ZeroMQ—指导

由 iMatix 公司的首席执行官 Pieter Hintjens <moc.xitami|hp#moc.xitami|hp>编写。感谢 Bill Desmarais, Brian Dorsey, CAF, 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, rofl0r, Mihail Minkov, Jeremy Avnet, Michael Compton, and Zed Shaw 的贡献，也感谢 Stathis Sideris，因为 Ditaa.。

请对所有的意见和勘误表进行问题跟踪。这个版本覆盖了 0MQ2.0 的版本，发表于周二 2010 年 11 月 9 日，9 时 32 分 19 秒。

第一章——基础的东西

修理（fixing）这个世界

怎么解释 0MQ？有些人会说它的所有美好的事情。它是类固醇（steroids）上的套接字。它像有路由的邮箱。它很快。别人想分享它的启蒙，当这一切变得越来越明显，人们开始顿悟了。事情变得更简单。再也不复杂了。它打开了人们的思维。其他人喜欢通过做比较的方式来解释。它更小，更简单，但是看起来仍然很熟悉。就我个人而言，我希望回忆起我们为什么要开发 0MQ，因为，这是读者们也很想知道的问题。

编程是装扮成艺术的科学，因为我们中的大多数不了解软件的物理过程。如果学过这方面的知识的话，也学的很少。软件的物理过程不是算法，数据结构，语言和抽象。这些只是我们研发，使用，然后扔掉的工具。软件真正的物理过程实际上是人们的思维过程。

我们都有自己的局限性，当事情变的复杂的时候，我们希望一起协作，把大的问题分成小的问题来处理。这就是科学的编程，开发人们能够理解和容易使用的模块。并且，人们会一起协作来解决很大的问题。

我们生活在一个连通的世界，当代的软件必须引导这个世界。因此，未来的大型软件的编连模块应该是连通的，并且是并行的。代码不能再是“强大而沉默”的。代码必须和代码对话。代码必须是能交谈的，友善的，容易连通的。代码必须像人脑一样运行。万亿的神经元彼此发送消息，一个没有中央控制的大规模并行网络，没有单点失败，能够解决很困难的问题。毫无疑问，将来的代码将会运行的像人脑，因为，每个网络的终端都看起来有些像人脑。

如果你做过一些关于线程，协议或者网络的工作，你就会意识到这是非常不可能的。它是一个梦。当你真正处理生活中的情形时，甚至利用少量的套接字连接少量的程序都让人不胜其烦。万亿？这个成本是没法想象的。连接计算机如此困难，所以，它的软件和技术实现的费用达十几亿美元。

因此，我们现在的状况是，布线的能力比我们能够使用它的能力超前了几年。我们在上世纪 80 年代遇到了软件危机，当时像 Fred Brooks 这样的人认为没有解决办法。免费的开

源软件解决了这次危机，让我们能够高效地分享知识。今天，我们遇到了另一个软件危机，但是我们对这个话题谈论的不多。只有最大，最有钱的公司能够建立连接的应用程序。虽然有云端运算网络，但是它是私有的。我们的数据 、知识从我们的电脑里消失，进入我们不能访问、不能竞争的云网络。谁拥有我们的社会网络？相反，它像大型电脑革命。

我们把关于政治哲学的问题留给别的书来解决吧。问题是，因特网提供了大量代码能够连接在一起的可能的同时，事实是对于我们中的大多数，都不能够实现它。因此，因为没有办法连接代码，很多人们感兴趣的问题（如健康，教育，经济，运输等）仍然没有解决，也没有办法连接人们的聪明来解决这些问题。

人们已经做过很多连接软件的尝试。因特网工程特别任务组制定了数以千记的规范，每个规范解决部分问题。对应用开发人员来说，HTTP 可能是一个很简单的解决方案，但是它却可能让问题变得更糟糕，它鼓励开发人员和工程师开发大型服务器和瘦的并且愚蠢的客户机。

因此，今天人们仍然通过原始 UDP 或者 TCP，专门的协议，HTTP，网络套接字来连接引用程序。它仍然很困难，慢，难以升级，并且很集中。分布式 P2P 结构大多数只能是玩玩，不能用于工作。有多少应用程序是使用 Skype 或者 Bittorrent 来交换数据的呢？

什么能够带我们回到科学的编程？为了修理（fixing）世界，我们需要做两件事。一件是，解决这个普遍问题“如何在任何地方把任意的代码连接起来。”另一件是，用人们容易理解和使用的简单的模块把它封装起来。

它听起来简单的荒谬。但可能它确实就是这么简单。它就是问题的所在。

用一百个字来描述 0MQ

0MQ 看起来像是嵌入的网络库，但行为像一个并发的框架。它提供带所有信息并且跨各种传输协议如进程间，进程内，广播等的套接字给你。你可以用多种方式实现 N 对 N 的套接字连接，如扇出，发布订阅，请求应答。它足够快，因此可以制作集群。他的异步输入输出模式为你提供了可扩展的多核应用，用来处理异步消息任务。它提供很多语言的套接字，并且能够在很多操作系统上运行.0MQ 是 iMatix 公司研发的，是开源协议。

一些假设

我们假设你使用的是最新版本的 0MQ，而且你是当今的 git 高手。我们假设你使用的是 Liux 系统或者类似的。我们假设你能够读懂 C 语言代码，因为我们的很多代码是用 C 语言编写的。我们假设如果程序需要，当我们写常量如 PUSH 或者 SUBSCRIBER 你能够想的到它们实际上是 ZMQ-PUSH 或者 ZMQ-SUBSCRIBER。

请求与响应

让我们开始写一些代码，我们当然是以世界你好这个例子开始。我们将会构造一个服务器和一个客户机。客户机发送“HELLO”给服务器，将会得到响应“World”。这是服务器程序，它在端口 5555 打开一个 0MQ 套接字，并在端口读取请求，并对每一个请求都以

“World”应答。

```
//
//  Hello World server
//  Binds REP socket to tcp://*:5555
//  Expects "Hello" from client, replies with "World"
//
#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main () {
    void *context = zmq_init (1);

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

    while (1) {
        //  Wait for next request from client
        zmq_msg_t request;
        zmq_msg_init (&request);
        zmq_recv (responder, &request, 0);
        printf ("Received request: [%s]\n",
            (char *) zmq_msg_data (&request));
        zmq_msg_close (&request);

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

        //  Send reply back to client
        zmq_msg_t reply;
        zmq_msg_init_size (&reply, 6);
        memcpy ((void *) zmq_msg_data (&reply), "World", 6);
        zmq_send (responder, &reply, 0);
        zmq_msg_close (&reply);
    }
    zmq_term (context);
    return 0;
}
```

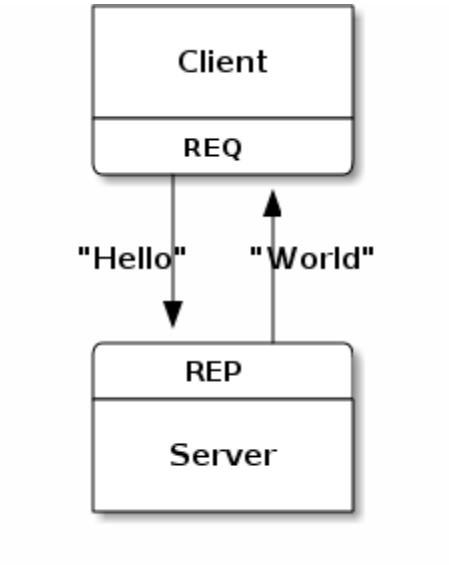

Figure 1 – Request-Reply

套接字对 REQ-REP 是步伐一致的。客户机调用 ZMQ 发送函数，然后调用函数 zmq_send(3)，然后调用函数 zmq_recv(3),这是一个循环（或者如果需要的话只调用一次）。使用别的任何序列（如在一行发送两个消息）都会出错。类似的，服务器按照先调用函数 zmq_recv(3)，然后调用函数 zmq_send(3)的顺序，调用次数根据需要而定。

0MQ 使用 C 语言作为参考语言，并且我们也用 C 语言作为例子。如果你是在线入读本文，这个例子下面的链接将会帮你转换为别的程序语言。让我们用 C++编写的相同的服务器程序作为对比。

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

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

    while (true) {
        zmq::message_t request;

        //  Wait for next request from client
        socket.recv (&request);
        printf ("Received request: [%s]\n",
```

```
            (char *) request.data ());

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

        //  Send reply back to client
        zmq::message_t reply (6);
        memcpy ((void *) reply.data (), "World", 6);
        socket.send (reply);
    }
    return 0;
}
```

你可以看到在 C 语言或者 C++语言中使用的 0MQ 套接字很相像。像 pathon 这样的语言，我们甚至可以隐藏更多，读起来更简单。

这个是 C 语言的客户机代码（单击源程序西面的链接，或者转换成你最喜欢的程序语言）：

```
//
//  Hello World client
//  Connects REQ socket to tcp://localhost:5555
//  Sends "Hello" to server, expects "World" back
//
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main () {
    void *context = zmq_init (1);

    //  Socket to talk to server
    printf ("Connecting to hello world server...\n");
    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++) {
        zmq_msg_t request;
        zmq_msg_init_data (&request, "Hello", 6, NULL, NULL);
        printf ("Sending request %d...\n", request_nbr);
        zmq_send (requester, &request, 0);
        zmq_msg_close (&request);
```

```
        zmq_msg_t reply;
        zmq_msg_init (&reply);
        zmq_recv (requester, &reply, 0);
        printf ("Received reply %d: [%s]\n", request_nbr,
            (char *) zmq_msg_data (&reply));
        zmq_msg_close (&reply);
    }
    zmq_close (requester);
    zmq_term (context);
    return 0;
```

这个看起来简单得不现实。但是，一个 0MQ 套接字就是当你把一个通用的 TCP 套接字，注入从苏维埃秘密核研究项目偷来的混合的放射性同位素，用上世纪 50 年代的宇宙射线轰炸它，再把它放在一个有膨胀的肌肉凸出在弹性合成纤维里的拙劣伪装成神物的，脑袋被药弄坏了的漫画书作者的手里得到的。0MQ 套接字是拯救网络世界的超级英雄。

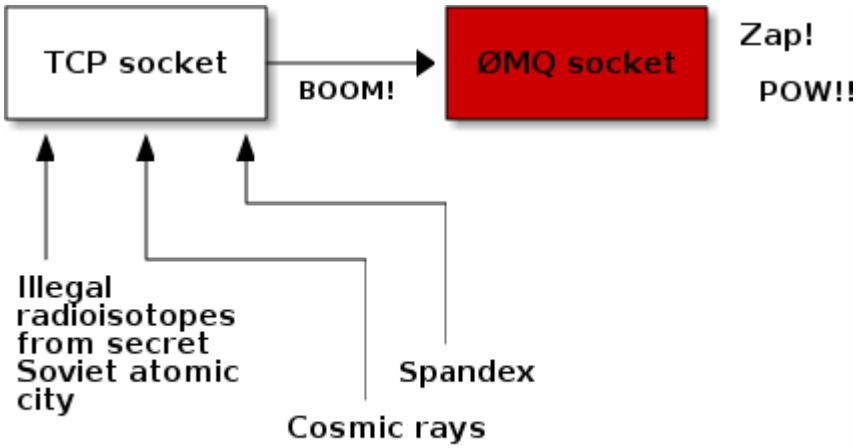

Figure 2 – A terrible accident...

一次让几千个客户机向服务器发送请求，它也可以工作的很好，很快。试着玩玩，先打开客户机，再打开服务器，看看它是怎么继续工作的，想一下这意味着什么。

让我简单的解释一下这两个程序实际上在干什么。他们创建了 0MQ 工作的环境，和套接字。不要担心不懂这些词语的意思。你慢慢会明白的。服务器在端口 5555 绑定它的 REP(应答)。服务器等待一个请求，在一个循环中，它每次都会用一个应答来响应。客户机发送一个请求并且读取来自服务器的应答。

在幕后发生了很多事，但是我们程序员关心的是代码是否简短和令人满意，甚至在重荷下它是否也不会经常崩溃掉。这是请求-应答模型，可能是 0MQ 中最简单的模型。它主要用于远程过程调用和典型的客户-服务器模式。

关于字符串的小小的注意事项

除了字节的大小外，0MQ 对你发送的数据一无所知。这就意味着你有责任对它进行安全的格式化，以便应用程序能够读回它。对象和复杂数据类型的安全格式化有专门的库比如协议缓冲区来完成。但是，即使是字符串，你也应该小心。

在 C 语言或者一些其它语言，字符串是以一个空字节结尾的，在 Hello World 的例子中，可以看到，我们把字符串和空字节一起发送。

```
zmq_msg_init_data (&request, "Hello", 6, NULL, NULL);
```

但是，如果你用另外一种语言发送数据，它可能不需要包括最后的空字节。例如，当我们用 Pathon 语言来发送相同的字符串。我们用下面的语句：

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

在线路上传输的是：

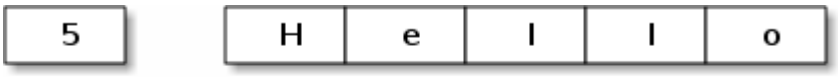

Figure 3 – A ØMQ string

如果你从 C 语言程序读到的，你可能会获得像字符串的东西，可能碰巧表现的像个字符串（很幸运，这五个字节后面恰好跟着一个空字节），但它不是一个正确的字符串。

这就意味着，如果你混合使用 C 语言和另外一种语言来编写 Hello World 的客户机服务器程序，你很可能会得到奇怪的结果。事实上，我希望你得到奇怪的结果，因为这样会帮助你理解这部分。

如果你接收到一个来自 0MQ 的字符串，是用 C 语言编写的，你不能轻易相信它是正确结尾的。每次收到一个字符串，你都需要分配一个新的有多余字节空间的缓冲区，复制这个字符串，并以一个空字符结尾。

因此，我们制定了规则， 0MQ 字符串是定长的，在线路上发送不会尾随着一个空字节。最简单的情况（在我们的例子中，我们将会做这件事）一个 0MQ 字符串对应一个 0MQ 消息结构，如上图所示，包括长度和一些字节。

这个程序描述的是我们接收来自 0MQ 的字符串，并且把它作为一个有用的字符串传送给应用程序的过程，使用的是 C 语言。

application as a valid C string:

```
//  Receive ØMQ string from socket and convert into C string
static char *
s_recv (void *socket) {
    zmq_msg_t message;
    zmq_msg_init (&message);
```

```
    zmq_recv (socket, &message, 0);
    int size = zmq_msg_size (&message);
    char *string = malloc (size + 1);
    memcpy (string, zmq_msg_data (&message), size);
    zmq_msg_close (&message);
    string [size] = 0;
    return (string);
}
```

这是一个很方便的帮助函数，它是本着有利于再利用的精神。我们编写一个相似的's_send'函数来以正确的形式发送字符串，并把它打包到我们能够重复利用的头文件中。

下面就是 zhelpers.h 文件，它让我们能够用 C 语言编写令人喜欢的，简短的应用程序。

```
/*
=====================================================================
====
    zhelpers.h

    Helper header file for example applications.



---------------------------------------------------------------------
----
    Copyright (c) 1991-2010 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it under
the
    terms of the GNU Lesser General Public License as published by the
Free
    Software Foundation; either version 3 of the License, or (at your
option)
    any later version.

    This software is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABIL-
    ITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General
    Public License for more details.
```

```
    You should have received a copy of the GNU Lesser General Public
License
    along with this program. If not, see <http://www.gnu.org/licenses/>.

=============================================================================
*/

#ifndef __ZHELPERS_H_INCLUDED__
#define __ZHELPERS_H_INCLUDED__

//  Include a bunch of headers that we will need in the examples

#include <zmq.h>

#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include <assert.h>

//  Bring Windows MSVC up to C99 scratch
#if (defined (__WINDOWS__))
    typedef unsigned long ulong;
    typedef unsigned int  uint;
    typedef __int64 int64_t;
#endif

//  Provide random number from 0..(num-1)
#define within(num) (int) ((float) (num) * random () / (RAND_MAX + 1.0))

//  Receive 0MQ string from socket and convert into C string
//  Caller must free returned string.
static char *
s_recv (void *socket) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    if (zmq_recv (socket, &message, 0))
        exit (1);           //  Context terminated, exit
```

```
    int size = zmq_msg_size (&message);
    char *string = malloc (size + 1);
    memcpy (string, zmq_msg_data (&message), size);
    zmq_msg_close (&message);
    string [size] = 0;
    return (string);
}

//  Convert C string to 0MQ string and send to socket
static int
s_send (void *socket, char *string) {
    int rc;
    zmq_msg_t message;
    zmq_msg_init_size (&message, strlen (string));
    memcpy (zmq_msg_data (&message), string, strlen (string));
    rc = zmq_send (socket, &message, 0);
    assert (!rc);
    zmq_msg_close (&message);
    return (rc);
}

//  Sends string as 0MQ string, as multipart non-terminal
static int
s_sendmore (void *socket, char *string) {
    int rc;
    zmq_msg_t message;
    zmq_msg_init_size (&message, strlen (string));
    memcpy (zmq_msg_data (&message), string, strlen (string));
    rc = zmq_send (socket, &message, ZMQ_SNDMORE);
    zmq_msg_close (&message);
    assert (!rc);
    return (rc);
}

//  Receives all message parts from socket, prints neatly
//
static void
s_dump (void *socket)
{
    puts ("----------------------------------------");
    while (1) {
        //  Process all parts of the message
        zmq_msg_t message;
```

```
        zmq_msg_init (&message);
        zmq_recv (socket, &message, 0);

        //  Dump the message as text or binary
        char *data = zmq_msg_data (&message);
        int size = zmq_msg_size (&message);
        int is_text = 1;
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            if ((unsigned char) data [char_nbr] < 32
            ||  (unsigned char) data [char_nbr] > 127)
                is_text = 0;

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

        int64_t more;           //  Multipart detection
        size_t more_size = sizeof (more);
        zmq_getsockopt (socket, ZMQ_RCVMORE, &more, &more_size);
        zmq_msg_close (&message);
        if (!more)
            break;      //  Last message part
    }
}

//  Set simple random printable identity on socket
//
static void
s_set_id (void *socket)
{
    char identity [10];
    sprintf (identity, "%04X-%04X", within (0x10000), within (0x10000));
    zmq_setsockopt (socket, ZMQ_IDENTITY, identity, strlen (identity));
}

//  Report 0MQ version number
//
static void
```

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

#endif
```

版本报告

0MQ 有几种版本，如果你发现有什么问题，它将会在更新的版本中得到改善。因此确切的知道你使用的是什么样的 0MQ 版本是一个很有用的技巧。这是一个查询版本的小程序，使用一个 zhelpers.h 头文件中的函数。

```
//
//  Report 0MQ version
//
#include "zhelpers.h"

int main (void)
{
    s_version ();
    return EXIT_SUCCESS;
}
```

获得消息

第二类模型是一种数据分发的方法，一台服务器把数据发送到一系列的客户机上。让我们来看一个发送有地点，温度和相对湿度组成的气候消息。我们将产生随机值，如真正的气象站所做的一样。

这是用 C 语言编写的服务器程序。我们将在这个应用程序中使用端口 5556.

```
//
//  Weather update server
//  Binds PUB socket to tcp://*:5556
//  Publishes random weather updates
//
#include "zhelpers.h"

int main () {
    //  Prepare our context and publisher
    void *context = zmq_init (1);
```

```
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5556");
    zmq_bind (publisher, "ipc://weather.ipc");

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

        //  Send message to all subscribers
        char update [20];
        sprintf (update, "%05d %d %d", zipcode, temperature,
relhumidity);
        s_send (publisher, update);
    }
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

这个消息留没有起点也没有终点，它像一个不会终止的广播。

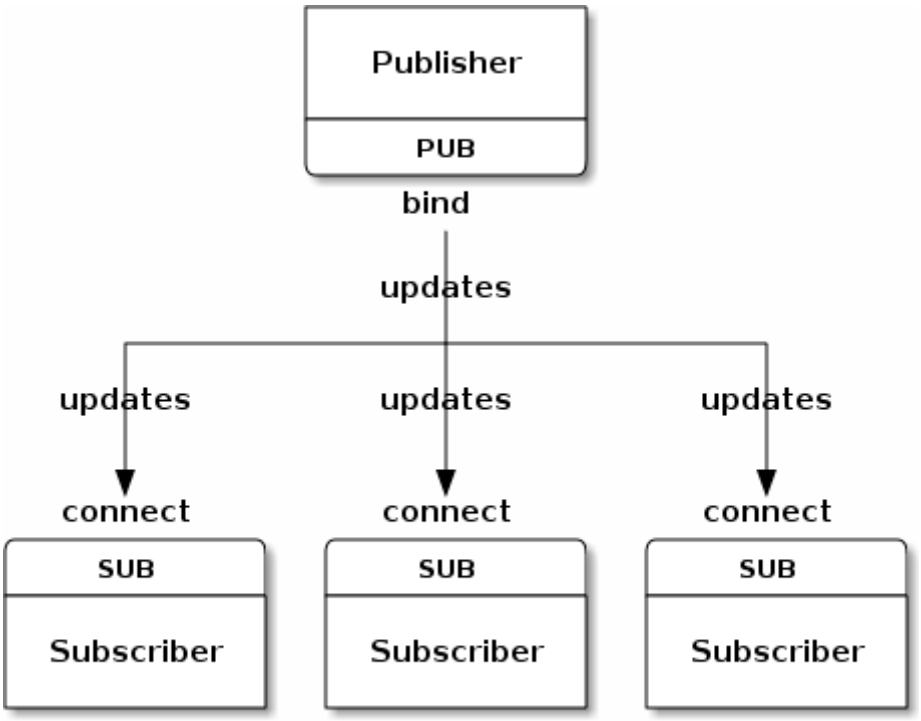

Figure 4 – Publish-Subscribe

这是服务器应用程序，它监听新的消息流，并且抓取所有与特定地址有关的消息。默认情况下，它获得的是纽约的消息，因为那是一个可以进行一切冒险的大城市。

```
//
//  Weather update client
//  Connects SUB socket to tcp://localhost:5556
//  Collects weather updates and finds avg temp in zipcode
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

    //  Socket to talk to server
    printf ("Collecting updates from weather server...\n");
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");

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

```
    //  Process 100 updates
    int update_nbr;
    long total_temp = 0;
    for (update_nbr = 0; update_nbr < 100; update_nbr++) {
        char *string = s_recv (subscriber);
        int zipcode, temperature, relhumidity;
        sscanf (string, "%d %d %d",
            &zipcode, &temperature, &relhumidity);
        total_temp += temperature;
        free (string);
    }
    printf ("Average temperature for zipcode '%s' was %dF\n",
        filter, (int) (total_temp / update_nbr));

    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

注意当你使用使用一个订阅套接字的时候，你必须利用函数 zmq_setsockopt(3)和 SUBSCRIBE 设置订阅，如这个例子中的代码一样。如果你不设置任何订阅，你将接收不到任何消息。这是初学者常犯的一个错误。也就是说，如果新消息匹配堪萨斯州，这个订阅方就会接收到它。订阅方也可以不订阅某些特定的订阅消息。订阅是定长的二进制数据块。查看函数 zmq_setsockopt(3)看看它是怎么工作的。

PUB-SUB 套接字对是异步的。客户机循环调用函数 zmq_recv(3)（或者只有一次，根据需要来定）。试图发送一个消息给一个 SUB 套接字就会出错。相似地，服务器根据需要调用函数 zmq_send(3)，但是不能用 PUB 套接字调用函数 zmq_recv(3)。

关于 PUB/SUB 套接字，有一件很重要的事情需要知道：你不可能精确的指导一个订阅方什么时候开始得到消息。即使你先打开订阅方，等一会，然后打开发布方，订阅方任然会错过一些发送方首先发生的那部分消息。这是因为当订阅方连接到发布方时（时间很短，但不为零），发布方可能把消息发送出去了。

这种“慢订阅连接”现象困惑了很多人，总是很多人，因此，我将要详细解释这个问题。记住，0MQ 在后台是异步输入/输出的。你有两个节点做这项工作，以这种顺序：

- 订阅方连接到一个终端，接收消息并计数。
- 发布方绑定到一个终端并立即发送 1000 个消息。

订阅方很可能什么也接收不到。你检查是否设置了一个正确的滤波器，然后在试一次，订阅方还是收不到任何消息。

如果利用 TCP 建立连接，涉及到来来回回的握手花费的几毫秒，具体需要的时间取决于你的网络和端点之间的跳的数量。在那段时间内 0MQ 可以发送很多消息。为了方便起见，假设建立连接需要 5 毫秒的时间，这样的链接每秒可以处理 1 兆的消息。在这 5 微秒内，订阅方连接到发布方而发不发发送 1 千的消息只需要 1 微秒的时间。

在第二章我将会解释怎样同步化发布方和订阅方，以便在订阅方连接上并准备好后你再发送才开始发布数据。有一个简单，笨的方法来延迟发布，就是利用休眠(sleep)。我从未在实际的应用程序中使用过这种方法。这种方法很脆弱，，不优雅，并且很慢。你自己使用休的方法看会发生什么事，然后请看第二章如何正确处理这件事。

同步化的转换将会假设发布的数据流是无穷的，没有开始，没有结束。这就是我们怎样建立我们的气象客户机例子。

客户机订阅到它选择的邮编代码，并且收集一千个有这样代码的新消息。如果邮编代码是随机分发的话，这就意味着大约 1 千万的新消息从服务器发出来。你可以先启动客户机，然后再启动服务器，并且客户机保持工作状态。你可以随意终止和重启服务器，但是客户机要保持工作状态。当客户机已经收集到它的一千个新消息，它计算平均值，打印，然后退出。

关于发布-订阅模式的一些问题：

- 一个订阅方实际上可以链接到不止一个发布方，每次使用一条连接命令。数据会交叉到达，因此没有哪个单一的发布方能够淹没其它发布方。
- 如果一个发布方没有与它连接的订阅方，它会简单的扔掉所有消息。
- 如果你使用 TCP，并且订阅方很慢，消息将会在发布方排队。我们将会看到随后如何使用“高水准”的方法来避免这种情况的出现。
  这是利用我的机子测试接收并滤波 10M 消息所消耗的时间的结果。我的机子是 Intel 4 核的，很快，但是没有什么特别地。

```
   ph@ws200901:~/work/git/0MQGuide/examples/c$ time wuclient
Collecting updates from weather server...
Average temperature for zipcode '10001 ' was 18F

real    0m5.939s
user    0m1.590s
sys     0m2.290s
```

分而治之

作为最后一个例子（你肯定已经厌倦了有趣的代码，希望回到关于相对抽象的语言规范的讨论上来），让我们做一点超级计算。然后我们就可以休息了。我们的超级计算应用程序是一个非常经典的并行处理模型：

- 有一个风扇（ventilator）,它产生能够并行处理的任务。
- 我们有一组处理任务的进程。
- 我们有一个监听器，收集工作进程返回的结果。

事实上，工作进程运行在非常快的机子（boxes）上，可能通过图形处理单元来处理困难的运算。这就是风扇（ventilator）的程序。。它产生 100 个任务，每个任务都是一个通知工作进程产生几微秒休眠（sleep）的消息。

```
//
//  Task ventilator
//  Binds PUSH socket to tcp://localhost:5557
//  Sends batch of tasks to workers via that socket
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

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

    printf ("Press Enter when the workers are ready: ");
    getchar ();
    printf ("Sending tasks to workers...\n");

    //  The first message is "0" and signals start of batch
    s_send (sender, "0");

    //  Initialize random number generator
    srandom ((unsigned) time (NULL));

    //  Send 100 tasks
    int task_nbr;
    int total_msec = 0;     //  Total expected cost in msecs
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        int workload;
        //  Random workload from 1 to 100msecs
        workload = within (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);
    sleep (1);              //  Give 0MQ time to deliver
```

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

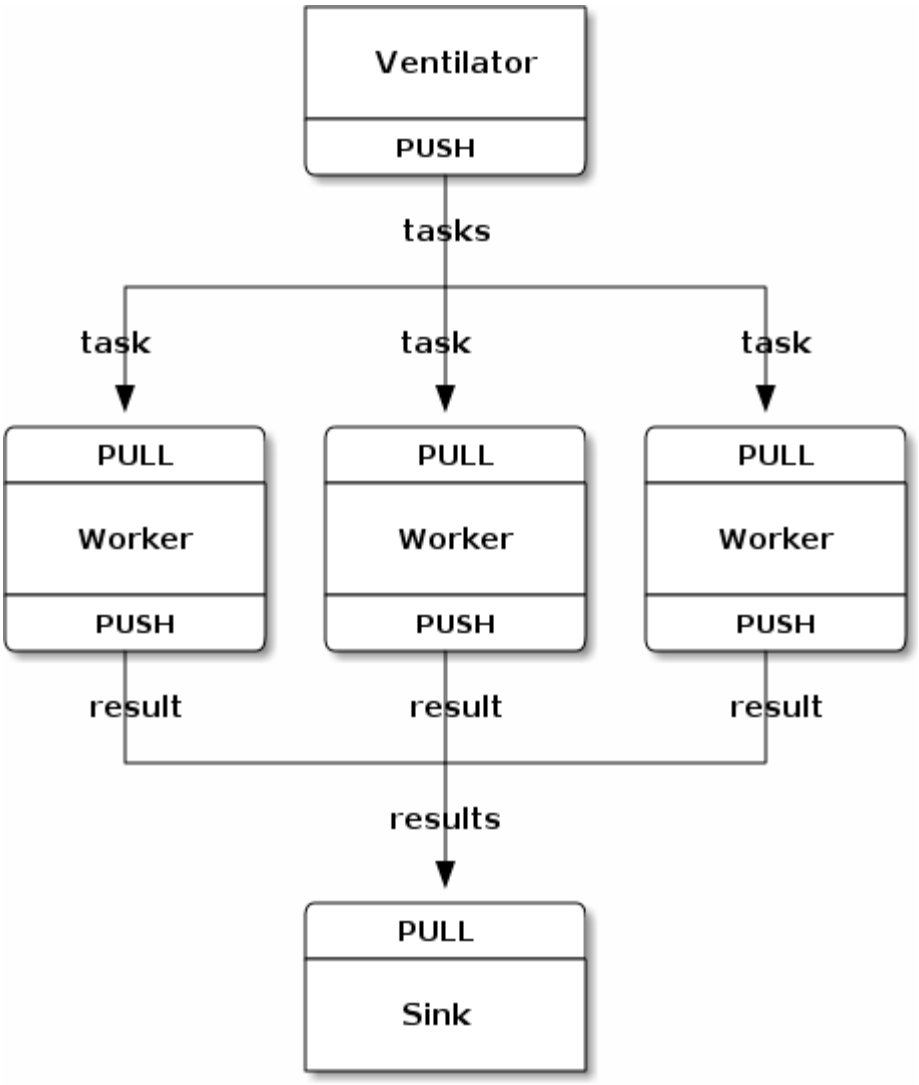

Figure 5 – Parallel Pipeline

这是工作进程应用程序，它接收一个消息，休眠（sleep）几秒后发送它已完成了的信号。

```
//
//  Task worker
//  Connects PULL socket to tcp://localhost:5557
//  Collects workloads from ventilator via that socket
//  Connects PUSH socket to tcp://localhost:5558
//  Sends results to sink via that socket
//
#include "zhelpers.h"
```

```
int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

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

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

    //  Process tasks forever
    while (1) {
        char *string = s_recv (receiver);
        struct timespec t;
        t.tv_sec = 0;
        t.tv_nsec = atoi (string) * 1000000;
        //  Simple progress indicator for the viewer
        fflush (stdout);
        printf ("%s.", string);
        free (string);

        //  Do the work
        nanosleep (&t, NULL);

        //  Send results to sink
        s_send (sender, "");
    }
    zmq_close (receiver);
    zmq_close (sender);
    zmq_term (context);
    return 0;
}
```

这是监听器的应用程序。它收集这 100 个任务，计算整体处理花费的时间，这样我们就能确定，当不止一个工作进程的时候，它们能够并行运行。

```
//
//  Task sink
//  Binds PULL socket to tcp://localhost:5558
//  Collects results from workers via that socket
//
#include "zhelpers.h"
```

```
int main (int argc, char *argv[])
{
    //  Prepare our context and socket
    void *context = zmq_init (1);
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

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

    //  Start our clock now
    struct timeval tstart;
    gettimeofday (&tstart, NULL);

    //  Process 100 confirmations
    int task_nbr;
    int total_msec = 0;     //  Total calculated cost in msecs
    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);
    }
    //  Calculate and report duration of batch
    struct timeval tend, tdiff;
    gettimeofday (&tend, NULL);

    if (tend.tv_usec < tstart.tv_usec) {
        tdiff.tv_sec = tend.tv_sec - tstart.tv_sec - 1;
        tdiff.tv_usec = 1000000 + tend.tv_usec -
tstart.tv_usec;
    }
    else {
        tdiff.tv_sec = tend.tv_sec - tstart.tv_sec;
        tdiff.tv_usec = tend.tv_usec - tstart.tv_usec;
    }
    total_msec = tdiff.tv_sec * 1000 + tdiff.tv_usec / 1000;
    printf ("Total elapsed time: %d msec\n", total_msec);
```

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

处理一批消息的平均时间是 5 秒。当我们启动 1，2，4 进程，我们从监听器上得到以下的结果：

```
#   1 worker
Total elapsed time: 5034 msec
#   2 workers
Total elapsed time: 2421 msec
#   4 workers
Total elapsed time: 1018 msec
```

让我们仔细看看这个代码的一些内容：

- 工作进程把向上的消息流连接到风扇（ventilator），向下的消息流连接到监听器。这就意味着你可以任意地增加工作进程。如果工作进程绑定到终端，每次你增加一个工作进程时你就需要（a）更多的终端（b）来修正风扇（ventilator）和/或监听器。我们说，在我们的结构中风扇（ventilator）和监听器是“固定”的部分，而工作进程是“变动”的部分。

- 我们必须启动并运行所有工作进程，来同步发送这批消息的开始（Start）。这在 0MQ 中是很平常的事情，并且没有更简单的方法。这个“连接”方法需要一定的时间。因此，当一组工作进吃呢个连接到风扇（ventilator）时，第一个成功连接的工作进程将会获得完整的消息，而别的工作进程还正在连接中。如果你因为某种原因没能同步这批消息的开始，系统不会完全并行运行。试试去掉等待时间，然后看结果如何。

- 风扇（ventilator）的 PUSH 套接字均等地分发消息给工作进程（假设它们在消息发出之前都已连接上）。这叫做负荷均衡，我们还会看到它的更详细的介绍。

- 监听器的 PULL 套接字从工作进程均等地收集结果。这叫做公平排队。

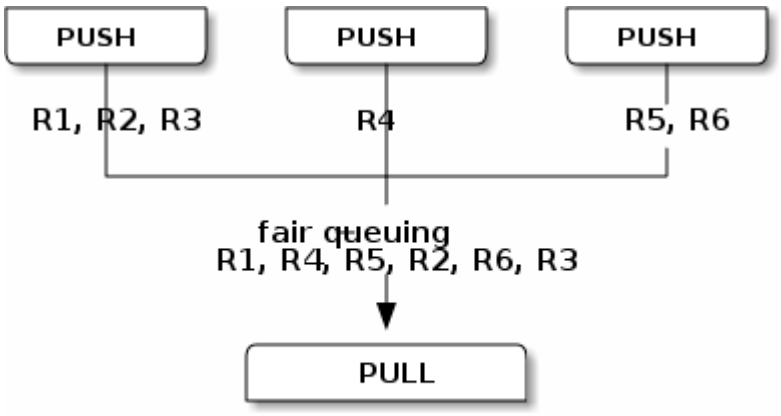

Figure 6 – Fair queuing

用 0MQ 编程

已经看到一些例子，你一定急于在应用程序中使用 0MQ。开始之前，深吸一口气，均匀呼出，并采取一些基本的办法来减轻你的压力和困惑。

- 一步一步的学习 0MQ。它只是简单的套接字，但是它却藏着无限的可能。慢慢的学习并掌握每一在种可能。

- 编写美好(nice)的代码。丑陋（ugly）的代码隐藏着问题，也让别人很难帮助你。你可能使用过没有意思的变量名，但是别人读不懂你的代码。使用有意思的单词作为变量名，而不要说“我太粗心了，所以没有告诉你这个变量是什么意思”。使用一致的缩排，整齐的界面。编写美好的代码，你的世界也会变得很舒服。

- 当你再做的时候，测试你做出了什么。当你的程序不运行，你应该知道问题在哪。使用 0MQ 的时候，第一次尝试的时候什么不运行，这点很重要。

- 当你发现代码如预料的一样不运行，给代码设置断点，测试每一部分代码，看看是那部分不工作。0MQ 允许用户编写模块化得代码。用它来发挥你的优势。

- 当你需要的时候，可以进行抽象化（类，方法，什么都可以）。如果你复制/粘贴很多代码，你也会复制/粘贴错误。

为了说明这一点，这是有人叫我帮着抽象的代码框架：

```
//  NOTE: do NOT reuse this example code!
static char *topic_str = "msg.x|";

void* pub_worker(void* arg){
   void *ctx = arg;
   assert(ctx);

   void *qskt = zmq_socket(ctx, ZMQ_REP);
   assert(qskt);

   int rc = zmq_connect(qskt, "inproc://querys");
   assert(rc == 0);

   void *pubskt = zmq_socket(ctx, ZMQ_PUB);
   assert(pubskt);

   rc = zmq_bind(pubskt, "inproc://publish");
   assert(rc == 0);

   uint8_t cmd;
```

```
    uint32_t nb;
    zmq_msg_t topic_msg, cmd_msg, nb_msg, resp_msg;

    zmq_msg_init_data(&topic_msg, topic_str,
strlen(topic_str) , NULL, NULL);

    fprintf(stdout,"WORKER: ready to recieve messages\n");
    //  NOTE: do NOT reuse this example code, It's broken.
    //  e.g. topic_msg will be invalid the second time through
    while (1){
    zmq_send(pubskt, &topic_msg, ZMQ_SNDMORE);

    zmq_msg_init(&cmd_msg);
    zmq_recv(qskt, &cmd_msg, 0);
    memcpy(&cmd, zmq_msg_data(&cmd_msg), sizeof(uint8_t));
    zmq_send(pubskt, &cmd_msg, ZMQ_SNDMORE);
    zmq_msg_close(&cmd_msg);

    fprintf(stdout, "recieved cmd %u\n", cmd);

    zmq_msg_init(&nb_msg);
    zmq_recv(qskt, &nb_msg, 0);
    memcpy(&nb, zmq_msg_data(&nb_msg), sizeof(uint32_t));
    zmq_send(pubskt, &nb_msg, 0);
    zmq_msg_close(&nb_msg);

    fprintf(stdout, "recieved nb %u\n", nb);

    zmq_msg_init_size(&resp_msg, sizeof(uint8_t));
    memset(zmq_msg_data(&resp_msg), 0, sizeof(uint8_t));
    zmq_send(qskt, &resp_msg, 0);
    zmq_msg_close(&resp_msg);

    }

    return NULL;
}
```

这是我改写的，为了查找问题：

```
static void *
worker_thread (void *arg) {
    void *context = arg;
    void *worker = zmq_socket (context, ZMQ_REP);
    assert (worker);
```

```
    int rc;
    rc = zmq_connect (worker, "ipc://worker");
    assert (rc == 0);

    void *broadcast = zmq_socket (context, ZMQ_PUB);
    assert (broadcast);
    rc = zmq_bind (broadcast, "ipc://publish");
    assert (rc == 0);

    while (1) {
        char *part1 = s_recv (worker);
        char *part2 = s_recv (worker);
        printf ("Worker got [%s][%s]\n", part1, part2);
        s_sendmore (broadcast, "msg");
        s_sendmore (broadcast, part1);
        s_send    (broadcast, part2);
        free (part1);
        free (part2);

        s_send (worker, "OK");
    }
    return NULL;
}
```

最后，该应用程序通过了先吃呢个间的套接字，它在 0MQ/2.0x 中奇怪的崩溃掉了，但是却固定(fix）在 0MQ/2.1x 中。这不是真正的代码错误，只是错误地使用了程序接口。

获得正确的背景（context）

0MQ 应用程序总是从创建一个背景(context)开始,然后利用它来创建套接字。用 C 语言是调用命令 zmq_init(3)。在你的进程中，你只能创建并使用一个背景。严格地说背景是一个单一进程中所有套接字的容器。它是进程内的套接字传输，这是连接一个进程中的线程的最快的方法。如果在运行中，进程有两个背景，它们像分开的 0MQ 例子（instance）。如果这就是你想要的，行，否则请记住：在你的主程序代码开始调用一次 zmq_init(3)，在主程序代码结束调用一次 zmq_term(3)。

如果你在使用 fork()系统调用，每个进程需要它自己的背景。如果在主进程中你先调用 zmq_init(3)再调用 fork(),子进程就会获得它们自己的背景。一般来说，如果你像做些子进程的有趣的东西，你只需要在父进程中控制就行了。

干净地退出

出色的程序员看到好的事物时都会给予箴言：总是会为你扫尾。但你通过 Python 这样的语言使用 0MQ 的时候，空间（stuff）会自动释放。但是当你使用 C 语言时，如果你

使用完一个对象，你必须小心地释放它，否则，你就会让内存泄露，是应用程序不稳定，产生坏的结果。

我们需要关心的是 0MQ 对象的消息，套接字和背景。幸运的是，它很简单。

- 它总会调用函数 zmq_msg_close(3)，关闭你正在使用的消息；
- 如果过你正在打开并关闭很多套接字，那可能意味着你将从新设计你的应用程序；
- 当你退出程序的时候，关闭你的套接字，然后调用函数 zmq_term(3).这样破坏了背景。

在 0MQ 的版本 2.0.x 及其以前的版本中，调用函数 zmq_close(3)，或者 zmq_term(3),或者退出主程序太频繁都会导致你认为已经发送的消息丢失。它是 0MQ 中的一个缺陷，但是已经在 2.1 及以后的版本中调整过来了。

在版本 2.1.x 中，如果你调用函数 zmq_term(3)时已经打开了套接字，则会导致 0MQ 永远挂起。它导致了一些争论，但是这个缺点有希望得到修正。

因此，当还有数据在发送的时候，所有的例子退出都做了这件事：

```
sleep (1);
```

这是一个快速而令人不喜欢（dirty）的解决问题的方法，但是对于 0MQ 版本 2.0.x 再没有别的解决方法了。

我们为什么需要 0MQ

你已经看到了运行的 0MQ，让我们回到为什么使用 0MQ 这个问题。

如今，很多应用程序的组成部分会跨越某些网络，或者是局域网，或者是互连网。很多开发人员最终都开始做消息方面的工作。一些开发人员使用消息队列产品，但很多时候他们利用 TCP 或者 UDP 做自己的产品。这些协议使用起来不难，但是以任何一种可靠的方式发送消息与从 A 发送几个字节到 B 有很大的不同。

让我们来看看当我们利用原始 TCP 套接字连接各部分（pieces）时所遇到得典型的问题。任何可以重复利用的消息层都会解决所有或者以下的所有或者绝大部分问题：

- 我们怎样处理输入/输出？让我们的应用程序阻塞，或者是我们在后台处理输入/输出？这是设计决策的关键。阻塞式输入/输出创建的构架不容易升级扩展。后台运行输入/输出很难保证正确。

- 我们怎样处理动态（dynamic）部分,即会暂时撤去的部分？我们要正式划分“客户“和”服务器“，并规定服务器不能撤离吗？如果我们想把服务器连接到服务器会怎样？我们应该每隔几秒就尝试着重新连接吗？

- 我们在线路上怎样表示消息？我们应该怎样组帧数据才能够方便读写，避免缓冲区溢出，使小的消息能够高效能够适合大的影片带帽子的跳舞猫的发送。

- 我们把消息队列存储在哪？如果部件读取消息队列很慢，导致我们的队列不断增大怎么办？我们应该采取什么策略呢？

- 我们怎样处理丢掉的消息？我们应该等待新的数据，或者请求重新发送，或者我们建立比较可靠的一层以确保消息不会丢失？如果这个层本身崩溃掉怎么办？

- 如果我们需要不同的网络来传输怎么办？用广播代替 TCP 单播？或者使用 Ipv6?我们需要重新编写应用程序吗？或者对传输协议进行抽象？

- 我们怎样对消息进行路由？我们能够把相同的消息发送到多个接收点（peer）吗?我们能够把应答发挥到请求着吗？

- 我们怎样用另一种语言编次应用程序接口函数？我们重新实现线路协议还是打包成一个库？如果是前者，我们怎样能够保证高效和稳定的栈？如果是后者，我们怎样能够保证通用性？

- 我们怎样表示数据才能够在不同的结构间读取？我们要执行特定编码的数据类型吗？什么时候这才会是消息系统的任务，而不是高层的？

- 我们怎样处理网络错误？我们是等待，再试，静静地忽略它们，或者中止呢？

看一个经典的开源项目 Hadoop Zookeeper 在目录 src/c/src/zookeeper.c.下读 C 语言编写的它的应用程序接口代码。他是 3200 行的神秘的(mystery)，无正式文件的（unducmented）,客户-服务器网络通信协议。我发现它很高效，因为它使用的是函数 poll()，而不是 select()。但是，事实上管理员应该使用一个通用的消息传递层和和明确记载的线路协议。对于一个组（team）来说，一遍又一遍的构造这样样的系统式很不值的。

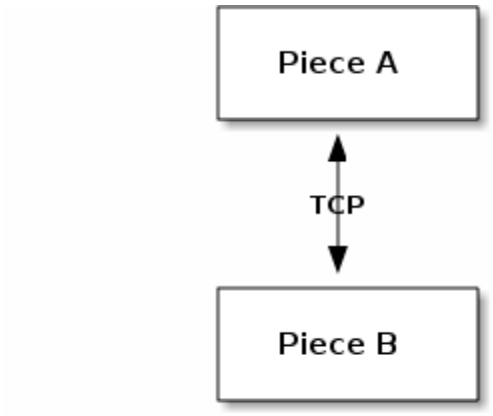

Figure 7 – Messaging as it starts

但是怎样构造一个能够重复使用的消息层？为什么这么多的项目都需要这项技术，而人们还是每次都是通过在代码中使用 TCP 套接字，这样困难的方法来解决那一长串列表中

的问题。

原来是因为建立可重复利用的消息系统真的很困难，这就是为什么没有什么自由的开源软件尝试过，并且这也是商用的消息产品复杂，昂贵，不灵活和脆弱的原因。在 2006 年，iMatix 公司设计了高级消息队列协议（AMQP），它给与了自由开源软件的开发人员第一种可以重负利用消息系统的方法。高级消息队列（AMQP）比别的设计（design）都运作的要好，但是还是复杂，昂贵，脆弱。学会使用它需要几周的时间，而创建当事情很糟糕时也不会崩溃的稳定的结构需要几个月的时间。

很多消息项目，如 AMQP，尝试着以一种可以重复利用的方式解决这一系列的问题，通过发明一个新的概念,“中间代理”(broker）,能够寻址，路由和排队。这就导致了一个客户-服务器模式的协议，或者封装在一些非正式协议之上的一系列应用程序接口函数，是应用程序能够与中间代理能够对话。

中间代理能够聪明的降低大型网络的复杂性。但是把基于中间代理的消息加到像 Zookeeper 这样的产品上会让它变的更糟，没有好处。它意味着增加一个额外的大的箱子（box）和一个新的单点故障。一个中间代理迅速变成瓶颈和一个新的管理危机。如果软件支持的话，我们可以增加第二个，第三个，第四个中间代理，并制定故障转移方案。人们这样做了。它创建了更多的移动模块，变得更复杂，更多的事情需要突破（break）。

一个中央中间代理的建立需要它自己的团队。你需要白天和晚上不停的监视中间代理，当它们运行错误的时候你就用一个棍敲打它们。你需要机子（boxes），并且你需要备份机子，你需要人们来管理这些机子。只有有很多移动模块的应用程序才值得这样做，有几组人员用几年的时间来完成。

因此，小的到中等大小的应用程序的陷入困境了。要么他们避免网络编程，编写大规模的不能升级的程序。或者他们进行网络编程，编写脆弱的、复杂的、难以维护的应用程序。或者他们致力于消息产品的开发，并最终开发出取决于昂贵的，容易的中间代理技术的可升级应用程序。这里没有真正好的选择，可能这就是为什么在上个世纪消息停止发展并激起巨大情绪的原因。

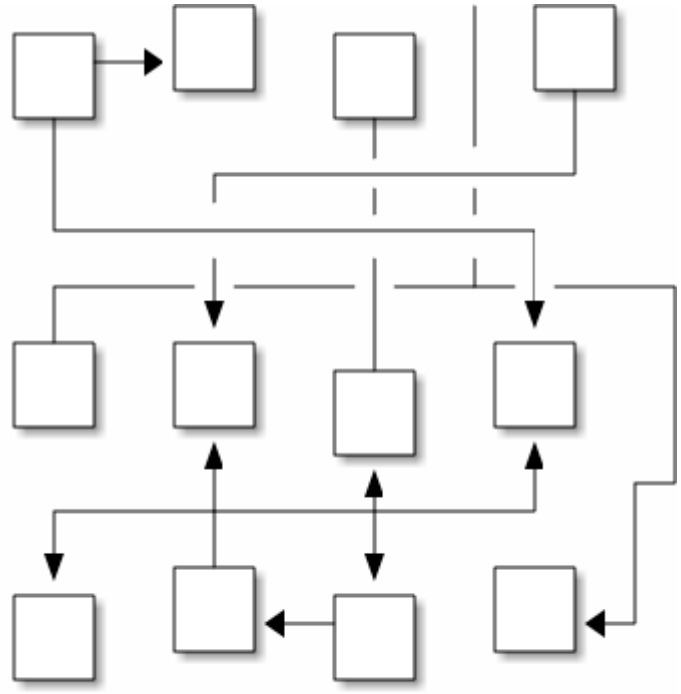

Figure 8 – Messaging as it becomes

我们需要的是做消息的工作但是很简单又便宜的方法，并且能够在任何应用程序中使用，接近零成本。它应该是一个你只需要链接的库，而没有别的依赖性。没有额外的移动模块，因此没有额外的危机。它应该能够运行在任何操作系统上，任何程序语言都能工作。

这就是 0MQ：一种高效地，可嵌入的库，它解决了很多问题，让应用程序在网络上有很好的可伸缩性，而不需要多大成本。

特别地：

- 它在后台线程处理输入/输出程序。这些应用程序进程线程之间的通信使用的是没有锁的数据结构，因此，0MQ 的应用程序不需要锁，信号量，或者别的等待状态。

- 可以自由加入和撤出部件，0MQ 可以动态地再连接。这就意味着你可以以任意的顺序开始部件。你可以创建：面向服务的结构“（SOAS），服务器可以随时加入或者离开网络。

- 当需要的时候消息会自动排队。在消息排队前，它聪明地把消息放得离接收者尽可能的接近。

- 它有办法处理过满队列（叫做“高水位标记“）。当一个队列满了以后，0MQ 自动地阻塞发送者，或者把消息扔掉，这取决于你处理的消息的类型（就是所谓的”模式“）。

- 它允许你的应用程序之间以任意的传输协议交谈：TCP,广播，进程间，或者进程内。你使用不同的传输协议的时候不需要改变代码。

- 它利用取决于消息模型的不同的策略安全地处理慢的/阻塞的读取方式（reader）；
- 它允许你用不同的模型路由消息，如请求-应答，和发布-订阅。这些模型就是你怎样创建的拓扑和网络结构。
- 当发送的时候，它就在线路上利用简单的帧把所有消息分发出去了。如果你写了 10K 的消息，你就会收到 10K 的消息。
- 它不给消息加任何格式。它们是一个整体，大小可从零到千兆字节。如果你想描述数据，你可以选择第三方产品，例如谷歌的协议栈，XDR，和别的。
- 它很聪明地处理网络错误。有时它会再试，有时它告诉你一个操作失败。
- 它减少你的碳（carbon）空间。做的多但是使用的处理器很少，意味着你的机子使用的功耗更低，这样你的老机子可以使用更长时间。人工智能的 Gore 公司应该会喜欢 0MQ。

事实上 0MQ 做的比这多，它对你如何开发应用程序的网络功能有颠覆性的影响。浅显地说它只是应用程序接口套接字，你可以在其中调用函数 zmq_recv(3)和 zmq_recv(3)。但是，消息处理迅速成为中央循环，并且你的应用程序迅速分解成一系列消息处理任务。它很简洁，很自然。它是可伸缩的：每个任务连接到一个节点，并且节点之间能够通过任意的传输协议通信。一个进程中的两个节点（一个节点就是一个线程），一台机子中的两个节点（一个节点就是一个进程），一个网络中的两个机子（一个机子就是一个节点）。都不会改变应用程序代码。

套接字的可伸缩性

让我们看看 0MQ 活动中的可伸缩性。这是一个启动气候服务器脚本框架的应用程序，然后是一系列平行的服务器程序。

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

当服务器运行，我们使用“顶“将在活动进程中看到类似这样的东西（在一个四核的机子上）：

