロジックの整理して設計を掘り下げてみましょう。

ローカルのクライアントと他のクラウドからのリクエストを受け付けるために 2 つのソケットが必要です。ローカルとクラウドから受信したメッセージを個別のキューに格納する必要が

ありますが、ØMQ ソケットは既にキューになっているので特に配慮する必要はありません。

これは負荷分散ブローカーで利用したテクニックですね。今回は 2 つフロントエンドソケットから読み込んだメッセージをバックエンドに送信し、2 つのバックエンドソケットから読み込んだメッセージをフロントエンドにルーティングします。また、バックエンドが接続してきていない場合は、フロントエンドのソケットを監視する必要は無いでしょう。

メインループはこんな風になります。

- バックエンドのソケットを監視してメッセージを受信します。これが「準備完了」メッセージであればなにもしません、そうでなければフロントエンドの local または cloud にルーティングして応答します。
- ワーカーからのメッセージ届けば、そのワーカーは空きになったと言えるのでキューに入れてカウントします。
- フロントエンドからのリクエストを受け付けた時、ワーカーの空きがあれば、ローカルのワーカーか、他のクラウドのどちらかをランダムに選んでルーティングします。

他のブローカーをワーカーと見なして分散させるのではなく、単純にランダムで選択するというのはあまり賢く無いですが、ここではこれを行います。

ブローカーはそれぞれのブローカー間でメッセージのルーティングを行うために ID を持っており、この ID はコマンドラインで指定しています。この ID はクライアントノードが生成する UUID と重複しないように注意して下さい。もし重複してしまったら、クライアントに返すべき応答がブローカーにルーティングされてしまいます。

ここからが実際に動作するコードです。注目に値する部分は、「ここからが面白い」とコメントで書いてあります。

peering2.c: local と cloud flow のプロトタイプ

```
// ブローカー同士の通信 (パート 2)
// リクエスト・応答フローのプロトタイプ

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define WORKER_READY   "\001"      // 準備完了シグナル

// このノードの名前
static char *self;

// クライアントタスクは同期 REQ ソケットを利用します。
```

```
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) {
        // リクエストを送信して、応答を受信します
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              // 割り込み
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

// ワーカータスクは負荷分散されたリクエストを REQ ソケットで受信します。

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);

    // ブローカーに準備完了を通知します
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    // 受信メッセージの処理
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              // 割り込み

        zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
```

```
    zctx_destroy (&ctx);
    return NULL;
}

// メインタスクではフロントエンドとバックエンドのソケットの準備を行いク
// ライアントタスクとワーカータスクを起動します。

int main (int argc, char *argv [])
{
    // 最初の引き数は自分自身のブローカー名を指定し、
    // 残りの引き数は、他のブローカーを指定します。

    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 ();

    // cloundfe をエンドポイントとして Bind
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    // 全ての cloudfe に接続します
    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);
    }
    // localfe と localbe の準備を行います
    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);

    // 準備ができたらユーザーに確認を求めます
    printf ("Press Enter when all brokers are started: ");
    getchar ();
```

```
    // ローカルのワーカーを起動します
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    // ローカルのクライアントを起動します
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);

    // ここからが面白い
    //
    // ここからがリクエスト・応答フローの処理です。
    // 複数のワーカーを常に監視して負荷分散を行います。

    // ワーカーの LRU キューを作成
    int capacity = 0;
    zlist_t *workers = zlist_new ();

    while (true) {
        // まず、ワーカーからの応答を待ちます。
        zmq_pollitem_t backends [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 }
        };
        // ワーカーがいなければ無期限に待ち続けます。
        int rc = zmq_poll (backends, 2,
            capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              // 割り込み

        // ローカルワーカーからの応答を処理します
        zmsg_t *msg = NULL;
        if (backends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          // 割り込み
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);
            capacity++;

            // 準備完了通知であった場合ルーティングしません
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
```

```
            zmsg_destroy (&msg);
        }
        // ブローカーからの応答を処理します
        else
        if (backends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          // 割り込み
            //  ブローカーの ID は特に利用しません
            zframe_t *identity = zmsg_unwrap (msg);
            zframe_destroy (&identity);
        }

        // ブローカーのアドレスであればルーティングします
        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);
        }
        // クライアントにルーテイングします
        if (msg)
            zmsg_send (&msg, localfe);

        // ワーカーの容量がある限りクライアントからのリクエストをルーティ
        // ングします。
        // ローカルクライアントからのリクエストは他のブローカに再ルーティ
        // ングするかもしれませんが、他のブローカーから再ルーティングさ
        // れてきたリクエストを再度、再ルーティングすることはありません。
        // 最ルーティングはここでは簡単にランダムで行います。次のバージョ
        // ンでは適切に cloud 容量を算出して再ルーティングを行います。

        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;
            // ブローカーからのリクエストを優先します
            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;      // 何もせずバックエンドの処理を行います

            // 再ルーティング可能な場合、20% の確率で他のブローカーに再
            // ルーテイングします。
            if (reroutable && argc > 2 && randof (5) == 0) {
                // ランダムなブローカーに再ルーティング
                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--;
            }
        }
    }
    // 終了処理を行います
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

これを試すには、ターミナルを 2 つ開いて 2 つのインスタンスを起動して下さい。

```
peering2 me you
peering2 you me
```

少し解説しておくと、

- C 言語の場合、機能を抽象化した zmsg_関数を利用することで短くて簡潔なコードになります。これをビルドするには、CZMQ ライブラリとリンクする必要があります。
- ピアブローカーからの状態情報が送られて来ない場合でも、普通に動作していると仮定

してしまいますので、コードの最初で全てのブローカーが起動しているかどうかを確認しています。実際にはブローカーが何も言わなくなった時は、タスクを送信しないようにしてやると良いでしょう。

あなたは動き続けているコードを眺めて満足するでしょう。もしもメッセージが誤った経路で流れると、クライアントは停止してブローカーはトレース情報を出力しなくなるでしょう。クライアントは応答が返ってくるまで待ち続けてしまいますので、こうなった場合はクライアントとブローカーを再起動するしかありません。

3.7.8 プログラムの結合

それではこれまでの仮実装のコードをひとつにまとめてみましょう。以前にも述べた通り、ここでは 1 クラスターの全てを 1 プロセスで実現します。そこで先程の 2 つのコードを合わせることで、クラスターをシミュレート出来るようになります。

先程の仮実装を合わせると、コードサイズは約 270 行程度になります。これはクライアントとワーカーを含む負荷分散クラスターを上手くシミュレートしています。コードはこちらです。

peering3.c: 完全なクラスターシミュレーション

```
// ブローカー同士の通信 (パート 3)
// プロトタイプの結合

#include "czmq.h"
#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define WORKER_READY   "\001"      // 準備完了シグナル

// このノードの名前
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:

// こちらはクライアントタスクです。これは要求を連続送信した後に数秒間ス
// リープします。
```

```
// これは突発的な大量のリクエストをシミュレートしています。同時に複数の
// クライアントが動作するとローカルワーカーは過負荷になるでしょう。
// クライアントは REQ ソケットで要求を送信し、統計情報を送信する為に PUSH
// ソケットを利用します。

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));

            // ランダムに生成したタスク ID を送信
            zstr_send (client, task_id);

            // 最大 10 秒待って、応答がなければエラーを送信する
            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;          // 割り込み

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
                    break;              // 割り込み
                // ワーカーはタスク ID を応答するはずです
                assert (streq (reply, task_id));
                zstr_send (monitor, "%s", reply);
                free (reply);
            }
            else {
                zstr_send (monitor,
                    "E: CLIENT EXIT - lost task %s", task_id);
                return NULL;
            }
        }
```

```
    }
    zctx_destroy (&ctx);
    return NULL;
}


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:

// ワーカータスクは負荷分散されたリクエストを REQ ソケットで受信します。
// これは前回のサンプルコードで見た内容と同じです。

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);

    // ブローカーに準備完了を通知します
    zframe_t *frame = zframe_new (WORKER_READY, 1);
    zframe_send (&frame, worker, 0);

    // 受信メッセージの処理
    while (true) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              // 割り込み

        // 負荷をシミュレートする為に 0 秒か 1 秒スリープします
        sleep (randof (2));
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}


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:


// メインタスクはソケットの準備を行います。localfe ソケットはクライアン
// トと通信し、localbe ソケットはワーカーと通信します。
// cloudfe ソケットには他のブローカーをクライアントとして通信し、
// cloudbe ソケットは他のブローカーをワーカーとして通信します。
// statebe ソケットは他のブローカーに対して状態メッセージを送信し、
// statefe ソケットは他のブローカーの状態メッセージを収集します。
// 最後に monitor ソケットを利用して各タスクからの処理結果を受信して表示
// します。

int main (int argc, char *argv [])
{

    // 最初の引き数は自分自身のブローカー名を指定し、
    // 残りの引き数は、他のブローカーを指定します。

    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));

    // localfe と localbe の準備を行います
    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);

    // cloundfe をエンドポイントとして Bind
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    // cloudbe ソケットで全てのブローカーに接続
    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);
    }

    // statebe エンドポイントを bind します
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    // 全ブローカーに statefe ソケットで接続します
    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);
    }
    // モニターソケットの準備
    void *monitor = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (monitor, "ipc://%s-monitor.ipc", self);

    // 全てのソケットを bind、接続した後にワーカータスクとクライアントタ
    // スクを起動します。
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (worker_task, NULL);

    // ローカルのクライアントを起動します
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (client_task, NULL);

    //  Queue of available workers
    // 有功なワーカーの数
    int local_capacity = 0;
    int cloud_capacity = 0;
    zlist_t *workers = zlist_new ();

    // メインループは大きく別けて 2 つの処理があります。
    // まず、ワーカーとその他 2 つのソケット (statefe と monitor) を監視しま
    // す。有効なワーカーが居ない場合はリクエストを受信しても意味があり
    // ません。
    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 }
    };
    // ワーカーがいなければ無期限に待ち続けます。
    int rc = zmq_poll (primary, 4,
        local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
    if (rc == -1)
        break;              //  割り込み

    // ワーカー容量の変更を検知するために退避
    int previous = local_capacity;
    zmsg_t *msg = NULL;     // ローカルワーカーからのメッセージ

    if (primary [0].revents & ZMQ_POLLIN) {
        msg = zmsg_recv (localbe);
        if (!msg)
            break;          //  割り込み
        zframe_t *identity = zmsg_unwrap (msg);
        zlist_append (workers, identity);
        local_capacity++;

        // 準備完了通知であった場合ルーティングしません
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), WORKER_READY, 1) == 0)
            zmsg_destroy (&msg);
    }
    // ブローカーからの応答を処理します
    else
    if (primary [1].revents & ZMQ_POLLIN) {
        msg = zmsg_recv (cloudbe);
        if (!msg)
            break;          //  割り込み
        // ブローカーの ID は特に利用しません
        zframe_t *identity = zmsg_unwrap (msg);
        zframe_destroy (&identity);
    }
    // ブローカーのアドレスであればルーティングします
    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);
    }
    // クライアントにルーテイングします
```

```
        if (msg)
            zmsg_send (&msg, localfe);

        // statefe もしくは monitor ソケットからの入力があればそれらのメッ
        // セージを処理します。
        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);
        }
          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.

        // ここではクライアントからのリクエストをルーティングします。
        // ローカルにワーカーが居る場合は localfe と cloudfe の両方を監視し
        // ます。ワーカーがクラウドにしか居ない場合は localfe のみを監視
        // します。
        // まずリクエストをローカルのワーカーにルーティングしようとし、
        // ローカルのワーカーが居ない場合にクラウドに流します。

        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;      // 何もせずバックエンドの処理を行います

            if (local_capacity) {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                local_capacity--;
            }
            else {
                // ランダムなブローカーにルーティング
                int peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [peer], strlen (argv [peer]));
                zmsg_send (&msg, cloudbe);
            }
        }
        We broadcast capacity messages to other peers; to reduce chatter,
        we do this only if our capacity changed.

        // ワーカー容量に変更があった場合、他のブローカーに自身のワーカー
        // 容量をブロードキャストします。
        if (local_capacity != previous) {
            // エンベロープに自分の ID を付与します
            zstr_sendm (statebe, self);
            // 現在のワーカー容量をブロードキャスト
            zstr_send (statebe, "%d", local_capacity);
        }
    }
    // 終了処理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

これはそこそこ大きなプログラムですので実装するのに 1 日かかりました。以下に要点をまとめます。

- クライアントスレッドはリクエストの失敗を検知して報告します。これはレスポンスが返ってくるまでの間ソケットを監視して、10 秒間何も返ってこなければエラーと判断します。
- クライアントスレッドは直接表示を行わず、PUSH ソケットでモニター用の PULL ソ

ケットに対してメッセージを送信し、ブローカーと同じメインスレッドで出力を行います。ここで初めて ØMQ ソケットをモニタリングとログ出力に利用しましたが、この使い方は重要ですので後で詳しく説明します。

- クライアントは様々な負荷をシミュレートし、そのタスクはクラウドに移されます。シミュレートする負荷は、クライアントとワーカーの数および遅延時間で制御出来ます。このプログラムを実行し、より現実的な用途に適用できるかどうかを確認してみて下さい。
- メインループでは 2 つの zmq_pollitem_t 構造体を利用しています。実際には 3 つに分けても良いでしょう。バックエンド容量が無い時にフロントエンドを監視しても意味が無いので、このサンプルコードでは 2 つに分けています。

このプログラムを開発するにあたって発生した問題をまとめると、

- クライアントはリクエストや応答が迷子になってしまうとフリーズしてしまう問題がありました。ROUTER ソケットはルーティング出来ないメッセージを捨ててしまうからです。最初に行った対策はクライアントスレッドでこれを検知して問題を報告するようにしました。そして、メッセージの受信を行った直後に `zmsg_dump()` を呼び出すようにしました。これで問題の所在が明らかになります。
- ブローカーのメインループで複数のソケットにメッセージが届いた場合、最初のメッセージを取りこぼすという問題がありました。これは最初に準備が出来たソケットからメッセージを受信する様にして修正しました。

このシミュレーションはクラウドの停止を検知しません。複数のクラウドを開始してひとつを停止した場合、一度でも処理容量を通知していれば他のクラウドはメッセージを送り続け、クライアントのタスクは消失していまいます。これは簡単にリクエストの消失状態を再現することが出来ます。解決方法は 2 つあります。1 つ目は、クラウドからの処理容量の通知が届かなくなり、一定の時間が経ったら処理容量を 0 に設定することです。もうひとつの方法は信頼性のあるリクエスト・応答モデルを構築することです。これについては次の章で説明します。

第 4 章

信頼性のあるリクエスト・応答パターン

第 3 章「リクエスト・応答パターンの応用」ではリクエスト・応答パターンの高度な応用方法を実際に動作する例と共に見てきました。この章では一般的な問題である信頼性を確保する方法、および様々な信頼性のあるメッセージングパターンの構築方法を紹介します。

この章では便利で再利用可能な以下のパターンを紹介します。

- ものぐさ海賊パターン: クライアント側による信頼性のあるリクエスト・応答パターン
- 単純な海賊パターン: 負荷分散を利用したリクエスト・応答パターン
- 神経質な海賊パターン: ハートビートを利用したリクエスト・応答パターン
- Majordomo パターン: サービス指向の信頼性のあるキューイング
- タイタニックパターン: ディスクベース・非接続な信頼性のあるキューイング
- バイナリースターパターン: プライマリー・バックアップ構成
- フリーランスパターン: ブローカー不在の信頼性のあるリクエスト・応答パターン

4.1 「信頼性」とは何でしょうか?

人々はよく「信頼性」という言葉を口にしますが、ほとんどの人はその本当の意味を理解していません。ここでは障害時における「信頼性」という言葉を定義します。もしも既知および未知の障害に対して適切にエラー処理を行うことができた場合、障害に対して信頼性があると言うことが出来ます。これ以上でもこれ以下でもありません。ですので、まずは ØMQ の分散アプリケーションで発生しうる障害について見て行きましょう。

- アプリケーションにとっての最大の罪はクラッシュして異常終了したり、フリーズして入力を受け付けなくなったり、メモリを使い果たして遅くなったりする事です。
- アプリケーションプログラムが原因で、ブローカーなどのシステムプログラムがクラッシュしてしまう障害。システムプログラムはアプリケーションプログラムより信頼性が求められますが、クラッシュしたり、メモリを使い果たしてしまう事は起こり得ます。
- システムプログラムが遅いクライアントの相手をすると、メッセージキューが溢れてしまう事があります。キューが溢れるとメッセージを捨ててしまうのでメッセージが喪失してしまう障害が発生します。
- ハードウェアに障害が発生すると、そのサーバー上で動作している全てのプロセスが影響を受けます。
- ネットワーク障害が発生すると、不可思議な現象を引き起こす場合があります。例えばスイッチのポートが故障することで部分的なネットワークにアクセス不能になります。
- データセンター全体で障害が発生する可能性があります。例えば地震、落雷、火事、空調の故障などです。

これら全ての障害に対して、信頼性を高める対策をソフトウェアで行うのは非常に高価で難しいことであり、本書の範疇を超えています。

現実世界で発生する障害の 99.9% は最初の 5 つに分類されるでしょう。(これは私が行った十分に科学的な調査です) もし最後の 2 例の問題にお金をつぎ込みたいと考えているお金の余っている大企業が居られましたら是非とも我社にご連絡下さい。ビーチハウスの裏側に高級プールを作って頂きたいです。

4.2 信頼性の設計

とても単純な事なのですが、信頼性とはコードがフリーズしたりクラッシュしてしまい、いわゆる「落ちた」という状況であっても、正しく動作し続けることです。しかし、正しく動作し続けるという事は思っている以上に大変な事です。まずは ØMQ メッセージングパターン毎にどの様な障害が発生し得るか考えてみる必要があるでしょう。

ひとつずつ見ていきます。

- リクエスト・応答パターン: リクエストの処理中にサーバーが落ちてしまった場合、クライアントは応答が返ってこないので障害の発生を検知することが出来ます。そしてリクエストを諦めたり、再試行を行ったり、別のサーバーを探すことが可能です。クライアントが落ちてしまった場合は「別のなにかの問題」として除外することが出来ます。
- Pub-sub パターン: クライアントが落ちた場合、サーバーはこれを検知することが出来

ません。Pub-sub パターンではクライアントからサーバーに対して一切の情報を送信しないからです。ただし、クライアントは別の経路、例えばリクエスト応答パターンを利用して「取りこぼしたメッセージを再送して下さい」と要求することは可能です。サーバーが落ちてしまった場合はここでは扱いません。また、サブスクライバーはパブリッシャーの動作が遅くなったことを検知して警告を通知したり、終了させたりすることができます。

- パイプラインパターン: ワーカーが処理中に落ちた場合でもベンチレーターはそれを検知することが出来ません。パイプラインパターンは回っている歯車のようなもので、処理は一方方向にしか流れません。しかし下流のコレクターは特定のタスクが処理されなかったことを検知することが可能です。そしてベンチレーターに対して「おい、タスク 324 を再送してくれ!」という様なメッセージを送信することが可能です。ベンチレーターやコレクターが落ちてしまった場合はどうでしょうか、上流のクライアントはリクエスト全体を再送することが可能です。これはあまり賢い方法ではありませんが、そもそもシステムコードは頻繁に落ちるべきではありません。

この章では、リクエスト・応答パターンに焦点を当てて信頼性のあるメッセージングを学んでいきます。

REQ ソケットが REP ソケットに対して同期的に送受信を行うリクエスト・応答パターンでは、極めて一般的な障害が発生します。もしもリクエストの処理中にサーバーがクラッシュした場合、クライアントは永久に固まってしまいます。そして、もしネットワークがリクエストや応答を喪失した場合でもクライアントは永久に固まってしまうでしょう。

リクエスト応答パターンは、ØMQ の再接続する機能や、負荷分散などの機能を持っており、通常の TCP ソケットに比べて十分優れています。しかし現実にはこれだけでは不十分でしょう。リクエスト・応答パターンにおいて信頼性があると言える唯一の状況は、同一プロセスにある 2 つのスレッドでこれを行う場合のみでしょう。

しかし、分散ネットワークの基礎となる幾つかの工夫を行うことで、私達が「海賊パターン」と呼んでいる信頼性のあるリクエスト応答パターンを実現する事が出来ます。

私の経験によると、サーバーからクライアントに接続する方法は 3 つに分類され、信頼性を高める方法はそれぞれ異なります。

- 複数のクライアントが単一のサーバーと直接通信する場合。考えられる障害はサーバーの再起動やクラッシュ、ネットワークの切断です。
- 複数のクライアントがブローカーなどのプロキシを経由して複数のワーカーと通信する場合。これはサービス指向のトランザクションを処理する場合などに使われます。考えられる障害はワーカーの再起動やクラッシュ、ワーカーのビジーループ、ワーカーの高負荷、キューのクラッシュや再起動、ネットワークの切断です。

