ZMQ 指南

作者: Pieter Hintjens ph@imatix.com, CEO iMatix Corporation.
翻译: 张吉 jizhang@anjuke.com, 安居客集团 好租网工程师

With thanks to 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, Mihail Minkov, Jeremy Avnet, Michael Compton, Kamil Kisiel, Mark Kharitonov, Guillaume Aubert, Ian Barber, Mike Sheridan, Faruk Akgul, Oleg Sidorov, Lev Givon, Allister MacLeod, Alexander D'Archangel, Andreas Hoelzlwimmer, Han Holl, Robert G. Jakabosky, Felipe Cruz, Marcus McCurdy, Mikhail Kulemin, Dr. Gergő Érdi, Pavel Zhukov, Alexander Else, Giovanni Ruggiero, Rick "Technoweenie", Daniel Lundin, Dave Hoover, Simon Jefford, Benjamin Peterson, Justin Case, Devon Weller, Richard Smith, Alexander Morland, Wadim Grasza, Michael Jakl, and Zed Shaw for their contributions, and to Stathis Sideris for Ditaa.

Please use the issue tracker for all comments and errata. This version covers the latest stable release of 0MQ and was published on Mon 10 October, 2011.

The Guide is mainly in C, but also in PHP and Lua.

第一章 ZeroMQ基础

拯救世界

如何解释ZMQ？有些人会先说一堆ZMQ的好：它是一套用于快速构建的套接字组件；它的信箱系统有超强的路由能力；它太快了！而有些人则喜欢分享他们被ZMQ点悟的时刻，那些被灵感击中的瞬间：所有的事情突然变得简单明了，让人大开眼界。另一些人则会拿ZMQ同其他产品做个比较：它更小，更简单，但却让人觉得如此熟悉。对于我个人而言，我则更倾向于和别人分享ZMQ的诞生史，相信会和各位读者有所共鸣。

编程是一门科学，但往往会乔装成一门艺术。我们从不去了解软件最底层的机理，或者说根本没有人在乎这些。软件并不只是算法、数据结构、编程语言、或者抽象云云，这些不过是一些工具而已，被我们创造、使用、最后抛弃。软件真正的本质，其实是人的本质。

举例来说，当我们遇到一个高度复杂的问题时，我们会群策群力，分工合作，将问题拆分为若干个部分，一起解决。这里就体现了编程的科学：创建一组小型的构建模块，让人们易于理解和使用，那么大家就会一起用它来解决问题。

我们生活在一个普遍联系的世界里，需要现代的编程软件为我们做指引。所以，未来我们所需要的用于处理大规模计算的构建模块，必须是普遍联系的，而且能够并行运作。那时，程序代码不能再只关注自己，它们需要互相交流，变得足够健谈。程序代码需要像人脑一样，数以兆计的神经元高速地传输信号，在一个没有中央控

制的环境下，没有单点故障的环境下，解决问题。这一点其实并不意外，因为就当今的网络来讲，每个节点其实就像是连接了一个人脑一样。

如果你曾和线程、协议、或网络打过交道，你会觉得我上面的话像是天方夜谭。因为在实际应用过程中，只是连接几个程序或网络就已经非常困难和麻烦了。数以兆计的节点？那真是无法想象的。现今只有资金雄厚的企业才能负担得起这种软件和服务。

当今世界的网络结构已经远远超越了我们自身的驾驭能力。二十世纪八十年代的软件危机，弗莱德•布鲁克斯曾说过，这个世上没有银弹。后来，免费和开源解决了这次软件危机，让我们能够高效地分享知识。如今，我们又面临一次新的软件危机，只不过我们谈论得不多。只有那些大型的、富足的企业才有财力建立高度联系的应用程序。那里有云的存在，但它是私有的。我们的数据和知识正在从我们的个人电脑中消失，流入云端，无法获得或与其竞争。是谁坐拥我们的社交网络？这真像一次巨型主机的革命。

我们暂且不谈其中的政治因素，光那些就可以另外出本书了。目前的现状是，虽然互联网能够让千万个程序相连，但我们之中的大多数却无法做到这些。这样一来，那些真正有趣的大型问题（如健康、教育、经济、交通等领域），仍然无法解决。我们没有能力将代码连接起来，也就不能像大脑中的神经元一样处理那些大规模的问题。

已经有人尝试用各种方法来连接应用程序，如数以千计的IETF规范，每种规范解决一个特定问题。对于开发人员来说，HTTP协议是比较简单和易用的，但这也往往让问题变得更糟，因为它鼓励人们形成一种重服务端、轻客户端的思想。

所以迄今为止人们还在使用原始的TCP/UDP协议、私有协议、HTTP协议、网络套接字等形式连接应用程序。这种做法依旧让人痛苦，速度慢又不易扩展，需要集中化管理。而分布式的P2P协议又仅仅适用于娱乐，而非真正的应用。有谁会使用Skype或者Bittorrent来交换数据呢？

这就让我们回归到编程科学的问题上来。想要拯救这个世界，我们需要做两件事情：一，如何在任何地点连接任何两个应用程序；二、将这个解决方案用最为简单的方式包装起来，供程序员使用。

也许这听起来太简单了，但事实确实如此。

ZMQ简介

ZMQ（ØMQ、ZeroMQ, 0MQ）看起来像是一套嵌入式的网络链接库，但工作起来更像是一个并发式的框架。它提供的套接字可以在多种协议中传输消息，如线程间、进程间、TCP、广播等。你可以使用套接字构建多对多的连接模式，如扇出、发布-订阅、任务分发、请求-应答等。ZMQ的快速足以胜任集群应用产品。它的异步I/O机制让你能够构建多核应用程序，完成异步消息处理任务。ZMQ有着多语言支持，并能在几乎所有的操作系统上运行。ZMQ是iMatix公司的产品，以LGPL开源协议发布。

需要具备的知识

- 使用最新的ZMQ稳定版本；
- 使用Linux系统或其他相似的操作系统；
- 能够阅读C语言代码，这是本指南示例程序的默认语言；
- 当我们书写诸如PUSH或SUBSCRIBE等常量时，你能够找到相应语言的实现，如ZMQ_PUSH、ZMQ_SUBSCRIBE。

获取示例

本指南的所有示例都存放于github仓库中，最简单的获取方式是运行以下代码：

```
git clone git://github.com/imatix/zguide.git
```

浏览examples目录，你可以看到多种语言的实现。如果其中缺少了某种你正在使用的语言，我们很希望你可以提交一份补充。这也是本指南实用的原因，要感谢所有做出过贡献的人。

所有的示例代码都以MIT/X11协议发布，若在源代码中有其他限定的除外。

提问-回答

让我们从简单的代码开始，一段传统的Hello World程序。我们会创建一个客户端和一个服务端，客户端发送Hello给服务端，服务端返回World。下文是C语言编写的服务端，它在5555端口打开一个ZMQ套接字，等待请求，收到后应答World。

hwserver.c: Hello World server

```
//
//  Hello World 服务端
//  绑定一个REP套接字至tcp://*:5555
//  从客户端接收Hello，并应答World
//
#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

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

    //  与客户端通信的套接字
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_bind (responder, "tcp://*:5555");

    while (1) {
        //  等待客户端请求
        zmq_msg_t request;
        zmq_msg_init (&request);
        zmq_recv (responder, &request, 0);
        printf ("收到 Hello\n");
        zmq_msg_close (&request);

        //  做些“处理”
        sleep (1);

        //  返回应答
```

```
        zmq_msg_t reply;
        zmq_msg_init_size (&reply, 5);
        memcpy (zmq_msg_data (&reply), "World", 5);
        zmq_send (responder, &reply, 0);
        zmq_msg_close (&reply);
    }
    //  程序不会运行到这里，以下只是演示我们应该如何结束
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
```