```
  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
```

让我们想一下这里究竟发生了什么。气候服务器有一个单一的套接字，我们在这让它发送数据到五个并行的客户机。我们有数千个并行的客户机。服务器应用程序看不到它们，不会直接与它们对话。

问题求解

当你开始用 0MQ 编程以后，你将不止一次地遇到这个问题：你丢掉了你认为会得到的消息。有一个基本的解决这个问题的办法，就是检查一遍导致这个问题的最常见的原因。不要担心对一些术语不熟悉，在下一章就会清楚的。

如果你在失败的代价很昂贵的背景中使用 0MQ，那么你需要做恰当的计划。首先，建立允许你学习并测试你的设计不同部分的原型。不断给它们加压，直到他们崩溃掉，这样你就知道你的设计有多强壮了。第二，投入到（investment）你的测试中。意思是建立测试框架，确保你能够访问有足够电脑电源的真实的设置，并花时间来认真的测试。事实上，第一组编写代码，第二组尝试破坏它。最后，让你的组织联系 iMax 公司来讨论我们怎样确保程序顺利运行，并且如果它被破坏了的话，如何修整。

简单地说，如果你还不能保证某个结构能够在现实条件下工作，那么它很容易在最坏情况下崩溃。

警告-不稳定的范例！

传统的网络编程是建立在一个一般的假设之上：一个套接字只与一个连接，一个端点对话。有广播协议，但是它们是特别的。当我们假设“一个套接字=一个连接“，我们以某种方法规范我们的结构。我们创建逻辑线程，每个线程与一个套接字，一个端点联系。我们使线程聪明且稳定。

在 0MQ 的世界里，套接字是多线程的应用进程，它自动地处理所有连接。打开，关闭，或者附加到这些连接上的状态你都看不到。当你使用阻塞发送或者接收，或者监听（poll），你只能与套接字交谈，而不是与它处理的连接交谈。连接时私有的，并且是看不见的，这就是 0MQ 稳定性的关键所在。

因为你的代码与一个套接字对话，可以处理任意数量的跨周围任意网络的连接，而不用做任何改变。0MQ 中的消息模式的扩展（scale）比应用程序代码中的消息模式扩展更便宜。

因此，这个一般的假设不再适用。当你读代码的例子的时候，你的大脑试图把它链接到你知道的一些东西里面。你会读到“套接字”，你就会想到“哦，它表示与另一个节点的连接”。那是错误地。你会读到“线程”你又会想“哦，一个线程表示了与另一个节点的连接”，然后，你就困惑了。

如果你第一次读这篇指导，记住，知道你真的写 0MQ 的代码一天或者两天（可能三天或者四天），你会感到困惑，特别是困惑于 0MQ 是如何把事情处理的这么简单的。可能你会试图把那个假设用于 0MQ 之上，但是还是行不通。然后你就会进入你自己的启蒙和信赖时刻，当这一切变得越来越明显，人们开始顿悟了。

第二章-中等的东西

在第一章我们把 0MQ 作为驱动，编写了一些 0MQ 主要套接字的基础的例子：请求-应答，发布-订阅，和管道模式。在这一章中，我们将会认真研究在实际的项目中如何使用这些工具。

我们将涉及到以下问题：

- 如何创建并利用 0MQ 套接字工作
- 如何通过套接字发送和接受消息
- 如何围绕 0MQ 的异步输入/输出模式建立你的应用程序
- 如何在一个线程中处理多个套接字
- 如何恰当处理关键的和非关键的错误
- 如何彻底地关闭一个 0MQ 应用程序
- 如何接收分段的消息
- 如何跨网络转交消息
- 如何建立一个简单的消息队列中间件代理
- 如何用 0MQ 写多线程的应用程序

- 如何利用 0MQ 在线程间发送信号
- 如何利用 0MQ 协调一个节点网络
- 如何创建利用套接字等式（identities）创建持久的套接字
- 如何创建和使用发布-订阅模型的消息封包（envelopes）
- 如何构建持久的可从崩溃会恢复的订阅方
- 使用高水位标记（high-water mark）（HWM）来防止内存溢出

零的含义

在 ØMQ 中 Ø 是权衡的意思。一方面，这个奇怪的名字，降低了 ØMQ 在 Google 和 Twitter 网站中的可见率。另一方面，它惹恼了一些丹麦人，他们写“ØMG røtfl"和“*Ø 不是一个看起来让人愉快的词*”和"*Rødgrød med Fløde!*"这样的信给我们，这显然是一种侮辱，这的意思是“可能你的邻居是格伦德尔的直接后裔！”看起来这是公平交易。

本来，零在 ØMQ 中的意思是“零中间代理”和（尽量做到）“零延迟”。其间，它又另外的目的：零管理，零成本，零浪费。更一般地，“零”指简约思想贯穿整个项目。它的本领体现在降低复杂性上，而不是增加新的功能。

应用程序接口套接字

很坦诚地说，ØMQ 起到带领你转变的作用。我们不用为这件事道歉，因为它对你有利，并且它对我们的影响比对你们大。它很像熟悉的应用程序接口 BSD 套接字，但是它隐藏了很多消息处理机制，而这些处理机制会慢慢地改变你如何设计和编写分布式软件的观念。

在网络编程中，套接字是事实上的标准应用程序接口，对于 stopping your eyes from falling onto your cheeks 也很有用。让 ØMQ 很合开发人员胃口的是它使用标准的 API 套机字。它把“面向消息的中间件”，这个让所有人都紧张的词组转变成了“特别有趣的套接字”，这就让我们很期待它，并且希望知道的更多。

正如意大利披萨一样令人满意，ØMQ 套接字很容易掌握。ØMQ 套接字一个生命周期由四部分组成，和 BSD 套接字一样：

- 创建和撤销套接字，它们一起形成了套接字生命期的因果循环（查看 zmq_socket(3),zmq_close(3)）。
- 设置套接字，给它们设置选项，并检查必要性（查看 zmq_setsockopt(3),zmq_getsockopt(3)）。

- 通过创建 ØMQ 与网络的连接，把 ØMQ 套接字接入网络中。
- 通过在套接字上写和接收消息来传递数据（查看 zmq_send(3), zmq_recv(3)）。

让我们看看：

```
void *mousetrap;

//  Create socket for catching mice
mousetrap = zmq_socket (context, ZMQ_PULL);

//  Configure the socket
int64_t jawsize = 10000;
zmq_setsockopt (mousetrap, ZMQ_HWM, &jawsize, sizeof
jawsize);

//  Plug socket into mouse hole
zmq_connect (mousetrap, "tcp://192.168.55.221:5001");

//  Wait for juicy mouse to arrive
zmq_msg_t mouse;
zmq_msg_init (&mouse);
zmq_recv (mousetrap, &mouse, 0);
//  Destroy the mouse
zmq_msg_close (&mouse);

//  Destroy the socket
zmq_close (mousetrap);
```

套接字是无返回值类型的指针，而消息（我们马上就会遇到）是一种数据结构。因此，在C语言中直接传递套接字,但是在所有函数中都是传递消息的地址,如函数zmq_send(3)和 mq_recv(3)。请注意，“在 ØMQ 中，套接字是属于我们的”但是消息只有在我们的代码中才是我们的。

对于任何对象，你都可以创建，撤销和设置如你所愿工作的套接字。但是，请记住，ØMQ 是异步的，可伸缩的结构。这会影响到我们如何把套接字接入到网络拓扑中，以及之后我们如何使用套接字的问题。

把套接字接入到网络拓扑中

为了在两个节点之间创建连接，你可以在一个节点使用函数 zmq_bind(3)，在另一个节点使用函数 zmq_connect(3)。按照通常的规则，使用 zmq_bind(3)的节点是一个“服务器”，它有一个公认的地址，而使用 zmq_connect(3)的节点是一个“客户机”，它的地址是不知道的，或者是任意的。因此我们说我们“绑定一个套接字到一个终端”和

“连接一个套接字到一个终端”，终端是指公认的网络地址。

ØMQ 的连接与守旧的 TCP 的连接有些不同。主要的不同是：

- 它们跨任意的传输协议（inproc;ipc;tcp; or epgm）。查看 zmq_inproc(7), zmq_ipc(7),zmq_tcp(7), zmq_pgm(7)，和 zmq_epgm(7)
- 当一个客户机调用 zmq_connect(3)连接到一个终端时它便开始存在，而不管服务器是否已经调用 zmq_bind(3)绑定到一个终端。
- 它们是异步的，并且当我们需要的时候队列就会神奇地存在。
- 根据在每各个端点使用的套接字类型，我们描述一种确定的“消息模型”。
- 一个套接字可能会有很多输入输出连接。
- 没有函数 zmq_accept()，当一个套接字绑定到一个终端的时候，它自动地开始接收连接。
- 应用程序不能直接与这些连接交流，它们是被封装在套接字之下的。

很多结构都采用的是客户-服务器模式，其中服务器是固定的部分，客户机多是动态的部分，即，它们经常会加入或者离开。有一些寻址方面的问题：服务器对客户机是可见的，但是客户机对服务器却未必可见。因此，通常哪个节点应该调用 zmq_bind(3)（服务器）和哪个节点应该调用 zmq_connect(3)（客户机）都是可见的。它也取决于你使用的套接字的类型，不寻常的网络结构例外。后面我们将会看看套接字类型。

现在假设我们先启动客户机再启动服务器。在传统的网络中我们会得到一个大的红色的失败标志。但是 ØMQ 允许我们任意的启动和停止各部件。一旦客户机调用函数 zmq_connect(3)，连接就建立起来了，节点能够开始写消息到套接字中。在某个时候（希望是在因排队消息太多，消息被丢弃之前）服务器开始工作，调用 zmq_bind(3) 和 ØMQ 分发消息。

一个服务器节点可以绑定到很多终端，而只需要一个单一的套接字就能够做到。这就意味着它会与不同的传输协议接收连接：

```
zmq_bind (socket, "tcp://*:5555");
zmq_bind (socket, "tcp://*:9999");
zmq_bind (socket, "ipc://myserver.ipc");
```

你不能两次绑定到相同的终端，这样会导致异常。

每次，一个客户机节点调用 zmq_connect(3)连接到任意的终端，服务器套接字建立另一个连接。没有固定一个套接字能够建立的连接数。客户机节点也能够通过一个单一的

套接字连接到任意数量的终端。

在多数情况下，哪个节点作为客户机，哪个节点作为服务器是一个网络拓扑的问题，而不是消息流的问题。然而，有些情况下（当连接撤销的时候重新发送）相同的套接字类型在客户机或者服务器中的行为不一样。

这就意味着你总是认为“服务器“是拓扑中的固定部分，有相对固定的终端地址，而“客户机”作为可以动态加入和撤出的部分。然后以这种模式设计你的应用程序。它运行起来会更好。

套接字有它的类型。套接字的类型定义了套接字的意义，它规定了向内或向外路由消息，排队等的方式。你可以把特定类型的套接字连接在一起，例如，一个出版类型的套接字和一个订阅类型的套接字。套接字以“消息模式“的形式一起工作。随后我们可以看到它的更详细的介绍。

以不同的方式连接套接字的能力让 ØMQ 有了作为消息队列的基础。在这上面有一层封装，例如设备（device）和主题路由，我们随后会讨论到。ØMQ 通过把各部分连接在一起的方式来定义你的网络结构，像一个孩子连接它的玩具模型一样。

利用套接字携带数据

你通过调用 zmq_send(3)和 zmq_recv(3)的方法来发送或者接收数据。名字没有改变，但是 ØMQ 的输入/输出模式和 TCP 的输入/输出模式很不一样，你需要花时间来分清楚。

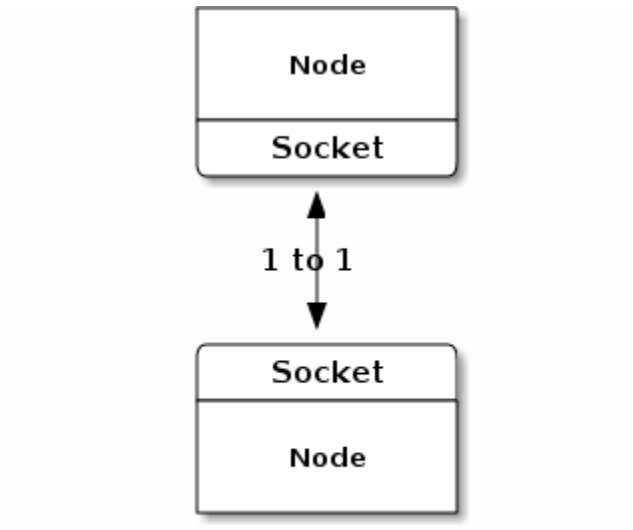

Figure 10 – TCP sockets are 1 to 1

我们来看看 TCP 套接字与 ØMQ 套接字在携带数据的时候主要的区别：

- ØMQ 携带的是消息，而不是字节（当使用 TCP 时）或者帧（当使用 UDP 时）。消息

是特定长度的二进制数据块。我们很快就会涉及到（come to）消息了，它是为了优化性能而设计的，因此，理解起来有点困难。

- ØMQ 套接字在后台线程中处理输入/输出。意思就是，消息到达本地的输入队列，并由本地的输出队列输出，而不用在乎你的应用程序正在忙什么。顺便说一下，这些都是可以设置的内存队列。

- ØMQ 套接字依据不同的套接字类型被连接到（或连接自，它是一样的）很多别的套接字。TCP 模仿一对一的电话呼叫模式，ØMQ 使用了一对多（像一个无线电广播），多对多（像邮局），多对一（像邮箱）的模式，也可以使用一对一的模式。

- ØMQ 套接字可以发送消息到很多端点（创建一个扇出模式），也可以从很多端点接收消息（创建一个扇入模式）。

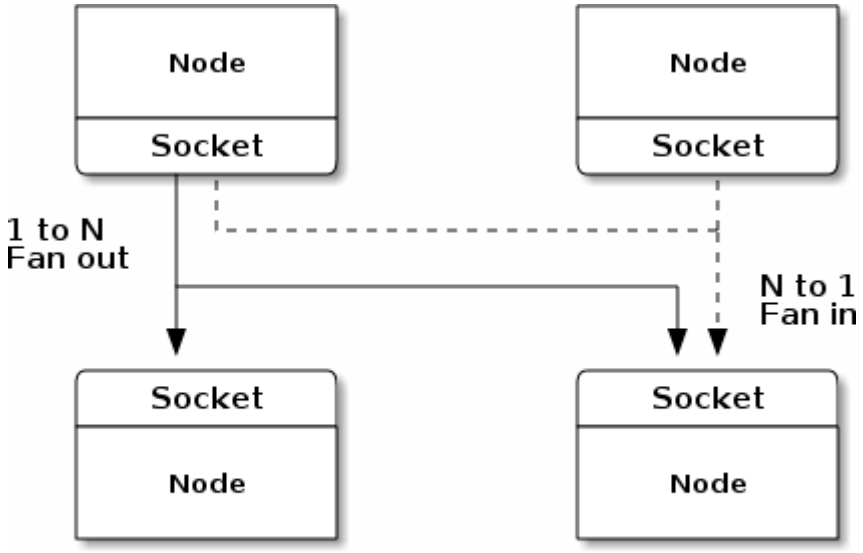

Figure 11 – ØMQ sockets are N to N

因此，写到一个套接字的消息可能被一次性的发送到不同的地方，反过来，一个套接字可以接受所有发送给它的消息。zmq_recv(3)方法使用的是公平排队算法，这样每一个发送者获得一个均等的机会。

zmq_send(3)并不真的把消息发送到与它相连的套接字。它给消息排队，这样输入/输出线程就能够异步发送它们。通常情况下消息不会阻塞。因此你的应用程序调用 zmq_send(3)时一定会发送消息。如果你使用 zmq_msg_init_data(3)创建一个消息，你将不能够再使用这个数据或者释放它，否则输入/输出线程很快就会发现它写重复了或者 unallocated garbage.对于初学者来说，这是一个常见的错误。一会我们将会看到如何处理消息。

单播传输

ØMQ 提供了很多单播传输协议(inproc, ipc, and tcp)和多播传输协议(epgm, pgm). 多播是一种高级的技术，我们一会将要讨论。如果你不知道怎样的扇出比例会让一到多的单播出错，请不要使用多播。

对于最常见的情况使用的是 tcp,它是一个非连接的 TCP 传输协议（看来此 TCP 非彼 TCP）。它是可伸缩的，轻便的，在很多时候都足够快。我们把它称作“非连接”是因为 ØMQ 的 tcp 传输不要求端点在你连接之前就已经存在。客户机和服务器可以在任何时候连接、绑定，撤销、加入，它对应用程序来说是透明的。进程间传输协议 ipc 和 tcp 协议很像，只是它是从局域网抽象得来的。因此你不需要修改 IP 地址和域名。对于某些目的，在这本书的例子里面我们会经常用到它。ØMQ 的 ipc 协议和 tcp 协议一样，也是非连接的。它存在一个局限，即它不能在窗口上工作。这可能会在 ØMQ 将来的版本中得到改善。按照惯例，我们给终端加上".ipc"的扩展名，以免和别的文件名发生冲突。在 UNIX 系统中，如果你使用 ipc 终端，你需要按照恰当的权限来创建它们，否者在用户 ID 不同的情况下进程间可能不能够分享它们。

线程间传输协议，inproc,是一个连接的信号传输协议。它比 tcp 和 ipc 要快的多。相对于 ipc 和 tcp 协议来说，inproc 协议有一个专门的限制：你在连接之前需要绑定。这点在 ØMQ 以后的版本中可能会得到改善，但是现在它还是对你使用 inproc 套接字会产生影响。当我们把它用在例子中时，在连接到它之前我们小心地绑定每一个终端。如果我们先连接，然后再绑定，那么连接就会返回一个错误，如果我们忽略这个错误，接收就会发生阻塞。

ØMQ 不是一个中立的传输者

使用 ØMQ 的新人常常问的一个问题（我问过自己）是，“我用 ØMQ 如何编写 XYZ 服务器？”例如，“我用 ØMQ 如何编写 HTTP 服务器？”

可以这样说，如果我们使用标准的套接字来携带 HTTP 的请求和响应，利用 ØMQ 套接字也能做到，并且更快更好。

很遗憾，答案是“它不是这样工作的”。 ØMQ 不是一个中立的传输者，它把组帧加到它使用的传输协议上。这个组帧与现存的协议是不兼容的，所以我们想使用我们自己的帧。例如，这是一个 HTTP 请求和一个 ØMQ 请求，它们都是用 TCP/IP 协议的。

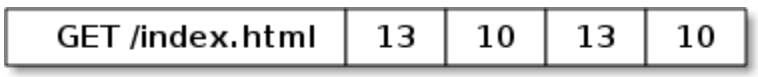

Figure 12 – HTTP request

HTTP 使用 CR-LF 作为它最简单的组帧分界符，而 ØMQ 是用定长的帧结构。

Figure 13 – ØMQ request

因此你可以利用 ØMQ 编写一个像 HTTP 协议的协议，使用请求-应答模式作为例子。但是它不是 HTTP。

对这个问题“我要怎样使用 ØMQ 构建我自己的新的 XYZ 服务器？”有一个好的答案。在我的例子中你可以使用任意你想到的协议，但是你应该把那个协议服务器（它可以非常小）链接到 ØMQ 的后台。你可以用任意语言编写的代码扩展你的后台，如你所愿地远程运行或者本地运行。Zed Shaw 的 Mongrel2 网页就是这种结构的一个很好的例子。

输入/输出线程

我们说道 ØMQ 在后台线程执行输入/输出。除了几个别的外，一个输入/输出线程（对于所有的套接字）对所有的应用程序都够用了。这就是我们创建一个背景的时候使用的神奇的“1”，意思是“使用一个输入/输出线程”。

```
void *context;
context = zmq_init (1);
```

ØMQ 应用程序与传统的网络应用程序有一个很大的不同，你不用每次连接的时候都创建一个套接字。对一个特定的端点一个套接字可以处理所有的输入和输出连接。例如，你只需要用一个套接字发布到一千多个订阅方。你在二十多个服务之间分布任务，只需要一个套接字就够了。你从一千多个应用程序网页收集数据只需要一个套接字就够了。

0MQ 对你如何编写应用程序有很重要的影响。传统的网络应用程序对每个远程连接都有一个进程或者线程，一个进程或者线程处理一个套接字。ØMQ 允许你把它整个的结构转换为一个单一的线程，当需要缩放的时候就可以终止它。

消息模式

在牛皮纸（brown paper）封装的 ØMQ 应用程序套接字下面的是消息模式的世界。如果你再消息队列的公司呆过，你会对这些感到有点熟悉。但是对于 ØMQ 的初学者来说，会很惊讶，我们习惯了用一个套接字描述一个节点的 TCP 模式。

让我们概括的描述一下 ØMQ 究竟做些什么。它快速而高效地发送整块的数据（消息）到节点。这里的节点可以是线程，进程或者机子（box）。它为你的应用程序提供一个单一的编程接口工作套接字，而不管使用的传输协议是什么（像进程间，进程内，TCP 或者多播）。各端点撤销或者接入的时候 ØMQ 套接字都会自动重新连接。当需要的时候，在发送方或者接收方它会自动给消息排队。它认真地处理这些队列，以确保进程不会耗尽内存而溢出到磁盘。它会处理套接字错误。它在后台线程中处理所有输入/输出。节点之间的会话使用的是无锁技术，因此从来不会存在锁定，等待，信号量（semaphores）和死锁的问

题。

跳过这个问题，它路由和排队都以消息模式为基础。就是这些模式给了 ØMQ 聪明。它封装了我们辛苦得来的分配数据和消息的最好的方法。ØMQ 的消息模式是硬编码的，但是在未来的版本中可能允许用户自定义模式。

ØMQ 模式是由匹配类型的套接字对来实现的。换句话说，为了理解 ØMQ 模型，你要先理解套接字类型以及它们如何一块工作的。这个问题主要是要认真学习，没有什么技巧。

基本的 ØMQ 模型是：

- 请求-应答，它把一组客户机连接到一组服务器。这是一种远程程序访问和任务分发方式。

- 发布-订阅，它把一组发布方连接到一组订阅方。这是一种数据分发方式。

- 管道。它以扇入/扇出模型连接节点，会有很多步骤和循环。这是一个并行任务收集和分发方式。

我们在第一章就已经见到了这三种模型。还有一种模型，是当人们把 ØMQ 当做传统 TCP 套接字的时候用用到的。

- 专用的套接字对，它用专门的对连接两个套接字。对于专门的，高级的用例来说，它是一种低级的模型。在本章的最后我们将看到一个例子。

zmq_socket(3)的手册页关于这个模型写的很清楚，你需要读几遍才能知道它的意义。我们会看到每一种模型以及它的用例。

这是一些套接字组合，它们组成有效地连接-绑定对（任何一边都可以绑定）：

- PUB and SUB
- REQ and REP
- REQ and XREP
- XREQ and REP
- XREQ and XREP
- XREQ and XREQ
- XREP and XREP
- PUSH and PULL
- PAIR and PAIR

任何别的组合都会导致不符合事实的和不可靠的结果，在 ØMQ 未来的版本中这样就会返回一个错误。你当然可以在别的套接字类型上通过代码建立连接，即，从一种类型套接字类型读取，而写入到另一种套接字类型中。

利用消息来工作

在线路上，ØMQ 消息可以是从零开始任意大小的字节块，适用于任意存储器。你可以通过谷歌协议栈，XDR，JSON,或者任意别的你的应用程序需要交谈的第三方软件。选择一种方便和快速的数据描述方法是很明智的，但是你可以作自己折中的决定。

在内存中，ØMQ 消息是 zmq_msg_t 结构（或者类，取决于你的语言）。以下列出的是在 C 语言中使用 ØMQ 消息的基本规则：

- 你创建并传递 zmq_msg_t 对象，数据不会阻塞。
- 为了读取一条消息，你首先利用 zmq_msg_init(3)创建一条空消息，然后你把它传送到 zmq_recv(3).。
- 为了从新数据中写一条消息，你使用 zmq_msg_init_size(3)创建一个消息，并同时分配一块有一定大小的数据。然后你用代码填充数据，并把消息传送到 zmq_send(3)。
- 为了释放（而不是撤销）一个消息你调用 zmq_msg_close(3)。这样扔掉了一个编号（reference）,事实上 ØMQ 将会毁坏这个消息。
- 为了访问消息的内容，你使用 zmq_msg_data(3)。为了知道消息中有多少数据，使用 zmq_msg_size(3).
- 不要使用 zmq_msg_move(3)，zmq_msg_copy(3)或者 zmq_msg_init_data(3)，除非你读了手册页并明确知道你为什么要用它们。

这就是利用消息工作的经典的代码块，如果你注意了的话，你就会发现它是比较熟悉的。这是我们在所有例子中都使用的文件 zhelpers.h 中的内容。

```
//  Receive ØMQ string from socket and convert into C string
static char *
s_recv (void *socket) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_recv (socket, &message, 0);
    int size = zmq_msg_size (&message);
    char *string = malloc (size + 1);
    memcpy (string, zmq_msg_data (&message), size);
    zmq_msg_close (&message);
    string [size] = 0;
    return (string);
}
```

```
//  Convert C string to ØMQ string and send to socket
static int
s_send (void *socket, char *string) {
    int rc;
    zmq_msg_t message;
    zmq_msg_init_size (&message, strlen (string));
    memcpy (zmq_msg_data (&message), string, strlen
(string));
    rc = zmq_send (socket, &message, 0);
    assert (!rc);
    zmq_msg_close (&message);
    return (rc);
}
```

你可以简单地扩展这些代码并接收任意长度的数据块。

注意，当你已经把一个消息传送到 zmq_send(3)，ØMQ 将会清除消息，即把消息的大小设置为零。你不能把相同的消息发送两次，并且在数据发送出去后你不能再访问它。

如果你想多次发送相同的消息，那么创建第二个消息，让后用 zmq_msg_init(3)来初始化，然后使用 zmq_msg_copy(3)创建第一个消息的副本。它复制的不是数据而是编号（reference）。你就可以发送者各消息两次（或者更多次，如果你创建更多的副本）了，并且只有在发送或关闭最后的副本后消息才被彻底的毁坏（destroy）。

ØMQ 也支持分段消息，它允许你把一系列数据块当做一条消息处理。它被广泛应用在实际的应用程序中，我们将在本章的后面部分以及第三章中会看到。

另外一些需要知道的关于消息的事：

- ØMQ 自动地发送和接收消息，即要么你收到一个完整的消息，要么你什么都收不到。
- ØMQ 不是立即发送消息，在一些不确定的时间之后再发。
- 你可以发送一个零长度的消息，例如从一个线程到另一个线程发送一个信号。
- 消息要适应内存。如果你想发送任意大小的文件，你应该把它们分块，并把每一块作为一个单独的消息来发。
- 如果使用的是在关闭一个作用域时不会自动毁坏（destroy）对象的语言，当你处理完一个消息以后，你必须调用 zmq_msg_close(3)。

很有必要重申一遍，还是不要使用 zmq_msg_init_data(3)。这是零拷贝的方法，避免你遇到创建的麻烦。在你开始担心（shaving off）微秒之前，还有关于 ØMQ 的重要的多的东西需要学习。

处理多功能的套接字

到现在为止的所有例子中，绝大多数例子的主要循环是：

1、 在套接字上等待消息；
2、 处理消息；
3、 重复。

如果我们想同时在多功能套接字中读取数据该怎么办？最简单的方法是把一个套接字连接到多个终端，然后使用 ØMQ 扇入。如果远程终端使用的是相同的模式的话，这样的操作是合法的，但是它也可能不合法，例如连接一个 PULL 套接字到一个 PUB 终端。可以玩玩，但是不合法。如果你使用混合的模型，你的结构将不稳定。

正确的方法是使用 zmq_poll(3)。一个更好的方法是把 zmq_poll(3)放在一个框架中，这样它就变成一个令人满意的事件驱动器了。这可能是最好的解决方法了，我们将在后面构造一个简单的反应器，

但主要的是它让我向你展示了如何使用非阻塞套接字读取方式。这是一个用非阻塞方式读两个套接字的例子。这个让人困惑的程序既像气象更新的订阅方，也像并行任务的工作进程：

```
//
//  Reading from multiple sockets
//  This version uses a simple recv loop
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    //  Prepare our context and sockets
    void *context = zmq_init (1);

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

    //  Connect to weather server
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

    //  Process messages from both sockets
```

```
    //  We prioritize traffic from the task ventilator
    while (1) {
        //  Process any waiting tasks
        int rc;
        for (rc = 0; !rc; ) {
            zmq_msg_t task;
            zmq_msg_init (&task);
            if ((rc = zmq_recv (receiver, &task, ZMQ_NOBLOCK))
== 0) {
                //  process task
            }
            zmq_msg_close (&task);
        }
        //  Process any waiting weather updates
        for (rc = 0; !rc; ) {
            zmq_msg_t update;
            zmq_msg_init (&update);
            if ((rc = zmq_recv (subscriber, &update,
ZMQ_NOBLOCK)) == 0) {
                //  process weather update
            }
            zmq_msg_close (&update);
        }
        //  No activity, so sleep for 1 msec
        struct timespec t;
        t.tv_sec = 0;
        t.tv_nsec = 1000000;
        nanosleep (&t, NULL);
    }
    //  We never get here but clean up anyhow
    zmq_close (receiver);
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

这种方法是以第一条消息的延迟为代价的（当没有消息等待处理的时候，休眠在循环的最后）。当微秒的延迟很重要的时候，这可能会给应用程序带来问题。你也需要检查nanosleep()的说明或者所有你使用的函数，以确定它没有忙循环（busy-loop）。

你可以从第一个开始公平的读取套接字，而不用像我们例子中一样优化它们。这叫做“公平排队”，当一个套接字从多个源接收到消息时，ØMQ 会自动地做这样的处理。

现在让我们看看这没有什么意义的应用程序如何正确运行，使用 zmq_poll(3)：

```
//
//  Reading from multiple sockets
//  This version uses zmq_poll()
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

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

    //  Connect to weather server
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

    //  Initialize poll set
    zmq_pollitem_t items [] = {
        { receiver, 0, ZMQ_POLLIN, 0 },
        { subscriber, 0, ZMQ_POLLIN, 0 }
    };
    //  Process messages from both sockets
    while (1) {
        zmq_msg_t message;
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmq_msg_init (&message);
            zmq_recv (receiver, &message, 0);
            //  Process task
            zmq_msg_close (&message);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmq_msg_init (&message);
            zmq_recv (subscriber, &message, 0);
            //  Process weather update
            zmq_msg_close (&message);
        }
    }
    //  We never get here
    zmq_close (receiver);
```

```
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

处理错误和 ETERM

ØMQ 的错误处理逻辑是混合快速失败和快速恢复。进程对内部错误应该进可能的脆弱，对外界的袭击和错误应该尽可能大的抵抗力。打个比喻，如果一个活细胞检测到一个内部错误它就会自我毁灭，然而它会采取所有可能的方法抵制外来侵略。不断询问 0MQ 代码的判断提示对于一个强壮的 0MQ 代码来说很重要，它们必须在细胞壁的正确的一边。并且也应该存在这样一个壁。如果不能确定一个错误是来自内部还是外部，那么设计就存在问题，需要改善。

在 C 语言中，判断提示会因为一个错误很快地终止应用程序。在别的语言中，可能会导致异常或者程序结束。

当 ØMQ 检测到一个外部错误时，它返回一个错误提示给调用程序。在一些罕见的情况下，如果没有一些可见的策略从错误恢复的话，它会悄悄的丢掉消息。在少数情况下，ØMQ 也会对外部错误生效，但这是它的缺陷。

到现在为止，我们看到的多数的 C 语言例子都没有错误处理部分。真正的代码对每一次 ØMQ 访问都要进行错误处理。如果你不是使用的 C 语言绑定，那么可能绑定语言会自动处理错误。但是在 C 语言中你需要自己处理。以下是一些简单的规则，以可移植操作系统接口协议开始：

- 如果创建对象的函数失败就会返回零；
- 别的函数如果成功才会返回零，如果发生异常（常常是失败）就会返回其它值（主要是（-1）；
- 错误代码由错误号或者 zmq_errno(3)提供；
- 描述错误记录的文本由 zmq_strerror(3)提供；

有两个主要的异常情况需要处理：

- 当一个线程以非阻塞方式调用 zmq_recv(3)，同时没有等待消息。ØMQ 将会返回-1 并设置错误号给 EAGIN。
- 当一个线程调用 zmq_term(3)而别的线程正以阻塞方式工作。zmq_term(3)调用关闭了背景，并且所有阻塞调用都返回-1 并退出，把错误号设置到 ETERM.

我们归结为，在 ØMQ 调用中，很多情况下我们可以这样设置判断提示信息：

```
  void *context = zmq_init (1);
  assert (context);
  void *socket = zmq_socket (context, ZMQ_REP);
  assert (socket);
  int rc;
  rc = zmq_bind (socket, "tcp://*:5555");
  assert (rc == 0);
```

在这个代码的第一个版本中，在函数中我调用了命令 assert()。这不是一个好办法，因为最优化设计会把所有宏指令转换为零，会对这些函数造成很大影响。使用返回代码，并让返回代码生效。

让我们看看如何完全关闭一个进程。我们将采用前面部分介绍的关于管道的例子。如果我们在后台打开了一大堆的工作进程，当批处理完成后我们希望关闭它们。我们通过发送一个关闭消息给工作进程来关闭它们。完成这个任务最好是在接收端，应为它知道什么时候批处理完成。

我们如何把监听器链接到工作进程呢？使用 PUSH/PUSH 套接字只是一种方法。ØMQ 的标准答案是：为每一种你需要解决的问题的类型创建一种新的套接字流。我们会使用发布-订阅模型发送关闭消息给工作进程。

- 监听器在一个终端创建一个 PUB 套接字。
- 工作进程把它们的输入套接字绑定到这个终端。
- 当监听器检测到一个批处理结束了它就发送一个结束消息给 PUB 套接字
- 当工作进程检测到结束消息，它就退出。

在这个监听器中没有多少新代码：

```
void *control = zmq_socket (context, ZMQ_PUB);
  zmq_bind (control, "tcp://*:5559");
  ...
  //  Send kill signal to workers
  zmq_msg_init_data (&message, "KILL", 5);
  zmq_send (control, &message, 0);
  zmq_msg_close (&message);
```

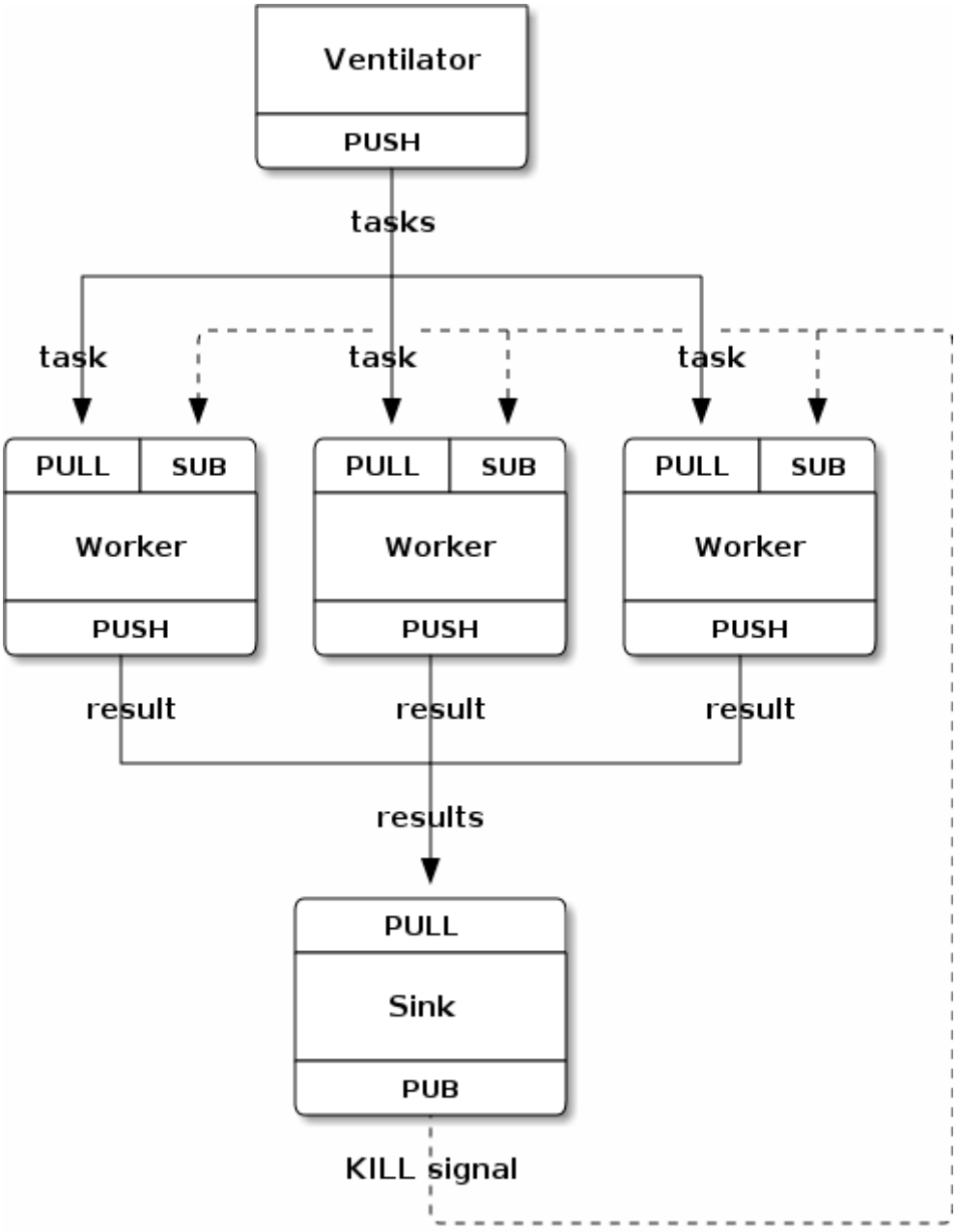

Figure 14 – Parallel Pipeline with Kill signalling

这就是使用我们之前见到的 zmq_poll(3)技术管理两个套接字（一个 PULL 套接字获得任务，一个 SUB 套接字接收控制命令）否认工作进程。

```
//
//  Task worker - design 2
//  Adds pub-sub flow to receive and respond to kill signal
//
#include "zhelpers.h"

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

```
{
   void *context = zmq_init (1);

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

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

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

   //  Process messages from receiver and controller
   zmq_pollitem_t items [] = {
      { receiver, 0, ZMQ_POLLIN, 0 },
      { controller, 0, ZMQ_POLLIN, 0 }
   };
   //  Process messages from both sockets
   while (1) {
      zmq_msg_t message;
      zmq_poll (items, 2, -1);
      if (items [0].revents & ZMQ_POLLIN) {
         zmq_msg_init (&message);
         zmq_recv (receiver, &message, 0);

         //  Process task
         int workload;         //  Workload in msecs
         struct timespec t;
         sscanf ((char *) zmq_msg_data (&message), "%d",
&workload);
         t.tv_sec = 0;
         t.tv_nsec = workload * 1000000;

         //  Do the work
         nanosleep (&t, NULL);

         //  Send results to sink
         zmq_msg_init (&message);
         zmq_send (sender, &message, 0);
```

```
            //  Simple progress indicator for the viewer
            printf (".");
            fflush (stdout);

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

这是修改了的监听器应用程序。当它完成了采集任务，它广播一个 KILL 消息给所有的工作进程：

```
//
//  Task sink - design 2
//  Adds pub-sub flow to send kill signal to workers
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

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

    //  Socket for worker control
    void *controller = zmq_socket (context, ZMQ_PUB);
    zmq_bind (controller, "tcp://*:5559");

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

    //  Start our clock now
```

```
    struct timeval tstart;
    gettimeofday (&tstart, NULL);

    //  Process 100 confirmations
    int task_nbr;
    int total_msec = 0;     //  Total calculated cost in msecs
    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);
    }
    //  Calculate and report duration of batch
    struct timeval tend, tdiff;
    gettimeofday (&tend, NULL);

    if (tend.tv_usec < tstart.tv_usec) {
        tdiff.tv_sec = tend.tv_sec - tstart.tv_sec - 1;
        tdiff.tv_usec = 1000000 + tend.tv_usec -
tstart.tv_usec;
    }
    else {
        tdiff.tv_sec = tend.tv_sec - tstart.tv_sec;
        tdiff.tv_usec = tend.tv_usec - tstart.tv_usec;
    }
    total_msec = tdiff.tv_sec * 1000 + tdiff.tv_usec / 1000;
    printf ("Total elapsed time: %d msec\n", total_msec);

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

    //  Finished
    sleep (1);              //  Give 0MQ time to deliver

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

分段消息

ØMQ 允许一条消息由几个帧构成，提供了“分段消息”的功能。现实的应用程序中广泛使用分段消息，特别是常用于制作“封包”。我们后面将会看到。我们现在要学的是如何安全地（但是无目的地）读和写分段消息，否则我们写的代码将不会与使用分段消息的应用程序工作。

当你使用分段消息的时候，每一段都是一条 zmq_msg。例如，如果你发送一条五段的消息，你将会构建，发送和毁坏五条 zmq_msg。你可以提前做这些（并且把 zmq_msg 存储在队列或者结构中），或者你一个一个地发送它们。

这是我们以分段消息的形式发送的帧（我们接收每一个帧到一个消息对象中）：

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

这是我们如何以一个消息的形式处理所有的帧的代码，不论是一个整体还是分段的：

```
while (1) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_recv (socket, &message, 0);
    //  Process the message part
    zmq_msg_close (&message);
    int64_t more;
    size_t more_size = sizeof (more);
    zmq_getsockopt (socket, ZMQ_RCVMORE, &more, &more_size);
    if (!more)
        break;      //  Last message part
}
```

关于分段消息需要知道的一些事项：

- 当你发送分段消息的时候，第一段（和接下来的所有段）会和最后一段数据同时发送。
- 如果你使用 zmq_poll(3)，当你接收到一条消息的第一段，接下来的所有段也都会到达。
- 你会接收到一条消息的所有部分，要么就什么也接收不到。
- 一个消息的每一段是一个独立的 zmq_msg 项。

- 不论你是否选择了 RCVMORE 选项，你都会接受到一条消息的所有部分。
- ØMQ 发送消息时，先把所有消息段排队，知道最后一段也进入队列，再把它们一起发送出去。
- 没有办法可以取消发送了一部分的消息，除非关闭套接字。

中间件和设备（Intermediates and Devices）

任意连接的装置都会有一个随着装置数量增加而变化的复杂的曲率。当装置的数量较少的时候，它们能够知道彼此，但是，当装置变大时，每个成员了解别的成员的成本就会线性增加，所有连接成员所用成本的总和就会指数增长。解决办法是把装置分解成更小的部分，并使用中间件把各部分连接起来。

这种模型在现实中非常广发的存在，这就是为什么我们的社会和经济生活中充满了中间件。中间件的作用就是降低较大网络的复杂性和缩减成本。中间件通常叫做批发商，分配器，管理器，等。

ØMQ 网络和别的网络一样，不使用中间件的话，不能超过一定的规模。在 ØMQ 中把它们叫做“设备”。当我们使用 ØMQ，我们经常建立一个网络上的一组节点 ，它们可以相互交谈，而不用通过中间件。

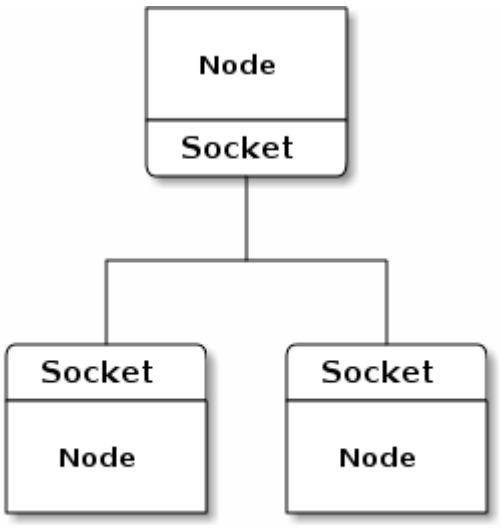

Figure 15 – Small scale ØMQ application

然后，我们跨一个更宽的网络扩展我们的应用程序，把设备放在专门的位置，扩大了节点的数量。

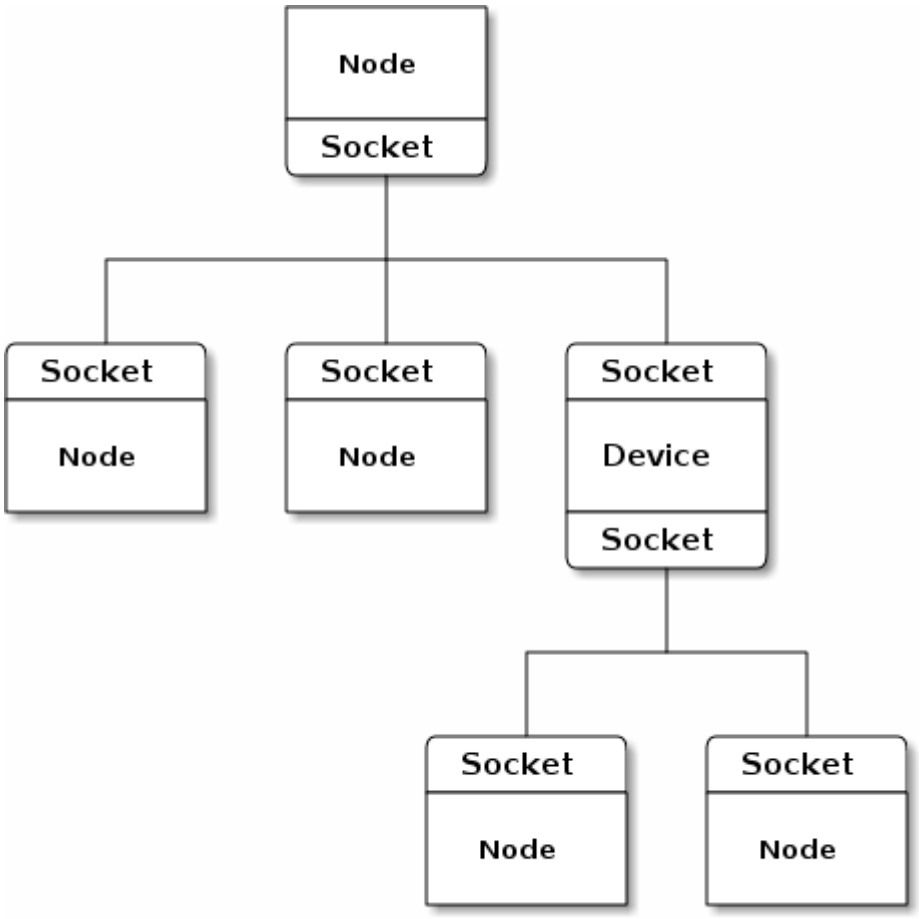

Figure 16 – Larger scale ØMQ application

ØMQ 设备通常把一组前向的套接字连接到一组后向的套接字，然而没有直接的设计规则。它们运行在无状态的情况下，因此可以通过尽可能多的中间件来延伸应用程序。你可以可以把它作为进程中的线程，或者作为单机中的进程来运行。ØMQ 提供了很多基础的设备，但是在实际应用中，你应该开发你自己的设备。

ØMQ 设备可以进行中介寻址，服务，排队，或者任何别的你小心定义在消息和套接字层上的抽象。不同的消息模型有不同复杂问题需要不同类型的中间件。例如，客户-服务器模型可用队列和服务器抽象，而发布-订阅模型可用流或者主题。

与传统的集中式中间代理相比 ØMQ 有趣的是你可以把设备准确地放在需要它们的位置，并且它们能够做最优化的中间代理。

发布-订阅代理服务器（proxy server）

常常会有在多个网络部分或者传输协议上扩展发布-订阅结构的请求。可能有一组订阅位于一个很远的位置。我们可能希望通过广播发布到本地的订阅方，并通过 TCP 发送到远处的订阅方。

我们要写一个简单的代理服务器，它位于一个发布方和一组订阅方之间，桥接两

个网络。这是作为一个有用的设备的最简单的例子。这个设备有两个套接字，前向套接字面对的是气象服务器处在内部的网络，后向套接字面对的是在外部网络的订阅方。它在前向套接字订阅气象服务，重新发布它的数据到后向套接字：

```
//
//  Weather proxy device
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

    //  This is where the weather server sits
    void *frontend = zmq_socket (context, ZMQ_SUB);
    zmq_connect (frontend, "tcp://192.168.55.210:5556");

    //  This is our public endpoint for subscribers
    void *backend  = zmq_socket (context, ZMQ_PUB);
    zmq_bind (backend, "tcp://10.1.1.0:8100");

    //  Subscribe on everything
    zmq_setsockopt (frontend, ZMQ_SUBSCRIBE, "", 0);

    //  Shunt messages out to our own subscribers
    while (1) {
        while (1) {
            zmq_msg_t message;
            int64_t more;

            //  Process all parts of the message
            zmq_msg_init (&message);
            zmq_recv (frontend, &message, 0);
            size_t more_size = sizeof (more);
            zmq_getsockopt (frontend, ZMQ_RCVMORE, &more,
&more_size);
            zmq_send (backend, &message, more? ZMQ_SNDMORE:
0);
            zmq_msg_close (&message);
            if (!more)
                break;      //  Last message part
        }
    }
    zmq_close (frontend);
```

```
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

我们把它叫做代理是因为它充当了发布方到订阅方，和订阅方到发布方的代理。你可以把这个设备装入网络中，而不会改变它（当然订阅方需要知道如何与代理交互）

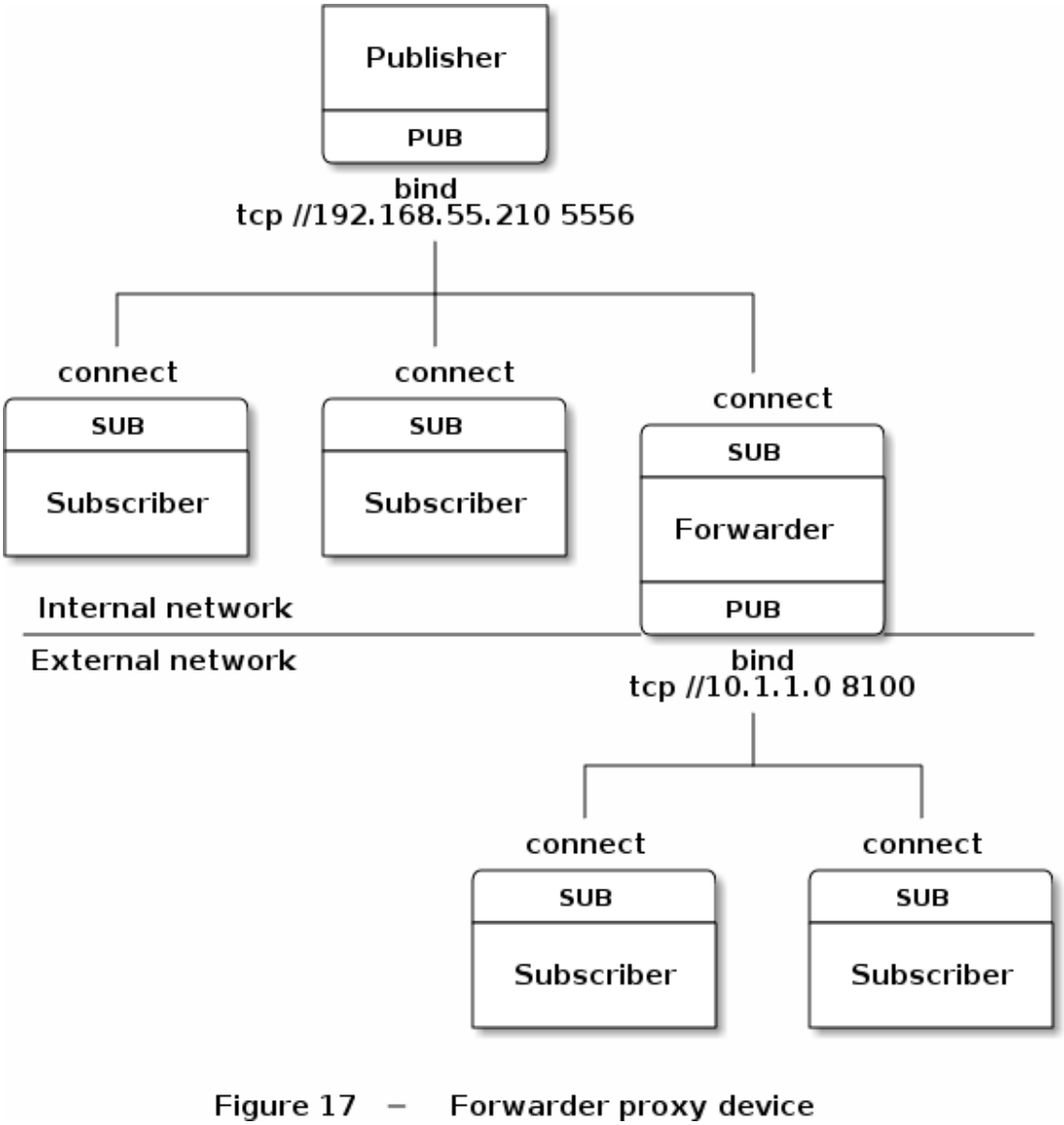

Figure 17 – Forwarder proxy device

注意这个应用程序是多重安全的。它正确的检测分段消息，和读它们一样发送它们。如果输出分段数据时我们不设置 ANDMORE 选项，最终的接收者将会获得一个毁坏的消息。你应该总是让你的设备多重安全的，这样才会保障你交换的数据没有风险。

请求-应答代理（broker）

让我们探究一下如何用 ØMQ 编写一个小的消息队列代理来解决缩放的问题。对于这种

情况我们将会考察客户-服务器模型。

在 HELLO WORLD 的客户-服务器模型中，一个客户机与一个服务器交互。然而现实中我们常常需要多个服务器和多个客户机。这样允许我们扩大服务器（多线程，或者多进程，或者多机子）的能力。唯一的限制是客户机必须是无状态的，所有的状态在请求中或者在一些共享的存储器如数据库中。

有两种方法可以把多个客户机连接到多个服务器。笨的方式是把每一个套接字链接到多个服务器端。一个客户机套接字可以连接到多个服务器套接字，请求在所有服务器之间是负荷均衡的。假设你连接一个客户机套接字到三个服务器终端 A,B 和 C。客户机发出请求 R1,R2,R3，R4。R1 和 R4 连接到服务器 A，R2 到 B，R3 到 C。

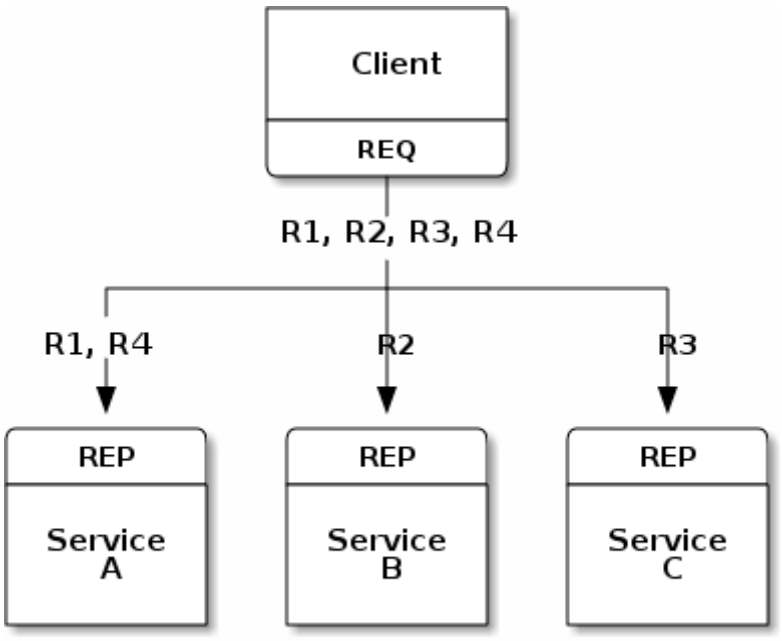

Figure 18 – Load balancing of requests

这样的设计允许你很便宜地增加更多的客户机。你也可以增加服务器。每个客户机都会负荷均衡到所有的服务器。每个客户机必须知道服务器的拓扑。如果你有 100 个客户机，并且你决定增加三个服务器。你需要重新配置和重启 100 个客户机，为的是让客户机知道新增加的三个服务器。

这当然不是我们希望在凌晨三点想做的事，这个时候我们的超级计算集群已经耗尽资源，而我们及其需要增加几百个新的服务器节点。太多的稳定部分就像流动的混凝土：知识是分布式的，你有更多的稳定部分，你就需要更努力地改变拓扑。我们需要的是位于客户机和服务器之间集中所有拓扑信息的东西。理想状况下，我们应该能够在任意时候增加或者删除客户机或者服务器，而不用触及拓扑的其余部分。

因此我们编写了一个提供给我们这种灵活性的消息队列中间代理。这个中间件绑定到两个终端，前端是客户机，后端是服务器。它使用 zmq_poll(3)监听这两个套接字的活性

（activity），当它们有活性的时候，它就在两个套接字之间来回的传送消息。它事实上不会显示地处理任何队列——ØMQ 会自动地在每个套接字上处理。

当你使用 REQ 与 REP 交谈，你获得的是一个直接同步的请求-应答对话。客户机发送一个请求，服务器读取请求并发送一个应答信号。然后客户机读取应答信号。不论是客户机还是服务器尝试做点别的事情（例如在一行发送两个请求而不等待一个响应）它们都会出错。

但是我们的代理应该是非阻塞的。很明显，我们可以使用 zmq_poll(3)等待任意一边的套接字的有活性，但是我们不能使用 REP 和 REQ.

幸运的是，有这两个套接字的非阻塞版本，叫做 XREQ 和 XREP.这些“扩展的请求/应答”套接字允许你通过中间件扩展请求-应答模型，例如我们的消息队列中间代理。

当我们扩展了请求-应答模型，REQ 与 XREP 会话，XREQ 与 REP 会话。在 XREQ 和 XREP 之间存在把消息从一个套接字拉下来，推到另一个套接字上的代码（像我们的中间代理一样）。

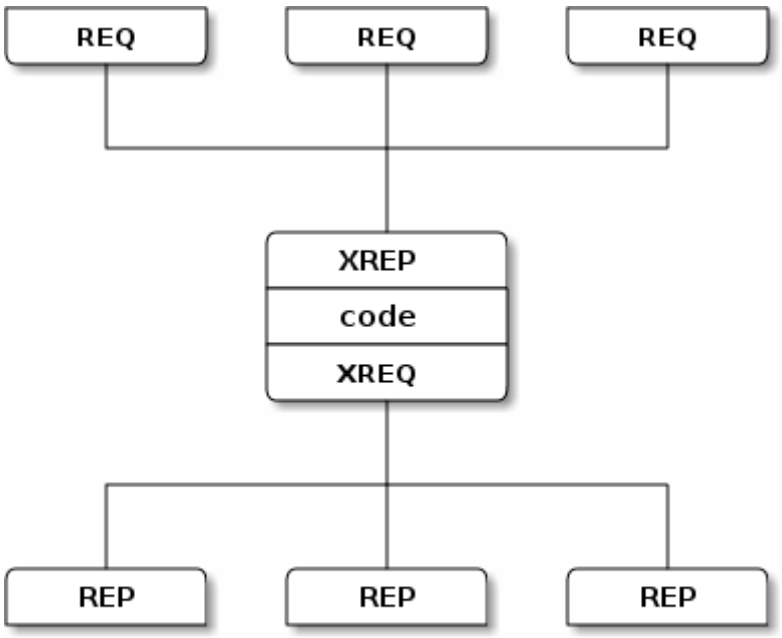

Figure 19 – Extending request-reply

请求-应答代理绑定到两个终端，一个（前端套接字）是为了与客户机连接，一个（后端套接字）是为了与服务器连接。为了测试这个代理，你希望改变你的服务器，使它们能够连接到后端套接字。这是一个 python 语言的客户机和服务器程序，它展示了我想说的：