- 複数のクライアントが複数のサーバーとプロキシを経由せずに通信する場合。名前解決などによるサービスの分散方法です。考えられる障害はサービスのビジーループ、サービスの高負荷、ネットワークの切断です。

これらの方法にはそれぞれ利点と欠点があり、時にはこれらが組み合わさる場合もあるでしょう。これら 3 つについて詳しく見ていきます。

4.3 クライアント側での信頼性 (ものぐさ海賊パターン)

クライアントにちょっとした工夫を行うことで、リクエスト・応答パターンの信頼性を高めることが可能です。そのためにはブロッキングで受信を行うのではなく、非同期で以下のことを行います。

- REQ ソケットを監視して、間違いなくメッセージが到着している場合のみ受信を行う。
- 一定の時間応答が返ってこない場合はリクエストを再送信する。
- 何度かリクエストを送信しても応答が返ってこなかった場合は諦めます。

REQ ソケットを利用して厳密に送信・受信の順序を守らなかった場合はエラーが発生します。技術的に説明すると、REQ ソケットは有限オートマトンとして実装されていて送受信を行うことで状態遷移を行います。そして異常な遷移が行われるとエラーコード「EFSM」を返します。ですから、REQ ソケットを利用してこの海賊パターンを行う場合、応答を受け取る前に送信する可能性があるので、ちょっと面倒な事が起こります。

この問題の強引で手っ取り早い解決方法は、REQ ソケットでエラーが発生したら、一旦クローズして再接続する事です。

lpclient.c: ものぐさ海賊クライアント

```
// ものぐさ海賊クライアント
// 安全にリクエスト・応答を行うために zmq_poll を利用してください
// サーバーの lpserver はランダムに異常終了します

#include "czmq.h"
#define REQUEST_TIMEOUT     2500    // ミリ秒 (> 1000!)
#define REQUEST_RETRIES     3       // リトライ回数
#define SERVER_ENDPOINT     "tcp://localhost:5555"
```

```
int main (void)
{
    zctx_t *ctx = zctx_new ();
    printf ("I: connecting to server...\n");
    void *client = zsocket_new (ctx, ZMQ_REQ);
    assert (client);
    zsocket_connect (client, SERVER_ENDPOINT);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    while (retries_left && !zctx_interrupted) {
        // リクエストを送信し、応答を受信します
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            // 応答ソケットを監視し、指定した時間でタイムアウトします
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          // 割り込み

            // ここではサーバーの応答を処理します。
            // 応答が返ってこなければリクエストを再送信し、一定数繰り返
            // したら諦めます。

            if (items [0].revents & ZMQ_POLLIN) {
                // サーバーからの応答を受信、シーケンス番号の一致を確認します
                char *reply = zstr_recv (client);
                if (!reply)
                    break;      // 割り込み
                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");
                // 古いソケットを閉じて再接続します。
                zsocket_destroy (ctx, client);
                printf ("I: reconnecting to server...\n");
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, SERVER_ENDPOINT);
                // 新しいソケットでリクエストを再送信します
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

こちらのサーバーも実行してください。

lpserver.c: ものぐさ海賊サーバー

```
// ものぐさ海賊クライアント
// 安全にリクエスト・応答を行うために zmq_poll を利用してください
// サーバーの lpserver はランダムに異常終了します

#include "czmq.h"
#define REQUEST_TIMEOUT     2500    // ミリ秒 (> 1000!)
#define REQUEST_RETRIES     3       // リトライ回数
#define SERVER_ENDPOINT     "tcp://localhost:5555"

int main (void)
{
    zctx_t *ctx = zctx_new ();
    printf ("I: connecting to server...\n");
    void *client = zsocket_new (ctx, ZMQ_REQ);
    assert (client);
    zsocket_connect (client, SERVER_ENDPOINT);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    while (retries_left && !zctx_interrupted) {
```

```
        // リクエストを送信し、応答を受信します
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            // 応答ソケットを監視し、指定した時間でタイムアウトします
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          // 割り込み

            // ここではサーバーの応答を処理します。
            // 応答が返ってこなければリクエストを再送信し、一定数繰り返
            // したら諦めます。

            if (items [0].revents & ZMQ_POLLIN) {
                // サーバーからの応答を受信、シーケンス番号の一致を確認します
                char *reply = zstr_recv (client);
                if (!reply)
                    break;      // 割り込み
                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");
                // 古いソケットを閉じて再接続します。
                zsocket_destroy (ctx, client);
                printf ("I: reconnecting to server...\n");
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, SERVER_ENDPOINT);
```

```
                // 新しいソケットでリクエストを再送信します
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

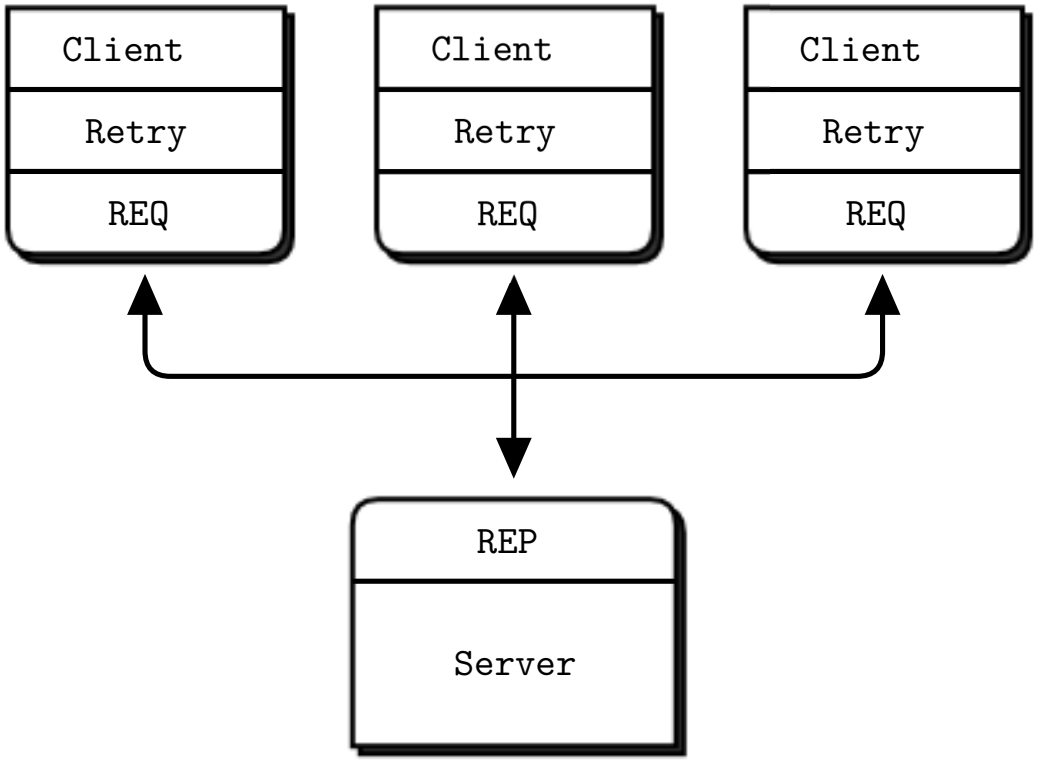


図 4.1　ものぐさ海賊パターン

このサンプルコードを実行するには、ターミナルを 2 つ立ち上げてクライアントとサーバーを起動します。このサーバーはランダムに障害をシミュレートし、以下の様なメッセージを出力します。

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

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

そして以下はクライアントの出力です。

```
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
```

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.

クライアントは応答メッセージのシーケンス番号を見て、メッセージが失われていないかどうかを確認しています。このメカニズムが期待通り動作しているか確信が持てるまでこのサンプルコードを何度でも実行してみてください。実際のアプリケーションではシーケンス番号は必要ありません、ここでは設計の正しさを確かめるために利用してるだけです。

クライアントは REQ ソケットを利用していますので、送受信の順序を守るために強制的にソケット閉じて再接続を行っています。ここで REQ ソケットの代わりに DEALER ソケットを使おうと考えるかもしれませんが、これはあまり良くありません。まず、REQ ソケットのエンベロープを模倣するのが面倒ですし、期待しない応答が返ってくる可能性があります。

これは複数のクライアントから単一のサーバーに対して通信する場合のみに適用できる障害対策であり、サーバーがクラッシュした場合は自動的に再起動することを期待しています。例えばハードウェア障害や電源供給が断たれるなどの恒久的なエラーが発生した場合はこの対策では不十分です。一般的にアプリケーションコードは障害の原因になりやすいので単一のサーバーに依存したアーキテクチャー自体があまり良くありません。

利点と欠点をまとめます。

- 利点: 理解しやすくて実装が簡単。
- 利点: サーバーアプリケーションの改修は必要なく、クライアント側の変更もわずかです。
- 利点: 接続が成功するまで ØMQ が自動的に再接続を行ってくれます。
- 欠点: 代替のサーバーにフェイルオーバーしません。

4.4 信頼性のあるキューイング (単純な海賊パターン)

2 番目に紹介する方法は複数のサーバーと透過的に通信を行うキュープロキシを用いてものぐさ海賊パターンを拡張します。まずは単純な海賊パターンが最低限動作する小さなモデルで実装していきます。

全ての海賊パターンにおいて、ワーカーはステートレスで動作します。もしアプリケーションがデーターベースなどに状態を保存したい場合でもメッセージングフレームワークはこれに関知しません。キュープロキシはクライアントについて何も知らずにやってくるメッセージをそのまま転送するだけの役割を持っています。こうした方がワーカーが落ちてしまった場合でも別のワーカーにメッセージを渡すだけで良いので都合が良いのです。これはなかなか単純で良いトポロジーですが中央キューが単一故障点になってしまうという欠点があります。

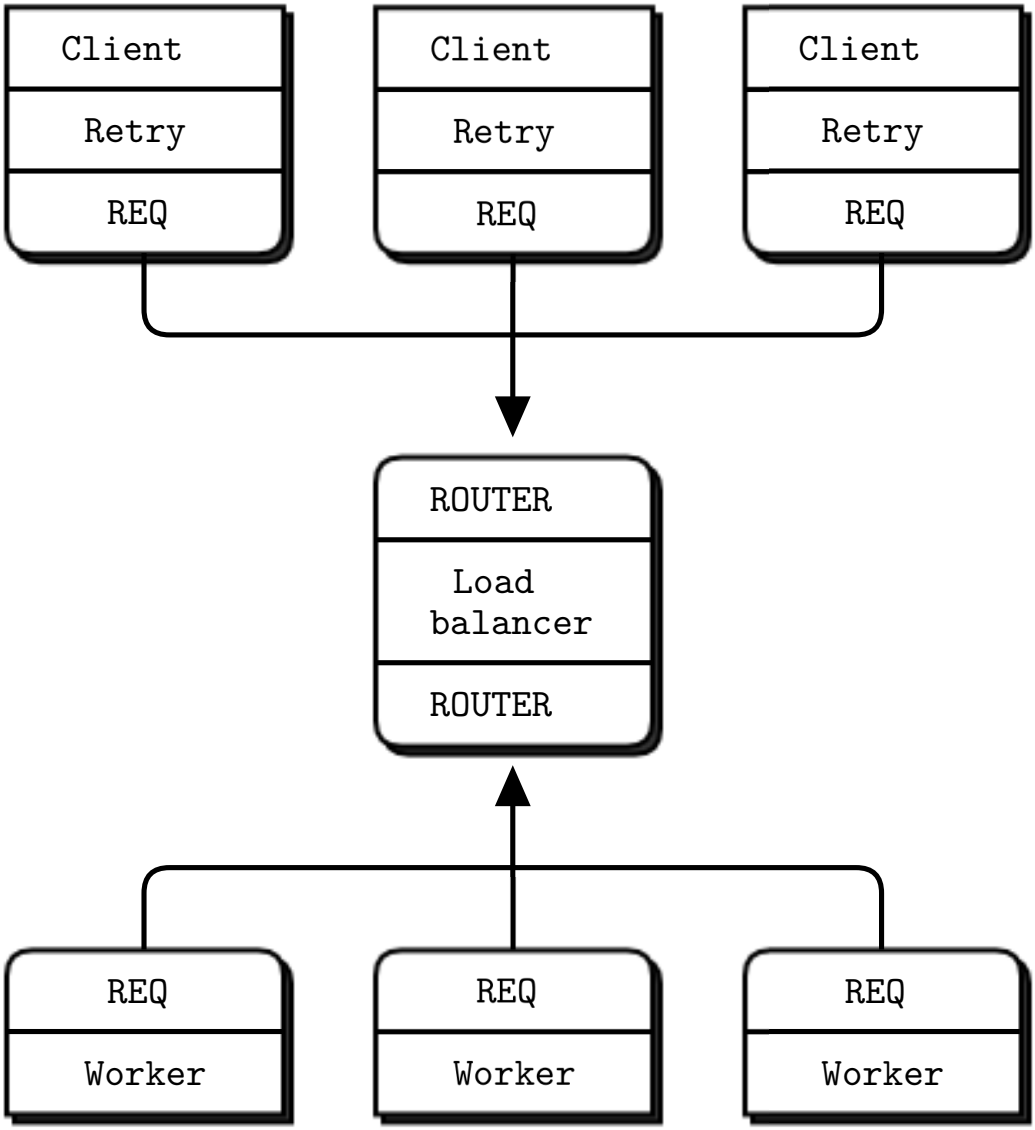


図 4.2　単純な海賊パターン

キュープロキシの基本的な仕組みは第 3 章「リクエスト・応答パターンの応用」で紹介した負荷分散ブローカーと同じです。ワーカーが落ちたりブロックしたりする障害に対して、どの様な対応を最低限行う必要があるでしょうか? クライアントには再試行が実装されていますので、負荷分散パターンが効果的に機能します。これはまさしく ØMQ の哲学に適合し、中間にプロキシを介する事で P2P パターンに拡張することが可能です。

これには特別なクライアントは必要ありません。先程のものぐさ海賊パターンと同じクライアントを利用します。こちらが負荷分散ブローカーと同等の機能を持ったキュープロキシのコードです。

spqueue.c: 単純な海賊ブローカー

```
// 単純な海賊ブローカー
// 信頼性のメカニズム以外は負荷分散パターンと同じです。

#include "czmq.h"
#define WORKER_READY   "\001"      // 準備完了シグナル

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");    // クライアント用
    zsocket_bind (backend,  "tcp://*:5556");    // ワーカー用

    // ワーカーキュー
    zlist_t *workers = zlist_new ();

    // この辺りのコードは lbbroker2 とまったく同じです
    while (true) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        // ワーカーが存在する場合のみフロントエンドソケットを監視します
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              // 割り込み

        // ワーカーとの通信を処理します
        if (items [0].revents & ZMQ_POLLIN) {
            // ワーカーの ID を負荷分散キューに追加します
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          // 割り込み
            zframe_t *identity = zmsg_unwrap (msg);
            zlist_append (workers, identity);

            // 準備完了通知でなければクライアントに転送します
            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) {
            // クライアントからのリクエストをワーカーにルーティングします
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    // 終了処理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

こちらがワーカーのコードです。ものぐさ海賊パターンのサーバーと同じような仕組みを負荷分散ブローカーに組み込んでいます。

spworker.c: 単純な海賊ワーカー

```
// 単純な海賊ワーカー
// REQ ソケットを利用してブローカーの tcp://*:5556 に接続します
// 負荷分散を行うワーカーの実装です

#include "czmq.h"
#define WORKER_READY   "\001"      // 準備完了シグナル

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);

    // 動作を理解しやすくするためにランダムな ID を設定します
    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");
```

```
    // ブローカーに準備完了を通知
    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;              // 割り込み

        // 数サイクル動作したら障害をシミュレートします
        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);
            sleep (3);
            if (zctx_interrupted)
                break;
        }
        printf ("I: (%s) normal reply\n", identity);
        sleep (1);              // 何らかの重い処理
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return 0;
}
```

これをテストするには幾つかのワーカーとものぐさ海賊クライアント、およびキュープロキシを起動してやります。順序はなんでも構いません。そうするとワーカーがクラッシュしたり固まったりするでしょうが、キュープロキシは機能を停止することなく動作し続けます。このモデルはクライアントやワーカーの数が幾つでも問題なく動作します。

4.5 頑丈なキューイング (神経質な海賊パターン)

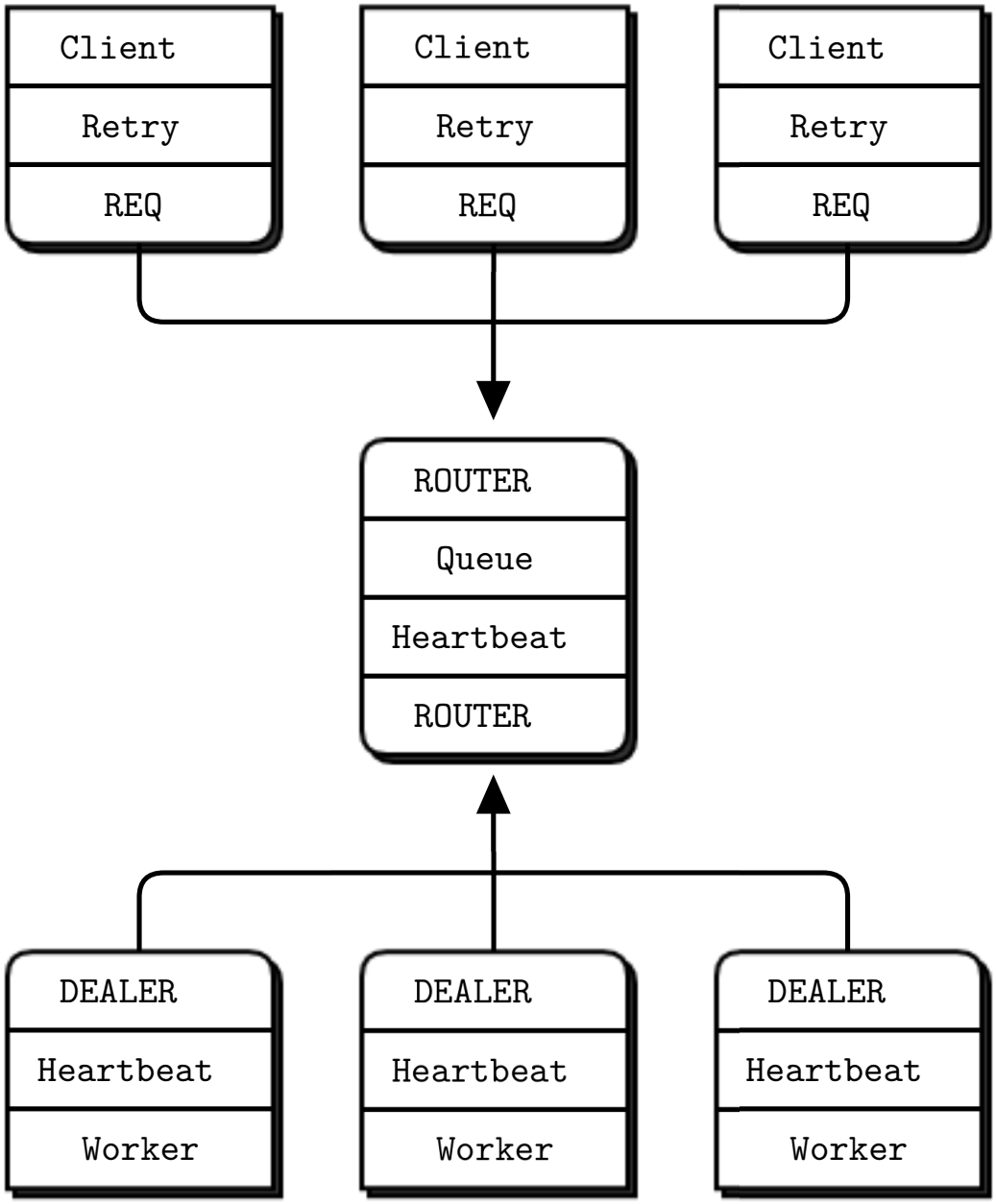


図 4.3　神経質な海賊パターン

単純な海賊パターンはものぐさ海賊パターンと組み合わせて上手く機能する障害対策でしたが、これには欠点があります。

- キューの再起動やクラッシュに対して堅牢ではありません。またクライアントは自動的に復旧しますがワーカーはそうではありません。ワーカーの ØMQ ソケットは自動的に再接続を行ってくれますが、準備完了メッセージを送信していませんのでメッセージが送られてきません。これを修正するにはキュープロキシからワーカーに対してハートビートを送って、ワーカーの存在を確認する必要があります。
- キュープロキシはワーカーの障害を検知できないため、待機中のワーカーが落ちてしまった場合にワーカーキューから該当のワーカーを削除することが出来ません。存在しないワーカーに対してメッセージを送信すると、クライアントは待たされてしまうでしょう。これは致命的な問題ではありませんが良くもありません。これを上手く動作させるには、ワーカーからキューに対してハートビートを送るワーカーの障害をキューがワーが検知できるようにするよ良いでしょう。

これらの欠点を神経質な海賊パターンで修正します。

これまでのワーカーは REQ ソケットを利用してきましたが、この神経質な海賊パターンのワーカーは DEALER ソケットを利用します。これにより、送受信の順序に拘らずにいつでもメッセージを送受信出来るというメリットがあります。デメリットはメッセージエンベロープを管理しなければならない事です。これについては第 3 章の「リクエスト・応答パターンの応用」で既に説明しました。

今回もまたものぐさ海賊パターンのクライアントを使いまわします。こちらは神経質な海賊キュープロキシです。

ppqueue.c: 神経質な海賊キュー