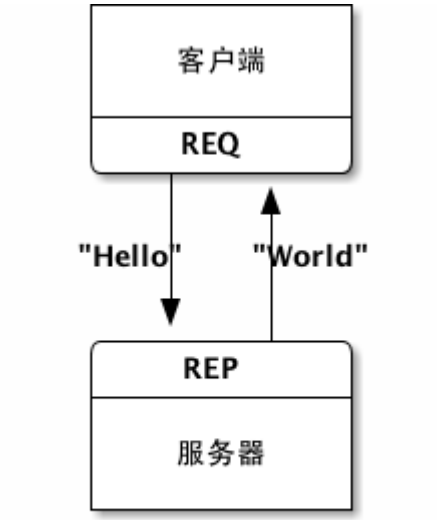


Figure 1 — Request-Reply

使用REQ-REP套接字发送和接受消息是需要遵循一定规律的。客户端首先使用zmq_send()发送消息，再用zmq_recv()接收，如此循环。如果打乱了这个顺序（如连续发送两次）则会报错。类似地，服务端必须先进行接收，后进行发送。

ZMQ使用C语言作为它参考手册的语言，本指南也以它作为示例程序的语言。如果你正在阅读本指南的在线版本，你可以看到示例代码的下方有其他语言的实现。如以下是C++语言：

hwserver.cpp: Hello World server

```
//
// Hello World 服务端 C++语言版
// 绑定一个REP套接字至tcp://*:5555
// 从客户端接收Hello，并应答World
//
#include <zmq.hpp>
#include <string>
#include <iostream>
#include <unistd.h>
```

```
int main () {
    // 准备上下文和套接字
    zmq::context_t context (1);
    zmq::socket_t socket (context, ZMQ_REP);
    socket.bind ("tcp://*:5555");

    while (true) {
        zmq::message_t request;

        // 等待客户端请求
        socket.recv (&request);
        std::cout << "收到 Hello" << std::endl;

        // 做一些“处理”
        sleep (1);

        // 应答World
        zmq::message_t reply (5);
        memcpy ((void *) reply.data (), "World", 5);
        socket.send (reply);
    }
    return 0;
}
```

可以看到C语言和C++语言的API代码差不多，而在PHP这样的语言中，代码就会更为简洁：

hwserver.php: Hello World server

```
<?php
/**
 * Hello World 服务端
 * 绑定REP套接字至 tcp://*:5555
 * 从客户端接收Hello，并应答World
 * @author Ian Barber <ian(dot)barber(at)gmail(dot)com>
 */

$context = new ZMQContext(1);

// 与客户端通信的套接字
$responder = new ZMQSocket($context, ZMQ::SOCKET_REP);
$responder->bind("tcp://*:5555");

while(true) {
    // 等待客户端请求
    $request = $responder->recv();
    printf ("Received request: [%s]\n", $request);
```

```
    // 做一些“处理”
    sleep (1);

    // 应答World
    $responder->send("World");
}
```

下面是客户端的代码：

hwclient: Hello World client in C

```
//
//  Hello World 客户端
//  连接REQ套接字至 tcp://localhost:5555
//  发送Hello给服务端，并接收World
//
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

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

    //  连接至服务端的套接字
    printf ("正在连接至hello world服务端...\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_size (&request, 5);
        memcpy (zmq_msg_data (&request), "Hello", 5);
        printf ("正在发送 Hello %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 ("接收到 World %d\n", request_nbr);
        zmq_msg_close (&reply);
    }
```

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

这看起来是否太简单了？ZMQ就是这样一个东西，你往里加点儿料就能制作出一枚无穷能量的原子弹，用它来拯救世界吧！

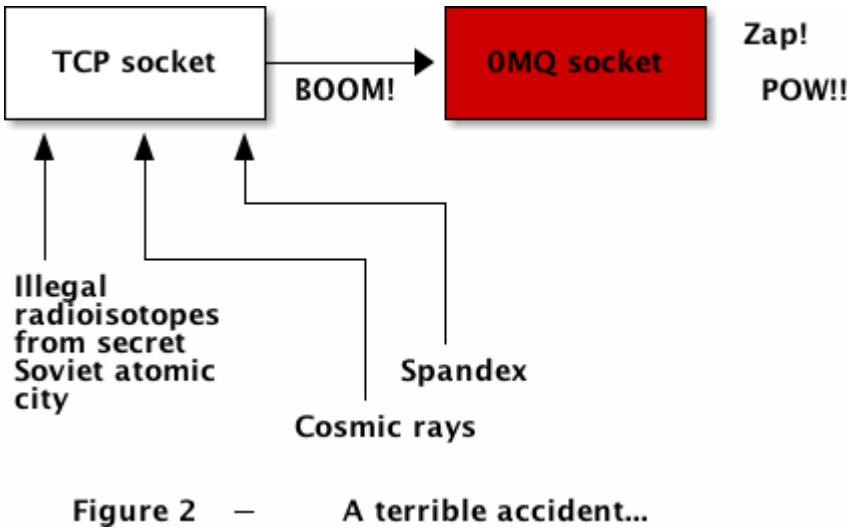


Figure 2 — A terrible accident...

理论上你可以连接千万个客户端到这个服务端上，同时连接都没问题，程序仍会运作得很好。你可以尝试一下先打开客户端，再打开服务端，可以看到程序仍然会正常工作，想想这意味着什么。

让我简单介绍一下这两段程序到底做了什么。首先，他们创建了一个ZMQ上下文，然后是一个套接字。不要被这些陌生的名词吓到，后面我们都会讲到。服务端将REP套接字绑定到5555端口上，并开始等待请求，发出应答，如此循环。客户端则是发送请求并等待服务端的应答。

这些代码背后其实发生了很多很多事情，但是程序员完全不必理会这些，只要知道这些代码短小精悍，极少出错，耐高压。这种通信模式我们称之为请求-应答模式，是ZMQ最直接的一种应用。你可以拿它和RPC及经典的C/S模型做类比。

关于字符串

ZMQ不会关心发送消息的内容，只要知道它所包含的字节数。所以，程序员需要做一些工作，保证对方节点能够正确读取这些消息。如何将一个对象或复杂数据类型转换成ZMQ可以发送的消息，这有类似Protocol Buffers的序列化软件可以做到。但对于字符串，你也是需要有所注意的。

在C语言中，字符串都以一个空字符结尾，你可以像这样发送一个完整的字符串：

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

但是，如果你用其他语言发送这个字符串，很可能不会包含这个空字节，如你使用Python发送：

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

实际发送的消息是：

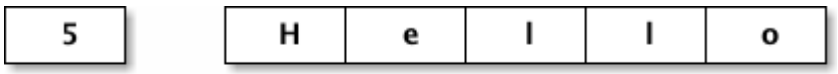


Figure 3 — A 0MQ string

如果你从C语言中读取该消息，你会读到一个类似于字符串的内容，甚至它可能就是一个字符串（第六位在内存中正好是一个空字符），但是这并不合适。这样一来，客户端和服务端对字符串的定义就不统一了，你会得到一些奇怪的结果。

当你用C语言从ZMQ中获取字符串，你不能够相信该字符串有一个正确的结尾。因此，当你在接受字符串时，应该建立多一个字节的缓冲区，将字符串放进去，并添加结尾。

所以，让我们做如下假设：**ZMQ的字符串是有长度的，且传送时不加结束符**。在最简单的情况下，ZMQ字符串和ZMQ消息中的一帧是等价的，就如上图所展现的，由一个长度属性和一串字节表示。

下面这个功能函数会帮助我们在C语言中正确的接受字符串消息：

```
// 从ZMQ套接字中接收字符串，并转换为C语言的字符串
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);
}
```

