# ØMQ - The Guide

**By Pieter Hintjens <ph@imatix.com>, CEO iMatix Corporation.**

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, 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 ØMQ and was published on Tue 7 June, 2011.

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

# Chapter One - Basic Stuff

top prev next

## Fixing the World

top prev next

How to explain ØMQ? Some of us start by saying all the wonderful things it does. *It's sockets on steroids. It's like mailboxes with routing. It's fast!* Others try to share their moment of enlightenment, that zap-pow-kaboom satori paradigm-shift moment when it all became obvious. *Things just become simpler. Complexity goes away. It opens the mind.* Others try to explain by comparison. *It's smaller, simpler, but still looks familiar.* Personally, I like to remember why we made ØMQ at all, because that's most likely where you, the reader, still are today.

Programming is a science dressed up as art, because most of us don't understand the physics of software, and it's rarely if ever taught. The physics of software is not algorithms, data structures, languages and abstractions. These are just tools we make, use, throw away. The real physics of software is the physics of people.

Specifically, our limitations when it comes to complexity, and our desire to work together to solve large problems in pieces. This is the science of programming: make building blocks that people can understand and use *easily*, and people will work together to solve the very largest problems.

We live in a connected world, and modern software has to navigate this world. So the building blocks for tomorrow's very largest solutions are connected and massively parallel. It's not enough for code to be "strong and silent" any more. Code has to talk to code. Code has to be chatty, sociable, well-connected. Code has to run like the human brain, trillions of individual neurons firing off messages to each other, a massively parallel network with no central control, no single point of failure, yet able to solve immensely difficult problems. And it's no accident that the future of code looks like the human brain, because the endpoints of every network are, at some level, human brains.

If you've done any work with threads, protocols, or networks, you'll realize this is pretty much impossible. It's a dream. Even connecting a few programs across a few sockets is plain nasty, when you start to handle real life situations. Trillions? The cost would be unimaginable. Connecting computers is so difficult that software and services to do this is a multi-billion dollar business.

So we live in a world where the wiring is years ahead of our ability to use it. We had a software crisis in the 1980s, when people like Fred Brooks believed there was no "Silver Bullet". Free and open source software solved that crisis, enabling us to share knowledge efficiently. Today we face another software crisis, but it's one we don't talk about much. Only the largest, richest firms can afford to create connected applications. There is a cloud, but it's proprietary. Our data, our knowledge is disappearing from our personal computers into clouds that we cannot access, cannot compete with. Who owns our social networks? It is like the mainframe-PC revolution in reverse.

We can leave the political philosophy for another book. The point is that while the Internet offers the potential of massively connected code, the reality is that this is out of reach for most of us, and so, large interesting problems (in health, education, economics, transport, and so on) remain unsolved because there is no way to connect the code, and thus no way to connect the brains that could work together to solve these problems.

There have been many attempts to solve the challenge of connected software. There are thousands of IETF specifications, each solving part of the puzzle. For application developers, HTTP is perhaps the one solution to have have been simple enough to work, but it arguably makes the problem worse, by encouraging developers and architects to think in terms of big servers and thin, stupid clients.

So today people are still connecting applications using raw UDP and TCP, proprietary protocols, HTTP, WebSockets. It remains painful, slow, hard to scale, and essentially centralized. Distributed p2p architectures are mostly for play, not work. How many applications use Skype or Bittorrent to exchange data?

Which brings us back to the science of programming. To fix the world, we needed to do two things. One, to solve the general problem of "how to connect any code to any code, anywhere". Two, to wrap that up in the simplest possible building blocks that people could understand and use *easily*.

It sounds ridiculously simple. And maybe it is. That's kind of the whole point.

## ØMQ in a Hundred Words

top prev next

ØMQ (ZeroMQ, 0MQ, zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry whole messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fanout, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. It has a score of language APIs and runs on most operating systems. ØMQ is from iMatix and is LGPL open source.

## Some Assumptions

top prev next

We assume you are using the latest stable release of ØMQ. We assume you are using a Linux box or something similar. We assume you can read C code, more or less, that's the default language for the examples. We assume that when we write constants like PUSH or SUBSCRIBE you can imagine they are really called ZMQ_PUSH or ZMQ_SUBSCRIBE if the programming language needs it.

## Getting the Examples

top prev next

The Guide examples live in the Guide's git repository. The simplest way to get all the examples is to clone this repository:

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

And then browse the examples subdirectory. You'll find examples by language. If there are examples missing in a language you use, you're encouraged to submit a translation. This is how the Guide became so useful, thanks to the work of many people.

All examples are licensed under MIT/X11, unless otherwise specified in the source code.

## Ask and Ye Shall Receive

top prev next

So let's start with some code. We start of course with a Hello World example. We'll make a client and a server. The client sends "Hello" to the server, which replies with "World". Here's the server in C, which opens a ØMQ socket on port 5555, reads requests on it, and replies with "World" to each request:

```
//
//  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)
{
    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 Hello\n");
        zmq_msg_close (&request);

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

        //  Send reply back to client
        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);
    }
    //  We never get here but if we did, this would be how we end
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
```

*hwserver.c: Hello World server*

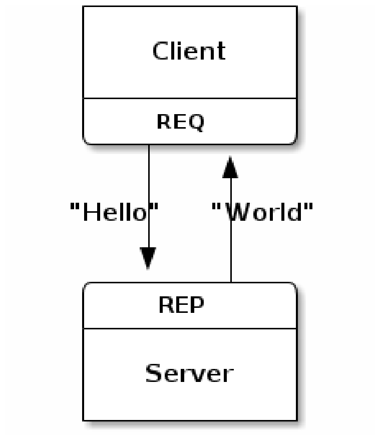


Figure 1 — Request-Reply

The REQ-REP socket pair is lockstep. The client does zmq_send(3) and then zmq_recv(3), in a loop (or once if that's all it needs). Doing any other sequence (e.g. sending two messages in a row) will cause an error. Similarly the service does zmq_recv(3) and then zmq_send(3) in that order, and as often as it needs to.

ØMQ uses C as its reference language and this is the main language we'll use for examples. If you're reading this on-line, the link below the example takes you to translations into other programming languages. Let's compare the same server in C++:

```
//
//  Hello World server in C++
//  Binds REP socket to tcp://*:5555
//  Expects "Hello" from client, replies with "World"
//
#include <zmq.hpp>
#include <string>
#include <iostream>
#include <unistd.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);
        std::cout << "Received Hello" << std::endl;

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

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

*hwserver.cpp: Hello World server*

You can see that the ØMQ API is similar in C and C++. In a language like PHP, we can hide even more and the code becomes even easier to read:

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

$context = new ZMQContext(1);

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

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

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

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

*hwserver.php: Hello World server*

Here's the client code (click the link below the source to look at, or contribute a translation in your favorite programming language):

```
//
//  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)
{
    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_size (&request, 5);
        memcpy (zmq_msg_data (&request), "Hello", 5);
        printf ("Sending 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 ("Received World %d\n", request_nbr);
        zmq_msg_close (&reply);
    }
    zmq_close (requester);
    zmq_term (context);
    return 0;
}
```

*hwclient.c: Hello World client*

Now this looks too simple to be realistic, but a ØMQ socket is what you get when you take a normal TCP socket, inject it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombard it with 1950-era cosmic rays, and put it into the hands of a drug-addled comic book author with a badly-disguised fetish for bulging muscles clad in spandex. Yes, ØMQ sockets are the world-saving superheros of the networking world.

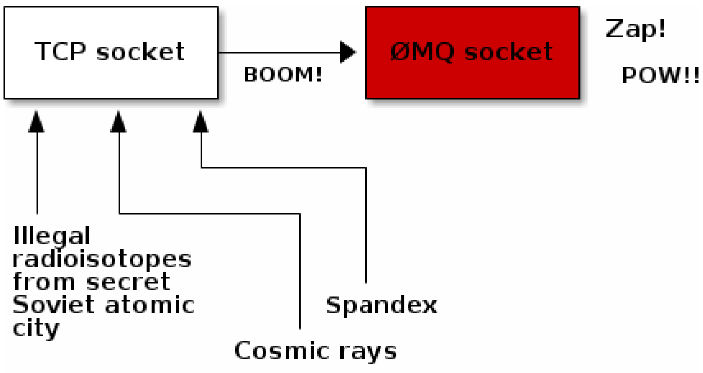


Figure 2 – A terrible accident...

You could literally throw thousands of clients at this server, all at once, and it would continue to work happily and quickly. For fun, try starting the client and *then* starting the server, see how it all still works, then think for a second what this means.

Let me explain briefly what these two programs are actually doing. They create a ØMQ context to work with, and a socket. Don't worry what the words mean. You'll pick it up. The server binds its REP (reply) socket to port 5555. The server waits for a request, in a loop, and responds each time with a reply. The client sends a request and reads the reply back from the server.

There is a lot happening behind the scenes but what matters to us programmers is how short and sweet the code is, and how often it doesn't crash, even under heavy load. This is the request-reply pattern, probably the simplest way to use ØMQ. It maps to RPC and the classic client-server model.

## A Minor Note on Strings

top prev next

ØMQ doesn't know anything about the data you send except its size in bytes. That means you are responsible for formatting it safely so that applications can read it back. Doing this for objects and complex data types is a job for specialized libraries like Protocol Buffers. But even for strings you need to take care.

In C and some other languages, strings are terminated with a null byte. We could send a string like "HELLO" with that extra null byte:

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

However if you send a string from another language it probably will not include that null byte. For example, when we send that same string in Python, we do this:

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

Then what goes onto the wire is:

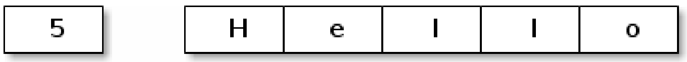


Figure 3 — A ØMQ string

And if you read this from a C program, you will get something that looks like a string, and might by accident act like a string (if by luck the five bytes find themselves followed by an innocently lurking null), but isn't a proper string. Which means that your client and server don't agree on the string format, you will get weird results.

When you receive string data from ØMQ, in C, you simply cannot trust that it's safely terminated. Every single time you read a string you should allocate a new buffer with space for an extra byte, copy the string, and terminate it properly with a null.

So let's establish the rule that **ØMQ strings are length-specified, and are sent on the wire *without* a trailing null**. In the simplest case (and we'll do this in our examples) a ØMQ string maps neatly to a ØMQ message frame, which looks like the above figure, a length and some bytes.

Here is what we need to do, in C, to receive a ØMQ string and deliver it to the application as a valid C string:

```
//  Receive 0MQ 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);
}
```

This makes a very handy helper function and in the spirit of making things we can reuse profitably, let's write a similar 's_send' function that sends strings in the correct ØMQ format, and package this into a header file we can reuse.

The result is `zhelpers.h`, which lets us write sweeter and shorter ØMQ applications in C. It is a fairly long source, and only fun for C developers, so read it at leisure.

## Version Reporting

top prev next

ØMQ does come in several versions and quite often, if you hit a problem, it'll be something that's been fixed in a later version. So it's a useful trick to know *exactly* what version of ØMQ you're actually linking with. Here is a tiny program that does that:

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

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

    return EXIT_SUCCESS;
}
```

*version.c: ØMQ version reporting*

## Getting the Message Out

top prev next

The second classic pattern is one-way data distribution, in which a server pushes updates to a set of clients. Let's see an example that pushes out weather updates consisting of a zipcode, temperature, and relative humidity. We'll generate random values, just like the real weather stations do.

Here's the server. We'll use port 5556 for this application:

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

int main (void)
{
    //  Prepare our context and publisher
    void *context = zmq_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     = randof (100000);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;

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

*wuserver.c: Weather update server*

There's no start, and no end to this stream of updates, it's like a never ending broadcast.

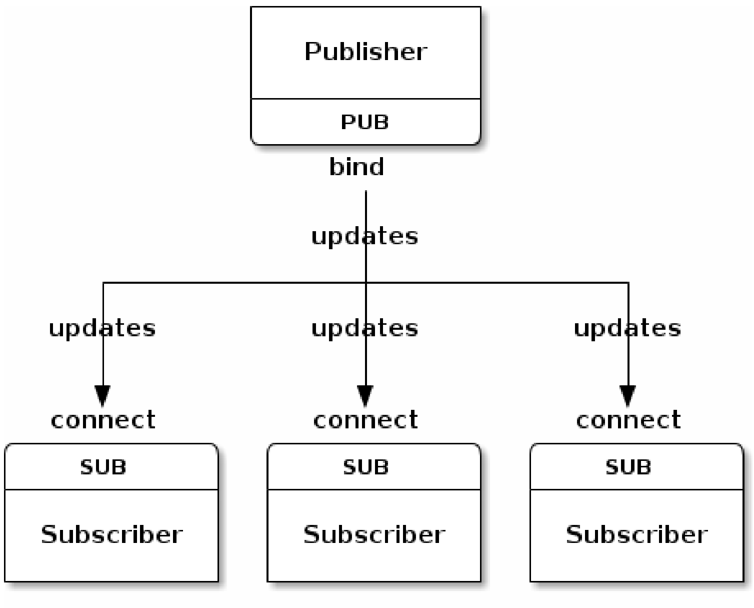


Figure 4 – Publish-Subscribe

Here is client application, which listens to the stream of updates and grabs anything to do with a specified zipcode, by default New York City because that's a great place to start any adventure:

```
//
//  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;
}
```

*wuclient.c: Weather update client*

Note that when you use a SUB socket you **must** set a subscription using zmq_setsockopt(3) and SUBSCRIBE, as in this code. If you don't set any subscription, you won't get any messages. It's a common mistake for beginners. The subscriber can set many subscriptions, which are added together. That is, if a update matches ANY subscription, the subscriber receives it. The subscriber can also unsubscribe specific subscriptions. Subscriptions are length-specified blobs. See zmq_setsockopt(3) for how this works.

The PUB-SUB socket pair is asynchronous. The client does zmq_recv(3), in a loop (or once if that's all it needs). Trying to send a message to a SUB socket will cause an error. Similarly the service does zmq_send(3) as often as it needs to, but must not do zmq_recv(3) on a PUB socket.

There is one important thing to know about PUB-SUB sockets: you do not know precisely when a subscriber starts to get messages. Even if you start a subscriber, wait a while, and then start the publisher, **the subscriber will always miss the first messages that the publisher sends**. This is because as the subscriber connects to the publisher (something that takes a small but non-zero time), the publisher may already be sending messages out.

This "slow joiner" symptom hits enough people, often enough, that I'm going to explain it in detail. Remember that ØMQ does asynchronous I/O, i.e. in the background. Say you have two nodes doing this, in this order:

- Subscriber connects to an endpoint and receives and counts messages.
- Publisher binds to an endpoint and immediately sends 1,000 messages.

Then the subscriber will most likely not receive anything. You'll blink, check that you set a correct filter, and try again, and the subscriber will still not receive anything.

Making a TCP connection involves to and fro handshaking that takes several milliseconds depending on your network and the number of hops between peers. In that time, ØMQ can send very many messages. For sake of argument assume it takes 5 msecs to establish a connection, and that same link can handle 1M messages per second. During the 5 msecs that the subscriber is connecting to the publisher, it takes the publisher only 1 msec to send out that 1K messages.

In Chapter Two I'll explain how to synchronize a publisher and subscribers so that you don't start to publish data until the subscriber(s) really are connected and ready. There is a simple and stupid way to delay the publisher, which is to sleep. I'd never do this in a real application though, it is extremely fragile as well as inelegant and slow. Use sleeps to prove to yourself what's happening, and then wait for Chapter 2 to see how to do this right.

The alternative to synchronization is to simply assume that the published data stream is infinite and has no start, and no end. This is how we built our weather client example.

So the client subscribes to its chosen zip-code and collects a thousand updates for that zip-code. That means about ten million updates from the server, if zip-codes are randomly distributed. You can start the client, and then the server, and the client will keep working. You can stop and restart the server as often as you like, and the client will keep working. When the client has collected its thousand updates, it calculates the average, prints it, and exits.

Some points about the publish-subscribe pattern:

- A subscriber can in fact connect to more than one publisher, using one 'connect' call each time. Data will then arrive and be interleaved so that no single publisher drowns out the others.
- If a publisher has no connected subscribers, then it will simply drop all messages.
- If you're using TCP, and a subscriber is slow, messages will queue up on the publisher. We'll look at how to protect publishers against this, using the "high-water mark" later.
- In the current versions of ØMQ, filtering happens at the subscriber side, not the publisher side. This means, over TCP, that a publisher will send all messages to all subscribers, which will then drop messages they don't want.

This is how long it takes to receive and filter 10M messages on my box, which is an Intel 4 core Q8300, fast but nothing special:

```
ph@ws200901:~/work/git/ØMQGuide/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
```

## Divide and Conquer

top prev next

As a final example (you are surely getting tired of juicy code and want to delve back into philological discussions about comparative abstractive norms), let's do a little supercomputing. Then coffee. Our supercomputing application is a fairly typical parallel processing model:

- We have a ventilator that produces tasks that can be done in parallel.
- We have a set of workers that process tasks.
- We have a sink that collects results back from the worker processes.

In reality, workers run on superfast boxes, perhaps using GPUs (graphic processing units) to do the hard maths. Here is the ventilator. It generates 100 tasks, each is a message telling the worker to sleep for some number of milliseconds:

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

int main (void)
{
    void *context = zmq_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 = randof (100) + 1;
        total_msec += workload;
        char string [10];
        sprintf (string, "%d", workload);
        s_send (sender, string);
    }
    printf ("Total expected cost: %d msec\n", total_msec);
    sleep (1);              //  Give 0MQ time to deliver

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

*taskvent.c: Parallel task ventilator*

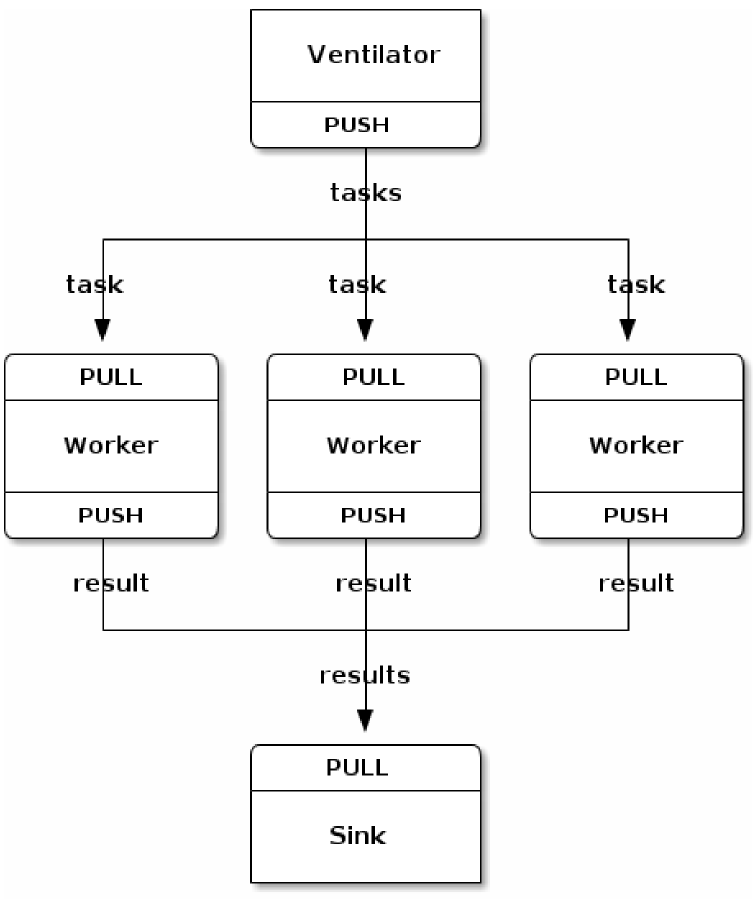


Figure 5 — Parallel Pipeline

Here is the worker application. It receives a message, sleeps for that number of seconds, then signals that it's finished:

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

int main (void)
{
    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);
        //  Simple progress indicator for the viewer
        fflush (stdout);
        printf ("%s.", string);

        //  Do the work
        s_sleep (atoi (string));
        free (string);

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

*taskwork.c: Parallel task worker*

Here is the sink application. It collects the 100 tasks, then calculates how long the overall processing took, so we can confirm that the workers really were running in parallel, if there are more than one of them:

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

int main (void)
{
    //  Prepare our context and socket
    void *context = zmq_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
    int64_t start_time = s_clock ();

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

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

*tasksink.c: Parallel task sink*

The average cost of a batch is 5 seconds. When we start 1, 2, 4 workers we get results like this from the sink:

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

Let's look at some aspects of this code in more detail:

- The workers connect upstream to the ventilator, and downstream to the sink. This means you can add workers arbitrarily. If the workers bound to their endpoints, you would need (a) more endpoints and (b) to modify the ventilator and/or the sink each time you added a worker. We say that the ventilator and sink are 'stable' parts of our architecture and the workers are 'dynamic' parts of it.

- We have to synchronize the start of the batch with all workers being up and running. This is a fairly common gotcha in ØMQ and there is no easy solution. The 'connect' method takes a certain time. So when a set of workers connect to the ventilator, the first one to successfully connect will get a whole load of messages in that short time while the others are also connecting. If you don't synchronize the start of the batch somehow, the system won't run in parallel at all. Try removing the wait, and see.

- The ventilator's PUSH socket distributes tasks to workers (assuming they are all connected *before* the batch starts going out) evenly. This is called *load-balancing* and it's something we'll look at again in more detail.

- The sink's PULL socket collects results from workers evenly. This is called *fair-queuing*:

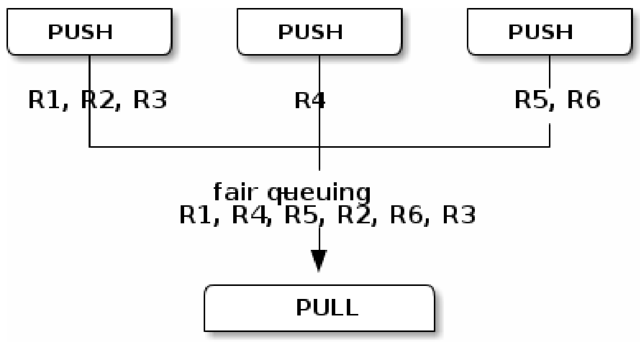


Figure 6 — Fair queuing

The pipeline pattern also exhibits the "slow joiner" syndrome, leading to accusations that PUSH sockets don't load balance properly. If you are using PUSH and PULL, and one of your workers gets way more messages than the others, it's because that PULL socket has joined faster than the others, and grabs a lot of messages before the others manage to connect.

## Programming with ØMQ

top prev next

Having seen some examples, you're eager to start using ØMQ in some apps. Before you start that, take a deep breath, chillax, and reflect on some basic advice that will save you stress and confusion.

- Learn ØMQ step by step. It's just one simple API but it hides a world of possibilities. Take the possibilities slowly, master each one.
- Write nice code. Ugly code hides problems and makes it hard for others to help you. You might get used to meaningless variable names, but people reading your code won't. Use names that are real words, that say something other than "I'm too careless to tell you what this variable is really for". Use consistent indentation, clean layout. Write nice code and your world will be more comfortable.
- Test what you make as you make it. When your program doesn't work, you should know what five lines are to blame. This is especially true when you do ØMQ magic, which just *won't* work the first times you try it.
- When you find that things don't work as expected, break your code into pieces, test each one, see which one is not working. ØMQ lets you make essentially modular code, use that to your advantage.
- Make abstractions (classes, methods, whatever) as you need them. If you copy/paste a lot of code you're going to copy/paste errors too.

To illustrate, here is a fragment of code someone asked me to help fix:

```
//  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,

    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;
}
```

This is what I rewrote it to, as part of finding the 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;
}
```

In the end, the problem was that the application was passing sockets between threads, which crashed weirdly. It became legal behavior in ØMQ/2.1, but remains dangerous and something we advise against doing.

## ØMQ/2.1

top prev next

History tells us that ØMQ/2.0 is when low-latency distributed messaging crawled out of the primeval mud, shook off a heavy coat of buzzwords and enterprise jargon, and reached its branches up to the sky, as if to cry, "no limits!". We've been using this stable branch since it spawned ØMQ/2.0.8 during the hot days of August, 2010.

But times change, and what was cool in 2010 is no longer *a la mode* in 2011. The ØMQ developers and community have been frantically busy redefining messaging chic, and anyone who's anyone knows that 2.1 is the new stable.

The Guide therefore assumes you're running 2.1.x. Let's look at the differences, as they affect your applications coming from the old 2.0:

- In 2.0, zmq_close(3) and zmq_term(3) discarded any in-flight messages, so it was unsafe to close a socket and terminate right after sending messages. In 2.1, these calls are safe: zmq_term will flush anything that's waiting to be sent. In 2.0 examples we often added a sleep(1) to get around the problem. In 2.1, this isn't needed.

- By contrast, in 2.0, it was safe to call zmq_term(3) even if there were open sockets. In 2.1, this is not safe, and it can cause zmq_term to block. So in 2.1 we *always close every socket*, before exiting. Furthermore, if you have any outgoing messages or connects waiting on a socket, 2.1 will by default wait forever trying to deliver these. You must *set the LINGER socket option* (e.g. to zero), on every socket which may still be busy, before calling zmq_term:

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

- In 2.0, zmq_poll(3) would return arbitrarily early, so you could not use it as a timer. We would work around this with a loop checked how much time was left, and called zmq_poll again as needed. In 2.1, zmq_poll properly waits for the full timeout if there are no events.

- In 2.0, ØMQ would ignore interrupted system calls, which meant that no libzmq call would ever return EINTR if a signal was received during its operation. This caused problems with loss of signals such as SIGINT (Ctrl-C handling), especially for language runtimes. In 2.1, any blocking call such as zmq_recv(3) will return EINTR if it is interrupted by a signal.

## Getting the Context Right

top prev next

ØMQ applications always start by creating a *context*, and then using that for creating sockets. In C, it's the zmq_init(3) call. You should create and use exactly one context in your process. Technically, the context is the container for all sockets in a single process, and acts as the transport for `inproc` sockets, which are the fastest way to connect threads in one process. If at runtime a process has two contexts, these are like separate ØMQ instances. If that's explicitly what you want, OK, but otherwise remember:

**Do one zmq_init(3) at the start of your main line code, and one zmq_term(3) at the**

**end.**

If you're using the fork() system call, each process needs its own context. If you do zmq_init(3) in the main process before calling fork(), the child processes get their own contexts. In general you want to do the interesting stuff in the child processes, and just manage these from the parent process.

## Making a Clean Exit

top prev next

Classy programmers share the same motto as classy hit men: always clean-up when you finish the job. When you use ØMQ in a language like Python, stuff gets automatically freed for you. But when using C you have to carefully free objects when you're finished with them, or you get memory leaks, unstable applications, and generally bad karma.

Memory leaks is one thing, but ØMQ is quite finicky about how you exit an application. The reasons are technical and painful but the upshot is that if you leave any sockets open, the zmq_term(3) function will hang forever. And even if you close all sockets, zmq_term(3) will by default wait forever if there are pending connects or sends. Unless you set the LINGER to zero on those sockets before closing them.

The ØMQ objects we need to worry about are messages, sockets, and contexts. Luckily it's quite simple, at least in simple programs:

- Always close a message the moment you are done with it, using zmq_msg_close(3).
- If you are opening and closing a lot of sockets, that's probably a sign you need to redesign your application.
- When you exit the program, close your sockets and then call zmq_term(3). This destroys the context.

If you're doing multithreaded work, it gets rather more complex than this. We'll get to multithreading in the next chapter, but because some of you will, despite warnings, will try to run before you can safely walk, below is the quick and dirty guide to making a clean exit in a *multithreaded* ØMQ application.

First, do not try to use the same socket from multiple threads. No, don't explain why you think this would be excellent fun, just please don't do it. Next, relingerfy and close all sockets, and terminate the context in the main thread. Lastly, this'll cause any blocking receives or polls or sends in attached threads (i.e. which share the same context) to return with an error. Catch that, and then relingerize and close sockets in *that* thread, and exit. Do not terminate the same context twice. The zmq_term in the main thread will block until all sockets it knows about are safely closed.

Voila! It's complex and painful enough that any language binding author worth his or her salt will do this automatically and make the socket closing dance unnecessary.

## Why We Needed ØMQ

top prev next

Now that you've seen ØMQ in action, let's go back to the "why".

Many applications these days consist of components that stretch across some kind of network, either a LAN or the Internet. So many application developers end up doing some kind of messaging. Some developers use message queuing products, but most of the time they do it themselves, using TCP or UDP. These protocols are not hard to use, but there is a great difference between sending a few bytes from A to B, and doing messaging in any kind of reliable way.

Let's look at the typical problems we face when we start to connect pieces using raw TCP. Any reusable messaging layer would need to solve all or most these:

- How do we handle I/O? Does our application block, or do we handle I/O in the background? This is a key design decision. Blocking I/O creates architectures that do not scale well. But background I/O can be very hard to do right.
- How do we handle dynamic components, i.e. pieces that go away temporarily? Do we formally split components into "clients" and "servers" and mandate that servers cannot disappear? What then if we want to connect servers to servers? Do we try to reconnect every few seconds?
- How do we represent a message on the wire? How do we frame data so it's easy to write and read, safe from buffer overflows, efficient for small messages, yet adequate for the very largest videos of dancing cats wearing party hats?
- How do we handle messages that we can't deliver immediately? Particularly, if we're waiting for a component to come back on-line? Do we discard messages, put them into a database, or into a memory queue?
- Where do we store message queues? What happens if the component reading from a queue is very slow, and causes our queues to build up? What's our strategy then?
- How do we handle lost messages? Do we wait for fresh data, request a resend, or do we build some kind of reliability layer that ensures messages cannot be lost? What if that layer itself crashes?
- What if we need to use a different network transport. Say, multicast instead of TCP unicast? Or IPv6? Do we need to rewrite the applications, or is the transport abstracted in some layer?
- How do we route messages? Can we send the same message to multiple peers? Can we send replies back to an original requester?
- How do we write an API for another language? Do we re-implement a wire-level protocol or do we repackage a library? If the former, how can we guarantee efficient and stable stacks? If the latter, how can we guarantee interoperability?
- How do we represent data so that it can be read between different architectures? Do we enforce a particular encoding for data types? How far is this the job of the messaging system rather than a higher layer?
- How do we handle network errors? Do we wait and retry, ignore them silently, or abort?

Take a typical open source project like Hadoop Zookeeper and read the C API code in src/c/src/zookeeper.c. It's 3,200 lines of mystery and in there is an undocumented, client-server network communication protocol. I see it's efficient because it uses poll() instead of select(). But really, Zookeeper should be using a generic messaging layer and an explicitly documented wire level protocol. It is incredibly wasteful for teams to be building this particular wheel over and over.

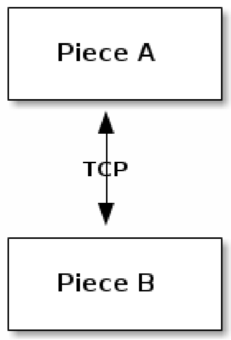


Figure 7 – Messaging as it starts

But how to make a reusable messaging layer? Why, when so many projects need this technology, are people still doing it the hard way, by driving TCP sockets in their code, and solving the problems in that long list, over and over?

It turns out that building reusable messaging systems is really difficult, which is why few FOSS projects ever tried, and why commercial messaging products are complex, expensive, inflexible, and brittle. In 2006 iMatix designed AMQP which started to give FOSS developers perhaps the first reusable recipe for a messaging system. AMQP works better than many other designs but remains relatively complex, expensive, and brittle. It takes weeks to learn to use, and months to create stable architectures that don't crash when things get hairy.

Most messaging projects, like AMQP, that try to solve this long list of problems in a reusable way do so by inventing a new concept, the "broker", that does addressing, routing, and queuing. This results in a client-server protocol or a set of APIs on top of some undocumented protocol, that let applications speak to this broker. Brokers are an excellent thing in reducing the complexity of large networks. But adding broker-based messaging to a product like Zookeeper would make it worse, not better. It would mean adding an additional big box, and a new single point of failure. A broker rapidly becomes a bottleneck and a new risk to manage. If the software supports it, we can add a second, third, fourth broker and make some fail-over scheme. People do this. It creates more moving pieces, more complexity, more things to break.

And a broker-centric set-up needs its own operations team. You literally need to watch the brokers day and night, and beat them with a stick when they start misbehaving. You need boxes, and you need backup boxes, and you need people to manage those boxes. It is only worth doing for large applications with many moving pieces, built by several teams of people, over several years.

So small to medium application developers are trapped. Either they avoid network programming, and make monolithic applications that do not scale. Or they jump into network programming and make brittle, complex applications that are hard to maintain. Or they bet on a messaging product, and end up with scalable applications that depend on expensive, easily broken technology. There has been no really good choice, which is maybe why messaging is largely stuck in the last century and stirs strong emotions. Negative ones for users, gleeful joy for those selling support and licenses.

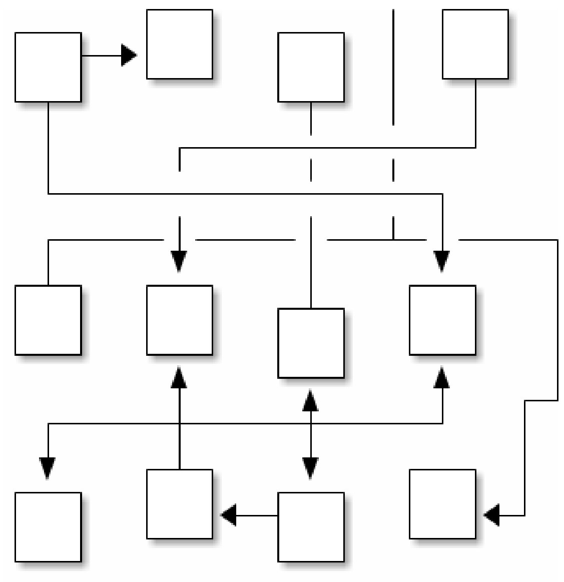


Figure 8 – Messaging as it becomes

What we need is something that does the job of messaging but does it in such a simple and cheap way that it can work in any application, with close to zero cost. It should be a library that you just link with, without any other dependencies. No additional moving pieces, so no additional risk. It should run on any OS and work with any programming language.

And this is ØMQ: an efficient, embeddable library that solves most of the problems an application needs to become nicely elastic across a network, without much cost.

Specifically:

- It handles I/O asynchronously, in background threads. These communicate with application threads using lock-free data structures, so ØMQ applications need no locks, semaphores, or other wait states.
- Components can come and go dynamically and ØMQ will automatically reconnect. This means you can start components in any order. You can create "service-oriented architectures" (SOAs) where services can join and leave the network at any time.
- It queues messages automatically when needed. It does this intelligently, pushing messages to as close as possible to the receiver before queuing them.
- It has ways of dealing with over-full queues (called "high water mark"). When a queue is full, ØMQ automatically blocks senders, or throws away messages, depending on the kind of messaging you are doing (the so-called "pattern").
- It lets your applications talk to each other over arbitrary transports: TCP, multicast, in-process, inter-process. You don't need to change your code to use a different transport.
- It handles slow/blocked readers safely, using different strategies that depend on the messaging pattern.
- It lets you route messages using a variety of patterns such as request-reply and publish-subscribe. These patterns are how you create the topology, the structure of your network.
- It lets you place pattern-extending "devices" (small brokers) in the network when you need to reduce the complexity of interconnecting many pieces.
- It delivers whole messages exactly as they were sent, using a simple framing on the wire. If you write a 10k message, you will receive a 10k message.

- It does not impose any format on messages. They are blobs of zero to gigabytes large. When you want to represent data you choose some other product on top, such as Google's protocol buffers, XDR, and others.

- It handles network errors intelligently. Sometimes it retries, sometimes it tells you an operation failed.

- It reduces your carbon footprint. Doing more with less CPU means your boxes use less power, and you can keep your old boxes in use for longer. Al Gore would love ØMQ.

Actually ØMQ does rather more than this. It has a subversive effect on how you develop network-capable applications. Superficially it's just a socket API on which you do zmq_recv(3) and zmq_send(3). But message processing rapidly becomes the central loop, and your application soon breaks down into a set of message processing tasks. It is elegant and natural. And it scales: each of these tasks maps to a node, and the nodes talk to each other across arbitrary transports. Two nodes in one process (node is a thread), two nodes on one box (node is a process), two boxes on one network (node is a box). With no application code changes.

## Socket Scalability

top prev next

Let's see ØMQ's scalability in action. Here is a shell script that starts the weather server and then a bunch of clients in parallel:

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

As the clients run, we take a look at the active processes using 'top', and we see something like (on a 4-core box):

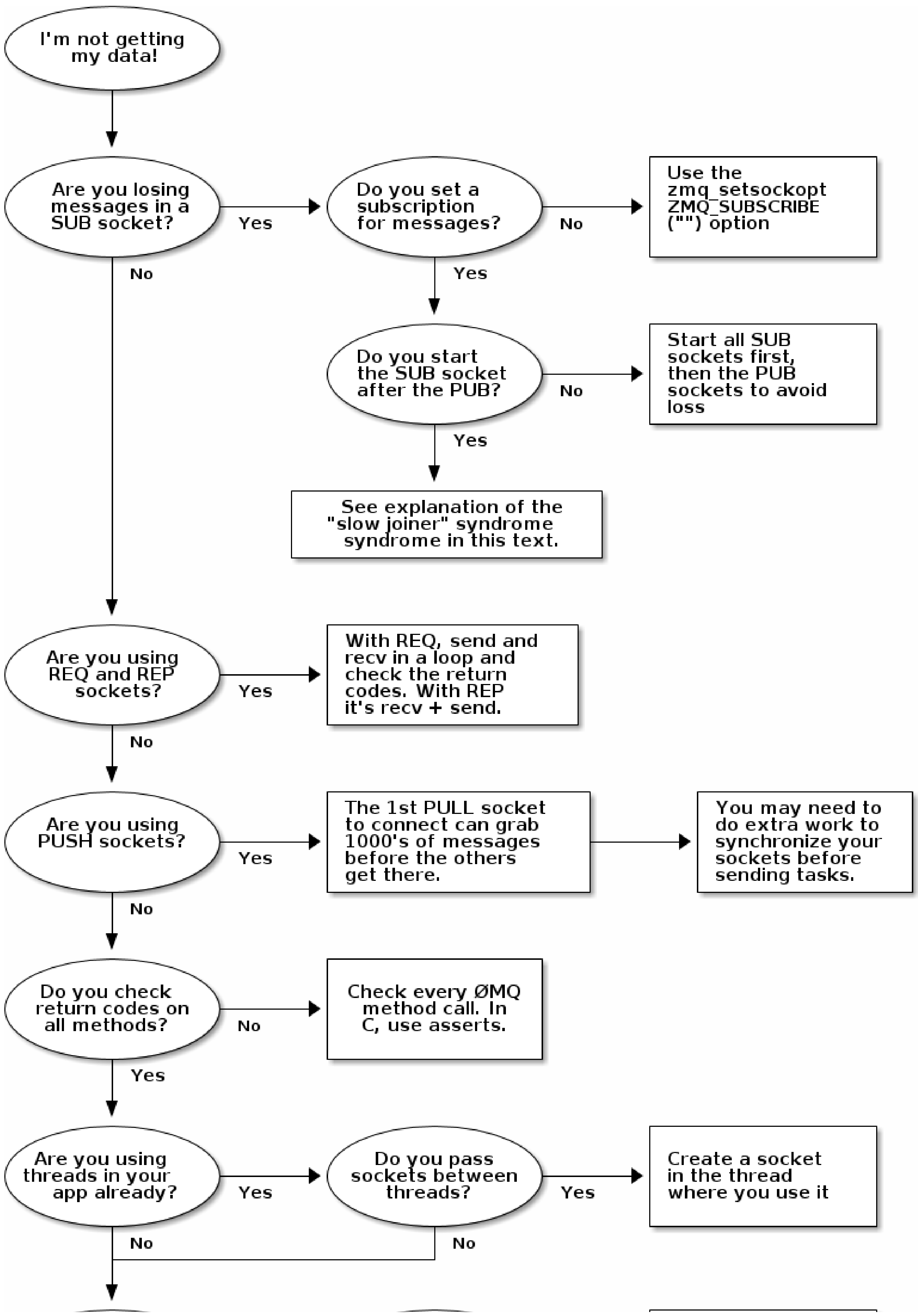
```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7136 ph        20   0 1040m 959m 1156 R  157 12.0  16:25.47 wuserver
 7966 ph        20   0 98608 1804 1372 S   33  0.0   0:03.94 wuclient
 7963 ph        20   0 33116 1748 1372 S   14  0.0   0:00.76 wuclient
 7965 ph        20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

Let's think for a second about what is happening here. The weather server has a single socket, and yet here we have it sending data to five clients in parallel. We could have thousands of concurrent clients. The server application doesn't see them, doesn't talk to them directly.

## Missing Message Problem Solver

top prev next

As you start to program with ØMQ you will come across one problem more than once: you lose messages that you expect to receive. Here is a basic problem solver that walks through the most common causes for this. Don't worry if some of the terminology is unfamiliar still, it'll become clearer in the next chapters.


I'm not getting my data!
Are you losing messages in a SUB socket?
Yes
No
Do you set a subscription for messages?
No
Use the zmq_setsockopt ZMQ_SUBSCRIBE ("") option
Yes
Do you start the SUB socket after the PUB?
No
Start all SUB sockets first, then the PUB sockets to avoid loss
Yes
See explanation of the "slow joiner" syndrome syndrome in this text.
Are you using REQ and REP sockets?
Yes
With REQ, send and recv in a loop and check the return codes. With REP it's recv + send.
No
Are you using PUSH sockets?
Yes
The 1st PULL socket to connect can grab 1000's of messages before the others get there.
You may need to do extra work to synchronize your sockets before sending tasks.
No
Do you check return codes on all methods?
No
Check every ØMQ method call. In C, use asserts.
Yes
Are you using threads in your app already?
Yes
Do you pass sockets between threads?
Yes
Create a socket in the thread where you use it
No
No

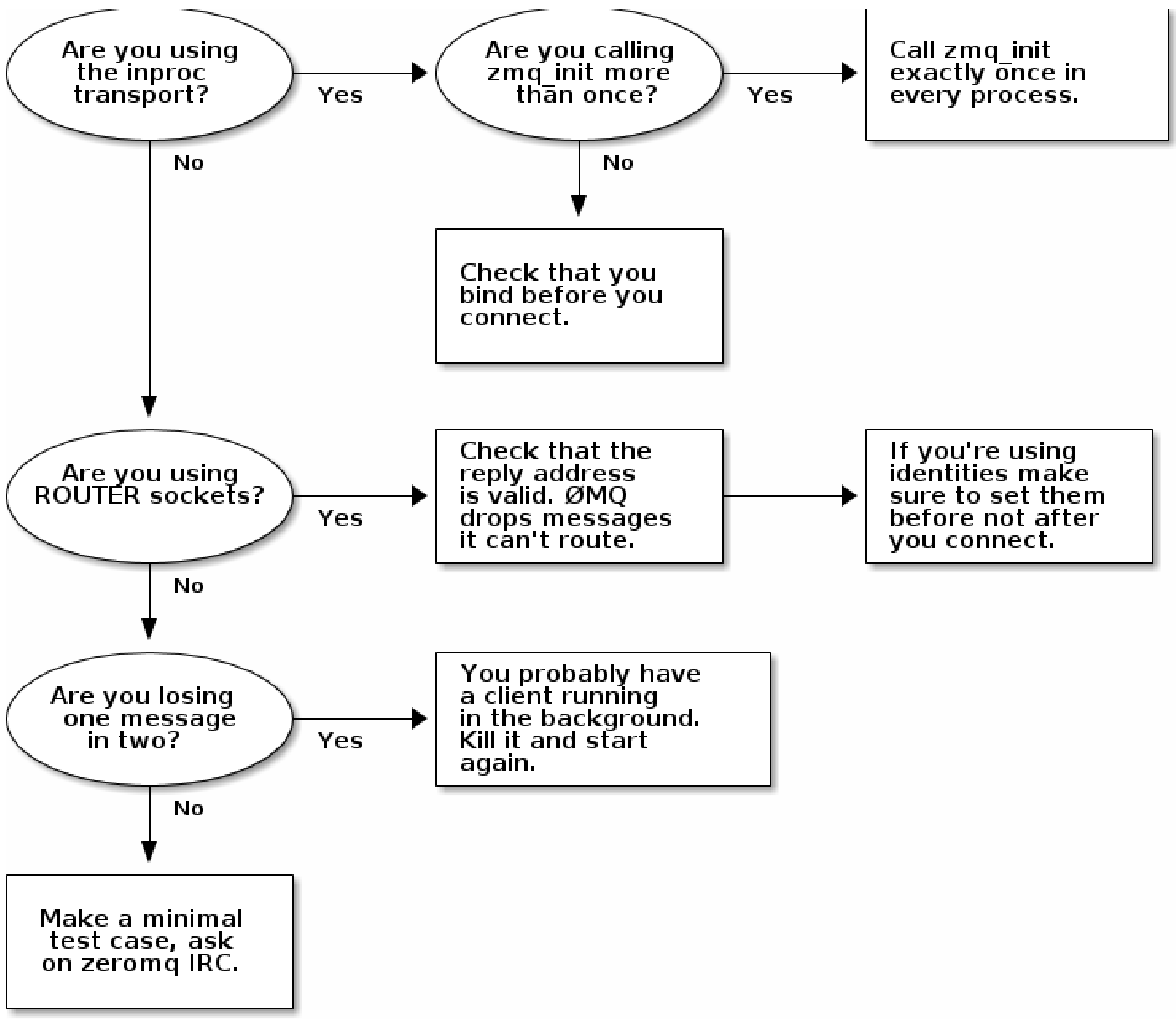

Are you using the inproc transport?
Yes
No
Are you calling zmq_init more than once?
Yes
No
Call zmq_init exactly once in every process.
Check that you bind before you connect.
Are you using ROUTER sockets?
Yes
No
Check that the reply address is valid. ØMQ drops messages it can't route.
If you're using identities make sure to set them before not after you connect.
Are you losing one message in two?
Yes
No
You probably have a client running in the background. Kill it and start again.
Make a minimal test case, ask on zeromq IRC.

Figure 9 – Missing Message Problem Solver

If you're using ØMQ in a context where failures are expensive, then you want plan properly. First, build prototypes that let you learn and test the different aspects of your design. Stress them until they break, so that you know exactly how strong your designs are. Second, invest in testing. This means building test frameworks, ensuring you have access to realistic setups with sufficient computer power, and getting time or help to actually test seriously. Ideally, one team writes the code, a second team tries to break it. Lastly, do get your organization to contact iMatix to discuss how we can help to make sure things work properly, and can be fixed rapidly if they break.

In short: if you have not proven an architecture works in realistic conditions, it will most likely break at the worst possible moment.

## Warning - Unstable Paradigms!

top prev next

Traditional network programming is built on the general assumption that one socket talks to one connection, one peer. There are multicast protocols but they are exotic. When we assume "one socket = one connection", we scale our architectures in certain ways. We create threads of logic where each thread work with one socket, one peer. We place intelligence and state in these threads.

In the ØMQ universe, sockets are clever multithreaded applications that manage a whole set of connections automagically for you. You can't see, work with, open, close, or attach state to these connections. Whether you use blocking send or receive, or poll, all you can talk to is the socket, not the connections it manages for you. The connections are private and invisible, and this is the key to ØMQ's scalability.

Because your code, talking to a socket, can then handle any number of connections across whatever network protocols are around, without change. A messaging pattern sitting in ØMQ can scale more cheaply than a messaging pattern sitting in your application code.

So the general assumption no longer applies. As you read the code examples, your brain will try to map them to what you know. You will read "socket" and think "ah, that represents a connection to another node". That is wrong. You will read "thread" and your brain will again think, "ah, a thread represents a connection to another node", and again your brain will be wrong.

If you're reading this Guide for the first time, realize that until you actually write ØMQ code for a day or two (and maybe three or four days), you may feel confused, especially by how simple ØMQ makes things for you, and you may try to impose that general assumption on ØMQ, and it won't work. And then you will experience your moment of enlightenment and trust, that *zap-pow-kaboom* satori paradigm-shift moment when it all becomes clear.

# Chapter Two - Intermediate Stuff

top prev next

In Chapter One we took ØMQ for a drive, with some basic examples of the main ØMQ patterns: request-reply, publish-subscribe, and pipeline. In this chapter we're going to get our hands dirty and start to learn how to use these tools in real programs.

We'll cover:

- How to create and work with ØMQ sockets.
- How to send and receive messages on sockets.
- How to build your apps around ØMQ's asynchronous I/O model.
- How to handle multiple sockets in one thread.
- How to handle fatal and non-fatal errors properly.
- How to handle interrupt signals like Ctrl-C.
- How to shutdown a ØMQ application cleanly.
- How to check a ØMQ application for memory leaks.
- How to send and receive multipart messages.
- How to forward messages across networks.
- How to build a simple message queuing broker.
- How to write multithreaded applications with ØMQ.
- How to use ØMQ to signal between threads.
- How to use ØMQ to coordinate a network of nodes.
- How to create durable sockets using socket identities.
- How to create and use message envelopes for publish-subscribe.
- How to do make durable subscribers that can recover from crashes.
- Using the high-water mark (HWM) to protect against memory overflows.

## The Zen of Zero

top prev next

The Ø in ØMQ is all about tradeoffs. On the one hand this strange name lowers ØMQ's visibility on Google and Twitter. On the other hand it annoys the heck out of some Danish folk who write us things like "ØMG røtfl", and "*Ø is not a funny looking zero!*" and "*Rødgrød med Fløde!*", which is apparently an insult that means "may your neighbours be the direct descendents of Grendel!" Seems like a fair trade.

Originally the zero in ØMQ was meant as "zero broker" and (as close to) "zero latency" (as possible). In the meantime it has come to cover different goals: zero administration, zero cost, zero waste. More generally, "zero" refers to the culture of minimalism that permeates the project. We add power by removing complexity rather than exposing new functionality.

## The Socket API

top prev next

To be perfectly honest, ØMQ does a kind of switch-and-bait on you. Which we don't apologize for, it's for your own good and hurts us more than it hurts you. It presents a familiar BSD socket API but that hides a bunch of message-processing machines that will slowly fix your world-view about how to design and write distributed software.

Sockets are the de-facto standard API for network programming, as well as being useful for stopping your eyes from falling onto your cheeks. One thing that makes ØMQ especially tasty to developers is that it uses a standard socket API. Kudos to Martin Sustrik for pulling this idea off. It turns "Message Oriented Middleware", a phrase guaranteed to send the whole room off to Catatonia, into "Extra Spicy Sockets!" which leaves us with a strange craving for pizza, and a desire to know more.

Like a nice pepperoni pizza, ØMQ sockets are easy to digest. Sockets have a life in four parts, just like BSD sockets:

- Creating and destroying sockets, which go together to form a karmic circle of socket life (see zmq_socket(3), zmq_close(3)).
- Configuring sockets by setting options on them and checking them if necessary (see zmq_setsockopt(3), zmq_getsockopt(3)).
- Plugging sockets onto the network topology by creating ØMQ connections to and from them (see zmq_bind(3), zmq_connect(3)).
- Using the sockets to carry data by writing and receiving messages on them (see zmq_send(3), zmq_recv(3)).

Which looks like this, in C:

```
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);
```

Note that sockets are always void pointers, and messages (which we'll come to very soon) are structures. So in C you pass sockets as-such, but you pass addresses of messages in all functions that work with messages, like zmq_send(3) and zmq_recv(3). As a mnemonic, realize that "in ØMQ all ur sockets are belong to us", but messages are things you actually own in your code.

Creating, destroying, and configuring sockets works as you'd expect for any object. But remember that ØMQ is an asynchronous, elastic fabric. This has some impact on how we plug sockets into the network topology, and how we use the sockets after that.

## Plugging Sockets Into the Topology

top prev next

To create a connection between two nodes you use zmq_bind(3) in one node, and zmq_connect(3) in the other. As a general rule of thumb, the node which does zmq_bind(3) is a "server", sitting on a well-known network address, and the node which does zmq_connect(3) is a "client", with unknown or arbitrary network addresses. Thus we say that we "bind a socket to an endpoint" and "connect a socket to an endpoint", the endpoint being that well-known network address.

ØMQ connections are somewhat different from old-fashioned TCP connections. The main notable differences are:

- They go across an arbitrary transport (`inproc`, `ipc`, `tcp`, `pgm` or `epgm`). See zmq_inproc(7), zmq_ipc(7), zmq_tcp(7), zmq_pgm(7), and zmq_epgm(7).
- They exist when a client does zmq_connect(3) to an endpoint, whether or not a server has already done zmq_bind(3) to that endpoint.
- They are asynchronous, and have queues that magically exist where and when needed.
- They may express a certain "messaging pattern", according to the type of socket used at each end.
- One socket may have many outgoing and many incoming connections.
- There is no zmq_accept() method. When a socket is bound to an endpoint it automatically starts accepting connections.
- Your application code cannot work with these connections directly; they are encapsulated under the socket.

Many architectures follow some kind of client-server model, where the server is the component that is most stable, and the clients are the components that are most dynamic, i.e. they come and go the most. There are sometimes issues of addressing: servers will be visible to clients, but not necessarily vice-versa. So mostly it's obvious which node should be doing zmq_bind(3) (the server) and which should be doing zmq_connect(3) (the client). It also depends on the kind of sockets you're using, with some exceptions for unusual network architectures. We'll look at socket types later.

Now, imagine we start the client *before* we start the server. In traditional networking we get a big red Fail flag. But ØMQ lets us start and stop pieces arbitrarily. As soon as the client node does zmq_connect(3) the connection exists and that node can start to write messages to the socket. At some stage (hopefully before messages queue up so much that they start to get discarded, or the client blocks), the server comes alive, does a zmq_bind(3) and ØMQ starts to deliver messages.

A server node can bind to many endpoints and it can do this using a single socket. This means it will accept connections across different transports:

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

You cannot bind to the same endpoint twice, that will cause an exception.

Each time a client node does a zmq_connect(3) to any of these endpoints, the server node's socket gets another connection. There is no inherent limit to how many connections a socket can have. A client node can also connect to many endpoints using a single socket.

In most cases, which node acts as client, and which as server, is about network topology rather than message flow. However, there *are* cases (resending when connections are broken) where the same socket type will behave differently if it's a server or if it's a client.

What this means is that you should always think in terms of "servers" as stable parts of your topology, with more-or-less fixed endpoint addresses, and "clients" as dynamic parts that come and go. Then, design your application around this model. The chances that it will "just work" are much better like that.

Sockets have types. The socket type defines the semantics of the socket, its policies for routing messages inwards and outwards, queueing, etc. You can connect certain types of socket together, e.g. a publisher socket and a subscriber socket. Sockets work together in "messaging patterns". We'll look at this in more detail later.

It's the ability to connect sockets in these different ways that gives ØMQ its basic power as a message queuing system. There are layers on top of this, such as devices and topic routing, which we'll get to later. But essentially, with ØMQ you define your network architecture by plugging pieces together like a child's construction toy.

## Using Sockets to Carry Data

top prev next

To send and receive messages you use the zmq_send(3) and zmq_recv(3) methods. The names are conventional but ØMQ's I/O model is different enough from TCP's model that you will need time to get your head around it.