```
// 神経質な海賊キュー

#include "czmq.h"
#define HEARTBEAT_LIVENESS  3       // 3〜5 が妥当
#define HEARTBEAT_INTERVAL  1000    // ミリ秒

// 神経質な海賊プロトコルの定数
#define PPP_READY       "\001"      // 準備完了シグナル
#define PPP_HEARTBEAT   "\002"      // ハートビートシグナル

// ここではワーカークラス (生成処理、開放処理、各メソッド) を定義します。
typedef struct {
    zframe_t *identity;         //  Identity of worker
    char *id_string;            //  Printable identity
    int64_t expiry;             //  Expires at this time
```

```
} worker_t;

// 新しいワーカーの作成
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;
}

// ワーカーオブジェクトの開放処理を行います

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;
    }
}

// この関数はワーカーをリストの最後に追加します

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);
}
```

```
// この関数は次に有効なワーカーの ID を返します。

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;
}

// この関数は有効期限の切れたワーカーをリストから削除します。

static void
s_workers_purge (zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              // 有効期限内であればそのままで

        zlist_remove (workers, worker);
        s_worker_destroy (&worker);
        worker = (worker_t *) zlist_first (workers);
    }
}

// メインタスクは負荷分散を行い、ハートビートを送信してワーカーのクラッ
// シュを検知します。

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");    // クライアント用
    zsocket_bind (backend,  "tcp://*:5556");    // ワーカー用

    // 有効なワーカーのリスト
    zlist_t *workers = zlist_new ();

    // ハートビートの間隔を設定します
```

```
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    while (true) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        // 有効なワーカーが存在する場合のみフロントエンドを監視します
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1,
            HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              // 割り込み

        // バックエンドからのイベントを処理
        if (items [0].revents & ZMQ_POLLIN) {
            // 負荷分散を行う為にワーカーの ID を利用します
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          // 割り込み

            // ワーカーからのメッセージは準備が出来ている事を意味します
            zframe_t *identity = zmsg_unwrap (msg);
            worker_t *worker = s_worker_new (identity);
            s_worker_ready (worker, workers);

            // 制御メッセージの検証、もしくはクライアントへの応答を行う
            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) {
            // クライアントからのリクエストを受信し、ワーカーにルーティングします
            zmsg_t *msg = zmsg_recv (frontend);
            if (!msg)
                break;          // 割り込み
            zmsg_push (msg, s_workers_next (workers));
            zmsg_send (&msg, backend);
```

```
        }

        // ソケットの受信処理を行った後に、ハードビートの処理を行います。
        // まず、全てのワーカーに対してハートビートを送信します。
        // 続いて、有効期限の切れたワーカーを排除します。

        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);
    }
    // 終了処理
    while (zlist_size (workers)) {
        worker_t *worker = (worker_t *) zlist_pop (workers);
        s_worker_destroy (&worker);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

このキュープロキシは負荷分散パターンを拡張してワーカーに対してハートビートを送信しています。ハートビートは単純な機能ですが、正しくこれを行うのは難しいので後ほど詳しく説明します。

以下は神経質な海賊パターンのワーカーです。

ppworker.c: 神経質な海賊ワーカー

```
// 神経質な海賊ワーカー

#include "czmq.h"
#define HEARTBEAT_LIVENESS  3       // 3〜5 が妥当
#define HEARTBEAT_INTERVAL  1000    // ミリ秒
#define INTERVAL_INIT       1000    // 最初の再接続間隔
#define INTERVAL_MAX       32000    // 最大の再接続間隔
```

```
// 神経質な海賊プロトコルの定数
#define PPP_READY       "\001"      // 準備完了シグナル
#define PPP_HEARTBEAT   "\002"      // ハートビートシグナル

// 神経質な海賊キューに接続を行うヘルパー関数

static void *
s_worker_socket (zctx_t *ctx) {
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");

    // 準備完了シグナルを送信
    printf ("I: worker ready\n");
    zframe_t *frame = zframe_new (PPP_READY, 1);
    zframe_send (&frame, worker, 0);

    return worker;
}

// これは神経質な海賊プロトコル (PPP) のワーカー側の実装です。
// ここで重要な点は、キューの障害を検知するためのハートビートを行ってい
// る所です。

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = s_worker_socket (ctx);

    // liveness が 0 になるとキューから切断されたと見なします
    size_t liveness = HEARTBEAT_LIVENESS;
    size_t interval = INTERVAL_INIT;

    // ハートビートの送信間隔
    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;              // 割り込み

        if (items [0].revents & ZMQ_POLLIN) {
```

```
            // メッセージの取得
            //  - 3 フレーム (envelope + content) であれば -> リクエスト
            //  - 1 フレーム (HEARTBEAT) であれば -> ハートビート
            zmsg_t *msg = zmsg_recv (worker);
            if (!msg)
                break;          // 割り込み

            // キュー側の頑丈さをテストする為にこの様にワーカー側でクラッ
            // シュや遅延などの障害をシミュレートします。
            // 最初の数サイクルは正常に動作し、しばらくすると障害を発生
            // させます。

            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");
                    sleep (3);
                    if (zctx_interrupted)
                        break;
                }
                printf ("I: normal reply\n");
                zmsg_send (&msg, worker);
                liveness = HEARTBEAT_LIVENESS;
                sleep (1);              // 何らかの処理
                if (zctx_interrupted)
                    break;
            }
            else
            // キューからのハートビートメッセージを受信したということは
            // キューは正常に動作していますので、liveness をリセットする
            // 必要があります。
            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
        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:

        // 一定の間、キューがハートビートを送信していない場合はソケット
        // を破棄して再接続を行います。
        // この方法は単純ですが送信中のメッセージは破棄されますので荒っ
        // ぽいです。

        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;
        }
        // 一定の間隔でキューにハートビートを送信します
        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;
}
```

このコードを解説すると、

- このコードは以前と同じく、障害をシミュレートするコードが入っています。
- このワーカーはものぐさ海賊パターンのクライアントと同様に再試行を行う戦略です。

再試行の間隔を指数的に増やしていき何度でも再試行を行います。

以下のスクリプトでこれらのコードを実行してみてください。

```
ppqueue &
for i in 1 2 3 4; do
    ppworker &
    sleep 1
done
lpclient &
```

これを実行すると、ワーカーがひとつずつクラッシュして終了していくことを確認できるでしょう。キュープロキシを再起動した場合でもワーカーは再接続して動作を継続し、ワーカーが 1 つでも動いていればクライアントは正しい応答を受け取る事が出来るでしょう。

4.6 ハートビート

ハートビートは相手が生きているか死んでいるかを知るための手段です。これは ØMQ 固有の概念ではありません。TCP のタイムアウト時間は約非常に長く、大抵 30 分程度が設定されています。これでは相手が生きているのか死んでいのるか、それともプラハに行って酒を飲んでいるか判断することは出来ません。

ハートビートを正しく実装するのは簡単なことではありません。神経質な海賊パターンのコードではリクエスト・応答のロジックは 10 分程度で実装できましたがハートビートを正しく動作させるためには 5 時間程度掛かりました。

それでは、ØMQ の利用者がハートビートを実装する際に直面する 3 つの問題を見て行きましょう。

4.6.1 Shrugging It Off

ØMQ アプリケーションのほとんどはハートビートを行いません。その場合どの様な問題が起こるでしょうか?

- アプリケーションが ROUTER ソケットを利用して接続を中継している場合、接続と切断を繰り返す度にメモリリークが発生します。そしてだんだん遅くなっていくでしょう。
- SUB ソケットや DEALER ソケットを利用してメッセージを受信する側は、データが送られてこない事が正常なのか異常なのかを判断することが出来ません。接続相手の異常

を検知できれば別の相手に切り替えることが出来ます。

- TCP で接続している場合、長い時間無通信が続くと接続が切られてしまう場合があります。この問題を避けるには「keep-alive」データを送信することで接続を継続することが出来ます。

4.6.2 片側ハートビート

2 番目の選択肢は片方のノードからもう片方のノードへ 1 秒に 1 回位の間隔でハートビートを送る方法です。応答が返らずにタイムアウトが発生した場合 (一般的に数秒間)、その相手は落ちたと見なします。これで本当に良いのでしょうか? これは上手く動作することもありますが、うまく行かない場合もあります。

pub-sub パターンではこの方法が使える唯一の方法です。SUB ソケットは PUB ソケットに対して話しかけることは出来きません、一方、PUB ソケットはサブスクライバーに対して「私は生きています」というメッセージを送信できます。

無駄をなくす為には、実際に送信すべきデータがない場合のみハートビートを送信すると良いでしょう。また、ネットワークが貧弱な場合 (例えばモバイルネットワークでバッテリーを節約したい場合) はハートビートの間隔を出来るだけ遅くするのが良いでしょう。サブスクライバーが障害を検知できさえすれば良いのです。

この設計の問題点を挙げると、

- 大量のデータが送信されている場合ハートビートのデータが遅延してしまい、不正確になる可能性があります。ハートビートが遅延してしまうとタイムアウトが発生して接続が切れてしまいます。従って受信者はハートビートを受信するかどうかに関わらず、全てのデータをハートビートとして扱う必要があります。
- 受信者が居なくなった場合、PUSH ソケットや DEALER ソケットであれば送信キューにキューイングされるのですが、pub-sub パターンの場合はメッセージを喪失してしまいます。ですのでハートビートの送出間隔以内に受信者が再起動を行った場合、ハートビートは全て受け取っていますが、メッセージは取りこぼしている可能性があります。
- この設計ではハートビートのタイムアウト時間は全て同じである事を前提にしています。しかしそれでは困る場合があります。素早く障害を検知したいノードに対しては積極的なハートビートを行い、電力消費を抑えたいノードに対しては控えめなハートビートを行いたいという事もあるでしょう。

4.6.3 PING-PONG ハートビート

3 番目の方法はピンポンのやりとりを行うことです。一方が PING コマンドを送信し、受信者は PONG コマンドを返信します。2 つのコマンドの相関性を確認するために、両方のコマンドはデータ部を持っています。ノードは「クライアント」とか「サーバー」といった役割を持っているかもしれませんが、基本的にどちらが PING を送信して、どちらが PONG を応答するかは任意です。しかしながらタイムアウトはネットワークのトポロジーに依存していますので、動的なクライアントが PING を行い、サーバーが PONG を返すのが適切でしょう。

これは ROUTER ソケットを使ったブローカーで上手く動作します。2 番目の方法で紹介した最適化はここでも有効です。送信者は実際に送信すべきデータがない場合のみ PING を送信し、受信者は全てのデータを PONG として扱う事です。

4.6.4 神経質な海賊パターンでのハートビート

先ほど説明した神経質な海賊パターンでは 2 番目の方法でハートビートを行いました。これは単純ではありますが、今となっては PING-PONG ハートビートを使ったほうが良いでしょう。基本的な所は前と同じです。双方向にハートビートメッセージを非同期で送信し、お互いに相手が落ちているかどうか確認することが出来ます。

キューブローカーからのハートビートをワーカーが処理するには、

- ハートビートのタイムアウトが何回発生したら相手が落ちたと判断するかという基準 (liveness) を決定します。ここでは 3 回を設定します。
- `zmq_poll` ループではハートビートの間隔の 1 秒間ブロックしてメッセージを待ちます。
- ハートビートに限らず、何らかのメッセージが届いたら `liveness` を 3 にリセットします。
- メッセージが届かずタイムアウトした場合に `liveness` をカウントダウンします。
- `liveness` が 0 になった時、キューブローカーに障害が発生したと判断します。
- 障害を検知したらソケットを破棄し、再接続を試みます。
- 大量のソケットが再接続を繰り返すのを避けるために sleep を行っています。これは再接続の度に 2 倍され、最大 32 秒まで増えます。

そしてキューブローカーがハートビートを処理する流れは以下の通りです。

- 次のハートビートを送信する時間を決定します。これは通信相手が 1 つの場合は単一の変数です。
- `zmq_poll` ループの中でこの時間を経過した場合にハートビートを送信します。

こちらがワーカーがハートビートを行う主要なコードです。

```
#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
    }
}
```

キューブローカー側もだいたい同じですがワーカー毎に有効時間を管理する必要があります。

以下はハートビートを実装する上でのアドバイスです。

- アプリケーションのメインループでは、zmq_poll かリアクターを使用して下さい。

- ハートビートを実装できたら、まず障害をシミュレートしてテストしてください。そしてその他のメッセージ処理を実装するのが良いでしょう。後からハートビートを実装するのは非常に難い事です。
- ターミナルに出力するなどして簡単に動作確認してください。メッセージの流れを追うには、zmsg が提供する dump 関数が役立ちます。これで想定通りのメッセージが流れているか確認しましょう。
- 実際のアプリケーションではハートビート間隔は設定で記述するか、ネゴシエートすべきでしょう。特定の接続相手には 10 ミリ秒程度の積極的なハートビートを行いたい事もあるでしょうし、30 秒程度の長い間隔で行いたい事もあります。
- ハートビート間隔が接続相手毎に異なる場合、`zmq_poll` のポーリング間隔はこれらの中で最短の間隔である必要があります。また、無限のタイムアウトを設定してはいけません。
- 実際のデータ通信と同じソケットでハートビート行って下さい。ハートービートはネットワークコネクションを維持するためのキープアライブとしての役割もあります。(不親切なルーターは通信が行われていない接続を切ってしまう事があるからです)

4.7 規約とプロトコル

ここまで注意深く読んできた読者は、神経質な海賊パターンと単純な海賊パターンを相互運用できないことに気がついたでしょう。しかし「相互運用」とはどの様なものでしょうか? 相互運用を保証するためには、異なる時間や場所で動作しているコードが協調して動作するための規約に同意する必要があります。これを私達は「プロトコル」と呼びます。

仕様の無いプロトコルで実験することは楽しいことですが、実際のアプリケーションでこれを行うのは賢明な判断とは言えません。例えばワーカーを別のプログラミング言語で書きたい場合はどうしますか? コードを読んで動作を調べますか? プロトコルを変更したい時はどうしますか? 元々は単純なプロトコルであっても、プロトコルが普及するに従って進化し、複雑になっていきます。

規約の欠如はアプリケーションが使い捨てにされる兆候です。というわけでプロトコルとしての規約を書いてみましょう。

公開された ØMQ の規約を集めた rfc.zeromq.org という Wiki サイトがあります。新しい仕様を作成するには、wiki のアカウントを登録して書かれている手順に従って下さい。技術文書を書くのが得意ではない方もいると思いますが、これはとっても簡単な事ですよ。

海賊パターンの仕様を書くのに 15 分程度掛かりました。これは大きな仕様ではありませんが、基本的な振る舞いを理解するためには十分です。このキューは神経質な海賊パターンと互

換性がありませんので必要に応じて修正して下さい。

実際の神経質な海賊パターンは以下のように動作します。

- プロトコルバージョンを区別できるように、READY コマンドでプロトコルバージョンを通知すべきでしょう。
- すでに、READY とハートビートコマンドというまったく異なるメッセージ種別が存在します。これらを区別するためにメッセージ構造に「メッセージ種別」を含める必要があるでしょう。

4.8 信頼性のあるサービス試行キューイング (Majordomo パターン)

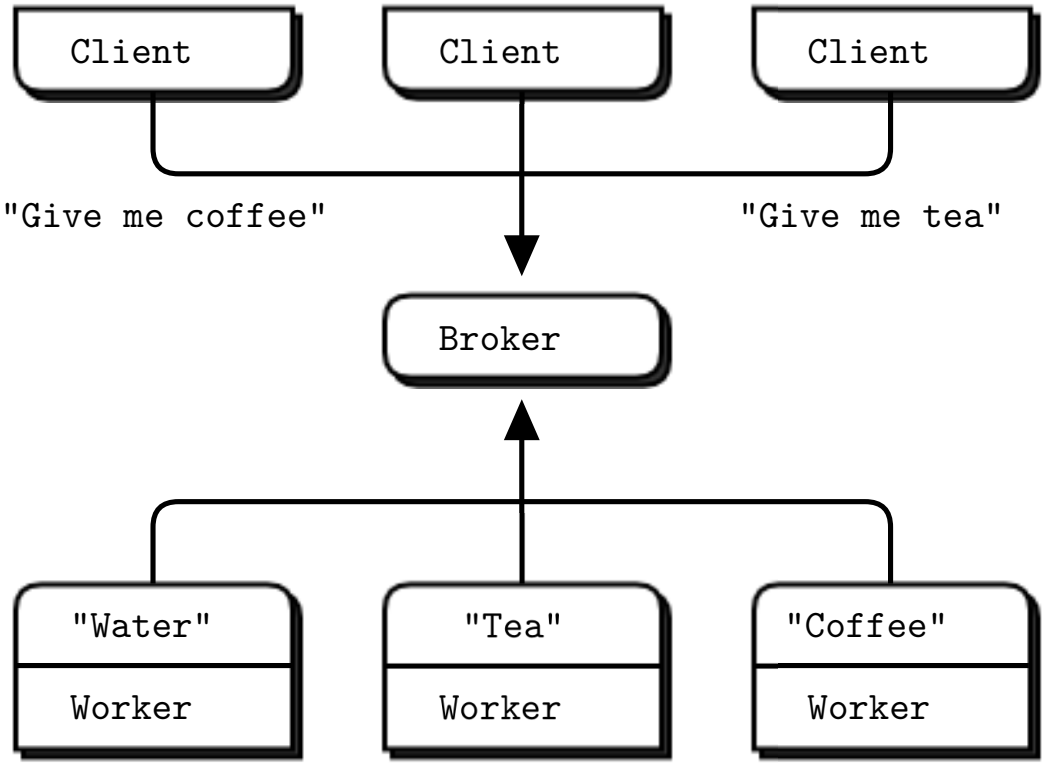


図 4.4　Majordomo パターン

Majordomop プロトコルには信頼性を高める効果もあります。この様な複雑なアーキテクチャを実装するには、まずプロトコル仕様書を書く必要があります。

Majordomo プロトコルは面白い方法で神経質な海賊パターンを拡張します。クライアント

が送信するリクエストに「サービス名」を付加し、特定のサービスに登録されたワーカーに振り分けられます。神経質な海賊キューにサービス名を追加するとサービス指向ブローカーになります。このパターンの良い所は、単純なプロトコル (神経質な海賊パターン) を元にしているため既存の問題が既に解決されていることと、実際に動作するコードが公開されていることです。この仕様は簡単に書くことが出来ました。

Majordomo プロトコルを実装するには、クライアントとワーカーのフレームワークを用意する必要があるります。全てのアプリケーション開発者がプロトコル仕様書を読んで理解するのは難しいことですから簡単に利用できる API を用意するのが良いでしょう。

1 つ目の仕様書では、分散アーキテクチャーの部品同士が通信を行う方法を定義します。そして 2 つ目の仕様書ではユーザーアプリケーションがこれらのフレームワークと通信する方法を定義します。

Majordomo プロトコルはクライアント側とワーカー側の 2 種類に分かれますので 2 つの API が必要です。こちらは単純なオブジェクト指向を利用して設計したクライアント側の API です。

```
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);
```

これだけです。ブローカとのセッションを張り、リクエストを送信して応答を受け取って接続を切っています。こちらはワーカー側の API です。

```
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);
```

これは対称的なコードに見えるかもしれませんが、ワーカー側のやりとりにちょっとした違いがあります。初回の recv() 呼び出しでは null を受け取り、続いてリクエストを受信します。