```
#
#   Request-reply client in Python
#   Connects REQ socket to tcp://localhost:5559
```

```
#   Sends "Hello" to server, expects "World" back
#
import zmq

#  Prepare our context and sockets
context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5559")

#  Do 10 requests, waiting each time for a response
for request in range(1,10):
    socket.send("Hello")
    message = socket.recv()
    print "Received reply ", request, "[", message, "]"
```

examples/Python/rrclient.py

All languages

```
#
#   Request-reply service in Python
#   Connects REP socket to tcp://localhost:5560
#   Expects "Hello" from client, replies with "World"
#
import zmq

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.connect("tcp://localhost:5560")

while True:
    message = socket.recv()
    print "Received request: ", message
    socket.send("World")
```

这是用 C 语言写的代理程序。你可以看到它是多重安全的：

```
//
//  Simple request-reply broker
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    //  Prepare our context and sockets
```

```
   void *context = zmq_init (1);
   void *frontend = zmq_socket (context, ZMQ_XREP);
   void *backend  = zmq_socket (context, ZMQ_XREQ);
   zmq_bind (frontend, "tcp://*:5559");
   zmq_bind (backend,  "tcp://*:5560");

   //  Initialize poll set
   zmq_pollitem_t items [] = {
      { frontend, 0, ZMQ_POLLIN, 0 },
      { backend,  0, ZMQ_POLLIN, 0 }
   };
   //  Switch messages between sockets
   while (1) {
      zmq_msg_t message;
      int64_t more;         //  Multipart detection

      zmq_poll (items, 2, -1);
      if (items [0].revents & ZMQ_POLLIN) {
         while (1) {
            //  Process all parts of the message
            zmq_msg_init (&message);
            zmq_recv (frontend, &message, 0);
            size_t more_size = sizeof (more);
            zmq_getsockopt (frontend, ZMQ_RCVMORE, &more,
&more_size);
            zmq_send (backend, &message, more? ZMQ_SNDMORE:
0);
            zmq_msg_close (&message);
            if (!more)
               break;     //  Last message part
         }
      }
      if (items [1].revents & ZMQ_POLLIN) {
         while (1) {
            //  Process all parts of the message
            zmq_msg_init (&message);
            zmq_recv (backend, &message, 0);
            size_t more_size = sizeof (more);
            zmq_getsockopt (backend, ZMQ_RCVMORE, &more,
&more_size);
            zmq_send (frontend, &message, more?
ZMQ_SNDMORE: 0);
            zmq_msg_close (&message);
            if (!more)
```

```
                break;      //  Last message part
            }
        }
    }
    //  We never get here but clean up anyhow
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

使用请求-应答代理可以使客户-服务模式更容易伸缩，因为客户机看不见服务器，服务器也看不见客户机。唯一稳定的节点是中间的代理。

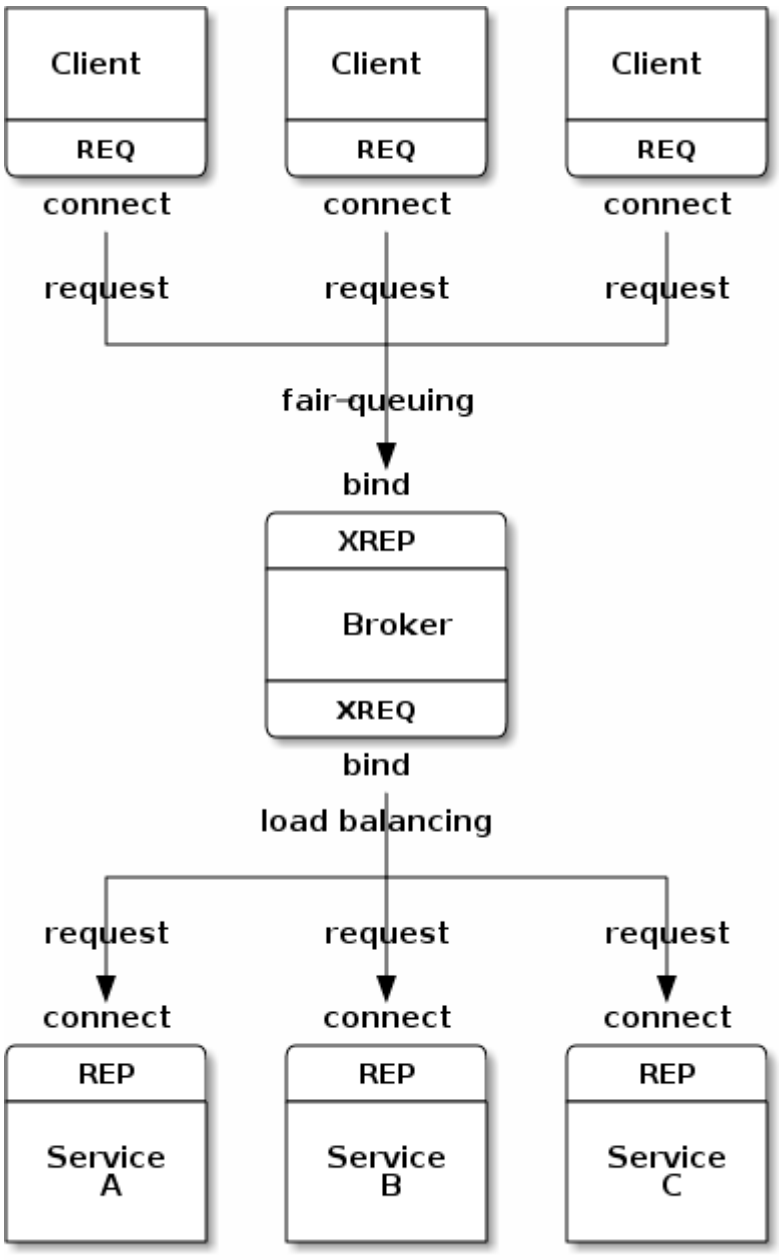

Figure 20 – Request reply broker

内置设备

虽然多数高级使用者会编写自己的设备，ØMQ 还是提供了一些内置设备。这些内置设备是：

- QUEUE，它像请求-应答代理。
- FORWARDER,它像发布-订阅代理服务器。
- STREAMER,它像 FORWARDER 但是用于管道流。

为了开始一个设备，我们调用 zmq_device(3)并且传递两个套接字给它，一个作为前端，一个作为后端。

```
zmq_device (ZMQ_QUEUE, frontend, backend);
```

你开始一个 QUEUE 设备，就如你把请求-应答代理装入你代码中一样。你需要创建套接字，绑定或者连接它们，并且可能在调用 zmq_device(3)之前还要设置它们。这是件琐碎的事情。这是重新编写的访问 QUEUE 的请求-应答代理，它 rebadged 像一个听起来很昂贵的代码（人们花费几小时编写的代码的作用更小）。

```
//
//  Simple message queuing broker
//  Same as request-reply broker but using QUEUE device
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

    //  Socket facing clients
    void *frontend = zmq_socket (context, ZMQ_XREP);
    zmq_bind (frontend, "tcp://*:5559");

    //  Socket facing services
    void *backend = zmq_socket (context, ZMQ_XREQ);
    zmq_bind (backend, "tcp://*:5560");

    //  Start built-in device
    zmq_device (ZMQ_QUEUE, frontend, backend);

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

内置设备会恰当地处理错误，但是我们展示的例子不是这样的。因为你可以根据你的需要来设置你的套接字，所以在开始设备之前，如果可以的话很值得使用内置设备。

如果你和绝大多数 ØMQ 的使用者一样，到这步你应该在想，“如果我把任意的套接字放入设备中，会有什么样的坏事情发生呢？”简单的答案是：不要这样做。你可以混合使

用套接字类型，但是结果会很奇怪。因此请坚持使用 XREP/XREQ 作为队列设备，SUB/PUB 作为传送器，PULL/PUSH 作为流转换器。

当你需要别的组合的时候，你就需要写自己的设备。

ØMQ 处理多线程

ØMQ 可能是编写多线程应用程序最好的方法。然而，如果你习惯了使用传统套接字，使用 ØMQ 套接字时会有一些调整。0MQ 会把你知道的所有关于多线程应用程序仍到公园里的垃圾堆，并把它们点燃。引起争论的书很少，但是很多关于并发编程的书是值得的。

为了编写极其完美的多线程应用程序（（我的意思是照字面上），我们不需要互斥，锁或者任何别的内部线程通信形式，只需要消息通过 ØMQ 套接字发送。

我用“完美”多线程程序是说代码很容易写和容易理解，可用任何语言在任何操作系统中使用统一的技术，可以跨任意数量的 CPU 而没有等待状态，也没有递减返回点。

如果你已经用了几年时间让你的多线程代码能够工作，还不用说速度，锁，信号量和临界区，当你意识到它什么也不为的时候你会感到厌烦。如果只有一件事是我们 30 年并发编程学到的，它就是：不要分享状态。它就像两个酒鬼想分享一杯啤酒一样。即使他们是哥们。他们迟早会打架。加入这个分享的人越多，他们彼此间为了啤酒打架的次数就会增多。可悲的多数多线程应用程序看起来就像酒吧中的战争。

当你编写经典的状态分享多线程代码的时候，你需要面对的一系列奇怪的问题可能会很滑稽，如果你不直接把它们翻译成压力或者危机的话，如代码在压力下突然运行失败。这有一张“在你多线程代码中的 11 个类似的问题”的表，它来自一个有处理错误代码经验的大公司：忘记同步，不正确的间隔尺度，分开的读和写，无锁，重新排序，锁护航，两步跳，优先级转换。

Yeah, we also counted seven, not eleven。虽然不是这个问题、问题是你真的希望控制电网或者股市的代码在一个很忙的星期三凌晨三点得到一个两部锁定？谁在意这句话究竟是什么意思呢？这不是把我们带到编程的东西，它会产生复杂的边界效应伴随复杂的 hacks。

用 ØMQ 编写多线程应用程序的时候你需要遵循以下规则：

- 不允许从多个线程访问相同的数据。
- 你必须为你的进程创建一个背景，并把它传送给所有的线程。这个背景采集 ØMQ 的状态。为了创建一个跨进程的连接：传输，服务器和客户机线程必须分享相同的背景对象。

- 不允许在线程间共享套接字，即使采用合适的排斥机制如信号量，锁定，或者互斥。在将来的某个时候 ØMQ 会允许在不同线程之间移动套接字。目前，创建套接字的线程是唯一能够使用这个套接字的线程。

- 如果你必须使用已经存在的代码，你还是应该尽力避免。不要使用互斥，信号量，临界区，而使用 ØMQ 消息。

如果你需要在一个进程中开始多个应用程序，例如，你希望各自运行在自己的线程中。在一个线程中创建设备套接字，并把它传送到另一个线程中的设备很容易出错。它可能看起来会运行，但是最终会随机的失败。记住：只有创建套接字的线程才能使用它。

如果你遵循了这些规则，当需要的时候，你就很容易把线程分解成分离的过程。应用程序逻辑可以位于线程，进程，机子：任意你需要的规模。

ØMQ 使用本身的线程，而不是虚拟的“绿色”线程。它的优点是你不需要任何新的线程应用程序接口，ØMQ 线程可以清楚地匹配你的操作系统。你可以使用标准工具像 Intel 的“ThreadChecker“来查看你的应用程序正在处理的事情。缺点是你的代码，例如要开始新的线程，将会不方便，并且，如果你有大量（几千个）线程，操作系统就会过压。

让我们看看它在实际中是如何工作的。我们将把之前的 Hello World 服务器变的更多功能的。原始的服务器是单线程的。如果每个请求都很慢，那就没有问题：一个 ØMQ 线程能够全速在 CPU 核上运行，没有等待，完成很多任务。但是实际的服务器必须完成重要的任务、当 10000 个客户机同时访问服务器的时候，单个核可能就不够用了。因此，实际的服务器必须启动多任务线程。然后，它尽可能快的接收请求，并把它们分发给它的工作线程。工作线程这处理任务，并最终把应答信号发送回去。

你当然可以通过一个队列设备或者外部工作进程来完成这些工作，但是启动一个有十六个核的进程比启动十六个各自拥有一个核的进程要简单。此外，通过线程运行任务能够解决网络跳，延迟和网络通信的问题。

多线程的 Hello World 服务从根本上使队列设备和工作进程变成一个单一的进程：

```
//
//  Multithreaded Hello World server
//
#include "zhelpers.h"

static void *
worker_routine (void *context) {
    //  Socket to talk to dispatcher
    void *receiver = zmq_socket (context, ZMQ_REP);
```

```
    zmq_connect (receiver, "inproc://workers");

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

int main () {
    //  Prepare our context and sockets
    void *context = zmq_init (1);

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

    //  Socket to talk to workers
    void *workers = zmq_socket (context, ZMQ_XREQ);
    zmq_bind (workers, "inproc://workers");……绑定到一个套接字，还是一个字符串？

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

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

所有这些代码你应该都了解了。它是怎样运行的呢：

- 服务器开始一组工作线程每个工作线程创建一个 REP 套接字，并在这个套接字上处理请求。工作线程就像单线程服务器，唯一的不同就是传输方式（inproc 而不是 tcp）,和绑定方向。

- 服务器创建一个 XREP（扩展的应答）套接字和客户机对话，把它绑定到外部接口（通过 tcp）

- 服务器创建一个 XREQ（扩展的请求）套接字和工作进程对话，把它绑定到内部接口（通过 inproc）

- 服务器启动一个连接两个套接字的 QUEUE 设备。QUEUE 设备对输入请求保持一个单一的队列，并把它分布给工作线程。它也会把应答信号路由回它们的源地址。

这里“工作“是一个一秒的脉冲。在工作线程中我们可以做任何事，包括与别的节点对话。这就是从 ØMQ 套接字和节点的角度看到的多线程服务器。注意请求-应答模型链式：REQ-XREP-queue-XREQ-REP。

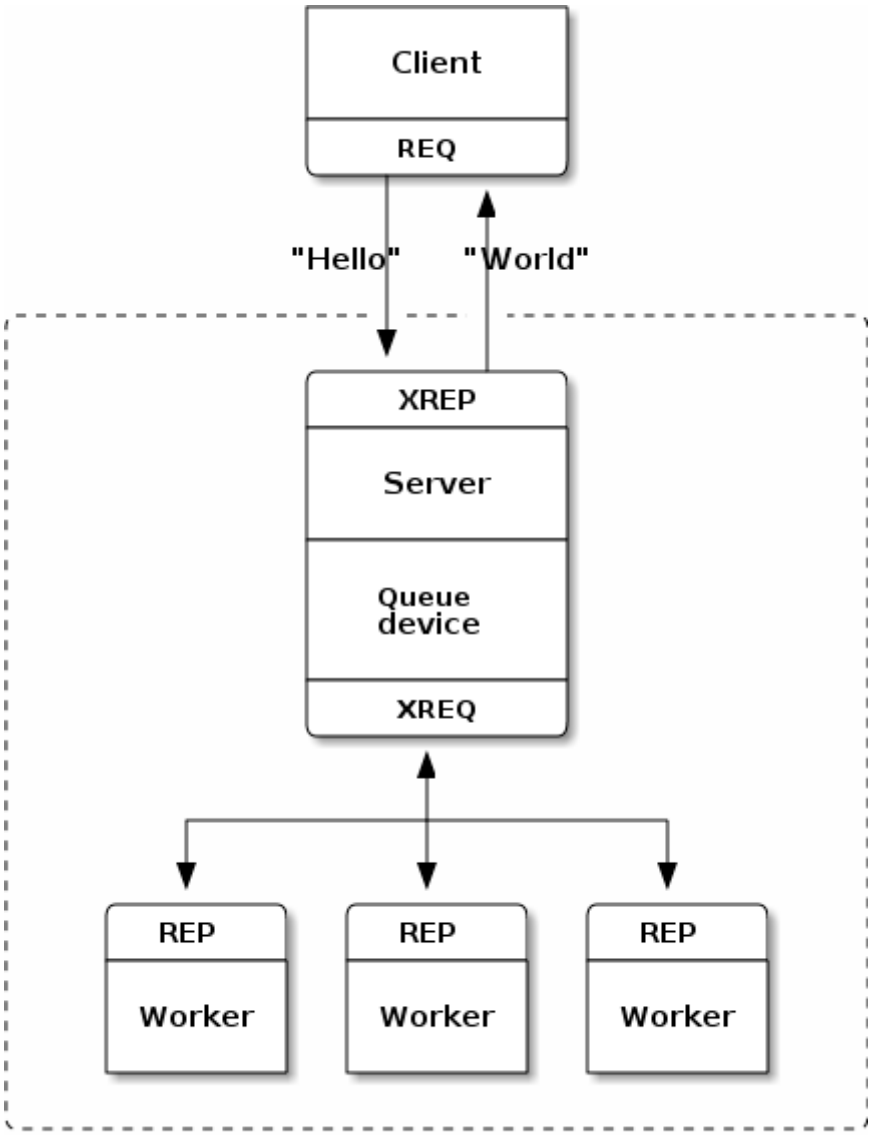

Figure 21 – Multithreaded server

线程组合

当你开始用 ØMQ 编写多线程应用程序，你将会面对如何组合你的线程的问题。虽然你可能打算插入“休眠“状态，或者使用多线程技术如：信号量，互斥，但是你唯一应该使用的机制是 ØMQ 消息。记住醉汉和啤酒瓶的故事。这个例子展示了三个线程在它们准备好后互相发送信号。

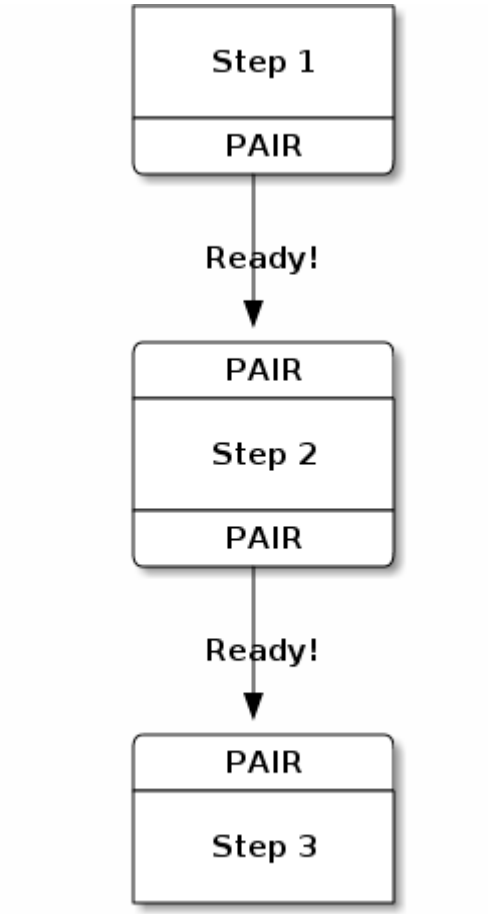

Figure 22 – The Relay Race

在这个例子中我们使用 PAIR 套接字和 inproc 传输方式：

```
//
//  Multithreaded relay
//
#include "zhelpers.h"

static void *
step1 (void *context) {
    //  Signal downstream to step 2
    void *sender = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (sender, "inproc://step2");
    s_send (sender, "");

    return NULL;
}

static void *
step2 (void *context) {
```

```
    //  Bind to inproc: endpoint, then start upstream thread
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step2");
    pthread_t thread;
    pthread_create (&thread, NULL, step1, context);

    //  Wait for signal
    char *string = s_recv (receiver);
    free (string);

    //  Signal downstream to step 3
    void *sender = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (sender, "inproc://step3");
    s_send (sender, "");

    return NULL;
}

int main () {
    void *context = zmq_init (1);

    //  Bind to inproc: endpoint, then start upstream thread
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step3");
    pthread_t thread;
    pthread_create (&thread, NULL, step2, context);

    //  Wait for signal
    char *string = s_recv (receiver);
    free (string);

    printf ("Test successful!\n");
    zmq_close (receiver);
    zmq_term (context);
    return 0;
}
```

这是我们第一次看到的使用 PAIR 的例子。为什么使用 PAIR 呢？别的套接字好像也会运行，但是它们都有会影响信号的边界效应。

- 你可以使用 PUSH 作为发送者，PULL 作为接收者。它看起来简单，并且也会运行，但是记住 PUSH 会负荷均衡消息到所有的接收者。如果你偶然打开了两个接收者（即已经有一个在运行你又打开了另一个），你将会丢失一半

的信号。PAIR 有拒绝多个连接的作用，它是专门的。

- 你可以使用 XREQ 作为发送者，XREP 作为接收者。然而 XREP 把你的消息包裹在一个“信封“中，意思是零个字节大小的信号变成那个了分段消息。如果你不考虑数据，把什么都当做有用信号，并且如果你只从这个套接字读取一次，那就不会有问题。然而，如果你决定发送真实的数据，你会发现 XREP 提供给你错误的消息。XREQ 也是负荷均衡的。与 PUSH 存在相同的危机。

- 你可以使用 PUB 作为发送者，SUB 作为接收者。当你发送消息的时候它们会正确的分发，并且 PUB 并不像 PUSH 或者 XREQ 一样是负荷均衡的。然而，你需要用一个空的订阅设置订阅方，这很让人厌烦。更糟的是，PUB-SUB 连接的可靠性是取决于时间的，如果 PUB 发送消息的时候 SUB 正在连接，那么消息就会丢失。

基于这些原因，对于特定的线程 PAIR 是最好的选择。

节点组合

你想组合节点的时候，PAIR 套接字将不会是好的选择。这是少数线程和节点不同地方之一。理论上节点可以随意加入或者撤销，而线程是稳定的。如果远程节点加入或者撤销的话 PAIR 套接字不会自动连接。

线程和节点之间第二个重要的不同在是通常你有固定数量的线程和不确定数量的节点。让我们使用之前的脚本（气象服务器和客户机）作为例子，并使用节点组合来确定启动的时候订阅方不会丢掉数据。

这说明了应用程序是如何工作的：

- 发布方事先知道它期望有多少连接的订阅方。这是它从某个地方获得的奇怪的数字。

- 发布方启动并等待所有的订阅方连接。这是节点组合部分。每个订阅方订阅并通过另一个套接字告诉发布方它准备好了。

- 当发布方连接了所有订阅方，它开始发布数据。

在这个例子中我们将会使用 REQ-REP 套接字流同步化发布方和订阅方。这是发布方程序：

```
//
//  Synchronized publisher
//
```

```
#include "zhelpers.h"

//  We wait for 10 subscribers
#define SUBSCRIBERS_EXPECTED  10

int main () {
    void *context = zmq_init (1);

    //  Socket to talk to clients
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5561");

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

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

    s_send (publisher, "END");

    sleep (1);              //  Give 0MQ/2.0.x time to flush
output
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

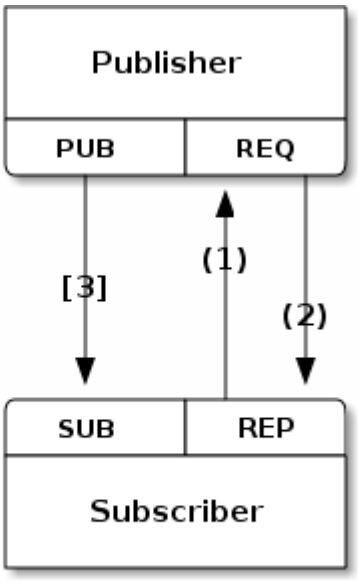

Figure 23 – Pub Sub Synchronization

这是订阅方程序：

```
//
//  Synchronized subscriber
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

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

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

    //  - send a synchronization request
    s_send (syncclient, "");

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

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

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

这个 Linux 脚本将会启动十个订阅方，然后是发布方：

```
echo "Starting subscribers..."
for a in 1 2 3 4 5 6 7 8 9 10; do
    syncsub &
done
echo "Starting publisher..."
syncpub
```

它给了我们满意的输出：

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

注意，在退出发布方之前我们调用了 sleep (1)。这是版本 ØMQ/2.0 的问题，如果你退出程序太快的话它就会丢失你还没有发送的消息。如果你使用的是版本 ØMQ/2.1 你可以删除这条休眠语句。但是在 ØMQ/2.0 版本中如果没有这个休眠的话就不能够恰当地退出发布方。

更坚固的模型是：

- 发布方打开 PUB 套接字并开始发送消息”Hello”（不是数据）。
- 当订阅方接收到一个 hello 消息的时候连接 SUB 套接字，并通过 REQ/REP 套接字对告诉发布方。
- 当发布方有了所有必须的确认，它开始发送真实的数据。

零拷贝

在第一章，当你还是一个新手的时候，关于零拷贝，我们戏弄了你。如果你已经走到了这步，你一样做好了使用另拷贝的准备了。然而，记住到成功的路有多种，到现在为止，提前的优化不是最让人喜欢或者有益的。当你的结构不完美的时候尝试另拷贝只会浪费时间并让事情变得更糟。

ØMQ 消息编程接口允许你直接应用程序缓冲区发送和接收消息而不用拷贝数据。为了使 ØMQ 在后台发送消息，零拷贝技术还需要别的技术。

为了能够实现另拷贝，使用 zmq_msg_init_data(3)创建一个消息利用 malloc()已经分配给堆得一块数据，然后你把它传送给 zmq_send(3)。当你创建消息的时候，你也传送了一个函数，当结束消息的发送的时候 ØMQ 会调用来释放这块数据区。这是最简单的例子，假设缓冲区是堆上分配的一块 1000 字节区。

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

在接收的时候没法另拷贝：ØMQ 分发给你一个缓冲区你想存储多久就能存储多久，但是但是不能直接把数据写入应用程序缓冲区。

在写方面 ØMQ 的分段消息与另拷贝协作的很好。在传统的消息中，你需要把不同的缓冲区整理成一个缓冲区这样你才能发送。这就意味着要拷贝数据。而 ØMQ 可以把来自不同源的多部分缓冲区作为独立的消息部分发送。把每部分作为长度分割的帧。对于应用程序，

它看起来像一系列发送和接收访问。但是在内部，只用单一的系统指令就可以把多部分写到网络上合从网络上接收数据，因此，它很高效。

临时的及持久的套接字

持久套接字的概念是又一个令人惊讶的 ØMQ 发明，当你看到它的时候你会好奇，为什么之前没有人想到它呢？在经典网络中，套接字是应用程序接口对象，它们的生存期仅限于使用它们的代码。但是如果你仔细看一个套接字，你会发现它收集很多资源-网络缓冲区-并且在某个阶段，ØMQ 使用者会问，“在程序崩溃的时候不能挂起吗，这样我就可以把它们取回了？”

这个想法看起来很有用。它不是万无一失的，它为 ØMQ 提供了一种可靠性，对于发布订阅的例子特别有用。我们马上会看到。

这好似一个一般的例子关于两个套接字高兴地谈论天气，谁在哪在什么地方吻了谁，在上次的员工派对上我听到了一些不同的东西，更不用说你看到

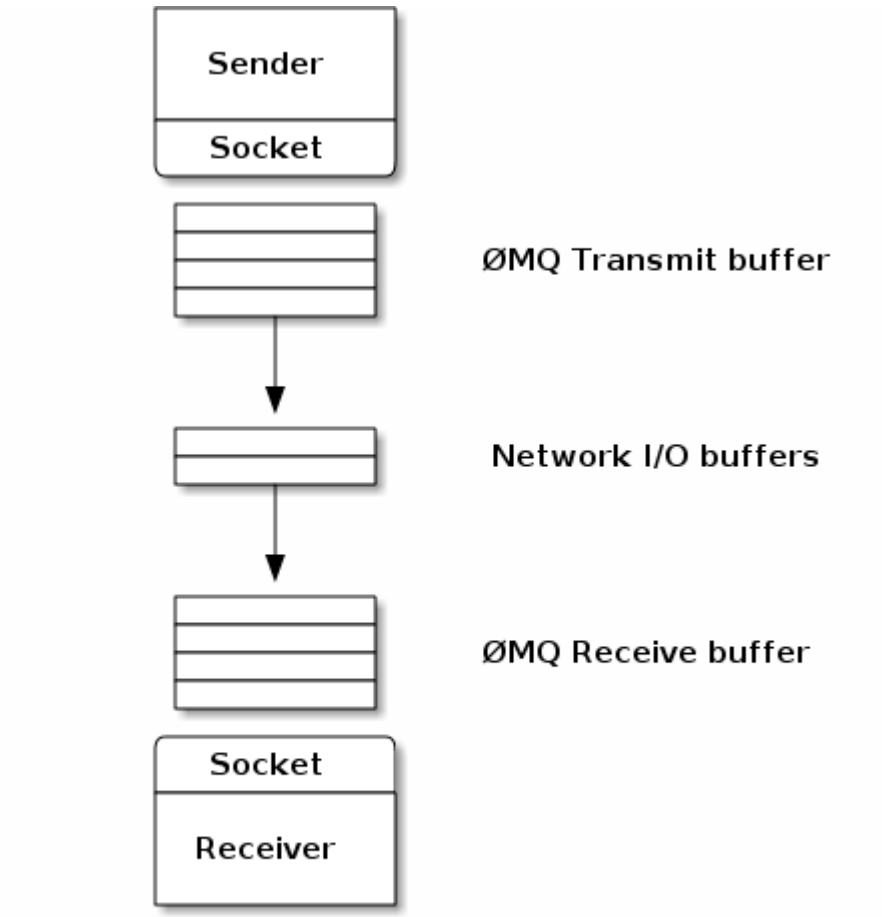

Figure 24 – Sender boring the pants off receiver

持久套接字允许 ØMQ 发送缓冲区与和发送者存在同样长的时间。如果接收者崩溃掉，它就会丢掉它的就收缓冲区，网络也会丢掉它的输入/输出缓冲区，但是发送者可以继续把

数据放到它的发送缓冲区，并且之后就收者可以接收到。

注意，ØMQ 的发送和接收缓冲区是不可见和自动地，就像 TCP 的缓冲区一样。

到现在为止我们使用的套接字都是临时的。为了把一个临时套接字转变为持久套接字，你给它一个身份。所有 ØMQ 套接字都有身份但是默认情况下它们用的是“专门的通用身份”（UUIDS）这样端点可以用它来反复调用它的谈话对象。

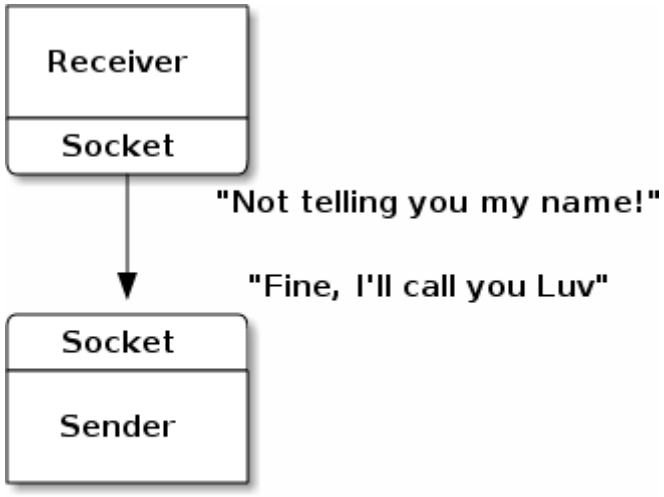

Figure 25 – Transient socket

在你不可见的背后，当一个套接字连接到另一个，这两个套机字相互交换身份。通常套接字不会告诉它们端点它们的身份，因此端点为彼此虚构随机的身份。

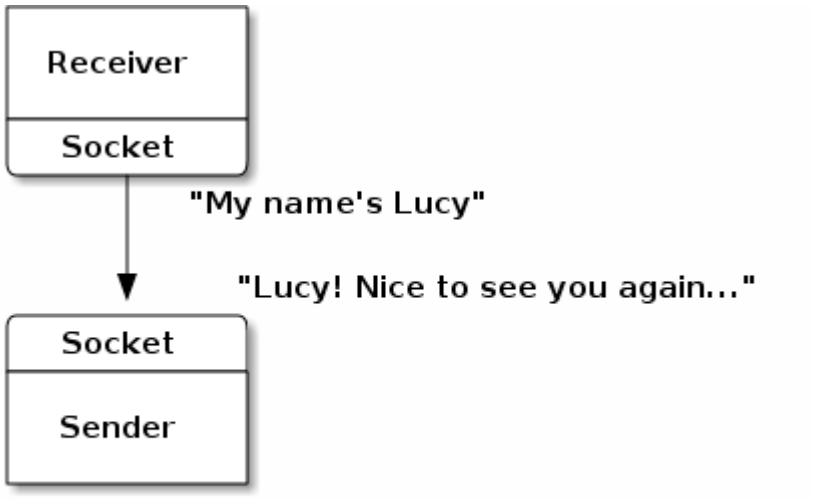

Figure 26 – Durable socket aka. Can't Escape

这显示了如何设置身份以创建持久套机字：

```
zmq_setsockopt (socket, ZMQ_IDENTITY, "Lucy", 4);
```

一些关于套接字身份的建议：

- 必须在连接或者绑定套接字之前设置它的身份。
- 身份是二进制字符串：以零字节开始的字符串是为 ØMQ 使用预留的
- 不要给两个套接字使用相同的身份。在 ØMQ/2.0.9 及以前的版本中，这样会让另一个套接字声明失败。是的，这是一个缺陷，将来会得到改善。
- 在绑定以后或者重启一个端点时不要再修改套接字的身份，这样会使连接到它的套接字声明失败。是的，这也是一个缺陷，将来也会得到改善。
- 在应用程序中不要使用创建很多套接字的随机身份。这样会导致大量的持久套接字堆积，事实上会使节点崩溃。
- 如果你需要知道发送给你消息的节点的身份，只有 XREP 套接字会自动为你处理。对于任何别的套接字你都需要发送一个地址作为消息的一部分。

看 zmq_setsockopt(3) 作为 ZMQ_IDENTITY 套接字选项的概括。注意 zmq_getsockopt(3)提供给你正在使用的套接字的身份，而不是它可能连接的任何端点的。

当一个失败的声明错误使用的时候，ØMQ 不会起作用，只有当它引起内部错误的时候，ØMQ 才会起作用。代码应该都有不会失败的错误提示。这在进行关于如何处理这些情况的讨论。一般规则下，如果使用 ØMQ 有判断提示的时候，把它报告给 zeromq-dev 表。

发布订阅消息封装

我们已经看到了分段消息。现在让我们看看它主要的用例，即消息封装。封装是一种用地址安全打包数据的方法，不用触及到数据本身。

在发布订阅模型中，封装至少有发布的关键词，以便滤波，但是你也能在封装中加入发送方的身份。

你可以自己构建法币订阅封装。它是可选择的，在以前的发布订阅例子中我们没有这样做。使用发布订阅封装对简单的例子需要更多的工作，但是它更加清楚，特别是对真实的情况，关键词和数据是分开的。如果你直接从应用程序缓冲区写数据，它会更快。

这是用封装发布订阅消息的样子：

Figure 27 – Pubsub envelope with separate key

再次调用发布订阅适合基于前缀的消息。把关键词放在一个单独的帧中，让匹配变得可见，因为应用程序没有偶然匹配部分数据的机会。

这是一个关于发布订阅的代码形式简单抽象的例子。发布方发送两个类型的消息 A 和 B。封装中有消息类型：

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

int main () {
    //  Prepare our context and publisher
    void *context = zmq_init (1);
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5563");

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

订阅方只需要 B 类型的消息：

```
//
```

```
//  Pubsub envelope subscriber
//
#include "zhelpers.h"

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

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

当你运行着两个程序的时候，发布方会显示给你：

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

这个例子显示了发布滤波器拒绝或者接收整个分段消息（关键字包含在数据中）。你永远不会得到分段消息的一部分。

如果你订阅到多个发布方并且希望知道它们的身份，以便你能够通过另外的套接字给它们发送数据（这是一个相当经典的用例），你创建一个三部分的消息。

Figure 28 – Pubsub envelope with sender address

构建一个（半）持久订阅方

身份使用于所有套接字。如果你有一个 PUB/SUB 套接字，并且订阅方给发布方它的身份，发布方保存数据直到它能够分发给订阅方。

它很棒也很恐惧。它很棒是因为更新数据可以在发布方的缓冲区中等待，直到你连接并收集它们。它很恐怖因为默认情况下你会快速结束你的发布方并且锁定你的系统。

如果使用持久套接字（即在一个 SUB 套接字上设置身份）你必须防止你的队列爆满，通过在发布套接字上使用高水位线。

如果你想证明，仍拿第一章的 wuclient 和 wuserver 来看，并在 wuclient 中连接之前加入这一行：

```
    zmq_setsockopt (subscriber, ZMQ_IDENTITY, "Hello", 5);
```

编译并运行这两个程序。它看起来很正常。但是看看发布方使用的内存，你就会发现当订阅方完成的时候，发布方的内存开始不断增长。如果你重启订阅方，发布方停止增长。订阅方一旦撤销，发布方有开始增长。它很快就会压垮你的系统。

我们将首先看看怎样做，然后看看怎样恰当的处理。这是使用第二章节点组合技术的个发布方和订阅方。发布方发送十个消息，每个消息之间等待一秒。这个等待是让你使用拷贝结束订阅，等待几秒，然后重启它。

这是发布方：

```
//
//  Publisher for durable subscriber
//
#include "zhelpers.h"

int main () {
    void *context = zmq_init (1);

    //  Subscriber tells us when it's ready here
    void *sync = zmq_socket (context, ZMQ_PULL);
```

```
    zmq_bind (sync, "tcp://*:5564");

    //  We send updates via this socket
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5565");

    //  Wait for synchronization request
    char *string = s_recv (sync);
    free (string);

    //  Now broadcast exactly 10 updates with pause
    int update_nbr;
    for (update_nbr = 0; update_nbr < 10; update_nbr++) {
        char string [20];
        sprintf (string, "Update %d", update_nbr);
        s_send (publisher, string);
        sleep (1);
    }
    s_send (publisher, "END");

    sleep (1);              //  Give 0MQ/2.0.x time to flush
output

    zmq_close (sync);
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

这是订阅方：

```
//
//  Durable subscriber
//
#include "zhelpers.h"

int main (int argc, char *argv[])
{
    void *context = zmq_init (1);

    //  Connect our subscriber socket
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_setsockopt (subscriber, ZMQ_IDENTITY, "Hello", 5);
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);
```

```
    zmq_connect (subscriber, "tcp://localhost:5565");

    //  Synchronize with publisher
    void *sync = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sync, "tcp://localhost:5564");
    s_send (sync, "");

    //  Get updates, expect random Ctrl-C death
    while (1) {
        char *string = s_recv (subscriber);
        printf ("%s\n", string);
        if (strcmp (string, "END") == 0) {
            free (string);
            break;
        }
        free (string);
    }
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

为了运行，在各自的窗口中启动发布方和订阅方。允许订阅方收集一条或者两条消息然后拷贝它。到了三就重新启动。你将会看到下面的结果：

```
$ durasub
Update 0
Update 1
Update 2
^C
$ durasub
Update 3
Update 4
Update 5
Update 6
Update 7
^C
$ durasub
Update 8
Update 9
END
```

仅看看不同的地方，注释掉订阅方中设置套接字身份的这一行，再运行。你会看到有消息丢失了。设置身份就把临时套接字转换为持久套接字了。实际中你需要认真选择身份，

或者从结构文件中提取，或者有 UUID 产生或者从存储它的某个地方。

当我们给 PUB 套接字设置高水位线，发布方存储那么多的消息，但不能够更多。在我们发布到套接字之前，让我们把发布方的高水位线设置成 2 个消息。：.....第一次出现高水位线

```
uint64_t hwm = 2;
zmq_setsockopt (publisher, ZMQ_HWM, &hwm, sizeof (hwm));
```

现在运行我们的测试，在几秒的脉冲后结束或者重启订阅方将会显示下面的结果：

```
$ durasub
Update 0
Update 1
^C
$ durasub
Update 2
Update 3
Update 7
Update 8
Update 9
END
```

仔细看：我们保存了两个消息（2 和 3），中间间隔几个消息，然后又开始更新。高水位线导致 ØMQ 丢掉了它不能放在队列中的消息，ØMQ 参考手册叫做“异常情况”。

简单地说，如果你使用订阅身份，你必须在发布方套接字设置高水位，否则你的服务器可能会耗尽内存并崩溃掉。然而，有一个解决方法。ØMQ 提供了叫做“分区”的东西，它是一个存储我们不能存储在队列中的消息的磁盘文件。它的使用非常简单：

```
//  Specify swap space in bytes
uint64_t swap = 25000000;
zmq_setsockopt (publisher, ZMQ_SWAP, &swap, sizeof (swap));
```

当仍然提供持久套接字给需要它的东西的时候。我们可以把这些放在一起以制造一个多功能的发布方，它能够避免慢速，阻塞或者订阅方不存在的情况。

```
//
//  Publisher for durable subscriber
//
#include "zhelpers.h"

int main () {
    void *context = zmq_init (1);
```

```
    //  Subscriber tells us when it's ready here
    void *sync = zmq_socket (context, ZMQ_PULL);
    zmq_bind (sync, "tcp://*:5564");

    //  We send updates via this socket
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5565");

    //  Prevent publisher overflow from slow subscribers
    uint64_t hwm = 1;
    zmq_setsockopt (publisher, ZMQ_HWM, &hwm, sizeof (hwm));

    //  Specify swap space in bytes, this covers all
subscribers
    uint64_t swap = 25000000;
    zmq_setsockopt (publisher, ZMQ_SWAP, &swap, sizeof
(swap));

    //  Wait for synchronization request
    char *string = s_recv (sync);
    free (string);

    //  Now broadcast exactly 10 updates with pause
    int update_nbr;
    for (update_nbr = 0; update_nbr < 10; update_nbr++) {
        char string [20];
        sprintf (string, "Update %d", update_nbr);
        s_send (publisher, string);
        sleep (1);
    }
    s_send (publisher, "END");

    sleep (1);              //  Give 0MQ/2.0.x time to flush
output

    zmq_close (sync);
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

实际中，如果把高水位线设置为一，并把所有的消息都放在洗盘将会导致发布订阅系统非常慢。对必须处理未知订阅方的发布方来说有一个更明智的“最佳实践”：

- 总是给套接字设置一个基于期望的订阅方数量的最大值，你打算用于队列的内存的数量，和一个消息平均大小的高水位线。例如，如果你希望有 5000 个订阅方，有 1G 的内存可有，消息平均 200 字节，那么一个安全的高水位线应该是（1000000000/200/5000）=1000.

- 如果你不希望因为订阅方太慢或者崩溃而丢失数据，设置一个基于订阅方数量、最大消息传送速率、消息的大小和你希望使用的消息的时间的足够大能够处理峰值的分区。例如，对于 5000 个订阅方，消息平均大小 200 字节，并以 100000 每秒的速度接收，你每秒需要 100MB 的磁盘空间。因此，为了能够通过长达一秒的中断，你需要 6GB 的磁盘空间，并且它必须很快，但是这是另一个问题。

关于持久订阅方的一些注意事项：

- 它取决于订阅方如何结束，数据更新的频率，网络缓冲区的大小，和你使用的传输协议，数据可能还会丢失。持久订阅方比临时的有更好的可靠性，但是它不完美。

- 分区文件不是可回收的，因此，如果一个发布方结束并重启，它将会丢掉在它的临时缓冲区中的数据，而它在网络输入/输出缓冲区。

使用高水位线的注意事项：

- 它会影响到单一套接字的发送和接收缓冲区。一些套接字（PUB,PUSH）只有发送缓冲区。一些（SUB,PULL,REQ,REP）只有接收缓冲区。一些（XREQ,XREPX,PAIR）既有接收缓冲区又有发送缓冲区。

- 在 ØMQ 未来的版本中可能会为发送和接收缓冲区提供单独的高水位线。但是这好像不是人们特别担心的事情。

- 当你的套接字达到它的高水位线，它要么阻塞要么丢掉数据，这取决于套接字类型。PUB 套接字如果达到高水位线会丢掉数据，而别的套接字则会阻塞。

纯粹的需要

ØMQ 就像由各部分粘成的盒子，唯一的限制是你的现象力和节制。

可变的伸缩性结构可能令你开了眼界。你可能需要喝一杯或者两倍咖啡。不要犯我犯过的错误买标有无咖啡因的德国咖啡。它不以为着“美味”。可伸缩性结构不是一个新方法基于流的编程和像 Erlang 这样的语言已经这样运行了-ØMQ 让它的使用比以前更简单。

如 Gonzo Diethelm 所说，“我直觉的感受概括在这句话里面：“如果 ØMQ 不存在，那么有必要发明它。”。意思是我再几年脑后台处理今年后进入 ØMQ 的世界，突然有种

感觉.....现在 ØMQ 对我来说是纯粹的需要。

第三章-高级请求应答

在第二章中我们看到了一些 ØMQ 的一些基础应用，通过开发一系列小的应用程序，每个应用程序都是对 ØMQ 新的部分的探索。在这章中我们将会继续使用这种方法，来探索 ØMQ 请求应答模型的高级部分。

我们会涉及到：

- 在请求-应答模型中如何使用消息封装。
- 如何是哟个套接字 REQ,REP,XREQ,XREP。
- 如何用身份设置手动应答寻址。
- 如何自定义随机分散路由。
- 如何自定义最近最少使用路由。
- 如何建立高水准消息类。
- 如何建立基础的请求应答代理。
- 如何为套接字选择好的名称。
- 如何模拟一个客户机和工作进程的集群。
- 如何建立一个请求应答集群的可扩展云。
- 如何使用管道套接字检测线程。

请求应答封装

在请求应答模型中，封装握有应答的返回地址。这就是无状态的 ØMQ 网络为什么能创建往返请求应答对话的原因。

在通常的应用中，你事实上不需要明白请求-应答封装是如何工作的。当你使用 REQ 和 REP，你的套接字会自动地建立和使用封装。当你写一个设备的时候，我们在上一章学过，你只需要读和写一条消息的所有部分。ØMQ 利用分段数据实现封装，因此，如果你安全的拷贝了分段数据，你也拷贝了封装。

然而，走到幕后，操作请求-应答封装只适合于高级的请求-应答任务。是该解释 XREP 是如何以封装的形式工作的了。

- 当你从 XREP 套接字接收一条消息，它会给消息加上一个牛皮纸信封并用不退色的墨水写上“它来自 Lucy”。让后把它给你。就是说 XREP 给你从传输线上得到的数据，并用写有应答地址的信封包裹起来。

- 当你发送一个消息给 XREP 套接字，它把牛皮纸信封撕掉，尝试读它自己的字迹，如果它知道“Lucy”是谁，把内容发回 Lucy。这是与接收一条消息相反的过程。

如果你把牛皮信封丢掉，然后把消息传送给另一个 XREP 套接字（例如发送给连接到 XREP 的 XREQ）。第二个 XREP 将会给它装上另一个信封，并在它上面写上那个 XREQ 的名称。

总体上是，XREP 知道如何把应答发送回正确的地方。在你的应用程序中你唯一需要做的是关心牛皮信封。现在 REP 套接字起作用了。它小心地撕开牛皮信封，一个接一个地，把它们安全地放在一边，并给你（有 REP 套接字的应用程序代码）原始的消息。当你发送应答，它重新把应答包裹在牛皮信封中，这样它就可以把处理了的牛皮包裹交回链的下一个 XREP 套接字。

它让你这样插入一个 XREP-XREQ 设备到请求-应答模型：

```
[REQ] <--> [REP]
[REQ] <--> [XREP--XREQ] <--> [REP]
[REQ] <--> [XREP--XREQ] <--> [XREP--XREQ] <--> [REP]
...etc.
```

如果你把一个 REQ 套接字连接到 XREP 套接字，然后发送一个请求消息，这就是你从 XREP 套接字得到的结果：

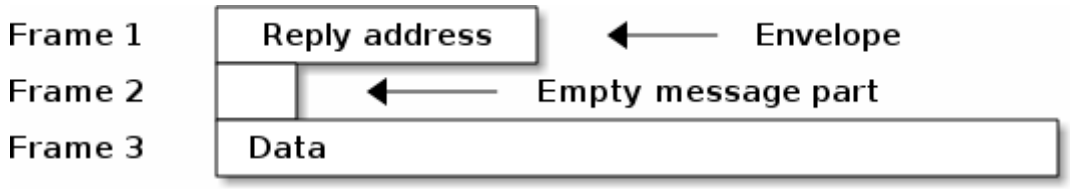

Figure 29 – Single hop request-reply envelope

把它们分类：

- 数据帧 3 是发送应用程序发送给 REQ 套接字的。

- 在第二帧的空消息部分是由 REQ 套接字在发送消息给 XREP 套接字时前置的。

- 帧 1 的应答地址是由 XREP 在发送消息给接收应用程序前前置的。

现在如果我们用一链的设备扩展，我们在信封上进行封装，最新的信封总是在栈的开始：

Figure 30 – Multihop request-reply envelope

这是一个关于我们用作请求应答模式四个套接字类型的更详细的解释：

- XREQ 会负荷均衡你发送的消息到连接的所有节点，并且对它接收到的消息公平排队。它确实像一个 PUSH 和 PULL 套接字组合。

- REQ 追加一个空消息部分在每个你发送的消息之前，并把空消息部分从你接收到的消息移去。然后它会像 XREQ 一样工作（事实上是以 XREQ 为基础）除非它也强制实行的是直接的发送/接收循环。

- XREP 对每一个它接收的消息在把它传递到应用程序之前追加一个带应答地址的信封。他也会砍掉它发送的每个消息的信封（第一个消息部分），并且通过应答地址决定消息应该发送到哪个端点。

- REP 存储了所有消息部分，包括空消息部分，当你接收一个消息的时候，他把剩下的（数据）传送给你的应用程序。当你发送一个应答，REP 把封装前置给消息并用相同的方式如 XREP（事实上 REP 是建立在 XREP 之上），但是强制执行直接的发送/接收循环的话就匹配 REQ。

REP 要求封装以空消息部分结束。如果在链的另一端没有使用 REQ，你必须自己增加空消息部分。

关于 XREP 公认的问题是，从哪获得应答地址呢？很显然答案是通过套接字的身份。我们学过，一个套接字可以是临时的，在这种情况下别的套接字（例如 XREP）产生一个身份，这样就可以和这个套接字联系。或者，如果套接字是持久的，它可以直接告诉别的套接字（又是 XREP）它的身份并且 XREP 可以直接使用而不是产生一个临时的标签。