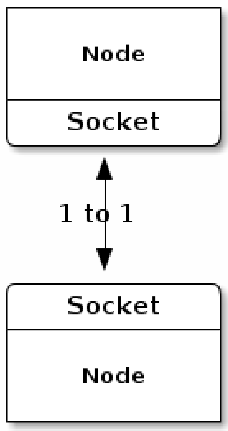


Figure 10 — TCP sockets are 1 to 1

Let's look at the main differences between TCP sockets and ØMQ sockets when it comes to carrying data:

- ØMQ sockets carry messages, rather than bytes (as in TCP) or frames (as in UDP). A message is a length-specified blob of binary data. We'll come to messages shortly, their design is optimized for performance and thus somewhat tricky to understand.
- ØMQ sockets do their I/O in a background thread. This means that messages arrive in a local input queue, and are sent from a local output queue, no matter what your application is busy doing. These are configurable memory queues, by the way.
- ØMQ sockets can, depending on the socket type, be connected to (or from, it's the same) many other sockets. Where TCP emulates a one-to-one phone call, ØMQ implements one-to-many (like a radio broadcast), many-to-many (like a post office), many-to-one (like a mail box), and even one-to-one.
- ØMQ sockets can send to many endpoints (creating a fan-out model), or receive from many endpoints (creating a fan-in model).

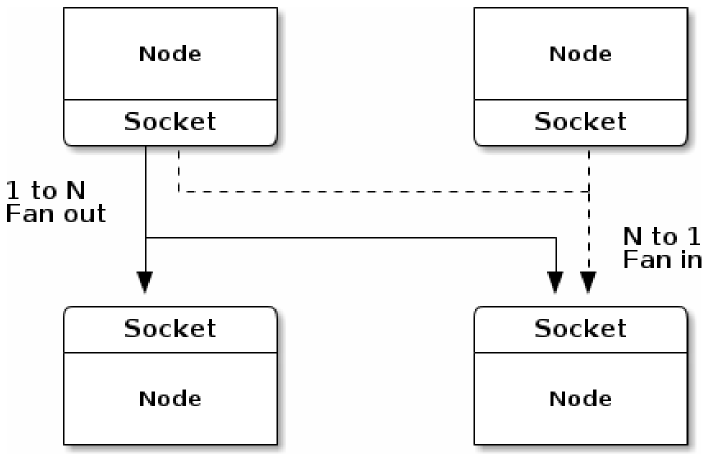


Figure 11 – ØMQ sockets are N to N

So writing a message to a socket may send the message to one or many other places at once, and conversely, one socket will collect messages from all connections sending messages to it. The zmq_recv(3) method uses a fair-queuing algorithm so each sender gets an even chance.

The zmq_send(3) method does not actually send the message to the socket connection(s). It queues the message so that the I/O thread can send it asynchronously. It does not block except in some exception cases. So the message is not necessarily sent when zmq_send(3) returns to your application. If you created a message using zmq_msg_init_data(3) you cannot reuse the data or free it, otherwise the I/O thread will rapidly find itself writing overwritten or unallocated garbage. This is a common mistake for beginners. We'll see a little later how to properly work with messages.

## Unicast Transports

top prev next

ØMQ provides a set of unicast transports (`inproc`, `ipc`, and `tcp`) and multicast transports (epgm, pgm). Multicast is an advanced technique that we'll come to later. Don't even start using it unless you know that your fanout ratios will make 1-to-N unicast impossible.

For most common cases, use `tcp`, which is a *disconnected TCP* transport. It is elastic, portable, and fast enough for most cases. We call this 'disconnected' because ØMQ's `tcp` transport doesn't require that the endpoint exists before you connect to it. Clients and servers can connect and bind at any time, can go and come back, and it remains transparent to applications.

The inter-process transport, `ipc`, is like `tcp` except that it is abstracted from the LAN, so you don't need to specify IP addresses or domain names. This makes it better for some purposes, and we use it quite often in the examples in this book. ØMQ's `ipc` transport is disconnected, like `tcp`. It has one limitation: it does not work on Windows. This may be fixed in future versions of ØMQ. By convention we use endpoint names with an ".ipc" extension to avoid potential conflict with other file names. On UNIX systems, if you use `ipc` endpoints you need to create these with appropriate permissions otherwise they may not be shareable between processes running under different user ids. You must also make sure all processes can access the files, e.g. by running in the same working directory.

The inter-thread transport, `inproc`, is a connected signaling transport. It is much faster than `tcp` or `ipc`. This transport has a specific limitation compared to `ipc` and `tcp`: **you must do bind before connect**. This is something future versions of ØMQ may fix, but at present this defines you use `inproc` sockets. We create and bind one socket, start the child threads, which create and connect the other sockets.

## ØMQ is Not a Neutral Carrier

top prev next

A common question that newcomers to ØMQ ask (it's one I asked myself) is something like, "*how do I write a XYZ server in ØMQ?*" For example, "how do I write an HTTP server in ØMQ?"

The implication is that if we use normal sockets to carry HTTP requests and responses, we should be able to use ØMQ sockets to do the same, only much faster and better.

Sadly the answer is "this is not how it works". ØMQ is not a neutral carrier, it imposes a framing on the transport protocols it uses. This framing is not compatible with existing protocols, which tend to use their own framing. For example, here is an HTTP request, and a ØMQ request, both over TCP/IP:

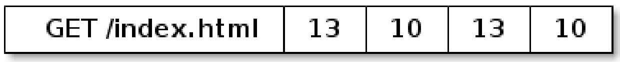


Figure 12 – HTTP request

Where the HTTP request uses CR-LF as its simplest framing delimiter, and ØMQ uses a length-specified frame:

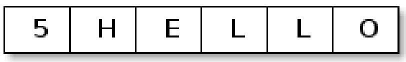


Figure 13 – ØMQ request

So you could write a HTTP-like protocol using ØMQ, using for example the request-reply socket pattern. But it would not be HTTP.

There is however a good answer to the question, "how can I make profitable use of ØMQ when making my new XYZ server?" You need to implement whatever protocol you want to speak in any case, but you can connect that protocol server (which can be extremely thin) to a ØMQ backend that does the real work. The beautiful part here is that you can then extend your backend with code in any language, running locally or remotely, as you wish. Zed Shaw's Mongrel2 web server is a great example of such an architecture.

## I/O Threads

top prev next

We said that ØMQ does I/O in a background thread. One I/O thread (for all sockets) is sufficient for all but the most extreme applications. This is the magic '1' that we use when creating a context, meaning "use one I/O thread":

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

There is a major difference between a ØMQ application and a conventional networked application, which is that you don't create one socket per connection. One socket handles all incoming and outcoming connections for a particular point of work. E.g. when you publish to a thousand subscribers, it's via one socket. When you distribute work among twenty services, it's via one socket. When you collect data from a thousand web applications, it's via one socket.

This has a fundamental impact on how you write applications. A traditional networked application has one process or one thread per remote connection, and that process or thread handles one socket. ØMQ lets you collapse this entire structure into a single thread, and then break it up as necessary for scaling.

## Core Messaging Patterns

top prev next

Underneath the brown paper wrapping of ØMQ's socket API lies the world of messaging patterns. If you have a background in enterprise messaging, these will be vaguely familiar. But to most ØMQ newcomers they are a surprise, we're so used to the TCP paradigm where a socket represents another node.

Let's recap briefly what ØMQ does for you. It delivers blobs of data (messages) to nodes, quickly and efficiently. You can map nodes to threads, processes, or boxes. It gives your applications a single socket API to work with, no matter what the actual transport (like in-process, inter-process, TCP, or multicast). It automatically reconnects to peers as they come and go. It queues messages at both sender and receiver, as needed. It manages these queues carefully to ensure processes don't run out of memory, overflowing to disk when appropriate. It handles socket errors. It does all I/O in background threads. It uses lock-free techniques for talking between nodes, so there are never locks, waits, semaphores, or deadlocks.

But cutting through that, it routes and queues messages according to precise recipes called *patterns*. It is these patterns that provide ØMQ's intelligence. They encapsulate our hard-earned experience of the best ways to distribute data and work. ØMQ's patterns are hard-coded but future versions may allow user-definable patterns.

ØMQ patterns are implemented by pairs of sockets with matching types. In other words, to understand ØMQ patterns you need to understand socket types and how they work together. Mostly this just takes learning, there is little that is obvious at this level.

The built-in core ØMQ patterns are:

- **Request-reply**, which connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- **Publish-subscribe**, which connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- **Pipeline**, connects nodes in a fan-out / fan-in pattern that can have multiple steps, and loops. This is a parallel task distribution and collection pattern.

We looked at each of these in the first chapter. There's one more pattern that people tend to try to use when they still think of ØMQ in terms of traditional TCP sockets:

- **Exclusive pair**, which connects two sockets in an exclusive pair. This is a low-level pattern for specific, advanced use-cases. We'll see an example at the end of this chapter.

The zmq_socket(3) man page is fairly clear about the patterns, it's worth reading several times until it starts to make sense. We'll look at each pattern and the use-cases it covers.

These are the socket combinations that are valid for a connect-bind pair (either side can bind):

- PUB and SUB
- REQ and REP
- REQ and ROUTER
- DEALER and REP
- DEALER and ROUTER
- DEALER and DEALER
- ROUTER and ROUTER
- PUSH and PULL
- PAIR and PAIR

Any other combination will produce undocumented and unreliable results and future versions of ØMQ will probably return errors if you try them. You can and will of course bridge other socket types *via code*, i.e. read from one socket type and write to another.

## High-level Messaging Patterns

top prev next

These four core patterns are cooked-in to ØMQ. They are part of the ØMQ API, implemented in the core C++ library, and guaranteed to be available in all fine retail stores. If one day the Linux kernel includes ØMQ, for example, these patterns would be there.

On top, we add *high-level patterns*. We build these high-level patterns on top of ØMQ and implement them in whatever language we're using for our application. They are not part of the core library, do not come with the ØMQ package, and exist in their own space, as part of the ØMQ community.

One of the things we aim to provide you with this guide are a set of such high-level patterns, both small (how to handle messages sanely) to large (how to make a reliable publish-subscribe architecture).

## Working with Messages

top prev next

On the wire, ØMQ messages are blobs of any size from zero upwards, fitting in memory. You do your own serialization using Google Protocol Buffers, XDR, JSON, or whatever else your applications need to speak. It's wise to choose a data representation that is portable and fast, but you can make your own decisions about trade-offs.

In memory, ØMQ messages are zmq_msg_t structures (or classes depending on your language). Here are the basic ground rules for using ØMQ messages in C:

- You create and pass around zmq_msg_t objects, not blocks of data.

- To read a message you use zmq_msg_init(3) to create an empty message, and then you pass that to zmq_recv(3).

- To write a message from new data, you use zmq_msg_init_size(3) to create a message and at the same time allocate a block of data of some size. You then fill that data using memcpy, and pass the message to zmq_send(3).

- To release (not destroy) a message you call zmq_msg_close(3). This drops a reference, and eventually ØMQ will destroy the message.
- To access the message content you use zmq_msg_data(3). To know how much data the message contains, use zmq_msg_size(3).
- Do not use zmq_msg_move(3), zmq_msg_copy(3), or zmq_msg_init_data(3) unless you read the man pages and know precisely why you need these.

Here is a typical chunk of code working with messages, which should be familiar if you have been paying attention. This is from the zhelpers.h file we use in all the examples:

```
//  Receive 0MQ 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 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);
}
```

You can easily extend this code to send and receive blobs of arbitrary length.

**Note than when you have passed a message to zmq_send(3), ØMQ will clear the message, i.e. set the size to zero. You cannot send the same message twice, and you cannot access the message data after sending it.**

If you want to send the same message more than once, create a second message, initialize it using zmq_msg_init(3) and then use zmq_msg_copy(3) to create a copy of the first message. This does not copy the data but the reference. You can then send the message twice (or more, if you create more copies) and the message will only be finally destroyed when the last copy is sent or closed.

ØMQ also supports *multipart* messages, which let you handle a list of blobs as a single message. This is widely used in real applications and we'll look at that later in this chapter and in Chapter Three.

Some other things that are worth knowing about messages:

- ØMQ sends and receives them atomically, i.e. you get a whole message, or you don't get it at all.

- ØMQ does not send a message right away but at some indeterminate later time.
- You can send zero-length messages, e.g. for sending a signal from one thread to another.
- A message must fit in memory. If you want to send files of arbitrary sizes, you should break them into pieces and send each piece as a separate message.
- You must call zmq_msg_close(3) when finished with a message, in languages that don't automatically destroy objects when a scope closes.

And to be necessarily repetitive, do not use zmq_msg_init_data(3), yet. This is a zero-copy method and guaranteed to create trouble for you. There are far more important things to learn about ØMQ before you start to worry about shaving off microseconds.

## Handling Multiple Sockets

top prev next

In all the examples so far, the main loop of most examples has been:

1. wait for message on socket
2. process message
3. repeat

What if we want to read from multiple sockets at the same time? The simplest way is to connect one socket to multiple endpoints and get ØMQ to do the fanin for us. This is legal if the remote endpoints are in the same pattern but it would be illegal to e.g. connect a PULL socket to a PUB endpoint. Fun, but illegal. If you start mixing patterns you break future scalability.

The right way is to use zmq_poll(3). An even better way might be to wrap zmq_poll(3) in a framework that turns it into a nice event-driven *reactor*, but it's significantly more work than we want to cover here.

Let's start with a dirty hack, partly for the fun of not doing it right, but mainly because it lets me show you how to do non-blocking socket reads. Here is a simple example of reading from two sockets using non-blocking reads. This rather confused program acts both as a subscriber to weather updates, and a worker for parallel tasks:

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

int main (void)
{
    //  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
        s_sleep (1);
    }
    //  We never get here but clean up anyhow
    zmq_close (receiver);
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

*msreader.c: Multiple socket reader*

The cost of this approach is some additional latency on the first message (the sleep at the end of the loop, when there are no waiting messages to process). This would be a problem in applications where sub-millisecond latency was vital. Also, you need to check the documentation for nanosleep() or whatever function you use to make sure it does not busy-loop.

You can treat the sockets fairly by reading first from one, then the second rather than prioritizing them as we did in this example. This is called "fair-queuing", something that ØMQ does automatically when one socket receives messages from more than one source.

Now let's see the same little senseless application done right, using zmq_poll(3):

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

int main (void)
{
    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;
}
```

*mspoller.c: Multiple socket poller*

## Handling Errors and ETERM

top prev next

ØMQ's error handling philosophy is a mix of fail-fast and resilience. Processes, we believe, should be as vulnerable as possible to internal errors, and as robust as possible against external attacks and errors. To give an analogy, a living cell will self-destruct if it detects a single internal error, yet it will resist attack from the outside by all means possible. Assertions, which pepper the ØMQ code, are absolutely vital to robust code, they just have to be on the right side of the cellular wall. And there should be such a wall. If it is unclear whether a fault is internal or external, that is a design flaw that needs to be fixed.

In C, assertions stop the application immediately with an error. In other languages you may get exceptions or halts.

When ØMQ detects an external faults it returns an error to the calling code. In some rare cases it drops messages silently, if there is no obvious strategy for recovering from the error. In a few places ØMQ still asserts on external faults, but these are considered bugs.

In most of the C examples we've seen so far there's been no error handling. **Real code should do error handling on every single ØMQ call**. If you're using a language binding other than C, the binding may handle errors for you. In C you do need to do this yourself. There are some simple rules, starting with POSIX conventions:

- Methods that create objects will return NULL in case they fail.
- Other methods will return 0 on success and other values (mostly -1) on an exceptional condition (usually failure).
- The error code is provided in `errno` or zmq_errno(3).
- A descriptive error text for logging is provided by zmq_strerror(3).

There are two main exceptional conditions that you may want to handle as non-fatal:

- When a thread calls zmq_recv(3) with the NOBLOCK option and there is no waiting data. ØMQ will return -1 and set errno to EAGAIN.
- When a thread calls zmq_term(3) and other threads are doing blocking work. The zmq_term(3) call closes the context and all blocking calls exit with -1, and errno set to ETERM.

What this boils down to is that in most cases you can use assertions on ØMQ calls, like this, in C:

```
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);
```

In the first version of this code I put the assert() call around the function. Not a good idea, since an optimized build will turn all assert() macros to null and happily wallop those functions. Use a return code, and assert the return code.

Let's see how to shut down a process cleanly. We'll take the parallel pipeline example from the previous section. If we've start a whole lot of workers in the background, we now want to kill them when the batch is finished. Let's do this by sending a kill message to the workers. The best place to do this is the sink, since it really knows when the batch is done.

How do we connect the sink to the workers? The PUSH/PULL sockets are one-way only. The standard ØMQ answer is: create a new socket flow for each type of problem you need to solve. We'll use a publish-subscribe model to send kill messages to the workers:

- The sink creates a PUB socket on a new endpoint.
- Workers bind their input socket to this endpoint.
- When the sink detects the end of the batch it sends a kill to its PUB socket.
- When a worker detects this kill message, it exits.

It doesn't take much new code in the sink:

```
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);
```

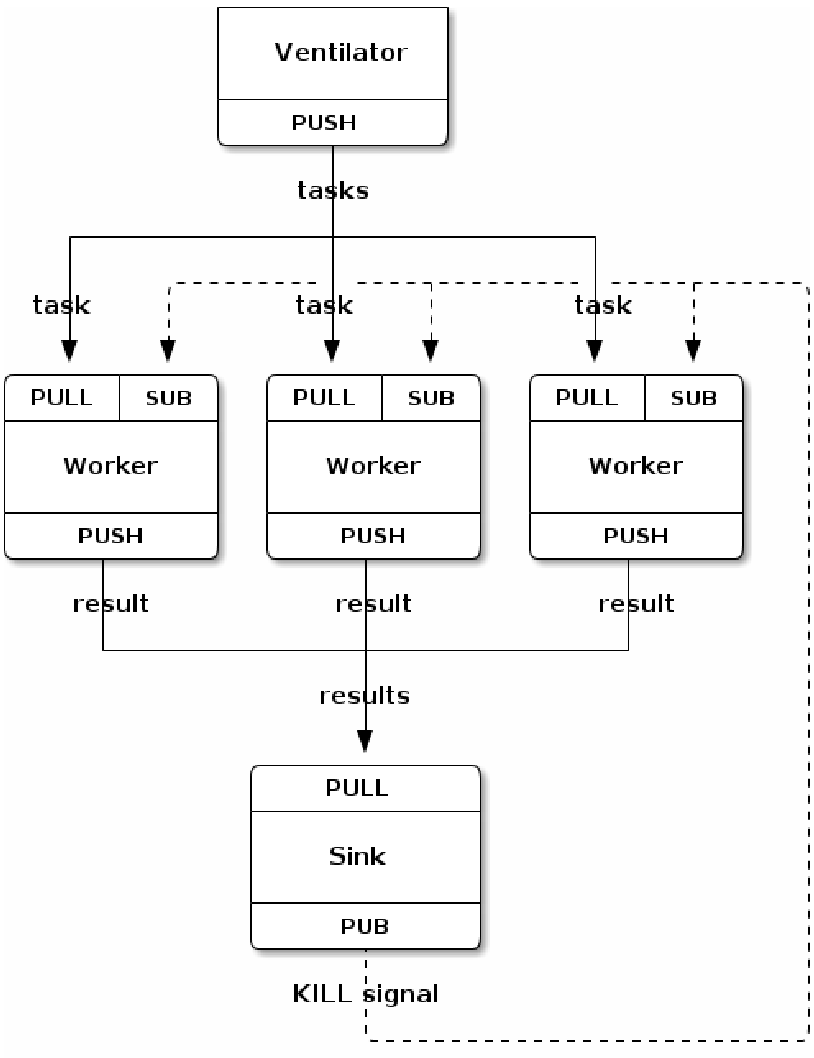


Figure 14 – Parallel Pipeline with Kill signaling

Here is the worker process, which manages two sockets (a PULL socket getting tasks, and a SUB socket getting control commands) using the zmq_poll(3) technique we saw earlier:

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

int main (void)
{
    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);

            //  Do the work
            s_sleep (atoi ((char *) zmq_msg_data (&message)));

            //  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;
}
```

*taskwork2.c: Parallel task worker with kill signaling*

Here is the modified sink application. When it's finished collecting results it broadcasts a KILL message to all workers:

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

int main (void)
{
    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
    int64_t start_time = s_clock ();

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

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

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

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

*tasksink2.c: Parallel task sink with kill signaling*

## Handling Interrupt Signals

top prev next

Realistic applications need to shutdown cleanly when interrupted with Ctrl-C or another signal such as SIGTERM. By default, these simply kill the process, meaning messages won't be flushed, files won't be closed cleanly, etc.

Here is how we handle a signal in various languages:

```
//
//  Shows how to handle Ctrl-C
//
#include <zmq.h>
#include <stdio.h>
#include <signal.h>
```

```
//  ---------------------------------------------------------------------
----
//  Signal handling
//
//  Call s_catch_signals() in your application at startup, and then
exit
//  your main loop if s_interrupted is ever 1. Works especially well
with
//  zmq_poll.

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

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

int main (void)
{
    void *context = zmq_init (1);
    void *socket = zmq_socket (context, ZMQ_REP);
    zmq_bind (socket, "tcp://*:5555");

    s_catch_signals ();
    while (1) {
        //  Blocking read will exit on a signal
        zmq_msg_t message;
        zmq_msg_init (&message);
        zmq_recv (socket, &message, 0);

        if (s_interrupted) {
            printf ("W: interrupt received, killing server…\n");
            break;
        }
    }
    zmq_close (socket);
    zmq_term (context);
    return 0;
}
```

*interrupt.c: Handling Ctrl-C cleanly*

Note that the zhelpers.h provides s_catch_signals() and s_interrupted. To use these correctly:

- Call s_catch_signals() at the start of your main code.
- Check s_interrupted after every blocking receive, or zmq_poll(3) call.
- Shutdown as normal if s_interrupted is set.

If you do not check s_interrupted then your application will become immune to Ctrl-C and SIGTERM, which may be useful but is usually not.

## Detecting Memory Leaks

top prev next

Any long-running application has to manage memory correctly, or eventually it'll use up all available memory and crash. If you use a language that handles this automatically for you, congratulations. If you program in C or C++ or any other language where you're responsible for memory management, here's a short tutorial on using valgrind, which among other things will report on any leaks your programs have.

- To install valgrind, e.g. on Ubuntu or Debian: `sudo apt-get install valgrind`.
- By default, ØMQ will cause valgrind to complain a lot. To remove these warnings, rebuild ØMQ with the ZMQ_MAKE_VALGRIND_HAPPY macro, thus:

```
$ cd zeromq2
$ export CPPFLAGS=-DZMQ_MAKE_VALGRIND_HAPPY
$ ./configure
$ make clean; make
$ sudo make install
```

- Fix your applications to exit cleanly after Ctrl-C. For any application that exits by itself, that's not needed, but for long-running applications (like devices), this is essential, otherwise valgrind will complain about all currently allocated memory.
- Build your application with -DDEBUG, if it's not your default setting. That ensures valgrind can tell you exactly where memory is being leaked.
- Finally, run valgrind thus:

```
valgrind --tool=memcheck --leak-check=full someprog
```

And after fixing any errors it reported, you should get the pleasant message:

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

## Multipart Messages

top prev next

ØMQ lets us compose a message out of several frames, giving us a 'multipart message'. Realistic applications use multipart messages heavily, especially to make "envelopes". We'll look at them later. What we'll learn now is simply how to safely (but blindly) read and write multipart messages because otherwise the devices we write won't work with applications that use multipart messages.

When you work with multipart messages, each part is a zmq_msg item. E.g. if you are sending a message with five parts, you must construct, send, and destroy five zmq_msg items. You can do this in advance (and store the zmq_msg items in an array or structure), or as you send them, one by one.

Here is how we send the frames in a multipart message (we receive each frame into a message object):

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

Here is how we receive and process all the parts in a message, be it single part or multipart:

```
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
}
```

Some things to know about multipart messages:

- When you send a multipart message, the first part (and all following parts) are only sent when you send the final part.
- If you are using zmq_poll(3), when you receive the first part of a message, all the rest have also arrived.
- You will receive all parts of a message, or none at all.
- Each part of a message is a separate zmq_msg item.
- You will receive all parts of a message whether or not you check the RCVMORE option.
- On sending, ØMQ queues message parts in memory until the last is received, then sends them all.
- There is no way to cancel a partially sent message, except by closing the socket.

## Intermediates and Devices

top prev next

Any connected set hits a complexity curve as the number of set members increases. A small number of members can all know about each other but as the set gets larger, the cost to each member of knowing all other interesting members grows linearly, and the overall cost of connecting members grows factorially. The solution is to break sets into smaller ones, and use intermediates to connect the sets.

This pattern is extremely common in the real world and is why our societies and economies are filled with intermediaries who have no other real function than to reduce the complexity and scaling costs of larger networks. Intermediaries are typically called wholesalers, distributors, managers, etc.

A ØMQ network like any cannot grow beyond a certain size without needing intermediaries. In ØMQ, we call these "devices". When we use ØMQ we usually start building our applications as a set of nodes on a network with the nodes talking to each other, without intermediaries:

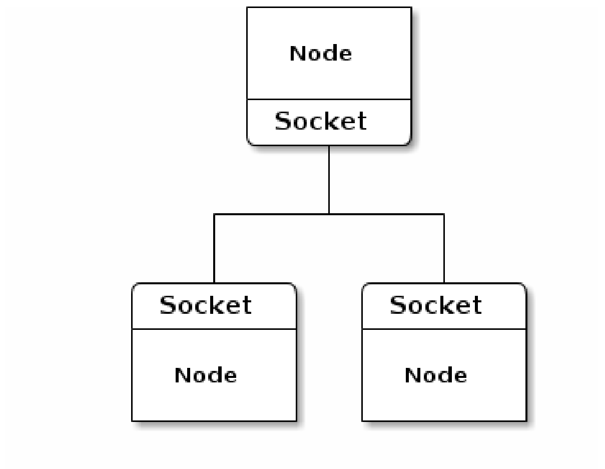


Figure 15 – Small scale ØMQ application

And then we extend the application across a wider network, placing devices in specific places and scaling up the number of nodes:

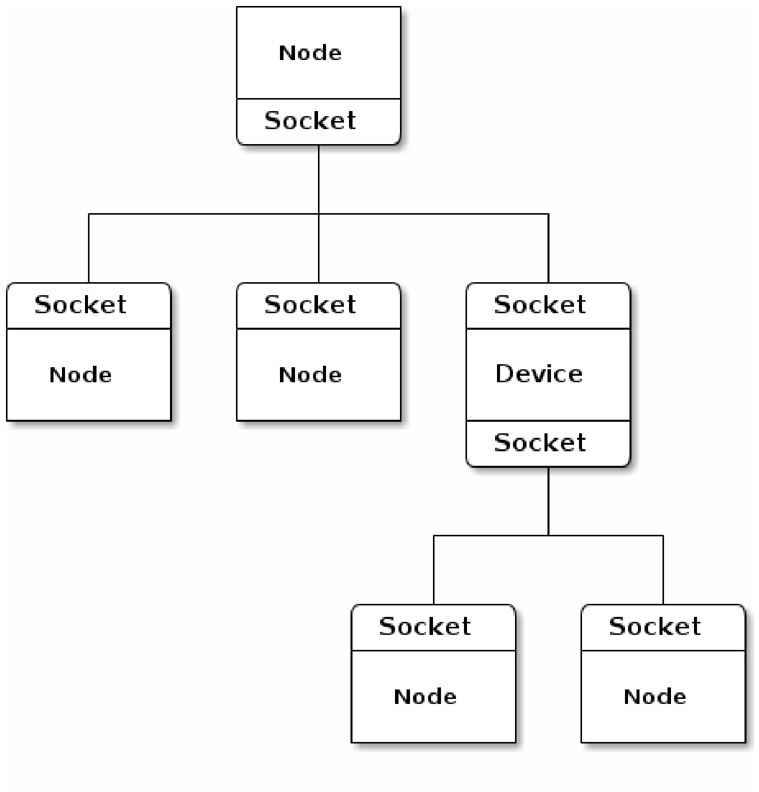


Figure 16 – Larger scale ØMQ application

ØMQ devices generally connect a set of 'frontend' sockets to a set of 'backend' sockets, though there are no strict design rules. They ideally run with no state, so that it becomes possible to stretch applications over as many intermediates as needed. You can run them as threads within a process, or as stand-alone processes. ØMQ provides some very basic devices but you will in practice develop your own.

ØMQ devices can do intermediation of addresses, services, queues, or any other abstraction you care to define above the message and socket layers. Different messaging patterns have different complexity issues and need different kinds of intermediation. For example, request-reply works well with queue and service abstractions, while publish-subscribe works well with streams or topics.

What's interesting about ØMQ as compared to traditional centralized brokers is that you can place devices precisely where you need them, and they can do the optimal intermediation.

## A Publish-Subscribe Proxy Server

top prev next

It is a common requirement to extend a publish-subscribe architecture over more than one network segment or transport. Perhaps there are a group of subscribers sitting at a remote location. Perhaps we want to publish to local subscribers via multicast, and to remote subscribers via TCP.

We're going to write a simple proxy server that sits in between a publisher and a set of subscribers, bridging two networks. This is perhaps the simplest case of a useful device. The device has two sockets, a frontend facing the internal network, where the weather server is sitting, and a backend facing subscribers on the external network. It subscribes to the weather service on the frontend socket, and republishes its data on the backend socket:

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

int main (void)
{
    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
        }
    }
    //  We don't actually get here but if we did, we'd shut down
neatly
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

*wuproxy.c: Weather update proxy*

We call this a *proxy* because it acts as a subscriber to publishers, and acts as a publisher to subscribers. That means you can slot this device into an existing network without affecting it (of course the new subscribers need to know to speak to the proxy).

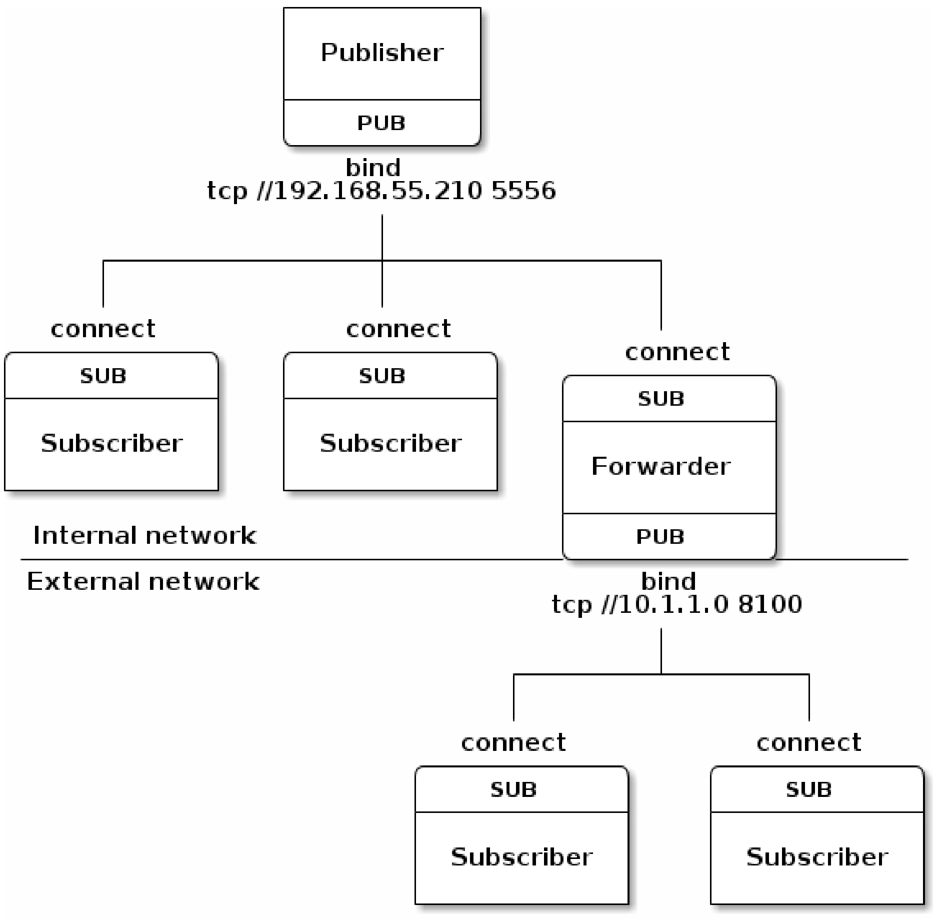


Figure 17 – Forwarder proxy device

Note that this application is multipart safe. It correctly detects multipart messages and sends them as it read them. If we did not set the SNDMORE option on outgoing multipart data, the final recipient would get a corrupted message. You should always make your devices multipart safe so that there is no risk they will corrupt the data they switch.

## A Request-Reply Broker

top prev next

Let's explore how to solve a problem of scale by writing a little message queuing broker in

ØMQ. We'll look at the request-reply pattern for this case.

In the Hello World client-server application we have one client that talks to one service. However in real cases we usually need to allow multiple services as well as multiple clients. This lets us scale up the power of the service (many threads or processes or boxes rather than just one). The only constraint is that services must be stateless, all state being in the request or in some shared storage such as a database.

There are two ways to connect multiple clients to multiple servers. The brute-force way is to connect each client socket to multiple service endpoints. One client socket can connect to multiple service sockets, and requests are load-balanced among these services. Let's say you connect a client socket to three service endpoints, A, B, and C. The client makes requests R1, R2, R3, R4. R1 and R4 go to service A, R2 goes to B, and R3 goes to service C.

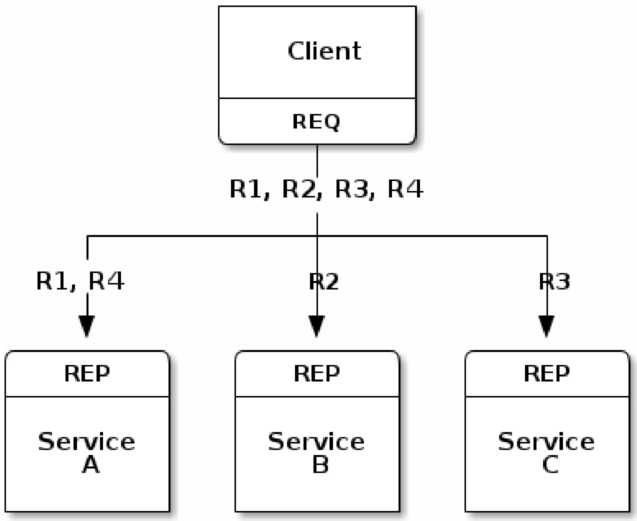


Figure 18 — Load balancing of requests

This design lets you add more clients cheaply. You can also add more services. Each client will load-balance its requests to the services. But each client has to know the service topology. If you have 100 clients and then you decide to add three more services, you need to reconfigure and restart 100 clients in order for the clients to know about the three new services.

That's clearly not the kind of thing we want to be doing at 3am when our supercomputing cluster has run out of resources and we desperately need to add a couple of hundred new service nodes. Too many stable pieces are like liquid concrete: knowledge is distributed and the more stable pieces you have, the more effort it is to change the topology. What we want is something sitting in between clients and services that centralizes all knowledge of the topology. Ideally, we should be able to add and remove services or clients at any time without touching any other part of the topology.

So we'll write a little message queuing broker that gives us this flexibility. The broker binds to two endpoints, a frontend for clients and a backend for services. It then uses zmq_poll(3) to monitor these two sockets for activity and when it has some, it shuttles messages between its two sockets. It doesn't actually manage any queues explicitly — ØMQ does that automatically on each socket.

When you use REQ to talk to REP you get a strictly synchronous request-reply dialog. The client sends a request, the service reads the request and sends a reply. The client then reads the reply. If either the client or the service try to do anything else (e.g. sending two requests in a row without waiting for a response) they will get an error.

But our broker has to be non-blocking. Obviously we can use zmq_poll(3) to wait for activity on either socket, but we can't use REP and REQ.

Luckily there are two sockets called DEALER and ROUTER that let you do non-blocking request-response. These sockets used to be called XREQ and XREP, and you may see these names in old code. The old names suggested that XREQ was an "extended REQ" and XREP was an "extended REP" but that's inaccurate. You'll see in Chapter Three how DEALER and ROUTER sockets let you build all kinds of asynchronous request-reply flows.

Now, we're just going to see how DEALER and ROUTER let us extend REQ-REP across a device, that is, our little broker.

In this simple stretched request-reply pattern, REQ talks to ROUTER and DEALER talks to REP. In between the DEALER and ROUTER we have to have code (like our broker) that pulls messages off the one socket and shoves them onto the other:

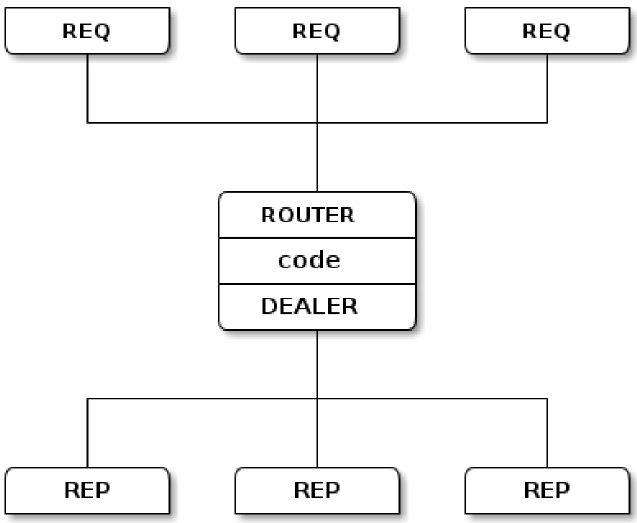


Figure 19 — Extended request-reply

The request-reply broker binds to two endpoints, one for clients to connect to (the frontend socket) and one for services to connect to (the backend). To test this broker, you will want to change your services so they connect to the backend socket. Here are a client and service that show what I mean:

```
//
//  Hello World client
//  Connects REQ socket to tcp://localhost:5559
//  Sends "Hello" to server, expects "World" back
//
#include "zhelpers.h"

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

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

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

*rrclient.c: Request-reply client*

Here is the service:

```
//
//  Hello World server
//  Connects REP socket to tcp://*:5560
//  Expects "Hello" from client, replies with "World"
//
#include "zhelpers.h"

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

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

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

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

        //  Send reply back to client
        s_send (responder, "World");
    }
    //  We never get here but clean up anyhow
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
```

*rrserver.c: Request-reply service*

And here is the broker. You will see that it's multipart safe:

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

int main (void)
{
    //  Prepare our context and sockets
    void *context = zmq_init (1);
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (frontend, "tcp://*:5559");
    zmq_bind (backend,  "tcp://*:5560");

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

*rrbroker.c: Request-reply broker*

Using a request-reply broker makes your client-server architectures easier to scale since clients don't see services, and services don't see clients. The only stable node is the device in the middle:

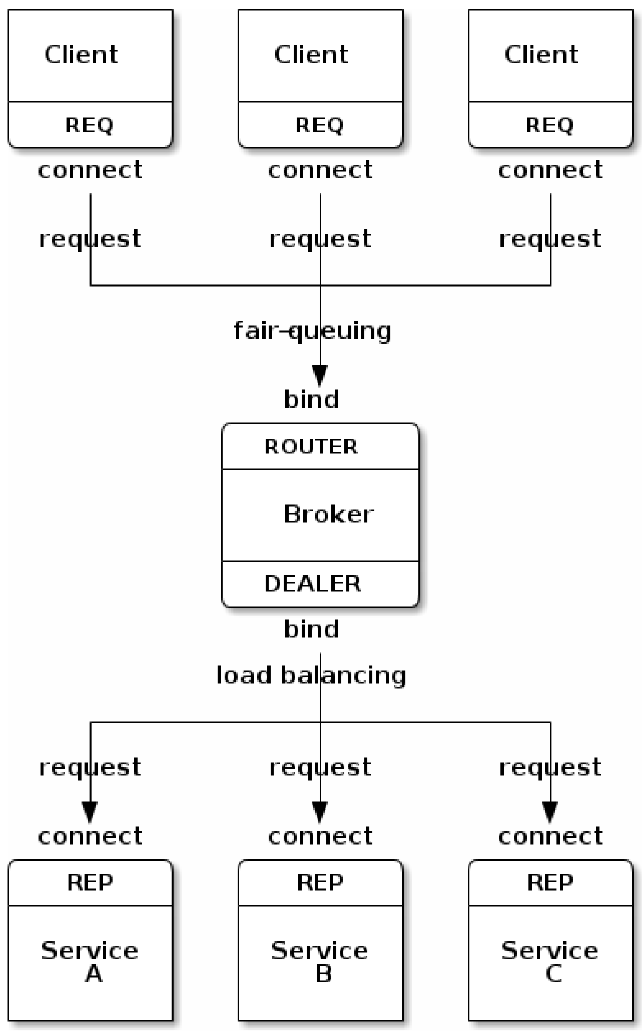


Figure 20 — Request reply broker

## Built-in Devices

top prev next

ØMQ provides some built-in devices, though most advanced users write their own devices. The built-in devices are:

- QUEUE, which is like the request-reply broker.
- FORWARDER, which is like the pub-sub proxy server.
- STREAMER, which is like FORWARDER but for pipeline flows.

To start a device, you call zmq_device(3) and pass it two sockets, one for the frontend and one for the backend:

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

Which if you start a QUEUE device is exactly like plugging the main body of the request-reply broker into your code at that spot. You need to create the sockets, bind or connect them, and possibly configure them, before calling zmq_device(3). It is trivial to do. Here is the request-reply broker re-written to call QUEUE and rebadged as an expensive-sounding "message queue" (people have charged houses for code that did less):

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

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

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

    //  Socket facing services
    void *backend = zmq_socket (context, ZMQ_DEALER);
    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;
}
```

*msgqueue.c: Message queue broker*

The built-in devices do proper error handling, whereas the examples we have shown don't. Since you can configure the sockets as you need to, before starting the device, it's worth using the built-in devices when you can.

If you're like most ØMQ users, at this stage your mind is starting to think, "*what kind of evil stuff can I do if I plug random socket types into devices?*" The short answer is: don't do it. You can mix socket types but the results are going to be weird. So stick to using ROUTER/DEALER for queue devices, SUB/PUB for forwarders and PULL/PUSH for streamers.

When you start to need other combinations, it's time to write your own devices.

## Multithreading with ØMQ

top prev next

ØMQ is perhaps the nicest way ever to write multithreaded (MT) applications. Whereas as ØMQ sockets require some readjustment if you are used to traditional sockets, ØMQ multithreading will take everything you know about writing MT applications, throw it into a heap in the garden, pour gasoline over it, and set it alite. It's a rare book that deserves burning, but most books on concurrent programming do.

To make utterly perfect MT programs (and I mean that literally) **we don't need mutexes, locks, or any other form of inter-thread communication except messages sent across ØMQ sockets.**

By "perfect" MT programs I mean code that's easy to write and understand, that works with one technology in any language and on any operating system, and that scales across any number of CPUs with zero wait states and no point of diminishing returns.

If you've spent years learning tricks to make your MT code work at all, let alone rapidly, with locks and semaphores and critical sections, you will be disgusted when you realize it was all for nothing. If there's one lesson we've learned from 30+ years of concurrent programming it is: *just don't share state*. It's like two drunkards trying to share a beer. It doesn't matter if they're good buddies. Sooner or later they're going to get into a fight. And the more drunkards you add to the pavement, the more they fight each other over the beer. The tragic majority of MT applications look like drunken bar fights.

The list of weird problems that you need to fight as you write classic shared-state MT code would be hillarious if it didn't translate directly into stress and risk, as code that seems to work suddenly fails under pressure. Here is a list of "*11 Likely Problems In Your Multithreaded Code*" from a large firm with world-beating experience in buggy code: forgotten synchronization, incorrect granularity, read and write tearing, lock-free reordering, lock convoys, two-step dance, and priority inversion.

Yeah, we also counted seven, not eleven. That's not the point though. The point is, do you really want that code running the power grid or stock market to start getting two-step lock convoys at 3pm on a busy Thursday? Who cares what the terms actually mean. This is not what turned us on to programming, fighting ever more complex side-effects with ever more complex hacks.

Some widely used metaphors, despite being the basis for billion-dollar industries, are fundamentally broken, and shared state concurrency is one of them. Code that wants to scale without limit does it like the Internet does, by sending messages and sharing nothing except a common contempt for broken programming metaphors.

You should follow some rules to write happy multithreaded code with ØMQ:

- You MUST NOT access the same data from multiple threads. Using classic MT techniques like mutexes are an anti-pattern in ØMQ applications. The only exception to this is a ØMQ context object, which is threadsafe.
- You MUST create a ØMQ context for your process, and pass that to all threads that you want to connect via `inproc` sockets.
- You MAY treat threads as separate tasks, with their own context, but these threads cannot communicate over `inproc`. However they will be easier to break into standalone processes afterwards.
- You MUST NOT share ØMQ sockets between threads. ØMQ sockets are not threadsafe. Technically it's possible to do this, but it demands semaphores, locks, or mutexes. This will make your application slow and fragile. The only place where it's remotely sane to share sockets between threads are in language bindings that need to do magic like garbage collection on sockets.

If you need to start more than one device in an application, for example, you will want to run each in their own thread. It is easy to make the error of creating the device sockets in one thread, and then passing the sockets to the device in another thread. This may appear to work but will fail randomly. Remember: *Do not use or close sockets except in the thread that created them.*

If you follow these rules, you can quite easily split threads into separate processes, when you need to. Application logic can sit in threads, processes, boxes: whatever your scale needs.

ØMQ uses native OS threads rather than virtual "green" threads. The advantage is that you don't need to learn any new threading API, and that ØMQ threads map cleanly to your operating system. You can use standard tools like Intel's ThreadChecker to see what your application is doing. The disadvantages are that your code, when it for instance starts new threads, won't be portable, and that if you have a huge number of threads (thousands), some operating systems will get stressed.

Let's see how this works in practice. We'll turn our old Hello World server into something more capable. The original server was a single thread. If the work per request is low, that's fine: one ØMQ thread can run at full speed on a CPU core, with no waits, doing an awful lot of work. But realistic servers have to do non-trivial work per request. A single core may not be enough when 10,000 clients hit the server all at once. So a realistic server must starts multiple worker threads. It then accepts requests as fast as it can, and distributes these to its worker threads. The worker threads grind through the work, and eventually send their replies back.

You can of course do all this using a queue device and external worker processes, but often it's easier to start one process that gobbles up sixteen cores, than sixteen processes, each gobbling up one core. Further, running workers as threads will cut out a network hop, latency, and network traffic.

The MT version of the Hello World service basically collapses the queue device and workers into a single process:

```
//
//  Multithreaded Hello World server
//
#include "zhelpers.h"
#include <pthread.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");
    }
    zmq_close (receiver);
    return NULL;
}

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

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

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

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

*mtserver.c: Multithreaded service*

All the code should be recognizable to you by now. How it works:

- The server starts a set of worker threads. Each worker thread creates a REP socket and then processes requests on this socket. Worker threads are just like single-threaded servers. The only differences are the transport (`inproc` instead of `tcp`), and the bind-connect direction.
- The server creates a ROUTER socket to talk to clients and binds this to its external interface (over `tcp`).
- The server creates a DEALER socket to talk to the workers and binds this to its internal interface (over `inproc`).
- The server starts a QUEUE device that connects the two sockets. The QUEUE device keeps a single queue for incoming requests, and distributes those out to workers. It also routes replies back to their origin.

Note that creating threads is not portable in most programming languages. The POSIX library is `pthreads`, but on Windows you have to use a different API. We'll see in Chapter Three how to wrap this in a portable API.

Here the 'work' is just a one-second pause. We could do anything in the workers, including talking to other nodes. This is what the MT server looks like in terms of ØMQ sockets and nodes. Note how the request-reply chain is `REQ-ROUTER-queue-DEALER-REP`:

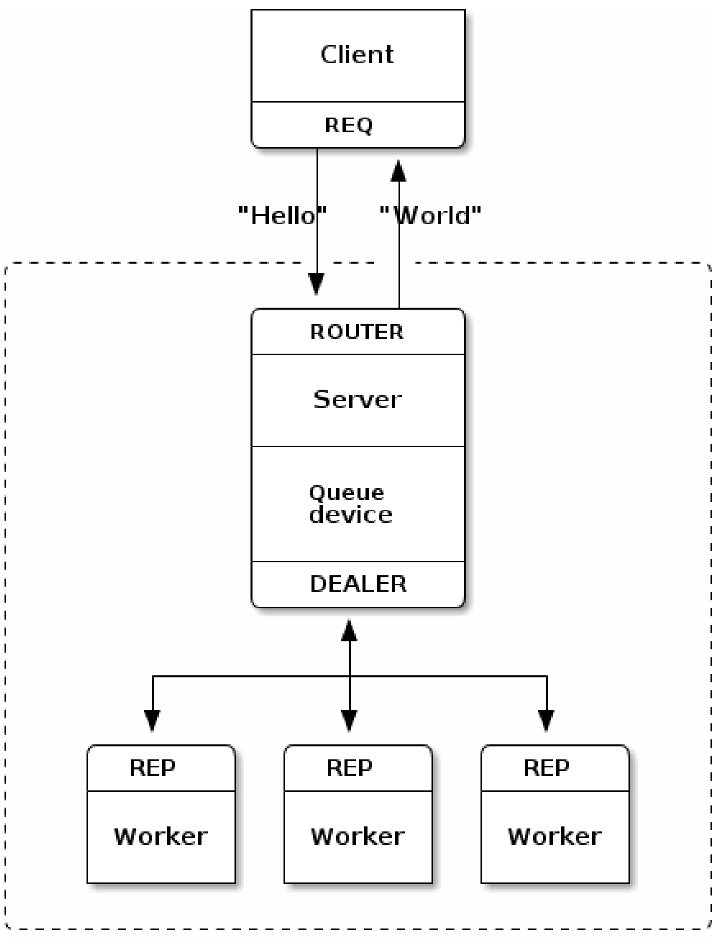


Figure 21 – Multithreaded server

## Signaling between Threads

top prev next

When you start making multithreaded applications with ØMQ, you'll hit the question of how to coordinate your threads. Though you might be tempted to insert 'sleep' statements, or use multithreading techniques such as semaphores or mutexes, **the only mechanism that you should use are ØMQ messages**. Remember the story of The Drunkards and the Beer Bottle.

Here is a simple example showing three threads that signal each other when they are ready.

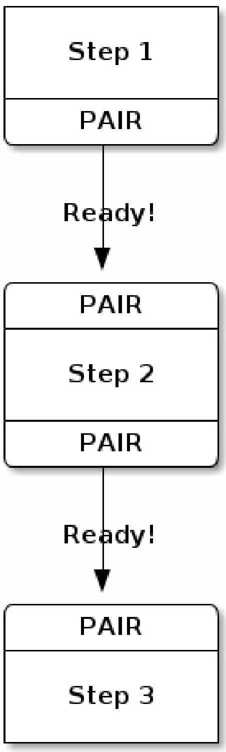


Figure 22 – The Relay Race

In this example we use PAIR sockets over the `inproc` transport:

```
//
//  Multithreaded relay
//
#include "zhelpers.h"
#include <pthread.h>

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

    return NULL;
}

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

    //  Wait for signal and pass it on
    char *string = s_recv (receiver);
    free (string);
    zmq_close (receiver);

    //  Connect to step3 and tell it we're ready
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step3");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}

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

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

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

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

*mtrelay.c: Multithreaded relay*

This is a classic pattern for multithreading with ØMQ:

1. Two threads communicate over `inproc`, using a shared context.
2. The parent thread creates one socket, binds it to an inproc:// endpoint, and *then* starts the child thread, passing the context to it.
3. The child thread creates the second socket, connects it to that inproc:// endpoint, and *then* signals to the parent thread that it's ready.

Note that multithreading code using this pattern is ***not scalable out to processes***. If you use `inproc` and socket pairs, you are building a tightly-bound application. Do this when low latency is really vital. For all normal apps, use one context per thread, and `ipc` or `tcp`. Then you can easily break your threads out to separate processes, or boxes, as needed.

This is the first time we've shown an example using PAIR sockets. Why use PAIR? Other socket combinations might seem to work but they all have side-effects that could interfere with signaling:

- You can use PUSH for the sender and PULL for the receiver. This looks simple and will work, but remember that PUSH will load-balance messages to all available receivers. If you by accident start two receivers (e.g. you already have one running and you start a second), you'll "lose" half of your signals. PAIR has the advantage of refusing more than one connection, the pair is *exclusive*.

- You can use DEALER for the sender and ROUTER for the receiver. ROUTER however wraps your message in an "envelope", meaning your zero-size signal turns into a multipart message. If you don't care about the data, and treat anything as a valid signal, and if you don't read more than once from the socket, that won't matter. If however you decide to send real data, you will suddenly find ROUTER providing you with "wrong" messages. DEALER also load-balances, giving the same risk as PUSH.

- You can use PUB for the sender and SUB for the receiver. This will correctly deliver you messages exactly as you sent them and PUB does not load-balance as PUSH or DEALER do. However you need to configure the subscriber with an empty subscription, which is annoying. Worse, the reliability of the PUB-SUB link is timing dependent and messages can get lost if the SUB socket is connecting while the PUB socket is sending its message.

For these reasons, PAIR makes the best choice for coordination between pairs of threads.

## Node Coordination

top prev next

When you want to coordinate nodes, PAIR sockets won't work well any more. This is one of the few areas where the strategies for threads and nodes are different. Principally nodes come and go whereas threads are stable. PAIR sockets do not automatically reconnect if the remote node goes away and comes back.

The second significant difference between threads and nodes is that you typically have a fixed number of threads but a more variable number of nodes. Let's take one of our earlier scenarios (the weather server and clients) and use node coordination to ensure that subscribers don't lose data when starting up.

This is how the application will work:

- The publisher knows in advance how many subscribers it expects. This is just a magic number it gets from somewhere.
- The publisher starts up and waits for all subscribers to connect. This is the node coordination part. Each subscriber subscribes and then tells the publisher it's ready via another socket.
- When the publisher has all subscribers connected, it starts to publish data.