The client and worker APIs were fairly simple to construct because they're heavily based on the Paranoid Pirate code we already developed. Here is the client API: クライアントとワーカーは既に実装済みの神経質な海賊パターンのコードを流用する事で、今回の API はとても簡単に設計することができました。

mdcliapi.c: Majordomo クライアント API

```
// mdcliapi クラス - Majordomo プロトコル クライアント API
// MDP/Client 仕様 (http://rfc.zeromq.org/spec:7) を実装しています
```

```
#include "mdcliapi.h"

// クラスの構造体
// これらのプロパティにはメソッドを経由してアクセスします。

struct _mdcli_t {
    zctx_t *ctx;                // コンテキスト
    char *broker;
    void *client;               // ブローカーへのソケット
    int verbose;                // 標準出力へのロギング
    int timeout;                // タイムアウト
    int retries;                // リトライ回数
};

// ブローカーに接続、もしくは再接続を行います

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);
}

// コンストラクタとデストラクタを定義します。

// コンストラクタ

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;           // ミリ秒
    self->retries = 3;              // 最大リトライ回数

    s_mdcli_connect_to_broker (self);
    return self;
```

```
}

// デストラクタ

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;
    }
}

// リクエストのタイムアウトを設定します

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}

// リクエストのリトライ回数を設定します

void
mdcli_set_retries (mdcli_t *self, int retries)
{
    assert (self);
    self->retries = retries;
}

// これは送信メソッドです。ブローカーに対してリクエストを送信し、応答を
// 受信します。
// 送信が完了したら、メッセージオブジェクトを破棄します。
// 指定した回数のリトライを行っても応答を得られない場合は NULL を返します。

zmsg_t *
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
```

```
    zmsg_t *request = *request_p;

    // リクエストに付加するプロトコルフレーム
    // フレーム 1: "MDPCxy"(6 バイト MDP/Client x.y)
    // フレーム 2: "サービス名"
    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 }
        };

        // libzmq の API はブロッキング呼び出しでエラーが発生すると-1 を返
        // します。
        // 実際にはさらにエラーコードをチェックする必要がありますが、こ
        // れはサンプルコードですので Ctrl-C での割り込みのチェックだけで
        // 十分でしょう。

        int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;          // 割り込み

        // 応答を受信して処理します
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->client);
            if (self->verbose) {
                zclock_log ("I: received reply:");
                zmsg_dump (msg);
            }
            // 実際のコードではもう少し厳密に応答を検証した方が良いでしょう
            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;     // 成功
        }
        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;          // 諦める
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing client...\n");
    zmsg_destroy (&request);
    return NULL;
}
```

それではクライアント API を動かしてみましょう。こちらは 10 万回のリクエスト・応答のサイクルを実行するテストコードです。

mdclient.c: Majordomo クライアントアプリケーション

```
// Majordomo プロトコル クライアント
// mdcli API を利用してプロトコルを隠蔽しています

// ライブラリをリンクするのではなく、単純に include しています
#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;              // 割り込み、もしくはエラー
    }
    printf ("%d requests/replies processed\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

そしてこちらはワーカーの API です。

mdwrkapi.c: Majordomo ワーカー API

```
// mdwrkapi クラス - Majordomo プロトコル ワーカー API
// MDP/Worker 仕様 (http://rfc.zeromq.org/spec:7) を実装しています

#include "mdwrkapi.h"

// 信頼性のパラメータ
#define HEARTBEAT_LIVENESS  3       // 3-5 が妥当です

// ワーカー API は擬似的なオブジェクト指向で実装しています

// クラスの構造体
// これらのプロパティにはメソッドを経由してアクセスします

struct _mdwrk_t {
    zctx_t *ctx;                // コンテキスト
    char *broker;
    char *service;
    void *worker;               // ブローカーに接続するソケット
    int verbose;                // ログを標準出力に表示

    // ハートビートの管理
    uint64_t heartbeat_at;      // ハートビートを送信する時刻
    size_t liveness;            // 試行回数
    int heartbeat;              // ハートビートの間隔 (ミリ秒)
    int reconnect;              // 再接続の間隔 (ミリ秒)

    int expect_reply;           // 開始時は 0
    zframe_t *reply_to;         // 応答先の ID
};
```

```
// ここでは 2 つのユーティリティ (メッセージの送信関数とブローカとの再接続
// 関数) を定義します

// メッセージの送信関数
// メッセージが null の場合、内部で生成します

static void
s_mdwrk_send_to_broker (mdwrk_t *self, char *command, char *option,
                        zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    // プロトコル仕様に従ってエンベロープを積み重ねます
    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);
}

// 接続と再接続関数

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);

    // ブローカーにサービスを登録
    s_mdwrk_send_to_broker (self, MDPW_READY, self->service, NULL);

    // liveness が 0 になると、切断したと判断して再接続を行います。
    self->liveness = HEARTBEAT_LIVENESS;
    self->heartbeat_at = zclock_time () + self->heartbeat;
```

```
}

// mdwrk クラスのコンストラクタとデストラクタを定義します

// コンストラクタ

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;     // ミリ秒
    self->reconnect = 2500;     // ミリ秒

    s_mdwrk_connect_to_broker (self);
    return self;
}

// デストラクタ

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;
    }
}

// ワーカー API には 2 つの設定メソッドがあります。

// ハートビート間隔の設定

void
```

```
mdwrk_set_heartbeat (mdwrk_t *self, int heartbeat)
{
    self->heartbeat = heartbeat;
}

// 再接続間隔の設定

void
mdwrk_set_reconnect (mdwrk_t *self, int reconnect)
{
    self->reconnect = reconnect;
}

// これは受信メソッドですが、最初に送信を行ってから応答を待つので少し名
// 前が良くありません。もっと良い名前があれば私に教えて下さい。

// 応答を行い、次のリクエストを待ちます

zmsg_t *
mdwrk_recv (mdwrk_t *self, zmsg_t **reply_p)
{
    // フォーマットを整えて応答を送信します。
    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;              // 割り込み

        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->worker);
            if (!msg)
                break;          // 割り込み
            if (self->verbose) {
```

```
            zclock_log ("I: received message from broker:");
            zmsg_dump (msg);
        }
        self->liveness = HEARTBEAT_LIVENESS;

        // エラー処理ではなく assert にしておいてください
        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)) {
            // 正しくは、空フレームまでのフレームを保存する必要があ
            // りますが、ここでは 1 つしか見ていません

            self->reply_to = zmsg_unwrap (msg);
            zframe_destroy (&command);

            Here is where we actually have a message to process; we
            return it to the caller application:
            // ここでメッセージをアプリケーションに返します。

            return msg;
        }
        else
        if (zframe_streq (command, MDPW_HEARTBEAT))
            ;               // ハートビートに対してはなにもしない
        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);
        }
        // ハートビートを送信します
        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;
}
```

ワーカーの API を用いて echo サービスを実装するテストコードを見て見ましょう。

mdworker.c: Majordomo ワーカーアプリケーション

```
// Majordomo プロトコル ワーカー
// mdcli API を利用してプロトコルを隠蔽しています

// ライブラリをリンクするのではなく、単純に include しています。
#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;              // 割り込み
        reply = request;        // echo は簡単ですね :-)
    }
    mdwrk_destroy (&session);
    return 0;
}
```

ワーカー API について注意すべき点を挙げます。

- この API はシングルスレッドで動作します。つまりバックグラウンドスレッドでハートビートを送信するような事はしていません。
- この API は指数的な間隔で再試行を行いません。
- この API はエラー報告を行いません。必要に応じてアサーションや例外を投げたりすると良いでしょう。これは仮の参照実装ですので実際のアプリケーションでは不正なメッセージに対して堅牢でなくてはなりません。

ØMQ は通信相手が落ちた場合に自動的に再接続を行うにも関わらず、ワーカー API でソケットを手動で閉じて再接続している事を不思議に思うかもしれません。単純な海賊パターンや神経質な海賊ワーカーを振り返って見てもらえると解ると思いますが、ワーカーはブローカーが落ちた際に再接続を行いますがこれだけでは十分ではありません。これには 2 つの解決方法があります。ワーカーはハートビートを利用してブローカーが落ちたことを検知すると、ソケットを閉じて新しいソケットで再接続を行います。もうひとつの方法は、未知のブローカーからハートビートを受け取った場合に再登録を行うようにする事です。すなわちプロトコルでの対応が必要になります。

それでは Majordomo ブローカーを設計してみましょう。基本的な構造としてサービス毎にひとつのキューを持っていて、このキューはワーカーが接続した時に生成されます。

こちらがブローカーのコードです。

mdbroker.c: Majordomo ブローカー

```
// Majordomo プロトコル ブローカー
// http://rfc.zeromq.org/spec:7 and http://rfc.zeromq.org/spec:8
// に定義されている Majordomo プロトコルの最小実装です

#include "czmq.h"
#include "mdp.h"

// 通常は何処かに設定されているパラメーターです

#define HEARTBEAT_LIVENESS  3       // 通常は 3-5
#define HEARTBEAT_INTERVAL  2500    // ミリ秒
#define HEARTBEAT_EXPIRY    HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS

// ブローカークラスを定義します

typedef struct {
    zctx_t *ctx;                // コンテキスト
    void *socket;               // クライアント&ワーカーとの通信ソケット
```

```
    int verbose;                // 詳細ログ
    char *endpoint;             // bind するエンドポイント
    zhash_t *services;          // サービスのハッシュ
    zhash_t *workers;           // ワーカーのハッシュ
    zlist_t *waiting;           // 待機中のワーカーリスト
    uint64_t heartbeat_at;      // ハートビートの送信時刻
} 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);

// サービスクラスを定義します

typedef struct {
    broker_t *broker;           // ブローカーインスタンス
    char *name;                 // サービス名
    zlist_t *requests;          // クライアントリクエストのリスト
    zlist_t *waiting;           // 待機中のワーカーリスト
    size_t workers;             // 保有ワーカー数
} 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);

// ワーカークラス

typedef struct {
    broker_t *broker;           // ブローカーインスタンス
    char *id_string;            // ワーカーの ID
    zframe_t *identity;         // ID フレーム
    service_t *service;         // サービス (もしあれば)
```

```
    int64_t expiry;             // ハートビートの有効期限
} 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);

// ブローカーのコンストラクタとデストラクタ

static broker_t *
s_broker_new (int verbose)
{
    broker_t *self = (broker_t *) zmalloc (sizeof (broker_t));

    // ブローカーの状態を初期化
    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;
```

```
    }
}

// このメソッドはブローカーをエンドポイントとして bind します。
// これは複数回呼び出すことが出来ますが、今回はクライアントとワーカーで
// ひとつのソケットを使用します。

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);
}

// このメソッドはワーカーからのメッセージ (READY, REPLY, HEARTBEAT,
// DISCONNECT) を処理します。

static void
s_broker_worker_msg (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 1);     // 最低限コマンドが含まれているはず

    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)               // 初回のコマンドではない
            s_worker_delete (worker, 1);
        else
        if (zframe_size (sender) >= 4  // 予約済みのサービス名
        &&  memcmp (zframe_data (sender), "mmi.", 4) == 0)
            s_worker_delete (worker, 1);
        else {
            // ワーカーとサービスに登録する
            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) {
            // クライアントが応答したエンベロープを取り除き、プロトコル
            // ヘッダーとサービス名を挿入し、エンベロープを再ラップしま
            // す。
            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);
}

// これはクライアントからのリクエストを処理する関数です。ここでは MMI リ
// クエストを直接実装しています。(今の所 mmi.service しか実装していません)

static void
s_broker_client_msg (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 2);     // サービス名と本体

    zframe_t *service_frame = zmsg_pop (msg);
    service_t *service = s_service_require (self, service_frame);

    // 応答 ID を設定します
```

```
    zmsg_wrap (msg, zframe_dup (sender));

    // MMI サービスリクエストである場合は内部的に処理します。
    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));

        // クライアントが応答したエンベロープを取り除き、プロトコルヘッ
        // ダーとサービス名を挿入し、エンベロープを再ラップします。
        zframe_t *client = zmsg_unwrap (msg);
        zmsg_push (msg, zframe_dup (service_frame));
        zmsg_pushstr (msg, MDPC_CLIENT);
        zmsg_wrap (msg, client);
        zmsg_send (&msg, self->socket);
    }
    else
        // それ以外のメッセージは要求されたサービスに振り分けます
        s_service_dispatch (service, msg);
    zframe_destroy (&service_frame);
}

// このメソッドは一定時間 PING を送信していないワーカーを削除します。
// 古いワーカーから順に確認し有効なワーカーを見つけたらそこで処理を終わ
// ります。これはワーカーのリストが多くなった場合を考慮しています。

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;   // 有効期限内のワーカーが見つかれば処理を終わらせる
        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);
    }
}

// このメソッドはサービスに関する処理を実装しています。
// 名前からサービスを検索し、サービス名が無ければ新しくサービスを作成す
// る簡易コンストラクタです。

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;
}

// このサービスのデストラクタはサービスが削除される際に自動的に呼び出さ
// れます。

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);
}

// このメソッドはワーカーに対してリクエストを送信します。

static void
s_service_dispatch (service_t *self, zmsg_t *msg)
{
    assert (self);
    if (msg)                    // メッセージをキューングします
        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);
    }
}

// このメソッドはワーカーの処理を実装しています。
// ID からワーカーを検索し、まだ存在しなければ新しくワーカーを登録する簡
// 易コンストラクタです。

static worker_t *
s_worker_require (broker_t *self, zframe_t *identity)
{
    assert (identity);

    // self->workers から ID を検索します
    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;
}

// このメソッドはワーカーの削除を行います

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);

    // 暗黙的に s_worker_destroy を呼び出します
    zhash_delete (self->broker->workers, self->id_string);
}

// broker->workers からワーカーが削除される際にこのデストラクタが自動的
// に呼び出されます。

static void
s_worker_destroy (void *argument)
{
    worker_t *self = (worker_t *) argument;
    zframe_destroy (&self->identity);
    free (self->id_string);
    free (self);
}

// このメソッドはワーカーへのコマンドを整形します。
// 呼び出し側で、コマンドオプションとメッセージの本体を渡します。
```

```
static void
s_worker_send (worker_t *self, char *command, char *option, zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    // プロトコルエンベロープを付加します
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);

    // ルーティング用のエンベロープを付加します
    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);
}

// 待機中のワーカーの処理です

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);
}

// 最後にこちらがメインタスクです。
// まずブローカーインスタンスを作成し、ソケットのメッセージを処理します。

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");
```

```
    // 割り込みが行われるまで永遠にメッセージの処理を行います
    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;              // 割り込み

        // 入力メッセージがあれば処理を行います
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->socket);
            if (!msg)
                break;          // 割り込み
            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);
        }

        // 有効期限切れのワーカを切断し、削除します。
        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;
}
```

このサンプルコードはこれまで見てきた中で最も複雑な例でしょう。約 500 行もあり、これをちゃんと動作させるために 2 日もかかりました。しかし、完全なサービス指向ブローカーとしては短い方です。

このブローカー注意点は、

- Majordomo ブローカーはワーカーとクライアントの両方からの接続を 1 つのソケットで受け付けます。エンドポイントが 2 つあるより 1 つの方が管理上便利でしょう。
- このブローカーはハートビートや不正なコマンドの処理など、MDP/0.1 の仕様が全て実装されています。
- アーキテクチャが大規模になる場合、クライアントとワーカーの組み合わせに対してひとつのスレッドが対応するような、マルチスレッドに拡張することも可能です。これによりとても大規模なアーキテクチャを構築できます。
- ブローカーは状態を持たないのでプライマリー／バックアップ、あるいはプライマリー／プライマリーという信頼性モデルの構築は簡単です。最初にどちらのブローカーに接続するかはクライアントやワーカー次第です。
- このサンプルコードでは動作を確認しやすい様に、ハートビートを 5 秒間の間隔で行っています。実際のアプリケーションではもっと短い間隔の方が良いでしょうが、サービスを再起動する場合を考えて、再試行間隔は 10 秒以上にした方が良いでしょう。

後に私達はこの Majordomo プロトコルの拡張と改良を行い、GitHub プロジェクトで公開しました。ちゃんとした Majordomo スタックを利用したい場合は GitHub プロジェクトを見て下さい。

4.9 非同期の Majordomo パターン

前節では愚直な Majordomo の実装を紹介しました。クライアントは単純な海賊モデルをセクシーな API でラップしただけのものです。クライアントとブローカーとワーカーを 1 台のサーバーで稼働させると 14 秒間の間に 10 万リクエスト処理できるはずです。すなわち CPU

リソースのある限りメッセージフレームをコピー出来るという事です。しかし現実的にはネットワークのラウンドトリップが問題になります。ØMQ は Nagle のアルゴリズムを無効にしており、ラウンドトリップは非常に遅いものなのです。

理論は理論として素晴らしいものですが、実践には実践の良さがあります。簡単なテストプログラムを作成して実際のラウンドトリップを計測してみましょう。

1 つ目のテストは大量のメッセージをひとつずつ送信と受信を繰り返します。もうひとつのテストは、まず大量のメッセージを送信して、あとから全ての応答を受信します。両方のテストは同じことを行っていますが、異なるテスト結果が得られるでしょう。テストコードは以下の通りです。

tripping.c: ラウンド・トリップの計測

```
// ラウンドトリップの計測
// ビルドと実行を簡略化する為にこのサンプルはシングルプロセスで実装して
// います。
// クライアントタスクの処理が完了するとメインタスクに対して終了通知を送
// 信します。

#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");
}

// こちらはワーカータスクです。
// 受信したメッセージを送信源に送り返します。

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;
}

// こちらはブローカータスクです。
// zmq_proxy 関数を利用してフロントエンドとバックエンドを繋ぎます

static void *
broker_task (void *args)
{
    // コンテキストとソケットの準備を行います
    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;
}

// 最後にこちらがメインタスクです。クライアントとワーカー、ブローカーを
// 起動し、クライアントからの通知を受け取るまで実行します。

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;
}
```

私の開発サーバーでは以下の結果が得られました。

```
Setting up test...
Synchronous round-trip test...
 9057 calls/second
Asynchronous round-trip test...
 173010 calls/second
```

クライアントは計測を開始する前に少しだけ sleep する必要があることに注意して下さい。ルーティング先が存在しない場合は ROUTER ソケットはメッセージを捨てるという「機能」があるからです。この例では負荷分散パターンを使用していませんので、sleep を入れないとワーカーの接続に時間がかかってしまった場合にメッセージが失われてしまいます。これでは計測のテストになりません。

ご覧の通り、1 つ目のテストは 2 番目の非同期のテストよりもラウンドトリップの影響で 20 倍程度遅い事が判ります。それではこの非同期の実装を Majordomo パターンに適用してみましょう。

まず、API を送信と受信の 2 つの関数に別けます。

```
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);
```

ほんの数分の手間で同期式クライアントを非同期に書き換えることが出来ました。

mdcliapi2.c: Majordomo 非同期クライアント API

```
// mdcliapi2 クラス - Majordomo プロトコル クライアント API
// MDP/Client 仕様 (http://rfc.zeromq.org/spec:7) を実装しています

#include "mdcliapi2.h"

// クラスの構造体
// これらのプロパティにはメソッドを経由してアクセスします。

struct _mdcli_t {
    zctx_t *ctx;                // コンテキスト
    char *broker;
    void *client;               // ブローカーへのソケット
    int verbose;                // 標準出力へのロギング
    int timeout;                // タイムアウト
};

// ブローカーに対して接続、または再接続を行います。
// この非同期版では REQ ソケットの替りに DEALER ソケットを利用します。
// これにより、応答を待たずにいくつでも要求を送信することが出来ます。

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);
}

// コンストラクタとデストラクタはリトライパラメーターが無い事を除いて
// mdcliapi と同じです。
// コンストラクタ

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;           // ミリ秒

    s_mdcli_connect_to_broker (self);
    return self;
}

// デストラクタ

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;
    }
}

// リクエストタイムアウトの設定

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}

// これは非同期で単一のメッセージを送信するメソッドです。
// 今回 DEALER ソケットを利用していますので、エンベロープの区切りの空フレー
// ムを明示的に送信する必要があります。

int
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
```

```
    zmsg_t *request = *request_p;

    // リクエストに付加するプロトコルフレーム
    //  フレーム 0:空フレーム (REQ ソケットのエミュレーション)
    //  フレーム 1: "MDPCxy"(6 バイト MDP/Client x.y)
    //  フレーム 2: "サービス名"
    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;
}

// これは受信メソッドです。
// このメソッドは受信メッセージがあればメッセージを返し、無ければ NULL を
// 返します。
// ここではブローカーの障害から復旧しようとしていません。送信リクエスト
// を保存しておかなければリクエストを再送する事は不可能です。

zmsg_t *
mdcli_recv (mdcli_t *self)
{
    assert (self);