这段代码我们会在日后的示例中使用，我们可以顺手写一个s_send()方法，并打包成一个.h文件供我们使用。

这就诞生了zhelpers.h，一个供C语言使用的ZMQ功能函数库。它的源代码比较长，而且只对C语言程序员有用，你可以在闲暇时看一看。

获取版本号

ZMQ目前有多个版本，而且仍在持续更新。如果你遇到了问题，也许这在下一个版本中已经解决了。想知道目前的ZMQ版本，你可以在程序中运行如下：

version: ØMQ version reporting in C

```
//
// 返回当前ZMQ的版本号
//
```

```c
#include "zhelpers.h"

int main (void)
{
    int major, minor, patch;
    zmq_version (&major, &minor, &patch);
    printf ("当前ZMQ版本号为 %d.%d.%d\n", major, minor, patch);

    return EXIT_SUCCESS;
}
```

让消息流动起来

第二种经典的消息模式是单向数据分发：服务端将更新事件发送给一组客户端。让我们看一个天气信息发布的例子，包括邮编、温度、相对湿度。我们生成这些随机信息，用来模拟气象站所做的那样。

下面是服务端的代码，使用5556端口：

wuserver: Weather update server in C

```c
//
//  气象信息更新服务
//  绑定PUB套接字至tcp://*:5556端点
//  发布随机气象信息
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和PUB套接字
    void *context = zmq_init (1);
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5556");
    zmq_bind (publisher, "ipc://weather.ipc");

    //  初始化随机数生成器
    srandom ((unsigned) time (NULL));
    while (1) {
        //  生成数据
        int zipcode, temperature, relhumidity;
        zipcode     = randof (100000);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;

        //  向所有订阅者发送消息
        char update [20];
```

```
        sprintf (update, "%05d %d %d", zipcode, temperature, relhumidity);
        s_send (publisher, update);
    }
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

这项更新服务没有开始、没有结束，就像永不消失的电波一样。

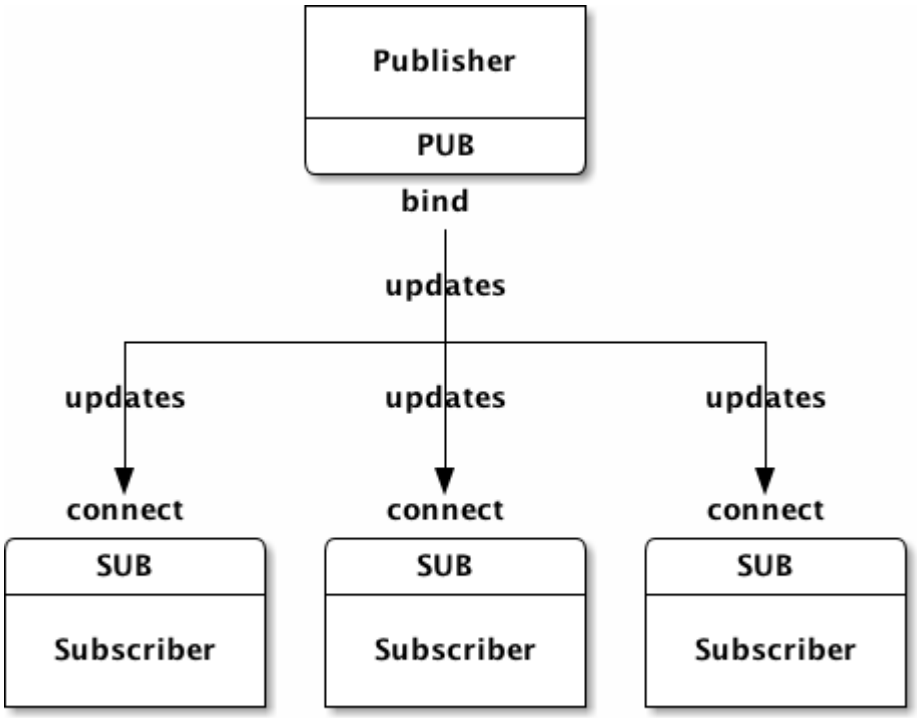


Figure 4 – Publish-Subscribe

下面是客户端程序，它会接受发布者的消息，只处理特定邮编标注的信息，如纽约的邮编是10001:

wuclient: Weather update client in C

```
//
//  气象信息客户端
//  连接SUB套接字至tcp://*:5556端点
//  收集指定邮编的气象信息，并计算平均温度
//
#include "zhelpers.h"

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

    //  创建连接至服务端的套接字
```

```
    printf ("正在收集气象信息...\n");
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");

    //  设置订阅信息，默认为纽约，邮编10001
    char *filter = (argc > 1)? argv [1]: "10001 ";
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, filter, strlen (filter));

    //  处理100条更新信息
    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 ("地区邮编 '%s' 的平均温度为 %dF\n",
        filter, (int) (total_temp / update_nbr));

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

需要注意的是，在使用SUB套接字时，必须使用zmq_setsockopt()方法来设置订阅的内容。如果你不设置订阅内容，那将什么消息都收不到，新手很容易犯这个错误。订阅信息可以是任何字符串，可以设置多次。只要消息满足其中一条订阅信息，SUB套接字就会收到。订阅者可以选择不接收某类消息，也是通过zmq_setsockopt()方法实现的。

PUB-SUB套接字组合是异步的。客户端在一个循环体中使用zmq_recv()接收消息，如果向SUB套接字发送消息则会报错；类似地，服务端可以不断地使用zmq_send()发送消息，但不能在PUB套接字上使用zmq_recv()。

关于PUB-SUB套接字，还有一点需要注意：你无法得知SUB是何时开始接收消息的。就算你先打开了SUB套接字，后打开PUB发送消息，这时SUB还是会丢失一些消息的，因为建立连接是需要一些时间的。很少，但并不是零。

这种“慢连接”的症状一开始会让很多人困惑，所以这里我要详细解释一下。还记得ZMQ是在后台进行异步的I/O传输的，如果你有两个节点用以下顺序相连：

- 订阅者连接至端点接收消息并计数；
- 发布者绑定至端点并立刻发送1000条消息。

运行的结果很可能是订阅者一条消息都收不到。这时你可能会傻眼，忙于检查有没有设置订阅信息，并重新尝试，但结果还是一样。

我们知道在建立TCP连接时需要进行三次握手，会耗费几毫秒的时间，而当节点数增加时这个数字也会上升。在这么短的时间里，ZMQ就可以发送很多很多消息了。举例来说，如果建立连接需要耗时5毫秒，而ZMQ只需要1毫秒就可以发送完这1000条消息。

第二章中我会解释如何使发布者和订阅者同步，只有当订阅者准备好时发布者才会开始发送消息。有一种简单的方法来同步PUB和SUB，就是让PUB延迟一段时间再发送消息。现实编程中我不建议使用这种方式，因为它太脆弱了，而且不好控制。不过这里我们先暂且使用sleep的方式来解决，等到第二章的时候再讲述正确的处理方式。

另一种同步的方式则是认为发布者的消息流是无穷无尽的，因此丢失了前面一部分信息也没有关系。我们的气象信息客户端就是这么做的。

示例中的气象信息客户端会收集指定邮编的一千条信息，其间大约有1000万条信息被发布。你可以先打开客户端，再打开服务端，工作一段时间后重启服务端，这时客户端仍会正常工作。当客户端收集完所需信息后，会计算并输出平均温度。

关于发布-订阅模式的几点说明：

- 订阅者可以连接多个发布者，轮流接收消息；
- 如果发布者没有订阅者与之相连，那它发送的消息将直接被丢弃；
- 如果你使用TCP协议，那当订阅者处理速度过慢时，消息会在发布者处堆积。以后我们会讨论如何使用阈值（HWM）来保护发布者。
- 在目前版本的ZMQ中，消息的过滤是在订阅者处进行的。也就是说，发布者会向订阅者发送所有的消息，订阅者会将未订阅的消息丢弃。

我在自己的四核计算机上尝试发布1000万条消息，速度很快，但没什么特别的：

```
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
```