In this case we'll use a REQ-REP socket flow to synchronize subscribers and publisher. Here is the publisher:

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

//  We wait for 10 subscribers
#define SUBSCRIBERS_EXPECTED  10

int main (void)
{
    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");

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

*syncpub.c: Synchronized publisher*

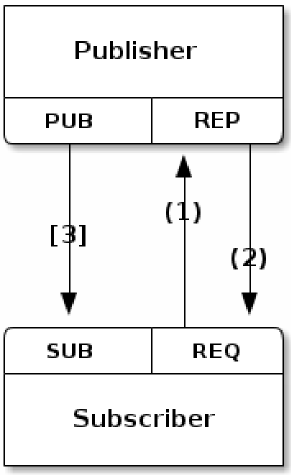


Figure 23 — Pub Sub Synchronization

And here is the subscriber:

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

int main (void)
{
    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);

    //  0MQ is so fast, we need to wait a while…
    sleep (1);

    //  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;
}
```

*syncsub.c: Synchronized subscriber*

This Linux shell script will start ten subscribers and then the publisher:

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

Which gives us this satisfying output:

```
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
```

We can't assume that the SUB connect will be finished by the time the REQ/REP dialog is complete. There are no guarantees that outbound connects will finish in any order whatsoever, if you're using any transport except `inproc`. So, the example does a brute-force sleep of one second between subscribing, and sending the REQ/REP synchronization.

A more robust model could be:

- Publisher opens PUB socket and starts sending "Hello" messages (not data).
- Subscribers connect SUB socket and when they receive a Hello message they tell the publisher via a REQ/REP socket pair.
- When the publisher has had all the necessary confirmations, it starts to send real data.

## Zero Copy

top prev next

We teased you in Chapter One, when you were still a ØMQ newbie, about zero-copy. If you survived this far, you are probably ready to use zero-copy. However, remember that there are many roads to Hell, and premature optimization is not the most enjoyable nor profitable one, by far. In English, trying to do zero-copy properly while your architecture is not perfect is a waste of time and will make things worse, not better.

ØMQ's message API lets you can send and receive messages directly from and to application buffers without copying data. Given that ØMQ sends messages in the background, zero-copy needs some extra sauce.

To do zero-copy you use zmq_msg_init_data(3) to create a message that refers to a block of data already allocated on the heap with malloc(), and then you pass that to zmq_send(3). When you create the message you also pass a function that ØMQ will call to free the block of data, when it has finished sending the message. This is the simplest example, assuming 'buffer' is a block of 1000 bytes allocated on the heap:

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

There is no way to do zero-copy on receive: ØMQ delivers you a buffer that you can store as long as you wish but it will not write data directly into application buffers.

On writing, ØMQ's multipart messages work nicely together with zero-copy. In traditional messaging you need to marshal different buffers together into one buffer that you can send. That means copying data. With ØMQ, you can send multiple buffers coming from different sources as individual message parts. We send each field as a length-delimited frame. To the application it looks like a series of send and recv calls. But internally the multiple parts get written to the network and read back with single system calls, so it's very efficient.

## Transient vs. Durable Sockets

top prev next

In classic networking, sockets are API objects, and their lifespan is never longer than the code that uses them. But if you look at a socket you see that it collects a bunch of resources - network buffers - and at some stage, a ØMQ user asked, "isn't there some way these could hang around if my program crashes, so I can get them back?"

This turns out to be very useful. It's not foolproof, but it gives ØMQ a kind of "better than a kick in the nuts" reliability, particularly useful for pub-sub cases. We'll look at that shortly.

Here is the general model of two sockets happily chatting about the weather, and who kissed who and where and when precisely, cause I heard something different, at the last staff party, not to mention did you see that new family up the road who do they think they are with that car and what's with the prices at the shops these days don't they know it's a crisis?

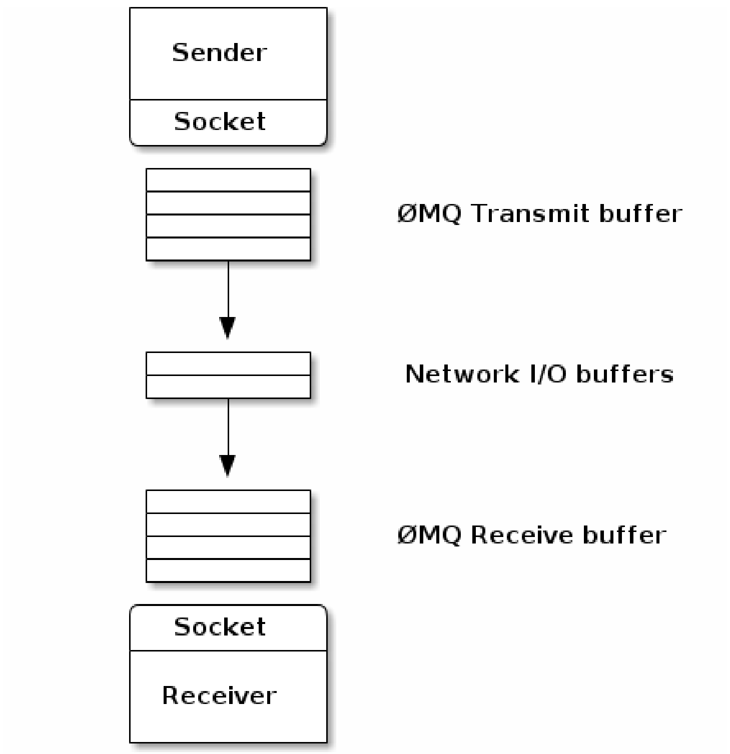


Figure 24 – Sender boring the pants off receiver

If a *receiver* (SUB, PULL, REQ) side of a socket sets an identity, then the *sending* (PUB, PUSH, PULL) side will buffer messages when they aren't connected up to the HWM. The sending side does not need to set an identity for this to work.

Note that ØMQ's transmit and receive buffers are invisible and automatic, just like TCP's buffers are.

All the sockets we've used so far were transient. To turn a transient socket into a durable one you give it an explicit *identity*. All ØMQ sockets have identities but by default they are generated 'unique universal identifiers' (UUIDs) that the peer uses to recall who it's talking to.

Behind the scenes, and invisibly to you, when one socket connects to another, the two sockets exchange identities. Normally sockets don't tell their peers their identity, so peers invent random identities for each other:

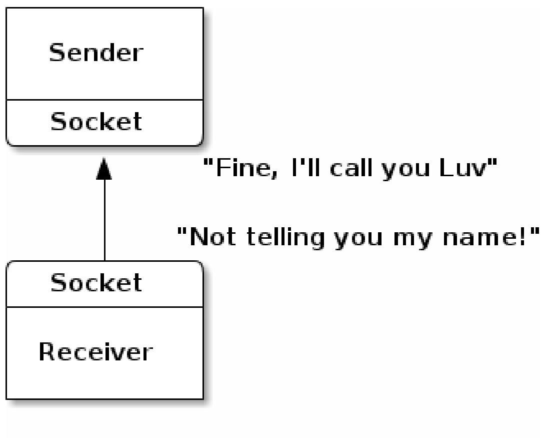


Figure 25 — Transient socket

But a socket can also tell the other its identity, and then the next time the two meet, it'll be "so as I was saying what I heard was quite different but anyhow you know how it goes at the office, they're all tattletales I'd never say anything about anyone that wasn't true or at least based on a sure thing".

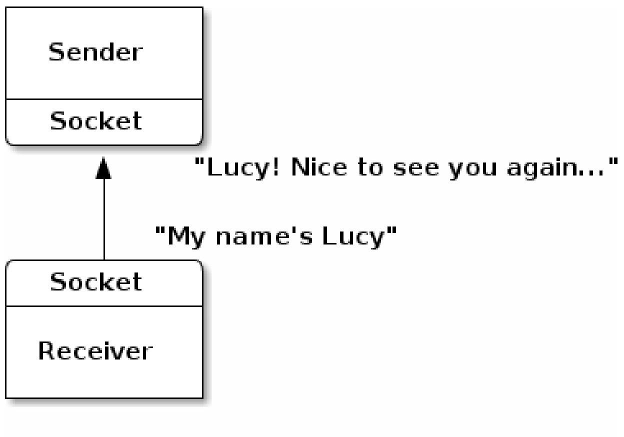


Figure 26 — Durable socket

Here's how you set the identity of a socket, to create a durable socket:

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

Some comments on setting a socket identity:

- If you want to set an identity you must do it *before* connecting or binding the socket.
- It's the receiver that sets an identity: it's kind of like a session cookie in an HTTP web application, except the client/sender is picking the cookie it will use.
- Identities are binary strings: identities starting with a zero byte are reserved for ØMQ use.
- Do not use the same identity for more than one socket. Any socket trying to connect using an identity already taken by another socket will just be disconnected.
- Do not use random identities in applications that create lots of sockets. What this will do is cause lots and lots of durable sockets to pile up, eventually crashing nodes.
- If you need to know the identity of the *peer* you got a message from, only the

ROUTER socket does this for you automatically. For any other socket type you must send the address explicitly, as a message part.

- Having said all this, using durable sockets is often a bad idea. It makes senders accumulate entropy, which makes architectures fragile. If we were making ØMQ again we'd probably not implement explicit identities at all.

See zmq_setsockopt(3) for a summary of the ZMQ_IDENTITY socket option. Note that the zmq_getsockopt(3) method gives you the identity of the socket you are working with, *not* any peer it might be connected to.

## Pub-sub Message Envelopes

top prev next

We've looked briefly at multipart messages. Let's now look at their main use-case, which is *message envelopes*. An envelope is a way of safely packaging up data with an address, without touching the data itself.

In the pub-sub pattern, the envelope at least holds the subscription key for filtering but you can also add the sender identity in the envelope.

If you want to use pub-sub envelopes, you make them yourself. It's optional, and in previous pub-sub examples we didn't do this. Using a pub-sub envelope is a little more work for simple cases but it's cleaner especially for real cases, where the key and the data are naturally separate things. It's also faster, if you are writing the data directly from an application buffer.

Here is what a publish-subscribe message with an envelope looks like:

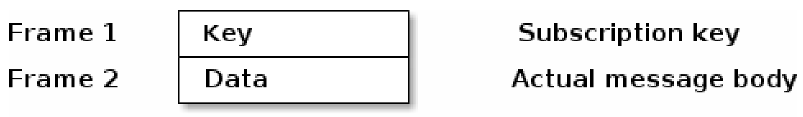


Figure 27 – Pub sub envelope with separate key

Recall that pub-sub matches messages based on the prefix. Putting the key into a separate frame makes the matching very obvious, since there is no chance an application will accidentally match on part of the data.

Here is a minimalist example of how pub-sub envelopes look in code. This publisher sends messages of two types, A and B. The envelope holds the message type:

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

int main (void)
{
    //  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;
}
```

*psenvpub.c: Pub-sub envelope publisher*

The subscriber only wants messages of type B:

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

int main (void)
{
    //  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;
}
```

*psenvsub.c: Pub-sub envelope subscriber*

When you run the two programs, the subscriber should show you this:

```
[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
...
```

This examples shows that the subscription filter rejects or accepts the entire multipart message (key plus data). You won't get part of a multipart message, ever.

If you subscribe to multiple publishers and you want to know their identity so that you can send them data via another socket (and this is a fairly typical use-case), you create a three-part message:

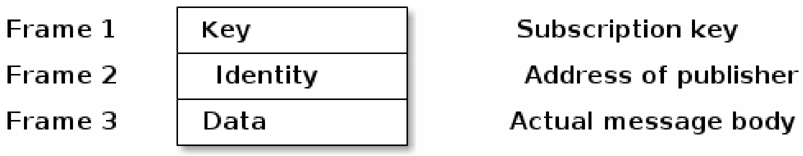


Figure 28 — Pub sub envelope with sender address

## (Semi-)Durable Subscribers and High-Water Marks

top prev next

Identities work on all socket types. If you have a PUB and a SUB socket, and the subscriber gives the publisher its identity, the publisher holds onto data until it can deliver it to the subscriber.

This is both wonderful and terrible at the same time. It's wonderful because it means updates can wait for you in the publisher's transmit buffer, until you connect and collect them. It's terrible because by default this will rapidly kill a publisher and lock up your system.

**If you use durable subscriber sockets (i.e. if you set the identity on a SUB socket) you *must* also guard against queue explosion by using the *high-water mark* or HWM, on the publisher socket.** The publisher's HWM affects all subscribers, independently.

If you want to prove this, take the wuclient and wuserver from Chapter One, and add this line to the wuclient before it connects:

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

Build and run the two programs. It all looks normal. But keep an eye on the memory used by the publisher, and you'll see that as the subscriber finishes, the publisher memory grows and grows. If you restart the subscriber, the publisher queues stop growing. As soon as the subscriber goes away, they grow again. It'll rapidly overwhelm your system.

We'll first look at how to do this, and then at how to do it properly. Here are a publisher and subscriber that use the 'node coordination' technique from Chapter Two to synchronize. The publisher then sends ten messages, waiting a second between each one. That wait is for you to kill the subscriber using Ctrl-C, wait a few seconds, and restart it.

Here's the publisher:

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

int main (void)
{
    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");

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

*durapub.c: Durable publisher*

And here's the subscriber:

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

int main (void)
{
    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, exit when told to do so
    while (1) {
        char *string = s_recv (subscriber);
        printf ("%s\n", string);
        if (strcmp (string, "END") == 0) {
            free (string);
            break;
        }
        free (string);
    }
    zmq_close (sync);
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

*durasub.c: Durable subscriber*

To run this, start the publisher, then the subscriber, each in their own window. Allow the subscriber to collect one or two messages, then Ctrl-C it. Count to three, and restart it. What you will see is something like this:

```
$ 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
```

Just to see the difference, comment out the line in the subscriber that sets the socket identity, and try again. You will see that it loses messages. Setting an identity turns a transient subscriber into a durable subscriber. You would in practice want to choose identities carefully, either taking them from configuration files, or generating UUIDs and storing them somewhere.

When we set a high-water mark on the PUB socket, the publisher stores that many messages, but no more. Let's test this by setting the publisher HWM to 2 messages, before we start publishing to the socket:

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

Now running our test, killing and restarting the subscriber after a couple of seconds' pause will show something like this:

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

Look carefully: we have two messages kept for us (2 and 3), then a gap of several messages, and then new updates again. The HWM causes ØMQ to drop messages it can't put onto the queue, something the ØMQ Reference Manual calls an "exceptional condition".

In short, if you use subscriber identities, you must set the high-water mark on publisher sockets, or else you risk servers that run out of memory and crash. However, there is a way out. ØMQ provides something called a "swap", which is a disk file that holds messages we can't store to the queue. It is very simple to enable:

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

We can put this together to make a cynical publisher that is immune to slow, blocked, or absent subscribers while still offering durable subscriptions to those that need it:

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

int main (void)
{
    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");

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

*durapub2.c: Durable but cynical publisher*

In practice, setting the HWM to 1 and shoving everything to disk will make a pub-sub system very slow. Here is a more reasonable 'best practice' for publishers that have to deal with unknown subscribers:

- **Always set a HWM on the PUB socket**, based on the expected maximum number of subscribers, the amount of memory you are willing to dedicated to queuing, and the average size of a message. For example if you expect up to 5,000 subscribers, and have 1GB of memory to play with, and messages of ~200 bytes, then a safe HWM would be (1,000,000,000 / 200 / 5,000) = 1,000.
- If you don't want slow or crashing subscribers to lose data, set a SWAP that's large enough to handle the peaks, based on the number of subscribers, peak message rate, average size of messages, and time you want to cover. For example with 5,000 subscribers and messages of ~200 bytes coming in at 100,000 per second, you will need up to 100MB of disk space per second. To cover an outage of up to 1 minute, therefore, you'd need 6GB of disk space, and it would have to be fast, but that's a different story.

Some notes on durable subscribers:

- Depending on how the subscriber dies, and the frequency of updates, and the size of network buffers, and the transport protocol you are using, data may be lost. Durable subscribers will have *much better* reliability than transient ones, but they will not be perfect.
- The SWAP file is not recoverable, so if a publisher dies and restarts, it will lose data that was in its transmit buffers, and that was in the network I/O buffers.

Some notes on using the HWM option:

- This affects both the transmit and receive buffers of a single socket. Some sockets (PUB, PUSH) only have transmit buffers. Some (SUB, PULL, REQ, REP) only have receive buffers. Some (DEALER, ROUTER, PAIR) have both transmit and receive buffers.
- When your socket reaches its high-water mark, it will either block or drop data depending on the socket type. PUB sockets will drop data if they reach their high-water mark, while other socket types will block.
- Over the `inproc` transport, the sender and reciever share the same buffers, so the real HWM is the sum of the HWM set by both sides. This means in effect that if one side does not set a HWM, there is no limit to the buffer size.

## A Bare Necessity

top prev next

ØMQ is like a box of pieces that plug together, the only limitation being your imagination and sobriety.

The scalable elastic architecture you get should be an eye-opener. You might need a coffee or two first. Don't make the mistake I made once and buy exotic German coffee labeled *Entkoffeiniert*. That does not mean "Delicious". Scalable elastic architectures are not a new idea - flow-based programming and languages like Erlang already worked like this - but ØMQ makes it easier to use than ever before.

As Gonzo Diethelm said, '*My gut feeling is summarized in this sentence: "if ØMQ didn't* *exist, it would be necessary to invent it". Meaning that I ran into ØMQ after years of brain-background processing, and it made instant sense... ØMQ simply seems to me a "bare necessity" nowadays.'*

# Chapter Three - Advanced Request-Reply Patterns

In Chapter Two we worked through the basics of using ØMQ by developing a series of small applications, each time exploring new aspects of ØMQ. We'll continue this approach in this chapter, as we explore advanced patterns built on top of ØMQ's core request-reply pattern.

We'll cover:

- How to create and use message envelopes for request-reply.
- How to use the REQ, REP, DEALER, and ROUTER sockets.
- How to set manual reply addresses using identities.
- How to do custom random scatter routing.
- How to do custom least-recently used routing.
- How to build a higher-level message class.
- How to build a basic request-reply broker.
- How to choose good names for sockets.
- How to simulate a cluster of clients and workers.
- How to build a scalable cloud of request-reply clusters.
- How to use pipeline sockets for monitoring threads.

## Request-Reply Envelopes

In the request-reply pattern, the envelope holds the return address for replies. It is how a ØMQ network with no state can create round-trip request-reply dialogs.

You don't in fact need to understand how request-reply envelopes work to use them for common cases. When you use REQ and REP, your sockets build and use envelopes automatically. When you write a device, and we covered this in the last chapter, you just need to read and write all the parts of a message. ØMQ implements envelopes using multipart data, so if you copy multipart data safely, you implicitly copy envelopes too.

However, getting under the hood and playing with request-reply envelopes is necessary for advanced request-reply work. It's time to explain how ROUTER works, in terms of envelopes:

- When you receive a message from a ROUTER socket, it shoves a brown paper envelope around the message and scribbles on with indelible ink, "This came from Lucy". Then it gives that to you. That is, the ROUTER socket gives you what came off the wire, wrapped up in an envelope with the reply address on it.

- when you send a message to a ROUTER socket, it rips off that brown paper envelope, tries to read its own handwriting, and if it knows who "Lucy" is, sends the contents back to Lucy. That is the reverse process of receiving a message.

If you leave the brown envelope alone, and then pass that message to another ROUTER socket (e.g. by sending to a DEALER connected to a ROUTER), the second ROUTER socket will in turn stick another brown envelope on it, and scribble the name of that DEALER on it.

The whole point of this is that each ROUTER knows how to send replies back to the right place. All you need to do, in your application, is respect the brown envelopes. Now the REP socket makes sense. It carefully slices open the brown envelopes, one by one, keeps them safely aside, and gives you (the application code that owns the REP socket) the original message. When you send the reply, it re-wraps the reply in the brown paper envelopes, so it can hand the resulting brown package back to the ROUTER sockets down the chain.

Which lets you insert ROUTER-DEALER devices into a request-reply pattern like this:

```
[REQ] <--> [REP]
[REQ] <--> [ROUTER--DEALER] <--> [REP]
[REQ] <--> [ROUTER--DEALER] <--> [ROUTER--DEALER] <--> [REP]
...etc.
```

If you connect a REQ socket to a ROUTER socket, and send one request message, this is what you get when you receive from the ROUTER socket:

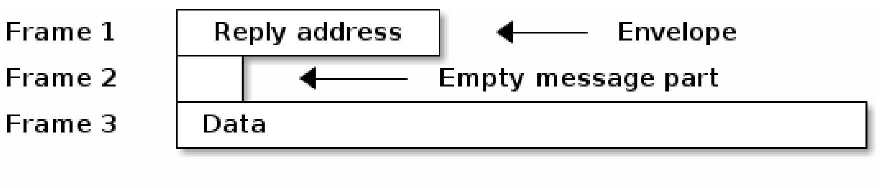


Figure 29 – Single hop request-reply envelope

Breaking this down:

- The data in frame 3 is what the sending application sends to the REQ socket.
- The empty message part in frame 2 is prepended by the REQ socket when it sends the message to the ROUTER socket.
- The reply address in frame 1 is prepended by the ROUTER before it passes the message to the receiving application.

Now if we extend this with a chain of devices, we get envelope on envelope, with the newest envelope always stuck at the beginning of the stack:

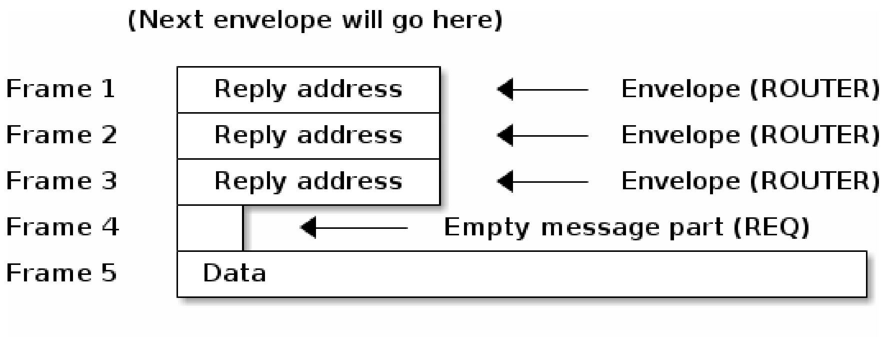


Figure 30 – Multihop request-reply envelope

Here now is a more detailed explanation of the four socket types we use for request-reply patterns:

- DEALER just load-balances (deals out) the messages you send to all connected peers, and fair-queues (deals in) the messages it receives. It is exactly like a PUSH and PULL socket combined.
- REQ prepends an empty message part to every message you send, and removes the

empty message part from each message you receive. It then works like DEALER (and in fact is built on DEALER) except it also imposes a strict send / receive cycle.

- ROUTER prepends an envelope with reply address to each message it receives, before passing it to the application. It also chops off the envelope (the first message part) from each message it sends, and uses that reply address to decide which peer the message should go to.
- REP stores all the message parts up to the first empty message part, when you receive a message and it passes the rest (the data) to your application. When you send a reply, REP prepends the saved envelopes to the message and sends it back using the same semantics as ROUTER (and in fact REP is built on top of ROUTER), but matching REQ, imposes a strict receive / send cycle.

REP requires that the envelopes end with an empty message part. If you're not using REQ at the other end of the chain then you must add the empty message part yourself.

So the obvious question about ROUTER is, where does it get the reply addresses from? And the obvious answer is, it uses the socket's identity. As we already learned, a socket can be transient in which case the *other* socket (ROUTER in this case) generates an identity that it can associate with the socket. Or, the socket can be durable in which case it explicitly tells the other socket (ROUTER, again) its identity and ROUTER can use that rather than generating a temporary label.

This is what it looks like for transient sockets:

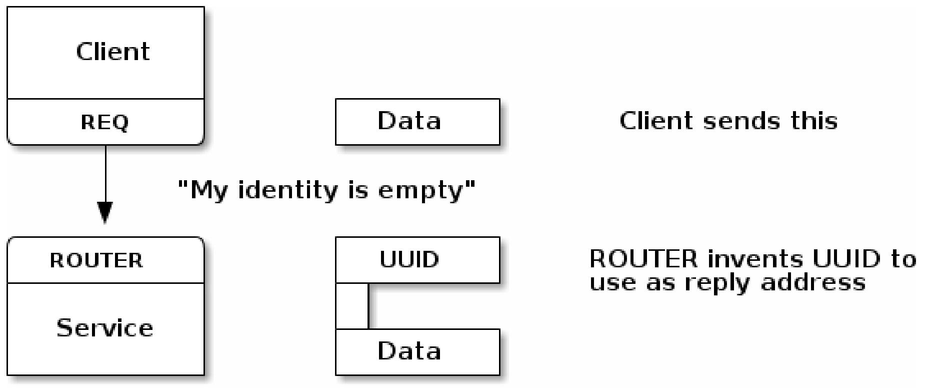


Figure 31 – ROUTER invents a UUID for transient sockets

This is what it looks like for durable sockets:

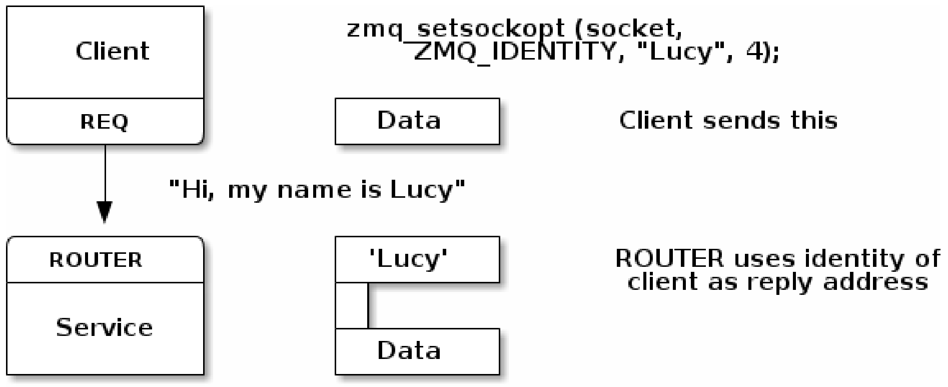

Client
REQ
zmq_setsockopt (socket, ZMQ_IDENTITY, "Lucy", 4);
Data
Client sends this
"Hi, my name is Lucy"
ROUTER
Service
'Lucy'
Data
ROUTER uses identity of client as reply address

Figure 32 – ROUTER uses identity if it knows it

Let's observe the above two cases in practice. This program dumps the contents of the message parts that a ROUTER socket receives from two REP sockets, one not using identities, and one using an identity '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)
{
    void *context = zmq_init (1);

    void *sink = zmq_socket (context, ZMQ_ROUTER);
    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, "ROUTER 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, "ROUTER socket uses REQ's socket identity");
    s_dump (sink);

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

*identity.c: Identity check*

Here is what the dump function prints:

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

## Custom Request-Reply Routing

top prev next

We already saw that ROUTER uses the message envelope to decide which client to route a reply back to. Now let me express that in another way: *ROUTER will route messages asynchronously to any peer connected to it, if you provide the correct routing address via a properly constructed envelope.*

So ROUTER is really a fully controllable router. We'll dig into this magic in detail.

But first, and because we're going to go off-road into some rough and possibly illegal terrain now, let's look closer at REQ and REP. Few people know this, but despite their kindergarten approach to messaging, REQ and REP are actually colorful characters:

- REQ is a **mama** socket, doesn't listen but always expects an answer. Mamas are strictly synchronous and if you use them they are always the 'request' end of a chain.
- REP is a **papa** socket, always answers, but never starts a conversation. Papas are strictly synchronous and if you use them, they are always the 'reply' end of a chain.

The thing about Mama sockets is, as we all learned as kids, you can't speak until spoken to. Mamas do not have simple open-mindedness of papas, nor the ambiguous "sure, whatever" shrugged-shoulder aloofness of a dealer. So to speak to a mama socket, you have to get the mama socket to talk to you first. The good part is mamas don't care if you reply now, or much later. Just bring a good sob story and a bag of laundry.

Papa sockets on the other hand are strong and silent, and pedantic. They do just one thing, which is to give you an answer to whatever you ask, perfectly framed and precise. Don't expect a papa socket to be chatty, or to pass a message on to someone else, this is just not going to happen.

While we usually think of request-reply as a to-and-fro pattern, in fact it can be fully asynchronous, as long as we understand that any mamas or papas will be at the end of a chain, never in the middle of it, and always synchronous. All we need to know is the address of the peer we want to talk to, and then we can then send it messages asynchronously, via a router. The router is the one and only ØMQ socket type capable of being told "send this message to X" where X is the address of a connected peer.

These are the ways we can know the address to send a message to, and you'll see most of these used in the examples of custom request-reply routing:

- If it's an transient socket, i.e. did not set any identity, the router will generate a UUID and use that to refer to the connection when it delivers you an incoming request envelope.

- If it is a durable socket, the router will give the peer's identity when it delivers you an incoming request envelope.

- Peers with explicit identities can send them via some other mechanism, e.g. via some other sockets.

- Peers can have prior knowledge of each others' identities, e.g. via configuration files or some other magic.

There are four custom routing patterns, one for each of the socket types we can connect to a router:

- Router-to-dealer.
- Router-to-mama (REQ).
- Router-to-papa (REP).
- Router-to-router.

In each of these cases we have total control over how we route messages, but the different patterns cover different use-cases and message flows. Let's break it down over the next sections with examples of different routing algorithms.

But first some warnings about custom routing:

- This goes against a fairly solid ØMQ rule: *delegate peer addressing to the socket*. The only reason we do it is because ØMQ lacks a wide range of routing algorithms.
- Future versions of ØMQ will probably do some of the routing we're going to build here. That means the code we design now may break, or become redundant in the future.
- While the built-in routing has certain guarantees of scalability, such as being friendly to devices, custom routing doesn't. You will need to make your own devices.

So overall, custom routing is more expensive and more fragile than delegating this to ØMQ. Only do it if you need it. Having said that, let's jump in, the water's great!

## Router-to-Dealer Routing

top prev next

The router-to-dealer pattern is the simplest. You connect one router to many dealers, and then distribute messages to the dealers using any algorithm you like. The dealers can be sinks (process the messages without any response), proxies (send the messages on to other nodes), or services (send back replies).

If you expect the dealer to reply, there should only be one router talking to it. Dealers have no idea how to reply to a specific peer, so if they have multiple peers, they will load-balance between them, which would be weird. If the dealer is a sink, any number of routers can talk to it.

What kind of routing can you do with a router-to-dealer pattern? If the dealers talk back to the router, e.g. telling the router when they finished a task, you can use that knowledge to route depending on how fast a dealer is. Since both router and dealer are asynchronous, it can get a little tricky. You'd need to use zmq_poll(3) at least.

We'll make an example where the dealers don't talk back, they're pure sinks. Our routing algorithm will be a weighted random scatter: we have two dealers and we send twice as many messages to one as to the other.

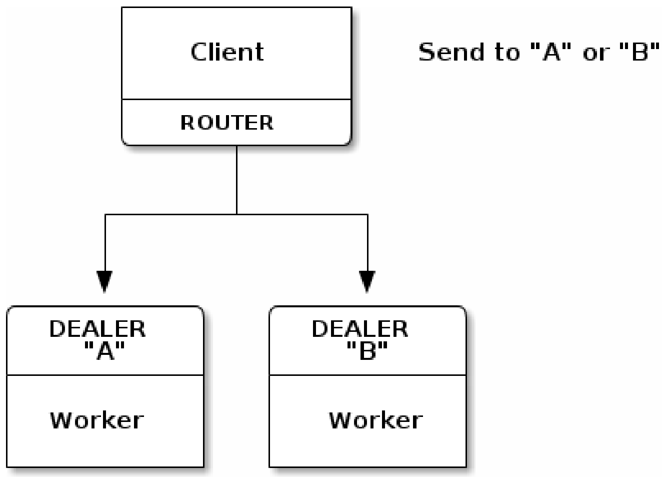


Figure 33 – Router to dealer custom routing

Here's code that shows how this works:

```
//
//  Custom routing Router to Dealer (ROUTER to DEALER)
//
```

```
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//
#include "zhelpers.h"
#include <pthread.h>

//  We have two workers, here we copy the code, normally these would
//  run on different boxes…
//
static void *
worker_task_a (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_DEALER);
    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++;
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}

static void *
worker_task_b (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_DEALER);
    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++;
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}
```

```
int main (void)
{
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (client, "ipc://routing.ipc");

    pthread_t worker;
    pthread_create (&worker, NULL, worker_task_a, NULL);
    pthread_create (&worker, NULL, worker_task_b, NULL);

    //  Wait for threads to connect, since otherwise the messages
    //  we send won't be routable.
    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 (randof (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");

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

*rtdealer.c: Router-to-dealer*

Some comments on this code:

- The router doesn't know when the dealers are ready, and it would be distracting for our example to add in the signaling to do that. So the router just does a "sleep (1)" after starting the dealer threads. Without this sleep, the router will send out messages that can't be routed, and ØMQ will discard them.
- Note that this behavior is specific to ROUTER sockets. PUB sockets will also discard messages if there are no subscribers, but all other socket types will queue sent messages until there's a peer to receive them.

To route to a dealer, we create an envelope like this:

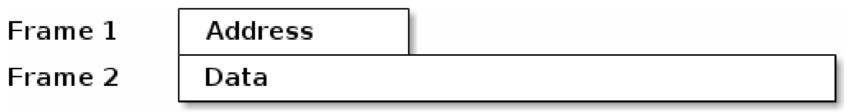


Figure 34 – Routing envelope for dealer

The router socket removes the first frame, and sends the second frame, which the dealer gets as-is. When the dealer sends a message to the router, it sends one frame. The router prepends the dealer's address and gives us back a similar envelope in two parts.

Something to note: if you use an invalid address, the router discards the message silently. There is not much else it can do usefully. In normal cases this either means the peer has gone away, or that there is a programming error somewhere and you're using a bogus address. In any case you cannot ever assume a message will be routed successfully until and unless you get a reply of some sorts from the destination node. We'll come to creating reliable patterns later on.

Dealers in fact they work exactly like PUSH and PULL combined. It's however illegal and pointless to connect PULL or PUSH to a request-reply socket.

## Least-Recently Used Routing (LRU Pattern)

top prev next

Like we said, mamas (REQ sockets, if you really insist on it) don't listen to you, and if you try to speak out of turn they'll ignore you. You have to wait for them to say something, *then* you can give a sarcastic answer. This is very useful for routing because it means we can keep a bunch of mamas waiting for answers. In effect, mamas tell us when they're ready.

You can connect one router to many mamas, and distribute messages as you would to dealers. Mamas will usually want to reply, but they will let you have the last word. However it's one thing at a time:

- Mama speaks to router
- Router replies to mama
- Mama speaks to router
- Router replies to mama
- etc.

Like dealers, mamas can only talk to one router and since mamas always start by talking to the router, you should never connect one mama to more than one router unless you are doing sneaky stuff like multi-pathway redundant routing. I'm not even going to explain that now, and hopefully the jargon is complex enough to stop you trying this until you need it.

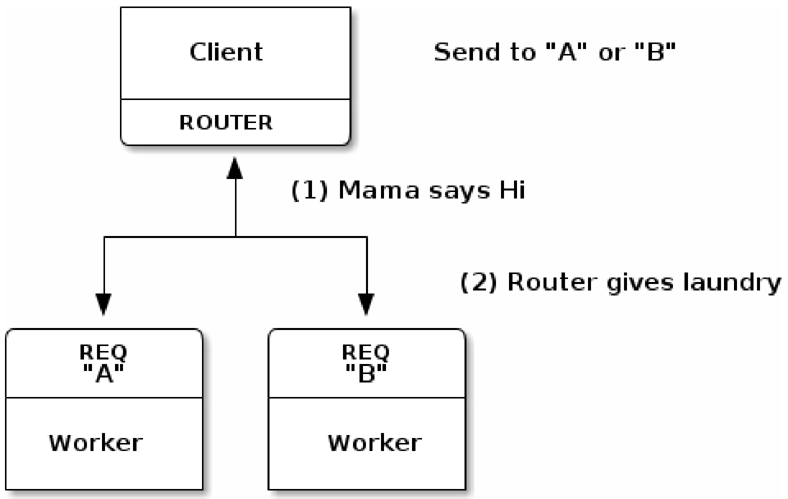


Figure 35 — Router to mama custom routing

What kind of routing can you do with a router-to-mama pattern? Probably the most obvious is "least-recently-used" (LRU), where we always route to the mama that's been waiting longest. Here is an example that does LRU routing to a set of mamas:

```
//
//  Custom routing Router to Mama (ROUTER to REQ)
//
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_WORKERS 10

static void *
worker_task (void *args)
{
    void *context = zmq_init (1);
    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
        s_sleep (randof (1000) + 1);
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}

int main (void)
{
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_ROUTER);
    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_task, NULL);
    }
    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);
    }
    zmq_close (client);
    zmq_term (context);
    return 0;
}
```

*rtmama.c: Router-to-mama*

For this example the LRU doesn't need any particular data structures above what ØMQ gives us (message queues) because we don't need to synchronize the workers with anything. A more realistic LRU algorithm would have to collect workers as they become ready, into a queue, and the use this queue when routing client requests. We'll do this in a later example.

To prove that the LRU is working as expected, the mamas print the total tasks they each did. Since the mamas do random work, and we're not load balancing, we expect each mama to do approximately the same amount but with random variation. And that is indeed what we see:

```
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
```

Some comments on this code

- We don't need any settle time, since the mamas explicitly tell the router when they are ready.
- We're generating our own identities here, as printable strings, using the zhelpers.h s_set_id function. That's just to make our life a little simpler. In a realistic application the mamas would be fully anonymous and then you'd call zmq_recv(3) and zmq_send(3) directly instead of the zhelpers s_recv() and s_send() functions, which can only handle strings.
- Worse, we're using *random* identities. Don't do this in real code, please. Randomized durable sockets are not good in real life, they exhaust and eventually kill nodes.
- If you copy and paste example code without understanding it, you deserve what you get. It's like watching Spiderman leap off the roof and then trying that yourself.

To route to a mama, we must create a mama-friendly envelope like this:

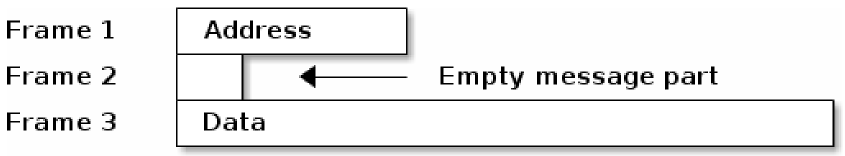


Figure 36 — Routing envelope for mama (REQ)

## Address-based Routing

top prev next

Papas are, if we care about them at all, only there to answer questions. And to pay the bills, fix the car when mama drives it into the garage wall, put up shelves, and walk the dog when it's raining. But apart from that, papas are only there to answer questions.

In a classic request-reply pattern a router wouldn't talk to a papa socket at all, but rather would get a dealer to do the job for it. That's what dealers are for: to pass questions onto random papas and come back with their answers. Routers are generally more comfortable talking to mamas. OK, dear reader, you may stop the psychoanalysis. These are analogies, not life stories.

It's worth remembering with ØMQ that the classic patterns are the ones that work best, that the beaten path is there for a reason, and that when we go off-road we take the risk of falling off cliffs and getting eaten by zombies. Having said that, let's plug a router into a papa and see what the heck emerges.

The special thing about papas, all joking aside, is actually two things:

- One, they are strictly lockstep request-reply.
- Two, they accept an envelope stack of any size and will return that intact.

In the normal request-reply pattern, papas are anonymous and replaceable (wow, these analogies *are* scary), but we're learning about custom routing. So, in our use-case we have reason to send a request to papa A rather than papa B. This is essential if you want to keep some kind of a conversation going between you, at one end of a large network, and a papa sitting somewhere far away.

A core philosophy of ØMQ is that the edges are smart and many, and the middle is vast and dumb. This does mean the edges can address each other, and this also means we want to know how to reach a given papa. Doing routing across multiple hops is something we'll look at later but for now we'll look just at the final step: a router talking to a specific papa:

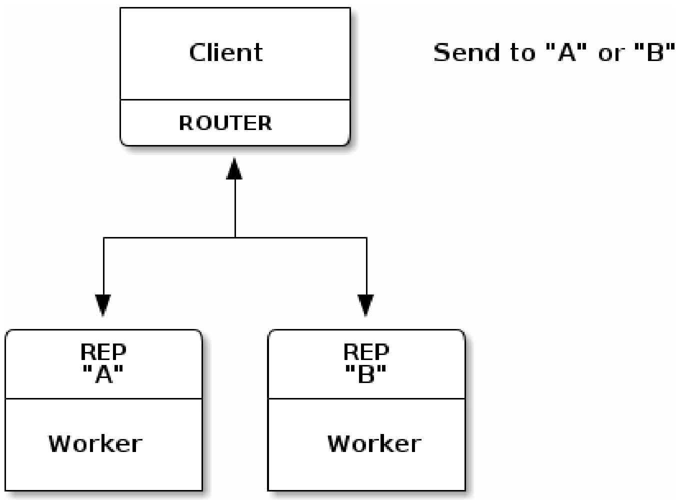


Figure 37 – Router to papa custom routing

This example shows a very specific chain of events:

- The client has a message that it expects to route back (via another router) to some node. The message has two addresses (a stack), an empty part, and a body.
- The client passes that to the router but specifies a papa address first.
- The router removes the papa address, uses that to decide which papa to send the message to.
- The papa receives the addresses, empty part, and body.
- It removes the addresses, saves them, and passes the body to the worker.
- The worker sends a reply back to the papa.
- The papa recreates the envelope stack and sends that back with the worker's reply to the router.
- The router prepends the papa's address and provides that to the client along with the rest of the address stack, empty part, and the body.

It's complex but worth working through until you understand it. Just remember a papa is garbage in, garbage out.

```
//
//  Custom routing Router to Papa (ROUTER to REP)
//
#include "zhelpers.h"

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

    void *client = zmq_socket (context, ZMQ_ROUTER);
    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 the worker to connect so that when we send a message
    //  with routing envelope, it will actually match the worker…
    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 ROUTER socket…
    s_dump (client);

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

*rtpapa.c: Router-to-papa*

Run this program and it should show you this:

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

Some comments on this code:

- In reality we'd have the papa and router in separate nodes. This example does it all in one thread because it makes the sequence of events really clear.
- zmq_connect(3) doesn't happen instantly. When the papa socket connects to the router, that takes a certain time and happens in the background. In a realistic application the router wouldn't even know the papa existed until there had been some previous dialog. In our toy example we'll just `sleep (1);` to make sure the connection's done. If you remove the sleep, the papa socket won't get the message. (Try it.)
- We're routing using the papa's identity. Just to convince yourself this really is happening, try sending to a wrong address, like "B". The papa won't get the message.
- The s_dump and other utility functions (in the C code) come from the zhelpers.h header file. It becomes clear that we do the same work over and over on sockets, and there are interesting layers we can build on top of the ØMQ API. We'll come back to this later when we make a real application rather than these toy examples.

To route to a papa, we must create a papa-friendly envelope like this:

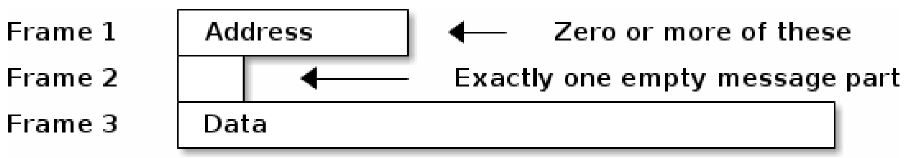


Figure 38 – Routing envelope for papa aka REP

## A Request-Reply Message Broker

top prev next

We'll recap the knowledge we have so far about doing weird stuff with ØMQ message envelopes, and build the core of a generic custom routing queue device that we can properly call a *message broker*. Sorry for all the buzzwords. What we'll make is a *queue device* that connects a bunch of *clients* to a bunch of *workers*, and lets you use *any routing algorithm* you want. What we'll do is *least-recently used*, since it's the most obvious use-case apart from load-balancing.

To start with, let's look back at the classic request-reply pattern and then see how it extends over a larger and larger service-oriented network. The basic pattern is:

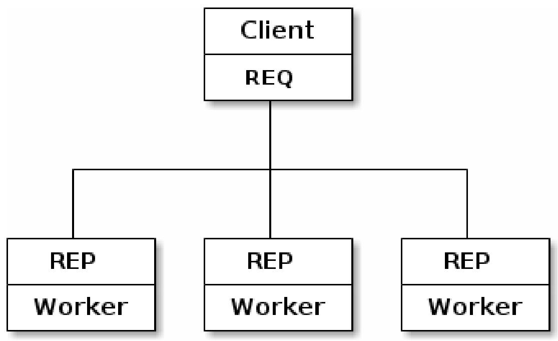


Figure 39 – Basic request-reply

This extends to multiple papas, but if we want to handle multiple mamas as well we need a device in the middle, which normally consists of a router and a dealer back to back, connected by a classic ZMQ_QUEUE device that just copies message parts between the two sockets as fast as it can:

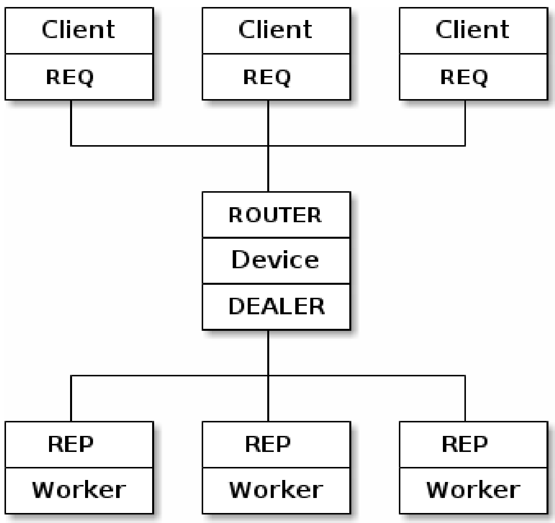


Figure 40 – Stretched request-reply

The key here is that the router stores the originating mama address in the request envelope, the dealer and papas don't touch that, and so the router knows which mama to send the reply back to. Papas are anonymous and not addressed in this pattern, all papas are assumed to provide the same service.

In the above design, we're using the built-in load balancing routing that the dealer socket provides. However we want for our broker to use a least-recently used algorithm, so we take the router-mama pattern we learned, and apply that:

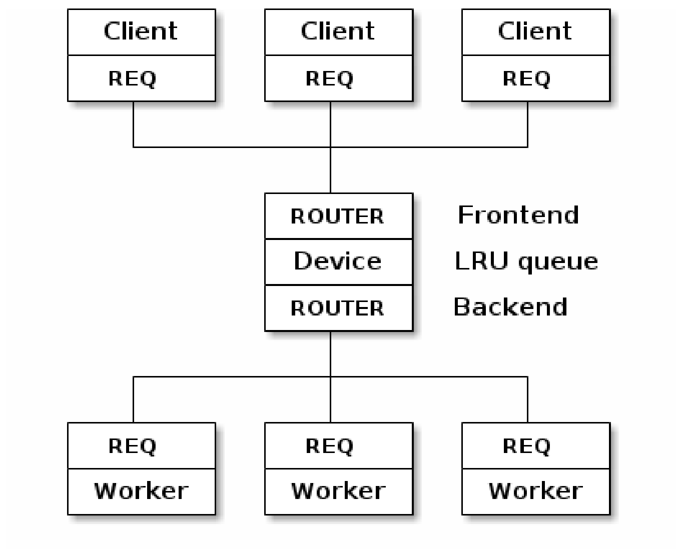


Figure 41 – Stretched request-reply with LRU

Our broker - a router-to-router LRU queue - can't simply copy message parts blindly. Here is the code, it's fairly complex but the core logic is reusable in any request-reply broker that wants to do LRU routing:

```
//
//  Least-recently used (LRU) queue device
//  Clients and workers are shown here in-process
//
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_CLIENTS 10
#define NBR_WORKERS 3

//  Dequeue operation for queue implemented as array of anything
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) - sizeof (q
[0]))

//  Basic request-reply client using REQ socket
//  Since s_send and s_recv can't handle 0MQ binary identities we
//  set a printable text identity to allow routing.
//
static void *
client_task (void *args)
{
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_REQ);
    s_set_id (client);          //  Set a printable identity
    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);
    zmq_close (client);
    zmq_term (context);
    return NULL;
}

//  Worker using REQ socket to do LRU routing
//  Since s_send and s_recv can't handle 0MQ binary identities we
//  set a printable text identity to allow routing.
//
static void *
worker_task (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_REQ);
    s_set_id (worker);          //  Set a printable identity
    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);
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}

int main (void)
{
    //  Prepare our context and sockets
    void *context = zmq_init (1);
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (frontend, "ipc://frontend.ipc");
    zmq_bind (backend,  "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++) {
        pthread_t client;
        pthread_create (&client, NULL, client_task, NULL);
    }
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_task, NULL);
    }
    //  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) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, available_workers? 2: 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--;
        }
    }
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

*lruqueue.c: LRU queue broker*

The difficult part of this program is (a) the envelopes that each socket reads and writes, and (b) the LRU algorithm. We'll take these in turn, starting with the message envelope formats.

First, recall that a mama REQ socket always puts on an empty part (the envelope delimiter) on sending and removes this empty part on reception. The reason for this isn't important, it's just part of the 'normal' request-reply pattern. What we care about here is just keeping mama happy by doing precisely what she needs. Second, the router always adds an envelope with the address of whomever the message came from.

We can now walk through a full request-reply chain from client to worker and back. In the code we set the identity of client and worker sockets to make it easier to print the message frames if we want to. Let's assume the client's identity is "CLIENT" and the worker's identity is "WORKER". The client sends a single frame:

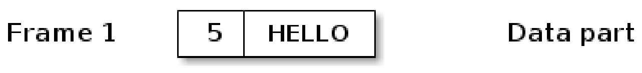


Figure 42 – Message that client sends

What the queue gets, when reading off the router frontend socket is this:

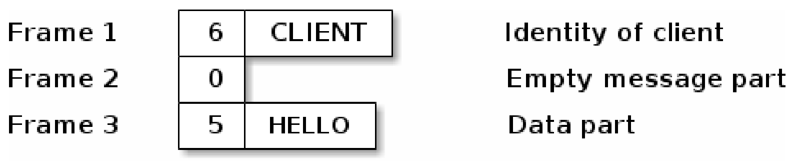


Figure 43 – Message coming in on frontend

The broker sends this to the worker, prefixed by the address of the worker, taken from the LRU queue, plus an additional empty part to keep the mama at the other end happy:

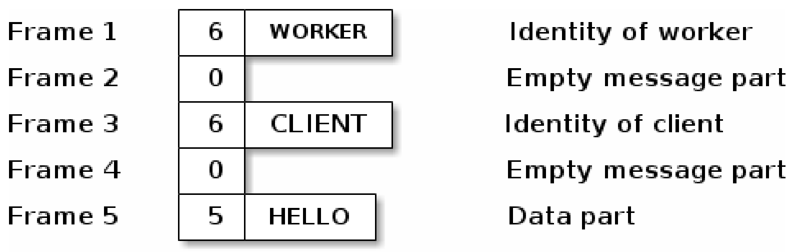


Figure 44 – Message sent to backend

This complex envelope stack gets chewed up first by the backend router socket, which removes the first frame. Then the mama socket in the worker removes the empty part, and provides the rest to the worker:

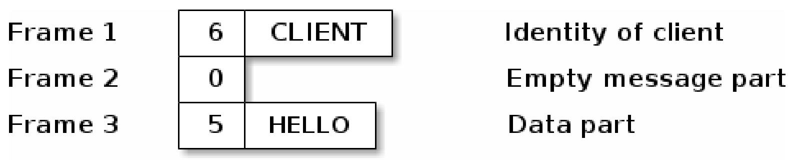


Figure 45 – Message delivered to worker

Which is exactly the same as what the queue received on its frontend router socket. The worker has to save the envelope (which is all the parts up to and including the empty message part) and then it can do what's needed with the data part.

On the return path the messages are the same as when they come in, i.e. the backend socket gives the queue a message in five parts, and the queue sends the frontend socket a message in three parts, and the client gets a message in one part.

Now let's look at the LRU algorithm. It requires that both clients and workers use mama sockets, and that workers correctly store and replay the envelope on messages they get.

The algorithm is:

- Create a pollset which polls the backend always, and the frontend only if there are one or more workers available.
- Poll for activity with infinite timeout.
- If there is activity on the backend, we either have a "ready" message or a reply for a client. In either case we store the worker address (the first part) on our LRU queue, and if the rest is a client reply we send it back to that client via the frontend.
- If there is activity on the frontend, we take the client request, pop the next worker (which is the least-recently used), and send the request to the backend. This means sending the worker address, empty part, and then the three parts of the client request.

You should now see that you can reuse and extend the LRU algorithm with variations based on the information the worker provides in its initial "ready" message. For example, workers might start up and do a performance self-test, then tell the broker how fast they are. The broker can then choose the fastest available worker rather than LRU or round-robin.

## A High-Level API for ØMQ

top prev next

Reading and writing multipart messages using the native ØMQ API is like eating a bowl of hot noodle soup, with fried chicken and extra vegetables, using a toothpick. Look at the core of the worker thread from our LRU queue broker:

```
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);
}
```

That code isn't even reusable, because it can only handle one envelope. And this code already does some wrapping around the ØMQ API. If we used the libzmq API directly this is what we'd have to write:

```
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);
    }
```

What we want is an API that lets us receive and send an entire message in one shot, including all envelopes. One that lets us do what we want with the absolute least lines of code. The ØMQ core API itself doesn't aim to do this, but nothing prevents us making layers on top, and part of learning to use ØMQ intelligently is to do exactly that.

Making a good message API is fairly difficult, especially if we want to avoid copying data around too much. We have a problem of terminology: ØMQ uses "message" to describe both multipart messages, and individual parts of a message. We have a problem of semantics: sometimes it's natural to see message content as printable string data, sometimes as binary blobs.

So one solution is to use three concepts: *string* (already the basis for s_send and s_recv), *frame* (a message part), and *message* (a list of one or more frames). Here is the worker code, rewritten onto an API using these concepts:

```
while (1) {
    zmsg_t *zmsg = zmsg_recv (worker);
    zframe_print (zmsg_last (zmsg), "Worker: ");
    zframe_reset (zmsg_last (zmsg), "OK", 2);
    zmsg_send (&zmsg, worker);
}
```

Replacing 22 lines of code with four is a good deal, especially since the results are easy to read and understand. We can continue this process for other aspects of working with ØMQ. Let's make a wishlist of things we would like in a higher-level API:

- *Automatic handling of sockets.* I find it really annoying to have to close sockets manually, and to have to explicitly define the linger timeout in some but not all cases. It'd be great to have a way to close sockets automatically when I close the context.

- *Portable thread management.* Every non-trivial ØMQ application uses threads, but POSIX threads aren't portable. So a decent high-level API should hide this under a portable layer.
- *Portable clocks.* Even getting the time to a millisecond resolution, or sleeping for some milliseconds, is not portable. Realistic ØMQ applications need portable clocks, so our API should provide them.
- *A reactor to replace zmq_poll(3).* The poll loop is simple but clumsy. Writing a lot of these, we end up doing the same work over and over: calculating timers, and calling code when sockets are ready. A simple reactor with socket readers, and timers, would save a lot of repeated work.
- *Proper handling of Ctrl-C.* We already saw how to catch an interrupt. It would be useful if this happened in all applications.

Turning this wishlist into reality gives us czmq, a high-level C API for ØMQ. This high-level binding in fact developed out of earlier versions of the Guide. It combines nicer semantics for working with ØMQ with some portability layers, and (importantly for C but less for other languages) containers like hashes and lists.

Here is the LRU queue broker rewritten to use czmq:

```
//
//  Least-recently used (LRU) queue device
//  Demonstrates use of the libczmq API
//
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  Signals worker is ready

//  Basic request-reply client using REQ socket
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    //  Send request, get reply
    while (1) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

```
//  Worker using REQ socket to do LRU routing
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "ipc://frontend.ipc");
    zsocket_bind (backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  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
    zlist_t *workers = zlist_new ();

    while (1) {
        //  Initialize poll set
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  Interrupted
        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            //  Use worker address for LRU routing
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  Interrupted
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);

            //  Forward message to client if it's not a READY
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Get client request, route to first available worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

*lruqueue2.c: LRU queue broker using czmq*

One thing czmq provides is clean interrupt handling. This means that Ctrl-C will cause any blocking ØMQ call to exit with a return code -1 and errno set to EINTR. The czmq message recv methods return NULL in such case. So, you can cleanly exit a loop like this:

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

Or, if you're doing zmq_poll, test on the return code:

```
    int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
    if (rc == -1)
        break;              //  Interrupted
```

The previous example still uses zmq_poll(3). So how about reactors? The czmq `zloop` reactor is simple but functional. It lets you:

- Set a reader on any socket, i.e. code that is called whenever the socket has input.
- Cancel a reader on a socket.
- Set a timer that goes off once or multiple times at specific intervals.