    // ソケットを監視します
    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;            // 割り込み

    // 受信メッセージがあれば処理します
    if (items [0].revents & ZMQ_POLLIN) {
        zmsg_t *msg = zmsg_recv (self->client);
        if (self->verbose) {
            zclock_log ("I: received reply:");
            zmsg_dump (msg);
        }
        // エラー処理を行わず単に assert しておきます
        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;     // 成功
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing client...\n");
    else
    if (self->verbose)
        zclock_log ("W: permanent error, abandoning request");

    return NULL;
}
```

変更点は、

- REQ ソケットの代わりに DEALER ソケットに置き換えましたのでエンベロープ意識する必要があります。
- API の中でリクエストの再試行を行いません。必要に応じてアプリケーション内で再試行を行って下さい。
- 同期的な送受信の API を廃止して、送信と受信の関数に別けました。
- send 関数は非同期ですので呼び出し後直ぐに戻ってきます。
- recv 関数はタイムアウト付きでブロックし、メッセージを受信すると呼び出しに戻ります。

そして 10 万メッセージを送信した後に 10 万メッセージを受信するテストプログラムをこの API を使って書き直すと以下のようになります。

mdclient2.c: Majordomo 非同期クライアントアプリケーション

```
// Majordomo プロトコル クライアント - 非同期版
// mdcli API を利用してプロトコルを隠蔽しています

// ライブラリをリンクするのではなく、単純に include しています
#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;              // 割り込み
    }
    printf ("%d replies received\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

プロトコル自体を変更した訳ではありませんので、ブローカーとワーカーのコードに変更はありません。それではパフォーマンスがどれだけ改善したか見てみましょう。以下は同期的クライアントでに一気に 10 万リクエストの送受信を行った際の実行結果です。

```
$ time mdclient
100000 requests/replies processed

real    0m14.088s
user    0m1.310s
sys     0m2.670s
```

そして、こちらが非同期クライアントで 1 つのワーカーにリクエストを行った際の実行結果です。

```
$ time mdclient2
100000 replies received

real    0m8.730s
user    0m0.920s
sys     0m1.550s
```

2 倍早くなりました、悪くありませんがワーカーを 10 に増やしてみましょう。

```
$ time mdclient2
100000 replies received

real    0m3.863s
user    0m0.730s
sys     0m0.470s
```

ワーカーは受け取ったメッセージを順番に処理しているのでこれは完全な非同期とは言えませんが、これはワーカーを増やすことで解決できます。私の PC は 4Core しか無いのでワーカーを 8 個以上に増やしてもそれ以上早くなりませんでした。しかし、まだブローカーについては最適化を行っていないにも関わらず、たった数分間の工夫で 4 倍ものパフォーマンスの向上が得られたのです。処理の大半はメッセージのコピーに費やされているので、ゼロコピーを行うことで更に早くなる余地があります。さして労力をかけずに 2.5 万回のリクエスト・応答を処理できればまあ十分でしょう。

ところで、非同期の Majordomo パターンに欠点が無いわけではありません。これにはブローカーのクラッシュから回復できないという根本的な欠点があります。mdcliapi2 のコードを読むと再接続を行っていない事が分かるでしょう。正しく再接続を行うには以下のことを行う必要があります。

- 全てのリクエストに番号を振り、応答と照合します。これにはプロトコルの変更が必要です。
- 送信する全てのリクエストをクライアント API でトラッキングし、受信していない応答を把握する必要があります。
- フェイルオーバーが発生した場合はまだ応答が返ってきていないリクエストをブローカーに再送します。

これらは無理なことではありませんが、確実に複雑性が増えます。これが本当に Majordomo に必要かどうかは用途に依存するでしょう。数千ものクライアントが接続する WEB フロントエンドでは必要かもしれませんが、DNS の様に 1 リクエストでセッションが完了する様なサービスでは必要ありません。

4.10 サービスディスカバリー

素晴らしいサービス指向ブローカーを作ることが出来ましたが、まだサービスが登録済みかどうかを知る方法がありません。動作していないサービスへのリクエストは失敗しますが、何故

失敗したのかが分かりません。そこで「echo サービスは動作していますか?」という様な問い合わせを行えると便利でしょう。最も解かりやすい方法は MDP のクライアント側のプロトコルを改修して新しいコマンドを追加することです。しかしこれでは MDP クライアントプロトコルの最大の魅力である単純さが失われてしまいます。

もうひとつの方法は E メールの様に無効なリクエストを返送することです。この方法は非同期の世界では上手く動作しますが、応答を受け取る際にどの様な応答かを適切に区別する必要があるため更に複雑性になってしまいます。

という訳で、既に私が用意した、MDP プロトコルを踏襲したサービスディスカバリーを使ってみましょう。サービスディスカバリーもそれ自体がサービスです。サービスを無効にしたり、サービスの利用統計提供するなどの管理サービスも必要になる可能性もあるでしょう。必要なのは、既存のプロトコルやアプリケーションに影響しない一般的で拡張性のあるソリューションです。

ここに MMI: Majordomo Management Interface という MDP プロトコルの上レイヤに構築した小さな仕様書があります。私達は既にこのプロトコルを実装しているのですが、コードをじっくり読んでいないのであれば恐らく見逃しているでしょう。これがどの様に動作するか説明すると。

- クライアントが mmi で始まるサービスに対してリクエストを行うと、ブローカーはワーカーにルーティングせずに内部的に処理します。
- このブローカーが行うサービスのひとつとして、mmi.service というサービス名でサービスディスカバリーを提供します。
- リクエストデータは実際に問い合わせを行うサービス名を指定します。
- ブローカーはそれが登録済みのサービスであれば「200」、存在しなければ「404」を返します。

以下はアプリケーションの中でサービスディスカバリーを使用する方法です。

mmiecho.c: Majordomo でのサービスディスカバリー

```
//  MMI による echo 問い合わせ

// ライブラリをリンクするのではなく、単純に include しています
#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);

    // 問い合わせを行うサービス名です
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");

    // リクエストを送信するサービスを指定します
    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;
}
```

ワーカーを起動していない状態でこのコードを実行すると「404」が返ってくるでしょう。この MMI 実装は手抜きですので、ワーカーが居なくなった場合も登録されたままになってしまいます。実際には、ワーカーが居なくなって一定のタイムアウトが経過するとサービスを削除する必要があります。

4.11 サービスの冪等性

冪等性と聞いてなんだか難しいものだと考える必要はありません。これは単に、何度繰り返し実行しても安全であるという意味です。時刻の確認は冪等性があります。クレジットカードを子供に貸す行為には冪等性はありません。クライアント・サーバーモデルの多くは冪等性がありますが、無いものもあります。例を挙げると、

- ステートレスなタスク分散処理には冪等性があります。例えばリクエストの内容のみに基づいてワーカーが計算を行うパイプラインモデルです。この様なケースでは非効率ではありますが、同じリクエストを何度実行しても安全です。
- 論理アドレスからエンドポイントのアドレスに変換する名前解決サービスには冪等性があります。同一の名前解決リクエストを何度行っても安全です。

冪等性の無いサービスの例を挙げると、

- ロギングサービス。同じログの内容が複数記録されていは困ります。
- 下流に影響を与えるサービス。受け取ったメッセージを他のノードに転送するサービスはが同一のメッセージを受け取ると、下流のノードも重複したメッセージを受け取るでしょう。
- 冪等性の無い方法で共有されているデータを更新するサービス。たとえば銀行口座の預金額を変更するサービスは工夫を行わなわない限り冪等性がありません。

アプリケーションに冪等性がない場合はクラッシュした際の対応はより慎重に考えなければなりません。アプリケーションがリクエストを受け付けていない段階で落ちた場合は特に問題ありませんが、データーベースのトランザクションの様な処理を行うアプリケーションにおいてリクエストの処理中に問題が発生した場合は問題となります。

クライアントに応答を返している最中にネットワーク障害が発生した場合に同じ問題が発生します。クライアントはサーバーが落ちたのだと思ってリクエストを再送し、サーバーは同じリクエストを 2 度処理します。これは望ましい動作ではありません。

非冪等性な操作を行う場合一般的には、重複したリクエストを処理しないような対策を行います。具体的には、

- クライアントは全てのリクエストにユニークなクライアント ID とユニークなメッセージ ID を付けて送信します。
- サーバーはリクエストに対して応答する前に、クライアント ID とメッセージ ID の組み合わせをキーにして保存します。
- 次にサーバーがリクエストを受け取った際に、まずクライアント ID とメッセージ ID の組み合わせを確認して同じリクエストを処理しないようにします。

4.12 非接続性の信頼性 (タイタニックパターン)

Majordomo が信頼性のあるメッセージブローカーとして機能する事が分かると、次にあなたはハードディスクへの永続化を行いたいと思うかもしれません。エンタープライズのメッセージングシステムにはそのような機能が用意されています。これは魅力的なアイディアですが、けして良いことばかりではありません。皮肉なことにこれは私の専門分野のひとつなのですが、必ずしも永続化が必須ではない幾つかの理由があります。

- これまで見てきたように、ものぐさ海賊パターンはとてもうまく機能します。異なるアーキテクチャに跨がった分散キュープロキシとして機能し、ワーカーはステートレスで冪等性があるとみなすことができましたが、永続化を行った場合はこうは行きません。

- 永続化はパフォーマンスを低下させ、管理しなければならない新たな部品を増やし、障害が発生すると業務に支障が出ないように朝の 6 時までに修理しなければなりません。海賊パターンの美しい所は単純な所です。このパターンはクラッシュが発生しません。もしハードウェア障害を心配しているのであれば P2P パターンに移行してブローカーを無くせば良いでしょう。これについては次の章で説明します。

とは言っても、永続化を行い非接続な非同期ネットワークの信頼性を高めるユースケースがひとつだけあります。これは海賊パターンの一般的な問題を解決します。クライアントとワーカーが散発的にしか接続しない場合 (E メールの様なシステムを思い浮かべて下さい) はステートレスなネットワークを利用できません。必ず中間に状態を持つ必要があります。

そこでタイタニックパターンです。メッセージをディスクに保存することで、散発的にクライアントやワーカーが接続して来た場合でもメッセージを失わない事を保証します。サービスディスカバリーと同様に、このタイタニックパターンを MDP プロトコルの上に追加実装します。これはブローカーに実装するのではなくワーカー側で信頼せい

- 単純な分割統治をおこなうので簡単です。
- ブローカーを実装する言語とは別の言語でワーカーを実装することが出来ます。
- fire-and-forget テクノロジーを独自に進化させます。

唯一の欠点は、ブローカーとハードディスクの間で幾つかのオーバーヘッドを必要とする事ですが、利点は欠点に勝るでしょう。

永続的なリクエスト・応答アーキテクチャを実現する方法はいくつもありますが、今回私達はその中で最も単純で痛みの無い方法を利用します。

タイタニックパターンは全てのワーカーに影響を与えるわけではありません。

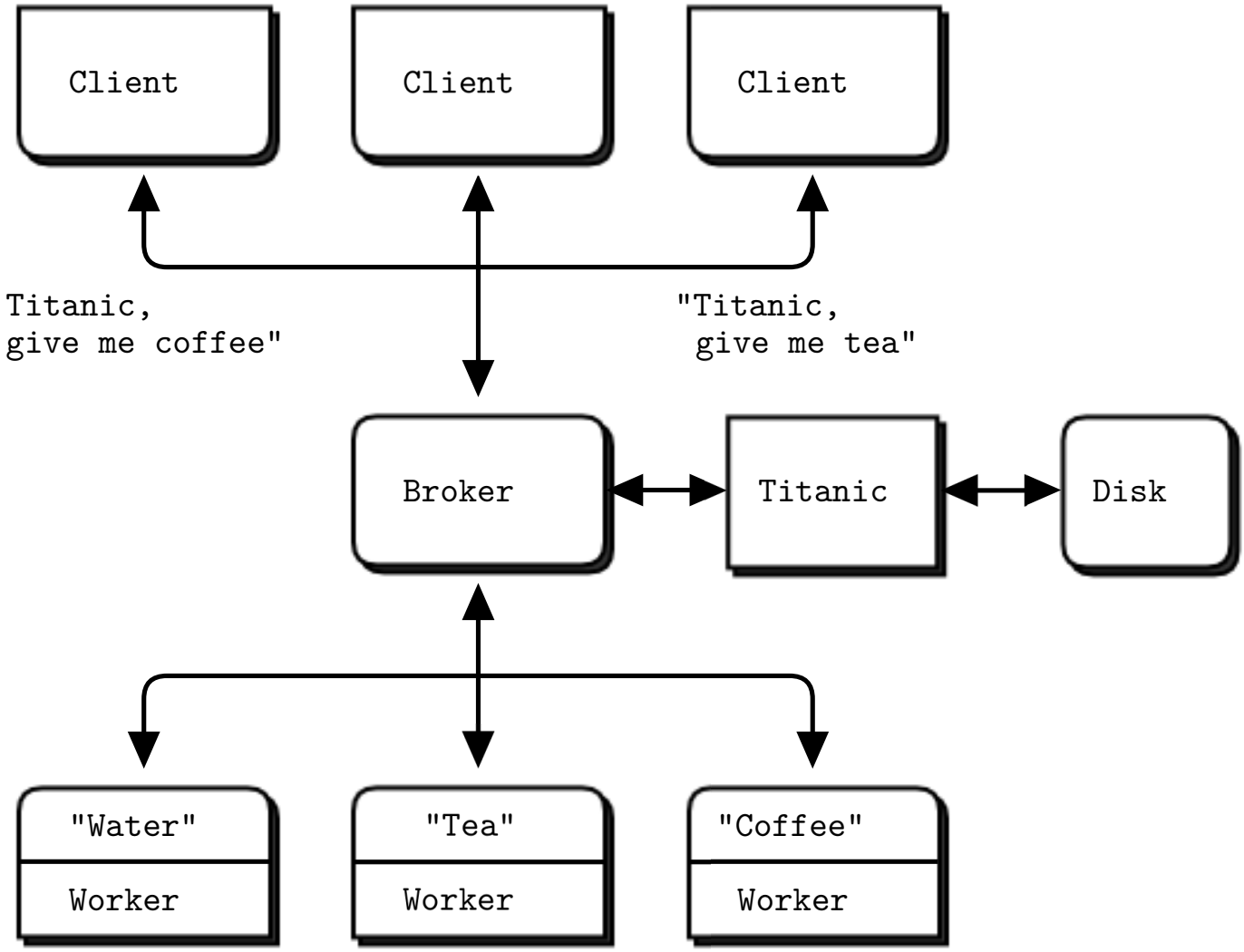


図 4.5　タイタニックパターン

この様に、タイタニックはワーカーとクライアントとの間に位置していて、クライアントとタイタニックのやりとりは以下通りです。

- クライアント「このリクエストを受け付けてくれる?」タイタニック「OK、完了」
- クライアント「それじゃあ応答を送ってくれる?」タイタニック「はい、これが応答ね」もしくは「そんなの無いよ」
- クライアント「さっき送信したリクエストを削除してもらえる?」タイタニック「OK、完了」

一方、タイタニックとブローカー間でのやりとりは以下の通りです。

- タイタニック「おーい、ブローカーさん。コーヒーサービスは動いてる?」ブローカー「うむ、動いているようだ」
- タイタニック「おーい、コーヒーサービスさん・このリクエストを処理してもらえる?」
- コーヒーサービス「はい、どうぞ」
- タイタニック「ありがとーーーーーー！」

それでは起こり得る障害のシナリオを見て行きましょう。リクエストを処理中のワーカーがクラッシュした場合、タイタニックは何度でも再試行します。応答が失われてしまった場合も再試行を行います。リクエストが処理されたにもかかわらず、クライアントが応答を受け取れなかった場合は再度問い合わせが行われます。リクエストの処理中にタイタニックがクラッシュした場合、クライアントは再試行を行います。リクエストが確実にディスクに書き込まれていれば、メッセージを失うことはありません。

やりとりはやや複雑ですが、パイプライン化が可能です。例えばクライアントが非同期な Majordomo パターンをしている場合、レスポンスを受け取るまでの間に多くの処理を行うことが可能です。

クライアントのリクエストに対して応答を返すには工夫が必要です。

異なる ID を持った多くのクライアントが同一のサービスに対してリクエストを行っているとします。安全で手頃な解決方法は以下の通りです。

- リクエストをキューに格納した後にタイタニックはクライアントに対して UUID を生成して返却します。
- クライアントはこの UUID を指定して応答を取得する必要があります。

実際のケースではクライアントはこの UUID をローカルのデータベースなどにに保存することになるでしょう。

正式な仕様を作成する楽しい作業に移る前に、まずクライアントとタイタニックがどの様な会話を行うか考えてみましょう。これには 2 種類の実装方法があります。ひとつ目は 1 つサービスで複数のリクエスト種別を扱う方法です。もう一方は、以下のような 3 つのサービスを用意する方法です。

- titanic.request: リクエストメッセージを保存し、UUID を返却するサービス。
- titanic.reply: UUID に対応する応答を返却するサービス。
- titanic.close: 応答が格納されたかどうかを確認するサービス。

ここでは、単純にマルチスレッドワーカーを作成して 3 つのサービスを提供します。まずはタイタニックがやりとりするメッセージフレームを設計してみましょう。これをタイタニック・サービス・プロトコル (以下 TSP) と呼びます。

MDP の場合と比較して、TSP は明らかにクライアントの作業量が多くなります。以下に短くて堅牢な「echo クライアント」のサンプルコードを示します。

ticlient.c: タイタニッククライアント

```
// タイタニック クライアント
// http://rfc.zeromq.org/spec:9 のクライアント側の実装です

// ライブラリを作成せずビルドするために
#include "mdcliapi.c"

// TSP サービスを呼び出します
// 成功 (ステータスコードが 200) の場合は応答を返し、失敗したら NULL を返し
// ます。

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);    // 割り込み、もしくは失敗

    zmsg_destroy (&reply);
    return NULL;        // 予期しない失敗
}

// メインタスクでは、エコーリクエストを送信してテストを行います
```

```
int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    // 1. タイタニックに対してエコーリクエストを送信します
    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. 応答が得られるまで待ちます
    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. リクエストを終わります
            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);      // 5 秒後に再試行を行います
        }
    }
    zframe_destroy (&uuid);
```

```
    mdcli_destroy (&session);
    return 0;
}
```

もちろん、この様なコードはフレームワークや API の中に隠蔽化されるでしょう。ただ、隠蔽化してしまうとメッセージングを学ぼうとしているアプリケーション開発者の障害になります。頭を悩ませ、時間を掛けて多くのバグを解決してこそ、知性が磨かれるのです。

クライアントがリクエストを行っている間はブロックするアプリケーションが多いですが、タスクを実行している間に別のタスクを行いたい事もあるでしょう。このような要件を実現するには工夫が必要です。Majordomo の時に行ったように、うまく API で隠蔽化してやれば一般的なアプリケーション開発者が誤用する事は少なくなります。

今度はタイタニックブローカーの実装です。先ほど提案した通り、このブローカーは 3 つのスレッドで 3 つのサービスを提供します。そしてメッセージ 1 つにつき 1 ファイルという最も単純かつ最も荒っぽい構成でディスクへの永続化を行います。唯一複雑な部分は、ディレクトリを何度も走査をするのを避けるために、全てのリクエストを別のキューで保持している所です。

titanic.c: タイタニックブローカー

```
// タイタニック クライアント
// http://rfc.zeromq.org/spec:9 のクライアント側の実装です

// ライブラリを作成せずビルドするために
#include "mdcliapi.c"

// TSP サービスを呼び出します
// 成功 (ステータスコードが 200) の場合は応答を返し、失敗したら NULL を返し
// ます。

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);    // 割り込み、もしくは失敗

    zmsg_destroy (&reply);
    return NULL;        // 予期しない失敗
}

// メインタスクでは、エコーリクエストを送信してテストを行います
int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    // 1. タイタニックに対してエコーリクエストを送信します
    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. 応答が得られるまで待ちます
    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. リクエストを終わります
            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);      // 5 秒後に再試行を行います
        }
    }
    zframe_destroy (&uuid);
    mdcli_destroy (&session);
    return 0;
}
```

これをテストするには、mdbroker と titanic を開始して ticlient を実行します。そして、mdworker が動作した段階でクライアントは応答を受け取るのを確認できるでしょう。