分布式处理

下面一个示例程序中，我们将使用ZMQ进行超级计算，也就是并行处理模型：

- 任务分发器会生成大量可以并行计算的任务；
- 有一组worker会处理这些任务；
- 结果收集器会在末端接收所有worker的处理结果，进行汇总。

现实中，worker可能散落在不同的计算机中，利用GPU（图像处理单元）进行复杂计算。下面是任务分发器的代码，它会生成100个任务，任务内容是让收到的worker延迟若干毫秒。

taskvent: Parallel task ventilator in C

```c
//
//  任务分发器
//  绑定PUSH套接字至tcp://localhost:5557端点
//  发送一组任务给已建立连接的worker
//
#include "zhelpers.h"

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

    //  用于发送消息的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_bind (sender, "tcp://*:5557");

    //  用于发送开始信号的套接字
    void *sink = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sink, "tcp://localhost:5558");

    printf ("准备好worker后按任意键开始: ");
    getchar ();
    printf ("正在向worker分配任务...\n");

    //  发送开始信号
    s_send (sink, "0");

    //  初始化随机数生成器
    srandom ((unsigned) time (NULL));

    //  发送100个任务
    int task_nbr;
    int total_msec = 0;     //  预计执行时间（毫秒）
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        int workload;
        //  随机产生1-100毫秒的工作量
        workload = randof (100) + 1;
        total_msec += workload;
        char string [10];
        sprintf (string, "%d", workload);
        s_send (sender, string);
    }
    printf ("预计执行时间: %d 毫秒\n", total_msec);
    sleep (1);              //  延迟一段时间，让任务分发完成
```

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

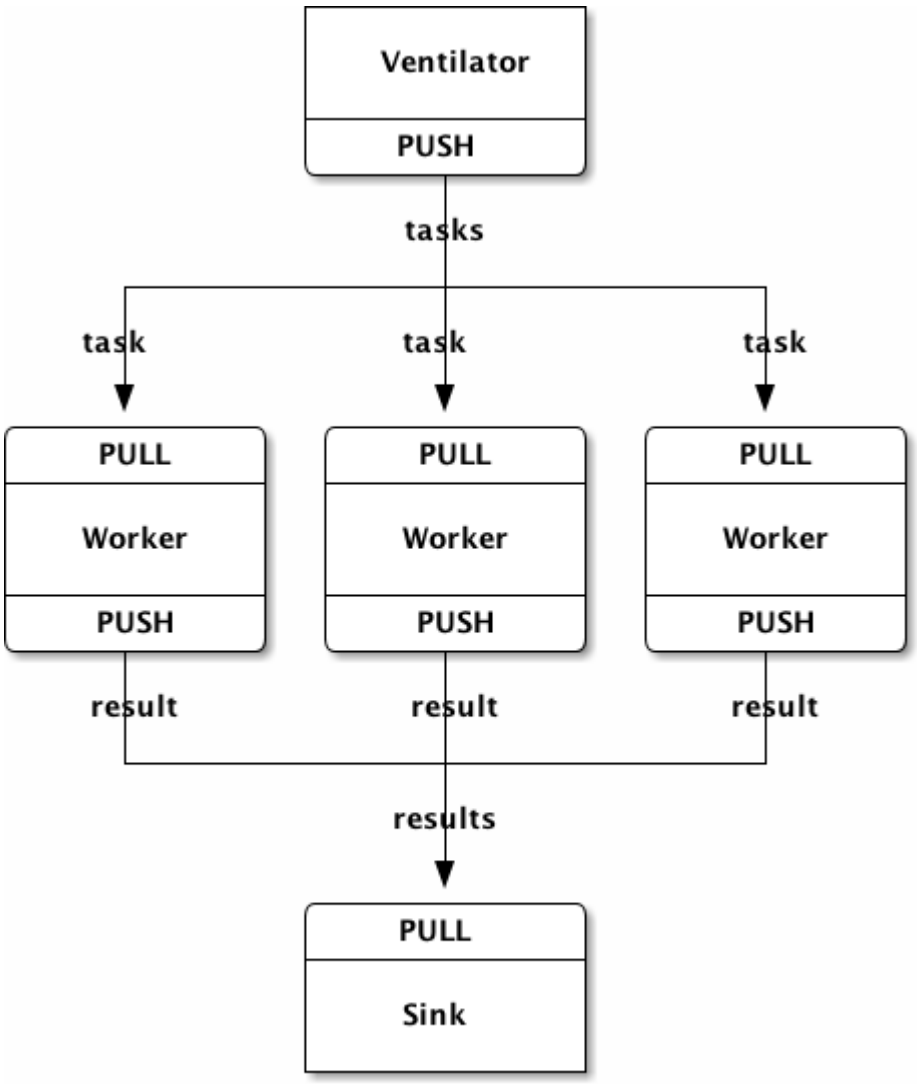


Figure 5 – Parallel Pipeline

下面是worker的代码，它接受信息并延迟指定的毫秒数，并发送执行完毕的信号：

taskwork: Parallel task worker in C

```
//
//  任务执行器
//  连接PULL套接字至tcp://localhost:5557端点
//  从任务分发器处获取任务
//  连接PUSH套接字至tcp://localhost:5558端点
//  向结果采集器发送结果
//
```

```c
#include "zhelpers.h"

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

    //  获取任务的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  发送结果的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sender, "tcp://localhost:5558");

    //  循环处理任务
    while (1) {
        char *string = s_recv (receiver);
        //  输出处理进度
        fflush (stdout);
        printf ("%s.", string);

        //  开始处理
        s_sleep (atoi (string));
        free (string);

        //  发送结果
        s_send (sender, "");
    }
    zmq_close (receiver);
    zmq_close (sender);
    zmq_term (context);
    return 0;
}
```