`zloop` of course uses zmq_poll(3) internally. It rebuilds its poll set each time you add or remove readers, and it calculates the poll timeout to match the next timer. Then, it calls the reader and timer handlers for each socket and timer that needs attention.

When we use a reactor pattern, our code turns inside out. The main logic looks like this:

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

While the actual handling of messages sits inside dedicated functions or methods. You may not like the style, it's a matter of taste. What it does help with is mixing timers and socket activity. In the rest of this text we'll use zmq_poll(3) in simpler cases, and `zloop` in more complex examples.

Here is the LRU queue broker rewritten once again, this time to use `zloop`:

```
//
//  Least-recently used (LRU) queue device
//  Demonstrates use of the libczmq API and reactor style
//
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  Signals worker is ready

//  Basic request-reply client using REQ socket
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    //  Send request, get reply
    while (1) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  Worker using REQ socket to do LRU routing
//
static void *
worker_task (void *arg_ptr)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  Our LRU queue structure, passed to reactor handlers
typedef struct {
    void *frontend;             //  Listen to clients
    void *backend;              //  Listen to workers
    zlist_t *workers;           //  List of ready workers
} lruqueue_t;

//  Handle input from client, on frontend
int s_handle_frontend (zloop_t *loop, void *socket, void *arg)
{
    lruqueue_t *self = (lruqueue_t *) arg;
    zmsg_t *msg = zmsg_recv (self->frontend);
    if (msg) {
        zmsg_wrap (msg, (zframe_t *) zlist_pop (self->workers));
        zmsg_send (&msg, self->backend);

        //  Cancel reader on frontend if we went from 1 to 0 workers
        if (zlist_size (self->workers) == 0)
            zloop_cancel (loop, self->frontend);
    }
    return 0;
}

//  Handle input from worker, on backend
int s_handle_backend (zloop_t *loop, void *socket, void *arg)
{
    //  Use worker address for LRU routing
    lruqueue_t *self = (lruqueue_t *) arg;
    zmsg_t *msg = zmsg_recv (self->backend);
    if (msg) {
        zframe_t *address = zmsg_unwrap (msg);
        zlist_append (self->workers, address);

        //  Enable reader on frontend if we went from 0 to 1 workers
        if (zlist_size (self->workers) == 1)
            zloop_reader (loop, self->frontend, s_handle_frontend,
self);

        //  Forward message to client if it's not a READY
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
            zmsg_destroy (&msg);
        else
            zmsg_send (&msg, self->frontend);
    }
    return 0;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    lruqueue_t *self = (lruqueue_t *) zmalloc (sizeof (lruqueue_t));
    self->frontend = zsocket_new (ctx, ZMQ_ROUTER);
    self->backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (self->frontend, "ipc://frontend.ipc");
    zsocket_bind (self->backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  Queue of available workers
    self->workers = zlist_new ();

    //  Prepare reactor and fire it up
    zloop_t *reactor = zloop_new ();
    zloop_reader (reactor, self->backend, s_handle_backend, self);
    zloop_start (reactor);
    zloop_destroy (&reactor);

    //  When we're done, clean up properly
    while (zlist_size (self->workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (self->workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&self->workers);
    zctx_destroy (&ctx);
    free (self);
    return 0;
}
```

*lruqueue3.c: LRU queue broker using zloop*

Getting applications to properly shut-down when you send them Ctrl-C can be tricky. If you use the zctx class it'll automatically set-up signal handling, but your code still has to cooperate. You must break any loop if zmq_poll returns -1 or if any of the recv methods (zstr_recv, zframe_recv, zmsg_recv) return NULL. If you have nested loops, it can be useful to make the outer ones conditional on `!zctx_interrupted`.

## Asynchronous Client-Server

top prev next

In the router-to-dealer example we saw a 1-to-N use case where one client talks asynchronously to multiple workers. We can turn this upside-down to get a very useful N-to-1 architecture where various clients talk to a single server, and do this asynchronously:

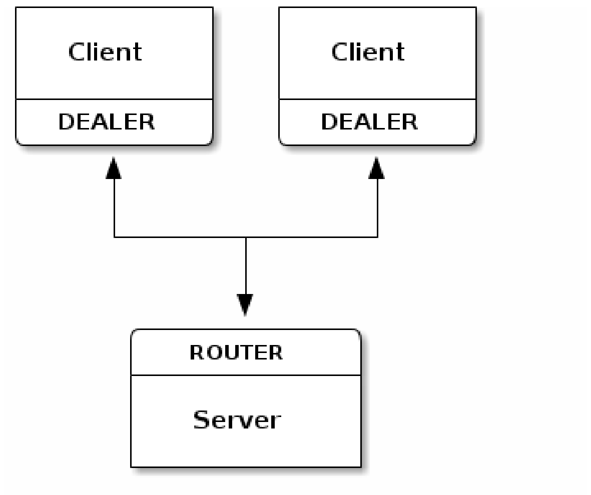


Figure 46 – Asynchronous Client Server

Here's how it works:

- Clients connect to the server and send requests.
- For each request, the server sends 0 to N replies.
- Clients can send multiple requests without waiting for a reply.
- Servers can send multiple replies without waiting for new requests.

Here's code that shows how this works:

```
//
//  Asynchronous client-to-server (DEALER to ROUTER)
//
//  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 "czmq.h"

//  ------------------------------------------------------------------
----
//  This is our client task
//  It connects to the server, and then sends a request once per
second
//  It collects responses as they arrive, and it prints them out. We
will
//  run several client tasks in parallel, each with a different
random ID.

static void *
```

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

    //  Set random identity to make tracing easier
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof
(0x10000));
    zsockopt_set_identity (client, identity);
    zsocket_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, 10 * ZMQ_POLL_MSEC);
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (client);
                zframe_print (zmsg_last (msg), identity);
                zmsg_destroy (&msg);
            }
        }
        zstr_sendf (client, "request #%d", ++request_nbr);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  -----------------------------------------------------------------
----
//  This is our server task
//  It uses the multithreaded server model to deal requests out to a
pool
//  of workers and route replies back to clients. One worker can
handle
//  one request at a time but one client can talk to multiple workers
at
//  once.

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

void *server_task (void *args)
{
    zctx_t *ctx = zctx_new ();

    //  Frontend socket talks to clients over TCP
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5570");

    //  Backend socket talks to workers over inproc
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://backend");

    //  Launch pool of worker threads, precise number is not critical
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++)
        zthread_fork (ctx, server_worker, NULL);
    //  Connect backend to frontend via a 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);
        }
    }
    zctx_destroy (&ctx);
    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 *args, zctx_t *ctx, void *pipe)
{
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "inproc://backend");

    while (1) {
        //  The DEALER socket gives us the address envelope and
message
        zmsg_t *msg = zmsg_recv (worker);
        zframe_t *address = zmsg_pop (msg);
        zframe_t *content = zmsg_pop (msg);
        assert (content);
        zmsg_destroy (&msg);

        //  Send 0..4 replies back
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            //  Sleep for some fraction of a second
            zclock_sleep (randof (1000) + 1);
            zframe_send (&address, worker, ZFRAME_REUSE +
ZFRAME_MORE);
            zframe_send (&content, worker, ZFRAME_REUSE);
        }
        zframe_destroy (&address);
        zframe_destroy (&content);
    }
}
```

```
//  This main thread simply starts several clients, and a server, and
then
//  waits for the server to finish.
//
int main (void)
{
    zctx_t *ctx = zctx_new ();
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, server_task, NULL);

    //  Run for 5 seconds then quit
    zclock_sleep (5 * 1000);
    zctx_destroy (&ctx);
    return 0;
}
```

*asyncsrv.c: Asynchronous client-server*

Just run that example by itself. Like other multi-task examples, it runs in a single process but each task has its own context and conceptually acts as a separate process. You will see three clients (each with a random ID), printing out the replies they get from the server. Look carefully and you'll see each client task gets 0 or more replies per request.

Some comments on this code:

- The clients send a request once per second, and get zero or more replies back. To make this work using zmq_poll(3), we can't simply poll with a 1-second timeout, or we'd end up sending a new request only one second *after we received the last reply*. So we poll at a high frequency (100 times at 1/100th of a second per poll), which is approximately accurate. This means the server could use requests as a form of heartbeat, i.e. detecting when clients are present or disconnected.

- The server uses a pool of worker threads, each processing one request synchronously. It connects these to its frontend socket using an internal queue. To help debug this, the code implements its own queue device logic. In the C code, you can uncomment the zmsg_dump() calls to get debugging output.

The socket logic in the server is fairly wicked. This is the detailed architecture of the server:

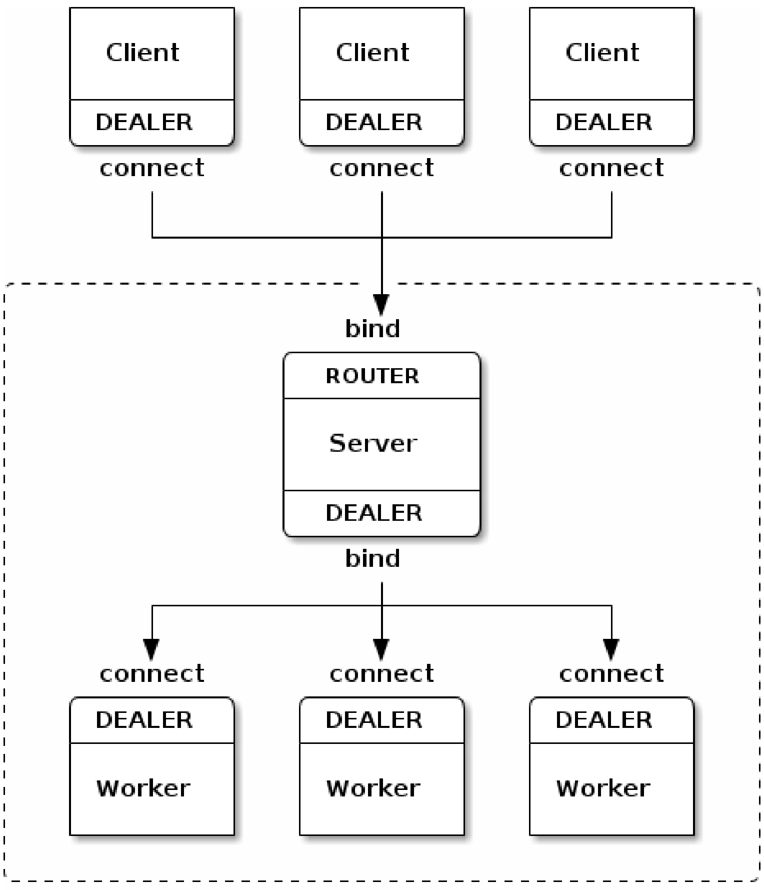


Figure 47 – Detail of asynchronous server

Note that we're doing a dealer-to-router dialog between client and server, but internally between the server main thread and workers we're doing dealer-to-dealer. If the workers were strictly synchronous, we'd use REP. But since we want to send multiple replies we need an async socket. We do *not* want to route replies, they always go to the single server thread that sent us the request.

Let's think about the routing envelope. The client sends a simple message. The server thread receives a two-part message (real message prefixed by client identity). We have two possible designs for the server-to-worker interface:

- Workers get unaddressed messages, and we manage the connections from server thread to worker threads explicitly using a router socket as backend. This would require that workers start by telling the server they exist, which can then route requests to workers and track which client is 'connected' to which worker. This is the LRU pattern we already covered.
- Workers get addressed messages, and they return addressed replies. This requires that workers can properly decode and recode envelopes but it doesn't need any other mechanisms.

The second design is much simpler, so that's what we use:

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

When you build servers that maintain stateful conversations with clients, you will run into a classic problem. If the server keeps some state per client, and clients keep coming and going, eventually it will run of resources. Even if the same clients keep connecting, if you're using transient sockets (no explicit identity), each connection will look like a new one.

We cheat in the above example by keeping state only for a very short time (the time it takes a worker to process a request) and then throwing away the state. But that's not practical for many cases.

To properly manage client state in a stateful asynchronous server you must:

- Do heartbeating from client to server. In our example we send a request once per second, which can reliably be used as a heartbeat.
- Store state using the client identity as key. This works for both durable and transient sockets.
- Detect a stopped heartbeat. If there's no request from a client within, say, two seconds, the server can detect this and destroy any state it's holding for that client.

## Router-to-Router (N-to-N) Routing

top prev next

We've seen ROUTER/router sockets talking to dealers, mamas, and papas. The last case is routers talking to routers. One use-case for this is a web farm that has redundant HTTP front-ends talking to an array of asynchronous back-end workers. Each worker accepts requests from any of the front-end HTTP servers, and processes them asynchronously, sending asynchronous replies back. A fully asynchronous worker has some internal concurrency but we don't really care about that here. What interests us is how N workers can talk to N front-ends.

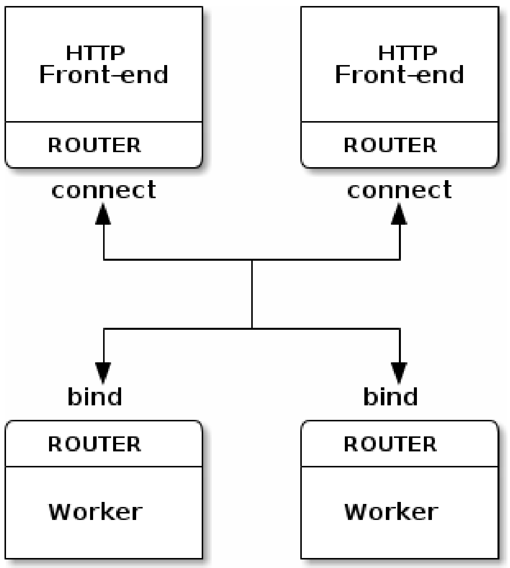


Figure 48 – N to N routing

Here's a simplified example with a single front-end and a single worker, cross connected and routing to each other. We just send a message each way, and dump the message envelopes:

```
//
//  Cross-connected ROUTER sockets addressing each other
//
#include "zhelpers.h"
```

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

    void *worker = zmq_socket (context, ZMQ_ROUTER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "WORKER", 6);
    zmq_bind (worker, "ipc://rtrouter.ipc");

    void *server = zmq_socket (context, ZMQ_ROUTER);
    zmq_setsockopt (server, ZMQ_IDENTITY, "SERVER", 6);
    zmq_connect (server, "ipc://rtrouter.ipc");

    //  Wait for the worker to connect so that when we send a message
    //  with routing envelope, it will actually match the worker…
    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;
}
```

*rtrouter.c: Cross-connected routers*

The program produces this output:

```
----------------------------------------
[008] SERVER
[000]
[014] send to worker
----------------------------------------
[006] WORKER
[000]
[014] send to server
```

Some comments on this code:

- We need to give the two sockets time to connect and exchange identities. If we don't, then they will discard the messages you try to send to them, not recognizing the address. Try commenting out the sleep(1), and then trying again.
- We can set and use identities on both bound and connected sockets, as this example shows.

Although the router-to-router pattern looks ideal for asynchronous N-to-N routing, it has some pitfalls. First, any design with N-to-N connections will not scale beyond a small number of clients and servers. You should really create a device in the middle that turns it into two 1-to-N patterns. This gives you a structure like the LRU queue broker, though you would use DEALER at the front-end and worker sides to get streaming.

Second, it may become confusing if you try to put two ROUTER sockets at the same logical level. One must bind, one must connect, and request-reply is inherently asymmetric. However, the next point takes care of this.

Third, one side of the connection has to know the identity of the other, *up front*. You cannot do router-to-router flows between two transient sockets. In practice this means you need a name service, configuration data, or some other magic to define and share the identities of one of the peers. It's convenient therefore to treat the more static side of the flow as 'server', and give it a fixed, known identity, and then treat the dynamic side as 'client'. The client will have to connect to the server, then send it a message using the server's known identity as address, and then the server can respond to the client.

## Worked Example: Inter-Broker Routing

top prev next

Let's take everything we've seen so far, and scale things up. Our best client calls us urgently and asks for a design of a large cloud computing facility. He has this vision of a cloud that spans many data centers, each a cluster of clients and workers, and that works together as a whole.

Because we're smart enough to know that practice always beats theory, we propose to make a working simulation using ØMQ. Our client, eager to lock down the budget before his own boss changes his mind, and having read great things about ØMQ on Twitter, agrees.

### Establishing the Details

top prev next

Several espressos later, we want to jump into writing code but a little voice tells us to get more details before making a sensational solution to entirely the wrong problem. What kind of work is the cloud doing?, we ask. The client explains:

- Workers run on various kinds of hardware, but they are all able to handle any task. There are several hundred workers per cluster, and as many as a dozen clusters in total.
- Clients create tasks for workers. Each task is an independent unit of work and all the client wants is to find an available worker, and send it the task, as soon as possible. There will be a lot of clients and they'll come and go arbitrarily.
- The real difficulty is to be able to add and remove clusters at any time. A cluster can can leave or join the cloud instantly, bringing all its workers and clients with it.
- If there are no workers in their own cluster, clients' tasks will go off to other available workers in the cloud.
- Clients send out one task at a time, waiting for a reply. If they don't get an answer within X seconds they'll just send out the task again. This ain't our concern, the client API does it already.
- Workers process one task at a time, they are very simple beasts. If they crash, they get restarted by whatever script started them.

So we double check to make sure that we understood this correctly:

- There will be some kind of super-duper network interconnect between clusters, right? The client says, yes, of course, we're not idiots.

- What kind of volumes are we talking about, we ask? The client replies, up to a thousand clients per cluster, each doing max. ten requests per second. Requests are small, and replies are also small, no more than 1K bytes each.

So we do a little calculation and see that this will work nicely over plain TCP. 2,500 clients x 10/second x 1,000 bytes x 2 directions = 50MB/sec or 400Mb/sec, not a problem for a 1Gb network.

It's a straight-forward problem that requires no exotic hardware or protocols, just some clever routing algorithms and careful design. We start by designing one cluster (one data center) and then we figure out how to connect clusters together.

## Architecture of a Single Cluster

top prev next

Workers and clients are synchronous. We want to use the LRU pattern to route tasks to workers. Workers are all identical, our facility has no notion of different services. Workers are anonymous, clients never address them directly. We make no attempt here to provide guaranteed delivery, retry, etc.

For reasons we already looked at, clients and workers won't speak to each other directly. It makes it impossible to add or remove nodes dynamically. So our basic model consists of the request-reply message broker we saw earlier:

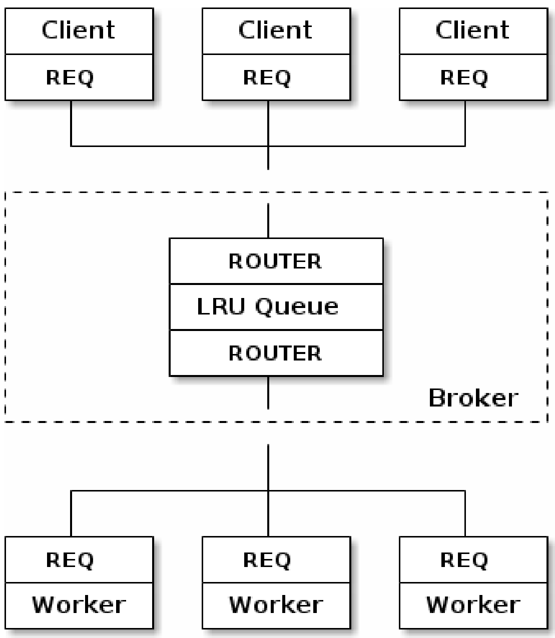


Figure 49 – Cluster architecture

## Scaling to Multiple Clusters

top prev next

Now we scale this out to more than one cluster. Each cluster has a set of clients and workers, and a broker that joins these together:

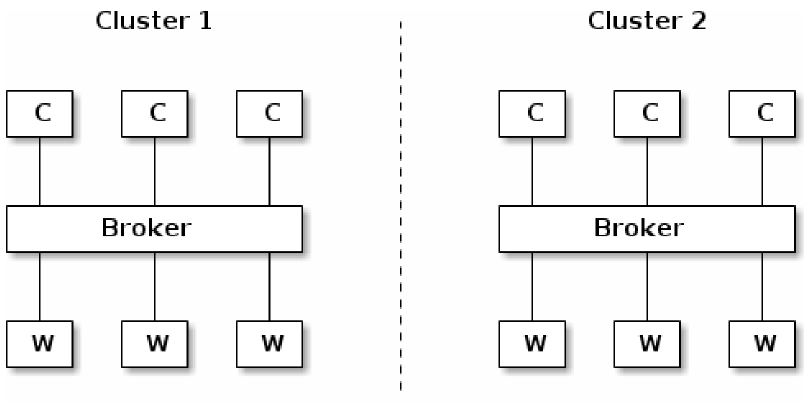


Figure 50 – Multiple clusters

The question is: how do we get the clients of each cluster talking to the workers of the other cluster? There are a few possibilities, each with pros and cons:

- Clients could connect directly to both brokers. The advantage is that we don't need to modify brokers or workers. But clients get more complex, and become aware of the overall topology. If we want to add, e.g. a third or forth cluster, all the clients are affected. In effect we have to move routing and failover logic into the clients and that's not nice.
- Workers might connect directly to both brokers. But mama workers can't do that, they can only reply to one broker. We might use papas but papas don't give us customizable broker-to-worker routing like LRU, only the built-in load balancing. That's a fail, if we want to distribute work to idle workers: we precisely need LRU. One solution would be to use router sockets for the worker nodes. Let's label this "Idea #1".
- Brokers could connect to each other. This looks neatest because it creates the fewest additional connections. We can't add clusters on the fly but that is probably out of scope. Now clients and workers remain ignorant of the real network topology, and brokers tell each other when they have spare capacity. Let's label this "Idea #2".

Let's explore Idea #1. Workers connecting to both brokers and accepting jobs from either:

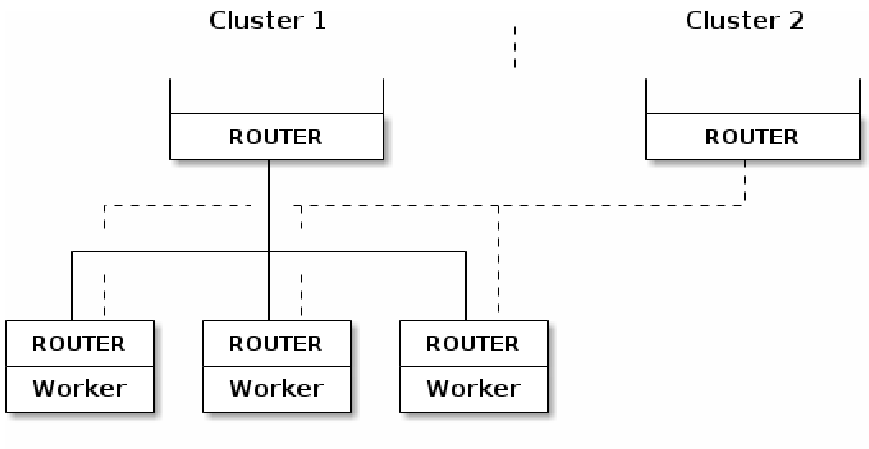


Figure 51 – Idea 1 – cross-connected workers

It looks feasible. However it doesn't provide what we wanted, which was that clients get local workers if possible and remote workers only if it's better than waiting. Also workers will signal "ready" to both brokers and can get two jobs at once, while other workers remain idle. It seems this design fails because again we're putting routing logic at the edges.

So idea #2 then. We interconnect the brokers and don't touch the clients or workers, which are mamas like we're used to:

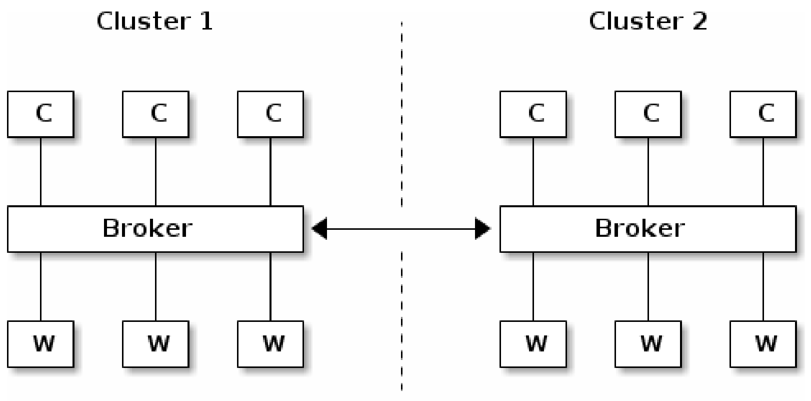


Figure 52 — Idea 2 — brokers talking to each other

This design is appealing because the problem is solved in one place, invisibly to the rest of the world. Basically, brokers open secret channels to each other and whisper, like camel traders, *"hey, I've got some spare capacity, if you have too many clients give me a shout and we'll deal".*

It is in effect just a more sophisticated routing algorithm: brokers become subcontractors for each other. Other things to like about this design, even before we play with real code:

- It treats the common case (clients and workers on the same cluster) as default and does extra work for the exceptional case (shuffling jobs between clusters).
- It lets us use different message flows for the different types of work. That means we can handle them differently, e.g. using different types of network connection.
- It feels like it would scale smoothly. Interconnecting three, or more brokers doesn't get over-complex. If we find this to be a problem, it's easy to solve by adding a super-broker.

We'll now make a worked example. We'll pack an entire cluster into one process. That is obviously not realistic but it makes it simple to simulate, and the simulation can accurately scale to real processes. This is the beauty of ØMQ, you can design at the microlevel and scale that up to the macro level. Thread become processes, become boxes and the patterns and logic remain the same. Each of our 'cluster' processes contains client threads, worker threads, and a broker thread.

We know the basic model well by now:

- The mama client (REQ) threads create workloads and pass them to the broker (ROUTER).
- The mama worker (REQ) threads process workloads and return the results to the broker (ROUTER).
- The broker queues and distributes workloads using the LRU routing model.

## Federation vs. Peering

top prev next

There are several possible way to interconnecting brokers. What we *want* is to be able to tell other brokers, "we have capacity", and then receive multiple tasks. We also need to be able to tell other brokers "stop, we're full". It doesn't need to be perfect: sometimes we may accept jobs we can't process immediately, then we'll do them as soon as possible.

The simplest interconnect is *federation* in which brokers simulate clients and workers for each other. We would do this by connecting our frontend to the other broker's backend socket. Note that it is legal to both bind a socket to an endpoint and connect it to other endpoints.

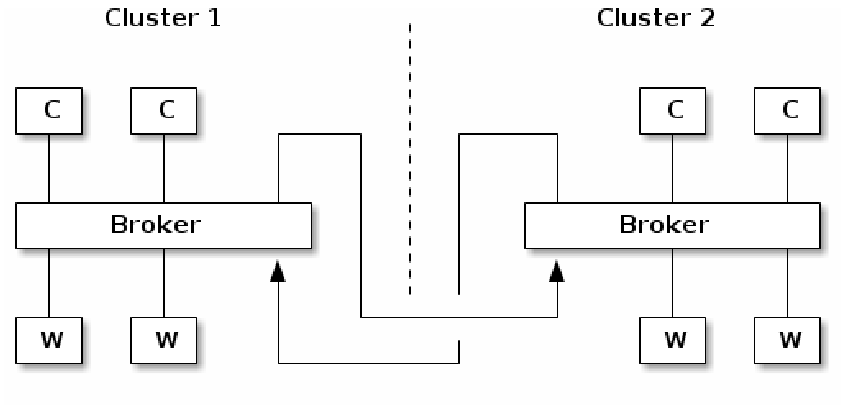


Figure 53 – Cross connected brokers in federation model

This would give us simple logic in both brokers and a reasonably good mechanism: when there are no clients, tell the other broker 'ready', and accept one job from it. The problem is also that it is too simple for this problem. A federated broker would be able to handle only one task at once. If the broker emulates a lock-step client and worker, it is by definition also going to be lock-step and if it has lots of available workers they won't be used. Our brokers need to be connected in a fully asynchronous fashion.

The federation model is perfect for other kinds of routing, especially service-oriented architectures or SOAs (which route by service name and proximity rather than LRU or load-balancing or random scatter). So don't dismiss it as useless, it's just not right for least-recently used and cluster load-balancing.

So instead of federation, let's look at a *peering* approach in which brokers are explicitly aware of each other and talk over privileged channels. Let's break this down, assuming we want to interconnect N brokers. Each broker has (N - 1) peers, and all brokers are using exactly the same code and logic. There are two distinct flows of information between brokers:

- Each broker needs to tell its peers how many workers it has available at any time. This can be fairly simple information, just a quantity that is updated regularly. The obvious (and correct) socket pattern for this is publish-subscribe. So every broker opens a PUB socket and publishes state information on that, and every broker also opens a SUB socket and connects that to the PUB socket of every other broker, to get state information from its peers.
- Each broker needs a way to delegate tasks to a peer and get replies back, asynchronously. We'll do this using router/router (ROUTER/ROUTER) sockets, no other combination works. Each broker has two such sockets: one for tasks it receives, one for tasks it delegates. If we didn't use two sockets it would be more work to know whether we were reading a request or a reply each time. That would mean adding more information to the message envelope.

And there is also the flow of information between a broker and its local clients and workers.

## The Naming Ceremony

top prev next

Three flows x two sockets for each flow = six sockets that we have to manage in the broker. Choosing good names is vital to keeping a multi-socket juggling act reasonably coherent in our minds. Sockets *do* something and what they do should form the basis for their names. It's about being able to read the code several weeks later on a cold Monday morning before coffee, and not feeling pain.

Let's do a shamanistic naming ceremony for the sockets. The three flows are:

- A *local* request-reply flow between the broker and its clients and workers.
- A *cloud* request-reply flow between the broker and its peer brokers.
- A *state* flow between the broker and its peer brokers.

Finding meaningful names that are all the same length means our code will align beautifully. It may seem irrelevant but such attention to such details turn ordinary code into something more like art.

For each flow the broker has two sockets that we can orthogonally call the "frontend" and "backend". We've used these names quite often. A frontend receives information or tasks. A backend sends those out to other peers. The conceptual flow is from front to back (with replies going in the opposite direction from back to front).

So in all the code we write for this tutorial will use these socket names:

- *localfe* and *localbe* for the local flow.
- *cloudfe* and *cloudbe* for the cloud flow.
- *statefe* and *statebe* for the state flow.

For our transport we'll use `ipc` for everything. This has the advantage of working like `tcp` in terms of connectivity (i.e. it's a disconnected transport, unlike `inproc`), yet we don't need IP addresses or DNS names, which would be a pain here. Instead, we will use `ipc` endpoints called *something*-`local`, *something*-`cloud`, and *something*-`state`, where *something* is the name of our simulated cluster.

You may be thinking that this is a lot of work for some names. Why not call them s1, s2, s3, s4, etc.? The answer is that if your brain is not a perfect machine, you need a lot of help when reading code, and we'll see that these names do help. It is a lot easier to remember "three flows, two directions" than "six different sockets".

Here is the broker socket arrangement, then:

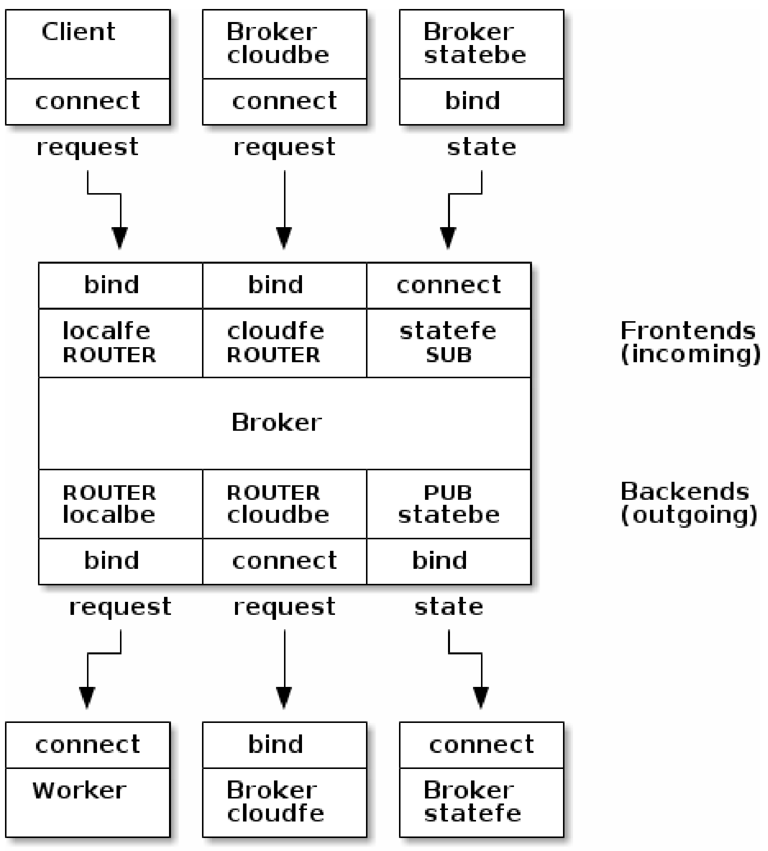


Figure 54 – Broker socket arrangement

Note that we connect the cloudbe in each broker to the cloudfe in every other broker, and likewise we connect the statebe in each broker to the statefe in every other broker.

## Prototyping the State Flow

top prev next

Since each socket flow has its own little traps for the unwary, we will test them in real code one by one, rather than try to throw the whole lot into code in one go. When we're happy with each flow, we can put them together into a full program. We'll start with the state flow:

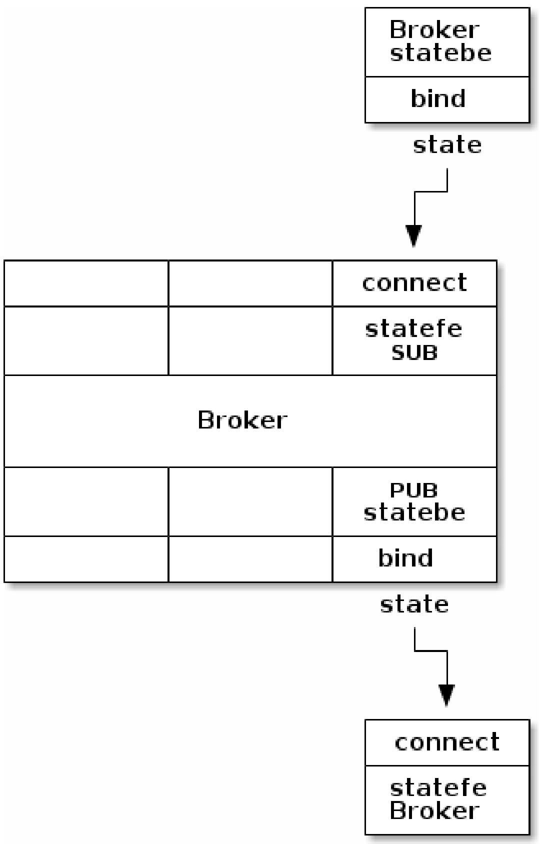


Figure 55 — The state flow

Here is how this works in code:

```
//
//  Broker peering simulation (part 1)
//  Prototypes the state flow
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    //  First argument is this broker's name
    //  Other arguments are our peers' names
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}…\n");
        exit (EXIT_FAILURE);
    }
    char *self = argv [1];
    printf ("I: preparing broker at %s…\n", self);
    srandom ((unsigned) time (NULL));

    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  Connect statefe to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  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
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle incoming status message
        if (items [0].revents & ZMQ_POLLIN) {
            char *peer_name = zstr_recv (statefe);
            char *available = zstr_recv (statefe);
            printf ("%s - %s workers free\n", peer_name, available);
            free (peer_name);
            free (available);
        }
        else {
            //  Send random value for worker availability
            zstr_sendm (statebe, self);
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

*peering1.c: Prototype state flow*

Notes about this code:

- Each broker has an identity that we use to construct `ipc` endpoint names. A real broker would need to work with TCP and a more sophisticated configuration scheme. We'll look at such schemes later in this book but for now, using generated `ipc` names lets us ignore the problem of where to get TCP/IP addresses or names from.
- We use a zmq_poll(3) loop as the core of the program. This processes incoming messages and sends out state messages. We send a state message *only* if we did not get any incoming messages *and* we waited for a second. If we send out a state message each time we get one in, we'll get message storms.
- We use a two-part pubsub message consisting of sender address and data. Note that we will need to know the address of the publisher in order to send it tasks, and the only way is to send this explicitly as a part of the message.
- We don't set identities on subscribers, because if we did then we'd get out of date state information when connecting to running brokers.
- We don't set a HWM on the publisher, since subscribers are transient. We might set a HWM of 1 but it's extra work for nothing here.

We can build this little program and run it three times to simulate three clusters. Let's call them DC1, DC2, and DC3 (the names are arbitrary). We run these three commands, each in a separate window:

```
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
```

You'll see each cluster report the state of its peers, and after a few seconds they will all happily be printing random numbers once per second. Try this and satisfy yourself that the three brokers all match up and synchronize to per-second state updates.

In real life we'd not send out state messages at regular intervals but rather whenever we had a state change, i.e. whenever a worker becomes available or unavailable. That may seem like a lot of traffic but state messages are small and we've established that the inter-cluster connections are super-fast.

If we wanted to send state messages at precise intervals we'd create a child thread and open the statebe socket in that thread. We'd then send irregular state updates to that child thread from our main thread, and allow the child thread to conflate them into regular outgoing messages. This is more work than we need here.

## Prototyping the Local and Cloud Flows

top prev next

Let's now prototype at the flow of tasks via the local and cloud sockets. This code pulls requests from clients and then distributes them to local workers and cloud peers on a random basis:

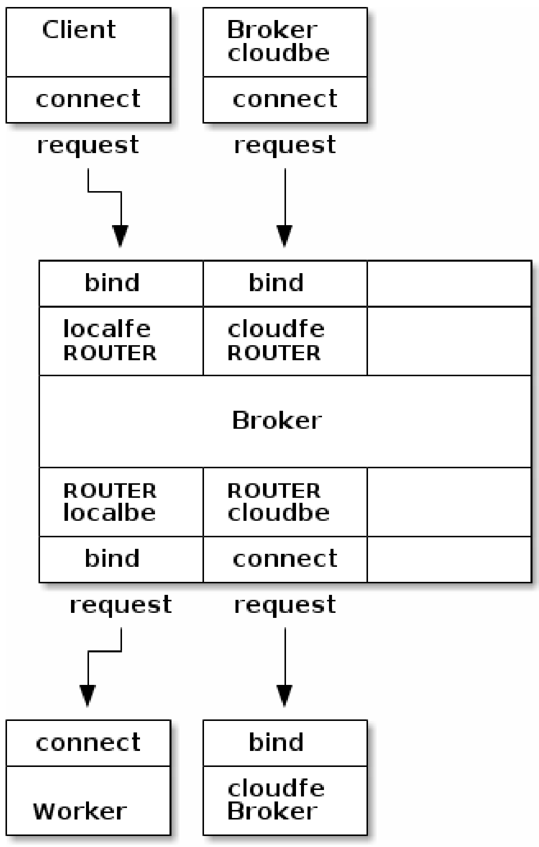


Figure 56 – The flow of tasks

Before we jump into the code, which is getting a little complex, let's sketch the core routing logic and break it down into a simple but robust design.

We need two queues, one for requests from local clients and one for requests from cloud clients. One option would be to pull messages off the local and cloud frontends, and pump these onto their respective queues. But this is kind of pointless because ØMQ sockets *are* queues already. So let's use the ØMQ socket buffers as queues.

This was the technique we used in the LRU queue broker, and it worked nicely. We only read from the two frontends when there is somewhere to send the requests. We can always read from the backends, since they give us replies to route back. As long as the backends aren't talking to us, there's no point in even looking at the frontends.

So our main loop becomes:

- Poll the backends for activity. When we get a message, it may be "READY" from a worker or it may be a reply. If it's a reply, route back via the local or cloud frontend.
- If a worker replied, it became available, so we queue it and count it.
- While there are workers available, take a request, if any, from either frontend and route to a local worker, or randomly, a cloud peer.

Randomly sending tasks to a peer broker rather than a worker simulates work distribution across the cluster. It's idiot but that is fine for this stage.

We use broker identities to route messages between brokers. Each broker has a name, which we provide on the command line in this simple prototype. As long as these names don't overlap with the ØMQ-generated UUIDs used for client nodes, we can figure out whether to route a reply back to a client or to a broker.

Here is how this works in code. The interesting part starts around the comment

"Interesting part".

```c
//
//  Broker peering simulation (part 2)
//  Prototypes the request-reply flow
//
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  Signals worker is ready

//  Our own name; in practice this'd be configured per node
static char *self;

//  Request-reply client using REQ socket
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);

    while (1) {
        //  Send request, get reply
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              //  Interrupted
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  Worker using REQ socket to do LRU routing
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  Process messages as they arrive
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted
        zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

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);
    }
    self = argv [1];
    printf ("I: preparing broker at %s…\n", self);
    srandom ((unsigned) time (NULL));

    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    char endpoint [256];

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%s'\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }
    //  Prepare local frontend and backend
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);
    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  Get user to tell us when we can start…
    printf ("Press Enter when all brokers are started: ");
    getchar ();

    //  Start local workers
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  Start local clients
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);

    //  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;
    zlist_t *workers = zlist_new ();

    while (1) {
        zmq_pollitem_t backends [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 }
        };
        //  If we have no workers anyhow, wait indefinitely
        int rc = zmq_poll (backends, 2,
            capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle reply from local worker
        zmsg_t *msg = NULL;
        if (backends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  Interrupted
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);
            capacity++;

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  Or handle reply from peer broker
        else
        if (backends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker address for anything
            zframe_t *address = zmsg_unwrap (msg);
            zframe_destroy (&address);
        }
        //  Route reply to cloud if it's addressed to a broker
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  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) {
                msg = zmsg_recv (cloudfe);
                reroutable = 0;
            }
            else
            if (frontends [0].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (localfe);
                reroutable = 1;
            }
            else
                break;      //  No work, go back to backends

            //  If reroutable, send to cloud 20% of the time
            //  Here we'd normally use cloud status information
            //
            if (reroutable && argc > 2 && randof (5) == 0) {
                //  Route to random broker peer
                int random_peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [random_peer], strlen (argv
[random_peer]));
                zmsg_send (&msg, cloudbe);
            }
            else {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                capacity--;
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

*peering2.c: Prototype local and cloud flow*

Run this by, for instance, starting two instance of the broker in two windows:

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

Some comments on this code:

- Using the zmsg class makes life much easier, and our code much shorter. It's obviously a abstraction that works, and which should form part of your toolbox as a ØMQ programmer.

- Since we're not getting any state information from peers, we naively assume they are running. The code prompts you to confirm when you've started all the brokers. In the real case we'd not send anything to brokers who had not told us they exist.

You can satisfy yourself that the code works by watching it run forever. If there were any misrouted messages, clients would end up blocking, and the brokers would stop printing trace information. You can prove that by killing either of the brokers. The other broker tries to send requests to the cloud, and one by one its clients block, waiting for an answer.

## Putting it All Together

top prev next

Let's put this together into a single package. As before, we'll run an entire cluster as one process. We're going to take the two previous examples and merge them into one properly working design that lets you simulate any number of clusters.

This code is the size of both previous prototypes together, at 270 LoC. That's pretty good for a simulation of a cluster that includes clients and workers and cloud workload distribution. Here is the code:

```
//
//  Broker peering simulation (part 3)
//  Prototypes the full flow of status and tasks
//
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Each thread has its own
//  context and conceptually acts as a separate process.
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define LRU_READY   "\001"      //  Signals worker is ready

//  Our own name; in practice this'd be configured per node
static char *self;

//  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_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "ipc://%s-monitor.ipc", self);

    while (1) {
        sleep (randof (5));
        int burst = randof (15);
        while (burst--) {
            char task_id [5];
            sprintf (task_id, "%04X", randof (0x10000));

            //  Send request with random hex ID
            zstr_send (client, task_id);

            //  Wait max ten seconds for a reply, then complain
            zmq_pollitem_t pollset [1] = { { client, 0, ZMQ_POLLIN, 0
} };
            int rc = zmq_poll (pollset, 1, 10 * 1000 *
ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  Interrupted

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
                    break;              //  Interrupted
                //  Worker is supposed to answer us with our task id
                puts (reply);
                assert (streq (reply, task_id));
                free (reply);
            }
            else {
                zstr_sendf (monitor,
                    "E: CLIENT EXIT - lost task %s", task_id);
                return NULL;
            }
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  Worker using REQ socket to do LRU routing
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

      //  Tell broker we're ready for work
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    while (1) {
        //  Workers are busy for 0/1/2 seconds
        zmsg_t *msg = zmsg_recv (worker);
        sleep (randof (2));
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    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: peering3 me {you}…\n");
        exit (EXIT_FAILURE);
    }
    self = argv [1];
    printf ("I: preparing broker at %s…\n", self);
    srandom ((unsigned) time (NULL));

    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    char endpoint [256];

    //  Bind cloud frontend to endpoint
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  Bind state backend / publisher to endpoint
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  Connect cloud backend to all peers
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to cloud frontend at '%s'\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }

    //  Connect statefe to all peers
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: connecting to state backend at '%s'\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  Prepare local frontend and backend
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);

    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  Prepare monitor socket
    void *monitor = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (monitor, "ipc://%s-monitor.ipc", self);

    //  Start local workers
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  Start local clients
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);
    //  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;
    zlist_t *workers = zlist_new ();

    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
        int rc = zmq_poll (primary, 4,
            local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  Interrupted

        //  Track if capacity changes during this iteration
        int previous = local_capacity;

        //  Handle reply from local worker
        zmsg_t *msg = NULL;

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

            //  If it's READY, don't route the message any further
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  Or handle reply from peer broker
        else
        if (primary [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  We don't use peer broker address for anything
            zframe_t *address = zmsg_unwrap (msg);
            zframe_destroy (&address);
        }
        //  Route reply to cloud if it's addressed to a broker
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  Route reply to client if we still need to
        if (msg)
            zmsg_send (&msg, localfe);

        //  Handle capacity updates
        if (primary [2].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (statefe);
            cloud_capacity = atoi (status);
            free (status);
        }
        //  Handle monitor message
        if (primary [3].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (monitor);
            printf ("%s\n", status);
            free (status);
        }

        //  Now route as many clients requests as we can handle
        //  - If we have local capacity we poll both localfe and
cloudfe
        //  - If we have cloud capacity only, we poll just localfe
        //  - 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)
                msg = zmsg_recv (localfe);
            else
            if (secondary [1].revents & ZMQ_POLLIN)
                msg = zmsg_recv (cloudfe);
            else
                break;      //  No work, go back to primary

            if (local_capacity) {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                local_capacity--;
            }
            else {
                //  Route to random broker peer
                int random_peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [random_peer], strlen (argv
[random_peer]));
                zmsg_send (&msg, cloudbe);
            }
        }
```

```
            if (local_capacity != previous) {
                //  We stick our own address onto the envelope
                zstr_sendm (statebe, self);
                //  Broadcast new capacity
                zstr_sendf (statebe, "%d", local_capacity);
            }
        }
        //  When we're done, clean up properly
        while (zlist_size (workers)) {
            zframe_t *frame = (zframe_t *) zlist_pop (workers);
            zframe_destroy (&frame);
        }
        zlist_destroy (&workers);
        zctx_destroy (&ctx);
        return EXIT_SUCCESS;
    }
```

*peering3.c: Full cluster simulation*

It's a non-trivial program and took about a day to get working. These are the highlights:

- The client threads detect and report a failed request. They do this by polling for a response and if none arrives after a while (10 seconds), printing an error message.
- Client threads don't print directly, but instead send a message to a 'monitor' socket (PUSH) that the main loop collects (PULL) and prints off. This is the first case we've seen of using ØMQ sockets for monitoring and logging; this is a big use case we'll come back to later.
- Clients simulate varying loads to get the cluster 100% at random moments, so that tasks are shifted over to the cloud. The number of clients and workers, and delays in the client and worker threads control this. Feel free to play with them to see if you can make a more realistic simulation.
- The main loop uses two pollsets. It could in fact use three: information, backends, and frontends. As in the earlier prototype, there is no point in taking a frontend message if there is no backend capacity.

These are some of the problems that hit during development of this program:

- Clients would freeze, due to requests or replies getting lost somewhere. Recall that the ØMQ ROUTER/router socket drops messages it can't route. The first tactic here was to modify the client thread to detect and report such problems. Secondly, I put zmsg_dump() calls after every recv() and before every send() in the main loop, until it was clear what the problems were.
- The main loop was mistakenly reading from more than one ready socket. This caused the first message to be lost. Fixed that by reading only from the first ready socket.
- The zmsg class was not properly encoding UUIDs as C strings. This caused UUIDs that contain 0 bytes to be corrupted. Fixed by modifying zmsg to encode UUIDs as printable hex strings.

This simulation does not detect disappearance of a cloud peer. If you start several peers and stop one, and it was broadcasting capacity to the others, they will continue to send it work even if it's gone. You can try this, and you will get clients that complain of lost requests. The solution is twofold: first, only keep the capacity information for a short time so that if a peer does disappear, its capacity is quickly set to 'zero'. Second, add reliability to the request-reply chain. We'll look at reliability in the next chapter.

# Chapter Four - Reliable Request-Reply

top prev next

In Chapter Three we looked at advanced use of ØMQ's request-reply pattern with worked examples. In this chapter we'll look at the general question of reliability and build a set of reliable messaging patterns on top of ØMQ's core request-reply pattern.

In this chapter we focus heavily on user-space 'patterns', which are reusable models that help you design your ØMQ architecture:

- The *Lazy Pirate* pattern: reliable request reply from the client side.
- The *Simple Pirate* pattern: reliable request-reply using a LRU queue.
- The *Paranoid Pirate* pattern: reliable request-reply with heartbeating.
- The *Majordomo* pattern: service-oriented reliable queuing.
- The *Titanic* pattern: disk-based / disconnected reliable queuing.
- The *Binary Star* pattern: primary-backup server failover.
- The *Freelance* pattern: brokerless reliable request-reply.

## What is "Reliability"?

top prev next

To understand what 'reliability' means, we have to look at its opposite, namely *failure*. If we can handle a certain set of failures, we are reliable with respect to those failures. No more, no less. So let's look at the possible causes of failure in a distributed ØMQ application, in roughly descending order of probability:

- Application code is the worst offender. It can crash and exit, freeze and stop responding to input, run too slowly for its input, exhaust all memory, etc.
- System code - like brokers we write using ØMQ - can die. System code should be more reliable than application code but can still crash and burn, and especially run out of memory if it tries to compensate for slow clients.
- Message queues can overflow, typically in system code that has learned to deal brutally with slow clients. When a queue overflows, it starts to discard messages.
- Networks can fail temporarily, causing intermittent message loss. Such errors are hidden to ØMQ applications since it automatically reconnects peers after a network-forced disconnection.
- Hardware can fail and take with it all the processes running on that box.
- Networks can fail in exotic ways, e.g. some ports on a switch may die and those parts of the network become inaccessible.
- Entire data centers can be struck by lightning, earthquakes, fire, or more mundane power or cooling failures.

To make a software system fully reliable against *all* of these possible failures is an enormously difficult and expensive job and goes beyond the scope of this modest guide.

Since the first five cases cover 99.9% of real world requirements outside large companies (according to a highly scientific study I just ran), that's what we'll look at. If you're a large company with money to spend on the last two cases, contact me immediately, there's a large hole behind my beach house waiting to be converted into a pool.

## Designing Reliability

top prev next

So to make things brutally simple, reliability is "keeping things working properly when code freezes or crashes", a situation we'll shorten to "dies". However the things we want to keep working properly are more complex than just messages. We need to take each core ØMQ messaging pattern and see how to make it work (if we can) even when code dies.

Let's take them one by one:

- Request-reply: if the server dies (while processing a request), the client can figure that out since it won't get an answer back. Then it can give up in a huff, wait and try again later, find another server, etc. As for the client dying, we can brush that off as "someone else's problem" for now.

- Publish-subscribe: if the client dies (having gotten some data), the server doesn't know about it. Pubsub doesn't send any information back from client to server. But the client can contact the server out-of-band, e.g. via request-reply, and ask, "please resend everything I missed". As for the server dying, that's out of scope for here. Subscribers can also self-verify that they're not running too slowly, and take action (e.g. warn the operator, and die) if they are.

- Pipeline: if a worker dies (while working), the ventilator doesn't know about it. Pipelines, like pubsub, and the grinding gears of time, only work in one direction. But the downstream collector can detect that one task didn't get done, and send a message back to the ventilator saying, "hey, resend task 324!" If the ventilator or collector die, then whatever upstream client originally sent the work batch can get tired of waiting and resend the whole lot. It's not elegant but system code should really not die often enough to matter.

In this chapter we'll focus on request-reply, and we'll cover reliable pub-sub and pipeline in the following chapters.

The basic request-reply pattern (a REQ client socket doing a blocking send/recv to a REP server socket) scores low on handling the most common types of failure. If the server crashes while processing the request, the client just hangs forever. If the network loses the request or the reply, the client hangs forever.

It is a lot better than TCP, thanks to ØMQ's ability to reconnect peers silently, to load-balance messages, and so on. But it's still not good enough for real work. The only use case where you can trust the basic request-reply pattern is between two threads in the same process where there's no network or separate server process to die.

However, with a little extra work this humble pattern becomes a good basis for real work across a distributed network, and we get a set of reliable request-reply patterns I like to call the "Pirate" patterns. RRR!

There are, roughly, three ways to connect clients to servers, each needing a specific approach to reliability:

- Multiple clients talking directly to a single server. Use case: single well-known server that clients need to talk to. Types of failure we aim to handle: server crashes and restarts, network disconnects.

- Multiple clients talking to a single queue device that distributes work to multiple servers. Use case: workload distribution to workers. Types of failure we aim to handle: worker crashes and restarts, worker busy looping, worker overload, queue crashes and restarts, network disconnects.

- Multiple clients talking to multiple servers with no intermediary devices. Use case: distributed services such as name resolution. Types of failure we aim to handle: service crashes and restarts, service busy looping, service overload, network disconnects.

Each of these has their trade-offs and often you'll mix them. We'll look at all three of these in detail.

## Client-side Reliability (Lazy Pirate Pattern)

top prev next

We can get very simple reliable request-reply with only some changes in the client. We call this the Lazy Pirate pattern. Rather than doing a blocking receive, we:

- Poll the REQ socket and only receive from it when it's sure a reply has arrived.
- Resend a request several times, it no reply arrived within a timeout period.
- Abandon the transaction if after several requests, there is still no reply.