このコードの注意点は、

- ループの開始時にメッセージを送信したり受信したりしていますが、これはタイタニックが役割の異なる、クライアントとワーカーの双方と通信するためです。
- このタイタニックブローカーは、動作しているサービスだけにメッセージを送信する為に、MMI サービスディスカバリープロトコルを利用します。しかしこのサンプルコードで実装した Majordomo ブローカーは少しお馬鹿さんなので、上手く動かない場合もあります。
- 新規リクエストを titanic.request サービスからメインディスパッチャーへ送信する手段にプロセス内通信を利用しています。これによりディスパッチャーが毎回ディレクトリを走査したり、時刻でソートしたりする処理を省くことが出来ます。

タイタニックで重要な事は、パフォーマンスは低いけれども、信頼性の保証された実装である事です。mdbroker と titanic を起動し、続いて、ticlient と echo サービスの mdworker を起動してみて下さい。これら全てのプログラムは-v オプションを指定して詳細な動きトレースすることが出来ます。メッセージを失うこと無く、全ての部品を再起動することを確認できるはずです。

実際のケースでこのタイタニックパターンを利用する場合、「どうすれば早くなるか?」を考

える必要があるでしょう。

以下は私が行った高速化の為の工夫です。

- 複数のファイルではなく、単一のファイルを利用します。一般的に OS は複数の小さなファイル扱うより、大きなファイルひとつを処理したほうが効率が良いです。
- 新規リクエストを連続した領域に書き込めるように調整してやります。
- 起動時にメモリ上にインデックスを構築する事で余計なディスクアクセスを回避できます。障害時にメッセージを失わないようにするには、受信の後に fsync すると良いでしょう。
- 錆びた鉄のプラッターの代わりに SSD を利用する。
- 必要に応じて増減可能な巨大なファイルをあらかじめ作成しておきます。これによりデータの連続性を保証し、フラグメンテーションを回避できます。

それと、高速なキー・バリュー・ストアではないデーターベースにメッセージを格納することはあまり推奨できません。特定のデーターベースが大好きだというなら話は別ですが、ディスクファイルを抽象化するために法外なコストを支払うことになります。

タイタニックの信頼性を更に高めたいのであれば、核攻撃の届かいない 2 台目のサーバーに重複したリクエストを送信すると良いでしょう。レイテンシが大きいでしょうけどね。

タイタニックの信頼性を下げて、リクエストと応答をメモリに格納する事で非接続性のネットワーク機能を実現できますが、タイタニックサーバー自体の障害でメッセージは失われてしまいます。

4.13 高可用性ペア (バイナリー・スターパターン)

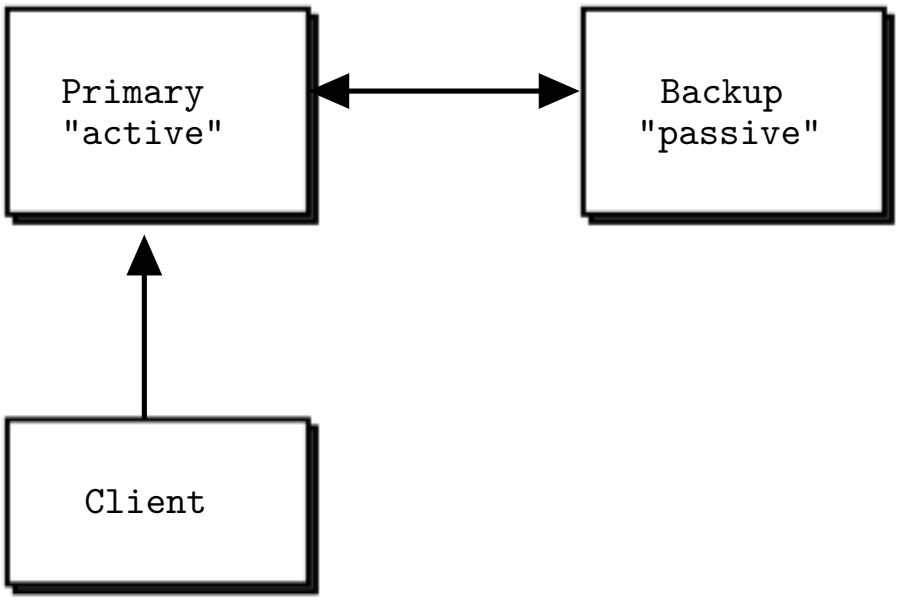


図 4.6 高可用性ペアの通常状態

バイナリー・スターパターンはプライマリー・バックアップに対応する 2 つのサーバーを用いて信頼性を高めます。ある時点では、ひとつのサーバー (アクティブ) がクライアントからの接続を受け付け、もう片方 (非アクティブ) はなにもしません。しかし、この 2 つのサーバーはお互いに監視しています。ネットワーク上からアクティブサーバーが居なくなると、すぐに非アクティブサーバーがアクティブサーバーの役割を引き継ぎます。

私達は OpenAMQ サーバーの頃にバイナリー・スターパターンを開発しました。以下の様に設計されています。

- 単純な高可用性ソリューションを提供する。
- 簡単に理解できて使いやすいこと。
- 必要な場合のみフェイルオーバーします。

バイナリー・スターパターンでは、以下のパターンでフェイルオーバーが発生します。

- プライマリーサーバーに致命的な問題 (爆発、火災、電源を引っこ抜いた、など) が発生した場合。アプリケーションはそれを確認し、バックアップサーバーに再接続を行います。

- プライマリーサーバーの居るネットワークセグメントで障害が発生した場合。恐らくルーターが高負荷になるでしょうが、アプリケーションはバックアップサーバーに再接続を行うでしょう。
- プライマリーサーバーがクラッシュした場合、もしくは再起動を行って自動的に起動しなかった場合。

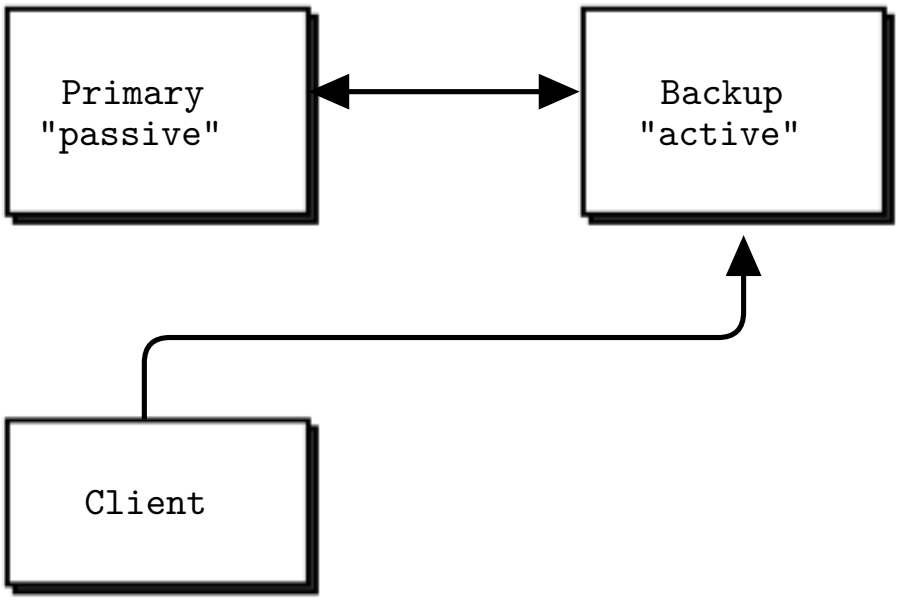


図 4.7　High-availability Pair During Failover

フェイルオーバーから復旧するには以下の事を行います。

- プライマリーサーバーを起動し、ネットワークから見えるようにしてやります。
- 一時的にバックアップサーバーを停止します。これによりアプリケーションからの接続が切断されます。
- アプリケーションがプライマリーサーバーに再接続するのを確認し、バックアップサーバーを起動します。

フェイルオーバーからの復旧は手動で行います。復旧を自動的に行うことが良くない事を私達は経験済みです。これには以下の理由があります。

- フェイルオーバーは恐らく 10〜30 秒のサービス停止を発生させます。そして復旧にも同等の時間が掛かります。これはユーザーの少ない時間帯に行ったほうが良いでしょう。

- 緊急事態に陥った場合、最も重要なのは確実に修復させる事です。自動復旧を行ったとしても、システム管理者のダブルチェック無しでは復旧した事を確証出来ません。
- 例えば一時的なネットワーク障害でフェイルオーバーした場合に自動復旧を行った場合、サービス停止の原因を特定することが難しくなります。

とは言っても、バイナリー・スターパターンではプライマリーサーバーに障害が発生し、その後バックアップサーバーで障害が発生すると、事実上自動復旧した様になります。

バイナリー・スターパターンをシャットダウンさせるには以下の方法があります。

- まず非アクティブサーバーを停止し、その後アクティブサーバーを停止する。
- 2 つのサーバーをほぼ同時に停止する。

アクティブサーバーをを停止し、時間を空けて非アクティブサーバーを停止した場合、アプリケーションは切断、再接続、切断という動作となりユーザーを混乱させてしまいます。

4.13.1 詳細な要件

バイナリー・スターパターンは出来るだけ単純に動作します。実は私達はこの設計を 3 度再設計したという経緯があります。以前の設計はとても複雑だと気がついたので簡単に理解して利用できるように機能を削ってきました。

高可用性アーキテクチャでは以下の要件を満たす必要があるでしょう。

- フェイルオーバーはハードウェア障害、火災などの重大なシステム障害に対する保険です。一般的な障害から復旧するための方法は既に学んだ様に単純な方法があります。
- フェイルオーバーに要する時間は 60 秒以下にすべきであり、できれば 10 秒以下が望ましいでしょう。
- フェイルオーバーは自動で行いますが、フェイルオーバーからの復旧は手動で行う必要があります。バックアップサーバーへの切り替えは自動的に行われて問題ありませんが、プライマリーサーバーへの切り替えは問題が修正されているかどうかをオペレーターが確認し、適切なタイミングを見極める必要があるからです。
- プロトコルは開発者が理解しやすいように単純かつ簡単にすべきであり、理想的にはクライアント API で隠蔽するのが良いでしょう。
- ネットワークが分断された際に発生するスプリットブレイン問題を回避するための明確なネットワーク設計手順が必要です。
- サーバーを起動する順序に依存せず動作しなければなりません。
- クライアントが停止すること無く（再接続は発生するでしょうが)、どちらのサーバーも停止したり再起動を行ったりできるようにしなければなりません。

- オペレーターは常に 2 つのサーバーを監視する必要があります。
- 2 つのサーバーは高速なネットワーク回線で接続され、フェイルオーバーは特定の IP 経路で同期する必要があります。

以下の事を仮定します。

- ひとつのバックアップサーバーで十分な保険であるとし、複数のバックアップサーバーを必要としません。
- プライマリーサーバーとバックアップサーバーは各 1 台でアプリケーションの負荷に耐えられるとします。これらのサーバーで負荷分散しないでください。
- 常時何も行わないバックアップサーバーを動作させるための予算を確保して下さい。

以下の事についてはここでは触れません。

- アクティブバックアップサーバー、あるいは負荷分散を行うこと。バイナリー・スターパターンではバックアップサーバーは非アクティブであり、プライマリーサーバーが非アクティブにならない限り利用できません。
- 信頼性の低いネットワークを利用していることを前提とした場合、何らかの方法でメッセージの永続化、もしくはトランザクションを行う必要があります。
- サーバーの自動検出。バイナリー・スターパターンではネットワークの設定を主導で行い、アプリケーションはこの設定を知っているとします。
- メッセージや状態のサーバー間でのレプリケーション。フェイルオーバーが発生するとセッションを 1 からやり直すこととします。

バイナリー・スターパターンで利用する用語は以下の通りです。

- プライマリー: 初期または通常状態でアクティブなサーバー。
- バックアップ: 通常状態で非アクティブなサーバー。プライマリーサーバーがネットワーク上から居なくなった際にアクティブになり、クライアントはこちらに接続します。
- アクティブ: クライアントからの接続を受け付けているサーバー。唯一ひとつのサーバーだけがアクティブになれます。
- 非アクティブ: アクティブが居なくなった際に役割を引き継ぐサーバー。バイナリー・スターパターンでは通常プライマリーサーバーがアクティブであり、バックアップサーバーが非アクティブです。フェイルオーバーが起こった際はこれが逆転します。

バイナリー・スターパターンでは以下の情報が設定されている必要があります。

1. プライマリーサーバーはバックアップサーバーのアドレスを知っていること
2. バックアップサーバーはプライマリーサーバーのアドレスを知っていること

3. フェイルオーバーの応答時間は 2 つのサーバーで同じである必要があります。

チューニングパラメーターとしては、フェイルオーバーを行うためにサーバー同士がお互いの状態を確認する間隔を設定します。今回の例ではフェイルオーバーのタイムアウトは既定で 2 秒とします。この数値を小さくすることでより迅速にバックアップサーバーがアクティブサーバーの役割を引き継ぐことが出来ます。しかし、予期しないフェイルオーバーが発生する可能性があります。例えば、プライマリーサーバーがクラッシュした場合に自動的に再起動を行うスクリプトを書いた場合、タイムアウトはプライマリーサーバーの再起動に要する時間より長く設定すべきでしょう。

バイナリー・スターパターンで正しくクライアントアプリケーションが正しく動作するには、クライアントを以下の様に実装する必要があります。

1. 2 つのサーバーのアドレスを知っている必要があります。
2. まず、プライマリーサーバーに接続し、失敗したらバックアップサーバーに接続します。
3. コネクションの切断を検出するために、ハートビートを行います。
4. 再接続を行う際、まずプライマリーサーバーに接続し、次にバックアップサーバーに接続します。リトライ間隔はフェイルオーバータイムアウトと同等の間隔で行います。
5. 再接続を行う歳、セッションを再生成します。
6. 信頼性を高めたい場合はフェイルオーバーが行われた際に失われたメッセージを再送信します。

これらを実装するのはそれほど簡単な仕事ではありませんので、通常は API の中に隠蔽すると良いでしょう。

バイナリー・スターパターンの主な制限は以下の通りです。

- 1 プロセスではバイナリー・スターパターンを構成できません。
- プライマリー・サーバーは 1 つのバックアップサーバーを持ち、これ以上は増やせません。
- 非アクティブサーバーは通常動作しません。
- プライマリーサーバーとバックアプサーバーは各々がアプリケーションの負荷に耐えられなければなりません。
- フェイルオーバーの設定は実行中に変更出来ません。
- クライアントアプリケーションはフェイルオーバーに対応する為のの機能を持っている必要があります。

4.13.2 スプリット・ブレイン問題の防止

クラスターが分断し、個々の部品が同じタイミングでアクティブになろうとするとスプリット・ブレイン問題が発生します。これはアプリケーションの停止を引き起こします。バイナリー・スターパターンはスプリット・ブレイン問題を検出し解決するアルゴリズムを持っています。サーバー同士がお互いに通信して判断するのではなく、クライアントからの接続を受けてから自分がアクティブであると判断します。

しかし、このアルゴリズムを騙す為の意図的なネットワークを構築することは可能です。この典型的なシナリオは、バイナリー・スターの対が 2 つの建物に分散されており、各々の建物にクライアントアプリケーションが存在するネットワークです。この時、建物間のネットワークが切断されるとバイナリー・スターの対は両方共アクティブになるでしょう。

このスプリット・ブレイン問題を防ぐには、単純にバイナリー・スターの対を同じネットワークスイッチに接続するか、クロスケーブルでお互いに直接接続すると良いでしょう。

バイナリー・スターパターンではアプリケーションの存在するネットワークを 2 つの島に分けてはいけません。この様なネットワーク構成である場合は、フェイルオーバーではなくフェデレーションパターンを利用すべきでしょう。

神経質なネットワークでは、単一では無く 2 つのクラスターを相互接続することがあります。さらに、場合によっては相互接続の為の通信とメッセージ処理の通信で異なるネットワークカードが利用される場合があるでしょう。まずはネットワーク障害とクラスター内の障害とを切り分ける事が重要です。ネットワークポートはかなりの頻度で壊れることがあるからです。

4.13.3 バイナリー・スターの実装

前置きはこれくらいにしおいて、実際に動作するバイナリー・スターサーバーの実装を見て行きましょう。プライマリーとバックアップの役割は実行時に指定しますので、コード自体は同じものです。

bstarsrv.c: バイナリー・スター サーバー

```
// バイナリ・スターの概念実装。このサーバーはバイナリー・スターのフェイ
// ルオーバーを実演しているのみで、実用的ではありません。

#include "czmq.h"

// とり得る状態の一覧です
```

```
typedef enum {
    STATE_PRIMARY = 1,          // プライマリ - 接続を待っている状態
    STATE_BACKUP = 2,           // バックアップ - 接続を待っている状態
    STATE_ACTIVE = 3,           // アクティブ - 接続を受け入れた状態
    STATE_PASSIVE = 4           // パッシブ - 接続を受け入れなかった状態
} state_t;

// 発生し得るイベントの一覧です
typedef enum {
    PEER_PRIMARY = 1,           // 相方がプライマリに遷移
    PEER_BACKUP = 2,            // 相方がバックアップに遷移
    PEER_ACTIVE = 3,            // 相方がアクティブになった
    PEER_PASSIVE = 4,           // 相方がパッシブになった
    CLIENT_REQUEST = 5          // クライアントからのリクエスト
} event_t;

// 有限状態オートマトン
typedef struct {
    state_t state;              // 現在の状態
    event_t event;              // 現在のイベント
    int64_t peer_expiry;        // 相方が「落ちている」かどうか
} bstar_t;

// 定期的に状態情報を送信します。
// もし 2 回のハートビートに応答が無かった場合、相手は落ちたと見なします。
#define HEARTBEAT 1000          // ミリ秒

// バイナリー・スターの根幹は有限状態オートマトン (FSM) です。
// FSM でイベントが発生すると、現在の状態に適用し新しい状態に遷移します。

static bool
s_state_machine (bstar_t *fsm)
{
    bool exception = false;

    // プライマリとバックアップ状態ではアクティブ、パッシブイベントを待
    // ちます。
    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;
        }
        // クライアントの接続を受け入れる
    }
    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
        // バックアップ状態なのでクライアントの接続を拒否します
        if (fsm->event == CLIENT_REQUEST)
            exception = true;
    }
    else
    // アクティブ、パッシブ状態の場合

    if (fsm->state == STATE_ACTIVE) {
        if (fsm->event == PEER_ACTIVE) {
            // 両方共アクティブ状態、すなわちスプリットブレインが発生しました
            printf ("E: fatal error - dual actives, aborting\n");
            exception = true;
        }
    }
    else
    // こちらがパッシブ状態で相手が落ちた場合、クライアントリクエストイ
    // ベントに起因してフェイルオーバーが発生します。
    if (fsm->state == STATE_PASSIVE) {
        if (fsm->event == PEER_PRIMARY) {
            // 相方が再起動しました。こちらがアクティブになり相手がパッ
            // シブになります。
            printf ("I: primary (passive) is restarting, ready active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_BACKUP) {
            // 相方が再起動しました。こちらがアクティブになり相手がパッ
            // シブになります。
            printf ("I: backup (passive) is restarting, ready active\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_PASSIVE) {
            // 両方共パッシブ、すなわちクラスターが応答不能になります
```

```
            printf ("E: fatal error - dual passives, aborting\n");
            exception = true;
        }
        else
        if (fsm->event == CLIENT_REQUEST) {
            // 相手がアクティブの状態でタイムアウトが発生すると、クライアン
            // トリクエストに起因してフェイルオーバーが発生します。

            assert (fsm->peer_expiry > 0);
            if (zclock_time () >= fsm->peer_expiry) {
                // 相手が落ちたのでこちらがアクティブ状態になります
                printf ("I: failover successful, ready active\n");
                fsm->state = STATE_ACTIVE;
            }
            else
                // 相手が生きていれば、クライアントの接続を拒否します。
                exception = true;
        }
    }
    return exception;
}

// こちらがメインタスクです。まず、お互いに bind と接続を行い正しい状態メッ
// セージが表示されることを確認してください。
// ここでは 3 つのソケットを利用します。ひとつは状態のパブリッシュ、もう
// ひとつは状態のサブスクライブ、そしてもうひとつでクライアントと通信し
// ます。
int main (int argc, char *argv [])
{
    // 引き数は以下のどちらかになります
    //      プライマリーサーバーの場合 -p tcp://localhost:5001
    //      バックアップサーバーの場合 -p 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);
    }
    // ここでは 2 つの入力ソケットからのメッセージを受信し、有限状態オー
    // トマトンとしてイベントを処理します。
    // クライアントからのリクエストに対しては単純にエコーを返します。

    // 次の状態メッセージまでのタイマーを設定
    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;              // コンテキストが終了した

        if (items [0].revents & ZMQ_POLLIN) {
            // クライアントリクエストを受信
            zmsg_t *msg = zmsg_recv (frontend);
            fsm.event = CLIENT_REQUEST;
            if (s_state_machine (&fsm) == false)
                // エコーを返します
                zmsg_send (&msg, frontend);
            else
                zmsg_destroy (&msg);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            // 相方から状態情報を受信、イベントを実行します
            char *message = zstr_recv (statesub);
```

```
            fsm.event = atoi (message);
            free (message);
            if (s_state_machine (&fsm))
                break;          // エラー
            fsm.peer_expiry = zclock_time () + 2 * HEARTBEAT;
        }
        // タイムアウトしたら相方に状態を送信
        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");

    // ソケットとコンテキストを開放
    zctx_destroy (&ctx);
    return 0;
}
```