下面是结果收集器的代码。它会收集100个处理结果，并计算总的执行时间，让我们由此判别任务是否是并行计算的。

tasksink: Parallel task sink in C

```c
//
//  任务收集器
//  绑定PULL套接字至tcp://localhost:5558端点
//  从worker处收集处理结果
//
#include "zhelpers.h"
```

```c
int main (void)
{
    //  准备上下文和套接字
    void *context = zmq_init (1);
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");

    //  等待开始信号
    char *string = s_recv (receiver);
    free (string);

    //  开始计时
    int64_t start_time = s_clock ();

    //  确定100个任务均已处理
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if ((task_nbr / 10) * 10 == task_nbr)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    //  计算并输出总执行时间
    printf ("执行时间: %d 毫秒\n",
        (int) (s_clock () - start_time));

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

一组任务的平均执行时间在5秒左右，以下是分别开始1个、2个、4个worker时的执行结果：

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

关于这段代码的几个细节：

- worker上游和任务分发器相连，下游和结果收集器相连，这就意味着你可以开启任意多个worker。但若worker是绑定至端点的，而非连接至端点，那我们就需要准备更多的端点，并配置任务分发器和结果收集器。所以说，任务分发器和结果收集器是这个网络结构中较为稳定的部分，因此应该由它们绑定至端点，而非worker，因为它们较为动态。

- 我们需要做一些同步的工作，等待worker全部启动之后再分发任务。这点在ZMQ中很重要，且不易解决。连接套接字的动作会耗费一定的时间，因此当第一个worker连接成功时，它会一下收到很多任务。所以说，如果我们不进行同步，那这些任务根本就不会被并行地执行。你可以自己试验一下。

- 任务分发器使用PUSH套接字向worker均匀地分发任务（假设所有的worker都已经连接上了），这种机制称为*负载均衡*，以后我们会见得更多。

- 结果收集器的PULL套接字会均匀地从worker处收集消息，这种机制称为*公平队列*：

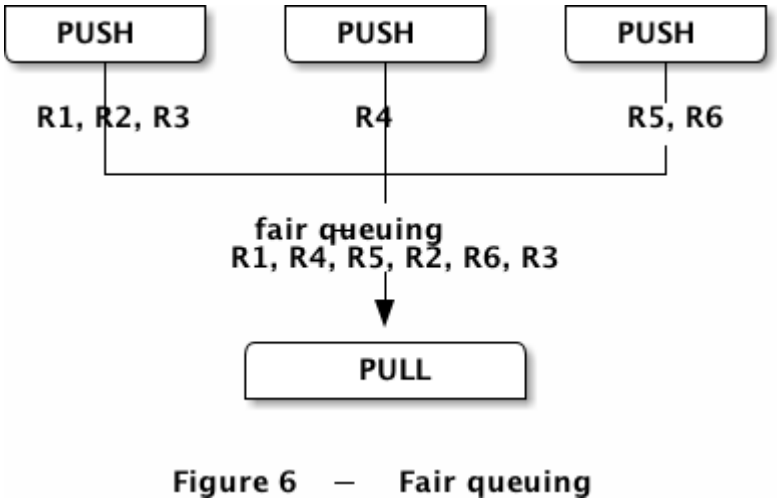


Figure 6 — Fair queuing

管道模式也会出现慢连接的情况，让人误以为PUSH套接字没有进行负载均衡。如果你的程序中某个worker接收到了更多的请求，那是因为它的PULL套接字连接得比较快，从而在别的worker连接之前获取了额外的消息。

使用ZMQ编程

看着这些示例程序后，你一定迫不及待想要用ZMQ进行编程了。不过在开始之前，我还有几条建议想给到你，这样可以省去未来的一些麻烦：

- 学习ZMQ要循序渐进，虽然它只是一套API，但却提供了无尽的可能。一步一步学习它提供的功能，并完全掌握。

- 编写漂亮的代码。丑陋的代码会隐藏问题，让想要帮助你的人无从下手。比如，你会习惯于使用无意义的变量名，但读你代码的人并不知道。应使用有意义的变量名称，而不是随意起一个。代码的缩进要统一，布局清晰。漂亮的代码可以让你的世界变得更美好。

- 边写边测试，当代码出现问题，你就可以快速定位到某些行。这一点在编写ZMQ应用程序时尤为重要，因为很多时候你无法第一次就编写出正确的代码。

- 当你发现自己编写的代码无法正常工作时，你可以将其拆分成一些代码片段，看看哪段没有正确地执行。ZMQ可以让你构建非常模块化的代码，所以应该好好利用这一点。

- 需要时应使用抽象的方法来编写程序（类、成员函数等等），不要随意拷贝代码，因为拷贝代码的同时也是在拷贝错误。

我们看看下面这段代码，是某位同仁让我帮忙修改的：

```
//  注意：不要使用这段代码！
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");
    //  注意：不要使用这段代码，它不能工作！
    //  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;
}
```

下面是我为他重写的代码，顺便修复了一些BUG：

```
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;
}
```

上段程序的最后，它将套接字在两个线程之间传递，这会导致莫名其妙的问题。这种行为在ZMQ 2.1中虽然是合法的，但是不提倡使用。

ZMQ 2.1版

历史告诉我们，ZMQ 2.0是一个低延迟的分布式消息系统，它从众多同类软件中脱颖而出，摆脱了各种奢华的名目，向世界宣告“无极限”的口号。这是我们一直在使用的稳定发行版。

时过境迁，2010年流行的东西在2011年就不一定了。当ZMQ的开发者和社区开发者在激烈地讨论ZMQ的种种问题时，ZMQ 2.1横空出世了，成为新的稳定发行版。

本指南主要针对ZMQ 2.1进行描述，因此对于从ZMQ 2.0迁移过来的开发者来说有一些需要注意的地方：

- 在2.0中，调用zmq_close()和zmq_term()时会丢弃所有尚未发送的消息，所以在发送完消息后不能直接关闭程序，2.0的示例中往往使用sleep(1)来规避这个问题。但是在2.1中就不需要这样做了，程序会等待消息全部发送完毕后再退出。

- 相反地，2.0中可以在尚有套接字打开的情况下调用zmq*term()，这在2.1中会变得不安全，会造成程序的阻塞。所以，在2.1程序中我们*会先关闭所有的套接字，*然后才退出程序。如果套接字中有尚未发送的消息，程序就会一直处于等待状态，*除非手工设置了套接字的LINGER选项_（如设置为零），那么套接字会在相应的时间后关闭。