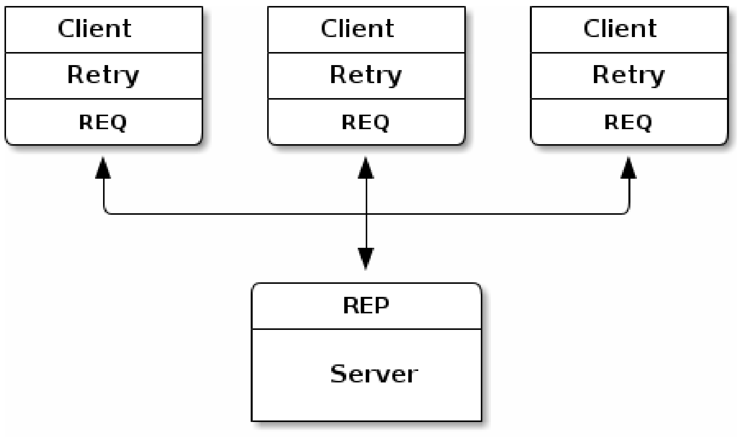


Figure 57 — Lazy Pirate pattern

If you try to use a REQ socket in anything than a strict send-recv fashion, you'll get an error (technically, the REQ socket implements a small finite-state machine to enforce the send-recv ping-pong, and so the error code is called "EFSM"). This is slightly annoying when we want to use REQ in a pirate pattern, because we may send several requests before getting a reply. The pretty good brute-force solution is to close and reopen the REQ socket after an error:

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

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

int main (void)
{
    zctx_t *ctx = zctx_new ();
    printf ("I: connecting to server…\n");
    void *client = zsocket_new (ctx, ZMQ_REQ);
    assert (client);
    zsocket_connect (client, SERVER_ENDPOINT);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    while (retries_left && !zctx_interrupted) {
        //  We send a request, then we work to get a reply
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);
        int expect_reply = 1;
        while (expect_reply) {
            //  Poll socket for a reply, with timeout
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 }
};
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT *
ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  Interrupted

            //  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 = zstr_recv (client);
                if (!reply)
                    break;      //  Interrupted
                if (atoi (reply) == sequence) {
                    printf ("I: server replied OK (%s)\n", reply);
                    retries_left = REQUEST_RETRIES;
                    expect_reply = 0;
                }
                else
                    printf ("E: malformed reply from server: %s\n",
                        reply);

                free (reply);
            }
            else
            if (--retries_left == 0) {
                printf ("E: server seems to be offline,
abandoning\n");
                break;
            }
            else {
                printf ("W: no response from server, retrying…\n");
                //  Old socket is confused; close it and open a new
one
                zsocket_destroy (ctx, client);
                printf ("I: reconnecting to server…\n");
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, SERVER_ENDPOINT);
                //  Send request again, on new socket
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

*lpclient.c: Lazy Pirate client*

Run this together with the matching server:

```
//
//  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 (2);
        }
        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;
}
```

*lpserver.c: Lazy Pirate server*

To run this testcase, start the client and the server in two console windows. The server will randomly misbehave after a few messages. You can check the client's response. Here is a typical output from the server:

```
I: normal request (1)
I: normal request (2)
I: normal request (3)
I: simulating CPU overload
I: normal request (4)
I: simulating a crash
```

And here is the client's response:

```
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
```

The client sequences each message, and checks that replies come back exactly in order: that no requests or replies are lost, and no replies come back more than once, or out of order. Run the test a few times until you're convinced this mechanism actually works. You don't need sequence numbers in reality, they just help us trust our design.

The client uses a REQ socket, and does the brute-force close/reopen because REQ sockets impose a strict send/receive cycle. You might be tempted to use a DEALER instead, but it would not be a good decision. First, it would mean emulating the secret sauce that REQ does with envelopes (if you've forgotten what that is, it's a good sign you don't want to have to do it). Second, it would mean potentially getting back replies that you didn't expect.

Handling failures only at the client works when we have a set of clients talking to a single server. It can handle a server crash, but only if recovery means restarting that same server. If there's a permanent error - e.g. a dead power supply on the server hardware - this approach won't work. Since the application code in servers is usually the biggest source of failures in any architecture, depending on a single server is not a great idea.

So, pros and cons:

- Pro: simple to understand and implement.
- Pro: works easily with existing client and server application code.
- Pro: ØMQ automatically retries the actual reconnection until it works.
- Con: doesn't do failover to backup / alternate servers.

## Basic Reliable Queuing (Simple Pirate Pattern)

top prev next

Our second approach takes Lazy Pirate pattern and extends it with a queue device that lets us talk, transparently, to multiple servers, which we can more accurately call 'workers'. We'll develop this in stages, starting with a minimal working model, the Simple Pirate pattern.

In all these Pirate patterns, workers are stateless, or have some shared state we don't know about, e.g. a shared database. Having a queue device means workers can come and go without clients knowing anything about it. If one worker dies, another takes over. This is a nice simple topology with only one real weakness, namely the central queue itself, which can become a problem to manage, and a single point of failure.

The basis for the queue device is the least-recently-used (LRU) routing queue from Chapter Three. What is the very *minimum* we need to do to handle dead or blocked workers? Turns out, its surprisingly little. We already have a retry mechanism in the client. So using the standard LRU queue will work pretty well. This fits with ØMQ's philosophy that we can extend a peer-to-peer pattern like request-reply by plugging naive devices in the middle:

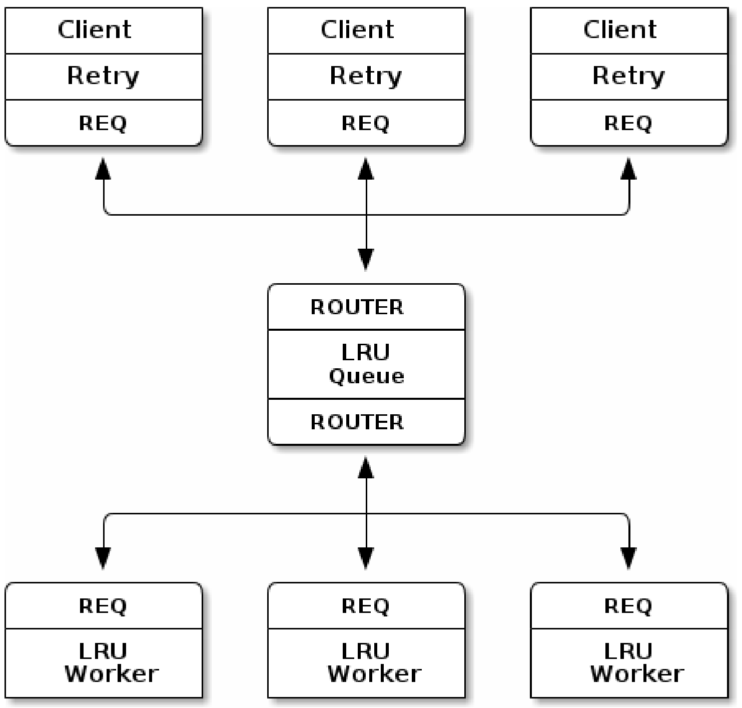


Figure 58 – Simple Pirate Pattern

We don't need a special client, we're still using the Lazy Pirate client. Here is the queue, which is exactly a LRU queue, no more or less:

```
//
//  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 "czmq.h"

#define LRU_READY   "\001"      //  Signals worker is ready

int main (void)
{
    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  For clients
    zsocket_bind (backend,  "tcp://*:5556");    //  For workers

    //  Queue of available workers
    zlist_t *workers = zlist_new ();

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  Interrupted
        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            //  Use worker address for LRU routing
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  Interrupted
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);

            //  Forward message to client if it's not a READY
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Get client request, route to first available worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

*spqueue.c: Simple Pirate queue*

Here is the worker, which takes the Lazy Pirate server and adapts it for the LRU pattern (using the REQ 'ready' signaling):

```
//
//  Simple Pirate worker
//  Connects REQ socket to tcp://*:5556
//  Implements worker part of LRU queueing
//
#include "czmq.h"
#define LRU_READY   "\001"      //  Signals worker is ready

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);

    //  Set random identity to make tracing easier
    srandom ((unsigned) time (NULL));
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof
(0x10000));
```

```
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen
(identity));
    zsocket_connect (worker, "tcp://localhost:5556");

    //  Tell broker we're ready for work
    printf ("I: (%s) worker ready\n", identity);
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    int cycles = 0;
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  Interrupted

        //  Simulate various problems, after a few cycles
        cycles++;
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating a crash\n", identity);
            zmsg_destroy (&msg);
            break;
        }
        else
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) simulating CPU overload\n", identity);
            sleep (3);
            if (zctx_interrupted)
                break;
        }
        printf ("I: (%s) normal reply\n", identity);
        sleep (1);              //  Do some heavy work
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return 0;
}
```

*spworker.c: Simple Pirate worker*

To test this, start a handlful of workers, a client, and the queue, in any order. You'll see that the workers eventually all crash and burn, and the client retries and then gives up. The queue never stops, and you can restart workers and clients ad-nauseam. This model works with any number of clients and workers.

## Robust Reliable Queuing (Paranoid Pirate Pattern)

top prev next

The Simple Pirate Queue pattern works pretty well, especially since it's just a combination of two existing patterns, but it has some weaknesses:

- It's not robust against a queue crash and restart. The client will recover, but the workers won't. While ØMQ will reconnect workers' sockets automatically, as far as the newly started queue is concerned, the workers haven't signalled "READY", so don't exist. To fix this we have to do heartbeating from queue to worker, so that the worker can detect when the queue has gone away.
- The queue does not detect worker failure, so if a worker dies while idle, the queue can only remove it from its worker queue by first sending it a request. The client

waits and retries for nothing. It's not a critical problem but it's not nice. To make this work properly we do heartbeating from worker to queue, so that the queue can detect a lost worker at any stage.

We'll fix these in a properly pedantic Paranoid Pirate Pattern.

We previously used a REQ socket for the worker. For the Paranoid Pirate worker we'll switch to a DEALER socket. This has the advantage of letting us send and receive messages at any time, rather than the lock-step send/receive that REQ imposes. The downside of DEALER is that we have to do our own envelope management. If you don't know what I mean, please re-read Chapter Three.

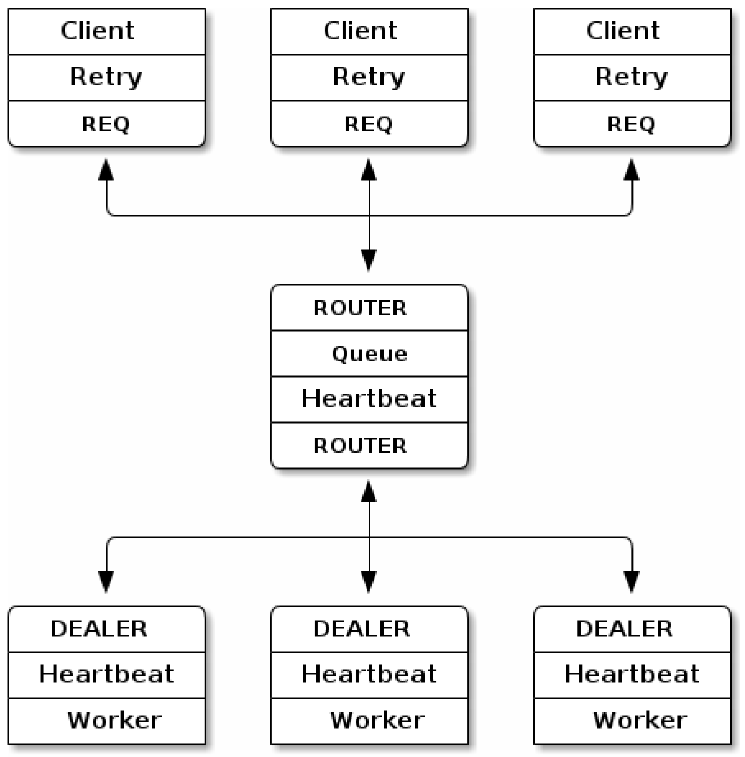


Figure 59 – Paranoid Pirate Pattern

We're still using the Lazy Pirate client. Here is the Paranoid Pirate queue device:

```
//
//  Paranoid Pirate queue
//
#include "czmq.h"

#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  1000    //  msecs

//  Paranoid Pirate Protocol constants
#define PPP_READY       "\001"      //  Signals worker is ready
#define PPP_HEARTBEAT   "\002"      //  Signals worker heartbeat

//  This defines one active worker in our worker list

typedef struct {
    zframe_t *address;          //  Address of worker
    char *identity;             //  Printable identity
    int64_t expiry;             //  Expires at this time
} worker_t;
```

```
//  Construct new worker
static worker_t *
s_worker_new (zframe_t *address)
{
    worker_t *self = (worker_t *) zmalloc (sizeof (worker_t));
    self->address = address;
    self->identity = zframe_strdup (address);
    self->expiry = zclock_time () + HEARTBEAT_INTERVAL *
HEARTBEAT_LIVENESS;
    return self;
}

//  Destroy specified worker object, including identity frame.
static void
s_worker_destroy (worker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        worker_t *self = *self_p;
        zframe_destroy (&self->address);
        free (self->identity);
        free (self);
        *self_p = NULL;
    }
}

//  Worker is ready, remove if on list and move to end
static void
s_worker_ready (worker_t *self, zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (streq (self->identity, worker->identity)) {
            zlist_remove (workers, worker);
            s_worker_destroy (&worker);
            break;
        }
        worker = (worker_t *) zlist_next (workers);
    }
    zlist_append (workers, self);
}

//  Return next available worker address
static zframe_t *
s_workers_next (zlist_t *workers)
{
    worker_t *worker = zlist_pop (workers);
    assert (worker);
    zframe_t *frame = worker->address;
    worker->address = NULL;
    s_worker_destroy (&worker);
    return frame;
}

//  Look for & kill expired workers. Workers are oldest to most
recent,
//  so we stop at the first alive worker.
static void
s_workers_purge (zlist_t *workers)
```

```
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              //  Worker is alive, we're done here

        zlist_remove (workers, worker);
        s_worker_destroy (&worker);
        worker = (worker_t *) zlist_first (workers);
    }
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend  = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  For clients
    zsocket_bind (backend,  "tcp://*:5556");    //  For workers

    //  List of available workers
    zlist_t *workers = zlist_new ();

    //  Send out heartbeats at regular intervals
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  Poll frontend only if we have available workers
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1,
            HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        //  Handle worker activity on backend
        if (items [0].revents & ZMQ_POLLIN) {
            //  Use worker address for LRU routing
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  Interrupted

            //  Any sign of life from worker means it's ready
            zframe_t *address = zmsg_unwrap (msg);
            worker_t *worker = s_worker_new (address);
            s_worker_ready (worker, workers);

            //  Validate control message, or return reply to client
            if (zmsg_size (msg) == 1) {
                zframe_t *frame = zmsg_first (msg);
                if (memcmp (zframe_data (frame), PPP_READY, 1)
                &&  memcmp (zframe_data (frame), PPP_HEARTBEAT, 1)) {
                    printf ("E: invalid message from worker");
                    zmsg_dump (msg);
                }
                zmsg_destroy (&msg);
            }
            else
```

```
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Now get next client request, route to next worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (!msg)
                break;          //  Interrupted
            zmsg_push (msg, s_workers_next (workers));
            zmsg_send (&msg, backend);
        }

        //  Send heartbeats to idle workers if it's time
        if (zclock_time () >= heartbeat_at) {
            worker_t *worker = (worker_t *) zlist_first (workers);
            while (worker) {
                zframe_send (&worker->address, backend,
                             ZFRAME_REUSE + ZFRAME_MORE);
                zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
                zframe_send (&frame, backend, 0);
                worker = (worker_t *) zlist_next (workers);
            }
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
        s_workers_purge (workers);
    }

    //  When we're done, clean up properly
    while (zlist_size (workers)) {
        worker_t *worker = (worker_t *) zlist_pop (workers);
        s_worker_destroy (&worker);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

*ppqueue.c: Paranoid Pirate queue*

The queue extends the LRU pattern with heartbeating of workers. It's simple once it works, but quite difficult to invent. I'll explain more about heartbeating in a second.

Here is the Paranoid Pirate worker:

```
//
//  Paranoid Pirate worker
//
#include "czmq.h"

#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  1000    //  msecs
#define INTERVAL_INIT       1000    //  Initial reconnect
#define INTERVAL_MAX       32000    //  After exponential backoff

//  Paranoid Pirate Protocol constants
#define PPP_READY       "\001"      //  Signals worker is ready
#define PPP_HEARTBEAT   "\002"      //  Signals worker heartbeat

//  Helper function that returns a new configured socket
//  connected to the Paranoid Pirate queue
```

```
static void *
s_worker_socket (zctx_t *ctx) {
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");

    //  Tell queue we're ready for work
    printf ("I: worker ready\n");
    zframe_t *frame = zframe_new (PPP_READY, 1);
    zframe_send (&frame, worker, 0);

    return worker;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = s_worker_socket (ctx);

    //  If liveness hits zero, queue is considered disconnected
    size_t liveness = HEARTBEAT_LIVENESS;
    size_t interval = INTERVAL_INIT;

    //  Send out heartbeats at regular intervals
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    srandom ((unsigned) time (NULL));
    int cycles = 0;
    while (1) {
        zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL *
ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        if (items [0].revents & ZMQ_POLLIN) {
            //  Get message
            //  - 3-part envelope + content -> request
            //  - 1-part HEARTBEAT -> heartbeat
            zmsg_t *msg = zmsg_recv (worker);
            if (!msg)
                break;          //  Interrupted

            if (zmsg_size (msg) == 3) {
                //  Simulate various problems, after a few cycles
                cycles++;
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: simulating a crash\n");
                    zmsg_destroy (&msg);
                    break;
                }
                else
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: simulating CPU overload\n");
                    sleep (3);
                    if (zctx_interrupted)
                        break;
                }
                printf ("I: normal reply\n");
                zmsg_send (&msg, worker);
                liveness = HEARTBEAT_LIVENESS;
                sleep (1);              //  Do some heavy work
                if (zctx_interrupted)
                    break;
            }
            else
            if (zmsg_size (msg) == 1) {
                zframe_t *frame = zmsg_first (msg);
                if (memcmp (zframe_data (frame), PPP_HEARTBEAT, 1) == 
0)
                    liveness = HEARTBEAT_LIVENESS;
                else {
                    printf ("E: invalid message\n");
                    zmsg_dump (msg);
                }
                zmsg_destroy (&msg);
            }
            else {
                printf ("E: invalid message\n");
                zmsg_dump (msg);
            }
            interval = INTERVAL_INIT;
        }
        else
        if (--liveness == 0) {
            printf ("W: heartbeat failure, can't reach queue\n");
            printf ("W: reconnecting in %zd msec…\n", interval);
            zclock_sleep (interval);

            if (interval < INTERVAL_MAX)
                interval *= 2;
            zsocket_destroy (ctx, worker);
            worker = s_worker_socket (ctx);
            liveness = HEARTBEAT_LIVENESS;
        }

        //  Send heartbeat to queue if it's time
        if (zclock_time () > heartbeat_at) {
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
            printf ("I: worker heartbeat\n");
            zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
            zframe_send (&frame, worker, 0);
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

*ppworker.c: Paranoid Pirate worker*

Some comments about this example:

- The code includes simulation of failures, as before. This makes it (a) very hard to debug, and (b) dangerous to reuse. When you want to debug this, disable the failure simulation.
- As for the Paranoid Pirate queue, the heartbeating is quite tricky to get right. See below for a discussion about this.
- The worker uses a reconnect strategy similar to the one we designed for the Lazy Pirate client. With two major differences: (a) it does an exponential back-off, and (b)

it never abandons.

Try the client, queue, and workers, e.g. using a script like this:

```
ppqueue &
for i in 1 2 3 4; do
    ppworker &
    sleep 1
done
lpclient &
```

You should see the workers die, one by one, as they simulate a crash, and the client eventually give up. You can stop and restart the queue and both client and workers will reconnect and carry on. And matter what you do to queues and workers, the client will never get an out-of-order reply: the whole chain either works, or the client abandons.

## Heartbeating

top prev next

When writing the Paranoid Pirate examples, it took about five hours to get the queue-to-worker heartbeating working properly. The rest of the request-reply chain took perhaps ten minutes. Heartbeating is one of those reliability layers that often causes more trouble than it saves. It is especially easy to create 'false failures', i.e. peers decide that they are disconnected because the heartbeats aren't sent properly.

Some points to consider when understanding and implementing heartbeating:

- Note that heartbeats are not request-reply. They flow asynchronously in both directions. Either peer can decide the other is 'dead' and stop talking to it.
- If one of the peers uses durable sockets, this means it may get heartbeats queued up that it will receive if it reconnects. For this reason, workers should *not* reuse durable sockets. The example code uses durable sockets for debugging purposes but they are randomized to (in theory) never reuse an existing socket.
- First, get the heartbeating working, and only *then* add in the rest of the message flow. You should be able to prove the heartbeating works by starting peers in any order, stopping and restarting them, simulating freezes, and so on.
- When your main loop is based on zmq_poll(3), use a secondary timer to trigger heartbeats. Do *not* use the poll loop for this, because it will either send too many heartbeats (overloading the network), or too few (causing peers to disconnect). The zhelpers package provides an s_clock() method that returns the current system clock in milliseconds. It's easy to use this to calculate when to send the next heartbeats. Thus, in C:

```
    //  Send out heartbeats at regular intervals
    uint64_t heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
    while (1) {
        …
        zmq_poll (items, 1, HEARTBEAT_INTERVAL * 1000);
        …
        //  Do this unconditionally, whatever zmq_poll did
        if (s_clock () > heartbeat_at) {
            … Send heartbeats to all peers that expect them
            //  Set timer for next heartbeat
            heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
            }
        }
```

- Your main poll loop should use the heartbeat interval as its timeout. Obviously, don't use infinity. Anything less will just waste cycles.
- Use simple tracing, i.e. print to console, to get this working. Some tricks to help you trace the flow of messages between peers: a dump method such as zmsg offers; number messages incrementally so you can see if there are gaps.
- In a real application, heartbeating must be configurable and usually negotiated with the peer. Some peers will want aggressive heartbeating, as low as 10 msecs. Other peers will be far away and want heartbeating as high as 30 seconds.
- If you have different heartbeat intervals for different peers, your poll timeout should be the lowest of these.
- You might be tempted to open a separate socket dialog for heartbeats. This is superficially nice because you can separate different dialogs, e.g. the synchronous request-reply from the asynchronous heartbeating. However it's a bad idea for several reasons. First, if you're sending data you don't need to send heartbeats. Second, sockets may, due to network vagaries, become jammed. You need to know when your main data socket is silent because it's dead, rather than just not busy, so you need heartbeats on that socket. Lastly, two sockets is more complex than one.
- We're not doing heartbeating from client to queue. We could, but it would add *significant* complexity for no real benefit.

## Contracts and Protocols

top prev next

If you're paying attention you'll realize that Paranoid Pirate is not compatible with Simple Pirate, because of the heartbeats.

In fact what we have here is a protocol that needs writing down. It's fun to experiment without specifications, but that's not a sensible basis for real applications. What happens if we want to write a worker in another language? Do we have to read code to see how things work? What if we want to change the protocol for some reason? The protocol may be simple but it's not obvious, and if it's successful it'll become more complex.

Lack of contracts is a sure sign of a disposable application. So, let's write a contract for this protocol. How do we do that?

- There's a wiki, at rfc.zeromq.org, that we made especially as a home for public ØMQ contracts.
- To create a new specification, register, and follow the instructions. It's straight-forward, though technical writing is not for everyone.

It took me about fifteen minutes to draft the new Pirate Pattern Protocol. It's not a big specification but it does capture enough to act as the basis for arguments ("your queue isn't PPP compatible, please fix it!").

Turning PPP into a real protocol would take more work:

- There should be a protocol version number in the READY command so that it's possible to create new versions of PPP safely.
- Right now, READY and HEARTBEAT are not entirely distinct from requests and replies. To make them distinct, we would want a message structure that includes a "message type" part.

## Service-Oriented Reliable Queuing (Majordomo Pattern)

top prev next

The nice thing about progress is how fast it happens when lawyers and committees aren't involved. Just a few sentences ago we were dreaming of a better protocol that would fix the world. And here we have it:

- http://rfc.zeromq.org/spec:7

This one-page specification takes PPP and turns it into something more solid. This is how we should design complex architectures: start by writing down the contracts, and only *then* write software to implement them.

The Majordomo Protocol (MDP) extends and improves PPP in one interesting way apart from the two points above. It adds a "service name" to requests that the client sends, and asks workers to register for specific services. The nice thing about MDP is that it came from working code, a simpler protocol, and a precise set of improvements. This made it easy to draft.

Adding service names is a small but significant change that turns our Paranoid Pirate queue into a service-oriented broker:

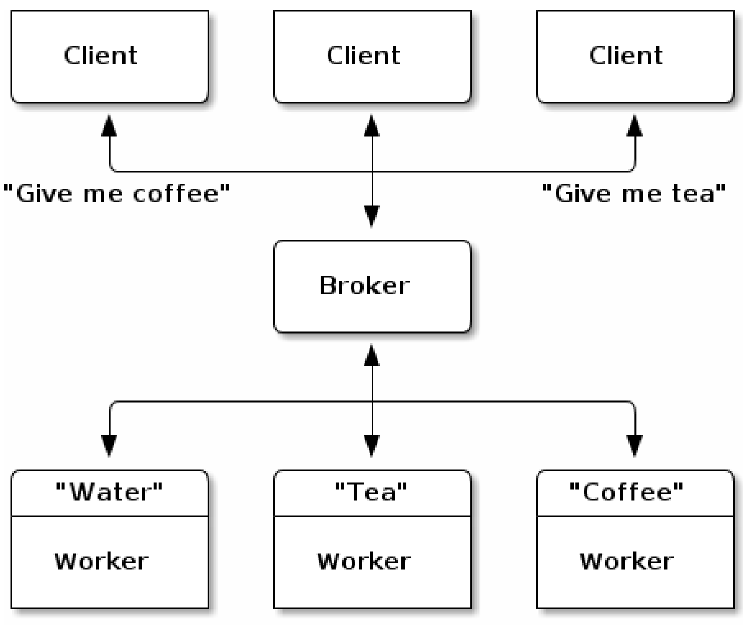


Figure 60 – Majordomo Pattern

To implement Majordomo we need to write a framework for clients and workers. It's really not sane to ask every application developer to read the spec and make it work, when they could be using a simpler API built and tested just once.

So, while our first contract (MDP itself) defines how the pieces of our distributed architecture talk to each other, our second contract defines how user applications talk to the technical framework we're going to design.

Majordomo has two halves, a client side and a worker side. Since we'll write both client and worker applications, we will need two APIs. Here is a sketch for the client API, using a simple object-oriented approach. We write this in C, using the style of the ZFL library:

```
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);
```

That's it. We open a session to the broker, we send a request message and get a reply message back, and we eventually close the connection. Here's a sketch for the worker API:

```
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);
```

It's more or less symmetrical but the worker dialog is a little different. The first time a worker does a recv(), it passes a null reply, thereafter it passes the current reply, and gets a new request.

The client and worker APIs were fairly simple to construct, since they're heavily based on the Paranoid Pirate code we already developed. Here is the client API:

```
/*
=====================================================================
    mdcliapi.c

    Majordomo Protocol Client API
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    ---------------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.

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

#include "mdcliapi.h"

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

```
struct _mdcli_t {
    zctx_t *ctx;                // Our context
    char *broker;
    void *client;               // Socket to broker
    int verbose;                // Print activity to stdout
    int timeout;                // Request timeout
    int retries;                // Request retries
};

// ---------------------------------------------------------------------
----
// Connect or reconnect to broker

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_REQ);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s…", self->broker);
}

// ---------------------------------------------------------------------
----
// Constructor

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           // msecs
    self->retries = 3;              // Before we abandon

    s_mdcli_connect_to_broker (self);
    return self;
}

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

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  ------------------------------------------------------------------
----
//  Set request timeout

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}

//  ------------------------------------------------------------------
----
//  Set request retries

void
mdcli_set_retries (mdcli_t *self, int retries)
{
    assert (self);
    self->retries = retries;
}

//  ------------------------------------------------------------------
----
//  Send request to broker and get reply by hook or crook
//  Takes ownership of request message and destroys it when sent.
//  Returns the reply message or NULL if there was no reply.

zmsg_t *
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  Prefix request with protocol frames
    //  Frame 1: "MDPCxy" (six bytes, MDP/Client x.y)
    //  Frame 2: Service name (printable string)
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    if (self->verbose) {
        zclock_log ("I: send request to '%s' service:", service);
        zmsg_dump (request);
    }

    int retries_left = self->retries;
    while (retries_left && !zctx_interrupted) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->client);

        while (TRUE) {
            //  Poll socket for a reply, with timeout
            zmq_pollitem_t items [] = {
                { self->client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, self->timeout *
ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  Interrupted

            //  If we got a reply, process it
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (self->client);
                if (self->verbose) {
                    zclock_log ("I: received reply:");
                    zmsg_dump (msg);
                }
                //  Don't try to handle errors, just assert noisily
                assert (zmsg_size (msg) >= 3);

                zframe_t *header = zmsg_pop (msg);
                assert (zframe_streq (header, MDPC_CLIENT));
                zframe_destroy (&header);

                zframe_t *reply_service = zmsg_pop (msg);
                assert (zframe_streq (reply_service, service));
                zframe_destroy (&reply_service);

                zmsg_destroy (&request);
                return msg;     //  Success
            }
            else
            if (--retries_left) {
                if (self->verbose)
                    zclock_log ("W: no reply, reconnecting…");
                //  Reconnect, and resend message
                s_mdcli_connect_to_broker (self);
                zmsg_t *msg = zmsg_dup (request);
                zmsg_send (&msg, self->client);
            }
            else {
                if (self->verbose)
                    zclock_log ("W: permanent error, abandoning");
                break;          //  Give up
            }
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing client…\n");
    zmsg_destroy (&request);
    return NULL;
}
```

*mdcliapi.c: Majordomo client API*

With an example test program that does 100K request-reply cycles:

```
//
//  Majordomo Protocol client example
//  Uses the mdcli API to hide all MDP aspects
//

//  Lets us build this source without creating a library
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        zmsg_t *reply = mdcli_send (session, "echo", &request);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  Interrupt or failure
    }
    printf ("%d requests/replies processed\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

*mdclient.c: Majordomo client application*

And here is the worker API:

```
/*
=====================================================================
    mdwrkapi.c

    Majordomo Protocol Worker API
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    -----------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.

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

#include "mdwrkapi.h"

//  Reliability parameters
#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable

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

struct _mdwrk_t {
    zctx_t *ctx;                //  Our context
    char *broker;
    char *service;
    void *worker;               //  Socket to broker
    int verbose;                //  Print activity to stdout

    //  Heartbeat management
    uint64_t heartbeat_at;      //  When to send HEARTBEAT
    size_t liveness;            //  How many attempts left
    int heartbeat;              //  Heartbeat delay, msecs
    int reconnect;              //  Reconnect delay, msecs

    //  Internal state
    int expect_reply;           //  Zero only at start

    //  Return address, if any
    zframe_t *reply_to;
};

//  ---------------------------------------------------------------------
----
//  Send message to broker
//  If no msg is provided, creates one internally

static void
s_mdwrk_send_to_broker (mdwrk_t *self, char *command, char *option,
                        zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  Stack protocol envelope to start of message
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);
    zmsg_pushstr (msg, "");

    if (self->verbose) {
        zclock_log ("I: sending %s to broker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->worker);
}

//  ---------------------------------------------------------------------
----
//  Connect or reconnect to broker

void s_mdwrk_connect_to_broker (mdwrk_t *self)
{
    if (self->worker)
        zsocket_destroy (self->ctx, self->worker);
    self->worker = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->worker, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s…", self->broker);
    //  Register service with broker
    s_mdwrk_send_to_broker (self, MDPW_READY, self->service, NULL);

    //  If liveness hits zero, queue is considered disconnected
    self->liveness = HEARTBEAT_LIVENESS;
    self->heartbeat_at = zclock_time () + self->heartbeat;
}

//  -----------------------------------------------------------------
----
//  Constructor

mdwrk_t *
mdwrk_new (char *broker,char *service, int verbose)
{
    assert (broker);
    assert (service);

    mdwrk_t *self = (mdwrk_t *) zmalloc (sizeof (mdwrk_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->service = strdup (service);
    self->verbose = verbose;
    self->heartbeat = 2500;     //  msecs
    self->reconnect = 2500;     //  msecs

    s_mdwrk_connect_to_broker (self);
    return self;
}

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

void
mdwrk_destroy (mdwrk_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdwrk_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self->service);
        free (self);
        *self_p = NULL;
    }
}

//  -----------------------------------------------------------------
----
//  Set heartbeat delay

void
mdwrk_set_heartbeat (mdwrk_t *self, int heartbeat)
{
    self->heartbeat = heartbeat;
}

//  -----------------------------------------------------------------
----
//  Set reconnect delay
```

```
void
mdwrk_set_reconnect (mdwrk_t *self, int reconnect)
{
    self->reconnect = reconnect;
}

//  --------------------------------------------------------------------
----
//  Send reply, if any, to broker and wait for next request.

zmsg_t *
mdwrk_recv (mdwrk_t *self, zmsg_t **reply_p)
{
    //  Format and send the reply if we were provided one
    assert (reply_p);
    zmsg_t *reply = *reply_p;
    assert (reply || !self->expect_reply);
    if (reply) {
        assert (self->reply_to);
        zmsg_wrap (reply, self->reply_to);
        s_mdwrk_send_to_broker (self, MDPW_REPLY, NULL, reply);
        zmsg_destroy (reply_p);
    }
    self->expect_reply = 1;

    while (TRUE) {
        zmq_pollitem_t items [] = {
            { self->worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, self->heartbeat *
ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->worker);
            if (!msg)
                break;          //  Interrupted
            if (self->verbose) {
                zclock_log ("I: received message from broker:");
                zmsg_dump (msg);
            }
            self->liveness = HEARTBEAT_LIVENESS;

            //  Don't try to handle errors, just assert noisily
            assert (zmsg_size (msg) >= 3);

            zframe_t *empty = zmsg_pop (msg);
            assert (zframe_streq (empty, ""));
            zframe_destroy (&empty);

            zframe_t *header = zmsg_pop (msg);
            assert (zframe_streq (header, MDPW_WORKER));
            zframe_destroy (&header);

            zframe_t *command = zmsg_pop (msg);
            if (zframe_streq (command, MDPW_REQUEST)) {
                //  We should pop and save as many addresses as there
are
                //  up to a null part, but for now, just save one…
                self->reply_to = zmsg_unwrap (msg);
                zframe_destroy (&command);
                return msg;     //  We have a request to process
            }
            else
            if (zframe_streq (command, MDPW_HEARTBEAT))
                ;               //  Do nothing for heartbeats
            else
            if (zframe_streq (command, MDPW_DISCONNECT))
                s_mdwrk_connect_to_broker (self);
            else {
                zclock_log ("E: invalid input message");
                zmsg_dump (msg);
            }
            zframe_destroy (&command);
            zmsg_destroy (&msg);
        }
        else
        if (--self->liveness == 0) {
            if (self->verbose)
                zclock_log ("W: disconnected from broker -
retrying…");
            zclock_sleep (self->reconnect);
            s_mdwrk_connect_to_broker (self);
        }
        //  Send HEARTBEAT if it's time
        if (zclock_time () > self->heartbeat_at) {
            s_mdwrk_send_to_broker (self, MDPW_HEARTBEAT, NULL,
NULL);
            self->heartbeat_at = zclock_time () + self->heartbeat;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing worker…\n");
    return NULL;
}
```

*mdwrkapi.c: Majordomo worker API*

With an example test program that implements an 'echo' service:

```
//
//  Majordomo Protocol worker example
//  Uses the mdwrk API to hide all MDP aspects
//

//  Lets us build this source without creating a library
#include "mdwrkapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdwrk_t *session = mdwrk_new (
        "tcp://localhost:5555", "echo", verbose);

    zmsg_t *reply = NULL;
    while (1) {
        zmsg_t *request = mdwrk_recv (session, &reply);
        if (request == NULL)
```

```
            break;              //  Worker was interrupted
        reply = request;        //  Echo is complex… :-)
    }
    mdwrk_destroy (&session);
    return 0;
}
```

*mdworker.c: Majordomo worker application*

Notes on this code:

- The APIs are single threaded. This means, for example, that the worker won't send heartbeats in the background. Happily, this is exactly what we want: if the worker application gets stuck, heartbeats will stop and the broker will stop sending requests to the worker.
- The worker API doesn't do an exponential backoff, it's not worth the extra complexity.
- The APIs don't do any error reporting. If something isn't as expected, they raise an assertion (or exception depending on the language). This is ideal for a reference implementation, so any protocol errors show immediately. For real applications the API should be robust against invalid messages.

Let's design the Majordomo broker. Its core structure is a set of queues, one per service. We will create these queues as workers appear (we could delete them as workers disappear but forget that for now, it gets complex). Additionally, we keep a queue of workers per service.

To make the C examples easier to write and read, I've taken the hash and list container classes from the ZFL project, and renamed them as [zlist and zhash, as we did with zmsg. In any modern language you can of course use built-in containers.

And here is the broker:

```
//
//  Majordomo Protocol broker
//  A minimal implementation of http://rfc.zeromq.org/spec:7 and
spec:8
//
#include "czmq.h"
#include "mdp.h"

//  We'd normally pull these from config data

#define HEARTBEAT_LIVENESS  3       //  3-5 is reasonable
#define HEARTBEAT_INTERVAL  2500    //  msecs
#define HEARTBEAT_EXPIRY    HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS

//  This defines a single broker
typedef struct {
    zctx_t *ctx;                //  Our context
    void *socket;               //  Socket for clients & workers
    int verbose;                //  Print activity to stdout
    char *endpoint;             //  Broker binds to this endpoint
    zhash_t *services;          //  Hash of known services
    zhash_t *workers;           //  Hash of known workers
    zlist_t *waiting;           //  List of waiting workers
    uint64_t heartbeat_at;      //  When to send HEARTBEAT
} broker_t;

//  This defines a single service
```

```
typedef struct {
    char *name;                 //  Service name
    zlist_t *requests;          //  List of client requests
    zlist_t *waiting;           //  List of waiting workers
    size_t workers;             //  How many workers we have
} service_t;

//  This defines one worker, idle or active
typedef struct {
    char *identity;             //  Identity of worker
    zframe_t *address;          //  Address frame to route to
    service_t *service;         //  Owning service, if known
    int64_t expiry;             //  Expires at unless heartbeat
} worker_t;

//  ---------------------------------------------------------------------
----
//  Broker functions
static broker_t *
    s_broker_new (int verbose);
static void
    s_broker_destroy (broker_t **self_p);
static void
    s_broker_bind (broker_t *self, char *endpoint);
static void
    s_broker_purge_workers (broker_t *self);

//  Service functions
static service_t *
    s_service_require (broker_t *self, zframe_t *service_frame);
static void
    s_service_destroy (void *argument);
static void
    s_service_dispatch (broker_t *self, service_t *service, zmsg_t
*msg);
static void
    s_service_internal (broker_t *self, zframe_t *service_frame,
zmsg_t *msg);

//  Worker functions
static worker_t *
    s_worker_require (broker_t *self, zframe_t *address);
static void
    s_worker_delete (broker_t *self, worker_t *worker, int
disconnect);
static void
    s_worker_destroy (void *argument);
static void
    s_worker_process (broker_t *self, zframe_t *sender, zmsg_t *msg);
static void
    s_worker_send (broker_t *self, worker_t *worker, char *command,
                   char *option, zmsg_t *msg);
static void
    s_worker_waiting (broker_t *self, worker_t *worker);

//  Client functions
static void
    s_client_process (broker_t *self, zframe_t *sender, zmsg_t *msg);

//  ---------------------------------------------------------------------
```

```
----
//  Main broker work happens here

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    broker_t *self = s_broker_new (verbose);
    s_broker_bind (self, "tcp://*:5555");

    //  Get and process messages forever or until interrupted
    while (TRUE) {
        zmq_pollitem_t items [] = {
            { self->socket,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL *
ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Interrupted

        //  Process next input message, if any
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->socket);
            if (!msg)
                break;          //  Interrupted
            if (self->verbose) {
                zclock_log ("I: received message:");
                zmsg_dump (msg);
            }
            zframe_t *sender = zmsg_pop (msg);
            zframe_t *empty  = zmsg_pop (msg);
            zframe_t *header = zmsg_pop (msg);

            if (zframe_streq (header, MDPC_CLIENT))
                s_client_process (self, sender, msg);
            else
            if (zframe_streq (header, MDPW_WORKER))
                s_worker_process (self, sender, msg);
            else {
                zclock_log ("E: invalid message:");
                zmsg_dump (msg);
                zmsg_destroy (&msg);
            }
            zframe_destroy (&sender);
            zframe_destroy (&empty);
            zframe_destroy (&header);
        }
        //  Disconnect and delete any expired workers
        //  Send heartbeats to idle workers if needed
        if (zclock_time () > self->heartbeat_at) {
            s_broker_purge_workers (self);
            worker_t *worker = (worker_t *) zlist_first (self-
>waiting);
            while (worker) {
                s_worker_send (self, worker, MDPW_HEARTBEAT, NULL,
NULL);
                worker = (worker_t *) zlist_next (self->waiting);
            }
            self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, shutting down…\n");

    s_broker_destroy (&self);
    return 0;
}

//  ---------------------------------------------------------------------
----
//  Constructor for broker object

static broker_t *
s_broker_new (int verbose)
{
    broker_t *self = (broker_t *) zmalloc (sizeof (broker_t));

    //  Initialize broker state
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->verbose = verbose;
    self->services = zhash_new ();
    self->workers = zhash_new ();
    self->waiting = zlist_new ();
    self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
    return self;
}

//  ---------------------------------------------------------------------
----
//  Destructor for broker object

static void
s_broker_destroy (broker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        broker_t *self = *self_p;
        zctx_destroy (&self->ctx);
        zhash_destroy (&self->services);
        zhash_destroy (&self->workers);
        zlist_destroy (&self->waiting);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
----
//  Bind broker to endpoint, can call this multiple times
//  We use a single socket for both clients and workers.

void
s_broker_bind (broker_t *self, char *endpoint)
{
    zsocket_bind (self->socket, endpoint);
    zclock_log ("I: MDP broker/0.1.1 is active at %s", endpoint);
}

//  ---------------------------------------------------------------------
----
//  Delete any idle workers that haven't pinged us in a while.
```

```
Workers
//  are oldest to most recent, so we stop at the first alive worker.

static void
s_broker_purge_workers (broker_t *self)
{
    worker_t *worker = (worker_t *) zlist_first (self->waiting);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              //  Worker is alive, we're done here
        if (self->verbose)
            zclock_log ("I: deleting expired worker: %s",
                worker->identity);

        s_worker_delete (self, worker, 0);
        worker = (worker_t *) zlist_first (self->waiting);
    }
}

//  ------------------------------------------------------------------
----
//  Locate or create new service entry

static service_t *
s_service_require (broker_t *self, zframe_t *service_frame)
{
    assert (service_frame);
    char *name = zframe_strdup (service_frame);

    service_t *service =
        (service_t *) zhash_lookup (self->services, name);
    if (service == NULL) {
        service = (service_t *) zmalloc (sizeof (service_t));
        service->name = name;
        service->requests = zlist_new ();
        service->waiting = zlist_new ();
        zhash_insert (self->services, name, service);
        zhash_freefn (self->services, name, s_service_destroy);
        if (self->verbose)
            zclock_log ("I: received message:");
    }
    else
        free (name);

    return service;
}

//  ------------------------------------------------------------------
----
//  Destroy service object, called when service is removed from
//  broker->services.

static void
s_service_destroy (void *argument)
{
    service_t *service = (service_t *) argument;
    //  Destroy all queued requests
    while (zlist_size (service->requests)) {
        zmsg_t *msg = zlist_pop (service->requests);
        zmsg_destroy (&msg);
    }
    zlist_destroy (&service->requests);
    zlist_destroy (&service->waiting);
    free (service->name);
    free (service);
}

//  ------------------------------------------------------------------
----
//  Dispatch requests to waiting workers as possible

static void
s_service_dispatch (broker_t *self, service_t *service, zmsg_t *msg)
{
    assert (service);
    if (msg)                    //  Queue message if any
        zlist_append (service->requests, msg);

    s_broker_purge_workers (self);
    while (zlist_size (service->waiting)
        && zlist_size (service->requests))
    {
        worker_t *worker = zlist_pop (service->waiting);
        zlist_remove (self->waiting, worker);
        zmsg_t *msg = zlist_pop (service->requests);
        s_worker_send (self, worker, MDPW_REQUEST, NULL, msg);
        zmsg_destroy (&msg);
    }
}

//  ------------------------------------------------------------------
----
//  Handle internal service according to 8/MMI specification

static void
s_service_internal (broker_t *self, zframe_t *service_frame, zmsg_t
*msg)
{
    char *return_code;
    if (zframe_streq (service_frame, "mmi.service")) {
        char *name = zframe_strdup (zmsg_last (msg));
        service_t *service =
            (service_t *) zhash_lookup (self->services, name);
        return_code = service && service->workers? "200": "404";
        free (name);
    }
    else
        return_code = "501";

    zframe_reset (zmsg_last (msg), return_code, strlen
(return_code));

    //  Remove & save client return envelope and insert the
    //  protocol header and service name, then rewrap envelope.
    zframe_t *client = zmsg_unwrap (msg);
    zmsg_push (msg, zframe_dup (service_frame));
    zmsg_pushstr (msg, MDPC_CLIENT);
    zmsg_wrap (msg, client);
    zmsg_send (&msg, self->socket);
}
```

```
//  ---------------------------------------------------------------------
----
//  Creates worker if necessary

static worker_t *
s_worker_require (broker_t *self, zframe_t *address)
{
    assert (address);

    //  self->workers is keyed off worker identity
    char *identity = zframe_strhex (address);
    worker_t *worker =
        (worker_t *) zhash_lookup (self->workers, identity);

    if (worker == NULL) {
        worker = (worker_t *) zmalloc (sizeof (worker_t));
        worker->identity = identity;
        worker->address = zframe_dup (address);
        zhash_insert (self->workers, identity, worker);
        zhash_freefn (self->workers, identity, s_worker_destroy);
        if (self->verbose)
            zclock_log ("I: registering new worker: %s", identity);
    }
    else
        free (identity);
    return worker;
}

//  ---------------------------------------------------------------------
----
//  Deletes worker from all data structures, and destroys worker

static void
s_worker_delete (broker_t *self, worker_t *worker, int disconnect)
{
    assert (worker);
    if (disconnect)
        s_worker_send (self, worker, MDPW_DISCONNECT, NULL, NULL);

    if (worker->service) {
        zlist_remove (worker->service->waiting, worker);
        worker->service->workers--;
    }
    zlist_remove (self->waiting, worker);
    //  This implicitly calls s_worker_destroy
    zhash_delete (self->workers, worker->identity);
}

//  ---------------------------------------------------------------------
----
//  Destroy worker object, called when worker is removed from
//  broker->workers.

static void
s_worker_destroy (void *argument)
{
    worker_t *worker = (worker_t *) argument;
    zframe_destroy (&worker->address);
    free (worker->identity);
    free (worker);
```

```
}

//  ---------------------------------------------------------------
----
//  Process message sent to us by a worker

static void
s_worker_process (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 1);      //  At least, command

    zframe_t *command = zmsg_pop (msg);
    char *identity = zframe_strhex (sender);
    int worker_ready = (zhash_lookup (self->workers, identity) !=
NULL);
    free (identity);
    worker_t *worker = s_worker_require (self, sender);

    if (zframe_streq (command, MDPW_READY)) {
        if (worker_ready)               //  Not first command in
session
            s_worker_delete (self, worker, 1);
        else
        if (zframe_size (sender) >= 4  //  Reserved service name
        &&  memcmp (zframe_data (sender), "mmi.", 4) == 0)
            s_worker_delete (self, worker, 1);
        else {
            //  Attach worker to service and mark as idle
            zframe_t *service_frame = zmsg_pop (msg);
            worker->service = s_service_require (self,
service_frame);
            worker->service->workers++;
            s_worker_waiting (self, worker);
            zframe_destroy (&service_frame);
        }
    }
    else
    if (zframe_streq (command, MDPW_REPLY)) {
        if (worker_ready) {
            //  Remove & save client return envelope and insert the
            //  protocol header and service name, then rewrap
envelope.
            zframe_t *client = zmsg_unwrap (msg);
            zmsg_pushstr (msg, worker->service->name);
            zmsg_pushstr (msg, MDPC_CLIENT);
            zmsg_wrap (msg, client);
            zmsg_send (&msg, self->socket);
            s_worker_waiting (self, worker);
        }
        else
            s_worker_delete (self, worker, 1);
    }
    else
    if (zframe_streq (command, MDPW_HEARTBEAT)) {
        if (worker_ready)
            worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
        else
            s_worker_delete (self, worker, 1);
    }
    else
    if (zframe_streq (command, MDPW_DISCONNECT))
        s_worker_delete (self, worker, 0);
    else {
        zclock_log ("E: invalid input message");
        zmsg_dump (msg);
    }
    free (command);
    zmsg_destroy (&msg);
}

//  -----------------------------------------------------------------
----
//  Send message to worker
//  If pointer to message is provided, sends that message. Does not
//  destroy the message, this is the caller's job.

static void
s_worker_send (broker_t *self, worker_t *worker, char *command,
               char *option, zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  Stack protocol envelope to start of message
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);

    //  Stack routing envelope to start of message
    zmsg_wrap (msg, zframe_dup (worker->address));

    if (self->verbose) {
        zclock_log ("I: sending %s to worker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->socket);
}

//  -----------------------------------------------------------------
----
//  This worker is now waiting for work

static void
s_worker_waiting (broker_t *self, worker_t *worker)
{
    //  Queue to broker and service waiting lists
    zlist_append (self->waiting, worker);
    zlist_append (worker->service->waiting, worker);
    worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
    s_service_dispatch (self, worker->service, NULL);
}

//  -----------------------------------------------------------------
----
//  Process a request coming from a client

static void
s_client_process (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 2);     //  Service name + body
    zframe_t *service_frame = zmsg_pop (msg);
    service_t *service = s_service_require (self, service_frame);

    //  Set reply return address to client sender
    zmsg_wrap (msg, zframe_dup (sender));
    if (zframe_size (service_frame) >= 4
    &&  memcmp (zframe_data (service_frame), "mmi.", 4) == 0)
        s_service_internal (self, service_frame, msg);
    else
        s_service_dispatch (self, service, msg);
    zframe_destroy (&service_frame);
}
```

*mdbroker.c: Majordomo broker*

This is by far the most complex example we've seen. It's almost 500 lines of code. To write this, and make it fully robust took two days. However this is still a short piece of code for a full service-oriented broker.

Notes on this code:

- The Majordomo Protocol lets us handle both clients and workers on a single socket. This is nicer for those deploying and managing the broker: it just sits on one ØMQ endpoint rather than the two that most devices need.
- The broker implements all of MDP/0.1 properly (as far as I know), including disconnection if the broker sends invalid commands, heartbeating, and the rest.
- It can be extended to run multiple threads, each managing one socket and one set of clients and workers. This could be interesting for segmenting large architectures. The C code is already organized around a broker class to make this trivial.
- A primary-failover or live-live broker reliability model is easy, since the broker essentially has no state except service presence. It's up to clients and workers to choose another broker if their first choice isn't up and running.
- The examples use 5-second heartbeats, mainly to reduce the amount of output when you enable tracing. Realistic values would be lower for most LAN applications. However, any retry has to be slow enough to allow for a service to restart, say 10 seconds at least.

## Asynchronous Majordomo Pattern

top prev next

The way we implemented Majordomo, above, is simple and stupid. The client is just the original Simple Pirate, wrapped up in a sexy API. When I fire up a client, broker, and worker on a test box, it can process 100,000 requests in about 14 seconds. That is partly due to the code, which cheerfully copies message frames around as if CPU cycles were free. But the real problem is that we're 'round-tripping'. ØMQ disables [http://en.wikipedia.org/wiki/Nagle's_algorithm], but round-tripping is still slow.

Theory is great in theory, but in practice, practice is better. Let's measure the cost of round-tripping with a simple test program. This sends a bunch of messages, first waiting for a reply to each message, and second as a batch, reading all the replies back as a batch. Both approaches do the same work, but they give very different results. We mockup a client, broker, and worker:

```
//
```

```
//  Round-trip demonstrator
//
//  While this example runs in a single process, that is just to make
//  it easier to start and stop the example. Client thread signals to
//  main when it's ready.
//
#include "czmq.h"

static void
client_task (void *args, zctx_t *ctx, void *pipe)
{
    void *client = zsocket_new (ctx, ZMQ_DEALER);
    zmq_setsockopt (client, ZMQ_IDENTITY, "C", 1);
    zsocket_connect (client, "tcp://localhost:5555");

    printf ("Setting up test…\n");
    zclock_sleep (100);

    int requests;
    int64_t start;

    printf ("Synchronous round-trip test…\n");
    start = zclock_time ();
    for (requests = 0; requests < 10000; requests++) {
        zstr_send (client, "hello");
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d calls/second\n",
        (1000 * 10000) / (int) (zclock_time () - start));

    printf ("Asynchronous round-trip test…\n");
    start = zclock_time ();
    for (requests = 0; requests < 100000; requests++)
        zstr_send (client, "hello");
    for (requests = 0; requests < 100000; requests++) {
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d calls/second\n",
        (1000 * 100000) / (int) (zclock_time () - start));

    zstr_send (pipe, "done");
}

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "W", 1);
    zsocket_connect (worker, "tcp://localhost:5556");

    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}
```

```
static void *
broker_task (void *args)
{
    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");
    zsocket_bind (backend,  "tcp://*:5556");

    //  Initialize poll set
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    while (1) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1)
            break;              //  Interrupted
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (frontend);
            zframe_t *address = zmsg_pop (msg);
            zframe_destroy (&address);
            zmsg_pushstr (msg, "W");
            zmsg_send (&msg, backend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (backend);
            zframe_t *address = zmsg_pop (msg);
            zframe_destroy (&address);
            zmsg_pushstr (msg, "C");
            zmsg_send (&msg, frontend);
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (void)
{
    //  Create threads
    zctx_t *ctx = zctx_new ();
    void *client = zthread_fork (ctx, client_task, NULL);
    zthread_new (ctx, worker_task, NULL);
    zthread_new (ctx, broker_task, NULL);

    //  Wait for signal on client pipe
    char *signal = zstr_recv (client);
    free (signal);

    zctx_destroy (&ctx);
    return 0;
}
```

*tripping.c: Round-trip demonstrator*

On my development box, this program says:

```
Setting up test...
```

```
Synchronous round-trip test...
 9057 calls/second
Asynchronous round-trip test...
 173010 calls/second
```

Note that the client thread does a small pause before starting. This is to get around one of the 'features' of the router socket: if you send a message with the address of a peer that's not yet connected, the message gets discarded. In this example we don't use the LRU mechanism, so without the sleep, if the worker thread is too slow to connect, it'll lose messages, making a mess of our test.

As we see, round-tripping in the simplest case is 20 times slower than "shove it down the pipe as fast as it'll go" asynchronous approach. Let's see if we can apply this to Majordomo.

First, let's modify the client API to have separate send and recv methods:

```
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);
```

It's literally a few minutes' work to refactor the synchronous client API to become asynchronous:

```
/*
=====================================================================
    mdcliapi2.c

    Majordomo Protocol Client API (async version)
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    -----------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.
```

```
======================================================================
*/

#include "mdcliapi2.h"

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

struct _mdcli_t {
    zctx_t *ctx;                //  Our context
    char *broker;
    void *client;               //  Socket to broker
    int verbose;                //  Print activity to stdout
    int timeout;                //  Request timeout
};

//  ------------------------------------------------------------------
----
//  Connect or reconnect to broker

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: connecting to broker at %s…", self->broker);
}

//  ------------------------------------------------------------------
----
//  Constructor

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  msecs

    s_mdcli_connect_to_broker (self);
    return self;
}

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

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
----
//  Set request timeout

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}

//  ---------------------------------------------------------------------
----
//  Send request to broker
//  Takes ownership of request message and destroys it when sent.

int
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  Prefix request with protocol frames
    //  Frame 0: empty (REQ emulation)
    //  Frame 1: "MDPCxy" (six bytes, MDP/Client x.y)
    //  Frame 2: Service name (printable string)
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    zmsg_pushstr (request, "");
    if (self->verbose) {
        zclock_log ("I: send request to '%s' service:", service);
        zmsg_dump (request);
    }
    zmsg_send (&request, self->client);
    return 0;
}

//  ---------------------------------------------------------------------
----
//  Returns the reply message or NULL if there was no reply. Does not
//  attempt to recover from a broker failure, this is not possible
//  without storing all unanswered requests and resending them all…

zmsg_t *
mdcli_recv (mdcli_t *self)
{
    assert (self);

    //  Poll socket for a reply, with timeout
    zmq_pollitem_t items [] = { { self->client, 0, ZMQ_POLLIN, 0 } };
    int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
    if (rc == -1)
        return NULL;            //  Interrupted

    //  If we got a reply, process it
    if (items [0].revents & ZMQ_POLLIN) {
        zmsg_t *msg = zmsg_recv (self->client);
        if (self->verbose) {
            zclock_log ("I: received reply:");
            zmsg_dump (msg);
        }
        //  Don't try to handle errors, just assert noisily
        assert (zmsg_size (msg) >= 4);

        zframe_t *empty = zmsg_pop (msg);
        assert (zframe_streq (empty, ""));
        zframe_destroy (&empty);

        zframe_t *header = zmsg_pop (msg);
        assert (zframe_streq (header, MDPC_CLIENT));
        zframe_destroy (&header);

        zframe_t *service = zmsg_pop (msg);
        zframe_destroy (&service);

        return msg;     //  Success
    }
    if (zctx_interrupted)
        printf ("W: interrupt received, killing client…\n");
    else
    if (self->verbose)
        zclock_log ("W: permanent error, abandoning request");

    return NULL;
}
```

*mdcliapi2.c: Majordomo asynchronous client API*

And here's the corresponding client test program:

```
//
//  Majordomo Protocol client example - asynchronous
//  Uses the mdcli API to hide all MDP aspects
//
//  Lets us build this source without creating a library
#include "mdcliapi2.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        mdcli_send (session, "echo", &request);
    }
    for (count = 0; count < 100000; count++) {
        zmsg_t *reply = mdcli_recv (session);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  Interrupted by Ctrl-C
    }
    printf ("%d replies received\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

*mdclient2.c: Majordomo client application*

The broker and worker are unchanged, since we've not modified the protocol at all. We see an immediate improvement in performance. Here's the synchronous client chugging through 100K request-reply cycles:

```
$ time mdclient
100000 requests/replies processed

real    0m14.088s
user    0m1.310s
sys     0m2.670s
```

And here's the asynchronous client, with a single worker:

```
$ time mdclient2
100000 replies received

real    0m8.730s
user    0m0.920s
sys     0m1.550s
```

Twice as fast. Not bad, but let's fire up 10 workers, and see how it handles:

```
$ time mdclient2
100000 replies received

real    0m3.863s
user    0m0.730s
sys     0m0.470s
```

It isn't fully asynchronous since workers get their messages on a strict LRU basis. But it will scale better with more workers. On my fast test box, after eight or so workers it doesn't get any faster. Four cores only stretches so far. But we got a 4x improvement in throughput with just a few minutes' work. The broker is still unoptimized. It spends most of its time copying message frames around, instead of doing zero copy, which it could. But we're getting 25K reliable request/reply calls a second, with pretty low effort.