そしてこちらがクライアントのコードです。

bstarcli.c: バイナリー・スター クライアント

```
// バイナリ・スターの概念実装。このクライアントはバイナリー・スターのフェ
// イルオーバーを実演しているのみで、実用的ではありません。


#include "czmq.h"
#define REQUEST_TIMEOUT     1000    // ミリ秒
#define SETTLE_DELAY        2000    // フェイルオーバーまでの時間

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) {
        // リクエストを送信し、応答を受け取る為の処理を行う
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            // 応答を受信するためにソケットを監視します
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          // 割り込み

              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:

            // ものぐさ海賊クライアントと同じ戦略を利用します。
            // 応答の取得までにタイムアウトが発生した場合、ソケットを閉
            // じて再試行を行います。
            // バイナリー・スターではクライアントの投票によりプライマリー
            // サーバーを決定します。従ってクライアントはサーバーへ順番
            // に接続しなければなりません。

            if (items [0].revents & ZMQ_POLLIN) {
                // サーバーから応答を受信したら、シーケンス番号を確認します。
                char *reply = zstr_recv (client);
                if (atoi (reply) == sequence) {
                    printf ("I: server replied OK (%s)\n", reply);
                    expect_reply = 0;
                    sleep (1);  // 1 秒間に 1 リクエストに制限
                }
                else
                    printf ("E: bad reply from server: %s\n", reply);
                free (reply);
            }
            else {
                printf ("W: no response from server, failing over\n");

                // 応答が無いので、新しいソケットで再接続します
                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]);

                // 新しいソケットでリクエストを再送信します
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

バイナリー・スターのテストを行うには、以下のように 2 つサーバーとクライアントを起動します。起動する順序はどちらが先でも構いません。

```
bstarsrv -p     # Start primary
bstarsrv -b     # Start backup
bstarcli
```

この状態でプライマリーサーバーを停止することにより、フェイルオーバーを発生させることができます。そしてプライマリーを起動し、バックアップを停止する事で復旧が完了します。フェイルオーバーや復旧のタイミングはクライアントが判断する事に注意して下さい。

バイナリー・スターは有限状態オートマトンにより動作します。「Peer Active」は相手側のサーバーがアクティブ状態であるという意味のイベントです。「Client Request」はクライアントからのリクエストを受け取ったことを意味するイベントです。「Client Vote」はパッシブ状態のサーバーがクライアントからのリクエストを受け取り、アクティブ状態に遷移します。

サーバー同士で状態を通知するために PUB-SUB ソケットを利用しています。他のソケットの組み合わせでは上手く動作しないでしょう。PUSH と DEALER ソケットの組み合わせでは通信相手がメッセージを受信する準備が出来ていない場合にブロックしてしまいます。PAIR ソケットでは通信相手と一時的に通信できなくなった場合に再接続を行いません。ROUTER ソケットではメッセージを送信する際に通信相手のアドレスが必要です。

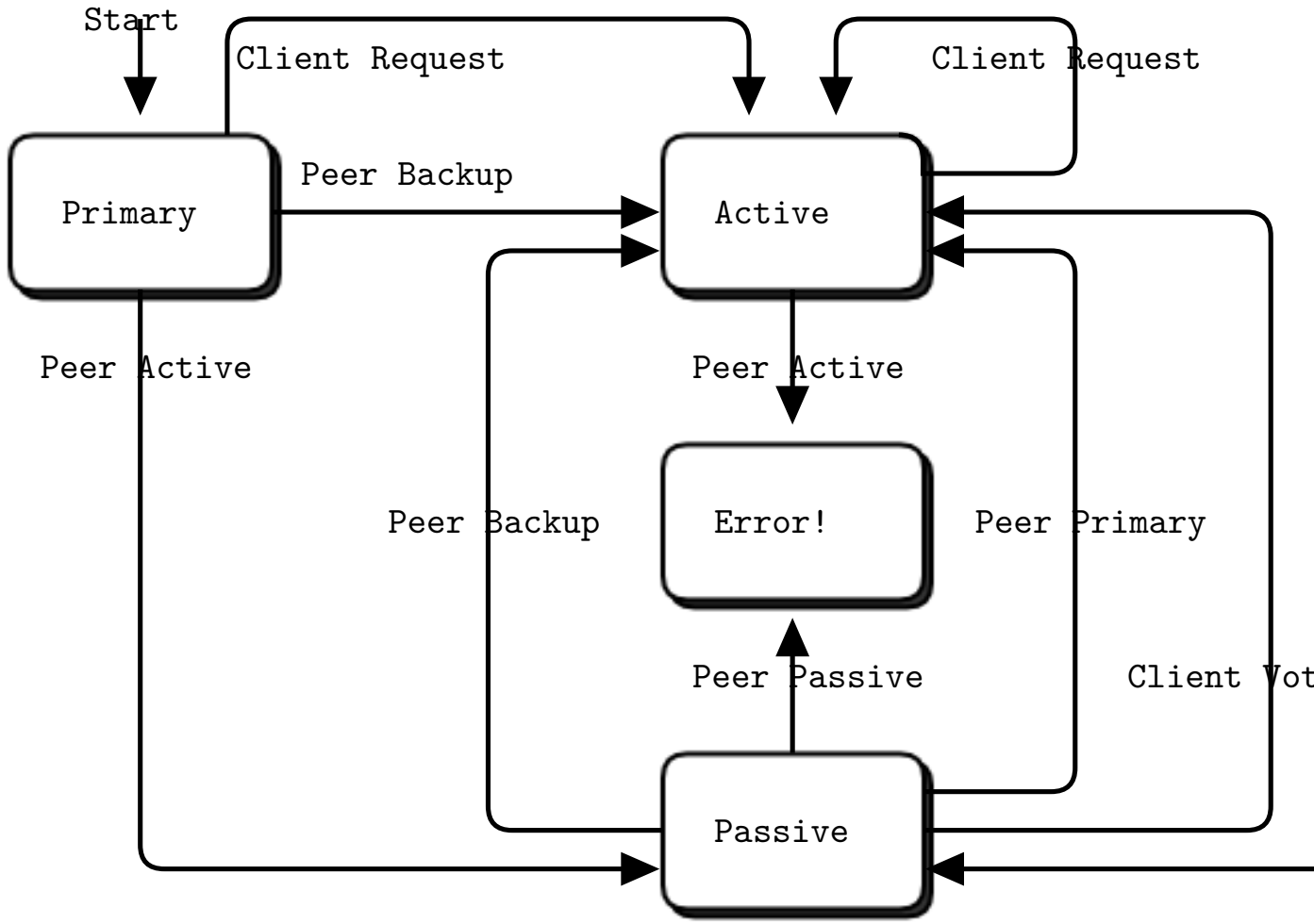


図 4.8　バイナリー・スター有限状態オートマトン

4.13.4 バイナリー・スター・リアクター

バイナリー・スターを再利用可能なリアクタークラスとしてパッケージングすると汎用的で便利です。リアクターにはメッセージを処理する関数を渡して実行します。既存のサーバーに対してバイナリー・スターの機能をコピペするよりはこちらの方が良いでしょう。

C 言語の場合、既に紹介した CZMQ の zloop クラスを利用します。zloop にはソケットやタイマーイベントに反応するハンドラーを登録する事ができます。バイナリー・スターの場合、アクティブから非アクティブへの遷移などの状態の変更に関するハンドラを登録します。こちらが bstar クラスの実装です。

bstar.c: バイナリー・スター リアクター

```
// bstar クラス - バイナリー・スター リアクター

#include "bstar.h"

// とり得る状態の一覧です
typedef enum {
    STATE_PRIMARY = 1,          // プライマリ - 接続を待っている状態
    STATE_BACKUP = 2,           // バックアップ - 接続を待っている状態
    STATE_ACTIVE = 3,           // アクティブ - 接続を受け入れた状態
    STATE_PASSIVE = 4           // パッシブ - 接続を受け入れなかった状態
} state_t;

// 発生し得るイベントの一覧です
typedef enum {
    PEER_PRIMARY = 1,           // 相方がプライマリに遷移
    PEER_BACKUP = 2,            // 相方がバックアップに遷移
    PEER_ACTIVE = 3,            // 相方がアクティブになった
    PEER_PASSIVE = 4,           // 相方がパッシブになった
    CLIENT_REQUEST = 5          // クライアントからのリクエスト
} event_t;

// クラスの構造

struct _bstar_t {
    zctx_t *ctx;                // コンテキスト
    zloop_t *loop;              // リアクターループ
    void *statepub;             // PUB ソケット
    void *statesub;             // SUB ソケット
    state_t state;              // 現在の状態
    event_t event;              // 現在のイベント
    int64_t peer_expiry;        // 相方が「落ちている」かどうか
    zloop_fn *voter_fn;         // 投票ハンドラ
    void *voter_arg;            // 投票ハンドラの引き数
    zloop_fn *active_fn;        // アクティブになると呼ばれるハンドラ
    void *active_arg;           // アクティブハンドラの引き数
    zloop_fn *passive_fn;       // パッシブになると呼ばれるハンドラ
    void *passive_arg;          // パッシブハンドラの引き数
};

// 有限状態オートマトンは前回のサーバーと同じです。
// このリアクターを詳しく理解したい場合は CZMQ zloop クラスを読んで下さい。

// 定期的に状態情報を送信します。
// もし 2 回のハートビートに応答が無かった場合、相手は落ちたと見なします。
#define BSTAR_HEARTBEAT     1000        //  In msecs
```

```
// 有限状態オートマトンがイベントを正常に処理した場合は 0 を返し、例外が
// 発生した場合は-1 を返します。

static int
s_execute_fsm (bstar_t *self)
{
    int rc = 0;
    // プライマリー状態であれば、接続を待ち、CLIENT_REQUEST イベントを受
    // け入れます。
    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) {
            // バックアップが長時間アクティブとして動作していない時は、
            // こちらがアクティブになることが出来ます。
            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
                // バックアップがアクティブ状態なので、プライマリはクラ
                // イアントに応答しません
                rc = -1;
        }
    }
    else
    // 相方に接続が来るのを待っています。
    // この状態で CLIENT_REQUEST が来た場合は拒否します
    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
    // アクティブ状態ですので CLIENT_REQUEST イベントを受け入れます
    if (self->state == STATE_ACTIVE) {
        if (self->event == PEER_ACTIVE) {
            // 両方共アクティブ状態、すなわちスプリットブレインが発生しました
            zclock_log ("E: fatal error - dual actives, aborting");
            rc = -1;
        }
    }
    else
    // パッシブ状態です
    // 相手が落ちていて、CLIENT_REQUEST が来た場合はフェイルオーバーします
    if (self->state == STATE_PASSIVE) {
        if (self->event == PEER_PRIMARY) {
            // 相方は再起動しています。
            // こちらがアクティブになり、相方はパッシブになります。
            zclock_log ("I: primary (passive) is restarting, ready as active");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_BACKUP) {
            // 相方は再起動しています。
            // こちらがアクティブになり、相方はパッシブになります。
            zclock_log ("I: backup (passive) is restarting, ready as active");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_PASSIVE) {
            // 両方共パッシブ、すなわちクラスターが応答不能になります
            zclock_log ("E: fatal error - dual passives, aborting");
            rc = -1;
        }
        else
        if (self->event == CLIENT_REQUEST) {
            // 相手がアクティブの状態でタイムアウトが発生すると、クライアン
            // トリクエストに起因してフェイルオーバーが発生します。
```

```
            assert (self->peer_expiry > 0);
            if (zclock_time () >= self->peer_expiry) {
                // 相手が落ちたのでこちらがアクティブ状態になります
                zclock_log ("I: failover successful, ready as active");
                self->state = STATE_ACTIVE;
            }
            else
                // 相手が生きていれば、クライアントの接続を拒否します。
                rc = -1;
        }
        // 必要に応じて状態変更のハンドラを呼び出します
        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;
}

// リアクターイベントハンドラ

// 状態を相方に伝えます。
int s_send_state (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    zstr_send (self->statepub, "%d", self->state);
    return 0;
}

// 相方の状態を受信し、有限状態オートマトンを実行します
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);
}
```

```
// アプリケーションは問い合わせ可能かどうか確認したいはずです
int s_voter_ready (zloop_t *loop, zmq_pollitem_t *poller, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    // サーバーが受付可能であればアプリケーションハンドラを呼び出します
    self->event = CLIENT_REQUEST;
    if (s_execute_fsm (self) == 0)
        (self->voter_fn) (self->loop, poller, self->voter_arg);
    else {
        // メッセージを破棄します、このメッセージは誰にも読まれません
        zmsg_t *msg = zmsg_recv (poller->socket);
        zmsg_destroy (&msg);
    }
    return 0;
}

// これは bstar クラスのコンストラクタです。ここで引き数を与えてプライマ
// リかバックアップかを伝えてやる必要があります。
// 同時にエンドポイントと相方への接続先を指定します。

bstar_t *
bstar_new (int primary, char *local, char *remote)
{
    bstar_t
        *self;

    self = (bstar_t *) zmalloc (sizeof (bstar_t));

    // バイナリー・スターの初期化
    self->ctx = zctx_new ();
    self->loop = zloop_new ();
    self->state = primary? STATE_PRIMARY: STATE_BACKUP;

    // 相方へ状態を通知する PUB ソケットの作成
    self->statepub = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->statepub, local);

    // 相方の状態を受け取る SUB ソケットの作成
    self->statesub = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_set_subscribe (self->statesub, "");
    zsocket_connect (self->statesub, remote);

    // リアクターイベントの準備
    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;
}

// bstar リアクターを終了するデストラクタ

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;
    }
}

// このメソッドは zloop リアクターを返します。
// ここでタイマーやリーダーを追加することが出来ます。

zloop_t *
bstar_zloop (bstar_t *self)
{
    return self->loop;
}

// このメソッドはクライアントの投票ソケットを登録します。
// メッセージを受信すると、バイナリー・スター FSM で CLIENT_REQUEST イベン
// トを発生させ、アプリケーションハンドラを呼び出します。

int
bstar_voter (bstar_t *self, char *endpoint, int type, zloop_fn handler,
             void *arg)
{
    // あとで利用するためにハンドラーと引き数を保持しておきます
    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);
```

```
}


// 状態を変更すると呼び出されるハンドラを登録します。

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;
}

// デバッグ用の詳細ログフラグです

void bstar_set_verbose (bstar_t *self, bool verbose)
{
    zloop_set_verbose (self->loop, verbose);
}

// reactor を開始します。
// いずれかのハンドラが-1 を返した場合や、プロセスが SIGINT や SIGTERM を受
// け取った場合に終了します。

int
bstar_start (bstar_t *self)
{
    assert (self->voter_fn);
    s_update_peer_expiry (self);
    return zloop_start (self->loop);
}
```

これを利用することでサーバーのメインプログラムはこんなにも短くなります。

bstarsrv2.c: リアクタークラスを利用したバイナリー・スター サーバー

```
//  リアクタークラスを利用したバイナリー・スター サーバー

// ライブラリをリンクせずビルドする為に
#include "bstar.c"

// エコーサービス
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 [])
{
    // 引き数は以下のどちらかになります
    //      プライマリーサーバーの場合 -p tcp://localhost:5001
    //      バックアップサーバーの場合 -p 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;
}
```

4.14 ブローカー不在の信頼性 (フリーランスパターン)

これまで ØMQ は「ブローカー不在のメッセージング」であると説明してきましたので、ブローカー中心の信頼性に頼ることは皮肉に見えるかもしれません。しかし、大抵の場合実際のメッセージング・アーキテクチャは分散とブローカーによるメッセージングを組み合わせて利用されます。あなたはトレードオフを理解した上で最良の方法を選択することができます。例えば、パーティ用のワインを 5 ケース買うために遠くの卸業者まで車を 20 分走らせることも出来ますし、夕飯のために 1 本のワインを買うだけであれば歩いて近くのスーパーに買いに行くという選択も出来ます。時間、エネルギー、価格に対する評価は現実の世界の経済では大きく状況に依存して決まります。そしてこれらはメッセージングアーキテクチャの最適化の為には不可欠な事です。

これが、ØMQ はブローカー中心のアーキテクチャを強制しないのにも関わらず、ブローカーやプロキシなどの例を多く説明してきた理由です。

それでは、これまで説明してきたブローカー中心の信頼性を解体し、フリーランスパターンと呼んでいる P2P な分散アーキテクチャを紹介してこの章を終わります。このパターンは名前解決システムの様な用途で利用します。ØMQ アーキテクチャにおいて、接続先のエンドポイントを知る方法は一般的な課題です。IP アドレスをハードコードなんてしたくは無いでしょうし、設定ファイルを作成すると管理者が悪夢を見るでしょう。あなたの利用している全ての PC や携帯電話のブラウザで「google.com」や「74.125.230.82」という文字を入力することを想像してみて下さい。

ここで実装する単純な ØMQ ネームサービスは以下の事を行う必要があります。

- 少なくとも、論理名から bind するエンドポイントへ名前解決を行い、エンドポイントに接続します。ひとつのサービス名は複数のエンドポイントを持つかもしれません。
- 例えば、テスト環境と本番環境のように複数の環境をコードを書き換えずに切り替えられる事が出来ます。
- これがサービス不能になるとアプリケーションがネットワークに接続できなくなるので信頼性は高くなければなりません。

サービス指向の Majordomo ブローカーの背後にネームサービスを配置することはそこそこ賢い考えです。しかし、クライアントがネームサービスへ直接接続する様にした方がより単純になるでしょう。

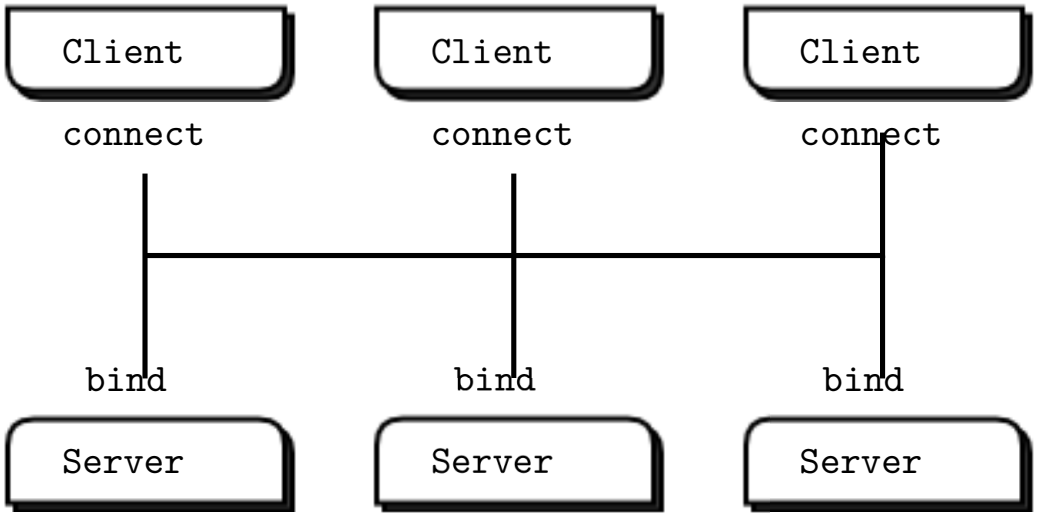


図 4.9　フリーランスパターン

ここで想定する障害は、サーバーのクラッシュやリスタート、バグによるビジーループ、サーバー負荷、ネットワーク障害です。信頼性を高める為に、複数のネームサーバーを複数配備する事で、1 台のサーバーがクラッシュしてもクライアントは別のサーバーに接続することが出来ます。実際には 2 台で十分でしょうが、この例では何台でも増やせるように設計します。

このアーキテクチャは、膨大なクライアント群が少数のサーバ群に対して直接接続を行い、サーバーは個別のアドレスを bind します。これはワーカーがブローカーに接続する Majordomo の様なアーキテクチャと根本的に異なります。クライアント側には幾つかの実装方法があります。

- REQ ソケットでものぐさ海賊パターンを利用します。これは簡単ですが落ちたサーバーに何度も再接続しないように工夫が必要です。
- DEALER ソケットを利用し、応答が得られるまで各サーバーに対してリクエストを投げ続けます。上品な方法ではありませんが効果的です。
- ROUTER ソケットを利用して特定のサーバーに接続します。しかしどうやってサーバーのソケット ID を知れば良いのでしょうか? いずれかのサーバーが最初にクライアントに対して PING を送る複雑な方法と、サーバーに固定的な ID をハードコードするという気持ちの悪い方法があります。

以下の節でそれぞれ方法を実装していきます。

4.14.1 モデル 1: 単純なリトライとフェイルオーバー

私達の目の前には 3 種類の選択肢が提示されています。単純な方法か、荒っぽいやり方か、複雑で面倒な方法です。まずひねくれてない単純な方法で実装してみましょう。というわけでものぐさ海賊パターンを複数のサーバーに対応するように書きなおします。

まず、引き数にエンドポイント名を指定して、1 つ以上のサーバーを起動して下さい。

flserver1.c: フリーランスサーバー モデル 1