这是临时套接字的情况：

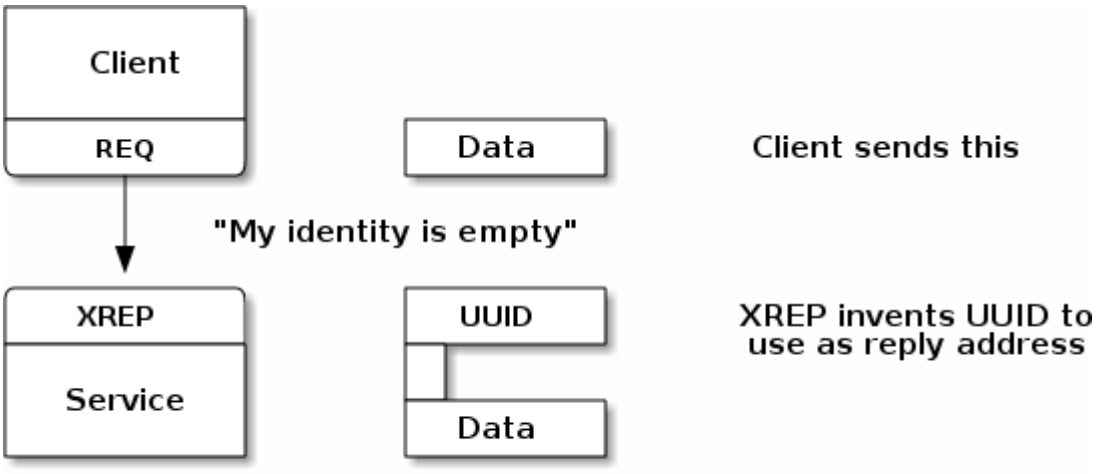

Figure 31 – XREP invents a UUID for transient sockets

这是持久套接字的情况：

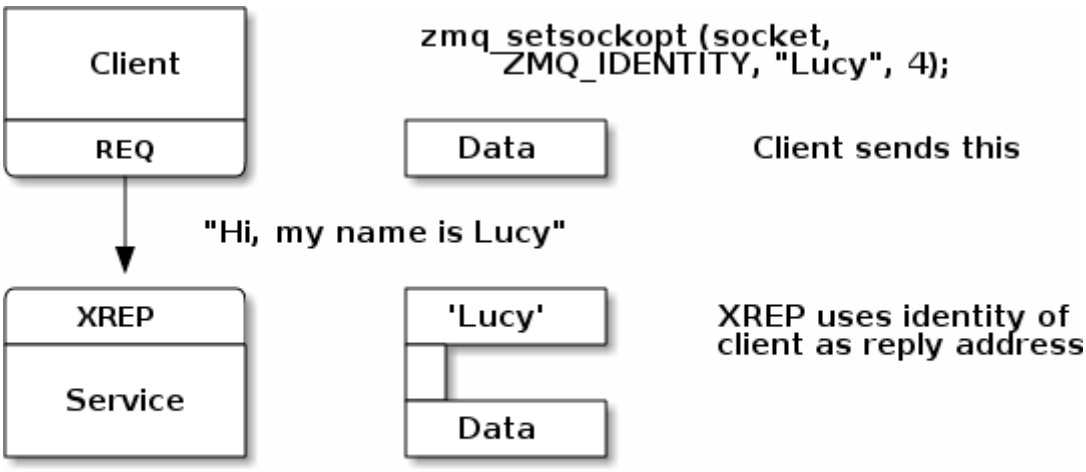

Figure 32 – XREP uses identity if it knows it

让我们在实际中看看以上两种情况。程序转存消息的内容部分这样 XREP 套接字从两个 REP 套接字接收消息，一个不使用身份，一个使用身份‘hello’：

```
//
//  Demonstrate identities as used by the request-reply
pattern.  Run this
//  program by itself.  Note that the utility functions s_ are
provided by
//  zhelpers.h.  It gets boring for everyone to keep repeating
this code.
//
#include "zhelpers.h"
```

```
int main () {
    void *context = zmq_init (1);

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

    //  First allow 0MQ to set the identity
    void *anonymous = zmq_socket (context, ZMQ_REQ);
    zmq_connect (anonymous, "inproc://example");
    s_send (anonymous, "XREP uses a generated UUID");
    s_dump (sink);

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

    zmq_close (anonymous);
    zmq_close (identified);
    zmq_term (context);
    return 0;
}
```

这是转存程序打印的东西：

```
----------------------------------------
[017] 00314F043F46C441E28DD0AC54BE8DA727
[000]
[026] XREP uses a generated UUID
----------------------------------------
[005] Hello
[000]
[038] XREP socket uses REQ's socket identity
```

自主请求-应答路由

我们已经看到 XREP 使用消息封装以决定把应答路由回哪个客户机。现在让我们用另一种方法来表现它：XREP 将会异步路由消息到任何与它相连的节点，只要你通过一个恰当的构造封装提供了一个正确的路由地址。

因此 XREP 是一个完全可控的路由器。让我们详细了解它的魔力。但是首先，让我们调整当我们试图分辨”REP”,”REQ”,”XREP”,”XREQ”时的痛苦。应该由法律来禁止名称如此相似。

为了容易读，并且因为我们将要从平顺的路出来，走上不平顺不规则的地方了。在本文的这部分，让我们重新命名这四种类型的套接字：

- REQ 是一个妈妈套接字，不要听，而是期望一个答案。妈妈套接字是一步的，如果你使用它们，它们永远是请求链的一个终端。

- REP 是一个爸爸套接字，总是回答，但是从不开始一个会话。爸爸套接字也是异步的，如果你使用它，它总是在应答链的终端。

- XREQ 是一个处理套接字，来回移动消息。处理套接字均匀地处理消息到 N 个爸爸套接字（发送请求等待应答）和/或一个路由器（等待请求，并发送应答）。

- XREP 是一个路由套接字，能够把消息路由到专门的节点。路由器可以和 N 个任意类型的节点会话，但它们通常是与妈妈套接字会话。

有件关于妈妈套接字的是，我们都像孩子一样听话，你不能说话，直到有人和你说。妈妈没有爸爸一样简单开放的胸襟，或者野心“当然，不管什么”耸耸肩，避开处理器套接字。为了与一个爸爸套接字对话，你必须先让妈妈套接字与你会话。好的是妈妈套接字不管你是否现在就应答，或者还要等会。

另一方面，爸爸套接字是强壮而沉默的，并且是 laundrypedantic。它们只做一件事，就是对于你问的任何问题，给你一个很规范和准确的答案。不要期望一个爸爸套接字话多，或者把消息传到别的地方，这是不会发生的。

处理套接字习惯于与爸爸套接字会话，它通过给每个爸爸套接字一个卡片，来公平地处理一组爸爸套接字。你不能贿赂处理套接字让它特别对待某个爸爸套接字，它对任何形式的劝说都有免疫力。

路由套接字是 ØMQ 请求-应答模型与外界通信的节点，能够与别的所有套接字会话。如果你让一个路由套接字把消息传送到它不知道的某个节点，它也不会抱怨或者说“不”，它会把消息分开地扔到垃圾桶中。不管什么时候如果你需要与一个专门的节点会话，你都需要一个路由套接字。

当我们经常认为请求-应答模型是一个到-和-自模型，事实上，它完全可以异步，只要我们明白任何妈妈套接字或者爸爸套接字会在链的终端，从不会在它的 中间，并且总是同步的。我们需要知道的是我们想要会话的对方的地址，然后我们就可以通过一个路由套接字与它异步地发送消息。路由套接字是 ØMQ 中唯一能够被告知“发送消息到 X”的套接字类型，其中 X 是一个连接节点的地址。

这是我们知道要发送消息到哪个地址的方法，你可以看到很多都在自主请求-应答路由的例子中用到了：

- 如果它是一个匿名的端点，即不设置任何身份，路由套接字将会产生一个 UUID 并在它传送给你一个输入请求封装的时候利用它来查找连接的节点。

- 如果它是一个有明确身份的节点，在路由提传送给你输入请求封装的时候将会为你提供这个身份。

- 有明确身份的节点能够通过另外的机制发送消息，例如通过一些别的套接字。

- 节点可以实现知道彼此的身份，例如通过配置文件或者别的方法。

- 这里有三个随机路由模型，每一个对应一种我们能够连接到路由器的套接字类型：

- 路由套接字-到-处理套接字，也叫做 XREP-到-XREQ

- 路由套接字-到-妈妈套接字，也叫做 XREP-到-REQ

- 路由套接字-到-爸爸套接字，也叫做 XREP-到-REP

- 路由套接字-到-路由套接字，也叫做 XREP-到-XREP

每一种情况我们都能够完全控制路由消息的方法，但是不同模型的用例和消息流不同。我们在下一部分将会用例子分别介绍各自不同的路由算法。

但是，首先看一些关于自主路由的警告：

- 这违背了相当固定的 ØMQ 规则：代理节点与套接字通信。我们使用它的唯一原因是 ØMQ 缺乏一个广泛使用的路由算法。

- 将来的 ØMQ 可能会恰当地做我们一会要建立的路由。意思是我们现在设计的代码可能会中断，或者在将来会显得多余。

- 当内置路由确实能够保证可扩展性，比如对设备友好，而自足路由不会。你需要构建你自己的设备。

因此，总的来说，自主路由更昂贵，并且更脆弱。只有你需要的时候采用它。已经说过了，让我们跳水吧，水很美妙。

自主离散路由

路由套接字-到-处理套接字模型是最简单的。你把一个路由套接字连接到很多处理套接字，然后使用任何你喜欢的算法分发消息给处理套接字。处理套接字可以是监听器（处理

没有任何响应的消息），代理（把消息发送到别的节点），或者服务器（发回应答）。

如果你希望处理套接字应答，那么需要一个套接字与它会话。处理套接字不能与专门的节点会话，因此，如果它们有多个节点，那么在它们之间会负荷均衡，这会很诡异。如果处理套接字是一个监听器，任何数量的路由套接字都可以与它会很。

什么样的路由你能够用路由套接字-到-路由套接字模型？如果处理套接字回答路哟套接字，例如告诉路由套接字它什么时候完成一个任务，你是否能用这种方式路由取决于你的处理套接字有多快。因为路由套接字和处理套接字都是异步的，这会变得有点复杂。你至少需要使用 zmq_poll(3)。

我们将会看一个处理套接字不会回话的例子，它是单纯的监听器。我们的算法是加权自主离散的。我们有两个处理套接字，并且发送两次消息发送到两个套接字的消息一样多。

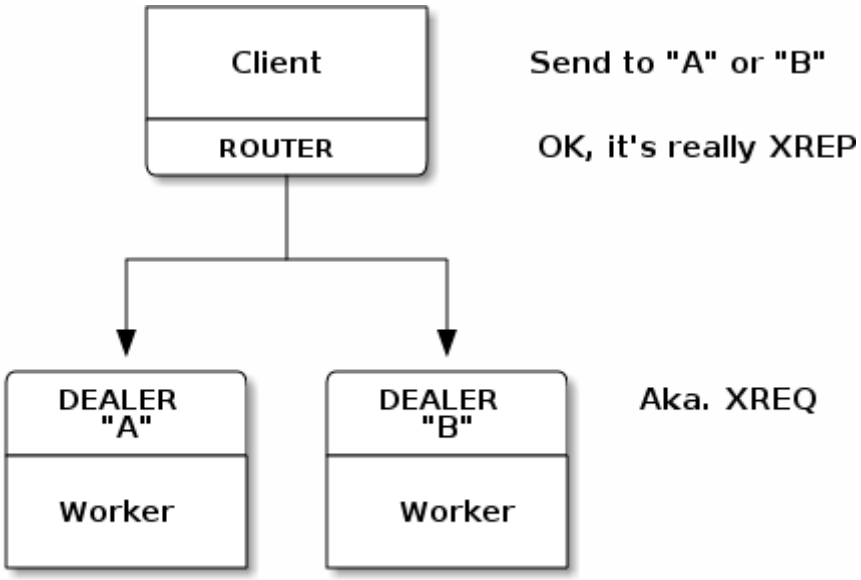

Figure 33 – Router to dealer custom routing

这的代码显示了它是如何工作的：

```
//
//  Custom routing Router to Dealer (XREP to XREQ)
//
#include "zhelpers.h"

//  We have two workers, here we copy the code, normally these
would
//  run on different boxes...
//
void *worker_a (void *context) {
    void *worker = zmq_socket (context, ZMQ_XREQ);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "A", 1);
    zmq_connect (worker, "ipc://routing.ipc");
```

```
    int total = 0;
    while (1) {
        //  We receive one part, with the workload
        char *request = s_recv (worker);
        int finished = (strcmp (request, "END") == 0);
        free (request);
        if (finished) {
            printf ("A received: %d\n", total);
            break;
        }
        total++;
    }
    return (NULL);
}

void *worker_b (void *context) {
    void *worker = zmq_socket (context, ZMQ_XREQ);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "B", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    int total = 0;
    while (1) {
        //  We receive one part, with the workload
        char *request = s_recv (worker);
        int finished = (strcmp (request, "END") == 0);
        free (request);
        if (finished) {
            printf ("B received: %d\n", total);
            break;
        }
        total++;
    }
    return (NULL);
}

int main () {
    void *context = zmq_init (1);

    void *client = zmq_socket (context, ZMQ_XREP);
    zmq_bind (client, "ipc://routing.ipc");

    pthread_t worker;
    pthread_create (&worker, NULL, worker_a, context);
```

```
    pthread_create (&worker, NULL, worker_b, context);

    //  Wait for threads to stabilize
    sleep (1);

    //  Send 10 tasks scattered to A twice as often as B
    int task_nbr;
    srandom ((unsigned) time (NULL));
    for (task_nbr = 0; task_nbr < 10; task_nbr++) {
        //  Send two message parts, first the address...
        if (within (3) > 0)
            s_sendmore (client, "A");
        else
            s_sendmore (client, "B");

        //  And then the workload
        s_send (client, "This is the workload");
    }
    s_sendmore (client, "A");
    s_send    (client, "END");

    s_sendmore (client, "B");
    s_send    (client, "END");

    sleep (1);              //  Give 0MQ/2.0.x time to flush
output
    zmq_close (client);
    zmq_term (context);
    return 0;
}
```

关于这个代码的一些意见：

- 路由套接字不知道什么时候处理套接字准备好了，如果我们的例子加入信号来解决这个问题可能会出差错。因此路由套接字使用在启动一个线程后调用了"sleep (1)"。

为了路由到一个处理套接字，我们这样创建一个封装：

路由套接字移走第一个帧，然后路由第二个帧，如处理套接字获得的一样。如果处理套接字打算应答，我们会在两部分获得相似的封装。

Figure 34 – Routing envelope for dealer

一些注意事项：如果你使用一个无效的地址，路由套接字会悄悄地丢掉消息。它不能再做什么别的事情。在通常的情况下这要么意味着节点已经撤掉，要么意味着编程的某个地方存在错误所以你使用了一个假的地址。可能将来 ØMQ 会通过一个记录总线系统报告消息的丢弃，并且能够分辨这两种不同的情况。在任何情况下你都不能假设一个消息会成功路由，除非你从目标节点获得某种应答。稍后我们将创建可靠的模型。

处理套接字看起来有点像 PULL 套接字，并且事实上它们如 PUSH 何 PULL 套接字组合一样运行。把 PULL 或者 PUSH 套接字连接到一个请求-应答套接字是非法的，并且没有意义，因此，不要这样做。

最近最少使用路由

正如我们所说的，妈妈套接字不会听你说，如果你要多嘴，它会忽略你。你必须等它们说些什么，然后你可以给一个挖苦的答案。这对于路由来说很有用因为这意味着我们可以让一组妈妈套接字等待答案。事实上当妈妈套接字准备好的时候会和我们说话。

你可以连接一个路由套接字到很多妈妈套接字，并分发消息给它们，像分发消息给处理套接字一样。妈妈套接字总是希望应答，但是他们会让你有最后的发言权。然而，一次只做一件事：

- 妈妈套接字与路由套接字谈话。
- 路由套接字响应妈妈套接字。
- 妈妈套接字与路由套接字谈话
- 路由套接字响应妈妈套接字。
- 等等

像处理套接字一样，妈妈套接字只能与一个路由套接字会话，因为妈妈套接字是通过与路由套接字会话启动的。你从来不能把妈妈套接字连接到多余一个得路由套接字，除非你在做什么鬼鬼祟祟的事情，像多通路冗余路由。我现在不会对它解释了，希望行货够狠，以免你在不必要的时候使用它。

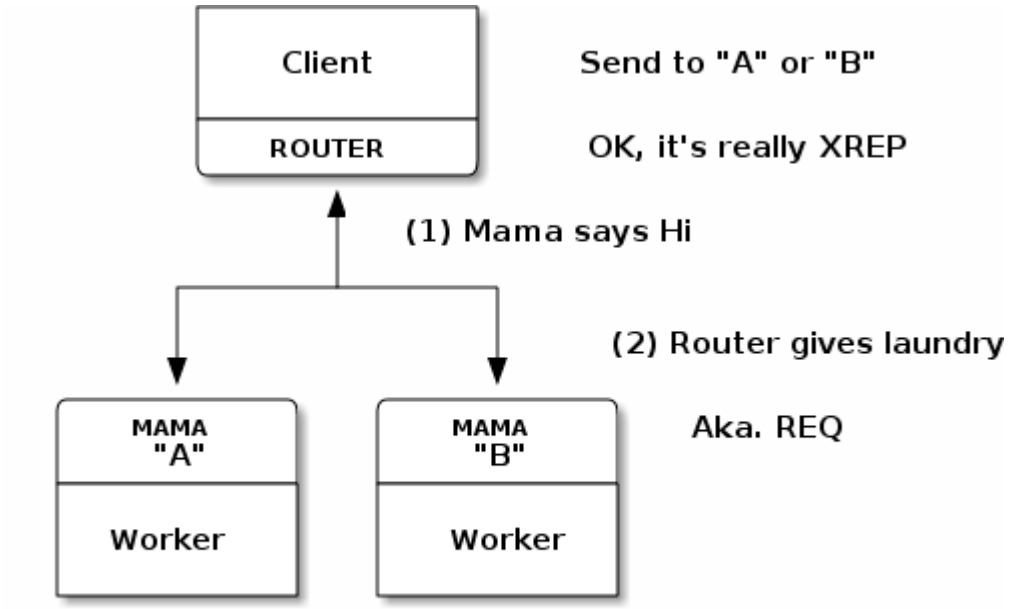

Figure 35 – Router to mama custom routing

利用路由-到-妈妈模型，你能进行什么类型的路由呢？可能最显然的答案就是“最近最少使用”（LRU），我们总是通过它路由到等待最长的妈妈。这是一个理由 LRU 路由一组妈妈的例子：

```
 //
//  Custom routing Router to Mama (XREP to REQ)
//
#include "zhelpers.h"

#define NBR_WORKERS 10

static void *
worker_thread (void *context) {
    void *worker = zmq_socket (context, ZMQ_REQ);

    //  We use a string identity for ease here
    s_set_id (worker);
    zmq_connect (worker, "ipc://routing.ipc");

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

        //  Get workload from router, until finished
        char *workload = s_recv (worker);
```

```
       int finished = (strcmp (workload, "END") == 0);
       free (workload);
       if (finished) {
          printf ("Processed: %d tasks\n", total);
          break;
       }
       total++;

       //  Do some random work
       struct timespec t;
       t.tv_sec = 0;
       t.tv_nsec = within (100000000) + 1;
       nanosleep (&t, NULL);
    }
    return (NULL);
}

int main () {
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_XREP);
    zmq_bind (client, "ipc://routing.ipc");
    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_thread,
context);
    }
    int task_nbr;
    for (task_nbr = 0; task_nbr < NBR_WORKERS * 10; task_nbr++)
{
       //  LRU worker is next waiting in queue
       char *address = s_recv (client);
       char *empty = s_recv (client);
       free (empty);
       char *ready = s_recv (client);
       free (ready);

       s_sendmore (client, address);
       s_sendmore (client, "");
       s_send (client, "This is the workload");
       free (address);
```

```
    }
    //  Now ask mamas to shut down and report their results
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS;
worker_nbr++) {
        char *address = s_recv (client);
        char *empty = s_recv (client);
        free (empty);
        char *ready = s_recv (client);
        free (ready);

        s_sendmore (client, address);
        s_sendmore (client, "");
        s_send (client, "END");
        free (address);
    }
    sleep (1);            //  Give 0MQ/2.0.x time to flush
output
    zmq_close (client);
    zmq_term (context);
    return 0;
}
```

在这个例子中，LRU 别的 ØMQ 提供给我们的（消息队列）以外的数据结构，因为我们不需要用什么东西来同步化我们的线程。一个更实际的 LRU 算法实在工作进程准备好后把它采集到一个队列，并在路由客户请求的时候使用这个队列。我们将在后面的例子中用到。

为了证明 LRU 如预料的一样工作，妈妈打印了它们各自各自做的总任务数。因为妈妈套接字做随机的工作，并且我们没有负荷均衡。我们希望每个妈妈完成大约相同的任务。这就是我们看大的结果：

```
Processed: 8 tasks
Processed: 8 tasks
Processed: 11 tasks
Processed: 7 tasks
Processed: 9 tasks
Processed: 11 tasks
Processed: 14 tasks
Processed: 11 tasks
Processed: 11 tasks
Processed: 10 tasks
```

关于这个代码的一些建议：

- 我们不需要决定时间，因为当妈妈准备好的时候会直接告诉路由器；

- 我们需要用 zhelpers.h s_set_id 函数来产生我们自己的身份，作为可打印的字符串。这样会让我们的生活简单一点。在实际的应用程序中妈妈是完全匿名的，它们会直接调用 zmq_recv(3)和 zmq_send(3)，而不是只会处理字符串的函数 zhelpers s_recv() 和 s_send()。

- 更糟的是我们使用的是随机身份。在实际的代码中请不要这样使用。随机化的持久套接字在实际应用中并不好。

- 如果你直接复制粘贴例子中的代码而不是理解它，你就应该遇到你遇到的情况。这就像你看到一个西班牙人从高空跳下然后你也这样做。

为了路由到一个妈妈，我们必须像这样创建一个妈妈套接字友好封装：

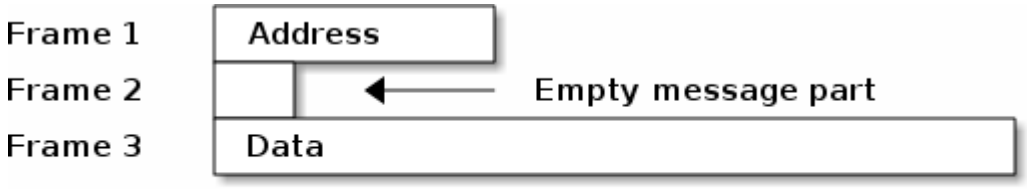

Figure 36 – Routing envelope for mama aka REQ

基于地址的路由

我们真正关心的是爸爸是回答问题的。还可以买药，当妈妈把车开到修车厂的时候他还要修车，防止隔板，在下雨天遛狗。除了这些以外，爸爸只是回答问题的。在一个经典的请求应答模型中，路由器完全不与爸爸交谈，而是让一个处理器为它做这项工作，处理器的作用是：把消息传递给随机的爸爸，并带回他们的答案。路由器更方便与妈妈交谈。后阿里，亲爱的读者，你可以停止心里分析了，这只是类比，而不是现实的故事。

关于 ØMQ 需要记住，经典的模型是工作得最好的。大路存在是有它的原因的。当你偏离大道，你就有掉下悬崖被僵尸吃掉的危险。说到这里，让我们插入一个路由器到爸爸中，看看究竟会出现什么情况。

言归正传，所有关于爸爸的事只有两件：

- 一件：他们直接支持请求-应答模型；

- 二件：他们接收任意大小的封装栈，并且完整地发回去。

在通常的请求-应答模型中，爸爸是匿名的，并且是可替代的（这样那个的类比有点吓人），但是我们在学习自主路由。因此，在我们的例子中，把请求发到爸爸 A 而不是爸爸 B 是有原因的。

ØMQ 的核心逻辑是边界小巧并且很多，中间部分大而且笨。这就是说边界可以相互寻址，也意味着我们想知道如何连接到一个已经给定的爸爸。我们随后会给出使用跨多跳路由的讨论，但是现在，我们只看最后一步：一个路由与一个特定的爸爸合作。

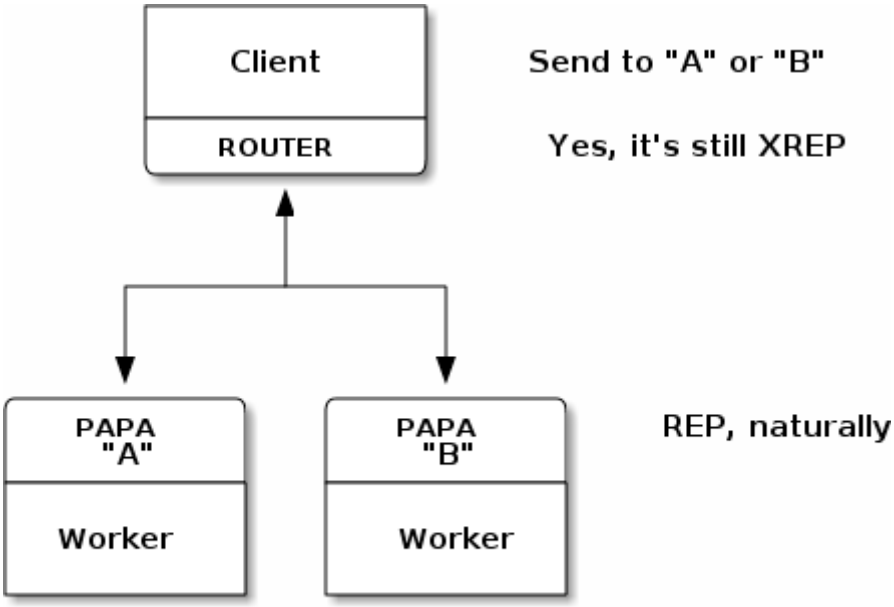

Figure 37 – Router to papa custom routing

这个例子展示了一个很特别的事件链：

- 客户机希望路由回（通过另一个套接字）某个节点。这个消息有两个地址（一个栈），一个空消息部分，和实体。
- 客户机把它传送给路由器，但是首先要专门化一个爸爸地址。
- 路由器把爸爸地址取出，用来决定消息要发送到哪里去。
- 爸爸接收地址，空消息部分和实体。
- 他把地址取出并保存，然后把实体传送给工作线程。
- 工作线程发送一个应答给爸爸。
- 右路器前置爸爸的地址，把它与剩下的地址栈，空部分和实体一起提供给客户机。

它很复杂，值得运行直到你明白。记住，爸爸是可以垃圾进，垃圾出的。

```
//
//  Custom routing Router to Papa (XREP to REP)
//
```

```
#include "zhelpers.h"

//  We will do this all in one thread to emphasize the sequence
//  of events...
int main () {
    void *context = zmq_init (1);

    void *client = zmq_socket (context, ZMQ_XREP);
    zmq_bind (client, "ipc://routing.ipc");

    void *worker = zmq_socket (context, ZMQ_REP);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "A", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    //  Wait for sockets to stabilize
    sleep (1);

    //  Send papa address, address stack, empty part, and
request
    s_sendmore (client, "A");
    s_sendmore (client, "address 3");
    s_sendmore (client, "address 2");
    s_sendmore (client, "address 1");
    s_sendmore (client, "");
    s_send    (client, "This is the workload");

    //  Worker should get just the workload
    s_dump (worker);

    //  We don't play with envelopes in the worker
    s_send (worker, "This is the reply");

    //  Now dump what we got off the XREP socket...
    s_dump (client);

    zmq_close (client);
    zmq_term (context);
    return 0;
}
```

运行这个程序，它会显示：

```
----------------------------------------
[020] This is the workload
```

```
----------------------------------------
[001] A
[009] address 3
[009] address 2
[009] address 1
[000]
[017] This is the reply
```

关于这个代码的一些意见：

- 在实际应用中我们会让爸爸套接字和路由器处于不同的节点。这个例子把它们一起放在一个线程中因为它让事件的顺序和清楚。

- zmq_connect(3)不会立即执行。爸爸套接字连接到路由器需要一定的时间，并且在后台执行。在实际的应用程序中在会话之前路由器甚至不知道爸爸套接字的存在。在我们的小例子中，我们会使用 sleep (1)：来确定连接应经建立。如果你删除了休眠语句，爸爸套接字将会收不到消息（可以试试）。

- 我们使用爸爸套接字的身份路由。为了让你确信真是这样的，可以尝试发送一个错误的地址如：“B”。爸爸套接字不会接收到这个消息。

- s_dump 和别的有用的函数（C 语言的）都是来自头文件 zhelpers.h。越来越明显，我们经常在套接字上做相同的工作，我们可以在 ØMQ 编程接口上建立新的有用的一层。当我们编写实际应用程序的时候会回到这一层来。

为了路由到爸爸套接字，我们必须这样创建一个爸爸友好的封装：

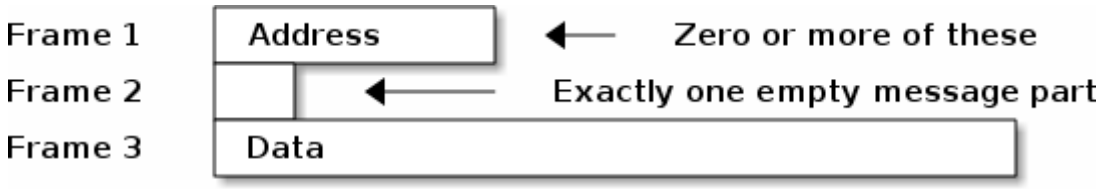

Figure 38 – Routing envelope for papa aka REP

请求-应答消息代理

我们将会将会拿回利用 ØMQ 信封做的奇怪的事情的方法，并且建立自主路由核心队列设备的核心，这样我们就能够恰当地调用一个消息代理。对于这些所有让人困惑的词语我感到抱歉。我们要构建的是一个把一组客户机绑定到一组服务器的队列设备，并允许你使用任何算法。我们要使用的是最近最少使用路由，因为它是处理负荷均衡外最明显的用例。

首先，让我们回顾经典的请求-应答模型，并看看它是如何通过一个越来越大的面向服务的网络扩展的。基础模型是：

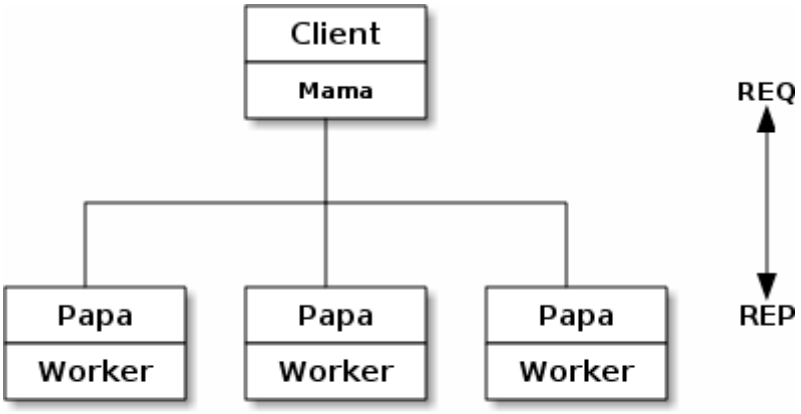

Figure 39 – Basic request-reply

它扩展到过个爸爸套接字，但是如果我们也想处理多个妈妈套接字，我们需要一个中间设备，它通常是由一个路由器和一个处理器背对背组成的，并由一个经典的的 ZMQ_QUEUE 设备连接，它会在两个套接字之间以最快的速度复制消息：

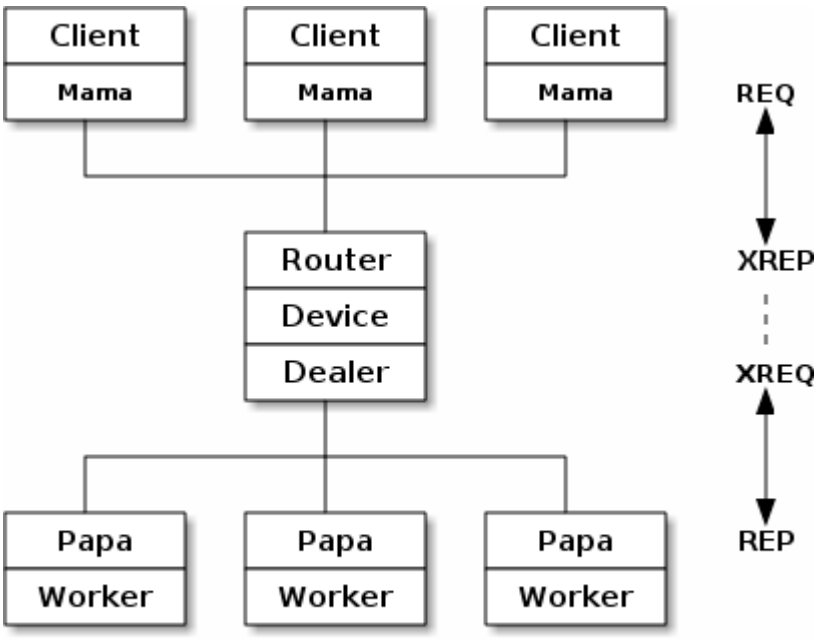

Figure 40 – Extended request-reply

关键点是，路由器在请求封装中存储原始的妈妈套接字地址，处理器和爸爸套接字都不会接触到，因此，路由器知道应该把应答返回到那个妈妈套接字。爸爸是匿名的，并且在这个模型中不会寻址它，假设所有的爸爸都提供相同的服务。

在上面的例子中我们使用处理器套接字提供的内置负荷均衡路由。然而我们希望我们的代理使用最近最少使用（LRU）算法，因此我们使用学过的路由器-妈妈套接字模型，并这样应用：

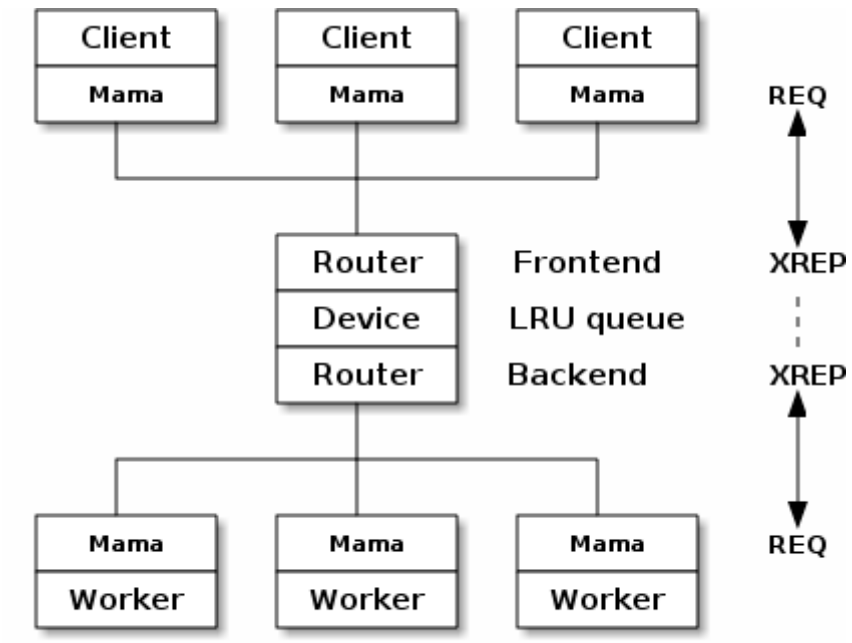

Figure 41 – Extended request-reply with LRU

我们的代理-一个路由器-到-路由器最近最少使用队列-不见简单地盲目复制消息部分。这是代码部分，它有点复杂，但是它的核心逻辑在任何希望使用最近最少使用路由的请求-应带代理中都能都重用。

```
//
//  Least-recently used (LRU) queue device
//  Clients and workers are shown here in-process
//
#include "zhelpers.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3

//  A simple dequeue operation for queue implemented as array
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) -
sizeof (q [0]))

//  Basic request-reply client using REQ socket
//
static void *
client_thread (void *context) {
    void *client = zmq_socket (context, ZMQ_REQ);
    s_set_id (client);          //  Makes tracing easier
    zmq_connect (client, "ipc://frontend.ipc");
```

```
    //  Send request, get reply
    s_send (client, "HELLO");
    char *reply = s_recv (client);
    printf ("Client: %s\n", reply);
    free (reply);
    return NULL;
}

//  Worker using REQ socket to do LRU routing
//
static void *
worker_thread (void *context) {
    void *worker = zmq_socket (context, ZMQ_REQ);
    s_set_id (worker);          //  Makes tracing easier
    zmq_connect (worker, "ipc://backend.ipc");

    //  Tell broker we're ready for work
    s_send (worker, "READY");

    while (1) {
        //  Read and save all frames until we get an empty frame
        //  In this example there is only 1 but it could be more
        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);
    }
    return NULL;
}

int main (int argc, char *argv[])
{
    //  Prepare our context and sockets
```

```
   void *context = zmq_init (1);
   void *frontend = zmq_socket (context, ZMQ_XREP);
   void *backend  = zmq_socket (context, ZMQ_XREP);
   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_thread,
context);
   }
   int worker_nbr;
   for (worker_nbr = 0; worker_nbr < NBR_WORKERS;
worker_nbr++) {
      pthread_t worker;
      pthread_create (&worker, NULL, worker_thread,
context);
   }
   //  Logic of LRU loop
   //  - Poll backend always, frontend only if 1+ worker ready
   //  - If worker replies, queue worker as ready and forward
reply
   //    to client if necessary
   //  - If client requests, pop next worker and send request
to it

   //  Queue of available workers
   int available_workers = 0;
   char *worker_queue [10];

   while (1) {
      //  Initialize poll set
      zmq_pollitem_t items [] = {
         //  Always poll for worker activity on backend
         { backend,  0, ZMQ_POLLIN, 0 },
         //  Poll front-end only if we have available
workers
         { frontend, 0, ZMQ_POLLIN, 0 }
      };
      if (available_workers)
         zmq_poll (items, 2, -1);
      else
```

```
          zmq_poll (items, 1, -1);

      //  Handle worker activity on backend
      if (items [0].revents & ZMQ_POLLIN) {
          //  Queue worker address for LRU routing
          char *worker_addr = s_recv (backend);
          assert (available_workers < NBR_WORKERS);
          worker_queue [available_workers++] = worker_addr;

          //  Second frame is empty
          char *empty = s_recv (backend);
          assert (empty [0] == 0);
          free (empty);

          //  Third frame is READY or else a client reply
address
          char *client_addr = s_recv (backend);

          //  If client reply, send rest back to frontend
          if (strcmp (client_addr, "READY") != 0) {
              empty = s_recv (backend);
              assert (empty [0] == 0);
              free (empty);
              char *reply = s_recv (backend);
              s_sendmore (frontend, client_addr);
              s_sendmore (frontend, "");
              s_send    (frontend, reply);
              free (reply);
              if (--client_nbr == 0)
                  break;     //  Exit after N messages
          }
          free (client_addr);
      }
      if (items [1].revents & ZMQ_POLLIN) {
          //  Now get next client request, route to LRU worker
          //  Client request is [address][empty][request]
          char *client_addr = 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_addr);
            s_sendmore (backend, "");
            s_send    (backend, request);

            free (client_addr);
            free (request);

            //  Dequeue and drop the next worker address
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            available_workers--;
        }
    }
    sleep (1);
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

这个程序的不同部分是每个套接字读和写的封装，和 LRU 算法。我们将顺序来说它们，线虫消息封装结构开始。

首先，重申一遍，妈妈套接字重视总加入一个空的部分（封装分界符）到发送的消息并把接收到得消息的空部分删除。这样做的原因不重要，这只是通常的请求-应答模型的一部分。这里我们关心的只是对妈妈的需要正确地处理以让妈妈高兴。其次，路由器总是郑家消息来源地地址的封装。

现在我们可以走完整个请求-应答链从客户机到服务器，然后回来。在代码中我们设置了客户机和工作线程套接字的身份，这样如果我们想打印罅隙帧的话就会简单些。让我们假设客户机的身份是“XLIENT”，并且工作线程的身份是”WORKER”.客户机发送一个单一的帧：

Figure 42 – Message that client sends

当从路由前置套接字读的时候，队列得到的是：

Figure 43 – Message coming in on frontend

代理发送到工作线程，前置从 LRU 获得的工作线程的地址，为了让妈妈高兴插入一个空的消息部分：

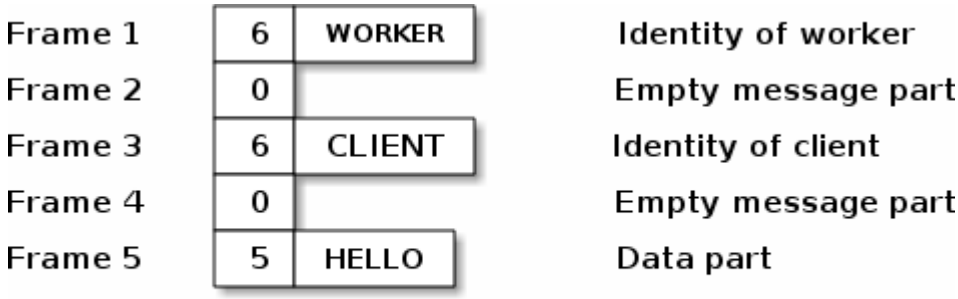

Figure 44 – Message sent to backend

复杂的封装栈首先由后向路由套接字处理，它移出第一个帧。然后在工作线程中的妈妈套接字移出消息空的消息部分，并把剩下部分提供给工作线程：

Figure 45 – Message delivered to worker

这与队列在它的前向路由套接字接收很相似。工作线程必须保存封装（它是所有的到达的部分，包括空消息部分），然后它会对数据部分进行处理。

在返回路线上消息与它们进入一样，即后向套接字以五部分提供给队列一个消息，然后队列发送给前向套接字一个三部分的消息，然后客户机获得一个整体的消息。

现在让我们看看 LRU 算法。它要求客户机和服务器都是用妈妈套接字，并且工作线程对它们接收到的消息正确的存储并重新进行封装。算法是：

- 创建一个 pollset，如果有一个或者多个可用工作线程，它会切断前向或者后向套接字。

- 进行无线超时的切断。

- 如果后向套接字有活动性，那么我们要么有一个“准备好的”消息，要么有一个客户机应答。在任何一种情况下我们都要存储 LRU 队列中的工作线程地址（第一部分），并且如果剩下部分是一个应答信号，我们通过一个前向套接字把它发送回客户机。

- 如果前向套接字有活动性，我们执行客户机请求，取出下一个工作线程（它是最近最少使用的），然后把请求发送回后向套接字。意思是发送工作线程地址，空消息部分和组成客户请求的三部分。

你现在可以看到，你能够利用基于工作线程在它原始的“准备好的“消息提供的信息的变量重新使用和扩展 LRU 算法。 例如，工作线程可以启动并进行自我性能测试，然后告诉代理它的速度。代理会选择最快的可用线程，而不是最近最少使用或者循环。

分段消息类

快速读写分段消息会很乏味并且容易出错。从我们的队列代理来看工作线程的内核：

```
 while (1) {
      //  Read and save all frames until we get an empty frame
      //  In this example there is only 1 but it could be more
      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);
   }
```

这个代码不是可重用的，因为它只能处理一个封装。并且这个代码已经对 ØMQ 接口程序进行了一些封装。如果我们小心使用原始编程接口套接字，我们就应该这样编写代码：

```
while (1) {
```

```
        //  Read and save all frames until we get an empty frame
        //  In this example there is only 1 but it could be more
        zmq_msg_t address;
        zmq_msg_init (&address);
        zmq_recv (worker, &address, 0);

        zmq_msg_t empty;
        zmq_msg_init (&empty);
        zmq_recv (worker, &empty, 0);

        //  Get request, send reply
        zmq_msg_t payload;
        zmq_msg_init (&payload);
        zmq_recv (worker, &payload, 0);
        int char_nbr;
        printf ("Worker: ");
        for (char_nbr = 0; char_nbr < zmq_msg_size (&payload);
char_nbr++)
            printf ("%c", *(char *) (zmq_msg_data (&payload)
+ char_nbr));
        printf ("\n");
        zmq_msg_init_size (&payload, 2);
        memcpy (zmq_msg_data (&payload), "OK", 2);

        zmq_send (worker, &address, ZMQ_SNDMORE);
        zmq_close (&address);
        zmq_send (worker, &empty, ZMQ_SNDMORE);
        zmq_close (&empty);
        zmq_send (worker, &payload, 0);
        zmq_close (&payload);
    }
```

我们需要的是一次能够发送和接收一个完整消息的编程接口，包括所有的封装。它允许我们用最少的代码做我们想做的事情。ØMQ 程序接收本身不是为了做这个的，但是但是没有什么阻止我们在上面进行一层封装，这样使用是一种聪明的使用 ØMQ 的方法。

设计编程接口程序最好的方法是编写测试代码，即看看编程接口在现实的代码中是怎样的。这就是我希望的工作西拿出代码：

```
  while (1) {
        zmsg_t *zmsg = zmsg_recv (worker);
        printf ("Worker: %s\n", zmsg_body (zmsg));
        zmsg_body_set (zmsg, "OK");
        zmsg_send (&zmsg, worker);
```

```
    }
```

用 4 行代码代替 22 行代码是一个好的处理方式，特别是因为结果很容易读和明白。我会使用本书中每一个本书中的 sneaky 假设，以让程序接口简略。

- 我们不关心另拷贝性能，消息有数据的副本。
- 我们总是通过调用构造函数来接收消息，因此文件哦么不需要另外的构造函数。
- 发送消息也是调用构造函数，并把消息参数置空。
- 消息部分（地址和数据）总是可打印的字符串。

最后一个比较难处理，因为 ØMQ 在产生的身份的开始使用二进制的零。在 C 语言中传送定长的数据块进出应用程序接口都很困难。因此，我们的消息类编码和解码 ØMQ 身份，以让它们能够在内部作为 C 语言字符串使用。这不是高性能设计但是它很简单，并且安全。在别的语言中，你可以编写一个编程接口套接字能够直接处理二进制消息部分而不用 hacks。

这就是这个类：

```
/*
=====================================================================
    zmsg.h

    Multipart message class for example applications.

    Follows the ZFL class conventions and is further developed
as the ZFL
    zfl_msg class.  See http://zfl.zeromq.org for more
details.



---------------------------------------------------------------------
    Copyright (c) 1991-2010 iMatix Corporation
<www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide:
http://zguide.zeromq.org
```

```
    This is free software; you can redistribute it and/or
modify it under the
    terms of the GNU Lesser General Public License as published
by the Free
    Software Foundation; either version 3 of the License, or
(at your option)
    any later version.

    This software is distributed in the hope that it will be
useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty
of MERCHANTABIL-
    ITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
General
    Public License for more details.

    You should have received a copy of the GNU Lesser General
Public License
    along with this program. If not, see
<http://www.gnu.org/licenses/>.

=============================================================
================
*/

#ifndef __ZMSG_H_INCLUDED__
#define __ZMSG_H_INCLUDED__

#include "zhelpers.h"

#ifdef __cplusplus
extern "C" {
#endif

//  Opaque class structure
typedef struct _zmsg_t zmsg_t;

//  Constructor and destructor
zmsg_t *zmsg_new     (void);
void   zmsg_destroy  (zmsg_t **self_p);

//  Receive and send message, wrapping new/destroy
zmsg_t *zmsg_recv    (void *socket);
void   zmsg_send    (zmsg_t **self, void *socket);
```

```
//  Report size of message
size_t  zmsg_parts   (zmsg_t *self);

//  Read and set message body part as C string
char   *zmsg_body    (zmsg_t *self);
void    zmsg_body_set (zmsg_t *self, char *body);
void    zmsg_body_fmt (zmsg_t *self, char *format, ...);

//  Generic push/pop message part off front
void    zmsg_push    (zmsg_t *self, char *part);
char   *zmsg_pop     (zmsg_t *self);

//  Read and set message envelopes
char   *zmsg_address  (zmsg_t *self);
void    zmsg_wrap    (zmsg_t *self, char *address, char
*delim);
char   *zmsg_unwrap   (zmsg_t *self);

//  Dump message to stderr, for debugging and tracing
void    zmsg_dump    (zmsg_t *self);

//  Selftest for the class
int     zmsg_test    (int verbose);

#ifdef __cplusplus
}
#endif

#endif

//  Pretty arbitrary limit on complexity of a message
#define ZMSG_MAX_PARTS  255

//  Structure of our class
//  We access these properties only via class methods

struct _zmsg_t {
    //  Part data follows message recv/send order
    unsigned char
         *_part_data [ZMSG_MAX_PARTS];
    size_t _part_size [ZMSG_MAX_PARTS];
    size_t _part_count;
};
```

```
//
--------------------------------------------------------------
-----------------
//  Constructor

zmsg_t *
zmsg_new (void)
{
   zmsg_t
      *self;

   self = malloc (sizeof (zmsg_t));
   memset (self, 0, sizeof (zmsg_t));
   return (self);
}

//
--------------------------------------------------------------
-----------------
//  Destructor

void
zmsg_destroy (zmsg_t **self_p)
{
   assert (self_p);
   if (*self_p) {
      zmsg_t *self = *self_p;

      //  Free message parts, if any
      while (self->_part_count)
         free (zmsg_pop (self));

      //  Free object structure
      free (self);
      *self_p = NULL;
   }
}

//
--------------------------------------------------------------
-----------------
//  Formats 17-byte UUID as 33-char string starting with '@'
//  Lets us print UUIDs as C strings and use them as addresses
```

```
//
static char *
s_encode_uuid (unsigned char *data)
{
   static char
      hex_char [] = "0123456789ABCDEF";

   assert (data [0] == 0);
   char *uuidstr = malloc (34);
   uuidstr [0] = '@';
   int byte_nbr;
   for (byte_nbr = 0; byte_nbr < 16; byte_nbr++) {
      uuidstr [byte_nbr * 2 + 1] = hex_char [data [byte_nbr
+ 1] >> 4];
      uuidstr [byte_nbr * 2 + 2] = hex_char [data [byte_nbr
+ 1] & 15];
   }
   uuidstr [33] = 0;
   return (uuidstr);
}

//
----------------------------------------------------------
-----------------
//  Formats 17-byte UUID as 33-char string starting with '@'
//  Lets us print UUIDs as C strings and use them as addresses
//
static unsigned char *
s_decode_uuid (char *uuidstr)
{
   static char
      hex_to_bin [128] = {
         -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*          */
         -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*          */
         -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*          */
          0, 1, 2, 3, 4, 5, 6, 7, 8, 9,-1,-1,-1,-1,-1,-1,    /*
0..9    */
         -1,10,11,12,13,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*   A..F    */
         -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*          */
```

```
         -1,10,11,12,13,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*   a..f    */

-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1 };  /*
*/

   assert (strlen (uuidstr) == 33);
   assert (uuidstr [0] == '@');
   unsigned char *data = malloc (17);
   int byte_nbr;
   data [0] = 0;
   for (byte_nbr = 0; byte_nbr < 16; byte_nbr++)
      data [byte_nbr + 1]
         = (hex_to_bin [uuidstr [byte_nbr * 2 + 1] & 127]
<< 4)
         + (hex_to_bin [uuidstr [byte_nbr * 2 + 2] & 127]);

   return (data);
}

//
------------------------------------------------------------
-----------------
//  Private helper function to store a single message part

static void
_set_part (zmsg_t *self, int part_nbr, unsigned char *data,
size_t size)
{
   self->_part_size [part_nbr] = size;
   self->_part_data [part_nbr] = malloc (size + 1);
   memcpy (self->_part_data [part_nbr], data, size);
   //  Convert to C string if needed
   self->_part_data [part_nbr][size] = 0;
}

//
------------------------------------------------------------
-----------------
//  Receive message from socket
//  Creates a new message and returns it
//  Blocks on recv if socket is not ready for input

zmsg_t *
```

```
zmsg_recv (void *socket)
{
   assert (socket);

   zmsg_t *self = zmsg_new ();
   while (1) {
      zmq_msg_t message;
      zmq_msg_init (&message);
      if (zmq_recv (socket, &message, 0)) {
         if (errno != ETERM)
            printf ("E: %s\n", zmq_strerror (errno));
         exit (1);
      }
      //  We handle 0MQ UUIDs as printable strings
      unsigned char *data = zmq_msg_data (&message);
      size_t       size = zmq_msg_size (&message);
      if (size == 17 && data [0] == 0) {
         //  Store message part as string uuid
         char *uuidstr = s_encode_uuid (data);
         self->_part_size [self->_part_count] = strlen
(uuidstr);
         self->_part_data [self->_part_count] = (unsigned
char *) uuidstr;
         self->_part_count++;
      }
      else
         //  Store this message part
         _set_part (self, self->_part_count++, data,
size);

      zmq_msg_close (&message);

      int64_t more;
      size_t more_size = sizeof (more);
      zmq_getsockopt (socket, ZMQ_RCVMORE, &more,
&more_size);
      if (!more)
         break;     //  Last message part
   }
   return (self);
}
```

```
//
-------------------------------------------------------------
-----------------
//  Send message to socket
//  Destroys message after sending

void
zmsg_send (zmsg_t **self_p, void *socket)
{
   assert (self_p);
   assert (*self_p);
   assert (socket);
   zmsg_t *self = *self_p;

   int part_nbr;
   for (part_nbr = 0; part_nbr < self->_part_count;
part_nbr++) {
      //  Could be improved to use zero-copy since we destroy
      //  the message parts after sending anyhow...
      zmq_msg_t message;

      //  Unmangle 0MQ identities for writing to the socket
      unsigned char *data = self->_part_data [part_nbr];
      size_t        size = self->_part_size [part_nbr];
      if (size == 33 && data [0] == '@') {
         unsigned char *uuidbin = s_decode_uuid ((char *)
data);
         zmq_msg_init_size (&message, 17);
         memcpy (zmq_msg_data (&message), uuidbin, 17);
         free (uuidbin);
      }
      else {
         zmq_msg_init_size (&message, size);
         memcpy (zmq_msg_data (&message), data, size);
      }
      int rc = zmq_send (socket, &message,
         part_nbr < self->_part_count - 1? ZMQ_SNDMORE: 0);
      assert (rc == 0);
      zmq_msg_close (&message);
   }
   zmsg_destroy (self_p);
}
```

```
//
--------------------------------------------------------------
-----------------
//  Report size of message

size_t
zmsg_parts (zmsg_t *self)
{
   return (self->_part_count);
}

//
--------------------------------------------------------------
-----------------
//  Return pointer to message body, if any
//  Caller should not modify the provided data

char *
zmsg_body (zmsg_t *self)
{
   assert (self);

   if (self->_part_count)
      return ((char *) self->_part_data [self->_part_count
- 1]);
   else
      return (NULL);
}

//
--------------------------------------------------------------
-----------------
//  Set message body as copy of provided string
//  If message is empty, creates a new message body

void
zmsg_body_set (zmsg_t *self, char *body)
{
   assert (self);
   assert (body);

   if (self->_part_count) {
      assert (self->_part_data [self->_part_count - 1]);
      free (self->_part_data [self->_part_count - 1]);
```

```
    }
    else
        self->_part_count = 1;

    _set_part (self, self->_part_count - 1, (void *) body,
strlen (body));
}

//
------------------------------------------------------------
-----------------
//  Set message body using printf format
//  If message is empty, creates a new message body
//  Hard-coded to max. 255 characters for this simplified
class

void
zmsg_body_fmt (zmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    zmsg_body_set (self, value);
}

//
------------------------------------------------------------
-----------------
//  Push message part to front of message parts

void
zmsg_push (zmsg_t *self, char *part)
{
    assert (self);
    assert (part);
    assert (self->_part_count < ZMSG_MAX_PARTS - 1);

    //  Move part stack up one element and insert new part
    memmove (&self->_part_data [1], &self->_part_data [0],
        (ZMSG_MAX_PARTS - 1) * sizeof (unsigned char *));
```

```
    memmove (&self->_part_size [1], &self->_part_size [0],
        (ZMSG_MAX_PARTS - 1) * sizeof (size_t));
    _set_part (self, 0, (void *) part, strlen (part));
    self->_part_count += 1;
}