```
int zero = 0;
zmq_setsockopt (mysocket, ZMQ_LINGER, &zero, sizeof (zero));
```

- 2.0中，zmq_poll()函数没有定时功能，它会在满足条件时立刻返回，我们需要在循环体中检查还有多少剩余。但在2.1中，zmq_poll()会在指定时间后返回，因此可以作为定时器使用。

- 2.0中，ZMQ会忽略系统的中断消息，这就意味着对libzmq的调用是不会收到EINTR消息的，这样就无法对SIGINT（Ctrl-C）等消息进行处理了。在2.1中，这个问题得以解决，像类似zmq_recv()的方法都会接收并返回系统的EINTR消息。

正确地使用上下文

ZMQ应用程序的一开始总是会先创建一个上下文，并用它来创建套接字。在C语言中，创建上下文的函数是zmq_init()。一个进程中只应该创建一个上下文。从技术的角度来说，上下文是一个容器，包含了该进程下所

有的套接字，并为inproc协议提供实现，用以高速连接进程内不同的线程。如果一个进程中创建了两个上下文，那就相当于启动了两个ZMQ实例。如果这正是你需要的，那没有问题，但一般情况下：

在一个进程中使用zmq_init()函数创建一个上下文，并在结束时使用zmq_term()函数关闭它

如果你使用了fork()系统调用，那每个进程需要自己的上下文对象。如果在调用fork()之前调用了zmq_init()函数，那每个子进程都会有自己的上下文对象。通常情况下，你会需要在子进程中做些有趣的事，而让父进程来管理它们。

正确地退出和清理

程序员的一个良好习惯是：总是在结束时进行清理工作。当你使用像Python那样的语言编写ZMQ应用程序时，系统会自动帮你完成清理。但如果使用的是C语言，那就需要小心地处理了，否则可能发生内存泄露、应用程序不稳定等问题。

内存泄露只是问题之一，其实ZMQ是很在意程序的退出方式的。个中原因比较复杂，但简单的来说，如果仍有套接字处于打开状态，调用zmq_term()时会导致程序挂起；就算关闭了所有的套接字，如果仍有消息处于待发送状态，zmq_term()也会造成程序的等待。只有当套接字的LINGER选项设为0时才能避免。

我们需要关注的ZMQ对象包括：消息、套接字、上下文。好在内容并不多，至少在一般的应用程序中是这样：

- 处理完消息后，记得用zmq_msg_close()函数关闭消息；
- 如果你同时打开或关闭了很多套接字，那可能需要重新规划一下程序的结构了；
- 退出程序时，应该先关闭所有的套接字，最后调用zmq_term()函数，销毁上下文对象。

如果要用ZMQ进行多线程的编程，需要考虑的问题就更多了。我们会在下一章中详述多线程编程，但如果你耐不住性子想要尝试一下，以下是在退出时的一些建议：

- 不要在多个线程中使用同一个套接字。不要去想为什么，反正别这么干就是了。
- 关闭所有的套接字，并在主程序中关闭上下文对象。
- 如果仍有处于阻塞状态的recv或poll调用，应该在主程序中捕捉这些错误，并在相应的线程中关闭套接字。不要重复关闭上下文，zmq_term()函数会等待所有的套接字安全地关闭后才结束。

看吧，过程是复杂的，所以不同语言的API实现者可能会将这些步骤封装起来，让结束程序变得不那么复杂。

我们为什么需要ZMQ

现在我们已经将ZMQ运行起来了，让我们回顾一下为什么我们需要ZMQ：

目前的应用程序很多都会包含跨网络的组件，无论是局域网还是因特网。这些程序的开发者都会用到某种消息通信机制。有些人会使用某种消息队列产品，而大多数人则会自己手工来做这些事，使用TCP或UDP协议。这些协议使用起来并不困难，但是，简单地将消息从A发给B，和在任何情况下都能进行可靠的消息传输，这两种情况显然是不同的。

让我们看看在使用纯TCP协议进行消息传输时会遇到的一些典型问题。任何可复用的消息传输层肯定或多或少地会要解决以下问题：

- 如何处理I/O？是让程序阻塞等待响应，还是在后台处理这些事？这是软件设计的关键因素。阻塞式的I/O操作会让程序架构难以扩展，而后台处理I/O也是比较困难的。

- 如何处理那些临时的、来去自由的组件？我们是否要将组件分为客户端和服务端两种，并要求服务端永不消失？那如果我们想要将服务端相连怎么办？我们要每隔几秒就进行重连吗？

- 我们如何表示一条消息？我们怎样通过拆分消息，让其变得易读易写，不用担心缓存溢出，既能高效地传输小消息，又能胜任视频等大型文件的传输？

- 如何处理那些不能立刻发送出去的消息？比如我们需要等待一个网络组件重新连接的时候？我们是直接丢弃该条消息，还是将它存入数据库，或是内存中的一个队列？

- 要在哪里保存消息队列？如果某个组件读取消息队列的速度很慢，造成消息的堆积怎么办？我们要采取什么样的策略？

- 如何处理丢失的消息？我们是等待新的数据，请求重发，还是需要建立一套新的可靠性机制以保证消息不会丢失？如果这个机制自身崩溃了呢？

- 如果我们想换一种网络连接协议，如用广播代替TCP单播？或者改用IPv6？我们是否需要重写所有的应用程序，或者将这种协议抽象到一个单独的层中？

- 我们如何对消息进行路由？我们可以将消息同时发送给多个节点吗？是否能将应答消息返回给请求的发送方？

- 我们如何为另一种语言写一个API？我们是否需要完全重写某项协议，还是重新打包一个类库？

- 怎样才能做到在不同的架构之间传送消息？是否需要为消息规定一种编码？