However the asynchronous Majordomo pattern isn't all roses. It has a fundamental weakness, namely that it cannot survive a broker crash without fmore work. If you look at the mdcliapi2 code you'll see it does not attempt to reconnect after a failure. A proper reconnect would require:

- That every request is numbered, and every reply has a matching number, which would ideally require a change to the protocol to enforce.
- That the client API tracks and holds onto all outstanding requests, i.e. for which no reply had yet been received.
- That in case of failover, the client API *resends* all outstanding requests to the broker.

It's not a deal breaker but it does show that performance often means complexity. Is this worth doing for Majordomo? It depends on your use case. For a name lookup service you call once per session, no. For a web front-end serving thousands of clients, probably yes.

## Service Discovery

top prev next

So, we have a nice service-oriented broker, but we have no way of knowing whether a particular service is available or not. We know if a request failed, but we don't know why. It is useful to be able to ask the broker, "is the echo service running?" The most obvious way would be to modify our MDP/Client protocol to add commands to ask the broker, "is service X running?" But MDP/Client has the great charm of being simple. Adding service discovery to it would make it as complex as the MDP/Worker protocol.

An other option is to do what email does, and ask that undeliverable requests be returned. This can work well in an asynchronous world but it also adds complexity. We need ways to distinguish a returned requests from a replies, and to handle these properly.

Let's try to use what we've already built, building on top of MDP instead of modifying it. Service discovery is, itself, a service. It might indeed be one of several management services, such as "disable service X", "provide statistics", and so on. What we want is a general, extensible solution that doesn't affect the protocol nor existing applications.

So here's a small RFC - MMI, or the Majordomo Management Interface - that layers this on top of MDP: http://rfc.zeromq.org/spec:8. We already implemented it in the broker, though unless you read the whole thing you probably missed that. Here's how we use the service discovery in an application:

```
//
//  MMI echo query example
//

//  Lets us build this source without creating a library
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  This is the service we want to look up
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");

    //  This is the service we send our request to
    zmsg_t *reply = mdcli_send (session, "mmi.service", &request);

    if (reply) {
        char *reply_code = zframe_strdup (zmsg_first (reply));
        printf ("Lookup echo service: %s\n", reply_code);
        free (reply_code);
        zmsg_destroy (&reply);
    }
    else
        printf ("E: no response from broker, make sure it's
running\n");

    mdcli_destroy (&session);
    return 0;
}
```

*mmiecho.c: Service discovery over Majordomo*

The broker checks the service name, and handles any service starting with "mmi." itself, rather than passing the request on to a worker. Try this with and without a worker running, and you should see the little program report '200' or '404' accordingly. The implementation of MMI in our example broker is pretty weak. For example if a worker disappears, services remain "present". In practice a broker should remove services that have no workers after some configurable timeout.

## Idempotent Services

top prev next

Idempotency is not something to take a pill for. What it means is that it's safe to repeat an operation. Checking the clock is idempotent. Lending ones credit card to ones wife is not. While many client-to-server use cases are idempotent, some are not. Examples of idempotent use cases include:

- Stateless task distribution, i.e. a pipeline where the servers are stateless workers that compute a reply based purely on the state provided by a request. In such a case it's safe (though inefficient) to execute the same request many times.
- A name service that translates logical addresses into endpoints to bind or connect to. In such a case it's safe to make the same lookup request many times.

And here are examples of a non-idempotent use cases:

- A logging service. One does not want the same log information recorded more than once.
- Any service that has impact on downstream nodes, e.g. sends on information to other nodes. If that service gets the same request more than once, downstream nodes will get duplicate information.
- Any service that modifies shared data in some non-idempotent way. E.g. a service that debits a bank account is definitely not idempotent.

When our server applications are not idempotent, we have to think more carefully about when exactly they might crash. If an application dies when it's idle, or while it's processing a request, that's usually fine. We can use database transactions to make sure a debit and a credit are always done together, if at all. If the server dies while sending its reply, that's a problem, because as far as its concerned, it's done its work.

if the network dies just as the reply is making its way back to the client, the same problem arises. The client will think the server died, will resend the request, and the server will do the same work twice. Which is not what we want.

We use the fairly standard solution of detecting and rejecting duplicate requests. This means:

- The client must stamp every request with a unique client identifier and a unique message number.
- The server, before sending back a reply, stores it using the client id + message number as a key.
- The server, when getting a request from a given client, first checks if it has a reply for that client id + message number. If so, it does not process the request but just resends the reply.

## Disconnected Reliability (Titanic Pattern)

top prev next

Once you realize that Majordomo is a 'reliable' message broker, you might be tempted to add some spinning rust[1]. After all, this works for all the enterprise messaging systems. It's such a tempting idea that it's a little sad to have to be negative. But that's one of my specialties. So, some reasons you don't want rust-based brokers sitting in the center of your architecture are:

- As you've seen, the Lazy Pirate client performs surprisingly well. It works across a whole range of architectures, from direct client-to-server to distributed queue devices. It does tend to assume that workers are stateless and idempotent. But we can work around that limitation without resorting to rust.

- Rust brings a whole set of problems, from slow performance to additional pieces to have to manage, repair, and create 6am panics as they inevitably break at the start of daily operations. The beauty of the Pirate patterns in general is their simplicity. They won't crash. And if you're still worried about the hardware, you can move to a peer-to-peer pattern that has no broker at all. I'll explain later in this chapter.

Having said this, however, there is one sane use case for rust-based reliability, which is an asynchronous deconnected network. It solves a major problem with Pirate, namely that a client has to wait for an answer in real time. If clients and workers are only sporadically connected (think of email as an analogy), we can't use a stateless network between clients and workers. We have to put state in the middle.

So, here's the Titanic pattern, in which we write messages to disk to ensure they never get lost, no matter how sporadically clients and workers are connected. As we did for service discovery, we're going to layer Titanic on top of Majordomo rather than extend MDP. It's wonderfully lazy because it means we can implement our fire-and-forget reliability in a specialized worker, rather than in the broker. This is excellent for several reasons:

- It's much, *much* easier.
- It lets us mix brokers written in one language with workers written in another.
- It lets us evolve the fire-and-forget technology independently.

The only downside is that there's an extra network hop between broker and hard disk. This is easily worth it.

There are many ways to make a persistent request-reply architecture. We'll aim for simple and painless. The simplest design I could come up with, after playing with this for a few hours, is Titanic as a "proxy service". That is, it doesn't affect workers at all. If a client wants a reply immediately, it talks directly to a service and hopes the service is available. If a client is happy to wait a while, it talks to Titanic instead and asks, "hey, buddy, would you take care of this for me while I go buy my groceries?"

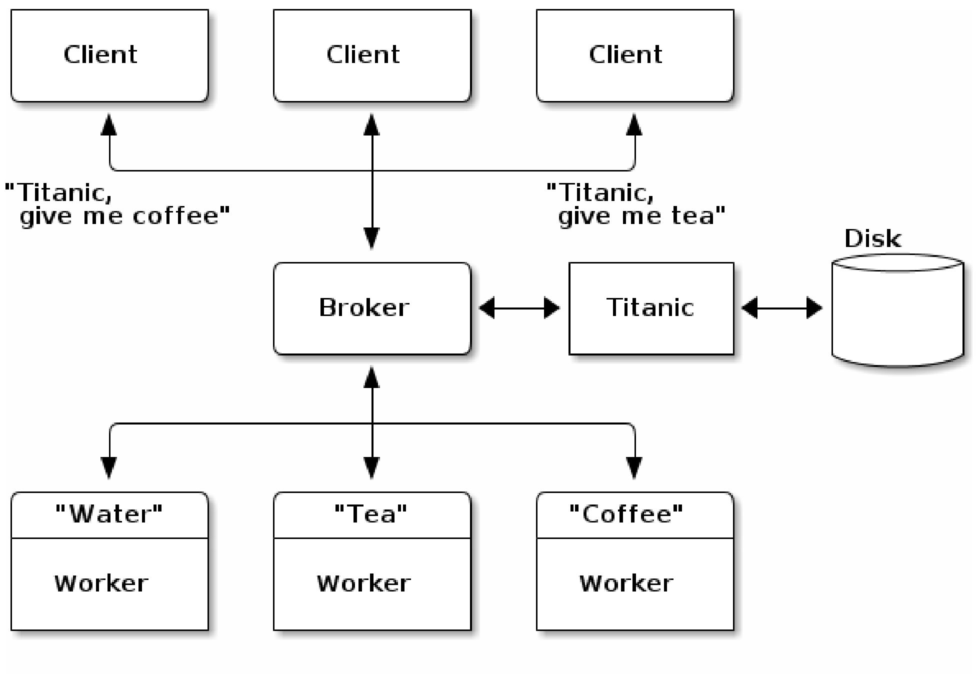


Figure 61 – Titanic Pattern

Titanic is thus both a worker, and a client. The dialog between client and Titanic goes along these lines:

- Client: please accept this request for me. Titanic: OK, done.
- Client: do you have a reply for me? Titanic: Yes, here it is. Or, no, not yet.
- Client: ok, you can wipe that request now, it's all happy. Titanic: OK, done.

Whereas the dialog between Titanic and broker and worker goes like this:

- Titanic: hey, broker, is there an echo service? Broker: uhm, yeah, seems like.
- Titanic: hey, echo, please handle this for me. Echo: sure, here you are.
- Titanic: sweeeeet!

You can work through this, and the possible failure scenarios. If a worker crashes while processing a request, Titanic retries, indefinitely. If a reply gets lost somewhere, Titanic will retry. If the request gets processed but the client doesn't get the reply, it will ask again. If Titanic crashes while processing a request, or a reply, the client will try again. As long as requests are fully committed to safe storage, work can't get lost.

The handshaking is pedantic, but can be pipelined, i.e. clients can use the asynchronous Majordomo pattern to do a lot of work and then get the responses later.

We need some way for a client to request *its* replies. We'll have many clients asking for the same services, and clients disappear and reappear with different identities. So here is a simple, reasonably secure solution:

- Every request generates a universally unique ID (UUID), which Titanic returns to the client when it's queued the request.
- When a client asks for a reply, it must specify the UUID for the original request.

This puts some onus on the client to store its request UUIDs safely, but it removes any need for authentication. What alternatives are there? We could use durable sockets, i.e. explicit client identities. That creates a management issue when we have many clients, and opens the door for the inevitable errors caused by two clients using the same identity.

Before we jump off and write yet another formal specification (fun, fun!) let's consider how the client talks to Titanic. One way is to use a single service and send it three different request types. Another way, which seems simpler, is to use three services:

- **titanic.request** - store a request message, return a UUID for the request.
- **titanic.reply** - fetch a reply, if available, for a given request UUID.
- **titanic.close** - confirm that a reply has been stored and processed.

We'll just make a multithreaded worker, which as we've seen from our multithreading experience with ØMQ, is trivial. However before jumping into code let's sketch down what Titanic would look like in terms of ØMQ messages and frames: http://rfc.zeromq.org/spec:9. This is the "Titanic Service Protocol", or TSP.

Using TSP is clearly more work for client applications than accessing a service directly via MDP. Here's the shortest robust 'echo' client example:

```
//
//  Titanic client example
//  Implements client side of http://rfc.zeromq.org/spec:9

//  Lets us build this source without creating a library
#include "mdcliapi.c"

//  Calls a TSP service
//  Returns reponse if successful (status code 200 OK), else NULL
//
static zmsg_t *
s_service_call (mdcli_t *session, char *service, zmsg_t **request_p)
{
    zmsg_t *reply = mdcli_send (session, service, request_p);
    if (reply) {
        zframe_t *status = zmsg_pop (reply);
        if (zframe_streq (status, "200")) {
            zframe_destroy (&status);
            return reply;
        }
        else
        if (zframe_streq (status, "400")) {
            printf ("E: client fatal error, aborting\n");
            exit (EXIT_FAILURE);
        }
        else
        if (zframe_streq (status, "500")) {
            printf ("E: server fatal error, aborting\n");
            exit (EXIT_FAILURE);
        }
    }
    else
        exit (EXIT_SUCCESS);    //  Interrupted or failed

    zmsg_destroy (&reply);
    return NULL;        //  Didn't succeed, don't care why not
}

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  1. Send 'echo' request to Titanic
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = s_service_call (
        session, "titanic.request", &request);

    zframe_t *uuid = NULL;
    if (reply) {
        uuid = zmsg_pop (reply);
        zmsg_destroy (&reply);
        zframe_print (uuid, "I: request UUID ");
    }

    //  2. Wait until we get a reply
    while (!zctx_interrupted) {
        zclock_sleep (100);
        request = zmsg_new ();
        zmsg_add (request, zframe_dup (uuid));
        zmsg_t *reply = s_service_call (
            session, "titanic.reply", &request);

        if (reply) {
            char *reply_string = zframe_strdup (zmsg_last (reply));
            printf ("Reply: %s\n", reply_string);
            free (reply_string);
            zmsg_destroy (&reply);

            //  3. Close request
            request = zmsg_new ();
            zmsg_add (request, zframe_dup (uuid));
            reply = s_service_call (session, "titanic.close",
&request);
            zmsg_destroy (&reply);
            break;
        }
        else {
            printf ("I: no reply yet, trying again…\n");
            zclock_sleep (5000);     //  Try again in 5 seconds
        }
    }
    zframe_destroy (&uuid);
    mdcli_destroy (&session);
    return 0;
}
```

*ticlient.c: Titanic client example*

Of course this can and in practice would be wrapped up in some kind of framework. Real application developers should never see messaging up close, it's a tool for more technically-minded experts to build frameworks and APIs. If we had infinite time to explore this, I'd make a TSP API example, and bring the client application back down to a few lines of code. But it's the same principle as we saw for MDP, no need to be repetitive.

Here's the Titanic implementation. This server handles the three services using three threads, as proposed. It does full persistence to disk using the most brute-force approach possible: one file per message. It's so simple it's scary, the only complex part is that it keeps a separate 'queue' of all requests to avoid reading the directory over and over:

```
//
//  Titanic service
//
//  Implements server side of http://rfc.zeromq.org/spec:9
//  Lets us build this source without creating a library
#include "mdwrkapi.c"
#include "mdcliapi.c"

#include "zfile.h"
#include <uuid/uuid.h>

//  Return a new UUID as a printable character string
//  Caller must free returned string when finished with it

static char *
s_generate_uuid (void)
{
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    uuid_generate (uuid);
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuid [byte_nbr] >> 4];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuid [byte_nbr] & 15];
    }
    return uuidstr;
}

//  Returns freshly allocated request filename for given UUID

#define TITANIC_DIR ".titanic"

static char *
s_request_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.req", uuid);
    return filename;
}

//  Returns freshly allocated reply filename for given UUID

static char *
s_reply_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.rep", uuid);
    return filename;
}

//  ---------------------------------------------------------------------
----
//  Titanic request service

static void
titanic_request (void *args, zctx_t *ctx, void *pipe)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.request", 0);
    zmsg_t *reply = NULL;

    while (TRUE) {
        //  Send reply if it's not null
        //  And then get next request from broker
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
```

```
                break;      //  Interrupted, exit

            //  Ensure message directory exists
            file_mkdir (TITANIC_DIR);

            //  Generate UUID and save message to disk
            char *uuid = s_generate_uuid ();
            char *filename = s_request_filename (uuid);
            FILE *file = fopen (filename, "w");
            assert (file);
            zmsg_save (request, file);
            fclose (file);
            free (filename);
            zmsg_destroy (&request);

            //  Send UUID through to message queue
            reply = zmsg_new ();
            zmsg_addstr (reply, uuid);
            zmsg_send (&reply, pipe);

            //  Now send UUID back to client
            //  Done by the mdwrk_recv() at the top of the loop
            reply = zmsg_new ();
            zmsg_addstr (reply, "200");
            zmsg_addstr (reply, uuid);
            free (uuid);
        }
        mdwrk_destroy (&worker);
    }

    //  ---------------------------------------------------------------------
    ----
    //  Titanic reply service

    static void *
    titanic_reply (void *context)
    {
        mdwrk_t *worker = mdwrk_new (
            "tcp://localhost:5555", "titanic.reply", 0);
        zmsg_t *reply = NULL;

        while (TRUE) {
            zmsg_t *request = mdwrk_recv (worker, &reply);
            if (!request)
                break;      //  Interrupted, exit

            char *uuid = zmsg_popstr (request);
            char *req_filename = s_request_filename (uuid);
            char *rep_filename = s_reply_filename (uuid);
            if (file_exists (rep_filename)) {
                FILE *file = fopen (rep_filename, "r");
                assert (file);
                reply = zmsg_load (file);
                zmsg_pushstr (reply, "200");
                fclose (file);
            }
            else {
                reply = zmsg_new ();
                if (file_exists (req_filename))
                    zmsg_pushstr (reply, "300"); //Pending
                else
                    zmsg_pushstr (reply, "400"); //Unknown
            }
            zmsg_destroy (&request);
            free (uuid);
            free (req_filename);
            free (rep_filename);
        }
        mdwrk_destroy (&worker);
        return 0;
    }

    //  ---------------------------------------------------------------------
    ----
    //  Titanic close service

    static void *
    titanic_close (void *context)
    {
        mdwrk_t *worker = mdwrk_new (
            "tcp://localhost:5555", "titanic.close", 0);
        zmsg_t *reply = NULL;

        while (TRUE) {
            zmsg_t *request = mdwrk_recv (worker, &reply);
            if (!request)
                break;      //  Interrupted, exit

            char *uuid = zmsg_popstr (request);
            char *req_filename = s_request_filename (uuid);
            char *rep_filename = s_reply_filename (uuid);
            file_delete (req_filename);
            file_delete (rep_filename);
            free (uuid);
            free (req_filename);
            free (rep_filename);

            zmsg_destroy (&request);
            reply = zmsg_new ();
            zmsg_addstr (reply, "200");
        }
        mdwrk_destroy (&worker);
        return 0;
    }

    //  Attempt to process a single request, return 1 if successful

    static int
    s_service_success (mdcli_t *client, char *uuid)
    {
        //  Load request message, service will be first frame
        char *filename = s_request_filename (uuid);
        FILE *file = fopen (filename, "r");
        free (filename);

        //  If the client already closed request, treat as successful
        if (!file)
            return 1;

        zmsg_t *request = zmsg_load (file);
        fclose (file);
        zframe_t *service = zmsg_pop (request);
        char *service_name = zframe_strdup (service);

        //  Use MMI protocol to check if service is available
        zmsg_t *mmi_request = zmsg_new ();
        zmsg_add (mmi_request, service);
        zmsg_t *mmi_reply = mdcli_send (client, "mmi.service",
    &mmi_request);
        int service_ok = (mmi_reply
            && zframe_streq (zmsg_first (mmi_reply), "200"));
        zmsg_destroy (&mmi_reply);

        if (service_ok) {
            zmsg_t *reply = mdcli_send (client, service_name, &request);
            if (reply) {
                filename = s_reply_filename (uuid);
                FILE *file = fopen (filename, "w");
                assert (file);
                zmsg_save (reply, file);
                fclose (file);
                free (filename);
                return 1;
            }
            zmsg_destroy (&reply);
        }
        else
            zmsg_destroy (&request);

        free (service_name);
        return 0;
    }

    int main (int argc, char *argv [])
    {
        int verbose = (argc > 1 && streq (argv [1], "-v"));
        zctx_t *ctx = zctx_new ();

        //  Create MDP client session with short timeout
        mdcli_t *client = mdcli_new ("tcp://localhost:5555", verbose);
        mdcli_set_timeout (client, 1000);  //  1 sec
        mdcli_set_retries (client, 1);     //  only 1 retry

        void *request_pipe = zthread_fork (ctx, titanic_request, NULL);
        zthread_new (ctx, titanic_reply, NULL);
        zthread_new (ctx, titanic_close, NULL);

        //  Main dispatcher loop
        while (TRUE) {
            //  We'll dispatch once per second, if there's no activity
            zmq_pollitem_t items [] = { { request_pipe, 0, ZMQ_POLLIN, 0
    } };
            int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;              //  Interrupted
            if (items [0].revents & ZMQ_POLLIN) {
                //  Ensure message directory exists
                file_mkdir (TITANIC_DIR);

                //  Append UUID to queue, prefixed with '-' for pending
                zmsg_t *msg = zmsg_recv (request_pipe);
                if (!msg)
                    break;          //  Interrupted
                FILE *file = fopen (TITANIC_DIR "/queue", "a");
                char *uuid = zmsg_popstr (msg);
                fprintf (file, "-%s\n", uuid);
                fclose (file);
                free (uuid);
                zmsg_destroy (&msg);
            }
            //  Brute-force dispatcher
            //
            char entry [] = "?…….:…….:…….:…….:";
            FILE *file = fopen (TITANIC_DIR "/queue", "r+");
            while (file && fread (entry, 33, 1, file) == 1) {
                //  UUID is prefixed with '-' if still waiting
                if (entry [0] == '-') {
                    if (verbose)
                        printf ("I: processing request %s\n", entry + 1);
                    if (s_service_success (client, entry + 1)) {
                        //  Mark queue entry as processed
                        fseek (file, -33, SEEK_CUR);
                        fwrite ("+", 1, 1, file);
                        fseek (file, 32, SEEK_CUR);
                    }
                }
                //  Skip end of line, LF or CRLF
                if (fgetc (file) == '\r')
                    fgetc (file);
                if (zctx_interrupted)
                    break;
            }
            if (file)
                fclose (file);
        }
        mdcli_destroy (&client);
        return 0;
    }
```

*titanic.c: Titanic broker example*

To test this, start `mdbroker` and `titanic`, then run `ticlient`. Now start `mdworker` arbitrarily, and you should see the client getting a response and exiting happily.

Some notes about this code:

- We use MMI to only send requests to services that appear to be running. This works as well as the MMI implementation in the broker.
- We use an inproc connection to send new request data from the **titanic.request** service through to the main dispatcher. This saves the dispatcher from having to scan the disk directory, load all request files, and sort them by date/time.

The important thing about this example is not performance (which is surely terrible, I've not tested it), but how well it implements the reliability contract. To try it, start the mdbroker and titanic programs. Then start the ticlient, and then start the mdworker echo service. You can run all four of these using the '-v' option to do verbose tracing of activity. You can stop and restart any piece *except* the client and nothing will get lost.

If you want to use Titanic in real cases, you'll rapidly be asking "how do we make this faster?" Here's what I'd do, starting with the example implementation:

* Use a single disk file for all data, rather than multiple files. Operating systems are usually better at handling a few large files than many smaller ones.
* Organize that disk file as a circular buffer so that new requests can be written contiguously (with very occasional wraparound). One thread, writing full speed to a disk file can work rapidly.
* Keep the index in memory and rebuild the index at startup time, from the disk buffer. This saves the extra disk head flutter needed to keep the index fully safe on disk. You would want an fsync after every message, or every N milliseconds if you were prepared to lose the last M messages in case of a system failure.
* Use a solid-state drive rather than spinning iron oxide platters.
* Preallocate the entire file, or allocate in large chunks allowing the circular buffer to grow and shrink as needed. This avoids fragmentation and ensures most reads and writes are contiguous.

And so on. What I'd not recommend is storing messages in a database, not even a 'fast' key/value store, unless you really like a specific database and don't have performance worries. You will pay a steep price for the abstraction, 10 to 1000x over a raw disk file.

If you want to make Titanic *even more reliable*, you can do this by duplicating requests to a second server, which you'd place in a second location just far enough to survive nuclear attack on your primary location, yet not so far that you get too much latency.

If you want to make Titanic *much faster and less reliable*, you can store requests and replies purely in memory. This will give you the functionality of a disconnected network, but it won't survive a crash of the Titanic server itself.

## High-availability Pair (Binary Star Pattern)

top prev next

### Overview

top prev next

The Binary Star pattern puts two servers in a primary-backup high-availability pair. At any given time, one of these accepts connections from client applications (it is the "master") and one does not (it is the "slave"). Each server monitors the other. If the master disappears from the network, after a certain time the slave takes over as master.

Binary Star pattern was developed by Pieter Hintjens and Martin Sustrik for the iMatix OpenAMQ server. We designed it:

* To provide a straight-forward high-availability solution.
* To be simple enough to actually understand and use.
* To failover reliably when needed, and only when needed.

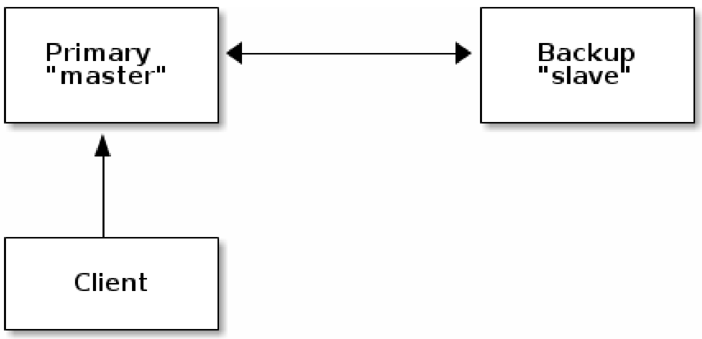


Figure 62 – High availability pair, normal operation

Assuming we have a Binary Star pair running, here are the different scenarios that will result in failover happening:

1. The hardware running the primary server has a fatal problem (power supply explodes, machine catches fire, or someone simply unplugs it by mistake), and disappears. Applications see this, and reconnect to the backup server.
2. The network segment on which the primary server sits crashes - perhaps a router gets hit by a power spike - and applications start to reconnect to the backup server.
3. The primary server crashes or is killed by the operator and does not restart automatically.

Recovery from failover works as follows:

1. The operators restart the primary server and fix whatever problems were causing it to disappear from the network.
2. The operators stop the backup server, at a moment that will cause minimal disruption to applications.
3. When applications have reconnected to the primary server, the operators restart the backup server.

Recovery (to using the primary server as master) is a manual operation. Painful experience teaches us that automatic recovery is undesirable. There are several reasons:

- Failover creates an interruption of service to applications, possibly lasting 10-30 seconds. If there is a real emergency, this is much better than total outage. But if recovery creates a further 10-30 second outage, it is better that this happens off-peak, when users have gone off the network.

- When there is an emergency, it's a Good Idea to create predictability for those trying to fix things. Automatic recovery creates uncertainty for system admins, who can no longer be sure which server is in charge without double-checking.

- Last, you can get situations with automatic recovery where networks will fail over, and then recover, and operators are then placed in a difficult position to analyze what happened. There was an interruption of service, but the cause isn't clear.

Having said this, the Binary Star pattern will fail back to the primary server if this is running (again) and the backup server fails. In fact this is how we provoke recovery.

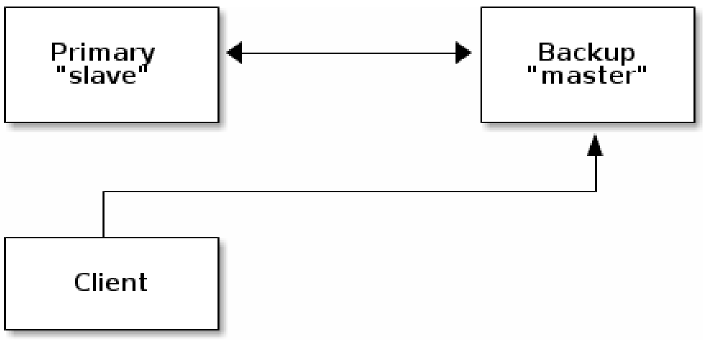


Figure 63 – –High availability pair, during failover

The shutdown process for a Binary Star pair is to either:

1. Stop the passive server and then stop the active server at any later time, or
2. Stop both servers in any order but within a few seconds of each other.

Stopping the active and then the passive server with any delay longer than the failover timeout will cause applications to disconnect, then reconnect, then disconnect again, which may disturb users.

## Detailed Requirements

top prev next

Binary Star is as simple as it can be, while still working accurately. In fact the current design is the third complete redesign. Each of the previous designs we found to be too complex, trying to do too much, and we stripped out functionality until we came to a design that was understandable and use, and reliable enough to be worth using.

These are our requirements for a high-availability architecture:

- The failover is meant to provide insurance against catastrophic system failures, such as hardware breakdown, fire, accident, etc. To guard against ordinary server crashes there are simpler ways to recover.
- Failover time should be under 60 seconds and preferably under 10 seconds.
- Failover has to happen automatically, whereas recover must happen manually. We want applications to switch over to the backup server automatically but we do not want them to switch back to the primary server except when the operators have fixed whatever problem there was, and decided that it is a good time to interrupt applications again.
- The semantics for client applications should be simple and easy for developers to understand. Ideally they should be hidden in the client API.
- There should be clear instructions for network architects on how to avoid designs that could lead to split brain syndrome in which both servers in a Binary Star pair think they are the master server.
- There should be no dependencies on the order in which the two servers are started.
- It must be possible to make planned stops and restarts of either server without stopping client applications (though they may be forced to reconnect).
- Operators must be able to monitor both servers at all times.
- It must be possible to connect the two servers using a high-speed dedicated network connection. That is, failover synchronization must be able to use a a specific IP route.

We make these assumptions:

- A single backup server provides enough insurance, we don't need multiple levels of backup.
- The primary and backup server are equally capable of carrying the application load. We do not attempt to balance load across the servers.
- There is sufficient budget to cover a fully redundant backup server that does nothing almost all the time.

We don't attempt to cover:

- The use of an active backup server or load balancing. In a Binary Star pair, the backup server is inactive and does no useful work until the primary server goes off-line.
- The handling of persistent messages or transactions in any way. We assuming a network of unreliable (and probably untrusted) servers or Binary Star pairs.
- Any automatic exploration of the network. The Binary Star pair is manually and explicitly defined in the network and is known to applications (at least in their

configuration data).

- Replication of state or messages between servers. All server-side state much be recreated by applications when they fail over.

Here is the key terminology we use in Binary Star:

- **Primary** - the primary server is the one that is normally 'master'.
- **Backup** - the backup server is the one that is normally 'slave', it will become master if and when the primary server disappears from the network, and when client applications ask the backup server to connect.
- **Master** - the master server is the one of a Binary Star pair that accepts client connections. There is always exactly one master server.
- **Slave** - the slave server is the one that takes over if the master disappears. Note that when a Binary Star pair is running normally, the primary server is master, and the backup is slave. When a failover has happened, the roles are switched.

To configure a Binary Star pair, you need to:

1. Tell the primary server where the backup server is.
2. Tell the backup server where the primary server is.
3. Optionally, tune the failover response times, which must be the same for both servers.

The main tuning concern is how frequently you want the servers to check their peering status, and how quickly you want to activate failover. In our example, the failover timeout value defaults to 2000 msec. If you reduce this, the backup server will take over as master more rapidly but may take over in cases where the primary server could recover. You may for example have wrapped the primary server in a shell script that restarts it if it crashes. In that case the timeout should be higher than the time needed to restart the primary server.

For client applications to work properly with a Binary Star pair, they must:

1. Know both server addresses.
2. Try to connect to the primary server, and if that fails, to the backup server.
3. Detect a failed connection, typically using heartbeating.
4. Try to reconnect to primary, and then backup, with a delay between retries that is at least as high as the server failover timeout.
5. Recreate all of the state they require on a server.
6. Retransmit messages lost during a failover, if messages need to be reliable.

It's not trivial work, and we'd usually wrap this in an API that hides it from real end-user applications.

These are the main limitations of the Binary Star pattern:

- A server process cannot be part of more than one Binary Star pair.
- A primary server can have a single backup server, no more.
- The backup server cannot do useful work while in slave mode.
- The backup server must be capable of handling full application loads.
- Failover configuration cannot be modified at runtime.
- Client applications must do some work to benefit from failover.

## Preventing Split-Brain Syndrome

top prev next

"Split-brain syndrome" is when different parts of a cluster think they are 'master' at the same time. It causes applications to stop seeing each other. Binary Star has an algorithm for detecting and eliminating split brain, based on a three-way decision mechanism (a server will not decide to become master until it gets application connection requests and it cannot see its peer server).

However it is still possible to (mis)design a network to fool this algorithm. A typical scenario would a Binary Star pair distributed between two buildings, where each building also had a set of applications, and there was a single network link between both buildings. Breaking this link would create two sets of client applications, each with half of the Binary Star pair, and each failover server would become active.

To prevent split-brain situations, we *MUST* connect Binary Star pairs using a dedicated network link, which can be as simple as plugging them both into the same switch or better, using a cross-over cable directly between two machines.

We must not split a Binary Star architecture into two islands, each with a set of applications. While this may be a common type of network architecture, we'd use federation, not high-availability failover, in such cases.

A suitably paranoid network configuration would use two private cluster interconnects, rather than a single one. Further, the network cards used for the cluster would be different to those used for message in/out, and possibly even on different PCI paths on the server hardware. The goal being to separate possible failures in the network from possible failures in the cluster. Network ports have a relatively high failure rate.

## Binary Star Implementation

top prev next

Without further ado, here is an implementation of the Binary Star server:

```
//
//  Binary Star server
//
#include "czmq.h"

//  We send state information every this often
//  If peer doesn't respond in two heartbeats, it is 'dead'
#define HEARTBEAT 1000          //  In msecs

//  States we can be in at any point in time
typedef enum {
    STATE_PRIMARY = 1,          //  Primary, waiting for peer to
connect
    STATE_BACKUP = 2,           //  Backup, waiting for peer to
connect
    STATE_ACTIVE = 3,           //  Active - accepting connections
    STATE_PASSIVE = 4           //  Passive - not accepting
connections
} state_t;

//  Events, which start with the states our peer can be in
typedef enum {
    PEER_PRIMARY = 1,           //  HA peer is pending primary
    PEER_BACKUP = 2,            //  HA peer is pending backup
    PEER_ACTIVE = 3,            //  HA peer is active
    PEER_PASSIVE = 4,           //  HA peer is passive
    CLIENT_REQUEST = 5          //  Client makes request
} event_t;

//  Our finite state machine
```

```
typedef struct {
    state_t state;              //  Current state
    event_t event;              //  Current event
    int64_t peer_expiry;        //  When peer is considered 'dead'
} bstar_t;

//  Execute finite state machine (apply event to state)
//  Returns TRUE if there was an exception

static Bool
s_state_machine (bstar_t *fsm)
{
    Bool exception = FALSE;
    //  Primary server is waiting for peer to connect
    //  Accepts CLIENT_REQUEST events in this state
    if (fsm->state == STATE_PRIMARY) {
        if (fsm->event == PEER_BACKUP) {
            printf ("I: connected to backup (slave), ready as
master\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: connected to backup (master), ready as
slave\n");
            fsm->state = STATE_PASSIVE;
        }
    }
    else
    //  Backup server is waiting for peer to connect
    //  Rejects CLIENT_REQUEST events in this state
    if (fsm->state == STATE_BACKUP) {
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: connected to primary (master), ready as
slave\n");
            fsm->state = STATE_PASSIVE;
        }
        else
        if (fsm->event == CLIENT_REQUEST)
            exception = TRUE;
    }
    else
    //  Server is active
    //  Accepts CLIENT_REQUEST events in this state
    if (fsm->state == STATE_ACTIVE) {
        if (fsm->event == PEER_ACTIVE) {
            //  Two masters would mean split-brain
            printf ("E: fatal error - dual masters, aborting\n");
            exception = TRUE;
        }
    }
    else
    //  Server is passive
    //  CLIENT_REQUEST events can trigger failover if peer looks dead
    if (fsm->state == STATE_PASSIVE) {
        if (fsm->event == PEER_PRIMARY) {
            //  Peer is restarting - become active, peer will go
passive
            printf ("I: primary (slave) is restarting, ready as
master\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_BACKUP) {
            //  Peer is restarting - become active, peer will go
passive
            printf ("I: backup (slave) is restarting, ready as
master\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_PASSIVE) {
            //  Two passives would mean cluster would be non-
responsive
            printf ("E: fatal error - dual slaves, aborting\n");
            exception = TRUE;
        }
        else
        if (fsm->event == CLIENT_REQUEST) {
            //  Peer becomes master if timeout has passed
            //  It's the client request that triggers the failover
            assert (fsm->peer_expiry > 0);
            if (zclock_time () >= fsm->peer_expiry) {
                //  If peer is dead, switch to the active state
                printf ("I: failover successful, ready as master\n");
                fsm->state = STATE_ACTIVE;
            }
            else
                //  If peer is alive, reject connections
                exception = TRUE;
        }
    }
    return exception;
}

int main (int argc, char *argv [])
{
    //  Arguments can be either of:
    //      -p  primary server, at tcp://localhost:5001
    //      -b  backup server, at tcp://localhost:5002
    zctx_t *ctx = zctx_new ();
    void *statepub = zsocket_new (ctx, ZMQ_PUB);
    void *statesub = zsocket_new (ctx, ZMQ_SUB);
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    bstar_t fsm = { 0 };

    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: Primary master, waiting for backup (slave)\n");
        zsocket_bind (frontend, "tcp://*:5001");
        zsocket_bind (statepub, "tcp://*:5003");
        zsocket_connect (statesub, "tcp://localhost:5004");
        fsm.state = STATE_PRIMARY;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: Backup slave, waiting for primary (master)\n");
        zsocket_bind (frontend, "tcp://*:5002");
        zsocket_bind (statepub, "tcp://*:5004");
        zsocket_connect (statesub, "tcp://localhost:5003");
        fsm.state = STATE_BACKUP;
    }
    else {
        printf ("Usage: bstarsrv { -p | -b }\n");
        zctx_destroy (&ctx);
        exit (0);
    }
    //  Set timer for next outgoing state message
    int64_t send_state_at = zclock_time () + HEARTBEAT;

    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { statesub, 0, ZMQ_POLLIN, 0 }
        };
        int time_left = (int) ((send_state_at - zclock_time ()));
        if (time_left < 0)
            time_left = 0;
        int rc = zmq_poll (items, 2, time_left * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Context has been shut down

        if (items [0].revents & ZMQ_POLLIN) {
            //  Have a client request
            zmsg_t *msg = zmsg_recv (frontend);
            fsm.event = CLIENT_REQUEST;
            if (s_state_machine (&fsm) == FALSE)
                //  Answer client by echoing request back
                zmsg_send (&msg, frontend);
            else
                zmsg_destroy (&msg);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  Have state from our peer, execute as event
            char *message = zstr_recv (statesub);
            fsm.event = atoi (message);
            free (message);
            if (s_state_machine (&fsm))
                break;          //  Error, so exit
            fsm.peer_expiry = zclock_time () + 2 * HEARTBEAT;
        }
        //  If we timed-out, send state to peer
        if (zclock_time () >= send_state_at) {
            char message [2];
            sprintf (message, "%d", fsm.state);
            zstr_send (statepub, message);
            send_state_at = zclock_time () + HEARTBEAT;
        }
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    //  Shutdown sockets and context
    zctx_destroy (&ctx);
    return 0;
}
```

*bstarsrv.c: Binary Star server*

And here is the client:

```
//
//  Binary Star client
//
#include "czmq.h"

#define REQUEST_TIMEOUT     1000    //  msecs
#define SETTLE_DELAY        2000    //  Before failing over

int main (void)
{
    zctx_t *ctx = zctx_new ();

    char *server [] = { "tcp://localhost:5001",
"tcp://localhost:5002" };
    uint server_nbr = 0;

    printf ("I: connecting to server at %s…\n", server [server_nbr]);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, server [server_nbr]);

    int sequence = 0;
    while (!zctx_interrupted) {
        //  We send a request, then we work to get a reply
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            //  Poll socket for a reply, with timeout
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 }
};
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT *
ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  Interrupted

            //  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 = zstr_recv (client);
                if (atoi (reply) == sequence) {
                    printf ("I: server replied OK (%s)\n", reply);
                    expect_reply = 0;
                    sleep (1);  //  One request per second
                }
                else {
                    printf ("E: malformed reply from server: %s\n",
                        reply);
                }
                free (reply);
            }
            else {
                printf ("W: no response from server, failing
over\n");
                //  Old socket is confused; close it and open a new
one
                zsocket_destroy (ctx, client);
                server_nbr = (server_nbr + 1) % 2;
                zclock_sleep (SETTLE_DELAY);
                printf ("I: connecting to server at %s…\n",
                        server [server_nbr]);
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, server [server_nbr]);

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

*bstarcli.c: Binary Star client*

To test Binary Star, start the servers and client in any order:

```
bstarsrv -p     # Start primary
bstarsrv -b     # Start backup
bstarcli
```

You can then provoke failover by killing the primary server, and recovery by restarting the primary and killing the backup. Note how it's the client vote that triggers failover, and recovery.

This diagram shows the finite state machine. States in green accept client requests, states in pink refuse them. Events are the peer state, so "Peer Active" means the other server has told us it's active. "Client Request" means we've receive a client request. "Client Vote" means we've received a client request AND our peer is inactive for two heartbeats.

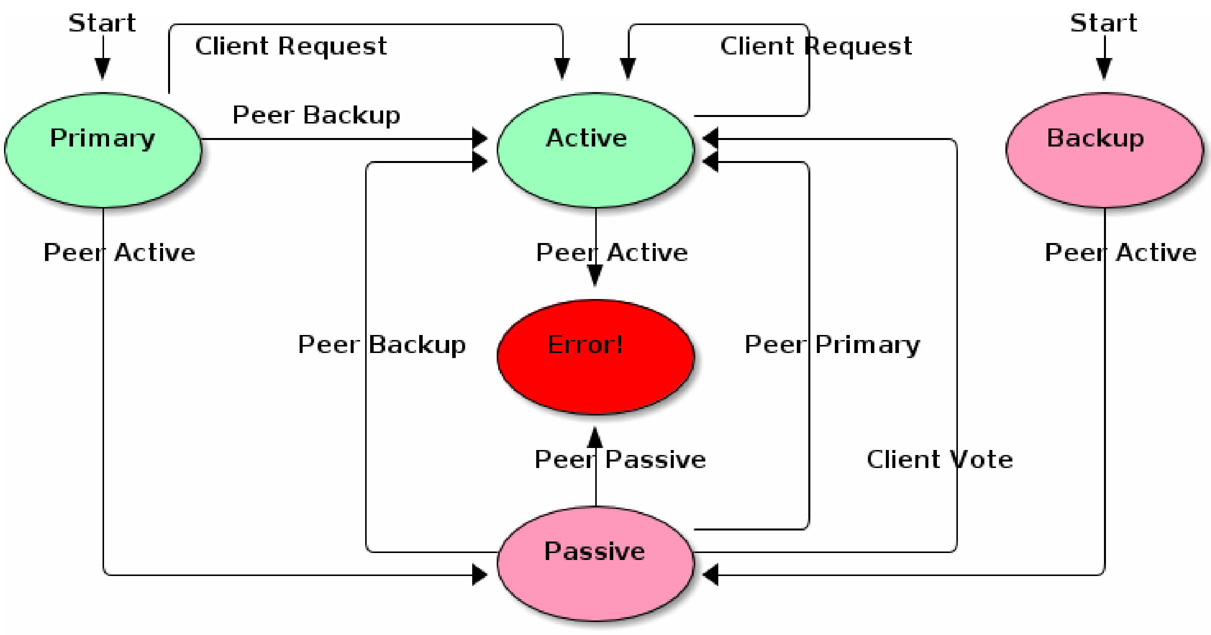


Figure 64 – Binary Star finite state machine

Note that the servers use PUB-SUB sockets for state exchange. No other socket combination will work here. PUSH and DEALER block if there is no peer ready to receive a message. PAIR does not reconnect if the peer disappears and comes back. ROUTER needs the address of the peer before it can send it a message.

These are the main limitations of the Binary Star pattern:

- A server process cannot be part of more than one Binary Star pair.
- A primary server can have a single backup server, no more.
- The backup server cannot do useful work while in slave mode.
- The backup server must be capable of handling full application loads.
- Failover configuration cannot be modified at runtime.
- Client applications must do some work to benefit from failover.

## Binary Star Reactor

top prev next

Binary Star is useful and generic enough to package up as a reusable reactor class. In C we wrap the czmq `zloop` class, though your milage may vary in other languages. Here is the `bstar` interface in C:

```
//  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);
```

And here is the class implementation:

```
/*
=====================================================================
    bstar - Binary Star reactor

    -----------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.