//
--------------------------------------------------------------
-----------------
//  Pop message part off front of message parts
//  Caller should free returned string when finished with it

char *
zmsg_pop (zmsg_t *self)
{
    assert (self);
    assert (self->_part_count);

    //  Remove first part and move part stack down one element
    char *part = (char *) self->_part_data [0];
    memmove (&self->_part_data [0], &self->_part_data [1],
        (ZMSG_MAX_PARTS - 1) * sizeof (unsigned char *));
    memmove (&self->_part_size [0], &self->_part_size [1],
        (ZMSG_MAX_PARTS - 1) * sizeof (size_t));
    self->_part_count--;
    return (part);
}

//
--------------------------------------------------------------
-----------------
//  Return pointer to outer message address, if any
//  Caller should not modify the provided data

char *
zmsg_address (zmsg_t *self)
{
    assert (self);

    if (self->_part_count)
        return ((char *) self->_part_data [0]);
    else
        return (NULL);
}
```

```
//
--------------------------------------------------------
-----------------
//  Wraps message in new address envelope
//  If delim is not null, creates two-part envelope

void
zmsg_wrap (zmsg_t *self, char *address, char *delim)
{
   assert (self);
   assert (address);

   //  Push optional delimiter and then address
   if (delim)
      zmsg_push (self, delim);
   zmsg_push (self, address);
}

//
--------------------------------------------------------
-----------------
//  Unwraps outer message envelope and returns address
//  Discards empty message part after address, if any
//  Caller should free returned string when finished with it

char *
zmsg_unwrap (zmsg_t *self)
{
   assert (self);

   char *address = zmsg_pop (self);
   if (*zmsg_address (self) == 0)
      free (zmsg_pop (self));
   return (address);
}

//
--------------------------------------------------------
-----------------
//  Dump message to stderr, for debugging and tracing

void
zmsg_dump (zmsg_t *self)
```

```
{
   int part_nbr;
   for (part_nbr = 0; part_nbr < self->_part_count;
part_nbr++) {
      unsigned char *data = self->_part_data [part_nbr];
      size_t        size = self->_part_size [part_nbr];

      //  Dump the message as text or binary
      int is_text = 1;
      int char_nbr;
      for (char_nbr = 0; char_nbr < size; char_nbr++)
         if (data [char_nbr] < 32 || data [char_nbr] > 127)
            is_text = 0;

      fprintf (stderr, "[%03d] ", (int) size);
      for (char_nbr = 0; char_nbr < size; char_nbr++) {
         if (is_text)
            fprintf (stderr, "%c", data [char_nbr]);
         else
            fprintf (stderr, "%02X", (unsigned char) data
[char_nbr]);
      }
      fprintf (stderr, "\n");
   }
   fflush (stderr);
}

//
-------------------------------------------------------------
-----------------
//  Runs self test of class

int
zmsg_test (int verbose)
{
   zmsg_t
      *zmsg;
   int rc;

   printf (" * zmsg: ");

   //  Prepare our context and sockets
   void *context = zmq_init (1);
   void *output = zmq_socket (context, ZMQ_XREQ);
```

```
    rc = zmq_bind (output, "ipc://zmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zmq_socket (context, ZMQ_XREP);
    rc = zmq_connect (input, "ipc://zmsg_selftest.ipc");
    assert (rc == 0);

    //  Test send and receive of single-part message
    zmsg = zmsg_new ();
    assert (zmsg);
    zmsg_body_set (zmsg, "Hello");
    assert (strcmp (zmsg_body (zmsg), "Hello") == 0);
    zmsg_send (&zmsg, output);
    assert (zmsg == NULL);

    zmsg = zmsg_recv (input);
    assert (zmsg_parts (zmsg) == 2);
    if (verbose)
        zmsg_dump (zmsg);
    assert (strcmp (zmsg_body (zmsg), "Hello") == 0);

    //  Test send and receive of multi-part message
    zmsg = zmsg_new ();
    zmsg_body_set (zmsg, "Hello");
    zmsg_wrap    (zmsg, "address1", "");
    zmsg_wrap    (zmsg, "address2", NULL);
    assert (zmsg_parts (zmsg) == 4);
    zmsg_send (&zmsg, output);

    zmsg = zmsg_recv (input);
    if (verbose)
        zmsg_dump (zmsg);
    assert (zmsg_parts (zmsg) == 5);
    assert (strlen (zmsg_address (zmsg)) == 33);
    free (zmsg_unwrap (zmsg));
    assert (strcmp (zmsg_address (zmsg), "address2") == 0);
    zmsg_body_fmt (zmsg, "%c%s", 'W', "orld");
    zmsg_send (&zmsg, output);

    zmsg = zmsg_recv (input);
    free (zmsg_unwrap (zmsg));
    assert (zmsg_parts (zmsg) == 4);
    assert (strcmp (zmsg_body (zmsg), "World") == 0);
    char *part;
    part = zmsg_unwrap (zmsg);
```

```
    assert (strcmp (part, "address2") == 0);
    free (part);

    //  Pull off address 1, check that empty part was dropped
    part = zmsg_unwrap (zmsg);
    assert (strcmp (part, "address1") == 0);
    assert (zmsg_parts (zmsg) == 1);
    free (part);

    //  Check that message body was correctly modified
    part = zmsg_pop (zmsg);
    assert (strcmp (part, "World") == 0);
    assert (zmsg_parts (zmsg) == 0);

    zmsg_destroy (&zmsg);
    assert (zmsg == NULL);

    printf ("OK\n");
    zmq_term (context);
    return 0;
}
```

examples/C/zmsg.c

这是一个测试包装：

```
//
//  Test zmsg class
//
#include "zhelpers.h"
#include "zmsg.c"

int main (int argc, char *argv[])
{
    zmsg_test (1);
    return EXIT_SUCCESS;
}
```

建立并运行这个测试包装，它会打印出一些像这样的东西：包括显示 ØMQ 产生的套接字身份的一组消息 dumps:

```
----------------------------------------
[017] 20B60BC50DC16542C2A9AB7B01872F359D
[005] Hello
```

```
----------------------------------------
[017] 20B60BC50DC16542C2A9AB7B01872F359D
[008] address2
[008] address1
[000]
[005] Hello
 * zmsg: OK
```

这是重写的 LRU 消息代理，为了使用类 zmsg。比较利用消息工作的代码，你会发现它很简短，并且仍然会运行：

```
//
//  Least-recently used (LRU) queue device
//  Demonstrates use of the zmsg class
//
#include "zhelpers.h"
#include "zmsg.c"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3

//  A simple dequeue operation for queue implemented as array
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) -
sizeof (q [0]))

//  Basic request-reply client using REQ socket
//
static void *
client_thread (void *context) {
    void *client = zmq_socket (context, ZMQ_REQ);
    s_set_id (client);        //  Makes tracing easier
    zmq_connect (client, "ipc://frontend.ipc");

    //  Send request, get reply
    s_send (client, "HELLO");
    char *reply = s_recv (client);
    printf ("Client: %s\n", reply);
    free (reply);
    return NULL;
}

//  Worker using REQ socket to do LRU routing
//
static void *
```

```
worker_thread (void *context) {
   void *worker = zmq_socket (context, ZMQ_REQ);
   s_set_id (worker);         //  Makes tracing easier
   zmq_connect (worker, "ipc://backend.ipc");

   //  Tell broker we're ready for work
   s_send (worker, "READY");

   while (1) {
      zmsg_t *zmsg = zmsg_recv (worker);
      printf ("Worker: %s\n", zmsg_body (zmsg));
      zmsg_body_set (zmsg, "OK");
      zmsg_send (&zmsg, worker);
   }
   return NULL;
}

int main (int argc, char *argv[])
{
   //  Prepare our context and sockets
   void *context = zmq_init (1);
   void *frontend = zmq_socket (context, ZMQ_XREP);
   void *backend  = zmq_socket (context, ZMQ_XREP);
   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_thread,
context);
   }
   int worker_nbr;
   for (worker_nbr = 0; worker_nbr < NBR_WORKERS;
worker_nbr++) {
      pthread_t worker;
      pthread_create (&worker, NULL, worker_thread,
context);
   }
   //  Logic of LRU loop
   //  - Poll backend always, frontend only if 1+ worker ready
   //  - If worker replies, queue worker as ready and forward
reply
```

```
    //   to client if necessary
    //  - If client requests, pop next worker and send request
to it

    //  Queue of available workers
    int available_workers = 0;
    char *worker_queue [NBR_WORKERS];

    while (1) {
        //  Initialize poll set
        zmq_pollitem_t items [] = {
            //  Always poll for worker activity on backend
            { backend,  0, ZMQ_POLLIN, 0 },
            //  Poll front-end only if we have available
workers
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        if (available_workers)
            zmq_poll (items, 2, -1);
        else
            zmq_poll (items, 1, -1);

        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *zmsg = zmsg_recv (backend);
            //  Use worker address for LRU routing
            assert (available_workers < NBR_WORKERS);
            worker_queue [available_workers++] = zmsg_unwrap
(zmsg);

            //  Forward message to client if it's not a READY
            if (strcmp (zmsg_address (zmsg), "READY") == 0)
                zmsg_destroy (&zmsg);
            else {
                zmsg_send (&zmsg, frontend);
                if (--client_nbr == 0)
                    break;     //  Exit after N messages
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Now get next client request, route to next
worker
            zmsg_t *zmsg = zmsg_recv (frontend);
            zmsg_wrap (zmsg, worker_queue [0], "");
```

```
            zmsg_send (&zmsg, backend);

            //  Dequeue and drop the next worker address
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            available_workers--;
        }
    }
    sleep (1);
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

异步客户-服务器

在路由去到处理器的例子中，我们看到一个 1-到-N 的例子，一个服务器与多个工作线程异步地会话。我们可以把这种由上到下的模式转换成一个非常有用的 N-到-1 的结构，即，各种客户机与一个单一的服务器会话，并且异步地进行。

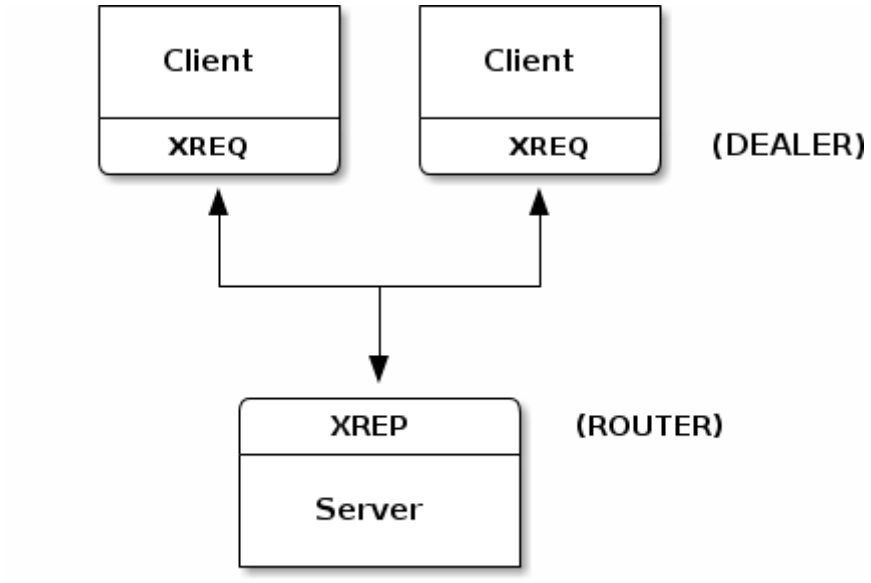

Figure 46 – Asynchronous Client Server

它的工作原理如下所示：

- 客户连接到服务器并发送请求。
- 对每个请求，服务器发送 0 到 N 个应答。
- 客户可以直接发送多个请求，而不用等到应答。
- 服务器可以发送多个应答而不用等待新的请求。

异步的意思是不必一个请求对应一个应答吗？

以下是它如何工作的代码：

```
//
//  Asynchronous client-to-server (XREQ to XREP)
//
//  While this example runs in a single process, that is just
to make
//  it easier to start and stop the example. Each task has
its own
//  context and conceptually acts as a separate process.

#include "zmsg.h"

//
---------------------------------------------------------
------------
//  This is our client task
//  It connects to the server, and then sends a request once
per second
//  It collects responses as they arrive, and it prints them
out. We will
//  run several client tasks in parallel, each with a different
random ID.

static void *
client_task (void *args)
{
   void *context = zmq_init (1);
   void *client = zmq_socket (context, ZMQ_XREQ);

   //  Generate printable identity for the client
   char identity [5];
   sprintf (identity, "%04X", randof (0x10000));
   zmq_setsockopt (client, ZMQ_IDENTITY, identity, strlen
(identity));
   zmq_connect (client, "tcp://localhost:5570");

   zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
   int request_nbr = 0;
   while (1) {
      //  Tick once per second, pulling in arriving messages
      int centitick;
      for (centitick = 0; centitick < 100; centitick++) {
         zmq_poll (items, 1, 10000);
         if (items [0].revents & ZMQ_POLLIN) {
```

```
            zmsg_t *zmsg = zmsg_recv (client);
            printf ("%s: %s\n", identity, zmsg_body
(zmsg));
            zmsg_destroy (&zmsg);
         }
      }
      zmsg_t *zmsg = zmsg_new (NULL);
      zmsg_body_fmt (zmsg, "request #%d", ++request_nbr);
      zmsg_send (&zmsg, client);
   }
   //  Clean up and end task properly
   zmq_close (client);
   zmq_term (context);
   return NULL;
}

//
----------------------------------------------------------
------------
//  This is our server task
//  It uses the multithreaded server model to deal requests
out to a pool
//  of workers and route replies back to clients. One worker
can handle
//  one request at a time but one client can talk to multiple
workers at
//  once.

static void *server_worker (void *socket);

void *server_task (void *args)
{
   void *context = zmq_init (1);

   //  Frontend socket talks to clients over TCP
   void *frontend = zmq_socket (context, ZMQ_XREP);
   zmq_bind (frontend, "tcp://*:5570");

   //  Backend socket talks to workers over inproc
   void *backend = zmq_socket (context, ZMQ_XREQ);
   zmq_bind (backend, "inproc://backend");

   //  Launch pool of worker threads, precise number is not
critical
```

```
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
        pthread_t worker_thread;
        pthread_create (&worker_thread, NULL, server_worker,
context);
    }
    //  Connect backend to frontend via a queue device
    //  We could do this:
    //      zmq_device (ZMQ_QUEUE, frontend, backend);
    //  But doing it ourselves means we can debug this more
easily

    //  Switch messages between frontend and backend
    while (1) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { backend,  0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (frontend);
            //puts ("Request from client:");
            //zmsg_dump (msg);
            zmsg_send (&msg, backend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (backend);
            //puts ("Reply from worker:");
            //zmsg_dump (msg);
            zmsg_send (&msg, frontend);
        }
    }
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return NULL;
}

//  Accept a request and reply with the same text a random
number of
//  times, with random delays between replies.
//
static void *
server_worker (void *context)
```

```
{
    void *worker = zmq_socket (context, ZMQ_XREQ);
    zmq_connect (worker, "inproc://backend");

    while (1) {
        //  The XREQ socket gives us the address envelope and
message
        zmsg_t *msg = zmsg_recv (worker);
        assert (zmsg_parts (msg) == 2);

        //  Send 0..4 replies back
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            //  Sleep for some fraction of a second
            s_sleep (randof (1000) + 1);
            zmsg_t *dup = zmsg_dup (msg);
            zmsg_send (&dup, worker);
        }
        zmsg_destroy (&msg);
    }
    zmq_close (worker);
    return NULL;
}

//  This main thread simply starts several clients, and a
server, and then
//  waits for the server to finish.
//
int main (void)
{
    s_version_assert (2, 1);

    pthread_t client_thread;
    pthread_create (&client_thread, NULL, client_task,
NULL);
    pthread_create (&client_thread, NULL, client_task,
NULL);
    pthread_create (&client_thread, NULL, client_task,
NULL);

    pthread_t server_thread;
    pthread_create (&server_thread, NULL, server_task,
NULL);
    pthread_join (server_thread, NULL);
```

```
    return 0;
}
```

让这个例子自己运行。和别的多任务例子一样，它在一个单一的进程中运行，但是每个任务有它自己的背景，并且理论上是一个分离的进程。你将会看到三个客户机（每个都有一个随机的身份），打印出它们从服务器获得的应答。仔细看你就会发现每个客户任务在每个请求后获得 0 个或者几个应答。

关于这个代码的一些建议：

- 客户机每秒发送一个请求，并且收到 0 个或者几个应答。为了让它这样运行，我们使用 zmq_poll(3)函数，我们不能简单地以一秒的超时来轮询，或者我们将要停止发送一个新的请求再我们接收到最后一个应答一秒后。因此，我们以一个高频（以每秒 100 次），它大概比较准确。这意味着服务器可以以心跳的形式作为请求，即，当服务器出现或者断开连接的时候检测。
- 这个服务器使用了一个工作线程池，每个异步地处理一个请求。它用一个内部队列把这些连接到它的前端套接字。为了帮助调试它，代码应用它自己的队列设备逻辑。在这个 C 语言代码中，你可以取消 zmsg_dump()调用来获得调试输出。

服务器中的套接字逻辑是相当有趣的。这是服务器的详细结构：

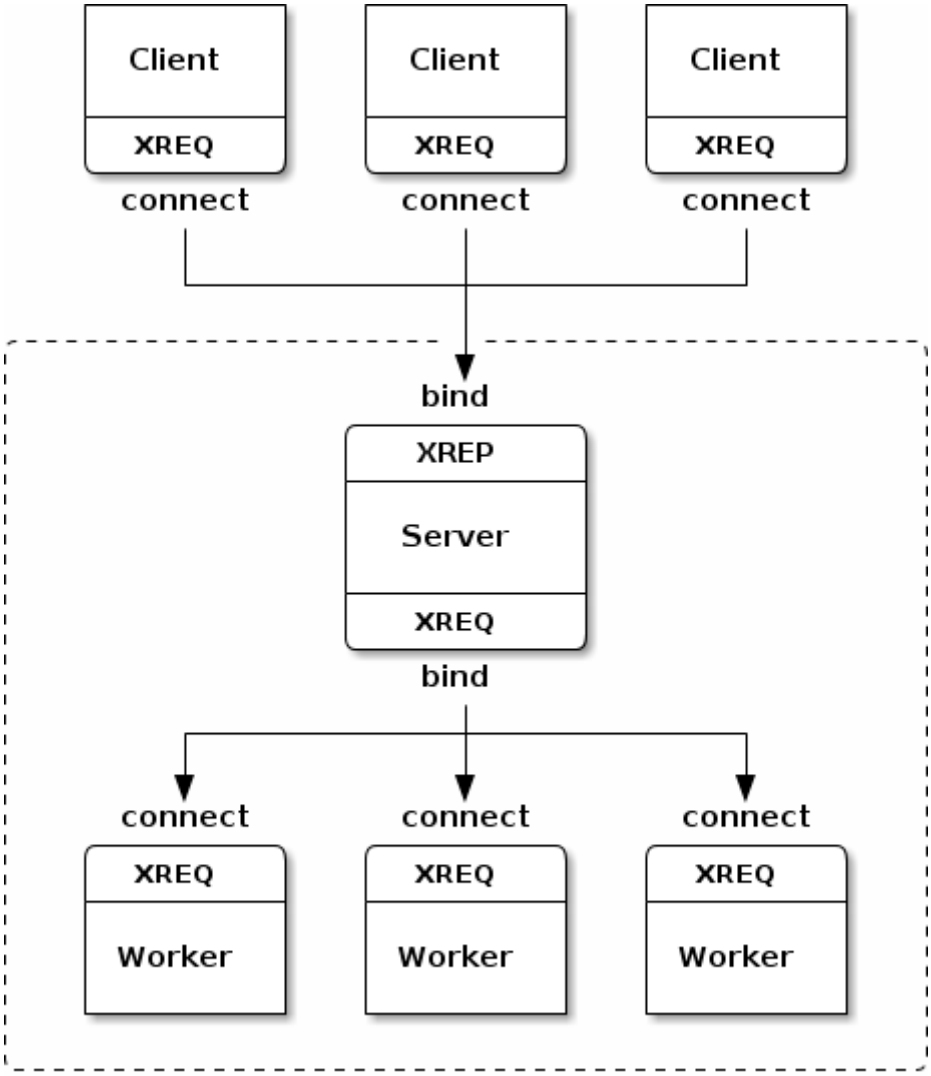

Figure 47 – Detail of async server

注意，在客户机和服务器之间我们使用的是处理器到路由器的对话，但是在内部，服务器的主线程和工作线程之间我们使用的是处理器-到-处理器的会话。如果工作线程是严格异步的，我们会使用 REP。但是因为我们希望发送多个应答我们需要一个异步套接字。我们不想路由应答，它们总是连接到发送我们请求的单一的服务器线程。

让我们考虑路由器封装。客户机发送一个简单的消息。服务器线程接收到两部分的消息（带有客户机身份的实际的消息部分）。在服务器-到-工作线程的接口间，我们有两种可能的设计：

- 工作线程获得没有带地址的消息，我们利用一个路由器套接字作为后向套接字直接处理服务器线程与工作线程之间的连接。这要求工作线程首先告诉服务器它们已经存在了，然后才能路由请求到工作线程并跟踪哪个客户机“连接”到哪个工作线程。这就是我们曾今讲到过的 LRU 模式。

- 工作线程获得带地址的消息，然后它们获得带地址的应答。这就要求工作线程能够恰当地解码和重新编码封装，但是它不需要任何别的机制。

第二种设计要简单的多，这就是我们用的：

```
   client          server      frontend      worker
  [ XREQ ]<---->[ XREP <----> XREQ <----> XREQ ]
         1 part        2 parts      2 parts
```

当你编译一个维持与客户机满状态会话的服务器，你将会碰到一个经典的问题。如果服务器与每个客户机保持一些状态，并且客户机保持进和出，事实上它会运行的很好。即使相同的客户机保持连接，如果你使用的是临时套接字（没有明确的身份），每个连接都会看起来像一个新的。

在上面的例子中我们有欺骗行为，我们以很短的时间（一个工作线程处理一个请求的时间）保持一个状态然后扔掉这个状态。但是这事实上对很多例子都不适用。

在一个满状态的异步服务器中为了恰当地管理服务器，你必须：

- 从客户机到服务器之间使用心跳。在我们的例子中我们每秒发送一个请求，它可以可靠地作为一个心跳来使用。
- 利用客户机身份作为关键字来存储状态。这适用于所有的持久和临时套接字。
- 检测一个停止的心跳。如果一个客户机没有请求，两秒钟，服务器就能够检测出来并毁坏为那个客户机保留的任何状态。

路由器-到-路由器（N-到-N）的路由

我们已经看到 XREP 套接字与处理器，爸爸，妈妈套接字会话。最后一种情况是路由器与路由器会话。它的一个用例是有过量的 HTTP 前端与一组后端工作线程会话的一个网页 farm。每个工作线程接受来自任意前端服务器的请求，并异步地处理它们，发送回一个异步应答。一个完全异步的线程有一些内部并发性，但是我们并不关心这个问题。我们关心的是 N 个工作线程是怎样与 N 各前端会话的。

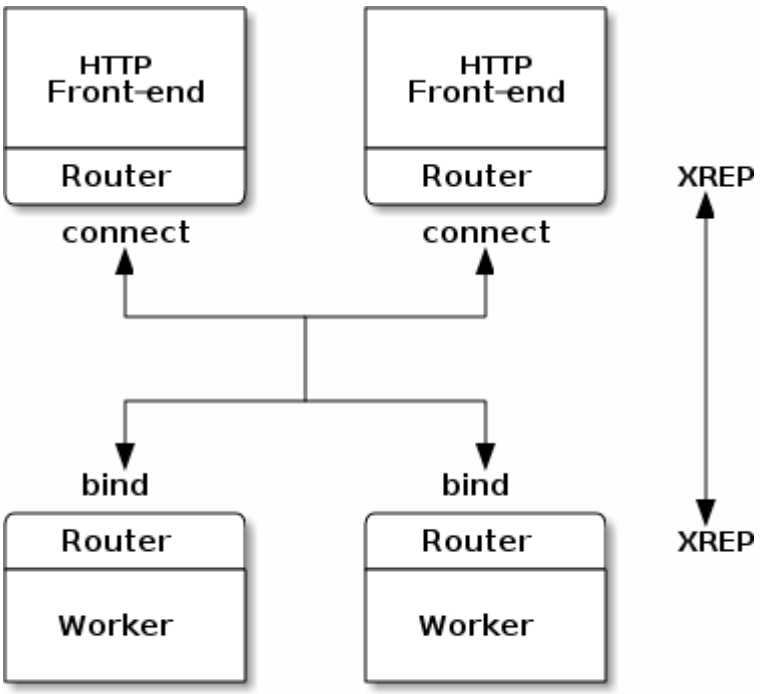

Figure 46 – N to N routing

这是一个单一的前端和单一的工作线程的例子，它们交叉连接和路由到对方：

```
//
//  Cross-connected XREP sockets addressing each other
//
#include "zhelpers.h"

int main () {
    void *context = zmq_init (1);

    void *worker = zmq_socket (context, ZMQ_XREP);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "WORKER", 6);
    zmq_bind (worker, "ipc://rtrouter.ipc");

    void *server = zmq_socket (context, ZMQ_XREP);
    zmq_setsockopt (server, ZMQ_IDENTITY, "SERVER", 6);
    zmq_connect (server, "ipc://rtrouter.ipc");

    sleep (1);
    s_sendmore (server, "WORKER");
    s_sendmore (server, "");
    s_send    (server, "send to worker");
    s_dump    (worker);
```

```
    s_sendmore (worker, "SERVER");
    s_sendmore (worker, "");
    s_send     (worker, "send to server");
    s_dump     (server);

    zmq_close (worker);
    zmq_close (server);
    zmq_term (context);
    return 0;
}
```

程序的输出是：

```
----------------------------------------
[008] SERVER
[000]
[014] send to worker
----------------------------------------
[006] WORKER
[000]
[014] send to server
```

对这个代码的一些意见：

- 这两个套接字连接和交换身份需要时间。如果你不提供时间，它们就识别不了地址，这样它们就会丢掉你发送给它们的消息。注释掉 sleep(1)然后运行看看。
- 我们可以在绑定和连接套接字上都设置和使用身份，如这个例子展示的。
- 在通过 inproc 交叉连接 XREP 的时候，现在的 ØMQ 还有个问题。绑定套接字不使用任何你设置的身份，而是仍然使用 UUID。

虽然对于异步的 N-到-N 路由，路由器-到-路由器看起来很完美，但是它有一些意想不到的错误。首先，任何 N-到-N 的连接的规模都不会超出少量的客户机和服务器。你必须创建一个中间设备，把它转换成两个一一到-N 的模型。这个结构像 LRU 队列代理，然而你需要在前段和工作线程这边使用 XREQ 来获得消息流。

第二，如果你想把两个 XREP 套接字放在同一个逻辑层的话这会变的很让人困惑。一个必须绑定，一个必须连接，并且请求-应答本来就是异步的。然而，下一点会关注这个问题。

第三，连接的一端必须知道另一端的身份。在两个异步套接字之间，你不能执行

xrep-到-xrep 流。在实际中，这意味着你需要一个名称服务器，配置数据，或者别的方法来分享一个端点的身份。它很方便，因此把流的较固定的部分当做“服务器”，并给它一个固定的，已知的身份，把动态部分当做“客户机”。客户机必须连接到服务器，然后把服务器的已知身份作为地址，向它发送一个消息，然后服务器就能够响应客户机了。

例子：内部代理路由

让我们把到现在为止学过的所有东西总结起来并放大。我们最好的客户紧急呼叫我们，让我们设计一个大的云计算设备。他对云的概念是涵盖很多数据中心，每个收拾一个有客户机和工作线程组成的集群，它们作为一个整体运行。

因为我们知道实际总是会胜过理论，我们提议用 ØMQ 进行工作仿真。我们的客户急于在他老板改变主意之前锁定预算，并且在 Twitter 上了解到很多关于 ØMQ 的情况，所以他同意了。

确定细节

几杯浓咖啡后，我们想要开始编写代码，但是一个声音告诉我们在对所有错误问题制定解决方法之前请获得更多的细节。我们问云究竟做什么工作呢？客户解释到：

- 工作线程运行在各种类型的硬件上，但是它们都能处理各种类型的任务每个集群有几百个工作线程，总共有十二个集群。
- 客户机为工作线程创建任务。每个任务是一个独立的工作单元，并且客户机想做的就是找到一个可用的工作线程，把任务尽快地发送给它。可以有很多客户机存在，并且它们可以任意地加入和撤销。
- 真正的困难是在任何时候都能加入和删除集群。集群可以带着它的客户机和任务快速加入或者离开云，。
- 如果在它们自己的集群中没有工作线程，客户的任务将会移交到这个云中别的工作线程。
- 客户机在一个时间发送一条任务，等待一个应答。如果它们在 X 秒内没有收到一个回答，它们会再次发出任务。这不是我们关注的，客户机应用程序接口已经把它处理好了。
- 工作线程一次处理一个任务，它们是很简单的东西。如果它们崩溃掉了，调用它的脚本可以重启它。

我们再次确定我们正确地理解了：

- 在集群之间有某种超级互连网络，对吗？客户说，对的。当然，我们不是白痴。

- 我们问，我们讨论的是多大的规模呢？客户回答，每个集群达到一千个客户机，每个最多每秒处理十个请求。请求和应答消息都很小，不会超过 1K 字节。

因此，我们进行一个小小的计算，我们会看到在平凡的 TCP 上，我们可以工作的很快乐。2500 客户机乘以每秒 10 个请求再乘以 1000 个字节再乘以 2 个方向等于 50 兆字节每秒或者 400 兆位每秒，对于 1G 字节的网络，这不是问题。

直接的问题是它不需要不稳定的硬件和协议，而是一些聪明的路由算法和小心地设计。我们通过设计一个集群（一个数据中心）开始，然后我们会指出如何把集群连接到一起。

单一集群的结构

工作线程和客户机是同步的。我们希望使用最近最少使用算法把任务路由到工作线程。工作线程总是平等的，我们的设备对不同的服务没有概念。工作线程是匿名的，客户机从来不会直接寻址它们。我们在这不打算为分发和重试等提供保障。

由于一些我们已经看到的原因，客户机和工作线程不会直接会话。这让动态地增加或者删除节点变得不可能。因此我们的基础模型由我们之前看到的请求-应答消息代理组成：

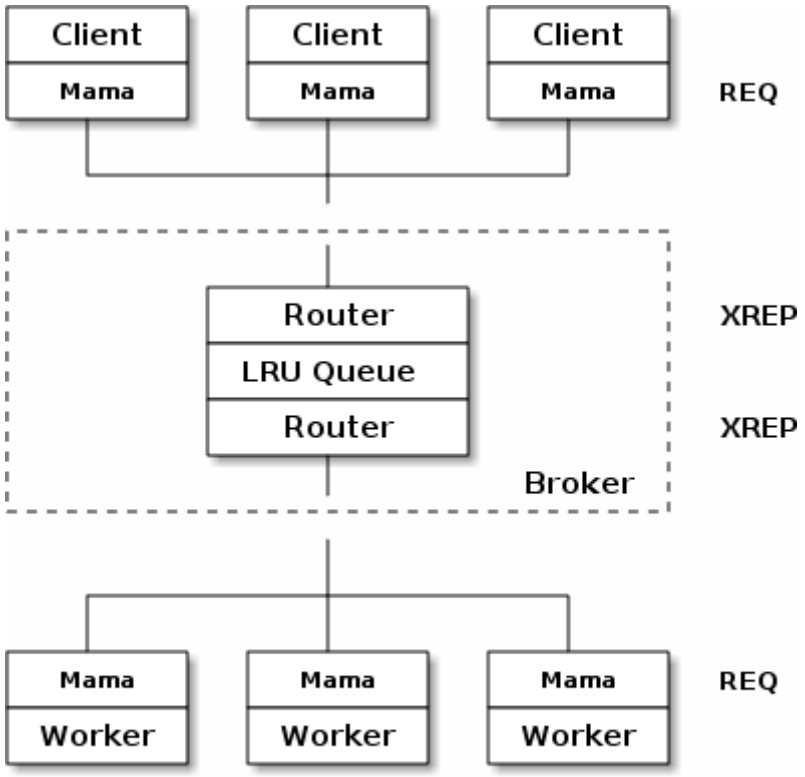

Figure 47 – Cluster architecture

升级到多集群

现在我们把它升级到不止一个集群。每个集群有一组客户机和工作线程，和一个把它们连接到一起的代理：

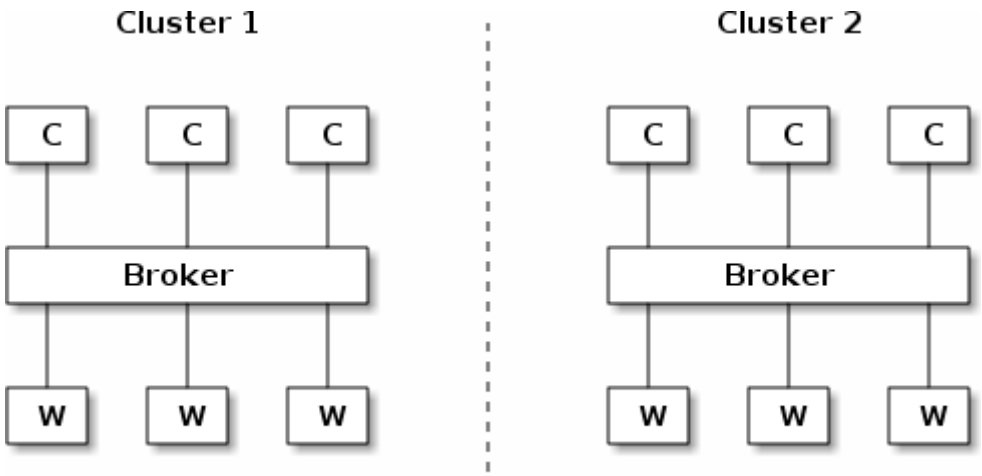

Figure 48 – Multiple clusters

问题是：我们我们如何让每个集群的客户机与另一个集群的工作线程会话？这有几个答案，各有利弊：

- 客户机能够直接连接到两个代理。它的优点是我们不需要修改工作线程或者代理。但是客户机变的更复杂，并且能感知整个拓扑。如果我们想要增加，例如第三个或者第四个集群，所有的客户机都会受到影响。事实上我们需要在客户机中加入路由是失败逻辑，这样显然不好。

- 工作线程可以直接连接到两个代理。但是妈妈工作线程不能这样做，它们只能应答一个代理。我们可能使用爸爸，但是爸爸不提供可用户化的代理-到-工作线程路由项 LRU，他只提供内置负荷均衡。这是个失败，如果我们想分发任务到空闲的工作线程：我们一定需要 LRU。一个解决方法是使用路由套接字作为工作线程节点。然我们把它标记为“方法#1”。

- 代理间可以彼此连接。这看起来是最好的，因为它创建了最少的附加连接。我们不能在空中增加集群，但是它是在范围之外的。现在客户机和工作线程也是忽略实际拓扑的，代理告诉彼此什么时候它们还有多余的能力。让我们把它标志位“方法#2”。

让我们先来研究方法#1.工作线程连接到两个代理，并从他们中的任何一个接收任务：

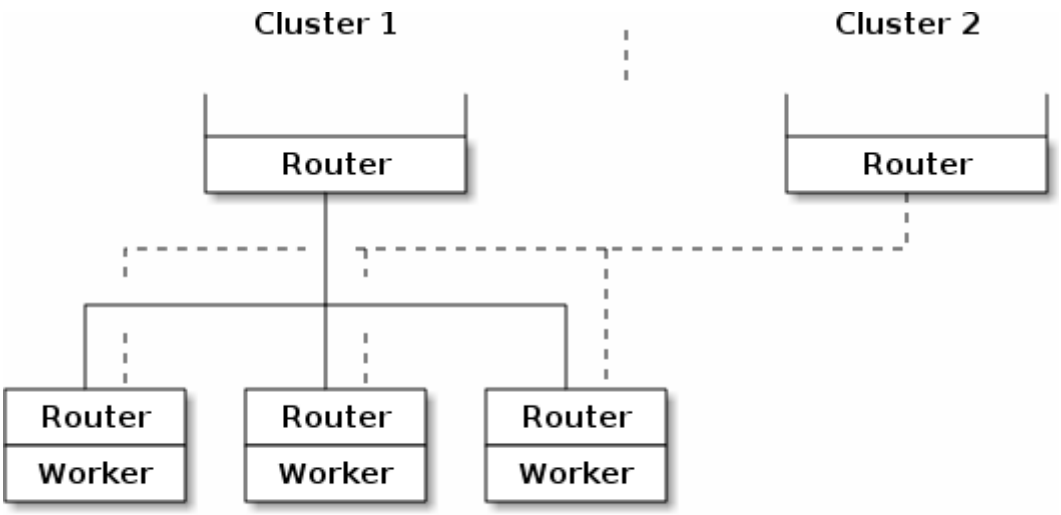

Figure 49 – Idea 1 – cross-connected workers

它看起来是可行的，但是它并没有给我们想要的，客户机由笨的可用的工作线程服务，只有在比等待更好的情况下才像远程工作线程请求服务。工作线程会对两个代理都发送“准备就绪”的信号，并且能够一次获得两个任务，当别的工作线程仍然空闲的时候。看起来设计又失败了我们把路由逻辑放一边了。

那么方法#2。我们互连代理，不触及客户机或者工作线程，这是妈妈希望我们习惯的：

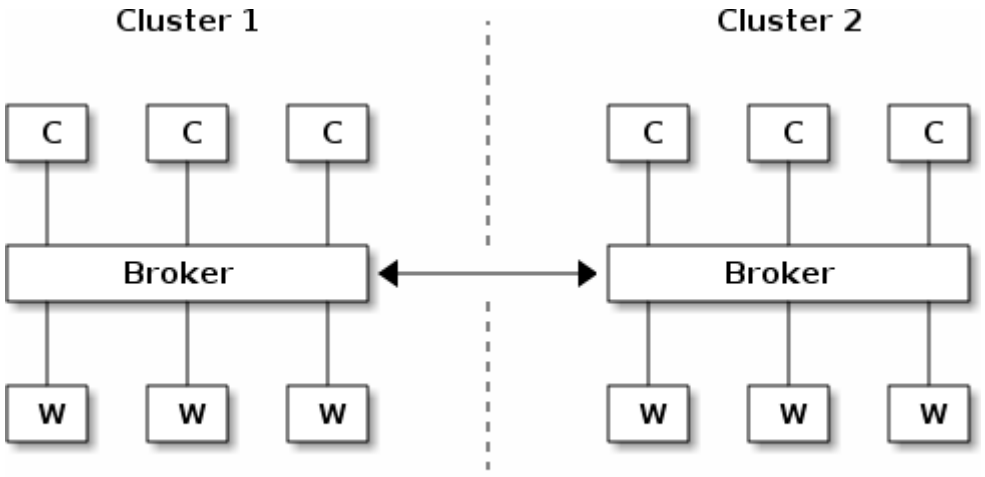

Figure 50 – Idea 2 – brokers talking to each other

这个设计是吸引人的，因为问题在一个地方得到解决了，对别的世界是不可见的。基础地，代理打开到别个代理的秘密通道，并且说悄悄话，像骆驼商一样，“嘿，我有多余能力，如果的客户太多的话给我留言，我会来处理“。

它事实上只是一个更高级的路由算法：代理编程彼此的转包商。别的都像这样设计，甚至在我们操作真实代码之前：

- 默认的是一般的情况（客户机和工作线程在同一个集群），把外部任务当做特别的情况（在集群之间移动任务）。

- 对不同类型的任务允许我们使用不同的消息流。意思是我们可以对它们进行不同的处理，例如使用不同类型的网络连接。

- 它可以很顺利地扩展。相互连接三个或者更多的代理也不会过于复杂。如果我们觉得这是一个问题，可以通过增加一个超级代理很容易地解决掉。

我们将会构造一个已经处理过的例子。我们把整个集群打包成一个进程。很明显这样不实际，但使它是仿真变得简单，并且这个仿真可以正确地扩大到实际的处理中。这就是 ØMQ 让人满意的地方，你可以在微观级设计它，然后把它扩大到宏观级。线程变成进程，机子，并且模型和逻辑仍然不变。集群过程包括客户机线程，工作线程，和一个代理线程。

我们现在对这个基础模型很了解了：

- 客户（妈妈/REQ）线程创建工作负荷并把它传送给代理（路由器/XREP）

- 工作线程（妈妈/REQ）线程处理工作负荷并把结果返回代理。

- 代理排队并用 LRU 路由模型分发负荷。

联合与互连

有几种可能的互连代理的方式。我们希望的是能够告诉别的代理，“我们还有能力”。然后接收多个任务。我们也需要能够告诉别的代理，“停止，我们满了。”它不需要完美：一些任务我们可能会接收，但是不能立即处理，我们会尽快地处理的。

最简单的互连是联合，其中代理与客户机和工作线程可以相互仿真。我们会通过连接我们的前向套接字到别的代理的后向套接字来完成这个任务。注意绑定套接字到一个终端和把它连接到另一个终端都是合法的。

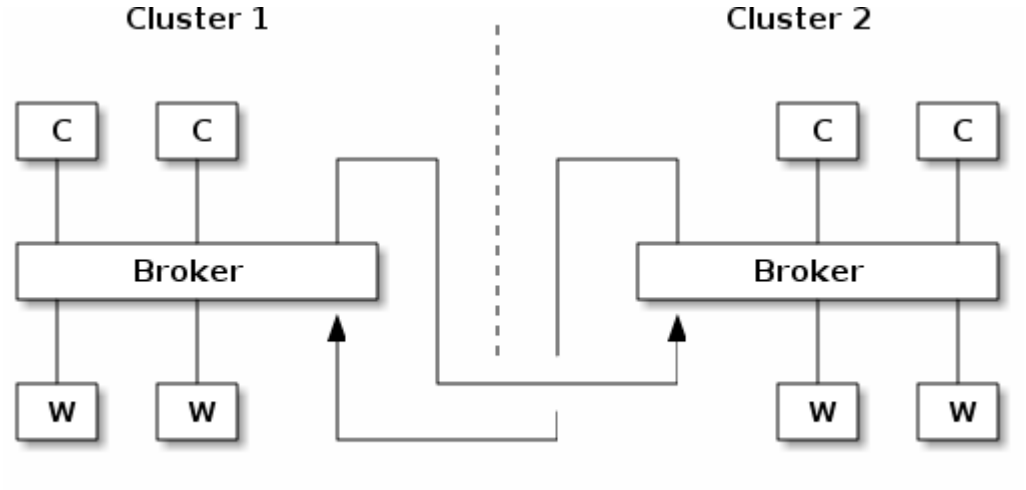

Figure 51 – Cross connected brokers in federation model

在代理和相当好的机制方面，它给我们提供了简单的逻辑：当没有客户机的时候，它

告诉代别的代理“准备好的”，然后从它接收一个任务。问题仍然是对这个问题来说太简单了。一个联合代理一次只能处理一个任务。如果代理模拟一个锁步客户机和工作线程，它也会被定义成锁步，并且如果它有很多工作线程，别的就不会被使用。我们的代理需要以完全异步的形式连接。

联合模型对别的类型的路由很完美，特别是面向服务的结构（通过服务名称和相似度路由，而不是 LRU 或者负荷均衡或者随机离散）。因此不要以为它没用而抛弃它，它只是不适合最近最少使用和符合均衡的情况。

处理联合，让我们看看互连方式。这种方式代理间可以直接相互感知，并通过秘密通道会话。假设我们想互连 N 个代理。每个代理有（N-1）个端点，并且所有代理都是用相同的逻辑和代码。在两个代理之间有完全不同的信息流：

- 每个代理在任何时候都需要告诉它的节点它有多少工作线程是可用的。这可以是一个相当简单的消息，只是一个有规律更新的数据。很明显（并正确）应该使用发布订阅套接字模型。因此每个代理打开一个 PUB 套接字，并在它上面发布状态信息，并且每个代理也打开一个 SUB 套接字，并把它连接到每个别的代理的 PUB 套接字，以从它的端点获得状态信息。

- 每个代理需要一种方式代理任务到一个端点并带异步的回应答信号。我们将利用那个路由器/路由器（XREP/XREP）套接字来完成这个任务，别的组合都不行。每个代理有两个这样的套接字：一个是为了它接收的任务，一个是为了它代理的任务。如果我们不使用两个套接字，那么需要更多工作才能知道每个时间我们是否在读一个请求或者应答。这意味着会增加更多的信息在消息封装上。

并且在一个代理和他的本地客户机和工作线程之间也有信息流。

命名仪式

三个流乘以每个流两个套接字等于我们必须在一个代理中处理六个流。好的名字很重要，它让多套接字代理在我们脑袋中一致运行。套接字做什么形成了它们名字的基础。这是关于能够在几周后的一个寒冷的周末的早上和咖啡之前能够读代码，并且还不会感觉痛苦。

让我们为套接字举行一个萨满的命名仪式。三个流是：

- 在代理和它的客户机和工作线程之间的本地请求-应答流。

- 代理与它的端点代理之间的云请求应答流。

- 代理与它的端点代理之间的状态流。

有意义的名字都是相同长度的，意味着我们的代码整齐漂亮。这看起来好像不行关，但是对细节注意可以把一般的代码转变成某种看起来像艺术的东西。

对于每个流，代理都有两个套接字我们叫做“前端”和“后端”。我们经常使用这些名字。前端接收信息或者任务。后端把它们发送到别的端点。概念上，流是从前到后的（应答走相反的方向从后到前）。

为这个指导编写的所有代码都将会使用这些套接字名：

- 本地流使用 localfe 和 locabe;
- 对于云流使用 cloudfe 和 cloudbe;
- 对于状态流使用 statefe 和 statebe.

我们所有的传输都会使用 ipc.它的优点是在连接方面和 tcp 一样（即它是非连接传输，不像 inproc）,然而我们不需要 IP 地址或者域名服务器地址，在这可能是一个烦恼，我们将使用 ipc 端点.叫做 something-local, something-cloud, 和 something-state，其中 something 是我们仿真集群的名字。

要做这么多工作来写名字。为什么不叫它们 S1,S2,S3,S4，等等，呢？答案是，如果你的大脑不是完美的机器，你就需要很多帮助来阅读代码，我们将会看到这些代码有帮助作用。记住“两个方向，三个流”比“六个不同的套接字”简单的多。

这就是代理套接字的布置：

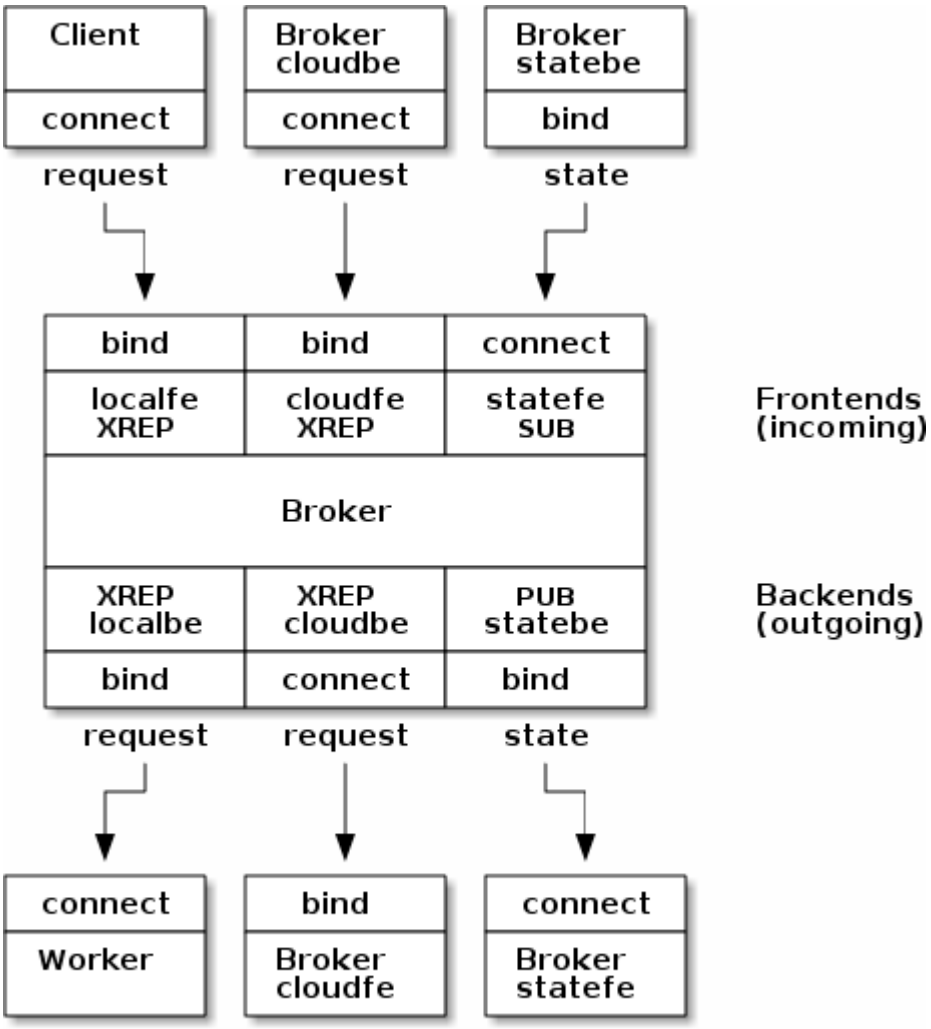

Figure 52 – Broker socket arrangement

注意我们把每个代理的 cloudbe 连接到别的任何一个 cloudfe 上，并且同样地，我们把 statebe 链接到每个别的代理的 statefe.