```
// フリーランスサーバー - モデル 1
// いつものエコーサービス

#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;          // 割り込み
        zmsg_send (&msg, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

続いて、1 つ以上のエンドポイントを指定してクライアントを起動します。

flclient1.c: フリーランスクライアント モデル 1

```
// フリーランスクライアント - モデル 1
// 問い合わせに REQ ソケットを利用します

#include "czmq.h"
#define REQUEST_TIMEOUT     1000
#define MAX_RETRIES         3       // 諦めるまでの回数

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);

    // リクエストを送信し、安全に応答をを待ちます
    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);

    // ソケットを閉じます。
    zsocket_destroy (ctx, client);
    return reply;
}

// このクライアントは通信するサーバーが 1 つであれば、ものぐさ海賊パター
// ンと同じ戦略になります。
// サーバーが 2 つ以上であれば、それぞれのサーバーに対して通信を行います。

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) {
        // エンドポイントが 1 つであれば数回再試行を行います
```

```
        int retries;
        for (retries = 0; retries < MAX_RETRIES; retries++) {
            char *endpoint = argv [1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          // 成功
            printf ("W: no response from %s, retrying...\n", endpoint);
        }
    }
    else {
        // 複数のエンドポイントがあればそれぞれにリクエストを行います
        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;
}
```

以下のように実行します。

```
flserver1 tcp://*:5555 &
flserver1 tcp://*:5556 &
flclient1 tcp://localhost:5555 tcp://localhost:5556
```

基本的にはものぐさ海賊パターンと同じですが、クライアントはただ一つの応答を得ることを目的としています。プログラムは動作しているサーバー数に応じて 2 つに分岐しています。

- サーバーが 1 つの場合、クライアントはものぐさ海賊パターンと同様に何度かリトライを行います。
- サーバーが複数の場合、応答が得られるまで全てのサーバーに対してリトライを行います。

ものぐさ海賊パターンにはバックアップサーバーにもつながらない場合に問題がありました

が、ここではこの欠点を解決しています。

しかし、この設計にも欠点があります。最初のサーバーがダウンしている場合、ユーザーは毎回痛みを伴うタイムアウトを待つことになります。

4.14.2 モデル 2: ショットガンをぶっ放せ

それでは DEALER ソケットに切り替えてみましょう。ここでの私達の目的は、一部のサーバーがダウンしてしたとして可能な限り素早く応答を得ることです。クライアントは以下の方針で実装します。

- 全てのサーバーに接続します。
- リクエストを行う際はサーバーに対してリクエストを何度も投げ続けます。
- 最初の応答が得られたらこれを読みます。
- 残りの応答は全て無視します。

これを行うと何が起こるかというと、全てのサーバーが動作している場合はそれぞれのサーバーがリクエストを受け取り、応答を返します。どれかのサーバーが落ちている時は、動作しているサーバーに対してリクエストが送信されます。従って、サーバーは 2 度以上の重複したリクエストを受け取る可能性があります。

クライアントにとって面倒なのは、返ってくる複数の応答を処理しなければならない事。しかもリクエストと応答はサーバーのクラッシュなどが原因で失われてしまう可能性があるので、いくつ返ってくるかを予め知ることは出来ません。

従って、クライアントは要求番号と一致しない応答を全て無視する必要があります。echo サーバーではこのモデルの題材にふさわしくありませんので、違う動作を行うサーバーを実装してみます。モデル 2 のサーバーは応答のメッセージを読み込み、要求番号と一致している事と「OK」というメッセージの本文を確認します。つまり、この応答メッセージは「シーケンス番号」と「本文」の 2 つのフレームを含んでいます。

バインドするエンドポイントを指定して 1 つ以上のサーバーを起動します。

flserver2.c: フリーランスサーバー モデル 2

```
// フリーランスサーバー - モデル 2
// リクエストを受け取ったら何らかの処理を行い、シーケンス ID を付けて OK を
// 返します

#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;          // 割り込み
        // 予期しないメッセージを受け取ると失敗します
        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;
}
```

そして、接続するエンドポイントを引き数に指定してクライアントを起動します。

flclient2.c: フリーランスクライアント モデル 2

```
// フリーランスクライアント - モデル 2
// DEALER ソケットを利用してリクエストをぶっ放します。

#include "czmq.h"

// クライアント API は CZMQ を利用してクラス化します
#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

// 単一のサービスがこの時間内に応答しなければ諦めます
#define GLOBAL_TIMEOUT 2500

int main (int argc, char *argv [])
{
    if (argc == 1) {
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
        return 0;
    }
    // フリーランスクライアントオブジェクトを生成
    flclient_t *client = flclient_new ();

    // 各エンドポイントに接続
    int argn;
    for (argn = 1; argn < argc; argn++)
        flclient_connect (client, argv [argn]);

    // 1 万回の名前解決リクエストを送信し、時間を計測する。
    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;
}

// これは flclient クラスの実装です。このクラスのインスタンスはコンテキス
// トとサーバーの通信に利用する DEALER ソケット、接続しているサーバー数の
// カウンター、リクエストのシーケンス番号を持っています。
struct _flclient_t {
    zctx_t *ctx;        // コンテキスト
    void *socket;       // サーバーの通信に利用する DEALER ソケット
    size_t servers;     // 接続しているサーバー数
    uint sequence;      // リクエストのシーケンス番号
};

// コンストラクタ

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;
}

// デストラクタ

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;
    }
}

// 新しいサーバーに接続
```

```
void
flclient_connect (flclient_t *self, char *endpoint)
{
    assert (self);
    zsocket_connect (self->socket, endpoint);
    self->servers++;
}

// このメソッドは多くの処理を行います。まず全ての接続済みのサーバーに対
// して並列にリクエストを送信します。(この時点で全てのサーバに接続して
// いる必要があります。) その後、ひとつの応答が返ってきたらこれを呼び出
// し元返し、それ以外の応答を破棄します。

zmsg_t *
flclient_request (flclient_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);
    zmsg_t *request = *request_p;

    // リクエストフレームの先頭にシーケンス番号と空のフレームを追加する
    char sequence_text [10];
    sprintf (sequence_text, "%u", ++self->sequence);
    zmsg_pushstr (request, sequence_text);
    zmsg_pushstr (request, "");

    // リクエストを全ての接続済みのサーバーにぶっ放す
    int server;
    for (server = 0; server < self->servers; server++) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->socket);
    }

    // 全てのサーバーからの応答を待ちます。
    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) {
            // 応答は [empty][sequence][OK] というフォーマットです
            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;
}
```

クライアントの実装について注意すべき点は以下の通りです。

- ØMQ コンテキストを作成する汚れ仕事やソケット、およびサーバーとの通信は綺麗に構造化されたクラスベースの API により隠蔽しています。
- 数秒間どのサーバーからも応答が無ければ、クライアントは応答を待つのを止めます。
- クライアントは、正しい REP エンベロープを作成する必要があります。例えばメッセージフレームの前に空のフレームを追加する事などです。

クライアントで 1 万回の名前解決を実行し、平均コストを計測してみます。私のテストマシンでは 1 台のサーバーと通信するのに 60 ミリ秒、3 台のサーバーと通信すると 80 ミリ秒掛かりました。

このショットガン方式の利点と欠点は、

- 利点: そこそこ単純で理解しやすいです。
- 利点: フェイルオーバーが機能し、少なくとも 1 台のサーバーが動作していれば迅速に動作します。。
- 欠点: 無駄なネットワークトラフィックが発生します。
- 欠点: プライマリーやセカンダリなど、サーバーの優先順位を決めることが出来ません。
- 欠点: ひとつのリクエストに対して全てのサーバーが処理を行う必要があります。

4.14.3 モデル 3: 複雑で面倒な方法

ショットガンをぶっ放すのがとても良い手段であることは真実です。しかし全ての代替案について検討してみるのが科学というものです。私達はこれから更に複雑で面倒な選択肢を提案しますが最終的にショットガンをぶっ放すのが望ましいと気がつくでしょう。これから紹介するのは私が辿った道です。

主な問題は ROUTER ソケットに置き換えることで解決可能です。クライアントは落ちているサーバーを予め知っておきリクエストを避ける方が一般的にスマートな方法でしょう。そし

て ROUTER ソケットに置き換える事でサーバーがシングルスレッドっぽくなる問題を解決できます。

しかし、ID の設定されていない匿名な 2 つの ROUTER ソケット同士で通信を行うことは出来ません。最初のメッセージを受け取る際に両側で接続相手の ID を生成すれば良いのですが、ID が無いので最初のメッセージを受け取る事が出来ません。この難問を解決する唯一の方法はカンニングです。つまり ID をハードコーディングするしかありません。クライアント・サーバーモデルにおけるカンニングの正しい方法は、クライアントがサーバーの ID を知っていることにすることです。他の方法もありますが複雑で厄介なのでやめたほうが良いでしょう。クライアントは何台立ち上がるか判らないからです。愚かで、複雑で、厄介な事をするのは暴君の特徴ですが、恐ろしいことにソフトウェアにも当てはまります。

管理を行うための新しい概念を発明するのではなく、接続エンドポイントを ID として利用します。エンドポイントは両者が事前知識なしに合意できるユニークな文字列です。これは 2 つの ROUTER ソケットを接続するための卑劣かつ有効な方法です。

ØMQ の ID がどの様に利用されるかを思い出して下さい。サーバーの ROUTER ソケットはソケットを bind する前に ID を設定します。クライアントが接続した際、メッセージをやりとりする前にちょっとしたハンドシェイクを行い ID を交換します。クライアント側の ROUTER ソケットは ID を設定せず、空の ID で送信します。そしてサーバーはランダムな UUID を生成してクライアントに送信します。

これはクライアントがメッセージをサーバーにルーティングできる事を意味します。ID としてサーバーのエンドポイントを指定すると直ちに接続が確立します。それは正確には `zmq_connect()` を実行した直後ではありませんがしばらく経てば接続されます。ここに問題があります。私達はサーバーへの接続が完了する正確なタイミングを知ることが出来ません。もしサーバーがオンラインであれば数ミリ秒で接続は完了するでしょうが、サーバーが落ちていてシステム管理者がお昼ごはんを食べていれば 1 時間ほどかかってしまうでしょう。

ここにちょっとしたパラドックスがあります。私達はサーバーへの接続が確立するタイミングを知る必要がありますが、これまで見てきたようにフリーランスパターンではサーバーと通信してみるまでサーバーがオンラインかどうかを知ることが出来ません。

そこで私はモデル 2 で利用したショットガンの方法を組み合わせてこの問題を解決しました。このある意味無害なショットを撃つことで、クライアントはサーバーの変化を知ることが出来ます。これには実際のリクエストを送るのではなくハートビートによる PING-PONG を行って確認を行います。

またプロトコルの話になりましたので、フリーランスクライアントがサーバーと PING-PONG コマンドやリクエスト・応答をやりとりする為の短い仕様を用意しました。

サーバー側の実装は短くて良い感じです。これを FLP プロトコルと呼んでいます。

flserver3.c: フリーランスサーバー モデル 3

```
// フリーランスサーバー - モデル 3
// 1 スレッドで ROUTER/ROUTER ソケットを利用します
//  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 ();

    // サーバーソケットを準備します
    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;          // 割り込み

        //  Frame 0: クライアントの ID
        //  Frame 1: PING、もしくはクライアントの制御フレーム
        //  Frame 2: リクエスト本体
        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_push (reply, identity);
```

```
        if (verbose && reply)
            zmsg_dump (reply);
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

こちらがフリーランスクライアントですが大きくなってしまいましたのでクラスに分離しました。こちらがメインプログラムです。

flclient3.c: フリーランスクライアント モデル 3

```
// フリーランスクライアント - モデル 3
// フリーランスパターンをカプセル化した flcliapi クラスを利用します。

// ライブラリをリンクせずビルドする為に
#include "flcliapi.c"

int main (void)
{
    // フリーランスクライアントオブジェクトの生成
    flcliapi_t *client = flcliapi_new ();

    // 複数のエンドポイントに接続
    flcliapi_connect (client, "tcp://localhost:5555");
    flcliapi_connect (client, "tcp://localhost:5556");
    flcliapi_connect (client, "tcp://localhost:5557");

    // 1000 回の名前解決リクエストを送信し、時間を計測します
    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;
}
```

そしてこちらが Majordomo ブローカーと同じくらい巨大で複雑になってしまったクライアント API クラスのコードです。

flcliapi.c: フリーランスクライアント API

```
// flcliapi クラス - フリーランスパターンのエージェントクラス
// 以下のフリーランスプロトコルの実装
// http://rfc.zeromq.org/spec:10

#include "flcliapi.h"

// この時間内にサーバーの応答がなければリクエストを破棄します
#define GLOBAL_TIMEOUT  3000    // ミリ秒
// PING 間隔
#define PING_INTERVAL   2000    // ミリ秒
// この時間サーバーが応答しない場合は落ちたと判断します
#define SERVER_TTL      6000    // ミリ秒

// この API は大きく別けて 2 つの部分から構成されています。1 つ目はアプリケー
// ションと動作するフロントエンド、もう一つはバックグラウンドスレッドで
// 動作するエージェントです。フロントエンドとバックエンドはプロセス内通
// 信ソケットを利用して通信します・

// フロントエンドクラスの構造

struct _flcliapi_t {
    zctx_t *ctx;        // コンテキスト
    void *pipe;         // flcliapi エージェントへのパイプ
};

// エージェントスレッドの本体です
static void flcliapi_agent (void *args, zctx_t *ctx, void *pipe);

// コンストラクタ

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;
}

// デストラクタ

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;
    }
}

// こちらは接続メソッドです。フロントエンドオブジェクトはバックエンドエー
// ジェントにマルチパートメッセージを送信します。最初のフレームは
// 「CONNECT」という文字列で次はエンドポイントを含めます。そして接続が
// 行われるまで 100 ミリ秒待ちます。これはあまりよい方法ではありませんが、
// 起動時に全てのリクエストが単一のサーバーに送られるのを防ぎます。

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);      // 接続が行われるまで待つ
}

// リクエストメソッドの実装です。フロントオブジェクトがバックエンドにメッ
// セージを送る際、「REQUEST」コマンドを指定してリクエストを行います。
```

```
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;
}

// こちらはサーバークラスです

typedef struct {
    char *endpoint;             // サーバーの ID/エンドポイント
    uint alive;                 // 生きていれば 1 になります
    int64_t ping_at;            // 次の PING 送信時刻
    int64_t expires;            // 有効期限
} 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;
}


// こちらは様々なソケットに到達するメッセージを処理するエージェントクラ
// スです

typedef struct {
    zctx_t *ctx;                // コンテキスト
    void *pipe;                 // アプリケーションと通信するソケット
    void *router;               // サーバーと通信するソケット
    zhash_t *servers;           // 接続済みのサーバー
    zlist_t *actives;           // 生きていることを確認済みのサーバー
    uint sequence;              // これまで送信したリクエスト数
    zmsg_t *request;            // 現在のリクエスト
    zmsg_t *reply;              // 現在の応答
    int64_t expires;            // リクエスト・応答のタイムアウト
} 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;
    }
}

// サーバーの削除時にコールバックされます

static void
s_server_free (void *argument)
{
    server_t *server = (server_t *) argument;
    server_destroy (&server);
}

// このメソッドはフロントエンドクラスからのメッセージを処理します。
// (それは CONNECT もしくは REQUEST コマンドのどちらかです)

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);    // リクエスト・応答のサイクルのチェック
        // リクエストにシーケンス番号と空のフレームを付加します
        char sequence_text [10];
        sprintf (sequence_text, "%u", ++self->sequence);
        zmsg_pushstr (msg, sequence_text);
        // リクエストメッセージの主導権を取ります
        self->request = msg;
        msg = NULL;
        // リクエストのタイムアウトを設定
        self->expires = zclock_time () + GLOBAL_TIMEOUT;
    }
    free (command);
    zmsg_destroy (&msg);
}

// このメソッドはサーバーからのメッセージを一つ処理します

void
agent_router_message (agent_t *self)
{
    zmsg_t *reply = zmsg_recv (self->router);

    // 最初のフレームは応答するサーバーのエンドポイントです
    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;

    // 次のフレームはシーケンス番号です
    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);
}

// こちらはエージェントタスクです。
// 2 つのソケットを監視して受信したメッセージを処理します

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) {
        // タイマーを一時間後に設定
        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;              // コンテキストの終了

        if (items [0].revents & ZMQ_POLLIN)
            agent_control_message (self);
```

```
        if (items [1].revents & ZMQ_POLLIN)
            agent_router_message (self);

        // リクエストの処理中であれば、次のサーバーに渡します
        if (self->request) {
            if (zclock_time () >= self->expires) {
                // リクエストの有効期限切れ、失敗させます
                zstr_send (self->pipe, "FAILED");
                zmsg_destroy (&self->request);
            }
            else {
                // 送信するサーバーを探すと同時に有効期限切れのサーバー
                // を削除する
                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;
                    }
                }
            }
        }
        // 有効期限切れのサーバーを切断、削除し、有効なサーバーにハート
        // ビートを送信します
        zhash_foreach (self->servers, server_ping, self->router);
    }
    agent_destroy (&self);
}
```

この API の実装はこれまでに紹介していない幾つかのテクニックを利用しています。

- マルチスレッド API: クライアント API は 2 つの部分から構成されています。アプリケーションから同期的に呼び出される flcliapi クラスとバックグラウンドで非同期に実行される agent クラスです。ØMQ はマルチスレッドアプリケーションを簡単に作成できると説明したことを思い出して下さい。flcliapi クラスと agent クラスはお互いにプロセス内通信を行っています。ØMQ に関する操作 (コンテキストの生成や破棄など) は API の中に隠蔽しています。agent はバックグラウンドでサーバーと通信しリクエスト

を行う際に有効なな接続先を選択できるようなちょっとしたブローカーのような役割を持っています。

- Tickless タイマー: これまで見てきたポーリングループでは 1 秒程度の固定のタイムアウト間隔を利用してきました。これは単純ですがタブレットやスマートフォンなどの非力なクライアントでは CPU コストを消費してしまうので最適ではありません。地球を救うために agent では Tickless タイマーを利用しましょう。Tickless タイマーは期待するタイムアウト値に基づいてポーリング時間を計算します。一般的な実装ではタイムアウトの順序リストを保持し、ポーリング時間を計算します。

4.15 まとめ

この章では、リクエスト・応答パターンに様々な信頼性を持たせる方法を見てきました。サンプルコードは最適化されていませんが、十分実用に使えるレベルです。なにより対照的なのは、ブローカーに信頼性持たせる Majordomo パターンとブローカーの無いフリーランスパターンです。

あとがき

ライセンス

私は多くの人達にこのテキストを雑誌、本、プレゼンテーションなどの仕事で再利用して欲しいと思っています。誰かがこの本を再編集したら他の誰かがまた再編集出来ることが条件になります。誰かが再編集したものを販売してもそれは私にとって名誉であり異論はありません。このテキストは cc-by-sa ライセンスで公開しています。

当初、サンプルコードは GPL で公開していましたが、上手くいかない事がすぐにわかりました。サンプルーコードはそのコード片を幅広い用途で再利用出来る必要があります。GPL ではそれが困難でした。そこで、サンプルコードのライセンスを MIT/X11 に切り替えました。規模が大きくてより複雑なサンプルコードに関しては LGPL でも上手くいくだろうと考えられます。

そして、サンプルコードは Majordomo の様に独立したプロジェクトに移行を始めており、そこでは LGPL を使用しています。重ねて言いますが、普及と再編集性に適しているからです。ライセンスはツールであり、目的があって選択しているのであってイデオロギーではありません。