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

#include "bstar.h"

//  States we can be in at any point in time
typedef enum {
    STATE_PRIMARY = 1,          //  Primary, waiting for peer to
connect
    STATE_BACKUP = 2,           //  Backup, waiting for peer to
connect
    STATE_ACTIVE = 3,           //  Active - accepting connections
    STATE_PASSIVE = 4           //  Passive - not accepting
connections
} state_t;

//  Events, which start with the states our peer can be in
typedef enum {
    PEER_PRIMARY = 1,           //  HA peer is pending primary
    PEER_BACKUP = 2,            //  HA peer is pending backup
    PEER_ACTIVE = 3,            //  HA peer is active
    PEER_PASSIVE = 4,           //  HA peer is passive
    CLIENT_REQUEST = 5          //  Client makes request
} event_t;

//  We send state information every this often
//  If peer doesn't respond in two heartbeats, it is 'dead'
#define BSTAR_HEARTBEAT     1000        //  In msecs

//  Structure of our class

struct _bstar_t {
    zctx_t *ctx;                //  Our private context
    zloop_t *loop;              //  Reactor loop
    void *statepub;             //  State publisher
    void *statesub;             //  State subscriber
    state_t state;              //  Current state
    event_t event;              //  Current event
    int64_t peer_expiry;        //  When peer is considered 'dead'
    zloop_fn *voter_fn;         //  Voting socket handler
    void *voter_arg;            //  Arguments for voting handler
    zloop_fn *master_fn;        //  Call when become master
    void *master_arg;           //  Arguments for handler
    zloop_fn *slave_fn;         //  Call when become slave
    void *slave_arg;            //  Arguments for handler
};

//  ---------------------------------------------------------------------
----
//  Binary Star finite state machine (applies event to state)
//  Returns -1 if there was an exception, 0 if event was valid.

static int
s_execute_fsm (bstar_t *self)
{
    int rc = 0;
    //  Primary server is waiting for peer to connect
    //  Accepts CLIENT_REQUEST events in this state
    if (self->state == STATE_PRIMARY) {
        if (self->event == PEER_BACKUP) {
            zclock_log ("I: connected to backup (slave), ready as
master");
            self->state = STATE_ACTIVE;
            if (self->master_fn)
                (self->master_fn) (self->loop, NULL, self-
>master_arg);
        }
        else
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: connected to backup (master), ready as
slave");
            self->state = STATE_PASSIVE;
            if (self->slave_fn)
                (self->slave_fn) (self->loop, NULL, self->slave_arg);
        }
        else
        if (self->event == CLIENT_REQUEST) {
            zclock_log ("I: request from client, ready as master");
            self->state = STATE_ACTIVE;
            if (self->master_fn)
                (self->master_fn) (self->loop, NULL, self-
>master_arg);
        }
    }
    else
    //  Backup server is waiting for peer to connect
    //  Rejects CLIENT_REQUEST events in this state
    if (self->state == STATE_BACKUP) {
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: connected to primary (master), ready as
slave");
            self->state = STATE_PASSIVE;
            if (self->slave_fn)
                (self->slave_fn) (self->loop, NULL, self->slave_arg);
        }
        else
        if (self->event == CLIENT_REQUEST)
            rc = -1;
    }
    else
    //  Server is active
    //  Accepts CLIENT_REQUEST events in this state
    //  The only way out of ACTIVE is death
    if (self->state == STATE_ACTIVE) {
        if (self->event == PEER_ACTIVE) {
            //  Two masters would mean split-brain
            zclock_log ("E: fatal error - dual masters, aborting");
            rc = -1;
        }
    }
    else
    //  Server is passive
    //  CLIENT_REQUEST events can trigger failover if peer looks dead
    if (self->state == STATE_PASSIVE) {
        if (self->event == PEER_PRIMARY) {
            //  Peer is restarting - become active, peer will go
passive
            zclock_log ("I: primary (slave) is restarting, ready as
master");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_BACKUP) {
            //  Peer is restarting - become active, peer will go
passive
            zclock_log ("I: backup (slave) is restarting, ready as
master");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_PASSIVE) {
            //  Two passives would mean cluster would be non-
responsive
            zclock_log ("E: fatal error - dual slaves, aborting");
            rc = -1;
        }
        else
        if (self->event == CLIENT_REQUEST) {
            //  Peer becomes master if timeout has passed
            //  It's the client request that triggers the failover
            assert (self->peer_expiry > 0);
            if (zclock_time () >= self->peer_expiry) {
                //  If peer is dead, switch to the active state
                zclock_log ("I: failover successful, ready as
master");
                self->state = STATE_ACTIVE;
            }
            else
                //  If peer is alive, reject connections
                rc = -1;
        }
        //  Call state change handler if necessary
        if (self->state == STATE_ACTIVE && self->master_fn)
            (self->master_fn) (self->loop, NULL, self->master_arg);
    }
    return rc;
}

//  -----------------------------------------------------------------
----
```

```
//  Reactor event handlers…

//  Publish our state to peer
int s_send_state (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    zstr_sendf (self->statepub, "%d", self->state);
    return 0;
}

//  Receive state from peer, execute finite state machine
int s_recv_state (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    char *state = zstr_recv (socket);
    if (state) {
        self->event = atoi (state);
        self->peer_expiry = zclock_time () + 2 * BSTAR_HEARTBEAT;
        free (state);
    }
    return s_execute_fsm (self);
}

//  Application wants to speak to us, see if it's possible
int s_voter_ready (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    //  If server can accept input now, call appl handler
    self->event = CLIENT_REQUEST;
    if (s_execute_fsm (self) == 0) {
        puts ("CLIENT REQUEST");
        (self->voter_fn) (self->loop, socket, self->voter_arg);
    }
    else {
        //  Destroy waiting message, no-one to read it
        zmsg_t *msg = zmsg_recv (socket);
        zmsg_destroy (&msg);
    }
    return 0;
}

//  -----------------------------------------------------------------
----
//  Constructor

bstar_t *
bstar_new (int primary, char *local, char *remote)
{
    bstar_t
        *self;

    self = (bstar_t *) zmalloc (sizeof (bstar_t));

    //  Initialize the Binary Star
    self->ctx = zctx_new ();
    self->loop = zloop_new ();
    self->state = primary? STATE_PRIMARY: STATE_BACKUP;

    //  Create publisher for state going to peer
    self->statepub = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->statepub, local);

    //  Create subscriber for state coming from peer
    self->statesub = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_connect (self->statesub, remote);

    //  Set-up basic reactor events
    zloop_timer (self->loop, BSTAR_HEARTBEAT, 0, s_send_state, self);
    zloop_reader (self->loop, self->statesub, s_recv_state, self);
    return self;
}

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

void
bstar_destroy (bstar_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        bstar_t *self = *self_p;
        zloop_destroy (&self->loop);
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  -----------------------------------------------------------------
----
//  Return underlying zloop reactor, lets you add additional timers
and
//  readers.

zloop_t *
bstar_zloop (bstar_t *self)
{
    return self->loop;
}

//  -----------------------------------------------------------------
----
//  Create socket, bind to local endpoint, and register as reader for
//  voting. The socket will only be available if the Binary Star
state
//  machine allows it. Input on the socket will act as a "vote" in
the
//  Binary Star scheme.  We require exactly one voter per bstar
instance.

int
bstar_voter (bstar_t *self, char *endpoint, int type, zloop_fn
handler,
             void *arg)
{
    //  Hold actual handler+arg so we can call this later
    void *socket = zsocket_new (self->ctx, type);
    zsocket_bind (socket, endpoint);
    assert (!self->voter_fn);
    self->voter_fn = handler;
    self->voter_arg = arg;
    return zloop_reader (self->loop, socket, s_voter_ready, self);
}

//  ---------------------------------------------------------------------
----
//  Register state change handlers

void
bstar_new_master (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->master_fn);
    self->master_fn = handler;
    self->master_arg = arg;
}

void
bstar_new_slave (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->slave_fn);
    self->slave_fn = handler;
    self->slave_arg = arg;
}

//  ---------------------------------------------------------------------
----
//  Enable/disable verbose tracing
void bstar_set_verbose (bstar_t *self, Bool verbose)
{
    zloop_set_verbose (self->loop, verbose);
}

//  ---------------------------------------------------------------------
----
//  Start the reactor, ends if a callback function returns -1, or the
//  process received SIGINT or SIGTERM.

int
bstar_start (bstar_t *self)
{
    assert (self->voter_fn);
    return zloop_start (self->loop);
}
```

*bstar.c: Binary Star core class*

Which gives us the following short main program for the server:

```
//
//  Binary Star server, using bstar reactor
//

//  Lets us build this source without creating a library
#include "bstar.c"

//  Echo service
int s_echo (zloop_t *loop, void *socket, void *arg)
{
    zmsg_t *msg = zmsg_recv (socket);
    zmsg_send (&msg, socket);
    return 0;
}

int main (int argc, char *argv [])
{
    //  Arguments can be either of:
    //      -p  primary server, at tcp://localhost:5001
    //      -b  backup server, at tcp://localhost:5002
    bstar_t *bstar;
    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: Primary master, waiting for backup (slave)\n");
        bstar = bstar_new (BSTAR_PRIMARY,
            "tcp://*:5003", "tcp://localhost:5004");
        bstar_voter (bstar, "tcp://*:5001", ZMQ_ROUTER, s_echo,
NULL);
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: Backup slave, waiting for primary (master)\n");
        bstar = bstar_new (BSTAR_BACKUP,
            "tcp://*:5004", "tcp://localhost:5003");
        bstar_voter (bstar, "tcp://*:5002", ZMQ_ROUTER, s_echo,
NULL);
    }
    else {
        printf ("Usage: bstarsrvs { -p | -b }\n");
        exit (0);
    }
    bstar_start (bstar);
    bstar_destroy (&bstar);
    return 0;
}
```

*bstarsrv2.c: Binary Star server, using core class*

## Brokerless Reliability (Freelance Pattern)

top prev next

It might seem ironic to focus so much on broker-based reliability, when we often explain ØMQ as "brokerless messaging". However in messaging, as in real life, the middleman is both a burden and a benefit. In practice, most messaging architectures benefit from a mix of distributed and brokered messaging. You get the best results when you can decide freely what tradeoffs you want to make. This is why I can drive 10km to a wholesaler to buy five cases of wine for a party, but I can also walk 10 minutes to a corner store to buy one bottle for a dinner. Our highly context-sensitive relative valuations of time, energy, and cost are essential to the real world economy. And they are essential to an optimal message based architecture.

Which is why ØMQ does not *impose* a broker-centric architecture, though it gives you the tools to build brokers, aka "devices", and we've built a dozen or so different ones so far, just for practice.

So we'll end this chapter by deconstructing the broker-based reliability we've built so far, and turning it back into a distributed peer-to-peer architecture I call the Freelance pattern. Our use case will be a name resolution service. This is a common problem with ØMQ architectures: how do we know the endpoint to connect to? Hard-coding TCP/IP addresses in code is insanely fragile. Using configuration files creates an administration nightmare. Imagine if you had to hand-configure your web browser, on every PC or mobile phone you used, to realize that "google.com" was "74.125.230.82".

A ØMQ name service (and we'll make a simple implementation) has to:

- Resolve a logical name into at least a bind endpoint, and a connect endpoint. A realistic name service would provide multiple bind endpoints, and possibly multiple connect endpoints too.
- Allow us to manage multiple parallel environments, e.g. "test" vs. "production" without modifying code.
- Be reliable, because if it is unavailable, applications won't be able to connect to the network.

Putting a name service behind a service-oriented Majordomo broker is clever from some points of view. However it's simpler and much less surprising to just expose the name service as a server that clients can connect to directly. If we do this right, the name service becomes the *only* global network endpoint we need to hard-code in our code or config files.

The types of failure we aim to handle are server crashes and restarts, server busy looping, server overload, and network issues. To get reliability, we'll create a pool of name servers so if one crashes or goes away, clients can connect to another, and so on. In practice, two would be enough. But for the example, we'll assume the pool can be any size:

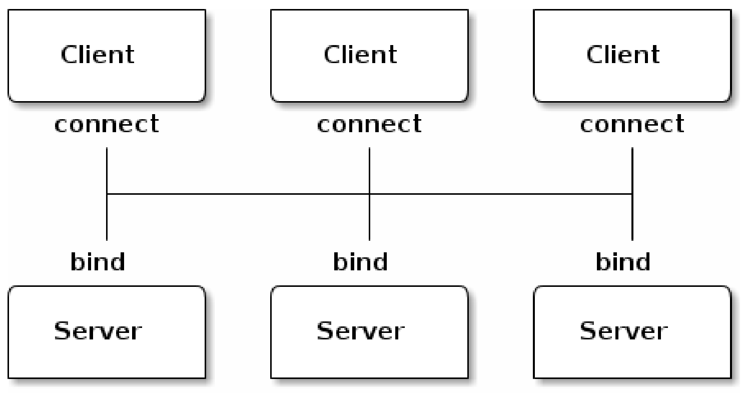


Figure 65 — The Freelance Pattern

In this architecture a large set of clients connect to a small set of servers directly. The servers bind to their respective addresses. It's fundamentally different from a broker-based approach like Majordomo, where workers connect to the broker. For clients, there are a couple of options:

- Clients could use REQ sockets and the Lazy Pirate pattern. Easy, but would need some additional intelligence to not stupidly reconnect to dead servers over and over.

- Clients could use DEALER sockets and blast out requests (which will be load balanced to all connected servers) until they get a reply. Brutal, but not elegant.

- Clients could use ROUTER sockets so they can address specific servers. But how does the client know the identity of the server sockets? Either the server has to ping the client first (complex), or the each server has to use a hard-coded, fixed identity known to the client (nasty).

## Model One - Simple Retry and Failover

top prev next

So our menu appears to offer: simple, brutal, complex, or nasty. Let's start with 'simple' and then work out the kinks. We take Lazy Pirate and rewrite it to work with multiple server endpoints. Start the server first, specifying a bind endpoint as argument. Run one or several servers:

```
//
//  Freelance server - Model 1
//  Trivial echo service
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: echo service is ready at %s\n", argv [1]);
    while (TRUE) {
        zmsg_t *msg = zmsg_recv (server);
        if (!msg)
            break;          //  Interrupted
        zmsg_send (&msg, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

*flserver1.c: Freelance server, Model One*

Then start the client, specifying one or more connect endpoints as arguments:

```
//
//  Freelance client - Model 1
//  Uses REQ socket to query one or more services
//
#include "czmq.h"

#define REQUEST_TIMEOUT     1000
#define MAX_RETRIES         3       //  Before we abandon

static zmsg_t *
s_try_request (zctx_t *ctx, char *endpoint, zmsg_t *request)
{
    printf ("I: trying echo service at %s…\n", endpoint);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, endpoint);

    //  Send request, wait safely for reply
    zmsg_t *msg = zmsg_dup (request);
    zmsg_send (&msg, client);
    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
    zmsg_t *reply = NULL;
    if (items [0].revents & ZMQ_POLLIN)
        reply = zmsg_recv (client);

    //  Close socket in any case, we're done with it now
    zsocket_destroy (ctx, client);
    return reply;
}

int main (int argc, char *argv [])
{
    zctx_t *ctx = zctx_new ();
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = NULL;

    int endpoints = argc - 1;
    if (endpoints == 0)
        printf ("I: syntax: %s <endpoint> …\n", argv [0]);
    else
    if (endpoints == 1) {
        //  For one endpoint, we retry N times
        int retries;
        for (retries = 0; retries < MAX_RETRIES; retries++) {
            char *endpoint = argv [1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  Successful
            printf ("W: no response from %s, retrying…\n", endpoint);
        }
    }
    else {
        //  For multiple endpoints, try each at most once
        int endpoint_nbr;
        for (endpoint_nbr = 0; endpoint_nbr < endpoints;
endpoint_nbr++) {
            char *endpoint = argv [endpoint_nbr + 1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  Successful
            printf ("W: no response from %s\n", endpoint);
        }
    }
    if (reply)
        printf ("Service is running OK\n");

    zmsg_destroy (&request);
    zmsg_destroy (&reply);
    zctx_destroy (&ctx);
    return 0;
}
```

*flclient1.c: Freelance client, Model One*

For example:

```
flserver1 tcp://*:5555 &
```

```
flserver1 tcp://*:5556 &
flclient1 tcp://localhost:5555 tcp://localhost:5556
```

While the basic approach is Lazy Pirate, the client aims to just get one successful reply. It has two techniques, depending on whether you are running a single server, or multiple servers:

- With a single server, the client will retry several times, exactly as for Lazy Pirate.
- With multiple servers, the client will try each server at most once, until it's received a reply, or has tried all servers.

This solves the main weakness of Lazy Pirate, namely that it could not do failover to backup / alternate servers.

However this design won't work well in a real application. If we're connecting many sockets, and our primary name server is down, we're going to do this painful timeout each time.

## Model Two - Brutal Shotgun Massacre

top prev next

Let's switch our client to using a DEALER socket. Our goal here is to make sure we get a reply back within the shortest possible time, no matter whether the primary server is down or not. Our client takes this approach:

- We set things up, connecting to all servers.
- When we have a request, we blast it out as many times as we have servers.
- We wait for the first reply, and take that.
- We ignore any other replies.

What will happen in practice is that when all servers are running, ØMQ will distribute the requests so each server gets one request, and sends one reply. When any server is offline, and disconnected, ØMQ will distribute the requests to the remaining servers. So a server may in some cases get the same request more than once.

What's more annoying for the client is that we'll get multiple replies back, but there's no guarantee we'll get a precise number of replies. Requests and replies can get lost (e.g. if the server crashes while processing a request).

So, we have to number requests, and ignore any replies that don't match the request number. Our Model One server will work, since it's an echo server, but coincidence is not a great basis for understanding. So we'll make a Model Two server that chews up the message, returns a correctly-numbered reply with the content "OK". We'll use messages consisting of two parts, a sequence number and a body.

Start the server once or more, specifying a bind endpoint each time:

```
//
//  Freelance server - Model 2
//  Does some work, replies OK, with message sequencing
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: service is ready at %s\n", argv [1]);
    while (TRUE) {
        zmsg_t *request = zmsg_recv (server);
        if (!request)
            break;          //  Interrupted
        //  Fail nastily if run against wrong client
        assert (zmsg_size (request) == 2);

        zframe_t *address = zmsg_pop (request);
        zmsg_destroy (&request);

        zmsg_t *reply = zmsg_new ();
        zmsg_add (reply, address);
        zmsg_addstr (reply, "OK");
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

*flserver2.c: Freelance server, Model Two*

Then start the client, specifying the connect endpoints as arguments:

```
//
//  Freelance client - Model 2
//  Uses DEALER socket to blast one or more services
//
#include "czmq.h"

//  If not a single service replies within this time, give up
#define GLOBAL_TIMEOUT 2500

//  We design our client API as a class

#ifdef __cplusplus
extern "C" {
#endif

//  Opaque class structure
typedef struct _flclient_t flclient_t;

flclient_t *
    flclient_new (void);
void
    flclient_destroy (flclient_t **self_p);
void
    flclient_connect (flclient_t *self, char *endpoint);
zmsg_t *
    flclient_request (flclient_t *self, zmsg_t **request_p);

#ifdef __cplusplus
```

```
}
#endif

int main (int argc, char *argv [])
{
    if (argc == 1) {
        printf ("I: syntax: %s <endpoint> …\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    //  Create new freelance client object
    flclient_t *client = flclient_new ();

    //  Connect to each endpoint
    int argn;
    for (argn = 1; argn < argc; argn++)
        flclient_connect (client, argv [argn]);

    //  Send a bunch of name resolution 'requests', measure time
    int requests = 10000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flclient_request (client, &request);
        if (!reply) {
            printf ("E: name service not available, aborting\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("Average round trip cost: %d usec\n",
        (int) (zclock_time () - start) / 10);

    flclient_destroy (&client);
    return 0;
}

//  ---------------------------------------------------------------------
---
//  Structure of our class

struct _flclient_t {
    zctx_t *ctx;        //  Our context wrapper
    void *socket;       //  DEALER socket talking to servers
    size_t servers;     //  How many servers we have connected to
    uint sequence;      //  Number of requests ever sent
};

//  ---------------------------------------------------------------------
---
//  Constructor

flclient_t *
flclient_new (void)
{
    flclient_t
        *self;

    self = (flclient_t *) zmalloc (sizeof (flclient_t));
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_DEALER);
    return self;
}

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

void
flclient_destroy (flclient_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flclient_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  -----------------------------------------------------------------
---
//  Connect to new server endpoint

void
flclient_connect (flclient_t *self, char *endpoint)
{
    assert (self);
    zsocket_connect (self->socket, endpoint);
    self->servers++;
}

//  -----------------------------------------------------------------
---
//  Send request, get reply
//  Destroys request after sending

zmsg_t *
flclient_request (flclient_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);
    zmsg_t *request = *request_p;

    //  Prefix request with sequence number and empty envelope
    char sequence_text [10];
    sprintf (sequence_text, "%u", ++self->sequence);
    zmsg_pushstr (request, sequence_text);
    zmsg_pushstr (request, "");

    //  Blast the request to all connected servers
    int server;
    for (server = 0; server < self->servers; server++) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->socket);
    }
    //  Wait for a matching reply to arrive from anywhere
    //  Since we can poll several times, calculate each one
    zmsg_t *reply = NULL;
    uint64_t endtime = zclock_time () + GLOBAL_TIMEOUT;
    while (zclock_time () < endtime) {
        zmq_pollitem_t items [] = { { self->socket, 0, ZMQ_POLLIN, 0
} };
        zmq_poll (items, 1, (endtime - zclock_time ()) *
ZMQ_POLL_MSEC);
        if (items [0].revents & ZMQ_POLLIN) {
            //  Reply is [empty][sequence][OK]
            reply = zmsg_recv (self->socket);
            assert (zmsg_size (reply) == 3);
            free (zmsg_popstr (reply));
            char *sequence = zmsg_popstr (reply);
            int sequence_nbr = atoi (sequence);
            free (sequence);
            if (sequence_nbr == self->sequence)
                break;
        }
    }
    zmsg_destroy (request_p);
    return reply;
}
```

*flclient2.c: Freelance client, Model Two*

Some notes on this code:

- The client is structured as a nice little class-based API that hides the dirty work of creating ØMQ contexts and sockets, and talking to the server. If a shotgun blast to the midriff can be called "talking".
- The client will abandon the chase if it can't find *any* responsive server within a few seconds.
- The client has to create a valid REP envelope, i.e. add an empty message part to the front of the message.

The client does 10,000 name resolution requests (fake ones, since our server does essentially nothing), and measures the average cost. On my test box, talking to one server, it's about 60 usec. Talking to three servers, it's about 80 usec.

So pros and cons of our shotgun approach:

- Pro: it is simple, easy to make and easy to understand.
- Pro: it does the job of failover, and works rapidly, so long as there is at least one server running.
- Con: it creates redundant network traffic.
- Con: we can't prioritize our servers, i.e. Primary, then Secondary.
- Con: the server can do at most one request at a time, period.

## Model Three - Complex and Nasty

top prev next

The shotgun approach seems too good to be true. Let's be scientific and work through all the alternatives. We're going to explore the complex/nasty option, even if it's only to finally realize that we preferred brutal. Ah, the story of my life.

We can solve the main problems of the client by switching to a ROUTER socket. That lets us send requests to specific servers, avoid servers we know are dead, and in general be as smart as we want to make it. We can also solve the main problem of the server (single threadedness) by switching to a ROUTER socket.

But doing ROUTER-to-ROUTER between two transient sockets is not possible. Both sides generate an identity (at their own end of the conversation) only when they receive a first message, and thus neither can talk to the other until it has first received a message. The only way out of this conundrum is to cheat, and use hard-coded identities in one direction. The proper way to cheat, in a client server case, is that the client 'knows' the identity of the server. Vice-versa would be insane, on top of complex and nasty. Great attributes for a genocidal dictator, terrible ones for software.

Rather than invent yet another concept to manage, we'll use the connection endpoint as identity. This is a unique string both sides can agree on without more prior knowledge than they already have for the shotgun model. It's a sneaky and effective way to connect two ROUTER sockets.

Remember how ØMQ identities work. The server ROUTER socket sets an identity before it binds its socket. When a client connects, they do a little handshake to exchange identities, before either side sends a real message. The client ROUTER socket, having not set an identity, sends a null identity to the server. The server generates a random UUID for the client, for its own use. The server sends its identity (which we've agreed is going to be an endpoint string) to the client.

This means our client can route a message to the server (i.e. send on its ROUTER socket, specifying the server endpoint as identity) as soon as the connection is established. That's not *immediately* after doing a zmq_connect, but some random time thereafter. Herein lies one problem: we don't know when the server will actually be available and complete its connection handshake. If the server is actually online, it could be after a few milliseconds. If the server is down, and the sysadmin is out to lunch, it could be an hour.

There's a small paradox here. We need to know when servers become connected and available for work. In the Freelance pattern, unlike the broker-based patterns we saw earlier in this chapter, servers are silent until spoken to. Thus we can't talk to a server until it's told us it's on-line, which it can't do until we've asked it.

My solution is to mix in a little of the shotgun approach from model 2, meaning we'll fire (harmless) shots at anything we can, and if anything moves, we know it's alive. We're not going to fire real requests, but rather a kind of ping-pong heartbeat.

This brings us to the realm of protocols again, so here's a short spec that defines how a Freelance client and server exchange PING-PONG commands, and request-reply commands:

- http://rfc.zeromq.org/spec:10

It is short and sweet to implement as a server. Here's our echo server, Model Three, now speaking FLP.

Model Three of the server is just slightly different:

```
//
//  Freelance server - Model 3
//  Uses an ROUTER/ROUTER socket but just one thread
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    zctx_t *ctx = zctx_new ();

    //  Prepare server socket with predictable identity
    char *bind_endpoint = "tcp://*:5555";
    char *connect_endpoint = "tcp://localhost:5555";
    void *server = zsocket_new (ctx, ZMQ_ROUTER);
    zmq_setsockopt (server,
        ZMQ_IDENTITY, connect_endpoint, strlen (connect_endpoint));
    zsocket_bind (server, bind_endpoint);
    printf ("I: service is ready at %s\n", bind_endpoint);

    while (!zctx_interrupted) {
        zmsg_t *request = zmsg_recv (server);
        if (verbose && request)
            zmsg_dump (request);
        if (!request)
            break;          //  Interrupted

        //  Frame 0: identity of client
        //  Frame 1: PING, or client control frame
        //  Frame 2: request body
        zframe_t *address = zmsg_pop (request);
        zframe_t *control = zmsg_pop (request);
        zmsg_t *reply = zmsg_new ();
        if (zframe_streq (control, "PONG"))
            zmsg_addstr (reply, "PONG");
        else {
            zmsg_add (reply, control);
            zmsg_addstr (reply, "OK");
        }
        zmsg_destroy (&request);
        zmsg_push (reply, address);
        if (verbose && reply)
            zmsg_dump (reply);
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

*flserver3.c: Freelance server, Model Three*

The Freelance client, however, has gotten large. For clarity, it's split into an example application and a class that does the hard work. Here's the top-level application:

```
//
//  Freelance client - Model 3
//  Uses flcliapi class to encapsulate Freelance pattern
//
//  Lets us build this source without creating a library
#include "flcliapi.c"

int main (void)
{
    //  Create new freelance client object
    flcliapi_t *client = flcliapi_new ();

    //  Connect to several endpoints
    flcliapi_connect (client, "tcp://localhost:5555");
    flcliapi_connect (client, "tcp://localhost:5556");
    flcliapi_connect (client, "tcp://localhost:5557");
    //  Send a bunch of name resolution 'requests', measure time
    int requests = 1000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flcliapi_request (client, &request);
        if (!reply) {
            printf ("E: name service not available, aborting\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("Average round trip cost: %d usec\n",
        (int) (zclock_time () - start) / 10);

puts ("flclient 1");
    flcliapi_destroy (&client);
puts ("flclient 2");
    return 0;
}
```

*flclient3.c: Freelance client, Model Three*

And here, almost as complex and large as the Majordomo broker, is the client API class:

```
/*
=====================================================================
    flcliapi - Freelance Pattern agent class
    Model 3: uses ROUTER socket to address specific services

    -----------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.

=====================================================================
*/
```

```
#include "flcliapi.h"

//  If no server replies within this time, abandon request
#define GLOBAL_TIMEOUT  3000    //  msecs
//  PING interval for servers we think are alive
#define PING_INTERVAL   2000    //  msecs
//  Server considered dead if silent for this long
#define SERVER_TTL      6000    //  msecs

//
=====================================================================
//  Synchronous part, works in our application thread

//  -----------------------------------------------------------------
----
//  Structure of our class

struct _flcliapi_t {
    zctx_t *ctx;        //  Our context wrapper
    void *pipe;         //  Pipe through to flcliapi agent
};

//  This is the thread that handles our real flcliapi class
static void flcliapi_agent (void *args, zctx_t *ctx, void *pipe);

//  -----------------------------------------------------------------
----
//  Constructor

flcliapi_t *
flcliapi_new (void)
{
    flcliapi_t
        *self;

    self = (flcliapi_t *) zmalloc (sizeof (flcliapi_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, flcliapi_agent, NULL);
    return self;
}

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

void
flcliapi_destroy (flcliapi_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flcliapi_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  -----------------------------------------------------------------
----
//  Connect to new server endpoint
//  Sends [CONNECT][endpoint] to the agent

void
flcliapi_connect (flcliapi_t *self, char *endpoint)
{
    assert (self);
    assert (endpoint);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, endpoint);
    zmsg_send (&msg, self->pipe);
    zclock_sleep (100);      //  Allow connection to come up
}

//  ---------------------------------------------------------------
----
//  Send & destroy request, get reply

zmsg_t *
flcliapi_request (flcliapi_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);

    zmsg_pushstr (*request_p, "REQUEST");
    zmsg_send (request_p, self->pipe);
    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *status = zmsg_popstr (reply);
        if (streq (status, "FAILED"))
            zmsg_destroy (&reply);
        free (status);
    }
    return reply;
}

//
=====================================================================
//  Asynchronous part, works in the background

//  ---------------------------------------------------------------
----
//  Simple class for one server we talk to

typedef struct {
    char *endpoint;             //  Server identity/endpoint
    uint alive;                 //  1 if known to be alive
    int64_t ping_at;            //  Next ping at this time
    int64_t expires;            //  Expires at this time
} server_t;

server_t *
server_new (char *endpoint)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));
    self->endpoint = strdup (endpoint);
    self->alive = 0;
    self->ping_at = zclock_time () + PING_INTERVAL;
    self->expires = zclock_time () + SERVER_TTL;
    return self;
```

```
}

void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->endpoint);
        free (self);
        *self_p = NULL;
    }
}

int
server_ping (char *key, void *server, void *socket)
{
    server_t *self = (server_t *) server;
    if (zclock_time () >= self->ping_at) {
        zmsg_t *ping = zmsg_new ();
        zmsg_addstr (ping, self->endpoint);
        zmsg_addstr (ping, "PING");
        zmsg_send (&ping, socket);
        self->ping_at = zclock_time () + PING_INTERVAL;
    }
    return 0;
}

int
server_tickless (char *key, void *server, void *arg)
{
    server_t *self = (server_t *) server;
    uint64_t *tickless = (uint64_t *) arg;
    if (*tickless > self->ping_at)
        *tickless = self->ping_at;
    return 0;
}

//  ---------------------------------------------------------------------
----
//  Simple class for one background agent

typedef struct {
    zctx_t *ctx;                //  Own context
    void *pipe;                 //  Socket to talk back to
application
    void *router;               //  Socket to talk to servers
    zhash_t *servers;           //  Servers we've connected to
    zlist_t *actives;           //  Servers we know are alive
    uint sequence;              //  Number of requests ever sent
    zmsg_t *request;            //  Current request if any
    zmsg_t *reply;              //  Current reply if any
    int64_t expires;            //  Timeout for request/reply
} agent_t;

agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->router = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->servers = zhash_new ();
    self->actives = zlist_new ();
    return self;
}

void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        zhash_destroy (&self->servers);
        zlist_destroy (&self->actives);
        zmsg_destroy (&self->request);
        zmsg_destroy (&self->reply);
        free (self);
        *self_p = NULL;
    }
}

//  Callback when we remove server from agent 'servers' hash table

static void
s_server_free (void *argument)
{
    server_t *server = (server_t *) argument;
    server_destroy (&server);
}

void
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);

    if (streq (command, "CONNECT")) {
        char *endpoint = zmsg_popstr (msg);
        printf ("I: connecting to %s…\n", endpoint);
        int rc = zmq_connect (self->router, endpoint);
        assert (rc == 0);
        server_t *server = server_new (endpoint);
        zhash_insert (self->servers, endpoint, server);
        zhash_freefn (self->servers, endpoint, s_server_free);
        zlist_append (self->actives, server);
        server->ping_at = zclock_time () + PING_INTERVAL;
        server->expires = zclock_time () + SERVER_TTL;
        free (endpoint);
    }
    else
    if (streq (command, "REQUEST")) {
        assert (!self->request);    //  Strict request-reply cycle
        //  Prefix request with sequence number and empty envelope
        char sequence_text [10];
        sprintf (sequence_text, "%u", ++self->sequence);
        zmsg_pushstr (msg, sequence_text);
        //  Take ownership of request message
        self->request = msg;
        msg = NULL;
```

```
            //  Request expires after global timeout
            self->expires = zclock_time () + GLOBAL_TIMEOUT;
        }
        free (command);
        zmsg_destroy (&msg);
    }

    void
    agent_router_message (agent_t *self)
    {
        zmsg_t *reply = zmsg_recv (self->router);

        //  Frame 0 is server that replied
        char *endpoint = zmsg_popstr (reply);
        server_t *server =
            (server_t *) zhash_lookup (self->servers, endpoint);
        assert (server);
        free (endpoint);
        if (!server->alive) {
            zlist_append (self->actives, server);
            server->alive = 1;
        }
        server->ping_at = zclock_time () + PING_INTERVAL;
        server->expires = zclock_time () + SERVER_TTL;

        //  Frame 1 may be sequence number for reply
        char *sequence = zmsg_popstr (reply);
        if (atoi (sequence) == self->sequence) {
            zmsg_pushstr (reply, "OK");
            zmsg_send (&reply, self->pipe);
            zmsg_destroy (&self->request);
        }
        else
            zmsg_destroy (&reply);
    }

    //  -----------------------------------------------------------------
    ----
    //  Asynchronous agent manages server pool and handles request/reply
    //  dialog when the application asks for it.

    static void
    flcliapi_agent (void *args, zctx_t *ctx, void *pipe)
    {
        agent_t *self = agent_new (ctx, pipe);

        zmq_pollitem_t items [] = {
            { self->pipe, 0, ZMQ_POLLIN, 0 },
            { self->router, 0, ZMQ_POLLIN, 0 }
        };
        while (!zctx_interrupted) {
            //  Calculate tickless timer, up to 1 hour
            uint64_t tickless = zclock_time () + 1000 * 3600;
            if (self->request
            &&  tickless > self->expires)
                tickless = self->expires;
            zhash_foreach (self->servers, server_tickless, &tickless);

            int rc = zmq_poll (items, 2,
                (tickless - zclock_time ()) * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;              //  Context has been shut down

            if (items [0].revents & ZMQ_POLLIN)
                agent_control_message (self);

            if (items [1].revents & ZMQ_POLLIN)
                agent_router_message (self);

            //  If we're processing a request, dispatch to next server
            if (self->request) {
                if (zclock_time () >= self->expires) {
                    //  Request expired, kill it
                    zstr_send (self->pipe, "FAILED");
                    zmsg_destroy (&self->request);
                }
                else {
                    //  Find server to talk to, remove any expired ones
                    while (zlist_size (self->actives)) {
                        server_t *server =
                            (server_t *) zlist_first (self->actives);
                        if (zclock_time () >= server->expires) {
                            zlist_pop (self->actives);
                            server->alive = 0;
                        }
                        else {
                            zmsg_t *request = zmsg_dup (self->request);
                            zmsg_pushstr (request, server->endpoint);
                            zmsg_send (&request, self->router);
                            break;
                        }
                    }
                }
            }
            //  Disconnect and delete any expired servers
            //  Send heartbeats to idle servers if needed
            zhash_foreach (self->servers, server_ping, self->router);
        }
        agent_destroy (&self);
    }
```

*flcliapi.c: Freelance client API*

This API implementation is fairly sophisticated and uses a number of techniques that we've not seen before:

**Asynchronous agent class**

The client API consists of two parts, a synchronous 'flcliapi' class that runs in the application thread, and an asynchronous 'agent' class that runs in the background. The flcliapi and agent classes talk to each other over an `inproc` socket. All ØMQ aspects (such as creating and terminating a context) are hidden in the API. The agent in effect acts like a mini-broker, talking to servers in the background, so that when we make a request, it can make a best effort to reach a server it believes is available.

**Patient connections**

ROUTER sockets have the feature of silently dropping messages they can't route. This means if you connect a client to a server, ROUTER-to-ROUTER, and then immediately try to send a message, it won't work. The flcliapi class does a short sleep when the application does an initial connect to a server. Thereafter, since these are durable sockets, ØMQ will never throw away messages to a server it's seen, even if the server does go away.

**Ping silence**

ØMQ will queue messages for a dead server indefinitely. So if a client repeatedly PINGs a dead server, when that server comes back to life it'll get a whole bunch of PING messages all at once. Rather than continuing to ping a server we know is offline, we count on ØMQ's handling of durable sockets to deliver the old PING messages when the server comes back online. As soon as a server reconnects, it'll get PINGs from all clients that were connected to it, it'll PONG back, and those clients will recognize it as alive again.

**Tickless poll timer**

In previous poll loops we always used a fixed tick interval, e.g. 1 second, which is simple enough but not excellent on power-sensitive clients, such as notebooks or mobile phones, where waking the CPU costs power. For fun, and to help save the planet, the agent uses a 'tickless timer', which calculates the poll delay based on the next timeout we're expecting. A proper implementation would keep an ordered list of timeouts. We just check all timeouts and calculate the poll delay until the next one.

## Conclusion

top prev next

In this chapter we've seen a variety of reliable request-reply mechanisms, each with certain costs and benefits. The example code is largely ready for real use, though it is not optimized. Of all the different patterns, the two that stand out are the Majordomo pattern, for broker-based reliability, and the Freelance pattern for brokerless reliability.

# Chapter Five - Advanced Publish-Subscribe

top prev next

In Chapters Three and Four we looked at advanced use of ØMQ's request-reply pattern. If you managed to digest all that, congratulations. In this chapter we'll focus on publish-subscribe, and extend ØMQ's core pub-sub pattern with higher-level patterns for performance, reliability, state distribution, and security.

We'll cover:

- How to handle too-slow subscribers (the *Suicidal Snail* pattern).
- How to design high-speed subscribers (the *Black Box* pattern).
- How to build a shared key-value cache (the *Clone* pattern).

## Slow Subscriber Detection (Suicidal Snail Pattern)

top prev next

A common problem you will hit when using the pub-sub pattern in real life is the slow subscriber. In an ideal world, we stream data at full speed from publishers to subscribers. In reality, subscriber applications are often written in interpreted languages, or just do a lot of work, or are just badly written, to the extent that they can't keep up with publishers.

How do we handle a slow subscriber? The ideal fix is to make the subscriber faster, but that might take work and time. Some of the classic strategies for handling a slow subscriber are:

- **Queue messages on the publisher**. This is what Gmail does when I don't read my email for a couple of hours. But in high-volume messaging, pushing queues upstream has the thrilling but unprofitable result of making publishers run out of memory and crash. Especially if there are lots of subscribers and it's not possible to flush to disk for performance reasons.

- **Queue messages on the subscriber**. This is much better, and it's what ØMQ does by default if the network can keep up with things. If anyone's going to run out of memory and crash, it'll be the subscriber rather than the publisher, which is fair. This is perfect for "peaky" streams where a subscriber can't keep up for a while, but can catch up when the stream slows down. However it's no answer to a subscriber that's simply too slow in general.

- **Stop queuing new messages after a while**. This is what Gmail does when my mailbox overflows its 7.554GB, no 7.555GB of space. New messages just get rejected or dropped. This is a great strategy from the perspective of the publisher, and it's what ØMQ does when the publisher sets a high water mark or HWM. However it still doesn't help us fix the slow subscriber. Now we just get gaps in our message stream.

- **Punish slow subscribers with disconnect**. This is what Hotmail does when I don't login for two weeks, which is why I'm on my fifteenth Hotmail account. It's a nice brutal strategy that forces subscribers to sit up and pay attention, and would be ideal, but ØMQ doesn't do this, and there's no way to layer it on top since subscribers are invisible to publisher applications.

None of these classic strategies fit. So we need to get creative. Rather than disconnect the publisher, let's convince the subscriber to kill itself. This is the Suicidal Snail pattern. When a subscriber detects that it's running too slowly (where "too slowly" is presumably a configured option that really means "so slowly that if you ever get here, shout really loudly because I need to know, so I can fix this!"), it croaks and dies.

How can a subscriber detect this? One way would be to sequence messages (number them in order), and use a HWM at the publisher. Now, if the subscriber detects a gap (i.e. the numbering isn't consecutive), it knows something is wrong. We then tune the HWM to the "croak and die if you hit this" level.

There are two problems with this solution. One, if we have many publishers, how do we sequence messages? The solution is to give each publisher a unique ID and add that to the sequencing. Second, if subscribers use ZMQ_SUBSCRIBE filters, they will get gaps by definition. Our precious sequencing will be for nothing.

Some use-cases won't use filters, and sequencing will work for them. But a more general solution is that the publisher timestamps each message. When a subscriber gets a message it checks the time, and if the difference is more than, say, one second, it does the "croak and die" thing. Possibly firing off a squawk to some operator console first.

The Suicide Snail pattern works especially when subscribers have their own clients and service-level agreements and need to guarantee certain maximum latencies. Aborting a subscriber may not seem like a constructive way to guarantee a maximum latency, but it's the assertion model. Abort today, and the problem will be fixed. Allow late data to flow downstream, and the problem may cause wider damage and take longer to appear on the radar.

So here is a minimal example of a Suicidal Snail:

```
//
//  Suicidal Snail
//
#include "czmq.h"

//  ---------------------------------------------------------------------
```

```
----
//  This is our subscriber
//  It connects to the publisher and subscribes to everything. It
//  sleeps for a short time between messages to simulate doing too
//  much work. If a message is more than 1 second late, it croaks.

#define MAX_ALLOWED_DELAY   1000    //  msecs

static void
subscriber (void *args, zctx_t *ctx, void *pipe)
{
    //  Subscribe to everything
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5556");

    //  Get and process messages
    while (1) {
        char *string = zstr_recv (subscriber);
        int64_t clock;
        int terms = sscanf (string, "%" PRId64, &clock);
        assert (terms == 1);
        free (string);

        //  Suicide snail logic
        if (zclock_time () - clock > MAX_ALLOWED_DELAY) {
            fprintf (stderr, "E: subscriber cannot keep up,
aborting\n");
            break;
        }
        //  Work for 1 msec plus some random additional time
        zclock_sleep (1 + randof (2));
    }
    zstr_send (pipe, "gone and died");
}

//  -----------------------------------------------------------------
----
//  This is our server task
//  It publishes a time-stamped message to its pub socket every 1ms.

static void
publisher (void *args, zctx_t *ctx, void *pipe)
{
    //  Prepare publisher
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");

    while (1) {
        //  Send current clock (msecs) to subscribers
        char string [20];
        sprintf (string, "%" PRId64, zclock_time ());
        zstr_send (publisher, string);
        char *signal = zstr_recv_nowait (pipe);
        if (signal) {
            free (signal);
            break;
        }
        zclock_sleep (1);           //  1msec wait
    }
}
```

```
//  This main thread simply starts a client, and a server, and then
//  waits for the client to signal it's died.
//
int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *pubpipe = zthread_fork (ctx, publisher, NULL);
    void *subpipe = zthread_fork (ctx, subscriber, NULL);
    free (zstr_recv (subpipe));
    zstr_send (pubpipe, "break");
    zclock_sleep (100);
    zctx_destroy (&ctx);
    return 0;
}
```

*suisnail.c: Suicidal Snail*

Notes about this example:

- The message here consists simply of the current system clock as a number of milliseconds. In a realistic application you'd have at least a message header with the timestamp, and a message body with data.
- The example has subscriber and publisher in a single process, as two threads. In reality they would be separate processes. Using threads is just convenient for the demonstration.

## High-speed Subscribers (Black Box Pattern)

top prev next

A common use-case for pub-sub is distributing large data streams. For example, 'market data' coming from stock exchanges. A typical set-up would have a publisher connected to a stock exchange, taking price quotes, and sending them out to a number of subscribers. If there are a handful of subscribers, we could use TCP. If we have a larger number of subscribers, we'd probably use reliable multicast, i.e. `pgm`.

Let's imagine our feed has an average of 100,000 100-byte messages a second. That's a typical rate, after filtering market data we don't need to send on to subscribers. Now we decide to record a day's data (maybe 250 GB in 8 hours), and then replay it to a simulation network, i.e. a small group of subscribers. While 100K messages a second is easy for a ØMQ application, we want to replay *much faster*.

So we set-up our architecture with a bunch of boxes, one for the publisher, and one for each subscriber. These are well-specified boxes, eight cores, twelve for the publisher. (If you're reading this in 2015, which is when the Guide is scheduled to be finished, please add a zero to those numbers.)

And as we pump data into our subscribers, we notice two things:

1. When we do even the slightest amount of work with a message, it slows down our subscriber to the point where it can't catch up with the publisher again.
2. We're hitting a ceiling, at both publisher and subscriber, to around say 6M messages a second, even after careful optimization and TCP tuning.

The first thing we have to do is break our subscriber into a multithreaded design so that we can do work with messages in one set of threads, while reading messages in another. Typically we don't want to process every message the same way. Rather, the subscriber will filter some messages, perhaps by prefix key. When a message matches some criteria, the subscriber will call a worker to deal with it. In ØMQ terms this means sending the message to a worker thread.

So the subscriber looks something like a queue device. We could use various sockets to connect the subscriber and workers. If we assume one-way traffic, and workers that are all identical, we can use PUSH and PULL, and delegate all the routing work to ØMQ. This is the simplest and fastest approach:

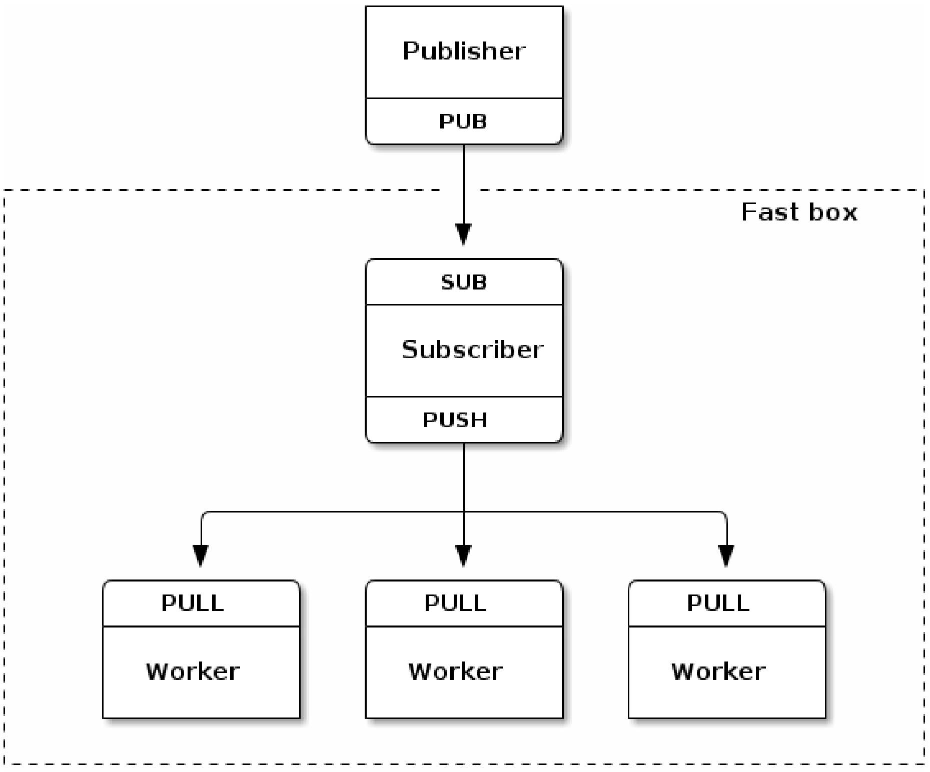


Figure 66 – Simple Black Box Pattern

The subscriber talks to the publisher over TCP or PGM. The subscriber talks to its workers, which are all in the same process, over inproc.

Now to break that ceiling. What happens is that the subscriber thread hits 100% of CPU, and since it is one thread, it cannot use more than one core. A single thread will always hit a ceiling, be it at 2M, 6M, or more messages per second. We want to split the work across multiple threads that can run in parallel.

The approach used by many high-performance products, which works here, is *sharding*, meaning we split the work into parallel and independent streams. E.g. half of the topic keys are in one stream, half in another. We could use many streams, but performance won't scale unless we have free cores.

So let's see how to shard into two streams:

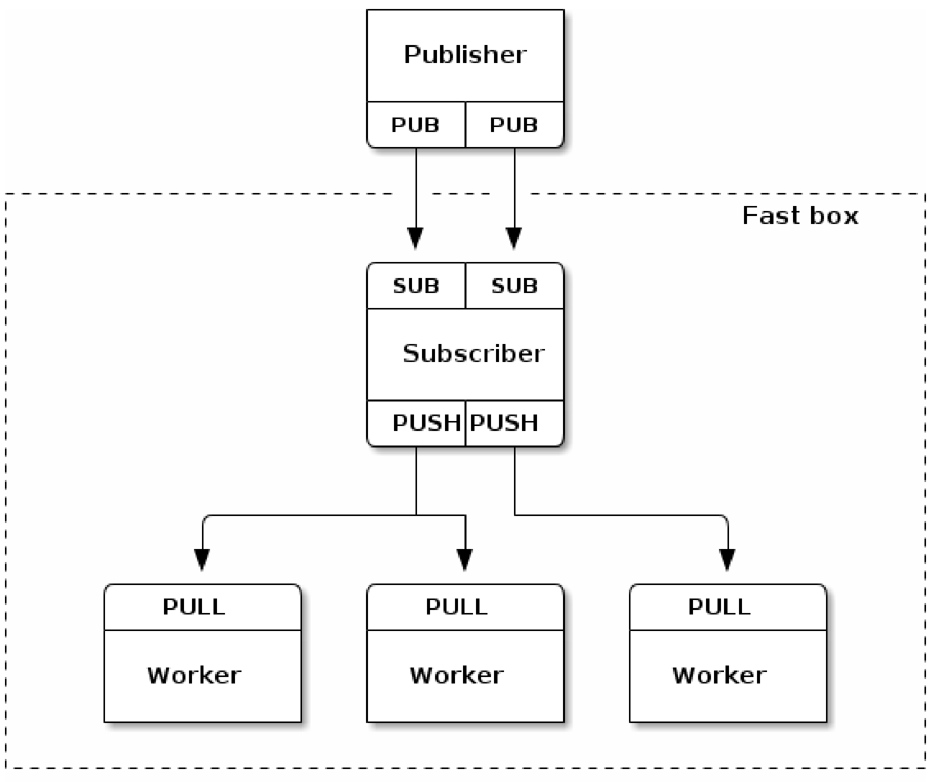


Figure 67 — Mad Black Box Pattern

With two streams, working at full speed, we would configure ØMQ as follows:

- Two I/O threads, rather than one.
- Two network interfaces (NIC), one per subscriber.
- Each I/O thread bound to a specific NIC.
- Two subscriber threads, bound to specific cores.
- Two SUB sockets, one per subscriber thread.
- The remaining cores assigned to worker threads.
- Worker threads connected to both subscriber PUSH sockets.

With ideally, no more threads in our architecture than we had cores. Once we create more threads than cores, we get contention between threads, and diminishing returns. There would be no benefit, for example, in creating more I/O threads.

## A Shared Key-Value Cache (Clone Pattern)

top prev next

Pub-sub is like a radio broadcast, you miss everything before you join, and then how much information you get depends on the quality of your reception. Surprisingly, for engineers who are used to aiming for "perfection", this model is useful and wide-spread, because it maps perfectly to real-world distribution of information. Think of Facebook and Twitter, the BBC World Service, and the sports results.

However, there are also a whole lot of cases where more reliable pub-sub would be valuable, if we could do it. As we did for request-reply, let's define ''reliability' in terms of what can go wrong. Here are the classic problems with pub-sub:

- Subscribers join late, so miss messages the server already sent.
- Subscriber connections are slow, and can lose messages during that time.
- Subscribers go away, and lose messages while they are away.

Less often, we see problems like these:

- Subscribers can crash, and restart, and lose whatever data they already received.
- Subscribers can fetch messages too slowly, so queues build up and then overflow.
- Networks can become overloaded and drop data (specifically, for PGM).
- Networks can become too slow, so publisher-side queues overflow, and publishers crash.

A lot more can go wrong but these are the typical failures we see in a realistic system.

We've already solved some of these, such as the slow subscriber, which we handle with the Suicidal Snail pattern. But for the rest, it would be nice to have a generic, reusable framework for reliable pub-sub.

The difficulty is that we have no idea what our target applications actually want to do with their data. Do they filter it, and process only a subset of messages? Do they log the data somewhere for later reuse? Do they distribute the data further to workers? There are dozens of plausible scenarios, and each will have its own ideas about what reliability means and how much it's worth in terms of effort and performance.

So we'll build an abstraction that we can implement once, and then reuse for many applications. This abstraction is a **shared value-key cache**, which stores a set of blobs indexed by unique keys.

Don't confuse this with *distributed hash tables*, which solve the wider problem of connecting peers in a distributed network, or with *distributed key-value tables*, which act like non-SQL databases. All we will build is a system that reliably clones some in-memory state from a server to a set of clients. We want to:

- Let a client join the network at any time, and reliably get the current server state.
- Let any client update the key-value cache (inserting new key-value pairs, updating existing ones, or deleting them).
- Reliably propagates changes to all clients, and does this with minimum latency overhead.
- Handle very large numbers of clients, e.g. tens of thousands or more.

The key aspect of the Clone pattern is that clients talk back to servers, which is more than we do in a simple pub-sub dialog. This is why I use the terms 'server' and 'client' instead of 'publisher' and 'subscriber'. We'll use pub-sub as the core of Clone but it is a bit more than that.

## Distributing Key-Value Updates

top prev next

We'll develop Clone in stages, solving one problem at a time. First, let's look at how to distribute key-value updates from a server to a set of clients. We'll take our weather server from Chapter One and refactor it to send messages as key-value pairs. We'll modify our client to store these in a hash table:

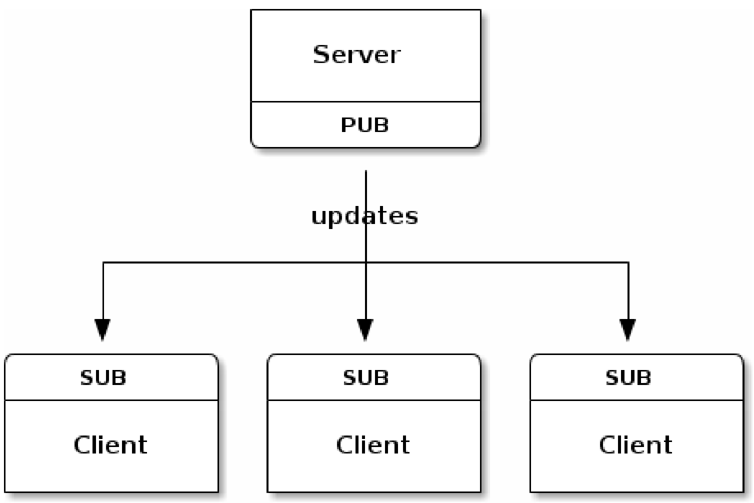


Figure 68 — Simplest Clone Model

This is the server:

```
//
//  Clone server Model One
//

//  Lets us build this source without creating a library
#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and publisher socket
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");
    zclock_sleep (200);

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    while (!zctx_interrupted) {
        //  Distribute as key-value message
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_store   (&kvmsg, kvmap);
    }
    printf (" Interrupted\n%d messages out\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

*clonesrv1.c: Clone server, Model One*

And here is the client:

```
//
//  Clone client Model One
//

//  Lets us build this source without creating a library
#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and updates socket
    zctx_t *ctx = zctx_new ();
    void *updates = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (updates, "tcp://localhost:5556");

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;

    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (updates);
        if (!kvmsg)
            break;          //  Interrupted
        kvmsg_store (&kvmsg, kvmap);
        sequence++;
    }
    printf (" Interrupted\n%d messages in\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

*clonecli1.c: Clone client, Model One*

Some notes about this code:

- All the hard work is done in a **kvmsg** class. This class works with key-value message objects, which are multipart ØMQ messages structured as three frames: a key (a ØMQ string), a sequence number (64-bit value, in network byte order), and a binary body (holds everything else).
- The server generates messages with a randomized 4-digit key, which lets us simulate a large but not enormous hash table (10K entries).
- The server does a 200 millisecond pause after binding its socket. This is to prevent "slow joiner syndrome" where the subscriber loses messages as it connects to the server's socket. We'll remove that in later models.
- We'll use the terms 'publisher' and 'subscriber' in the code to refer to sockets. This will help later when we have multiple sockets doing different things.

Here is the kvmsg class, in the simplest form that works for now:

```
/*
=====================================================================
    kvsimple - simple key-value message class for example
applications

    -----------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.

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

#include "kvsimple.h"
#include "zlist.h"

//  Keys are short strings
#define KVMSG_KEY_MAX   255

//  Message is formatted on wire as 4 frames:
//  frame 0: key (0MQ string)
//  frame 1: sequence (8 bytes, network order)
//  frame 2: body (blob)
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_BODY      2
#define KVMSG_FRAMES    3

//  Structure of our class
struct _kvmsg {
    //  Presence indicators for each frame
    int present [KVMSG_FRAMES];
    //  Corresponding 0MQ message frames, if any
    zmq_msg_t frame [KVMSG_FRAMES];
    //  Key, copied into safe C string
    char key [KVMSG_KEY_MAX + 1];
};

//  ---------------------------------------------------------------------
----
//  Constructor, sets sequence as provided

kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    kvmsg_set_sequence (self, sequence);
    return self;
```

```
}

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

//  Free shim, compatible with zhash_free_fn
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  Destroy message frames if any
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        //  Free object itself
        free (self);
    }
}

void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}

//  -----------------------------------------------------------------
----
//  Reads key-value message from socket, returns new kvmsg instance.

kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
    kvmsg_t *self = kvmsg_new (0);

    //  Read all frames off the wire, reject if bogus
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_recvmsg (socket, &self->frame [frame_nbr], 0) == -1)
{
            kvmsg_destroy (&self);
            break;
        }
        //  Verify multipart framing
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsockopt_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    return self;
}

//  ---------------------------------------------------------------------
----
//  Send key-value message to socket; any empty frames are sent as
such.

void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_sendmsg (socket, &copy,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}

//  ---------------------------------------------------------------------
----
//  Return key from last read message, if any, else NULL

char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}

//  ---------------------------------------------------------------------
----
//  Return sequence nbr from last read message, if any

int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}

//  ------------------------------------------------------------------
----
//  Return body from last read message, if any, else NULL

byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}

//  ------------------------------------------------------------------
----
//  Return body size from last read message, if any, else zero

size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}

//  ------------------------------------------------------------------
----
//  Set message key as provided

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}
```

```
//  ---------------------------------------------------------------
----
//  Set message sequence number

void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}

//  ---------------------------------------------------------------
----
//  Set message body

void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}

//  ---------------------------------------------------------------
----
//  Set message key using printf format

void
kvmsg_fmt_key (kvmsg_t *self, char *format, …)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}

//  ---------------------------------------------------------------
----
```

```
//  Set message body using printf format

void
kvmsg_fmt_body (kvmsg_t *self, char *format, …)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}

//  ---------------------------------------------------------------------
----
//  Store entire kvmsg into hash map, if key/value are set
//  Nullifies kvmsg reference, and destroys automatically when no
longer
//  needed.

void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (self->present [FRAME_KEY]
        &&  self->present [FRAME_BODY]) {
            zhash_update (hash, kvmsg_key (self), self);
            zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
        }
        *self_p = NULL;
    }
}

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

void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self));
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd] ", size);
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}

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

int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg: ");

    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  Test send and receive of simple message
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  Shutdown and destroy all objects
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

*kvsimple.c: Key-value message class*

We'll make a more sophisticated kvmsg class later, for using in real applications.

Both the server and client maintain hash tables, but this first model only works properly if we start all clients before the server, and the clients never crash. That's not 'reliability'.

## Getting a Snapshot

top prev next

In order to allow a late (or recovering) client to catch up with a server it has to get a snapshot of the server's state. Just as we've reduced "message" to mean "a sequenced key-value pair", we can reduce "state" to mean "a hash table". To get the server state, a client opens a REQ socket and asks for it explicitly:

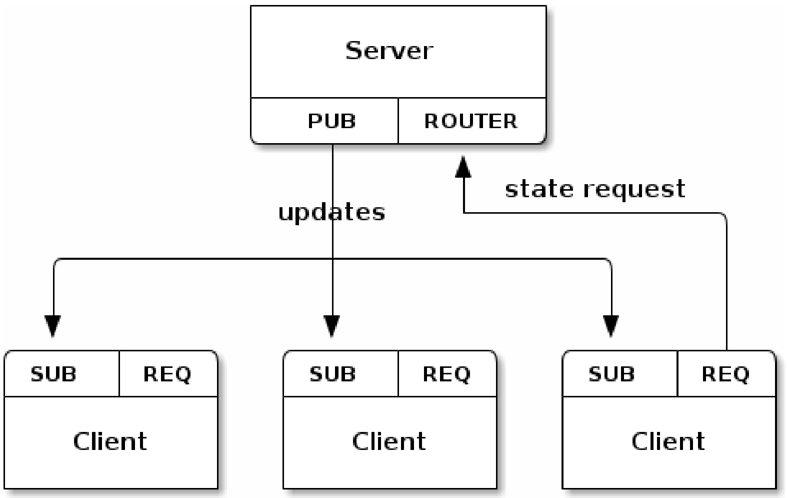


Figure 69 – State Replication

To make this work, we have to solve the timing problem. Getting a state snapshot will take a certain time, possibly fairly long if the snapshot is large. We need to correctly apply updates to the snapshot. But the server won't know when to start sending us updates. One way would be to start subscribing, get a first update, and then ask for "state for update N". This would require the server storing one snapshot for each update, which isn't practical.

So we will do the synchronization in the client, as follows:

- The client first subscribes to updates and then makes a state request. This guarantees that the state is going to be newer than the oldest update it has.
- The client waits for the server to reply with state, and meanwhile queues all updates. It does this simply by not reading them: ØMQ keeps them queued on the socket queue, since we don't set a HWM.
- When the client receives its state update, it begins once again to read updates. However it discards any updates that are older than the state update. So if the state update includes updates up to 200, the client will discard updates up to 201.
- The client then applies updates to its own state snapshot.

It's a simple model that exploits ØMQ's own internal queues. Here's the server:

```
//
//  Clone server Model Two
//

//  Lets us build this source without creating a library
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);
static void state_manager (void *args, zctx_t *ctx, void *pipe);

int main (void)
```

```
{
    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");

    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    //  Start state manager and wait for synchronization signal
    void *updates = zthread_fork (ctx, state_manager, NULL);
    free (zstr_recv (updates));

    while (!zctx_interrupted) {
        //  Distribute as key-value message
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_send     (kvmsg, updates);
        kvmsg_destroy (&kvmsg);
    }
    printf (" Interrupted\n%d messages out\n", (int) sequence);
    zctx_destroy (&ctx);
    return 0;
}

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
} kvroute_t;

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  Send identity of recipient first
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}

//  This thread maintains the state and handles requests from
//  clients for snapshots.
//
static void
state_manager (void *args, zctx_t *ctx, void *pipe)
{
    zhash_t *kvmap = zhash_new ();

    zstr_send (pipe, "READY");
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");

    zmq_pollitem_t items [] = {
        { pipe, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    int64_t sequence = 0;       //  Current snapshot version number
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1 && errno == ETERM)
            break;              //  Context has been shut down

        //  Apply state update from main thread
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (pipe);
            if (!kvmsg)
                break;          //  Interrupted
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        //  Execute state snapshot request
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted

            //  Request is in second frame of message
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: bad request, aborting\n");
                break;
            }
            //  Send state snapshot to client
            kvroute_t routing = { snapshot, identity };

            //  For each entry in kvmap, send kvmsg to client
            zhash_foreach (kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            printf ("Sending state shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    zhash_destroy (&kvmap);
}
```

*clonesrv2.c: Clone server, Model Two*

And here is the client:

```
//
//  Clone client Model Two
//

//  Lets us build this source without creating a library
```

```
#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and subscriber
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");

    zhash_t *kvmap = zhash_new ();

    //  Get state snapshot
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("Received snapshot=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  Done
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    //  Now apply pending updates, discard out-of-sequence messages
    while (!zctx_interrupted) {
        kvmsg_t *kvmsg = kvmsg_recv (subscriber);
        if (!kvmsg)
            break;          //  Interrupted
        if (kvmsg_sequence (kvmsg) > sequence) {
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

*clonecli2.c: Clone client, Model Two*

Some notes about this code:

- The server uses two threads, for simpler design. One thread produces random updates, and the second thread handles state. The two communicate across PAIR sockets. You might like to use SUB sockets but you'd hit the "slow joiner" problem where the subscriber would randomly miss some messages while connecting. PAIR sockets let us explicitly synchronize the two threads.