状态流原型

因为每个套接字有他自己容易出错的地方，我们将在实际代码中对它们一个一个地测试，而不是一次性地把所有的都放在代码中测试。当我们对每个流都满意的时候，我们可以把它们一起放在一个完整的程序中。我们将从状态流开始：

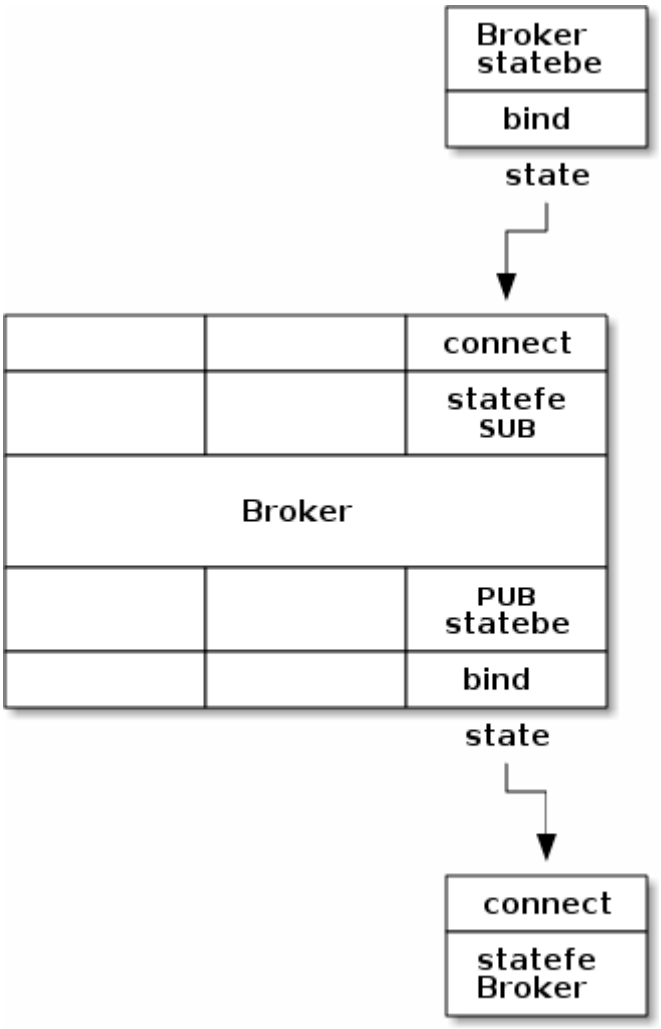

Figure 53 – The state flow

这是它的代码表示：

```
//
//  Broker peering simulation (part 1)
//  Prototypes the state flow
//
#include "zhelpers.h"
#include "zmsg.c"

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}...\n");
        exit (EXIT_FAILURE);
```

```
    }
    char *self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  Prepare our context and sockets
    void *context = zmq_init (1);
    char endpoint [256];

    //  Bind statebe to endpoint
    void *statebe = zmq_socket (context, ZMQ_PUB);
    snprintf (endpoint, 255, "ipc://%s-state.ipc", self);
    int rc = zmq_bind (statebe, endpoint);
    assert (rc == 0);

    //  Connect statefe to all peers
    void *statefe = zmq_socket (context, ZMQ_SUB);
    zmq_setsockopt (statefe, ZMQ_SUBSCRIBE, "", 0);

    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n",
peer);
        snprintf (endpoint, 255, "ipc://%s-state.ipc", peer);
        rc = zmq_connect (statefe, endpoint);
        assert (rc == 0);
    }
    //  Send out status messages to peers, and collect from
peers
    //  The zmq_poll timeout defines our own heartbeating
    //
    while (1) {
        //  Initialize poll set
        zmq_pollitem_t items [] = {
            { statefe, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll for activity, or 1 second timeout
        rc = zmq_poll (items, 1, 1000000);
        assert (rc >= 0);

        //  Handle incoming status message
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *zmsg = zmsg_recv (statefe);
```

```
            printf ("%s - %s workers free\n",
                zmsg_address (zmsg), zmsg_body (zmsg));
            zmsg_destroy (&zmsg);
        }
        else {
            //  Send random value for worker availability
            zmsg_t *zmsg = zmsg_new ();
            zmsg_body_fmt (zmsg, "%d", within (10));
            //  We stick our own address onto the envelope
            zmsg_wrap (zmsg, self, NULL);
            zmsg_send (&zmsg, statebe);
        }
    }
    //  We never get here but clean up anyhow
    zmq_close (statefe);
    zmq_term (context);
    return EXIT_SUCCESS;
}
```

关于这个代码的注意事项：

- 每个代码都有我们构建终端名字的身份。一个真实的代理需要 TCP 和一个更高级的配置方案。我们将在本书的后面介绍这个方案，但是现在，使用生成的 ipc 名字允许我们忽略在哪获得 TCP/IP 地址和名字的问题。

- 我们使用 zmq_poll(3)循环作为程序的核心。它会处理输入的消息并发送状态消息。只有我们没有获得任何输入消息的时候，我们等待一会然后发送状态消息。如果我们接收到一个输入消息就发送一个状态消息，将会导致消息风暴。

- 我们使用由发送者地址以及数据两部分组成的发布订阅消息。为了给发布者发送任务，我们需要知道它的地址，唯一的方式是把它作为发送消息的一部分。

- 我们在订阅方不会设置身份，否者在连接到运行代理的时候，我们会丢失状态消息。

- 我们在发布方不设置高水位标识，因为订阅方是临时的。我们肯能会把高水位标识设为 1，但这是没有意义的额外工作。

我们可以编译这个小程序然后运行三次来仿真三个集群。让我们把它们称作 DC1,DC2,和 DC3（名字是任意的）。我们在三个独立的窗口运行这三个程序：

```
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
```

你会看到每个集群报告它的端点状态，一会它将会每秒打印随机数字。试一试让三个代理都匹配，并同步化到每秒的状态更新。

现实中我们并不是以规定的时间间隔发送状态消息，而是当有状态改变的时候，即当一个工作线程变的可用或者不可用，才发送。它看起来有很多消息，但是状态消息很小并且我们已经建立了，集群内连接很快。

如果我们想要以确定的间隔发送状态消息，我们创建一个子线程，并在线程中打开状态套接字。然后我们从主线程发送一个不规则的状态更新到子线程并允许子线程把它们合并为规则的输出消息。它比我们在这个代码中需要更多的工作。

原型化本地和云流

现在让我们通过本地和云套接字 protopyte at 任务流。代码从客户机取得任务然后以随机的方式分配给本地工作线程和云终端：

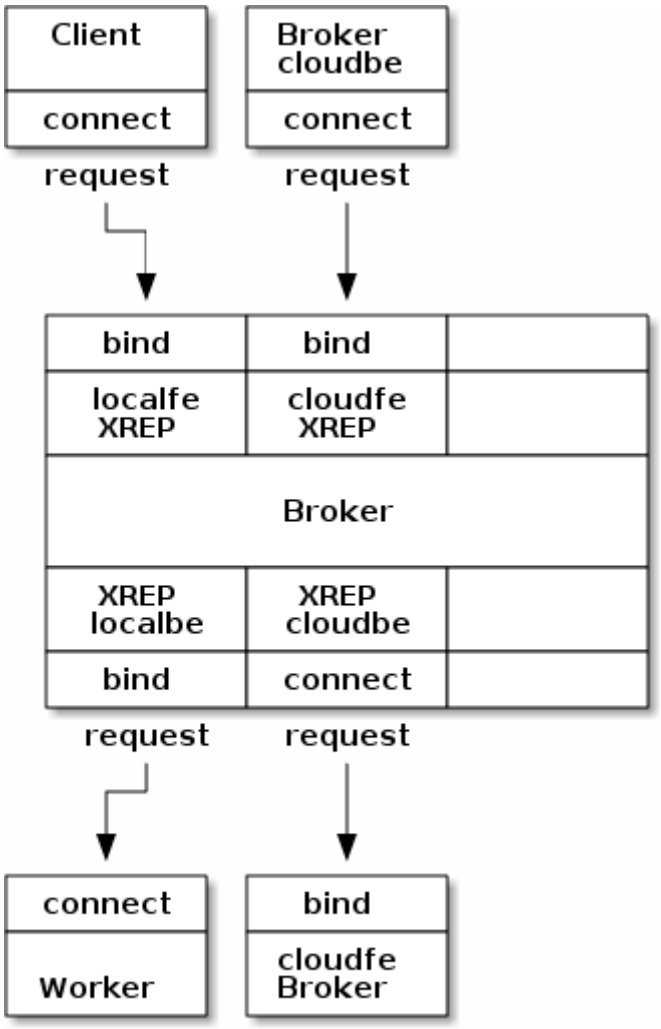

Figure 54 – The flow of tasks

在我们研究比较复杂的代码之前，我们先来描绘核心的路由逻辑，并把它简化成简单但是强壮的设计。

我们需要两个队列，一个是为了存储来自本地服务机的请求，一个是存储云客户机的请求。一种方法是从本地或者云前端获得消息，然后把它们放到可靠的队列。但是这样没有多大意思，因为 ØMQ 套接字本身就是队列。因此让我们使用 ØMQ 套接字缓冲区作为队列。

这是我们使用在 LRU 队列代理中的技术，它运行的很好。当有地方可发送请求的时候，我们只需要从两个前端套接字读取消息。我们也可以从后端套接字读取消息，因为它们给我们提供了路由回来的应答。只要后端不与我们谈话，访问前端就没有意义了。

因此，主要的循环是：

- 轮询后端。当我们获得一个消息，它可能是一个来自工作线程的“READY”信号或者它是一个应答信号。如果是应答信号，通过本地或者云前端路由回去。

- 如果一个工作线程应答，它变的可用，我们对它进行排队并计算它。

- 如果有可用的工作线程，如果有的话，处理一个来自前端并路由到本地工作线程，或者一个随机的云端节点的请求。

随机发送任务到一个终端代理而不是一个工作线程仿真跨集群的任务分发。这个方法很笨，但是对这步来说很好。

我们利用代理身份在代理之间路由消息。每个代理都有一个名字，我们在简单的原型化指令行提供的。只要它们的名称与 ØMQ 产生的 UUID 不重叠，我们就可以指出应该把一个应答路由回一个客户机或者一个代理。

这是它的工作线程代码：

```
//
//  Broker peering simulation (part 2)
//  Prototypes the request-reply flow
//
#include "zmsg.c"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3

//  A simple dequeue operation for queue implemented as array
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) -
sizeof (q [0]))

//  Request-reply client using REQ socket
//
static void *
client_thread (void *context) {
    void *client = zmq_socket (context, ZMQ_REQ);
    zmq_connect (client, "inproc://localfe");

    zmsg_t *zmsg = zmsg_new ();
    while (1) {
        //  Send request, get reply
        zmsg_body_set (zmsg, "HELLO");
        zmsg_send (&zmsg, client);
        zmsg = zmsg_recv (client);
        printf ("I: client status: %s\n", zmsg_body (zmsg));
    }
```

```
    return (NULL);
}

//  Worker using REQ socket to do LRU routing
//
static void *
worker_thread (void *context) {
    void *worker = zmq_socket (context, ZMQ_REQ);
    zmq_connect (worker, "inproc://localbe");

    //  Tell broker we're ready for work
    zmsg_t *zmsg = zmsg_new ();
    zmsg_body_set (zmsg, "READY");
    zmsg_send (&zmsg, worker);

    while (1) {
        zmsg = zmsg_recv (worker);
        //  Do some 'work'
        sleep (1);
        zmsg_body_fmt (zmsg, "OK - %04x", within (0x10000));
        zmsg_send (&zmsg, worker);
    }
    return (NULL);
}

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
        printf ("syntax: peering2 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    char *self = argv [1];
    printf ("I: preparing broker at %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  Prepare our context and sockets
    void *context = zmq_init (1);
    char endpoint [256];

    //  Bind cloud frontend to endpoint
    void *cloudfe = zmq_socket (context, ZMQ_XREP);
```

```
   snprintf (endpoint, 255, "ipc://%s-cloud.ipc", self);
   zmq_setsockopt (cloudfe, ZMQ_IDENTITY, self, strlen
(self));
   int rc = zmq_bind (cloudfe, endpoint);
   assert (rc == 0);

   //  Connect cloud backend to all peers
   void *cloudbe = zmq_socket (context, ZMQ_XREP);
   zmq_setsockopt (cloudbe, ZMQ_IDENTITY, self, strlen
(self));

   int argn;
   for (argn = 2; argn < argc; argn++) {
      char *peer = argv [argn];
      printf ("I: connecting to cloud frontend at '%s'\n",
peer);
      snprintf (endpoint, 255, "ipc://%s-cloud.ipc", peer);
      rc = zmq_connect (cloudbe, endpoint);
      assert (rc == 0);
   }

   //  Prepare local frontend and backend
   void *localfe = zmq_socket (context, ZMQ_XREP);
   zmq_bind (localfe, "inproc://localfe");
   void *localbe = zmq_socket (context, ZMQ_XREP);
   zmq_bind (localbe, "inproc://localbe");

   //  Get user to tell us when we can start...
   printf ("Press Enter when all brokers are started: ");
   getchar ();

   //  Start local workers
   int worker_nbr;
   for (worker_nbr = 0; worker_nbr < NBR_WORKERS;
worker_nbr++) {
      pthread_t worker;
      pthread_create (&worker, NULL, worker_thread,
context);
   }
   //  Start local clients
   int client_nbr;
   for (client_nbr = 0; client_nbr < NBR_CLIENTS;
client_nbr++) {
      pthread_t client;
```

```
       pthread_create (&client, NULL, client_thread,
context);
   }

   //  Interesting part
   //
-------------------------------------------------------------
----
   //  Request-reply flow
   //  - Poll backends and process local/cloud replies
   //  - While worker available, route localfe to local or
cloud

   //  Queue of available workers
   int capacity = 0;
   char *worker_queue [NBR_WORKERS];

   while (1) {
      zmq_pollitem_t backends [] = {
         { localbe, 0, ZMQ_POLLIN, 0 },
         { cloudbe, 0, ZMQ_POLLIN, 0 }
      };
      //  If we have no workers anyhow, wait indefinitely
      rc = zmq_poll (backends, 2, capacity? 1000000: -1);
      assert (rc >= 0);

      //  Handle reply from local worker
      zmsg_t *zmsg = NULL;
      if (backends [0].revents & ZMQ_POLLIN) {
         zmsg = zmsg_recv (localbe);

         assert (capacity < NBR_WORKERS);
         //  Use worker address for LRU routing
         worker_queue [capacity++] = zmsg_unwrap (zmsg);
         if (strcmp (zmsg_address (zmsg), "READY") == 0)
            zmsg_destroy (&zmsg);   //  Don't route it
      }
      //  Or handle reply from peer broker
      else
      if (backends [1].revents & ZMQ_POLLIN) {
         zmsg = zmsg_recv (cloudbe);
         //  We don't use peer broker address for anything
         free (zmsg_unwrap (zmsg));
      }
```

```
       //  Route reply to cloud if it's addressed to a broker
       for (argn = 2; zmsg && argn < argc; argn++) {
          if (strcmp (zmsg_address (zmsg), argv [argn]) ==
0)
             zmsg_send (&zmsg, cloudfe);
       }
       //  Route reply to client if we still need to
       if (zmsg)
          zmsg_send (&zmsg, localfe);

       //  Now route as many clients requests as we can handle
       //
       while (capacity) {
          zmq_pollitem_t frontends [] = {
             { localfe, 0, ZMQ_POLLIN, 0 },
             { cloudfe, 0, ZMQ_POLLIN, 0 }
          };
          rc = zmq_poll (frontends, 2, 0);
          assert (rc >= 0);
          int reroutable = 0;
          //  We'll do peer brokers first, to prevent
starvation
          if (frontends [1].revents & ZMQ_POLLIN) {
             zmsg = zmsg_recv (cloudfe);
             reroutable = 0;
          }
          else
          if (frontends [0].revents & ZMQ_POLLIN) {
             zmsg = zmsg_recv (localfe);
             reroutable = 1;
          }
          else
             break;     //  No work, go back to backends

          //  If reroutable, send to cloud 20% of the time
          //  Here we'd normally use cloud status information
          //
          if (reroutable && argc > 2 && within (5) == 0) {
             //  Route to random broker peer
             int random_peer = within (argc - 2) + 2;
             zmsg_wrap (zmsg, argv [random_peer], NULL);
             zmsg_send (&zmsg, cloudbe);
          }
          else {
```

```
            zmsg_wrap (zmsg, worker_queue [0], "");
            zmsg_send (&zmsg, localbe);

            //  Dequeue and drop the next worker address
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            capacity--;
        }
    }
    //  We never get here but clean up anyhow
    zmq_close (localbe);
    zmq_close (cloudbe);
    zmq_term (context);
    return EXIT_SUCCESS;
}
```

例如通过在两个窗口开始代理的例子：

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

关于这个代码的一些建议：

- 使用 zmsq 类让我们的生活变的简单很多，并且我们的代码更短了。很明显，任务的抽象是你作为 ØMQ 程序员的工具。

- 因为我们没有从端点获得任何状体消息，我们天真的建设它们正常运行。当你启动了所有的代理，代码会提示你去人。实际中我们不会发送任何消息给没有告诉我们它存在的代理。

把它们放在一起

让我们它们一起打成一个单一的程序包。和以前一样，我们将会运行一个完全的集群作为一个进程。我们将会使用前面两个例子，并把它们整合成一个适合你仿真任意数量集群的任务设计。

代码是两个原型一起的大小。它很适合仿真包括客户机，工作线程和云工作负荷分发的集群。这是代码：

```
//
//  Broker peering simulation (part 3)
//  Prototypes the full flow of status and tasks
```

```
//
#include "zhelpers.h"
#include "zmsg.c"

#define NBR_CLIENTS 10
#define NBR_WORKERS 5

//  A simple dequeue operation for queue implemented as array
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) -
sizeof (q [0]))

//  Request-reply client using REQ socket
//  To simulate load, clients issue a burst of requests and
then
//  sleep for a random period.
//
static void *
client_thread (void *context) {
   int rc;
   void *client = zmq_socket (context, ZMQ_REQ);
   rc = zmq_connect (client, "inproc://localfe");
   assert (rc == 0);
   void *monitor = zmq_socket (context, ZMQ_PUSH);
   rc = zmq_connect (monitor, "inproc://monitor");
   assert (rc == 0);

   zmsg_t *zmsg = zmsg_new ();
   while (1) {
      sleep (within (5));

      int burst = within (15);
      while (burst--) {
         //  Send request with random hex ID
         char task_id [5];
         sprintf (task_id, "%04X", within (0x10000));
         zmsg_body_set (zmsg, task_id);
         zmsg_send (&zmsg, client);

         //  Wait max ten seconds for a reply, then complain
         zmq_pollitem_t pollset [1] = {
            { client, 0, ZMQ_POLLIN, 0 }
         };
         rc = zmq_poll (pollset, 1, 10 * 1000000);
         assert (rc >= 0);
```

```
          if (pollset [0].revents & ZMQ_POLLIN) {
             zmsg = zmsg_recv (client);
             //  Worker is supposed to answer us with our task
id
             assert (strcmp (zmsg_body (zmsg), task_id) ==
0);
          }
          else {
             zmsg = zmsg_new ();
             zmsg_body_fmt (zmsg,
                "E: CLIENT EXIT - lost task %s", task_id);
             zmsg_send (&zmsg, monitor);
             return (NULL);
          }
       }
    }
    return (NULL);
}

//  Worker using REQ socket to do LRU routing
//
static void *
worker_thread (void *context) {
    void *worker = zmq_socket (context, ZMQ_REQ);
    int rc = zmq_connect (worker, "inproc://localbe");
    assert (rc == 0);

    //  Tell broker we're ready for work
    zmsg_t *zmsg = zmsg_new ();
    zmsg_body_set (zmsg, "READY");
    zmsg_send (&zmsg, worker);

    while (1) {
       //  Workers are busy for 0/1/2 seconds
       zmsg = zmsg_recv (worker);
       sleep (within (2));
       zmsg_send (&zmsg, worker);
    }
    return (NULL);
}

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
```

```
   //  Other arguments are our peers' names
   //
   s_version ();
   if (argc < 2) {
      printf ("syntax: peering3 me {you}...\n");
      exit (EXIT_FAILURE);
   }
   char *self = argv [1];
   printf ("I: preparing broker at %s...\n", self);
   srandom ((unsigned) time (NULL));

   //  Prepare our context and sockets
   void *context = zmq_init (1);
   char endpoint [256];

   //  Bind cloud frontend to endpoint
   void *cloudfe = zmq_socket (context, ZMQ_XREP);
   snprintf (endpoint, 255, "ipc://%s-cloud.ipc", self);
   zmq_setsockopt (cloudfe, ZMQ_IDENTITY, self, strlen
(self));
   int rc = zmq_bind (cloudfe, endpoint);
   assert (rc == 0);

   //  Bind state backend / publisher to endpoint
   void *statebe = zmq_socket (context, ZMQ_PUB);
   snprintf (endpoint, 255, "ipc://%s-state.ipc", self);
   rc = zmq_bind (statebe, endpoint);
   assert (rc == 0);

   //  Connect cloud backend to all peers
   void *cloudbe = zmq_socket (context, ZMQ_XREP);
   zmq_setsockopt (cloudbe, ZMQ_IDENTITY, self, strlen
(self));

   int argn;
   for (argn = 2; argn < argc; argn++) {
      char *peer = argv [argn];
      printf ("I: connecting to cloud frontend at '%s'\n",
peer);
      snprintf (endpoint, 255, "ipc://%s-cloud.ipc", peer);
      rc = zmq_connect (cloudbe, endpoint);
      assert (rc == 0);
   }
```

```
    //  Connect statefe to all peers
    void *statefe = zmq_socket (context, ZMQ_SUB);
    zmq_setsockopt (statefe, ZMQ_SUBSCRIBE, "", 0);

    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n",
peer);
        snprintf (endpoint, 255, "ipc://%s-state.ipc", peer);
        rc = zmq_connect (statefe, endpoint);
        assert (rc == 0);
    }
    //  Prepare local frontend and backend
    void *localfe = zmq_socket (context, ZMQ_XREP);
    zmq_bind (localfe, "inproc://localfe");
    void *localbe = zmq_socket (context, ZMQ_XREP);
    zmq_bind (localbe, "inproc://localbe");

    //  Prepare monitor socket
    void *monitor = zmq_socket (context, ZMQ_PULL);
    zmq_bind (monitor, "inproc://monitor");

    //  Start local workers
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS;
worker_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_thread,
context);
    }
    //  Start local clients
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS;
client_nbr++) {
        pthread_t client;
        pthread_create (&client, NULL, client_thread,
context);
    }

    //  Interesting part
    //
-----------------------------------------------------------
----
    //  Publish-subscribe flow
```

```
   //  - Poll statefe and process capacity updates
   //  - Each time capacity changes, broadcast new value
   //  Request-reply flow
   //  - Poll primary and process local/cloud replies
   //  - While worker available, route localfe to local or
cloud

   //  Queue of available workers
   int local_capacity = 0;
   int cloud_capacity = 0;
   char *worker_queue [10];

   while (1) {
      zmq_pollitem_t primary [] = {
         { localbe, 0, ZMQ_POLLIN, 0 },
         { cloudbe, 0, ZMQ_POLLIN, 0 },
         { statefe, 0, ZMQ_POLLIN, 0 },
         { monitor, 0, ZMQ_POLLIN, 0 }
      };
      //  If we have no workers anyhow, wait indefinitely
      rc = zmq_poll (primary, 4, local_capacity? 1000000:
-1);
      assert (rc >= 0);

      //  Track if capacity changes during this iteration
      int previous = local_capacity;

      //  Handle reply from local worker
      zmsg_t *zmsg = NULL;

      if (primary [0].revents & ZMQ_POLLIN) {
         assert (local_capacity < NBR_WORKERS);
         //  Use worker address for LRU routing
         zmsg = zmsg_recv (localbe);
         worker_queue [local_capacity++] = zmsg_unwrap
(zmsg);
         if (strcmp (zmsg_address (zmsg), "READY") == 0)
            zmsg_destroy (&zmsg);   //  Don't route it
      }
      //  Or handle reply from peer broker
      else
      if (primary [1].revents & ZMQ_POLLIN) {
         zmsg = zmsg_recv (cloudbe);
         //  We don't use peer broker address for anything
```

```
            free (zmsg_unwrap (zmsg));
        }
        //  Route reply to cloud if it's addressed to a broker
        for (argn = 2; zmsg && argn < argc; argn++) {
            if (strcmp (zmsg_address (zmsg), argv [argn]) ==
0)
                zmsg_send (&zmsg, cloudfe);
        }
        //  Route reply to client if we still need to
        if (zmsg)
            zmsg_send (&zmsg, localfe);

        //  Handle capacity updates
        if (primary [2].revents & ZMQ_POLLIN) {
            zmsg = zmsg_recv (statefe);
            cloud_capacity = atoi (zmsg_body (zmsg));
            zmsg_destroy (&zmsg);
        }
        //  Handle monitor message
        if (primary [3].revents & ZMQ_POLLIN) {
            zmsg_t *zmsg = zmsg_recv (monitor);
            printf ("%s\n", zmsg_body (zmsg));
            zmsg_destroy (&zmsg);
        }

        //  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
        //  - Route any request locally if we can, else to cloud
        //
        while (local_capacity + cloud_capacity) {
            zmq_pollitem_t secondary [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            if (local_capacity)
                rc = zmq_poll (secondary, 2, 0);
            else
                rc = zmq_poll (secondary, 1, 0);
            assert (rc >= 0);

            if (secondary [0].revents & ZMQ_POLLIN)
```

```
            zmsg = zmsg_recv (localfe);
         else
         if (secondary [1].revents & ZMQ_POLLIN)
            zmsg = zmsg_recv (cloudfe);
         else
            break;      //  No work, go back to primary

         if (local_capacity) {
            zmsg_wrap (zmsg, worker_queue [0], "");
            zmsg_send (&zmsg, localbe);

            //  Dequeue and drop the next worker address
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            local_capacity--;
         }
         else {
            //  Route to random broker peer
            printf ("I: route request %s to cloud...\n",
                zmsg_body (zmsg));
            int random_peer = within (argc - 2) + 2;
            zmsg_wrap (zmsg, argv [random_peer], NULL);
            zmsg_send (&zmsg, cloudbe);
         }
      }
      if (local_capacity != previous) {
         //  Broadcast new capacity
         zmsg_t *zmsg = zmsg_new ();
         zmsg_body_fmt (zmsg, "%d", local_capacity);
         //  We stick our own address onto the envelope
         zmsg_wrap (zmsg, self, NULL);
         zmsg_send (&zmsg, statebe);
      }
   }
   //  We never get here but clean up anyhow
   zmq_close (localbe);
   zmq_close (cloudbe);
   zmq_close (statefe);
   zmq_close (monitor);
   zmq_term (context);
   return EXIT_SUCCESS;
}
```

它是不平凡的程序，并且需要一天才能工作。这是重点：

- 客户现场检查和报告一个错误的请求。它们通过轮询一个响应来完成这个任务，并且如果在议会（10 秒）后没有消息到达，它将会打印一个错误消息。

- 客户机线程不会直接打印，，而是发送一个消息到“监听”套接字（PUSH）这样主循环采集（PULL）并打印。这就是我们看到的使用 ØMQ 套接字监听和记录的第一种情况；这是一个很大的用例，我们还会回到它上面来。

- 客户机仿真不同的负荷以达到集群百分百咋随机的状态，以至于任务在云上转移。客户机和工作线程的数量在客户机延迟，并由工作线程控制。随时处理处理它们，看看你能否做出更实用的仿真。

- 首先绑定所有的 ipc 终端，然后连接它们。这是进程内传输必须的。

- 主要的循环使用两个 pollset。它事实上可以使用三个：信心，后端和前端。在更早的原型中，如果有后端能力的话，使用一个前端消息就没有意义了。

在开发这个程序的时候遇到一些问题：

- 如果请求或者应答丢失的话，客户机会冻结。再次重申，ØMQ 的 XREP/ 路由器套接字会丢掉它不能路由的消息。第一种方法是修改客户机线程来检测和报告这个问题。第二种是在主循环中每个 recv() 函数后，send() 函数前调用 zmsq_dump()，知道发现问题是什么。

- 主循环错误地从不止一个准备好的套接字读取数据这导致了第一个消息丢失。这个问题通过只从第一个准备好的消息读数据改正过来了。

- Zmsq 类不能合适地把 UUID 解码成 C 语言串。这就导致了 UUID 因为只包含零个字节而损坏了。这个问题通过把 UUID 解码成可打印的文件字符串修正过来了。

这个方正不检测云终端的消失。如果你开始了几个终端，并终止其中的一个正向别的报告容量的终端，它们将会继续给它发送任务，即使它已经撤离。你可以试试，客户机会向你抱怨丢失了请求。解决方法是双重的：首先，只保存性能消息一个很短的时间，这样，如果一个终端消失了，它的性能很快设置为零。跌停，为请求-应答链增加可靠性。在下一章，我们将会看到两种情况。

第四章 可靠性

在第三章中我们看到了请求-应答模式的高级应用，并且看到了工作例子。在这章中我们将会看看这个常见的可靠性的问题，并且在 0MQ 核心请求应答模型中编译一系列的可靠消息模型。

我们会涉及到：

- 在 0MQ 应用程序中如何定义“可靠性”。
- 在 0MQ 核模型上怎样实现可靠性。
- 在 0MQ 端点之间怎样实现心跳。
- 怎样写一个可重用的视频说明。
- 怎样设计一个面向服务器结构的 API.
- 怎样实现服务器发现。
- 怎样实现非等幂服务器应用程序。
- 怎样实现基于磁盘的可靠性。

在这一章中我们主要集中于使用者‘模式‘，它是一种可重用模式，可以帮助你设计你的 0MQ 结构：

- 懒海盗模式：来自客户端得可靠的请求应答。
- 简单的海盗模式：利用 LRU 队列的可靠的请求应答。
- 妄想者海盗模式：利用心跳的可靠的请求应答模式。
- 大管家模式：面向服务器的可靠排队。
- 泰坦尼克模式：基于磁盘/非连接可靠排队。
- 自由模式：无代理的可靠请求应答。

什么是“可靠性”

为了明白什么是“可靠性”，我们必须看看它的反义词，叫做*失败*。如果我们能够处理一系列确定的失败，那么对于这些失败我们是可靠的。没有更多，没有更少。让我们看看在一个分布式 0MQ 应用程序中导致失败的可能原因，按可能性递减的顺序：

- 应用程序是最差的犯规者。它能够崩溃和退出，冻结和阻止输入的反应，对于它的输入运行的很慢，消耗所有的内存，等等。
- 我们用 0MQ 编写的系统类码代理容易死亡。系统代码应该比应用程序代码更加可靠，但是也能够崩溃和燃烧，并且如果它试图补偿满客户的话会用完内存。
- 消息队列可能溢出，特别是在它尝试野蛮地处理慢客户的时候。当一个队列溢出，它会扔掉消息。
- 网络会暂时地失败，导致间歇的消息丢失。这样的错误隐藏在 0MQ 应用程序中，因为在一个网络导致的连接断开后，它自动地再连接端点。
- 硬件可能失败，并导致所有连接在这个机子上的所有进程都失败。
- 网络可能以异常的方式失败，即，一个开关上的一些端点可能死亡并且网络的这些部分变得不可访问。

让一个软件系统面对所有这些可能的失败变得完全可靠是一个很大的困难，并且是一个昂贵的任务，超出了这个中等指导的范围。

因为头五个覆盖了大公司外的 99.9%的实际请求（依据我刚进行的高的科学研究），那就是我们将会研究的。如果你是一个大公司，有钱花在最后两个例子中，尽快联系我，

在我的海滨房子的后面有一个很大的洞，等待着修建成一个水池。

可靠性设计

因此，为了让事情变得非常的简单，可靠性能够在代码冻结或者崩溃的时候让程序仍然正常运行，一种我们会缩短到死的状态。然而，我们希望持续正常运行的代码比消息更复杂。我们需要查看每个 0MQ 消息模式并看看怎样让他工作（如果我们能够），即使在代码死亡的时候。

让我们一个一个地看：

- 请求-应答：如果服务器死了（正在处理一个请求的时候），客户机能够指出它，因为它没有获得一个应答。然后它会生气地放弃，等待一会然后再试，发现另一个服务器，等等。如果客户机死掉，我们可以把它作为别人的问题而不去理睬。
- 发布-订阅：如果客户机死掉（已经接受到一些数据），服务器不知道它。发布订阅不会从客户机发送任何消息回服务器。但是客户机能够通过不同的途径联系到服务器，即，通过请求应答问，“能再次发送我丢失的消息吗？”如果服务器死掉，这种情况超出了这讨论的范围。订阅方也能自己核对它们没有运行的很慢，并采取行动（即，通知操作系统并且死掉）如果它们是的话。
- 管道：如果一个工作线程死掉（当在工作的时候），风扇不知道这个事。管道，如发布订阅一样，时间档只在一个方向上运行。但是下游的采集器能够检测到一个任务没有处理，并发送一个消息回风扇，说“再发送一遍任务 324”如果这个风扇或者采集器死掉，任何上有发送任务的客户机都会不愿等待并再发送所有的。它不优雅，但是系统代码对于问题不应该死的太平凡。

在这章中，我们集中讲请求-应答模式，我们也会涉及到可靠的发布订阅和管道，在接下来的章节中。

基础的请求应答模式（一个 REQ 客户机套接字与一个 REP 套接字实现一个阻塞的发送/接收）来处理最平凡类型的失败。如果在处理请求的时候服务器崩溃掉了，客户机会永远挂起。如果是网络丢掉了请求或者应答，客户机也会永远挂起。

由于 0MQ 在端点之间能够自动再连接，负荷均衡消息，等一些原因，0MQ 比 TCP 优秀的多。但是对于实际任务，它还是不够好。唯一能够信任 0MQ 的例子是一个进程中两个线程之间的请求应答模式，没有会导致死亡的网络或者分离的服务器进程。

然而，再做一点工作，这个谦虚的模式对于一个跨分布式网络的真实任务变成一个很好的基础，我们得到一个很好的请求应答模式我叫做“海盗“模式。RRR！

这里有三种方式连接客户机到服务器，每一个需要一种专门的方法来带到可靠的目的：

- 对个客户机直接地与一个客户机会话。用例：客户机要回话的是一个众所周知的服务器。我们需要努力处理的失败的类型是：服务器崩溃和重启，网络连接断开。
- 对个客户机与一个单一的队列设备会话，这个队列设备会分发任务到多个服务器。

用例：工作量分发到工作线程。我们努力处理的失败类型：工作线程绷紧或者重启，工作线程忙循环，工作线程超负荷，队列崩溃和重启，网络连接断开。

- 多个客户机与多个服务器会话而没有中间设备。用例：分发服务，如名字方案。我们努力解决的失败是：服务器崩溃和重启，服务器忙循环，服务器超负荷，网络连接断开。

每个都有它自己的协议，并且你常常会混合它们。这三个我们都会详细看。

客户边的可靠性（懒海盗模式）

只要在客户机做一些改变，我们就可以获得间的可靠的请求应答。我们调用懒海盗模式。而不是使用阻塞的接收，我们:

- 我们轮询 REQ 套接字，并且只有当它确定一个应答到达的时候才从它接收。
- 如果在截至时间范围内没有应答到达，再发几次这个请求。
- 如果在几个请求后还是没有应答，禁止这个任务。

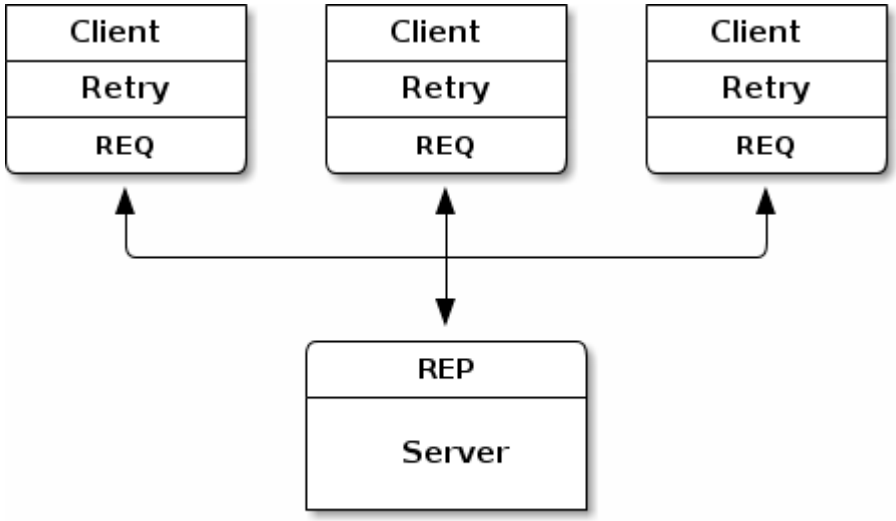

Figure 57 – Lazy Pirate pattern

如果你尝试以任何方式使用 REQ 套接字，而不是严格地发送-接收方式，你将会获得一个 EFSM 错误。这是我们以海盗模式使用 REQ 套接字遇到的让人有点郁闷的问题，因为在获得一个应答之前，我们可能发送几个请求。这个很好的直接断电方法是在一个错误后关闭并重新打开 REQ 套接字。

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

```
#define REQUEST_TIMEOUT    2500    //  msecs, (> 1000!)
#define REQUEST_RETRIES    3      //  Before we abandon

//  Helper function that returns a new configured socket
//  connected to the Hello World server
//
static void *
s_client_socket (void *context) {
   printf ("I: connecting to server...\n");
   void *client = zmq_socket (context, ZMQ_REQ);
   zmq_connect (client, "tcp://localhost:5555");

   //  Configure socket to not wait at close time
   int linger = 0;
   zmq_setsockopt (client, ZMQ_LINGER, &linger, sizeof
(linger));
   return client;
}

int main (void)
{
   s_version_assert (2, 1);
   void *context = zmq_init (1);
   void *client = s_client_socket (context);

   int sequence = 0;
   int retries_left = REQUEST_RETRIES;
   while (retries_left) {
      //  We send a request, then we work to get a reply
      char request [10];
      sprintf (request, "%d", ++sequence);
      s_send (client, request);

      int expect_reply = 1;
      while (expect_reply) {
         //  Poll socket for a reply, with timeout
         zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN,
0 } };
         zmq_poll (items, 1, REQUEST_TIMEOUT * 1000);

         //  If we got a reply, process it
         if (items [0].revents & ZMQ_POLLIN) {
```

```
            //  We got a reply from the server, must match
sequence
            char *reply = s_recv (client);
            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");
            //  Old socket will be confused; close it and
open a new one
            zmq_close (client);
            client = s_client_socket (context);
            //  Send request again, on new socket
            s_send (client, request);
         }
      }
   }
   zmq_close (client);
   zmq_term (context);
   return 0;
}
```

把它与匹配的服务器一起运行：

```
//
//  Lazy Pirate server
//  Binds REQ socket to tcp://*:5555
```

```
//  Like hwserver except:
//   - echoes request as-is
//   - randomly runs slowly, or exits to simulate a crash.
//
#include "zhelpers.h"

int main (void)
{
    srandom ((unsigned) time (NULL));

    void *context = zmq_init (1);
    void *server = zmq_socket (context, ZMQ_REP);
    zmq_bind (server, "tcp://*:5555");

    int cycles = 0;
    while (1) {
        char *request = s_recv (server);
        cycles++;

        //  Simulate various problems, after a few cycles
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: simulating a crash\n");
            break;
        }
        else
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: simulating CPU overload\n");
            sleep (5);
        }
        printf ("I: normal request (%s)\n", request);
        sleep (1);            //  Do some heavy work
        s_send (server, request);
        free (request);
    }
    zmq_close (server);
    zmq_term (context);
    return 0;
}
```

为了运行这个测试用例，在两个控制窗口开启客户机和服务器。在几个消息后，服务器会随机地不正常运行。你可以检查客户机的反应。这是服务器的典型的输出：

```
I: normal request (1)
```

```
I: normal request (2)
I: normal request (3)
I: simulating CPU overload
I: normal request (4)
I: simulating a crash
```

这是客户机的反应：

```
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
```

客户机按顺序排列每个消息，并按顺序准确地检测回来的应答：没有请求或者应答丢失，并且没有应答回来两次，或者顺序错误。运行这个测试几次直到你确信这个机制事实上行得通。在实际中你不需要排列数字，它只是帮助我们信任我们的设计。

客户机使用 REQ 套接字，并使用直接的电源关闭/重启，因为 REQ 套接字使用的是严格地发送/接收循环。你可能打算使用 XREQ 代替，但是这可能不是一个好的决定。首先，它意味着仿真。。。REQ 是带有封装的（如果你已经忘掉那是什么，它是一个你不希望必须做的好的设计）。第二，它意味着可能获得你不希望的应答。

当我们有一系列客户机与单个服务器会话的时候，只在客户机工作线程处处理失败。它能够处理服务器崩溃，但是只要恢复就意味着重启相同的服务器。如果这里有一个永久的错误，即，服务器硬件的断电，这种方法就不适用。因为在任何结构中，服务器中的应用程序代码都是最大的错误源，依靠一个单一的服务器不是一个好的方法。

因此，利与弊如下：

- 利：容易明白和实现。
- 利：对于已经存在的客户机和服务器应用程序代码好处理。
- 利：0MQ 自动尝试重新连接知道它工作。
- 弊：不会对备份文件/交替的服务器做容错转移。

基础的可靠排队（简单海盗模式）

我们的第二种方法采用的是懒海盗模式并且用一个队列设备扩大它。这个队列设备对

多个服务器是透明的，我们就可以更准确地通知工作线程。我们会逐步研究它，以一个小的工作模式开始，这个简单海盗模式。

在所有的这些海盗模式中，工作线程都是无状态的，或者有一些我们不知道的共享状态，即，例如，一个共享数据库。有一个队列设备意味着工作线程的加入和撤销客户机完全不知道。如果一个工作线程死掉，则另一个就会接管。这是一个很好的拓扑，只有一个问题，就是中心队列本身，它可能会成为管理的问题，并且是单一的失败点。

队列设备的基础是第三章的最近最少使用路由队列。处理死掉或者阻塞的工作线程我们至少需要做什么？结果有一点让人吃惊。在客户机我们已经有一个重新尝试机制。因此，利用标准 LRU 队列能够工作的很好。这适合 0MQ 的哲学，我们能够可以通过嵌入简单的设备扩大端到端的模式。

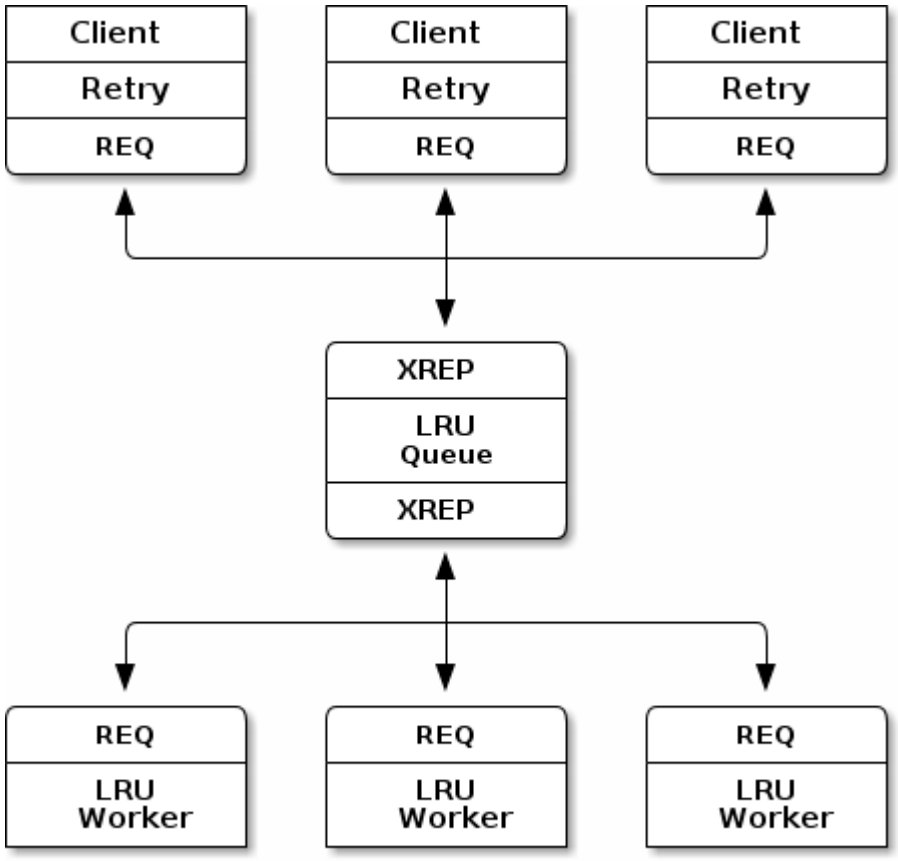

Figure 58 – Simple Pirate Pattern

我们不需要一个特别的客户机，还是使用懒海盗客户机。这是这个队列，它是一个准确的 LRU 队列，不多不少：

```
//
//  Simple Pirate queue
//  This is identical to the LRU pattern, with no reliability
mechanisms
```

```
//  at all. It depends on the client for recovery. Runs
forever.
//
#include "zmsg.h"

#define MAX_WORKERS 100

//  Dequeue operation for queue implemented as array of
anything
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) -
sizeof (q [0]))

int main (void)
{
   s_version_assert (2, 1);

   //  Prepare our context and sockets
   void *context = zmq_init (1);
   void *frontend = zmq_socket (context, ZMQ_XREP);
   void *backend  = zmq_socket (context, ZMQ_XREP);
   zmq_bind (frontend, "tcp://*:5555");   //  For clients
   zmq_bind (backend,  "tcp://*:5556");   //  For workers

   //  Queue of available workers
   int available_workers = 0;
   char *worker_queue [MAX_WORKERS];

   while (1) {
      zmq_pollitem_t items [] = {
         { backend,  0, ZMQ_POLLIN, 0 },
         { frontend, 0, ZMQ_POLLIN, 0 }
      };
      //  Poll frontend only if we have available workers
      if (available_workers)
         zmq_poll (items, 2, -1);
      else
         zmq_poll (items, 1, -1);

      //  Handle worker activity on backend
      if (items [0].revents & ZMQ_POLLIN) {
         zmsg_t *zmsg = zmsg_recv (backend);
         //  Use worker address for LRU routing
         assert (available_workers < MAX_WORKERS);
```

```
            worker_queue [available_workers++] = zmsg_unwrap
(zmsg);

            //  Return reply to client if it's not a READY
            if (strcmp (zmsg_address (zmsg), "READY") == 0)
                zmsg_destroy (&zmsg);
            else
                zmsg_send (&zmsg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Now get next client request, route to next
worker
            zmsg_t *zmsg = zmsg_recv (frontend);
            //  REQ socket in worker needs an envelope
delimiter
            zmsg_wrap (zmsg, worker_queue [0], "");
            zmsg_send (&zmsg, backend);

            //  Dequeue and drop the next worker address
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            available_workers--;
        }
    }
    //  We never exit the main loop
    return 0;
}
```

这是工作线程，它采用懒海盗服务器，并让它适应 LRU 模式（使用 REQ 的'ready'信号）：

```
//
//  Simple Pirate worker
//  Connects REQ socket to tcp://*:5556
//  Implements worker part of LRU queueing
//
#include "zmsg.h"

int main (void)
{
    srandom ((unsigned) time (NULL));

    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_REQ);
```

```
    //  Set random identity to make tracing easier
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof
(0x10000));
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen
(identity));
    zmq_connect (worker, "tcp://localhost:5556");

    //  Tell queue we're ready for work
    printf ("I: (%s) worker ready\n", identity);
    s_send (worker, "READY");

    int cycles = 0;
    while (1) {
        zmsg_t *zmsg = zmsg_recv (worker);

        //  Simulate various problems, after a few cycles
        cycles++;
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating a crash\n", identity);
            zmsg_destroy (&zmsg);
            break;
        }
        else
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating CPU overload\n",
identity);
            sleep (5);
        }
        printf ("I: (%s) normal reply - %s\n", identity,
zmsg_body (zmsg));
        sleep (1);            //  Do some heavy work
        zmsg_send (&zmsg, worker);
    }
    zmq_close (worker);
    zmq_term (context);
    return 0;
}
```

为了测试它，以任意顺序开始工作线程，客户机和队列。你将会看到工作线程崩溃并且烧坏，客户机从新尝试然后放弃。队列从不停止，并且你能够任意地重新启动客户机和工作线程。这种模式可以有任意数量的客户机和工作线程。

强壮的可靠排队（妄想者海盗模式）

简单的海盗队列模式工作的很好，特别是因为它仅仅是两个已经存在的模式的组合，但是它也有一些弱点：

- 对于队列崩溃和重启，它不够强壮。客户机会恢复，但是工作线程不会。当 0MQ 自动地重新连接工作线程的套接字，只要考虑新开始的队列，工作线程还没有发送信号“READY”，因此不存在。为了满足这个要求，从队列到工作线程我们使用心跳，因此，当队列离开的时候，工作线程能够检测的到。

- 对列不会检测工作线程的失败，如果一个工作线程在空闲的时候死掉，队列只会从它的工作队列中移调，并发送它一个请求。客户机等待并免费重新尝试。它不是一个关键的问题，但是，它不好。为了让它恰当地工作，从工作线程到队列我们使用心跳，这样队列在每一步都能检测到一个丢掉的工作线程。

我们以一个合适的书生气的妄想者海盗模式来实现：

首先，对于工作线程我们用一个 REQ 套接字。对这个妄想者海盗工作线程我们将会转到一个 XREQ 套接字。它有一个优势，就是允许我们在任意时候发送和接收消息，而不是 REQ 必须用的锁步发送/接收。XREQ 向下，我们要进行我们自己的封装管理。如果你不知道我说的是什么意思，请再读第三章。

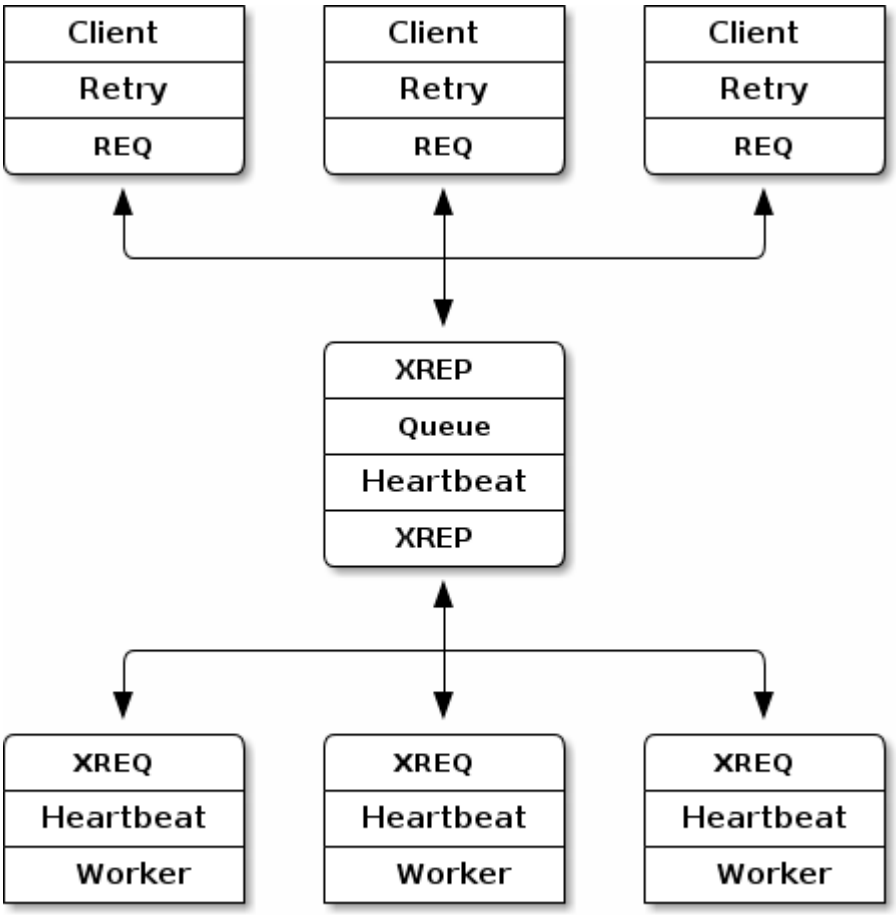

Figure 59 – Paranoid Pirate Pattern

我们仍然会使用懒海盗客户机。这就是妄想者队列设备：

```
//
//  Paranoid Pirate queue
//
#include "zmsg.h"

#define MAX_WORKERS        100
#define HEARTBEAT_LIVENESS  3      //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  1000   //  msecs

//  This defines one active worker in our worker queue

typedef struct {
   char *identity;           //  Address of worker
   int64_t expiry;           //  Expires at this time
} worker_t;

typedef struct {
```

```
    size_t size;             //  Number of workers
    worker_t workers [MAX_WORKERS];
} queue_t;

//  Dequeue operation for queue implemented as array of
anything
#define DEQUEUE(queue, index) memmove (     \
    &(queue) [index], &(queue) [index + 1],  \
    (sizeof (queue) / sizeof (*queue) - index) * sizeof (queue
[0]))

//  Insert worker at end of queue, reset expiry
//  Worker must not already be in queue
static void
s_worker_append (queue_t *queue, char *identity)
{
    int index;
    for (index = 0; index < queue->size; index++)
        if (strcmp (queue->workers [index].identity, identity)
== 0)
            break;

    if (index < queue->size)
        printf ("E: duplicate worker identity %s", identity);
    else {
        assert (queue->size < MAX_WORKERS);
        queue->workers [queue->size].identity = identity;
        queue->workers [queue->size].expiry = s_clock ()
            + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS;
        queue->size++;
    }
}

//  Remove worker from queue, if present
static void
s_worker_delete (queue_t *queue, char *identity)
{
    int index;
    for (index = 0; index < queue->size; index++)
        if (strcmp (queue->workers [index].identity, identity)
== 0)
            break;

    if (index < queue->size) {
```

```
        free (queue->workers [index].identity);
        DEQUEUE (queue->workers, index);
        queue->size--;
    }
}

//  Reset worker expiry, worker must be present
static void
s_worker_refresh (queue_t *queue, char *identity)
{
    int index;
    for (index = 0; index < queue->size; index++)
        if (strcmp (queue->workers [index].identity, identity)
== 0)
            break;

    if (index < queue->size)
        queue->workers [index].expiry = s_clock ()
            + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS;
    else
        printf ("E: worker %s not ready\n", identity);
}

//  Pop next available worker off queue, return identity
static char *
s_worker_dequeue (queue_t *queue)
{
    assert (queue->size);
    char *identity = queue->workers [0].identity;
    DEQUEUE (queue->workers, 0);
    queue->size--;
    return identity;
}

//  Look for & kill expired workers
static void
s_queue_purge (queue_t *queue)
{
    //  Work backwards from oldest so we don't do useless work
    int index;
    for (index = queue->size - 1; index >= 0; index--) {
        if (s_clock () > queue->workers [index].expiry) {
            free (queue->workers [index].identity);
            DEQUEUE (queue->workers, index);
```

```
            queue->size--;
            index--;
        }
    }
}

int main (void)
{
    s_version_assert (2, 1);

    //  Prepare our context and sockets
    void *context = zmq_init (1);
    void *frontend = zmq_socket (context, ZMQ_XREP);
    void *backend  = zmq_socket (context, ZMQ_XREP);
    zmq_bind (frontend, "tcp://*:5555");   //  For clients
    zmq_bind (backend,  "tcp://*:5556");   //  For workers

    //  Queue of available workers
    queue_t *queue = (queue_t *) calloc (1, sizeof (queue_t));

    //  Send out heartbeats at regular intervals
    uint64_t heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        if (queue->size)
            zmq_poll (items, 2, HEARTBEAT_INTERVAL * 1000);
        else
            zmq_poll (items, 1, HEARTBEAT_INTERVAL * 1000);

        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (backend);
            char *identity = zmsg_unwrap (msg);

            //  Return reply to client if it's not a control
message
            if (zmsg_parts (msg) == 1) {
                if (strcmp (zmsg_address (msg), "READY") == 0)
{
```

```
                s_worker_delete (queue, identity);
                s_worker_append (queue, identity);
            }
            else
            if (strcmp (zmsg_address (msg), "HEARTBEAT") ==
0)
                s_worker_refresh (queue, identity);
            else {
                printf ("E: invalid message from %s\n",
identity);
                zmsg_dump (msg);
                free (identity);
            }
            zmsg_destroy (&msg);
         }
         else {
            zmsg_send (&msg, frontend);
            s_worker_append (queue, identity);
         }
      }
      if (items [1].revents & ZMQ_POLLIN) {
         //  Now get next client request, route to next
worker
         zmsg_t *msg = zmsg_recv (frontend);
         char *identity = s_worker_dequeue (queue);
         zmsg_wrap (msg, identity, "");
         zmsg_send (&msg, backend);
         free (identity);
      }

      //  Send heartbeats to idle workers if it's time
      if (s_clock () > heartbeat_at) {
         int index;
         for (index = 0; index < queue->size; index++) {
            zmsg_t *msg = zmsg_new ("HEARTBEAT");
            zmsg_wrap (msg, queue->workers
[index].identity, NULL);
            zmsg_send (&msg, backend);
         }
         heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
      }
      s_queue_purge (queue);
   }
   //  We never exit the main loop
```

```
    //  But pretend to do the right shutdown anyhow
    while (queue->size)
        free (s_worker_dequeue (queue));
    free (queue);
    zmq_close (frontend);
    zmq_close (backend);
    return 0;
}
```

关于这个例子的一些建议：

- 用 C 语言，管理任何类型的数据结构都很让人讨厌。这个队列真的需要两种类型的数据结构：服务器的最近最少使用列表，和同类服务器的分区。C 代码不是最优的，不会这样的扩大规模。一个合适的使用分区和列表容器的版本如 ZFL 工程提供的。

- 队列利用工作线程心跳扩大 LRU 模式。一旦它开始工作，就很简单，但是很难创造。关于心跳的问题，一会我会解释更多。

这就是妄想者海盗工作线程：

```
//
//  Paranoid Pirate worker
//
#include "zmsg.h"

#define HEARTBEAT_LIVENESS  3      //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  1000   //  msecs
#define INTERVAL_INIT      1000    //  Initial reconnect
#define INTERVAL_MAX      32000    //  After exponential
backoff

//  Helper function that returns a new configured socket
//  connected to the Hello World server
//
static char identity [10];

static void *
s_worker_socket (void *context) {
    void *worker = zmq_socket (context, ZMQ_XREQ);

    //  Set random identity to make tracing easier
```

```
    sprintf (identity, "%04X-%04X", randof (0x10000), randof
(0x10000));
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen
(identity));
    zmq_connect (worker, "tcp://localhost:5556");

    //  Configure socket to not wait at close time
    int linger = 0;
    zmq_setsockopt (worker, ZMQ_LINGER, &linger, sizeof
(linger));

    //  Tell queue we're ready for work
    printf ("I: (%s) worker ready\n", identity);
    s_send (worker, "READY");

    return worker;
}

int main (void)
{
    s_version_assert (2, 1);
    srandom ((unsigned) time (NULL));

    void *context = zmq_init (1);
    void *worker = s_worker_socket (context);

    //  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 = s_clock () + HEARTBEAT_INTERVAL;

    int cycles = 0;
    while (1) {
        zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN,
0 } };
        zmq_poll (items, 1, HEARTBEAT_INTERVAL * 1000);

        if (items [0].revents & ZMQ_POLLIN) {
            //  Get message
            //  - 3-part envelope + content -> request
            //  - 1-part "HEARTBEAT" -> heartbeat
```

```
          zmsg_t *msg = zmsg_recv (worker);

          if (zmsg_parts (msg) == 3) {
             //  Simulate various problems, after a few
cycles
             cycles++;
             if (cycles > 3 && randof (5) == 0) {
                printf ("I: (%s) simulating a crash\n",
identity);
                zmsg_destroy (&msg);
                break;
             }
             else
             if (cycles > 3 && randof (5) == 0) {
                printf ("I: (%s) simulating CPU overload\n",
identity);
                sleep (5);
             }
             printf ("I: (%s) normal reply - %s\n",
                identity, zmsg_body (msg));
             zmsg_send (&msg, worker);
             liveness = HEARTBEAT_LIVENESS;
             sleep (1);            //  Do some heavy work
          }
          else
          if (zmsg_parts (msg) == 1
          && strcmp (zmsg_body (msg), "HEARTBEAT") == 0)
             liveness = HEARTBEAT_LIVENESS;
          else {
             printf ("E: (%s) invalid message\n", identity);
             zmsg_dump (msg);
          }
          interval = INTERVAL_INIT;
       }
       else
       if (--liveness == 0) {
          printf ("W: (%s) heartbeat failure, can't reach
queue\n",
             identity);
          printf ("W: (%s) reconnecting in %zd msec...\n",
             identity, interval);
          s_sleep (interval);

          if (interval < INTERVAL_MAX)
```

```
            interval *= 2;
         zmq_close (worker);
         worker = s_worker_socket (context);
         liveness = HEARTBEAT_LIVENESS;
      }

      //  Send heartbeat to queue if it's time
      if (s_clock () > heartbeat_at) {
         heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
         printf ("I: (%s) worker heartbeat\n", identity);
         s_send (worker, "HEARTBEAT");
      }
   }
   zmq_close (worker);
   zmq_term (context);
   return 0;
}
```