- 我们如何处理网络通信错误？等待并重试，还是直接忽略或取消？

我们可以找一个开源软件来做例子，如Hadoop Zookeeper，看一下它的C语言API源码，[src/c/src/zookeeper.c](http://github.com/apache/zookeeper/blob/trunk/src/c/src/zookeeper.c src/c/src/zookeeper.c)。这段代码大约有3200行，没有注释，实现了一个C/S网络通信协议。它工作起来很高效，因为使用了poll()来代替select()。但是，Zookeeper应该被抽象出来，作为一种通用的消息通信层，并加以详细的注释。像这样的模块应该得到最大程度上的复用，而不是重复地制造轮子。

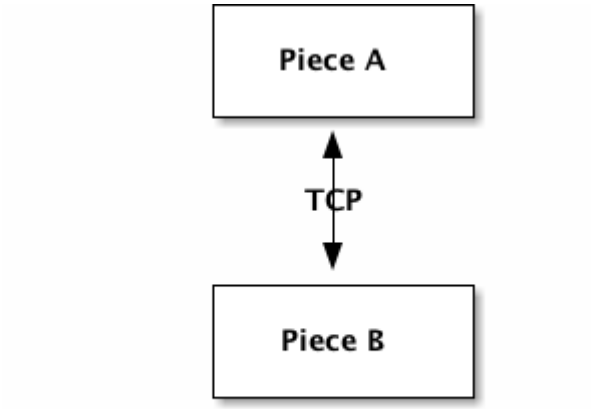


Figure 7 — Messaging as it starts

但是，如何编写这样一个可复用的消息层呢？为什么长久以来人们宁愿在自己的代码中重复书写控制原始TCP套接字的代码，而不愿编写这样一个公共库呢？

其实，要编写一个通用的消息层是件非常困难的事，这也是为什么FOSS项目不断在尝试，一些商业化的消息产品如此之复杂、昂贵、僵硬、脆弱。2006年，iMatix设计了AMQP协议，为FOSS项目的开发者提供了可能是当时第一个可复用的消息系统。AMQP比其他同类产品要来得好，但仍然是复杂、昂贵和脆弱的。它需要花费几周的时间去学习，花费数月的时间去创建一个真正能用的架构，到那时可能为时已晚了。

大多数消息系统项目，如AMQP，为了解决上面提到的种种问题，发明了一些新的概念，如“代理”的概念，将寻址、路由、队列等功能都包含了进来。结果就是在一个没有任何注释的协议之上，又构建了一个C/S协议和相应的API，让应用程序和代理相互通信。代理的确是一个不错的解决方案，帮助降低大型网络结构的复杂度。但是，在Zookeeper这样的项目中应用代理机制的消息系统，可能是件更加糟糕的事，因为这意味了需要添加一台新的计算机，并构成一个新的单点故障。代理会逐渐成为新的瓶颈，管理起来更具风险。如果软件支持，我们可以添加第二个、第三个、第四个代理，构成某种冗余容错的模式。有人就是这么做的，这让系统架构变得更为复杂，增加了隐患。

在这种以代理为中心的架构下，需要一支专门的运维团队。你需要昼夜不停地观察代理的状态，不时地用棍棒调教他们。你需要添加计算机，以及更多的备份机，你需要有专人管理这些机器。这样做只对那些大型的网络应用程序才有意义，因为他们有更多可移动的模块，有多个团队进行开发和维护，而且已经经过了多年的建设。

这样一来，中小应用程序的开发者们就无计可施了。他们只能设法避免编写网络应用程序，转而编写那些不需要扩展的程序；或者可以使用原始的方式进行网络编程，但编写的软件会非常脆弱和复杂，难以维护；亦或者他们选择一种消息通信产品，虽然能够开发出扩展性强的应用程序，但需要支付高昂的代价。似乎没有一种选择是合理的，这也是为什么在上个世纪消息系统会成为一个广泛的问题。

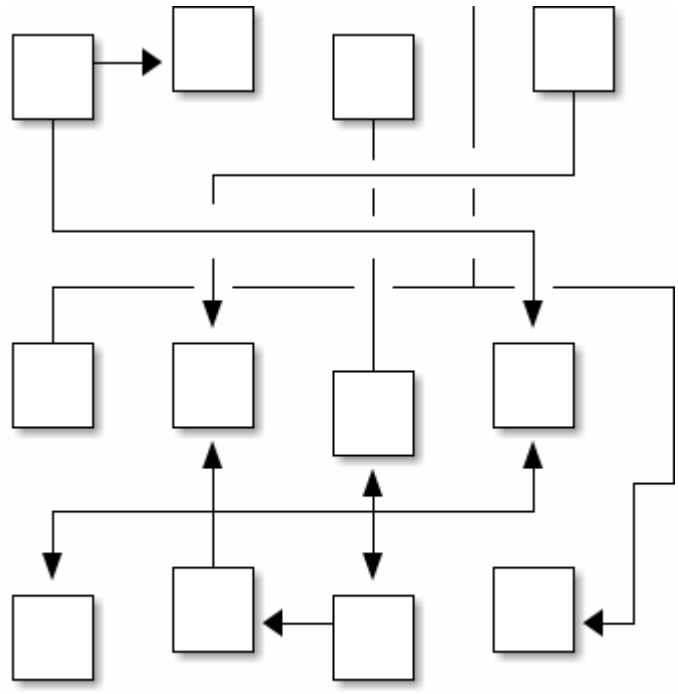


Figure 8 — Messaging as it becomes

我们真正需要的是这样一种消息软件，它能够做大型消息软件所能做的一切，但使用起来又非常简单，成本很低，可以用到所有的应用程序中，没有任何依赖条件。因为没有了额外的模块，就降低了出错的概率。这种软件需要能够在所有的操作系统上运行，并能支持所有的编程语言。

ZMQ就是这样一种软件：它高效，提供了嵌入式的类库，使应用程序能够很好地在网络中扩展，成本低廉。

ZMQ的主要特点有：

- ZMQ会在后台线程异步地处理I/O操作，它使用一种不会死锁的数据结构来存储消息。
- 网络组件可以来去自如，ZMQ会负责自动重连，这就意味着你可以以任何顺序启动组件；用它创建的面向服务架构（SOA）中，服务端可以随意地加入或退出网络。
- ZMQ会在有必要的情况下自动将消息放入队列中保存，一旦建立了连接就开始发送。
- ZMQ有阈值（HWM）的机制，可以避免消息溢出。当队列已满，ZMQ会自动阻塞发送者，或丢弃部分消息，这些行为取决于你所使用的消息模式。
- ZMQ可以让你用不同的通信协议进行连接，如TCP、广播、进程内、进程间。改变通信协议时你不需要去修改代码。
- ZMQ会恰当地处理速度较慢的节点，会根据消息模式使用不同的策略。
- ZMQ提供了多种模式进行消息路由，如请求-应答模式、发布-订阅模式等。这些模式可以用来搭建网络拓扑结构。
- ZMQ中可以根据消息模式建立起一些中间装置（很小巧），可以用来降低网络的复杂程度。
- ZMQ会发送整个消息，使用消息帧的机制来传递。如果你发送了10KB大小的消息，你就会收到10KB大小的消息。
- ZMQ不强制使用某种消息格式，消息可以是0字节的，或是大到GB级的数据。当你表示这些消息时，可以选用诸如谷歌的protocol buffers，XDR等序列化产品。
- ZMQ能够智能地处理网络错误，有时它会进行重试，有时会告知你某项操作发生了错误。
- ZMQ甚至可以降低对环境的污染，因为节省了CPU时间意味着节省了电能。