- We set a HWM on the updates socket pair, since hash table insertions are relatively slow. Without this, the server runs out of memory. On `inproc` connections, the real HWM is the sum of the HWM of *both* sockets, so we set the HWM on each socket.

- The client is really simple. In C, under 60 lines of code. A lot of the heavy lifting is done in the kvmsg class, but still, the basic Clone pattern is easier to implement than it seemed at first.

- We don't use anything fancy for serializing the state. The hash table holds a set of kvmsg objects, and the server sends these, as a batch of messages, to the client requesting state. If multiple clients request state at once, each will get a different snapshot.
- We assume that the client has exactly one server to talk to. The server **must** be running; we do not try to solve the question of what happens if the server crashes.

Right now, these two programs don't do anything real, but they correctly synchronize state. It's a neat example of how to mix different patterns: PAIR-over-inproc, PUB-SUB, and ROUTER-DEALER.

## Republishing Updates

top prev next

In our second model, changes to the key-value cache came from the server itself. This is a centralized model, useful for example if we have a central configuration file we want to distribute, with local caching on each node. A more interesting model takes updates from clients, not the server. The server thus becomes a stateless broker. This gives us some benefits:

- We're less worried about the reliability of the server. If it crashes, we can start a new instance, and feed it new values.
- We can use the key-value cache to share knowledge between dynamic peers.

Updates from clients go via a PUSH-PULL socket flow from client to server:

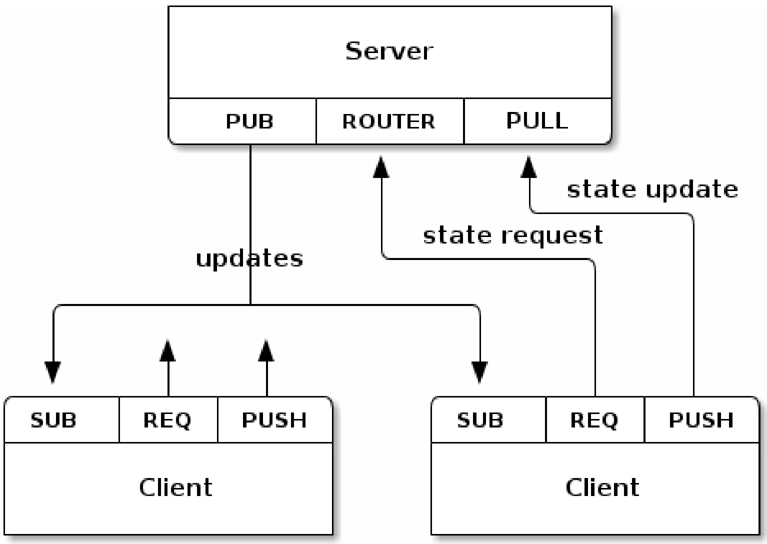


Figure 70 — Republishing Updates

Why don't we allow clients to publish updates directly to other clients? While this would reduce latency, it makes it impossible to assign ascending unique sequence numbers to messages. The server can do this. There's a more subtle second reason. In many applications it's important that updates have a single order, across many clients. Forcing all updates through the server ensures that they have the same order when they finally get to clients.

With unique sequencing, clients can detect the nastier failures - network congestion and queue overflow. If a client discovers that its incoming message stream has a hole, it can take action. It seems sensible that the client contact the server and ask for the missing messages, but in practice that isn't useful. If there are holes, they're caused by network stress, and adding more stress to the network will make things worse. All the client can really do is warn its users "Unable to continue", and stop, and not restart until someone has manually checked the cause of the problem.

We'll now generate state updates in the client. Here's the server:

```
//
//  Clone server Model Three
//

//  Lets us build this source without creating a library
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
} kvroute_t;

int main (void)
{
    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  Apply state update sent from client
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  Interrupted
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: publishing update %5d\n", (int) sequence);
        }
        //  Execute state snapshot request
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted
            //  Request is in second frame of message
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: bad request, aborting\n");
                break;
            }
            //  Send state snapshot to client
            kvroute_t routing = { snapshot, identity };

            //  For each entry in kvmap, send kvmsg to client
            zhash_foreach (kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            printf ("I: sending shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    printf (" Interrupted\n%d messages handled\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  Send identity of recipient first
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}
```

*clonesrv3.c: Clone server, Model Three*

And here is the client:

```
//
//  Clone client Model Three
//

//  Lets us build this source without creating a library
#include "kvsimple.c"

int main (void)
{
    //  Prepare our context and subscriber
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  Get state snapshot
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: received snapshot=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  Done
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    int64_t alarm = zclock_time () + 1000;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 }
};
        int tickless = (int) ((alarm - zclock_time ()));
        if (tickless < 0)
            tickless = 0;
        int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Context has been shut down

        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (subscriber);
            if (!kvmsg)
                break;          //  Interrupted

            //  Discard out-of-sequence kvmsgs, incl. heartbeats
            if (kvmsg_sequence (kvmsg) > sequence) {
                sequence = kvmsg_sequence (kvmsg);
                kvmsg_store (&kvmsg, kvmap);
                printf ("I: received update=%d\n", (int) sequence);
            }
            else
                kvmsg_destroy (&kvmsg);
        }
        //  If we timed-out, generate a random kvmsg
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
    }
    printf (" Interrupted\n%d messages in\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

*clonecli3.c: Clone client, Model Three*

Some notes about this code:

- The server has collapsed to one thread, which collects updates from clients and redistributes them. It manages a PULL socket for incoming updates, a ROUTER socket for state requests, and a PUB socket for outgoing updates.
- The client uses a simple tickless timer to send a random update to the server once a second. In reality, updates would be driven by application code.

## Clone Subtrees

top prev next

A realistic key-value cache will get large, and clients will usually be interested only in parts of the cache. Working with a subtree is fairly simple. The client has to tell the server the subtree when it makes a state request, and it has to specify the same subtree when it subscribes to updates.

There are a couple of common syntaxes for trees. One is the "path hierarchy", and another is the "topic tree". These look like:

- Path hierarchy: "/some/list/of/paths"
- Topic tree: "some.list.of.topics"

We'll use the path hierarchy, and extend our client and server so that a client can work with a single subtree. Working with multiple subtrees is not much more difficult, we won't do that here but it's a trivial extension.

Here's the server, a small variation on Model Three:

```
//
//  Clone server Model Four
//

//  Lets us build this source without creating a library
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
    char *subtree;          //  Client subtree specification
} kvroute_t;

int main (void)
{
    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  Apply state update sent from client
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  Interrupted
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: publishing update %5d\n", (int) sequence);
        }
        //  Execute state snapshot request
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted

            //  Request is in second frame of message
            char *request = zstr_recv (snapshot);
            char *subtree = NULL;
            if (streq (request, "ICANHAZ?")) {
                free (request);
                subtree = zstr_recv (snapshot);
            }
            else {
                printf ("E: bad request, aborting\n");
                break;
            }
            //  Send state snapshot to client
            kvroute_t routing = { snapshot, identity, subtree };

            //  For each entry in kvmap, send kvmsg to client
            zhash_foreach (kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            printf ("I: sending shapshot=%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    printf (" Interrupted\n%d messages handled\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  Send identity of recipient first
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}
```

*clonesrv4.c: Clone server, Model Four*

And here is the client:

```
//
//  Clone client Model Four
//

//  Lets us build this source without creating a library
#include "kvsimple.c"

#define SUBTREE "/client/"

int main (void)
{
    //  Prepare our context and subscriber
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");
    zsockopt_set_subscribe (subscriber, SUBTREE);
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  Get state snapshot
    int64_t sequence = 0;
    zstr_sendm (snapshot, "ICANHAZ?");
    zstr_send  (snapshot, SUBTREE);
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: received snapshot=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  Done
        }
        kvmsg_store (&kvmsg, kvmap);
    }

    int64_t alarm = zclock_time () + 1000;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 }
};
        int tickless = (int) ((alarm - zclock_time ()));
        if (tickless < 0)
            tickless = 0;
        int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  Context has been shut down

        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (subscriber);
            if (!kvmsg)
                break;          //  Interrupted

            //  Discard out-of-sequence kvmsgs, incl. heartbeats
            if (kvmsg_sequence (kvmsg) > sequence) {
                sequence = kvmsg_sequence (kvmsg);
                kvmsg_store (&kvmsg, kvmap);
                printf ("I: received update=%d\n", (int) sequence);
            }
            else
                kvmsg_destroy (&kvmsg);
        }
        //  If we timed-out, generate a random kvmsg
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%s%d", SUBTREE, randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
    }
    printf (" Interrupted\n%d messages in\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

*clonecli4.c: Clone client, Model Four*

## Ephemeral Values

top prev next

An ephemeral value is one that expires dynamically. If you think of Clone being used for a DNS-like service, then ephemeral values would let you do dynamic DNS. A node joins the network, publishes its address, and refreshes this regularly. If the node dies, its address eventually gets removed.

The usual abstraction for ephemeral values is to attach them to a "session", and delete them when the session ends. In Clone, sessions would be defined by clients, and would end if the client died.

The simpler alternative to using sessions is to define every ephemeral value with a "time to live" that tells the server when to expire the value. Clients then refresh values, and if they don't, the values expire.

I'm going to implement that simpler model because we don't know yet that it's worth making a more complex one. The difference is really in performance. If clients have a handful of ephemeral values, it's fine to set a TTL on each one. If clients use masses of ephemeral values, it's more efficient to attach them to sessions, and expire them in bulk.

First off, we need a way to encode the TTL in the key-value message. We could add a frame. The problem with using frames for properties is that each time we want to add a new property, we have to change the structure of our kvmsg class. It breaks compatibility. So let's add a 'properties' frame to the message, and code to let us get and put property values.

Next, we need a way to say, "delete this value". Up to now servers and clients have always blindly inserted or updated new values into their hash table. We'll say that if the value is empty, that means "delete this key".

Here's a more complete version of the kvmsg class, which implements a 'properties' frame (and adds a UUID frame, which we'll need later on). It also handles empty values by deleting the key from the hash, if necessary:

```
/*
=====================================================================
    kvmsg - key-value message class for example applications

    -----------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.
```

```
=====================================================================
*/

#include "kvmsg.h"
#include <uuid/uuid.h>
#include "zlist.h"

//  Keys are short strings
#define KVMSG_KEY_MAX   255

//  Message is formatted on wire as 4 frames:
//  frame 0: key (0MQ string)
//  frame 1: sequence (8 bytes, network order)
//  frame 2: uuid (blob, 16 bytes)
//  frame 3: properties (0MQ string)
//  frame 4: body (blob)
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_UUID      2
#define FRAME_PROPS     3
#define FRAME_BODY      4
#define KVMSG_FRAMES    5

//  Structure of our class
struct _kvmsg {
    //  Presence indicators for each frame
    int present [KVMSG_FRAMES];
    //  Corresponding 0MQ message frames, if any
    zmq_msg_t frame [KVMSG_FRAMES];
    //  Key, copied into safe C string
    char key [KVMSG_KEY_MAX + 1];
    //  List of properties, as name=value strings
    zlist_t *props;
    size_t props_size;
};

//  Serialize list of properties to a message frame
static void
s_encode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    if (self->present [FRAME_PROPS])
        zmq_msg_close (msg);

    zmq_msg_init_size (msg, self->props_size);
    char *prop = zlist_first (self->props);
    char *dest = (char *) zmq_msg_data (msg);
    while (prop) {
        strcpy (dest, prop);
        dest += strlen (prop);
        *dest++ = '\n';
        prop = zlist_next (self->props);
    }
    self->present [FRAME_PROPS] = 1;
}

//  Rebuild properties list from message frame
static void
s_decode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    self->props_size = 0;
    while (zlist_size (self->props))
        free (zlist_pop (self->props));

    size_t remainder = zmq_msg_size (msg);
    char *prop = (char *) zmq_msg_data (msg);
    char *eoln = memchr (prop, '\n', remainder);
    while (eoln) {
        *eoln = 0;
        zlist_append (self->props, strdup (prop));
        self->props_size += strlen (prop) + 1;
        remainder -= strlen (prop) + 1;
        prop = eoln + 1;
        eoln = memchr (prop, '\n', remainder);
    }
}

//  -----------------------------------------------------------------
----
//  Constructor, sets sequence as provided

kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    self->props = zlist_new ();
    kvmsg_set_sequence (self, sequence);
    return self;
}

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

//  Free shim, compatible with zhash_free_fn
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  Destroy message frames if any
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        //  Destroy property list
        while (zlist_size (self->props))
            free (zlist_pop (self->props));
        zlist_destroy (&self->props);

        //  Free object itself
        free (self);
    }
}

void
```

```
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------
----
//  Create duplicate of kvmsg

kvmsg_t *
kvmsg_dup (kvmsg_t *self)
{
    kvmsg_t *kvmsg = kvmsg_new (0);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr]) {
            zmq_msg_t *src = &self->frame [frame_nbr];
            zmq_msg_t *dst = &kvmsg->frame [frame_nbr];
            zmq_msg_init_size (dst, zmq_msg_size (src));
            memcpy (zmq_msg_data (dst),
                    zmq_msg_data (src), zmq_msg_size (src));
            kvmsg->present [frame_nbr] = 1;
        }
    }
    kvmsg->props = zlist_copy (self->props);
    return kvmsg;
}

//  ---------------------------------------------------------------
----
//  Reads key-value message from socket, returns new kvmsg instance.

kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
    kvmsg_t *self = kvmsg_new (0);

    //  Read all frames off the wire, reject if bogus
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_recvmsg (socket, &self->frame [frame_nbr], 0) == -1)
{
            kvmsg_destroy (&self);
            break;
        }
        //  Verify multipart framing
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsockopt_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    if (self)
        s_decode_props (self);
    return self;
}

//  ---------------------------------------------------------------------
----
//  Send key-value message to socket; any empty frames are sent as
such.

void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    s_encode_props (self);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_sendmsg (socket, &copy,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}

//  ---------------------------------------------------------------------
----
//  Return key from last read message, if any, else NULL

char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}

//  ---------------------------------------------------------------------
----
//  Return sequence nbr from last read message, if any

int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}

//  ---------------------------------------------------------------------
----
//  Return UUID from last read message, if any, else NULL

byte *
kvmsg_uuid (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_UUID]
    &&  zmq_msg_size (&self->frame [FRAME_UUID]) == sizeof (uuid_t))
        return (byte *) zmq_msg_data (&self->frame [FRAME_UUID]);
    else
        return NULL;
}

//  ---------------------------------------------------------------------
----
//  Return body from last read message, if any, else NULL

byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}

//  ---------------------------------------------------------------------
----
//  Return body size from last read message, if any, else zero

size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}
```

```
//  ---------------------------------------------------------------------
----
//  Set message key as provided

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}

//  ---------------------------------------------------------------------
----
//  Set message sequence number

void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}

//  ---------------------------------------------------------------------
----
//  Set message UUID to generated value

void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
    uuid_generate (uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid));
    memcpy (zmq_msg_data (msg), uuid, sizeof (uuid));
    self->present [FRAME_UUID] = 1;
}
```

```
//  ---------------------------------------------------------------------
----
//  Set message body

void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}

//  ---------------------------------------------------------------------
----
//  Set message key using printf format

void
kvmsg_fmt_key (kvmsg_t *self, char *format, …)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}

//  ---------------------------------------------------------------------
----
//  Set message body using printf format

void
kvmsg_fmt_body (kvmsg_t *self, char *format, …)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}

//  ---------------------------------------------------------------------
----
//  Get message property, if set, else ""

char *
kvmsg_get_prop (kvmsg_t *self, char *name)
{
    assert (strchr (name, '=') == NULL);
    char *prop = zlist_first (self->props);
    size_t namelen = strlen (name);
    while (prop) {
        if (strlen (prop) > namelen
        &&  memcmp (prop, name, namelen) == 0
        &&  prop [namelen] == '=')
            return prop + namelen + 1;
        prop = zlist_next (self->props);
    }
    return "";
}

//  -----------------------------------------------------------------
----
//  Set message property
//  Names cannot contain '='. Max length of value is 255 chars.

void
kvmsg_set_prop (kvmsg_t *self, char *name, char *format, …)
{
    assert (strchr (name, '=') == NULL);

    char value [255 + 1];
    va_list args;
    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);

    //  Allocate name=value string
    char *prop = malloc (strlen (name) + strlen (value) + 2);

    //  Remove existing property if any
    sprintf (prop, "%s=", name);
    char *existing = zlist_first (self->props);
    while (existing) {
        if (memcmp (prop, existing, strlen (prop)) == 0) {
            self->props_size -= strlen (existing) + 1;
            zlist_remove (self->props, existing);
            free (existing);
            break;
        }
        existing = zlist_next (self->props);
    }
    //  Add new name=value property string
    strcat (prop, value);
    zlist_append (self->props, prop);
    self->props_size += strlen (prop) + 1;
}

//  -----------------------------------------------------------------
----
//  Store entire kvmsg into hash map, if key/value are set.
//  Nullifies kvmsg reference, and destroys automatically when no
longer
//  needed. If value is empty, deletes any previous value from store.

void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (kvmsg_size (self)) {
            if (self->present [FRAME_KEY]
            &&  self->present [FRAME_BODY]) {
                zhash_update (hash, kvmsg_key (self), self);
                zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
            }
        }
        else
            zhash_delete (hash, kvmsg_key (self));

        *self_p = NULL;
    }
}

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

void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self));
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd] ", size);
        if (zlist_size (self->props)) {
            fprintf (stderr, "[");
            char *prop = zlist_first (self->props);
            while (prop) {
                fprintf (stderr, "%s;", prop);
                prop = zlist_next (self->props);
            }
            fprintf (stderr, "]");
        }
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}

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

int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;
    printf (" * kvmsg: ");

    //  Prepare our context and sockets
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  Test send and receive of simple message
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  Test send and receive of message with properties
    kvmsg = kvmsg_new (2);
    kvmsg_set_prop (kvmsg, "prop1", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value1");
    kvmsg_set_prop (kvmsg, "prop2", "value2");
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_uuid (kvmsg);
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_destroy (&kvmsg);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
    kvmsg_destroy (&kvmsg);

    //  Shutdown and destroy all objects
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

*kvmsg.c: Key-value message class - full*

The Model Five client is almost identical to Model Four. Diff is your friend. It uses the full kvmsg class instead of kvsimple, and sets a randomized 'ttl' property (measured in seconds) on each message:

```
kvmsg_set_prop (kvmsg, "ttl", "%d", randof (30));
```

The Model Five server has totally changed. Instead of a poll loop, we're now using a reactor. This just makes it simpler to mix timers and socket events. Unfortunately in C the reactor style is more verbose. Your mileage will vary in other languages. But reactors seem to be a better way of building more complex ØMQ applications. Here's the server:

```
//
//  Clone server Model Five
//

//  Lets us build this source without creating a library
#include "kvmsg.c"

//  zloop reactor handlers
static int s_snapshots  (zloop_t *loop, void *socket, void *args);
static int s_collector  (zloop_t *loop, void *socket, void *args);
static int s_flush_ttl  (zloop_t *loop, void *socket, void *args);

//  Our server is defined by these properties
typedef struct {
    zctx_t *ctx;                //  Context wrapper
    zhash_t *kvmap;             //  Key-value store
    zloop_t *loop;              //  zloop reactor
    int port;                   //  Main port we're working on
    int64_t sequence;           //  How many updates we're at
    void *snapshot;             //  Handle snapshot requests
    void *publisher;            //  Publish updates to clients
    void *collector;            //  Collect updates from clients
} clonesrv_t;

int main (void)
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));

    self->port = 5556;
    self->ctx = zctx_new ();
    self->kvmap = zhash_new ();
    self->loop = zloop_new ();
    zloop_set_verbose (self->loop, FALSE);

    //  Set up our clone server sockets
    self->snapshot  = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_PULL);
    zsocket_bind (self->snapshot,  "tcp://*:%d", self->port);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  Register our handlers with reactor
    zloop_reader (self->loop, self->snapshot, s_snapshots, self);
    zloop_reader (self->loop, self->collector, s_collector, self);
    zloop_timer  (self->loop, 1000, 0, s_flush_ttl, self);

    //  Run reactor until process interrupted
    zloop_start (self->loop);

    zloop_destroy (&self->loop);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);
    return 0;
}

//  -----------------------------------------------------------------
----
//  Send snapshots to clients who ask for them

static int s_send_single (char *key, void *data, void *args);

//  Routing information for a key-value snapshot
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
    char *subtree;          //  Client subtree specification
} kvroute_t;

static int
s_snapshots (zloop_t *loop, void *snapshot, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (snapshot);
    if (identity) {
        //  Request is in second frame of message
        char *request = zstr_recv (snapshot);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (snapshot);
        }
        else
            printf ("E: bad request, aborting\n");

        if (subtree) {
            //  Send state socket to client
            kvroute_t routing = { snapshot, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            zclock_log ("I: sending shapshot=%d", (int) self-
>sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    return 0;
}

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
```

```
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  Send identity of recipient first
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}

//  ---------------------------------------------------------------------
----
//  Collect updates from clients

static int
s_collector (zloop_t *loop, void *collector, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (collector);
    if (kvmsg) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
        if (ttl)
            kvmsg_set_prop (kvmsg, "ttl",
                "%" PRId64, zclock_time () + ttl * 1000);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing update=%d", (int) self->sequence);
    }
    return 0;
}

//  ---------------------------------------------------------------------
----
//  Purge ephemeral values that have expired

static int s_flush_single (char *key, void *data, void *args);

static int
s_flush_ttl (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}

//  If key-value pair has expired, delete it and publish the
//  fact to listening clients.
static int
s_flush_single (char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing delete=%d", (int) self->sequence);
    }
    return 0;
}
```

*clonesrv5.c: Clone server, Model Five*

## Clone Server Reliability

top prev next

Clone models one to five are relatively simple. We're now going to get into unpleasantly complex territory here that has me getting up for another espresso. You should appreciate that making "reliable" messaging is complex enough that you always need to ask, "do we actually need this?" before jumping into it. If you can get away with unreliable, or "good enough" reliability, you can make a huge win in terms of cost and complexity. Sure, you may lose some data now and then. It is often a good trade-off. Having said, that, and since the espresso is really good, let's jump in!

As you play with model three, you'll stop and restart the server. It might look like it recovers, but of course it's applying updates to an empty state, instead of the proper current state. Any new client joining the network will get just the latest updates, instead of all of them. So let's work out a design for making Clone work despite server failures.

Let's list the failures we want to be able to handle:

- Clone server process crashes and is automatically or manually restarted. The process loses its state and has to get it back from somewhere.
- Clone server machine dies and is off-line for a significant time. Clients have to switch to an alternate server somewhere.
- Clone server process or machine gets disconnected from the network, e.g. a switch dies. It may come back at some point, but in the meantime clients need an alternate server.

Our first step is to add a second server. We can use the Binary Star pattern from Chapter four to organize these into primary and backup. Binary Star is a reactor, so it's useful that we already refactored the last server model into a reactor style.

We need to ensure that updates are not lost if the primary server crashes. The simplest technique is to send them to both servers.

The backup server can then act as a client, and keep its state synchronized by receiving updates as all clients do. It'll also get new updates from clients. It can't yet store these in its hash table, but it can hold onto them for a while.

So, Model Six introduces these changes over Model Five:

- We use a pub-sub flow instead of a push-pull flow for client updates (to the servers). The reasons: push sockets will block if there is no recipient, and they round-robin, so we'd need to open two of them. We'll bind the servers' SUB sockets and connect the clients' PUB sockets to them. This takes care of fanning out from one client to two servers.

- We add heartbeats to server updates (to clients), so that a client can detect when the primary server has died. It can then switch over to the backup server.
- We connect the two servers using the Binary Star `bstar` reactor class. Binary Star relies on the clients to 'vote' by making an explicit request to the server they consider "master". We'll use snapshot requests for this.
- We make all update messages uniquely identifiable by adding a UUID field. The client generates this, and the server propagates it back on re-published updates.
- The slave server keeps a "pending list" of updates that it has received from clients, but not yet from the master server. Or, updates it's received from the master, but not yet clients. The list is ordered from oldest to newest, so that it is easy to remove updates off the head.

It's useful to design the client logic as a finite state machine. The client cycles through three states:

- The client opens and connects its sockets, and then requests a snapshot from the first server. To avoid request storms, it will ask any given server only twice. One request might get lost, that'd be bad luck. Two getting lost would be carelessness.
- The client waits for a reply (snapshot data) from the current server, and if it gets it, it stores it. If there is no reply within some timeout, it fails over to the next server.
- When the client has gotten its snapshot, it waits for and processes updates. Again, if it doesn't hear anything from the server within some timeout, it fails over to the next server.

The client loops forever. It's quite likely during startup or fail-over that some clients may be trying to talk to the primary server while others are trying to talk to the backup server. The Binary Star pattern handles this, hopefully accurately. (One of the joys of making designs like this is we cannot prove they are right, we can only prove them wrong. So it's like a guy falling off a tall building. So far, so good… so far, so good…)

We can draw the client finite state machine, with events in caps:

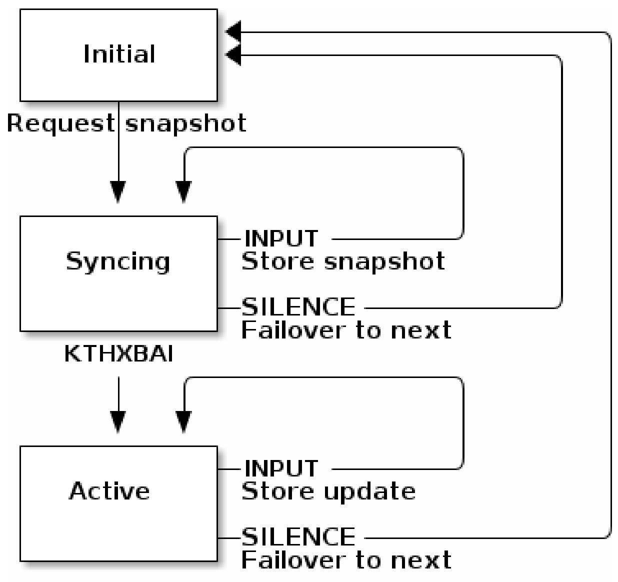


Figure 71 – Clone client FSM

Fail-over happens as follows:

- The client detects that primary server is no longer sending heartbeats, so has died. The client connects to the backup server and requests a new state snapshot.
- The backup server starts to receive snapshot requests from clients, and detects that primary server has gone, so takes over as primary.
- The backup server applies its pending list to its own hash table, and then starts to process state snapshot requests.

When the primary server comes back on-line, it will:

- Start up as slave server, and connect to the backup server as a Clone client.
- Start to receive updates from clients, via its SUB socket.

We make some assumptions:

- That at least one server will keep running. If both servers crash, we lose all server state and there's no way to recover it.
- That multiple clients do not update the same hash keys, at the same time. Client updates will arrive at the two servers in a different order. So, the backup server may apply updates from its pending list in a different order than the primary server would or did. Updates from one client will always arrive in the same order on both servers, so that is safe.

So here is our high-availability server pair, using the Binary Star pattern:

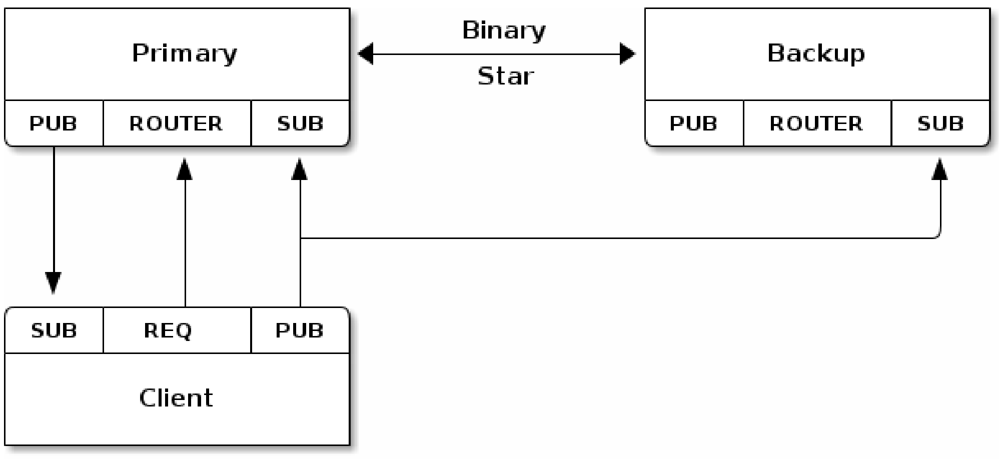


Figure 72 – High availability Clone Server Pair

As a first step to building this, we're going to refactor the client as a reusable class. This is partly for fun (writing asynchronous classes with ØMQ is like an exercise in elegance), but mainly because we want Clone to be really easy to plug-in to random applications. Since resilience depends on clients behaving correctly, it's much easier to guarantee this when there's a reusable client API. When we start to handle fail-over in clients, it does get a little complex (imagine mixing a Freelance client with a Clone client). So, reusability ahoy!

My usual design approach is to first design an API that feels right, then to implement that. So, we start by taking the clone client, and rewriting it to sit on top of some presumed class API called **clone**. Turning random code into an API means defining a reasonably stable and abstract contract with applications. For example, in Model Five, the client opened three separate sockets to the server, using endpoints that were hard-coded in the source. We could make an API with three methods, like this:

```
    //  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");
```

But this is both verbose and fragile. It's not a good idea to expose the internals of a design to applications. Today, we use three sockets. Tomorrow, two, or four. Do we really want to go and change every application that uses the clone class? So to hide these sausage factory details, we make a small abstraction, like this:

```
    //  Specify primary and backup servers
    clone_connect (clone, "tcp://localhost:5551");
    clone_connect (clone, "tcp://localhost:5561");
```

Which has the advantage of simplicity (one server sits at one endpoint) but has an impact on our internal design. We now need to somehow turn that single endpoint into three endpoints. One way would be to bake the knowledge "client and server talk over three consecutive ports" into our client-server protocol. Another way would be to get the two missing endpoints from the server. We'll take the simplest way, which is:

- The server state router (ROUTER) is at port P.
- The server updates publisher (PUB) is at port P + 1.
- The server updates subscriber (SUB) is at port P + 2.

The clone class has the same structure as the flcliapi class from Chapter Four. It consists of two parts:

- An asynchronous clone agent that runs in a background thread. The agent handles all network I/O, talking to servers in realtime, no matter what the application is doing.

- A synchronous 'clone' class which runs in the caller's thread. When you create a clone object, that automatically launches an agent thread, and when you destroy a clone object, it kills the agent thread.

The frontend class talks to the agent class over an `inproc` 'pipe' socket. In C, the czmq thread layer creates this pipe automatically for us as it starts an "attached thread". This is a natural pattern for multithreading over ØMQ.

Without ØMQ, this kind of asynchronous class design would be weeks of really hard work. With ØMQ, it was a day or two of work. The results are kind of complex, given the simplicity of the Clone protocol it's actually running. There are some reasons for this. We could turn this into a reactor, but that'd make it harder to use in applications. So the API looks a bit like a key-value table that magically talks to some servers:

```
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);
```

So here is Model Six of the clone client, which has now become just a thin shell using the clone class:

```
//
```

```
//  Clone client Model Six
//

//  Lets us build this source without creating a library
#include "clone.c"

#define SUBTREE "/client/"

int main (void)
{
    //  Create distributed hash instance
    clone_t *clone = clone_new ();

    //  Specify configuration
    clone_subtree (clone, SUBTREE);
    clone_connect (clone, "tcp://localhost", "5556");
    clone_connect (clone, "tcp://localhost", "5566");

    //  Set random tuples into the distributed hash
    while (!zctx_interrupted) {
        //  Set random value, check it was stored
        char key [255];
        char value [10];
        sprintf (key, "%s%d", SUBTREE, randof (10000));
        sprintf (value, "%d", randof (1000000));
        clone_set (clone, key, value, randof (30));
        sleep (1);
    }
    clone_destroy (&clone);
    return 0;
}
```

*clonecli6.c: Clone client, Model Six*

And here is the actual clone class implementation:

```
/*
=====================================================================
    clone - client-side Clone Pattern class

    -----------------------------------------------------------------
----
    Copyright (c) 1991-2011 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
    MERCHANTABILITY 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/>.

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

#include "clone.h"

//  If no server replies within this time, abandon request
#define GLOBAL_TIMEOUT  4000    //  msecs
//  Server considered dead if silent for this long
#define SERVER_TTL      5000    //  msecs
//  Number of servers we will talk to
#define SERVER_MAX      2

//
=====================================================================
//  Synchronous part, works in our application thread

//  -----------------------------------------------------------------
----
//  Structure of our class

struct _clone_t {
    zctx_t *ctx;                //  Our context wrapper
    void *pipe;                 //  Pipe through to clone agent
};

//  This is the thread that handles our real clone class
static void clone_agent (void *args, zctx_t *ctx, void *pipe);

//  -----------------------------------------------------------------
----
//  Constructor

clone_t *
clone_new (void)
{
    clone_t
        *self;

    self = (clone_t *) zmalloc (sizeof (clone_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, clone_agent, NULL);
    return self;
}

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

void
clone_destroy (clone_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        clone_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
----
//  Specify subtree for snapshot and updates, do before connect
//  Sends [SUBTREE][subtree] to the agent

void clone_subtree (clone_t *self, char *subtree)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SUBTREE");
    zmsg_addstr (msg, subtree);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------------
----
//  Connect to new server endpoint
//  Sends [CONNECT][endpoint][service] to the agent

void
clone_connect (clone_t *self, char *address, char *service)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, address);
    zmsg_addstr (msg, service);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------------
----
//  Set new value in distributed hash table
//  Sends [SET][key][value][ttl] to the agent

void
clone_set (clone_t *self, char *key, char *value, int ttl)
{
    char ttlstr [10];
    sprintf (ttlstr, "%d", ttl);

    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SET");
    zmsg_addstr (msg, key);
    zmsg_addstr (msg, value);
    zmsg_addstr (msg, ttlstr);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------------
----
//  Lookup value in distributed hash table
//  Sends [GET][key] to the agent and waits for a value response
//  If there is no clone available, will eventually return NULL.

char *
```

```
clone_get (clone_t *self, char *key)
{
    assert (self);
    assert (key);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "GET");
    zmsg_addstr (msg, key);
    zmsg_send (&msg, self->pipe);

    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *value = zmsg_popstr (reply);
        zmsg_destroy (&reply);
        return value;
    }
    return NULL;
}

//
=====================================================================
//  Asynchronous part, works in the background

//  -----------------------------------------------------------------
----
//  Simple class for one server we talk to

typedef struct {
    char *address;              //  Server address
    int port;                   //  Server port
    void *snapshot;             //  Snapshot socket
    void *subscriber;           //  Incoming updates
    uint64_t expiry;            //  When server expires
    uint requests;              //  How many snapshot requests made?
} server_t;

static server_t *
server_new (zctx_t *ctx, char *address, int port, char *subtree)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));

    zclock_log ("I: adding server %s:%d…", address, port);
    self->address = strdup (address);
    self->port = port;

    self->snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (self->snapshot, "%s:%d", address, port);
    self->subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "%s:%d", address, port + 1);
    zsockopt_set_subscribe (self->subscriber, subtree);
    return self;
}

static void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->address);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
----
//  Our agent class

//  States we can be in
#define STATE_INITIAL       0   //  Before asking server for state
#define STATE_SYNCING       1   //  Getting state from server
#define STATE_ACTIVE        2   //  Getting new updates from server

typedef struct {
    zctx_t *ctx;                //  Context wrapper
    void *pipe;                 //  Pipe back to application
    zhash_t *kvmap;             //  Actual key/value table
    char *subtree;              //  Subtree specification, if any
    server_t *server [SERVER_MAX];
    uint nbr_servers;           //  0 to SERVER_MAX
    uint state;                 //  Current state
    uint cur_server;            //  If active, server 0 or 1
    int64_t sequence;           //  Last kvmsg processed
    void *publisher;            //  Outgoing updates
} agent_t;

static agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->kvmap = zhash_new ();
    self->subtree = strdup ("");
    self->state = STATE_INITIAL;
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    return self;
}

static void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        int server_nbr;
        for (server_nbr = 0; server_nbr < self->nbr_servers;
server_nbr++)
            server_destroy (&self->server [server_nbr]);
        zhash_destroy (&self->kvmap);
        free (self->subtree);
        free (self);
        *self_p = NULL;
    }
}

//  Returns -1 if thread was interrupted
static int
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);
    if (command == NULL)
        return -1;

    if (streq (command, "SUBTREE")) {
        free (self->subtree);
        self->subtree = zmsg_popstr (msg);
    }
    else
    if (streq (command, "CONNECT")) {
        char *address = zmsg_popstr (msg);
        char *service = zmsg_popstr (msg);
        if (self->nbr_servers < SERVER_MAX) {
            self->server [self->nbr_servers++] = server_new (
                self->ctx, address, atoi (service), self->subtree);
            //  We broadcast updates to all known servers
            zsocket_connect (self->publisher, "%s:%d",
                address, atoi (service) + 2);
        }
        else
            zclock_log ("E: too many servers (max. %d)", SERVER_MAX);
        free (address);
        free (service);
    }
    else
    if (streq (command, "SET")) {
        char *key = zmsg_popstr (msg);
        char *value = zmsg_popstr (msg);
        char *ttl = zmsg_popstr (msg);
        zhash_update (self->kvmap, key, (byte *) value);
        zhash_freefn (self->kvmap, key, free);

        //  Send key-value pair on to server
        kvmsg_t *kvmsg = kvmsg_new (0);
        kvmsg_set_key  (kvmsg, key);
        kvmsg_set_uuid (kvmsg);
        kvmsg_fmt_body (kvmsg, "%s", value);
        kvmsg_set_prop (kvmsg, "ttl", ttl);
        kvmsg_send     (kvmsg, self->publisher);
        kvmsg_destroy (&kvmsg);
puts (key);
        free (ttl);
        free (key);             //  Value is owned by hash table
    }
    else
    if (streq (command, "GET")) {
        char *key = zmsg_popstr (msg);
        char *value = zhash_lookup (self->kvmap, key);
        if (value)
            zstr_send (self->pipe, value);
        else
            zstr_send (self->pipe, "");
        free (key);
        free (value);
    }
    free (command);
    zmsg_destroy (&msg);
    return 0;
```

```
}

//  -------------------------------------------------------------------
----
//  Asynchronous agent manages server pool and handles request/reply
//  dialog when the application asks for it.

static void
clone_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    while (TRUE) {
        zmq_pollitem_t poll_set [] = {
            { pipe, 0, ZMQ_POLLIN, 0 },
            { 0,    0, ZMQ_POLLIN, 0 }
        };
        int poll_timer = -1;
        int poll_size = 2;
        server_t *server = self->server [self->cur_server];
        switch (self->state) {
            case STATE_INITIAL:
                //  In this state we ask the server for a snapshot,
                //  if we have a server to talk to…
                if (self->nbr_servers > 0) {
                    zclock_log ("I: waiting for server at %s:%d…",
                        server->address, server->port);
                    if (server->requests < 2) {
                        zstr_sendm (server->snapshot, "ICANHAZ?");
                        zstr_send  (server->snapshot, self->subtree);
                        server->requests++;
                    }
                    server->expiry = zclock_time () + SERVER_TTL;
                    self->state = STATE_SYNCING;
                    poll_set [1].socket = server->snapshot;
                }
                else
                    poll_size = 1;
                break;
            case STATE_SYNCING:
                //  In this state we read from snapshot and we expect
                //  the server to respond, else we fail over.
                poll_set [1].socket = server->snapshot;
                break;
            case STATE_ACTIVE:
                //  In this state we read from subscriber and we
expect
                //  the server to give hugz, else we fail over.
                poll_set [1].socket = server->subscriber;
                break;
        }
        if (server) {
            poll_timer = (server->expiry - zclock_time ())
                       * ZMQ_POLL_MSEC;
            if (poll_timer < 0)
                poll_timer = 0;
        }
        //  -------------------------------------------------------
---
```

```
        //  Poll loop
        int rc = zmq_poll (poll_set, poll_size, poll_timer);
        if (rc == -1)
            break;              //  Context has been shut down

        if (poll_set [0].revents & ZMQ_POLLIN) {
            if (agent_control_message (self))
                break;          //  Interrupted
        }
        else
        if (poll_set [1].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (poll_set [1].socket);
            if (!kvmsg)
                break;          //  Interrupted

            //  Anything from server resets its expiry time
            server->expiry = zclock_time () + SERVER_TTL;
            if (self->state == STATE_SYNCING) {
                //  Store in snapshot until we're finished
                server->requests = 0;
                if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                    self->sequence = kvmsg_sequence (kvmsg);
                    self->state = STATE_ACTIVE;
                    zclock_log ("I: received from %s:%d snapshot=%d",
                        server->address, server->port,
                        (int) self->sequence);
                    kvmsg_destroy (&kvmsg);
                }
                else
                    kvmsg_store (&kvmsg, self->kvmap);
            }
            else
            if (self->state == STATE_ACTIVE) {
                //  Discard out-of-sequence updates, incl. hugz
                if (kvmsg_sequence (kvmsg) > self->sequence) {
                    self->sequence = kvmsg_sequence (kvmsg);
                    kvmsg_store (&kvmsg, self->kvmap);
                    zclock_log ("I: received from %s:%d update=%d",
                        server->address, server->port,
                        (int) self->sequence);
                }
                else
                    kvmsg_destroy (&kvmsg);
            }
        }
        else {
            //  Server has died, failover to next
            zclock_log ("I: server at %s:%d didn't give hugz",
                    server->address, server->port);
            self->cur_server = (self->cur_server + 1) % self-
>nbr_servers;
            self->state = STATE_INITIAL;
        }
    }
    agent_destroy (&self);
}
```

*clone.c: Clone class*

Finally, here is the sixth and last model of the clone server:

```
//
//  Clone server Model Six
//

//  Lets us build this source without creating a library
#include "bstar.c"
#include "kvmsg.c"

//  Bstar reactor handlers
static int s_snapshots  (zloop_t *loop, void *socket, void *args);
static int s_collector  (zloop_t *loop, void *socket, void *args);
static int s_flush_ttl  (zloop_t *loop, void *socket, void *args);
static int s_send_hugz  (zloop_t *loop, void *socket, void *args);
static int s_new_master (zloop_t *loop, void *unused, void *args);
static int s_new_slave  (zloop_t *loop, void *unused, void *args);
static int s_subscriber (zloop_t *loop, void *socket, void *args);

//  Our server is defined by these properties
typedef struct {
    zctx_t *ctx;                //  Context wrapper
    zhash_t *kvmap;             //  Key-value store
    bstar_t *bstar;             //  Bstar reactor core
    int64_t sequence;           //  How many updates we're at
    int port;                   //  Main port we're working on
    int peer;                   //  Main port of our peer
    void *publisher;            //  Publish updates and hugz
    void *collector;            //  Collect updates from clients
    void *subscriber;           //  Get updates from peer
    zlist_t *pending;           //  Pending updates from clients
    Bool primary;               //  TRUE if we're primary
    Bool master;                //  TRUE if we're master
    Bool slave;                 //  TRUE if we're slave
} clonesrv_t;

int main (int argc, char *argv [])
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));
    if (argc == 2 && streq (argv [1], "-p")) {
        zclock_log ("I: primary master, waiting for backup (slave)");
        self->bstar = bstar_new (BSTAR_PRIMARY, "tcp://*:5003",
                                 "tcp://localhost:5004");
        bstar_voter (self->bstar, "tcp://*:5556", ZMQ_ROUTER,
                     s_snapshots, self);
        self->port = 5556;
        self->peer = 5566;
        self->primary = TRUE;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        zclock_log ("I: backup slave, waiting for primary (master)");
        self->bstar = bstar_new (BSTAR_BACKUP, "tcp://*:5004",
                                 "tcp://localhost:5003");
        bstar_voter (self->bstar, "tcp://*:5566", ZMQ_ROUTER,
                     s_snapshots, self);
        self->port = 5566;
        self->peer = 5556;
        self->primary = FALSE;
    }
    else {
        printf ("Usage: clonesrv4 { -p | -b }\n");
        free (self);
        exit (0);
    }
    //  Primary server will become first master
    if (self->primary)
        self->kvmap = zhash_new ();

    self->ctx = zctx_new ();
    self->pending = zlist_new ();
    bstar_set_verbose (self->bstar, TRUE);

    //  Set up our clone server sockets
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  Set up our own clone client interface to peer
    self->subscriber = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "tcp://localhost:%d", self-
>peer + 1);

    //  Register state change handlers
    bstar_new_master (self->bstar, s_new_master, self);
    bstar_new_slave (self->bstar, s_new_slave, self);

    //  Register our other handlers with the bstar reactor
    zloop_reader (bstar_zloop (self->bstar), self->collector,
s_collector, self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_flush_ttl,
self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_send_hugz,
self);

    //  Start the Bstar reactor
    bstar_start (self->bstar);

    //  Interrupted, so shut down
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_destroy (&kvmsg);
    }
    zlist_destroy (&self->pending);
    bstar_destroy (&self->bstar);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);

    return 0;
}

//  -----------------------------------------------------------------
----
//  Send snapshots to clients who ask for them

static int s_send_single (char *key, void *data, void *args);

//  Routing information for a key-value snapshot
```

```
typedef struct {
    void *socket;           //  ROUTER socket to send to
    zframe_t *identity;     //  Identity of peer who requested state
    char *subtree;          //  Client subtree specification
} kvroute_t;

static int
s_snapshots (zloop_t *loop, void *snapshot, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (snapshot);
    if (identity) {
        //  Request is in second frame of message
        char *request = zstr_recv (snapshot);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (snapshot);
        }
        else
            printf ("E: bad request, aborting\n");

        if (subtree) {
            //  Send state socket to client
            kvroute_t routing = { snapshot, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  Now send END message with sequence number
            zclock_log ("I: sending shapshot=%d", (int) self-
>sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    return 0;
}

//  Send one state snapshot key-value pair to a socket
//  Hash item data is our kvmsg object, ready to send
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  Send identity of recipient first
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}
```

```
//  ---------------------------------------------------------------------
----
//  Collect updates from clients
//  If we're master, we apply these to the kvmap
//  If we're slave, or unsure, we queue them on our pending list

static int s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg);

static int
s_collector (zloop_t *loop, void *collector, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (collector);
    kvmsg_dump (kvmsg);
    if (kvmsg) {
        if (self->master) {
            kvmsg_set_sequence (kvmsg, ++self->sequence);
            kvmsg_send (kvmsg, self->publisher);
            int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
            if (ttl)
                kvmsg_set_prop (kvmsg, "ttl",
                    "%" PRId64, zclock_time () + ttl * 1000);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: publishing update=%d", (int) self-
>sequence);
        }
        else {
            //  If we already got message from master, drop it, else
            //  hold on pending list
            if (s_was_pending (self, kvmsg))
                kvmsg_destroy (&kvmsg);
            else
                zlist_append (self->pending, kvmsg);
        }
    }
    return 0;
}

//  If message was already on pending list, remove it and
//  return TRUE, else return FALSE.

static int
s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg)
{
    kvmsg_t *held = (kvmsg_t *) zlist_first (self->pending);
    while (held) {
        if (memcmp (kvmsg_uuid (kvmsg),
                    kvmsg_uuid (held), sizeof (uuid_t)) == 0) {
            zlist_remove (self->pending, held);
            return TRUE;
        }
        held = (kvmsg_t *) zlist_next (self->pending);
    }
    return FALSE;
}

//  ---------------------------------------------------------------------
----
```

```
//  Purge ephemeral values that have expired

static int s_flush_single (char *key, void *data, void *args);

static int
s_flush_ttl (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}

//  If key-value pair has expired, delete it and publish the
//  fact to listening clients.
static int
s_flush_single (char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing delete=%d", (int) self->sequence);
    }
    return 0;
}

//  -----------------------------------------------------------------
----
//  Send hugz to anyone listening on the publisher socket

static int
s_send_hugz (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_new (self->sequence);
    kvmsg_set_key  (kvmsg, "HUGZ");
    kvmsg_set_body (kvmsg, (byte *) "", 0);
    kvmsg_send     (kvmsg, self->publisher);
    kvmsg_destroy (&kvmsg);

    return 0;
}

//  -----------------------------------------------------------------
----
//  State change handlers
//  We're becoming master
//
//  The backup server applies its pending list to its own hash table,
//  and then starts to process state snapshot requests.

static int
s_new_master (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    self->master = TRUE;
    self->slave = FALSE;
    zloop_cancel (bstar_zloop (self->bstar), self->subscriber);

    //  Apply pending list to own hash table
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: publishing pending=%d", (int) self-
>sequence);
    }
    return 0;
}

//  ---------------------------------------------------------------------
----
//  We're becoming slave

static int
s_new_slave (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zhash_destroy (&self->kvmap);
    self->master = FALSE;
    self->slave = TRUE;
    zloop_reader (bstar_zloop (self->bstar), self->subscriber,
                  s_subscriber, self);

    return 0;
}

//  ---------------------------------------------------------------------
----
//  Collect updates from peer (master)
//  We're always slave when we get these updates

static int
s_subscriber (zloop_t *loop, void *subscriber, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    //  Get state snapshot if necessary
    if (self->kvmap == NULL) {
        self->kvmap = zhash_new ();
        void *snapshot = zsocket_new (self->ctx, ZMQ_DEALER);
        zsocket_connect (snapshot, "tcp://localhost:%d", self->peer);
        zclock_log ("I: asking for snapshot from:
tcp://localhost:%d",
                    self->peer);
        zstr_send (snapshot, "ICANHAZ?");
        while (TRUE) {
            kvmsg_t *kvmsg = kvmsg_recv (snapshot);
            if (!kvmsg)
                break;          //  Interrupted
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                kvmsg_destroy (&kvmsg);
                break;          //  Done
            }
            kvmsg_store (&kvmsg, self->kvmap);
        }
        zclock_log ("I: received snapshot=%d", (int) self->sequence);
        zsocket_destroy (self->ctx, snapshot);
    }
    //  Find and remove update off pending list
    kvmsg_t *kvmsg = kvmsg_recv (subscriber);
    if (!kvmsg)
        return 0;

    if (strneq (kvmsg_key (kvmsg), "HUGZ")) {
        if (!s_was_pending (self, kvmsg)) {
            //  If master update came before client update, flip it
            //  around, store master update (with sequence) on
pending
            //  list and use to clear client update when it comes
later
            zlist_append (self->pending, kvmsg_dup (kvmsg));
        }
        //  If update is more recent than our kvmap, apply it
        if (kvmsg_sequence (kvmsg) > self->sequence) {
            self->sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: received update=%d", (int) self-
>sequence);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    else
        kvmsg_destroy (&kvmsg);

    return 0;
}
```

*clonesrv6.c: Clone server, Model Six*

This main program is only a few hundred lines of code, but it took some time to get working. To be accurate, building Model Six was a bitch, and took about a full week of "sweet god, this is just too complex for the Guide" hacking. We've assembled pretty much everything and the kitchen sink into this small application. We have failover, ephemeral values, subtrees, and so on. What surprised me was that the upfront design was pretty accurate. But the details of writing and debugging so many socket flows is something special. Here's how I made this work:

- By using reactors (bstar, on top of zloop), which remove a lot of grunt-work from the code, and leave what remains simpler and more obvious. The whole server runs as one thread, so there's no inter-thread weirdness going on. Just pass a structure pointer ('self') around to all handlers, which can do their thing happily. One nice side-effect of using reactors is that code, being less tightly integrated into a poll loop, is much easier to reuse. Large chunks of Model Six are taken from Model Five.

- By building it piece by piece, and getting each piece working **properly** before going onto the next one. Since there are four or five main socket flows, that meant quite a lot of debugging and testing. I debug just by printing stuff to the console (e.g. dumping messages). There's no sense in actually opening a debugger for this kind of work.

- By always testing under Valgrind, so that I'm sure there are no memory leaks. In C this is a major concern, you can't delegate to some garbage collector. Using proper and consistent abstractions like kvmsg and czmq helps enormously.

I'm sure the code still has flaws which kind readers will spend weekends debugging and fixing for me. I'm happy enough with this model to use it as the basis for real applications.

To test the sixth model, start the primary server and backup server, and a set of clients, in any order. Then kill and restart one of the servers, randomly, and keep doing this. If the design and code is accurate, clients will continue to get the same stream of updates from whatever server is currently master.

## Clone Protocol Specification

top prev next

After this much work to build reliable pub-sub, we want some guarantee that we can safely build applications to exploit the work. A good start is to write-up the protocol. This lets us make implementations in other languages and lets us improve the design on paper, rather than hands-deep in code.

Here, then, is the Clustered Hashmap Protocol, which "*defines a cluster-wide key-value hashmap, and mechanisms for sharing this across a set of clients. CHP allows clients to work with subtrees of the hashmap, to update values, and to define ephemeral values.*"

- http://rfc.zeromq.org/spec:12

(More coming soon…)

## Footnotes

1. I.e. ferrous-based hard disk platters.