关于这个代码的一些建议：

- 和之前一样，这个代码包括了对失败的仿真。这让它很难调试，并且对于重复利用来说很危险。当你试图调试它，让对失败的仿真不可用。
- 对于妄想者海盗队列，让心跳正确很棘手。看以下关于心跳的讨论。
- 工作线程使用了与我们为懒海盗设计的相似的重新连接策略。有两个主要的不同：它使用越来越快的补偿，并且它从不禁止。

尝试客户机，队列，和工作线程，例如，使用一个这样的脚本：

```
ppqueue &
for i in 1 2 3 4; do
   ppworker &
   sleep 1
done
lpclient &
```

你会看到工作线程一个接一个地死掉，因为它们仿真一个崩溃，并且客户机事实上放弃了。你可以停止并重新启动队列，客户机和工作线程都会重新连接并继续运行。并且不管你对队列和工作线程做了什么，客户机都不会获得无序的应答：整个链要么是工作线程要么是客户机禁止。

心跳

当写这个妄想者海盗例子，大概花了五个小时让队列-到-工作线程心跳正常地工作。请求-应答链的剩余部分大概用了十分钟。心跳是容易导致很多问题的可靠层之一。很容易创建假的失败，即，端点决定它们会撤销连接，因为心跳没有正确的发送。

当理解并应用心跳的时候需要考虑的一些问题：

注意，心跳不是请求-应答。它们在两个方向异步地流动。任何一个端点能够检测到别的死掉了并停止与它会话。

如果一个端点使用持久套接字，这就意味着它可能会获得排队的心跳，它将会接受，如果它重新连接。就这个原因，工作线程不应该重用持久套接字。例子代码利用持久套接字以达到调试的目的，它们是随机的（理论上）从来不要重复利用一个已经存在的套接字。

首先，让心跳工作，并且在剩下的消息流中增加。你应该能够证明以任何顺序开始、结束、重新启动端点、仿真冻结、等等，心跳都能工作。

合同与协议

如果你注意了的，你就会意识到妄想者海盗模式与简单海盗模式是不兼容的，原因是心跳。

事实上在这我们需要的是一个需要写下来的协议。没有规则的尝试很有趣，但是对于实际的应用程序，这样做事不明智的。如果我们想要用另外一种语言来写这个代理会怎样？你必须读代码知道它是怎么工作的吗？如果由于某种原因我们想要改变协议呢？协议可能简单，但是它不是公认的，并且，如果它成功了，它将变得更复杂。

缺少合同是一次性应用程序的明显标志。因此，让我们为这个协议写一个合同。我们怎样做呢？

- 这是一个在 rfc.zeromq.org 中的网络文摘，我们专门把它设置为公共 0MQ 合同的主页。
- 为了创建新的规范，注册，和接下来的说明。它是直截了当的，虽然技术的描述不是对所有的都适用。

我大概用了五十分钟来起草新的海盗模式协议。它不是一个大的规范，但是它确实抓住了足够的东西，能够成为争论的规范。（“你的队列不是点到点兼容的，请修改它！”）

把点到点协议转换为真正的协议还需要更多的工作：

- 在 READY 指令中应该有一个版本号，这样它才可能安全地创建新版的点到点协议。
- 现在，请求和应答的 READY 和 HEARTBEAT 并不是完全不同的。为了让它们不同，我们想要一个包含“消息类型”部分的消息结构。

面向服务器的可靠的队列（管家模式）

前不久，我们还梦想有一个更好的协议能够修理这个世界。现在，我们有了：

这个一页纸纸的规范采用的是点到点协议，并把它转换成更可靠的东西。我们怎眼设计一个复杂的结构：通过写下合同，然后编写软件实现它。

管家婆协议（MDP）在除上面讲到的两点外以一种有趣的方式扩展并改善了点到点协议。对客户机发送的请求它增加了一个“服务器名字”，并要求工作线程注册到专门的服务器。关于 MDP 好的是它来自工作线程代码，一个更简单的协议，一种恰到好处的改善。这让草拟很简单。

增加服务器名字是一个小的但是很重要的改变，它让我们的妄想者海盗队列变成一个面向服务器的代理：

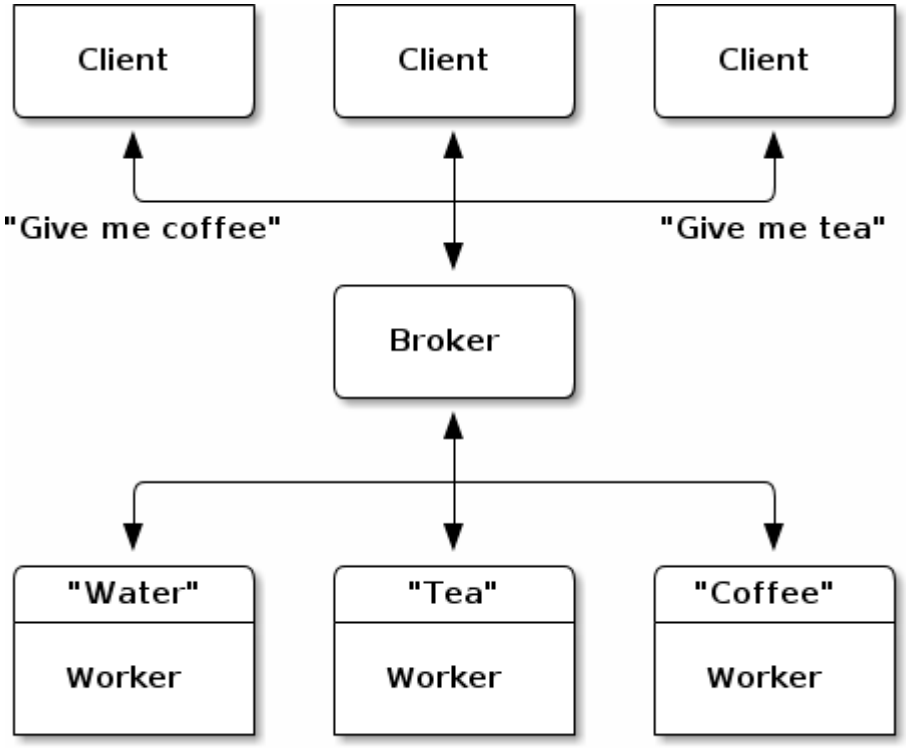

Figure 60 – Majordomo Pattern

为了实现管家婆，我们需要为客户机和工作线程编写一个框架。当你能够使用一个更简单的应用程序接口函数编译并进行一次测试的时候，让每一个应用程序开发者读这个说明并让它生效是不明智的。

因此，当我们的第一个合同（管家婆协议本身）定义了我们的分布式结构的各部分怎样与各部分通信，我们的第二个合同定义了用户的应用程序怎样与我们正在设计的技术框架会话。

管家婆协议有两方，一边是客户机，一边是工作线程。因为我们要同时写客户机和工

作线程的应用程序，我们需要两个应用程序接口函数。这是客户机接口函数框架，使用一种简单的面向对象的方法。我们用 C 语言编写的，使用 ZFL 库类型：

```
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);
```

我们打开一个会话到代理，我们发送一个请求，并获得一个应答信号，最后我们关闭连接。这是代理接口函数的构架：

```
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);
```

它或多或少是对称的，但是线程的会话有点不同。当工作线程第一次调用 recv()的时候，它传送一个空的应答信号，在此之后，它传送一个直接的应答信号，并获得一个新的请求。

客户机和工作线程的应用程序接口函数很容易构建，因为它们主要是基于我们已经开发的妄想者海盗代码。这就是客户机的引用程序接口函数。
mdcliapi: Majordomo client API in C

Lua | Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

有一个测试程序有 100K 的请求-应答循环。

mdclient: Majordomo client application in C

C++ | Lua | Ada | Basic | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

这是工作线程的应用程序接口函数：

mdwrkapi: Majordomo worker API in C

Lua | Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

这是个实现“应声”服务的测试程序：

mdworker: Majordomo worker application in C

C++ | Lua | Ada | Basic | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

关于这个代码的注意事项：

- 这些应用程序接口函数是单线程的。这就意味着，例如，在后台工作线程不会发送心跳。很好，这就是我们想要的：如果工作线程被卡住，心跳就会停止，并且代理会停止发送请求到工作线程。

- 工作线程 API 不会做额外的补偿，它不比增加额外的复杂性。

- 这些 API 不会做任何的错误报告。如果有什么异常情况，它扔出一个中断（或者是异常，取决于语言）。对于一个访问的实现，它很完美，因此，任何协议错误都会很快显示出来。对于实际的应用程序，API 应该对无效的消息有免疫力。

让我们设计管家婆代理。它的核心结构是一些列的队列，每个提供一种功能。当工作线程出现的时候，我们将会创建这些队列（在工作线程消失的时候，我们将会删除它们，但是暂时忘记它吧，它变的复杂了）。除此以外，我们对每一个服务，我们保持一个工作线程队列。

为了让 C 语言代码的例子容易读和写，我们已经采用了来自 ZFL 工程的分区和列表容器类并当我们处理 zmsg 的时候把它们重新命名为 zlist 和 zhash。在任何一种模式语言中，你自然可以使用内置容器。

这是代理：

mdbroker: Majordomo broker in C

C++ | Lua | Ada | Basic | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

这是到现在为止，我们见过的最复杂的例子。它大概有 500 行代码。为了编写它，并让它足够强壮，我用了两天的时间。但是，对于一个完全面向服务器的代理来说，只是一小部分。

关于这个代码的注意事项：

- 管家婆协议让我们在一个单一的套接字上处理客户机和服务器程序。对于展示和管理代理来说这样更好了：它只是设置在 0MQ 的一个终端，而不是想绝大多数设备一样，两端都需要。

- 代理正确地实现所有的 MDP/0.1（据我所知），包括如果代理发送无效的指令，心跳，等则断开连接。

- 它可以扩大到运行多个线程，每个管理一个套接字和一系列的客户机和工作线程。这对于分段大的结构来说很有意义。C 代码已经是围绕代理类组织来让它变的简单。

- 一个初级的容错转移或者活的到活的代理可靠性模式是简单的，因为除非服务存在，代理基本上是无状态的。如果它们的第一个选择没有实现，它根据客户机和工作线程来选择另一个代理。

- 这个例子使用了 5 秒的心跳，主要是当你使用追踪的时候来减少输出的数量。对于很多的本地应用程序。然而，任意的重新尝试对于一个服务器重新启动来说都太慢，至少会达到十秒。

异步管家婆模式

前面我们实现管家婆模式的那种方法简单同时很愚蠢。客户机仅仅是原始的简单海盗模式，包裹在一个别样的 API 中。当我在一台测试机上启动一个客户机，代理，和工作线程，它可以在 14 秒内处理大约 100000 个请求。这部分取决于当 CPU 循环空闲的时候复制消息构架的代码。但是，实际问题是，我们是迂回套利的。0MQ 让 [http://en.wikipedia.org/wiki/Nagle 算法不可用，但是迂回套利仍然很慢。

理论在理论中更好，但是在实际中，实际更好。让我们用一个简单的程序来测试迂回套利的代价。发送一束消息，首先等待每个消息的应答信号，然后批处理地读回所有的应答。两种方法做了同样的工作，但是它们的结果是完全不同的。我们构造一个客户机，代理和工作线程的模型：

tripping: Round-trip demonstrator in C

C++ | Lua | PHP | Ada | Basic | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | Python | Ruby

在我的开发机子上，程序的结果是：

```
Setting up test...
Synchronous round-trip test...
 9057 calls/second
Asynchronous round-trip test...
 173010 calls/second
```

注意客户机线程在开始之前有一个小的延迟。这是根据路由套接字的一个特征：如果你发送一个带还没有连接上的端点的地址的消息，消息将会被丢弃。在这个例子中，我们不使用 LRU 机制，因此，没有休眠，如果工作线程连接的太慢，它将会丢掉消息，把我们的测试弄乱。

正如我们看到的一样，在最简单的案例中，迂回套利方式比“当它离开的时候，直接把它推入管道”的异步方式慢了 20 倍。让我们看看我们是否能够把它应用于管家婆模式。

首先，让我们修改客户机 API 来得到分离的发送和接收方法：

```
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);
```

只需要几分钟的时间实现同步客户机 API 到异步的转换：

mdcliapi2: Majordomo asynchronous client API in C

Lua | Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

这是对应的客户机处理程序：

mdclient2: Majordomo client application in C

C++ | Lua | Ada | Basic | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

代理和工作线程是不变的，因为我们一点也没有改变协议。我们可以看到在性能方面提高了很多。这是发出 10K 请求-应答循环的同步客户机。

```
$ time mdclient
100000 requests/replies processed

real    0m14.088s
user    0m1.310s
sys     0m2.670s
```

这是有一个单一线程的异步的客户机：

```
$ time mdclient2
100000 replies received

real    0m8.730s
user    0m0.920s
sys     0m1.550s
```

快两倍。还不坏，但是，让我们打开十个工作线程，看看它处理的如何：

```
$ time mdclient2
100000 replies received

real    0m3.863s
user    0m0.730s
sys     0m0.470s
```

它不完全是异步的，因为工作线程接收消息是基于严格 LRU 算法的。但是，当有多个线程的时候它会处理的更好。在我的告诉测试主机上，在 8 个左右的工作线程后，它没有变的更快。四个核只能延伸这么远了。但是在几分钟的任务内，在吞吐量上，我们获得了四倍的提高。代理仍然不是最优的。它花了绝大多数的时间来复制消息帧，而不是实现它本来能够办到的零复制。但是在很少的努力地情况下，我们获得了每秒 25K 的可靠的请求/应答调用。

然而，异步的管家婆模式并不是只有优点。它有一个很大的弱点，即，没有更多的工作的话，它不能够从一个代理崩溃恢复。如果你看看 mdcliapi2 代码，你将会看到在一次失败以后，它并不打算重新连接。一个恰当的重新连接要求：

- 每个请求都是编号了的，并且每个应答都有匹配的号码，这需要改变协议来加强。
- 客户机 API 追踪并且持有所有的为完成的请求，即，哪些还没有收到应答的请求。
- 根据容错转移，客户机 API 再发送为完成的请求到代理。

它不是一个破坏者，但是它确实表示了性能的提高就意味着程序更加复杂。对于管家婆模式，有这个必要吗？它取决于你的使用情况。对于一个名字查找服务器，每个会话调用一次的情况不适合。对于一个有几千个客户机的前端到后端的网页服务器，有这个必要。

发现服务器

我们有一个好的面向服务器的代理，但是我们没有方法知道某个特定的服务器是否存在。我们知道某个请求是否失败，但是我们不知道为什么它失败了。很有必要咨询代理“应答服务器运行了吗？”最明显的方式是修改我们的 MDP/客户机协议来增加询问代理的指令，“服务器 X 是否运行了？”但是 MDP/客户机最大的魅力所在就是简单。增加服务器发现功能就会让他变得和工作线程协议一样复杂。

另一种方法是如同 Email 一样，并且让无法投递的请求返回。在一个异步环境中，这样会实现的很好，但是，它也增加了复杂性。我们需要一种来分辨来自一个应答方返回的请求，并且恰当地处理它们。

让我们尝试我们已经建立的，并且在 MDP 上编译，而不是修改它。服务器发现本身是一种服务。它可能需要几个管理服务，例如“服务器 X 不可用”，“提供统计值”，等等。我们想要的是一种通用的，可扩展的解决方案，它不影响已经存在的协议和应用程序。

这是一个小的 RFC-MMI,或者管家婆模式管理界面-在 MDP 的顶层：http://rfc.zeromq.org/spec:8 在代理中我们已经实现了它，但是，除非你读了所有的东西，否则你可能错过它。这显示了我们如何在一个应用程序中使用服务器发现的：

mmiecho: Service discovery over Majordomo in C

Lua | Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

代理检查服务器的名字，并且，任何服务都以“mmi”开始它自己，而不是传送一个请求到工作线程。用或者不用一个运行的工作线程，你将会看到这个小程序将会分别报告‘200’或者‘404’。在我们的示例代理中，实现 MMI 还是比较难办的。例如，如果一二工作线程消失了，服务仍然是”存在“。在实际中，当在指定的截至时间后如果没有工作线程，一个代理应该将服务删除。

等幂服务

等幂性不是什么特别的东西。它的意思是重复一个操作是安全的。检查时钟是非等幂的。当很多客户机-到-一个服务器用例不是等幂的时候，一些不是。等幂的用例包括：

- 无状态的任务分发，即，一个管道，其中服务器是无状态的工作线程，它计算一个基于一个请求提供的应答。在这样的例子中，多次执行相同的请求是安全的（即使无效）。
- 一个命名服务器把逻辑地址转换为终端地址以绑定和连接。在这样一个例子中，实现相同的查阅请求多次是安全的。

这是非等幂用例：

- 一个登录服务器。它不希望相同的登录信息不止出现一次。
- 任何对向下的节点有影响的服务，例如，发送一条信息到另一个节点。如果那个服务器不止一次获得相同的信息，下游的节点将会获得完全一样的信息。
- 以非等幂方式修改共享数据的任意服务，例如，计入银行账户肯定不是等幂的。

当我们的服务器应用程序不是等幂的，我们要更加认真地考虑它们什么时候可能崩溃。如果一个应用程序在空闲的时候死掉，或者当它在处理一个请求的死掉，通常没有什么问题。我们可以用一个数据库来确保借方和结余总是一起工作的。如果服务器在发送应答的时候死掉，那就是个问题了，因为，它被认为是在处理它的工作。

如果网络在应答路由回客户机的时候死掉，也会产生相同的问题。客户机将会认为服务器死掉了，将会重新发送请求，服务器会做两次相同的工作。这不是我们希望的。

我们使用相当标准的方法检测并排斥重复的请求。即：

- 客户机必须用一个专门的客户机身份和专门的消息号标识一个请求。
- 服务器在发送一个应答回去之前，用客户机 ID+消息号作为关键字来存储的。
- 当服务器从一个给定的客户机获得一个请求后，首先检查它是否有一个给这个客户机 ID+消息号的应答。如果有，它不处理这个消息，仅仅是再发送一遍这个应答。

断开连接的可靠性（泰坦尼克模式）

一旦你意识到管家婆模式是一种可靠的消息代理，你可能增加一些防锈。毕竟，它对所有公司的消息系统都适用。它是一个吸引人的主意，所以一般不会被忽略。但是，它是我的一个专长。所以，一些关于你不想知道的关于生锈的代理位于你的构架中心的原因是：

- 正如你已经看到了的一样，懒海盗客户机表现的让人吃惊的好。从直接的客户机-到-服务器到分布式队列，跨整个范围的体系结构它都工作。它事实上假设工作线程是无状态的和等幂的。但是我们可以在这个范围内工作，而不用求助于铁锈。
- 铁锈带来了一系列的问题，从慢性能到额外的必须管理，修复和创建的部分。通常，海盗模式的优点是它们的简单性。它们不会崩溃。并且，如果你还是担心硬件，你可以转到完全没有代理的端到端模式。随后在这章中我还会讲到。

已经说了这些，然而，有一种基于铁锈的可靠性用例，它是一种异步的非连接网络。它解决了海盗模式的一个主要的问题，即一个客户机载实际时间内必须等待一个应答。如果客户机和工作线程只是偶尔地连接（想想 email 做为类比），在客户机和工作线程之间我们不能使用无状态网络。我们必须在中间加上状态。

因此，这就是泰坦尼克模式，其中，我们写一个消息存储到硬盘中，以确保它不会丢失，不管客户机和工作线程的连接怎样偶然。当我们使用服务发现，我们将会在管家婆上使用泰坦尼克，而不是扩展 MDP。它相当懒，因为它意味着我们可以在一个专门的工作线程中实现我们的开启-和-忘记可靠性，而不是在代理中实现。这样做在以下几方面体现出它的聪明：

- 它简单很多，很多。
- 它允许我们混合用一种语言写的代理到另一种语言写的代理中去。

- 它允许我们独立地发展开启-和-忘记技术。

唯一的缺点是在代理和硬盘之间有一个额外的网络跳数。我们的目的是简单和不纠结。在尝试它一段时间后，我能够想到的最简单的设计是把泰坦尼克做为一个代理服务器。因此，它一点也不影响工作线程。如果一个客户机想要一个立即的应答，它直接与一个服务会话，并且希望这个服务是可用的。如果一个客户机高兴等待一会，它就会和泰坦尼克会话，它问道“你好，伙伴，当我去买我的食品的时候你能帮我保管它吗？”

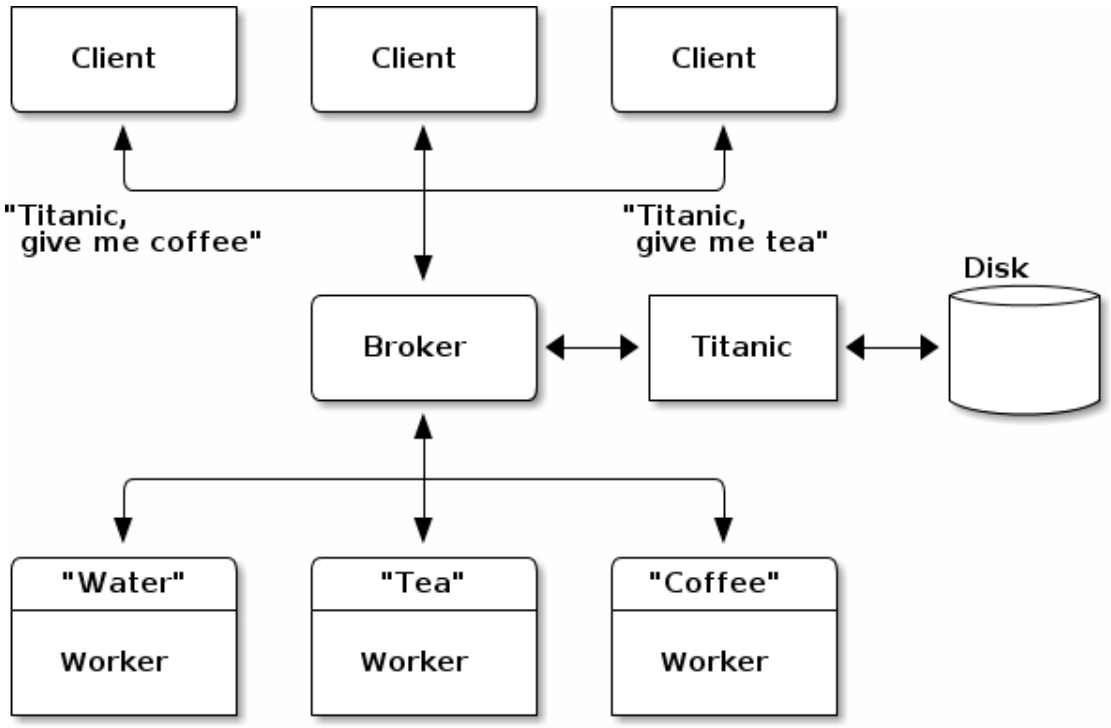

Figure 61 – Titanic Pattern

泰坦尼克是一个工作线程同时是一个客户机。客户机和泰坦尼克之间的会话沿着这样的路线进行：

- 客户机：请为了接受这个请求。泰坦尼克：好的，没问题。
- 客户机：你那有我的一个应答码？泰坦尼克：是的，这呢。或者，还没有。
- 客户机：好的，现在你可以抹去那个请求了，它进行的很好。泰坦尼克：好的，没有问题。

然而，在泰坦尼克和代理之间的会话沿着这样的路线：

- 泰坦尼克：嗨，代理，你这有一个应答服务吗？代理：呃，是的，好像是。
- 泰坦尼克：嗨，应答，请为我处理它。应答：当然，给你。
- 泰坦尼克：太好了！

你可以这样工作，可能的失败情况集合是。当处理一个请求的时候，如果一个工作线程崩溃可，泰坦尼克会永不停止地重新尝试。如果一个应答在某个地方丢了，泰坦尼克会重新尝试。如果请求处理了，但是客户机没有获得应答，它会再次请求。如果在处理一个请求或者应答的时候泰坦尼克崩溃了，客户机将会重新尝试。如果请求是安全存储的，任务就不会丢失。

信息交换是呆板的，但是也可以是管道的，即，客户机可以使用异步的管家婆模式来处理很多的任务，之后再获得应答信号。

对一个客户机来请求它的应答来说，我们需要某种方式。我们将会有很多歌客户机请求相同的服务，客户机消失并以不同的身份重新出现。因此，这是一个简单的，明智的安全的解决方案：

- 每个请求产生一个通用的特定 ID（UUID），它是在客户机排队请求的时候有泰坦尼克返回给客户机的。
- 当一个客户机请求一个应答，它必须制定原始请求的 UUID.

这样就给客户机增加了存储它的请求 UUID 的责任，但是它身份验证的需要。这有什么备选方案吗？我们可以使用持久套接字，即明确的客户机身份。当我们有很多客户机的时候，它就产生了一个管理问题，并且也存在两个客户机使用相同身份的隐患。

在我们开始写另一个正式的规范前，让我们考虑一下客户机是怎样与泰坦尼克会话的。一种方式是使用一种单一的服务，并发送给它三种不同的请求类型。另一种方式，它看起来更简单，是使用三种服务：

- 泰坦尼克请求-存储一个请求消息，并为这个请求返回一个 UUID。
- 泰坦尼克应答-如果可以的话，为一个给定的请求 UUID 取回一个应答。
- 泰坦尼克关闭-确认一个应答已经存储并且处理了。

我们使用了一个多线程工作线程，正如我们的 0MQ 多线程实验所看到的，它很详细。然而，在我们研究代码之前，让我们描绘一下泰坦尼克以 0MQ 消息和帧的形式看起来像什么： http://rfc.zeromq.org/spec:9. 这就是“泰坦尼克服务协议”，或者叫做 TSP.

对于客户机应用程序来说，使用 TSP 肯定比通过 MDP 直接访问一个服务器需要更多的任务。这是最简单的强壮应答客户机例子：

ticlient: Titanic client example in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

当然，在实际中它会以某种框架封装起来。实际的应用程序开发者从来都不会看到消息的打开和关闭，它是一种让更多技术型专家建立框架和 API 的一种工具。如果我有无限的时间来开发的话，我将会构造一个 TSP 的例子，并且把把客户机应用程序变回几行代码。但是，它和我们看到的 MDP 是相同的理论，没有必要重复了。

这是泰坦尼克的实现。服务器利用三个线程处理三种服务。对于最强壮的可能的方式存储方式，它非常有毅力。它如此的简单，可怕，唯一复杂的地方是，对所有的请求，它保持了一个分离的队列以避免重复读取目录：

titanic: Titanic client example in C

Ada | Basic | C+ + | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

为了测试它，代开 mdbroker 和泰坦尼克，然后运行 ticlient.现在任意地开始 mdworker，你会看到客户机得到一个应答，并且高兴地存在。

关于这个代码的一些注意事项：

- 我们使用 MMI 仅仅把请求发送到在运行的服务。它也能在 MMI 所在的代理中实现。
- 我们使用来发送来自泰坦尼克请求服务的新的请求数据到新的发报机。它可以让发报机不用扫描目录，加载所有的请求文件，并以日期和时间来分类。

关于这个代码重要的问题不是性能（它当然很可怕，我没有测试它），但是它能够很好地实现可靠的协议。为了测试它，开始 mdbroker 和泰坦尼克程序。然后开始 ticlient，然后开始 mdworker 应答服务。你可以通过使用‘-v’选项运行这四个来做详细的活性跟踪。除非客户机或者任何部分都没有丢掉，你可以停止并重启任何部分。

如果在实际的例子中你想要用到泰坦尼克，你将会快速地被问到“我们怎样能将它变的更快？”这就是我要做的，以例子的实现开始：

- 对所有的数据，使用一个单一的硬盘文件，而不是多个文件。操作系统通常会更好的处理少量的大的文件而不是很多小的文件。

- 作为一个循环的栈来建立这个硬盘文件，这样新的请求可以连续地写入（可能偶尔会有环绕式处理）。一个全速写到一个硬盘文件的线程可以工作的很快。

- 阻止内存中的参数进入硬盘缓冲区，并且在启动时重新编译参数。它节省了额外的硬盘头飘动的需要，以保证硬盘中的参数完全。在每个消息后，你会希望有一个 fsync,或者每 N 微秒，如果由于系统错误，你打算丢掉最后的 M 各消息。

- 使用一个固定状态驱动而不是氧化铁盘片旋转。

- 预先分配整个文件，或者在大的允许循环缓冲的组块中爱分配。这样就避免了分裂并并确保了很多读和写是连续的。

等等。我不建议在数据库中存储消息，即使是一个快的关键字和值的存储，除非你真的喜欢一个专门的数据库并且没有性能方面的担心。你将会为这个抽象付出更多的代价，是一个裸盘文件的 10 到 1000 倍。如果你想让一个泰坦尼克更加可靠，你可以把请求和应

答直接存储在内存中。它将会为你提供非连接网络的功能，但是它不会重泰坦尼克本身崩溃中恢复。

高可靠性对（二进制星形模式）

二进制星形模式把两个服务器放进一个初级备份高可靠性对。在任意给定的时间，其中一个接收来自客户机应用程序的连接（它是主人）另一个不接受（它是奴隶）。两个服务器互相监测。如果主人从网络上消失，一段时间后，奴隶接管过来作为主人。

二进制星形模式由 Pieter Hintjens 和 Martin Sustrik 为 Matix OpenAMQ server 服务器开发。我们设计它：

- 为了提供一个直接的前向高可靠性方案。
- 为了更简单以便明白和使用。
- 为了在需要的时候进行可靠的容错转移，并只在需要的时候。

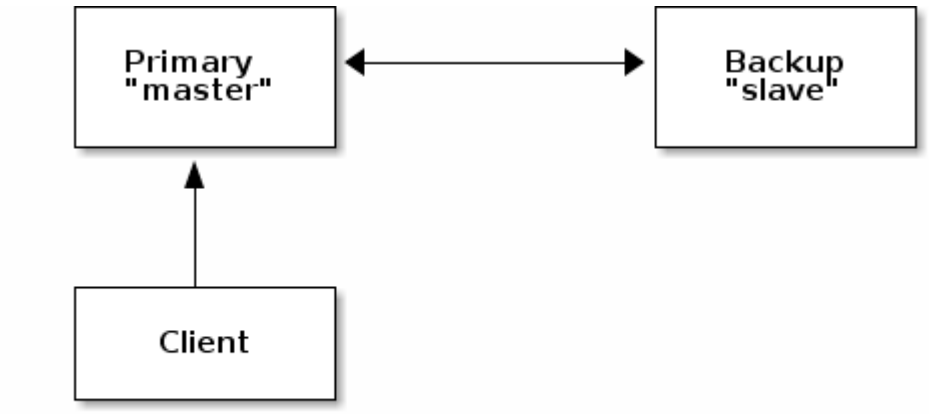

Figure 62 – High availability pair, normal operation

假设我们有二进制新型对在运行，以下是会导致容错转移的可能的情况：

1. 运行首个服务器的硬件有一个致命的问题（供电支持爆炸，机器着火，或者某人错误不小心拔去了插头），消失了。应用程序发现了这个问题，重新连接到备用服务器。
2. 前个服务器崩溃的网络段-可能是一个路由遭遇了尖峰电压-应用程序重新连接到备用服务器。
3. 前一个服务器崩溃或者被操作者终止并且没有自动重启。

像这样从容错转移任务恢复：

1. 操作者重新启动前一个服务器并且休整任意导致它重网络上消失的问题。
2. 操作者在会给应用程序带来最小破坏的时刻终止备用服务器。
3. 当应用程序已经重新连接到前一个服务器，操作者重新启动后一个服务器。

恢复（使用前一个服务器作为主人）是一个手动的操作。痛苦的经历教育我们自动恢复是不如人意的。以下是几点原因：

- 容错转移给应用程序创建了容错的中断，可能延迟 10-30 秒。如果存在一个紧急情况，这比完全的停止运行好很多。但是如果恢复创建比 10-30 秒更久的中断，它发生在非高峰期会更好，这时使用者下线了。
- 当遇到一个紧急情况，最好为这些尝试修理的东西创建可预见性。自动恢复为系统管理人员创建了不确定性，如果不两次检测，他将不知道究竟是那台服务器在管理。
- 最后，在网络将故障转移，然后恢复，然后把操作符放到一个困难的地方来分析究竟什么发生的时候，就可以实现自动恢复了。这是一个中断服务，但是原因还不清楚。

说了这么多，如果这些（重新）运行的话，二进制星形模式将会失败返回到前一个服务器，备用服务器将失败。事实上这些就是我们引发恢复的方法。

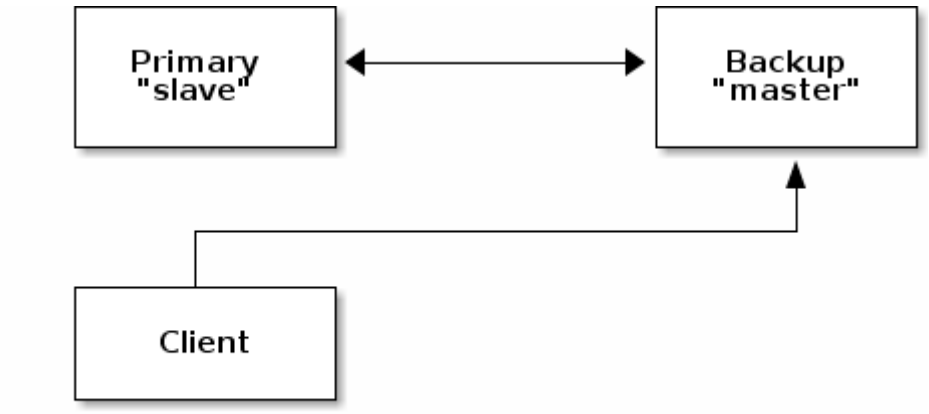

Figure 63 – –High availability pair, during failover

对二进制星形对关闭的进程是为了：

1. 停止被动的服务器，并且在随后的任意时间停止主动的服务器，或者
2. 以任意顺序关闭两台服务器，但是关闭的时间间隔不要超过几秒。

先关闭主动的服务器，在关闭被动的服务器时，如果两者关闭的时间间隔超过容错截至时间将会使应用程序断开连接，然后重新连接，然后再断开连接，这样会是使用者感到很困惑。

详细的请求

二进制星形是在精确工作的前提下尽可能的简单的情况。事实上当前的设计是第三种完成了的新设计。我们发现前面两种设计都太复杂了，想做的太多，我们抽出它的功能，知道我们找到了一种设计，它可以理解和使用，并且对于使用来说足够可靠。

以下是我们对高性能结构的要求：

- 容错转移是为了阻止毁灭性的系统失败，如硬件崩溃，起火，意外事件等。为了保护原始的服务器崩溃，有更简单的恢复的方法。
- 容错转移应该在 60 秒内，最好是在 10 秒内。
- 容错转移必须自动发生，然而恢复必须手动进行。我们希望应用程序自动转换到备用的服务器，但是我们不希望它转换到前一个服务器，除非操作人员修整了存在的任意问题，并且认为那是一个在此中断应用程序的好时间。
- 对客户机应用程序的场景应该尽量简单，并且对于开发者来说应该要容易理解。理想情况是它们隐藏在客户机 API 中。
- 关于怎样避免在星形对中两个服务器都以为自己是主人的情况应该进行详细的说明。
- 应该对两台服务器的打开顺序没有依赖。
- 应该能够让停止和重新启动服务器不要影响到客户机应用程序（虽然他们可能强制性地重新连接）。
- 操作员必须要能够一直监测两个服务器。
- 用专门的高速的网络必须要能够连接两个服务器。即，容错转移必须使用一个专门的 IP 路由。

我们做以下的假设：

- 一个单一的备用服务器提供足够的保障，我们不需要多级的备用服务器。
- 开始的和备用的服务器有等同的携带应用程序负荷的能力。我们不打算再两个服务器之间实现负荷均衡。
- 有足够的预算来支持完全多余的备用服务器，它有可能一直什么都不做。

我们不打算涉及到：

- 主动备用服务器的使用或者符合均衡。在二进制星形对中，备用服务器是非主动的，并且在前一个服务器下线之前都没有有用的任务。
- 持久消息处理，或者任意方式的交易。我们假设一个不可靠的（或者不可信任的）服务器的网络或者二进制星形对。
- 关于网络的任意自动地探索。二进制星形对是手动的，并且直接在网络中定义，被应用程序知道（至少在它们的配置数据中）。
- 在服务器之间复制消息或者状态。当它们失败的时候，应用程序应该创建所有服务器的状态。

以下是我们在二进制星形模型中使用的关键术语：

- 前面的-前面的服务器是通常的‘主人’。
- 备用的-备用的服务器是通常的‘奴隶‘，当前面的服务器从网络上消失，并且客户机应用程序请求备用服务器连接的时候它将会变成主人。
- 主人-主人服务器是接受客户机连接的二进制星形对中的那个。通常只有一个主人服务器。

- 奴隶-奴隶服务器是如果主人消失的话它就会接替的那个服务器。注意，当二进制星形对运行正常地时候，前面的服务器是主人，备用服务器是奴隶。当容错转移发生的时候，角色就互换了。

为了配置二进制星形对，你需要：

1. 告诉前面的服务器，备用服务器在哪。
2. 告诉备用服务器，前面的服务器在哪。
3. 可选择地，调整容错转移反应时间，这对两个服务器都一样。

关于调整主要考虑的是希望服务器以 的频率来检查它们的端点状态，并且你希望的主动的容错转移的频率是怎样的。在我们的例子中，容错转移的默认时间是 2000 微秒。如果你减少它们，备用服务器将会更快地代替主人，但是如果前面的服务器恢复的话，它也可能被代替。例如，你可能已经把前面的服务器封装到一个当它崩溃的时候可以重新启动它的脚本中。在那种情况下，截止时间应该比需要重新启动前端服务器的时间更高。

为了恰当地在二进制星形对中恰当地工作，客户机应用程序应该：

1. 知道两个服务器的地址。
2. 尝试连接前面的服务器，并且，如果失败了的话，连接到备用服务器。
3. 检测一个失败的连接，典型的应用是心跳。
4. 尝试重新连接到前面的，然后连接到后向的，之间的延迟时间至少要达到服务器容错转移的时间。
5. 重新创建它们对服务器要求的所有的状态。
6. 如果消息需要是可靠的的话，重新传输在容错转移的时候丢掉的消息。

它不是繁琐的工作，并且我们经常把它封装到一个对实际终端用户应用程序隐藏的应用程序中。

以下是二进制星形模型的主要的限制：

- 一个服务器进程不能属于超过一个二进制星形对。
- 一个前端服务器只能有一个备用服务器，不能有更多的了。
- 当备用服务器处于奴隶模式的时候不能做有用的工作。
- 备用服务器必须有能力处理所有的应用程序任务。
- 在运行时，容错配置不能被修改。
- 客户机应用程序需要做一些工作以从容错转移中获益。

防止裂脑综合症

“裂脑综合症”是当一个集群的不同部分同时都认为它们是主人。它阻止应用程序停止相互看到对方。二进制星形模式对于检测和删除脑裂有一种算法，基于三种方式定义机制（直到只有当获得应用程序的连接请求以后才会决定变成一个主人，并且它不能看到它的端点服务器）。

然而，仍然有可能设计一个网络来愚弄这种算法。一个典型的场景是一个二进制星形对分布在两个建筑之间，每个建筑都有一系列的应用程序，并且在两栋建筑之间有单一的网络连接。打开这个连接就会创建两套客户机应用程序，每套有二进制星形对的一半，并且每个容错转移服务器都会变成主动的。

为了阻止脑裂状态，我们必须用专门的网络链来连接二进制星形对，我们只需要简单地把它们两插入到相同的开关，或者更好，我们直接在两个机器之间使用交叉光纤。

我们不能把二进制星形结构分离成两部分，每部分由一组应用程序。因为这是通常网络结构的类型，在样的例子中，我们应该使用联合，而不是高可靠性容错转移。

一个适宜的狂妄的配置应该使用两个私有的集群交叉连接，而不是单一的。除此之外，集群的网络卡应该与消息输入/输出的网络卡不同，可能在在服务器硬件上，在不同的 PCI 方式下。分开可能来自网络网络的失败和在集群内的失败。网络端口有相对高的失败率。

二进制星形实现

不需要更多的力气，这就是二进制星形服务器的实现：

bstarsrv: Binary Star server in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这是客户机：

bstarcli: Binary Star client in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

为了测试它，以任意顺序启动服务器和客户机：

```
bstarsrv -p    # Start primary
bstarsrv -b    # Start backup
bstarcli
```

你可以通过关闭前面的服务器来引发容错转移，并通过重新启动前面的服务器和结束备用服务器来实现恢复。注意客户机是怎样决定容错转移和恢复的。

图表显示了有限的状态机器。用绿色标注的状态接收客户机的请求，粉红色标注的状态拒绝接收。事件是端点的状态，因此“端点积极的”意味着别的服务器已经告诉我们它是主动的。“客户机请求”意味着我们已经接收到一个客户机请求。“客户机选择”意思是我们接收到一个客户机请求并且我们的端点对两个心跳是非主动的。

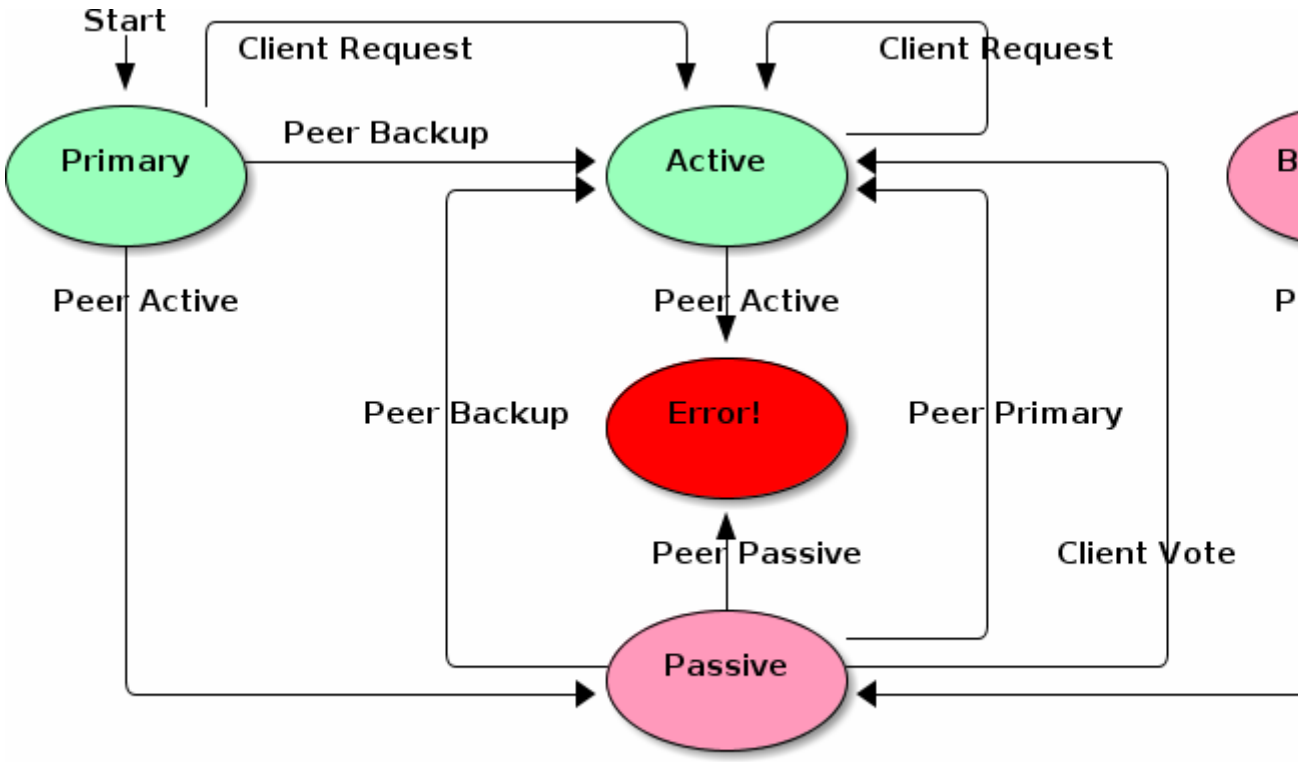

Figure 64 – Binary Star finite state machine

注意，对状态调换，服务器使用 PUB-SUB 套接字。别的套接字连接都不会生效。如果没有端点准备接收一个消息，PUSH 和 DEALER 套接字将会阻塞。如果端点消失又重新回来，PAIR 套接字不会重新连接。ROUTER 套接字在它发送消息之前需要知道端点的地址。

二进制星形反应器

二进制星形模型很有用，并且可以作为可重复使用的反应器类通用。用 C 语言，我们封装 libzapi zloop 类，用别的语言花费可能不同。这是 C 语言编写的二进制星形模型：

```
//  Create a new Binary Star instance, using local (bind) and
//  remote (connect) endpoints to set-up the server peering.
bstar_t *bstar_new (int primary, char *local, char *remote);

//  Destroy a Binary Star instance
void bstar_destroy (bstar_t **self_p);

//  Return underlying zloop reactor, for timer and reader
//  registration and cancelation.
zloop_t *bstar_zloop (bstar_t *self);
```

```
//  Register voting reader
int bstar_voter (bstar_t *self, char *endpoint, int type,
                 zloop_fn handler, void *arg);

//  Register main state change handlers
void bstar_new_master (bstar_t *self, zloop_fn handler, void *arg);
void bstar_new_slave (bstar_t *self, zloop_fn handler, void *arg);

//  Start the reactor, ends if a callback function returns -1, or the
//  process received SIGINT or SIGTERM.
int bstar_start (bstar_t *self);
```

这是类的实现：

bstar: Binary Star core class in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

它为我们提供了服务器的主要程序：

bstarsrv2: Binary Star server, using core class in C

代理可靠性（自由职业模式）

当我们经常把 0MQ 解释为“非代理消息系统”的时候，集中这么多精力在基于代理的可靠性上看起来很有挖苦意味。然而，在发送消息的过程中，如在现实生活中一样，中间人同时是负债方和受益方。在实际中，在实际中，很多消息结构从分布式和代理的混合消息结构获益。当你能够自由地决定你想做什么交易的时候，你获得最大的收益。这就是为什么我可以驾车 10 千米到一个批发商那买为一个聚会买 5 像啤酒的原因，但是我也可以走十分钟到一个角落的商店买一瓶来供晚餐。对于实际的经济效益，我们高度的与环境相关的时间，能量和成本的估计很重要。并且对于最优的基于消息的结构，它们很重要。

0MQ 为什么不实现以代理为中心的结构，虽然它为你提供了编译代理的工具，又叫做设备，并且到现在为止我们已经建立了十二个左右的代理。因为我们仅仅是为了练习。

因此，我们将会以毁坏到现在为止我们已经建立的基于代理的可靠性来结束本章，并且转到分布式的端到端结构，我把它叫做自由职业模式。我们的用例将是一个名称解析服务。这是 0MQ 结构常有的问题：我们怎样知道连接到的终端？用代码硬编码 TCP/IP 地址会非常地脆弱。想象一下，在你使用的个人电脑或者移动电话上，你必须手动配置你的网页浏览器，为了知道“gool.com”是” 74.25.230.82 “。

一个 0MQ 名称解析服务（如我们将会进行的简单的实现）必须：

- 把一个逻辑名字至少分解为一个绑定的终端，和一个连接的终端。实际的名称解析服务会提供多个绑定终端，并且可能也会提供多个连接终端。

- 允许我们管理多个平行的环境，即，测试和产品之间不用修改代码。
- 可靠，因为如果它不可靠，应用程序将不能连接到网络。

从某些点来看，方一个名称解析服务器在面向服务的管家婆代理后事聪明的。然而，让名称接续服务作为一个客户机可以直接连接的服务器会更简单。如果我们正确做到了，名字解析服务编程唯一的需要在我们的代码或 config 文件中硬编码的全局网络终端。

我们致力于处理的失败的类型是服务器崩溃和重新启动，服务器忙循环，服务器超载，和网络问题。为了得到可靠性，我们将会创建一个名字解析服务器池，这样如果一个崩溃或者离开了，客户机可以连接到另一个，等等。在实际中，两个就足够了。但是对于这个例子，我们将会假设这个吃可以任意大。

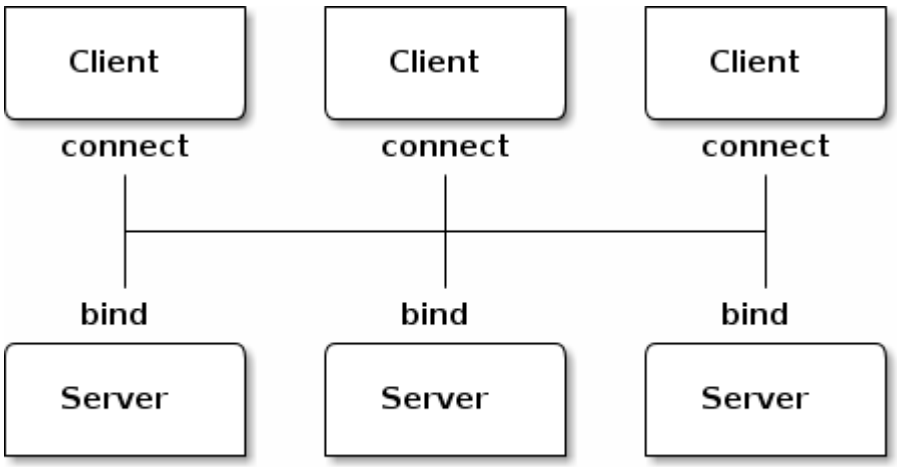

Figure 65 – The Freelance Pattern

在这种结构中，一大组客户机直接连接到一小组服务器。服务器绑定到它们各自的地址。它与工作线程连接到代理的基于代理的方法如管家婆模式有本质的不同。对于客户机来说，这有两个选择：

- 客户机可以使用 REQ 套接字和懒海盗模式。简单，但需要在愚蠢地反复重新连接死掉的服务器的基础上增加一点聪明的元素。
- 客户机可以使用 DEALER 套接字和爆破请求（它会在所有连接的服务器之间负荷均衡）直到它们获得一个请求。很直接，但是不优雅。
- 客户机可以使用 ROUTER 套接字，这样它们就可以寻址专门的服务器。但是客户机怎样知道服务器套接字的身份？或者服务器首先叫客户机（复杂），或者每个服务器必须使用一个硬编码，所有客户机都知道的身份（令人讨厌的）。

模式一-简单的重新尝试和容错转移

因此我们的项目看起来提供了：简单，直接，复杂，或者讨厌。让我们以简单开始，然后算出 kinks。我们采用懒海盗模式并且重新书写它以便能与多个服务器终端一起工作。首先打开服务器，指定一个绑定终端作为参数。运行一个活几个服务器：

flserver1: Freelance server, Model One in C

Lua | Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

然后打开客户机，指定一个或者多个连接终端作为参数：

flclient1: Freelance client, Model One in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

例如：

```
flserver1 tcp://*:5555 &
flserver1 tcp://*:5556 &
flclient1 tcp://localhost:5555 tcp://localhost:5556
```

当基本的方法是懒海盗模式，客户机的目的仅仅是得到一个成功的应答。它有两种技术，取决于你是运行一个单一的服务器还是多个服务器：

- 对于单一的服务器，客户机会重新尝试几次，事实上作为懒海盗。
- 对于多个服务器，客户机会尝试每个服务器最多一次，直到它接收到一个应答，或者尝试了所有的服务器。

它解决了懒海盗的主要的弱点，即它不能容错转移到备用/替换的服务器。

然而在实际的应用程序中，这样的设计不会工作的很好。如根据哦我们连接了很多的套接字，并且我们前面的名字解析服务器下线了，每次我们都必须处理这个痛苦的截至时间。

模式二-凶残的鸟枪杀戮

让我们的客户机转为使用 DEALER 套接字。在这我们的目标是在最短的可能时间内确定我们得到了一个应答，而不管前面的服务器是否关闭。我们客户机采用以下的方法：

- 我们设置事件，连接到所有的服务器。
- 当我们有一个请求的时候，我们爆破它我们拥有的服务器那么多次。
- 我们等待第一个应答，并处理它。
- 我们忽略任何别的应答。

在实际中，当所有的服务器都在运行的时候，会发生什么呢？0MQ 会分发请求，这样每个服务器都可以获得一个请求，并且发送一个应答。当任何一个服务器下线，并且断开连接，OMQ 将会把这个请求分发给剩下的服务器。这样一个服务器可能在某种情况下不止一次获得相同的请求。

对客户机来说更厌烦的是我们将获得多个回来的应答，但是不能确保我们能够获得应答的数量。请求和应答都能够丢失（即，当处理一个请求的时候服务器崩溃了）。

因此，我们必须给请求编号，并且忽略任何与请求号不匹配的应答。我们模式一的服务器会工作，因为它是一个应答服务器，但是对于理解来说偶然不是原因。因此我们将会构造模式二的服务器，返回一个带有内容“OK”的编号的应答。我们将使用由两部分组成的消息，一个序列号和消息体。

打开服务器一次，或者更多次，每次指定一个绑定的终端：

flserver2: Freelance server, Model Two in C

Lua | Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

然后打开客户机，以参数的形式指定连接的终端：

flclient2: Freelance client, Model Two in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

关于这个代码的注意事项：

- 客户机是以一个漂亮的小的基于类的 API 来构造的，创建 0MQ 背景和套接字以及与服务器会话的工作隐藏在它之下。
- 如果在几秒内客户机没有发现任何有反应的服务器，它将会放弃。
- 客户机必须创建有效的 REP 封装，即，增加一个空消息部分到消息体的前面。

客户机发送 10000 个名称解析请求，并计算平均的成本。在我的测试机上，与一个服务器会话花费大概 60 微秒。与三个服务器会话，大概要 80 微秒。

因此我们的鸟枪方式的优点和缺点是：

- 优点：它简单，容易构造和理解。
- 优点：它进行容错转移，并且工作的很快，只有有至少一个服务器运行。
- 缺点：它产生了冗余的网络交通。
- 缺点：我们不能为我们的服务器划分优先顺序，即，先是前面的服务器，再是第二个服务器。
- 缺点：服务器每次最多只能处理一个请求。

模式三-复杂和厌恶的

鸟枪方式看起来好得不真实。让我们科学地实现所有选择。我们将会开发复杂并且让人厌恶的选择，只是它只是为了最终证明我们更喜欢直接的方式。这是我的生活故事。

通过转换到 ROUTER 套接字，我们可以解决客户机的主要的问题。它允许我们发送请求到专门的服务器，而不是我们知道的已经死掉的服务器，并且做到我们希望的小巧。我们也能通过切换到 ROUTER 套接字解决服务器主要的问题。

但是，在两个传输套接字之间实现 ROUTER-到-ROUTER 是不可能的。两边都产生一个身份（在会话的它们自己的那边），只有当它们接收到第一个消息，并且在它们接收到一个消息之前没有哪个能与对方。走出这个难题的唯一方式是欺骗，在一方使用硬编码身份。在客户服务器例子中，正确的欺骗方式是客户机知道服务器的身份。反之亦然的话将会十分危险，达到复杂和厌恶的顶端。很容易导致灭绝性的独裁者，对于软件来说很恐怖。

我们使用连接终端作为身份，而不是另一个管理理论。这是一个两边都知道的字符串，而不需要比鸟枪模式知道更多的东西。它是一种卑鄙但有效的连接两个 ROUTER 套接字的方式。

记住，0MQ 身份是怎样工作的。ROUTER 套接字在它绑定到它的套接字之前设置一个身份。当一个客户机连接的时候，在任意一方发送一个实际的消息之前，它们进行很少的交换身份的交流。客户机 ROUTER 套接字没有设置身份，发送一个空身份到服务器。服务器为客户机产生一个随机的 UUID，作为它自己使用。服务器发送它的身份（我们已经同意的终端字符串）到客户机。

这就意味着一旦连接建立，我们的客户机能够路由一个消息到服务器（即，发送到它的 ROUTER 套接字，以身份的形式指定服务器终端）。它不是在调用 zmq-connect 后立即，但是在此后一些随机的时间。这仍然存在一个问题，我们不知道什么时候服务器真的可用并且完成了它的连接交流。如果服务器事实上已经在线，它会是几微秒。如果服务器不在线，并且系统管理程序也在休息，它可能会是一个小时。

这里有一个矛盾。我们不知道什么时候服务器连接上并且能够工作。在自由职业模式中，不像在这章的前面我们看到的基于代理的模式，在与它会话之前服务器都是安静的。因此，在服务器告诉我们它在线之前，我们都不能与一个服务器会话，而我们要先与它会话。

我的解决方式是混合一个来自模式 2 的鸟枪模式，意思是我们将会我们会射击任意我们能够射击的东西，如果有什么在移动的话，我们知道它是活着的。我们不会发生真实的请求，但是会发送一种心跳。

为了重新把我们带入协议的领域，这里定义了自由职业客户机和服务器是怎样交换 PING-PONG 命令，和请求应答命令。

- http://rfc.zeromq.org/spec:10

它作为服务器实现很短，并且很甜。这是我们的应答服务器，模式三，现在叫做 FLP.