其实ZMQ可以做的还不止这些，它会颠覆人们编写网络应用程序的模式。虽然从表面上看，它不过是提供了一套处理套接字的API，能够用zmq_recv()和zmq_send()进行消息的收发，但是，消息处理将成为应用程序的核心部分，很快你的程序就会变成一个个消息处理模块，这既美观又自然。它的扩展性还很强，每项任务由一个节点（节点是一个线程）、同一台机器上的两个节点（节点是一个进程）、同一网络上的两台机器（节点是一台机器）来处理，而不需要改动应用程序。

套接字的扩展性

我们来用实例看看ZMQ套接字的扩展性。这个脚本会启动气象信息服务及多个客户端：

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

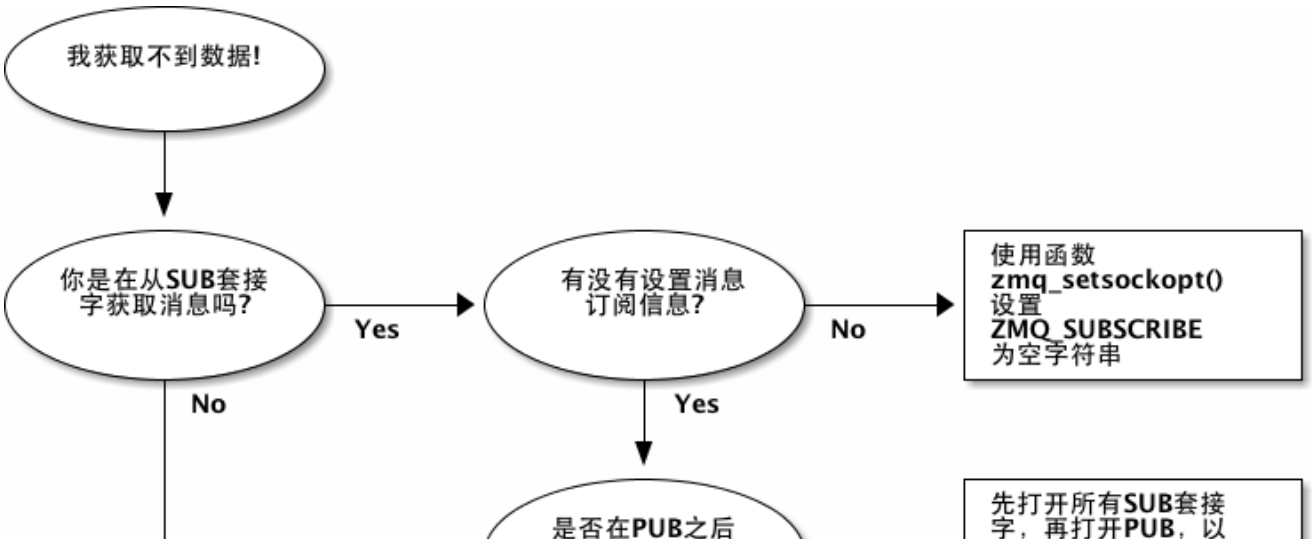
执行过程中，我们可以通过top命令查看进程状态（以下是一台四核机器的情况）：

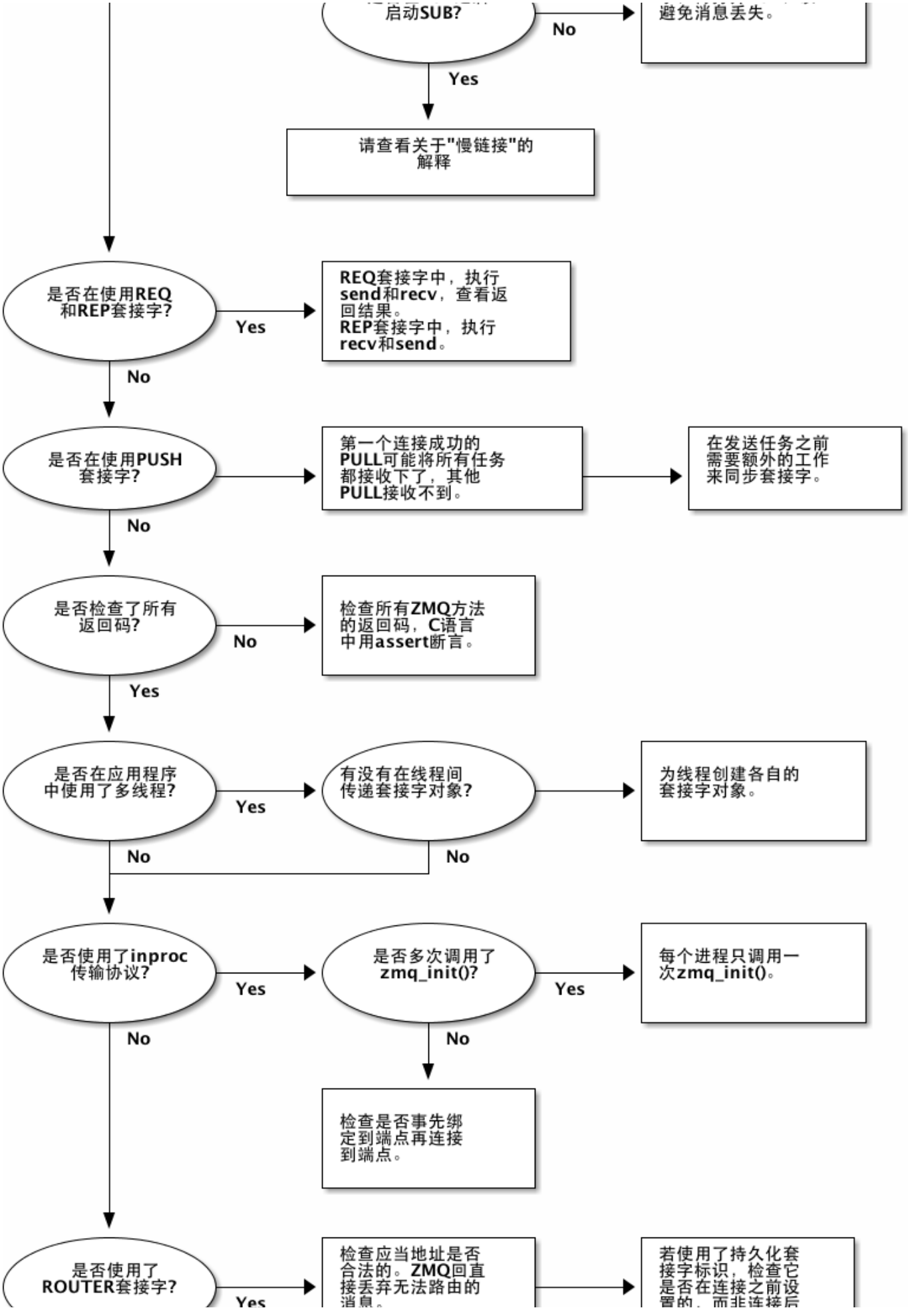
```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7136 ph        20   0 1040m 959m 1156 R  157 12.0  16:25.47 wuserver
 7966 ph        20   0 98608 1804 1372 S   33  0.0   0:03.94 wuclient
 7963 ph        20   0 33116 1748 1372 S   14  0.0   0:00.76 wuclient
 7965 ph        20   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
```

我们想想现在发生了什么：气象信息服务程序有一个单独的套接字，却能同时向五个客户端并行地发送消息。我们可以有成百上千个客户端并行地运作，服务端看不到这些客户端，不能操纵它们。

如果解决丢失消息的问题

在编写ZMQ应用程序时，你遇到最多的问题可能是无法获得消息。下面有一个问题解决路线图，列举了最基本的出错原因。不用担心其中的某些术语你没有见过，在后面的几章里都会讲到。




启动SUB?
No
避免消息丢失。
Yes
请查看关于"慢链接"的
解释
是否在使用REQ
和REP套接字?
Yes
REQ套接字中，执行
send和recv，查看返
回结果。
REP套接字中，执行
recv和send。
No
是否在使用PUSH
套接字?
第一个连接成功的
PULL可能将所有任务
都接收下了，其他
PULL接收不到。
在发送任务之前
需要额外的工作
来同步套接字。
No
是否检查了所有
返回码?
No
检查所有ZMQ方法
的返回码，C语言
中用assert断言。
Yes
是否在应用程序
中使用了多线程?
Yes
有没有在线程间
传递套接字对象?
为线程创建各自的
套接字对象。
No
No
是否使用了inproc
传输协议?
Yes
是否多次调用了
zmq_init()?
Yes
每个进程只调用一
次zmq_init()。
No
No
检查是否事先绑
定到端点再连接
到端点。
是否使用了
ROUTER套接字?
Yes
检查应当地址是否
合法的。ZMQ回直
接丢弃无法路由的
若使用了持久化套
接字标识，检查它
是否在连接之前设

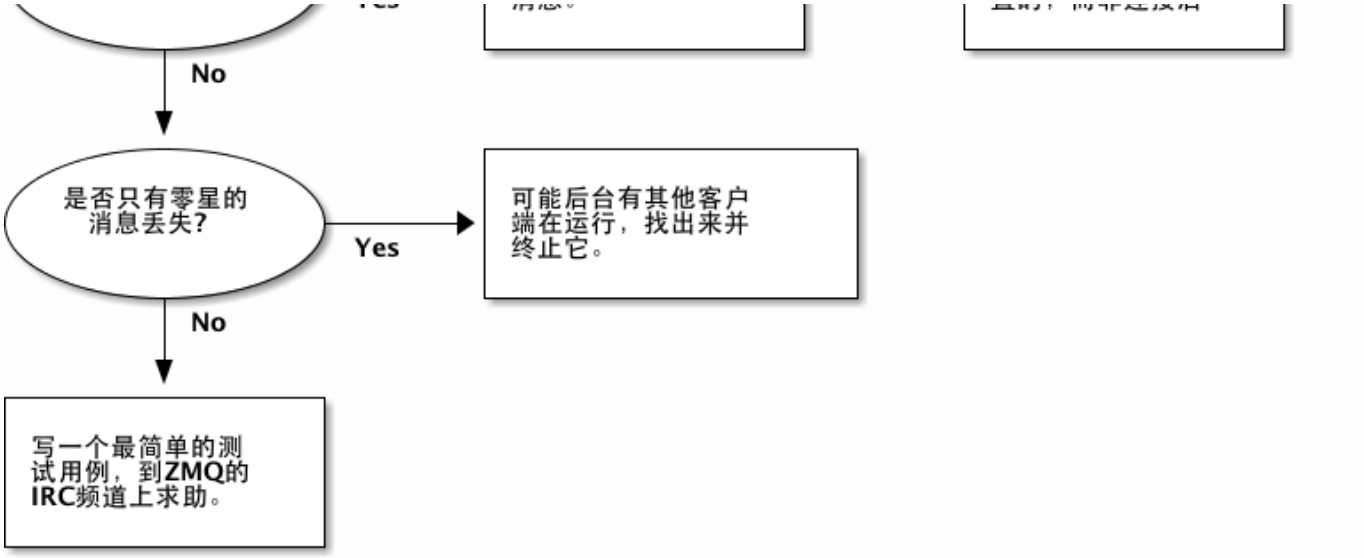


Figure 9 — Missing Message Problem Solver

如果ZMQ在你的应用程序中扮演非常重要的角色，那你可能就需要好好计划一下了。首先，创建一个原型，用以测试设计方案的可行性。采取一些压力测试的手段，确保它足够的健壮。其次，主攻测试代码，也就是编写测试框架，保证有足够的电力供应和时间，来进行高强度的测试。理想状态下，应该由一个团队编写程序，另一个团队负责击垮它。最后，让你的公司及时联系iMatix，获得技术上的支持。

简而言之，如果你没有足够理由说明设计出来的架构能够在现实环境中运行，那么很有可能它就会在最紧要的关头崩溃。

警告：你的想法可能会被颠覆！

传统网络编程的一个规则是套接字只能和一个节点建立连接。虽然也有广播的协议，但毕竟是第三方的。当我们认定“一个套接字 = 一个连接”的时候，我们会用一些特定的方式来扩展应用程序架构：我们为每一块逻辑创建线程，该线程独立地维护一个套接字。

但在ZMQ的世界里，套接字是智能的、多线程的，能够自动地维护一组完整的连接。你无法看到它们，甚至不能直接操纵这些连接。当你进行消息的收发、轮询等操作时，只能和ZMQ套接字打交道，而不是连接本身。所以说，ZMQ世界里的连接是私有的，不对外部开放，这也是ZMQ易于扩展的原因之一。

由于你的代码只会和某个套接字进行通信，这样就可以处理任意多个连接，使用任意一种网络协议。而ZMQ的消息模式又可以进行更为廉价和便捷的扩展。

这样一来，传统的思维就无法在ZMQ的世界里应用了。在你阅读示例程序代码的时候，也许你脑子里会想方设法地将这些代码和传统的网络编程相关联：当你读到“套接字”的时候，会认为它就表示与另一个节点的连接——这种想法是错误的；当你读到“线程”时，会认为它是与另一个节点的连接——这也是错误的。

如果你是第一次阅读本指南，使用ZMQ进行了一两天的开发（或者更长），可能会觉得疑惑，ZMQ怎么会让事情便得如此简单。你再次尝试用以往的思维去理解ZMQ，但又无功而返。最后，你会被ZMQ的理念所折服，拨云见雾，开始享受ZMQ带来的乐趣。