服务器的模式三只有少许的不同：

flserver3: Freelance server, Model Three in C

Lua | Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | PHP | Python | Ruby

然而，自由职业客户机已经变得大了。对于真相来说，它分离成例子程序和一个处理复杂工作的类。这是顶级应用程序：

flclient3: Freelance client, Model Three in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这是它的客户机 API,几乎和管家婆代理一样大和复杂：

flcliapi: Freelance client API in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这个 API 应用是相当复杂的，并且使用了一些我们在前面见过的技术：

异步代理类

客户机 API 由两部分组成，一个运行在应用程序线程的同步的‘flciapi’类，和一个运行在后台的异步代理类。Flcliapi 和代理类通过进程间套接字相互会话。所有的 0MQ 部分（如创建和终止一个背景）都隐藏在这个 API 中。代理事实上像一个微型的代理，在后台与服务器会话，这样，当我们构造一个请求，它会做最好的努力来传送到一个它认为可靠的服务器。

耐心的连接

ROUTER 套接字有静静地扔掉它不能路由的消息的特性。这就意味着，如果你连接一个客户机到服务器，ROUTER-到-ROUTER,然后快速地尝试发送一个消息，它不会工作。当一个应用程序起初连接到一个服务器的时候，Flcliapi 类会有一个短暂的休眠。此后，因为有持久套接字，0MQ 从来不会把消息扔到它看到了的服务器，即使服务器离开了。

平安静

对于一个死掉的消息，0MQ 会无限地排队消息。因此，如果一个服务器反复地 PING 一个死掉的服务器，但那个服务器又重新活过来后，它将会一次性获得所有的 PING 消息。当服务器回来的时候，我们计算 0MQ 处理的发送就到 PING 消息的持久套接字，而不是继续 PING 一个我们知道已经下线了的消息。一旦服务器重新连接，它将会获得来自所有连接到它的客户机的 PING，并且这些客户机将会意识到它又重新活过来了。

无票的选票触发器

在之前的选票池，我们总是使用一个固定的选票间隔，例如，1 秒，这样很简单，但是对电力敏感的客户机来说还不够聪明，如笔记本，或者移动电话，它们会唤醒 CPU 的成本电力。为了玩玩，和帮助节约环境，代理使用了无票的触发，它计算基于下一个截至时间的选票延迟。正确的实现方法是保持一个有序的截止时间的列表。我们检查所有的截止时间，并计算知道下一个的选举延迟。

结论

在这一章我们看到了各种可靠的请求应答机制，每种都有一定的代价和利益。例子代码可以作为真实的使用，虽然它不是最优的。对于所有不同的模式，两种杰出的模式是基于代理可靠性的管家婆模式,和基于非代理可靠性的自由职业者模式。

第五章 高级的发布-订阅

在第三章和第四章，我们看到了高级的 0MQ 请求应答模式。如果你能够把它们完全消化，那么恭喜你。在这一章中，我们集中在发布-订阅，和扩展的的 0MQ 核心发布-订阅模式，在性能，可靠性，状态分布和安全方面，它是高级的模式。

我们会涉及到：

- 怎样处理非常慢的订阅方（自我毁灭的蜗牛模式）。
- 怎样设计告诉的订阅方（黑色盒子模式）。
- 怎样建立一个共享的关键字贮藏（克隆模式）。

慢订阅者检查（自我毁灭的蜗牛模式）

在实际生活中，当你使用发布-订阅模式的时候你通常会遇到的问题是订阅方很慢。在理想的情况下，我们的数据流全速地从发布方到订阅方。在实际中，订阅方应用程序通常是用判断的语言写的，或者仅仅是做很多工作，或者写的不好不容易扩展，这样它们就不能跟上订阅方。

我们怎样处理一个慢的订阅者？完美的补救办法是让订阅方变得更快，但是那样可能会花费任务和时间。一些经典的处理慢订阅方的策略是：

- **在发布方排队消息。**这就是在我没有读我的 email 前几个小时，Gmail 做的事情。但是在大量消息发送的时候，逆向地排队会使发布方用完内存并崩溃掉。特别是在有很多的订阅方并且由于性能的原因不可能清除到硬盘的情况下。
- **在订阅方排队消息。**这样更好，并且在网络给力的情况下，这是 0MQ 默认的情况。如果有某个将要用完内存并崩溃掉，它将会是订阅方而不是发布方，它是公平的。这对于订阅方不能保持一会的情况来说很好，但是在流慢下来的时候能够更上。然而，它很慢，通常情况下没有答案。

- **一定时间后停止排队新消息。**这就是没有 7.555GB 的空间的情况下，我的邮箱溢出它的 7.554GB 的空间 Gmail 做的事情。新消息会被拒绝或者丢掉。从发布方来说，这是一个很大的策略，这也是在发布方设置一个高水位标识的时候 0MQ 做的事情。然而，它仍然没有帮助我们不就慢订阅方。现在，我们仅仅是在我们的消息流中获得了一个缺口。
- **断开慢订阅方的连接。**这就是当我有两周没有登录的时候 Hotmail 做的事情，这就是为什么我有了第十五个 Hotmail 账号。它是一个漂亮的直接的策略来强迫订阅方关注，并且应该比较完美，但是 0MQ 不会这样做，没有办法把它放到顶端，因为订阅方对发布方应用程序是不可见的。

哪个经典的模式都不适用。因此我们要有创造力。而不是断开发不方的连接，让我们说服订阅方关闭自己。这就是自杀式的蜗牛模式。当一个订阅者检测到它运行的很慢（其中“非常慢”可能是一个配置选择，它真的意味着“太慢了，如果你总是在这，请大声说话让我知道，这样我就能够补救！”），它粗声说话并且死掉。

订阅方怎样能够检测到它呢》一种方法是按顺序排列消息（为它们按顺序编号），并在发布方使用 HWM。现在，如果订阅者检测到一个间断，（即编号是不连续的），它知道有什么东西出错了，然后我们把 HWM 转到“如果你遇到它的话粗声说话并死掉”级。

对于这种解决方法，这里有两个问题。一个是，如果我们有很多发布方，我们怎样给消息按顺序编号？解决方法是给每个发布方一个特别的 ID，并把它增加到序号中。第二，如果订阅方使用 ZMQ-SUBSCRIBER 滤波器，它们将会获得定义的间断。我们精确的序列号将没有意义。

一些用例不会使用滤波器，排序会帮它们做工作。但是一个更通常的解决方法是发布方用时间标志每个消息。当一个订阅方获得消息，它检查时间，如果间隔超过一秒，它就触发“粗声说话并死掉”事件。可能首先会向操作员控制台发送一个粗历的叫声。

自杀式蜗牛模式主要工作在订阅者有它们自己的客户机和服务水平协议和需哟啊保证确定的最大延迟的情况。中止一个订阅可能看起来不像保证最大延迟的有助的方式，但是它是申明的模式。今天中止，问题会得到补救。允许之后的某天顺流而下，这个问题可能导致更大的损坏并且会花费更大的时间来出现在雷达上。

这是自杀式蜗牛模式的最小例子：

suisnail: Suicidal Snail in C

C++ | Lua | PHP | Ada | Basic | C# | CLisp | Erlang | Go | Haskell | Java | Objective-C | ooc | Perl | Python | Ruby

关于这个代码的注意事项：

- 这的消息由现在的系统时钟作为微秒的编号。在实际的应用程序中，你至少有一个时间标志的消息头，和一个带数据的消息体。

- 这个例子的订阅和发布在一个例子中以两个线程的形式进行。在实际中它们可能是分离的进程。使用线程是为了论证的方便。

高速订阅（黑盒子模式）

对于发布订阅，一个通常的用例是分发大的数据流。例如，来自股票交易的市场数据。典型的设置应该是一个订阅连接到一个股票交易，处理报价，并把它们发送到一些订阅方。如果有少数的订阅者，我们会使用 TCP.如果有大量的订阅者，我们会使用可靠的多播，即，pgm.

让我们假设我们的馈送平均每秒有 100000100 字节的消息。那是个典型的频率，在滤波市场数据后，我们不需要发送到订阅方。现在我们决定记录一天的数据（可能 8 小时内有 250GB 的数据），然后把它重新放到一个模拟的网络，即，即以小组的订阅者。当每秒有 100K 的消息对 0MQ 应用程序来说很简单，我们需要重放的更快。

因此，我们用一组机子来创建我们的结构，一个提供给发布方，然后每个订阅方一个机子。这有很好规定的机子，8 个核，十二个是给发布方的。（如果你在 2015 年读这个 guide，它是这个 guide 预定的完成时间，请在所有号码前加上 0.）

我们把数据放入我们的订阅方时，要注意两件事：

1. 我们用消息即使做最轻任务的时候，当不能跟上发布方的时候，它会使我们的订阅方变慢。
2. 大约每秒 6M 的消息，在发布方和订阅方我们都达到了顶层，即使在仔细的优化和 TCP 调整后。

第一件我们必须做的事是把我们的订阅分解成多线程设计，这样当另一个线程在读消息的时候，我们就可以在一系列线程中处理带消息的任务。通常，我们不希望用相同的方式处理任何消息。然而，订阅方将会滤掉一些消息，可能是通过前缀关键字。当一个消息匹配某个标准，订阅者将会调用一个工作线程来处理它。以 0MQ 的形式，这就意味着发送一个消息到工作线程。

因此，订阅方看起来有点像一个队列设备。我们能够使用不同的套接字来连接订阅方和工作线程。如果我们假设一种方式的交通，并且所有的工作线程都是等同的，我们可以使用 PUSH 和 PULL 套接字，并且服了所有 0MQ 的路由任务。这是最简单和最快的方法：

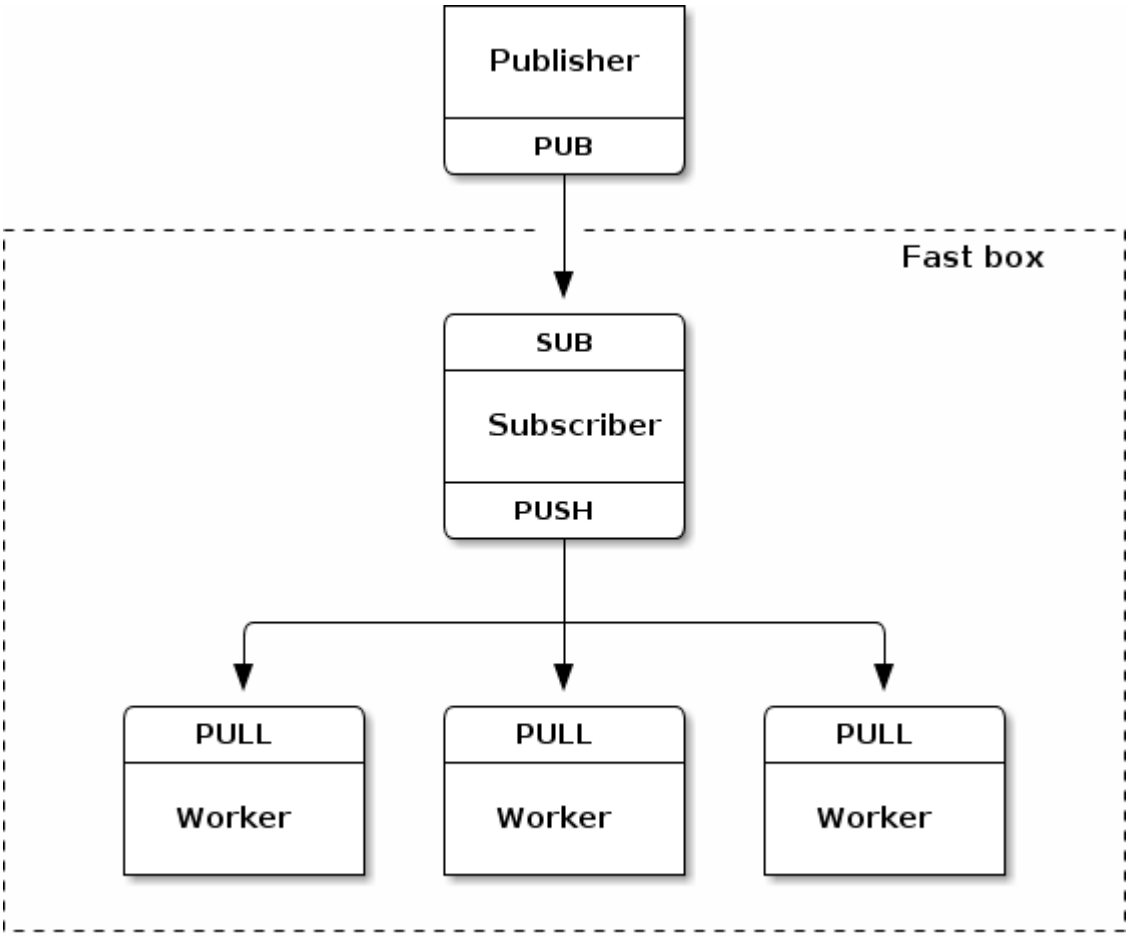

Figure 66 – Simple Black Box Pattern

订阅者通过 TCP 或者 PGM 与发送者会话。订阅者与它的工作线程会话，所有的工作线程都在相同的进程中，通过的 inproc 协议。

现在，为了打破那个上线，订阅者线程会遇到 100%的 CPU，并且因为它是一个线程，它不能使用超过一个核。单线程常常会达到上限，不管是每秒 2M,6M,或者更多的消息。我们希望通过多个线程来分离这些可以平行运行的任务。

这种在这工作的被高性能产品使用的方法叫做 sharding,意思是我们把任务分解成平行的和独立的流。例如，一半的主题关键字在一个流中，一半在另一个流中。我们可以使用很多流，但是除非有自由的核，否则性能不会被扩展。

让我们看看怎样变成两个流：

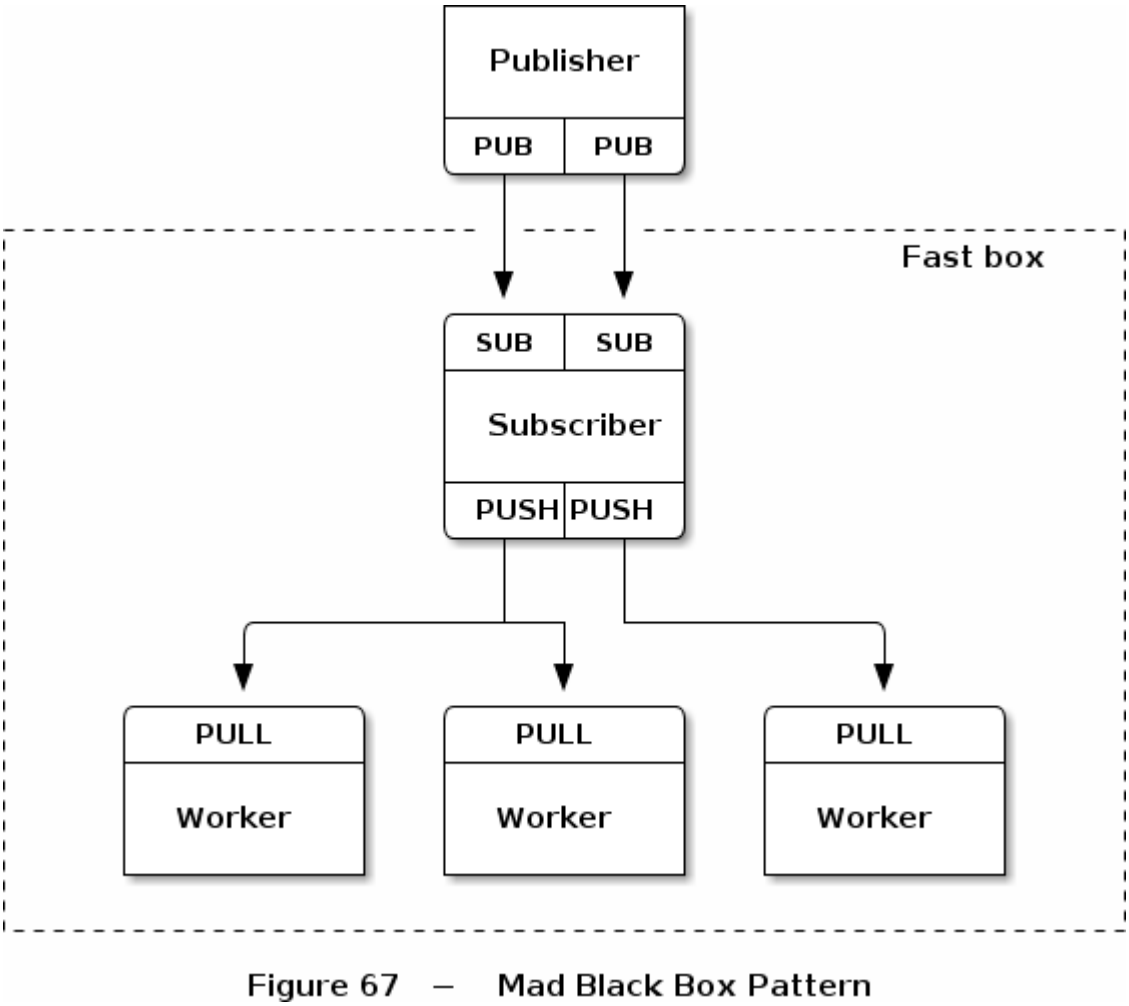

Figure 67 – Mad Black Box Pattern

有两个流全速工作，我们应该像下面这样配置 0MQ：

- 两个输入/输出线程，而不是一个。
- 两个网络接口（NIC），每个订阅者一个。
- 每个输入/输出线程绑定到专门的 NIC。
- 两个订阅线程绑定到专门的核。
- 两个 SUB 套接字，每个订阅线程一个。
- 剩下的核分配给工作线程。
- 工作线程连接到两个订阅者的 PUSH 套接字。

理想情况是，在我们的结构中，先吃呢个的数量不会超过核的数量。一旦我们创建比核更多的线程，在线程之间就会产生竞争，并且返回值会逐渐缩小。这样没有好处，例如，在创建更多的输入/输出线程的时候。

共享的关键值（克隆模式）

发布订阅像一个无线广播，在你加入之前你会错过所有的东西，并且，之后的获得多少的信息取决于你的接收量。令人吃惊的是，对于追求完美的工程师来说，这种模式很有用，并且很广泛，它把完美匹配到了现实中消息的分发。考虑 Facebook 和 Twitter，这

BBC 世界服务器，和这个输出结果。

然而，在很多情况下，更可靠的发布订阅方式会很有用，如果我们能够办到的话。如我们为请求应答所做的一样，让我们以什么能够变坏的方式定义“可靠性”一下是关于发布订阅的典型的问题：

- 订阅者加入太迟，因此丢掉了服务器已经发送的消息。
- 订阅者连接的很慢，就可能在这个时间间隔内丢掉消息。
- 订阅者离开，在它们离开的时间里会丢失消息。

通常较少，我们看到这样的问题：

- 订阅方崩溃，又重新启动，并且丢掉了它们已经接收到的任何数据。
- 订阅者取回消息很慢，因此队列建立起来，并且很快就溢出。
- 网络超载，并丢掉数据（特别是对于 PGM 来说）。
- 网络变得非常慢，因此，发布方的队列溢出，发布方崩溃。

更多的东西可能会出错，这些事我们在现实系统中看到的典型的情况。

我们已经解决了其中的一些问题，例如慢订阅者，用的是自杀式蜗牛模式来解决的。但是，剩下的问题，应该构建一个漂亮的，能够被发布-订阅重用的框架。

难点在于，我们不知道我们的目标应用程序想用它们的数据来做什么。它们会过滤这些消息，然后只处理这些消息的一个子集吗？它们会记录这些数据到某个地方以备以后使用吗？它们会分发这些数据到更远的工作线程？有几十种可能的情景，每种都有它自己关于可靠性，以及为提高性能应该做的努力的定义。

因此，我们会建立一种抽象的，只要我们实现一次，就可以被很多应用程序重复使用。这种抽象是共享的关键字贮藏，它存储在一系列有专门的关键字编的索引的块中。

不要把它与分布式哈希表混淆，在分布式网络中，它会解决关于连接网络的更大的问题，也不要与表现得像非-SQL 数据库的分布式关键字表相混淆。我们所要建立的是一个把内存状态从一个服务器克隆到一组客户机的系统。我们希望：

- 在任何时候允许一个客户机加入网络，并可靠地获得当前的服务器状态。
- 允许任何客户机更新关键字缓冲区（插入新的关键字对，更新已经存在的，或者删除它们）。
- 可靠的传播改变，并且以最小的延迟来做到。
- 处理很大数量的客户机，即，数万，或者更多。

克隆模式的关键部分是客户机会和服务器顶嘴，这在一个简单的发布-订阅对话中不存在。这就是为什么我使用术语“服务器”和“客户机”，而不是“发布者”和“订阅者”。我们会使用发布-订阅作为克隆模式的核，但是不是只有它。

分布式关键值更新：

我们会一步一步地开发克隆模式，一次解决一个问题。首先，让我们看看怎样从一组客户机到服务器来分发关键值更新。我们将会采用第一章的天气服务器，并以关键值对的形式重构它来发送消息。我们将会修改我们的客户机来把这些存储在一个哈希表中。

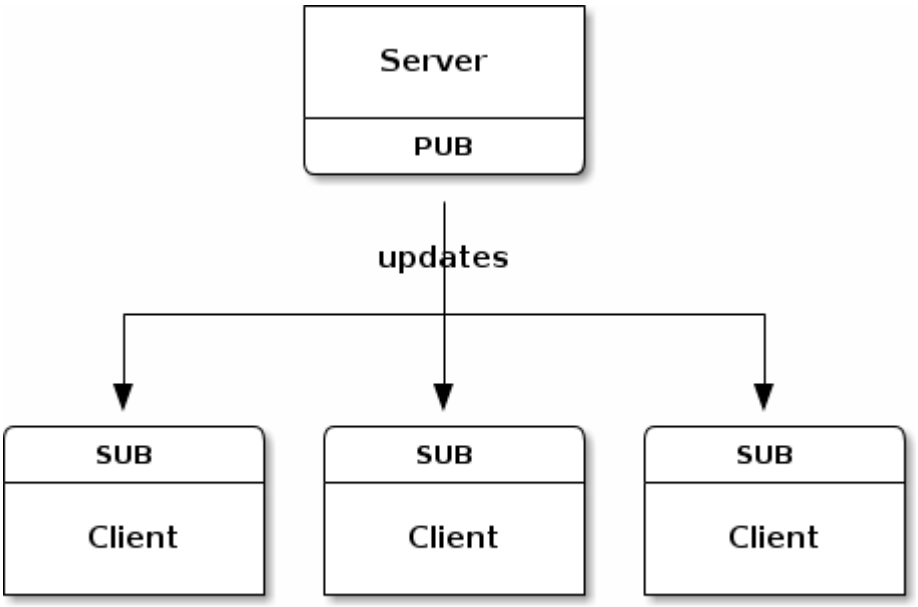

Figure 68 – Simplest Clone Model

这是服务器：

clonesrv1: Clone server, Model One in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这是客户机：

clonecli1: Clone client, Model One in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

关于这个代码的一些注意事项：

- 所有困难的工作都在 kvmsg 类中处理了。这个类与关键值消息对象一起做事，它们是分段的 0MQ 消息结构，有三个帧：一个关键字（一个 0MQ 字符串），一个序列号（64 位的值，以网络字节顺序），和一个二进制消息体（拥有所有别的东西）。
- 在服务器绑定到它的套接字后，会有 200 微秒的脉冲。这是为了阻止订阅者连接

到服务器套接字时丢掉消息的“慢加入综合症”。在之后的模式中，我们会删除它。

- 鉴于套接字的原因，我们在这个代码中会使用术语“发布”和“订阅”。这在我们有多个套接字做不同工作的时候会有帮助。

这是 kvsmg 类，是到目前为止能够工作的最简单的模式。：

kvsimple: Key-value message class in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

等会我们将会构造一个更先进的 kvmsg 类，以便能在实际应用程序中使用。

服务器和客户机都保存着哈希表，但是第一种模式只在所有的客户机在服务器之前打开的那种情况下才会正常工作，并且客户机从来不会崩溃。这不是“可靠性”。

得到一个简介：

为了允许随后的（或者再生的）客户机赶上服务器，必须要获得一个服务器状态的说明。就像我们把消息减少到“一个排序的关键值对”，我们可以把状态减少到一个“哈希表”。为了得到服务器状态，客户机打开一个 REQ 套接字，并直接向它请求。

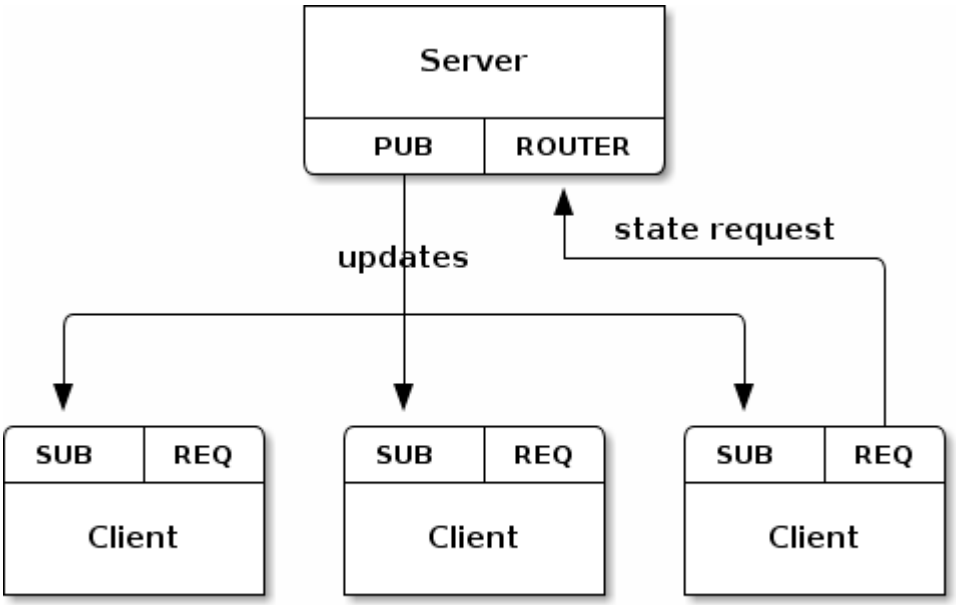

Figure 69 – State Replication

为了让它能够正常工作，我们必须解决时间掌握的问题。获得一个状态说明会花费一定的时间，如果说明大的话，可能时间还会相当的长。我们需要正确地把更新应用到说明。但是服务器不知道什么时候把更新发送给我们。一种方式是开始订阅，获得第一个更新，然后请求“更新 N 的状态”。这就要求服务器为每个更新存储一个说明，这不适用。

因此我们在客户机上实现同步化，方法如下：

- 客户机首先订阅到更新，然后进行一个状态请求。这会保证将会比他有的最新的更新更新。
- 客户机等待服务器返回它的状态，同时排队所有的更新。它会简单地做这些而不会读它们：0MQ 让它们在套接字队列中排队，因为我们没有设置 HWM.
- 当客户机接收到它的状态更新，它开始再一次读取更新。然而它会丢掉任何比它更旧的更新。因此，如果状态更新包括达到 200 的更新，客户机将会丢掉达到 201 的更新。
- 然后客户机客户机把它自己的更新应用到他自己的状态说明。

这是一种简单的开发 0MQ 自己内部队列的方式。这是服务器：

clonesrv2: Clone server, Model Two in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这是客户机：

clonecli2: Clone client, Model Two in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

关于这个代码的一些注意事项：

- 为了更简单的设计，服务器使用两个线程。一个线程产生随机的更新，第二个线程处理状态。两个线程通过 PAIR 套接字交流。你可能喜欢使用 SUB 套接字，但是你可能会遇到“慢连接”问题，订阅者在连接的时候可能会随机地丢掉一些消息。PAIR 套接字让我们能够直接地同步这两个线程。
- 我们在更新的套接字对中设置一个高水位线，因为哈希表的插入相对比较慢。没有它的话，服务器会用完内存。关于 inproc 连接，真实的 HWM 是两个套接字 HWM 的和，因此，我们可以在每个套接字上设置 HWM.
- 客户机真的很简单。用 C 语言，代码不足 60 行。很多困难的提高都在 kvmsg 类中，但是基本的克隆模式还是比它之前看起来的要简单。
- 我们不使用任何设想来连载这个状态。哈希表拥有一组 kvmsg 对象，服务器把它们作为一组消息发送到客户机请求状态。如果一次有多个客户机请求状态，每个都会获得不同的说明。
- 我们假设客户机有一个会话的服务器。服务器必须运行；我们不尝试解决服务器崩溃的问题。

现在两个程序不做什么实际的事情，但是，它们正确地同步状态。它是一个关于怎样混合不同模式的简洁的例子：inproc 撒谎国内的 PAIR，PUB-SUB,ROUTER-DEALER.

重新发布更新：

在我们的第二种模式中，改变到关键值的贮藏来自服务器自身。这是一个集中式模式，很有用，例如，如果我们有一个我们想要发布的本地贮藏在每个节点的中央配置文件。一个更有趣的模式处理来自客户机而不是服务器的更新。因此，服务器变成一个无状态的代理。它可以提供给我们一些好处：

- 我们不必再那么担心服务器的可靠性。如果它崩溃掉了，我们可以立刻开始一个新的，并为它提供新的值。
- 我们可以使用关键值贮藏来在两个动态端点之前分享知识。

来自客户机的更新经过客户机到服务器的 PUSH-PUSH 套接字流。

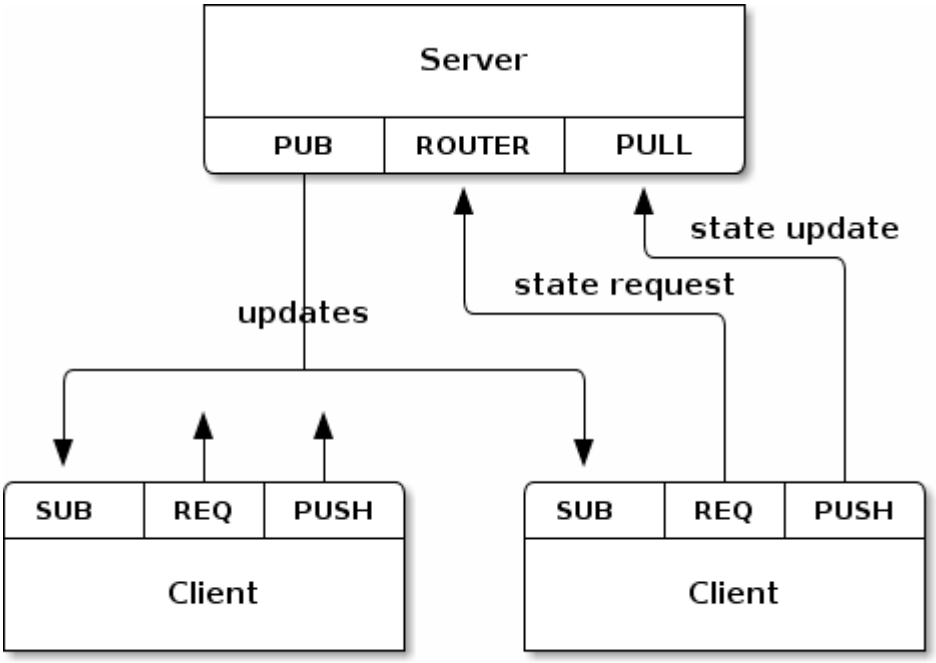

Figure 70 – Republishing Updates

我们为什么不允许客户机发送更新到别的客户机？当这样可以减少延迟的时候，它是给消息分配上升的专门的序列号变得不可能。服务器可以这样做。有更微妙的第二个问题。在很多应用程序中，跨很多客户机的更新数据有一个单一的顺序很重要。强迫所有的更新通过服务器，以确保当它们最后到达客户机的时候有相同的顺序。

由于有专门的序列顺序，客户机能够检测到最厉害的失败—网络阻塞和队列溢出。如果一个客户机发现它的输入消息有一个洞，它可以采取行动。它看起来是敏感的，客户机联系服务器，并请求丢掉的消息，但是在实际中，这样不适用。如果这里有漏洞，它们由网络压力导致的，并且给网络更多的压力，这样，事情会变得更糟。所有客户机能够做的就是警告它的用户“不能够继续了”，请停止，并且在手动检测出问题之前都不会停止。

我们现在就会在客户机端产生状态更新。这是服务器：

clonesrv3: Clone server, Model Three in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这是客户机：

clonecli3: Clone client, Model Three in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

关于这个代码的一些注意事项：

- 服务器崩溃到一个线程，它从客户机采集数据并且重新分发它们。对于输入的更新，它管理一个 PULL 套接字，对于状态更新管理一个 ROUTER 套机字，对于输出更新管理一个 PUB 套接字。
- 客户机使用一个简单的无票触发来每秒发送一个随机的更新到服务器。在实际中，更新会被应用程序代码驱动。

克隆子树

一个真实的关键值贮藏将会变得更大，客户机通常只对部分贮藏感兴趣。通过一个子树来工作相当的简单。当它进行状态请求的时候，客户机必须告诉服务器这个子树，并且当它订阅到更新的时候，它们必须指定相同的子数。

这是对于树的常规的句法。一个是“路径层次体系”，另一个是“主题树”。它们看起来像：

- 路径层次体系：“/some/list/of/paths”
- 主题树：“some.list.of.topics”

我们将会使用路径层次结构，并且扩展我们的客户机和服务器，这样一个客户机就可以和一个单一的子数一起工作。与多个子树一起工作不会困难多少，在这我们不会这样做，但是它是一个琐碎的扩展。

这是服务器，在模式三上进行了很小的变动：

clonesrv4: Clone server, Model Four in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这是客户机：

clonecli4: Clone client, Model Four in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

短暂的值

一个短暂的值是会动态终止的。如果你认为克隆被用在一个像 DNS 一样的服务器，那么短暂的值，那么短暂的值将会允许你实现动态的 DNS。一个节点加入网络，发布它的地址，并有规则地恢复它。如果节点死掉，它的地址事实上就删除了。

对短暂值的有用的抽象是把它们联系到一个“会话”，并且在会话终止的时候删除它们。在克隆模式中，会话应该由客户机定义，并且在客户机撕掉的时候会结束。

一个使用会话的更简单的选择是用“生存时间”来定义每个短暂的值，它告诉服务器什么时候终止这个值。然后客户机恢复这个值，否则，这个值就终止。

我将会实现那个更简单的模式，因为我们还不知道有必要弄一个更复杂的。在性能方面确实不同。如果客户机有很多短暂的值，在每个上设置 TTL 将会不错。如果客户机使用很多短暂的值，把它们来接到会话会更有效，并且一块终止它们。

我们需要一种方式来编码 TTL 到关键值消息。我们可以增加一个帧。对于使用一个帧作为属性的问题是，每次我们想要加入一个新的属性，我们都必须改变 kvmsg 类的结构。它违背了方便性原则。因此，让我们增加一个属性帧到消息，并且编码允许我们获得和放置属性。

接下来，我们需要一种方式来说“删除这个值”。到现在为止，客户机和服务器总是盲目地删除或者更新一个个新的指导它们的哈希表。如果值是空的，就意味着“删除了这个关键字”。

这是关于 kvmsg 类更完整的版本，它实现了一个属性帧（并且增加了一个 UUID 帧一会我们就会用到）。它也会通过把关键字从哈希表中删除来处理空值，如果有必要的话：

kvmsg: Key-value message class - full in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

模式五的客户机几乎等同于模式四。不同时你的朋友。它使用完全的 kvmsg 类，而不是 kvsimple,并且在每个消息上设置随机的‘ttl’属性（以秒计算）：

```
kvmsg_set_prop (kvmsg, "ttl", "%d", randof (30));
```

模式五的服务器完全改变了。不是投票循环，我们现在使用一个反应器。这样会让混合一个触发器和套接字事件更简单。不幸的是，在 C 语言中，反应器模式更累赘。在别的语言中，你的利益将会改变。但是，反应器看起来是变异一个更复杂的 0MQ 应用程序的更好的方式。这是服务器：

clonesrv5: Clone server, Model Five in C

克隆服务器可靠性

克隆模式一到五都相对地简单。我们将要进入令人不高兴的复杂的区域，到这我又喝了一杯浓咖啡。你将会注意到，做到可靠的消息发送真的很复杂，以至于在做之前你会问：“我们真的需要这样做吗？”如果你能够摆脱不可靠，或者足够好的可靠性，那么你能够在代价和复杂性方面获得很大的胜利。当然，你可能会不时地丢掉消息。它常常是一个好的折中方案。说到这，因为咖啡很好，我们还是来弄弄它吧。

但你在尝试模式三的时候，你会停止和重启服务器。它可能看起来像一个恢复，但是，它在把更新应用到一个空的状态，而不是合适的直接的状态。任何新加入网络的客户机都只会获得最新的更新数据，而不是所有的更新。因此，让我们设计出一个让克隆模式不去理会服务器失败的方案。

让我们列出我们希望能够处理的失败：

- 克隆服务器处理崩溃，并且会自动或者手动重启。进程丢掉它的状态并且必须从某个地方重新找回来。
- 克隆服务器机器死掉，并且在一个重要的时间掉线了。客户机必须切换到某处备用的服务器。
- 克隆服务器进程或者机器从网络上断开连接，例如，一个开关断了。它可能胡在某个时刻回来，但是，在此期间客户机需要一个备用服务器。

我们的第一步是增加一个另外的服务器。我们可以用第四章的二进制星形模式来把它们组织成前面的和备用的。二进制星形模式是一个反应堆，因此，我们把最新的服务器模式重构成一个反应堆模式很有用的。

我们需要确保前面的服务器崩溃的时候更新不会丢失。最简单的技术是把他们发送到两个服务器。

备用服务器可以表现为一个客户机，并且通过和所有客户机一样接受更新，以保证它的状态同步。它也能从客户机获得新的更新。它也能把这些存储到一个哈希表中，但是它只能保持它们一会。

模式六在模式五的基础上引进了这些改变：

- 对于客户机（到服务器）的更新数据，我们使用一个 pub-sub 流，而不是 push-pull 流。原因是，如果没有接受者的话，push 套接字会阻塞，并且它们实现循环赛，因此我们需要打开两个。我们会绑定服务器的 SUB 套接字，并连接客户机的 PUB 套接字到它们。这需要从一台客户机扇出到两台服务器。
- 我们给服务器（到客户机的）更新增加心跳，这样客户机可以检测到什么时候前面的服务器死掉。之后它就可以连接到备用的服务器。
- 我们利用二进制星形反应堆类连接两个服务器。二进制星形依赖于客户机通过发

送一个直接的请求到它们认为是“主人”的服务器来选举的。我们将会使用说明请求。

- 通过增加一个 UUID 域，我们让每个更新消息都是可辨别的。它由客户机产生，服务器以一个重新发布的更新的形式把它传播回去。
- 奴隶服务器保持一个挂起的从客户机接收的更新数据列表，但是不是来自主人服务器。或者是来自主人服务器的更新，而不是客户机。这个列表从最老的排到最新的，这样就很容易从头删除更新。

把客户机设计成一个有限的状态机器很有用。客户机的周期跨三个状态：

- 客户机打开并连接到它的套接字，然后请求一个来自第一个服务器的说明。为了避免请求风暴，它之后询问任何给定的服务器两次。一个请求可能丢失，那比较倒霉。两个都丢失了就是不小心。
- 客户机等待一个来自当前服务器的应答（说明数据），并且如果它获得了，它就存储起来。如果在某个截止时间内没有应答，它会转向另一个服务器。
- 当客户机得到它的说明，它等待并处理更新。如果在截至时间内，它没有得到来自服务器的任何数据，它会失败并转向另一个服务器。

客户机一直循环。有可能在启动或者失败期间，有的客户机尝试着与前面的服务器会话，而别的客户机尝试着与备用的服务器会话。二进制星形模式会处理这个，并且有希望做的很精确。（关于做这个设计的一个笑话是我们不能证明它们是对的，我们只能证明它们是错的。它就像一个年轻人从一座高高的大楼摔下。这么远，这么好，这么远，这么好。）

我们可以描绘客户机有限的状态机制。

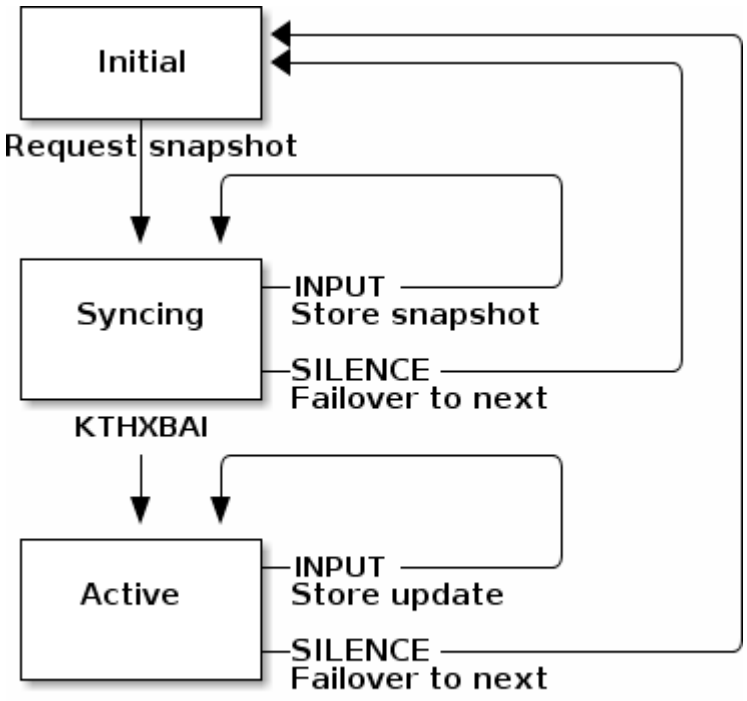

Figure 71 – Clone client FSM

故障切换这样发生：

- 客户机检测到前面的服务器不再发送心跳，因此判断它死掉了。客户机连接到备用服务器，并请求新的状态说明。
- 备用服务器开始接收来自客户机的说明请求，并且检测到前面的服务器已经离开，因此接任前面服务器的工作。
- 备用服务器应用它的挂起列表到它自己的哈希表，然后开始处理状态说明请求。

当前面的服务器重新上线，它将会：

- 作为一个奴隶服务器，兵器作为一个克隆客户机连接到备用服务器。
- 开始通过它自己的 SUB 套接字接收来自客户机的更新。

我们做一些假设：

- 至少一个服务器会运行。如果两个服务器都崩溃，我们丢掉所有的服务器状态，并且没法恢复。
- 多个客户机不同时更新相同的哈希关键字。客户机更新将会以不同的顺序到达两个服务器。这样，备用服务器可能以不同顺序使用来自它的挂起列表的更新，而不是像前面服务器所做的那样。来自一个客户机的更新总是会以相同的顺序到达两个服务器，因此它是安全的。

这是我们的高可靠性服务器对，使用二进制星形模式：

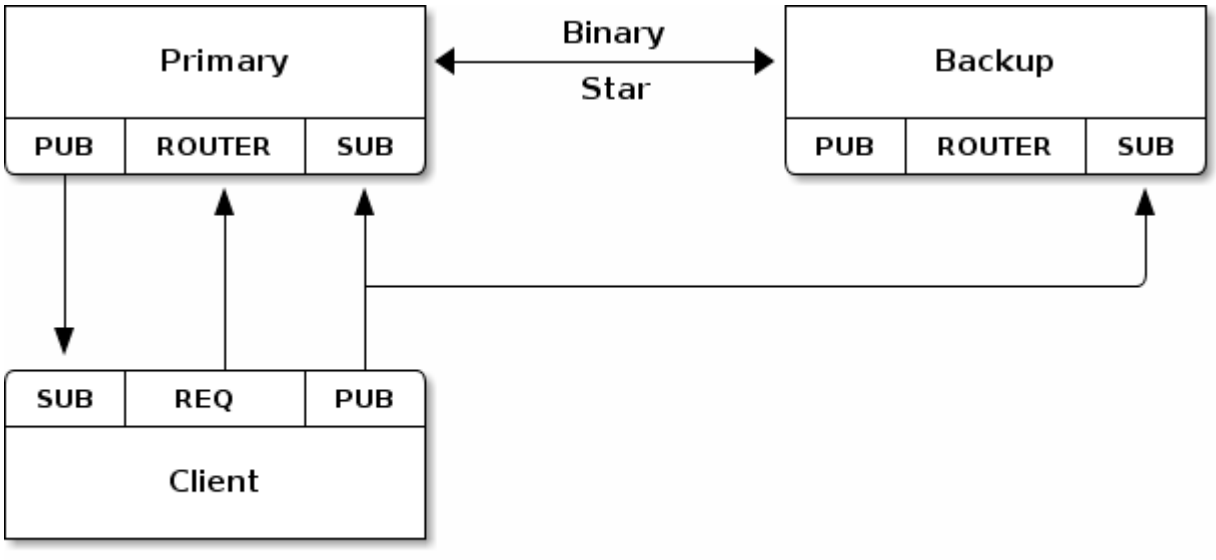

Figure 72 – High availability Clone Server Pair

建立它的第一步是，我们将会重构客户机为一个可重用的类。这有一点玩笑的意味（用 0MQ 编写异步的类像一种高雅的经历），但是主要是因为我们克隆模式对于插入到随机的应

用程序来说真的简单。因为回复力直接取决于客户机的活动能力。当有一个可重用的客户机 API 的时候，更容易保证它。当我们在客户机端开始处理故障切换，它确实变得有点复杂（想象一下混合一个自由职业者客户机到一个克隆客户机）。因此，可重用性哦。

我的设计方法是首先设计一个感觉正确的 API,然后实现它。因此，我们通过采用克隆客户机开始，然后把它重写，然后设定到假定的叫做克隆的 API 类。把随机代码转换为 API 意味着用应用程序定义一个聪明稳定和抽象的类。例如，在模式五中，客户机打开分离的套接字到服务器，使用在源文件中硬编码的终端。我们可以用三种方法构造一个 API，像这样：

```
//  Specify endpoints for each socket we need
clone_subscribe (clone, "tcp://localhost:5556");
clone_snapshot  (clone, "tcp://localhost:5557");
clone_updates   (clone, "tcp://localhost:5558");

//  Times two, since we have two servers
clone_subscribe (clone, "tcp://localhost:5566");
clone_snapshot  (clone, "tcp://localhost:5567");
clone_updates   (clone, "tcp://localhost:5568");
```

但是它既累赘又脆弱。这不是一个把一个设计用于应用程序的好的方法。今天，我们使用三个套接字。明天，两个或者四个。我们真的希望改变使用克隆类的每个应用程序吗？因此，为了隐藏这些细节，我们构造一个小的抽象，像这样：

```
//  Specify primary and backup servers
clone_connect (clone, "tcp://localhost:5551");
clone_connect (clone, "tcp://localhost:5561");
```

它的优点是简单，（一个服务器位于一个终端）但是对我们的内部设计有影响。我们现在需要把单一的终端转换为三个终端。一种方式是把“客户机与服务器在三个连续的端口会话”转换为我们的客户机-服务器协议。另一种方式是得到两个丢掉的来自服务器的终端。我们会采用最简单的方式，即：

- 服务器状态路由（ROUTER）是端口 P。
- 服务器更新发布（PUB）是端口 P+1.
- 服务器更新订阅（SUB）是端口 P+2.

克隆类和四章的 flcliapi 有相同的结构。它由两部分呢组成：

- 一个运行在后台线程的异步克隆代理。这个代理处理所有的网络输入/输出，实时地与服务器会话，而不管应用程序做什么。
- 一个运行在调用者线程的同步的“克隆”类。当你创建一个克隆对象，就会自地

发动一个代理线程，并且在毁坏一个可怜对象的时候也就终止了一个代理线程。

前一个类通过基于 inprocde ‘pipe’套接字与这个代理会话。用 C 语言，libzapi 线程层子啊它开始一个“连接线程”的时候自动为我们创建 pipe。这对于 0MQ 的多线程是一种自然的模式。

没有 0MQ，这种异步类的设计将会需要几周的艰难的工作。利用 0MQ，它只是一两天的工作。结果有点复杂，它的实际的运行为克隆协议提供了直接性。关于它是有一些原因的。我们把它转变为一个反应堆，但是它让它在应用程序中更难使用。因此，这个 API 看起来有点像一个与一些服务器神秘会话的关键值表。

```
clone_t *clone_new (void);
void clone_destroy (clone_t **self_p);
void clone_connect (clone_t *self, char *address, char *service);
void clone_set (clone_t *self, char *key, char *value);
char *clone_get (clone_t *self, char *key);
```

这是克隆客户机的模式六，它变成一个简单的在克隆类中使用的贝壳了。

clonecli6: Clone client, Model Six in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

这是实现的实际的克隆类：

clone: Clone class in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

最后，这是克隆服务器的第六种也是最后一种模式：

clonesrv6: Clone server, Model Six in C

Ada | Basic | C++ | C# | CLisp | Erlang | Go | Haskell | Java | Lua | Objective-C | ooc | Perl | PHP | Python | Ruby

主程序只有几百行代码，但是它花了一些时间才工作。为了精确，带着埋怨建立了模式六，并且大概花了一周“天哪，对于 guide 来说太复杂了”。我们组合很多好的东西在这个应用程序中。我们有容错转移，说明值，子树，等等。令我吃惊的是，这个诚实的设计很精确。但是写和调试这么多套接字流的细节是不一样的。这就是我怎样让它工作的：

- 通过使用反应堆（bstar,在 zloop 上）,它删除了很多来自代码的繁重的工作，让事情 变得更简单更容易理解。整个服务器以单线程方式运行，因此没有内部线程的问题。仅仅传送一个结构指针（self）到所有的处理者，处理者就可以很高兴地处理事情了。使用反应堆的一个边界效应是那些代码，松整合到一个选票循环中，

更容易重用。模式六的绝大部分来自模式五。

- 通过一部分一部分地建立它，并且在开始下一部分之前会让每部分工作的很好。因为有四个或者五个主要的套接字流，就意味着很多的调试和测试。我的调试仅仅是简单地把缓冲区打印到控制台（例如倒的消息）。在实际中，为了这种事打开一个调试器是没有意义的。
- 总是在 Valgrind 下测试，以至于我确定没有内存泄露。用 C 语言时，这是一个主要要考虑的问题，我们不能代理到某个垃圾采集者。使用恰当的，合适的抽象，像 kvmsg 和 libzapi 帮助很大。

我确信代码还有瑕疵，读者话可以花周末来调试并帮我校正。对于这种模式我很高兴能把它作为实际应用程序的基础。

为了测试第六种模式，打开前面的服务器和备用的服务器，和一组客户机，以任意的顺序。然后关闭并重新启动一个服务器，随机地，并持续这样做。如果涉及和代码准确，客户机将会持续获得来自任意当前主人服务器的相同的更新流。

克隆协议说明

在做了这么多的建立可靠的发布订阅模式后，我们想保障我们能够安全建立的应用程序来开发这个任务。一个好的开始时评论这个协议。它允许我们用另一种语言来实现，并允许我们在在纸上提高这个设计，而不是改代码。

这是 Clustered Hashmap 协议，它“定义一个集群范围内的关键值 hash 图，和分享跨一组客户机的机制。CHP 允许客户机用 hashmap 的子树工作，来更新值，并定义说明值。”

- http://rfc.zeromq.org/spec:12